Journal of Computer Virology and Hacking Techniques (2022) 18:315–331
https://doi.org/10.1007/s11416-022-00417-2

ORIGINAL PAPER

PEzoNG: Advanced Packer For Automated Evasion On Windows

Giorgio Bernardinetti[1] · **Dimitri Di Cristofaro**[2] · **Giuseppe Bianchi**[1]

Received: 12 July 2021 / Accepted: 21 January 2022 / Published online: 7 February 2022

Abstract
The ability to evade Antivirus analyses is a highly coveted goal in the cybersecurity field, especially in the case of Red Team operations where advanced external threats against a target infrastructure are performed. In this paper we present the design and implementation of *PEzoNG*, a framework for automatically creating stealth binaries that target a very low detection rate in a Windows environment. *PEzoNG* features a custom loader for Windows binaries, polymorphic obfuscation, a payload decryption process and a number of anti-sandbox and anti-analysis evasion mechanisms, including a novel user space unhooking technique. In addition, the custom loader supports a large amount of Windows executable files, and features stealth and advanced memory allocation schemes. We evaluate the effectiveness of PEzoNG by testing various malicious payloads against up to 29 commercial Antivirus solutions, and we highlight and discuss the assets and differences of PEzoNG with respect to similar tools.

Keywords Malware · Evasion · Windows · Packer

1 Introduction

Executing payloads without being detected by an Antivirus is one of the elements that leads to the success of a Penetration Test [1] or a Red Team [2] campaign; being invisible to the eyes of an Antivirus provides a wide range of possibilities both to gain remote access and to escalate privileges on a target machine. The amount of time required to figure out the right payload to be deployed on a target machine mainly depends on gathering as much information as possible about this target, e.g. Operative System (OS) version, any Antivirus (AV) installed, patches applied, etc. Once the payload has been conceived, there might be the need to make it undetectable by an Antivirus, especially if the target OS is Windows. The attacker (which may either be a penetration tester, a red teamer, or a real malicious threat) needs to set up an environment similar to the target one in order to verify that the payload will not trigger any alarm once deployed on the target. *PEzoNG* aims to automate all the process of making a payload undetectable, providing an automatic way to embed any payload into a custom loader which takes care of being invisible. The input of *PEzoNG*, which can be either a normal Windows executable file (a *Portable Executable*—PE [3]) or a Windows *shellcode*, is encrypted and embedded inside the loader; the loader is then obfuscated to obtain a polymorphic binary in the output. This final result is another payload which has two main features: 1—it's a payload with a low detection rate (i.e. both static and dynamic analysis are bypassed) and 2—the behavior of the original payload is unchanged.

Our work starts from *PEzor* [4], an existing opensource PE and shellcode packer. As we describe in more details in Sect. 2, this tool, and some of its dependencies, have limitations which may either trigger an AV alarm or leave known artifacts in memory (which can be identified by a forensic analysis). *PEzoNG* overcomes all of them, while at the same time provides new technologies and implements both static and dynamic analysis bypass methodologies. Moreover, at the time of writing *PEzoNG* it's a completely different project from *PEzor* as they only share a part of the name and the building environment. We tested *PEzoNG* by packing well known payloads identified as malicious by many AV software. The resulting payload was successfully executed and

✉ Giorgio Bernardinetti
giorgio.bernardinetti@cnit.it

Dimitri Di Cristofaro
dimitri.di.cristofaro@secforce.com

Giuseppe Bianchi
giuseppe.bianchi@uniroma2.it

[1] CNIT/University of Rome "Tor Vergata", Rome, Italy

[2] SECFORCE LTD, London, England, UK

not detected by the AV thus demonstrating the effectiveness of *PEzoNG*. Moreover, *PEzoNG* was designed to be stealth even to the human eyes of a Blue Team [5], opposed to the Red Team. As discussed later, allocating a private memory area with the Read, Write and Execute (RWX) flags could be harmless for a number of AVs but suspicious to a human being and to modern EDR systems, both while the malicious payload is running and while conducting a forensic analysis of the RAM content. For this reason, allocating memory in such a way is considered to be an issue in the next sections. Finally, we propose a new way for unhooking hooked functions in Windows libraries that would allow the unhooking process without the need of reading the original library from disk.

In summary, the contribution of this paper and the unique assets of *PEzoNG* are the following:

- An environment in which embedding malicious payloads to make them undetectable
- A novel unhooking technique
- A custom PE loader with stealth memory allocation
- A custom payload double-encryption process
- A custom *function call obfuscation* method to invoke Windows APIs

The remainder of this paper is organized as follows: Sect. 2 is an overview of related work; Sect. 3 describes the design and implementation of *PEzoNG*; Sect. 4 shows the results we achieved and Sect. 5 concludes this paper.

2 Background and related work

PEzor (version 1.0) [4] is the PE packer from which *PEzoNG* was born; at the time of writing the two projects have diverged, meaning that *PEzor* focused on different features than *PEzoNG*. For this reason, in the remainder of this paper, when referring to *PEzor* we consider version 1.0. *PEzor* contains many state of the art tools and combines them to generate a PE containing a malicious payload with a low detection rate; we analyzed all of them and the focus of our research is to overcome their limitation while at the same time provide new features for evasion. *PEzor* makes extensive use of *Donut* [6]: "*Donut is a position-independent code that enables in-memory execution of VBScript, JScript, EXE, DLL files and dotNET assemblies*". To summarize, *Donut* converts an executable file to shellcode by prepending a small custom loader before the actual executable. An issue of this loader is that it leaves known artifacts in memory: *Donut* allocates memory (where the executable file bytes are copied to) using the `VirtualAlloc` API (Application Programming Interface): as we discuss in more details in Sect. 3.5, memory allocated in this way can be quickly identified; moreover, the allocation flags of this memory area include the $Executable$ Flag, i.e. it contains code that is going to be executed at some stage.

PEzoNG does not use *Donut* but, like *Donut*, needs to allocate memory somehow, so it overcomes this issue by changing the allocation scheme to a modified version of the *Dll Hollowing* [7] technique: memory allocated this way has the same properties of memory allocated by Windows when loading a Dll into the process address space.

PEzor also employs the *Shikata Ga Nai Encoder* [8] as an encoding mechanism for the malicious payload in order to obtain a polymorphic payload, different at each generation, to bypass static detection mechanisms. SGN also has a problem, namely, it needs the memory allocated where the payload resides to necessarily have RWX flags. This is also an issue that *PEzoNG* successfully overcomes.

Additionally, *PEzor* uses a custom loader to load the payload into memory (transformed into shellcode with *Donut* and made polymorphic with *SGN*), and exploits the classic pattern of shellcode allocation and execution on Windows, namely the sequence of invocations: `VirtualAlloc`, `WriteProcessMemory` and `CreateRemoteThread`; moreover, instead of calling these *Application Programming Interfaces* ($APIs$), the underlying syscalls are invoked—which, as we'll see later, has the advantage of avoiding some detection mechanisms- using the $inline\_syscall$ project [9] which, however, has a limitiation; namely it does not work if the system call wrapper in the Windows library *NTDLL.dll* is hooked [10]. This happens because the $inline\_syscall$ project parses *NTDLL.dll* searching for symbols name (e.g. `NtClose`) and gets the system call number by reading at an offset of the symbol address. This approach won't work if the stub is hooked because the system call number won't be there. An example targeting `NtClose` is showed in Fig. 1.

This limitation entails a reliability issue: if the functions used by the loader are hooked, the loader must unhook them before the payload is loaded otherwise the loading process will fail. *PEzor* implements the unhooking feature by using *DLLRefresher* project [11] which can trigger some AVs because of hooked functions and malicious behaviour (e.g. NTDLL.dll is loaded from disk—more about this is described in details in Sect. 3.4.2). Moreover, for what concerns user-space unhooking, in [12] 7 Antivirus software are analyzed and different unhooking techniques are evaluated so as to discuss their effectiveness against the same Antivirus; in this paper, and more specifically in Sect. 3.4.2, we analyze 16 Antivirus and discuss a novel technique for user-space unhooking.

PEzoNG implemented syscalls invocation with the *syswhispers2* project [13], and allows memory allocation with the aforementioned new allocation scheme—derived from the $ModuleOverloading$ [14,15] technique— which is described in more details in Sect. 3.

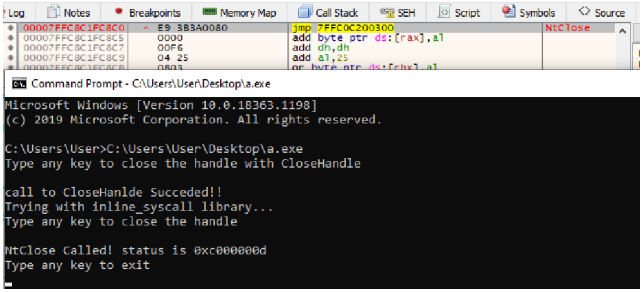

Fig. 1 Usage example of inline_syscall library calling NtClose when it is hooked by BitDefender Total Security

With regards to static analysis bypass, the malware has to be obfuscated in a way that the same source code would result in different binary files at each compilation. In particular, implementing obfuscation techniques that allows to obtain multiple different outputs allows to avoid trivial signature based detection since no unique signature can be computed to identify them. Using metamorphic obfuscation techniques has been proven to be effective against static analysis [16,17].

