Antivirus Evasion Techniques 1 (2021) 1-48 Submitted 2/21; Published 2/21

Antivirus Evasion Techniques

Chen Gleichger GLEICHGER@GMAIL.COM
Roman Gotsdiner ROMANXTREME@GMAIL.COM
Yarin Zeevi YARINZEEVI2803@GMAIL.COM
Noam Moshe NOAMISWAYCOOLER@GMAIL.COM
Chen Shem Tov CHENSHEMTOV2@GMAIL.COM
Sharon Nachshony SHARON.NACHSHONY@GMAIL.COM

Guidance: Dr. Nezer Jacob Zaidenberg NEZERZA@COLMAN.AC.IL
Department of Computer Science
The Collage of Management
Rishon Le-Tzion, Israel

Abstract

This paper describes a final project in bachelor of science in computer science, with cyber security speciality. The project lasts for 4 months, and includes hard work, dozens of hours and many COVID-19 closures, accompanied and guided by Dr. Nezer Zaindenberg, head of cyber security specilization. In this article, we will define, detail, explain and demonstrate a variety of methods to evade antivirus engines.

Keywords: Malware, Antivirus, Evasion, Signatures, Heuristics, Hashes, Behaviors

1. Introduction

In these days there is an ongoing race between attackers and defenders in the cyber security filed. As part of this race the attacks become more sophisticated and as a result the security tools are improving as well to deal with those attacks.
One of the most common ways to deal with malware is anti-virus, which in these days exists on almost every workstation.

The main aim of the attackers is to evade detection by anti-viruses and to achieve this goal they use anti-virus evasion techniques. In our project, we conducted a research about five different techniques to avoid detection by anti-viruses:

1. Code Injection
2. Shellcode Injection
3. Source Code Injection
4. Process Hollowing
5. Code Normalisation and Obsufication

©2021 Chen Gleichger, Roman Gotsdiner, Yarin Zeevi, Chen Shem Tov, Noam Moshe and Sharon Nachshony.

Furthermore, we tried to find ways to improve the source code of each module and make it harder to detect by anti-virus engines.
We used VirusTotal to tests our code against known anti-virus engines.

Before we present the different types of code injections that can be used to evade antivirus engines, we need to define several basic terms that will be used by us during the presentation of the various antivirus evasion techniques:

- **Antivirus:** Software used to prevent, detect, and remove malware, including computer viruses, worms, and Trojan horses. This software may also prevent and remove adware, spyware, and other forms of malware.
- **Signature-based AV:** This type of antivirus engine is produced based on several signatures (patterns created to identify a particular sequence of bytes) created to identify what type of code could have contained the next activation. If the code or template of bytes is in the blacklisted content, then AV alert is enabled.
- **Heuristics-based AV:** A modern technique for detecting malicious charge by defining some rules and algorithms so that if the code contains a specific set of code / instructions or operating segments that perform certain actions that may prove malicious, then an AV alert is enabled.
- **Behavior-based AV and dynamic scans:** This technique analyzes the behavior of the binary and classifies it as malicious and non-malicious. Most antiviruses today are based on a dynamic approach, that is, the potential code file as maliciously runs on a virtual environment for a short time while comparing its behavior against the list of signatures + heuristics. During the analysis, it will be possible to discover the essence of the run code even if it is an encrypted file without decoding and by examining its behavior and signatures only.

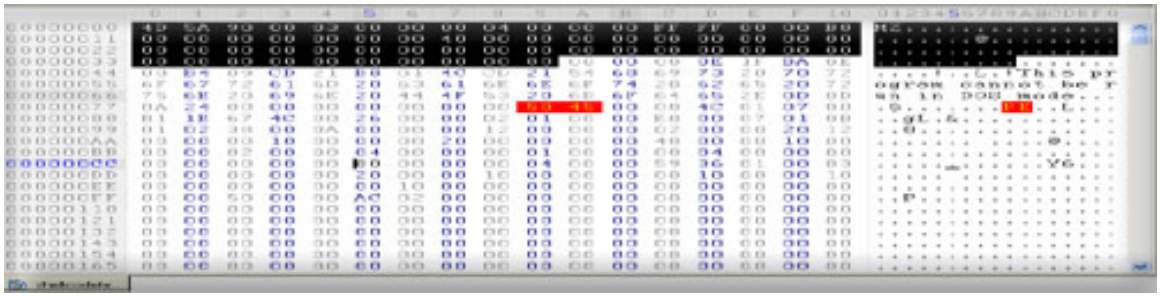
Antivirus plays a key role in system security. For Hackers / Pentesters, there are some major issues in the post-intrusion phase, usually the attacker will want to upload some tools to the victim's machine for better control, but here the antivirus "plays" with our tools and detects them as a malicious file and deletes them. It may be possible to evade identification if a tool created specifically for this attack and written exclusively is used, but this is a rare situation. In case the tools exposed to the public are used, it is more likely to be caught by antivirus.
A solution to the problem described above can be found in one of two main ways:

1. Hide code that may be identified as malicious. This is usually done through encryption
2. Encoding in such a way that will prevent the AV in the system from detecting the run code as a virus malicious file.

2. Evasion Techniques

2.1 Code Injection

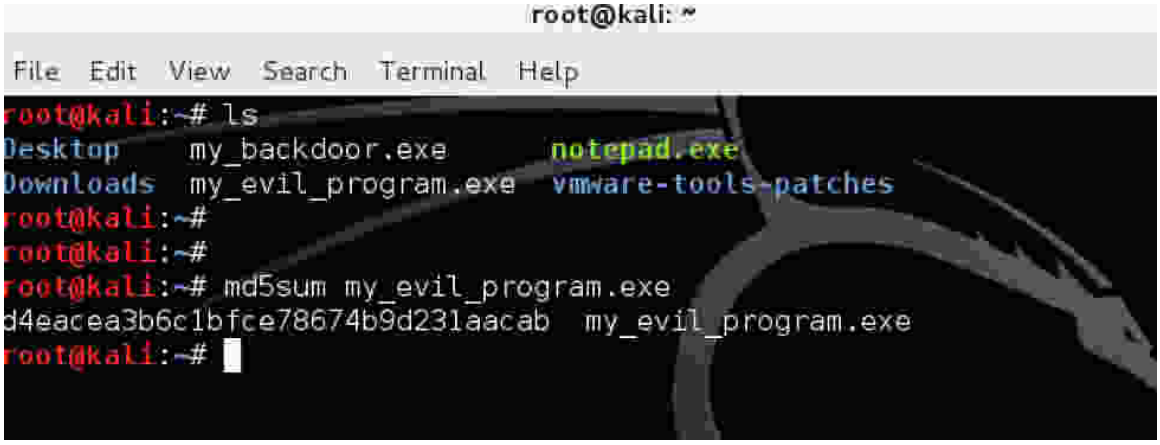
Code injection consists of running code in the memory of another process. Generally using DLL injection but there are other options and even an entire EXE can be injected. The complexity of the process lies in the fact that the injected code must find a way to execute itself without being loaded by the system (especially since the base address is not the same). For DLL injection this is done when the DLL is loaded. In order to inject code, the code should be able to change its memory pointers based on Reallocation.

