Detecting fileless malware using Endpoint Detection and Response tools

Lautaro Lecumberry
Supervised by
Prof. Nicolás Wolovick and
Dr. Michael Denzel

Facultad de Matemática, Astronomía, Física y Computación
Universidad Nacional de Córdoba, 2023

Abstract

Damage caused by malware has been ramping up in the last five years [1]. One kind of malware is fileless malware, which increased 900 percent in 2020, and it is expected to be half of the attacks against enterprise environments in 2022 [8] [9]. To detect fileless malware, we matched code segments from the executables loaded into Random Access Memory to the original executable file stored on hard disk, using Endpoint Detection and Response tools to implement it. Furthermore, we tested the technique against real malware families, resulting in a detection rate of 77.78 percent, with a sensitivity rate of 92.11 percent. In summary, we present a technique to detect fileless malware, and the results of the testing phase sound promising.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Damage done by cyberattacks has been on the rise, ramping up in the last 5 years. Cybercrime reported to Internet Crime Complaint Center (IC3) alone has caused an estimated 7 billion US dollars in damage during 2021 [1]. Furthermore, even the lives of people are at risk because of some issues caused by cyberattacks [2].

Another target that is on the rise is critical infrastructure [3]. Infrastructure like pipelines and power grids are becoming more likely to get attacked with the goal of interrupting the targets everyday living, while causing economical damage to them [4] [5].

Attackers are constantly looking for new techniques to improve the attacks, both for new infection mechanisms and also for staying undetected afterwards. An example of the use of unprecedented techniques to achieve a successful cyberattack is the Stuxnet attack [6].

One of the possible methods are fileless attacks. These attacks are the ones that are not 100 percent based on files, which gives them the advantage of being harder to detect as there is no file to discover the successful attack. Considering that fileless attacks raised 900 percent during 2020 and it is predicted that they would consist on 50 percent of the all attacks against enterprise environments in 2022, it is extremely important to detect the threat [8] [9].

The memory based attacks also render antivirus traditional approach useless, since it consists in comparing the malware file hash with all the hashes from already known malware stored on a database, but there is no file to calculate the hash [7]. To fix the previous issue, techniques related to Random Access Memory (RAM) forensics can be used [10].

Another available tools are EDR. EDR are tools that can perform a

broad different functions related to forensics on different devices [11]. Some of the functions are related to memory forensics.

In this project, a detection for fileless malware will be presented. The previous will be done based on memory forensic techniques using EDRs in order to get the necessary information.

1.1 Airbus

This research project has been done during an internship in the Computer Security Incident Response Team (CSIRT) at Airbus Protect GmbH. Although the project was carried out in this context, the main purpose was academic and all decisions made were aimed at obtaining the best possible result from a research point of view.

The team consisted in a group of incident responders and a group of penetration testers.

Chapter 2

Background and literature review

2.1 Forensics

According to the Cambridge Dictionary, forensics are the "scientific methods of solving crimes, that involve examining objects or substances related to a crime" [12].

2.1.1 Digital Forensics

With classic forensics in mind, it is possible to transfer the concept into the digital world and define the area of digital forensics as the scientific methods involving examining digital objects. As expected, it is the idea behind SysAdmin, Audit, Network and Security (SANS) definition of digital forensics as "the forensic discipline that deals with the preservation, examination and analysis of digital evidence" [13].

From the time perspective, the first place where digital forensics has been applied was to storage devices. In this approach, first the machine is plugged off, second the storage device is imaged, i.e. create a bit-by-bit copy of it, and third the image is analyzed. Considering that forensics involves scientific methods, the steps done during the analysis phase should be reproducible by someone else, which is also important if the results of the analysis are part of a court process [14].

One of the downsides of this approach is that the context of the computer is lost in the moment the computer has been turned off, so information, like active network connections and running processes, is lost, which could have

a huge impact on the result of the investigation.

Another downside is the lack of scalability. It can take a long amount of time to copy a, for example, 1 Terabyte (TB) storage device. After copying the content, it is also difficult to analyze an image of that size. Taking into account that nowadays a 1 TB storage device is something possible to have on a personal computer and that, most certain, those numbers will keep growing, this problem is likely to get bigger and bigger [18].

In addition, the presence of Network Attached Storage (NAS) devices with several TB of storage is not unlikely, which massively expands the volume of analysis data [19].

2.1.2 Live forensics

The concept of live forensics is summarized by Adelstein [20] on the title of one of his articles: "diagnosing your system without killing it first". Without shutting down the computer, the idea of this approach is to constantly get information of the storage devices, instead of creating an image out of it and then analyze the static approach.

Not killing the computer has many consequences. One of them is that the complete context is maintained because the computer is not turned off. Also, it is possible to access the volatile information stored on RAM but also the non-volatile storage devices, so more information is available.

Furthermore, the author claim that another advantage is the possibility to constantly get additional information, which leads to a more scalable approach because it is not necessary to get all information at once. It may not even be necessary to gather all the information at the beginning as it is possible to get it when it is necessary.

However, one downside of this technique is that the reproducibility of the analysis phase is lost because the source of the information is constantly changing [18] [21].

2.2 Portable executable file format

The PE file format is a type of files which specifies the structure of executable files and object files in the entire Microsoft Windows family of operating systems, where the word "portable" in the name comes from, as is not for a specific architecture but for the entire family [22].

This file format is also known as Common Object File Format (COFF) files and the most common extensions are *.exe* and *.dll*. Mach-O in macOS and Executable and Linkable Format (ELF) in Linux are the similar formats.

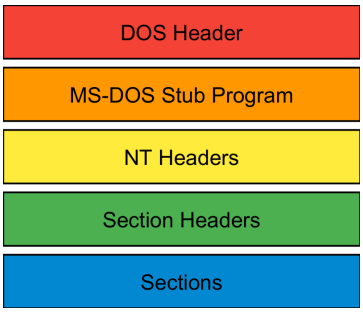

Figure 2.1: Portable Executable (PE) file format structure.
Each color means a different portion of the PE file format.

As shown in figures 2.1 and 2.2, there are different types of headers present in this file format. The first one is the Disk Operating System (DOS) header, which is required for the executable file to run under Microsoft DOS because if this header is present then the DOS stub is executed instead of the actual executable, but, without this header, it will produce an error. Then, the previously mentioned DOS stub is a valid application that is able to run on Microsoft DOS but it only prints "This program cannot be run in DOS mode" [23].

After, the DOS stub there is the Rich Header, which is an undocumented structure present when the executable file is compiled using Visual Studio and it has been used by malware creators to pretend that another group had developed certain malware samples [24].

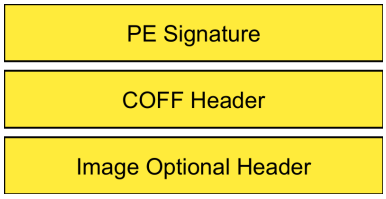

Figure 2.2: NT header subsections.
All entries are yellow because they all belong to the NT headers portion.

Following, there are the NT headers, which is divided in three parts. First, the PE signature, which is the 4 bytes `PE\0\0` and it identifies the file as a PE one. The second one is a standard COFF header, which has information related to the Central Processing Unit (CPU) type where the file can run, the number of sections present on the file, information related to the symbols, the size of the optional header, among other things. Finally,

there is the optional header which, in fact, is not optional: every image file requires one, but it is not required on some other files, like object files.

The first eight fields of the optional header are standard for every COFF implementation and contains the size of the code section or the sum of them if there are more than one, size of initialized and uninitialized data, address of the entry point, which is the starting address for program images, among other information. It is also a little different if it is a PE32 format or a PE32+ format.

After the COFF fields, there are the Windows-specific fields, in which is the image base address, e.g. the preferred address to load the image into memory, the alignment of the file and the sections when they are loaded into memory, size of the image as loaded in memory, size of all the headers combined, image file checksum, and information related to the versions of the operating system, image and subsystem, along with others.

The last header is the section table, which has to be immediately after the optional header because there no direct pointer to this section in the file header. It has one entry for every section in the file and each entry has the name of the section, the virtual address of the section relative to the image base when the section is loaded into memory, information related to the relocation for the entry, the permissions the section should has when loaded into memory, size of the section when loaded into virtual memory and the size of the raw data on disk, among others.

Figure 2.3: Portable Executable (PE) sections.
All entries are blue because they all belong to the sections portion.

As seen in Fig. 2.3, there is an undefined number of sections on a PE file. These sections can have a different combination of characteristics depending on their needs without following any special rule. However, there are some special sections that are common among PE files. Some of them are the following:

- **.bss**: this section contains uninitialized data.
- **.data**: this section contains initialized data.

- **.edata**: this section contains the export tables.
- **.idata**: this section contains the import tables.
- **.pdata**: this section contains exception information.
- **.rdata**: this section contains read-only initialized data.
- **.reloc**: this section contains information related to image relocations.
- **.rsrc**: this section contains the resources used by the program, including images or embedded binaries.
- **.text**: this section contains the executable code, and it is the only special section with executable permissions.
- **.tls**: standing for thread local storage, provides storage for the program's executing threads.

2.3 Memory

Modern operating systems provide an abstraction of the memory to the processes, which is called virtual memory. Memory virtualization is the technique of using virtual memory and has the goal of efficiency and protection, as the processes do not interact with the physical memory. Instead, they interact with the abstraction presented to them by the operating system and the operating system is the one with full control over the physical memory [39].

The address space is the name of this abstraction and each process has a complete address space for itself. It stores all the state of the memory, such as the program code, and the size of the address space is not related with the amount of physical RAM in the computer. As each process has its own, it also serves isolation purposes because it does not directly share memory addresses.

2.3.1 Memory pages

A memory page is a contiguous fragment of virtual memory with a fixed size. It is the smallest division of virtual memory and the equivalent division of physical memory is named page frame [39].

