HookTracer: A System for Automated and Accessible API Hooks Analysis

Andrew Case[a], Mohammad M. Jalalzai[b,c], Md Firoz-Ul-Amin[c], Ryan D. Maggio[b,c], Aisha Ali-Gombe[d], Mingxuan Sun[c], Golden G. Richard III[b,c]

[a] *Volatility Foundation*
[b] *Center for Computation and Technology, Louisiana State University*
[c] *School of Electrical Engineering & Computer Science, Louisiana State University*
[d] *Department of Computer and Information Sciences, Towson University*

Abstract

The use of memory forensics is becoming commonplace in digital investigation and incident response, as it provides critically important capabilities for detecting sophisticated malware attacks, including memory-only malware components. In this paper, we concentrate on improving analysis of API hooks, a technique commonly employed by malware to hijack the execution flow of legitimate functions. These hooks allow the malware to gain control at critical times and to exercise complete control over function arguments and return values. Existing techniques for detecting hooks, such the Volatility plugin *apihooks*, do a credible job, but generate numerous false positives related to non-malicious use of API hooking. Furthermore, deeper analysis to determine the nature of hooks detected by *apihooks* typically requires substantial skill in reverse engineering and an extensive knowledge of operating systems internals. In this paper, we present a new, highly configurable tool called *hooktracer*, which eliminates false positives, provides valuable insight into the operation of detected hooks, and generates portable signatures called *hook traces*, which can be used to rapidly investigate large numbers of machines for signs of malware infection.

Keywords: memory forensics, malware, memory analysis, API hooks, *unicorn*, emulation

1. Introduction

The last decade has seen the rise of memory forensics from a research-grade idea to a standard procedure in digital forensics workflows. This adoption has largely been driven by the wide-spread creation and use of memory-only malware and malware components that require little-to-no interaction with the local filesystem. To detect such threats, investigators must rely on analysis of the data structures and artifacts contained within volatile memory. Fortunately, significant open-source memory forensics research and tool development has been performed that enables a wide variety of analysis tasks, including malware detection, insider threat investigations, system audits, and more [5, 21, 31]. One of the most significant drawbacks of all of these tools, however, is the inaccessibility of several critical analysis tasks to less experienced investigators, especially those with little previous background in operating system internals and malware reverse engineering. One of the most glaring examples of this is the detection and analysis of API hooks by userland malware on Windows systems. The use of API hooks by malware allows it to inspect, filter, and modify any data being passed to and returned by functions within running programs, including any associated libraries [17] . By placing such hooks, malware is then able to perform a wide variety of tasks, such as keystroke logging, password stealing, hiding processes and files, hijacking network connections, preventing security tools from loading, and nearly anything else that it wishes to perform on the system. Due to the power that API hooks gives malware over a system, detection of such threats is a high priority for digital investigators [16, 29].

The current inaccessibility of API hook triage and analysis to all but the most experienced investigators significantly reduces the scalability of memory forensics and presents a significant bottleneck within the workflow of organizations. In this paper, we demonstrate these issues through the use of the industry-standard *apihooks* [32] plugin in Volatility and our newly developed Volatility plugin, *hooktracer*. Our plugin performs post-processing of *apihooks*-generated output in conjunction with our own memory analysis algorithms. The goal of our plugin is to automate significant portions of API hook triage, make the analysis results accessible to novice investigators, and generate data that can be fed into other automated analysis engines, such as machine learning and security analytics systems. Our plugin is intended to benefit investigators in enterprise environments, where a significant number of 3rd party applications and security monitors are installed af-

Email addresses: andrew@dfir.org (Andrew Case), mjalal7@lsu.edu (Mohammad M. Jalalzai), mfiroz1@lsu.edu (Md Firoz-Ul-Amin), rmaggi2@lsu.edu (Ryan D. Maggio), aaligombe@towson.edu (Aisha Ali-Gombe), msun@csc.lsu.edu (Mingxuan Sun), golden@cct.lsu.edu (Golden G. Richard III)

ter the initial Windows installation, which populates memory with many disparate artifacts. In these environments, whitelists of memory-resident data and system-wide instrumentation are generally not deployed or realistically even possible, making the large amount of noise generated by certain memory forensic techniques untenable.

This paper begins by providing an overview of API hooks and how Volatility's existing *apihooks* plugin detects them. It then illustrates the specific deficiencies in the existing *apihooks* plugin that make it largely unusable in real-world, enterprise environments. This discussion is followed by presentation of the algorithm that drives our new analysis plugin along with the results of our plugin against a variety of operating system versions, security software, and malware samples.

2. API Hooks Background

2.1. Code Injection

As mentioned in the previous section, the use of API hooks allows malware to have nearly complete control of a running system. To place API hooks within target processes, malware must first be able to run code inside a process. A variety of code injection techniques are available to malware to accomplish this goal [24]. These techniques allow injection of blocks of code, commonly known as shellcode, or entire library files (DLL files) into foreign processes. In nearly all modern investigations, these blocks of shellcode or DLL files will be entirely memory-resident. Detection of code injection techniques can be accomplished with Volatility's existing *malfind*, *messagehooks*, and *eventhooks* plugins, among others [15].

Once malware is injected into a victim process, it often inserts API hooks [14] within the victim's address space. The hooks effectively replace the implementation of an existing function with one implemented by the malware. Such hooks can take one of two forms, both of which are detected by Volatility's *apihooks* plugin, explained next.

2.2. IAT and EAT Hooks

Portable executable (PE) files are the native executables for Windows environments [39]. At compile time, generated PE files specify which libraries and external functions are needed for the application to operate correctly. When a Windows application is loaded, the runtime loader will then load and initialize these libraries from the file system using the *LoadLibrary* API [20] and resolve the runtime addresses of needed functions through the *GetProcAddress* API [37].

