ENSILO
SILO YOUR DATA FROM THREAT ACTORS

Captain Hook

Pirating AVs to Bypass Exploit Mitigations

By

enSilo Research Team

August 2016

TABLE OF CONTENTS

User-mode hooks are used by most of the end-point security vendors today, specifically Anti-Virus (AV) products, and Anti-Exploitation products such as EMET. Beyond their usage in security, hooks are used in other invasive applications such as Application Performance Management (APM) technologies to track performance bottlenecks.

Hooking itself is a very intrusive coding operation where function calls (mainly operating system functions) are intercepted in order to alter or augment their behavior.

Given the sensitivity of hooking implementations, we sought to find their robustness. For our research, we investigated about a dozen popular security products. Our findings were depressing – we revealed six different security problems and vulnerabilities stemming from this practice.

Our findings were depressing– we revealed six different security problems and vulnerabilities stemming from this practice.

HOOKING IN A NUTSHELL

The use of hooks allows intrusive software to intercept and monitor sensitive API calls. In particular, security products use hooking to detect malicious activity. For example, most Anti-Exploitation solutions monitor memory allocation functions, such as VirtualAlloc and VirtualProtect, in an attempt to detect vulnerability exploitation.

On the other side of the security spectrum, hooks are also used extensively by malware for various nefarious purposes, the most popular being Man-In-The-Browser (MITM) attacks.

The most common form of hooking in real-life products, especially security products, is inline hooking. Inline hooking is performed by overwriting the first few instructions in the hooked function and redirecting it to the hooking function. Although there are other forms of hooking, such as Import Address Table (IAT)-hooking, this research focuses only on inline hooks.

Hooking in user-mode is usually implemented within a DLL which is loaded into a process address space. We refer to this DLL as the “Hooking Engine”.

In this paper we dive into inline user-mode hooking. We also take a deep look into injection techniques, specifically kernel-to-user injections, since these are usually used to load the hooking engine into the process address space. Kernel-to-user injections are not trivial to implement and accordingly, some of the most severe issues that we found were not in the hooking engine itself but rather in the implementation of the kernel-to-user injection.

UNDER-THE-HOOD OF INLINE USER-MODE HOOKING

Although hooking is quite common and there are several common hooking libraries out there, such as Microsoft Detours, it seems that most security vendors develop their own hooking engines. That said, apart from a few exceptions, most of these in-house inline hooking implementations are pretty much similar.

INLINE HOOKING ON 32-BIT PROCESSES

Hooking 32-bit functions is straight forward most of the time. The hooking engine disassembles the first few instructions of the target function in order to replace it with a 5 byte jmp instruction. After at least 5 bytes of disassembled instructions are found, the hooking engine copies the instructions to a dynamically allocated code stub and follows with a jmp which returns the code to the original function. At that stage, the hooking engine overwrites the instructions with a jmp to the actual hooking function.

For example, let's see how a hook on InternetConnectW looks in a windbg:

```
0:000:x86> u WININET!InternetConnectW
WININET!InternetConnectW:
77090ec0 8bff            mov     edi,edi
77090ec2 55              push    ebp
77090ec3 8bec            mov     ebp,esp
77090ec5 83e4f8          and     esp,0FFFFFFF8h
77090ec8 83ec7c          sub     esp,7Ch
77090ecb 53              push    ebx
77090ecc 56              push    esi
77090ecd 57              push    edi
```

Figure 1: InternetConnectW before the hook is set (Marked in red are the instructions that will be replaced)

```
0:014:x86> u WININET!InternetConnectW
WININET!InternetConnectW:
77090ec0 e97b7a0e89      jmp     00178940
77090ec5 83e4f8          and     esp,0FFFFFFF8h
77090ec8 83ec7c          sub     esp,7Ch
77090ecb 53              push    ebx
77090ecc 56              push    esi
77090ecd 57              push    edi
```

Figure 2: After the hook is set

We can see that the jmp instruction leads to 0x178940, which is the hooking function itself. Disassembling the code at 0x178940 provides:

```
00178940 55              push    ebp
00178941 8bec            mov     ebp,esp
00178943 53              push    ebx
00178944 8b5d1c          mov     ebx,dword ptr [ebp+1Ch]
00178947 56              push    esi
00178948 57              push    edi
00178949 ff7524          push    dword ptr [ebp+24h]
0017894c 33f6            xor     esi,esi
0017894e ff7520          push    dword ptr [ebp+20h]
00178951 53              push    ebx
00178952 ff7518          push    dword ptr [ebp+18h]
00178955 ff7514          push    dword ptr [ebp+14h]
00178958 ff7510          push    dword ptr [ebp+10h]
0017895b ff750c          push    dword ptr [ebp+0Ch]
0017895e ff7508          push    dword ptr [ebp+8]
00178961 ff152cf21900    call    dword ptr [0019f22c]
```

Figure 3: Disassembled code at 0x178940

This code calls the original InternetConnectW function, leading to:

```
0:014:x86> u poi(0019f22c)
03110000 8bff            mov     edi,edi
03110002 55              push    ebp
03110003 8bec            mov     ebp,esp
03110005 e9bb0ef873      jmp     WININET!InternetConnectW+0x5 (77090ec5)
0311000a 90              nop
0311000b 90              nop
0311000c 90              nop
```

Figure 4: Original instructions of the function followed by a jmp

As shown, the original instructions of the function are followed by a jmp to the original function.

