Hooktracer: Automatic Detection and Analysis of Keystroke Loggers Using Memory Forensics

Andrew Case

Volatility Foundation

Ryan D. Maggio

Division of Computer Science and Engineering, Louisiana State University

Md Firoz-Ul-Amin

Division of Computer Science and Engineering, Louisiana State University

Mohammad M. Jalalzai

Division of Computer Science and Engineering, Louisiana State University

Aisha Ali-Gombe

Department of Computer Science, Towson University

Mingxuan Sun

Division of Computer Science and Engineering, Louisiana State University

Golden G. Richard III*

Center for Computation and Technology and Division of Computer Science and Engineering, Louisiana State University

Abstract

Advances in malware development have led to the widespread use of attacker toolkits that do not leave any trace in the local filesystem. This negatively impacts traditional investigative procedures that rely on filesystem analysis to reconstruct attacker activities. As a solution, memory forensics has replaced filesystem analysis in these scenarios. Unfortunately, existing memory forensics tools leave many capabilities inaccessible to all but the most experienced investigators, who are well versed in operating systems internals and reverse

*Corresponding author

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

engineering. The goal of the research described in this paper is to make investigation of one of the greatest threats that organizations face, userland keyloggers, less error-prone and less dependent on manual reverse engineering. To accomplish this, we have added significant new capabilities to HookTracer, which is an engine capable of emulating code discovered in a physical memory captures and recording all actions taken by the emulated code. Based on this work, we present new memory forensics capabilities, embodied in a new Volatility plugin, *hooktracer_messagehooks*, that uses Hooktracer to automatically decide whether a hook in memory is associated with a malicious keylogger or benign software. We also include a detailed case study that illustrates our technique's ability to successfully analyze very sophisticated keyloggers, such as Turla.

Keywords: memory forensics; keystroke loggers; malware detection; emulation; incident response; reverse engineering

---

1. Introduction

The rise of memory-only malware and attack payloads has led to the nearly ubiquitous use of volatile memory analysis in incident response. Volatile memory analysis, also known as memory forensics, is the technique of acquiring and then analyzing a sample of volatile memory (RAM) obtained from a running computer system or virtual machine. Whereas traditional filesystem forensics can recover only those artifacts that the operating system and running applications choose to record to disk, memory analysis techniques allow an investigator to fully examine and reconstruct the entire state of a system. This state includes all of the in-kernel and userland data structures, code, user-generated input and output, and more. When focused against malware, these capabilities allow an investigator to detect and analyze all of a malware sample's actions, regardless of the techniques employed by the malware to become resident within memory.

As recently documented by Microsoft [42], nearly all modern malware and attacker toolkits have at least one memory-only component and many of them reside solely in memory. One of the most infamous of these was Duqu, which was used to compromise a significant portion of Kaspersky's corporate network environment [10]. Duqu leveraged memory-only rootkits that were installed by exploiting then zero-day vulnerabilities in Microsoft Windows. Duqu utilized no persistence mechanisms, meaning a reboot fully removed it from a system. In Kaspersky's environment, this provided little relief, however, as other Duqu-infected systems would probe the network for rebooted systems and then re-infect them. As documented by Kaspersky in their post-mortem report, full detection and understanding of Duqu was only achieved after memory forensics was used by its incident response team. Careto [39], Skeleton Key [19], and Poison Ivy [22] are other examples of powerful malware that execute in a memory-only or near memory-only manner and that require memory forensics to detect and analyze. The popular open source Metasploit [63] and PowerShell Empire [1] attack frameworks also run memory-only unless the user chooses to

store files within the filesystem. Combined, these malware samples and frameworks provide attackers with complete control of a system in a manner that requires no storage of any data anywhere on the local filesystem or within its contained stores, such as the registry.

As the trend of attackers leveraging memory-only toolkits continues to grow, the need for memory forensic tools and techniques that are accessible to forensic investigators with a wide range of skill levels has become essential. Unfortunately, current memory forensics tools do not meet this need. Volatility [2] is the most widely used and powerful memory forensic framework currently available. It is open source and contains over 200 plugins that support deep inspection of memory-resident artifacts contained within volatile memory captures of Windows, Linux, and Mac systems. While extremely powerful, a major shortcoming of Volatility is that many of its capabilities are only accessible to expert digital forensics investigators. This is particularly true when malware analysis tasks are involved, as they often require manual reverse engineering within Volatility by the analyst. This is primarily a result of Volatility being able to correctly identify malware hooks in memory, but not providing post-processing algorithms capable of successfully differentiating hooks placed by legitimate software from malicious ones.