The *PEzor* loader is obfuscated using *LLVM*-based obfuscators, e.g., *YansoLLVM* [18], to obtain a final PE that is also polymorphic; *PEzoNG* also uses this mechanism to obfuscate the code, generating a polymorphic binary. However, the author of *PEzor* didn't release the source code for obfuscation and a comparison with *PEzoNG* is not possible. The usage of *LLVM*-based obfuscators is a well-known evasion technique [19] that allows to obfuscate the code at compile time. The *LLVM* framework allows to easily add further steps to the compilation (i.e. operations to manipulate the intermediate representation of the code) while supporting a large number of programming languages and output architectures which makes it a good candidate for obfuscating binaries.

Finally, *PEzoNG* implements additional evasion mechanisms with respect to *PEzor*, as described in Sects. 3.4 and 3.1, as well as *function call obfuscation* to invoke APIs (Sect. 3.2), a custom *PE* loader (Sect. 3.5) and a novel userland unhooking technique (Sect. 3.4.2).

In [20] many open source packers are evaluated against Bitdefender [21] which, according to the referenced statistics, is the most effective Antivirus software. In total 9 packers were evaluated and the maximum evasion rate was 50%, meaning that half of the payloads were detected by Bitdefender; in particular, two of the payloads that are always detected regardless of the packer are meterpreter [22] implants. By packing the same payloads with *PEzoNG* we show in Sect. 4 that the evasion rate is 100% against not only Bitdefender, but also a number of other AV software. In [23] an evaluation similar to the previous one was carried out, testing 5 different Antivirus software against 4 open source PE packers. The best evasion rate in this paper is 60%, and the packed payloads are meterpreter implants and custom reverse shells.

3 PEzoNG

PEzoNG is a project written in C and C++. Although this project targets Windows only, it has to be compiled using the *Mingw-w64* [24] development environment together with the *LLVM toolchain* [25] in order to compile and link. The toolchain made up of *Mingw-w64* and *LLVM/clang* can cross-compile Windows executables from a GNU/Linux machine.

PEzoNG source code is made up of three main components:

1. the malicious payload, i.e. the input of *PEzoNG*,
2. the evasion code, which allows to evade from Antivirus sandboxes and Endpoint Detection and Response (EDR) solutions, and finally
3. the main loader, which loads the malicious payload into memory and executes it.

PEzoNG is built with modularity in mind and allows to add new features in a simple way by adding new modules that could implement different techniques with a fine grained detail. The project is organized in the following modules:

- Encryption
- APIs
- Syscalls
- Evasion
- PE loader
- Shellcode injection

Each module can implement different techniques that can be chosen when packing a malicious payload. Moreover, this structure allows to decouple the implementation of the techniques from the actual packer giving the flexibility to mix different techniques together as well as adding new ones with low effort.

The process of compilation and linking is not trivial and it is divided into many steps (Fig. 2):

- The encrypted payload is embedded in the template source code
- The source code (all but the payload) is compiled into *LLVM* Intermediate Representation (IR)
- The IR obtained in the previous step is obfuscated using *YansoLLVM*
- The obfuscated IR as well as the payload are compiled and linked into binary format

Since the evasion code and *PEzoNG* loader are obfuscated using *YansoLLVM*, the generated output is polymorphic, and as such trivial static signature detection methods used by Antivirus software are not effective. Moreover, we recall that the malicious payload is not obfuscated using *YansoLLVM*: as we will discuss in more details later on, the payload is actually encrypted in two different stages so as to decrypt it during execution by reversing those stages, in a way that allows to bypass AV logical paths hijacking. [26].

The high level operations performed by the generated PE can be divided in the same way as the three main components of *PEzoNG* (Fig. 3), along with the modules involved in each phase:

1. The evasion code is executed
   - Evasion, Syscalls
2. The malicious payload is decrypted
   - Encryption
3. The custom loader is invoked
   - APIs, Syscalls, PE loader, Shellcode injection

The next sections describe the aforementioned modules in more details.

3.1 Payload decryption

As briefly explained before, the malicious payload is encrypted in two steps during *PEzoNG* compilation so as the actual payload cannot be trivially extracted from the final packed binary. The encryption keys are randomly generated using *openssl* during each packing process and their length is fixed to 256 bit; then a Python script is used to encrypt the payload and the encryption keys are embedded in a header file of the crypto module of *PEzoNG*. The encryption algorithm can be selected by the user, and the current choices range from a baseline XOR encryption up to AES256-CBC. We remark that our usage of an encryption algorithm is *not* related to the need to protect confidentiality, but "just" with the goal of evading static analysis and sandboxing. Therefore, even a semantically insecure algorithm (such as XOR with a constant random pattern) meets our needs.

Decryption happens in two stages too, because an AV can modify logical paths taken by an application in order to analyze its behavior [26]: if there is an if-else branch in the code, the AV can choose to always run one branch by changing the result of the checked condition (e.g. run all the branches as if they were all true or false). For this reason, *PEzoNG* implements two branches inside a loop which, under normal conditions, are both evaluated as true for a single value of the loop iterator; in this way, there is only one possible path that allows the complete decryption of the payload and, this path cannot be taken if the logic inside the if statements is changed (Listing 1).

Note that our two stages of encryption are devised to bypass AV logical path hijacking. Therefore, the usage of a stream cipher or even an XOR encryption (where encrypting twice is actually equivalent to a single encryption) is correct in our context. Indeed, the AV can only see random bytes in memory until the double-decryption step is executed—those bytes are going to become meaningful only after the two decryption steps, i.e. after the two logical conditions are executed without tampering by an Antivirus.

Listing 1 Payload decryption steps

```
a = 1337;
c = 1337;
for (int i = 0; i < 100000000; i++) {
    if(a == 1337 && i == 98765400 && c != 7331) {
        compute(); // Huge Computation
        decrypt1(); // First stage Decryption
    }
    if(a != 7331 && i == 98765400 && c == 1337){
        decrypt2(); // Second stage Decryption
    }
}
```

If the AV changes either one or both of them, the final decryption will be wrong and the next execution stage will fail (the main loader), so *PEzoNG* will not execute any potentially malicious code. Moreover, there will be no malicious artifacts left in memory (i.e. the original payload) because of the wrong decryption, but only a sequence of nonsense bytes.

3.2 APIs

In order to setup its environment *PEzoNG* needs to call multiple Windows APIs, many of them usually used by many malicious payloads. When an executable file makes use of Windows APIs, their names are included in the *PE Import Table*, so that the OS can load them at run-time and make them available to the process. The *Import Table* is part of the PE metadata so every imported API implies the presence of a string containing the API name inside the PE. Even the mere presence of certain strings inside an executable file may mark it as suspicious and trigger the AV to perform deeper analysis, so we implemented an automatic *function call obfuscation* method which allows to dynamically resolve any Windows API address at run-time without ever specifying the API name. A Python script is used to compute the hash of all the used Windows APIs using a compile-time salt. At run-time each API is called using its corresponding hash—transparently to the programmer, which continues to use the API name—and its address is resolved similarly to what the *PEzoNG* main loader does—as explained in Sect. 3.5. In particular a function belonging to a dynamic library already mapped into the process address space can be resolved by parsing a linked list inside the *Process Environment Block*

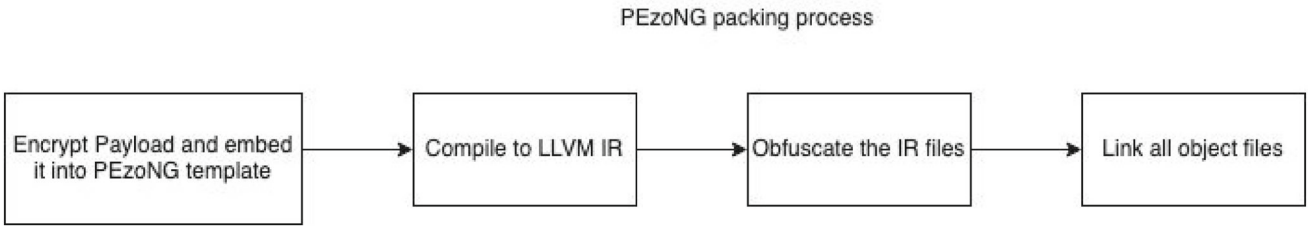

Fig. 2 Packing process in PEzoNG

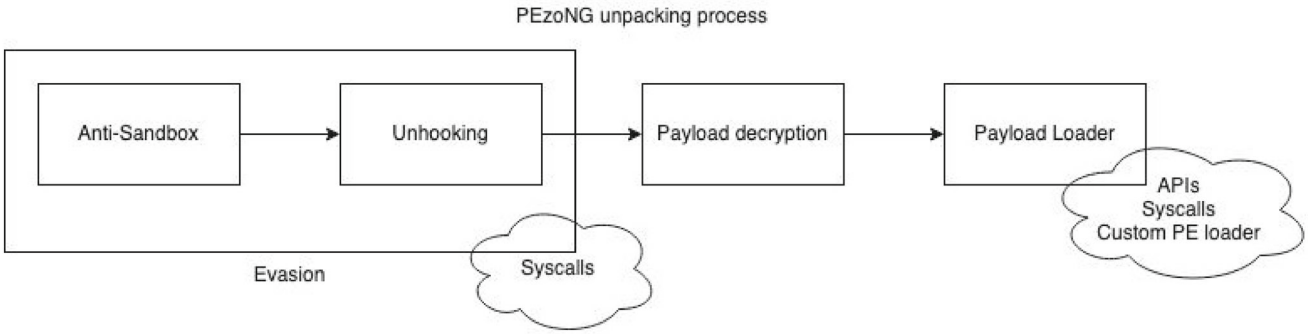

Fig. 3 UnPacking process in PEzoNG

[27]; we iterate over this linked list until we find an API whose hash is equal to the one provided by the caller. Since all the APIs used by *PEzoNG* belongs to two dynamic libraries—ntdll.dll and kernelbase.dll—which are always loaded by Windows into every process address space, all of them can be correctly resolved at run-time. Following is an example of how to call an API using the method we provide:

```
API_CALL(ApiName, param1, param2);
```

The are two main advantages of using this method; the first one is that the API names will never appear inside the PE, but only their hashes will, and since the salt is changed on every compilation each PE packed by *PEzoNG* will contain different hashes. The second advantage is that we do not rely on two other Windows APIs to perform the run-time API resolution, i.e. *LoadLibrary* and *GetProcAddress*, which are also usually employed by malicious software.

