



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: Novel approaches to the detection of backdoors
Student: Bc. Jan Vojtěšek
Supervisor: Ing. Josef Kokeš
Study Programme: Informatics
Study Branch: Computer Security
Department: Department of Information Security
Validity: Until the end of summer semester 2020/21

Instructions

Study the available methods used for static analysis of a binary code, with a particular focus on Windows-native (Portable Executable) file format.

Research the techniques used by malware developers to embed their code into a benign application in general.

Analyze how your findings apply to high-profile backdoors (such as ShadowPad or ShadowHammer).

Propose heuristics which could be used for finding backdoors.

Implement several chosen heuristics and measure their performance against a dataset of malicious and benign software.

Discuss your results and the possibility of their real-world application.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague September 25, 2019



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Novel Approaches to the Detection of Backdoors

Bc. Jan Vojtěšek

Department of Information Security

Supervisor: Ing. Josef Kokeš

May 26, 2020

Acknowledgements

I would like to express my deep gratitude to Ing. Josef Kokeš for all his dedication, guidance, and insight while he was supervising this thesis. I am also very grateful to my family and friends, who supported me throughout my studies. Furthermore, I would like to thank the antivirus company Avast for letting me use its collection of samples for this research. Last but not least, a big thank you goes out to everybody who helped me review this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 26, 2020

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2020 Jan Vojtěšek. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Vojtěšek, Jan. *Novel Approaches to the Detection of Backdoors*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstract

This thesis presents a detailed examination of application backdoors hidden in Portable Executable files and proposes novel anomaly-based methods for their heuristic detection. Four application backdoors used in large-scale supply chain attacks were reverse-engineered and shown to exhibit anomalous properties that could be utilized in the search for similar backdoors. These anomalous properties serve as the basis for three heuristic detections that were implemented and had their performance evaluated on a dataset composed of both benign and backdoored applications.

Keywords backdoor, malware, static analysis, anomaly detection, reverse engineering

Abstrakt

Tato práce prezentuje detailní průzkum aplikačních backdoorů ukrytých v souborech formátu Portable Executable a navrhuje nové metody jejich heuristické detekce. Čtyři aplikační backdoory použité v rozsáhlých útocích na softwarové dodavatele byly zanalyzovány za použití reverzního inženýrství a bylo ukázáno, že vykazují anomální vlastnosti, kterých lze využít při hledání podobných backdoorů. Tyto anomální vlastnosti slouží jako základ tří heuristických detekcí, které byly naimplementovány a jejichž výkon byl vyhodnocen na datasetu obsahujícím jak neškodné, tak backdoorované aplikace.

Klíčová slova backdoor, škodlivý software, statická analýza, detekce anomálií, reverzní inženýrství

Contents

Introduction	1
1 Static Analysis of PE Files	3
1.1 The PE File Format	4
1.2 Static Analysis Tools	8
2 Backdoored PE Files	11
2.1 Backdoor Classes	11
2.2 Trojanized PE Files	14
2.3 Defense Against Trojanization	15
2.4 Techniques for Trojanizing PE Files	16
2.5 Automatic Trojanization of PE Files	22
3 Backdoor Case Studies	25
3.1 ShadowHammer	25
3.2 Asian Gaming Industry Incidents	30
3.3 The CCleaner Incident	32
3.4 ShadowPad	36
3.5 Summary	38
4 Call Graph Extractor	41
4.1 Design of the Extractor	41
4.2 Implementation of the Extractor	46
4.3 Description of the Extracted Properties	48
4.4 Issues with the Extractor	54
5 Heuristic Detections	57
5.1 Entry Point Hijacking	59
5.2 Anomalous Cross References	62
5.3 Import Call Hijacking	66

5.4 Other Heuristic Detections	71
5.5 Evaluation	72
Conclusion	75
Bibliography	77
A Acronyms	85
B Contents of the Enclosed CD	87

List of Figures

1.1	The PE header	5
2.1	A PE file trojanized through packing	17
2.2	A PE file trojanized through entry point hijacking	20
3.1	The hijack method used in the ShadowHammer backdoor	28
3.2	The hijack method used in the Asian Gaming Industry Incidents	31
3.3	The first hijack method used in the CCleaner Incident	33
3.4	The second hijack method used in the CCleaner Incident	34
4.1	An example of a suspicious code fragment	43
4.2	Other examples of suspicious code fragments	52
5.1	Functions most commonly cross-referenced from the entry point	60
5.2	The official implementation of the <code>__crtExitProcess</code> function	63
5.3	A stub function that resolves the <code>CreateProcessW</code> function	70

Introduction

Supply chain attacks conducted using application backdoors pose a serious threat to the cybersecurity of countless organizations. In such attacks, malicious actors attempt to compromise a software vendor with the intention of secretly implanting a backdoor into some widely distributed software. Even though the software vendor gets attacked first, it is not the primary target of these attacks. Instead, the attackers are commonly targeting a carefully selected subset of the backdoored software's users. When the backdoored software gets executed by one of those users, the backdoor activates itself and provides remote control of the infected machine to the attackers.

Defenders employed by software vendors are generally aware of the risks associated with supply chain attacks. However, the attackers' incentives to perform such attacks are extremely high, which attracted advanced malicious actors who managed to pull off large-scale attacks, such as ShadowPad [23] and ShadowHammer [25]. One of the main incentives is the enormous spread potential: backdooring a product with a large install base can give the attackers access to millions of machines just by compromising a single target. Supply chain attacks might also make it easier for attackers to hack organizations that implement effective security measures. They can choose a supplier with poor defensive capabilities and get into the target organization indirectly through this supplier.

Supply chain attacks are hard to defend against, both for the software vendors and for the targeted users. For instance, the backdoor could be inserted by a maliciously modified compiler, which would mean that the attacked software vendor would not be able to find traces of the backdoor in its source code. The vendor could then unwittingly push the backdoored software via an automatic update to its users' machines where the backdoor would operate silently in the background. Since the backdoor would not interfere with the legitimate functionality of the backdoored software, it would be virtually impossible for a regular user to notice that something is wrong. Security is often based on trust. This attack scenario shows that it might not

be enough to trust the suppliers' good intentions. One also has to be confident of the suppliers' ability to defend themselves in order to lower the chances of attacks like this.

Most backdoors used in supply chain attacks were discovered by defenders who investigated a malware incident and managed to trace it back to its initial infection vector, the backdoored application [23, 16]. Unfortunately, the discovery often took place months after the backdoor first became active in-the-wild. This naturally begs the question of whether there is a way to detect these backdoors sooner, before they inflict too much damage. There could also be other, undiscovered backdoors currently active in-the-wild, which is why it would be beneficial to have some tools ready to help in the hunt for such backdoors.

Designing such tools is precisely the goal of this thesis. As is shown in Chapter 3, backdoored applications often exhibit various anomalous properties. These anomalous properties tend to be caused by the invasive manner in which the backdoor gets implanted inside a benign application. The main idea behind this thesis is to design static heuristic detections that point out applications exhibiting such anomalous properties, since those applications might be hiding yet undiscovered backdoors.

The structure of this thesis is as follows. Chapter 1 introduces the Portable Executable file format and describes techniques applicable for static analysis of files in this format. Chapter 2 focuses on backdoors and details various methods that can be used to backdoor an executable file. Chapter 3 highlights several backdoors used in high-profile supply chain attacks and discusses anomalous properties observed in those backdoors. Chapter 4 describes the design and implementation of a tool that processes Portable Executable files and outputs a multitude of features extracted from those files. Finally, Chapter 5 proposes several heuristic detections that could help detect undiscovered backdoors. This final chapter also contains an evaluation of these heuristic detections, based on a dataset containing both benign and backdoored software.

Static Analysis of PE Files

The two main approaches to software reverse engineering are static analysis and dynamic analysis [49]. Dynamic analysis involves executing the analyzed program in a controlled environment, such as a sandbox or an emulator. This environment usually makes it possible to inspect the behavior of the analyzed program in order to learn about its functionality. The other approach, static analysis, is the process of examining a program without actually executing it [52]. When static analysis is applied to Portable Executable (PE) files with no available source code, it often involves reading disassembled or decompiled code in order to obtain more information about the analyzed file.

Unfortunately, dynamic analysis can only explore the code paths that actually got executed [28]. This could pose a problem, especially if the analyzed program attempts to detect if it is running in an analysis environment and deliberately avoids executing some of its code if it concluded that it is [27]. In contrast, static analysis can be much more comprehensive, but it is usually also more time-consuming and complex. Static analysis can also be made even more difficult by various obfuscation techniques, such as string encryption or control flow flattening [12]. Since both approaches to reverse engineering have their advantages and disadvantages, selecting the right approach for the given task is crucial. Both approaches also complement each other well, so it is often beneficial to combine them. For example, static analysis can be used to make assumptions about program behavior which can then be verified using dynamic analysis.

Static analysis is also commonly applied in areas other than reverse engineering. Most notably, it is often used by compilers to generate more efficient code [43] or for quality assurance to investigate software for potential bugs and vulnerabilities [35]. For such purposes, static analysis can be performed on the source code level, object code level, or even on an intermediate representation. For reverse engineering, however, source code is usually not available, so static analysis generally has to be performed on the object code level. In such cases, static analysis often consists of

disassembling/decompiling code or parsing executable files in order to extract information from their headers and various other data structures.

This chapter contains a brief introduction into the structure of the PE file format, which is used throughout Microsoft Windows for storing executable code. Some of the tools useful for static analysis of PE files are also presented. The two most important types of such tools are a disassembler and a decompiler. While a disassembler's job is to translate machine code back into human-readable assembly, a decompiler attempts to reconstruct the original source code in a higher-level programming language.

1.1 The PE File Format

The Portable Executable is an executable file format that is widely used on Microsoft Windows. It is meant to encapsulate machine code with multiple headers and data structures that help the operating system load and execute the embedded code. Its structure is based on the Common Object File Format (COFF), which is nowadays still used on Microsoft Windows for object files [41]. The PE file format is designed to be as generic as possible, and it therefore supports multiple instruction set architectures and multiple subsystems. A subsystem essentially defines the environment that the PE file can be executed in. Depending on the subsystem, a PE file can be used as a Win32 application, a device driver, an Extensible Firmware Interface (EFI) application, and more. A PE file can also be used as an executable file or as a software library, such as a Dynamic-link Library (DLL). Since PE files can be used in so many contexts, many extensions are commonly associated with them. The most common ones are `.exe`, `.dll`, and `.sys`. Note that even though the format was initially designed to contain primarily executable machine code, there are nowadays even PE files that do not contain any meaningful custom native code. Examples of this are `.NET` executables and resource-only PE files.

The basic structure of the main headers in a PE file can be seen in Figure 1.1. The very first 64 bytes of a PE file are occupied by the MS-DOS header [39] (formally `IMAGE_DOS_HEADER`, but it is also often referred to as the *MZ header*, based on the magic number that identifies it). This header exists mostly for backwards compatibility, and it is often accompanied by a simple program for MS-DOS that prints a warning that the PE file cannot be executed under MS-DOS. The most important part of the MS-DOS header is called `e_lfanew`, and it contains a 32-bit file offset to where the actual PE header is located.

The PE header is made up of the COFF Header and the Optional Header¹. It starts with a 32-bit signature that spells out ASCII letters `PE` followed

¹The name *Optional Header* might be a bit misleading, since a PE file is required to have it, otherwise it will not be loaded by the operating system.

1.1. The PE File Format

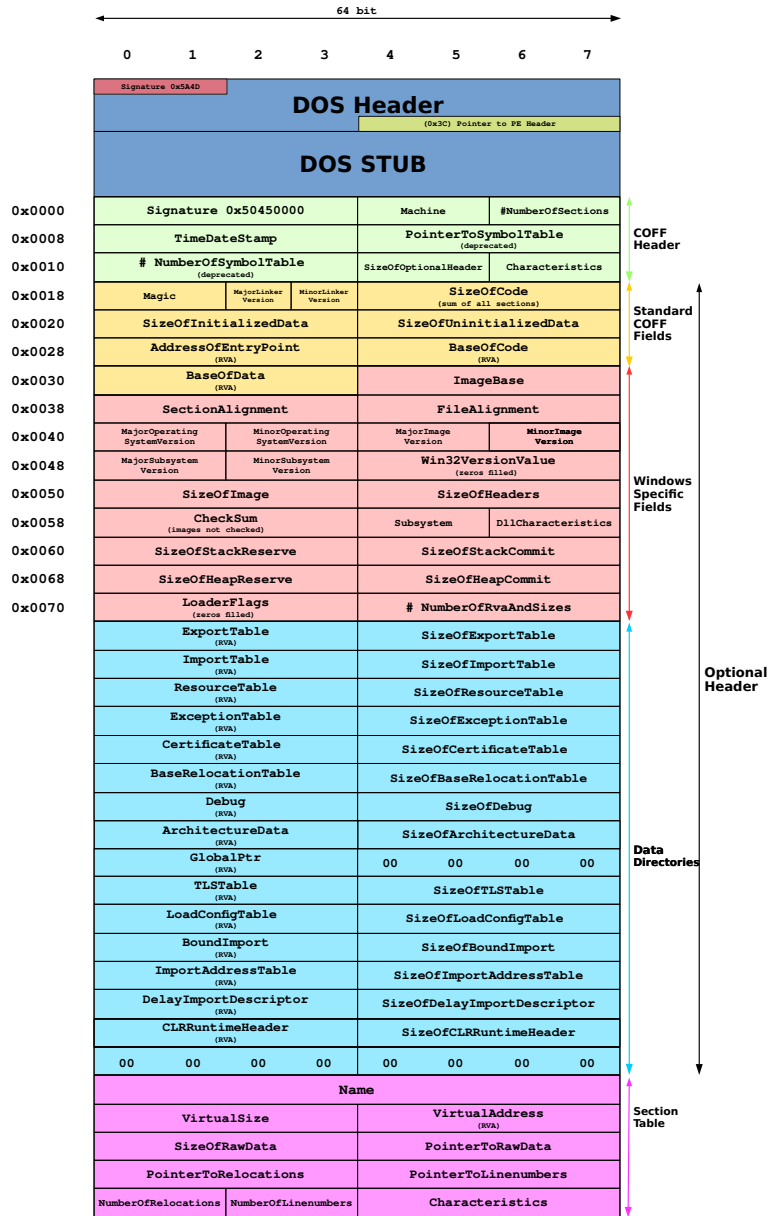


Figure 1.1: A visualization of the headers in a PE file. Retrieved from [11].

by two null bytes. The PE header contains many essential items that are necessary for the program's functionality. The fields `Machine` and `Subsystem` identify the instruction set architecture and subsystem required for the PE file's execution. `ImageBase` contains the preferred base address where the PE file should be mapped when it gets loaded². `AddressOfEntryPoint` points to the first instruction in the PE file that gets executed.

The Optional Header ends with an array that contains the locations and sizes of data directories. Each data directory is optional and can contain data structures that further describe some additional characteristics of the PE file. For example, `IMAGE_DIRECTORY_ENTRY_EXPORT` can point to the `IMAGE_EXPORT_DIRECTORY` header, which enumerates all functions exported by the particular PE file. These exported functions can then be imported into and called from another PE file, which allows interaction between pieces of code that are situated in different PE files. Another example is `IMAGE_DIRECTORY_ENTRY_SECURITY`, which might point to the Authenticode digital signature that can be used to verify the PE file's integrity.

Another important header that can be found in a PE file is the Section Table. It is located directly after the Optional Header, and it is an array of Section Headers, each of which describes an individual section in the PE file. A section is essentially a contiguous chunk of memory that gets loaded into the virtual address space of a program before a PE file can be executed. Each Section Header contains the fields `PointerToRawData` and `SizeOfRawData`, which describe where can the section be found in the raw PE file and how big it is. It also contains the fields `VirtualAddress` and `VirtualSize`, which describe where the section should be loaded in the virtual address space and how much space should it occupy there. Note that `VirtualSize` does not always have to equal `SizeOfRawData`. If `SizeOfRawData` is less than `VirtualSize`, the remainder of the virtual address space will be zero-filled by the loader [39]. Each section can also be given a name that has to fit into eight bytes. While the section name can be arbitrary and does not carry much significance, certain conventions are often followed, such as naming sections that contain machine code `.text` or `.code` and naming sections that contain constant data `.rdata`. Each Section Header also contains section `Characteristics`, which hold multiple flags that further describe the section. Out of those flags, the most important ones for this thesis are `IMAGE_SCN_MEM_READ`, `IMAGE_SCN_MEM_WRITE`, and `IMAGE_SCN_MEM_EXECUTE`. They express respectively whether the section can be read from, written to, or executed as code. Attempting to perform an illegal operation on data from some section would result in a memory access violation exception.

²External factors such as Address Space Layout Randomization might cause the PE file to be mapped to another address, but this field is still necessary for the loader to apply relocation fixups.

The PE file format is utilized extensively in the rest of this thesis. In order to provide the reader with the necessary background, some additional concepts related to PE files have to be explained.

- **Relative virtual address**

Items in a PE file are usually referenced either by their file offset or by their relative virtual address (RVA). A file offset is simply the position of the item within the raw PE file as it is stored on the filesystem, whereas RVA is the address of the item once it gets loaded in the virtual address space, relative to the base address of the image. Therefore, an item's RVA can be simply computed by subtracting the module's base address from the virtual address where the item got loaded. RVAs are used extensively in the PE header, since they are efficient to compute and can refer even to addresses that are not backed by the PE file (and that therefore cannot be referred to by the file offset).

- **Overlay**

Arbitrary data can be appended to the end of a PE file. If this data does not belong to any section, it is called the overlay. The overlay is sometimes used to bundle additional data inside a single PE file. Since it does not belong to any section, it does not get loaded into the virtual address space. Therefore, accessing the appended data entails reading the image's raw PE file at runtime and finding the overlay's file offset.

- **PE resources**

Another way to bundle arbitrary data in a PE file is to use PE resources. Unlike the overlay, PE resources are a part of the PE file format and can be found by following the `IMAGE_DIRECTORY_ENTRY_RESOURCE` data directory. The content of PE resources can commonly be found in a section called `.rsrc` and can be accessed using functions such as `FindResource` and `LoadResource`. Resources are organized in a tree structure, where each resource has an associated type and a language identifier. The same resource can be defined in multiple languages in order to localize content for the end users.

- **Relocations**

Machine code is often position-dependent, which means that it will only execute correctly when it is loaded at its preferred address. This is caused by the fact that it can contain absolute addresses to other pieces of code or data, which unfortunately change when a PE file gets loaded at an unexpected address. If it should be possible to load a PE file at an arbitrary base address (which is definitely a desirable property because of possible virtual address space collisions and Address Space Layout Randomization (ASLR)), the PE file should

provide the `IMAGE_DIRECTORY_ENTRY_BASERELOC` data directory. This data directory contains a list of relocation entries (sometimes also called *fixups*) that hold a list of RVAs that have to be fixed when a PE file gets loaded at an unexpected address. Each relocation entry also specifies the type of the fixup that should be applied. The most common type consists of just adding the 32-bit difference between the actual and the preferred base address.

- **Rich header**

The Rich header is an undocumented structure containing information about the tools used to build a PE file [14]. It can be found in PE files created using modern versions of the Microsoft Visual C++ (MSVC) linker, where it logs the number of times a particular version of a certain build tool has been used to build the final executable [59].

1.2 Static Analysis Tools

There are many different tools that make static analysis of PE files easier. Some of them are briefly introduced in this section in order to illustrate how they can be used during the reverse engineering process.

1.2.1 IDA Pro

IDA Pro³ is a proprietary disassembler that runs on Microsoft Windows, Linux, and Mac OS X. It supports a wide range of instruction sets and executable file formats. It can show disassembly in two views that a user can switch between. The first one is called *text view*, which is essentially a linear disassembly listing. The other one is called *graph view*. It displays functions in customizable graphs where basic blocks are represented as graph nodes. IDA Pro allows its users to interact with the disassembly in various ways. Users can rename items, add comments, (un)define instructions and functions, create custom data structures, and more.

In order to speed up reverse engineering efforts, users can also purchase and use the Hex-Rays decompiler, which is tightly integrated into IDA Pro. At the time of writing, it is capable of decompiling machine code from five different instruction set architectures into C-style pseudocode. The decompiler uses a custom intermediate language internally, which should make it easier to add support for decompilation of additional instruction set architectures in the future.

Many executable files contain statically linked library functions. IDA Pro attempts to automatically identify such functions and assign them descriptive names so that its users do not have to delve deep into library functions

³<https://www.hex-rays.com/products/ida/>

that are not relevant for the analyzed program's functionality. IDA ships with a custom technology called F.L.I.R.T. (Fast Library Identification and Recognition Technology) that uses a list of function signatures and attempts to match them to all unrecognized functions.

Dynamic analysis can also be conducted with IDA Pro, since it contains an integrated debugger. The debugger supports the usual features, such as single-stepping, memory patching, and placing breakpoints, as well as some more advanced features, such as remote debugging, tracing, and taking memory snapshots.

