Antivirus evasion by user mode unhooking on Windows 10

Tom Broumels BSc
MSc Security and Network Engineering
University of Amsterdam
tom.broumels@os3.nl

Sander Ubink MSc
Graduation mentor
KPMG Netherlands
ubink.sander@kpmg.nl

ABSTRACT

Several antivirus products place inline user mode hooks in an attempt to recognize malicious behaviour of processes. Removing these hooks is called unhooking and can make malicious activity invisible to the antivirus software.
We applied unhooking on a Windows 10 system that was running antivirus software that placed user mode hooks. Malicious payloads that call hooked functions were created as target processes for five implemented unhooking techniques. These five techniques consist of three existing ones: Section Remapping, Prologue Restoring and Perun's Fart, as well as two novel techniques introduced in this study: Interprocess Function Copying and Interprocess Section Copying.
We found that all unhooking techniques were able to remove hooks and none of them were detected. In most cases a triggered hooked function did not lead to an antivirus alert.
Although the quantity of antivirus products and tested hooked functions do not allow us to make statements in general, this study shows that user mode unhooking can be used for antivirus evasion today. Alternatives to user mode hooking should be found in security mechanisms that do not allow tampering by using user permissions.

1 INTRODUCTION

Threat actors have been using malware to, among other things, establish a foothold within targeted organizations for years [1, 2]. Developments in both malware detection and malware evasion have led to an ongoing arms race [3].

One way for an organization to gain insight in its resilience to these threat actors is to employ red teaming assessments that simulate the approaches of Advanced Persistent Threat (APT) groups [4, 5]. Such APT groups are often well-resourced, work in an sophisticated and stealthy manner and persist in a target network for a longer period of time [1, 2]. For a red team it is necessary to have insight in current techniques to evade detection mechanisms to be able to realistically simulate the modus operandi of APTs.

User mode unhooking has been found in recent malware and is recognized by Mitre ATT&CK Enterprise as a way to impair defenses by disabling or modifying tools [6, 7].

This research concerns evading antivirus (AV) products that place user mode hooks in an attempt to recognize malicious behaviour of processes. Removing such hooks from a process, known as unhooking, can hide malicious activity from the AV software's view. The goal of our research is to gain insight in current AV unhooking techniques for today's AV software on Windows 10.

2 BACKGROUND

This section describes user mode hooking and architectural aspects of Windows 10 that are relevant for user mode (un)hooking. Next, the use of direct syscalls for unhooking is explained.

2.1 User mode hooking

Up to Windows Vista, security software vendors used kernel hooks on the System Service Dispatch Table (SSDT). This allowed them to tap into fundamental system services and as such get the required input for detecting malicious behaviour [8, 9].

Because malware was exploiting kernel hooking as well and incorrectly implemented kernel patches by AV vendors could make a system unstable, Microsoft introduced PatchGuard in Windows Vista in 2006. This led to the situation that security vendors who utilize hooking could either bypass PatchGuard and risk malfunctioning products after Microsoft patches, or use non kernel hooking solutions such as user mode hooking. The latter was preferred and some vendors are still using user mode hooks today.

The relevant user mode hooking technique for this research, inline user mode hooking, is briefly introduced below based on the work of Shaid and Maarof [10].

2.1.1 Inline user mode hooking.

Inline hooking typically targets the machine code in a function of an imported Dynamic-Link Library (DLL). The first bytes of the machine code are replaced by a 5 byte jump instruction to a custom made method that allows or blocks execution of an API call, after which it executes the original functionality of the API call. To be able to do this, it makes sure it has a trampoline function that contains the overwritten machine code followed with a jump instruction to the first byte of unaltered machine code in the original API method. In Figure 1 an example of a hooked CopyFileExW method of kernelbase.dll is shown.

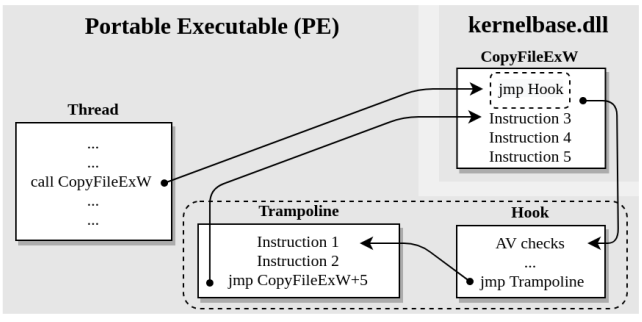

Figure 1: Example of Inline hooking on a process importing the CopyFileExW method of a kernelbase.dll instance.

The hook replaces instructions 1 and 2 that are saved in the Trampoline function. If CopyFileExW is called, first the custom code in the Hook method is executed. If the hook does not block the call, the original instructions 1 and 2 are executed next, followed by the original instructions 3, 4 and 5 in CopyFileExW. These hooks are sometimes called Detour hooks because Microsoft introduced the Microsoft Detours library to create such hooks for monitoring and instrumentation of API calls on Windows [11].

To unhook a hooked function, the original bytes should be restored or the jump address should be changed to point at a location containing the original bytes.

User mode hooks are created by AV products using user permissions. This means that a process with user permissions can remove the hook if a way can be found to not trigger the hooks in the process.

2.2 Windows 10 architectural aspects

After a process is created, the AV product hooks certain functions of imported Windows DLLs called modules. Interesting functions to hook are close to the point where program execution is handed over to kernel mode, i.e. methods that perform a "syscall" assembly instruction after setting up the right registers with the expected argument data by the syscall. The majority of such methods are located in kernelbase.dll (or kernel32.dll in earlier Windows versions) and ntdll.dll, which exports common system calls to user mode [12, 13]. Both DLLs are imported by a considerable amount of other modules.

DLLs are memory-mapped and shared with other processes using virtual memory. The relative module address in memory is in practice the same for each process. Changes to a loaded module are process specific and stored in memory using copy-on-write.

2.3 Direct syscalls

To evade user mode hooks altogether, direct syscalls can be used [6, 14, 15]. This means that the officially undocumented syscall assembly instruction gets executed directly instead of by calling a possible user mode hooked Windows API function that calls the same syscall internally. To make this work, the right registry values should be set before execution, among others the Windows version dependent system call index in register eax.

If a payload is implemented using direct syscalls where possible, there is in theory no need for unhooking because the hooks will already be evaded. However, creating payloads that way is cumbersome in practice. Especially for closed source payloads that are already compiled into an executable.

3 RELATED WORK

Shaid and Maarof identified different user mode hooking techniques and the corresponding forensic artefacts that make the detection of such hooks possible [10]. The general processes of hooking and unhooking are documented in several blog posts as well [16, 17]. These articles mainly describe .text section hooks (i.e. inline- or Detour hooking) in the context of AV evasion.