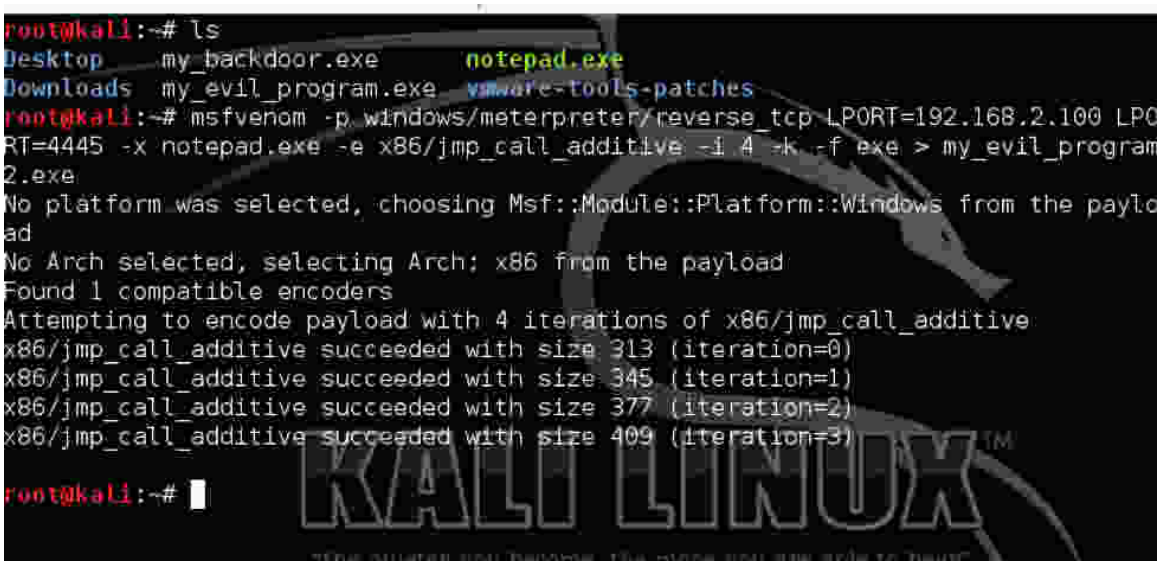


1. **Using Binders:**
   Binders are used to link two or more EXE files to a single EXE file.
   This way viruses or worms can be hidden using Binders and they can operate without detection of AV. The original signature of the malicious files is transferred for offsetting in the newly created binary and can easily evade any static antivirus products.
   Binder usually binds other EXE files to itself and creates a new binary. For example, the original size of the binder file is 20KB and the attached EXE size is 35KB so the final size of the newly created EXE will be 55KB.

2. **Using Packers:**
   Works very similar to how Binders work but the only difference between them in the case of Packers, the malicious binary is compressed before it gets embedded in the binary of the package to create the final EXE. This makes any antivirus product helpless in detecting the malicious binaries because changes have been made to its signature due to compression.
   Also, there are techniques that can be used to convert an executable file or any other file (like .pif or .scr) to a vbs file and in executing a vbs file the hidden binary will be executed automatically.

3. **Using Cryptors:**
   This mechanism cryptographically modifies the program / activation code and performs a decoding or sub-process function like stub. When the program is sent it is encrypted and the decryption plug is hidden. Make it seem completely useless for AV, thus bypassing it. Decrypting an encrypted code works in memory, and a decrypted activation code remains on the disk. Hence AV could not remove it before performing it.





To sum up , **Code injection** is the exploitation of a computer bug caused by illegal data processing. Injection is used to validate (or "inject") code into a vulnerable computer program and modify the execution. The result of a successful code injection can be disastrous, for example, by allowing computer worms to spread.
Code injection vulnerabilities occur when an application sends untrusted data to an interpreter. Injection "defects" can most often be found in SQL, LDAP, XPath, or NoSQL queries; Operating system commands; XML parsers, SMTP headers, program arguments, etc. Injection defects in the source code are easier to detect than in the scan. Injection may result in data loss or corruption, irresponsibility, or denial of access. Injection can sometimes lead to complete takeover of the machine.

2.2 Shellcode Injection

Shellcode Injection is a short code that is used as a Payload in exploiting memory-based weakness, for example: Buffer Overflow. This concept is called Shellcode because usually the code in question initiates a Command Shell through which the attacker can perform whatever he pleases in the victim position. Usually the code will be written in machine language. This code will be concise and will contain specific HEX commands called opcodes. In this chapter we will learn what the types of Shellcode Injection attacks are and how we can write dedicated code ourselves.

2.2.1 Types of Shellcode Injection

In general, a dedicated code is divided into two - a local dedicated code and a remote dedicated code. Details about each type:

1. **Local Shellcode**
   This code is used in a situation where the attacker has limited access to the victim station but can exploit this or that vulnerability with high permissions, for example as we have already seen "Buffer Overflow". This type of shell code allows an attacker to escalate privileges.

2. **Remote Shellcode**
   This code is used by the attacker when it aims to attack a vulnerable process which is running on another computer on the network from the computer it is holding, thus gaining access to it, and proceeding on the network. Usually remote dedicated code initiates TCP / IP Socket connections to gain access to the destination, although an attacker would prefer not to initiate a new connection (for fear of encountering a firewall that would block the communication), but would rather "ride" on existing communication, which is defined legitimate for the same destination station.
   This code will be categorized according to the quality of the connection it creates:

   - If the dedicated code initiates the communication, it will be called "Reverse Shell" or "Connect-Back Shellcode", since the shell connects back to the attacker's position.
   - If the attacker initiates the communication in front of the attacked position, the Shellcode will be called "Bindshell" because the Shellcode is bind to a specific port in the victim's position.
   - When a particular vulnerability initiates the connection to the vulnerable process at the victim position, it is a "Socket-Reuse Shellcode". This approach is more complex because the dedicated code is required to understand which connection is available to use, and the victim position may have several open connections.

3. **Download and Execute**
   This technique guides the attacked target destination to download software from a particular network / website, i.e. a particular threat, so that once activated, the Shellcode will be able to run. Another method is to download software libraries from the network. The advantage of this technique is that the code is relatively small, since

it does not require the shellcode to create new processes in the attacking position and there is no need to clear evidence.

4. **Staged**
   This method represents a situation where the dedicated code is too large to be injected into the same process that the attacker wishes to exploit in the target position. Therefore the attacker's approach would be to run the dedicated code in stages - in the first stage only a small part of the dedicated code will run, which will download a larger part of the dedicated code which will be injected into the process memory and run, and this is actually the second stage.

5. **Egg-Hunt**
   This type of Shellcode is a kind of multi-stage dedicated code. This approach is used by the attacker when he wants to inject a relatively large shellcode into the victim position process so that he does not know to determine where the run memory will end up. Therefore, a small dedicated code is injected into the desired process and run. This code will look for the address of the process in memory so that the larger code can run (the "Egg").

6. **Omelette**
   This shell code works similar to Egg-Hunt, but it injects a number of different code packets into the process and the "searching" code will know how to find them, logically connect them and run them together in the same utilized process.