IDA Pro also supports scripting in either IDC (a custom scripting language designed just for IDA) or in IDAPython, which is a Python interpreter that is available from within IDA. Scripts executed in it can interact with IDA's API in order to automate various reverse engineering tasks. IDA is also extensible through plugins written in C++. These plugins may interact closely with IDA's static analysis engine and can therefore be used to provide additional functionality, such as parsing otherwise unsupported executable file formats.

1.2.2 PE-Bear

PE-Bear⁴ is a freeware tool that parses PE headers and presents their decoded content in a well-arranged graphical user interface (GUI). It displays, among others, the COFF Header, the Optional Header, the Rich Header, the Section Table, and the Import Directory. It features a built-in disassembler and a hex editor so that its users can take a more detailed view at the analyzed PE file. PE-Bear can also be used to perform basic modifications of a PE file, such as editing the PE header, adding new sections and imports, and nopping out instructions.

It is possible to compare two PE files in PE-Bear as well. The built-in comparison tool highlights any differences between the compared files so that its users can quickly figure out which items from which headers have changed.

1.2.3 Resource Hacker

Resource Hacker⁵ is a freeware utility for viewing and editing PE resources. It is capable of parsing the Resource Table of both 32-bit and 64-bit PE files and listing all the individual resources contained within. Each resource is displayed in an appropriate way based on its resource type. Icons and cursors are displayed as images, dialog boxes are shown in a preview, string resources are displayed as text, and so forth. Each resource can also be analyzed in its raw binary form in an embedded hex viewer. Existing resources can be exported, deleted, or replaced, and new resources can be added. It is possible

⁴<https://hshrzd.wordpress.com/pe-bear/>

⁵<http://www.angusj.com/resourcehacker/>

to edit many types of resources directly in Resource Hacker with a custom editor based on the type of the edited resource.

Resource Hacker also features a compiler for resource script files, and it can open compiled resource files (often associated with the `.res` extension). It is commonly used as a GUI application, but it also provides a command-line interface that exposes most of the functionality that is available from the GUI.

1.2.4 Detect It Easy

Detect It Easy⁶ is a cross-platform and portable packer identifier. It is especially helpful during the initial static analysis when it can quickly determine the category of an unknown PE file. To do so, it contains an extensive database of signatures for various packers, protectors, compilers, linkers, installers, and self-extracting archives. A signature match provides valuable information to analysts, who can decide their next analysis steps based on the name and type of the matched signature.

The signatures are essentially algorithmic detections written in a scripting language similar to JavaScript. This distinguishes Detect It Easy from other packer identifiers such as PeID⁷, whose signatures are composed of plain data which usually just describes the bytes that should be present at the entry point for a signature to match. Signatures for Detect It Easy are not difficult to comprehend, and users can contribute by creating new ones or improving existing ones.

There are also signatures for other executable file formats such as the Executable and Linkable Format (ELF) and Mach-O. There are signatures for generic file formats as well, which allows Detect It Easy to be used similarly as the Unix `file` command. Detect It Easy also contains many other features, such as graphing the Shannon entropy of different parts of the analyzed file, extracting Uniform Resource Locators (URLs), and identifying the usage of cryptographic algorithms. As is the case with PE-Bear, it is capable of parsing PE headers and displaying them in a human-readable format.

⁶<https://github.com/horsicq/Detect-It-Easy>

⁷<https://www.aldeid.com/wiki/PEiD>

Backdoored PE Files

In computer security, backdoors can be defined as deliberately crafted means of bypassing authentication or other security controls [60]. They can be implanted into systems for legitimate reasons or with malicious intent. While the mere existence of a backdoor is frequently criticized [7], especially troublesome are backdoors whose very existence is hidden from the public [17, 6]. By creating such covert backdoors, users' trust is being abused, regardless of the backdoor's purpose.

There are both hardware and software backdoors. Hardware backdoors are usually created by including some secret logic, software backdoors by including hidden code. Software backdoors can be further divided into system backdoors and application backdoors [60]. While system backdoors allow attackers continued access to an already compromised system, application backdoors are legitimate pieces of software that were compromised and modified to perform malicious actions. Only application backdoors are discussed further, with a particular focus on backdoors that are hidden inside otherwise legitimate PE files. Since the above-given definition of a backdoor is very broad, there are numerous forms that an application backdoor can take. In the following section, an overview of several different classes of backdoors is given.

2.1 Backdoor Classes

The first class of PE backdoors is a trojanized executable. Such backdoors are created by implanting malicious code into another application. The execution flow of the original program is typically modified so that the inserted malicious code gets executed. This kind of backdoor is often created in such a way that the functionality of the benign application is preserved as much as possible and that the execution of the malicious code is hidden. This way, the user of the backdoored application might not notice any signs of malicious behavior. The functionality of the malicious code does vary from case to case. Sometimes, a full-blown remote administration tool is implanted [56]. In many

2. BACKDOORED PE FILES

cases [33, 25], however, the backdoor authors try to implant as little malicious code as possible, so they only include a small stub that is able to download the malware's next stage from the Internet or give the attackers a remote shell. An advantage of this approach is that if the backdoor gets discovered, they can shut down the control server, and that there is less malicious code in the trojanized executable, which makes detection and attribution of the attack harder.

The second class are backdoors that take the form of a security vulnerability. For instance, an attacker might introduce a remote code execution vulnerability to a server application. Another example would be a cryptographic vulnerability that enables the attackers to perform actions for which they lack authorization. These kinds of backdoors are usually very dangerous because anybody who knows about them might be able to exploit them. Parties who introduce them often believe that *nobody but them* [10] knows about the backdoor, so they are the only ones capable of abusing them. In some cases, however, not everyone who knows about these vulnerabilities will be able to exploit them. An example of this is a cryptographical backdoor that only the parties who have access to certain predetermined values can exploit [6]. Vulnerability backdoors can also emerge as a security bug unwittingly created by the application's developers. These vulnerabilities become backdoors at the moment an attacker learns about them and chooses not to disclose them. This leads to an interesting property of many such backdoors: plausible deniability [51]. Even if somebody learns about a backdoor vulnerability and knows who is responsible, it would be hard for them to tell if the vulnerability is a result of malice or if it was unintentionally created as a software bug.

A backdoor might also be created by modifying pieces of data in a target benign application. For example, the URL from which application updates are downloaded could be changed to an attacker-controlled server. A cryptocurrency miner could have its "donate address" changed to an address of the attacker. An additional set of credentials could be added to an authentication program, and so forth. Similarly, some piece of code could be altered to modify the application's functionality for the attacker's benefit. For instance, a cryptocurrency wallet could be altered to divert selected transactions to the attacker. A network traffic analysis tool could be modified to hide certain packets. An online game client could be sabotaged to put some of its users at an unfair disadvantage. Clearly, these are just a few examples, and there are many more possibilities depending on the target application and the attacker's motives.

While not strictly an application backdoor, it should also be mentioned that legitimate application updates can also be abused in order to turn an application into a backdoor. In this case, the attacker infiltrates an update server of the application and starts to serve malicious updates. The attacker can either target a chosen subset of users or infect all of them indiscriminately.

In 2017, malware dubbed NotPetya [32] was spread through the update process of an otherwise benign tax software. If the updated software does not sufficiently check the integrity of the update, the attacker could also abuse a Man-in-the-Middle (MitM) position to hijack the update process. For instance, the notorious Flame [5] malware abused a vulnerability in Windows Update in order to spread itself. Depending on the exact update validation performed, the attacker could also steal the private keys used to sign the update and forge a fraudulent signature for a malicious update. This might enable the attacker to abuse the update process from a MitM position even without having a foothold on the update server.

Backdoors can also be characterized by the stage of the software build process during which they are inserted into the application. If the application to be backdoored is being compiled, the backdoor can be included either before compilation (typically as a piece of source code), after compilation (typically as a piece of object code), or even during compilation if the compiler itself is backdoored [58, 22]. In open-source applications, backdoors may be easier to spot, since anyone who reviews the code might notice them. Analyzing object code of closed-source applications often requires reverse engineering, so such backdoors are much harder for the public to find. To illustrate this point, a backdoor existed in Borland Interbase for at least seven years until it was discovered shortly after the software became open source [60]. On the other hand, it is harder to introduce backdoors to the source code of a closed-source application, since access to it is often heavily guarded, and only selected engineers can work on it. In the open-source world, anyone can contribute code (and thus attempt to introduce backdoors), so security depends on the amount of scrutiny with which any proposed code is analyzed. Since open-source applications are also in some cases vulnerable to backdoors introduced during or after the build process, efforts have been made to introduce reproducible builds [31]. These attempt to guarantee that object code corresponds to known source code and is without any additional modifications. Backdoors in source code can be discovered by looking at either the object code or the source code, while backdoors in object code can only be found in object code. However, when one analyzes object code, a backdoor that was created in source code usually blends much more in and so is harder to spot.

While almost all backdoors can be found by inspecting source code or reverse engineering binaries, this manual approach does not scale well. It seems impossible to design an approach that would detect all possible kinds of backdoors, so existing backdoor detection strategies [60] focus only on specific types of backdoors. Therefore, to limit the scope of this research, only backdoors that are created by trojanizing a PE file during or after compilation are discussed in the rest of this thesis. This focus is not arbitrary, since this type of backdoor is arguably one of the most prevalent and harmful ones [24, 9].

2.2 Trojanized PE Files

Trojanized PE files are created by taking a legitimate executable and modifying it in order to make it covertly execute extra malicious code. The trojanized PE file often behaves very similarly to the original legitimate executable, which makes it very hard for victims to notice the difference. There are several reasons why a malicious actor might want to trojanize a PE file. A non-exhaustive listing of the most common motives follows.

- **Software supply chain attack**

In a software supply chain attack [61], the attacker typically compromises a software vendor in order to attack the users of its software. One of the most straightforward ways to do so is to trojanize some code that the software vendor distributes — it can be either an application or a software library. One of the biggest targets for the attackers are developers of operating systems. Successful execution of a supply chain attack against them could potentially enable the attackers to gain access to an enormous number of systems. In practice, attackers often prioritize software vendors whose software is likely to be used by the real targets and smaller software vendors whose software ends up on a large number of systems. Unless the attackers make an operational security error, it might be impossible to fully attribute such attacks to them.

- **File-infecting virus**

Many malware families propagate by infecting accessible PE files. For example, this might enable them to spread to other systems by infecting PE files on a shared network drive or when an infected executable gets copied to another system and gets executed there. The infection is performed by trojanizing legitimate files and including the malware's own code as the implanted malicious payload. Sometimes, file-infecting viruses also mutate the malicious payload for each trojanized file in order to make detection harder [18]. Since file-infecting viruses often trojanize applications that offer necessary functionality, removing them can often be much harder than just deleting the infected files. Often, a dedicated removal tool that cleans the infected files has to be developed.

- **MitM code injection**

When a PE file gets downloaded, an attacker who is able to tamper with the network traffic might be able to trojanize the downloaded executable on-the-fly [46]. For example, an attacker who compromised a vulnerable router can make the router trojanize all PE files that get downloaded over an insecure HTTP connection. While downloading executable files through HTTP is by itself not dangerous, sufficient integrity checks

have to be made before actually executing the downloaded files. Some websites that offer downloads over insecure HTTP provide cryptographic hashes of the downloaded files. If the hashes are also served through insecure HTTP, they do not sufficiently guarantee the integrity of the downloaded files, even though they might fool the vast majority of automatic injectors.

- **Trojan horse download**

Users do not always install applications by downloading them directly through the official distribution channel [8]. They might, for example, go to third-party websites that offer software downloads or get it through a peer-to-peer network. A third-party that redistributes the software can act maliciously and trojanize it. Since the trojanized application might behave exactly the same as the original benign application, users are not likely to even notice anything suspicious. An attacker could also use social engineering to get a victim to execute a trojanized application. To avoid downloading trojanized executables, it is widely recommended that users only download and install software from trusted sources [8].

- **Hiding malware**

When a PE file gets trojanized, the actually trojanized part of the PE file is often fairly small. Especially when choosing a big host PE file to embed the malicious payload in, the vast majority of the trojanized PE file will still correspond to the original benign PE file. This might make it harder for defenders to find such malware, especially since most static features of the PE file (such as the contents of the PE header) will stay the same. Therefore, attackers sometimes hide their malware by wrapping it inside an arbitrary benign PE file in an attempt to evade detection [15]. Attackers might also use this trojanizing approach for stealthier persistence. Instead of writing their persistent payload to a standalone file, they might trojanize a PE file that they know will get executed at some point.

2.3 Defense Against Trojanization

There is a mechanism designed by Microsoft to combat PE trojanization: Authenticode Signatures [42]. In a nutshell, this allows software vendors to sign their PE files and users to verify the origin of signed files. A valid signature can be used to determine the origin of the file and prove its integrity. An attacker's attempt to trojanize an already signed PE file would invalidate the signature. However, Authenticode signatures are usually not enforced, meaning the attacker can just strip the invalidated signature from the PE file. While the lack of a signature might raise some suspicion for certain

applications, there are still plenty of legitimate unsigned PE files in-the-wild, so the lack of a signature alone often cannot be used to draw any conclusions about the file's origin and integrity. However, in some cases, the lack of a signature can stop a PE file from running. With the so-called *driver signature enforcement* feature, enabled by default on 64-bit versions of Microsoft Windows, only drivers with a valid signature will be loaded by the operating system. Naturally, Authenticode can only be used to detect modifications performed after code signing. If an attacker inserts the malicious payload before signing takes place, the PE file might be signed with the backdoor included. Similarly, an attacker with access to the signing keys is able to backdoor and re-sign the trojanized PE file.

The PE header contains a checksum of the PE file, which would get invalidated by most trojanization attempts. However, it is just a simple 32-bit additive checksum, so nothing prevents an attacker from simply recalculating this checksum after trojanizing a PE file. Since the Windows loader does not check the validity of this checksum for common executables, many PE files contain an invalid checksum full of zero bytes. File-infecting viruses often recalculate the checksum, zero it out, or even use it as an infection marker to prevent multiple infections of the same file [57]. In summary, both Authenticode Signatures and PE checksums can be used as indicators of trojanization, but unfortunately, they are not always applicable.

Some application developers also implement their own custom protection against trojanization. The custom protection most often consists of computing checksums of some parts of the PE file and comparing them against predetermined values on startup. If the checksums do not match, the executable was likely patched. While this will detect most automatic trojanization attempts, a sufficiently skilled attacker who trojanizes an executable manually can bypass such protection (for example, by patching the checksum comparison itself). Custom protection against trojanization is also provided by many runtime packers and protectors [44].

2.4 Techniques for Trojanizing PE Files

As discussed, the objective of PE trojanization is to make a benign PE file execute extra malicious payloads. There are two main approaches that can achieve this goal: packing and patching.

Packing works just like a runtime packer, but with added malicious code. The executable to be infected is *packed* (or stored as data) inside of a completely new malicious executable (see Figure 2.1). When the new malicious executable is run, it first performs some malicious actions (such as infecting other files) and then unpacks the original executable, loads it, and runs it. If the time spent on performing malicious actions and unpacking is reasonably short, a victim might not notice the startup delay or any change in application

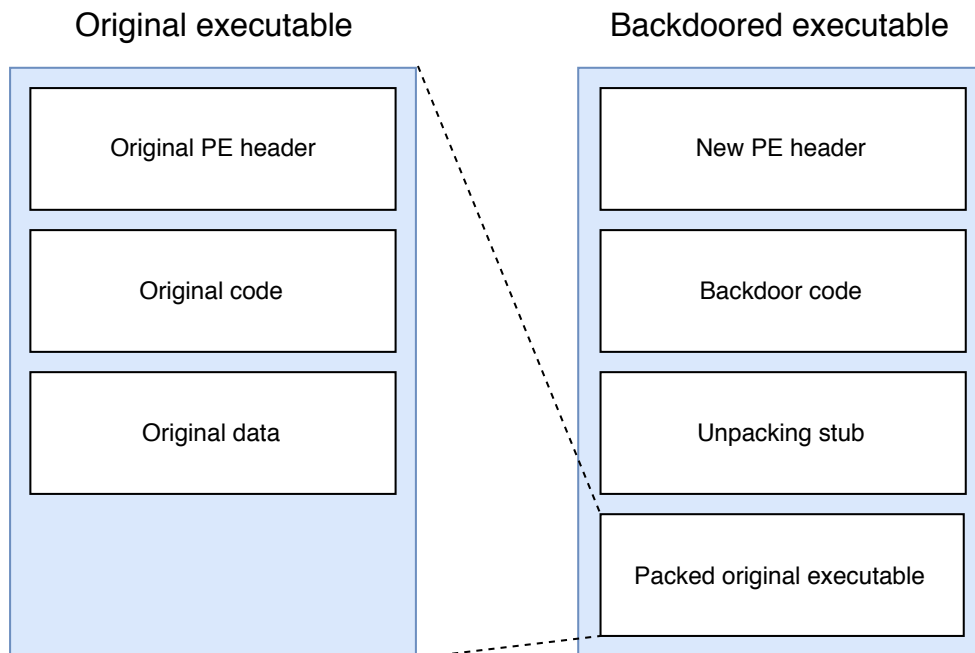


Figure 2.1: A PE file trojanized through packing. The original executable (on the left) is stored in a packed form in the backdoored executable. The backdoored executable first runs its malicious code, then unpacks, loads, and runs the original executable.

behavior. Some file-infesting viruses also run the malicious code in a separate thread in order to decrease the startup delay.

The packed executable is usually not stored in plaintext — it is often encrypted or compressed. It can be stored almost anywhere in the infected executable, but it is often located in PE resources, in the overlay, or in a brand new PE section. Storing it in the overlay is especially popular among malware authors because automatic file infection is then very easy to implement. In order to infect a PE file, malware authors can just simply prepend the executable to be infected with the malicious packer [21]. For example, a file-infesting virus called Neshta [3] uses the packing approach and stores the encrypted original executable in the overlay. The packed executable is usually executed from memory using a custom PE loader. However, some malware avoids having to implement a PE loader and just unpacks the executable, drops it to disk, and spawns a new process to execute it [46].

The packing approach completely changes what the infected file looks like, which makes it easy to notice the infection using static analysis. Simple packing also changes the icon of the PE file (because it is stored in the original executable's resources), which makes the backdoor more noticeable for the

potential victims⁸. The other approach, patching, leaves most parts of the backdoored executable intact and can thus be much more subtle and harder to detect.

Patching consists of hiding malicious code inside of the trojanized executable and subverting execution flow of the executable so that the malicious code gets executed. There are many places where the malicious code can be hidden and many techniques to subvert execution flow. The techniques used to hide malicious code and hijack execution flow are mostly independent of each other, which means that almost any combination of them can be used to backdoor an executable.

To make the backdoor harder to detect, backdoor authors attempt to make the malicious code small and hide it in a large legitimate PE file. Due to Data Execution Prevention (DEP) implementation on Microsoft Windows, the malicious code has to be placed in executable pages of memory, which can put constraints on the size of the malicious code. To get around these problems, backdoor authors often use a multistage approach. The malicious code that gets executed first is just a small stub which loads the next stage into executable memory and passes control to it. The next stage can then be hidden as data inside of the executable or even downloaded from the Internet.

Because of ASLR, backdoor authors might not have control over the location of the backdoor's first stage in memory [1]. Therefore, the first stage often consists of position-independent code (referred to as *shellcode* in the rest of this thesis). It can be hidden in the backdoored executable in one of the following ways:

- **Adding an extra PE section**

In this scenario, backdoor authors add a new executable section to the backdoored PE file and hide the backdoor code in it. To avoid having to fix many absolute and relative references, the new section is usually inserted as the last one. While this is easy to implement, it is also not particularly stealthy because the extra section is bound to raise suspicion, especially since it is executable [4].

- **Abusing code caves**

The size of the raw data of each section in a PE file has to be aligned [39]. The alignment factor is given by the `FileAlignment` field in the PE header. Its minimum, default, and most frequently used value is 512 bytes. This leads to a padding of up to 511 bytes at the end of each section. When this padding is inside of an executable section, it is commonly referred to as a *code cave*. This padding is not needed for the functionality of the program, so backdoor authors can freely overwrite it

⁸Of course, backdoor authors can avoid this problem by copying the original icon and using it in the trojanized executable.

with their malicious code. However, the backdoor's first-stage shellcode may not fit in any code cave. In that case, backdoor authors could also choose a non-executable section with a sufficiently large padding and make it executable (or split the shellcode across multiple code caves).

While `FileAlignment` refers to the alignment of each section as it is stored in the PE image, `SectionAlignment` refers to the alignment of each section when it is loaded into memory. This alignment must be larger than `FileAlignment` and is set to the page size by default (usually 4096 bytes). As a result, there are also bigger code caves of up to 4095 bytes at the end of each section. However, abusing them is slightly harder, since it requires shifting subsequent sections in the raw PE image.