In 2017, Tang published proof-of-concept code titled "DLLRefresher" that removes the hooks of a process by reloading the original instructions from the Windows executable files and overwriting the instructions in memory that where added for hooking. A relevant finding is that the thread used for unhooking can be detected due to the API calls that are executed [16].

In 2019, De Plaa demonstrated unhooking of ntdll.dll for AV bypassing to create a memory dump in an unnoticed way [14]. Direct system calls are used to avoid calling potentially hooked APIs while unhooking. Inspecting the PoC reveals that the first 5 bytes of the hooked DLL are overwritten with the original bytes that are hard coded in the unhooking code.

A year later Saha created an utility for removing user mode hooks in the .text section of a DLL [15]. Different bypassing techniques are mentioned, of which section remapping is implemented: the hooked .text section is overwritten with the original DLL content on disk. Direct system calls are used to evade the AV while unhooking. The unhooking code itself is compiled as a Position Independent Code (PIC) reflectively loadable DLL.

Misgav and Yavo identified different tactics for unhooking based on malware analysis [6]. The techniques described earlier can be classified as implementations of these tactics. After their publication more techniques emerged. Winter-Smith introduced Firewalker that aims to trace the original thunk by interpreting the assembly code of the hook [18]. The latest unhooking technique published, Perun's Fart, starts a new process and copies part of its ntdll.dll code before AV software has hooked it [19].

Some tactics are mentioned to prevent the hooks from being triggered while loading shellcode for the purpose of unhooking, e.g., avoiding names in the assembly source code that reflect the names of the corresponding legitimate Windows API calls, or by obfuscating bytes in the shellcode that are used to overwrite hooks in memory [20].

Various sources describe tests or evasions of specific security products. These sources discuss older or unknown versions of the AV software used, or do not mention user mode unhooking explicitly [21–23].

To the best of our knowledge, no recent research was publicly available at the time of writing that describes the effectiveness and detectability of different user mode unhooking techniques on today's AV software.

4 RESEARCH QUESTION

"What is the effectiveness and detectability of today's user mode unhooking techniques on a x64 process on a Windows 10 Enterprise system that is protected by antivirus software?"

We define an unhooking technique to be *effective* if the targeted AV hook is removed for the duration of at least one minute to allow a malicious payload to be executed successfully. Otherwise it is *ineffective*. This means that techniques that evade calling the hooked function in its entirety are by definition ineffective, because no hooks are removed and an unmodified payload might still invoke a hooked function and trigger the hook.

An unhooking action is defined as *undetected* if no alert is registered by the AV software for the process that is unhooking and the unhooking code terminates gracefully. Otherwise it is *detected*.

5 METHODOLOGY

Briefly summarized, we answered the research question as follows. We started by selecting AV products that place user mode hooks. Based on the found hooks, we selected unhooking techniques designed for removing user mode hooks and created payloads that called specific functions that were hooked by at least one of the AV products. Finally we applied all unhooking techniques on all payloads for all AV products in a series of experiments. Conclusions about effectiveness and detectability are drawn based on: AV and Sysmon alerts, exit codes of executed payloads and unhooking techniques, and hooks in memory of payloads before and after unhooking was used. Each step is documented in more detail below. All source code and scripts are available on GitHub [24].

5.1 Included user mode hooking AV software

We installed several AV products meant for business use that are listed in the AVTEST published in April 2021 [25]. The default installation settings were used. We only included those products of which a free trial version was available for immediate download. These are listed in Table 1.

	Antivirus software	User mode hooks
✓	Comodo Internet Security Pro 10	Inline
✓	F-Secure Computer Protection Premium 21.5	Inline
✓	Sophos Intercept X Advanced with EDR 10.8	Inline, IAT
	G Data Endpoint Protection Business 15.0	None
	Kaspersky Small Office Security 21.3	None
	Malwarebytes Premium 4.4.0	None
	Microsoft Defender Antivirus 4.18	None
	Avast Business Antivirus Pro Plus 20.10	Unknown
	Bitdefender Endpoint Security 6.6	Unknown

Table 1: Used user mode hooks by AV products. Checked products are included in our experiments.

We identified the user mode hooks in the following way. We deployed each AV product on a separate VirtualBox image using DetectionLab software [26, 27]. We also created an additional image containing no AV software at all that was used as a baseline image containing no AV hooks placed. Next we ran three applications, a predeployed Microsoft executable, and two executables compiled from the Microsoft Visual Studio Enterprise 2019 boilerplate code for an C++ Console and Desktop application [28].

HookShark64 was used for identifying hooks, and by using Process Dump we saved memory of the processes to disk [29, 30]. Using Ghidra, we inspected the memory dumps and confirmed the existence of the hooks found by HookShark64 [31]. Because we were looking for user mode hooks, the three executables as well as Process Dump were ran with user mode permissions.

Four products placed no hooks on the three executables. Results for Avast and Bitdefender are inconclusive due to incompatibility with VirtualBox.

The Import Address Table (IAT) hook found on Sophos is related to the Sophos specific HitManProAlert DLL called hmpalert.dll, it hooks the imported LoadLibraryExW function of kernel32.dll by redirecting the location of the method in the IAT. Since this is the only IAT hook found in the three AV products, we have left further analysis for future work.

The amount of functions hooked in DLLs that were inline user mode hooked by at least two of the three AV products are listed in Table 2.

		kernel32.dll	kernelbase.dll	ntdll.dl	user32.dll
Comodo	notepad.exe	0	10	36	3
	Console application	4	10	52	43
	Desktop application	4	10	52	43
F-Secure	notepad.exe	0	0	0	0
	Console application	1	11	0	
	Desktop application	1	11	0	2
Sophos	notepad.exe	1	1	7	0
	Console application	1	1	7	
	Desktop application	**1**	1	7	0

Table 2: Amount of hooked functions for the DLLs that are at least hooked by two antivirus products. An empty cell indicates a not imported DLLs for the specified application.

We observed differences between AV products in amounts of hooked functions on DLLs. We also found that single AV products place different hooks for different processes.

The full table with the amounts of hooked DLLs can be found in "Appendix A - Inline Hooks placed by antivirus software".

The hashes on disk for the DLLs that are inline hooked have the same SHA256, SHA512 and MD5 hashes before and after installation of AV software. This indicates that the hooks are placed on a process after loading DLLs in memory. For the sake of brevity, we only listed the SHA256 hashes in "Appendix B - Hashes of inline hooked DLLs".