OTHER TECHNIQUES

There are also other ways to achieve function hooking:

- **One Byte Patching** – This technique is most used by malware. The idea is simple, hooking is performed by patching the first byte with an illegal instruction (or with an instruction that generates an exception) and installing an exception handler. Whenever the code executes, an exception will occur whereas the exception handler will handle it and act as the hooking function.
- **Microsoft Hot-Patching** – Hot-Patching was developed by Microsoft to enable patching without the need to reboot. The patching itself is done through the inline-hooking of the relevant function. To make things easy, Microsoft decided to keep a 5-bytes' space between functions and change the first instruction to a 2-byte NOP, specifically mov edi, edi instructions.

```
0:027> ub kernelbase!loadlibraryW+5 L8
KERNELBASE!CreateSemaphoreExW+0x9b:
7532b61b cc              int     3
7532b61c cc              int     3
7532b61d cc              int     3
7532b61e cc              int     3
7532b61f cc              int     3
KERNELBASE!LoadLibraryW:
7532b620 8bff            mov     edi,edi
7532b622 55              push    ebp
7532b623 8bec            mov     ebp,esp
```

Figure 5: Prior to hot-patching

The patch is done by replacing the 2-byte NOP with a short jmp instruction and replacing the 5-byte gap with a long jmp. This way the hooking code doesn't need to copy any of the original instructions.

```
0:033> ub kernelbase!loadlibraryW+5 L4
KERNELBASE!CreateSemaphoreExW+0x9b:
74dfb61b e900d00080      jmp     f4e08620
KERNELBASE!LoadLibraryW:
74dfb620 ebf9            jmp     KERNELBASE!CreateSemaphoreExW+0x9b (74dfb61b)
74dfb622 55              push    ebp
74dfb623 8bec            mov     ebp,esp
```

Hooking Function

Figure 6: After hot-patching

POSSIBLE COMPLICATIONS

In other 32-bit hooking scenarios, hooking is not that straight forward. For example:

- **Relative instructions** - If one of the instructions is a relative call/jmp it must be fixed before being copied.
- **Very short functions** – If a function is less than 5 bytes it might be hard to patch without overriding adjacent function.
- **Jmp/Jxx to function's start** – If some instruction in the function jumps back to the start of the function, the instruction will jump to the middle of the jmp instruction, resulting in a crash. This scenario is very difficult to solve without the full disassembly of the target function (or through one byte patch). However, this scenario is extremely rare.

A nice read on possible hooking issues can be found in Binary Hooking Problems by Gil Dabah.

INLINE HOOKING ON 64-BIT PROCESSES

Hooking on 64-bit processes is a bit more difficult than on 32-bit because the address space is much larger. This means that 5 bytes jmp instruction might not be enough in order to install a x64 hook since it is limited to a 2GB range from the its location.

There are several solutions to this problem, some of them are described in Trampolines in X64 by Gil Dabah.

The most common solution to this issue is to allocate code stub within 2GB range from the hooked function and use the following code template:

```
MOV RAX, <Hooking Function>
JMP RAX
```

For example, let's take a look at a hook on the 64-bit version of InternetConnectA.

```
0:000> u WININET!InternetConnectA
WININET!InternetConnectA:
000007fe`fe3b70a0 48895c2408      mov     qword ptr [rsp+8],rbx
000007fe`fe3b70a5 48896c2410      mov     qword ptr [rsp+10h],rbp
000007fe`fe3b70aa 4889742418      mov     qword ptr [rsp+18h],rsi
000007fe`fe3b70af 57              push    rdi
000007fe`fe3b70b0 4154            push    r12
000007fe`fe3b70b2 4155            push    r13
000007fe`fe3b70b4 4156            push    r14
```

Figure 7: The original InternetConnectA function

```
0:009> u WININET!InternetConnectA
WININET!InternetConnectA:
000007fe`fe3b70a0 e95b7fe4ff      jmp     000007fe`fe1ff000
000007fe`fe3b70a5 58              pop     rax
000007fe`fe3b70a6 90              nop
000007fe`fe3b70a7 90              nop
000007fe`fe3b70a8 90              nop
000007fe`fe3b70a9 90              nop
000007fe`fe3b70aa 4889742418      mov     qword ptr [rsp+18h],rsi
000007fe`fe3b70af 57              push    rdi
```

Figure 8: The function after the hook is set.