- **Enlarging an existing PE section**

Backdoor authors can also create space for the malicious code by enlarging an existing section. When the backdoor is inserted after compilation, backdoor authors would probably choose to enlarge (and make executable) the last section of the PE file because enlarging a section in the middle would require them to fix all memory references to all the sections that come after it. If the backdoor is inserted during early stages of compilation, there is no such problem because the compiler will make sure that all sections are placed at proper relative addresses.

- **Overwriting existing content**

While in the previous scenarios, the backdoor authors hid the backdoor code in unused space (or created extra unused space), it is also possible that the backdoor code would replace existing code or data. For example, the backdoor code could replace rarely (or never) used code that is not critical for the program's functionality. If this is done incorrectly, it could cause the backdoored program to crash or otherwise misbehave. Doing this properly, while minimizing the impact on the backdoored program's behavior, might require the backdoor authors to have a strong understanding of the affected software.

There are also many different ways in which backdoor authors can hijack execution flow in order to execute the backdoor code. Note that if backdoor authors do not wish to break any existing functionality, effort has to be put into restoring execution context to the correct state after the backdoor code has finished execution. This is commonly done by using a pair of instructions such as `pushad` and `popad`⁹, patching instructions back to their original state, or copying and executing the overwritten instructions from elsewhere. Common methods of hijacking execution flow include:

⁹`pushad` saves the current content of general-purpose registers on the stack, `popad` loads the saved content from the stack back into the registers [20].

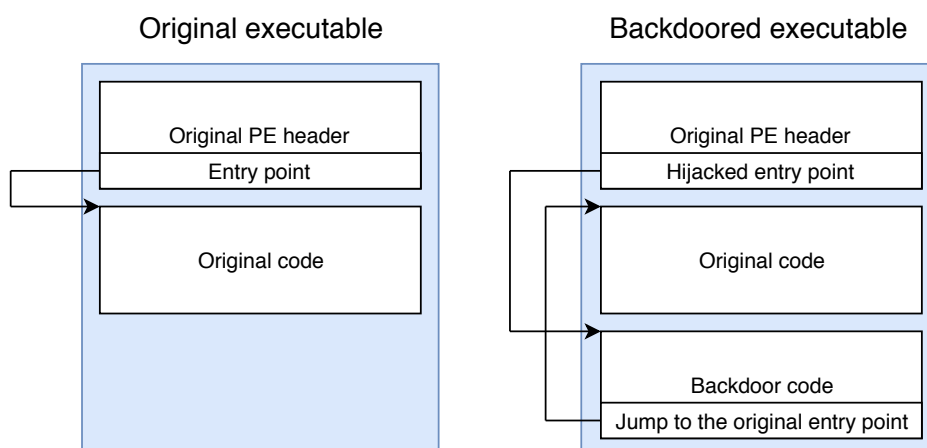


Figure 2.2: An illustration of backdooring through entry point hijacking. The pointer to the entry point in the backdoored executable is overwritten to ensure that the backdoor code gets executed first. A jump back to the original entry point is inserted into the backdoor code in order to preserve the original functionality.

- **Hijacking the entry point**

The header of a PE file contains the `AddressOfEntryPoint` field, which points to the first instruction of the PE file that should get executed [39]. Backdoor authors can modify this value to point to the backdoor code and place an unconditional jump back to the original entry point into the backdoor code (see Figure 2.2). This way, the code of the backdoor gets executed first, and after it has finished, the original program starts executing. If the backdoored executable is a DLL, backdoor authors can also similarly hijack any exported function by modifying the address of the hijacked exported function in the export table. This method is popular among backdoor authors because it is easy to implement both manually and automatically.

- **Adding a TLS callback**

The entry point is not always the first piece of code that gets executed when a PE file is loaded. The PE header can optionally contain the Thread Local Storage (TLS) directory, which contains an array of TLS callback functions [39]. These functions are meant to contain initialization for thread local storage and get executed even before the executable's entry point. Backdoor authors can add a new callback function (or hijack an existing one) in order to execute the malicious code every time the backdoored executable is run.

- **Hijacking a function pointer**

Many function pointers can often be found in a PE file. Usually, there is some piece of code that reads the function pointer and calls the function that it points to. Backdoor authors can choose to overwrite the function pointer and point it to the backdoor code. When the piece of code reads the function pointer and calls it, the backdoor code will start executing instead of the original function. In order not to break the program, the backdoor code will probably finish by restoring the execution context and jumping to the original function that the overwritten pointer used to point to. For example, the `initterm` method is included in PE files created by the MSVC compiler. It is executed during program initialization, and it processes an array of function pointers and calls the function pointed to by each one of them [37]. Backdoor authors can hijack an existing function pointer in this array or even add a new one [47].

- **Overwriting existing code**

Backdoor authors can also overwrite some existing code with the backdoor code. In order to get the backdoor code to execute, the piece of code that gets overwritten has to be something that would naturally get executed. The overwriting can, of course, cause the original program to crash or misbehave, so backdoor authors usually choose to overwrite code that is not strictly necessary for the original program's functionality, such as security checks, resource deallocation, logging, and so forth. Alternatively, some backdoor authors also store the overwritten code elsewhere in the PE file and either re-execute it from another address or patch it back in when the backdoor code finishes executing.

- **Instruction patching**

Instead of overwriting many instructions, backdoor authors might also patch just a single instruction. This is typically done by inserting a control flow altering instruction such as `jmp` or `call` in order to redirect control flow to the backdoor code. For instance, many tutorials [1, 13] on PE backdoor manufacturing recommend inserting a jump to the backdoor code directly at the entry point. Backdoor authors also often hijack an existing `call` or `jmp` instruction. If such an instruction has its target encoded inside itself, backdoor authors can just modify it and change the instruction's target to the backdoor code. In some cases, the original target of the instruction does not even have to be executed, such as when it is `__security_check_cookie` or `free`. Alternatively, the `ret` instruction sometimes gets patched into a `jmp` to the backdoor code. Even though it takes more bytes to encode a `jmp` instruction than a `ret` instruction on x86, this is not a problem, since the `ret` instruction

is often followed by a padding between functions that can be freely overwritten.

- **Object file modification**

The methods of hijacking execution flow described above were mostly concerned with backdoor authors trojanizing an already compiled PE file. But what if the backdoor gets inserted during linking? In that case, backdoor authors do not need to hijack the execution flow at the instruction level (or the PE format level). Instead, backdoor authors can just modify one or more precompiled object files and recompile them with some added backdoor code. If that precompiled object file is statically linked into the final executable, the backdoor gets seamlessly incorporated in the final PE file. The difference from the previous methods is that in this case, the backdoor code can be created at the source code level (yet the victim still cannot find the backdoor in their source code), and so the backdoor authors do not have to worry about many low-level issues outlined above (such as finding space for the shellcode or restoring execution context). An example of this would be trojanizing CRT's (C Run-time) `.lib` files, which often get statically linked into PE files produced by the MSVC linker.

2.5 Automatic Trojanization of PE Files

The methods described so far can be implemented both manually and automatically. Some of them are suitable for automatic backdooring (such as packing or entry point hijacking) while others are very hard to implement automatically (such as overwriting existing code). Backdooring an executable manually gives much more freedom to the backdoor authors, since they can use techniques tailored to the specific executable that is being trojanized, which can make the whole process much easier. However, manual backdooring is not suitable for many of the backdooring motives described in Section 2.2. Often, backdoor authors want to trojanize executables in real-time or in large quantities, and that cannot be performed manually.

While many advanced malicious actors will perform trojanization manually or with custom tools, there are also publicly available tools for trojanizing PE files. Even advanced malicious actors might use these tools sometimes because it saves time and makes their attack look less professional (and makes attribution harder). The main goal of this thesis is to create a framework for heuristically detecting selected types of backdoors. The detection of backdoors created with a specific tool is not an objective, since these backdoors can be easier detected with a detection dedicated just to that specific tool. However, a brief overview of two of the most widely used backdooring tools is given, since they are good examples of how backdooring works in practice.

2.5.1 The Backdoor Factory

The Backdoor Factory [45] is an open-source tool for trojanizing PE, ELF and Mach-O executable binaries. It can backdoor them either with a custom user-supplied shellcode or with some standard default shellcode payload (e.g., configurable reverse shell). The Backdoor Factory uses the *patching* trojanization approach discussed in Section 2.4. To hijack execution flow, it uses the *instruction patching* technique. Specifically, it patches the instruction(s) directly at the PE entry point into a `jmp` instruction pointing to the embedded shellcode.

The Backdoor Factory hides the shellcode in code caves by default, but it can also be configured to hide it in an extra PE section. It lists the available code caves to its user, who can choose which specific code cave should be used. If the shellcode does not fit into any one code cave, it is also possible to split it across multiple code caves. When a code cave is chosen, the shellcode is written into it, and the PE section containing the code cave is made both writable and executable.

When The Backdoor Factory trojanizes a signed PE file, it strips its digital signature by default because the trojanization would invalidate it anyway. Interestingly, it neither recalculates nor wipes the 32-bit checksum in the PE header, so the backdoored executables contain an invalid checksum.

2.5.2 Shellter

Shellter [29] is a tool for trojanizing 32-bit PE files with shellcode. As is the case with The Backdoor Factory, the trojanized PE files can be infected with an arbitrary custom shellcode, but there is also a small library of standard shellcodes for convenience. Shellter infects PE files using the *patching* approach, more specifically by overwriting existing code. Before trojanization, it first obtains an instruction trace of the PE file that it intends to infect and uses the collected instruction trace to select pieces of code that can be overwritten with the shellcode. However, it does not overwrite this code directly with the user-selected shellcode¹⁰. Instead, it uses a multistage approach where the sole purpose of the first stage is to decrypt the second stage, load it into executable memory, and pass control to it. The second stage then runs the user-selected shellcode in a new thread and patches the overwritten code in memory back to its original clean state. Finally, it restores register values using the `popf` and `popa` instructions and jumps back to the restored code in order to preserve the program's original functionality. The original code that was overwritten with shellcode is stored in the last section of the trojanized PE file, which is enlarged in order to accommodate it.

¹⁰This is only true if Shellter's *stealth mode* is enabled. If it is disabled, the shellcode injection is much simpler, but the original functionality of the infected PE file will not be preserved.

2. BACKDOORED PE FILES

Since Shellter does not wish to add new imports to the trojanized executable (or use an import-less technique [55] such as iterating over the `InLoadOrderModuleList` in order to resolve imports), it abuses functions that are already imported by the targeted executable. For example, if the targeted application already imports the `VirtualAllocEx` function, the first-stage shellcode can use it to allocate executable memory for the second stage. There are currently eight methods to allocate executable memory (some of them require multiple imported functions such as `HeapCreate` and `HeapAlloc`). If the program does not import functions to satisfy any of those eight methods, it will default to making an existing section writable and executable.

Shellter attempts to make its trojanization hard to detect, so it can be configured to generate polymorphic code for the first and second stage. Interestingly, the first stage is not position-independent, and Shellter tackles this problem by clearing the `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE` flag in the PE header, thus making the whole trojanized PE file not relocatable. Shellter strips away the digital signature of trojanized executables, and it also recomputes the PE checksum.

Backdoor Case Studies

This chapter focuses on the analysis of several real-world backdoors and examines how the findings from the previous chapter apply to them. The backdoors that are analyzed here are widespread legitimate applications that were trojanized by a malicious actor in a supply chain attack. Most of them went undetected for months, which shows that detecting such backdoors is extremely hard, and that there is a lot of room for research about how to detect them sooner and thus mitigate the harm caused by them. It is even possible that there are more such backdoors currently active out there that the public still does not know about. One of the main goals of this thesis is to attempt to find them and make it easier for other researchers to search for such backdoors in the future.

All of the backdoors analyzed in this chapter are multistage backdoors, where the later stages are usually either embedded in an encrypted form or downloaded from the Internet. Since this thesis aims to create heuristics for finding unknown backdoors statically (i.e., without running the potentially trojanized application), the later stages are not a focus, since they are usually either not present in the analyzed PE file or extremely hard to distinguish from some random data. Instead, a particular focus is given to anomalous static properties of these backdoors that can be used to build heuristics for their detection.

3.1 ShadowHammer

ShadowHammer [25] was a supply chain attack against the Taiwan-based hardware manufacturer ASUS. Researchers from Kaspersky Lab found that malicious actors trojanized the *Asus Live Update Utility*, which comes pre-installed on almost all ASUS computers and is supposed to automatically update drivers and various other pieces of software [24]. The trojanized executables were digitally signed by ASUS and downloadable from their official update servers. While first trojanized versions of the utility started to be

delivered to their victims back in June 2018, this attack was only discovered in January 2019, which means that it took about seven months to uncover it. The backdoor’s command and control (C&C) domain `asushotfix.com` was registered at the start of May 2018, which shows that the attack must have been planned for quite some time in advance.

Even though the backdoor was installed on a large number of computers, it was never activated on the vast majority of them. The reason for this was that the backdoor first hashed the potential victims’ MAC addresses and searched for them in a whitelist that was embedded in the backdoor. Only if at least one MAC address was whitelisted did the backdoor continue execution by downloading its next stage from the C&C server¹¹. It is not clear how did the backdoor authors obtain their targets’ MAC addresses, but the list of whitelisted MAC addresses changed over time, and there were only about 600 targets in total [25].

The trojanization was performed by patching an outdated version of the utility from 2015. The malicious actors trojanized this outdated version with their malicious payload, re-signed it, and substituted the legitimate binary with it. This made ASUS unwittingly start distributing the trojanized binary to its users. Interestingly, even though the trojanized executable was served to the users as an update and used an up-to-date certificate, it still retained its original compilation timestamp from 2015 in its PE header.

The backdoor authors used two different techniques to trojanize the same executable. The older technique consisted of overwriting the `WinMain` function. The backdoor authors later switched to a more stealthy technique where they hijacked an existing `call` instruction that initially pointed to the `crtCorExitProcess` function.

3.1.1 The First Trojanization Technique

In the first trojanization technique, the malicious actors overwrote the `WinMain` function with a custom shellcode loader. Since `WinMain` was the function responsible for dispatching execution to the rest of the original code, this naturally completely broke the original functionality of the backdoored application. As a result, most of the original code in the backdoored application would never get executed under normal circumstances, and so the trojanized application did not appear to do anything useful.

The shellcode loader that replaced the `WinMain` function was extremely simple. It allocated writable and executable memory using `VirtualAlloc`, copied the second-stage shellcode into it, and started executing it from a certain hardcoded offset. The second-stage shellcode was embedded in the executable by overwriting an existing PE resource, and it was wrapped in a

¹¹For some targets, the whitelist consisted of pairs of MAC addresses, and the victim had to have both of them for the malware to activate.

PE file of its own¹². The address of the resource that contained the second stage was not obtained using the standard functions such as `FindResource` and `LoadResource`. Instead, the backdoor authors just hardcoded an address to the resource directly in the code of the backdoor's first stage.

Since the backdoor's first stage is smaller than the original `WinMain` function, it did not overwrite the whole `WinMain` function. The backdoor authors decided to fill the rest of the available space with 59 `0xCC` bytes (which are commonly used as padding between functions by many compilers¹³). Such an unconventionally large padding can be considered an anomaly, since padding between functions in the backdoored executable is otherwise only used to align the start of each function to sixteen bytes. Even though other executables created with other compilers (or with other compiler settings) can produce larger paddings, such paddings would generally be used to align the start of the function with an alignment factor that is a power of two. In this case, there is a padding of size `0x3B`, and the next function starts at an address whose hexadecimal representation ends with `A10`, so that is clearly not the case here.

Comparing the trojanized executable with the original benign executable shows that the backdoor authors did not tamper with the vast majority of its content. There are only a handful of patches that were performed in order to embed the malicious payload. The content of the `WinMain` function was replaced with the backdoor's first stage, and an existing PE resource was partially replaced with the backdoor's second stage. These changes also prompted the backdoor authors to recompute and patch the PE checksum and to re-sign the executable. Finally, the backdoor authors also replaced three relocation entries. This was necessary since the executable was relocatable, and the backdoor's first stage was not position-independent. The backdoor's first stage contained the aforementioned hardcoded address to the second-stage shellcode and calls to imported functions through addresses in the import address table (IAT). Since those addresses would change if the executable was not loaded to its preferred base address, the backdoor authors made them relocatable by hijacking existing relocation entries¹⁴.

As can be seen, this trojanization technique was not particularly stealthy. First of all, the trojanized executable did not retain its original functionality, which might have caused someone to start investigating why. Secondly, the malicious code was located in the `WinMain` function, which is the function that most advanced disassemblers such as IDA Pro show at first to its users, which

¹²This PE file was not loaded or otherwise used, so the second stage can just be considered shellcode with an extra redundant PE header.

¹³`0xCC` is an opcode for the `int 3` instruction on x86, which is often used as a software breakpoint.

¹⁴These hijacked entries initially belonged to some benign code. Since this code would never get executed anyway in the backdoored executable, this modification did not break any functionality.

3. BACKDOOR CASE STUDIES

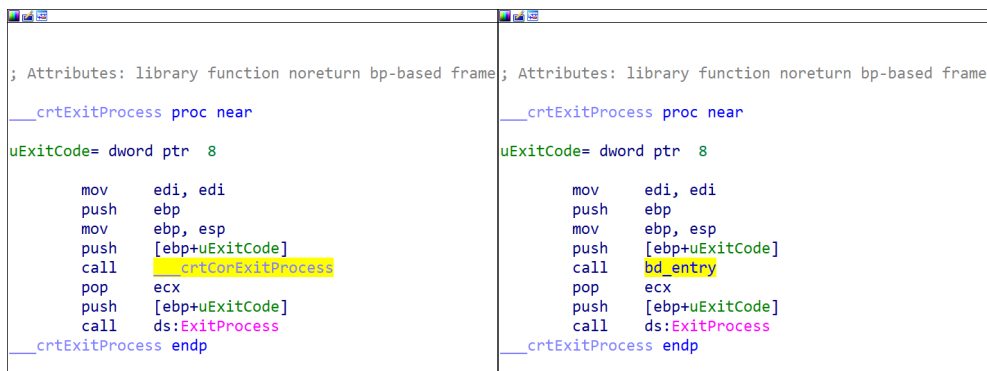


Figure 3.1: The `__crtExitProcess` function in a clean version of *Asus Live Update* (on the left) and in a trojanized version (on the right). The only difference is the fifth instruction, which used to call `___crtCorExitProcess` but was patched to call the embedded malicious code.

means that if someone actually reverse engineered the trojanized executable, there was a high chance they would have immediately noticed the backdoor. Finally, the backdoor’s second stage was stored unencrypted in PE resources, which made it possible to detect its code using static heuristic detections. Amusingly, the backdoor authors even forgot to wipe the PDB path¹⁵ of the second stage: `D:\C++\AsusShellCode\Release\AsusShellCode.pdb`.

To make the backdoor less noticeable, the backdoor authors later switched to the second trojanization technique in which they encrypted the second-stage shellcode, hid the first stage in a code cave, and did not completely destroy the original application’s functionality.

3.1.2 The Second Trojanization Technique

As was discussed, the first version of the backdoor was not particularly stealthy. The backdoor authors probably realized this and consequently implemented an upgraded trojanization technique in which they continued to backdoor the same legitimate executable with the same malicious payload but in a less noticeable way. Namely, they redirected control flow by patching a single `call` instruction in the `___crtExitProcess` function (see Figure 3.1). That call instruction used to point to the `___crtCorExitProcess` function, but it was patched to point to the backdoor’s first stage instead, which was located in a code cave at the end of the `.text` section.

The backdoor’s first stage starts out by calling `GetModuleHandle` in order to retrieve the main module’s base address. This is the only position-

¹⁵The PDB path is a filesystem path to a file containing debugging information. It is often removed from executables by malware authors because it can hint at the project name or even at the malware author’s identity.

dependent action — all subsequent addresses are obtained relatively by adding offsets (relative virtual addresses) to this base address. For example, the second stage of the backdoor is found at RVA `0x16EC78` from the base address, and `VirtualAlloc`'s IAT entry is located at RVA `0x11C27C`. The backdoor then proceeds to allocate writable and executable memory, copy the second stage into it, decrypt it using a custom xor-based cipher, and finally execute it.

Same as in the first trojanization technique, the trojanization was performed by in-place patching of a benign executable. This means that the size of the executable did not change (disregarding the overlay which contained the Authenticode signature) and that simply comparing a trojanized executable with the original benign executable will reveal all of the modifications performed by the backdoor authors. The most evident modifications are patching the `call` instruction in `___crtExitProcess`, overwriting a code cave with the backdoor's first stage, and partially overwriting an existing PE resource with the encrypted second stage. The backdoor authors also patched an existing relocation entry so that the call to `GetModuleHandle` would work regardless of where the code would be based because of ASLR. Interestingly, they also modified an existing string from `ASUS Live Update` to `ASUS Live Updata`. There is code that deletes a registry value named like that from the registry key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, so it seems this modification was performed in order for the backdoor to remain persistent. Finally, the backdoor authors also fixed the PE checksum and the Authenticode signature.