As sometimes the RAM is a limited resource, the operating system has mechanisms to reduce the use of physical memory. These mechanisms are page swapping and demand paging [14].

Page swapping is a mechanism where the operating system stores some of the information that should be in memory in other sources, usually in disk.

Demand paging is a mechanism where the information is not loaded into memory until that information is written or read.

2.3.2 Memory segmentation

In the context of memory management, a segment is a contiguous portion of virtual memory of a non-specific length, for example the program code, the stack and the heap of a certain process. Segmentation is the technique of dividing the memory into segments. Each segment consists in one or more memory pages.

This is used for security purposes as it is known which segments are being used and it is possible to give different permissions to the different segments. The possible permissions are read, write, execute, and the combination of them. With these, the operating system can restrict unwanted behaviors such as executing user's input stored in the stack because that section has read and write permissions only, so it is not possible to execute that section [39].

When more than one process maps the exact same segment, e.g. a library code segment, the operating system can map all the segments in each process virtual memory to the same segment in physical memory. Using this technique the segments are shared between processes for efficiency purposes. However, this is not possible with any segment, only with the ones that would not be modified by the different processes and does not exposed private information.

2.3.3 Portable Executable file format mapping

In order to be accessed and executed on run time, all the PE sections mention in section 2.2 should be present in memory, with the addition of one stack section for each running thread of the process and one heap section for the entire process. Due to segmentation, there is the possibility of having different permissions for each of the sections of the PE file.

Even though each section can have whatever permissions, the special sections has some predefined options as shown in Table 2.1 [22].

Name	Mem read	Mem write	Mem execute
.bss	✓	✓	
.data	✓	✓	
.edata	✓		
.idata	✓	✓	
.pdata	✓		
.rdata	✓		
.reloc	✓		
.rsrc	✓		
.text	✓		✓
.tls	✓	✓	

Table 2.1: Permissions of the Portable Executable (PE) sections when mapped into memory.

2.3.4 Operating system data structures

As the operating system needs information about the current state of the computer, it is necessary to store such information in some kind of data structure, which are stored in memory.

In the Windows family of operating systems one of them is the Virtual Address Descriptor (VAD), which is a tree data structure that describes the virtual memory segments reserved by each process, and information related to them such as memory protection.

```
typedef struct _FILE_OBJECT {
  CSHORT                          Type;
  CSHORT                          Size;
  PDEVICE_OBJECT                  DeviceObject;
  ...
  ULONG                           Flags;
  UNICODE_STRING                  FileName;
  LARGE_INTEGER                   CurrentByteOffset;
  ...
} FILE_OBJECT, *PFILE_OBJECT;
```

Listing 2.1: `_FILE_OBJECT` struct.

Each VAD entry is created when a process allocates memory using `VirtualAlloc`, it does not wait until the process actual references that memory segment. If a file is mapped into this segment, a `_FILE_OBJECT` data structure can be accessed to, from which it is possible to get the file-

name of the file mapped to the segment [34] [35]. The `_FILE_OBJECT` data structured represents an open file, device, directory, or volume [41].

Another important kernel data structure is the `EPROCESS` structure, where information about the running processes are stored [42].

```
typedef struct _EPROCESS {
  KPROCESS Pcb;
  EX_PUSH_LOCK ProcessLock;
  LARGE_INTEGER CreateTime;
  LARGE_INTEGER ExitTime;
  EX_RUNDOWN_REF RundownProtect;
  PVOID UniqueProcessId;
  LIST_ENTRY ActiveProcessLinks;
  ULONG QuotaUsage[3];
  ULONG QuotaPeak[3];
  ULONG CommitCharge;
  ULONG PeakVirtualSize;
  ULONG VirtualSize;
  ...
  ULONG ActiveThreads;
  ULONG ImagePathHash;
  ULONG DefaultHardErrorProcessing;
  LONG LastThreadExitStatus;
  PPEB Peb;
  ...
  MM_AVL_TABLE VadRoot;
  ULONG Cookie;
  ALPC_PROCESS_CONTEXT AlpcContext;
} EPROCESS, *PEPROCESS;
```

Listing 2.2: `EPROCESS` struct.

2.3.5 Memory forensics

The content of the RAM of the computer can be added in order to expand the scope of traditional digital forensics. Adding that information, it would be possible to access more information about the current state of the machine.

Following the steps of the traditional digital forensics techniques over storage devices, the first step is to image the raw physical memory and then analyze it to retrieve some high level information out of the bytes of the image. There are frameworks to help in the analysis phase because images can get excessively big to do it without any help. Some of these frameworks are Volatility, and rekall.

Even as these approach can sound extremely reliable, there are some

problems related to the acquisition phase. One of them is page smearing, which is the inconsistency between state of the memory described by the page tables and the actual content of the memory. This occurs because of the time difference between the acquisition of the different sections of the memory [14].

Another problem is that the image is done over the physical memory content, so all the swapped pages, and demand pages are not included in the image, which can affect analysis as some information is missing.

2.3.6 Live memory forensics

Live memory forensics is the result of combining live and memory forensics. The idea is to access the content of the memory while the computer is still running instead of creating an image of the physical memory [48].

By not shutting down the computer, accessing swapped and demand pages are no longer a problem because the external sources where these pages can be stored are also accessible [20].

It also avoids page smearing because it is no longer necessary to create an image, so there is no issue of copying the different pages at different times.

However, as it happens in live forensics, the results are no longer reproducible.

2.4 Endpoint Detection and Response (EDR)

An Endpoint Detection and Response (EDR) tool is a software that monitors the activities of the end hosts in real time with the goal to raise an alert if malicious behavior is detected. The collected data from each endpoint can be sent to a centralized database where all the information from multiple endpoints is correlated [36].

What traditional antivirus do is comparing the checksum of the files stored on the computer with the checksums stored in databases of known malware, if those checksums matches then the file on the computer is consider a dangerous file.

However, with the previous approach it is granted that antivirus that use this technique will not detect any new malware, they will only detect the ones that are already known.

As EDRs raises alerts based on behavior and not based on signatures, it is possible to fix the previous issue and detect new malware [37].

In Table 2.2 is shown a comparison between different EDR tools.

Name	Open source	Reme-diation	Dead RAM	Live RAM	Expand-able	Immediate deploy
Carbon Black		✓				
Crowdstrike		✓	✓	✓		✓
OSQuery	✓	✓	✓			
OSSEC	✓	✓			✓	✓
Sysmantec		✓				
TheHive	✓					
Velociraptor	✓	✓	✓	✓	✓	✓
Wazuh	✓					

Table 2.2: Endpoint Detection and Response (EDR) tools comparison. The columns highlighted in green indicates the requirements of Section 5.1. The empty cells mean that the tools do not have that feature.

2.5 Malware

Malware, which stands for malicious software, is any program that is created with the goal of harming a computer o a network. In order to achieve such harming, there are many known techniques used overcome the difficulties of each step. Some of these techniques will be introduced in the following chapters [25] [27].

2.5.1 Fileless malware

Fileless malware is a kind of malware that does not use traditional executable as the main resource to perform its activities. Considering this, it is able to evade signature-based detection systems.

This kind of malware uses trusted, legitimate processes and tools already included in the operating systems in order to attack the computer and hide afterwards [38].

2.5.2 Process injection

Process injection is a technique used to execute code chosen by the attacker in the virtual memory address space of a separate live process. The idea behind the technique is to create a trusted and legitimate process and run the chosen code as it is the trusted process and not a malicious one. The consequences of this is accessing the targeted process memory, accessing the targeted process resources, or gaining elevated privileges as well as evading detection from the user [28].

It is worth noting that after the injection has been done, the checksum of the file does not change because the content on disk does is still the same so, the checksum will also be maintained [15] [16].

2.5.3 Process hollowing

Process hollowing is a sub-technique of process injection that consists in creating a new process in suspended mode. Then, allocating a new virtual memory segment, followed by writing the chosen code into the recently allocated segment. Finally, it is necessary to run the process at the recently injected code [29].

It is also possible to unmap the existing code segment, so the injected code is the only code mapped in the process virtual memory [16] [17].

In Windows operating systems, the previously described technique is achieved by calling the following Windows API functions:

1. `CreateProcess` for creating a new process in suspended mode.
2. `VirtualAllocExe` for allocating a new virtual memory segment.
3. `WriteProcessMemory` for writing the code to inject.
4. `SetThreadCreate` or `ResumeThread` for resuming the process.
5. `NtUnmapViewOfSection` for unmapping the original code segment.

2.6 Tools

I use additional tools to gather information about the processes.

One of them is Process Monitor. It is a tool in the Windows Internals framework that gives information about the running processes. This tool has the ability to obtain certain information related to the process injection and process hollowing techniques, such as what processes are created from the malware process. It has a GUI that shows the process activity and it is possible to filter the desired information [30].

Another tool is drstrace. It traces the function calls to the Windows kernel by a certain process [40].

2.7 Velociraptor Query Language (VQL)

VQL is a programming language used to query endpoints in order to collect information and the state of the endpoint. Afterwards, the data is processed on the server [49] [50].

```
SELECT Pid
FROM pslist()
WHERE Name =~ "notepad"
```

Listing 2.3: VQL query example.

As an example, a query written using VQL can be seen in Listing 2.3. This query gets all the process running, return by the `pslist()` plugin. Then, it filters the processes to keep only those whose name matches with the regular expression `notepad`. Finally, it gets the Process Identifier (PID) from all the filtered processes.

VQL keeps the same structure and keywords as Structured Query Language (SQL).

2.8 Related work

In this section, I introduce some other state-of-the-art techniques that looks for process injections using memory forensics related techniques. These techniques are:

- **Block et Al. [43] algorithm**: it is an algorithm that returns the VADs entries with executable pages, which can contain malicious code. It uses page table entries values to get the actual protection of the pages, in order to know which ones are the executable ones.
- **HollowFind [44] [45]**: it detects process hollowing attacks, comparing the content of the Process Environment Block (PEB) with the VAD.
- **malfind [46]**: it finds hidden or injected code in memory, using VAD entries information and page permissions.
- **Srivastava et Al. [47] technique**: it uses execution stack analysis to locate injected code.