3.3 System calls

PEzoNG performs a number of tasks by directly invoking the underlying syscalls used by an API without invoking the API itself (e.g. the high-level API `VirtualAlloc` calls the system call `NtAllocateVirtualMemory` at some point of its execution), thus avoiding the user-land hooking engine [28] implemented by Antivirus (AV) and Endpoint Detection and Response (EDR) software vendors.

Windows provides wrappers for system calls that are meant to hide the internal structure and possible changes of the internal operating system services. System calls wrappers

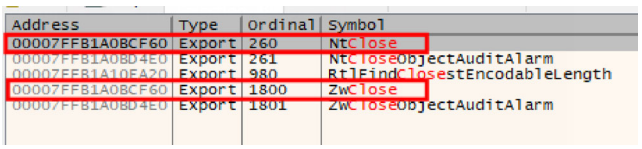

Fig. 4 ZwClose and NtClose pointing to the same address

use a name convention, namely user-space system service function names start with `Nt` and the corresponding kernel-level functions start with `Zw`. A user-space program does not have access to kernel-space routines thus, in user-space `Zw*` functions are at the same address of the corresponding `Nt*` function. Figure 4 shows the user-space system call wrapper `NtClose` and the corresponding `ZwClose` kernel-level function pointing to the same address in the Export Directory of *NTDLL.dll*.

PEzoNG implements direct system calls with the help of the *Syswhispers2* project [13] which allows to resolve the system call numbers dynamically at runtime even if the system calls have been hooked in user space.

The technique was popularized by ElephantSe4l [29] and MDSec Research [30]; it is based on the observation that the system call number is used as an offset to identify the position in memory of the real system service. In particular, system call numbers can be obtained by ordering by address all the Zw* functions in *NTDLL.dll* so that a smaller system call number will correspond to a lower position in memory. For example, "The stub with the lowest memory in Windows

10 1909 is NtAccessCheck and if we check the associated syscall number... it is 0!" [29]

3.4 Evasion

PEzoNG implements different evasion techniques to defeat anti-malware monitoring capabilities used for dynamic analysis. In particular, *PEzoNG* addresses sandbox execution as well as user space hooking.

PEzoNG can be extended by adding more evasion techniques to this stage of execution even though the mechanisms we implemented are sufficient for the commercial AV solutions that we tested (Sect. 4).

3.4.1 Anti-sandbox

Many anti-sandbox techniques implements delayed execution by sleeping for X seconds before executing the malicious code. However, EDRs in the first place, but also some AVs, may ignore the call to the `sleep()` function, thus executing the payload without delay and triggering alarms. Because of that, PEzoNG implements dynamic analysis evasion using a slightly modified version of the *Offer you have to refuse* [31] technique. This technique is based on the concept that AV engines cannot use large amount of resources to analyze a potentially malicious program. The implemented technique executes useless instructions that are memory dependent between each other and whose execution time is about X seconds. Since the sandbox cannot execute the code for a long time for performance reasons to avoid degrading usage experience, if the malicious payload is triggered after the time used by the sandbox engine to analyze the binary, the binary results harmless and thus, there is no detection of the malicious behavior. After many experimental tests (Sect. 4) we found the optimal amount of useless computations needed in the average case.

3.4.2 User-space UnHooking

User-space API hooking is a well-known technique used by AVs and EDRs to monitor the execution of a process at runtime in order to detect malicious patterns. In particular, a number of system functions are hijacked by the security product overwriting the first instructions of the function with a jump instruction which redirects execution flow to a piece of code controlled by the security software before returning to the original API code. Which exact functions are hooked depends on the security product in use, however, functions that are commonly used for malicious purposes are often hooked.

Even if *PEzoNG* is extremely careful in using stealth techniques to invoke Windows APIs and syscalls, the embedded payload may not be so careful thus it may still raise alarms if user-space hooking is employed by an anti-malware software. For this reason *PEzoNG* allows to patch the hooking procedure in order to make Anti-Virus software blind.

API Hooking is a very effective detection technique as it allows to take actions basing on real-time events that could trigger the detection of the malicious software after it has started to run. For example, *AVG Internet Security* [32] was able to detect a *Cobalt Strike* [33] raw stageless beacon shellcode packed with *PEzoNG* without us having enabled the unhooking feature of our packer. In particular, the malware was not detected when the file was placed on disk (static analysis), nor when the beacon was loaded in memory (dynamic analysis) nor when it connected to the Command-and-Control server but rather when a certain command was executed on the system through the beacon. The reason behind this is that once the beacon was run, the packer couldn't protect it anymore because the AV software employed run-time detection techniques, namely by hooking Windows APIs.

As we show in Sect. 4, we successfully executed malicious payloads without getting caught by many anti-malware software by packing them with *PEzoNG* after enabling the unhooking feature with the novel *Whisper2Shout* technique.

From an attacker's perspective, one way to bypass these security products is to attempt to remove the hooking. There are many documented techniques to remove user-space hooking [34–37] but all of them require either reading the original library (DLL) from disk or reading its contents from a remote process'memory space before the library is already hooked by the security product. Detection of those techniques is usually implemented with the support of Windows kernel, by using minifilter drivers [38]. Windows allows anti-malware software to register callbacks for a number of system events including file operations and process creation [39]. This means that the AV will be notified when such events happen in the system and could trigger a deeper analysis that would potentially lead to detection. For example, reading the contents of the NTDLL.dll file, which should only be loaded during process creation, can be considered suspicious and could lead to detection.

PEzoNG implements two unhooking techniques that can be chosen by the operator

1. *Shellycoat* [40]
2. *Whisper2Shout*

Shellycoat technique is a well-known technique which unhooks a hooked DLL by loading a clean version from disk. This technique uses `NtCreateFile`,`NtCreateSection` and `NtMapViewOfSection` to load a fresh copy of the DLL in the process address space, it copies the original bytes of its text section in the text section of the hooked DLL and

Table 1 Antivirus employing API Hooking

Antivirus	API Hooking
AVG	✓
Avast	✓
BitDefender	✓
Comodo	✓
MalwareBytes	✓
ESET Internet Security	✓
Sophos Home 3.0	✓
Norton 360	✓
Trend Micro	✓
Dr. Web	✓
Windows Defender	×
Kaspersky	×
Avira Prime	×
McAfee Total Protection	×
Webroot	×
Qihoo 360	×

finally calls `NtUnmapViewOfSection` to unload the previously loaded library.

However this technique, as well as all the aforementioned existing ones, could be detected because of three main reasons:

1. `NtCreateFile` is called to open a system DLL that is not usually opened by user-space programs
2. `NtMapViewOfSection` is called to map a DLL that is already loaded in the process address space (e.g. NTDLL.dll is always loaded by the OS)
3. There is a (small) period of time in which the DLL is mapped twice in the process address space

To solve the issues mentioned above *PEzoNG* implements a new technique, not documented at the time of writing, that we called *Whisper2Shout*. This technique came out as the result of a research, on 16 different Antivirus software, whose purpose was to evaluate which AV employs API hooking, and most importantly how they implement it. We evaluated *if* and *how* each of the Antivirus provides user-space API hooking by reading the content of a number of Windows DLLs in RAM and checking, for every API, if the execution flow was hijacked towards a memory location outside of the same DLL.

Table 1 shows which of the 16 AV software—the same that are used in Sect. 4 for the final tests of *PEzoNG* as a platform—provides user-space API hooking as one of their detection mechanisms, while *how* they actually implement hooking will be explained later on in this Section.

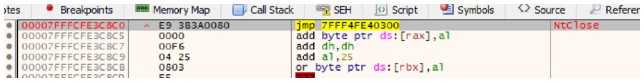

Fig. 5 NtClose hooked by BitDefender Total Security

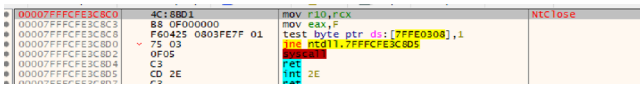

Fig. 6 NtClose unhooked by writing the original system call stub at the symbol address

The *Whisper2Shout* technique uses a number of observations to restore the prologue of hooked functions with the original bytes without the need of reading the contents of the original library.

First of all, the technique discriminates the hooked functions between system calls and higher level APIs. The system call case exploits the same property used in *Syswhispers2*: as explained in Sect. 3.3, system call numbers can be obtained by ordering by address the Zw* functions in NTDLL.dll even if the user-space system call stub has been hooked. Once the correct system call number is obtained, if a system call stub is hooked, restoring the original bytes is trivial as the stub used to call a system call is well-known; thus, it is possible to unhook any system call stub hooked in NTDLL by overwriting the instructions with the system call stub using the right system call number.

