Windows Antivirus Evasion Techniques: How to Stay Ahead of the Hooks

Giorgio Bernardinetti[1,2,*,†], Dimitri Di Cristofaro[3,†] and Giuseppe Bianchi[1,2]

¹CNIT Natl. Network Assessment Assurance and Monitoring Lab, Rome, IT

²University of Rome "Tor Vergata", Rome, IT

³SECFORCE LTD, London, U.K.

Abstract

The practice of API hooking in user-space is a common technique used by antivirus (AV) and endpoint detection and response (EDR) software to monitor and control software execution on Windows systems. This method of detection allows for the interception and examination of interactions between processes and operating system services, making it a potential target for both simulated penetration testing and malicious attacks. After an extensive analysis on how commercial antivirus software do implement hooking techniques, this paper introduces a new approach, named `Whisper2Shout`, which enables users to bypass API hooking in user-space. Unlike established unhooking methods, `Whisper2Shout` does not rely on any operating system service monitored by antivirus or endpoint detection and response software. We compare the advantages and disadvantages of `Whisper2Shout` with respect to similar tools and we evaluate the effectiveness of `Whisper2Shout` by circumventing hooks on 16 commercial antivirus programs and 4 endpoint detection and response software.

Keywords

windows evasion, api hooking, user-space unhooking

1. Introduction

API hooking [1] is a technique in computer programming that allows modification of the behavior of an application by intercepting its function calls to certain APIs[1]. This technique is used by a variety of software, including *Antivirus* (AV) and *Endpoint Detection and Response* (EDR) programs, to monitor the behavior of suspicious processes and detect malicious activities [2]. AV and EDR software utilize API hooking to keep track of both API and system calls made by an application, allowing them to detect any suspicious behavior that might indicate malware or other security threats. By hooking the APIs used by a process, the software can monitor the data being passed to and from the API, and take appropriate actions based on the information collected. This provides a powerful tool for security software to detect and respond to potential threats in real-time, and helps to keep systems protected from malicious actors. API hooking is

¹Application Programming Interfaces

a remarkably effective method of detection, as it allows to take actions based on real-time events that could trigger the identification of malicious software after it has started its execution.

From the perspective of an attacker, a way to evade these security products is to try to remove the hooking. There are numerous documented techniques to remove user-space hooking (refer to Section 2 for more details), though the sheer majority of them result in some *Indicator Of Compromise* (IoC) [2] that could alert the AV/EDR.

In this paper we present `Whisper2Shout`, a novel unhooking technique which does not require knowledge of the original (unhooked) library. This paper expands on our previous work [3] by looking into a more comprehensive analysis of hooking techniques implemented by AV/EDR and providing a detailed examination of the inner workings of our approach.

The idea behind our strategy is that even if an AV/EDR employs hooking in user-space, it must save all the information about the original code of the hooked APIs in memory. This is because if a process is considered not suspicious, the AV/EDR should allow transparent access to the APIs without interfering with the execution of legitimate applications. The technique focuses on identifying the memory location where this information is stored, and utilizing it to retrieve the original bytes of the hooked APIs. Moreover, these operations should be executed without alerting the AV/EDR. In summary, the contribution and novelties of this paper are:

- via an extensive analysis carried out on 16 AVs and 4 EDRs, we identify and classify hooking techniques used by real world Windows AV/EDRs;
- we devise `Whisper2Shout`, a novel unhooking technique which does not rely on any monitored OS service, and allows to selectively remove hooks on user-defined APIs;

The remainder of this paper is organized as follows: Section 2 highlights differences between `Whisper2Shout` and other existing techniques, Section 3 discusses how hooking is employed in the commercial AV/EDRs analyzed, Section 4 explains how our technique has been devised and implemented; its effectiveness is assessed in Section 5, and Section 6 concludes this paper.

2. Related Work

Since *API hooking* is a technique commonly used by security software to detect suspicious processes [2], there are various documented methods for the circumvention of user-space hooking, such as those cited in [4], [5], [6], and [7]. However, these techniques need to retrieve the unaltered library file (*Dynamic Link Library* - DLL) either by reading it from the disk or from the memory of a remote process before the security software places the hooks. The detection of such techniques is commonly facilitated by the Windows kernel via the deployment of minifilter drivers [8]. Anti-malware software can register callbacks for a variety of system events, such as file operations and process creation, through Windows [9], which notifies the AV and prompts a deeper analysis that may result in detection. For instance, reading of the `ntdll.dll` file, which should only be loaded at the time of process creation, is deemed suspicious and could trigger detection.