Chapter 3

Methodology

3.1 Samples

In order to get a better understanding whether the idea and the implementation were correct, I detonate samples of different malware families to see if it is possible to detect malicious activities in them. I will refer to the term detonate malware as the action of executing malware in a controlled environment.

I choose malware families that use process injection and process hollowing techniques according to the Mitre Att&ck framework classification. Once I select the families, I look for samples of the previously selected families in malware repositories where countless malware samples from different families are available. The repositories mentioned before are MalwareBazaar, TheZoo and the Malware Family Explorer.

I will introduce each of the malware repositories in detail:

- Bazaar is hosted by the security company abuse.ch but anyone can upload samples to it. The website gives metadata related to the sample, such as file extension, SHA256 hash, SHA1 hash, and, in some cases, it assigns a malware family signature to the sample. It is also possible for the uploader to tag the samples according to criteria like malware family, target operating system, among others. I used both tags and signatures in order to find relevant samples.

- TheZoo is a GitHub hosted repository of live malware where also anyone can create a pull request in order to upload new samples. It is possible to find directories named after the samples, with the samples and metadata about them inside each directory.

- Malware Family Explorer is a collection of malware created by vx underground that also has a directory structure with the name of each family and inside each there are samples corresponding to that family.

For the latter two sources, I used the directory names in order to get the correct samples.

Out of the repositories mentioned before, I obtain 79 samples from 41 different families. All the downloaded content are zip files and the actual samples were inside the zip file.

In addition to the public samples mentioned before, the penetration testers of the team made available to me two more state of the art samples. These samples are developed by themselves and they use process injection and process hollowing techniques.

3.2 Malware test architecture

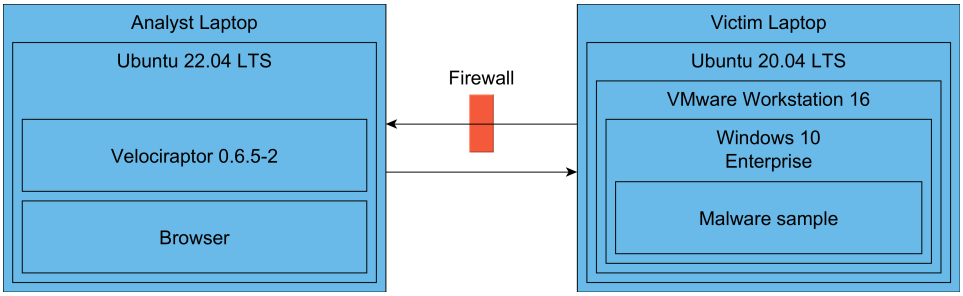

Figure 3.1: Architecture for malware testing.

As displayed in Fig. 3.1, I use two laptops to detonate the samples. Even if I could run the samples in one of the laptops, I prefer this option to hinder the malware to escape the testing environment.

The setup consisted of one of the laptops used as the victim machine, running Ubuntu 20.04 LTS kernel version 5.15.0-46-generic as the host operating system with VMware Workstation 16 Pro version 16.2.4 build-20089737 on it. This laptop executes a Windows virtual machine as a guest operating system, with Windows 10 Enterprise 2016 LTSB version 1607 operating system build 14393.0. Additionally, all Windows Defender options are disabled and the `C:\` directory is whitelisted.

Also, Velociraptor version 0.6.5-2 runs in client mode to gather information from this computer and send it to the Velociraptor server running on the other laptop.

I create a snapshot of the virtual machine while running in clean state to be able to restore to this snapshot after running each malware sample. The goal of this is to recover the clean state of the virtual machine to continue running the following samples.

The second laptop is used as the analyst machine. It runs Ubuntu 22.04 LTS and Velociraptor version 0.6.5-2 running in server mode in order to query the victim machine and extract the results out of it. Additionally, Wi-Fi is turned off and ufw firewall is blocking all incoming connections except port 8000 which is used by Velociraptor to get the results. Additionally, Mozilla Firefox browser is running in order to access Velociraptor's GUI.

In order to achieve the desire isolation, none of the computers were connected to any network. The only exception was connecting the victim laptop to a temporary mobile hotspot with the sole purpose of downloading the malware samples. However, this laptop was only connected to that network while the malware was being downloaded, with the hotspot turned off and the password being changed as soon as the downloading phase was finished.

3.3 Detonating the malware

Once I had the architecture ready, I proceeded to detonate the malware. This architecture is the one described in Section 3.2, consisting of a victim machine and an analyst one. To detonate the malware, I executed the following steps in order:

1. Victim: recover the snapshot.
2. Victim: move the sample zip file from the host to the guest virtual machine.
3. Victim: unzip the file.
4. Analyst: gather data with Velociraptor in order to know the state of the computer before the malware being detonated.
5. Victim: detonate the malware by double clicking in the executable file unzipped in the step before.
6. Analyst: run own Velociraptor queries in order to know whether the malware is detected.

To get more information about what happens during the execution of the malware, I run Process Monitor during the entire execution.

Additionally, I detonate the samples a second time also running drstrace alongside Process Monitor. I decide to run the samples a second time and not just to run drstrace alongside the other tools during the first execution because some malware do not behave in the same way if they are being monitored by an external process in this way.

Chapter 4

Results

4.1 Own implementation

In order to get the necessary information to detect whether any injection is happening in any of the running processes, I implemented two own artifacts in Velociraptor: `Mem2Disk` and `ExtraX`. Their source code can be found on GitHub [51].

`ExtraX` aims to get all the memory sections of the processes with executable permissions, except for the code segment of each `.exe` file, which is covered by the `Mem2Disk` query. To do so, I first create a temporary table with the PID and the name of all the processes currently running. Then, I create a second temporary table with all the VAD tree entries from a certain process, which represent memory sections, that has executable permissions and has no mapping name. It is worth noting that the `vad()` plugin will fail on some processes, such as `System`, because Velociraptor does not have the necessary privileges to access those processes.

Finally, I achieved the desired result of the query by combining the two previous queries. That means that I get the VAD entries that meets the requirements mentioned above for each entry of the first table, that is, every running process. The code of this artifact is in Listing 4.1.

The goal of the `Mem2Disk` query is to check whether the content of the `.text` section of the PE file stored in a non-volatile storage device and the content of the code segment in memory are the same.

```
LET GetPids =
  SELECT Pid,
    Name
  FROM pslist()

LET Compare =
  SELECT Pid,
    Name,
    MappingName,
    Protection
  FROM vad(pid=Pid)
  WHERE Protection =~ "x"
    AND NOT MappingName

SELECT * FROM foreach(
  row=GetPids,
  query=Compare
)
```

Listing 4.1: ExtraX query Velociraptor Query Language (VQL) code.

First, I use the same subquery used in `ExtraX` to get the processes running in the system. Then, I get the address and path from the VAD tree entries that has an existing mapping name, executable and readable permissions and the path has an `.exe` extension. Afterwards, I use the path extracted from the VAD entry to access the portable executable file and get header of the `.text` section, which is first of all the sections of the file. The path, the address and the `.text` header is stored in the `GetMetadata` table.

With the information from the `GetMetadata` table, I get the code segment content from memory, accessing the offset specified in the VAD tree entry, and get the content of the entire `.text` section by reading again the PE file from disk. Later on, I compare both contents and, finally, I repeat the process for every process running on the system.

The code of the `Mem2Disk` artifact is in Listing 4.2.

```
LET GetPids =
  SELECT Pid,
    Username
  FROM pslist()

LET InfoFromVad =
  SELECT Address,
    format(
      format='''C:\%s''',
      args=strip(
        prefix='''\Device\HarddiskVolume2\''',
        string=MappingName)
      ) AS Path
  FROM vad(pid=Pid)
  WHERE MappingName
    AND Protection =~ "xr-"
    AND Path =~ "(exe)$"

LET GetMetadata =
  SELECT Path,
    Address,
    parse_pe(file=Path).Sections[0] AS TextSegmentData
  FROM InfoFromVad

LET GetContent =
  SELECT *,
    format(format="%#x", args=Address) AS AddressHex,
    format(format="%x",
      args=read_file(accessor="process",
        offset=Address,
        filename=format(format="/%d", args=Pid),
        length=TextSegmentData.Size)
      ) AS MemoryData,
    format(format="%x",
      args=read_file(accessor="file",
        offset=TextSegmentData.FileOffset,
        filename=Path,
        length=TextSegmentData.Size)
      ) AS DiskData
  FROM GetMetadata

LET Compare =
  SELECT Pid,
    Path,
    TextSegmentData.Size,
    MemoryData = DiskData AS comparison
  FROM GetContent
  WHERE NOT comparison

SELECT * FROM foreach(
  row=GetPids,
  query=Compare
)
```

Listing 4.2: Mem2Disk query Velociraptor Query Language (VQL) code.

4.2 Testing results

In the following section, I present the test results of the detection techniques running both malware and non-malicious software. The information in the tables is the following:

- **Name**: name of the malware family or software run.
- **SHA256 hash**: the first 8 digits of the executed `.exe` file's SHA256 hash. The full SHA256 hashes can be found in Table A.1.
- **extra alloc**: whether extra executable sections have been allocated.
- **proc create**: whether some additional process has been created.
- **ExtraX det**: whether the ExtraX detection worked.
- **Mem2Disk det**: whether the Mem2Disk detection worked.

4.2.1 Non-malicious software testing

The results of running the non-malicious software are presented in Table 4.1. As shown in the previous table, some of the non-malicious software are detected by `ExtraX` and `Mem2Disk` techniques.