5.2 Included unhooking techniques

We started by listing all techniques that were documented as unhooking techniques for x64 processes. Next, we splitted these techniques in two categories:

(1) Techniques that remove the hook or alter the hook in a way that it is no longer triggered, e.g. by prologue restoring or section remapping.
(2) Techniques that evade usage of the hooked Windows API calls, e.g. by using syscalls, loading a copy of the hooked DLL or embedding copies of functions in an executable.

The resulting list can be found in "Appendix C - Unhooking techniques overview". In our research we only included techniques of the first category that remove the hook because these give an adversary the advantage of being able to execute payloads that are not aware of hooks, e.g., closed source payloads.

A total of five unhooking techniques are included as can be seen in Table 3. Three of those are existing techniques, the other two are new techniques that we present in this paper.

	Unhooking technique	Description
✓	Section Remapping [15, 32]	Reload the hooked .text section in memory by the .text section in the original DLL on disk.
✓	Prologue Restoring [6, 14]	Overwrite the first bytes of the hook with the known original bytes stored in the unhooking program.
✓	Perun's Fart [19]	Overwrite hooked .text section with code of a not yet hooked ntdll.dll in a created suspended process.
	Rebuilding Function Stubs [6]	Point the jump to a trampoline containing the original first bytes and a jump to right after the hook.
	Original Thunk Tracing [18]	Find the original code in memory by analysing the code of the hook and alter hook to jump there.
✓	Interprocess Function Copying [this paper]	Overwrite hooked function bytes with the original bytes found in an other not hooked process.
✓	Interprocess Section Copying [this paper]	Overwrite the .text section in memory with original .text found in an other not hooked process.

Table 3: Unhooking techniques that are able to prevent the code of the hook from running. Checked techniques are included in our experiments.

Due to time constraints, Rebuilding Function Stubs and Original Thunk Tracing have not been implemented. Compared to the other techniques, we expect these techniques to be less interesting to adversaries because of the additional effort needed to implement the techniques.

We wanted to measure the effect of the unhooking techniques themselves. Therefore, we implemented an executable containing only the code needed for unhooking a specified process. In our experiments this was a different process than the unhooking process. We abstained from implementing code to perform other evasion techniques in the unhooking application, as is the case for the proof-of-concept code published for Section Remapping.

We knew that direct syscalls would not be detected by the very same user mode hooks we were trying to unhook [6, 14, 15]. For that reason, direct syscalls were used instead of API calls for the implementation of the unhooking techniques.

The selected techniques were implemented in a way that empty handles and erroneous NTSTATUS codes would exit the unhooking program with an exit code other than zero. This gave us an indication of success of an unhooking attempt as perceived by the attacker.

Technique specific implementation details are described below.

5.2.1 *Section remapping.*

For section remapping we used Saha's Shellycoat source code as starting point [15]. We changed a few things: we ran the code as a commandline executable so we changed the entrypoint and supplied a Process IDentifier (PID) as input. This also meant that the targeted process and the unhooking process were no longer the same process and we had to replace the memcpy function by a direct syscall similar to NtWriteVirtualMemory to avoid an access violation on writing in the targeted process.

5.2.2 *Prologue Restoring.*

For Prologue Restoring we used the commandline interface version of the Dumpert source code by De Plaa as starting point [14]. By default it also targeted the current process, so we adjusted the code to take a PID of a target process to unhook as input.

In the proof-of-concept a fixed amount of 5 bytes is assumed for restoring prologues. The bytes itself were stored as an array in the code. While an assumed five bytes was enough for most functions wrapping a syscall in ntdll.dll, it was not enough for hooks on other functions. If the first five bytes were replaced by an one byte JMP assembly instruction to a four byte address, but the last overwritten instruction was not ending on byte five, the remaining bytes of the instruction would be set to an INT 3 (0xCC) instruction that would terminate the process when called. By manually checking the size of the actual hooks, we added the right amount of bytes for Windows 10 20H2 for the three functions mentioned in Table 4.

5.2.3 *Perun's Fart.*

No source code was available for Perun's Fart at the time of writing. The proof-of-concept demo video suggested that only part of the functions of ntdll.dll, those wrapping syscall instructions, were unhooked. Since we could not pinpoint the exact unhooked functions in the video, we instead implemented it in a way that restored the entire .text section at once.

5.2.4 *Introduced technique: Interprocess Function Copying.*

We observed that F-Secure and Comodo would not hook exactly the same functions of a DLL for each process. This could be exploited for the removal of hooks by comparing the first bytes of a hooked function with the bytes of the same function in an other process that was not hooked but owned by the same user. If differences were found, the hooked bytes (that were recognizable by a jump instruction) were overwritten with the unhooked version of the bytes. We will refer to this technique as Interprocess Function Copying.

5.2.5 *Introduced technique: Interprocess Section Copying.*

We used the same principle as Interprocess Function Copying to copy the entire .text section of a DLL at once. We will refer to this technique as Interprocess Section Copying.

5.3 Payloads for triggering hooked functions

Since the unhooking techniques target an other malicious process, we needed programs that fulfilled that role.

Three of such programs that we will call payloads have been implemented in a way that for each AV product at least one of the hooked functions is called when all three payloads have been executed in a successful way. In Table 4 we can see which payload triggers which potentially hooked function and which AV products hooked the function. The code for the payloads is based on common malicious behaviour and implemented by combining and adjusting online examples that use Windows API calls. The payloads are described below.

5.3.1 *Malicious File Copying.*

This payload mimics copying files with a malicious content. First,

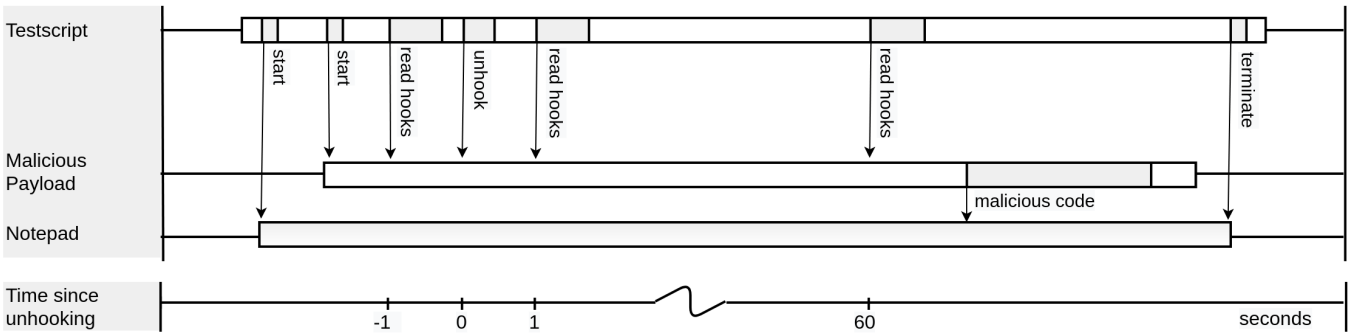

Figure 2: Execution of a single test including unhooking.