Shellycoat [10] is a renowned technique that un-hooks a hooked DLL by loading an unaltered version from disk. This technique employs the syscalls `NtCreateFile`, `NtCreateSection`,

and `NtMapViewOfSection` to i) load a fresh copy of the DLL in the process' address space, ii) copy the original bytes of its text section to the text section of the hooked DLL, and finally iii) call `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-reserved DLL that is not usually accessed by user-space programs
2. `NtMapViewOfSection` is called to map a DLL that is already loaded in the process address space (i.e. `ntdll.dll` is always loaded by the OS)
3. There is a (small) amount of time in which the DLL is mapped twice in the process address space

Perun's Fart [7] is another unhooking technique which does not require reading the clean DLL from disk, and its main steps are:

- Spawn a new process in suspended state.
- Read memory from the new spawned process, at which point the target DLL has not been tampered yet, so the bytes of the functions to be unhooked can be copied.
- Resume/kill the suspended process

Such a technique could trigger an IoC, i.e. reading the contents of the `ntdll.dll` file from a suspended target process, which would be considered an abnormal and suspicious operation by AVs and EDRs.

There are alternative means of covertly invoking API calls on Windows, as referenced by [11] and [12]. However, their aim is not solely to evade hooking, but rather to develop more comprehensive functionalities that can bypass antivirus software. Additionally, other strategies for monitoring APIs, such as [13] and [14], exist. Nevertheless, our study concentrates on the hooking methods utilized by commercial AV/EDR software, which are analyzed in greater depth in Section 3.

The `Whisper2Shout` technique distinguishes itself from the previously documented methods in that it does not require any syscall or API monitored in kernel-space to execute the unhooking process. In contrast, all of these techniques necessitate, at some point, the ability to read the contents of a *fresh* version of the DLL to be unhooked, whether it be by reading it from the disk or another process' memory, a task which involves the utilization of either APIs or system calls. `Whisper2Shout` only requires the capability to traverse a series of pointers and the capability to retrieve metadata about a memory region, abilities that are enabled by default for all processes on Windows. Additionally, `Whisper2Shout` is resilient against hook monitoring. The previous unhooking techniques remove the hooks from the original DLL and presume that the AV/EDR will not interfere with its memory again. However, as we will explain later in Section 4, certain AVs periodically verify if their hooks are in place and restore them if they are not. `Whisper2Shout` circumvents this mechanism by misleading the AV during its hook check. Lastly, it is worth to mention that all previously documented techniques aim to remove all hooks within a specific DLL whereas `Whisper2Shout` enables selective removal of certain hooks, a concept that will be made clearer in the following sections.
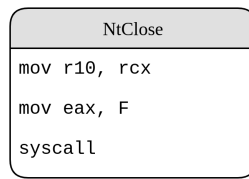
Figure 1: Layout of the `NtClose` function before being hooked.
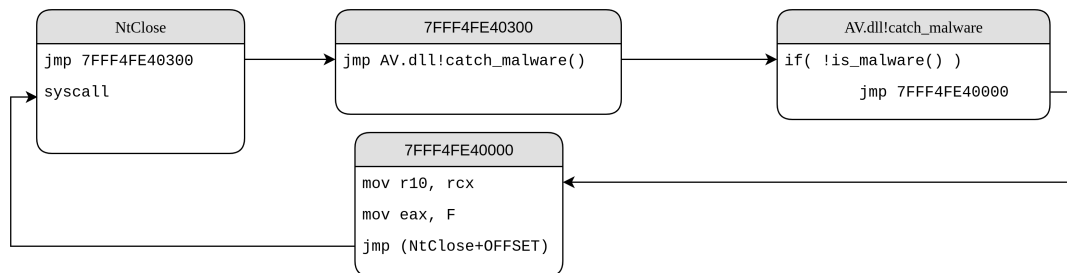
Figure 2: Layout of the `NtClose` function after being hooked.