The following is an example of how a system call stub looks like. We omitted some instructions between the number of the system call and the `syscall` instruction as they are not important for the purpose of the example.

Listing 2 NtClose system call stub on Windows 10 1909

```
mov r10, rcx
mov eax, F
...
syscall
```

During our research on the 16 different Antivirus software —the same AVs listed in Table 1 and in Sect. 4—we found that the methods used to hook a system call were 2: (i) a 5-byte `jmp` instruction and (ii) a 7-byte sequence of instructions `mov eax, N; jmp rax;` Since the first instructions of a syscall stub `mov r10, rcx; mov eax, system_call_number` are always 8 bytes long, the knowledge of the system call number is enough to reconstruct the correct stub. Figure 5 shows the system call stub of `NtClose` hooked by *BitDefender Total Security* [21] and Fig. 6 shows the reconstructed stub after the unhooking process.

The API case is less trivial because there are a number of different techniques that could be used by AV/EDRs to hook a Windows API.

We analysed previous research on the topic [41] as well as the aforementioned security software to understand which

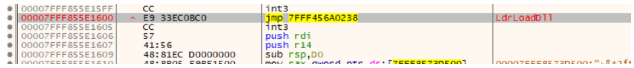

Fig. 7 NTDLL.LdrLoadDll hooked by AVG using inline hooking

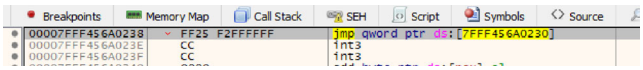

Fig. 8 Trampoline for jumping to AVG Dll

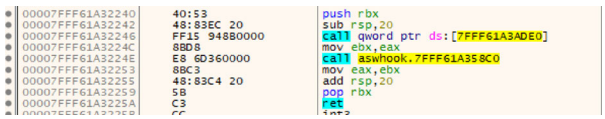

Fig. 9 AV Checker function in AVG Dll

techniques were used and we developed a general unhooking technique working for each hooking method we found.

In particular, we recall that the jump to the trampoline stub (which is allocated and written by the AV dll at runtime) can be done in the two aforementioned ways, i.e. short jump or a `mov eax, N; jmp rax;` sequence. It should be noted that those are not the only possible ways to hook a function, however, during our research we found that in practice only these two techniques are used.

After jumping to the AV controlled area, there must be a way to jump back to the original function. Since now execution is in the AV controlled area, there is not any restriction on the number of instructions to use in order to restore the execution flow.

In particular, we identified the following two unique techniques to execute back the original function from the hook:

1. Jump back to the original function with a `jmp` instruction (implemented by `Detours` hooking library [42])
2. Double-Push technique [41]

During our research we identified a common pattern with regards to the memory allocated to storing pointers and trampolines needed for hooking. We found that the memory type of all the regions containing useful information regarding the hooks was marked as *Private* (namely `MEMORY_BASIC_INFORMATION.Type == MEM_PRIVATE`) [43].

The previous observation is the fundamental block of this unhooking technique because that private memory region will contain all the information necessary for the unhooking process. Figures 7, 8, 9, 10 shows the blocks used by *AVG Internet Security* to hook the function *NTDLL.LdrLoadDll*

So when a function is hooked, the pointer to the symbol in the *Export Directory* of the DLL points to a jump instruction or to a set of well-known instructions that divert the execution to a target address located inside a *Private* memory region

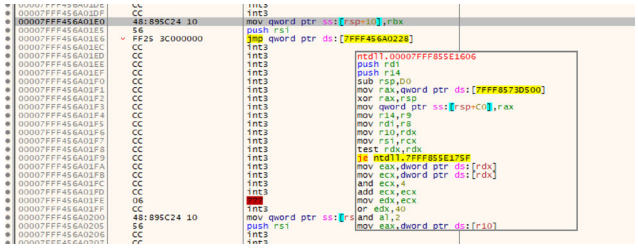

Fig. 10 Trampoline create by AVG to execute back the hooked function

Fig. 11 AVG Private memory region

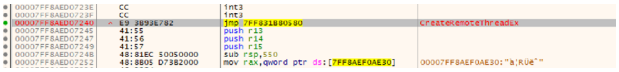

Fig. 12 Kernelbase.CreateRemoteThreadEx Hooked by BitDefender

Fig. 13 BitDefender Private memory region

(Figs. 7 and 11 shows the hook and the private memory for AVG while 12 and 13 for BitDefender).

This (private) memory region contains trampolines to the hooking dll (which will be used to hijack the execution of the function towards the anti-malware software) as well as trampolines to the hooked (original) dll (which will be used if the call has been identified as legitimate by the anti-malware software and thus the execution should continue as normal). Even when there are multiple *Private* memory regions, both trampolines reside in the same memory area. This means that by using the destination address of the jump located at the symbol address, we can call `VirtualQuery` to get the memory region where the prologue of the hooked function is stored (Figs. 14, 15, 16).

Once this memory region is identified, it is necessary to parse it, searching for the trampolines used to jump back to the original function. Each of those trampolines will contain the original prologue of a hooked function as well as a pointer to an address near the position of the hooked function—a few

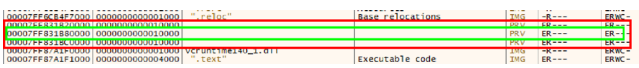

Fig. 14 BitDefender Private memory region where the trampoline of CreateRemoteThreadEx resides (Green)

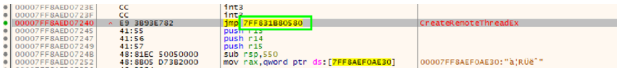

Fig. 15 CreateRemoteThreadEx Hooked by BitDefender

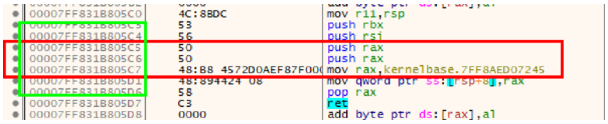

Fig. 16 BitDefender trampoline to execute back the hooked function

Fig. 17 Layout of the NtClose function before being hooked

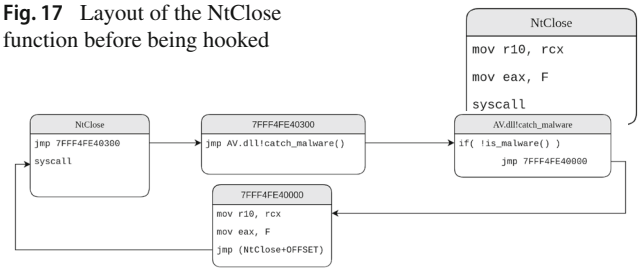

Fig. 18 Layout of the NtClose function after being hooked

Fig. 19 BitDefender Private memory region

bytes after its first instruction. Figures 17 and 18 shows the layout of the NtClose function before and after being hooked respectively.

At this point, the actions that have to be taken differ between hooking techniques as there are different ways to understand if the trampoline is pointing to the function we are trying to unhook.

When a jump is used to execute back the hooked function (first hooking technique), it is necessary to find all the jumps in that region so that we can analyze the destination address of each jump searching for the memory region where the original function is located (Figs. 11, 10). In particular, we scan the *private* memory region searching for (i) long jumps (used by the *Detours* library) with opcode 0xFF25 and (ii) short jumps (used e.g. by *MalwareBytes* [44]) with opcode 0xE9. On the other side, when the second technique (double-push) is used, it is necessary to find all the sequences of `push rax; push rax; mov rax, addr` so that the destination address could be extracted to check if it points to the hooked function (Figs. 19, 20).

Once we have identified the correct trampoline, then the bytes prepending the aforementioned stub are the original bytes that have been overwritten by the initial hook. To unhook the function, we have to copy those bytes back to the original address of the symbol.

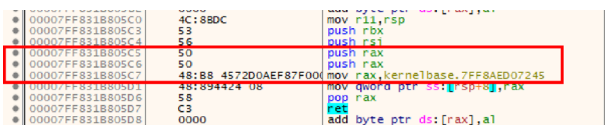

Fig. 20 BitDefender trampoline to execute back the hooked functiopn

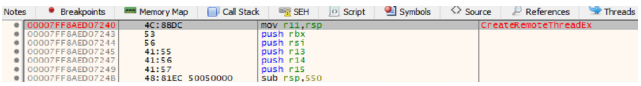

Fig. 21 CreateRemoteThreadEx UnHooked

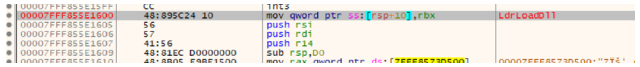

Fig. 22 LdrLoadDll unhooked

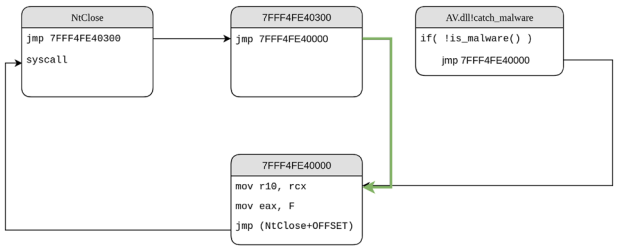

Fig. 23 Layout of the NtClose function after being unhooked.