As these addresses are resolved, they are stored in optimized lookup tables so that future calls will not require loader-related overhead. For functions that an application or library imports, the resolved addresses are stored in the module's import address table (IAT). For functions that are exported for use by other modules, the resolved addresses are stored in the module's export address table (EAT).

Malware can effectively hijack the operation of resolved functions by overwriting the corresponding entries within these lookup tables. Once addresses are overwritten with the addresses of malicious functions, all future calls to the victim function are completely under the control of the malware.

Volatility's *apihooks* plugin detects such hooks by first enumerating every module (the main application and its dependent DLLs) in a process' address space and then verifying that every entry in the IAT and EAT for each module points back into its owning module or, if it points outside the module, that it matches a whitelist of known redirected functions. Otherwise, any entry whose implementation points to an address outside the owning module is reported as hooked.

Figure 1 shows how IAT and EAT hooks are reported in Volatility. In this output, the type of the hook (IAT), the process that is hooked (svchost.exe) and the function *(SLGenerateOfflineInstallationId)* that was hooked inside of the victim DLL *(slc.dll)* are shown. Additional information includes the module responsible for the redirection (*sppc.ddl*) and a disassembly of the first few redirected instructions.

```
Hook mode: Usermode
Hook type: Import Address Table (IAT)
Process: 880 (svchost.exe)
Victim module: sppcomapi.dll (0x7fefac20000 - 0x7fefac5d0(
Function: slc.dll!SLGenerateOfflineInstallationId
Hook address: 0x7fefac695cc
Hooking module: sppc.dll
Disassembly(0):
0xfac695cc 48               DEC EAX
0xfac695cd 895c2410         MOV [ESP+0x10], EBX
0xfac695d1 48               DEC EAX
0xfac695d2 896c2418         MOV [ESP+0x18], EBP
0xfac695d6 56               PUSH ESI
0xfac695d7 57               PUSH EDI
0xfac695d8 41               INC ECX
0xfac695d9 56               PUSH ESI
```

Figure 1: An IAT hook.

2.3. Inline/Trampoline Hooks

The second technique used for API hooking is known as inline or trampoline hooks. These hooks work by overwriting the first few instructions of a function to redirect control flow to a malicious implementation. This type of hook has two advantages for malware authors compared to IAT/EAT hooks. First, inline hooks are stealthier in memory as automated disassembly is required to detect them, instead of a verification of the IAT and EAT. Second, inline hooks can target any function within a module, not just those that are directly imported or exported.

To detect these types of hooks, Volatility's *apihooks* plugin performs some relatively simple static analysis. The plugin enumerates all functions within all loaded modules

of a process, and then disassembles the first few instructions to see if control flow leaves the containing function. If such a control flow change occurs, the plugin will report output as shown in Figure 2. This catches most inline hooks, but may miss hooks inserted deeper into a function.

```
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 420 (IEXPLORE.EXE)
Victim module: mswsock.dll (0x71a50000 - 0x71a8f000)
Function: mswsock.dll!WSPStartup at 0x71a5c35b
Hook address: 0x27000a
Hooking module: <unknown>

Disassembly(0):
0x71a5c35b e9aa3c818e       JMP 0x27000a
0x71a5c360 81ec24010000     SUB ESP, 0x124
0x71a5c366 a12c72a871       MOV EAX, [0x71a8722c]
0x71a5c36b 8945fc           MOV [EBP-0x4], EAX
0x71a5c36e 8b4550           MOV EAX, [EBP+0x50]
0x71a5c371 53               PUSH EBX
0x71a5c372 8b               DB 0x8b

Disassembly(1):
0x27000a   e94b36ffff       JMP 0x26365a
```

Figure 2: An inline/trampoline hook.

3. Drawbacks of Current Memory Forensic Detection of API Hooks

While existing memory forensic algorithms for enumerating API hooks are capable of detecting most hooking mechanisms, the amount of data produced by such algorithms on modern operating systems is too much for even subject matter experts to handle. To make matters worse, analyzing a reported hook to determine if it was placed by legitimate software or malware requires reverse engineering of in-memory code and understanding the context of each hook within the process. Manual examination of each hook clearly doesn't scale without refining how *apihooks* operates, as we discuss in the next section.

3.1. Overwhelming Number of Legitimate Hooks

When the memory forensics algorithms for detecting API hooks were originally developed (circa Windows XP), there were almost no hooks present on systems not infected with malware. This meant that any reported hooks were likely malicious and deserved investigation. Unfortunately, this situation has drastically changed in modern versions of Windows, as API hooks are explicitly used by Windows to support backwards compatibility. Specifically, hooks are used to ensure that applications will execute the required version of some function. Many investigators are familiar with the largest of these backwards compatibility subsystems, the Application Compatibility Cache, more commonly referred to by the digital forensics community as the *shimcache* [38].

To illustrate this problem, Table 1 documents the number of API hooks present in a clean/default install of various Windows versions. For our testing, the state of each system was a clean install of the 32bit version of the operating system followed by the default user logging in and then launching the default browser (either Internet Explorer or Microsoft Edge). Each install was done in a new VMware Fusion virtual machine. The memory capture was acquired by suspending the virtual machine and copying the produced *vmem* and *vmss* files [19].

Operating System	Number of API Hooks
Windows XP	36
Windows 7	296
Windows 8	622
Windows 10	32,458

Table 1: Operating system version and corresponding number of legitimate hooks.