As shown, the function jumps to 0x7fefe1ff000.

```
0:009> u 00007fe`fe1ff000
000007fe`fe1ff000 48b8c094006800000000 mov rax,00000000`680094c0
000007fe`fe1ff00a ffe0            jmp     rax
000007fe`fe1ff00c 90              nop
000007fe`fe1ff00d 90              nop
```

Figure 9: Disassembling the code in address0x7fefe1ff000

If we follow the hooking function like we did in the 32-bit version we get to the following code stub which redirects the execution back to the original function:

```
00000000`00380000 48895c2408      mov     qword ptr [rsp+8],rbx
00000000`00380005 48896c2410      mov     qword ptr [rsp+10h],rbp
00000000`0038000a 50              push    rax
00000000`0038000b 48b8a5703bfefe070000 mov rax,offset WININET!InternetConnectA+0x5 (000007fe`fe3b70a5)
00000000`00380015 ffe0            jmp     rax
```

Figure 10: 64-bit code stub

OTHER TECHNIQUES

There are also other ways to achieve function hooking:

- **6-Byte Patching** – It is possible to avoid using trampolines by patching 6-bytes instead of 5 bytes, and making sure that the target is in a 32-bit address space. The idea is simply to use a push-ret instructions to do the jmp. This is how it looks like:

```
0:004> u kernelbase!loadlibraryA
KERNELBASE!LoadLibraryA:
00007ffc`9c8d8760 6800000300      push    30000h
00007ffc`9c8d8765 c3              ret
00007ffc`9c8d8766 89742410        mov     dword ptr [rsp+10h],esi
00007ffc`9c8d876a 57              push    rdi
```

Figure 11: 6-byte patching

- **Double Push (Nikolay Igotti) –** One of the problem of the classic techniques is that it trashes the rax register. One way to avoid it while still being able to jump anywhere in the address space is by pushing the lower 4-byte of the address into the stack and then copying the high 4-bytes of the address into the stack and then returning to that address.

```
0:004> u kernelbase!loadlibrarya
KERNELBASE!LoadLibraryA:
00007ffc`9c8d8760 6800000300          push    30000h
00007ffc`9c8d8765 c7442404fc7f0000 mov     dword ptr [rsp+4],7FFCh
00007ffc`9c8d876d c3                  ret
00007ffc`9c8d876e 20488b              and     byte ptr [rax-75h],cl
```

Figure 12: Double-push patching

POSSIBLE COMPLICATIONS

Complications in 64-bit hooking are similar to those in 32-bit hooking. However, since 64-bit code supports an instruction-pointer relative instructions there is a greater chance that the hooking engine will need to fix Instruction-pointer relative code. For example:

```
MOV RAX, QWORD [RIP+0x15020]
```

INJECTING THE HOOK ENGINE

Regardless of the way the hooking engine is implemented, a prerequisite for it to do its job is to inject it into the target process. Most vendors use kernel-to-user DLL injections to perform this. In this section we cover the most common methods used by security vendors.

Import Injection

This method is quite common and is relatively clean as it doesn't require any code modifications. As far as we know this injection technique was never used by malware.
It works by adding an import to the main image. These are the steps for import injection:

1. Register load image callback using PsSetLoadImageNotifyRoutine and wait for main module to load.
2. After the main module is loaded, the import table is copied to a different location and a new row that imports the hook engine is added to the beginning of the table. The RVA of the import table is modified to point to the new table. This is how it looks like in Internet Explorer:

```
0:000> !dh iexplore

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       5 number of sections
53F262AC time date stamp Mon Aug 18 23:31:40 2014

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
     102 characteristics
            Executable
            32 bit word machine

OPTIONAL HEADER VALUES
     10B magic #
   11.00 linker version
    3A00 size of code
   BEA00 size of initialized data
       0 size of uninitialized data
    1DDD address of entry point
    1000 base of code
         ----- new -----
00000000008c0000 image base
    1000 section alignment
     200 file alignment
       2 subsystem (Windows GUI)
    6.03 operating system version
    6.03 image version
    6.01 subsystem version
   C6000 size of image
     400 size of headers
   CAEE4 checksum
0000000000100000 size of stack reserve
000000000000e000 size of stack commit
0000000000100000 size of heap reserve
0000000000001000 size of heap commit
    8040  DLL characteristics
            Dynamic base
            Terminal server aware
       0 [         0] address [size] of Export Directory
FF7C0000 [        8C] address [size] of Import Directory
    7000 [     BD408] address [size] of Resource Directory
       0 [         0] address [size] of Exception Directory
   C2800 [      3CB8] address [size] of Security Directory
   C5000 [       328] address [size] of Base Relocation Directory
    4828 [        38] address [size] of Debug Directory
       0 [         0] address [size] of Description Directory
       0 [         0] address [size] of Special Directory
       0 [         0] address [size] of Thread Storage Directory
    2D88 [        40] address [size] of Load Configuration Directory
       0 [         0] address [size] of Bound Import Directory
    6000 [       138] address [size] of Import Address Table Directory
    45E0 [        A0] address [size] of Delay Import Directory
       0 [         0] address [size] of COR20 Header Directory
       0 [         0] address [size] of Reserved Directory
```

Import Directory RVA is out of image

Figure 13: Internet Explorer patched import table

This is the new import table:

The new row

```
0:000:x86> dd /c5 80000
00080000  ff7c009c ffffffff ffffffff ff7c00b4 ff7c008c
00080014  00006230 00000000 00000000 00006224 00006000
00080028  00006294 00000000 00000000 00006214 00006064
0008003c  00006328 00000000 00000000 000061e8 000060f8
00080050  00006348 00000000 00000000 000061d8 00006118
00080064  00006360 00000000 00000000 000061b0 00006130
```

Figure 14: The new import table

3. When the module completes loading, the RVA of the original import table is restored.

ENTRYPOINT PATCHING

To the best of our knowledge, this kind of injection method was first used by the infamous Duqu malware and is well documented. It is also used by security vendors.