Payload and Function called	Hooked by Comodo	Hooked by F-Secure	Hooked by Sophos
Malicious File Copying Calls CopyFileExW in kernelbase.dll	✓	✓	✗
Basic Process Injection Calls CreateRemoteThreadEx in kernelbase.dll	✗	✓	✗
Process Injection by Section Remapping Calls NtMapViewOfSection in ntdll.dll	✗	✗	✓

Table 4: Payloads designed to trigger hooks of specific AV software by calling specific functions.

a temporary folder "C:\tempdir" is created. Next, a text file containing an executable containing a reverse shell is copied to an executable file in the temp directory and an attempt is made to overwrite "ntdll.dll" and "lsass.exe", and to create new files called "ntdll2.dll' and 'lsass2.exe' using the same malicious file. Next, similar actions are performed using a text file containing the EICAR test string. A similar process is repeated using files that originate from a networkshare (in our case a VirtualBox shared folder). The files are copied using CopyFileExW and because we set directive #UNICODE, this resolves to a call to CopyFileExW.

5.3.2 *Basic Process injection.*

This payload injects shellcode for showing a MessageBox into a remote process and spawns a thread in that remote process to execute the shellcode. First, memory is allocated in the opened targeted process (OpenProcess) where the shellcode will be stored (VirtualAllocEx). Next, the shellcode is copied in the allocated memory (WriteProcessMemory). Next, a new thread is created in the target process that starts executing the injected shellcode (CreateRemoteThread). This last step will trigger a call of CreateRemoteThreadEx in kernelbase.dll.

5.3.3 *Process Injection by Section Remapping.*

This payload results in executing the same shellcode mentioned, but other functions are used for delivering the shellcode to the remote process and executing it. First the payload will create a section in its own process (NtCreateSection) and map a view to it (NtMapViewOfSection). The received section handle is then used to map a view of the same section in the target process (NtMapViewOfSection) right after opening the remote process (OpenProcess). Next, the shellcode is copied to the section view of the payload (memcpy) resulting in the shellcode being available for the target process. Finally, a new thread is started in the target process (RtlCreateThread).

5.4 Automated unhooking experiments

To measure the effect of the unhooking techniques on AV software, we ran the payloads without unhooking and with unhooking with each included technique. A PowerShell script was created to perform the experiments in an repeatable and convenient way. A single experiment is executed as shown in Figure 2.

First, a new notepad.exe process is created that will be targeted by our injection malware. This process is also used as the source process of unhooked bytes for both Interprocess copying techniques. Next, the malware started with a ninety second sleep period to allow us to identify any hooks on the malware before we started unhooking. We identify hooks in memory one second and one minute after we started unhooking using a custom made application (HookDetector). This enabled us to see if removed hooks were removed for a longer period of time. Next, the malicious activity of the payload was ran and finally the notepad.exe process was killed.

The exit codes of the process applying the unhooking technique and the malicious payload were outputted to the screen.

After the execution of all experiments on an AV product, we manually checked for AV events generated by AV software (both with user and administrator permissions, both on the local system and in a remote central dashboard when available).

Also, Sysmon alerts generated using the Sysmon Modular configuration were manually checked [33].

For each experiment an unique version of the payload was compiled by changing a string value in a printf statement to make sure the malware was not blocked on forehand due to detection of malicious behaviour by the same executable during earlier tests. To be able to identify the exact cause of an AV or Sysmon event, we gave each payload executable an unique filename. We excluded the application that reads the hooks from the memory of the payload from being monitored by the AV software to make sure the hooks were registered.

		Detectability		Effectiveness	
Malware	**Unhooking technique**	**AV alert on payload**	**Payload exit code 0**	**Unhooking exit code 0**	**Hook removed**
F-Secure Computer Protection Premium 21.5					
Malicious File Copying	None	✗[1]	✓	-	-
	Interprocess Function Copying	✗[1]	✓	✓	✓
	Interprocess Section Copying	✗[1]	✓	✓	✓
Hook: kernelbase.dll	Peruns Fart	✗[1]	✓	✓	✗[3]
CopyFileExW	Prologue Restoring	✗[1]	✓	✓	✓
	Section Remapping	✗[1]	✓	✓	✓
Shellcode Injection - Basic	None	✓[2]	✗	-	-
	Interprocess Function Copying	✗	✓	✓	✓
	Interprocess Section Copying	✗	✓	✓	✓
Hook: kernelbase.dll	Peruns Fart	✓[2]	✗[3]	✓	✗[3]
CreateRemoteThreadEx	Prologue Restoring	✗	✓	✓	✓
	Section Remapping	✗	✓	✓	✓
Sophos Intercept X Advanced with EDR 10.8					
Shellcode Injection - Section Mapping	None	✗	✓	-	-
	Interprocess Function Copying	✗	✗[4]	✓	✗[6]
	Interprocess Section Copying	✗	✗[4]	✓	✗[6]
Using ntdll.dll	Peruns Fart	✗	✓	✓	✓
NtMapViewOfSection	Prologue Restoring	✗	✓	✓	✓
	Section Remapping	✗	✓	✓	✓
Comodo Internet Security Pro 10					
Malicious File Copying	None	✗[1]	✓	-	-
	Interprocess Function Copying	✗[1]	✓	✓	✗[5]
	Interprocess Section Copying	✗[1]	✓	✓	✗[5]
Using kernelbase.dll	Peruns Fart	✗[1]	✓	✓	✗[3]
CopyFileExW	Prologue Restoring	✗[1]	✓	✓	✓
	Section Remapping	✗[1]	✓	✓	✓

[1] Antivirus event related to the harmful content of copied files, not the payload itself.
[2] Reason: Exploit:W32/ShellCodeInjection.A!DeepGuard.
[3] Peruns Fart can only copy ntdll.dll because no other dll's are available for a sleeping process.
[4] Execution failed because hooks are copied from source process refering to relative memory not existing in target process.
[5] Hook is overwritten by code containing exactly the same hooks.
[6] Hook is overwritten by code containing hooks referring to nonexistent memory.

Table 5: Results for experiments of which the payload called a function that was hooked by the antivirus software.

6 RESULTS

Table 5 contains the results for those experiments for which the AV software hooked the function that was called by the payload. The results for the full series of experiments per AV product can be found in "Appendix D - Full experimental results".