The goal of the research presented in this paper is to automate and make accessible to investigators of all skill levels one of Volatility's most powerful capabilities - detection of userland keyloggers. Keyloggers are one of the most dangerous threats facing users [74] as they allow recording and exfiltration of keystrokes entered by users, contents of copy and paste buffers, and data displayed within applications. Sophisticated malware often bundles additional capabilities within its keylogger modules, including the ability to take screenshots and record audio from microphones and video from web cameras [36]. As discussed in the next section, the Windows API commonly abused by keyloggers, *SetWindowsHookEx* [53], is one type of hook that Volatility can correctly find, but with no post-processing capability. As documented on the Volatility Labs blog [44], making this determination currently requires a labor-intensive mix of reverse engineering in conjunction with running multiple Volatility plugins. This process also assumes deep knowledge of the Windows API and systems internals. These requirements make the technique inaccessible to all but the most experienced investigators. Furthermore, even for subject matter experts, the process is still labor intensive, manual, and time consuming. Given the substantial amount of evidence that investigators must sift through in modern investigations [56], any portion of the workflow that requires manual examination by senior staff members causes a severe bottleneck in an organization's incident response capabilities. Our work, embodied in a new Volatility plugin called *hooktracer_messagehooks*, leverages the significant additions that we made to the *HookTracer* engine [13] to enable automated and scalable detection and analysis of this threat.

2. The SetWindowsHookEx API

Although heavily abused by malware, *SetWindowsHookEx* also has a number of legitimate uses. These include allowing software to monitor for nearly all hardware events, such as USB device insertion, mouse movement, or keystrokes being typed. Common legitimate uses of keyboard monitoring include implementing computer-based training (CBT) and application-specific hot keys. As shown in Figure 1, to register a hook, a module must call *SetWindowsHookEx* with four parameters set.

```
HHOOK SetWindowsHookExA(
int       idHook,
HOOKPROC  lpfn,
HINSTANCE hmod,
DWORD     dwThreadId
);
```

Figure 1: The function prototype of SetWindowsHookEx.

The first parameter, *idHook*, dictates which event the hook wants to monitor, such as for keystrokes or mouse clicks. The second parameter, *lpfn*, is the callback function within the hooking module that receives each triggered message. The third parameter, *hmod*, is a handle to the DLL containing the hooking procedure or NULL. The fourth parameter, *dwThreadId*, is either zero or the thread ID of the thread for which the hook should be active. The specific process(es) in which a hook will be active and where the hook code will reside is fully dependent on the values of the last two arguments to the function. If *dwThreadId* is non-zero, then it specifies the thread ID within the calling process to hook. Otherwise, if *dwThreadId* is zero, then all threads within the same desktop will be hooked. This also affects the behaviour of *hmod*. If *dwThreadId* is zero and *hmod* is a handle to a DLL, then that DLL will get loaded (injected) into every hooked process once a thread of the process generates a registered hook (e.g., receives keyboard input). If *dwThreadId* is zero and *hmod* is NULL, then the callback at the offset specified by *pfn* inside the calling executable will be triggered from the context of the hooked threads.

2.1. Abuse by Malware

As reported by Kaspersky in 2011 [28], abuse of *SetWindowsHookEx* is the most common method for userland keyloggers to gain access to victim's keystrokes. This remains true today [71, 32]. This popularity has held as the only other two methods available to userland keyloggers are 1) extremely noisy to system monitors; 2) require the malware to inject code into foreign processes on its own instead of relying on features of the runtime loader; and 3) require polling the keyboard state in a cyclical fashion, which misses keystrokes entered between polling operations. From an incident response perspective, the in-memory artifacts introduced by code injection are trivially detected and reported as malicious by Volatility's *malfind* and *apihooks* plugins [73, 45].

On the other hand, userland keyloggers that utilize *SetWindowsHookEx* have much of the hard work done for them by the operating system. For example, if the keylogger wishes to monitor keystrokes across all processes within a desktop, then all it needs to do is set *dwThreadId* to zero. Similarly, if the keylogger wants its malicious DLL injected into all of its victim processes automatically, then it can simply pass a handle to the DLL in the *hmod* parameter. All of this can be done without suspicious and noisy API calls, such as *AdjustTokenPrivileges*, *WriteProcessMemory*, and *CreateRemoteThread*. Volatility's *apihooks* and *malfind* plugins will not detect DLLs loaded through *SetWindowsHookEx* as maliciously injected since they are loaded through normal operating system procedures.