Starting with Windows 7, hooks placed by the backwards compatibility engine, browser engine, and other operating system components make the number of hooks to manually analyze completely impractical. Furthermore, as we discuss in the related work section, to date there has not been any effort to effectively whitelist such hooks in a scalable and accessible manner. In Section 5, we discuss our efforts to implement effective API hook whitelisting as well as document how the usability of *apihooks* becomes far worse when anti-virus applications are installed on a system.

3.2. Diagnosis Requires Manual Reverse Engineering

The overwhelming number of API hooks present in default installs of modern versions of Windows and particularly, systems with anti-virus enabled, would not be such a burden for experienced investigators if existing algorithms were able to produce better indicators of which hooks were actually suspicious. Instead, if an investigator wishes to examine an API hook, they must use a combination of the *apihooks*, *volshell*, and *vadinfo* plugins. As discussed in [33, 32, 18], the *volshell* plugin allows programmatic exploration of memory samples, including disassembling arbitrary regions of process memory. The *vadinfo* plugin maps addresses within a process' address space to a file path on disk or the anonymous memory region that backs it. Using these plugins in combination allows an investigator to determine the source of a single API hook, but again, this is a very labor intensive, manual process. Even ignoring the tedium, this procedure is realistically only accessible to experienced reverse engineers.

4. Automating Analysis of API Hook Behavior

To provide automated analysis and filtering of API hooks within a memory sample, we developed a new

Volatility plugin, *hooktracer*. Algorithm 1 illustrates *hooktracer* internals at a very high level. First, a set of API hooks is gathered by executing *apihooks* from Volatility (line 1). Emulation is then performed on each API hook to determine the basic blocks that are executed. Then each basic block is mapped to its hosting memory regions. Finally, traversed regions are displayed in one of several accessible formats (line 2-8).

Algorithm 1: Hooktracer

```
1 ApiHookSet ← API hooks from Volatility's apihooks
2 foreach hook in ApiHookSet do
3 |   BasicBlocks ← Emulate(hook)
4 |   CodeMemRegs ← Map(BasicBlocks)
5 |   foreach Region in CodeMemRegs do
6 |   |   Display(Region)
7 |   end
8 end
```

The following sections describe the implementation of Algorithm 1 in more detail.

4.1. Gathering API Hooks

The set of API hooks present within each process can be gathered using the techniques employed by the existing Volatility *apihooks* plugin. This process is relatively slow, as it must check thousands of functions to be thorough. The current implementation of our tool consumes the output of *apihooks* formatted using JSON.

4.2. Hook Emulation Engine

To determine the code paths that a particular API hook takes, we rely on runtime emulation [43]. Emulation is a technique for "executing" code in a software environment that mimics physical hardware. The use of emulation has a long history in the security and malware analysis communities [10, 12, 27, 9, 40, 46], with QEMU being perhaps the most well-known emulator. We chose to leverage emulation to avoid the pitfalls of the current *apihooks* plugin, which statically analyzes instructions and uses several hard-coded patterns to detect control flow redirection outside of the hosting module. While this is useful for detecting hooks, analysis of more than a few instructions per function is extremely difficult and often quite brittle. Our choice of emulator for our plugin was *unicorn* [41], which is used in a variety of security and forensics software [6], and has Python bindings to allow complete control of its emulation environment from Volatility.

4.3. Initializing the Emulation Environment

Before emulation using *unicorn* can begin, the emulator environment must be initialized. This is left largely to the developer and provides a great deal of flexibility. To be useful, code using the emulator needs to register callbacks within the emulated environment to monitor the emulated code's behavior. Our Volatility plugin currently registers emulator callbacks for the following events exposed by *unicorn*:

- Instruction tracing
- Basic block tracing
- Memory reads and writes
- Memory accesses (read, write, or execute) to invalid or unmapped memory regions

After registering our callbacks, our plugin initializes a virtual address space for analysis of each API hook.

4.4. Implementing the Emulated Stack

The first aspect of the virtual address space that our plugin initializes is the stack. By default, *unicorn* provides no stack and the programmer must initialize a memory region within the emulated address space and set the stack pointer register to point to it. Implementing a fake stack and maintaining correct operations presented two main challenges.

First, the stack region chosen must live within a region not currently in-use by the application and one which would not be inadvertently overwritten by the emulated code. To avoid this issue, we chose a region within the kernel virtual address space to place our emulated stack. When running on a real Windows system, userland code can never access kernel ranges so this does not break any operations. We also implemented our read, write, and execution monitor callbacks to stop emulation if they detect access attempts to kernel memory ranges that are not within our chosen stack region. The effect of this setup and associated monitors is that the emulated code can store and retrieve data on the stack as usual, and we can ensure that data within the process' memory is not trampled by our stack emulation.

The second challenge we faced related to the stack was how to correctly determine when an emulated hook finished executing. This was essential to ensure that we let the entire API hook call chain be emulated without letting execution branch to incorrect locations after completion. To meet this goal, we instrumented our read and write operation callbacks to monitor access to the stack base address. Since our emulated stack starts 'empty', the plugin's initialization code sets a global flag to False and only updates it if the stack base is written to by emulated code. Our memory read callback is set to monitor for reads to the stack base and halts emulation if the stack base is read from before being written. The motivation behind this monitoring is that when an API hook executes its final *ret* instruction to return control flow from itself, the *ret* will attempt to read from the initial stack base to gather the address to continue execution. We know that the *ret* will be pointed at the stack base as the API hook handler is the initial function emulated, and any/all sub-procedures

called by the hook will have already adjusted the stack pointer before returning.

With these challenges dealt with, our plugin is able to provide a fully functional stack to the emulated code.