The Sysmon events and AV details for alerts are described below.

6.1 Sysmon events

For all three AV products used, a Sysmon event 8 (CreateRemoteThread, Process injection) was generated within one second of the execution of the actual Process Injection code. The alert identified notepad.exe as the target image of the process injection. Also, all three AV products generated alerts for files that were copied by payload Malicious File Copying, but not for the payload itself.

Within 0.3 seconds after the creation of a payload- or unhooking process on F-Secure or Sophos, the following Sysmon events were generated:

- Event 7 (Image Load, DLL Side-Loading)
- Event 10 (Process Accessed)
- Event 25 (Process Tampering)

Sysmon registered suspended processes created by Perun's Fart as events with ID 4688 (Process Creation) and mentioned that the Perun's Fart process was the parent process.

AV product specific observations related to Sysmon, application and AV logs are discussed below.

6.1.1 F-Secure specific observations.

When the exit code of the payload was non-zero, we observed an Application Closed event in the F-Secure Graphical User Interface (GUI) with reason "Exploit:W32/ShellCodeInjection.A!DeepGuard".

6.1.2 Sophos.

In addition to the warning in the GUI, an application warning log with event ID 32 (Sophos Anti-Virus) was created. It mentioned the names of malicious files that were copied and marked as "virus/spyware". For files removed by Sophos, an event with ID 36 was generated.

An application error event ID 1000 (Faulting Application) was reported within one second of the execution of the malicious payload Process Injection By Section Mapping if a variant of Interprocess Copying was used that was terminated with a non-zero exit code.

During the experiments we observed 15 informative SecurityCenter logs with event ID 15 ("Updated Sophos Anti-Virus status successfully to SECURITY_PRODUCT_STATE_ON.") with intervals between one second and 5:51 minutes.

6.1.3 Comodo.

All executed payloads and unhooking techniques were marked as unrecognized by Comodo. This led to containerized execution.

The experimental results were the same if the experiment was repeated with the auto-containment feature set to off.

Malicious files were listed as "Detected Threats" in the AV Events list of the Comodo GUI.

7 DISCUSSION

If we look at the results in "Appendix D - Full experimental results" we can observe that all unhooking techniques were terminated gracefully with exit code zero. This means the processes were not blocked by AV and that none of the checks in the unhooking techniques themselves failed. This is not surprising because the security mechanism (hook) is created with user permissions, so the user has the permissions needed to tamper with it. This issue is inherent to user mode hooking and an alternative mechanism should be found that is not accessible through user permissions. Using Event Tracing for Windows (ETW) might be a suitable alternative for real time tracing and acting on both user and kernel mode events on a granular level [34]. Additional kernel tracing events have been introduced in Windows 10 build 1809 as a means for AV vendors to receive events from kernel mode memory management (MM) and Asynchronous Procedure Calls (APC) APIs. The related ETW provider is called "Microsoft-Windows-Threat-Intelligence" and can only be consumed by security vendors with proper signing certificates recognized by Microsoft [35]. Also, drivers have been used as a means to get control beyond user mode, but the possibilities are limited compared to the aforementioned alternative.

In most cases the hooks in memory were removed because the expected original bytes were found in memory after the unhooking technique was performed. There were two exceptions:

- If Perun's Fart was executed when a non ntdll.dll function was hooked. This makes sense, since only an unhooked ntdll.dll was accessible in the suspended process that was created.
- If any of the two variants of Interprocess Copying were used and both the function in source (notepad.exe) and target of copying were hooked. This resulted in a hook being overwritten by a hook. For Sophos this led to failing payloads because the location of the hooks differed between source and target process, resulting in a jump to nonexistent memory by the target of the copy action. An additional check in the unhooking code to confirm that the source is hook free before copying can solve this issue.

Payloads were executed successfully in all cases, except when the payload failed because of invalid hook addresses as mentioned above due to Interprocess Copying, or if Basic Process Injection was ran on F-Secure without unhooking or using Perun's Fart.

In all experiments, the hooks are exactly the same after one second and one minute of unhooking. Since it is unlikely that AV products (temporarily) remove their own hooks, we assume that no hooks are restored within one minute after unhooking.

AV alerts are only generated by F-Secure if no unhooking is used or unhooking is unsuccessful. Comodo and Sophos are not generating alerts for any of the three payloads and neither when no unhooking is used. This is an unexpected result because the payloads used were performing malicious behaviour and no shellcode obfuscation was applied.

One explanation for the shellcode injections not being detected could be that the AV recognizes a MessageBox as not harmful and allowed it. However, a rerun of the experiments with a clearly malicious payload (using shellcode for an unencoded reverse shell instead of a MessageBox) led to similar results.

To prove that the hooked functions are in fact executed, we created a copy of the Prologue Restoring payload and replaced the original bytes with INT 3 assembly instructions (0xCC). This causes a trap normally used for debugging that results in unsuccessful termination of a program when it executes the instructions. This way, we confirmed that all payloads on all AV products were in fact executed because all payloads exited with a non-zero exit code.

A possible explanation for Sophos and Comodo accepting the shellcode injection is that this behaviour is accepted by design. After all, one user mode program is injecting in an other user mode program for the same user. But this would not explain why these hooks are placed at all.

Other possible explanations that are left for future work are:

- Usage of hooks for the collection of data that make AV trigger after aggregation of events.
- AV software utilizing machine learning that marks activity early after the installation of the AV software as normal activity.
- Hooks being placed due to legacy code, but the hooks are no longer used to act upon.

For Comodo the latter possibility is not unrealistic because a blog post published in 2013 already explains how to evade Comodo user mode hooks [22]. Furthermore, the Host Intrusion Protections System feature of Comodo seem to be triggered based on generated events, not on the lines of code in a payload that call hooked functions.

For F-Secure we can observe that all unhooking techniques that are able to unhook hooks on kernelbase.dll make a difference between a blocked Basic Shellcode Injection payload or an unblocked one. This means unhooking was successfully used for AV evasion.

For all Malicious File Copying payloads, copied malicious files are detected by all AV products, even if unhooking is successful. The event generated does not mention the payload as source of the problem and the payload process is not blocked. This could indicate that file scanning functionality of AV, instead of the hooked function, is responsible for detection and removal of these files.

The Sysmon alerts that are created shortly after the execution of a process by Sophos or F-Secure can be attributed to these AV products placing the user mode hooks based on the moment that these events are generated. These events should therefore been seen as false positives from a defensive point of view.

Sysmon event 8 is only generated right after a successful Process Injection (Basic or Section mapping variant). For that reason we attribute these events to actual Process Injection. The events with ID 8 could therefore be used by AV products as an indicator of Process Injection, instead of using unhooking.