3. Memory Analysis of Hooks

3.1. Volatility's messagehooks Plugin

The *messagehooks* plugin of Volatility can enumerate all hooks registered through *SetWindowsHookEx*. To enumerate each hook, as explained in The Art of Memory Forensics [46], *messagehooks* first enumerates every window station of every user logon session. It then enumerates the desktop(s) associated with each window station. For each desktop, it then enumerates its global hooks as well as the local hooks for each active thread. After enumerating all global and local hooks, *messagehooks* prints one block of output per hook. An example of a global hook is shown in Figure 2 and Figure 3 shows the corresponding local hook in a specific thread.

```
Offset(V)  : 0xfea009d8
Session    : 0
Desktop    : WinSta0\Default
Thread     : <any>
Filter     : WH_CALLWNDPROC
Flags      : HF_GLOBAL
Procedure  : 0x1160
ihmod      : 0
Module     : C:\Windows\system32\wls0wndh.dll
```

Figure 2: A global message hook as displayed by messagehooks.

As seen in Figure 2, this message hook is active inside the Default desktop of logon session 0. We can tell it is a global hook, since its thread is identified as "<any>". This is Volatility's method for signifying that the hook has been set in all threads associated with the same desktop. We also see that the *WH_CALLWNDPROC* message type is being monitored. Finally, we see the registered callback is the code beginning at offset 0x1160 inside of *C:\Windows\system32\wls0wndh.dll*. Examining Figure 3, we see it is reporting that the global hook is active inside of thread 3180 of the *spoolsv* process with process ID 1360. *messagehooks* will report one of these blocks for each thread of the desktop.

```
Offset(V)   : 0xfea009d8
Session     : 0
Desktop     : WinSta0\Default
Thread      : 3180 (spoolsv.exe 1360)
Filter      : WH_CALLWNDPROC
Flags       : HF_GLOBAL
Procedure   : 0x1160
ihmod       : 0
Module      : C:\Windows\system32\wls0wndh.dll
```

Figure 3: A local message hook as displayed by messagehooks.

3.2. Analyzing a Reported Hook

Once an analyst has used *messagehooks* to generate the list of hooks present in a memory sample, they must determine if each hook was set by legitimate software or by malware. Unfortunately, Volatility provides little in the way of guidance for this process. Instead, the investigator must revert to manual reverse engineering. To illustrate this process, Figure 4 shows the steps to begin analyzing a hook. To start, the analyst must run the *dlllist* plugin to determine the load address of the DLL. In this output, we have grepped for the DLL name to reduce output, and are interested in the first column's value (0x6e6d0000). Since the hook is present inside the DLL, we need its load address in memory to calculate where the hook is inside the process' address space. Next, we load the *volshell* plugin, and use the *dis* function to disassemble the beginning of the hook. For the parameter to *dis*, we calculate the load address from *dlllist* as well as the callback (*pfn*) offset given in Figure 3. As can be seen, the output from *dis* is simply the raw assembly of the hook without any additional, helpful information, such as function names, parameter values, or instruction cross references. This makes an already tedious process even slower, and as mentioned previously, understanding the assembly well enough to make a determination regarding a hook's legitimacy requires significant reverse engineering skills. Furthermore, in Figure 4 we show the disassembly of the initial hook callback, but as readers with assembly knowledge can see, there are calls to three other functions. This means the analyst has to not only reverse engineer the initial handling function, but also any subroutines that are called to get a complete picture.