2.2.2 How do we write Shellcode?

In writing the shellcode we will consider some important factors for running it.
The writing of the shellcode will be done in the language of the machine assembly since it is important to us that the shellcode be as limited as possible so that we do not encounter problems when utilizing memory-based weakness. Other programming languages are eventually translated into commands in the assembler, meaning that every command in the language will be translated and therefore we do not want to get into a situation where the code will be translated into more lines than necessary. This means we will strive to write the shellcode with a minimum of commands.
In addition to writing polymorphic code to evade recognizing the shellcode when running. The intention is to write different commands that will perform the same thing, so if AV has already signed a similar shellcode, we will be able to evade identification by it.
When writing the shellcode, one should be careful of characters in the code that may impair the process of running the shellcode. For example, the character NULL character may cause the shellcode to stop running.
At the end of writing the shellcode, we will make sure that it is as short as we could and if it is found that we will not reduce it as much as we can in order to try and ensure its run. After finishing writing the shellcode, we want to verify its effectiveness by analyzing it with the help of a disassembler.
In addition, we want to verify that the shellcode does indeed address the appropriate system calls and we will need to write code that will run it. System functions are used to escalate privileges, stop code and more.

2.3 Source Code Injection

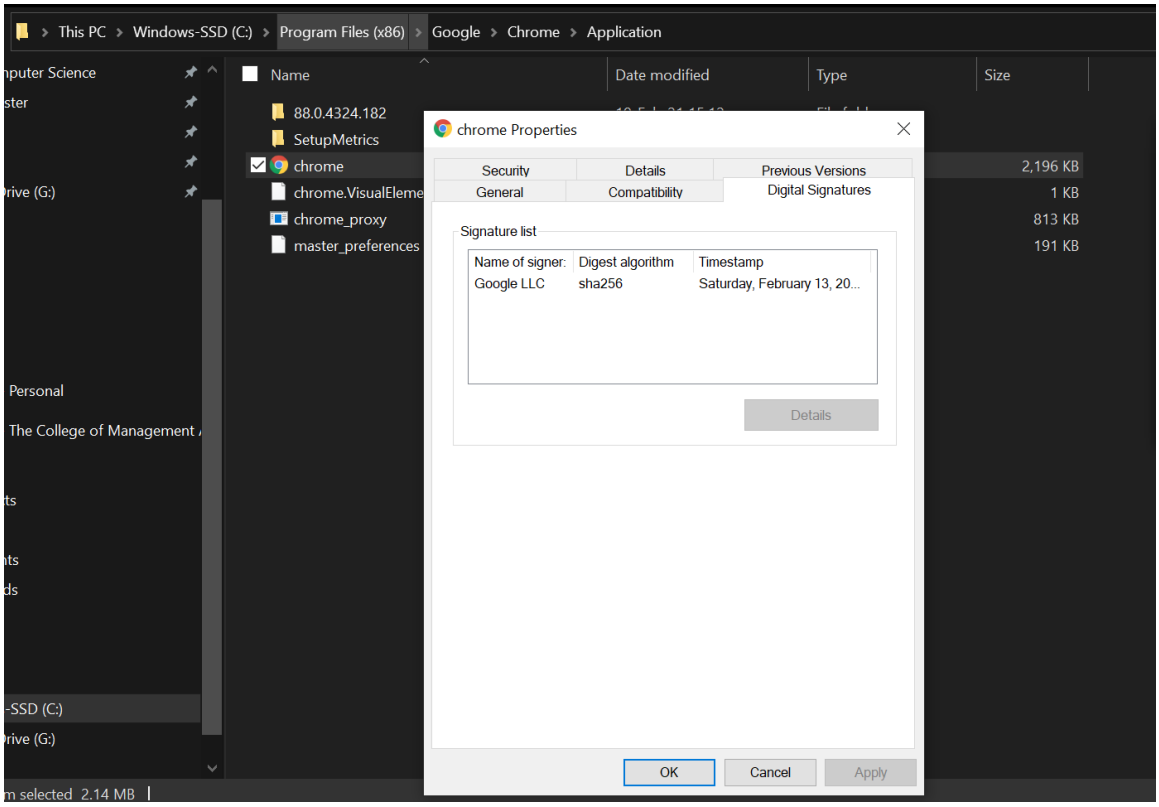
Source Code Injection is a technique for injecting malicious code, which has not been seen "In the Wild" more than a few times, when in the times that it was observed, it was powerful elements that carried out attacks against giant companies.
The idea behind Source Code Injection is to inject malicious code into part of an existing civil product code repository, during the product development phase. Since it is still in the development stage, the malicious code is bound under the legitimate code of the company and when it is compiled into a product, it will also include the malicious code, without any attention from the main developer.
The basis of the attack is that every product of such a huge company has thousands of lines of code, so that it can be assimilated without any attention from the development teams. Moreover, such a product is a complicated product, and therefore it performs a large number of operations, thus another operation will not attract any attention from the development teams and / or antivirus engines.
Because the malware is part of a known software, which usually belongs to a known giant company, the antivirus engines will exclude the software and its operations, because it is known, genuine and reliable.

Every software product produced by a huge company is signed, thus ensuring that the software is reliable software and has not been modified by a malicious entity, and therefore its actions are reliable and legitimate.

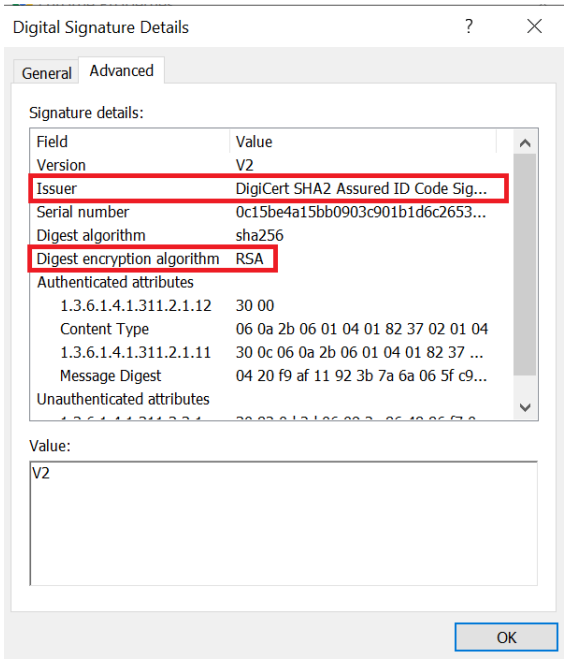
2.3.1 How Does it Work?

The way the product is signed is by the global certificate authority mechanism, where there are several specific companies that are allowed to provide signatures, and all computers rely on the signatures provided by those companies. The way to make sure that the company has indeed signed the product is through an asymmetric encryption mechanism - RSA protocol. Any such signature is signed by the company's private key and can be decrypted by the public key only. When the antivirus mechanism sees that the software is indeed signed by the global certificate authority mechanism, then it trusts the software and its operations Because it is defined as safe.