Name	SHA256 hash	extra alloc	proc create	ExtraX det	Mem2Disk det
Adobe Acrobat Reader	966cfa95...	✓	✓	✓	✗
Command Prompt	935c1861...	✗	✓	✗	✗
Discord	a3f9b57e...	✓	✓(own)	✓	✓
Google Chrome	af0bd408...	✓	✓	✓	✓
LibreOffice Writer	7e1ef3b9...	✓	✓	✓	✗
LibreOffice Calc	961ef417...	✓	✓	✓	✗
Microsoft Edge	eb8da1b8...	✓	✓	✓	✓
Microsoft Paint	061d41c4...	✗	✗	✗	✗
Microsoft Teams	8a94b091...	✓	✓(own)	✓	✓
Mozilla Firefox	ead605af...	✓	✓	✓	✓
Spotify	3b84962c...	✓	✓(own)	✓	✓
VLC	1a403269...	✗	✗	✗	✓
Windows Calculator	d7b378a4...	✗	✗	✗	✗
WordPad	0092cd4a...	✗	✗	✗	✗
Zoom	f61f4548...	✗	✓(own)	✗	✗

Table 4.1: Non-malicious software run results.
A ✓in the ExtraX and Mem2Disk det columns means that the detections have detected the programs, while a ✗means they have not.
`(own)` means those processes have started another instance of themselves.

4.2.2 Malware testing

The results of detonation of the malware families mentioned in section 3.1 are presented in Table 4.2.

Out of the detected malware families, many of them create multiple new process, not only the ones to be hollowed. Some families like Ryuk, HyperBro, remcos, among others, execute binaries from temporary folders that are not there before executing the malware. This is a known technique used by different malware families [52].

The tools show that the malware is hollowing some of the temporary binaries after being executed.

However, some of the additional created processes are not being used with the same purpose among the different families. For example, I can detect that AgentTesla and netwire samples are creating an instance of `schtasks.exe`, which is a process to schedule tasks on Windows operating systems. Although the netwire sample is hollowing the process, the AgentTesla sample `8118d7c7...` is not.

Also, the different samples from the same families often have the same behavior among them, as it is the case of GuLoader, and pandora, among

other families.

Even when there are difference, sometimes they are minor, as it is the case of the lokibot family. The difference between the samples of this family is that sample `97301bb...` hollows `msinfo32.exe` process, while sample `afe2844c...` injects the code in `ReAgentC.exe` process.

Furthermore, WhisperGate and remcos hollow the `WerFault.exe` process. This process is the one that pops the alert sign on Windows.

The Hoplight family deletes the original binary after injecting other processes.

Both analyzed samples of the Pandora family unmap the text segment segment and add extra executable segments to execute the desired code.

Name	SHA256 hash	extra alloc	proc create	ExtraX det	Mem2Disk det
AgentTesla	eeedc8ec...	✓	✓	✓	✓
AgentTesla	8118d7c7...	✓	✓	✓	✗
AssemblyInjection	de4cd0c5...	✓	✓	✗	✗
Astaroth	972cae6f...	✓	✓	✗	✗
Astaroth	ce2928ab...	✓	✓	✗	✗
Azorult	08a6193d...	✓	✓	✓	✓
Azorult	5ebb3cc4...	✓	✗	✓	✗
BADNEWS	d07adf20...	✓	✓	✗	✓
bandook	4bf9325f...	✓	✓	✓	✓
bazar	300c0dab...	✓	✗	✓	✗
bazar	534d6039...	✓	✗	✓	✗
Donut	bada6d6d...	✓	✓	✓	✗
dtrack	bf8e3f03...	✓	✗	✗	✓
Dyre	1f8ba528...	✓	✓	✓	✓
Dyre	0350d2ab...	✓	✓	✓	✗
Empire	4a23326d...	✓	✓	✓	✗
formbook	d2f58b0f...	✗	✗	✗	✗
formbook	2c7540c6...	✓	✗	✓	✗
Gazer	c5db84fe...	✗	✗	✗	✗
Gazer	ca9e3ea2...	✗	✗	✗	✗
Gh0stRAT	5a9f06e3...	✓	✓	✓	✓
Gh0stRAT	10ed8da5...	✓	✓	✓	✓
GuLoader	3ae4d65b...	✓	✓	✓	✓
GuLoader	cdbe7a2f...	✓	✓	✓	✓
HopLight	032ccd6a...	✓	✓	✓	✗
HopLight	0237b186...	✓	✓	✓	✗
HTran	1b32e680...	✓	✗	✓	✗
HTran	15b529d0...	✓	✗	✓	✓
HyperBro	6e32c33c...	✗	✗	✗	✗
HyperBro	07f87f7b...	✓	✓	✓	✓
InjectionPoC	e75b681c...	✓	✓	✓	✗
InvisiMole	2debef67...	✗	✗	✗	✗
InvisiMole	d0062f47...	✗	✗	✗	✗
ISMAgent	33c187cf...	✓	✓	✗	✓
ISMAgent	74f61b6f...	✓	✓	✗	✓
Kimsuky	7d89a16f...	✓	✗	✓	✗
Kimsuky	b207265b...	✗	✗	✗	✗
lokibot	97301bb7...	✓	✓	✓	✓
lokibot	afe2844c...	✓	✓	✓	✓
netwire	ea32c3c3...	✓	✓	✓	✓
Pandora	1f172321...	✓	✓	✓	✓
Pandora	5b56c5d8...	✓	✗	✓	✓
PlatinumGroup	021bb772...	✓	✓	✗	✗
PlatinumGroup	46a9ac06...	✓	✓	✓	✗
poshc2	3c4c4cb0...	✓	✓	✓	✓

poshc2	8ba619e1...	✓	✓	✓	✗
qakbot	3be90506...	✓	✓	✓	✗
qakbot	6f00837f...	✓	✓	✓	✗
remcos	03e29815...	✓	✓	✓	✓
remcos	944ec3ee...	✓	✓	✓	✓
REvil	0c10cf1b...	✓	✓	✓	✓
REvil	00d015ed...	✗	✗	✗	✗
RokRAT	af61993f...	✓	✓	✓	✗
RokRAT	9b383ebc...	✓	✓	✓	✓
Ryuk	5e2c9d80...	✓	✓	✗	✓
shadowpad	aef610b6...	✗	✗	✗	✗
sliver	5adc6b62...	✗	✗	✗	✗
sliver	38895ca4...	✗	✗	✗	✗
SlothfulMedia	320cf030...	✓	✗	✗	✓
SlothfulMedia	83131292...	✓	✓	✗	✓
smokeloader	041a05dd...	✓	✓	✓	✓
synack	5b9dee21...	✗	✓	✗	✗
trickbot	b7cbc5e5...	✗	✗	✗	✗
trickbot	47ba62ce...	✗	✗	✗	✗
TsCookie	3cad2031...	✗	✗	✗	✗
TsCookie	80ffaea1...	✓	✓	✓	✗
Turla	44d6d67b...	✗	✗	✗	✗
Turla	95d1f440...	✗	✗	✗	✗
ursnif	104e6094...	✓	✓	✓	✓
ursnif	e3fb27a6...	✓	✓	✓	✓
WarzoneRAT	6c34c666...	✓	✓	✓	✓
WarzoneRAT	8590ebe9...	✓	✓	✓	✓
WhisperGate	b50fb203...	✗	✗	✗	✗
WhisperGate	dcbbae5a...	✓	✓	✓	✓

Table 4.2: Malware detonation results.

A ✓in the ExtraX and Mem2Disk det columns means that the detections have detected the programs, while a ✗means they have not.

Chapter 5

Discussion

5.1 Decisions

5.1.1 Operating system family selection

I select Microsoft Windows as the focus operating system family because it has around 76 percent of the market share for desktops and it was the target of 41.4 million new malware samples during the first half of 2022 out of the 43.8 million in total, that accounts to 94.5 percent of the samples [31] [33].

Furthermore, over 3 million samples of `.exe` files and around 1.7 million of `.dll` files were uploaded to virustotal in the first week of November 2022. This means the second and fourth most uploaded file formats are Windows specific ones. Moreover, these are the first and second most uploaded file types for operating system specific file formats, considering that the first one in the overall ranking are HyperText Markup Language (HTML) files with 3.1 million samples and the third one are JavaScript files with a little more than 2 million samples [32].

The previous fact grants a large set of samples to test whatever techniques I decide to implement, ensuring that the lack of samples will not be a problem.

However, the idea behind the detection relies on having a file where the trusted code is stored and then that content is loaded into memory in order to be executed. As these characteristics are not Windows specific, this general idea is generic and can be implemented for other operating systems.

5.1.2 Malware selection

As my detection technique is focused on the memory behavior of the process, the malware that I consider interesting for testing is the one that has some kind of memory activity.

Even though it is possible to detonate any other samples and try to detect them, it would not be ideal as the detection is not likely to work because it is not intended to do so. Thus, I selected malware families that use process injection and process hollowing, as shown in Table 5.1. This seemed like a representative set to evaluate the implemented techniques.

Name	Technique	Name	Technique
AgentTesla	Process Hollowing	lokibot	Process Hollowing
AssemblyInjection	Process Injection	netwire	Process Hollowing
Astaroth	Process Hollowing	Pandora	Process Injection
Azorult	Process Hollowing	PlatinumGroup	Process Injection
BADNEWS	Process Hollowing	poshc2	Process Injection
bandook	Process Hollowing	qakbot	Process Hollowing
bazar	Process Hollowing	remcos	Process Injection
Donut	Process Injection	REvil	Process Injection
dtrack	Process Hollowing	RokRAT	Process Injection
Dyre	Process Injection	Ryuk	Process Injection
Empire	Process Injection	shadowpad	Process Injection
formbook	Process Hollowing	sliver	Process Injection
Gazer	Process Injection	SlothfulMedia	Process Injection
Gh0stRAT	Process Injection	smokeloader	Process Hollowing
GuLoader	Process Injection	synack	Process Hollowing
HopLight	Process Injection	trickbot	Process Hollowing
HTran	Process Injection	TsCookie	Process Injection
HyperBro	Process Injection	Turla	Process Injection
InjectionPoC	Process Injection	ursnif	Process Hollowing
InvisiMole	Process Injection	WarzoneRAT	Process Injection
ISMAgent	Process Hollowing	WhisperGate	Process Injection

Table 5.1: Different techniques used by each malware family.