Worse, *messagehooks* has no filtering capability to enable re-use of knowledge gained during previous manual inspections of hooks. This forces the investigator to resort to mental recall or adhoc filtering scripts. Both of these methods are brittle and error-prone, as they do not account for code changes within modules between operating system versions and updates. Furthermore, the ability to write scripts is not universal among digital forensic investigators, and mental recall for single investigators obviously does not scale across a team. The main focus of our research effort is to completely remove the need for an investigator to reverse engineer in-memory code and to allow results of previous analysis to be used in an automated and scalable manner.

```
$ python vol.py --profile=Win7SP1x86 -f win7x886.vmem dlllist\
-p 1360 | grep wls0wndh.dll
0x6e6d0000     0x6000        0x1 2019-01-03 04:24:44 UTC+0000\
C:\Windows\system32\wls0wndh.dll

$ python vol.py --profile=Win7SP1x86 -f win7x886.vmem volshell -p 1360
In [2]: dis(0x6e6d0000+0x1160)
0x6e6d1160 6a14                            PUSH 0x14
0x6e6d1162 68a8116d6e                      PUSH DWORD 0x6e6d11a8
0x6e6d1167 e824ffffff                      CALL 0x6e6d1090
0x6e6d116c 33f6                            XOR ESI, ESI
0x6e6d116e 8975fc                          MOV [EBP-0x4], ESI
0x6e6d1171 8b7d10                          MOV EDI, [EBP+0x10]
0x6e6d1174 397508                          CMP [EBP+0x8], ESI
0x6e6d1177 750a                            JNZ 0x6e6d1183
0x6e6d1179 837f0818                        CMP DWORD [EDI+0x8], 0x18
0x6e6d117d 0f846a030000                    JZ 0x6e6d14ed
0x6e6d1183 c745fcfeffffff                  MOV DWORD [EBP-0x4], 0xfffffffe
0x6e6d118a e841000000                      CALL 0x6e6d11d0
0x6e6d118f 57                              PUSH EDI
0x6e6d1190 ff750c                          PUSH DWORD [EBP+0xc]
0x6e6d1193 ff7508                          PUSH DWORD [EBP+0x8]
0x6e6d1196 56                              PUSH ESI
0x6e6d1197 ff152c106d6e                    CALL DWORD [0x6e6d102c]
0x6e6d119d e83effffff                      CALL 0x6e6d10e0
0x6e6d11a2 c20c00                          RET 0xc

[instructions after function returns cut from figure]
```

Figure 4: Analyzing message hooks using Volatility.

4. HookTracer

HookTracer is a custom emulator environment, which provides an API for programmatic analysis and observation of code in volatile memory samples. HookTracer is built on top of the *unicorn* emulator [62] and operates as a Python API for Volatility plugins. *unicorn* is an open source emulator written in C that contains Python language bindings. *unicorn* is used in many security projects [3], and it provides programmers with full control over emulated code.

Before the research and development described in this paper, HookTracer's main functionality and purpose was to enumerate the memory regions (VADs) that emulated code traversed. These capabilities were sufficient for analyzing API hooks [13], but are insufficient for properly understanding and evaluating message hooks. This occurs as the number of instructions encountered when emulating API hooks is significantly smaller than compared to message hooks and the use of system APIs in API hooks occurs much less frequently. The goal of nearly all malicious API hooks is to either filter data back to the caller, such as to remove a malicious process from the process list before returning it to an anti-virus engine, or to steal sensitive parameters sent to functions, such as passwords and encryption keys, and write them to disk. As shown later, malicious message hooks perform a wide array of activity, such as clipboard snooping, gathering the names of GUI elements, performing code injection, and much more.

The amount of work performed by message hooks compared to API hooks necessitated a significant amount of new research and development to integrate new functionality into HookTracer. To add these missing capabilities, our team developed a new set of features and APIs for HookTracer, totaling nearly 2,000 lines of new Python code. The main purpose of these additions was to 1) add full support for 32-bit executables and libraries, as the existing implementation was focused on 64-bit code, and 2) provide function call interception capabilities. Our new function call interception APIs allow HookTracer to internally monitor when particular Windows APIs are about to be called by emulated code and then provide callbacks for the following purposes:

- Stable emulation
- Supporting system resource access
- Faking return values of called functions
- Faking parameters sent to functions
- Recording parameters passed to emulated functions

The remainder of this section describes each of these issues in more detail.

4.1. Providing Stable Emulation

Whole system emulators, such as QEMU [9], emulate entire operating systems and, as a result, can allow an application full access to all resources, such as the network, filesystem, and other running processes. This also implies that global resources, such as semaphores, locks, and mutexes, can be queried and changed while the emulated code is running. Conversely, when emulating code from a memory sample, there is no operating system active and the sole actor is the code being emulated. This led to a few situations where we needed to internally implement certain Windows APIs to provide stable emulation. To accomplish this, before emulation begins, we record the runtime address of all exported functions of every loaded module within a hooked process. This is performed using Volatility's existing APIs. When problematic functions are later called by emulated code, HookTracer then overrides the native Windows implementation with a stable implementation provided by HookTracer itself.