Example:

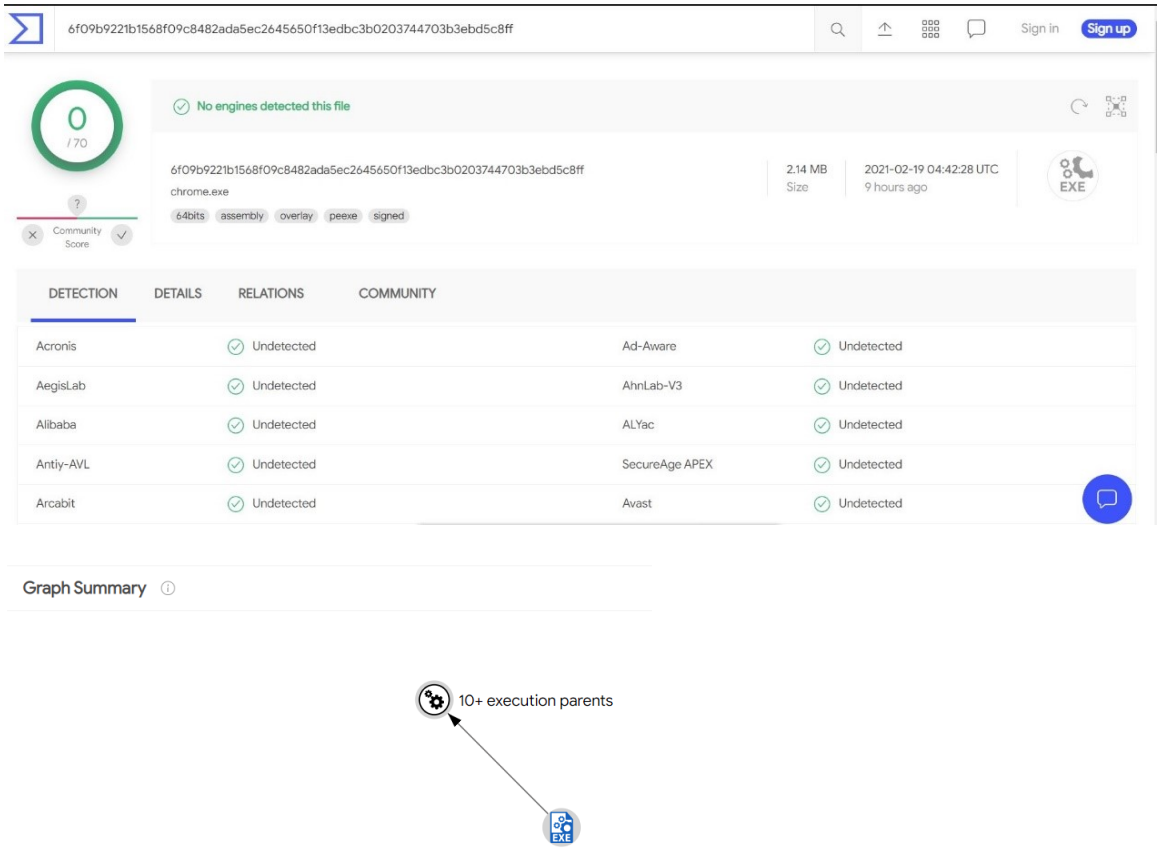


This example shows the signature of the Google Chrome software:



In this example it can be seen that the software is signed by a Certificate Authority called DigiCert and verified by the asymmetric encryption protocol RSA.
In addition, when the antivirus engine sees that the software is signed, it excludes it and claims that it is not malicious, even if it does things that will be identified with malware in terms of the nature of the actions, in the heuristic form.
Google Chrome, for example, uses a lot of code-injection methods for other Google Chrome processes, to enable proper management of the software.
If unsigned software behaved similarly, the antivirus engine would classify it as potentially malicious software. Moreover, even when a protective body encounters signed software, it usually classifies it as proper software and will not examine it in depth.

For example, if we upload the Google Chrome EXE file to VirusTotal.com, we will get a diagnosis of all antivirus engines on that file:



This behavior of running multiple processes in parallel on the host computer and their debug operation, is a common operation in several attacks, but is not identified here as malicious.

A famous example of this type of attack is the attack that occurred during 2020 on SolarWinds company, in which the APT system infiltrated the company with malicious code that was distributed as part of its software updates. By doing so, the attack system has gained remote access to thousands of computers, including companies and giant organizations. This attack has not been detected for a long time, as it was part of a regular software update. According to reports, behind this attack was a powerful Russian offensive line.

To sum up, **Source Code Injection** is an attack method for injecting malicious code, which is not well known and does not receive enough attention. With this method, it is possible to integrate malicious code within a legitimate code of a software product and thus evade the discovery of antivirus engines and / or product development teams, for a long time.

2.4 Process Hollowing

Process Hollowing is process injection sub-technique, where the attacker replaces the legitimate code the was mapped to the memory from a running process, with his malicious code, and by this way, the malicious code will run in the same permissions as the legitimate code that was created in the beginning.
Attacker will use process hollowing when he will want to disguise his malicious code, without the risk that he will damage the activity of the legitimate running process and make the user to suspect the something is not right on the computer.
Process Hollowing is used as AV Evasion technique, that the attacker will use, to avoid detection by AV engines. One of the most famous worms, which used Process Hollowing, was Stuxnet in 2010.

2.4.1 Implementation of the Attack:

In process Hollowing the malware creates new instance of legitimate process in suspended mode, for example svchost.exe.
After the creation of the process in suspended mode, the malware unmapped the legitimate code from the memory and instead the legitimate code, mapped the malicious code into the memory in the same place, and resume the process, so he will start running as svchost.exe.
The steps malware takes, to perform Process Hollowing (According to "Mastering Malware Analysis" - Stuxnet secret technique-process hollowing [163-165]):

1. First, the malware creates a legitimate process in suspended mode.
   By creating the process in suspended mode, the process is created, but his first thread is not running.
2. Hollowing out the legitimate process, by using VirtualFreeEx API for example.
3. Allocate the same space in memory as the unloaded PE image for the malware PE image.
4. Inject the malware executable into the freed memory space by loading the PE file and fixing its import table.
5. Change the thread starting point to the malware entry point.
6. Resume the suspended thread to execute the malware from the entry point.

Note:
There are many ways to implement Process Hollowing, and this is just an example of one way to implement this technique.

2.4.2 Process Internals

In order to understand the next parts it is important to be familiar with process internals and with two important terms: PEB and VAD.
Each windows process is represented by an executive process structure called EPROCESS.
EPROCESS contains attributes that are related to the process and points to other data

structures.
Most of the data structures that EPROCESS points to exists in the kernel-mode, except from one data structure – Process Environment Block (PEB).

PEB is user-mode data structure, and the reason why it is part of the user-mode is because it contains information that applications access to.
PEB contains the pointers to the process' DLL list, command line arguments, environment variables, heaps, current working directory, and standard handles.

The second term that it is important to understand is "Virtual Address Descriptor" (VAD).
The memory manager maintains a set of VADs for each process, in order to track, which address space has been reserved in the process address space and which has not.
VAD is kernel-mode structure.