5.1.3 Endpoint Detection and Response selection

In order to enable a larger amount of people to reproduce our results and to easily adjust core features of the tool if it is necessary, I decide to use an open source EDR. Also, I choose an EDR with a plugin interface for

expandability, considering it would a necessary feature in order to perform any research project. Finally, to detect fileless malware RAM access is necessary. Both dead and live analysis are possible, but I prefer the live one for the advantages mentioned in section 2.4.

In summary, I pick Velociraptor as the EDR to use because, as seen in Table 2.2, the tool is open source, has a plugin interface, and enables live memory access.

5.1.4 Techniques selection

The idea behind the creation of my artifacts is to cover two places where the injections can happen, since memory segments with executable permissions for the attacker are needed to perform the injection. With the above assumption, there are two methods where the injection is more likely to be executed: allocating a new segment or using the executable memory segments already available.

The first idea it is covered by the `ExtraX` artifact and the second one it is mostly covered by the `Mem2Disk` artifact. However, the code segment of all the mapped Dynamic-Link Libraries (DLLs) are a place where an injection, but I decide to not focus on DLLs memory as it is possible to inject an entire new Dynamic-Link Library (DLL) with the needed code without the need to inject into an existing DLL.

In the `Mem2Disk` artifact I use the sections with no mapping name. I choose these sections because if the region is intended to be used for a mapped file, then the `_FILE_OBJECT` structure is present in the control area that can be found from the VAD entry. However, if the allocation is done by the user, there will not be a `_FILE_OBJECT` associated because it is not supposed to have a mapped file related to the section, and, therefore, this section should not have executable permissions [34].

5.1.5 Detonating malware supported by additional tools

As I detonate the first samples, I consider it necessary to have information related to the process activity.

First, I decide to look for processes being created from the supposed malware process that I detonate. If there is no new process created from the original process, then the process hollowing is not likely to happen. In order to get this information I used Process Monitor.

Also, I decide to track all the function calls to the Windows kernel. With this, it is possible to realize whether the process has allocated extra

executable memory, which can be used to inject code. To track the calls I used drstrace.

5.2 Comparison to other techniques

Compared to the techniques mentioned in Section 2.8, the `ExtraX` technique is similar to `malfind` [46], as it uses VAD tree information and also both of them uses memory pages information in order to detect the injection.

However, they are not the same, as `malfind` focuses on detecting hidden and injected executable sections [46], while `ExtraX` looks for all the executable sections.

Nevertheless, the `Mem2Disk` technique is not directly comparable to any of the presented techniques. It uses VAD content as Block et Al. [43] algorithm and `malfind`, but the technique itself is not similar to them.

While `Hollowfind` uses VAD information to detect the injections, it also uses information from the PEB, making it different from `Mem2Disk` and `ExtraX`.

Furthermore, neither `Mem2Disk` nor `ExtraX` have a similar procedure to the one used in Srivastava et Al. [47]. The three of them only share the goal of detecting injected code.

5.3 Results analysis

5.3.1 Detection rate, sensitivity, and accuracy

As shown in Table 4.2, I tested the `Mem2Disk` and `ExtraX` techniques against different malware families. As only 59 samples have been tested, it has to be considered with caution. However, the preliminary results appear to be valid, and I consider the analysis is reasonably sound. The results of the testing phase are shown in Table 5.2.

	Not-detected	**Detected**	**Total**
Non-malware	10% (6)	17% (10)	27% (16)
Malware	14% (8)	59% (35)	73% (43)
Total	24% (14)	76% (45)	100% (59)

Table 5.2: Results of non-malicious software, and all malware families. The true negatives (TN) is 10 percent, while the false positives (FP) is 17 percent. Also, the false negatives (FN) is 14 percent, and the true positives (TP) is 59 percent. Numbers in brackets are the absolute values.

However, it is worth noting that 5 of the families do not run at all. Most of them show the Windows compatibility "This app can't run on your PC" alert. These families are Gazer, InvisiMole, shadowpad, sliver and Turla. Even though it is possible to think that the Windows compatibility alert is the malware trying to recreate a valid user interface to make the user falls into clicking it, I am confident through my analysis with Process Monitor and drstrace that the malware samples did not run.

I discard the previous 5 families from the discussion, as they cannot run. Table 5.2 is presented for completeness.

	Not-detected	**Detected**	**Total**
Non-malware	11% (6)	19% (10)	30% (16)
Malware	6% (3)	64% (35)	70% (38)
Total	17% (9)	83% (45)	100% (54)

Table 5.3: Results of non-malicious software, and malware families that can be executed.
The true negatives (TN) is 11 percent, while the false positives (FP) is 19 percent. Also, the false negatives (FN) is 6 percent, and the true positives (TP) is 64 percent. Numbers in brackets are the absolute values.

In Table 5.3, I present the results in confusion matrix format with the 5 families already discarded. With these results, I will calculate the detection rate, sensitivity, and accuracy of the techniques. These rates have been calculated according to Ceponis et Al. [53].

As shown in Equation 5.1, 5.2, and 5.3, the detection rate is 77.78 percent, while the sensitivity is 92.11 percent, and the accuracy is 75.93 percent.

$$Detection\ rate = \frac{TP}{TP + FP} * 100 = \frac{35}{35 + 10} * 100 = 77.78 \tag{5.1}$$

5.1: Calculation of the detection rate.

$$Sensitivity = \frac{TP}{TP + FN} * 100 = \frac{35}{35 + 3} * 100 = 92.11 \tag{5.2}$$

5.2: Calculation of the sensitivity rate.

The detection rate is the percentage of probability that the technique is correct when it predicts that the process is malicious. The lower the detection rate, the more non-malicious software is wrongly classified as malicious.

$$Accuracy = \frac{TN + TP}{TN + TP + FN + FP} * 100 = \frac{6 + 35}{6 + 35 + 6 + 10} * 100 = 75.93 \tag{5.3}$$

5.3: Calculation of the accuracy rate.

It is possible to think about it as how many times the EDR tool bothers the user, while it is not actually necessary.

The sensitivity is the percentage of the samples that are correctly classified. The lower the detection rate, the more real malware families the technique is missing to label as such.

Furthermore, the accuracy is the percentage of correct results, including both true positives and true negatives. This rate does not exclude any of the values that are present in the confusion matrix from Table 5.3.

These rates are affected by the false positive results. As detection rate, and accuracy, both where false positives are involved, are lower than the sensitivity.

However, the high percentage of false positives is intrinsique to the design of the technique. My supervisors and I choose to be strict and detect any change that happens in memory. While this approach leads to minimize the amount of false negatives, this also increases the amount of false positives. However, we prefer this as we consider the impact of undetected malware can be bigger than wrongly labeling a non-malicious process as malware.

5.3.2 Test results

With a false negatives of 6 percent, I believe it is not a priority to focus on reducing it. Instead, this percentage should at least be maintained while the other limitations are addressed.

I attribute the false negative results mainly to the timing issues mentioned in Subsection 5.3.3. One of the families that is worth analyzing as an example of this issue is the Ryuk family.

After running the Ryuk sample, `Mem2Disk` and `ExtraX` cannot detect any trace of the activity shown by Process Explorer and drstrace. However, the tools can record memory activity and new processes being created from the original sample. So, I consider the attack is happening but it is too fast to be detected but the processes are already terminated, which means it is no longer possible to access to the process memory.

To fix this, I set the newly created processes status to suspended mode,

so the process is not finished, and I am able to access their memory. When I access the memory, it is possible to detect a modification in the code segment of `icacls.exe`, one of the processes being created by the Ryuk process.

The families that the technique cannot detect also present similar behaviors according to the tools. However, I cannot suspend the processes of the undetected families, at least at the appropriate time.

Since the observed behaviors of the undetected families are similar to the behaviors of the families with timing issues, I presume the lack of detection is related to the timing issue. I consider that these families would be detected if the injected process is not killed, and it is possible to access the memory of this process.

Furthermore, I consider the true positives of 64 percent a good starting point for the first iteration of the technique, especially if it is possible to maintain this percentage with a larger test set.

It is worth noting that binary signing is a technique for publishers to validate the code inside the executable, so antivirus sometimes do not scan signed binaries [56]. However, the `Mem2Disk` technique detects that the BADNEWS family is using a signed binary to start the attack and then replacing its own text segment by other code, that I presume it is malicious.

The 19 percentage of false positives seems high. In the view of the fact that it is more than half of the non-malicious programs tested, I consider important to further understand and improve this problem.

For the false positives cases of the `ExtraX` detection, I presume that one of the reasons is the additional executable sections present in memory are related to binaries loaded on-demand. Especially for browsers, sometimes it is necessary to run code after the binary has been loaded into memory. One example of it, it is running WebAssembly in JavaScript, as it is explained in Mozilla documentation [55].

As for the modified text segment detected by `Mem2Disk`, I believe it is related to some mechanism that changes an offset while the executable file is loaded into memory.

One of the programs detected is VLC because there are differences between the text segment in RAM and the code section in disk. However, they are never longer than one contiguous byte and, as displayed in Table 5.4, always one value in disk is replaced by the same byte in memory.

One modification that my supervisors and I consider possible is Address Space Layout Randomization (ASLR). ASLR is a technique that randomizes memory addresses of processes in order to make harder for attacks to be successful [54]. In practice, the entire address does not change but the memory page changes, so only a part of the address is actually modified.

