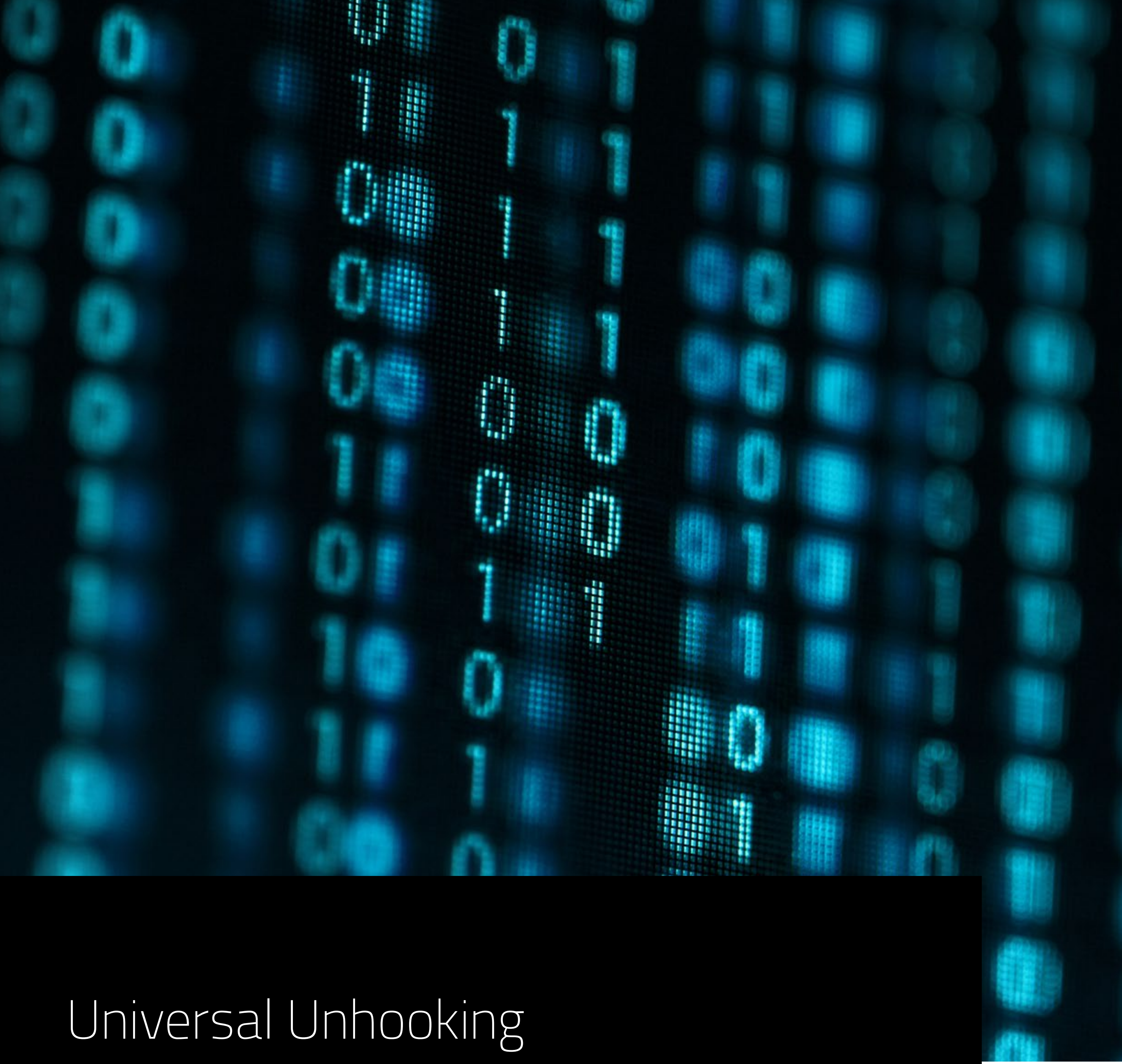


Universal Unhooking

Blinding Security Software

CYLANCE™

Code hooking is a technique used for redirecting a computer's execution flow to modify software. Essentially, a 'hook' is something that will allow the developer to see, view, and interact with something that is already going on in the system. Code hooks can perform a wide variety of functionality, both innocent and nefarious, including:

- Patching a bug
- Performance monitoring
- Disabling digital rights management systems
- Capturing keystrokes (AKA keylogging)
- Hiding processes and files (rootkits)

The antivirus (AV) industry uses code hooking to monitor processes for potentially malicious behavior, protect applications by injecting anti-exploitation checks, and isolate processes by sandboxing or virtualization. The technique can also be used by bad actors, for instance, to implement a rootkit to hide processes and network connections from the end-user and security software. Bad actors have also determined how to monetize the technique by hooking browsers to invisibly modify online banking sites to steal credentials or initiate fraudulent transfers.