3. Hooking

The `Whisper2Shout` method was developed through an exploration of various commercial AV/EDR programs with the aim of evaluating their utilization of API hooking and the way in which it was executed. The research analyzed the contents of several Windows DLLs stored in RAM to determine *if* and *how* the execution flow for each API was being redirected to a location outside of the same DLL.

In general, API hooking is implemented by AVs/EDRs by injecting a custom library (DLL) into the address space of a new process. The purpose of this library is to install hooks within DLLs which contain critical APIs and/or syscalls. Let's take as an example the `NtClose` API contained in `ntdll.dll` and which is a simple wrapper for a syscall on Windows. The layout of this API, and its corresponding assembly instructions, are shown in Figure 1. The insertion of the hook occurs by overwriting the initial assembly instructions with a JMP [15] instruction, redirecting the execution flow to a memory region dynamically allocated by the AV DLL and classified as *private*. This memory region acts as a *trampoline* between the hooked API and the DLL injected by the AV/EDR. As a matter of fact, this memory region often contains a JMP to the AV DLL, which contains the code to determine whether the API invocation can be deemed suspicious or not. If the invocation is not considered suspicious, the execution will proceed running the original API instructions. However, if the invocation triggers an alarm, the execution will be diverted to somewhere else, in a memory area controlled by the security product. This layout is depicted in Figure 2, which shows how the execution flow of `NtClose` is altered after the AV placed its hook.

Figure 3: NTDLL.LdrLoadDll hooked by *AVG* using inline hooking

Figure 4: Trampoline for jumping to *AVG* DLL

Figure 5: AV Checker function in *AVG* DLL

This analysis can be validated by looking at Figures 3, 4, 5 and 6, which show how *AVG*[3] implements user-space API hooking. In particular:

- Figure 3 shows how *AVG* overwrites the original bytes of the API LdrLoadDll contained in ntdll.dll.
- Figure 4 shows the content of the memory area which contains the *trampoline* to *AVG* DLL.
- Figure 5 shows the code of *AVG* DLL which implements the logic to determine whether the process should be classified as malicious or not.
- Figure 6 shows the *trampoline* containing the original instructions of LdrLoadDll, followed by a JMP to go back to ntdll.dll

In order to classify AVs and EDRs with respect to hooking, and based on the observations above, there are three main steps in which each AV/EDR can implement hooks differently from the others, i.e. there are three features that can be used to classify:

1. how the AV implements the jump to the area containing the *trampoline*
2. how the AV DLL allocates the area containing the *trampolines*
3. how the AV implements the *trampoline* back to the original API.

For the first point, we observed that only two different techniques are used by AVs/EDRs to divert execution of an API, i.e. i) a jmp instruction and ii) the sequence mov eax, N; jmp rax;. These two techniques we observed are not exhaustive, however, our research determined that these are the only techniques utilized in practice.

Figure 6: Trampoline created by *AVG* to execute back the hooked function

Figure 7: *BitDefender* trampoline to execute back the hooked function

For the second point, our research uncovered a recurring pattern with respect to memory allocation to store pointers and trampolines essential for hooking. We discovered that the memory type of all regions holding relevant information regarding hooks was designated as *Private* (namely, MEMORY_BASIC_INFORMATION.Type == MEM_PRIVATE) [16].

For the third and last point, we identified the following two distinct techniques to execute the original function from the hook: i) a jmp instruction that jumps back to the original function (employed by the Detours hooking library [17]) and ii) the double-push technique [18]. An example of hooking using the first of these two techniques is shown in Figure 6 (*AVG*); the second one is shown in Figure 7 (*Bitdefender*[4]).

Our analysis was based on 16 commercial AV products and 4 commercial EDR software. The results are shown in Table 1: for each AV/EDR we identified how they implement the three main steps of hooking i.e. i) how they jump to the trampoline area ii) how they allocate the trampoline area and iii) how they jump back to the original API.

4. Unhooking

The Whisper2Shout technique is based upon the aforementioned observations aimed at restoring the preamble of hooked functions to their original state, without the requirement of retrieving the content of the original library. To understand the techniques utilized, we

---

Table 1
Comparison of how API hooking is employed by different AV and EDR software.