With this in mind, it is possible that the modification is only one byte compared to the addresses of the binary stored on disk.

Memory content	Disk content	Difference	Times occurred
0x8D	0x40	0x4D	81
0x8E	0x41	0x4D	4
0x8F	0x42	0x4D	224
0x90	0x43	0x4D	67
0x91	0x44	0x4D	38
0x92	0x45	0x4D	57
0x93	0x46	0x4D	49
0x94	0x47	0x4D	127
0x95	0x48	0x4D	85
0x96	0x49	0x4D	145
0x97	0x4A	0x4D	109
0x98	0x4B	0x4D	1462
0x99	0x4C	0x4D	4980
0x9A	0x4D	0x4D	598

Table 5.4: `vlc.exe` content modification.

A similar situation to VLC happens in the Firefox process. As Table 5.5 shows, only blocks of one contiguous bytes are modified, and the replacements are the same each time it happens.

Furthermore, in the `firefox.exe` process every byte that changes there are 109 bytes where the content in memory is equal to the content in disk, so the entire difference is less than a 1 percent. Fig 5.1 shows the distribution of the changed bytes. It is also worth noting that the difference between the bytes are the same among all the replacements in each process.

I injected a 1024 bytes code into the code segment of `firefox.exe` using a debugger. The bitmap of the injected code segment is shown in Fig 5.2. This figure shows how a long block of contiguous bytes in memory are different and not only one byte every 100 of equal bytes, as it looks the bitmap of the non-malicious software in Fig 5.1.

Memory content	Disk content	Difference	Times occurred
0xBF	0x40	0x7F	27
0xC0	0x41	0x7F	3
0xC1	0x42	0x7F	19
0xC2	0x43	0x7F	9
0xC3	0x44	0x7F	4358
0xC4	0x45	0x7F	548

Table 5.5: `firefox.exe` content modification.

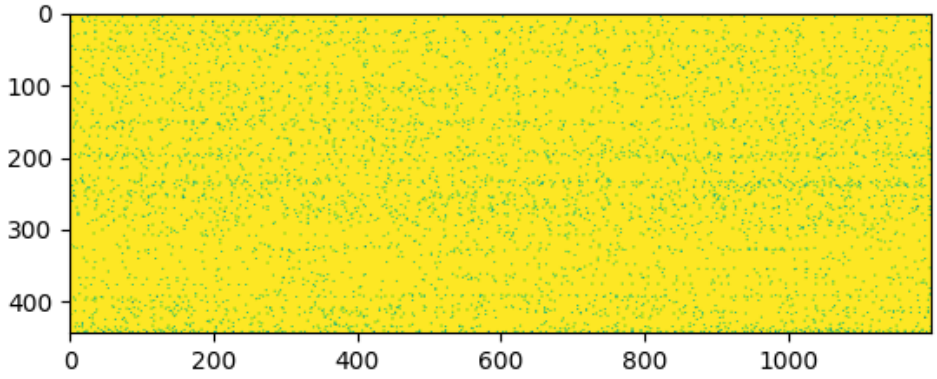

Figure 5.1: `firefox.exe` code segment bitmap.
Every bit in the bitmap is one byte of RAM. Yellow dots mean the byte value in memory is the same as the value in disk, and blues mean the contents are different.

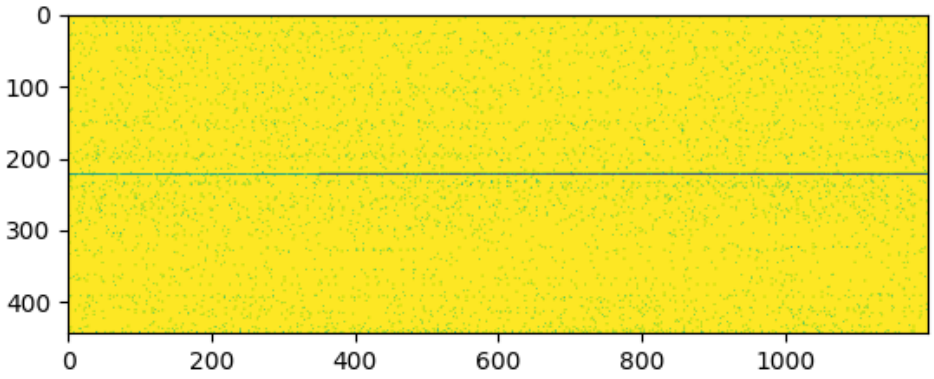

Figure 5.2: Injected malware code segment bitmap.
Every bit in the bitmap is one byte of RAM. Yellow dots mean the byte value in memory is the same as the value in disk, and blues mean the contents are different.

5.3.3 Limitations

One of the biggest issue that I have for detecting the memory injections is timing. Running the Velociraptor artifacts takes time due to the need to access many data structures stored in memory and, especially, it has to read the content from disk.

Considering that the RAM of the computer is continuously changing its state, one of the biggest challenges for this technique is to get the memory with the injected code still present. Sometimes this code is executed in a short period of time and then the process is killed, which clears the virtual memory of the process leaving no trace of the injection. If the injected code is no longer present then it is not possible to detect the malware with our approach.

Out of the 35 detected samples, timing issues are found in 8 of them, those are: ISMAgent, donut, GuLoader, Pandora, TsCookie, Ryuk and WarzoneRAT. On top of that, from the 3 families the techniques which cannot detected, Process Monitor and drstrace indicate they have similar behavior to the other samples but the execution is faster.

Although this is clearly a disadvantage of the technique, it is an issue inherent in forensics as an area: if the detective cannot arrive on time then it is likely that the evidence will not be there. More specifically to memory forensics, there will be gaps in detection due to the constantly changing state of the memory.

As mentioned before, another limitation is the size of the malware families set. I consider that 43 families it is not enough for a final decision on

the technique. I believe it is necessary to increase this number, and I think the automation of the testing phase as an option to increase this number.

Chapter 6

Conclusion

In summary, I present a technique to detect memory based malware, focusing on process hollowing and process injections attacks. To achieve these goals, I compare the content between RAM and disk in order to check that all the executable segments in memory have not been manipulated. Also, I look for additional executable segments that are allocated after the program is initially loaded into memory.

Results were promising with a detection rate of 77.78 percent and a sensitivity rate of 92.11 percent when excluding malware samples that do not run.

For future work further research in the false positives rate is necessary so it can be reduced, and the timing issue should be mitigated. Also, it is necessary to increase the amount of malware families tested to generate a more sound results table.

Bibliography

[1] Published by Statista Research Department, & 3, A. (2022, August 3). Cyber crime: Reported damage to the IC3 2021. Statista. Retrieved November 30, 2022, from https://www.statista.com/statistics/267132/total-damage-caused-by-by-cyber-crime-in-the-us/ Accessed 23 Nov. 2022.

[2] Eddy, Melissa, and Nicole Perlroth. "Cyber Attack Suspected in German Woman's Death." The New York Times, The New York Times, 18 Sept. 2020, https://www.nytimes.com/2020/09/18/world/europe/cyber-attack-germany-ransomeware-death.html. Accessed 04 Nov. 2022.

[3] "Cost of a Data Breach 2022." IBM, https://www.ibm.com/reports/data-breach. Accessed 28 Nov. 2022.

[4] Baram, Gil. "Analysis — How the Cyberwar between Iran and Israel Has Intensified." The Washington Post, WP Company, 25 July 2022, https://www.washingtonpost.com/politics/2022/07/25/iran-israel-cyber-war/. Accessed 28 Nov. 2022.

[5] Baram, Gil. "Analysis — How the Cyberwar between Iran and Israel Has Intensified." The Washington Post, WP Company, 25 July 2022, https://www.washingtonpost.com/politics/2022/07/25/iran-israel-cyber-war/. Accessed 28 Nov. 2022.

[6] Zhioua, Sami. "The Middle East under Malware Attack Dissecting Cyber Weapons." 2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops (2013): 11-16.

[7] Masdari, Mohammad and Hemn Khezri. "A survey and taxonomy of the fuzzy signature-based Intrusion Detection Systems." Appl. Soft Comput. 92 (2020): 106301.

[8] WatchGuard Technologies, Inc. "New Research: Fileless Malware Attacks Surge by 900% and Cryptominers Make a Comeback, While Ransomware Attacks Decline." GlobeNewswire News Room, WatchGuard Technologies, Inc, 30 Mar. 2021, https://www.globenewswire.com/news-release/2021/03/30/2201173/0/en/New-Research-Fileless-Malware-Attacks-Surge-by-900-and-Cryptominers-Make-a-Comeback-While-Ransomware-Attacks-Decline.html. Accessed 28 Nov. 2022.

[9] "Only in Memory: Fileless Malware – an Elusive TTP." CIS, https://www.cisecurity.org/insights/blog/only-in-memory-fileless-malware-an-elusive-ttp. Accessed 29 Nov. 2022.

[10] Kara, Ilker. "Fileless Malware Threats: Recent Advances, Analysis Approach Through Memory Forensics and Research Challenges." SSRN Electronic Journal (2022)

[11] Gartner Inc. "Named: Endpoint Threat Detection and Response." Anton Chuvakin, 18 June 2015, https://blogs.gartner.com/anton-chuvakin/2013/07/26/named-endpoint-threat-detection-response/.

[12] "Forensics." Cambridge Dictionary, https://dictionary.cambridge.org/dictionary/english/forensics.

[13] "For308.1: Introduction to Digital Investigation." Digital Forensics Essentials Course — SANS FOR308, http://www.sans.org/cyber-security-courses/digital-forensics-essentials.

[14] Case, Andrew and Golden G. Richard. "Memory forensics: The path forward." Digit. Investig. 20 (2017): 23-33.

[15] Sun, Yixin et al. "Detecting Malware Injection with Program-DNS Behavior." 2020 IEEE European Symposium on Security and Privacy (EuroS&P) (2020): 552-568.