4.5. Emulating an API Hook

Once the emulator environment is initialized, we use *unicorn* to begin emulation at the starting address of the API hook. Since this address is not yet mapped into the emulated address space, the initial execution attempt will trigger a call to our invalid memory access callback with the address and size of the access set as parameters. If the address is within a valid memory region of the analyzed process, then our plugin will attempt to read it from the memory sample. When the accessed page is present within the memory sample, our callback will first read the data from the memory sample and then copy the data to the corresponding address in the emulated address space. This allows the emulator to continue processing and for our plugin to fill the emulated address space on demand. The same procedure occurs when control flow pivots to previously unmapped pages or when data is read from or written to pages for the first time.

In situations where a needed page is not accessible, our plugin's callback will optionally "patch" in data where possible to allow execution to proceed for as long as possible. When enabled for write operations, the plugin maps a blank page into the emulated address space and then allows the write to occur on the new page. For execution attempts on new pages caused by a *CALL* instruction, our plugin maps in the target page and fills the target address with the opcodes corresponding to the *MOV EAX, 0; RET;* instruction sequence. These instructions set a return value of zero, which mimics the usual error condition of Windows APIs. The calling function can then branch based on the error condition and continue execution.

4.6. Gathering and Analyzing Basic Blocks

As an API hook is being emulated, *unicorn* triggers a callback event when new basic blocks are reached. Basic blocks are units of code (instructions) that execute linearly and in an unconditional manner. The *hooktracer* plugin leverages this callback to record every basic block executed by a particular API hook. Once emulation of a hook is complete, the plugin leverages Volatility's API to map every basic block to its containing memory region. By gathering these regions in the order of their execution, a wide variety of analysis can be performed as described in the following section.

5. Automated Analysis with *hooktracer*

Figure 3 shows the output of our plugin against an API hook inserted by the Core Flood [1] malware. In this output, the plugin reports that a process with PID 2044 and name *IEXPLORE.EXE* has an API hook on the *GetMessageA* function inside *user32.dll*. This information comes directly from the JSON data generated by *apihooks*. The rest of the information is generated by our analysis algorithm. Each subsequent line lists, in order, the memory region where at least one basic block was executed.

```
2044 IEXPLORE.EXE     user32.dll!GetMessageA
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x7ff80000
0x7ffadfff>
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\WINDOWS\
system32\user32.dll (10)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\WINDOWS\
system32\ntdll.dll (5)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\WINDOWS\
system32\user32.dll (3)
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x7ff80000
0x7ffadfff> (12)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\WINDOWS\
system32\kernel32.dll (3)
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x7ff80000
0x7ffadfff> (2)
```

Figure 3: Hooktracer output for Coreflood malware.

In interpreting this output, we first see that control flow of our hooked API was redirected to a non-file backed region starting at 0x7ff80000. We also see that the permissions of the region are executable, readable, and writable. This raises several red flags, the first being that legitimate code should be mapped from a file on-disk, not stored and executed directly from memory. Second, having all three permissions bits enabled is a common sign of malware that is utilizing memory-only code, as these permissions allow injection of shellcode. In legitimate applications that do not contain self-modifying code, executable regions should be readable and executable, but not writable. The permissions also assist in detecting hollowed processes. As described by Cysinfo [36], DLL files loaded through normal APIs, such as *LoadLibrary*, will have their permissions set to *PAGE_EXECUTE_WRITECOPY*. For hollowed processes, the permissions will always be something else, generally *PAGE_EXECUTE_READWRITE*. Finally, we note that the paths displayed by our plugin are derived from the in-kernel data structures (VADs) that track the memory region. This prevents name-overwriting attacks against the userland loader from affecting our output [8].

The remainder of the output in Figure 3 illustrates that the legitimate *ntuser.dll* and *user32.dll* handled the actual API request and then later returned control back to the malicious handler. The number in parenthesis after each region is the number of basic blocks that were executed in a memory region before control flow was transferred outside the region. This numbering makes the output more concise and helps to focus attention on regions in which significant numbers of instructions were executed.

The usefulness of grouping memory regions becomes even more clear when examining API hooks inserted by one of the most prolific pieces of malware in history, TDSS [2]. An API hook related to TDSS is illustrated in Figure 4. In the beginning of this output, we see that the

API hook initially begins executing in the memory region starting at 0x270000 but then later transfers control to a second malicious region starting at 0x260000. Based on this output, the investigator can quickly deduce that there are two regions hosting suspicious code, as opposed to just the original one. No reverse engineering was required to gain this insight. Furthermore, Volatility provides several plugins that permit extraction of memory regions once the base address is determined [44].

```
420 IEXPLORE.EXE     mswsock.dll!WSPStartup at 0x71a5c35b
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x270000
0x270fff>
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x260000
0x26efff> (2)
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x270000
0x270fff> (18)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\WINDOWS\
system32\mswsock.dll (2)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\WINDOWS\
system32\ntdll.dll (12)
```

Figure 4: Hooktracer output for TDSS malware.

5.1. Hook Analysis with Security Tools Present

Based on the previous figures, readers may draw the same conclusion that many investigators do, which is that any API hook that initially starts execution in non-file backed memory is illegitimate. Unfortunately, this is often an incorrect conclusion, as nearly all anti-virus and endpoint security monitors employ malware-like tactics to gain visibility into system activity as well as to remain as hidden as possible. Visibility is often gained by utilizing API hooks to monitor parameters passed to functions as well as for system events, such as a process starting or a DLL loading. Stealthiness is enhanced by using non-file backed regions to disassociate executing endpoint security code from files that might be identified and flagged by malware. Unfortunately, these hooks are detected by the *apihooks* plugin, potentially creating a large number of false positives for an investigator looking for malware.