In both 'good' and 'bad' uses, hooks are inserted by overwriting the first few bytes of a target function to hijack execution flow. Typically, a JMP or CALL is inserted to transfer execution to new code that will perform the new functionality and, depending on the intended behavior, may return to the original function to complete its execution.

A Pen Tester's Perspective

One of the first things a penetration tester performs after gaining initial access to a system is to dump password hashes. Exploits aren't always reliable, but having a working set of password hashes means pen testers can expand access and move laterally across an internal network. The meterpreter shell — part of the Metasploit framework — has a built-in capability to dump hashes with the hashdump command [0]. However, security software vendors have caught onto this technique and have implemented a variety of detection and prevention mechanisms to combat hash dumping.

Code hooking is one of these techniques: anti-exploitation software will insert code hooks into Windows APIs to detect and prevent the hashdump command from executing.

The penetration test would be quite tedious if it stopped there. If you had the capability to remove security hooks, you could evade detection and prevention, and continue deeper into the network.

The capability does exist. The following will demonstrate how to universally remove hooks in memory to bypass protection mechanisms.

A Brief Detour: Code Hooking

The diagram below depicts the general outline of code hooking. The first five bytes of the original function is overwritten with a JMP to a detour which contains the hook code to perform the new functionality. Once the hook code completes, it calls the trampoline that contains the original function bytes, which were overwritten, and jumps to the original function code. Other hooking methods may overwrite the address of a target function in a table (export address table, global offset table, procedure linkage table) to point to a new function that will execute instead of the original function.

Overview of Code Hooking

Original Function	
8b ff	mov edi, edi
55	push ebp
8b ec	mov ebp, esp
	<function code>
c3	ret

Hooked Function	
e9 ..	jmp detour
	...
	<function code>
c3	ret

Detour	
	<hook code>
	...
	call trampoline
c3	ret

Trampoline	
8b ff	mov edi, edi
55	push ebp
8b ec	mov ebp, esp
e9	jmp <function + 5>

PatchGuard

The most common method for instrumenting the Windows operating system to gather system behavioral information used to be through kernel-mode hooking, to redirect execution to the software vendor's instrumentation engine. Kernel drivers installed with security software would insert hooks into critical kernel functions to monitor the behavior of the system by inspecting parameters and return values.

However, most implementations contained bugs, which caused system instability and resulted in the infamous blue screen of death (BSOD).

Microsoft released Kernel Patch Protection - also known as PatchGuard - with the introduction of the 64-bit Windows OS in 2005, to protect the integrity of the Windows kernel. Microsoft's PatchGuard prevents modification to the Windows kernel and critical kernel data structures such as the interrupt descriptor table, global descriptor table, system service descriptor table, and much more.

PatchGuard also routinely scans for changes in the background and raises a BSOD when other software has compromised the kernel's integrity.

RETurning To User-Mode

While PatchGuard bypasses have been discovered [1][2][3] and even employed by malware such as the Uroboros rootkit [4], security software vendors must take a conservative approach and rely on user-mode hooking instead of kernel-mode hooking. This is because Microsoft could update PatchGuard at a moment's notice to block bypasses.

Security software will hook specific user space API functions that are commonly used by malware. For example, a code hook installed on winsock.connect can examine the IP and port of an outgoing network connection and decide whether the connection should be allowed or blocked. A combination of hooks installed on OpenProcess, VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread detect malicious process injection.

However, there's a downside to relying on user-mode hooks: they can be removed easily. At the same time, poorly implemented user-mode hooks are increasing the attack surface and introducing additional vulnerabilities, as documented by the enSilo research team [5].

Unhooking Programs in Memory

Removing user-mode hooks is a well-known technique that has been documented previously [6]. In the past, these techniques have focused on removing hooks from specific products by looking for known hooking patterns or signatures.

The universal unhooking technique approaches the problem by mimicking the Windows loader to load a replica of each binary (DLLs and EXEs) from disk and comparing them to the original image in memory. If the original image in memory is different from the replica, it is confirmed that it has been hooked or modified.

There are a few tricks when performing this technique:

- Imports must be manually resolved to avoid bringing in additional hooks
- API sets must be resolved to support Windows 7+
- Relocations must be performed with the base address of the original image in memory and not the replica

Resolving Imports

When an EXE or DLL is loaded, the Windows PE loader is responsible for resolving the addresses of imported functions. The universal unhooking code cannot use the Windows PE loader when loading the replica into memory, so it must manually resolve the address of imported functions. Windows provides the GetProcAddress API for performing these resolutions.

However, it may be hooked to provide the addresses of other hooked functions, which would defeat our purpose. Instead, the unhooking code must also implement the GetProcAddress functionality by walking the loaded modules list and iterating over the export address tables.