2.4.3 **Detection of Process Hollowing**

In order to detect Process Hollowing on a computer, the most common way is by using Volatility plugins.
When a malware performed Process Hollowing and Hollows the suspended process Windows removes all the connection between the PE file and the hollowed process, but only from the EPROCESS kernel object and not in the PEB.
The comparison between the output about all the loaded modules inside a process in the user-mode from the PEB and all the loaded modules according to the kernel-mode objects, will be good indication if Process Hollowing was performed on this process.
The commands that need to be used in order to do this comparison are "dlllist", which that lists all the loaded modules from the PEB, and "ldrmodules", which lists all the loaded modules from EPROCESS kernel objects.
There is plugin in volatility that perform the comparison between the loaded modules in the kernel-mode and the user-mode called "HollowFind".
One more way to detect Process Hollowing is by using another Volatility plugin called "malfind", which do comparison between information from the PEB to information in the VADs.

2.4.4 **POC**

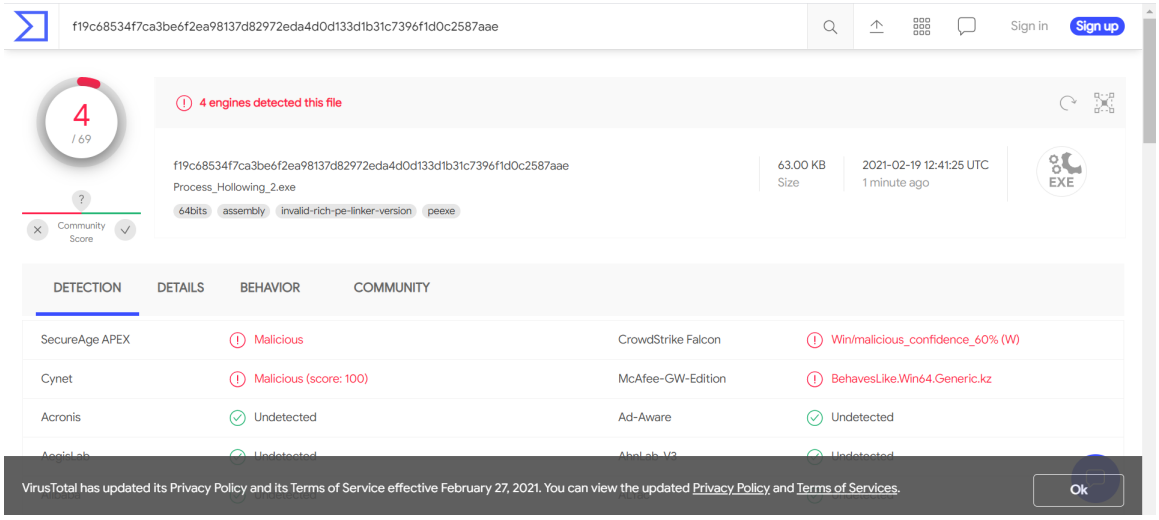
The base code for the POC is taken from:
https://github.com/idan1288/ProcessHollowing32-64

As part of trying to improve the current code the base code was changed.
At the beginning the main file was getting two arguments: the process, which will be hollowed and the process which replace it in memory.
To the base code, there were added a simple anti-sandbox technique and anti-assembly technique to make the detection a little bit harder.
This project support 64-bit files only.

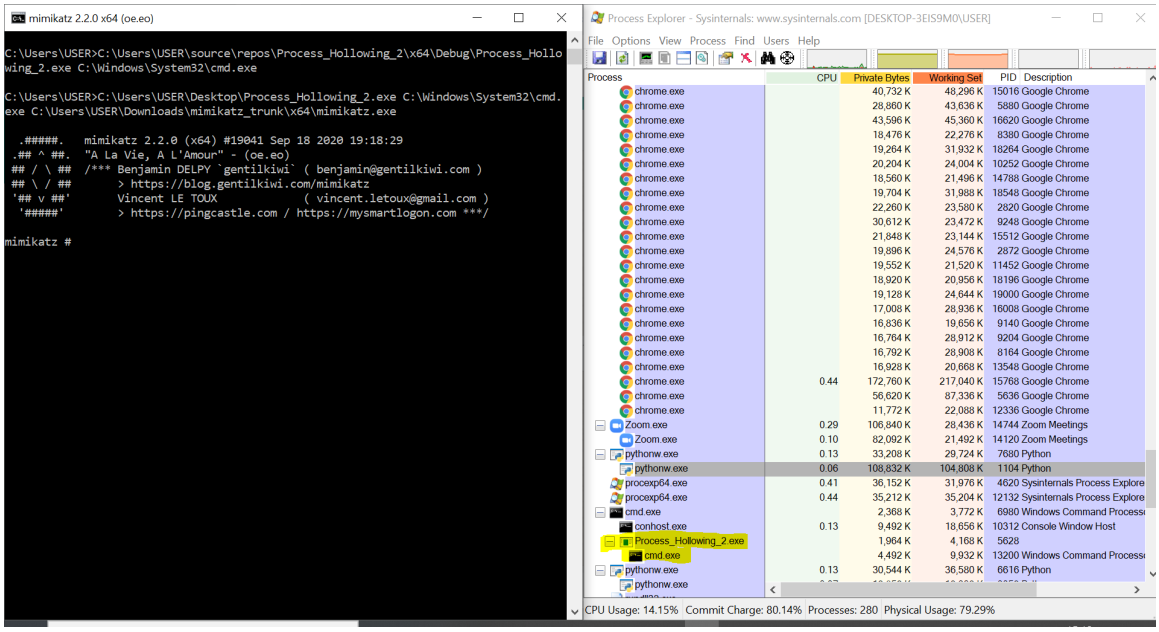
```
int count = 0;
for(int i = 0; i < MAX_OP; i++)
{
    count++;
}

if (count == MAX_OP)
{
    FILE* fp = fopen("c:\\windows\\system.ini", "rb");
    if (fp == NULL)
        return 0;
    fclose(fp);
```

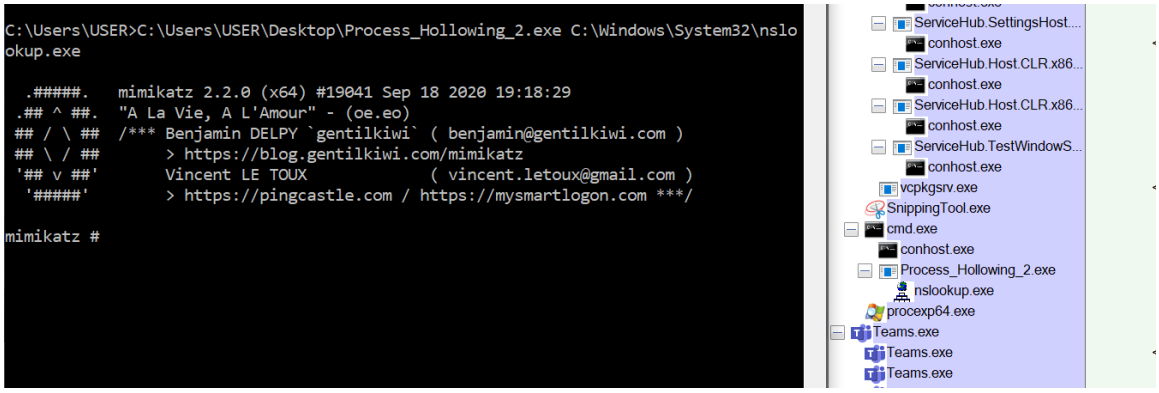
After the first change in the source code only four anti-viruses detected this exe file as malicious.