At first, we have implemented this idea by cycling over each hooked dll and performing the following steps:

1. Use a direct system call to `NtProtectVirtual Memory` to set the memory permissions of the `.text` section to RW
2. Unhook the functions by writing each original stub at the corresponding symbol address
3. Call `NtProtectVirtualMemory` to restore the original memory permissions (RX)

The final results are shown in Fig. 21 for *CreateRemoteThreadEx* and in Fig. 22 for *LdrLoadDll*.

However, at some point we faced a problem with security products which were using a more advanced method and were monitoring the integrity of their hooks which would undo our modifications.

To solve this problem, we changed our approach and decided that rather than overwriting the hooks at the symbol address, we could overwrite the AV hooking trampoline with a jump to the original prologue function (located in the same Private memory area).

In this way, when the AV checks for its hooks it will find all the initial jumps unmodified. Those jumps will also point to the very same addresses the AV placed the hooking trampolines in, however the instructions there will not divert the execution to the AV Dll anymore. We are basically bypassing the hooking trampoline so that when the function is called, the execution will behave exactly like no hooks were in place even if a jump is placed at the symbol address. Figure 23 shows the layout of the NtClose function after being unhooked in this way.

It should be noted that all the previous observations are still valid and they allow us to retrieve all the original stubs by walking the process address space in a clever way.

We have all the information that is necessary to restore the original execution path:

- We know the destination address of each jump located at the symbol address
- We know where the original function stub is located

After collecting all this information, we can start the unhooking process:

- Use a direct system call to NtProtectVirtualMemory to set the protection of the memory area that stores the stub to RW
- Add a short jump instruction opcode: 0xe9 to jump to the original prologue
- Set back the memory to RX using another direct system call to NtProtectVirtualMemory

Finally, it is worth mentioning that, using this technique, it is no longer necessary to differentiate between system calls and APIs - although the knowledge of the system call stub can be used as a verification to understand if a function has been hooked/unhooked correctly—and therefore the unhooking process will be exactly the same, namely:

- Check if the function has been hooked
- Get the pointers to "hooking" and "original" stubs
- Overwrite the "hooking" stub with a jump to the "original" stub.

3.5 Main loader

Once the payload has been decrypted, the execution enters in the loading phase. Here a distinction must be made between shellcode and PE payloads: in both cases memory can be allocated using either the classic `VirtualAlloc`—`VirtualProtect` scheme (possibly by calling the corresponding system calls) or our modified dll hollowing scheme (default behavior) and then, in the first case execution goes directly to the shellcode while in the second case, the control is given to the custom PE loader before the actual payload is executed.

As said, the memory is allocated by default using as a basic principle that of dll hollowing: we were strongly inspired by the idea of *Phantom Dll Hollowing* [45] and we introduced some modifications to overcome its limitations.

Phantom Dll Hollowing looks for a dll on disk that has not already been loaded into memory and that is large enough to host the malicious payload. Once a feasible Dll is found, the loader opens the Dll using a *transacted NTFS* (*TxF*) [46] and maps it to memory using `NTDLL.DLL!NtCreateSection` and `NTDLL.DLL!NtMapViewOfSection` thus obtaining a memory area allocated in the same way as all the dll libraries. At this point the sections of the mapped dll in memory is overwritten with the bytes of the malicious PE exploiting the properties of *NTFS Transacted* filesystem which allows to have a copy of the section completely isolated from external applications.

In particular, *TxF* can be used to "preserve the integrity of data on disk caused by unexpected error conditions and help resolve concurrent file-system user scenarios by isolating your changes from others while the changes are being made." [46] To optimize memory usage, Windows shares mapped views of image sections created from Dlls (e.g. only one copy of `kernel32.dll` actually resides in physical memory); if the mapped view of a shared section is modified, the modified copy of the shared section is stored within the process address space. Without the usage of *TxF*, this region is marked as Private and this artifact can be used by defenders as a warning of malicious behaviour.

Phantom DLL Hollowing uses *TxF* to edit mapped views before they are actually mapped in the process address space, without having the modified sections marked as Private. The function `NTDLL.DLL!NtCreateSection` is called to load a Microsoft signed library from disk using the flag `SEC_IMAGE`: when this flag is used, the initial permissions parameter is ignored resulting in an initial allocation of RWXC. The resulting section can be mapped into memory using `NTDLL.DLL!NtMapViewOfSection` which allows to use transacted file handles as input. Since it is possible to modify the view of the loaded Dll by calling `WriteFile` using the transacted handle, when calling `NTDLL.DLL!NtMapViewOfSection`, the process will have a modified view of the library but the file object underlying the mapped image will point to the original unmodified Mifrosoft library.

However, this technique has a strong prerequisite: the file must be opened with Write access otherwise, the call to `WriteFile` will fail. Even if, as suggested by the author, this issue can be easily solved by copying the Dll in a directory where the attacker has write privileges, this is not ideal and thus we tried to overcome this limitation avoiding the use of *TxF*.

PEzoNG memory allocation removes the prerequisite of having write access to the target Dll by using the concept of *Module Overloading* [14,15]. Our approach uses `NTDLL.DLL!NtCreateSection` and `NTDLL.DLL!NtMapViewOfSection` to allocate memory but instead of using `WriteFile` on a phantom file handle to overwrite the dll with the malicious payload, it uses the handle of the mapped memory, changing the sections' memory protection according to the headers of the injected PE by using `VirtualProtect` (or `NtProtectVirtualMemory` if

Fig. 24 Memory allocated with VirtualAlloc

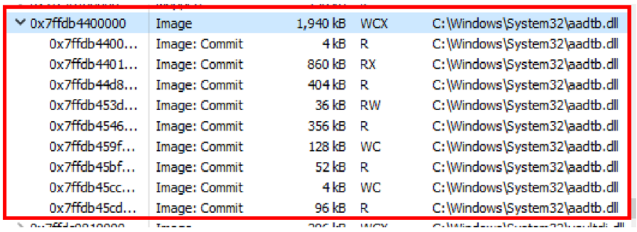

Fig. 25 Memory allocated with DLL Hollowing

PEzoNG was compiled with syscall support). This technique allows to open the sacrificial Dll file with *READONLY* access, thus removing the write access constraint but maintaining the `Image` flag for mapped memory (as opposed to `Private`). Moreover, this approach is different from the classic implementation of Dll Hollow which instead rely on `LoadLibrary` API to load the sacrificial Dll. To summarize, the following are the steps used to allocate memory for the injected payload:

- open a sacrificial DLL with `READONLY` flag using `CreateFileW` API.
- call `NtCreateSection` with `SEC_IMAGE` and `READONLY` flags using the handle of the file opened in the previous step
- call `NtMapViewOfSection` with `READWRITE` flag to allow overwriting of the sections
- return the pointer to the mapped section

Finally, it is necessary to add the module to the PEB's list of loaded modules so as to avoid having a mismatch between loaded libraries and mapped images that could be used by AVs as an indicator of compromise (IOC).

Once the loader has the pointer to the mapped section, it overwrites the memory with the payload to be injected and the sections are marked with the appropriate permissions.

It should be noted that, after this operation, the content of the overloaded dll in ram and on disk is different. One way to identify the injection is to compare the content on disk with the content in ram for each dll loaded by the process. In case this allocation fails or the user has explicitly decided to not use the *dll hollowing* scheme, the classic allocation scheme with `VirtualAlloc` and `VirtualProtect` is used by either using the API submodule (Sect. 3.2) or by using the system call submodule (Sect. 3.3). It should be noted that this allocation method leaves known artifact in memory, allowing a simpler detection by checking the attributes of the allocated memory. In particular, allocating memory using `VirtualAlloc` (or using the corresponding system call `NtAllocateVirtualMemory`) will cause the allocated memory region to be flagged as `Private` memory with the state field of `MEMORY_BASIC_INFORMATION` set to `MEM_COMMIT`. (Fig. 24)

On the other side, using the *dll hollowing* technique, the allocated memory is flagged as `Image` (with the state field of `MEMORY_BASIC_INFORMATION` set to `MEM_COMMIT`) making it indistinguishable from memory allocated by the system to load dll libraries (Fig. 25). Moreover, as previously mentioned, the dll on disk remains unchanged, and only its run-time version in memory is different from the original.

After allocating memory, the custom loader intervenes to replicate the behavior of the operating system when running a PE. Our loader has a wide support as it manages most of the features present in a PE except some niche cases; examples of these features are (i) Imports, (ii) Relocations, (iii) TLS callbacks, (iv) PE resource management and (v) Exception handlers.

At this point the PEB of the current process (i.e. *PEzoNG* loader) is modified to change the entry-point and the base address to those required by the target PE in order to hide information about the loader in memory. This also has the advantage of allowing the use of resources in the target PE (that otherwise could not be used). Finally, the memory area where the loader's PE header is located is cleaned up and execution control is given to the entry-point of the target PE.

An important feature of the loader is about how function resolving is handled. While a PE is being loaded, its *Import Address Table* (*IAT*) [3] must be filled with the addresses of the imported function names. Microsoft provides two APIs to resolve a function name: `LoadLibrary` and `GetProcAddress`; the first one is used to load a Dll into the process address space, while the second one is an API used to find the address of a function inside a given Dll; these two APIs can be used to fill the *IAT* of the target PE, however, they are also frequently employed by malware for malicious tasks. For this reason, *PEzoNG* features a custom function resolving mechanism based on the Windows loader information which reside in the *PEB* structure [27]: a function belonging to a Dll already mapped into the process address space can be resolved by parsing a linked list inside the *PEB*; on the other side, if a Dll is not already mapped, `LoadLibrary` must be called: the PE loader calls `LoadLibrary` by searching its address in `Kernel32.dll` which is referenced by *PEB*. Manually scraping imported Dlls searching for `LoadLibrary` allows to hide the function from the *Import Address Table* (*IAT*)

of the packed executable since the function is dynamically resolved. Notably, most of the Dll libraries needed by a PE are already loaded in most cases, and as such `LoadLibrary` invokation are very rare. In the case of forwarded exports this method results in an infinite loop if the function to be imported is part of the *ApiSet Map* [47], so parsing of the *ApiSetMap* has been added as well.