The most evident anomalies can be found in the `___crtExitProcess` function. While this function should normally call `___crtCorExitProcess` and `ExitProcess`, an unknown function is called in the trojanized samples instead. This unknown function is located at the end of the code section (which can generally indicate that it might have been patched into a code cave), and the usual target, `___crtCorExitProcess`, is located right before the `___crtExitProcess` function in memory. This is not a coincidence, since both functions come from the same module and are normally executed together, so it makes sense for the linker to put them close together. The `___crtCorExitProcess` is still present in the backdoored executable even though it is never called, which can also be considered unusual, since an optimizing compiler tends to remove unused functions from the final executable.

The backdoor's entry point is called in exactly the same way as the `___crtCorExitProcess` function due to the way that execution flow was hijacked. It is called with a single argument, an exit status code. However, this argument is never used inside the backdoor code. This could actually be a common pattern in executables trojanized by patching of a `call` instruction. The code of the backdoor should generally be independent of the rest of

the trojanized executable, so it should not use any arguments passed to it. However, the patched call instruction might use some arguments, which can result in a mismatch between the number of arguments a function takes and the number of arguments a function uses. Such a mismatch can also happen in legitimate code, which makes this observation only useful in combination with other heuristics.

3.2 Asian Gaming Industry Incidents

In March 2019, researchers from ESET uncovered a supply chain attack [33] against three separate Asian companies in the gaming industry. This attack resulted in all three of them distributing signed and trojanized binaries. The trojanization was performed the same way in all cases, and the embedded payload was also the same (except for the backdoor configuration, which was target-specific). First trojanized binaries were distributed to users in November 2018, so this attack went undetected for approximately four months.

Trojanization was achieved by hijacking execution flow almost directly at the PE entry point. The entry point of binaries built with MSVC commonly contains a call to `___security_init_cookie` and a jump to `___tmainCRTStartup` (see Figure 3.2a). However, there was an extra function in the trojanized binaries (see Figures 3.2b and 3.2c). This extra function was executed instead of `___tmainCRTStartup`, and it contained a call to the backdoor's first stage followed by the expected jump to `___tmainCRTStartup`. This trojanization method effectively injected the backdoor to the CRT startup and ensured that the backdoor will always get executed before the rest of the trojanized application.

The backdoor's first stage is located at the start of the executable `.text` section. It consists of 16 functions, making it more complex than ShadowHammer's first stage, which contained at most two functions. The first stage starts out by finding the base address of `kernel32.dll` by iterating over the `InInitializationOrderModuleList`. It uses this base address to resolve addresses of several functions exported by `kernel32.dll`. The configuration of the first stage and the encrypted second stage is located right after the first stage code in memory. The first stage reads an RC4 key from this configuration and decrypts it with a simple xor cipher [34]. The decrypted key is then used to decrypt the second stage, which is a PE file that is supposed to be loaded in memory and executed.

The second stage creates a new thread for the backdoor code to run in, since it does not want to block the original functionality of the backdoored application. It does not perform any malicious activities when analysis tools such as `procmon.exe` are running or when the system language is Russian or Chinese [33]. It then collects information about the infected machine (including the victim's MAC address) and sends it to its C&C server. The

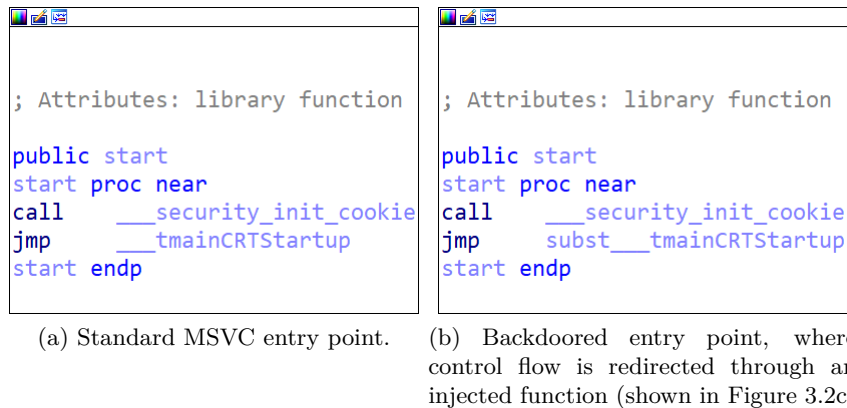


Figure 3.2: The hijack method used in the Asian gaming industry incidents.

C&C server analyzes the collected information and responds either with the third stage or with a command to deactivate.

In the ShadowHammer case, it was clear that the backdoor code was patched in after compilation and that it did not originally belong to the backdoored executable. The Asian gaming industry incidents backdoors were different because there was no sign of patching and the backdoor itself was located at the start of the `.text` section. It actually turned out that the backdoor was inserted during linking because the build environment itself was trojanized [25]. More specifically, a malicious DLL was injected into the linker process, and it hooked the `CreateFile` function. During its standard operation, the linker reads some `.lib` files that contain compiled library code and statically links their content into the produced executable. The hook redirected the linker from reading a legitimate `.lib` file into reading a maliciously modified `.lib` file, and this made the linker produce executables that had the backdoor seamlessly incorporated in them¹⁶.

On one hand, it does seem that this backdoor would be easy to notice. The execution flow hijack happens very close to the entry point, and the backdoor code is located at the very start of the first section of the trojanized executables. But reverse engineers might not even view the entry point — it is very common to start analyzing unknown samples from the `main` function,

¹⁶The hook also made sure that the linker included the backdoor only in the targeted executables. It did not perform any redirection when other executables were being built.

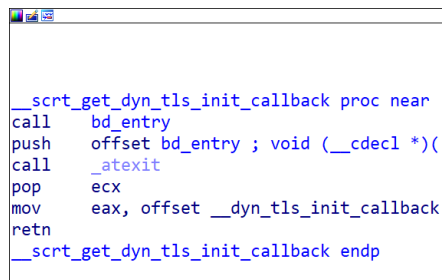
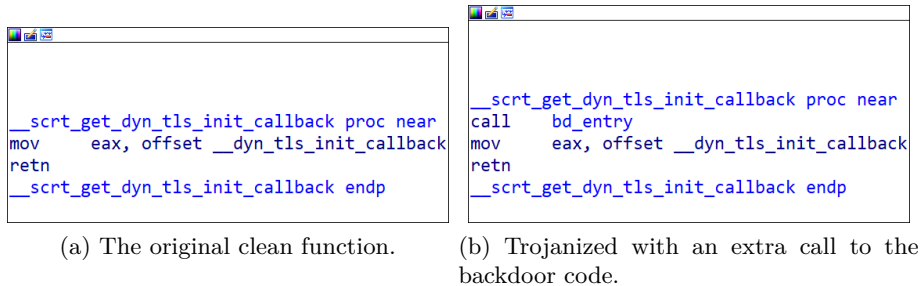
since the CRT startup code is often not very interesting from the reverse engineer's point of view. Advanced disassemblers such as IDA Pro often attempt to find the `main` function automatically and present it to its users at first instead of the entry point. Furthermore, even if the reverse engineer did view the entry point, it contains the typical `call`, `jmp` sequence, so it looks legitimate at first glance because one might not notice that the `jmp` does not point to its usual destination. However, the entry point is definitely anomalous. When there is a call to `___security_init_cookie` directly at the entry point, one can assume that the rest of the entry point will look like standard CRT startup. Since there is an extra injected function in the trojanized samples, their entry point differs significantly from standard CRT startup, which allows building heuristics for detecting similar anomalies.

As discussed, the second stage of the backdoor is stored in the executable `.text` section. Since it is encrypted with RC4, from the perspective of static analysis, it is just a high-entropy data blob that is indistinguishable from random data. It is unusual for such large high-entropy data blobs to reside in executable sections, since they are supposed to be stored in nonexecutable data sections. However, heuristics that look for suspiciously large high-entropy data blobs in executable sections would probably have a lot of false positives. For instance, some of them could be caused by cryptographic libraries that store constants or lookup tables close to the code that is using them.

The first stage of the backdoor also makes use of techniques that are typical for malicious loaders. This includes resolving imported functions by hashing their names, finding base addresses of loaded DLLs by iterating over structures in the Process Environment Block (PEB), loading PE files in memory, or obtaining the current instruction pointer by executing a `call` instruction followed by a `pop` instruction. Since the first stage is unencrypted in the `.text` section, the usage of these techniques can be detected both statically and dynamically. While their usage is, of course, no direct evidence of maliciousness, these techniques are very common among malware and rare among legitimate executables. Therefore, when the usage of such techniques is detected in a legitimate-looking executable, it might be worth looking into the reason why such a technique should be used, especially if its usage was not observed in previous versions of the same software.

3.3 The CCleaner Incident

Another supply chain attack that is worth discussing here is the CCleaner incident [16]. In this incident, malicious actors implanted a backdoor into CCleaner executables that were released on August 15, 2017. The backdoor was discovered almost a month later, on September 12. Even though the backdoor was found on a large number of machines, it targeted only a tiny fraction of them and remained dormant on the rest of them. The targeting



(c) Trojanized in such a way that the backdoor code also gets executed when the program normally terminates.

Figure 3.3: The first hijack method used in the CCleaner incident.

was implemented by collecting information about the infected machines (such as usernames, MAC addresses, lists of installed programs, and so forth) and exfiltrating this information to a C&C server. The server processed the collected information and used it to decide whether a machine is targeted (more specifically, whether it should receive and execute the next stage). Since the C&C server was seized and analyzed, we know that this decision was based solely on the domain name assigned to the potential victim¹⁷ and that the attacker’s ultimate goal was to breach the perimeter of select companies [9].

As was the case in the Asian gaming industry incidents, the backdoor was seamlessly incorporated in the trojanized executable. The trojanization did not leave any signs of patching behind, and it affected CRT code, which suggests that the backdoor was inserted during the build process. The backdoor code was launched by hijacking execution flow in CRT code even before the `WinMain` function. The specific execution flow hijack method varied in individual backdoored executables. In the main CCleaner binary, the `__sCRT_get_dyn_tls_init_callback` function was modified. While this function usually just returns a function pointer (see Figure 3.3a), it contained an extra call to the backdoor code in the trojanized executables (see

¹⁷This domain name was retrieved using the `GetComputerNameEx` function with `NameType` equal to `ComputerNameDnsDomain`.

3. BACKDOOR CASE STUDIES

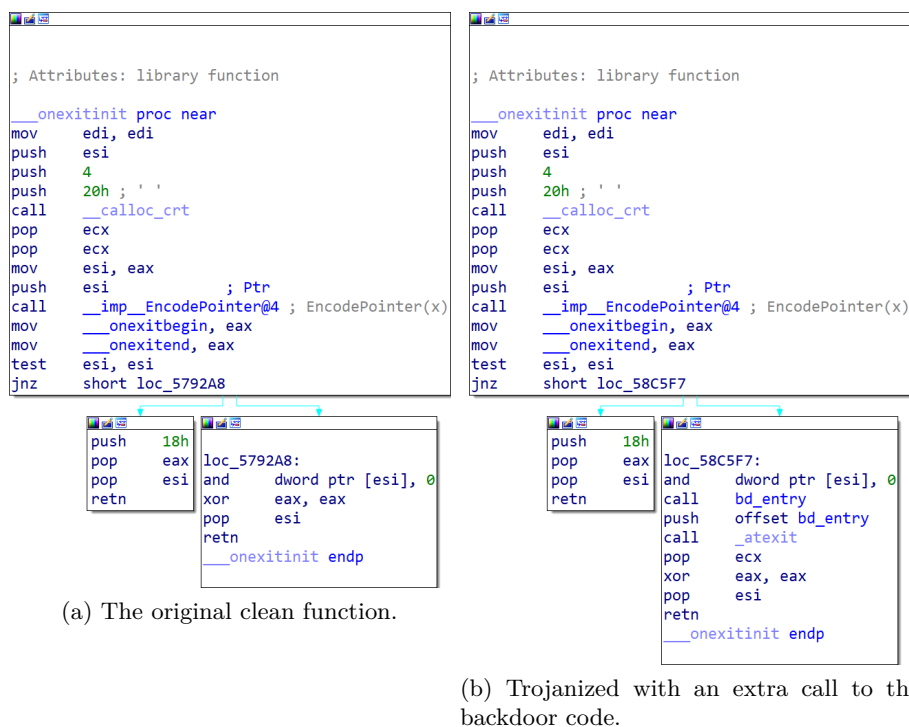


Figure 3.4: The second execution flow hijack method used in the CCleaner incident. Note that the left branch contains extremely rare error handling, so the backdoor gets executed virtually every time along with the `__onexitinit` function.

Figure 3.3b). In CCleaner Cloud, the `__sCRT_get_dyn_tls_init_callback` function was backdoored similarly, but the backdoor code was also registered to get executed on process exit (see Figure 3.3c). Another execution flow hijack method targeted the `__onexitinit` function. A call to the backdoor code was added at the end of this function along with a call to `_atexit` so that the backdoor would also run when the program terminates (see Figure 3.4).

The first stage of the backdoor is extremely simple. All it does is decrypt a shellcode located in the `.data` section, copy it into executable memory, and execute it. Executable memory is obtained by creating a custom executable heap using the `HeapCreate` function with the `HEAP_CREATE_ENABLE_EXECUTE` flag and allocating a chunk of memory from it using `HeapAlloc`. This heap is used only for this single allocation. It is destroyed after the shellcode returns, and its content is wiped by overwriting it with zero bytes. The shellcode’s purpose is to load a PE file that is located right after it in memory. The PE file to be loaded does not have any `IMAGE_DOS_HEADER`, and its full PE header is wiped in memory once it is fully loaded.

The loaded PE file contains most of the backdoor's functionality. Its code is executed in a separate thread so that the main thread can return and does not block the rest of the backdoored application. The new thread starts out by attempting to evade sandbox analysis. Then it proceeds to collect information about the victim and to exfiltrate this information to a hardcoded IP address via HTTP¹⁸. A response from the C&C server is then decoded and decrypted, and if it contains shellcode of the next stage, it will get executed.

The backdoor uses several techniques that are typically found in malicious software. It loads a PE file in memory and wipes its PE header in an attempt to hide it. It uses string encryption and wipes the decrypted strings in memory once they are no longer needed. It uses anti-sandbox tricks in an attempt not to exhibit malicious behavior in an analysis environment. It uses a domain generation algorithm (DGA) if the primary C&C IP address is unresponsive. The usage of such techniques in an executable with a trusted Authenticode signature should raise suspicion and ideally launch an investigation to find out why they are used.

The fact that the `__sCRT_get_dyn_tls_init_callback` function calls an unknown function in the backdoored executables can be considered anomalous. Microsoft even distributes the source code of this function with the MSVC development environment, and it can be seen in the `dyn_tls_init.c` source file that this function is supposed to just return a single function pointer. However, someone could change the implementation of this function for legitimate reasons, and that is why the call can only be considered anomalous (and not definitely malicious) until it is actually reverse-engineered and found to be anomalous for malicious reasons. In some backdoored executables, the backdoored `__sCRT_get_dyn_tls_init_callback` function is located with the rest of the backdoor code at the very start of the `.text` section. This function is otherwise usually located along with the rest of the CRT startup code, close to the function that calls it. This seems to be a result of the way the backdoor was inserted. The `__sCRT_get_dyn_tls_init_callback` symbol was likely defined in a different object file than usually, which resulted in it being placed in an entirely different location in memory.

Another interesting anomaly that indicates the way the backdoor was inserted can be found in the Rich header. The Rich header of the compromised version of CCleaner differs from the Rich header in preceding and consecutive clean versions. Specifically, it logs one new object file built with Visual Studio 2010 (which was otherwise not used at all according to the Rich header) and one less object file built with Visual Studio 2015. Unfortunately, it is impossible to say for sure if that one object file is the backdoor itself from just the executable, but it seems very probable that the backdoor was built separately using Visual Studio 2010 on an attacker's machine and it was then

¹⁸If the hardcoded IP address does not respond, a fallback domain generation algorithm (DGA) will be used.

used to replace the legitimate object file (which was built using Visual Studio 2015) on the build machine.

3.4 ShadowPad

In July 2017, software distributed by NetSarang was compromised in yet another supply chain attack [23]. NetSarang is a vendor of server management tools, so this attack was probably targeted at larger businesses that would be likely to use the affected software. This supply chain attack was first detected by the cybersecurity company Kaspersky after investigating suspicious DNS requests used by the backdoor for C&C communication [23]. As is the case with the previously described attacks, the backdoor did not fully activate on most of the machines that it found itself on. Conditional targeting was implemented by exfiltrating information (such as domain names, hostnames, and so forth) about potential targets to a C&C server. Based on that information, the C&C server selectively responded with an activation key only to the systems that were really targeted. If a victim did not receive the activation key, the backdoor did not perform any additional malicious actions. Unfortunately, it is not publicly known how many times was the backdoor activated, but at least one case was confirmed in a company in Hong Kong [23].

The backdoor itself was hidden in the `nsock2.dll` library. This library exposes a higher-level networking API and is used by multiple NetSarang tools, which might be the reason why the attackers chose it for trojanization. Backdooring a single DLL allowed them to compromise all NetSarang applications that used it, and the network connections initiated by the backdoor were more likely to blend in, since the library was legitimately supposed to be used for networking.

The malicious code is started from the `_initempty` function. This function is called during CRT initialization even before the `DllMain` function. It processes an array of function pointers and calls each one of them. The attackers managed to insert a pointer to the backdoor's entry point into this array, which caused the `_initempty` function to call it. There are no signs of patching, which suggests that the backdoor was inserted during the build process. However, it is also possible that the backdoor was inserted directly into the source code, since functions can also be registered to be called from `_initempty` in source code.

The backdoored version of `nsock2.dll` contains only two extra malicious functions. They are responsible for allocating writable and executable memory, decrypting a shellcode from the `.rdata` section into it, and executing it. This shellcode is obfuscated and uses simple anti-disassembly techniques [54]. Its purpose is to load the next stage, which is in a format very similar to the PE format but with a different header layout. The next stage

obtains a domain name using a DGA with the current month as an input [26]. It then prepends a random-looking subdomain made up of encoded exfiltrated data to the generated domain name. A DNS query is performed for the TXT record of this domain name every eight hours. The DNS response can contain an activation key that would be used to decrypt the backdoor's final stage.

The final stage is essentially a modular remote administration tool (RAT) that gives its operators full control over the infected machines. It is interesting that this RAT is hidden under several layers of encryption in the backdoored `nsock2.dll` file, especially since the other analyzed backdoors tended to download the later stages only for the targeted victims. On one hand, it seems that this allows anyone to analyze the later stages in order to tell the backdoor's real purpose and to attempt to attribute the attack. However, the final stage is encrypted with an activation key that only the C&C server can provide, and decrypting it without this key is not trivial. There might also have been a technical reason for this, since the DNS-based C&C communication protocol is not suitable for transporting a large amount of data, so the final stage would either have to be transported over multiple DNS queries (which would be even more likely to trigger some alert) or over another C&C communication channel.

The legitimate version of `nsock2.dll` is relatively small at around 110 KB. The backdoored version contains a lot of malicious code in addition to everything the legitimate version contained, so it is not surprising that it is larger at around 180 KB. This shows that tracking changes in size of PE files can also help detect potential backdoors: it is suspicious if there is a substantial increase in size that is not accompanied by significant changes in the source code or in the compiler settings. Indeed, when the trojanized DLL file is compared to the last legitimate version preceding it, there are no significant changes in functionality. Upon further investigation, it can be seen that the added 70 KB mostly correspond to random-looking high-entropy data, which is another red flag that points to potential trojanization.

As is the case with the previously described backdoors, ShadowPad utilizes several techniques commonly associated with malicious software. There is obfuscation, anti-disassembly tricks, string encryption, custom executable formats, DGA, and more. Since the vast majority of the backdoor code is encrypted and is only decrypted at runtime, it is almost impossible to detect the usage of these techniques by purely static heuristics. However, dynamic analysis can be very effective at finding such backdoors. In fact, what led to ShadowPad's discovery was its anomalous abuse of the DNS protocol for C&C communication.

Trojanization artifacts can also be found in the Rich header, similarly to the CCleaner incident. Legitimate `nsock2.dll` binaries released years before or years after the supply chain attack all have the very same Rich header. However, Rich headers of trojanized `nsock2.dll` binaries differ because they contain one extra field corresponding to C++ code compiled

with Visual Studio 2012. While it cannot be confirmed beyond any doubt, it is almost certain that this extra field corresponds to the backdoor.