Antivirus/EDR	jmp to trampoline	private area	trampoline back
AVG	`jmp`	✔	`long jmp`
Avast	`jmp`	✔	`long jmp`
BitDefender	`jmp`	✔	`double push`
Comodo	`jmp`	✔	`long jmp`
MalwareBytes	`jmp`	✔	`jmp`
ESET Internet Security	`jmp`	✔	`long jmp`
Sophos Home 3.0	`jmp`	✔	`long jmp`
Norton 360	`jmp`	✔	`long jmp`
Trend Micro	`jmp`	✔	`jmp`
Dr. Web	`mov eax, N; jmp rax;`	✔	`jmp`
Windows Defender	✘	✘	✘
Kaspersky	✘	✘	✘
Avira Prime	✘	✘	✘
McAfee Total Protection	✘	✘	✘
Webroot	✘	✘	✘
Qihoo 360	✘	✘	✘
SentinelOne *EDR*	`jmp`	✔	`long jmp`
CrowdStrike *EDR*	`jmp`	✔	`jmp`
Cortex *EDR*	✘	✘	✘
Sophos *EDR*	✘	✘	✘

analyzed prior research on the topic [18] and the AVs/EDRs of Table 1, and thus devised a general unhooking technique that can be applied to each identified hooking method. At its core, the `Whisper2Shout` approach is to trace the arrows as depicted in Figure 2, traverse the *trampoline* area, the AV DLL and the hooked DLL in order to locate the original instructions of the hooked API and ultimately restore these instructions without alerting the AV/EDR.

The observation made in the previous section about the recurring pattern with respect to memory allocation constitutes the fundamental block of our unhooking technique, as that *private* memory regions contain all the necessary information for the unhooking process. Figures 3, 4, 5, 6 illustrate the blocks utilized by *AVG Internet Security* to hook the function `NTDLL.LdrLoadDll`. When a function is hooked, the pointer to the symbol within the the *Export Directory*[5] of the DLL is diverted to either a jump instruction or a sequence of well-known instructions that redirect execution to a designated address located within a private memory region - namely, `short jmp` or `mov eax, N; jmp rax;` - as demonstrated in Figures 8 and 9 for the case of the *BitDefender* antivirus. This *private* memory region contains the trampolines both to the hooking DLL, which serves to divert the execution to the anti-malware software, and to the original, hooked DLL, which will be utilized if the call is deemed legitimate by the anti-malware software and the execution must proceed as intended. By accessing the destination address of the jump located at the symbol address (i.e. the first arrow of Figure 2),

---

[5]https://learn.microsoft.com/en-us/windows/win32/debug/pe-format

Figure 8: `CreateRemoteThreadEx` hooked by *BitDefender*

Figure 9: *BitDefender* private memory region (highlighted in red) where the trampoline of `CreateRemoteThreadEx` resides (highlighted in green)

we can call `VirtualQuery`[6] to obtain the base address of the memory region that stores the original prologue of the hooked function (Figure 8 shows the destination address of the jump, Figure 9 depicts the *private* memory region containing that address and Figure 7 shows that the *trampoline* to return back to the hooked function resides within the *private* area). It should be noted that some security solutions might use different memory spaces to store the trampoline containing the original function prologue. In the event the trampoline cannot be located in the same memory area, it is necessary to scan all the *Private* memory regions in the address space using multiple calls to `VirtualQuery`.

Once the memory region has been identified, it is necessary to examine it in search of trampolines that facilitate a return to the original function. Each trampoline will comprise the original prologue of a hooked function, as well as a pointer to an area located near to the hooked function location - just a few bytes past its first instructions. Figures 1 and 2 display the structure of the `NtClose` function before and after being hooked, respectively.

At this stage, the steps required to be taken diverge, as the method of determining whether the trampoline leads to the desired unhooked function depends on the hooking technique employed, i.e. a `jump` and the `double-push` technique.

When a `jump` is employed to return to the hooked function (as in the first hooking approach), it becomes necessary to locate all jumps within that memory region, so as to evaluate the destination of each jump and locate the memory region that houses the original function. Our research has revealed that there exist two `jump` patterns utilized in assembly by anti-virus programs, namely `long` and `short` jumps. In particular, we examine the *private* memory sector searching for:

- long jumps (utilized by the e.g. *Detours* library) with opcode `0xFF25`
- short jumps (employed by e.g. *MalwareBytes* [7]) with opcode `0xE9`.