Finally, the loader also has support for loading *.NET* executables from memory. Before starting the loading process, both AMSI and ETW are disabled by patching the functions `AmsiScanBuffer` and `EtwEventWrite` respectively as documented by [48,49]. The loading process is then started by loading the Common Language Runtime (*CLR*) in the process, then the *.NET* PE is loaded in memory by passing the assembly bytes to the `Load_3` function defined in the *CLR*. Finally, the assembly is executed by calling the `Invoke_3` function defined in the *CLR* as well.

4 Experimental results

4.1 Sandbox timing

The first experimental step was focused on identifying the average time used by AV sandboxes to analyze the payload so that we could tune the computation done when *PEzoNG* starts to evade sandboxes. In particular, Windows Defender was taken as reference for sandbox evasion since the experiments showed that the same values could be successfully used on other AV vendors. The experiments identified that the sandbox was successfully evaded if the computation lasted for more than about 13 seconds. It should be noted that these experiments have to take into account the time (namely, the milliseconds of useless work needed to successfully evade) and not the computation needed (namely, the number of iterations) because the latter is strongly dependent on the computational power of the system and thus, different systems will result in a different number of iterations needed.

4.2 Testing methodology

A Web Application was developed in order to automate the packing process: all *PEzoNG* modules can be enabled/disabled with ease and configured. A bash script is also available with the same capabilities. Figure 26 shows the Web Interface of *PEzoNG*.

We tested *PEzoNG* running packed known malware on Windows 10 protected by the following 16 AV software.

- Windows Defender [50]
- BitDefender [21]
- Kaspersky [51]
- ESET [52]

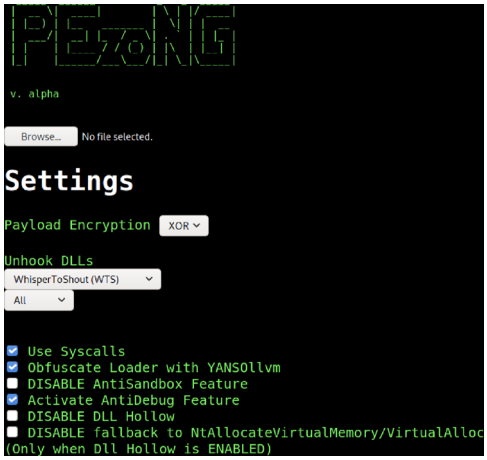

Fig. 26 *PEzoNG* Web Application

- Norton 360 [53]
- Avast [54]
- MalwareBytes [44]
- AVG Internet Security [32]
- Sophos Home 3.0 [55]
- McAfee Total Protection [56]
- Webroot [57]
- Avira Prime [58]
- Qihoo 360 Total Security Business [59]
- Comodo [60]
- Trend Micro [61]
- Dr. Web [62]

The testing environment consists of Windows 10 virtual machines, each of them provisioned with one Antivirus software. All the AVs were configured such that all the available security features and hardening mechanisms were enabled—which by default is true on most of the AV but not all of them. We say that a test is successful on a virtual machine if i) the AV does not statically detect the packed binary, ii) we are able to execute the packed binary and iii) the behavior is the same of the original payload. It's worth to note that i) and ii) are not sufficient for a test to be successful because a payload can be detected and its process killed by the Antivirus at some point during the execution. Furthermore, we used antiscan.me service [63] to test the packed binaries over 26 AV engines; all the aforementioned 16 AV software are included in this service except for Qihoo 360, Norton 360 and Trend Micro, thus increasing the total number of tested Antivirus software to 29.

The list of all the tested payloads includes tools employed during real Penetration Tests and Red Team engagements i.e. *Cobalt Strike beacons* [33], *Mimikatz* [64], *Meterpreter* [22] implant, *UACme* [65], *Rubeus* [66], *SharpHound* [67], *SeatBelt* [68], Netcat [69], and other reverse shell custom payloads. All the AV software used for the experiments

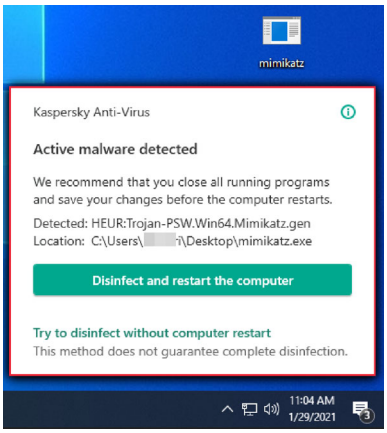

Fig. 27 Mimikatz detected by Kaspersky

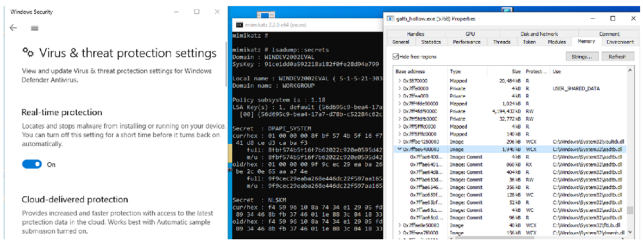

Fig. 28 Mimikatz packed with PEzoNG executed with Windows Defender enabled

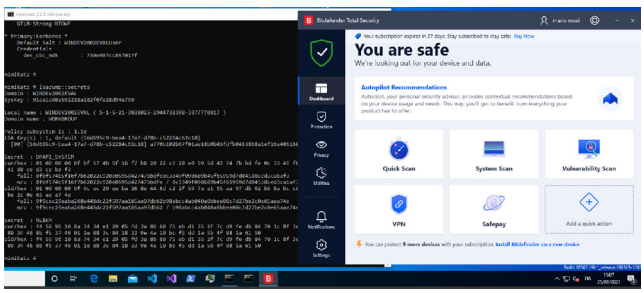

Fig. 29 Mimikatz packed with PEzoNG executed with BitDefender enabled

flagged the binaries as malicious, and as an example Fig. 27 shows how *Kaspersky* detects the *Mimikatz* executable.

4.3 Results

After packing our payloads with *PEzoNG* we were able to successfully execute the payload on the Windows 10 machines protected by the aforementioned 16 AV software; as an example, Figs. 28, 29, 30, 31 and 32 show the results of executing the post-exploitation tool *Mimikatz* against Windows Defender, BitDefender, Ka-spersky, ESET and Norton 360 respectively. The detection engine of all the aforementioned AV software was not able to detect our payloads thus proving the effectiveness of *PEzoNG*.

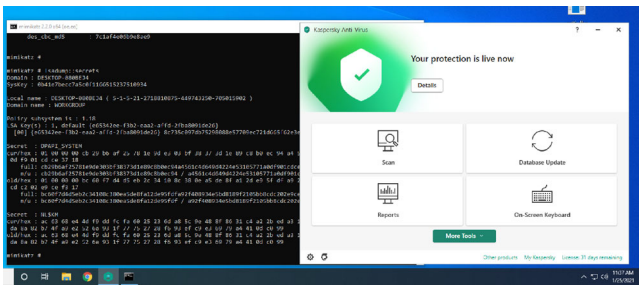

Fig. 30 Mimikatz packed with PEzoNG executed with Kaspersky enabled

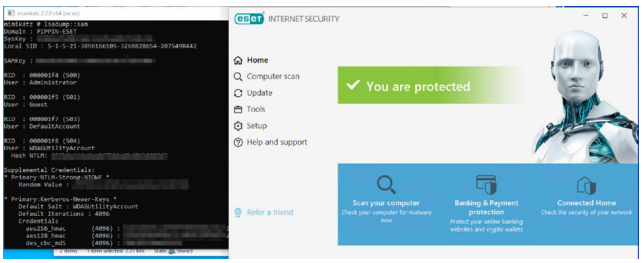

Fig. 31 Mimikatz packed with PEzoNG executed with ESET enabled

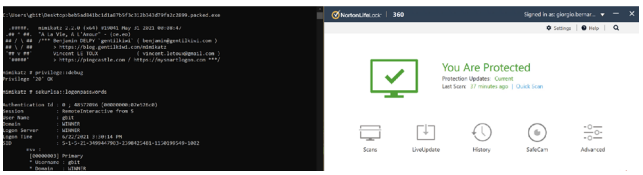

Fig. 32 Mimikatz packed with PEzoNG executed with Norton 360 enabled

Furthermore, we used antiscan.me service [63] to test the packed binaries over 26 AV engines resulting in 0/26 detection rate (Figs. 33 and 34 shows the results for mimikatz and a meterpreter payload respectively); as previously mentioned, all the manually tested 16 AV software are included in this service except for Qihoo 360, Norton 360 and Trend Micro, thus increasing the global detection rate to 0/29. In addition, we were able to test *PEzoNG* against 2 business EDR solutions, i.e. Cyber Reason [70] and Microsoft Defender Endpoint [71], and the tests were successful; in this case the environment was provided by a third party and we didn't have access to the configuration.