These are the steps for entrypoint patching:

1. Register load image callback using PsSetLoadImageNotifyRoutine and wait for main module to load.
2. Read the instructions from the entrypoint and allocate a payload to load the hook engine.

Patch the entry point with a jmp to the payload. This is how entry point patching looks like in Internet Explorer:

```
iexplore+0x1ddd:
00c51ddd e91ee257ff      jmp     001d0000
00c51de2 e955f9ffff      jmp     iexplore+0x173c (00c5173c)
00c51de7 90              nop
00c51de8 90              nop
00c51de9 90              nop
00c51dea 90              nop
00c51deb 90              nop
00c51dec 8bff            mov     edi,edi
0:000:x86> uf 001d0000
001d0000 55              push    ebp
001d0001 8bec            mov     ebp,esp
001d0003 83ec48          sub     esp,48h
001d0006 eb50            jmp     001d0058
```

JMP to the payload

Figure 15: Internet Explorer patched entrypoint

3. When the payload executes, it first loads the hooking engine and then restores the bytes that were copied from the original image.

```
iexplore+0x1ddd:
00c51ddd e91ee257ff      jmp     001d0000
00c51de2 e955f9ffff      jmp     iexplore+0x173c (00c5173c)
00c51de7 90              nop
00c51de8 90              nop
00c51de9 90              nop
00c51dea 90              nop
00c51deb 90              nop
00c51dec 8bff            mov     edi,edi
0:000:x86> uf 001d0000
001d0000 55              push    ebp
001d0001 8bec            mov     ebp,esp
001d0003 83ec48          sub     esp,48h
001d0006 eb50            jmp     001d0058

001d0058 6a40            push    40h
001d005a 6808001d00      push    1D0008h
001d005f 8d45b8          lea     eax,[ebp-48h]
001d0062 50              push    eax
001d0063 b840238077      mov     eax,offset ntdll32!memcpy (77802340)
001d0068 ffd0            call    eax
001d006a 8d45b8          lea     eax,[ebp-48h]
001d006d 50              push    eax
001d006e b8f3487276      mov     eax,offset kernel32!LoadLibraryW (767248f3)
001d0073 ffd0            call    eax
001d0075 8be5            mov     esp,ebp
001d0077 5d              pop     ebp
001d0078 55              push    ebp
001d0079 8bec            mov     ebp,esp
001d007b 83ec08          sub     esp,8
001d007e c745f800000000  mov     dword ptr [ebp-8],0
001d0085 c745fc02000000  mov     dword ptr [ebp-4],2
001d008c c745f8dd1dc500  mov     dword ptr [ebp-8],offset iexplore+0x1ddd (00c51ddd)
001d0093 8d45fc          lea     eax,[ebp-4]
001d0096 50              push    eax
001d0097 6a40            push    40h
001d0099 6805000000      push    5
001d009e 8b4df8          mov     ecx,dword ptr [ebp-8]
001d00a1 51              push    ecx
001d00a2 b827437276      mov     eax,offset kernel32!VirtualProtect (76724327)
001d00a7 ffd0            call    eax
001d00a9 6805000000      push    5
001d00ae 68df001d00      push    1D00DFh
001d00b3 68dd1dc500      push    offset iexplore+0x1ddd (00c51ddd)
001d00b8 b840238077      mov     eax,offset ntdll32!memcpy (77802340)
001d00bd ffd0            call    eax
001d00bf 8d55fc          lea     edx,[ebp-4]
001d00c2 52              push    edx
001d00c3 8b45fc          mov     eax,dword ptr [ebp-4]
001d00c6 50              push    eax
001d00c7 6805000000      push    5
001d00cc 8b4df8          mov     ecx,dword ptr [ebp-8]
001d00cf 51              push    ecx
001d00d0 b827437276      mov     eax,offset kernel32!VirtualProtect (76724327)
001d00d5 ffd0            call    eax
001d00d7 8be5            mov     esp,ebp
001d00d9 5d              pop     ebp
001d00da e9fe1ca800      jmp     iexplore+0x1ddd (00c51ddd)
```

Load the hooking engine

Restore the code of the entrypoint

Jump back to the entrypoint

Figure 16: Restoring the bytes from the original image

User-APC

Kernel-to-user DLL injection using User Mode APC (Asynchronous Procedure Call) is probably the most documented and common method. This method was also extensively used by malware, TDL and Zero-Access for example.