3.5 Summary

Four of the most impactful recently discovered application backdoors were analyzed in this chapter. While each of them contained some unique features, they all followed a similar formula. It sounds reasonable to expect the same to hold for potential future backdoors as well. They will all probably contain some unique features because their authors want them to evade detection mechanisms based on previously discovered backdoors. However, they will probably also share similarities at a higher level because some techniques have proven to be effective for achieving the attackers' ultimate goal: infecting select users of some software in a stealthy way. This section focuses on those high-level similarities, since they can be useful for building heuristic detections for unknown backdoors.

All of the discussed backdoors implemented some way to target only a subset of the users of the trojanized application. This was probably done in order to make the backdoor harder to find because each time the backdoor is activated, there is a certain chance that it will get discovered. Additionally, since a typical malware sandbox will likely not be targeted, this also lowers the amount of suspicious behavior that would be exhibited when the trojanized application gets executed in a sandbox. In most cases, the targeting was performed by sending a profile of the potential victim to a C&C server and letting the server decide whether the specific victim is to be targeted or ignored. The only exception is the ShadowHammer backdoor, which contained the targeting logic hardcoded in the trojanized executable and would only infect victims with specific whitelisted MAC addresses.

In most cases, the backdoor was likely inserted during the build process. This also meant that the attackers did not have to worry about signing and delivery of the trojanized executables, since it would get done automatically during the release engineering process. The only exception to this is again the ShadowHammer backdoor, which was trojanized by patching an old legitimate build.

All of the backdoors also implement techniques that are often used by other malware. Backdoor authors use these techniques to make the backdoors harder to detect and analyze, but fortunately, this is a double-edged sword, since the usage of such techniques can also lead to discovery, especially by security software protecting one of the targeted victims. However, most security companies suppress detections that detect PE files with trusted Authenticode signatures in order to prevent disastrous false positives. These backdoors underline the need to always analyze the root cause of suppressed detections,

as the suppressed detection might have actually been caused by an executable trojanized in a supply chain attack.

The discussed backdoors also all featured a similar multistage design. They started with an execution flow hijack from CRT code. This hijack would trigger execution of the backdoor's first stage, which was often extremely simple, and its only purpose was to decrypt the next stage in memory and pass execution to it. Consequently, the first stage was the only malicious code that could be analyzed automatically statically, since all of the following stages were only decrypted in memory. Due to the way the application was trojanized, the first stage was sometimes located at the very start or at the very end of the executable `.text` section. The first stage was also often independent of the execution context from the rest of the trojanized application, which means that the backdoor would execute the same regardless of the way the execution flow had been hijacked.

The second stage was either shellcode or a PE-like payload with a custom in-memory loader. Its primary purpose was to identify whether the infected machine was targeted and if so, deliver the third stage of the backdoor. In order to do that, it communicated with its C&C server and was sometimes executed in a separate thread so that the network latency of C&C communication would not slow down the trojanized application.

While the first two stages were often relatively small and did not contain many distinguishing characteristics (likely in an attempt to make attribution and detection harder), the third stage was a much more complicated piece of malware. It was usually only delivered to the targeted victims from the C&C server, so it was relatively protected, since the C&C server could be shut down after discovery. Unfortunately, this means that actually finding the third stage is a hard task that might not be possible anymore after the C&C server stops responding.

Call Graph Extractor

Several heuristic detections for anomalous PE files are implemented in the practical part of this thesis. Since these detections are meant to operate on hundreds of thousands of files, it would not be feasible to run them directly on the input PE files. Therefore, the practical part of this thesis is split into two parts: preprocessing and heuristic detections. During preprocessing, each sample is analyzed statically, and certain properties are extracted from it. These properties are saved in a database that serves as an input for the actual heuristic detections.

The main advantage of this approach is that each sample has to be processed only once. On the other hand, plenty of information about the samples is lost during preprocessing. Therefore, it is crucial to discuss what properties of the analyzed samples are extracted so that the desired anomalies are still visible in the extracted data.

This chapter details the preprocessing stage. First, there is a discussion about the selection of properties to be extracted. Then, the actual implementation of the preprocessing stage is described.

4.1 Design of the Extractor

The main properties that are extracted from the analyzed PE files are functions and control flow relations between them. Essentially, the extractor attempts to reconstruct the static call graph of the analyzed samples. A static call graph is a directed multigraph in which subroutines of a program are modeled as vertices, and there is an edge if and only if the subroutine corresponding to the source vertex may call the subroutine corresponding to the destination vertex [30]. Note that while call graphs are usually constructed from source code, the extractor described here attempts to construct them from PE files. Unfortunately, this means that it is not always possible to construct an accurate call graph. Some information is lost during compilation, and statically deducing the possible destinations of indirect `call`

and `jmp` instructions is difficult, so the extracted call graph is at best only an approximation of the real call graph of the analyzed program.

The reconstructed call graph should serve as a basis for detecting anomalous CRT functions. As was shown in the previous chapter, many backdoors trojanized legitimate CRT functions by either injecting a call to the backdoor code or by hijacking an existing call in order to redirect it to the backdoor code. From the call graphs of many legitimate samples, it should be possible to deduce the usual targets called from the analyzed CRT function and report samples in which an unusual target function is observed.

4.1.1 Selection of Extracted Properties

To obtain a call graph, the extractor has to first identify functions and their boundaries, which is a challenging task that can be approached from many different angles [36]. Implementing function extraction properly requires a good understanding of compilers and the used instruction set architecture. It is an error-prone task, since there are many challenges, such as data intermixed with instructions (e.g., jump tables), non-returning functions, code blocks shared among multiple functions, and tail calls [36]. The original names of the identified functions are also lost during compilation unless the function is exported or annotated in some other way. Since some heuristics would benefit from having the original function names of library functions, it would be beneficial to attempt to recover them. This is also a hard and error-prone problem that has been tackled multiple times [50, 53]. Since solving either of these problems is beyond the scope of this thesis, the extractor has to rely on third-party solutions. As is shown later, IDA Pro's static analysis engine is used for extracting function information, and F.L.I.R.T. [19] is used for recovering library function names.

For each identified function, the extractor retrieves its RVA, size, and function name (if any was recovered). Since some heuristics could benefit from having a more fine-grained way to compare functions, a hash of the function body is also extracted. In order to be able to measure similarity between functions, multiple additional properties of each function could be extracted, such as the number of function arguments, the number of local variables, the stack frame size, and the number of function chunks¹⁹. Note that the extraction of some of those properties is also a difficult task that will not always be performed correctly. For example, the number of local variables is often inferred by analyzing offsets and sizes of accesses to the function's stack frame. Unfortunately, the compiler might use the same memory on the stack for multiple variables (assuming it can prove that they will not be needed

¹⁹A function chunk is the largest contiguous piece of code that belongs to the same function. Optimizing compilers sometimes split functions into multiple chunks to separate rarely executed code from frequently executed code. This might allow them to improve instruction cache performance.


```
xor    ecx, ecx
mov    esi, fs:[ecx+30h]
mov    esi, [esi+0Ch]
mov    esi, [esi+1Ch]

loc_401024:

mov    eax, [esi+8]
mov    edi, [esi+20h]
mov    esi, [esi]
cmp    [edi+18h], cx
jnz   short loc_401024
cmp    byte ptr [edi], 'k'
jz    short loc_40103C
cmp    byte ptr [edi], 'K'
jnz   short loc_401024

loc_40103C:
pop    esi
mov    esi, eax
```

Figure 4.1: An example of a suspicious code fragment found in the trojanized executables from the previous chapter. This fragment represents a position-independent method of obtaining the base address of *kernel32.dll*.

simultaneously), and there is no way to always correctly figure out how many variables are mapped to the same memory. Since advanced function analysis is outside the scope of this thesis, the extraction of such function properties also has to rely on an external static analysis engine. It might not always extract these properties correctly, but some errors are acceptable, as long as they are consistent. After all, the main point of extracting these properties is for function comparison, so if the extractor exhibits consistent errors, the comparison will not suffer that much.

All of the backdoors described in the previous chapter contained suspicious code fragments. These are pieces of code that are not necessarily malicious but that are much more frequently found in malicious executables than in legitimate ones. An example of such a suspicious code fragment could be accessing the `InLoadOrderModuleList` (see Figure 4.1) or using a *geteip* method to obtain the address where the currently executed code is located. The body of each extracted function is scanned for some indicators of these fragments. These indicators can then be used when comparing functions. For example, extracted CRT functions with the same recovered name can be compared. If only one of them exhibits an indicator for the use of the *geteip* method, while the rest of them do not, there may be malicious code injected into that one function.

To link the individual extracted functions together, relations between functions are also retrieved. The most important type of a relation is one function calling another, but all types of control flow transitions between functions should be extracted. For each control flow transition, its exact position within the source and destination function is also helpful. For example, if one function calls another function, the byte offset of the `call` instruction within the calling function is extracted. Note that not all control flow transitions target the entry point of the called function, so the transition's byte offset within the called function is also a useful piece of information.

Control flow transitions to functions imported from other DLLs are also extracted. These transitions are very similar to relations between functions. The difference is that the target cannot be found in the analyzed PE file. It is instead identified by the combination of a DLL name and a function name/ordinal. A significant advantage of these transitions is that they do not have to rely on information about the destination function. The name recovered for an ordinary function might not be accurate, but the identifier of an imported function will always be extracted correctly.

As was shown in the discussion of the ShadowPad backdoor, execution flow can also be hijacked by modifying a function pointer. In order to be able to detect this execution flow hijack method, the analyzed PE file is also scanned for function pointers. Any pointer to a start of a function is extracted. The idea behind this is that there might be some rules that often hold for legitimate PE files. For instance, a sequence of pointers to functions A, B, and C might often be found in legitimate PE files. If only a small number of PE files contain a sequence of pointers to functions A, D, C, the pointer to function B may have been hijacked with a pointer to function D.

Finally, some information about the analyzed samples should also be retrieved. Each sample is identified by its cryptographical hash. To get an idea of what the sample is supposed to be, the `VERSIONINFO` resource is parsed. This resource is often provided by software developers, and it contains basic information about a PE file, such as the name of the company that develops the software, a description of the PE file, and the version of the software [40]. This information can be used for analysis of consecutive versions of the same software to identify potentially suspicious modifications. The validity of the PE checksum is also verified, since a malicious actor could have patched an executable without recomputing its checksum. As was shown in the previous chapter, trojanization artifacts can sometimes be found in the Rich header. Therefore, the extractor also parses and retrieves information from Rich headers of the analyzed samples.

4.1.2 Extractor Design Decisions

Since many malicious PE files are packed, static analysis is often complemented by an unpacking engine. Static analysis of packed samples

would otherwise extract only static properties of the packer. This raises the question of whether the extractor should perform unpacking or not. To answer this question, the percentage of packed samples in the dataset to be analyzed should be taken into account. The heuristic detections developed in this thesis are meant to be executed on supposedly legitimate executables to identify potential backdoors hidden in them. After manually examining some of the PE files that are supposed to be analyzed by the extractor, the percentage of packed files was found to be small enough that unpacking was not deemed necessary. It should also be noted that the potential backdoor can be hidden in the packer itself. Unpacking samples packed with a trojanized packer can be counterproductive because unpacking would completely destroy evidence of trojanization in such cases.

The extractor is designed to process PE files, but it was not yet specified which types of PE files are supposed to be supported. The PE file format can handle multiple instruction set architectures (even though x86 and x86-64 are by far the most common ones) and multiple subsystems. Putting all of them into the same dataset might introduce some unwanted anomalies, since each architecture and subsystem has its own rules that might not be followed in other architectures and subsystems. There are also some functions with the same name that have different implementations based on the architecture or subsystem, which could also make some anomaly detection algorithms worse. Therefore, the extractor only supports 32-bit x86 PE files for the `IMAGE_SUBSYSTEM_WINDOWS_GUI` and `IMAGE_SUBSYSTEM_WINDOWS_CUI` subsystems (this most notably excludes drivers). Both EXE and DLL files are supported. This subset of PE files was chosen because it was the most prevalent one in the available dataset, and the differences between EXE/DLL and GUI/CUI are not significant enough to rationalize further division. Supporting only one instruction set architecture also allows extracting architecture-specific properties. However, note that the majority of the extractor is independent of the architecture and the subsystem, which means that it can be used with minor modifications on other types of PE files as well. For other architectures, different architecture-specific properties could be extracted. For instance, x86-64 PE files commonly contain the `.pdata` section with exception handling information. Function boundaries could be extracted from the data in this section and used to detect patching-based anomalies.

Static analysis of PE files is a CPU-intensive task that can take weeks or even months for a large dataset. Since the analysis of a single sample is an independent task, multiple samples can be analyzed in parallel. Therefore, to speed up the preprocessing stage, there should be a way to configure the extractor to use multiple threads. To avoid losing progress due to unexpected shutdowns and similar issues, the extractor should save analysis results as soon as possible so that it is possible to resume the extraction from where it left off. Finally, since it should be possible to analyze a large dataset of PE

files, the extractor should not require the whole dataset to be available on a local drive. Instead, it should support downloading the analyzed samples dynamically and deleting them locally once they are no longer needed.

4.2 Implementation of the Extractor

The extractor is implemented as a Python script that processes PE files and inserts properties extracted from them into an SQL database. Internally, it uses the `pefile`²⁰ Python module for parsing PE files and IDA Pro for more advanced static analysis that requires a disassembler. For each analyzed file, the extractor runs IDA in a so-called batch mode (which allows starting IDA in the background without opening its GUI). When IDA finishes its standard autoanalysis, a custom IDAPython script is executed in order to collect the analysis results and to perform extraction of additional custom properties.

There are several reasons why IDA Pro was chosen for performing static analysis. First of all, IDA is very good at the analysis of functions and relations between them (also called *cross references* or shortly *xrefs* in IDA). In many cases, it is even able to deduce that there is a cross reference between two functions through indirect calls. Extracting some of the desired function properties discussed in the last section is as easy as calling a single IDAPython API function. Compared with other disassemblers, IDA is also very good at recognizing functions that were split into multiple chunks and at recovering library function names using the built-in F.L.I.R.T. technology. Out of all the considered options (where the most viable alternatives were Ghidra²¹ and Radare²²), IDA also seemed to be the most mature static analysis engine, exhibiting fewer analysis errors than the alternatives. On the other hand, a significant disadvantage of IDA Pro is that it is a commercial piece of software, unlike Ghidra and Radare, which are open source. While there is a freeware version of IDA, it lacks support for IDAPython scripting, so the extractor cannot be used with it. IDAPython API is also neither backward nor forward compatible. The extractor was written for IDA 7.4 (which is the current version at the time of writing), but since the API was changed recently, it does not support versions older than 7.0.

One of the requirements for the extractor was that it should be able to handle a large number of samples. Storing all of the input samples on a local drive might require too much storage, so the extractor only downloads samples to a temporary directory shortly before they are needed and deletes them once they are processed. It attempts to always have a certain number of samples ready in the temporary directory in order to mitigate the possible negative performance impact of waiting for the download of the next sample. The

²⁰<https://github.com/erocarrera/pefile>

²¹<https://ghidra-sre.org/>

²²<https://rada.re/>

download time should not be a performance bottleneck, since static analysis is expected to take much longer than file download, and since each sample is supposed to be downloaded while another one is being analyzed. The way that samples are downloaded is expected to be provided by the user of the extractor, but for convenience, two download methods are already implemented. The first one downloads samples from the malware-scanning service VirusTotal²³ (a VirusTotal API key with sufficient permissions is required), the other one just fetches samples from a custom directory (this method can be used with shared folders or with local folders for smaller datasets).

Since the heuristic detections presented in the following chapter mostly focus on the analysis of library functions, it is not strictly necessary to extract properties of all functions. Instead, particular functions of interest are identified first. These include functions with a recovered library name, entry point functions, and functions at section boundaries. The set of functions of interest is then recursively expanded up until a certain depth by adding functions with a cross reference from one of the functions that is already in the set. Finally, functions that are not in this set are not extracted in order to keep the size of the database smaller.

The extractor is capable of using multiple threads in order to analyze multiple samples in parallel. Any time it finishes the analysis of a sample, it inserts all extracted information about that analyzed sample into the database in a single SQL transaction. This makes it possible to use the database to store progress and stop/resume the extraction at any time. When the extractor is launched, it simply queries the database in order to get a list of already analyzed samples so that it can avoid analyzing the same sample multiple times. Potential errors that happen during extraction are logged, and an error code is inserted into the database so that it is possible to fix these errors and reanalyze the affected samples.

Configuration of the extractor is given by a JSON file. It contains configuration parameters necessary for the extractor's functionality, such as the filesystem path to the IDA Pro executable or the hostname of the output database. The only parameter that is worth mentioning here is `ida_analysis_timeout`. It can limit the maximum amount of time spent on analyzing a single sample. If the analysis time reaches this timeout, the offending sample is skipped and never analyzed again. The motivation for this parameter is that for a tiny percentage of samples, the analysis can take a disproportionately large amount of time. The user of the extractor might wish to skip analyzing such samples in order to improve overall performance.

²³<https://www.virustotal.com/>

4.3 Description of the Extracted Properties

This section contains a brief description of properties extracted from the analyzed PE files. Note that not all of them are used in the heuristic detections described in the next chapter. The extractor was implemented before the heuristic detections, so it was not yet clear exactly which properties will be needed. Therefore, a bigger number of properties was extracted in order to make it possible to experiment with many different approaches to backdoor detection. This approach also allows for implementing further heuristic detections later on without having to modify and rerun the extractor.

4.3.1 Sample Properties

The extractor obtains the following properties of each analyzed PE file:

- **sha256**
The SHA-256 digest of the whole PE file.
- **ep_rva**
The relative virtual address of the entry point.
- **total_number_of_funcs**
The number of functions identified by IDA Pro. Note that this also counts functions that were skipped for extraction.
- **invalid_chksum**
(In)validity of the 32-bit checksum in the PE header. Checksums full of zero bytes are considered to be valid.
- **ver_company_name**
The `CompanyName` string from the `VERSIONINFO` resource.
- **ver_file_description**
The `FileDescription` string from the `VERSIONINFO` resource.
- **ver_file_version**
The `FileVersion` string from the `VERSIONINFO` resource.
- **ver_original_filename**
The `OriginalFilename` string from the `VERSIONINFO` resource.
- **ver_product_name**
The `ProductName` string from the `VERSIONINFO` resource.

- **ver_product_version**
The `ProductVersion` string from the `VERSIONINFO` resource.
- **pdb_path**
The PDB path embedded in the analyzed sample.

4.3.2 Function Properties

For each analyzed function in the PE file, the following properties are extracted:

- **sample_id**
The identifier of the PE file that contains this function.
- **func_name**
The function name extracted by F.L.I.R.T. If one sample contains multiple functions that match the same F.L.I.R.T. signature, IDA assigns each of them a unique name by appending an underscore and an incrementing numerical suffix to the function name (e.g., `nullsub_1` and `nullsub_2`). Since having multiple function names for the same signature would be detrimental to some heuristic detections, this suffix is removed by the extractor.
- **rva**
The relative virtual address of the function's entry point.
- **size**
The size of the function body in bytes.
- **hash**
The MD5 digest of the function body. This digest skips over some bytes (such as call displacement values) in order to achieve higher stability.
- **num_chunks**
The number of functions chunks that the function is composed of.
- **num_unusual_outside_cref**
The number of code cross references to other functions that are not represented by the most common opcodes for `call` or `jmp`.
- **flag_no_ret**
The value of the `FUNC_NORET` function flag assigned by IDA. This flag represents whether the function is expected not to return.

- **flag_bp_frame**

The value of the `FUNC_FRAME` function flag assigned by IDA. It represents whether the `ebp` register is used by the function to access its stack frame.
- **flag_fuzzy_sp**

The value of the `FUNC_FUZZY_SP` function flag assigned by IDA. It is set if the analyzed function modifies the stack pointer in a way that causes stack pointer analysis to fail.
- **flag_thunk**

The value of the `FUNC_THUNK` function flag assigned by IDA. It is set if the analyzed function is just a jump thunk to another function.
- **local_var_num**

The number of local variables used by the analyzed function.
- **local_var_size**

The sum of sizes of local variables used by the analyzed function.
- **arg_num**

The number of arguments the analyzed function takes.
- **arg_size**

The sum of sizes of arguments that the analyzed function takes.
- **frame_size**

The size of the whole function frame of the analyzed function (extracted using the `idc.get_frame_size` function).
- **is_badstack**

Indicates whether stack pointer analysis of this function failed (extracted from the presence of the `PR_BADSTACK` *problem* in IDA).
- **num_code_xrefs_from_undef_funcs**

The number of code cross references to the analyzed function that originate from code that does not belong to any function.
- **num_code_xrefs_to_undef_funcs**

The number of code cross references from the analyzed function that point to code that does not belong to any function.
- **num_jmps_without_xref**

The number of `call` and `jmp` instructions that do not result in a code cross reference (i.e., number of `call` and `jmp` instructions for which IDA cannot statically deduce the target).

- **num_code_xrefs_to_unextracted_funcs**

The number of code cross references that point to a function that was not extracted.