In the case of the second technique (`double-push`), it is necessary to locate all sequences of `push rax; push rax; mov rax, addr` in order to extract the destination address and confirm that it points to the hooked function.

---

[6] https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualquery
[7] https://www.malwarebytes.com/
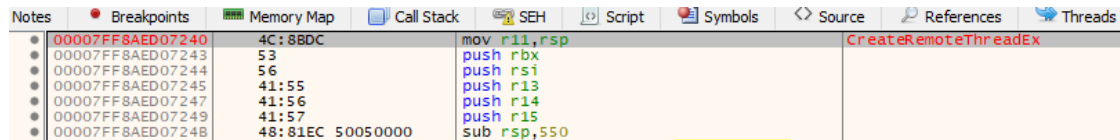
Figure 10: `CreateRemoteThreadEx` unhooked

Once the trampoline has been correctly located, the bytes that precede the aforementioned stub are the original bytes that were overwritten by the initial hook, and they must be rewritten to the original symbol address in order to unhook the function.

First version of `Whisper2Shout`: Initially, we carried out the unhooking idea by iterating over each hooked DLL and executing the following procedures:

1. Employ a direct system call to `NtProtectVirtualMemory` to adjust 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 conclusive results are shown in Figure 10 for `CreateRemoteThreadEx`.

Final version of `Whisper2Shout`: During the evaluation of the initial version of our unhooking technique, we were met with a challenge posed by security products that employed a more sophisticated technique and monitored the stability of their hooks, thereby nullifying our modifications. To address this issue, we adjusted our strategy by opting to overwrite the AV hooking trampoline with a jump to the original prologue function instead of overwriting the hooks at the symbol address (as we previously did).

As a result, when the AV inspects its hooks, all the original jumps will remain unaltered, directing towards the same locations where the AV positioned the hooking trampolines. However, the instructions there will no longer divert the execution towards the AV DLL. Essentially, we have circumvented the hooking trampoline, guaranteeing the seamless execution of the function as if no hooks were present, even in the presence of a jump at the symbol address. Figure 11 depicts the layout of the `NtClose` function after being unhooked in this manner.

It is worth mentioning that all previous observations are still valid and allow us to traverse the process address space and retrieve all original stubs 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 have knowledge of the location of the original function stub.

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

- Use a direct system call to `NtProtectVirtualMemory` to adjust the protection of the memory region that stores the stub to RW.
- Add a short jump instruction - opcode `0xe9` - to jump to the original prologue.
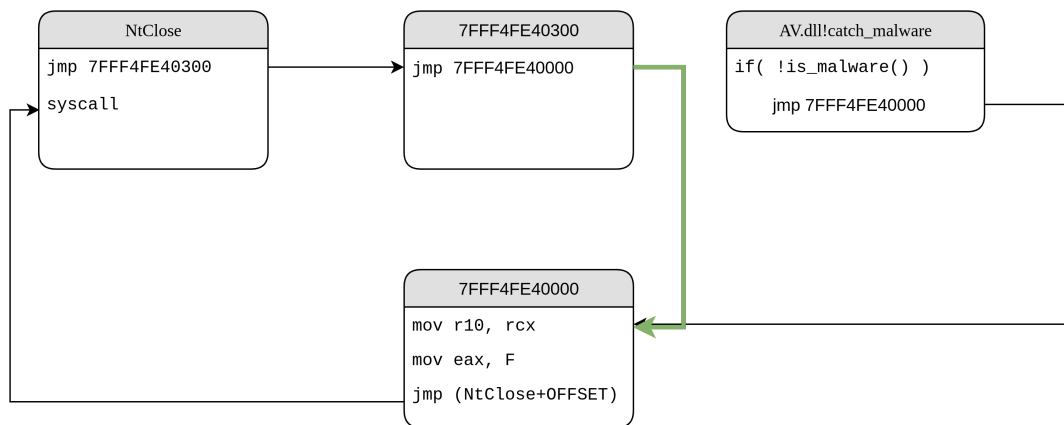
Figure 11: Layout of the `NtClose` function after being unhooked.