8 LIMITATIONS

There are some limitations to this research. A limited amount of AV products have been analyzed for user mode hooks. Therefore, no reliable statements can be made about the effectiveness and detectability of the techniques on AV products in general.

The default settings of AV products were used. Using different settings might have led to different results. A quick exploration of enabling the Host Intrusion Protection System (HIPS) mode on Comodo showed numerous alerts asking the user for permission to execute certain actions. Answering all questions with "no" would block the execution of payloads and unhooking techniques altogether.

For each AV product, one or two hooked functions were triggered. Our conclusions are based on the assumption that AV products treat all hooks the same way. It is not unthinkable that results differ if other hooked functions of the same product are triggered.

Selected unhooking methods were based on the inline user mode hooks that were used by the three AV products. Other AV products might use other types of user mode hooks such as Import Address Table (IAT) hooks and require different unhooking techniques.

Proof-of-concept code for Perun's Fart was not yet available. The original implementation by the author of the blog of the technique might differ. The proof-of-concept code of Shellycoat and Dumpert has been adapted to run as a commandline program unhooking an other process using direct syscalls. Running the original techniques might have led to different results, e.g. because virus signatures might exist for the original executables.

9 CONCLUSION

Our work showed that unhooking might still be used for antivirus evasion today, depending on the AV product used. We have demonstrated this for F-Secure using different techniques.

We published code for testing user mode unhooking techniques on payloads in an automated fashion.

Section Remapping and Prologue Restoring effectively unhooked user mode hooks on each Windows 10 system that we installed with a default configuration of Comodo, Sophos or F-Secure. Perun's Fart was limited to unhooking functions in ntdll.dll. None of the unhooking attempts were detected by antivirus software.

There are no indications that removed hooks were restored by AV software in the first minute after removal of a hook.

We introduced two variants of a new unhooking technique based on copying memory from an unhooked process to a hooked process and we called these techniques Interprocess Function Copying and Interprocess Section Copying. We have shown that these techniques were effective and undetected as long as a source process was available without hooks on the same function as the function that was being unhooked.

A security issue inherent to using user mode hooks is that user permissions allow tampering with the hooks. Alternatives to user mode hooking should be found in security mechanisms that do not allow tampering by using user permissions, such as EventTracing for Windows.

10 FUTURE WORK

Implementing the two remaining unhooking techniques and adding more payloads for other hooked functions will lead to a better impression of the resilience of an AV product to user mode unhooking techniques.

While analysing hooks, we found an import address table hook on Sophos. Because these hooks are also placed in user mode, further analysis might uncover similar security issues as we have seen for inline user mode hooking.

Finally, additional research is required to see if unnecessary unhooking attempts are detected as malicious behaviour on AV products that do not place user mode hooks themselves.

11 RESPONSIBLE DISCLOSURE

We have disclosed our findings to F-Secure prior to the publication of this paper or source code.

12 ACKNOWLEDGEMENTS

This research was carried out in the context of red teaming security assessments performed by KPMG. The author wishes to thank Sander Ubink at KPMG for his in-depth technical advise and detailed reviews.

Also, I want to thank my friends and family for their support. Especially my wife and my friends Martijn and Erik.

REFERENCES

[1] Ping Chen, Lieven Desmet, and Christophe Huygens. In *Communications and Multimedia Security*, pages 63–72, Berlin, Heidelberg, 2014.

[2] Colin Tankard. Advanced persistent threats and how to monitor and deter them. *Network security*, 2011(8):16–19, 2011.

[3] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense, 2017.

[4] B.J Wood and R.A Duggan. Red teaming of advanced information assurance concepts. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 112–118 vol.2. IEEE, 2000.

[5] Ivan Kovacevic and Stjepan Gros. Red teams - pentesters, apts, or neither. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1242–1249. Croatian Society MIPRO, 2020.

[6] Omri Misgav and Udi Yavo. Bypassing user-mode hooks: Analyzing malware evasion trend. https://www.first.org/resources/papers/telaviv2019/Ensilo-Omri-Misgav-Udi-Yavo-Analyzing-Malware-Evasion-Trend-Bypassing-User-Mode-Hooks.pdf, 2019. Visited: 2021-05-31.

[7] Mitre. Mitre att&ck enterprise matrix, impair defenses: Disable or modify tools. https://attack.mitre.org/techniques/T1562/001/, 2021. Visited: 2021-04-29.

[8] Gartner. Mcafee ad highlights ongoing microsoft security skirmish. https://www.gartner.com/resources/144000/144073/mcafee_ad_highlights_ongoing_144073.pdf, 2006-10-10. Visited: 2021-05-30.

[9] Wikipedia. Kernel patch protection. https://en.wikipedia.org/wiki/Kernel_Patch_Protection/, 2021. Visited: 2021-05-30.

[10] Syed Zainudeen Mohd Shaid and Mohd Aizaini Maarof. In memory detection of windows api call hooking technique. In *2015 International conference on computer, communications, and control technology (I4CT)*, pages 294–298. IEEE, 2015.

[11] Galen Hunt and David Tarditi. Event tracing. https://www.microsoft.com/en-us/research/project/detours/, 2002-01-16. Visited: 2021-06-06.

[12] Geoff Chappell. Ntdll. https://www.geoffchappell.com/studies/windows/win32/ntdll/index.htm?tx=4, 2016. Visited: 2021-05-30.

[13] Wikipedia. Native api. https://en.wikipedia.org/wiki/Native_API, 2021. Visited: 2021-04-29.

[14] Cornelis de Plaa. Red team tactics: Combining direct system calls and srdi to bypass av/edr. https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/, 2019-06-19. Visited: 2021-05-15.

[15] Upayan Saha. Shellycoat. https://github.com/slaeryan/AQUARMOURY/tree/master/Shellycoat, 2020-11-4. Visited: 2021-05-28.

[16] Jeffrey Tang. Universal unhooking: Blinding security software. https://blogs.blackberry.com/en/2017/02/universal-unhooking-blinding-security-software, 2017-02-28. Visited: 2021-05-22.

[17] Mantvydas Baranauskas. Full dll unhooking with c++. https://www.ired.team/offensive-security/defense-evasion/how-to-unhook-a-dll-using-c++, 2020-06. Visited: 2021-04-29.

[18] Peter Winter-Smith. Firewalker: A new approach to generically bypass user-space edr hooking. https://www.mdsec.co.uk/2020/08/firewalker-a-new-approach-to-generically-bypass-user-space-edr-hooking/, 2020-08. Visited: 2021-05-28.