- **num_push_page_rwx_bef_call**

The number of times that the constant `0x40` was used as the third or the fourth argument in an indirect function call (see Figure 4.2a for an example). If interpreted as a memory protection constant, `0x40` stands for `PAGE_EXECUTE_READWRITE`, which is used in functions such as `VirtualAlloc` or `VirtualProtect` to obtain memory that is both writable and executable.

This is the first of four indicators of suspicious code fragments. It attempts to detect traces of manually resolving one of the above-mentioned functions and using it to prepare memory for the backdoor's next stage. As is the case with the other indicators, a certain number of false positives is expected because the constant `0x40` can clearly represent much more than just `PAGE_EXECUTE_READWRITE`.

- **num_susp_fs_access**

The number of times that the `fs` segment override prefix was used in the analyzed function. This prefix is commonly used in shellcodes to obtain the address of the Process Environment Block (PEB) and consequently find the base address of some loaded DLL (see Figure 4.1 for an example). As is the case with `num_push_page_rwx_bef_call`, this is just an indicator, since there are many legitimate reasons for using this prefix. Some common uses of this prefix (such as setting up or unwinding Structured Exception Handling frames) were filtered out in order to lower the number of functions in which this indicator is nonzero.

- **num_xor_loops**

The number of `xor` instructions that are in the middle of a loop that spans only a single basic block (see Figure 4.2c for an example). `xor` instructions that have the same source and destination operand are not counted, since they are just used to zero out a register. This is an indicator of the usage of encryption/hashing, since many cryptographic algorithms contain short loops that perform the `xor` operation.

- **num_get_eip**

The number of times that an attempt to obtain the current value of the instruction pointer was detected. This counts `pop` instructions found at a `call` target (see Figure 4.2b) and the usage of the `fstenv` and `fnstenv` instructions. These two so-called *geteip* methods can often be found in position-independent shellcodes that attempt to obtain a pointer to some data that is located at some offset relative to the shellcode.

4. CALL GRAPH EXTRACTOR

```
push    PAGE_EXECUTE_READWRITE ; 0x40    push    ebp
push    MEM_COMMIT ; 0x1000             mov     ebp, esp
push    20h ; ' '                       call   $+5
push    0                                pop     eax
call    edi ; VirtualAlloc               pop     ebp
                                           retn
```

(a) A piece of code that uses the `PAGE_EXECUTE_READWRITE` constant as an argument in an indirect call.

(b) A piece of code that uses a `geteip` method.

```
loc_401012:
mov     eax, [ebp+arg_ptr]
imul   ecx, 47A6547h
mov     dl, cl
xor     [eax+esi], dl
shr    ecx, 8
inc     esi
cmp     esi, [ebp+arg_size]
jnl    short loc_401012
```

(c) An example of a loop that spans a single basic block and uses a `xor` instruction for encryption.

Figure 4.2: Examples of indicators of suspicious code fragments that were found in the backdoored PE files discussed in the previous chapter.

4.3.3 Other Properties

The following properties characterize cross references between extracted functions:

- **func_from_id**

The identifier of the function that the cross reference originates from.

- **func_from_chunk_num**

The number of the function chunk containing the source of the cross reference. Together with `func_from_chunk_offset`, it can be used to infer the exact location of the cross reference within the source function.

- **func_from_chunk_offset**

The offset from the start of the chunk that is identified by `func_from_chunk_num` to the source of the cross reference.

- **func_to_id**

The identifier of the function that the cross reference points to.

- **func_to_chunk_num**

The number of the function chunk containing the destination of the cross reference.

- **func_to_chunk_offset**

The offset from the start of the chunk that is identified by `func_to_chunk_num` to the destination of the cross reference. Note that `func_to_chunk_num` and `func_to_chunk_offset` are usually both zero because most cross references target the very entry point of the referenced function.

- **xref_type**

The type of the cross reference as identified by IDA Pro. Both code cross references (such as a `call` or a `jmp` instruction to another function) and data cross references (such when one function contains an address of another function) are extracted.

Cross references to functions imported from external DLLs are stored in a similar fashion. The only difference is that the properties prefixed with `func_to` are replaced with a new property called `import_name`, which is a string that identifies the imported function.

Static pointers that point to the entry point of a function are extracted in the following way:

- **from_rva**

The relative virtual address where the pointer is located.

- **func_to_id**

The identifier of the function that the pointer points to.

Finally, each entry in the Rich header is fully extracted:

- **sample_id**

The identifier of the PE file that this Rich header entry was extracted from.

- **tool**

The identifier of the build tool that was used to produce the executable (sometimes also referred to as a *product identifier* or *ProdID*).

- **build_num**

The minor version of that tool.

- **count**

The number of items produced by that tool.

4.4 Issues with the Extractor

As was already discussed, the quality of the data produced by the extractor is heavily dependent on the quality of IDA Pro's analysis. Since static analysis of PE files is an incredibly difficult task, some inaccuracies during the analysis are expected. This is also one of the reasons why such a seemingly large number of properties is extracted: if some subset of properties would exhibit too many inaccuracies, the heuristic detections might switch to using other properties.

Some of the issues identified during the extraction are worth pointing out. They are not severe enough to invalidate the extracted data, but they should be taken into account when heuristic detections are designed in order to avoid possible problems that could be caused by them.

IDA Pro is used to recover function names of library functions. The recovered function names can be used in heuristic detections for clustering similar functions together. Unfortunately, IDA Pro sometimes assigns function names based on the context that a function is used in, instead of the actual functionality of the function. For example, entry points of threads started by the `CreateThread` function are often assigned the function name `StartAddress`. Clustering functions based on such names would result in clusters of poor quality, since the content of various functions named `StartAddress` will vary wildly. Therefore, the extractor contains a blacklist of similar function names and avoids assigning such problematic names to analyzed functions.

Some linkers in some configurations merge duplicate functions (for example, in MSVC this can be controlled by the `/OPT:ICF` linker switch). This means that if they encounter two or more functions that have the exact same function body, they generate only one instance of that function. This could pose a problem for some heuristic detections because a merged function might be used in multiple seemingly unrelated contexts, which could be considered an anomaly by itself. Since F.L.I.R.T. only recovers a single function name for any function, the extractor might produce some cross references on unusually named functions if there is a call to a merged function that is named after one of the other functions.

The compiler might decide to split a function into multiple function chunks. In addition to that, some function chunks might even be shared across multiple functions. A straightforward approach to adapting to this fact would be having a table in the database that would describe individual function chunks and would have a many-to-many relationship with the table of functions. Nevertheless, this database design was not used. Instead, there is no table of function chunks, and all information extracted from shared function chunks is replicated across all functions that contain that chunk. This design was chosen in order to improve performance. After all, the number of shared

function chunks is too low to justify having to pay for the performance impact of joining an additional table in most database queries.

Heuristic Detections

Malicious actors understandably strive to make their backdoors as stealthy as possible. The analysis of backdoors in this thesis showed that they often achieve trojanization by modifying statically linked runtime library functions. These functions are ideal for their purposes, since they are usually not inspected in detail by reverse engineers who instead often only make assumptions about these functions based on recovered function names. Reverse engineers also frequently start the analysis of unknown samples directly from a function like `main` and ignore the library functions that get executed before it. This makes it even more tempting for backdoor authors to hide their malicious payload in one of those functions.

There are naturally many various ways that a backdoor can be inserted, but if backdoor authors insert the backdoor during or after compilation, the act of inserting the backdoor often creates one or more anomalies in the backdoored executable. The goal of this chapter is to design and implement heuristic detections that detect such anomalies because finding and analyzing them might lead to the discovery of yet undetected backdoors.

The detections should not target all kinds of anomalies, since most types of anomalies are caused by benign reasons. Instead, the detections should only focus on anomalies for which there is a reason to believe that they could have been caused by trojanization. All heuristic detections discussed in this chapter target anomalies that have either been observed by analyzing in-the-wild backdoors, or that would have been caused by some straightforward trojanization method. Specifically, the proposed algorithms target anomalies that would be caused by standard execution flow hijack methods.

It might seem overly complicated to design such anomaly detection algorithms instead of focusing directly on detecting backdoors. The reason for this indirection is that it is extremely hard to predict what might an undiscovered backdoor look like. On the other hand, types of anomalies caused by backdoors were discussed in this thesis, and there is a reason to believe that future backdoors will produce similar anomalies. A disadvantage of the

anomaly detection approach is that it is often necessary to perform reverse engineering in order to discover the root cause of the detected anomalies. Only when one understands the root cause can the detection algorithm be improved so that it better separates anomalies caused by trojanization from anomalies caused by benign factors.

Verifying whether an anomalous sample is indeed trojanized should also be done through manual reverse engineering. If a certain heuristic detection reports too many anomalies, and it is not feasible to analyze all of them, some triage methods could be used to identify anomalies that should receive attention first. One such triage method could be modifying the entry point of a sample to the location of the anomaly and using dynamic analysis to see if the modified sample will execute self-decrypting code. Since all of the backdoors discussed in this thesis did not depend on the previous execution context, changing their PE files' entry point to the entry point of the backdoor would still fully execute all of their self-decrypting stages. On the other hand, just changing the entry point of a benign PE file to the location of some random anomaly would, in most cases, corrupt the PE file and cause it to crash soon after it starts executing.

To evaluate the effectiveness of the proposed heuristic detections, a dataset composed of both benign and trojanized executables was compiled and processed by the extractor described in the previous chapter. Access to this dataset was provided by the antivirus company Avast. It contains one million of the most prevalent legitimate PE files and ten trojanized PE files. The database containing information extracted from this dataset is relatively large at over 400 GB. As can be seen, this dataset is extremely skewed, since it contains vastly more benign executables than trojanized ones. This is because the number of backdoors discovered in-the-wild is fairly limited, which makes it not possible to create a large and balanced dataset. The skewness of the dataset is one of the main reasons why standard machine learning algorithms were not considered: there are simply not enough backdoored executables to train a model for their detection. Instead, the large quantity of benign executables is used to establish a "baseline" of ordinary executables in order to detect executables that deviate from this baseline. Note that since one of the goals of this thesis is to attempt to find yet undetected backdoors in the benign part of the dataset, detected anomalies among the supposedly benign executables should not be immediately treated as false positive detections.

Performance evaluation of binary classification algorithms is usually conducted by employing the confusion matrix and measures such as precision, recall, or F-score [48]. Since there are only ten trojanized binaries in the dataset, the number of true positives and false negatives exhibited by any binary classification algorithm is certainly going to be very low. Unfortunately, this could cause the above-mentioned measures to be extremely volatile and make it tempting to optimize the heuristic detections for these measures (after all, all that is needed for a "perfect score" is to find a way to isolate those ten

backdoors in the dataset). Therefore, the above-mentioned measures are not computed for the proposed heuristics. Instead, each detection is evaluated based on a discussion of the types of backdoors that would be detected by it and its false positive rate (which is one of the few measures that can be computed reasonably accurately, since it depends only on the number of false positives and true negatives).

All of the heuristic detections discussed in this chapter are implemented as Python scripts that make SQL queries to the database extracted from the analyzed dataset. Once a heuristic detection was implemented, it was executed, and randomly selected anomalies detected by it were manually analyzed. If some benign class of anomalies was present in large quantities, an attempt was made to improve the detection and filter out this anomaly class in order to reduce the detection's false positive rate.

5.1 Entry Point Hijacking

One of the most straightforward ways to trojanize an executable is by hijacking its entry point. Executables trojanized by entry point hijacking are forced to execute an injected malicious payload first and then return to executing code from the original entry point (detailed discussion of this technique can be found in Section 2.4). The idea behind this heuristic detection is to compile a list of functions that are typically only found at the entry point. Samples that contain one of these “typical entry point functions” at a location different from the entry point are then reported as anomalous, since they could be affected by entry point hijacking. Most compilers have a typical function that they usually emit to the entry point. This function should generally only be found at the entry point, so if it is located elsewhere, the location of the entry point itself could have been changed. For instance, most binaries built with recent versions of MSVC contain the function shown in Figure 3.2a at the entry point. If that function would still be present in an executable but would not be located at the very entry point, the entry point of the executable might have been altered with malicious intent.

The typical entry point functions are identified by their recovered function names. This was found to work better than identifying them using their function hashes because the recovered function names are much more generic. Unfortunately, there is a problem with using function names of entry point functions, since they are often named not based on their functionality but based on the fact that they are located at the entry point (IDA commonly assigns them generic names such as `start` or `DllEntryPoint`). To get around this problem, this detection does not work directly with the entry point functions but instead with functions that are targeted with a cross reference from the entry point function. A prime example of such a function is `__security_init_cookie`. This function is usually called directly from

function name	count
___security_init_cookie	313641
___tmainCRTStartup	248734
lstrcatW	96501
_fast_error_exit	94776
__amsg_exit	86888
_initterm	72531
lstrcatA	70767
__XcptFilter	65624
_exit	56495
__setargv	56229

Figure 5.1: A list of functions that are most commonly cross-referenced from the entry point. The *count* column contains the number of unique samples in which the corresponding function is cross-referenced from the entry point function. The first two functions are called from MSVC startup, while *lstrcatW* is called from the entry point function of NSIS installers.

the entry point, and since it performs a one-time setup, it does not have to be called again from other locations.

5.1.1 Implementation

This heuristic detection starts by compiling a list of functions (grouped by the recovered function name) that are commonly cross-referenced from the entry point function (see Figure 5.1 for the top ten functions in this list). For each function name in this list, two sets of samples are created. The first one consists of samples where a function with that name is cross-referenced from the entry point function. The second one consists of samples where it is cross-referenced from a location other than the entry point function. If the ratio between the size of the first set and the size of the second set is sufficiently high (meaning that the function is most of the time only cross-referenced from the entry point), all samples that are in the second set and at the same time not in the first set are reported as anomalous. The actual entry point of those samples is reported as the location of the anomaly because that is where the suspected backdoor’s entry point could be.

A casual examination of the reported anomalies revealed that in some of them, the examined functions are cross-referenced indirectly from the entry point function through a thunk function²⁴. This is because the reported binaries are mainly incrementally linked debug builds with support for the

²⁴A thunk function is a function containing nothing but a `jmp` instruction to another function.

Edit and Continue feature. Since these are not anomalies that would be interesting for the purpose of discovering backdoors, cross references from thunk functions to the examined functions were filtered out.

5.1.2 Evaluation

The first version of this heuristic detection reported 1374 anomalous samples. After filtering out cross references from thunk functions, this number was reduced to 957. Since no new trojanized samples were found among these anomalies, this represents a false positive rate of 0.000956.

Note that a *false positive* in this context denotes a reported anomaly that was not caused by trojanization. Many of the reported anomalous samples actually *had* a hijacked entry point, it was just not hijacked with malicious code. For example, some samples contained a check at the entry point if the MMX instruction set was supported by the processor. If it was, the code at the original entry point would be executed. If it was not, an error message would be displayed, alerting the user that the processor was not supported, and the sample would exit prematurely.

Samples with slightly modified startup functions were also among the false positives. For instance, some samples initialized the security cookie slightly later than usual. Since this detection learned that the `__security_init_cookie` function should typically only be called from the entry point function, such samples were unfortunately also reported as anomalous.

This detection was designed to detect standard cases of entry point hijacking. To its credit, it also detected the anomaly from the supply chain attack described in Section 3.2. Even though that sample was not trojanized with entry point hijacking, the execution flow hijack happened so close to the entry point that it also got picked up by this detection, since the function `__tmainCRTStartup` was not called directly from the entry point function (see Figure 3.2).

It has to be acknowledged that not all cases of entry point hijacking would be detected. First of all, this detection relies on one of the “typical entry point functions” being at the original entry point of the trojanized executables. If an executable with a rare entry point function gets trojanized (or with an entry point function that only cross-references functions that are commonly cross-referenced from other locations as well), this detection would most likely miss that. Secondly, it relies on an accurate recovery of library function names. F.L.I.R.T. failing to recognize one of the “typical entry point functions” can also lead to false negatives. Thirdly, it requires IDA to actually create a function at the original entry point. Since IDA might not even recognize the control flow transition from the malicious code to the original entry point, the code at the original entry point might, in the worst case, not even be disassembled and could be treated as data instead.

To summarize, this heuristic detection has a reasonably low false positive rate, which makes it usable, since one can verify all the reported anomalies even for somewhat larger datasets. However, there are multiple methods to trojanize a PE file, and this detection is designed to detect only one of them: entry point hijacking. Furthermore, since it relies on certain assumptions about the trojanized binaries, it is not guaranteed to detect all cases of entry point hijacking.

5.2 Anomalous Cross References

Backdoors described in this thesis often hijacked the execution flow by modifying the content of library functions. In the ShadowHammer backdoor, the `___crtExitProcess` function was patched in order to redirect a call instruction's target to the backdoor's entry point (see Figure 3.1). In the CCleaner incident, functions `__sCRT_get_dyn_tls_init_callback` and `___onexitinit` were tampered with, and an extra call to the backdoor's entry point was inserted into them (see Figures 3.3 and 3.4). In both cases, this malicious modification of a library function resulted in an anomalous cross reference: the modified library function simply transferred control flow to a function that it was not originally supposed to.

This heuristic detection is based on the assumption that the expected functionality of some library functions should stay relatively constant. For such functions, it should be possible to build a list of target functions that are typically cross-referenced from them. If one instance of such a function cross-references an unknown function that is not in this list, the body of that instance could have been forcibly modified in order to transfer control to some malicious code.

For example, the `___crtExitProcess` function can be found in 198 353 samples in the analyzed dataset. A cross reference to a function whose recovered function name was not `___crtCorExitProcess`, can be found in only 524 of them²⁵. It should not come as a surprise that this number is so low, since the source code of `___crtExitProcess` is published by Microsoft (see Figure 5.2) and it can be seen that it should call only the `___crtCorExitProcess` function. Therefore, if there is a sample where it cross-references some function other than `___crtCorExitProcess`, then either the body of `___crtExitProcess` was modified for any reason, or there was a mistake in IDA's analysis (such as F.L.I.R.T. failing to recognize the `___crtCorExitProcess` function).

²⁵This heuristic detection only takes into account cross references to functions present in the same executable and disregards cross references to imported functions. The function `___crtExitProcess` also regularly cross-references the function `ExitProcess` imported from `kernel32.dll`.

```
void __cdecl __crtExitProcess(int status) {  
    __crtCorExitProcess(status);  
    ExitProcess(status);  
}
```

Figure 5.2: The official implementation of the *__crtExitProcess* function (from the *crt0dat.c* source file). Comments were removed for brevity.

The number of anomalies can be even further reduced if only cross references from the offset 8 of the `__crtExitProcess` function are taken into account. From this offset, there are 107 510 cross references to a function with the recovered name `__crtCorExitProcess` and only four other cross references. One of those four anomalous cross references targets the entry point of the ShadowHammer backdoor. In the remaining three cases, the function `__crtCorExitProcess` was indeed cross-referenced from that offset, but F.L.I.R.T. unfortunately failed to recognize it.

5.2.1 Implementation

This heuristic detection starts by randomly selecting a subset of the extracted dataset and using it as a training set. Cross references observed in the training set are clustered together and used to build a set of rules that hold true for all samples in the training set. When the set of rules is finalized, the rules are applied to the rest of the dataset. If any sample violates any of the generated rules, it is reported as anomalous.

Each rule contains a cross reference source location (rule source) and a list of targets that were typically observed cross-referenced from that location (rule destinations). A rule is violated if there is a cross reference whose source matches the source of some rule but whose destination is not among the rule destinations. The rule source is identified by the quadruple of a recovered function name, function size, cross reference source chunk number, and cross reference source chunk offset²⁶. A rule destination is identified only by the recovered function name. Getting back to the ShadowHammer example, a sample rule could be described followingly: if there is a `__crtExitProcess` function whose size is 23 bytes and which cross-references another function from its first chunk at byte offset 8, that cross reference should target the `__crtCorExitProcess` function. As was mentioned, this exact rule holds true for 107 510 samples and is violated in only four samples. Samples in the rest of the dataset do not contain a cross reference that would match the rule source, which means that this rule cannot even be applied to them.

²⁶This detection would of course also work with various other selections of properties that would identify the rule source, but this one was found to work best in practice.

To build the set of rules, all cross references in the training set are clustered according to the quadruple that identifies the rule source. For each cluster, the cross reference targets from the source are grouped by their recovered function names, and the number of times each function name was cross-referenced is counted. If the count of all function names is above (or equal to) a certain threshold, a new rule is created. Otherwise, if there is a function name with a count below the threshold, no rule is created, since that rule source is too volatile and would probably cause too many false positives. The threshold essentially controls the quantity and the quality of generated rules. If it is set to a high value, a low number of high-quality rules will be created. If it is set to a low value, a higher number of rules will be created, but there might be some rules of questionable quality.