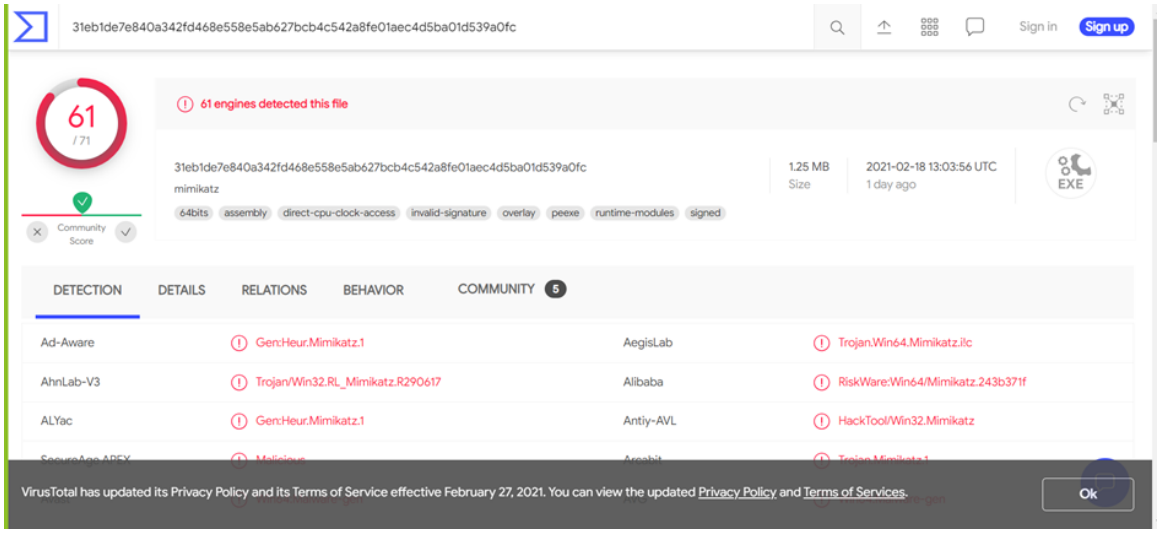
In the following picture, cmd is running, but the attacker has mimikatz instead.



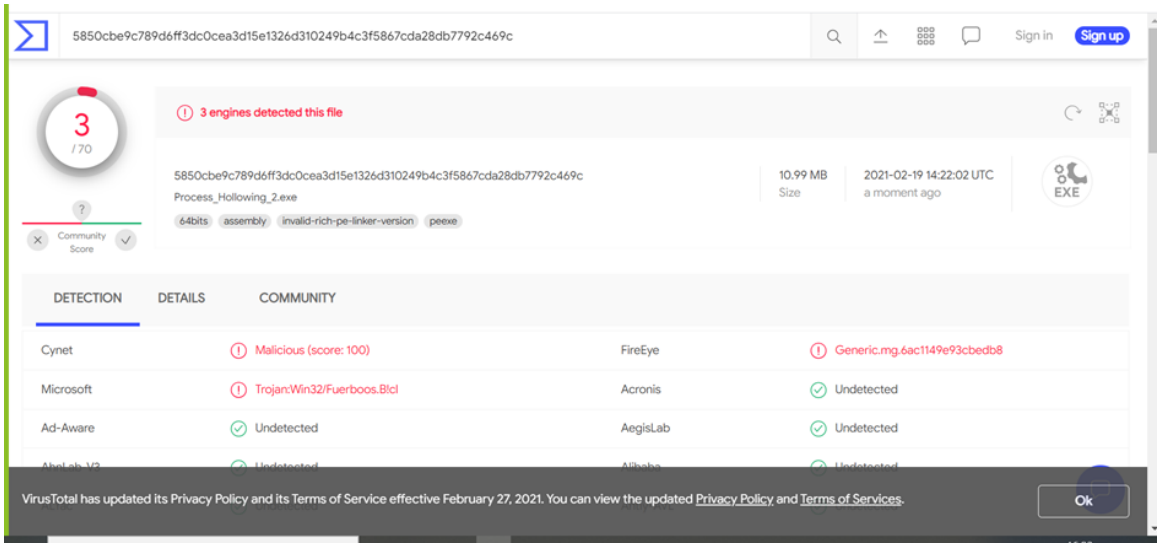
In order to improve and make the attack more sophisticated, we used simple script in Python, which read the file in hexadecimal and put it as array of chars in the code.
By this way, the malicious software (in our case Mimikatz), will be part of the code.
After the change, our file will get one argument, which will be the hollowed process. In this example, the process in the process explorer will be nslookup, but the real file is Mimikatz.



If we will check Mimikatz on VirusTotal 61/71 engines will detect the file as malicious:



However, when we checked our file, which contains the content in hex of Mimikatz 3/70 engines detected it.



To sum up, **Process Hollowing** is advance Process Injection sub-technique, which is used by common and known malwares and attacks.
This technique can be implemented in several ways, and the main way to detect this attack is by memory forensic.
As it was written above, the detection is being done by comparison between the PEB in the user-mode and kernel-mode objects. Because it is much easier to make changes on user-mode objects it is possible to avoid detection while using this technique. In our AV-Evasion platform we will use Process hollowing as one of our modules in order to avoid anti-virus engines.

2.5 Code Normalisation and Obsufication

As of today, most Anti-virus engines use two main methods to detect and decide if an executable is malicious or not. Those two methods are **Detection of malicious data blocks** (either strings or code blocks) and **Heuristics'-based detection of malicious actions**. Using those two methods, most Anti-virus engine manage to detect most "in the wild" viruses, either previously detected un-signed viruses and newly created/never discovered viruses.

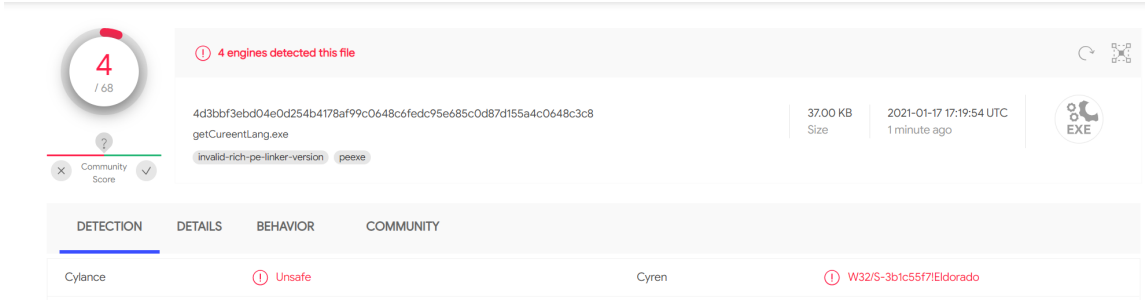
2.5.1 Malicious Block Detection

A known technique in which the Anti-virus engine searches in an executable for strings or data blocks which are "signed" and known to the Anti-virus company as malicious. In fact, most Anti-virus Engines keep a Database of virus signatures. Those blocks can be part of a string which was detected in a common virus, Cryptographic keys known to be part of an attack, shellcodes and more. When the executable is created on disk (and sometimes, when it's loaded into memory by a new process), the AV engine will scan it – sometimes pre-running and sometimes mid-execution. If the executable contains those signed blocks, the AV engine will detect it and declare it malicious.