[19] Sector7. Perun's fart - yet another unhooking method. https://blog.sektor7.net/#!res/2021/perunsfart.md, 2021-04-21. Visited: 2021-06-08.

[20] Fabian Mosch. A tale of edr bypass methods. https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/, 2021-01-31. Visited: 2021-05-22.

[21] Mantvydas Baranauskas. Bypassing cylance and other avs/edrs by unhooking windows apis. https://www.ired.team/offensive-security/defense-evasion/bypassing-cylance-and-other-avs-edrs-by-unhooking-windows-apis, 2021-01. Visited: 2021-05-31.

[22] George Nicolaou. Bypassing cylance and other avs/edrs by unhooking windows apis. http://rce.co/why-usermode-hooking-sucks-bypassing-comodo-internet-security/, 2012-05-13. Visited: 2021-06-01.

[23] Mitre Engenuity. Att&ck evaluations. https://attackevals.mitre-engenuity.org/, 2021. Visited: 2021-05-31.

[24] Tom Broumels. User mode unhooking test script. https://github.com/TomOS3/UserModeUnhooking, 2021-07-04. Visited: 2021-07-04.

[25] AVTEST. Test antivirus software for windows 10 - april 2021. https://www.av-test.org/en/antivirus/business-windows-client/windows-10/april-2021/, 2021-04. Visited: 2021-05-30.

[26] Oracle. Virtualbox version 6.1.22 for windows. https://download.virtualbox.org/virtualbox/6.1.22/VirtualBox-6.1.22-144080-Win.exe, 2021. Visited: 2021-05-29.

[27] Chris Long. Detectionlab. https://github.com/clong/DetectionLab, 2021. Visited: 2021-05-29.

[28] Microsoft Visual Studio Enterprise 2019 version 16.9.2. https://docs.microsoft.com/en-us/visualstudio/releases/2019/release-notes-v16.9#--visual-studio-2019-version-1692, 2021. Visited: 2021-05-29.

[29] Geoff McDonald. Process dump version 2.1. http://www.geoffmcdonald.ca/pd.html, 2021. Visited: 2021-05-29.

[30] HookShark64 version beta 0.1. https://www.unknowncheats.me/forum/pc-software/72799-hookshark64-beta-0-1-a.html, 2011. Visited: 2021-05-29.

[31] National Security Agency. Ghidra version 9.2. https://ghidra-sre.org/releaseNotes_9.2.html#9_2, 2021. Visited: 2021-05-29.

[32] Theodoros Apostolopoulos, Vasilios Katos, Kim-Kwang Raymond Choo, and Constantinos Patsakis. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems*, 116:393–405, 2021.

[33] Olaf Hartong. Sysmon modular. https://github.com/olafhartong/sysmon-modular, 2021. Visited: 2021-05-29.

[34] Matthew Eidelberg. Endpoint detection and response: How hackers have evolved. https://www.optiv.com/insights/source-zero/blog/endpoint-detection-and-response-how-hackers-have-evolved, 2021-02-02. Visited: 2021-05-31.

[35] Filip Olszak. Endpoint detection and response: How hackers have evolved. https://blog.redbluepurple.io/windows-security-research/kernel-tracing-injection-detection, 2021-04-07. Visited: 2021-05-31.

[36] ShengHao Ma. Wow hell: Rebuilding heavens gate. https://conference.hitb.org/hitbsecconf2021ams/sessions/wow-hell-rebuilding-heavens-gate/, 2021-05-27. Visited: 2021-06-01.

APPENDIX A - INLINE HOOKS PLACED BY ANTIVIRUS SOFTWARE

Table 6 shows the amount of hooked functions placed by antivirus products on three different applications.

		advapi32.dll	combase.dll	fltlib.dll	gdi32.dll	kernel32.dll	kernelbase.dll	ntdll.dll	rpcrt4.dll	sechost.dll	shell32.dll	user32.dll	win32u.dll
No antivirus	Notepad	0	0		0	0	0	0	0	0	0	0	0
	Console					0	0	0					
	Desktop	0	0		0	0	0	0	0	0		0	0
Comodo	Notepad	0	0	3	6	0	10	36	0	0	2	3	29
	Console	7	2	3	6	4	10	52	1	14	7	43	32
	Desktop	7	2	3	6	4	10	52	1	14	7	43	32
F-Secure	Notepad	0	0		0	0	0	0	0	0	0	0	0
	Console					1	11	0					
	Desktop	0	0		0	1	11	0	0	0		2	0
Sophos	Notepad	0	0		0	1	1	7	0	0	0	0	0
	Console					1	1	7					
	Desktop	0	0		0	1	1	7	0	0		0	0

Table 6: Inline hooks placed by AV products on MS Notepad, and two Visual Studio 2019 boilerplate applications, a Console and a Desktop application. If a DLL was not imported by an application, the corresponding cell is left empty.

APPENDIX B - HASHES OF INLINE HOOKED DLLS

The SHA256 hashes for the inline hooked DLLs are listed in Table 7.

DLL	SHA256 hash value of DLL
advapi32.dll	D9B3BB2AC2CE939DD485EDDD1184684698A9D7CA2A04F4E79DC32B2E806077E3
combase.dll	E2556FC888D8A419234C7559B2C2D255A958223AE69E8D0E7587F77CA688CDFE
fltlib.dll	353F8D4E647A11F235F4262D913F7BAC4C4F266EAC4601EA416E861AFD611912
gdi32.dll	4FC9460147D5A3AA1F9498863CF10BB7BE6099BEF1A3D55503164CA51E36E27E
kernel32.dll	DC1BCAE0A3DD5ED5C91C1A26C8DC9DEF93CA9926D3335226803479D27C3C0377
kernelbase.dll	733BD624C3BFC8F57FAD79835D31790F4489582352D5CB696B4C3717DB31E820
ntdll.dll	2CF67E1CC1E6F43637FDA35315FFE16B2CA140BCBA149944D5E4B8ECC49391B1
rpcrt4.dll	78C4D3F9607A399DCD96E9067A14F143E9838FB02E78E7775675BC7D46D8D703
sechost.dll	D5DE7C00A5BD8E766EB1ECAA651687E84A227ACDFC110847E3F595342953E29C
shell32.dll	BC8637E2F6D6E18CD60452498FE48DB54FF4742CE7252EE0953CB72F3A4A5E3D
user32.dll	6ECB526489C26424A7D8A8C58C770CC3E41377CCB3B87B5E126F21ADBEA316C0
win32u.dll	5462B14DCD51B06596F0B6A8154A36F8346C42D5C4DE6B31CFC1F4CA9977D662

Table 7: Hash values of the DLL files on the lab systems

APPENDIX C - UNHOOKING TECHNIQUES OVERVIEW

Table 8 shows known techniques that can be used to remove or evade inline user mode hooks.