As an example, after we installed the free edition of AVG Anti-Virus [42] in our default Windows 7 install, the number of API hooks reported went from 296 as shown in the Table 1 to 1,625. This occurred because AVG places numerous hooks in every process to monitor activity.

Figure 5 shows the output of Volatility's *apihooks* plugin against one of the AVG hooks. Obviously, the *apihooks* plugin does not provide any indication that the hook is associated with AVG. Instead, it simply lists the first two hops in the control flow chain, with the second hop transferring control to an unknown third destination. For an investigator to determine the hook's source, they must load *volshell*, as previously discussed, to begin reverse engineering the hook's code and manually following the jumps. The investigator might then use Volatility's *vadinfo* plugin to map the jump destinations to memory regions.

In comparison, Figure 6 shows this hook as reported by *hooktracer*. In this output, the investigator can see that

```
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 3068 (iexplore.exe)
Victim module: ntdll.dll (0x77640000 - 0x7777c000)
Function: ntdll.dll!LdrLoadDll at 0x776a22b8
Hook address: 0x74c601f8
Hooking module: <unknown>
Disassembly(0):
0x776a22b8 e93bdf5bfd       JMP 0x74c601f8
<cut>
Disassembly(1):
0x74c601f8 e9c3daabeb       JMP 0x6071dcc0
<cut>
```

Figure 5: AVG API hook detected by Volatility's apihooks plugin.

control flow transfers from the API hook at 0x776a22b8 to the non-file backed region at 0x74c60000, and then to several DLLs inside of the AVG Program Files subfolder. Given that the hook has likely been placed by a well-known security product, the investigator can instead dedicate time to looking for other signs of malware infection.

```
3068 iexplore.exe     ntdll.dll!LdrLoadDll at 0x776a22b8
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x74c60000
0x74c6afff>
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\
Program Files\AVG\Antivirus\snxhk.dll (2)
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x74c60000
0x74c6afff> (46)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\
Program Files\AVG\Antivirus\aswhookx.dll (2)
PAGE_EXECUTE_READWRITE <Non-File Backed Region: 0x6f670000
0x6f67ffff> (4)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\Windows\
System32\ntdll.dll (2)
```

Figure 6: AVG API hook detected by *hooktracer*.

5.2. Filtering Legitimate DLLs

Even with the accessibility of API hooks analysis provided by *hooktracer*, the sheer number of API hooks present on even non-infected systems makes manually scrolling through the output time consuming. To help alleviate this burden, we added filtering support to the plugin. Two of these filters are described in this section and the third filter is described in the next section.

The first filter allows excluding an API hook from output if every memory region accessed during emulation matches a given file or folder path. The most common use of this filter is to exclude API hooks where all code paths are handled by file-backed regions originating under the `System32` directory. This is possible as modern Windows versions protect DLLs in this directory from modification, which prevents malicious overwriting of these files. As an example, Figure 7 shows our plugin's output against a legitimate API hook from our clean Windows 10 system.

In the output, an API hook of the *CryptUninstallCancelRetrieval* function is shown as well as that every code path for the hook is inside DLL files under `System32`.

```
992 svchost.exe    cryptnet.dll!CryptUninstallCancelRetrieval
at 0x634c80f0
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Windows\
System32\crypt32.dll
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Windows\
System32\ntdll.dll (4)
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Windows\
System32\crypt32.dll (9)
```

Figure 7: Legitimate API hook in Windows 10 detected by *hooktracer*.

This is precisely what thousands of hooks look like in memory when *shimcache* and other built-in hooks are active, which is the default starting in Windows 7.

To exclude such hooks from the output of *hooktracer*, investigators can re-run the plugin with an "All Containing" filter of `\Windows\System32`. For "All Containing" filters, our plugin compares the path of every memory region found during basic block tracing to the path(s) specified in the filter. If every region matches the filter (e.g., they are all in the Windows `System32` directory), then information about the API hook is suppressed. By applying this filter to our clean Windows 10 sample, the number of hooks reported drops from 32,458 to only 178. This shows that by simply filtering every API hook whose implementation exists solely in DLLs stored under `System32`, we have removed over 99% of the plugin's default output.

When examining the remaining 178 hooks, two hook patterns emerge, as illustrated in Figures 8 and 9. These hooks are related to the Visual C++ runtime and to Microsoft's OneDrive application and files associated with these components are not stored under the `System32` directory. If we re-run the plugin with filtering added for these DLLs, the number of hooks reported goes from 178 to zero. Thus by starting with a filter for hooks targeting `System32` DLLs and then adding new string-based filters for observed legitimate hooks, we are able to quickly determine that no malicious hooks are present on the system. This process required no reverse engineering and each execution of our plugin takes less than 30 seconds on a typical laptop computer.

```
6044 Microsoft.Phot   MSVCP140_APP.dll!?uncaught_exceptions@std@@YAHXZ
at 0x5b2aa470
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Program Files\
WindowsApps\Microsoft.VCLibs.140.00_14.0.25426.0_x86__8wekyb3d8bbwe\
vcruntime140_app.dll
```

Figure 8: An unfiltered API hook related to use of the *vcruntime140_app.dll* in the WindowsApps folder.

```
632 OneDrive.exe     MSVCP140.dll!?uncaught_exception@std@@YA_NXZ
at 0x6e7ad2b0
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume2\Users\x\AppData\
Local\Microsoft\OneDrive\18.143.0717.0002\vcruntime140.dll
```

Figure 9: An unfiltered API hook due to use of Microsoft's OneDrive components.