For example, we wrote a simple WinAPI program, which detects what language is currently in use in the computer.

```
int main()
{
    while (1) {
        HWND hForegourndWindow = GetForegroundWindow();
        //HWND frgrndHandle = (HWND)OpenProcess(PROCESS_ALL_ACCESS, NULL, 9012); //Explorer PID
        if (hForegourndWindow == NULL) {
            printf("getForegroundWindow Failed... GLE=%d", GetLastError());
            exit(0);
        }
        DWORD fouregroundTID= GetWindowThreadProcessId(hForegourndWindow, NULL);
        printf("tid: %d\n", fouregroundTID);
        HKL keyboardLayout = GetKeyboardLayout(fouregroundTID);
        printf("keyboard layout: %x\n", keyboardLayout);
        Sleep(1000);
    }
    return 0;
```

Obviously, this project is **NOT** malicious, and is using methods which every user may and might use. And when will check it against most Anti-virus engines out there, we will see that indeed most of them will not detect this program as malicious (only 4 out of 68 programs will detect this program).

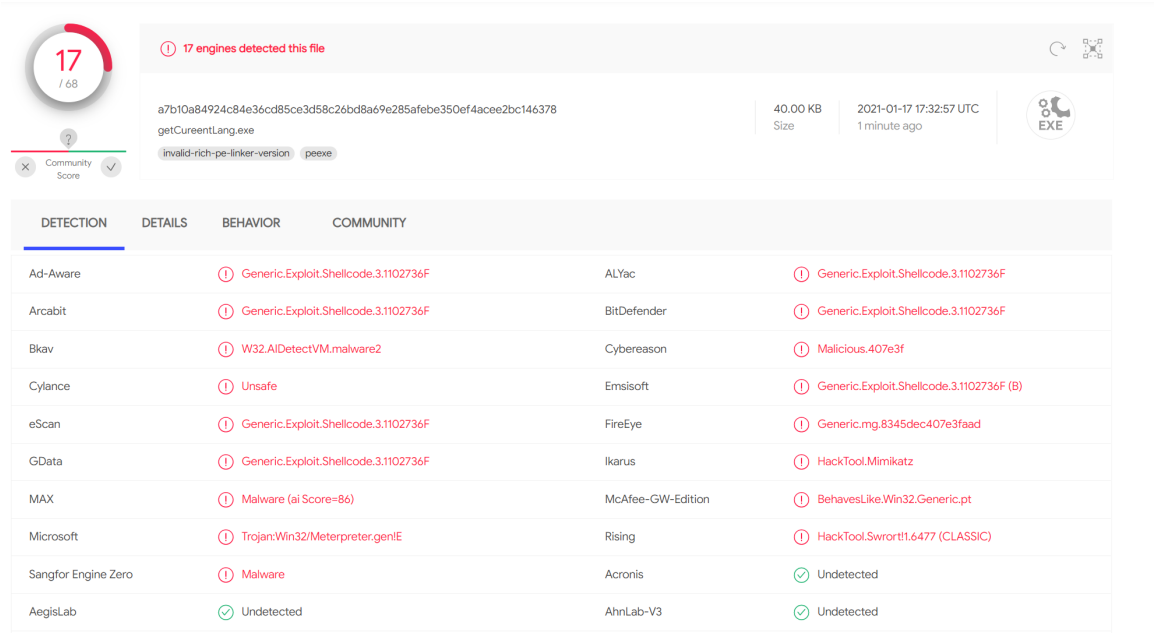


Now, we will edit our project, without changing it's behavior, and we'll add a few strings. Those strings are taken from known attack oriented tools published on github, for example from the Metasploit project (a Post-Explotation tool built for attackers and red-teamers alike), Impacket (a project built for letheral movement and Windows-Domain attacks) and Mimikatz (a tool built for grabbing windows password from a computer).

```
int main()
{
    unsigned char buf[] =   "\xfc\xe8\x8f\x00\x00\x00\x60\x31\xd2\x89\xe5\x64\x8b\x52\x30\x8b\x52\x0c\x8b\x52\
    unsigned char buf2[] = "python -c \"import sys;import ssl;u=__import__('urllib'+{2:'',3:'.request'}[sys.ve
    unsigned char buf3[] = "powershell.exe -nop -w hidden -e WwBOAGUAdAAuAFMAZQByAHYAaQBjAGUAUABvAGkAbgB0AE0AY
    while (1) {
        printf( " .#####.   mimikatz 2.0 alpha (x86) release \"Kiwi en C\" (Apr  6 2014 22:02:03)\n"\
                ".## ^ ##.\n"\
                "## / \ ##  /* * *\n"\
                "## \ / ##   Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )\n"\
                "'## v ##'   https://blog.gentilkiwi.com/mimikatz            (oe.eo)\n"\
                "'#####'                                     with  13 modules * * */\n"
        );

        HWND hForegourndWindow = GetForegroundWindow();
        //HWND frgrndHandle = (HWND)OpenProcess(PROCESS_ALL_ACCESS, NULL, 9012); //Explorer PID
        if (hForegourndWindow == NULL) {
            printf("getForegroundWindow Failed... GLE=%d", GetLastError());
            exit(0);
```

Now, after adding those "malicious" strings (please note that our project's behavior and actions did not change), we'll check our program again against most Anti-virus engines, and we'll see that now, many more engines detect our program as malicious, including many "Big Players" in the industry that hold a big share of the market in the likes of Windows Defender, Cyberreason FireEye and more.



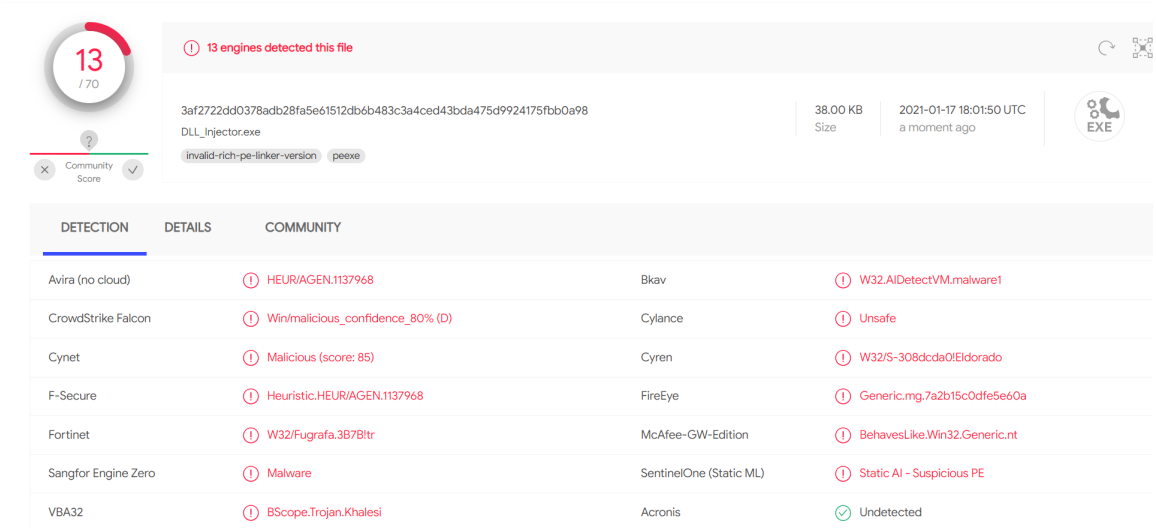
2.5.2 **Heuristics'-based detection of malicious actions**