This heuristic detection is a randomized algorithm, since the training set is randomly sampled from the entire extracted dataset. Therefore, a different set of anomalies might be reported each time this detection is executed. This can be considered an advantage, since new anomalies can be discovered just by simply restarting the algorithm. However, sometimes a certain level of consistency might be demanded. For this reason, the detection always prints the randomly selected seed used to initialize the pseudorandom number generator (PRNG). If the heuristic detection is executed multiple times with the same seed, it will produce the same results.

The size of the training set is an important parameter of this detection algorithm. Note that because of the design of this algorithm, no anomalies in samples that belong to the training set will ever be reported. Furthermore, if at least one instance of some type of anomaly is present in the training set, no other instances of said anomaly type will be reported, even when found in samples that do not belong to the training set. Therefore, if the training set is too large, the detection might miss many interesting anomalies. On the other hand, if the training set is too small, the set of generated rules might fit the training set too closely, and that could result in a large number of false positives.

As was the case with the heuristic detection for entry point hijacking, a problem with indirect calls through thunk functions was encountered. The problem was that thunk functions were often called from the same rule source as the real destination function but much less frequently. This prevented the algorithm from building many rules, since the rule destination represented by the thunk function was often less prevalent than the rule creation threshold. Fortunately, IDA usually names thunk functions the same as the real destination function, only prefixed with the string `j_`. This heuristic detection solves the problem with thunk functions by removing this prefix and thus treating the thunk function as if it were the real destination function.

5.2.2 Evaluation

Since this heuristic detection is randomized, it was executed multiple times in order to evaluate its performance. Out of ten runs, an average of 312 rules was generated from the randomly selected training set (minimum was 285, maximum 337). These rules reported an average of 2120 anomalies in 565 samples (the lowest number of reported samples was 313, the highest one was 895). Considering that the size of the dataset is one million samples, this represents an average false positive rate of 0.000564. In total, 1615 unique samples were reported as anomalous in at least one run. The ShadowHammer backdoor was detected in nine runs. In the run where it was not detected, the training set contained one of the four anomalous samples discussed at the beginning of this section, and that prevented the detection from creating a rule for the `___crtExitProcess` function.

This shows a potential limitation of the “training set approach”. The training set should ideally be composed of a substantial number of backdoor-free executables. However, if this algorithm is used to look for undiscovered backdoors in a large dataset of supposedly benign (but possibly trojanized) executables, it gets very hard to create such a training set. If the training set is chosen at random, there is a certain risk that a backdoored executable could end up in the training set, which would cause false negatives. This would not be such a problem if only one instance of a given backdoor was present in the dataset. The algorithm could simply be restarted multiple times, and the probability that the backdoor would find itself in the training set in every run would rapidly decrease. However, if there would be multiple instances of the same backdoor in the entire extracted dataset, then the probability that at least one of those instances would be included in the randomly selected training set could become significantly higher.

The vast majority of the reported anomalies were false positives caused by F.L.I.R.T. failing to recognize the target function of a cross reference. In these cases, the reported cross references did indeed target one of the functions that were expected to be cross-referenced from the given rule source, but since there was no function name recovered for the target function, it was treated by the heuristic detection as if an unknown function was called. Other false positives were caused by slight variations in the recovered function name. Examples of this include a cross reference to `_getptd` instead of `__getptd` or a cross reference to `__SEH_prolog` instead of `__SEH_prolog4`. The number of such false positives could be reduced by aliasing similar problematic function name pairs and treating them as the same rule destination.

As was already mentioned, this heuristic detection reported on average 2120 anomalies in 565 samples. This means that the average reported sample contains almost four anomalies. The samples that contain multiple anomalies usually seem to be a result of unusual compiler/linker options or low-level optimizations. Given the relatively low false positive rate, there is a good

chance that if a backdoored executable was reported as anomalous by this heuristic detection, it would only contain the one anomaly that points to the place where the execution flow was hijacked. Therefore, triaging of the reported anomalies could be implemented by starting with samples where only a single anomaly was identified²⁷. This observation could also be used to improve the quality of the training set. Samples that were previously observed to contain multiple anomalies could be excluded from the training set (or from the entire extracted dataset), which should increase the number of generated rules.

Unfortunately, this heuristic detection will only detect a specific type of backdoors. A backdoor reported by this detection would have to hijack an existing cross reference located in a library function. This hijack could be accomplished by modifying the source code, tampering with an object file, or just simply patching an instruction after compilation. However, functions that were backdoored by recompiling them with extra code (such as in the CCleaner incident) would not be detected.

Some other conditions also have to be met for a backdoor to get detected. Firstly, the library function where execution flow gets hijacked has to be common enough so that multiple similar instances of it can be found in the training set. The hijacked cross reference also has to have only a handful of valid targets that are successfully identified by F.L.I.R.T. Otherwise, no rule would be created for that cross reference. Unfortunately, these conditions make it seem like this heuristic detection would have a very high false negative rate. However, it did reasonably consistently detect the ShadowHammer backdoor, which at least indicates that it would be able to detect some backdoors that use similar trojanization techniques.

5.3 Import Call Hijacking

The heuristic detection described in the previous section was limited to detecting hijacked cross references that initially targeted a statically linked library function. Hijacked cross references that used to target a function imported from another module would not get detected. The heuristic detection described in this section is meant to remedy this shortcoming and focuses on detecting hijacked cross references that originally targeted imported functions. Even though no backdoor described in this thesis hijacked such a cross reference, this technique might still be appealing to backdoor authors. This is because calling some imported functions is not strictly necessary for the functionality of the backdoored program. For example, function calls

²⁷Another easy-to-implement way to triage the anomalies would be to analyze the position of the function that is the target of the anomalous cross reference. As was discussed, the first stage of the backdoor was often found at section boundaries, so it would make sense to prioritize anomalies where the new destination function is either at the start or the end of a section.

meant to deallocate resources (such as calls to `HeapFree` or `CloseHandle`) might be entirely replaced by calls to the backdoor code. Since not calling the original imported function might not break the functionality of the backdoored executable, the backdoor authors have an easier job because they do not have to worry about making sure that the original destination function still gets executed.

In order to detect anomalies that might have been caused by import call hijacking, this heuristic detection compiles a list of locations that commonly serve as sources of cross references to imported functions. If cross references from such locations target in the vast majority of cases only one specific imported function, all samples where any other function is cross-referenced from that location are reported as anomalous. As was the case with the previously described heuristic detections, this is based on the premise that backdoors caused by import call hijacking are very rare. Therefore, it should be possible to learn which imported functions are typically cross-referenced from which statically linked library functions and report anomalous samples where these typical relationships are not preserved, since they might have been broken because of import call hijacking.

Note that backdoor authors do not have much incentive to hijack a cross reference to an imported function with a cross reference to another imported function. If they wanted to hijack execution flow this way, they would have to put the backdoor code into another PE file, i.e., they would be required to needlessly make malicious modifications to at least two PE files. Even if they did choose to make modifications to multiple PE files, there would be easier ways to accomplish their goal, such as adding a dummy import of a malicious DLL and putting the backdoor code at the entry point of that DLL. While malware authors sometimes do not choose the easiest way to accomplish things in an attempt to evade existing detection and to confuse malware analysts, a backdoor that would be introduced by hijacking a cross reference to an imported function with a cross reference to another imported function still seems extremely unlikely. Therefore, this heuristic detection will not report anomalies that seem to be caused by such hijacks in order to keep its false positive rate low.

5.3.1 Implementation

This heuristic detection starts by clustering together similar library functions. The clustering is based on the recovered function name and function size: if two functions have the same values of these two properties, they are placed into the same cluster. The clusters are then analyzed one at a time, starting with the most prevalent ones. Clustering based on function hashes was also explored, but it was found to perform worse than this clustering method, since it resulted in a lower number of larger clusters. However, a different implementation of function hashing (such as one that would skip over more

volatile bytes or one that would be based on fuzzy hashing) could possibly yield better results.

Within each cluster of similar functions, all locations that frequently serve as sources of cross references to imported functions are analyzed. The locations are identified by their chunk number and their chunk offset within the function. For each location, the imported function that is most frequently cross-referenced from there is obtained, along with the number of times that it is cross-referenced. If this number is significantly higher than the number of cross references from that location which target functions that are not imported, the location is used for reporting anomalies. Specifically, all samples that contain a cross reference from that location to a function that is not imported are reported as anomalous, with the location of the anomaly being the destination of that cross reference. The idea behind this is that if the vast majority of cross references from a given location target some specific imported function, but there are also few cross references which target functions that are not imported from that same location, these few cross references can be considered anomalous.

To illustrate how this heuristic detection works in practice, consider an example regarding the `___security_init_cookie` function. This function's purpose is to initialize the stack cookie to a hard-to-predict value. To do so, it commonly gathers entropy using imported functions `GetTickCount`, `GetCurrentProcessId`, `GetCurrentThreadId`, `GetSystemTimeAsFileTime`, and `QueryPerformanceCounter`. One of the more prevalent clusters in the dataset consists of `___security_init_cookie` functions whose size is exactly 150 bytes. Among the 92 719 functions in this cluster, the detection identified five locations that are commonly a source of a cross reference to an imported function. One of them can be found at byte offset 83. From this offset, the imported function `GetTickCount` was cross-referenced 92 662 times, and there was no other imported function cross-referenced from this offset. The extractor identified only one cross reference from that offset to a function that was not imported. Since there are significantly more cross references to `GetTickCount`, the sample containing this one cross reference gets reported as anomalous.

Unfortunately, IDA is not always consistent with assigning names to imported functions. For example, the `LeaveCriticalSection` function imported from `kernel32.dll` might be labeled `LeaveCriticalSection`, `__imp_LeaveCriticalSection`, or `__imp__LeaveCriticalSection@4`, depending on whether there is a thunk function wrapping calls to it. Having three different names for the same imported function would be detrimental to this heuristic detection since it could artificially lower the prevalence of the most commonly cross-referenced imported function and thus result in fewer locations being used for reporting anomalies. To resolve this issue, the detection normalizes names of imported functions by removing various prefixes and suffixes so that aliases of the same imported function are merged together.

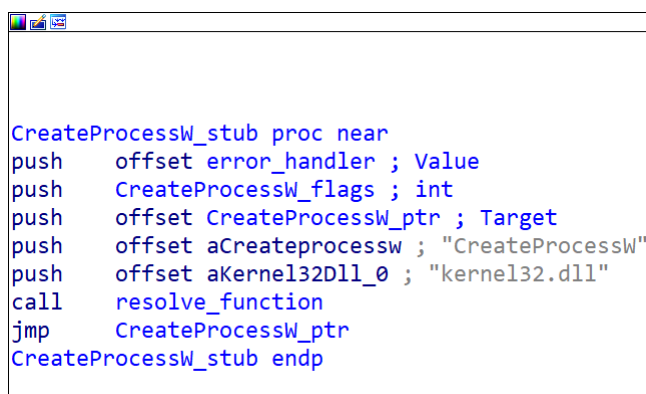
5.3.2 Evaluation

When this heuristic detection was applied to all function clusters that contained at least 10 000 functions (which was 4857 clusters), it reported 225 anomalies in 102 unique samples. Since none of the identified anomalies were found to be caused by trojanization, this represented a false positive rate of 0.000102. Note that this is a much lower false positive rate than the one associated with the heuristic detection described in the previous section. This might seem surprising at first, since both detections are based on a very similar concept. However, note that the most significant source of false positives in the previous heuristic detection was F.L.I.R.T. failing to recognize the names of functions commonly cross-referenced from some location. In contrast, this detection focuses on imported functions, and therefore it can obtain the names of commonly cross-referenced imported functions from the import table. Parsing the import table is a much more reliable way of recognizing functions, which means that this source of false positives is almost nonexistent in this heuristic detection.

The anomalies reported by this heuristic detection were caused by a lot of various factors, but most of them seemed to be a result of developers deliberately altering the way the imported functions were called. For example, one sample had all calls to `IsDebuggerPresent` hijacked with calls to a custom function that constantly returned zero. Since the `IsDebuggerPresent` function is used to detect if the calling process is being debugged by a user-mode debugger [38], this resulted in the program always receiving a negative answer (that there is no user-mode debugger attached).

Some samples contained multiple anomalies, since they implemented a custom lazy import resolution mechanism, similar in functionality to Procedure Linkage Table (PLT) stubs. Calls to imported functions were realized indirectly through function pointers that were initialized to point to custom stub functions (see Figure 5.3). The stub function's purpose was to dynamically resolve the imported function's address, replace the function pointer with it, and finally execute the imported function. This effectively ensured that the imported function got resolved the first time it was called, and any subsequent calls to it were no longer hijacked, since they bypassed the stub function and instead executed the imported function directly.

Other anomalies were reported in samples with a corrupted import table. IDA did not identify any imported functions in these samples, so there were naturally no cross references to imported functions either. However, many statically linked library functions in these samples were still implemented in the usual way, with cross references originating from the usual locations. Cross references that would typically target imported functions were still present but with an unresolved destination. The heuristic detection unfortunately reported such cross references as anomalous, since they did not target the expected imported function. These anomalies are not very interesting for

A screenshot of a window displaying assembly code for a stub function. The code is as follows:

```
CreateProcessW_stub proc near
push  offset error_handler ; Value
push  CreateProcessW_flags ; int
push  offset CreateProcessW_ptr ; Target
push  offset aCreateProcessW ; "CreateProcessW"
push  offset aKernel32Dll_0 ; "kernel32.dll"
call  resolve_function
jmp   CreateProcessW_ptr
CreateProcessW_stub endp
```

Figure 5.3: A stub function that gets executed the first time *CreateProcessW* is called. All function calls to *CreateProcessW* are performed indirectly through *CreateProcessW_ptr*, which is initialized to point to this stub function, and which is set to point to the real *CreateProcessW* function in *resolve_function*.

the purpose of searching for backdoors and could be filtered out by ignoring samples that contain no cross references to imported functions.

As was the case with the heuristic detections described in the previous two sections, this detection is designed to detect backdoors that use a particular execution flow hijack method. Specifically, the hijack method that is the focus here is called import call hijacking, which is what happens when backdoor authors replace a cross reference to an imported function with a cross reference to the backdoor code. Unfortunately, this detection will not detect all instances of import call hijacking because some conditions have to be met for a backdoor to get detected. The first condition is that the hijacked cross reference has to originate from a location that is commonly used as a source of cross references to imported functions. The second condition is that there can only be a limited number of samples that contain a cross reference from that location to a function that is not imported.

In summary, the main limitation of this heuristic detection is that it is only designed to detect a rather small subset of backdoors, specifically ones that use a particular execution flow hijack method. Since this method has never been observed in a backdoor in-the-wild, it was not possible to demonstrate that it would successfully detect a real trojanized executable. The lack of backdoors that would use this method also makes it hard to estimate this detection's false negative rate. On the other hand, this detection produces only very few false positives, and it was consequently possible to manually verify all the anomalies that it reported. All of them did indeed contain anomalous cross references, and some of them were suspicious enough to warrant further analysis.

5.4 Other Heuristic Detections

Three heuristic detections designed to discover suspicious anomalies were proposed in this chapter. However, note that far more detections could be implemented on top of the extracted data, especially since the set of properties extracted from each sample was intentionally selected to be very broad. In fact, various other heuristic detections were also tested during the research for this thesis. One such detection was based on clustering functions according to their recovered function names, observing typical properties of functions within each cluster, and reporting samples that contained functions with atypical properties²⁸. Another detection was designed to look for suspicious changes between Rich headers of consecutive versions of some software in order to detect trojanization artifacts similar to the ones observed in the ShadowPad and CCleaner backdoors. There were also detections intended to search for potential new variants of existing backdoors: it is possible that the backdoor authors would not implement a completely new backdoor each time, but would instead reuse some techniques from one of the already discovered backdoors. These detections attempted to uncover such backdoors by searching for sequences of API calls and suspicious code fragments similar to the ones observed in backdoors that were discussed back in Chapter 3.

There were also many ideas for more complex heuristic detections, which were unfortunately not explored due to the limited scope of this thesis. The main ideas will be outlined here, since they could inspire further research into this topic. First of all, it might be a good idea to begin by clustering together samples built with similar compiler settings and applying the heuristic detections to each cluster individually. Samples within a cluster should share a more homogeneous implementation of statically linked library functions, so this approach could significantly lower the number of false positives. Various different clustering methods might be used. One promising approach involves clustering by the contents of the Rich header, but this would, of course, only work for samples that actually contain one.

Secondly, it would be interesting to consider using one of the existing graph-based anomaly detection methods [2]. Since the output of the extractor is essentially a graph where functions are treated as vertices and cross references as edges, these methods could be applied directly to the extracted data. However, these methods might detect only generic anomalies, and it

²⁸The motivation for this heuristic detection was the possibility that backdoor authors would recompile an existing library function with added malicious code. If they did that, F.L.I.R.T. might still recognize the recompiled function by its original name, but other properties of this function could differ considerably. The backdoor authors could also trojanize a library function by inserting an unconditional `jmp` instruction to the backdoor code. That could cause the library function to suddenly contain more function chunks than usual, which would also be an anomaly that is supposed to be reported by this heuristic detection.

will be very challenging to guide them towards focusing on the detection of anomalies that could have been caused by trojanization.

Other data mining techniques used for anomaly detection could also be employed. However, note that some of these techniques provide only very little explainability. For instance, they might report a sample as anomalous without actually pointing to a specific address in the sample where the anomaly resides and without providing a simple explanation as to what makes this specific sample anomalous. Since the heuristic detections often rely on manual verification of the reported anomalies, low explainability could make this process very time-consuming. In the absence of any explanation, the whole reported sample would have to be analyzed. Therefore, it would be best to select techniques that make it easy to verify if the reported anomalies are benign in nature or if they were caused by trojanization.

Finally, one could also combine multiple heuristic detections into a single detection algorithm. For instance, the training set of some detections could be chosen randomly from samples that were not reported as anomalous by any other heuristic detection. This would ensure that the training set contains a lower percentage of anomalous samples, which would presumably improve the detection's performance, since the anomalous samples would no longer taint the training set and prevent the detection from learning what most regular executables look like.

5.5 Evaluation

The false positive rate of the three examined heuristic detections was found to be below 0.001, which means less than one in a thousand benign samples got reported as anomalous. While this may seem like a reasonably low number, a lot of false positives might still be generated for larger datasets. That could be problematic, since each anomaly should ideally be manually verified by reverse engineering. Searching for backdoors in a very large dataset would therefore require a substantial amount of reverse engineering, unless the heuristic detections are improved to better filter out benign anomalies or unless some triage method is used to prioritize the most suspicious anomalous samples. Fortunately, all of the detections also report the locations of anomalies within the anomalous sample. This makes it much easier to verify the reported anomalies, since the reverse engineer can focus only on a small part of the sample instead of having to reverse engineer all of it.

The proposed heuristics were designed to aid in the hunt for undiscovered backdoors that could potentially be active in-the-wild. For this purpose, a dataset of supposedly backdoor-free executables has to be collected in order to scour it for potential backdoors. However, note that since many situations in which the heuristics would fail to discover a backdoor were discussed, these detections cannot be relied on if one wants to be absolutely sure that there

is no backdoor in the analyzed dataset. Even worse, the false negative rates of the proposed heuristics were not adequately quantified, which means that it is not yet entirely clear how likely they would be to actually detect an undiscovered backdoor.

The size of the analyzed dataset is an important consideration. It would make sense to build a dataset as large as possible in order to maximize the chance that it would contain an undiscovered backdoor. However, enormous datasets could be problematic because of their storage demands, the CPU time required by the extractor, and perhaps most importantly, the number of reported false positives. When applied to the dataset of one million samples, the heuristic detections reported hundreds of false positives, so it is easy to see that the number of false positives could quickly get out of hand for larger datasets. Note that while one million might seem low compared to the total number of PE samples in-the-wild, the dataset was assembled from the most prevalent samples, since backdoors found in them would be among the most impactful. The size of the dataset could be increased by adding less and less prevalent samples, but at some point, backdoors found in the freshly added samples might not be interesting enough because of their low prevalence.

Software vendors could also use these heuristics as an additional safety measure in order to lower the chance they would release a piece of software with an implanted backdoor²⁹. They could obtain a dataset of benign samples, add their software to it, and run the heuristic detections on the resulting dataset. Since they would be interested only in anomalies found in their software, the number of false positives should not pose a problem. In theory, they could even tweak the parameters of some heuristic detections in order to improve their recall (true positive rate) at the expense of a higher false positive rate. Also, note that many false positives could still provide valuable information to them. As was shown in this chapter, most false positives actually pointed out unusual characteristics of the affected samples. In some cases, these unusual characteristics seem to be a result of misconfiguration or software bugs that the vendor might not have been aware of.

A significant percentage of the reported anomalies have also been caused by imperfections of static analysis, especially by F.L.I.R.T. failing to recognize statically linked library functions. This leads to another potential application of the heuristic detections: they could be used to point out errors exhibited by a static analysis tool in order to suggest new ways to improve the tool. For instance, the heuristic detection discussed in Section 5.2 could be modified so that it would report potential misses of F.L.I.R.T. These misses could be used to update F.L.I.R.T. signatures, either manually or automatically.