For detailed information on this injection method we refer the reader to:

- http://www.opening-windows.com/techart_windows_vista_apc_internals2.htm
- http://rsdn.ru/article/baseserv/InjectDll.xml

This is how it works:

1. Register load image callback using PsSetLoadImageNotifyRoutine and wait for the target module to load.
2. Once the module is loaded, a payload for loading the hook engine is injected into the process and a function that will be called during the startup of the process is patched with a jmp or push/ret to the payload. On user32.dll the patched function is used is usually UserClientDllInitialize. On ntdll.dll the patched function is usually LdrLoadDLL. In this case, the push/ret sequence is used to divert execution to the injected payload.

```
0:000> u ntdll32!LdrLoadDll
ntdll32!LdrLoadDll:
00000000`77e0c4dd 6800000077      push    77000000h
00000000`77e0c4e2 c3              ret
00000000`77e0c4e3 cc              int     3
00000000`77e0c4e4 cc              int     3
```

Figure 17: LdrLoadDLL is used for injection

3. Once the payload executes it loads the hook engine and restores the original code in the patched function.

THE SECURITY ISSUES OF HOOKING

As stated above hooking has many benefits and is extensively used by many security vendors. However, hooking is also a very intrusive operation and implementing it correctly is not a simple matter.
Our research of more than a dozen security products revealed six separate security issues stemming from hooking-related implementations.

1. UNSAFE INJECTION

Severity: Very High
Affected Underlying Systems: All Windows versions
Description: This issue is a result of a bad DLL injection implementation. We have seen two cases of this issue which although had the same effect, differed in their technical details.

Description: This issue is a result of a bad DLL injection implementation. We have seen two cases of this issue which although had the same effect, differed in their technical details.

- **LoadLibrary from relative path:** In this case, the implementation uses the entrypoint patching injection method to load its hooking engine. The problem is that the DLL isn't loaded using a full path, making injected processes vulnerable to DLL hijacking vulnerability. An attacker also uses this as a persistence mechanism by placing a malicious DLL in the folder of the target process.
- **Unprotected injected DLL file:** In this case, the vendor loads the DLL using a full path but the DLL is placed in the %appdata%\..\Local\Vendor folder. The problem is that an attacker could replace the DLL with a malicious DLL thus causing the vendor to load the malicious DLL into every injected process.

Impact: In both cases, the attacker could use the affected implementation as a way to inject into most processes in system. This is a very convenient way to achieve persistency on the target system.

Exploitability: In both cases, exploitation of this issue is very simple. Although we believe that most attackers will not use vendor specific persistency mechanisms, security vendors should not weaken the integrity of the operating system.

2. PREDICTABLE RWX CODE STUBS (UNIVERSAL)

Severity: Very High
Affected Underlying Systems: All Windows versions
Description: In this case, the implementation uses a constant address - both for 32-bit and 64-bit processes, to allocate its injection code stub and leaves it as RWX. We have seen this issue only with one vendor. We decided not to show the exact code stub of the vendor to avoid exploitation of the issue.

Impact: An attacker can leverage this issue as part of the exploitation process by overwriting the code of the injection code stub with malicious code. Since the code stub also contains addresses of system functions it also causes the following issues:

- **Bypassing ASLR:** Most of these code stubs contain addresses of important system functions, such as LdrLoadDll, NtProtectVirtualMemory and more. These functions can be very useful as part of an exploitation process. In the cases we researched, it was also possible to leak the address of ntdll.dll.
- **Bypassing Hooks:** In cases where the hooks code stubs are allocated at a constant address it is possible to easily bypass the hook by calling directly to the function prolog. Note that in all the cases we saw the offsets of the code stubs were at a constant offset.
- **Code Reuse:** An attacker can also use the code in these code stubs as part of a code reuse attack. For example, an attacker can build a ROP chain that uses the part of the code which is used for loading the hook engine DLL. Attackers can manipulate the arguments in a way that their own DLL will be loaded.

All these issues make it possible to easily exploit vulnerabilities that will be otherwise very hard to exploit.

Exploitability: Past research of ours showed that these kind of issues are significant by weaponizing an old vulnerability in Adobe Acrobat Reader v.9.3 CVE-2010-0188.

Later that year, on September 22, Tavis Ormandy from ProjectZero wrote a very interesting post, "Kaspersky: Mo Unpackers, Mo Problems." about a vulnerability he discovered in Kaspersky that showed that these threats are real. To exploit the vulnerability he found, Tavis used a second flaw in Kaspersky which allocated RWX memory in a predictable address. To quote from Tavis's blog "Kaspersky have enabled /DYNAMICBASE for all of their modules which should make exploitation unreliable. Unfortunately, a few implementation flaws prevented it from working properly."

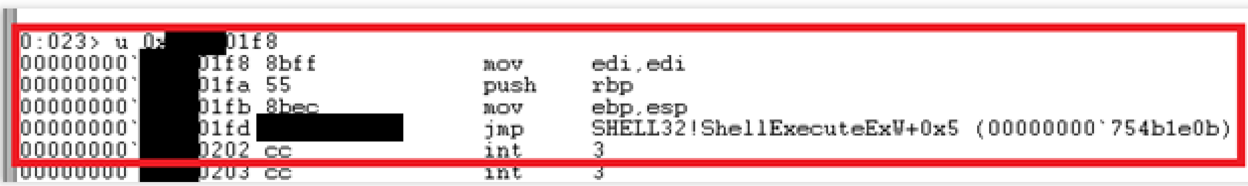
3. PREDICTABLE RX CODE STUBS (UNIVERSAL)

Severity: High
Affected Underlying Systems: All Windows versions
Description: This issue usually occurs when the implementation uses a constant address to allocate its injection code stub. One vendor we researched also uses a constant address to allocate the code stubs for its hooks.
Impact: Depending on the exact implementation, an attacker can leverage this to bypass ASLR, bypass Hooks or for code reuse as described in the previous issue (Predictable RWX Code Stubs - System independent).