Another method which most Anti-virus engines are using is Heuristics'-based search of action which the Anti-virus company defined might be malicious. Those actions were defined as potentially malicious by detecting similar behavior in many known attack and viruses, and most time those actions are needed in order to complete the full chain of attack on a computer/network (and example for which will be shown in later on). This search for behavior is more effective than searching for signed blocks, because it isn't based on a specific data (URL's, keys etc...) an attacker might change as he pleases, but is based on a set of actions which an attacker must perform in order to execute his malicious code/attack.
As an example, we will create a project that performs a classic DLL Injetion. It is important to note – in this project we did not use any blocks/strings that are connected to a known attack/project. Therefore, if our project will be detected as malicious – it will be detected by its behavior.
To perform a DLL Injection, we will create a Handle to another process (in this instance – notepad.exe). Then we will allocate space in the remote process' address space and write the path to our dll there in order to dynamically load it. Then we'll search for LoadLibrary address and create a remote thread that loads our library.

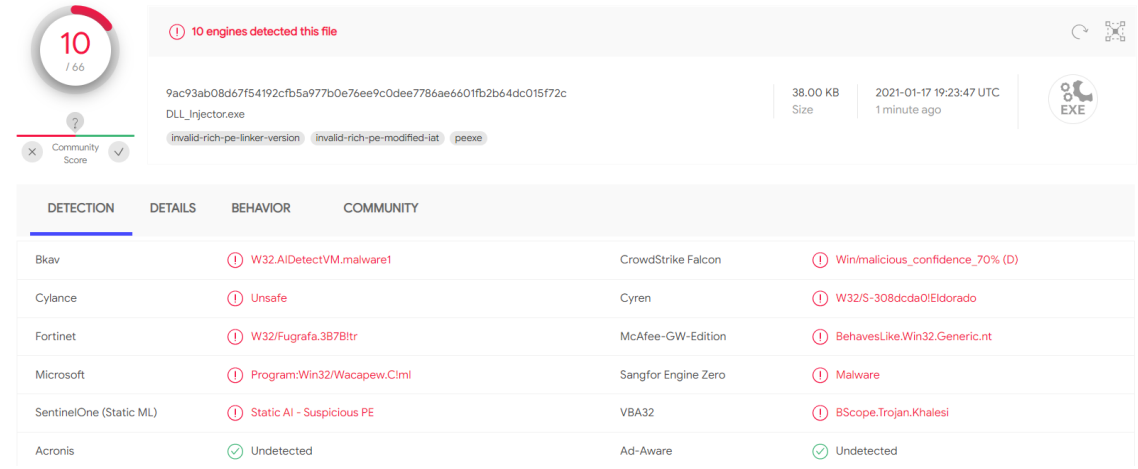
```cpp
void GLE(const char* err) {
    printf("Error at: %s, GLE=%d\n", err, GetLastError());
}

int main()
{
    DWORD processId = getProcessId(L"notepad.exe");
    if (processId == 0)
        GLE("getProcessId");
    char dllLibFullPath[] = "c:\\Windows\\System32\\user32.dll";
    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, false, processId);
    if (hProcess == NULL)
        GLE("OpenProcess");
    LPVOID dllAllocatedMemory = VirtualAllocEx(hProcess, NULL, strlen(dllLibFullPath), MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    if (dllAllocatedMemory == NULL)
        GLE("VirtualAllocEx");
    if (!WriteProcessMemory(hProcess, dllAllocatedMemory, dllLibFullPath, strlen(dllLibFullPath) + 1, NULL)) {
        GLE("WriteProcessMemory");
    LPVOID loadLibrary = (LPVOID)GetProcAddress(GetModuleHandleA("kernel32.dll"), "LoadLibraryA");
    if (loadLibrary == NULL)
        GLE("GetProcAddress");
    HANDLE remoteThreadHandler = CreateRemoteThread(hProcess, NULL, 0, (LPTHREAD_START_ROUTINE)loadLibrary, dllAllocatedMemory, 0, NULL);
    if (remoteThreadHandler == NULL)
        GLE("CreateRemoteThread");
    CloseHandle(hProcess);
    return 0;
```

Now we'll look how many Engines detect our program as malicious. And indeed, many engines detected our program.



It is interesting to note that when we added a few calls to Sleep, the number of Antivirus engines that detected our program was lowered, but not majorly enough.


10
/ 66
Community Score
10 engines detected this file
9ac93ab08d67f54192cfb5a977b0e76ee9c0dee7786ae6601fb2b64dc015f72c
DLL_Injector.exe
invalid-rich-pe-linker-version
invalid-rich-pe-modified-lat
peexe
38.00 KB
Size
2021-01-17 19:23:47 UTC
1 minute ago
EXE
DETECTION
DETAILS
BEHAVIOR
COMMUNITY
Bkav
W32.AIDetectVM.malware1
CrowdStrike Falcon
Win/malicious_confidence_70% (D)
Cylance
Unsafe
Cyren
W32/S-308dcda0!Eldorado
Fortinet
W32/Fugrafa.3B7B!tr
McAfee-GW-Edition
BehavesLike.Win32.Generic.nt
Microsoft
Program:Win32/Wacapew.C!ml
Sangfor Engine Zero
Malware
SentinelOne (Static ML)
Static AI - Suspicious PE
VBA32
BScope.Trojan.Khalesi
Acronis
Undetected
Ad-Aware
Undetected

Acknowledgments

We would like to acknowledge support for this project from our guide, Dr. Nezer Zaidenberg and our academic institute, The Collage of Management, Rishon Le Tzion, Israel.

References

- Shellcoding, edition 49-101, Digital Whisper.
- Threadmap Documentation – by KSL group.
- The Art of Memory Forensic by Michael Hale Ligh, Andrew Case, Jamie Lavy and Aaron Walters.
- Mastering Malware Analysis by Alexey Kleymenov and Amr Thabet.
- Windows Internals 7th edition by Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich and David A. Solomon.
- https://en.wikipedia.org/wiki/Shellcode
- https://www.solarwinds.com/sa-overview/securityadvisory
- https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html
- https://github.com/idan1288/ProcessHollowing32-64
- http://www.rohitab.com/discuss/topic/40262-dynamic-forking-process-hollowing/