The three categories where such implementations are provided are:

- Lock access - Since no other processes or threads are active within the emulated environment, the state of a lock is "stuck" at its value at the time of memory capture
- Memory region allocations - Handling of memory region allocation requests, such as through the *VirtualAlloc* API, requires in-kernel code that is not supported by the emulator
- Debugging APIs - These APIs allow reading and writing memory of other processes, none of which are present within the emulated environment

4.2. Supporting System Resource Access

Of particular importance to forensic investigators is malware's access and activity related to the filesystem, the registry, and the network. While our goal is to detect malware that leaves absolutely no traces in the local filesystem, we certainly do not want to miss detecting malware that does. Since these resources provide persistence, lateral movement, data gathering, and data exfiltration, we added extensive support for each.

4.2.1. Filesystem Access

Since there is no actual filesystem inside the emulated environment, HookTracker must monitor calls to filesystem related functions and provide implementations of important calls. For calls to open a file handle, such as *OpenFile* or *CreateFile*, HookTracer will return a unique handle from its internal handle table. For subsequent calls to read from the file handle, such as the use of *ReadFile*, HookTracer will attempt to read from the cached version of the file in memory through Volatility's API [75]. If the file is not cached, then HookTracer returns a buffer of null bytes. For write operations, HookTracer implements a "shadow" cached file and internally overlays written data on top of the cached file data. For calls to close a file handle, HookTracer simply removes it from the tracking array.

Another commonly used set of filesystem functions include those that return metadata about a file or directory. On Windows the most popular of these is *ZwQueryInformationFile* [4], which allows gathering all of a file's metadata. HookTracer's implementation of this function handles the most common requests to ensure that emulated code can execute properly. HookTracer also emulates the common *GetFileSize* call and attempts to return the size of the file reported by the file cache or otherwise simply returns a value of 256KB for the file.

4.2.2. Registry Access

Malware often uses the registry to gather data to be exfiltrated or to maintain persistence. Similar to how HookTracer now simulates filesystem access, it also now handles registry accesses. First, HookTracer maintains an in-memory registry tree to handle calls for creating registry keys and values. For queries to registry keys and values, if the requested node is not currently in the internal tree, then HookTracer attempts to access the in-memory registry data accessible through Volatility's APIs [75]. This allows the malware to branch based on real data where possible and also allows HookTracer to use the precise information gathered and stored by emulated code.

4.2.3. Network Access

Particularly when emulating malicious code, it is essential that HookTracer fakes as much network connectivity and activity as possible. Malware will commonly perform checks to see if it has internet access before executing its actual payload. Since many sandbox systems and malware analysis labs are isolated,

network connectivity checks by malware will often thwart automated and dynamic analysis. The widespread use of network connectivity checks by malware for this purpose led to the creation of projects such as FakeNet [31] and FakeNet-NG [23]. These software projects present fake instances of network services, such as DNS and HTTP, to malware running inside of automated analysis systems to trick the malware into thinking it has real internet access. Use of these projects is now common in the industry.

HookTracer implements a strategy similar to FakeNet in that it intercepts calls to network functions and returns fake data. Full discussion of returning faked data is discussed in the next subsection. For DNS resolution attempts, HookTracer simply returns the public IP address of Google. For calls to functions that perform HTTP requests, HookTracer will construct a reply that matches the requested protocol and file (or mime) type. For calls to receive raw data, such as *recv*, HookTracer returns the English alphabet repeating for the length requested. For network APIs that don't require generating fake data, such as *bind*, *accept*, and *socket*, HookTracer will fake return values that indicate success. Combined, this capability allows HookTracer to successfully emulate a significant amount of network activity generated by emulated code.

4.3. Faking Returned Data

To provide the most complete emulation possible, we added the ability for HookTracer to insert a "fake" return value for functions where emulation would not succeed or where emulation may not succeed. Faking a return value includes setting the return value (the EAX register on 32-bit systems and RAX on 64-bit systems) to zero or non-zero, which is the common Windows pattern, as well as filling in any data structure(s) that the calling code expects to be populated. The rest of this subsection describes the functions and function types that HookTracer now fakes by default for this purpose.

GetComputerName: Malware often gathers the network name of infected systems to definitively associate stolen data with a particular system. HookTracer fills the buffer expected by *GetComputerName* callers with a static "Windows7Desktop".

GetModuleFileNameW: This function is used to get the full path of an executable module loaded in a process. Malware often uses it to find the full path of where it is running from disk. HookTracer leverages Volatility's APIs to determine the full path of the module requested by the calling function and copies it to the passed in memory buffer.

System Time Functions: Malware often uses *GetSystemTime* to gather the local time of a system followed by functions, such as *SystemTimeToFileTime*, to convert the timestamp to a human readable value that can be logged to disk or sent across the network. To return a reasonable timestamp, HookTracer uses Volatility's APIs to determine the time of the system when the memory sample was acquired and returns its value.