²⁹In a similar fashion, organizations could also use these heuristics to try to detect trojanized software running in their network. They could create a dataset by collecting PE files from the machines they control and examine this dataset using the proposed heuristic detections.

No new backdoors were found in this thesis. This could be either because there was no undiscovered backdoor in the analyzed dataset or because the proposed heuristic detections failed to find one. The detections did at least discover many samples affected by some execution flow hijack method, but, apart from the already known backdoors, none of those samples were found to be malicious.

The heuristic detections were designed to detect techniques used by backdoor authors to hijack execution flow. As was discussed in Section 2.4, there are many various techniques that they can implement in order to achieve that goal. Unfortunately, not all of them were addressed by the heuristic detections proposed in this thesis. This indicates that the proposed heuristic detections are very narrowly focused on detecting backdoors similar to the ones discussed in Chapter 3. Since the number of PE application backdoors discovered in-the-wild is extremely low, it is hard to predict just how similar will the next backdoor be and if it would get discovered by one of the proposed detections. Once the next backdoor gets discovered, it can be used to further evaluate the proposed heuristic detections and to suggest further improvements.

Perhaps the biggest challenge encountered in this thesis was lowering the false positive rate. The proposed heuristic detections were similar in nature to unsupervised anomaly detection algorithms, which are often associated with a high false positive rate. The analyzed dataset was found to be extremely heterogeneous, with a large number of anomalous samples, but the vast majority of them were benign. The anomalies in the benign samples were often very similar to the anomalies observed in in-the-wild backdoors despite them not being caused by trojanization. Instead, these anomalies often originated from unusual low-level optimizations, unusual compiler settings, and from software protection systems. These anomalies could perhaps be more easily filtered out with a labeled dataset approach, which would enable the usage of heuristic detections more similar to supervised anomaly detection algorithms. However, creating such a dataset would be very time-consuming, since labeling a sample as benign would require proving that there is no highly elusive backdoor hidden in it.

Conclusion

The beginning of this thesis is devoted to the theory of application backdoors. First, various kinds of backdoors are described, along with the diverse motives that malicious actors could have for creating them. Then, the existing mechanisms designed to prevent backdoors are discussed, and it is explained that a sufficiently skilled attacker might be able to bypass them. This is followed by a description of two approaches to backdooring Portable Executable files. The first approach, packing, is easier to implement but also less stealthy. The second approach, patching, can be implemented in a multitude of ways, depending on where the malicious payload gets stored and how the execution flow gets hijacked.

This theory is then illustrated with the analysis of four application backdoors that were previously discovered in-the-wild. All the analyzed backdoors were created by hijacking execution flow in statically linked code from the C Runtime library. They all also follow a similar multistage design, and their final stage is typically downloaded from the Internet only for a specific targeted subset of users. While their authors seemed to put a lot of effort into making the backdoors hard to find, they still left some artifacts behind in the backdoored applications. These include additional Rich header entries, suspicious code fragments, and PDB paths containing malware-related strings. These artifacts are pointed out in this thesis, in hopes that they could lead to the discovery of similar backdoors. The analyzed backdoored applications also contain anomalous cross references which are a side effect of the used execution flow hijack method, and it is argued that similar cross references could also be used to find yet undiscovered backdoors.

This thesis then presents a new tool that processes Portable Executable files and extracts their static features into an SQL database. Its primary purpose is to make it possible to quickly search for anomalies using the extracted features of millions of Portable Executable files. It is built on top of IDA Pro, and it mostly focuses on the extraction of function information. Each analyzed function is characterized by over twenty features in the output

database. This tool deliberately extracts more features than are needed for the purposes of this thesis in order to make it easier to develop and evaluate further heuristic detections in the future.

Finally, three heuristic detections that use the extracted features were implemented. They all focus on detecting anomalies that could have been caused by the use of some execution flow hijack method. The first detection is designed to find instances of entry point hijacking. It managed to find some such instances in the evaluation dataset, but none of them were malicious. Surprisingly, it also managed to detect a backdoor that was not created using entry point hijacking but instead using a very similar technique. The second detection is inspired by the ShadowHammer backdoor, and it is supposed to detect hijacked cross references to statically linked library functions. Unfortunately, apart from the ShadowHammer backdoor itself, it did not find anything malicious. The last detection is designed to detect hijacked cross references to imported functions. It managed to find some interesting hijacks, but none of them were a part of a backdoor.

As indicated, none of the proposed heuristic detections found any new backdoors in the evaluation dataset, but this could very well be because there were no undiscovered backdoors in there. Admittedly, the detections are very narrowly focused on detecting only backdoors that were created using specific execution flow hijack methods. While future backdoors could continue using similar methods, backdoor authors could also invent new ones that would not be covered by the proposed heuristic detections. Therefore, further research could focus on identifying additional execution flow hijack methods and designing new detections that would recognize their usage. Another way to improve on this research would be to figure out how to further reduce the number of false positives generated by the proposed heuristic detections. While the current false positive rates were found to be reasonably low, lowering them even further would undoubtedly make the heuristic detections easier to work with.

Bibliography

- [1] abatchy17. *Introduction to Manual Backdooring*. 2017, <https://www.exploit-db.com/docs/english/42061-introduction-to-manual-backdooring.pdf> [accessed 05-May-2020].
- [2] Akoglu, L.; Tong, H.; et al. *Graph-based Anomaly Detection and Description: A Survey*. *Data Mining and Knowledge Discovery*, volume 29, no. 3, 2015: p. 626–688, ISSN 1384-5810, doi:10.1007/s10618-014-0365-y.
- [3] Alvarez, R. *Bird's Nest*. Technical report, Fortinet, Canada, 2014, <https://www.virusbulletin.com/virusbulletin/2014/08/bird-s-nest> [accessed 05-May-2020].
- [4] Balci, E. *Art of Anti Detection 2: PE Backdoor Manufacturing*. Technical report, Invictus Cyber Security Services, 2017, <https://pentest.blog/art-of-anti-detection-2-pe-backdoor-manufacturing/> [accessed 05-May-2020].
- [5] Bencsáth, B.; Pék, G.; et al. *The Cousins of Stuxnet: Duqu, Flame, and Gauss*. *Future Internet*, volume 4, 2012: pp. 971–1003.
- [6] Bernstein, D. J.; Lange, T.; et al. *Dual EC: A Standardized Back Door*. In *LNCS Essays on The New Codebreakers – Volume 9100*, Springer-Verlag, 2015, ISBN 9783662493007, p. 256–281, doi:10.1007/978-3-662-49301-4_17, <https://eprint.iacr.org/2015/767> [accessed 05-May-2020].
- [7] Blaze, M. *Protocol Failure in the Escrowed Encryption Standard*. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security, CCS '94*, Association for Computing Machinery, 1994, ISBN 0897917324, p. 59–67, doi:10.1145/191177.191193, <https://www.mattblaze.org/papers/eesproto.pdf> [accessed 05-May-2020].

- [8] Braskamp, C.; Soffronoff, J. *The Dangerous World of Counterfeit and Pirated Software*. Technical report, Harrison Group, 2011, <https://news.microsoft.com/download/archived/presskits/antipiracy/docs/GenuineMSProducts.pdf> [accessed 05-May-2020].
- [9] Brumaghin, E.; Carter, E.; et al. *CCleaner Command and Control Causes Concern*. Technical report, Cisco Talos Intelligence, 2017, <https://blog.talosintelligence.com/2017/09/ccleaner-c2-concern.html> [accessed 04-May-2020].
- [10] Buchanan, B. *Nobody But Us: The Rise and Fall of the Golden Age of Signals Intelligence*. Hoover Working Group on National Security, Technology, and Law, 2017, https://www.hoover.org/sites/default/files/research/docs/buchanan_webready.pdf [accessed 05-May-2020].
- [11] ByteBiter; Benhut1. *Portable Executable 32 bit Structure in SVG fixed*. https://commons.wikimedia.org/wiki/File:Portable_Executable_32_bit_Structure_in_SVG_fixed.svg [accessed 04-May-2020].
- [12] Collberg, C.; Thomborson, C.; et al. *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs*. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, Association for Computing Machinery, 1998, ISBN 0897919793, p. 184–196, doi:10.1145/268946.268962.
- [13] _Dark_Knight_. *Backdooring PE Files – Part 2*. 2013, <https://sector876.blogspot.com/2013/03/backdooring-pe-files-part-2.html> [accessed 05-May-2020].
- [14] Dubyk, M. *Leveraging the PE Rich Header for Static Malware Detection and Linking*. 2019, <https://www.sans.org/reading-room/whitepapers/reverseengineeringmalware/leveraging-pe-rich-header-static-malware-detection-linking-39045> [accessed 05-May-2020].
- [15] Giorgos, P. *Advanced Antivirus Evasion Techniques*. Master’s thesis, University of Piraeus, Department of Digital Systems, 2015, http://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/9122/Poulios_Giorgos.pdf [accessed 05-May-2020].
- [16] Gorelik, M. *Morphisec Discovers CCleaner Backdoor Saving Millions Of Avast Users*. Technical report, Morphisec Technologies, 2017, <https://blog.morphisec.com/morphisec-discovers-ccleaner-backdoor> [accessed 04-May-2020].

-
- [17] Halderman, J. A.; Felten, E. W. *Lessons from the Sony CD DRM Episode*. In Proceedings of the 15th Conference on USENIX Security Symposium – Volume 15, USENIX-SS’06, USENIX Association, 2006, https://www.usenix.org/legacy/events/sec06/tech/full_papers/halderman/halderman.pdf [accessed 05-May-2020].
- [18] Hardikar, A. *Malware 101 – Viruses*. 2020, <https://www.sans.org/reading-room/whitepapers/incident/malware-101-viruses-32848> [accessed 05-May-2020].
- [19] Hex-Rays. *IDA F.L.I.R.T. Technology: In-Depth*. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml [accessed 04-May-2020].
- [20] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual*. 2016, <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf> [accessed 04-May-2020].
- [21] ir3t. *Introduction to Various File Infection Techniques*. 2010, <http://index-of.es/Viruses/EN-Introductiontovariousfileinfectiontechniques.pdf> [accessed 05-May-2020].
- [22] Karger, P. A.; Schell, R. R. *Thirty Years Later: Lessons from the Multics Security Evaluation*. In 18th Annual Computer Security Applications Conference, 2002., 2002, pp. 119–126.
- [23] Kaspersky GReAT. *ShadowPad in Corporate Networks*. Technical report, 2017, <https://securelist.com/shadowpad-in-corporate-networks/81432/> [accessed 04-May-2020].
- [24] Kaspersky GReAT/AMR. *Operation ShadowHammer*. Technical report, 2019, <https://securelist.com/operation-shadowhammer/89992/> [accessed 04-May-2020].
- [25] Kaspersky GReAT/AMR. *Operation ShadowHammer: a High-profile Supply Chain Attack*. Technical report, 2019, <https://securelist.com/operation-shadowhammer-a-high-profile-supply-chain-attack/90380/> [accessed 04-May-2020].
- [26] Kaspersky Lab. *ShadowPad: Popular Server Management Software Hit In Supply Chain Attack*. Technical report, 2017, https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2017/08/07172148/ShadowPad_technical_description_PDF.pdf [accessed 04-May-2020].

- [27] Keragala, D. *Detecting Malware and Sandbox Evasion Techniques*. 2016, <https://www.sans.org/reading-room/whitepapers/forensics/detecting-malware-sandbox-evasion-techniques-36667> [accessed 04-May-2020].
- [28] Kruegel, C.; Robertson, W.; et al. *Static Disassembly of Obfuscated Binaries*. In Proceedings of the 13th Conference on USENIX Security Symposium – Volume 13, SSYM'04, USENIX Association, 2004, p. 18, https://sites.cs.ucsb.edu/~chris/research/doc/usenix04_disasm.pdf [accessed 04-May-2020].
- [29] kyREcon. *Shellter*. 2020, <https://www.shellterproject.com/Downloads/Shellter/Readme.txt> [accessed 05-May-2020].
- [30] Lakhotia, A. *Constructing Call Multigraphs Using Dependence Graphs*. In Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93, Association for Computing Machinery, 1993, ISBN 0897915607, p. 273–284, doi:10.1145/158511.158647.
- [31] Linderud, M. *Reproducible Builds: Break a Log, Good Things Come in Trees*. Master's thesis, University of Berges, Department of Information Science and Media Studies, 2019.
- [32] LogRhythm Labs. *NotPetya Technical Analysis*. Technical report, 2017, <https://logrhythm.com/pdfs/threat-intelligence-reports/notpetya-technical-analysis-threat-intelligence-report.pdf> [accessed 04-May-2020].
- [33] Léveillé, M.-E. *Gaming Industry Still in the Scope of Attackers in Asia*. Technical report, ESET, 2019, <https://www.welivesecurity.com/2019/03/11/gaming-industry-scope-attackers-asia/> [accessed 04-May-2020].
- [34] Léveillé, M.-E.; Tartare, M. *Connecting The Dots: Exposing the Arsenal and Methods of the Winnti Group*. Technical report, ESET, 2019, https://www.welivesecurity.com/wp-content/uploads/2019/10/ESET_Winnti.pdf [accessed 04-May-2020].
- [35] Mahmood, R.; Mahmoud, Q. H. *Evaluation of Static Analysis Tools for Finding Vulnerabilities in Java and C/C++ Source Code*. ArXiv, volume abs/1805.09040, 2018, <https://arxiv.org/ftp/arxiv/papers/1805/1805.09040.pdf> [accessed 04-May-2020].
- [36] Meng, X.; Miller, B. P. *Binary Code is Not Easy*. In Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Association for Computing Machinery, 2016, ISBN 9781450343909, p. 24–35, doi:10.1145/2931037.2931047.

-
- [37] Microsoft Corporation. *_initterm, _initterm_e*. <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/initterm-initterm-e> [accessed 04-May-2020].
- [38] Microsoft Corporation. *IsDebuggerPresent Function*. <https://docs.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent> [accessed 04-May-2020].
- [39] Microsoft Corporation. *PE Format*. <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format> [accessed 04-May-2020].
- [40] Microsoft Corporation. *VERSIONINFO Resource*. <https://docs.microsoft.com/en-us/windows/win32/menurc/versioninfo-resource> [accessed 04-May-2020].
- [41] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*. 1999, <http://www.skyfree.org/linux/references/coff.pdf> [accessed 04-May-2020].
- [42] Microsoft Corporation. *Windows Authenticode Portable Executable Signature Format*. 2008, https://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/authenticode_pe.docx [accessed 04-May-2020].
- [43] Møller, A.; Schwartzbach, M. I. *Static Program Analysis*. Department of Computer Science, Aarhus University, 2018, <https://cs.au.dk/~amoeller/spa/> [accessed 04-May-2020].
- [44] Oreans Technologies. *Themida Help File*. 2010, <https://www.oreans.com/ThemidaHelp.pdf> [accessed 04-May-2020].
- [45] Pitts, J. *The Backdoor Factory*. GitHub repository, 2013, <https://github.com/secretsquirrel/the-backdoor-factory> [accessed 05-May-2020].
- [46] Pitts, J. *Repurposing OnionDuke: A Single Case Study Around Reusing Nation State Malware*. 2015, <https://www.blackhat.com/docs/us-15/materials/us-15-Pitts-Repurposing-OnionDuke-A-Single-Case-Study-Around-Reusing-Nation-State-Malware-wp.pdf> [accessed 05-May-2020].
- [47] Poslušný, M. *BackSwap Malware Finds Innovative Ways to Empty Bank Accounts*. Technical report, ESET, 2018, <https://www.welivesecurity.com/2018/05/25/backswap-malware-empty-bank-accounts/> [accessed 04-May-2020].

- [48] Powers, D. M. W. *Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness And Correlation*. 2007, <https://csem.flinders.edu.au/research/techreps/SIE07001.pdf> [accessed 05-May-2020].
- [49] Priya, R. H.; Bhagavan, K. *Anti-Reverse Engineering Techniques Employed by Malware*. International Journal of Innovative Technology and Exploring Engineering, volume 8, 2019, <https://www.ijitee.org/wp-content/uploads/papers/v8i6s3/F10670486S319.pdf> [accessed 04-May-2020].
- [50] Qiu, J.; Su, X.; et al. *Library Functions Identification in Binary Code by Using Graph Isomorphism Testings*. In 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015, pp. 261–270.
- [51] Schneier, B.; Fredrikson, M.; et al. *Surreptitiously Weakening Cryptographic Systems*. IACR Cryptology ePrint Archive, volume 2015, 2015: p. 97, <https://www.schneier.com/academic/paperfiles/paper-weakening.pdf> [accessed 05-May-2020].
- [52] Scrinzi, F. *Behavioral Analysis of Obfuscated Code*. Master’s thesis, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science, 2015, https://essay.utwente.nl/67522/1/Scrinzi_MA_SCS.pdf [accessed 04-May-2020].
- [53] Shirani, P.; Wang, L.; et al. *BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape*. In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2017, ISBN 978-3-319-60875-4, pp. 301–324, doi:10.1007/978-3-319-60876-1_14.
- [54] Sikorski, M.; Honig, A. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, first edition, 2012, ISBN 1593272901.
- [55] skape. *Understanding Windows Shellcode*. 2003, <http://www.hick.org/code/skape/papers/win32-shellcode.pdf> [accessed 05-May-2020].
- [56] Středa, A.; Camastra, L. *The Tangle of WiryJMPer’s Obfuscation*. Technical report, Avast Software, 2019, <https://decoded.avast.io/adolfstreda/the-tangle-of-wiryjmpers-obfuscation/> [accessed 04-May-2020].
- [57] Szor, P. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005, ISBN 0321304543.

- [58] Thompson, K. *Reflections on Trusting Trust*. Communications of the ACM, volume 27, no. 8, 1984: p. 761–763, ISSN 0001-0782, doi: 10.1145/358198.358210.
- [59] Webster, G.; Kolosnjaji, B.; et al. *Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage*. 2017.
- [60] Wysopal, C. *Static Detection of Application Backdoors*. 2010, <https://www.veracode.com/sites/default/files/Resources/Whitepapers/static-detection-of-backdoors-1.0.pdf> [accessed 05-May-2020].
- [61] Yeboah-Ofori, A.; Islam, S. *Cyber Security Threat Modeling for Supply Chain Organizational Environments*. Future Internet, volume 11, 2019: p. 63.

Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
C&C	Command and Control
COFF	Common Object File Format
CPU	Central Processing Unit
CRT	C Run-time
CUI	Character User Interface
DEP	Data Execution Prevention
DGA	Domain Generation Algorithm
DLL	Dynamic-link Library
DNS	Domain Name System
EFI	Extensible Firmware Interface
ELF	Executable and Linkable Format
F.L.I.R.T.	Fast Library Identification and Recognition Technology
GUI	Graphical User Interface
IAT	Import Address Table
IDA	Interactive Disassembler
JSON	JavaScript Object Notation

A. ACRONYMS

MAC	Media Access Control
MitM	Man in the Middle
MMX	MultiMedia eXtension
MS-DOS	Microsoft Disk Operating System
MSVC	Microsoft Visual C++
NSIS	Nullsoft Scriptable Install System
PDF	Portable Document Format
PE	Portable Executable
PEB	Process Environment Block
PIC	Position-independent Code
PLT	Procedure Linkage Table
PRNG	Pseudorandom Number Generator
RAT	Remote Administration Tool
RVA	Relative Virtual Address
SEH	Structured Exception Handling
SQL	Structured Query Language
TLS	Thread Local Storage
URL	Uniform Resource Locator

Contents of the Enclosed CD

README.md.....	the description of the CD contents
thesis.pdf.....	the text of this thesis
extractor.....	the call graph extractor described in Chapter 4
├─ config.json.....	the extractor's configuration file
├─ extractor.py.....	the extractor's main Python module
├─ LICENSE.....	the free software license used for the extractor
├─ README.md.....	instructions on how to use the extractor
heur.....	the heuristic detections described in Chapter 5
├─ anomalous_xrefs.py..	the heuristic detection described in Section 5.2
├─ hijacked.ep.py.....	the heuristic detection described in Section 5.1
├─ import_hijack.py....	the heuristic detection described in Section 5.3
├─ LICENSE.....	the free software license used for the heuristic detections
├─ README.md.....	instructions on how to run the heuristic detections
thesis.....	L ^A T _E X source code of the thesis
zoo.....	analyzed backdoors and their IDA Pro database files
├─ asian_gaming_industry..	samples from the gaming industry incidents
├─ bdf.....	analyzed Backdoor Factory samples
├─ ccleaner.....	analyzed samples from the CCleaner incident
├─ shadowhammer.....	analyzed ShadowHammer samples
├─ shadowpad.....	analyzed ShadowPad samples
├─ shellter.....	analyzed Shellter samples