Hooktracer also supports a second filter type that can be used alone or in conjunction with "All Containing" filters. The second type, "Any Containing", will exclude API hooks from output when the path of at least one memory region matches the filter. This type of filter is extremely powerful when analyzing hooks placed by security tools, such as AVG. As shown previously in Figure 6, AVG hooks every process with DLLs that live under the `\ProgramFiles\AVG\Antivirus` directory. To exclude AVG's hooks from the plugin's output, we can use an "Any Containing" filter configured with the AVG directory path.

As mentioned previously, *apihooks* found 1,625 userland API hooks in our memory sample with AVG active as compared to 296 before it was installed. By using an "Any Containing" filter set for AVG in conjunction with our previous "All Containing" filters for `System32` and vcruntime, the number of hooks is reduced to 175, an 89% reduction. Examining the remaining hooks shows that 122 of them are inside of Internet Explorer processes and are browser compatibility hooks that redirect into *IEShims.dll* or *ieframe.dll*, as shown in Figure 10.

```
3068 iexplore.exe     kernel32.dll!CreateDirectoryA
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\
Program Files\Internet Explorer\IEShims.dll
[listing truncated]
3068 iexplore.exe     USER32.dll!IsDialogMessageW
at 0x777b4104
PAGE_EXECUTE_WRITECOPY \Device\HarddiskVolume1\Windows\
System32\ieframe.dll
```

Figure 10: Browser hooks redirect into *IEShims.dll* or *ieframe.dll*.

The remaining hooks involve processes that loaded Visual C++ runtime DLLs from the Windows Side-by-Side directory and the *AcLayers.dll* component of the *shimcache*, which is stored in the `\Windows\AppPatch` directory. By running the plugin again with filters targeting these components, the number of hooks reported drops to zero.

5.3. Grouping Hooks Across Processes

Another powerful capability of *hooktracer* is the ability to group sets of hooks across processes. This allows investigators to understand the full scope of infections on a single system as well as build simple and reliable indicators of compromise that can be used on any number of memory samples across a number of systems. For this case study, we will analyze our previously clean Windows 7 system, which we infected with the infamous Zeus malware [25, 45].

Executing *apihooks* against this memory sample produces 480 API hooks compared to the 296 present in our clean sample. This large increase is due to Zeus' aggressive behaviour of injecting code into every process that it has permission to access, as well as hooking 41 functions within each victim process. Without any filters, *hooktracer* will produce many similar blocks of output per Zeus hook, as

shown in Figure 11. Note that the permissions indications have been removed for readability.

```
2384 taskhost.exe     ntdll.dll!NtCreateUserProcess at
0x779f5778
<Non-File Backed Region: 0x850000 0x87bfff>
<Non-File Backed Region: 0x9a0000 0x9a0fff> (2)
\Device\HarddiskVolume1\Windows\System32\ntdll.dll (2)
<Non-File Backed Region: 0x850000 0x87bfff> (5)
```

Figure 11: hooktracer output for a Zeus API hook.

The hook's control flow starts with two anonymous regions followed by a DLL file under `System32` and then exiting from the original anonymous memory region. All of the hooks placed by Zeus follow this same pattern of two anonymous regions to start followed by the legitimate API being handled by a varying number of DLLs inside of `System32`.

To allow investigators to avoid manually examining 41 of these hooks per process, we implemented a grouping capability in *hooktracer* that allows filtering the output to include only the processes and victim functions hooked by the same malware code. To generate a grouping, an analyst runs *hooktracer* and specifies the process ID and victim function name of the hook to be grouped. As shown in Figure 11, the PID is 2384 and the function is *ntdll!NtCreateUserProcess*. This instructs *hooktracer* to create an ordered record for the first three memory regions executed by the hook so that they can later be re-identified. This record will include the size for non-file backed regions and the full path on disk for file-backed ones. We chose the size for non-file backed regions as the identity marker as it is highly consistent across injections. Other attributes, such as the starting and ending address or a region's contents, are not reliable, due to both address layout space randomization (ASLR) as well as code and data changes that occur within a region at runtime and across processes.

Once the grouping record is generated, the analyst can re-run *hooktracer* with the record specified. This will instruct the plugin to display only processes and hooked functions that match the record's pattern of region sizes and file paths. As shown in Figure 12, *hooktracer's* grouping capability uses the record from one hook in one process to identify every other process and function infected with Zeus. This figure has some of the output truncated for brevity's sake, but in total *hooktracer* was able to automatically find and report the 41 hooked functions across all 8 infected processes.

Investigators can also use hook records when analyzing other Windows memory samples. In real-world investigations, where numerous machines may need to be investigated quickly, being able to rapidly determine which are infected and which are not is key. By integrating *hooktracer's* grouping capability into their investigative workflow, an investigator can whittle an entire investigation's worth of systems down to only the infected ones within minutes.

```
1024 iexplore.exe
                ntdll.dll!NtCreateUserProcess
                ntdll.dll!ZwCreateUserProcess
                kernel32.dll!GetFileAttributesExW
                WININET.dll!InternetCloseHandle
                CRYPT32.dll!PFXImportCertStore
                USER32.dll!BeginPaint
                USER32.dll!CallWindowProcA
[hook listing truncated]

2468 explorer.exe
                ntdll.dll!NtCreateUserProcess
                ntdll.dll!ZwCreateUserProcess
                kernel32.dll!GetFileAttributesExW
                WININET.dll!InternetCloseHandle
                CRYPT32.dll!PFXImportCertStore
                USER32.dll!BeginPaint
                USER32.dll!CallWindowProcA
[hook listing truncated]

2564 conhost.exe
                ntdll.dll!NtCreateUserProcess
                ntdll.dll!ZwCreateUserProcess
                kernel32.dll!GetFileAttributesExW
                WININET.dll!InternetCloseHandle
                CRYPT32.dll!PFXImportCertStore
                USER32.dll!BeginPaint
                USER32.dll!CallWindowProcA
[hook listing truncated]
```

Figure 12: hooktracer grouping Zeus' API hooks.