Timestomping Functions: Timestomping [15] is a very popular anti-forensics technique used by malware to alter the timestamps of its files on disk.

To accomplish this, malware will call *GetFileTime* with a handle to a common Windows file, such as *kernel32.dll*, to get its timestamps. The malware will then use *SetFileTime* to copy these timestamps for the malicious file. This allows malicious files to blend in with other files during timeline analysis and filesystem anomaly detection. HookTracer handles this call sequence by returning timestamps matching the time when the memory sample was taken and allowing *SetFileTime* calls to succeed.

Filesystem Enumeration: The *FindFirstFile* and *FindNextFile* APIs allow programs to enumerate all files and sub-directories of a given directory. This functionality is often abused by malware to find files to infect, encrypt, delete, or exfiltrate. Since there is no actual filesystem present, HookTracer fakes a realistic looking directory structure.

Process Enumeration: The list of running processes is queried by malware for many purposes, such as finding processes to inject code into, searching for security monitoring software, or gathering a list of process names for exfiltration. Enumerating processes is accomplished through a set of calls to *CreateToolhelp32Snapshot*, *Process32First*, and *Process32Next*. HookTracer fakes the corresponding data structures for these calls and uses the actual process list generated by Volatility's APIs to return the processes active in the memory sample.

4.4. Faking Function Parameters

To trigger the real payload of a malicious function in memory, the parameters passed to the function must match those that the function expects. Since HookTracer emulates code in a standalone environment, it must first setup fake parameters to functions before initially emulating them. As described in Section 5.2, to trigger the payloads of malicious keylogger hooks, our plugin needs to use our new HookTracer APIs to create the parameters and data structures that match those that a running Windows system generates after each keystroke.

4.5. Recording Function Parameters

The original HookTracer implementation simply monitored which memory regions were accessed by emulated code, but to deeply examine malicious message hooks, we needed the ability to fully understand the functionality of emulated code. For example, the original HookTracer engine would simply list the fact that a certain number of basic blocks were executed inside of DLLs, such as *kernel32.dll* or *user32.dll*. Unfortunately, legitimate message hooks will use many APIs inside of these DLLs. To gain a deeper understanding, we needed to know which Windows APIs were actually used and what parameters were sent to them. To provide this capability, we developed an API where a list of Windows APIs to be monitored can be passed in, and then for each call to a monitored function by emulated code, the HookTracer analysis plugin will receive a callback notification. It can then inspect the name of the function being called as well as its parameters. The incredible insight that can be gained with this feature is shown in Section 6.3 as analysis of the Turla malware is showcased.

5. Message Hooks Analysis

5.1. hooktracer_messagehooks

To automate analysis of message hook handlers and to remove the need for manual reverse engineering, we developed a new Volatility plugin, *hooktracer_messagehooks*. Our plugin leverages the new and improved *HookTracer* API to emulate message hook handlers and then runs a variety of analysis results to aid the investigator. To use the plugin, an investigator first runs the Volatility *messagehooks* plugin, saves its output, and then feeds this output to *hooktracer_messagehooks*. We chose to use the output of *messagehooks* as this allows the investigator to quickly run our plugin a number of times with different analysis options, without having to regenerate the hook data. Furthermore, since Volatility is a command line tool, it is already common practice for investigators to run individual plugins manually or within a script and save the output to files. For the remainder of this section, we explain how *hooktracer_messagehooks* works in more detail and in the next section we document a case study using the plugin against the Turla malware.

5.2. Faking a Keystroke

Before our plugin begins emulation of a hook, it first creates the data structures that would be generated by a user typing 'A'. These data structures are then mapped into the emulated address space through the HookTracer API. When emulation of the hook begins, the hooked code will read our "fake" values and keystroke data. This is necessary, as keyloggers typically check the state of the keyboard and the action that generated the keystroke before executing their real payload. As shown in Figure 5, message hook handlers are sent three parameters for each event.

```
LRESULT CALLBACK KeyboardProc(int nCode,
WPARAM wParam, LPARAM lParam)
```

Figure 5: Function prototype for a message hook handler.

The first parameter, *nCode*, can assume several values. These are *HC_ACTION* or *HC_NOREMOVE*. Keyloggers will verify that the value is *HC_ACTION* so that they are ensured they have received a current keystroke. The second parameter, *wParam*, is the action that generated the keystroke event. Keyloggers generally filter for *WM_KEYDOWN* events, but can also monitor for a *WM_KEYDOWN* followed by a *WM_KEYUP*. The third parameter, *lParam*, describes the key's value and associated attributes (i.e., it is an extended key, etc.).