Table 2 shows the detection rate of a meterpreter payload against 6 of the 16 Antivirus software where only some of the features of *PEzoNG* were enabled. In the table, every column specifies that the corresponding feature is enabled, along with all the previous columns. That is, the "Syscall" column shows the results when both the *Syscall* and the *Encryption* modules are enabled. The "✓" symbol means that the Antivirus was able to detect the payload whereas the "×" symbol means that the payload was not detected and therefore it was executed. It should be noted that Table 2 shows one use case of *PEzoNG*

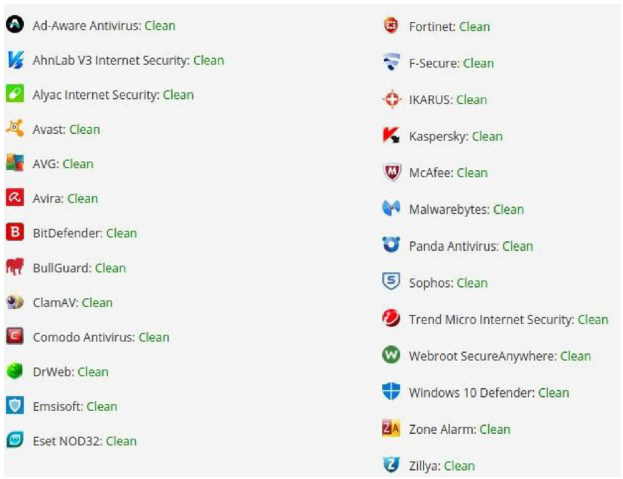

Fig. 33 Antiscan result for mimikatz packed with *PEzoNG*

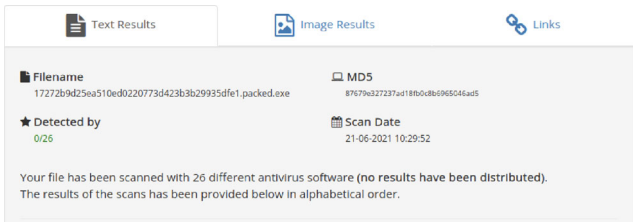

Fig. 34 Antiscan result for meterpreter reverse shell packed with *PEzoNG*

were features are enabled incrementally in order to try to bypass Antivirus software; the order in which the features are enabled can be changed and it's worth mentioning that one Antivirus could be theoretically bypassed by enabling less features.

4.4 Entropy analysis

A number of Antivirus software use entropy as a first indicator to determine if an executable file is malicious or not. In particular, "malware authors also tend to rely heavily on packing, compression, and encryption to obfuscate their tools on order to evade signature based detection systems"

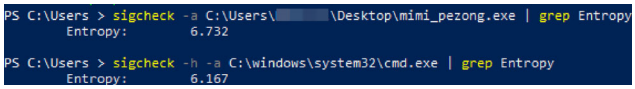

Fig. 35 Entropy comparison between the packed version of *Mimikatz* and the Windows Command Prompt

[72] thus leading to an increase of entropy. We computed the binary entropy of the packed executable files and compared them with a legit PE, i.e. the Windows Command Prompt (cmd.exe). Figure 35 shows the comparison between the entropy—computed with *SigCheck* [73]—of *Mimikatz* packed with *PEzoNG* and cmd.exe, which are 6.732 and 6.167 respectively, while malicious packed executables usually have entropy greater than 7.2 [72].

In our scenario, both the *PEzoNG* loader and *YansoLLVM* help increase the total number of instructions, so as long as the embedded payload size is limited compared to the entire packed PE, the entropy of the packed PE is reduced. Because usually the code of our packer is much larger than the embedded payload, even if the malicious payload is encrypted, it will not affect the final entropy of the binary file. Conversely, if the size of the payload we want to pack is comparable to the size of *PEzoNG*, then high entropy can be detected in the final PE. It is worth noting here that even in this case, other techniques can be applied to obtain entropy reduction, for example, we attach another legit PE to the packed PE.

4.5 Comparison with PEzor

The same payloads were packed with *PEzor* and checked against the same AV software. As an example, we show that the detection rate of *Mimikatz* is 5/26 (Fig. 36): the malicious payload was packed with all the evasion features enabled, i.e. unhooking, syscalls, antidebug, payload encoding with SGN.

The entropy comparison result is shown in Figs. 35 and 37: it is clear that the binary packed with *PEzor* contains an encrypted payload because the entropy is very high, and in particular greater than the threshold value of 7.2, while

Table 2 Detection rate of meterpreter payload (*windows/meterpreter/reverse_tcp*) with different features of *PEzoNG* enabled

Antivirus	Raw	Encryption	Syscall	Dll Hollowing	Anti-sandbox	Unhooking	All(=+Obfuscation)
Defender	✓	×	×	×	×	×	×
AVG	✓	×	×	×	×	×	×
BitDefender	✓	✓	✓	✓	×	×	×
MalwareBytes	✓	✓	✓	✓	×	×	×
Norton 360	✓	✓	✓	✓	✓	×	×
ESET Int. Sec.	✓	✓	✓	✓	×	×	×
McAfee	✓	✓	✓	✓	×	×	×

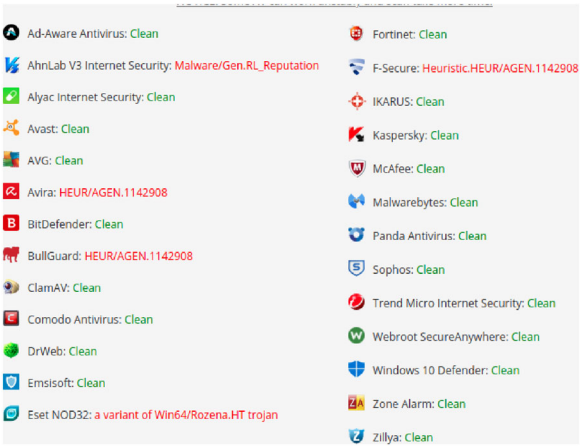

Fig. 36 Detection rate of *Mimikatz* packed with PEzor

```
PS C:\Users > sigcheck -a C:\temp\mimi_pezor.exe | grep Entropy
        Entropy:        7.996
```

Fig. 37 Entropy of *Mimikatz* packed with PEzor

the same binary packed with *PEzoNG* has an entropy more similar to cmd.exe.

5 Conclusion

The results we achieved in this paper demonstrate that it is indeed feasible to automate the process of payload obfuscation and Antivirus evasion, as commonly used Anti-virus software fail in detecting payloads embedded into *PEzoNG*.

It would be possible however to build detection strategies by actively analysing the system: if Dll Hollow is used to store the payload, scraping the RAM memory and comparing the content of the sacrificial Dll with the content of the Dll on disk may be used as an indicator that the library was overwritten. In particular, besides the actual data being different, in the general case, PE sections in RAM won't match the sections on disk (e.g. size, permissions); while this indicator might lead to false positives (i.e. dotnet binaries are used to change the memory layout while running), it may be used as a red flag to trigger further analysis. Moreover, if the malicious payload is a PE, PE headers might be different from the headers stored on disk when *PEzoNG* is configured to overwrite the original PE headers (i.e. when the payload needs support for resources). Hasherezade's hollows_hunter [74] and Volatility Hollowfind [75] plugins can be used though they generates many false positives [76]; in addition, the memory allocation scheme used in PEzoNG provide more stealthiness than standard Process Hollowing Injection techniques, which can be quickly identified by defenders [77]. If defenders have the choice of running code in kernel space (i.e. installing a kernel driver), it would be possible to catch the event of an image (e.g. Dll) loaded into a process (e.g. `NtMapViewOfSection`, `LoadLibrary`, `LoadLibraryEx`) by registering a callback using the API `PsSetLoadImageNotifyRoutine` [78] in a minifilter driver.

As regards the unhooking technique *Whisper2Shout*, from a defender's perspective, user-space hooking is a very important mechanism, and even though bypasses are possible it is important to have it in place following a defense in depth approach. Moreover, security products that monitor the integrity of the hooks should be preferred as they make attacker's life harder increasing the likelihood of detection. In particular, since the malicious payload have full control over its own memory address space, the only way to detect hooking removal is monitoring the hooked dlls and the hooking stubs searching for changes in the instructions stored in those memory areas.

For what concerns the embedded payload, even if *PEzoNG* mitigates the presence of user space hooks and provides an almost completely safe environment to execute malicious payloads to the eyes of AV/EDRs, those payloads can still raise different alarms if other detecting techniques are employed—i.e. network traffic analysis—by a firewall, for example, for which *PEzoNG* provides no protection.

Finally, it is important to note that these Antivirus products are not bullet-proof solutions that will protect systems from every possible threat, they are tools that defenders can use to identify anomalies in the monitored systems. Setting and tuning a security software are fundamental steps when a new AV is placed in the network: being able to receive meaningful alerts would help defenders to detect and react to stealth attacks that are not automatically detected as malicious but looks suspicious.

Author Contributions GB and DDC are joint first two authors.