API Set Schema

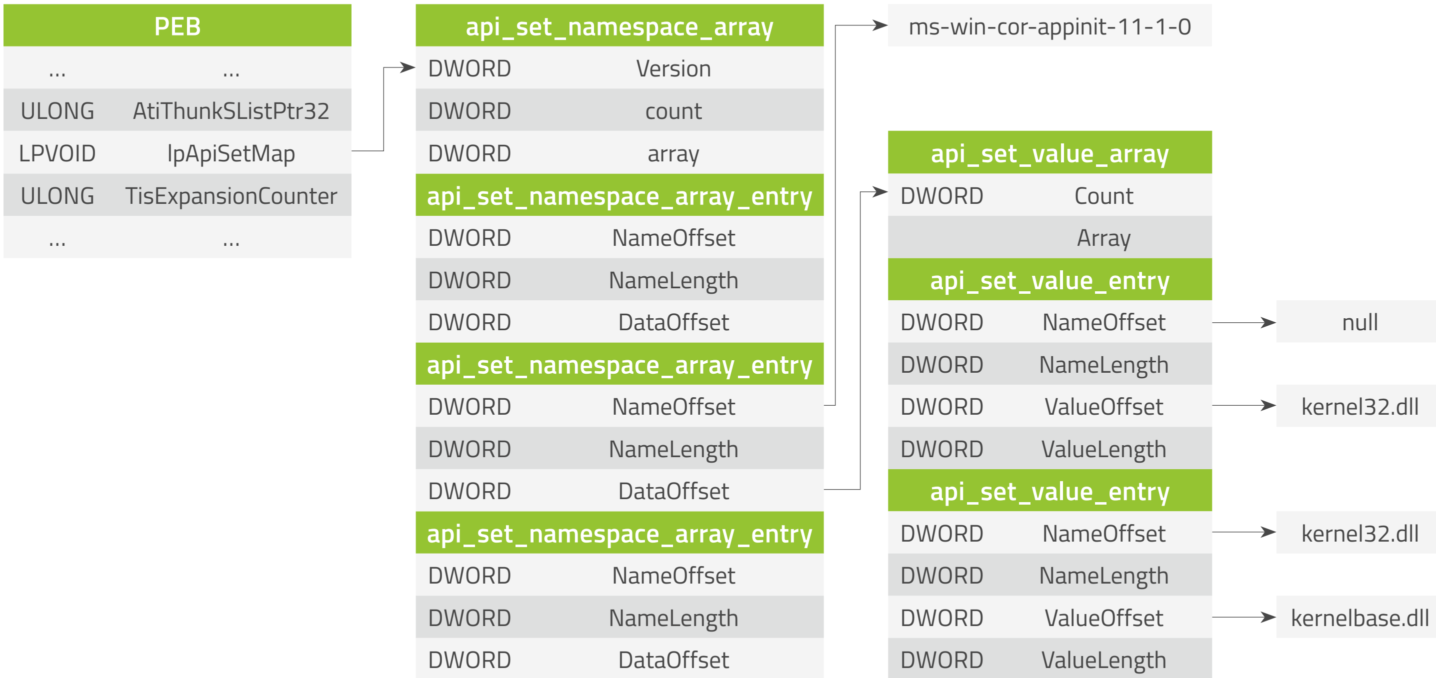
Microsoft introduced the concept of API sets starting with Windows 7 as part of their 'MinWin' effort to reorganize and clean up the internal Windows architecture [7]. API Sets are virtual DLLs — modules beginning with 'api-' or 'ext-' — that map to logical DLLs, which contain the actual implementation.

The unhooking code must support API sets when resolving the addresses of imported functions. For example, if the code needs to resolve the LoadAppInitDlls function, it must load api-ms-win-core-appinit-l1-1-0.dll.

The diagram below demonstrates the process for resolving a virtual DLL for API Set Schema version 2 (Win 7). The process environment block (PEB) contains the API set map pointer, lpApiSetMap, to an API_SET_NAMESPACE_ARRAY structure. Each array entry contains an offset (note: each offset is added to lpApiSetMap to obtain the virtual address), NameOffset, to a Unicode string containing the virtual DLL name without the api- prefix. Once the matching module name is found, the DataOffset points to an API_SET_VALUE_ARRAY structure. Each entry in the value structure contains an offset to the name of the logical DLL providing the implementation of the virtual DLL.

In this example, api-ms-win-core-appinit-l1-1-0.dll is redirected to kernel32.dll and subsequently redirected again to kernelbase.dll.

Resolving API Set Schemas for api-ms-win-core-appinit-l1-1-0.dll



For further details, Geoff Chappell provides an excellent resource on the evolution of API Set Schema [8].