6. Related Work

6.1. Emulation for Malware Analysis

The use of emulation to analyze the behavior of malware is a powerful technique with over a decade of research behind it [47, 26, 13, 28]. Until recently, however, all of these emulation efforts required access to an original malware executable file as well as the ability to emulate that executable in a heavyweight environment, such as Bochs [30] or QEMU [11], to instrument and observe execution. While these techniques are powerful, such approaches are not directly applicable to memory analysis, as executables in memory go through substantial transformations from the time they are loaded from disk until a memory capture is taken. This generally prevents the executables from being later extracted from memory and then natively executed. Furthermore, the rise of memory-only malware means that much of the malware found in modern investigations cannot be easily encapsulated into a functional executable file at all. This prevents existing whole-system emulators from being able to analyze the malware. Finally, existing hook detection architectures require substantial instrumentation and specialized lab setups that are not realistically feasible in incident response handling across diverse enterprise infrastructures. Other modern techniques for live analysis, such as virtual machine introspection [7], face many of the same challenges and are not applicable in post-compromise scenarios.

6.2. Memory Forensics and Emulation

After the introduction of *unicorn* and its accessible Python bindings, there have been two recent research efforts besides ours that integrated *unicorn* with Volatility. The first, ROPEMU [22, 4], uses *unicorn* to automatically detect ROP chains [35] within memory. ROP is used by system-level exploits to perform code-reuse attacks. Such attacks are necessarily memory-only and can be difficult to detect with traditional Volatility plugins.

The second project [23] also hunts for ROP chains and was specifically developed to detect the "Gargoyle" attack [34] that hides executable code using permission changes and timers. Detection of Gargoyle is implemented by emulating the handler of each registered timer found by Volatility and checking if calls are made to any Windows API functions leveraged by the Gargoyle attack.

Although neither of the referenced projects are related to API hooks, we consider them to be important related work, as they both leverage *unicorn* in conjunction with Volatility to significantly expand the state-of-the-art in memory forensics.

6.3. Analysis of In-Memory API Hooks

The difficulties of analyzing API hooks on enterprise systems without a filtering capability led to a research project and Volatility plugin named *apihooksdeep* [3]. This plugin filters API hooks based on the fuzzy hash set of the code that implements the initial handling of the hooks. To generate this hash set, the investigator must use a combination of Volatility plugins to determine the owning module and then also use other plugins to extract the module from memory. A standalone tool must then be run against the module to generate the hash set. Once the hash set is created, it can then be used as a filter to exclude API hooks that match the the same initial hooking code. Using the plugin to filter legitimate hooks that start in non-file backed memory, as was shown with AVG, is not currently supported by the plugin and would likely be problematic, since most of these hooks are simply control flow transfers.

While *apihooksdeep* is a substantial research effort and the best effort to date for filtering of Volatility's *apihooks*, its accessibility to a wide range of investigators and its scalability are quite limited. These limitations occur as generating the hash sets for a single API hook takes several steps, including running multiple Volatility plugins and a separate standalone tool. Furthermore, these hash sets need to be built on a clean system that closely matches the one under investigation or the filter will be ineffective. While this is possible in mature, enterprise IT environments with "gold builds" for each system, it still requires at least one member of the security team to build new hash sets for each new build and for every update to 3rd party applications, such as anti-virus and other endpoint security monitors. This also places utilization of the hash sets outside of the reasonable workflow of consultants, who must operate in a wide variety of non-uniform environments. Finally, since the hash sets are based on the actual code of modules implementing API hooks, the hash sets will change substantially over time. The use of fuzzy hashing offsets this risk some since the plugin can report a percentage of how similar two hooks are, but then this requires the analyst to tune the acceptable threshold.

Compared to our filters, which are string-based and gathered from DLL listings produced by our plugin, the fuzzy hash sets are more complicated to use, burdensome to maintain, and more brittle.

7. Conclusion

In this paper we have demonstrated a new Volatility plugin, *hooktracer*, which makes userland API hooks analysis both more efficient and accessible to a wide range of investigators. Our plugin performs emulation of API hooks to determine the implementing module(s) and relationships between these module(s). Through such analysis, we allow an investigator to quickly determine which, if any, API hooks are suspicious and require deeper investigation. Given the large-scale reliance on API hooks by Microsoft and endpoint security vendors as well as malware authors, it is essential that quick and easy-to-use filtering is available to investigators.

As we have demonstrated, *hooktracer* meets these needs and makes API hook analysis available to investigators of all experience levels. The whitelisting system of *hooktracer* allows highly effective filters to be created using only simple strings, and since these filters are based on the pathnames of files and not code, they will rarely, if ever, need to change. *hooktracer* also supports generation of hook records that allow re-identification of previously discovered malware. Given the large number of systems that a typical incident response effort might target, hook records have the potential to save a substantial amount of investigative time.

References

[1] U.S. Government Takes Down Coreflood Botnet. URL `https://krebsonsecurity.com/2011/04/u-s-government-takes-down-coreflood-botnet/`.

[2] Microsoft Security Intelligence Report. `http://download.microsoft.com/download/8/1/B/81B3A25C-95A1-4BCD-88A4-2D3D0406CDEF/Microsoft_Security_Intelligence_Report_volume_9_Battling_Botnets_English.pdf#page=24`, July 2010.

[3] Volatility Plugin SSDeep for malfind and apihooks. `https://blog.superponible.com/2014/08/30/volatility-plugin-ssdeep-for-malfind-and-apihooks/`, 2014.