Technique	Category
Original Thunk Tracing [18]	Removal
Perun's Fart [19]	Removal
Prologue Restoring [6, 14]	Removal
Rebuild function stubs with trampoline [6]	Removal
Section Remapping [15, 32]	Removal
Clone and load DLL from disk [6]	Evasion
DLL Parsing for using direct syscalls [6]	Evasion
New Heaven's Gate (WoW Hell) [36]	Evasion
Manually load DLL from disk [6, 16]	Evasion

Table 8: Known techniques for removal or evasion of user mode hooks.

APPENDIX D - FULL EXPERIMENTAL RESULTS

The results for all the experiments performed on F-Secure, Sophos and Comodo can be found in tables 9, 10 and 11 respectively.

F-Secure Computer Protection Premium 21.5		**Detectability**		**Effectiveness**	
Malware	**Unhooking technique**	**AV alert on payload**	**Payload exit code 0**	**Unhooking exit code 0**	**Hook removed**
Malicious File Copying	None	✗[1]	✓	-	-
	Interprocess Function Copying	✗[1]	✓	✓	✓
	Interprocess Section Copying	✗[1]	✓	✓	✓
Hook: kernelbase.dll	Peruns Fart	✗[1]	✓	✓	✗[3]
CopyFileExW	Prologue Restoring	✗[1]	✓	✓	✓
	Section Remapping	✗[1]	✓	✓	✓
Shellcode Injection - Basic	None	✓[2]	✗	-	-
	Interprocess Function Copying	✗	✓	✓	✓
	Interprocess Section Copying	✗	✓	✓	✓
Hook: kernelbase.dll	Peruns Fart	✓[2]	✗[3]	✓	✗[3]
CreateRemoteThreadEx	Prologue Restoring	✗	✓	✓	✓
	Section Remapping	✗	✓	✓	✓
Shellcode Injection - Section Mapping	None	✗	✓	-	-
	Interprocess Function Copying	✗	✓	✓	✗[7]
	Interprocess Section Copying	✗	✓	✓	✗[7]
Hook: ntdll.dll	Peruns Fart	✗	✓	✓	✗[7]
NtMapViewOfSection	Prologue Restoring	✗	✓	✓	✗[7]
	Section Remapping	✗	✓	✓	✗[7]

[1] Antivirus event related to the harmful content of copied files, not the payload itself.
[2] Reason: Exploit:W32/ShellCodeInjection.A!DeepGuard.
[3] Peruns Fart can only copy ntdll.dll because no other dll's are available for a sleeping process.
[7] Function not hooked by antivirus.

Table 9: Results for all experiments on a system provided with F-Secure.

Sophos Intercept X Advanced with EDR 10.8		Detectability			Effectiveness
Malware	**Unhooking technique**	**AV alert on payload**	**Payload exit code 0**	**Unhooking exit code 0**	**Hook removed**
Malicious File Copying Hook: kernelbase.dll CopyFileExW	None	✗[1]	✓	-	-
	Interprocess Function Copying	✗[1]	✓	✓	✗[7]
	Interprocess Section Copying	✗[1]	✓	✓	✗[7,8]
	Peruns Fart	✗[1]	✓	✓	✗[7]
	Prologue Restoring	✗[1]	✓	✓	✗[7]
	Section Remapping	✗[1]	✓	✓	✗[7]
Shellcode Injection - Basic Hook: kernelbase.dll CreateRemoteThreadEx	None	✗	✓	-	-
	Interprocess Function Copying	✗	✓	✓	✗[7]
	Interprocess Section Copying	✗	✓	✓	✗[7,8]
	Peruns Fart	✗	✓	✓	✗[7]
	Prologue Restoring	✗	✓	✓	✗[7]
	Section Remapping	✗	✓	✓	✗[7]
Shellcode Injection - Section Mapping Hook: ntdll.dll NtMapViewOfSection	None	✗	✓	-	-
	Interprocess Function Copying	✗	✗[4]	✓	✗[6]
	Interprocess Section Copying	✗	✗[4]	✓	✗[6]
	Peruns Fart	✗	✓	✓	✓
	Prologue Restoring	✗	✓	✓	✓
	Section Remapping	✗	✓	✓	✓

[1] Antivirus event related to the harmful content of copied files, not the payload itself.
[4] Execution failed because hooks are copied from source process refering to relative memory not existing in target process.
[6] Hook is overwritten by code containing hooks referring to nonexistent memory.
[7] Function not hooked by antivirus.
[8] Hook unchanged but other functions overwritten by code containing hooks referring to nonexistent memory.

Table 10: Results for all experiments on a system provided with Sophos.

Comodo Internet Security Pro 10		Detectability			Effectiveness
Malware	**Unhooking technique**	**AV alert on payload**	**Payload exit code 0**	**Unhooking exit code 0**	**Hook removed**
Malicious File Copying Hook: kernelbase.dll CopyFileExW	None	✗[1]	✓	-	-
	Interprocess Function Copying	✗[1]	✓	✓	✗[5]
	Interprocess Section Copying	✗[1]	✓	✓	✗[5]
	Peruns Fart	✗[1]	✓	✓	✗[3]
	Prologue Restoring	✗[1]	✓	✓	✓
	Section Remapping	✗[1]	✓	✓	✓
Shellcode Injection - Basic Hook: kernelbase.dll CreateRemoteThreadEx	None	✗	✓	-	-
	Interprocess Function Copying	✗	✓	✓	✗[7]
	Interprocess Section Copying	✗	✓	✓	✗[7]
	Peruns Fart	✗	✓	✓	✗[3,7]
	Prologue Restoring	✗	✓	✓	✗[7]
	Section Remapping	✗	✓	✓	✗[7]
Shellcode Injection - Section Mapping Hook: ntdll.dll NtMapViewOfSection	None	✗	✓	-	-
	Interprocess Function Copying	✗	✓	✓	✗[7]
	Interprocess Section Copying	✗	✓	✓	✗[7]
	Peruns Fart	✗	✓	✓	✗[7]
	Prologue Restoring	✗	✓	✓	✗[7]
	Section Remapping	✗	✓	✓	✗[7]

[1] Antivirus event related to the harmful content of copied files, not the payload itself.
[3] Peruns Fart can only copy ntdll.dll because no other dll's are available for a sleeping process.
[5] Hook is overwritten by code containing exactly the same hooks.
[7] Function not hooked by antivirus.

Table 11: Results for all experiments on a system provided with Comodo.