Figure 6 shows the initial code of the keyboard event handler of the Gozi malware. Gozi steals keystrokes and passwords, captured screenshots, and infected systems throughout the world. Its source code was eventually leaked online and is still accessible on GitHub [12]. As shown, before processing the

```
if ( !g_bLoggerEnabled || ( nCode != HC_ACTION ) || !HookStruct ){
break;
}

if ( (UINT)wParam != WM_KEYDOWN ){ //message
break;
}
```

Figure 6: Leaked Gozi source code.

keystroke and executing its malicious payload, Gozi first checks that the action matches *HC_ACTION* and that the event is *WM_KEYDOWN*. Analysis of Gozi's message hook handler is is described in Section 7.3.

5.3. Reporting Malicious Hooks

An important feature of our plugin is the ability to automatically determine if a message hook is malicious or benign without intervention by an analyst. A study of the previously referenced Microsoft documentation for *SetWindowsHookEx* shows that its legitimate uses include functionality such as hot keys, 'magic' keys, reader accessibility enhancements, and on-screen training modules (CBT). All of these legitimate uses only access keystroke data for a short time and typically keep the data solely within memory. Furthermore, legitimate hooks do not typically perform other actions indicative of data exfiltration. We consider a message hook as potentially malicious (and certainly worthy of investigation) if any of the following conditions are met:

1. Keystroke data is logged to a file
2. Keystroke data is sent over the network
3. Keystroke data is written to the registry
4. Contents of the clipboard are accessed
5. Screenshots are generated
6. Debugging APIs are used (code injection)
7. The hook immediately calls *CallNextHookEx*

The first three criteria cover malware that exfiltrate keystrokes. Given the security risk, there is no reason for legitimate applications to store keystrokes in local storage or over the network. Criterion 4 exists as keylogging malware will often grab the clipboard contents at the same as the latest keystroke in an attempt to capture passwords copied from password managers or text files. The stealing of clipboard contents was initially popularized by banking trojans and now occurs in a wide variety of malware. Criterion 5 covers many malware families, such as Zeus [33], that take screenshots of victims' systems in their message hook handler when targeted applications, such as web browsers, are in the foreground. This allows the malware operator to see the websites visited by a user, view visible username values, and potentially sensitive information, such as bank account numbers. Criterion 6 covers debugging APIs as they are used

for code injection as well as for scanning memory of the process the malware is active in. There is no reason for this activity to occur in legitimate message hook handlers.

To monitor such activity, our plugin compares each API call made by an emulated hook and matches it to a list of functions commonly used to implement the malicious activities. If any of these calls are found then the hook is reported as malicious. The only exception to this is our monitoring of hooks that perform no action except to immediately call the *CallNextHookEx* function. *CallNextHookEx* tells the runtime system that the currently executing hook handler has finished processing the event and that the next one in the chain can be called. Malware will often abuse *SetWindowsHookEx* to inject a malicious DLL into many processes, but then not actually analyze the requested hook. This allows the injected DLL to perform other malicious actions without concern for specific keystrokes or other actions by the victim users. Laqma [44] was one of the first advanced malware samples to use this technique. The infamous Carberp malware, which was used to steal an estimated $250M USD from victims [38], also utilized this technique, as illustrated in its leaked source code [57].

5.4. Generating Trace Records

Beyond making a simple yes/no determination of a hook being malicious, *hooktracer_messagehooks* can also generate a listing of the precise activity performed by a hook. This base version of a trace will provide a list of the APIs called by the hook in the order they were called. The extended version will report the parameters sent to a select set of functions. This includes APIs related to filesystem, network, and registry activity, string manipulation, creation of mutexes and atoms, and code injection. By reviewing the list, an analyst can determine the exact resources accessed by a particular hook.

Once a malicious hook is identified, the investigator can then re-run *hooktracer_messagehooks* to generate a trace record of the hook. This record can be used later to automatically re-identify the malicious hook on the same system or other systems. Multiple trace records can also be combined for analysis in one plugin run as *hooktracer_messagehooks* supports newline-separated input files that can include any number of malware traces. This has many advantages, including allowing entire teams to pool their gathered trace records into a single source. The use of trace records provides full automation and repeatability of analysis, regardless of the skill level of the investigator performing analysis.