Exploitability: This issue is very simple to exploit. All an attacker has to do is use the information in the hardcoded address. Moreover, in all the cases that we have seen, the address was constant for both 32-bit and 64-bit processes. In most cases, it is also possible to use these code stubs to inject DLL into the target process using methods similar to the ones described in a former research of ours, [Injection On Steroids](#).

Technical Breakdown

Let's see how it looks in a vulnerable hooking engine. In this case, the hooks are set in Internet-Explorer and always at a constant address. An attacker can simply call 0xXXXX01f8 in order to call ShellExecuteExW.



```
0:023> u 0x[redacted]01f8
00000000`[redacted]01f8 8bff        mov     edi,edi
00000000`[redacted]01fa 55          push    rbp
00000000`[redacted]01fb 8bec        mov     ebp,esp
00000000`[redacted]01fd [redacted]  jmp     SHELL32!ShellExecuteExW+0x5 (00000000`754b1e0b)
00000000`[redacted]0202 cc          int     3
00000000`[redacted]0203 cc          int     3
```

4. PREDICTABLE RWX CODE STUBS (ON WINDOWS 7 AND BELOW)

Severity: High

Affected Underlying Systems: Windows 7 and below

Description: This issue is very common and was described thoroughly in our blog post “[Vulnerability Patching: Learning from AVG on Doing it Right](#)”, as well as in a follow-up blog post 6 months later “[Sedating the Watchdog: Abusing Security Products to Bypass Mitigations](#)”. In all the cases we have seen, the issue was caused by the kernel-to-user dll injection and not by the hooking engine itself.

Impact: Similar to the above “Predictable RX Code Stubs (System independent)” issue.
The impact severity is lower here, since not all version of the operating system are affected.

Exploitability: Similar to the above “Predictable RX Code Stubs (System independent)” issue.

5. RWX HOOK CODE STUBS

Severity: Medium

Affected Underlying Systems: All Windows versions

Description: This is the most common issue in the hooking engines we researched. Most hooking engines leave their hook code stubs as RWX. We assume that the main reason for this is to avoid changing the code stub page protection whenever a new hook is set.

Impact: This can potentially be used by an attacker as part of exploitation process by overwriting the code stubs with malicious code. Overwriting such stubs can make it much easier for an attacker to bypass exploit mitigations such as Windows 10 Control-Flow-Guard (CFG) or Anti-Exploitation hooks. For example, an attacker that achieved arbitrary read/write in the target process may find the hook stub by following the hook's code and overwriting it. At that stage, the attacker only needs to trigger the execution of the hooked function (or even directly call the hook stub) in order to achieve code execution, effectively bypassing CFG mitigation.

Exploitability: We believe that an attacker that achieved arbitrary read/write will whatever find a way to complete the exploit without taking advantage of such an issue. Thus, it is unlikely that an attacker will actually exploit this issue in a real-life scenario. That said, we believe that security vendors should do their best not to weaken system's protections.

Technical Breakdown

Let's see how it looks in a vulnerable hooking engine. In this case, the hook is set on LdrLoadDLL function:

```
0:028:x86> u ntdll_76f70000!LdrLoadDll
ntdll_76f70000!LdrLoadDll:
76fac4dd e9163d0889       jmp     000301f8
76fac4e2 a10cf7f976       mov     eax,dword ptr [ntdll_76f70000!wcsnicmp+0xb1 (76f9f70c)]
76fac4e7 83ec0c           sub     esp,0Ch
76fac4ea 53               push    ebx
76fac4eb 83c801           or      eax,1
```

Figure 18: The hooking engine in windbg

If we check the permissions on the jmp target we will see that its permissions are RWX:

```
0:028:x86> !address 000301f8
Usage:                  <unclassified>
Allocation Base:        00030000
Base Address:           00030000
End Address:            0003b000
Region Size:            0000b000
Type:                   00020000    MEM_PRIVATE
State:                  00001000    MEM_COMMIT
Protect:                00000040    PAGE_EXECUTE_READWRITE
```

Figure 19: Permissions on the jmp target

6. RWX HOOKED MODULES:

Severity: Medium

Affected Underlying Systems: All Windows versions

Description: Some hooking engines leave the code of the hooked modules as RWX. This happens both as part of the initial dll injection code and in the hooking engine code. This issue is not very common and frankly, the appearance of this issue took us by surprise since we didn't even look for it given that we couldn't think of any good reason for a hooking engine to be implemented this way.

Impact: An attacker can leverage this issue as part of the exploitation process by overwriting the code of the hooked modules with malicious code, thus simplifying the bypassing of Windows' mitigations such as Windows 10 Control-Flow-Guard.
For example, an attacker that achieved arbitrary read/write in the target process may then find the hooked code and overwrite those permissions. At that stage, the attacker only needs to trigger the execution of the hooked function in order to achieve code execution, effectively bypassing CFG mitigation.