[16] Tien, Chin-Wei et al. "Memory forensics using virtual machine introspection for Malware analysis." 2017 IEEE Conference on Dependable and Secure Computing (2017): 518-519.

[17] Security, Microsoft. "Detecting Stealthier Cross-Process Injection Techniques with Windows Defender ATP: Process Hollowing and Atom Bombing." Microsoft Security Blog, 22 July 2019, https://www.microsoft.com/en-us/security/blog/2017/07/12/detecting-stealthier-cross-process-

injection-techniques-with-windows-defender-atp-process-hollowing-and-atom-bombing/.

[18] Garfinkel, Simson L.. "Digital forensics research: The next 10 years." Digit. Investig. 7 (2010): S64-S73.

[19] Nance, Kara L. et al. "Digital Forensics: Defining a Research Agenda." 2009 42nd Hawaii International Conference on System Sciences (2009): 1-6.

[20] Frank Adelstein. 2006. Live forensics: diagnosing your system without killing it first. Commun. ACM 49, 2 (February 2006), 63–66. https://doi.org/10.1145/1113034.1113070

[21] C. P. Grobler, C. P. Louwrens and S. H. von Solms, "A Multi-component View of Digital Forensics," 2010 International Conference on Availability, Reliability and Security, 2010, pp. 647-652, doi: 10.1109/ARES.2010.61.

[22] Karl-Bridge-Microsoft. "PE Format - win32 Apps." Win32 Apps — Microsoft Learn, https://learn.microsoft.com/en-us/windows/win32/debug/pe-format.

[23] 0xRick. "A Dive into the PE File Format - Introduction." 0xRick's Blog, 22 Oct. 2021, https://0xrick.github.io/win-internals/pe1/.

[24] Authors GReAT, et al. "The Devil's in the Rich Header." Securelist English Global Securelistcom, 13 May 2021, https://securelist.com/the-devils-in-the-rich-header/84348/. Accessed 17 Sep. 2022.

[25] "The Difference between Malware and a Virus: CrowdStrike." Crowdstrike.com, 15 Aug. 2022, https://www.crowdstrike.com/cybersecurity-101/malware/malware-vs-virus/.

[26] "What Is Malware and How Cybercriminals Use It." McAfee, https://www.mcafee.com/en-us/antivirus/malware.html. Accessed 27 Nov. 2022.

[27] "Mitre ATT&CK®." MITRE ATT&CK®, https://attack.mitre.org/.

[28] "Process Injection." Process Injection, Technique T1055 - Enterprise — MITRE ATT&CK®, https://attack.mitre.org/techniques/T1055/.

[29] "Process Injection: Process Hollowing." Process Injection: Process Hollowing, Sub-Technique T1055.012 - Enterprise — MITRE ATT&CK®, https://attack.mitre.org/techniques/T1055/012/.

[30] Markruss. "Process Explorer - Sysinternals." Process Explorer - Sysinternals — Microsoft Learn, https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer.

[31] "Desktop Operating System Market Share Worldwide." StatCounter Global Stats, https://gs.statcounter.com/os-market-share/desktop/worldwide.

[32] Virustotal, https://www.virustotal.com/gui/stats. Accessed 07 Nov. 2022.

[33] "Linux Malware on a Rise Reaching All-Time High in H1 2022 - Atlas VPN." AtlasVPN, https://atlasvpn.com/blog/linux-malware-on-a-rise-reaching-all-time-high-in-h1-2022.

[34] Dolan-Gavitt, Brendan. "The VAD tree: A process-eye view of physical memory." Digit. Investig. 4 (2007): 62-64.

[35] Hash, Imp. "Windows Process Internals: A Few Concepts to Know before Jumping on Memory Forensics [Part 4] -..." Medium, Medium, 4 Sept. 2020, https://imphash.medium.com/windows-process-internals-a-few-concepts-to-know-before-jumping-on-memory-forensics-part-4-16c47b89e826.

[36] Hassan, Wajih Ul et al. "Tactical Provenance Analysis for Endpoint Detection and Response Systems." 2020 IEEE Symposium on Security and Privacy (SP) (2020): 1172-1189.

[37] Karantzas, G.; Patsakis, C. An Empirical Assessment of Endpoint Detection and Response Systems against Advanced Persistent Threats Attack Vectors. J. Cybersecur. Priv. 2021, 1, 387-421.

[38] Sudhakar and Sushil Kumar. "An emerging threat Fileless malware: a survey and research challenges." Cybersecurity 3 (2020): 1-12.

[39] Arpaci-Dusseau, Remzi H.. "Operating Systems: Three Easy Pieces." login Usenix Mag. 42 (2017).

[40] System Call Tracer for Windows, https://drmemory.org/page_drstrace.html. Accessed 14 Oct. 2022.

[41] Barrygolden. "_file_object (WDM.H) - Windows Drivers." _FILE_OBJECT (Wdm.h) - Windows Drivers — Microsoft Learn, https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_file_object.

[42] Tedhudek. "Windows Kernel Opaque Structures - Windows Drivers." Windows Drivers — Microsoft Learn, https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/eprocess. Accessed 19 Jul. 2022.

[43] Block, Frank and Andreas Dewald. "Windows Memory Forensics: Detecting (Un)Intentionally Hidden Injected Code by Examining Page Table Entries." Digit. Investig. 29-Supplement (2019): S3-S12.

[44] monnappa22, "Monnappa22/Hollowfind." GitHub, https://github.com/monnappa22/HollowFind. Accessed 13 Oct. 2022.

[45] MtlOop, et al. "Detecting Deceptive Process Hollowing Techniques Using HollowFind Volatility Plugin." Cysinfo, 24 Sept. 2016, https://cysinfo.com/detecting-deceptive-hollowing-techniques/.

[46] Volatilityfoundation. "Home · Volatilityfoundation/Volatility Wiki." GitHub, https://github.com/volatilityfoundation/volatility/wiki/Command-ReferenceMal#malfind. Accessed 05 Aug. 2022.

[47] Srivastava, Anurag and James H. Jones. "Detecting code injection by cross-validating stack and VAD information in windows physical memory." 2017 IEEE Conference on Open Systems (ICOS) (2017): 83-89.

[48] Thing, Vrizlynn L. L. et al. "Live memory forensics of mobile phones." Digit. Investig. 7 (2010): S74-S82.

[49] Cohen, Mike. "The Velociraptor Query Language Pt 1." Medium, Velociraptor IR, 14 June 2020, https://velociraptor.velocidex.com/the-velociraptor-query-language-pt-1-d721bff100bf.

[50] "Basic VQL." Velociraptor, https://docs.velociraptor.app/vql_reference/basic/. Accessed 29 Jul. 2022.

[51] https://github.com/lautarolecumberry/DetectingFilelessMalware. Accessed 10 Dec. 2022.

[52] Yan, Tao. “New Wine in Old Bottle: New Azorult Variant Found in Findmyname Campaign Using Fallout Exploit Kit.” Unit 42, 11 Dec. 2018, https://unit42.paloaltonetworks.com/unit42-new-wine-old-bottle-new-azorult-variant-found-findmyname-campaign-using-fallout-exploit-kit/.

[53] Ceponis, Dainius and Nikolaj Goranin. “Investigation of Dual-Flow Deep Learning Models LSTM-FCN and GRU-FCN Efficiency against Single-Flow CNN Models for the Host-Based Intrusion and Malware Detection Task on Univariate Times Series Data.” Applied Sciences 10 (2020): 2373.

[54] Marco-Gisbert, Héctor and Ismael Ripoll Ripoll. “Address Space Layout Randomization Next Generation.” Applied Sciences (2019).

[55] ”Loading and running WebAssembly code”. https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running. Accessed 12 Dec. 2022.

[56] Kim, Doowon et al. “Certified Malware: Measuring Breaches of Trust in the Windows Code-Signing PKI.” Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (2017).

Appendix A

Appendix

A.1 Malware samples complete SHA-256 hash

Name	Complete SHA256 hash
AgentTesla	eeedc8ecc3623353fedfcedf3f5402ad5c4ea2d7cff37be3a730415df2e8a68c
AgentTesla	8118d7c7ac15618f5517b0fec626096796084a2b44bba85a65ee7d2e7b4f1fc9
AssemblyInjection	de4cd0c58f2b44050feea01c56fe1f63b85eb178a15b2f087d963bbef113c644
Astaroth	972cae6f1dbd11ea90e931b27e78d640f0621abc2a5431269d39b2287653cc6c
Astaroth	ce2928ab5086fe869d7b16ffb0e01519fd7a735253e40e40983ad03f33f80600
Azorult	08a6193d0afc12de32573390251740b4b1d7a1af0b19ef0cc3a12c078db76449
Azorult	5ebb3cc4e09a0fb9434d07543cd821538008462dc037c6d6323a32b8bd26dd6e
BADNEWS	d07adf2032c1274ef810aa9146e0407dbe76335b2121c6f98667f67a90f63ce3
bandook	4bf9325fe8d721e60c2a5beee8dbdf275ab9c5de309e162ecc81d1cdf7369cef
bazar	300c0dab0af5de260c5e0a30ff799fd26758b39bb933870674ce632be22841a5
bazar	534d60392e0202b24d3fdaf992f299ef1af1fb5efef0096dd835fe5c4e30b0fa
Donut	bada6d6d493416c0992a375de60fe574ced09bef5496ebfac07c19a8b2785494
dtrack	bf8e3f0430a2a53608432cca208ac7d932e84a557defcfcdcb468b68cfacd7f8
Dyre	1f8ba528cef45357d5c7376e510077aeb9c580af0378b3b80f7d4f94dc531b2f
Dyre	0350d2ab1cb791672d7b3927c57bcdfe71fa4d2e3609201dd8c2f288ac341f4c
Empire	4a23326def54ca250c558925ef891ad92ecd9a1a6870ea85760f8b97fe28613f
formbook	d2f58b08f8abfe5055f3c1f0b8d991dfe1deb62807a5336b134ce2fb36d87284
formbook	2c7540c6d066510b73a1a5c668dc74ec6d0d3f0716bb3adb6cd83afdd07f35ff
Gazer	c5db84fe0f762ebc2abe484d59d51fdf35a37f3a32e6f44d8197b1e8cda98e84
Gazer	ca9e3ea2e21483612ec2d9ff4a91693e97ab24175ac00ccb52da89e4b89230c9
Gh0stRAT	10ed8da5de4785261aeb9a2c113b8d82316b2b0e7946ebb4203b3bf4c3b355e0
Gh0stRAT	5a9f06e3346cb716c79bdfaf347944539fccaeb2e503b6b5d434e47c458c6618
GuLoader	3ae4d65b8e2c2ac4866500331532095749c24778b2ba55a7cf75b7615676d299
GuLoader	cdbe7a2fc091f0b3b4ed8e35d600df24deadc377d7323d92aebf44ed38f41413
HopLight	032ccd6ae0a6e49ac93b7bd10c7d249f853fff3f5771a1fe3797f733f09db5a0
HopLight	0237b186086fa4d13e8c854dcf2d0f8a19fcbe62a58a415e9a5a933f1154e7d8
HTran	15b529d02b4d9effdc660b7546b4aab6f266af87bfca48a5a97ae7b46a725e64