- Revert the memory back to RX using another direct system call to `NtProtectVirtualMemory`

The aforementioned operations can be executed to successfully unhook one API. At this stage, we may opt to either i) indiscriminately apply this technique to all APIs within a user-defined DLL, or ii) methodically iterate over all APIs, or a selected subset of user-defined APIs, and remove the hook only if it is present.

To summarize, the final unhooking process of Whisper2Shout involves iterating the following steps for each API the user desires to unhook:

- verify if the API has been hooked
- obtain the pointers to the *hooking* and *original* stubs
- overwrite the *hooking* stub with a jump to the *original* stub

5. Results

The efficacy of `Whisper2Shout` was tested against the antivirus software and endpoint detection and response systems listed in Table 1.

The testing process was executed in several steps. Initially, an anti-virus or endpoint detection and response system was installed, followed by the creation of a test program using the C programming language, for instance a simple "Hello World" program. The unhooking code was then added to the program, which was opened in a debugger. Prior to executing the unhooking code, it was verified that certain APIs were hooked, such as the `LdrLoadDll` API. Finally, after executing the unhooking code, the same APIs were verified to be unhooked by walking through the pointers in the debugger's graphical user interface.

The number of hooked APIs goes up to approximately 90 in `ntdll.dll` and approximately 20 in `kernel32.dll`. A comprehensive list of hooked APIs for EDRs is provided in [19]. As previously mentioned, it is worth to mention that `Whisper2Shout` can be configured to either

unhook all APIs within the memory space of a given process or methodically unhook a subset of user-defined APIs. In the results provided in this paper the first option was adopted, in order to obtain a more comprehensive analysis on the capabilities of our technique.

It was noted that not all of the anti-virus programs listed in the study employed API hooking. However, for those that did, `Whisper2Shout` was able to successfully remove the hooks, and we validated this result by analyzing the hooked APIs of each DLL and verifying that the hooks set by the AVs/EDRs were no longer present.

As a demonstration of its potential, `Whisper2Shout` was utilized to pack a Cobalt Strike beacon shellcode and several other payloads that were detected as malicious. The technique was able to effectively bypass the security products that rely on API hooking, showcasing its efficacy as a tool for evading detection.

6. Conclusions

The technique discussed in this paper allows to bypass user-space API hooking in a universal way. The core of this mechanism lies in the observation that memory allocated by AVs/EDRs is designated as *Private* in all products analyzed. This makes identifying the stub effortless, as it cannot be mistaken for any other central libraries in memory, enabling us to pinpoint the memory area assigned by the anti-virus `Dll` to accommodate the stubs. The steps taken to acquire the necessary information are executed in a stealth manner as they only require reading memory and following pointers within our own process address space. Moreover, calls to `NtProtectVirtualMemory` are kept to a bare minimum, as we only use the system call twice per *Private* memory area, once to set the region to RW and again to set it back to RX. However, `Whisper2Shout` still suffers from the following two limitations:

- calls to `NtProtectVirtualMemory`, although kept to a bare minimum, can still alert some AV/EDR
- the scanning technique itself could still be detected, for example, by setting a GUARD page bit [20] on the memory of the trampolines and validate from the exception handler what is the address of the offending instruction with respect to the return address on stack.

From a defensive viewpoint, user-space hooking is a crucial mechanism, and though bypasses may be feasible, its utilization as part of a defense-in-depth strategy is imperative. Furthermore, security products that monitor hook integrity are preferable, as they render attacker's efforts more arduous, increasing the likelihood of detection.

Finally, it is important to emphasize that anti-malware solutions are not impenetrable security mechanisms that can guarantee protection against every possible threat, they are tools that defenders can utilize to identify anomalies in monitored systems. Proper setup and tuning of security software are crucial steps when installing a new anti-virus program in a network. The ability to receive meaningful alerts would facilitate defenders in detecting and responding to stealth attacks that may not be automatically identified as malicious but appear suspicious.

7. Acknowledgments

References

[1] M. F. Marhusin, H. Larkin, C. Lokan, D. Cornforth, An evaluation of api calls hooking performance, in: 2008 International Conference on Computational Intelligence and Security, volume 1, 2008, pp. 315–319. doi:`10.1109/CIS.2008.199`.