Funding CNIT and SECFORCE LTD.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. NIST, Penetration testing. https://csrc.nist.gov/glossary/term/penetration_testing. Accessed Dec 2021
2. NIST, Red team. https://csrc.nist.gov/glossary/term/red_team. Accessed Dec 2021
3. Microsoft, Portable executable format. https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. Accessed May 2021
4. Phra, P.: https://github.com/phra/PEzor. Accessed May 2021
5. NIST, Blue team. https://csrc.nist.gov/glossary/term/blue_team. Accessed Dec 2021

6. TheWover, D.: https://github.com/TheWover/donut. Accessed May 2021
7. @spotheplanet, Module stomping for shellcode injection. https://www.ired.team/offensive-security/code-injection-process-injection/modulestomping-dll-hollowing-shellcode-injection. Accessed Dec 2021
8. EgeBalci, Shikata ga nai encoder. https://github.com/EgeBalci/sgn. Accessed May 2021
9. JustasMasiulis, Windows inline syscalls. https://github.com/JustasMasiulis/inline_syscall. Accessed May 2021
10. SecRat, Api hooking. https://resources.infosecinstitute.com/topic/api-hooking/. Accessed Dec 2021
11. Cylance Vulnerability Research Team, Reflectivedll-refresher. https://github.com/CylanceVulnResearch/ReflectiveDLLRefresher. Accessed May 2021
12. Broumels, T., Ubink, S.: Antivirus evasion by user mode unhooking on windows 10. https://rp.os3.nl/2020-2021/p68/report.pdf (2021). Accessed Dec 2021
13. Syswhispers2, J.T.: https://github.com/jthuraisamy/SysWhispers2. Accessed May 2021
14. TheWover, Module overloading. https://twitter.com/TheRealWover/status/1193284444687392768?s=20. Accessed Jan 2021
15. Hasherezade, Module overloading. https://github.com/hasherezade/module_overloading. Accessed Jan 2021
16. Borello, J.-M., Mé, L,: Code obfuscation techniques for metamorphic viruses. J. Comput. Virol. **4**(3), 211–220 (2008). https://doi.org/10.1007/s11416-008-0084-2
17. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. J. Comput. Virol. **7**, 201–214 (2011)
18. Emc2314, Yansollvm. https://github.com/emc2314/YANSOllvm. Accessed Jan 2021
19. Tamboli, T., Austin, T.H., Stamp, M.: Metamorphic code generation from llvm bytecode. J. Comput. Virol. Hacking Techn. **10**(3), 177–187 (2014). https://doi.org/10.1007/s11416-013-0194-3
20. Ahmed, A., Garba, F., Abba, A.: Evaluating antivirus evasion tools against bitdefender antivirus 10 (2021)
21. Bitdefender, Bitdefender av. https://www.bitdefender.com/. Accessed June 2021
22. Rapid7, Meterpreter. https://www.offensive-security.com/metasploit-unleashed/about-meterpreter/. Accessed Dec 2021
23. Kalogranis, C.: Antivirus software evasion: an evaluation of the av evasion tools. https://dione.lib.unipi.gr/xmlui/handle/unipi/11232 (2018)
24. Mingw, Mingw-w64. http://mingw-w64.org/doku.php. Accessed Jan 2021
25. LLVM Foundation, Llvm. https://llvm.org/. Accessed Jan 2021
26. Sahita, R., Li, X., Lu, L., Deng, L., Shepsen, A., Xu, X., Huang, L., Liu, H., Huang, K.: Executing full logical paths for malware detection, 2016, uS Patent No. US10210331B2. [Online]. Available: https://patents.google.com/patent/US10210331B2/en
27. Microsoft, Peb structure. https://docs.microsoft.com/en-us/windows/win32/api/winternl/ns-winternl-peb. Accessed Jan 2021
28. R0-crew, Kaspersky hooking engine analysis. https://forum.reverse4you.org/t/kaspersky-hooking-engine-analysis/543. Accessed May 2021
29. Crummie5, Freshycalls. https://www.crummie5.club/freshycalls/. Accessed Jan 2021
30. MDSec Research, Bypassing user-mode hooks and direct invocation of system calls for red teams. https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/. Accessed June 2021
31. Nasi, E.: Bypass antivirus dynamic analysis. https://blog.sevagas.com/IMG/pdf/BypassAVDynamics.pdf. Accessed Jan 2021
32. AVG, Avg internet security. https://www.avg.com/en-us/internet-security#pc. Accessed June 2021
33. HelpSystems, Cobalt strike beacon. https://www.cobaltstrike.com/features/. Accessed Dec 2021
34. Mosch, F.: A tale of edr bypass methods. https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/. Accessed May 2021
35. Tang, J.: Universal unhooking: Blinding security software. https://blogs.blackberry.com/en/2017/02/universal-unhooking-blinding-security-software. Accessed May 2021
36. Bui, H.: Bypass edr's memory protection, introduction to hooking. https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6. Accessed May 2021
37. Sektor7, Perun's fart - yet another unhooking method. https://blog.sektor7.net/#!res/2021/perunsfart.md. Accessed May 2021
38. Microsoft, Writing preoperation and postoperation callback routines. https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/writing-preoperation-and-postoperation-callback-routines. Accessed June 2021
39. RedBluePurple, Detecting process injection with etw. https://blog.redbluepurple.io/windows-security-research/kernel-tracing-injection-detection. Accessed June 2021
40. Slaeryan, Shellycoat. https://github.com/slaeryan/AQUARMOURY/tree/master/Shellycoat. Accessed May 2021
41. Bitton, T., Yavo, U.: Captain hook: Pirating avs to bypass exploit mitigations, 2016, blackHat USA. https://www.blackhat.com/us-16/briefings/schedule/#captain-hook-pirating-avs-to-bypass-exploit-mitigations-4057
42. Microsoft, Detours. https://github.com/microsoft/Detours. Accessed May 2021
43. Microsoft, Winnt memory basic information. https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-memory_basic_information. Accessed June 2021
44. MalwareBytes, Malwarebytes anti-malware. https://www.malwarebytes.com/. Accessed June 2021
45. Orr, F.: Phantom dll hollowing. https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing. Accessed Jan 2021
46. Microsoft, Transactional ntfs (txf). https://docs.microsoft.com/en-us/windows/win32/fileio/transactional-ntfs-portal. Accessed Jan 2021
47. Cysinfo, Runtime dll name resolution: Apisetschema part i. https://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-i.html. Accessed Jan 2021
48. Chung, W.-J.: Hunting for amsi bypasses. https://blog.f-secure.com/hunting-for-amsi-bypasses/. Accessed May 2021
49. Chester, A.: Hiding your .net - etw. https://blog.xpnsec.com/hiding-your-dotnet-etw/. Accessed May 2021
50. Microsoft, Microsoft defender av. https://docs.microsoft.com/en-gb/windows/security/threat-protection/microsoft-defender-antivirus/microsoft-defender-antivirus-in-windows-10. Accessed June 2021
51. Kaspersky, Kaspersky av. https://www.kaspersky.co.uk/. Accessed June 2021
52. ESET, Eset av. https://www.eset.com/. Accessed June 2021
53. Norton, Norton 360. https://us.norton.com/360. Accessed June 2021
54. Avast, Avast av. https://www.avast.com/. Accessed June 2021
55. Sophos, Sophos home. https://home.sophos.com/en-us.aspx. Accessed June 2021
56. McAfee, Mcafee total protection. https://www.mcafee.com/en-us/antivirus/mcafee-total-protection.html. Accessed June 2021
57. Webroot, Webroot internet security. https://www.webroot.com/. Accessed June 2021
58. Avira, Avira prime. https://www.avira.com/it/prime. Accessed June 2021

59. Qihoo, 360 total security business. https://www.360totalsecurity.com/it/business/. Accessed June 2021
60. Comodo, Comodo internet security. https://www.comodo.com/home/internet-security/antivirus.php. Accessed June 2021
61. Trend Micro, Trend micro antivirus. https://www.trendmicro.com. Accessed June 2021
62. Dr. Web, Dr. web antivirus. https://www.drweb.com/. Accessed June 2021
63. Antiscan.me service. https://antiscan.me/. Accessed June 2021
64. Gentilkiwi, Mimikatz. https://github.com/gentilkiwi/mimikatz. Accessed June 2021
65. hfiref0x, Uacme. https://github.com/hfiref0x/UACME. Accessed Dec 2021
66. GhostPack, Rubeus. https://github.com/GhostPack/Rubeus. Accessed Dec 2021
67. BloodHoundAD, Sharphound. https://github.com/BloodHoundAD/SharpHound3. Accessed Dec 2021
68. GhostPack, Seatbelt. https://github.com/GhostPack/Seatbelt. Accessed Dec 2021
69. Nmap.org, Ncat. https://nmap.org/ncat/. Accessed Dec 2021
70. CyberReason, Cyberreason edr. https://www.cybereason.com/. Accessed Dec 2021
71. Microsoft, Microsoft defender endpoint. https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-endpoint?view=o365-worldwide. Accessed Dec 2021
72. Lester, M.: Threat hunting with file entropy. https://practicalsecurityanalytics.com/file-entropy/. Accessed Dec 2021
73. Microsoft, Sigcheck. https://docs.microsoft.com/en-us/sysinternals/downloads/sigcheck. Accessed Dec 2021
74. Hasherezade, hollows_hunter. https://github.com/hasherezade/hollows_hunter. Accessed June 2021
75. monnappa22, Hollowfind. https://github.com/monnappa22/HollowFind. Accessed June 2021
76. Balaoura, S.: Process injection techniques and detection using the volatility framework. https://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/11578/Balaoura_MTE1623.pdf (2018)
77. Cysinfo, Detecting deceptive process hollowing techniques using hollowfind volatility plugin. https://cysinfo.com/detecting-deceptive-hollowing-techniques/. Accessed May 2021
78. Microsoft, Pssetloadimagenotifyroutine function. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/ntddk/nf-ntddk-pssetloadimagenotifyroutine. Accessed Dec 2021