Relocations

The Portable Executable (PE) file format supports address relocations by defining an IMAGE_BASE_RELOCATION table, which is pointed to by the IMAGE_DIRECTORY_ENTRY_BASERELOC header attribute. When a PE file such as an EXE or a DLL is loaded, the Windows PE loader iterates through each entry in the table and performs the necessary relocation calculations.

In a typical relocation operation, the PE loader subtracts the image base defined in the PE's optional header and adds in the base address at which the module was loaded. The universal unhooking code performs a similar operation but uses the base address of the original image instead of the base address of the replica. This creates a byte-exact replica of what was originally loaded by the Windows PE loader.

Unhooking

The unhooking code compares the byte-exact replica to the original image. If any changes are detected, the code will overwrite the original image's section with the replica's section, thereby removing all hooks installed within that section.

Packing It Up

The universal unhooking code is packaged up as a reflective DLL [9] to provide flexibility in usage. The reflective DLL can be injected into a process in a meterpreter session, appended to arbitrary binaries as a thread local storage (TLS) callback, or even inserted into a packed binary with UPX [10].

Meterpreter provides a reflective DLL injection module (post/windows/manage/reflective_dll_inject) for injecting the unhooking DLL on endpoints that have already been compromised, allowing the hashdump command to execute. The next step is to modify the meterpreter payload to embed the unhooking code so all meterpreter activity would be immune to user space hooks.

Borja Merino has a nifty Python script for injecting shellcode into a 32-bit Windows executable as a TLS callback [11]. The reflective unhooking DLL is fairly close to shellcode and with a little modification to the tlsInjector.py script to add some bootstrapping shellcode, we are able to inject the unhooking code into arbitrary 32-bit Windows executables as TLS callbacks.

Utilizing UPX

UPX is a popular packing utility used by malware for obfuscating and evading AV signatures. We extended UPX to insert our universal unhooking DLL as an additional section into the packed output binary and modified the decoder stub to call the reflective loader prior to jumping into the original binary.

In theory, this shouldn't work because the loaded binary in memory is now different (unpacked) when compared to disk, and our unhooking code should detect the change and replace it with the packed code from disk. However, our unhooking code only checks sections that are non-writable, *Sections of a UPX Packed Binary Appended with the Universal Unhooking DLL*

upx.x86.packed.exe

upx.x86.packed_dll.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Reloc Number	Linenumbers	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
UPX0	00154000	00001000	00000000	00000400	00000000	00000000	0000	0000	E0000080
UPX1	0006D000	00155000	0006CC00	00000400	00000000	00000000	0000	0000	E0000040
.rsrc	00001000	001C2000	00000400	0006D000	00000000	00000000	0000	0000	C0000040
UPX3	00012C00	001C3000	0012C00	0006D400	00000000	00000000	0000	0000	60000040

and UPX conveniently marks every section as writable, so it's ignored by the unhooking code. This provides us with the ability to unhook all the imports without interfering with the packed binary.

Now, anyone can pack the universal unhooking DLL in a variety of ways to bypass disk signatures and evade behavior-based detection systems.

This is merely a small sample of what is possible. Cylance will continue to explore new methods and techniques for employing the universal unhooking DLL. In addition, Cylance will be releasing its universal unhooking tool on its public GitHub repository soon!

References

[0] Safe, Reliable, Hash Dumping - https://community.rapid7.com/community/metasploit/blog/2010/01/01/safe-reliable-hash-dumping

[1] Bypassing PatchGuard on Windows x64 - http://uninformed.org/?v=3&a=3

[2] Subverting PatchGuard Version 2 - http://uninformed.org/?v=6&a=1

[3] PatchGuard Reloaded - http://uninformed.org/?v=8&a=5

[4] The Windows 8.1 Kernel Patch Protection - http://blog.talosintel.com/2014/08/the-windows-81-kernel-patch-protection.html

[5] Captain Hook: Pirating AVs to Bypass Exploit Mitigations - https://breakingmalware.com/vulnerabilities/captain-hook-pirating-avs-bypass-exploit-mitigations/

[6] Defeating Antivirus Real-time Protection From The Inside - https://breakdev.org/defeating-antivirus-real-time-protection-from-the-inside/

[7] Inside "MinWin": the Windows 7 Kernel Slims Down - http://arstechnica.com/information-technology/2009/11/inside-minwin-the-windows-7-kernel-slims-down/

[8] The API Set Schema - http://www.geoffchappell.com/studies/windows/win32/apisetschema/index.htm

[9] Reflective DLL Injection - http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf

[10] UPX - https://upx.github.io/

[11] TLS Injector: Running Shellcodes Through TLS Callbacks - http://www.shelliscoming.com/2015/06/tls-injector-running-shellcodes-through.html

+1-877-97DEFEND
proservices@cylance.com
www.cylance.com/consulting
18201 Von Karman Ave., Suite 700, Irvine, CA 92612

CYLANCE