[2] M. Botacin, F. D. Domingues, F. Ceschin, R. Machnicki, M. A. Zanata Alves, P. L. de Geus, A. Grégio, Antiviruses under the microscope: A hands-on perspective, Computers & Security 112 (2022) 102500. URL: https://www.sciencedirect.com/science/article/pii/S0167404821003242. doi:`https://doi.org/10.1016/j.cose.2021.102500`.

[3] G. Bernardinetti, D. Di Cristofaro, G. Bianchi, Pezong: Advanced packer for automated evasion on windows, Journal of Computer Virology and Hacking Techniques 18 (2022). doi:`10.1007/s11416-022-00417-2`.

[4] F. Mosch, A tale of edr bypass methods, 2021. URL: https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/, accessed: 2023-01-13.

[5] J. Tang, Universal unhooking: Blinding security software, 2017. URL: https://blogs.blackberry.com/en/2017/02/universal-unhooking-blinding-security-software, accessed: 2022-12-14.

[6] H. Bui, Bypass edr's memory protection, introduction to hooking, 2019. URL: https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6, accessed: 2023-01-10.

[7] Sektor7, Perun's fart - yet another unhooking method, 2021. URL: https://blog.sektor7.net/#!res/2021/perunsfart.md, accessed: 2022-12-14.

[8] Microsoft, Writing preoperation and postoperation callback routines, 2021. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/writing-preoperation-and-postoperation-callback-routines, accessed: 2022-12-14.

[9] RedBluePurple, Detecting process injection with etw, 2021. URL: https://blog.redbluepurple.io/windows-security-research/kernel-tracing-injection-detection, accessed: 2022-12-14.

[10] Slaeryan, Shellycoat, 2020. URL: https://github.com/slaeryan/AQUARMOURY/blob/master/Shellycoat/README.md, accessed: 2023-01-10.

[11] Y. Kawakoya, E. Shioji, Y. Otsuki, M. Iwamura, T. Yada, Stealth loader: Trace-free program loading for api obfuscation, in: M. Dacier, M. Bailey, M. Polychronakis, M. Antonakakis (Eds.), Research in Attacks, Intrusions, and Defenses, Springer International Publishing, Cham, 2017, pp. 217–237.

[12] D. C. D'Elia, L. Invidia, L. Querzoni, Rope: Covert multi-process malware execution with

return-oriented programming, in: Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I, Springer-Verlag, Berlin, Heidelberg, 2021, p. 197–217. URL: https://doi.org/10.1007/978-3-030-88418-5_10. doi:10.1007/978-3-030-88418-5_10.

[13] Y. Kawakoya, M. Iwamura, E. Shioji, T. Hariu, Api chaser: Anti-analysis resistant malware analyzer, in: S. J. Stolfo, A. Stavrou, C. V. Wright (Eds.), Research in Attacks, Intrusions, and Defenses, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 123–143.

[14] D. C. D'Elia, S. Nicchi, M. Mariani, M. Marini, F. Palmaro, Designing robust api monitoring solutions, IEEE Transactions on Dependable and Secure Computing 20 (2021) 1–1. doi:10.1109/TDSC.2021.3133729.

[15] x86 Instruction Set Reference, Jmp, 2016. URL: https://c9x.me/x86/html/file_module_x86_id_147.html, accessed: 2022-12-13.

[16] Microsoft, Winnt memory basic information, 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-memory_basic_information, accessed: 2022-12-14.

[17] Microsoft, Detours, 2022. URL: https://github.com/microsoft/Detours, accessed: 2022-12-13.

[18] T. Bitton, U. Yavo, Captain hook: Pirating avs to bypass exploit mitigations, 2016. URL: https://www.blackhat.com/us-16/briefings/schedule/#captain-hook-pirating-avs-to-bypass-exploit-mitigations-4057, blackHat USA.

[19] Mr-Un1k0d3r, Edrs, 2021. URL: https://github.com/Mr-Un1k0d3r/EDRs, accessed: 2022-12-13.

[20] Microsoft, Creating guard pages, 2021. URL: https://learn.microsoft.com/en-us/windows/win32/memory/creating-guard-pages, accessed: 2022-12-14.