Exploitability: We believe that an attacker that achieved arbitrary read/write will whatever find a way to complete the exploit without taking advantage of such an issue. Thus, it is unlikely that an attacker will actually exploit this issue in a real-life scenario. That said, we believe that security vendors should do their best not to weaken system's protections.

Technical Breakdown

As an example, we show how the issue appears as part for kernel-to-user mode DLL injection. Here, the LdrLoadDll is used to inject the hooking engine.

```
0:000> u ntdll!ldrloaddll
ntdll!LdrLoadDll:
77be2576 6813040178      push    78010413h
77be257b c3              ret
77be257c cc              int     3
77be257d 90              nop
77be257e 48              dec     eax
77be257f 78bd            js      ntdll!RtlLengthRequiredSid+0x16 (77be253e)
77be2581 7753            ja      ntdll!LdrLoadDll+0x60 (77be25d6)
77be2583 56              push    esi
```

Figure 20: Hooking engine injection using LdrLoadDll in a windbg

As shown, the LdrLoadDLL was patched with a push-ret sequence in order to jump to the code stub which is located at 0x78919413. If we let windbg run we can see that the original code was restored:

```
                         jmp     ntdll!LdrLoadDll+0xaa (77be2620)
0:011> u ntdll!LdrLoadDll
ntdll!LdrLoadDll:
77be2576 8bff            mov     edi,edi
77be2578 55              push    ebp
77be2579 8bec            mov     ebp,esp
77be257b 51              push    ecx
77be257c 51              push    ecx
77be257d a14878bd77      mov     eax,dword ptr [ntdll!RtlUpcaseUnicodeChar+0x51 (77bd7848)]
77be2582 53              push    ebx
77be2583 56              push    esi
```

Figure 21: the original code is restored

However, when we check the permissions we can see that the code is still RWX:

```
Usage:                  Image
Allocation Base:        77b80000
Base Address:           77be2000
End Address:            77be3000
Region Size:            00001000
Type:                   01000000     MEM_IMAGE
State:                  00001000     MEM_COMMIT
Protect:                00000040     PAGE_EXECUTE_READWRITE
More info:              lmv m ntdll
More info:              !lmi ntdll
More info:              ln 0x77be2576
```

Figure 22: Code permissions were not restored

3RD PARTY HOOKING ENGINES

As we showed, implementing a robust hooking engine is not a simple task. For this reason many vendors choose to buy a commercial hooking engine or just use an open-source engine. Doing so saves the vendor a lot of development and QA time. It's also clear that the implications of security issues in a wide-spread hooking engine are much more serious for the following reasons:

- **Affects Multiple Vendors** – every vendor using the vulnerable engine will also be potentially vulnerable.
- **Hard to Patch** – Each vendor which uses the affected hooking engine will need to update its product.

When we started the research we didn't even look into mature hooking engines since we assumed that given their wide-spread use and massive amount of QA such engines are probably safe. We were wrong.

EASY-HOOK OPEN-SOURCE HOOKING-ENGINE

EasyHook is as its name suggests, is a simple to use hooking engine with advanced hooking features that supports 32-bit and 64-bit platforms. To mention a few:

- Kernel Hooking support
- "Thread Deadlock Barrier" – deals with problems related to hooking of unknown APIs.
- RIP-relative address relocation for 64-bit
- ...

However is has two drawbacks when it comes to security:

1. RWX Hooked Modules – EasyHook doesn't restore the page-protection after the hook is set on hooked modules.

2. RWX Code Stubs – EasyHook leaves its code stub as RWX. Moreover, when compiled in release it uses non-executable heap for its code-stub allocations. In order to make its

allocations executable, it uses VirtualProtect. The problem with this approach is that the heap doesn't guarantee that the code stub will be page-aligned which means that it may inadvertently convert data to code.

DEVIARE2 OPEN-SOURCE HOOKING-ENGINE

Deviare2 is an open-source hooking engine with a dual-license, GPL for open-source and Commercial for closed-source, that supports both 32-bit and 64-bit platforms. Like EasyHook it has an extensive list of features:

- Defer Hook – Set a hook only when and if a module is loaded
- .NET Function hooking
- Interface for many languages: (C++, VB, Python, C#,…)
- …

In Deviare2 we found only a single security issue – RWX Code Stubs. Deviare2 allocates its code using VirtualAlloc function with PAGE_EXECUTE_READWRITE and leaves it as such. Deviare2 has released a patch with a couple of days from notification.

MADCODEHOOK – COMMERCIAL HOOKING ENGINE

madCodeHook hooking engine a powerful commercial hooking engine by Mathias Rauen that supports both 32-bit and 64-bit platforms and even support windows 95. It used by many vendors – about 75% of which are security-related products, for instance, used by Emsisoft anti-virus. To list some of its features:

- Injection Driver – Used to perform kernel-injection into processes
- IPC API – Used to easily communicate with some main process
- IAT Hooking
- …

In madCodeHook engine we also found a single security issue - RWX Code Stubs.