[4] ROPMEMU. `https://github.com/Cisco-Talos/ROPMEMU/`, 2016.

[5] The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. `https://github.com/volatilityfoundation/volatility`, 2017.

[6] Unicorn showcase. `http://www.unicorn-engine.org/showcase/`, 2018.

[7] `http://libvmi.com/`, 2019.

[8] b33f. Powershell-suite. https://github.com/FuzzySecurity/PowerShell-Suite/blob/master/Masquerade-PEB.ps1, 2016.

[9] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 281–289. ACM, 2003.

[10] Daniel Bartholomew. Qemu: A Multi-host, Multi-target Emulator. *Linux Journal*, 2006(145):3, 2006.

[11] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2005.

[12] Michael Bilzor, Ted Huffmire, Cynthia Irvine, and Tim Levin. Security Checkers: Detecting Processor Malicious Inclusions at Runtime. In *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 34–39, 2011.

[13] Lutz Böhne. Pandoras Bochs: Automatic Unpacking of Malware. *University of Mannheim*, 2008.

[14] Jurriaan Bremer. x86 API Hooking Demystified. https://jbremer.org/x86-api-hooking-demystified/, 2012.

[15] Andrew Case. Automating Detection of Known Malware Through Memory Forensics. https://volatility-labs.blogspot.com/2016/08/automating-detection-of-known-malware.html, 2016.

[16] Andrew Case and Golden G. Richard III. Detecting Objective-C Malware Through Memory Forensics. *Proceedings of the 16th Annual Digital Forensics Research Workshop (DFRWS)*, 2016.

[17] The MITRE Corporation. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. https://attack.mitre.org/techniques/T1179/.

[18] Tyler Dean. Uroburos Rootkit Hook Analysis and Driver Extraction. https://spresec.blogspot.com/2014/03/uroburos-rootkit-hook-analysis-and.html, 2014.

[19] Volatility Foundation. VMware Snapshot File. https://github.com/volatilityfoundation/volatility/wiki/VMware-Snapshot-File, 2014.

[20] Michael Galkovsky. DLLs the Dynamic Way. https://docs.microsoft.com/en-us/previous-versions/ms810279(v=msdn.10), 2009.

[21] Google. Rekall. https://github.com/google/rekall, 2016.

[22] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. ROPMEMU: A Framework for the Analysis of Complex Code-reuse Attacks. In *ASIACCS 2016, 11th ACM Asia Conference on Computer and Communications Security, May 30-June 3, 2016, Xi'ian, China*, 2016.

[23] Aliz Hammond. Hunting for Gargoyle Memory Scanning Evasion. https://countercept.com/blog/hunting-for-gargoyle/.

[24] Ashkan Hosseini. Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques, year = 2017. https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques\-technical-survey-common-and-trending-process.

[25] IOActive. Reversal and Analysis of Zeus and SpyEye Banking Trojans. Technical report, 2012.

[26] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating Emulation-resistant Malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 11–22, New York, NY, USA, 2009.

[27] William B Kimball and Rusty O Baldwin. Emulation-based Software Protection, October 9 2012. US Patent 8,285,987.

[28] Christopher Kruegel. Full System Emulation: Achieving Successful Automated Dynamic Analysis of Evasive Malware. In *Proc. BlackHat USA Security Conference*, pages 1–7, 2014.

[29] Peter Klnai. volatility-browserhooks. https://github.com/eset/volatility-browserhooks, 2018.

[30] Kevin P Lawton. Bochs: A Portable PC Emulator for Unix/X. *Linux Journal*, 1996.

[31] M. Ligh, A. Case, J. Levy, and A. Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, New York, 2014.

[32] Michael Ligh. HowTo: Extract "Hidden" API-Hooking BHO DLLs. https://volatility-labs.blogspot.com/2013/01/howto-extract-hidden-api-hooking-bho.html, 2013.

[33] Michael Ligh. Stuxnet's Footprint in Memory with Volatility 2.0. https://mnin.blogspot.com/2011/06/examining-stuxnets-footprint-in-memory.html, 2016.

[34] Josh Lospinoso. Gargoyle: A Memory Scanning Evasion Technique. https://github.com/JLospinoso/gargoyle, 2017.

[35] David Maloney. Return Oriented Programming (ROP) Exploits Explained. https://www.rapid7.com/resources/rop-exploit-explained/.

[36] K. A. Monnappa. Detecting Deceptive Hollowing Techniques. https://cysinfo.com/detecting-deceptive-hollowing-techniques/, 2016.

[37] MSDN. GetProcAddress Function. https://docs.microsoft.com/en-us/windows/desktop/api/libloaderapi/nf-libloaderapi-getprocaddress, 2018.

[38] Timothy Parisi. Caching Out: The Value of Shimcache for Investigators. https://www.fireeye.com/blog/threat-research/2015/06/caching_out_the_val.html, 2015.

[39] Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. https://docs.microsoft.com/en-us/previous-versions/ms809762(v=msdn.10), 2010.

[40] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 15–27, 2006.

[41] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn: Next Generation CPU Emulator Framework. *Black Hat USA*, 2015.

[42] AVAST Software s.r.o. AVG Anti-Virus Free. https://www.avg.com/en-us/free-antivirus-download.

[43] Kenneth L. Stevens. *The Emulation User's Guide*. Lulu.com, 2008. ISBN 9781435753730.

[44] Forensics Wiki. List of volatility plugins. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference, 2012.

[45] James Wyke. What is Zeus. Technical report, SophosLabs UK, 2011.

[46] Heng Yin and Dawn Song. Temu: Binary Code Analysis via Whole-system Layered Annotative Execution. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[47] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, 2008.