HTran	1b32e6800b3a80e74f135b75925f3c1e081662adfac53262ec9a8a830398ff64
HyperBro	6e32c33c82efaf05822a0d5c610adbc2c1e8fd4d99955b1050496ad29ec927de
HyperBro	07f87f7b3313acd772f77d35d11fc12d3eb7ca1a2cd7e5cef810f9fb657694a0
InjectionPoC	e75b681cd8f4a69663dfcf0cc5337bc00dd8595c45f884e7e91136d764e27886
InvisiMole	2debef67c4e8e9a28af920688b23e858876be623573f2cf23edcd50856388622
InvisiMole	d0062f473c4350dc934752dc4f876c962eeb1c43968647695460f4c8ed629a46
ISMAgent	33c187cfd9e3b68c3089c27ac64a519ccc951ccb3c74d75179c520f54f11f647
ISMAgent	74f61b6ff0eb58d76f4cacfb1504cb6b72684d0d0980d42cba364c6ef28223a8
Kimsuky	7d89a16fc0d3afa3cd78cc51e7ae6a81343cb14de6fdca9325142deca5133515
Kimsuky	b207265be3bdb32536b3118e0d94660241988e2f862a0ccaf968d25d86b87a04
lokibot	97301bb711f9921f5e24aa2e249a4e76cc3ffa359f73153d77712bcc11a20f15
lokibot	afe2844c27424c053cc0e578dae9d5d1bea6cab5ab0227edbaf983289961452e
netwire	ea32c3c39c8c3f83e95916262d37b5aa49c920a9356dcebfa639b8391413ae9a
Pandora	1f172321dfc7445019313cbed4d5f3718a6c0638f2f310918665754a9e117733
Pandora	5b56c5d86347e164c6e571c86dbf5b1535eae6b979fede6ed66b01e79ea33b7b
PlatinumGroup	021bb772775dc4c7df1569c3ee8ed957207df810837bdff71104ea6e905e4681
PlatinumGroup	46a9ac069c20c505e6bc5fcd6de9a0f1d3a8ed3073133913e57d54604a0e8e8e
poshc2	3c4c4cb0e9a48e8203ebe67da38dcfdc0d888213424ddd335a767f6a04e798ff
poshc2	8ba619e1fb38bc0232347892b8fa0f0a3350be8d7397179de74549c07d684bab
qakbot	3be905066595dc785c9b6b98bfb2d9e0478f32df31337a8aeec96d7ccd52769e
qakbot	6f00837f83703021bc4f718a4df8a7fbdadf5fff50728dc09c050efa5259db89
remcos	03e298154f6a21a7f3e98d06193f5f3c902325887428ab1d7c9390685b239d02
remcos	944ec3ee0ba63f9535753135f92b2147efab49a3116c9d428335fcff92ba24ab
REvil	0c10cf1b1640c9c845080f460ee69392bfaac981a4407b607e8e30d2ddf903e8
REvil	00d015edbfb34e16b5b4086d25174ae435ca86d8cd267e0ed9b32db7d1d8ae2f
RokRAT	af619936fa29b7d0cf0c8441674bbf062cea427f9aaad4ea3173b5942956720b
RokRAT	9b383ebc1c592d5556fec9d513223d4f99a5061591671db560faf742dd68493f
Ryuk	5e2c9d80fa4528fe9777738a9cba9ede08cdae353fd4cb2d9caf0c9801fd5711
shadowpad	aef610b66b9efd1fa916a38f8ffea8b988c20c5deebf4db83b6be63f7ada2cc0
sliver	5adc6b62d26ad39c99407b3dfe2869f89a14d174ada9a732f3e1ef0c851c036f
sliver	38895ca4da6111265ad5d5f995d306085ccfcff13fcb2175d4596307a42135b1
SlothfulMedia	320cf030b3d28fcddcf0a3ef541dea15599d516cb6edaad53ec9be6b708d15c2
SlothfulMedia	83131292833103948d70b354b95905e484c34c9992cecd00fe9ab5eb1bbc7987
smokeloader	041a05dd902a55029449bf412cedbe59a593f8d4e67d4ae37cf7a9289e2f22ca
synack	68b87153d663663bd8e2e6644eeff9f5291167a2dca1a807e4918976e739ba95
trickbot	b7cbc5e5dc182c8d99809cd64d36734abeb6bfac15e6efc2ebcc2c57254bf172
trickbot	9da8a5a0b5957db6112e927b607a8fd062b870f2132c4ae3442eb63235f789e1
TsCookie	3cad20318f36b020cf4d6b44320eb5a6dae0a78339a0fdc3a1fe5e280a8507f1
TsCookie	80ffaea12a5ffb502d6ce110e251024e7ac517025bf95daa49e6ea6ddd0c7d5b
Turla	44d6d67b5328a4d73f72d8a0f9d39fe4bb6539609f90f169483936a8b3b88316
Turla	95d1f4407f3c725be2100cb72bb30e4fba08960ea83b51b5ead8e210eea51ec4
ursnif	104e6094ef239aae7e4317433e868b67108b8157627dc222f996cb087795334f
ursnif	e3fb27a6761d3a9403ff5b3ddbc86e5231664980149c8fd85bcfb319cc1ebb8c
WarzoneRAT	6c34c6663c544d9d1d255733c2f8d0bb090730467d0cf19b10e0b6abcbbd6fb4
WarzoneRAT	8590ebe9a64020b717f0fda3bc22a35bc6f4f488e63d9a5f8deadd67aa89921e
WhisperGate	b50fb20396458aec55216cc9f5212162b3459bc769a38e050d4d8c22649888ae
WhisperGate	dcbbae5a1c61dbbbb7dcd6dc5dd1eb1169f5329958d38b58c3fd9384081c9b78

Table A.1: Malware samples complete SHA-256 hash.

A.2 Non-malicious software complete SHA-256 hash

Name	Complete SHA256 hash
Adobe Acrobat Reader	966cfa9539314f7ff8bd0d403217708ffa17402959e9ef831d11ef7edd502fba
Command Prompt	935c1861df1f4018d698e8b65abfa02d7e9037d8f68ca3c2065b6ca165d44ad2
Discord	a3f9b57ec492d1ca666943899da3c6d01d22dbbc3a5cedb2e575cba8912f4a16
Google Chrome	af0bd408548165770baf8e5b455d7ecaa6de127c8d696c063806d07df7c3a6d5
LibreOffice Writer	7e1ef3b95fe273bd4c16aab6cf485221a7099e8d881a58258156bc18554ebe69
LibreOffice Calc	961ef417d04570f25a72509b52c72c0bcc8b6bb31e304072d5749f0d380f3ac9
Microsoft Edge	eb8da1b8b138179cfa9c97ebbd64af67e22a6ab1673a900c7aaadeb72944439e
Microsoft Paint	061d41c4239a0dbe2d9578d4d10a5db2a8f21c1c0f9b63aae490f1cdd683340a
Microsoft Teams	8a94b091015de99b0449e2df666563a03e35462843e80b709ffa1fb0d2e96a75
Mozilla Firefox	ead605af5446603cbcfa890e7f44cc380bdb69794487ce3ed68103e486b1ed28
Spotify	3b84962c10248d8128d52a3531432e786b236d5f4fcc2b24df0127eb41741d67
VLC	1a403269242218a67e401c7e321bf466ef6b381bc7cb8a56ea77d504f7f81a44
Windows Calculator	d7b378a4bc4deae748462d216d14a20ccb1bac1d3ffbc67711db2cc1d8b182b7
WordPad	0092cd4a00fb6095663dbbd50b8846cdb03f954274a7bf6a84d1face3db4eaf4
Zoom	f61f45486c1a3ea5330a87522e6fa139c3ebf33addd2e3bd3dfd9b2881b06917

Table A.2: Non-malicious software complete SHA-256 hash.

A.3 Acronyms

ASLR Address Space Layout Randomization

COFF Common Object File Format

CPU Central Processing Unit

CSIRT Computer Security Incident Response Team

DOS Disk Operating System

DLL Dynamic-Link Library

EDR Endpoint Detection and Response

ELF Executable and Linkable Format

HTML HyperText Markup Language

IC3 Internet Crime Complaint Center

NAS Network Attached Storage

PE Portable Executable

PEB Process Environment Block

PID Process Identifier

RAM Random Access Memory

SANS SysAdmin, Audit, Network and Security

SQL Structured Query Language

TB Terabyte

VAD Virtual Address Descriptor

VQL Velociraptor Query Language