MICROSOFT DETOURS

Microsoft Detours is the most popular and probably the most mature hooking engine in the world, from Microsoft's web site:

> *“Under commercial release for over 10 years, Detours is licensed by over 100 ISVs and used within nearly every product team at Microsoft.”*

As far as we know, its also the only major hooking engine out there that supports ARM processors. It is also used by many Microsoft own applications, for example Microsoft's Application Virtualization Technology.

Since a patch was not yet released for Detours, we will not disclose the specifics of the vulnerability. An updated version of this paper is expected to be released on 15.8.2016.

However, these are the implications:

- Potentially affects millions of users
- Introduces security issues into numerous products, including security products
- Hard to patch since it involved recompilation of affected products

SUMMARY

Our research encompassed more than a dozen security products. As findings unveiled, we worked closely with all affected vendors in order to fix the issues we found as fast as possible. Most vendors responded professionally and in a timely manner.

As shown, some vendors implement their own proprietary hooking code, while others integrate a third-party vendor for hooking. Given these third party hooking engines, these issues have become widespread, affecting security and non-security products.

This pie chart shows a breakdown of the disclosed issues per the number of vendors suffering from the issue:

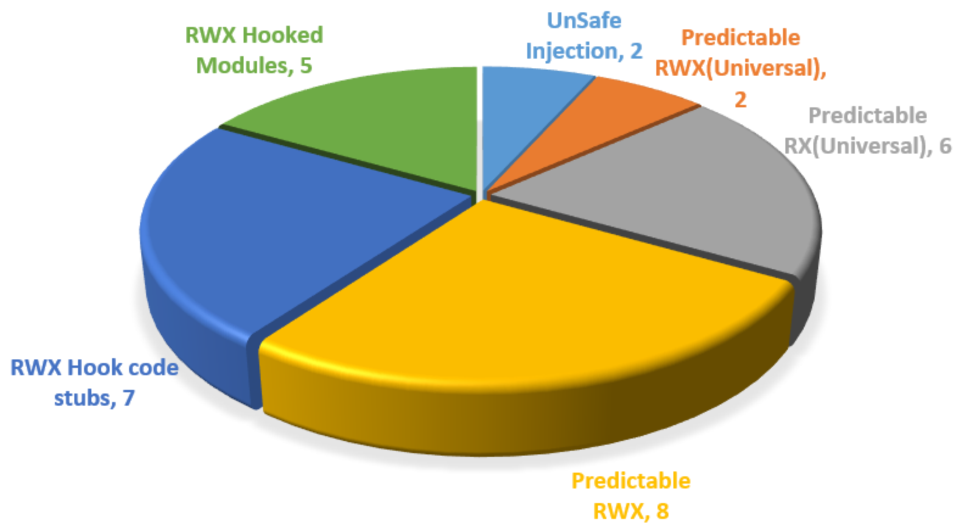


Figure 23: Breakdown of issue type per number of affected vendor

Products/Vendors	UnSafe Injection	Predictable RWX(Universal)	Predictable RX(Universal)	Predictable RWX	RWX Hook code stubs	RWX Hooked Modules	Time To Fix (Days)
Symantec				X			90
McAfee				X	X		90
Major AV*		X	X (Initial Fix)		X		210<
Kaspersky			X	X			90
AVG				X			30
BitDefender					X	X	30
WebRoot			X			X	29
AVAST			X		X		30
Emsisoft					X		90
Citrix - Xen Desktop					X	X	90
Microsoft Office*			X				150
Websense	X			X		X	30
Vera	X			X			?
Invincea		X(64-bit)			X	X	?
Anti-Exploitation*				X			?
BeyondTrust			X	X			Fixed Independently
TOTALS	2	2	6	8	7	5	79.9

Figure 24: Breakdown of issue type per number of affected vendor

Unfortunately, our scope of research was limited given the endless number of products (security and non-security) that integrate hooking into their technologies. We urge consumers of intrusive products to turn to their vendors, requesting a double check of their hooking engines to ensure that they are aware of these issues and make sure they are addressed.

We urge consumers of intrusive products to turn to their vendors, requesting a double check of their hooking engines to ensure that they are aware of these issues and make sure they are addressed.

HOW ENSILO WORKS

enSilo prevents the consequences of cyber - attacks, stopping data from being altered (encrypted), wiped or stolen, while enabling legitimate operations to continue unaffected. The solution hones in on and shuts down any malicious or unauthorized activity performed by an external threat actor, while allowing business to go on as usual. As soon as the platform blocks a malicious communication attempt, it sends an alert that contains the detailed information that the security team will need for their breach remediation process.

ENSILO BENEFITS

enSilo buys organizations the time and peace of mind they need to protect and remediate their sensitive information.

REDUCE OPEX

One alert per one live threat

- ✓ Low number of alerts
- ✓ Forensics on your own time
- ✓ Lower forensics costs
- ✓ No action required

REDUCE OPEX

Real-time, exfiltration prevention

Real-time, ransomware prevention

- ✓ Prevent the consequences of an advanced attack
- ✓ Before Real-time, It never starts
- ✓ You don't need to know where the data lives

REDUCE OPEX

Frictionless security

- ✓ Allow working on a compromised environment
- ✓ Only stop the malicious communication or process