5.5. Generating IOCs

Beyond trace records that can be reused by *hooktracer_messagehooks*, the plugin can also automatically generate standard indicators of compromise (IOCs). These can then be used across a variety of incident response frameworks, endpoint security monitoring agents, and security information and event management (SIEM) systems. IOCs are ubiquitously used throughout the security industry to describe malware based on the filesystem, registry, and network

activity performed. By generating IOCs for message hooks, our plugin allows incident response teams to search for threats throughout their environment in a consistent manner, using existing workflows.

5.6. Monitoring Data Transformations

Keyloggers often transform logged data to avoid detection by security scanners looking for common logging patterns in files on disk and in network traffic. The most common transformations consist of arithmetic operations, such as XOR and ROL, but some malware also uses "real" cryptography. To aid investigators analyzing malware that encrypts logged data, *hooktracer_messagehooks* monitors for transformations of gathered keystrokes. In situations where a transformation occurred, the plugin will attempt to automatically generate a script capable of decrypting a given key log file.

hooktracer_messagehooks currently supports automated decoding and script generation against keyloggers that leverage the XOR and ROL operations with a static key (shift value). To accomplish this, the plugin monitors for transformation that utilizes these instructions, and when detected, records the source value. For XOR, this is the integer key used to transform the destination value. For ROL, this is the integer that specifies by how many bits to shift the data. When a static key (shift) is used for every offset of transformed data, then *hooktracer_messagehooks* generates a simple Python script that takes a file path from the command line, transforms every byte of the file with the monitored operation and static key, and then writes it to an output file. When used in conjunction with the plugin's automatic component extraction, this allows an investigator to decrypt all previously logged data on infected systems.

For transformations beyond XOR and ROL, the plugin currently reports the address ranges of the instructions that transformed the data. This tells an investigator where to begin analyzing the encryption routine. This process does then require reverse engineering, but our plugin pinpoints where this effort should begin and provides the associated data that must analyzed.

5.7. Automatically Extracting Components

Analysts can optionally instruct *hooktracer_messagehooks* to automatically extract all components referenced by a malicious hook. When enabled, the following artifacts are extracted as they are encountered:

- The executable or memory region hosting the malicious hook
- Files written to or read
- Registry data
- Network data

Extracting malicious executables directly from memory allows the investigator to perform further binary analysis tasks, such as reverse engineering with IDA Pro or running signature checks. This does not provide the capability to

load the executable into a traditional sandbox, however, as substantial transformations occur during runtime loading. Extracting files, registry keys, and network data from memory allows an investigator to determine the extent of the infection on the system being investigated as well as view the exact data being referenced and generated by the keylogger.

For extracted components, files are created using a naming convention of *<file|reg|http> . <pid> . <thread id> . <instruction address> . <instruction count>.extracted*. This convention ensures that all created files are unique and conveys the context for extracted data, enhancing investigative efficiency as well as documentation efforts associated with legal proceedings.

6. Case Study: Turla

6.1. The Turla Malware

Turla is both the name of a high-profile advanced persistent threat (APT) group as well this group's digital espionage platform. As documented by MITRE [16] and Kaspersky [40], the Turla group is responsible for compromising victims in over 45 countries, with the majority of the victims belonging to government agencies, military departments, and embassy operators. It is widely believed that the Turla team is Russia's most advanced hacking group inside of its intelligence agencies, and its past attack campaigns have involved hacking satellites to target victims in remote areas and compromising entire ISPs to deliver targeted malware to a single victim [14].

Of the many capabilities provided by the Turla espionage platform, logging of keystrokes and environmental data is a central focus. As part of its payload, Turla leverages *SetWindowsHookEx* to gather and record keystrokes along with other system data. As documented in two lengthy blog posts by *malware.news* [48, 49], Turla's hook handler performs a substantial number of operations per-keystroke and to manually uncover the actions taken requires days of expert-level reverse engineering. To showcase our HookTracer engine and our *hooktracer_messagehooks* plugin, we now present each feature of the plugin as it analyzes Turla's malicious message hooks.

6.2. Analysis Environment

6.2.1. Operating Environment

To execute Turla, we setup a new VMWare Fusion virtual machine running the 64-bit Professional version of Windows 7. We then allowed the virtual machine to install all security and operating system updates.

6.2.2. Infecting with Turla

The MD5 hash of the Turla sample used in our analysis is:

59b57bdabee2ce1fb566de51dd92ec94

This sample is a DLL deployed by the Turla platform during operations. To activate the DLL, we leveraged *rundll32.exe* to execute the DLL as if it was a traditional application executable (EXE file). This commonly used technique is documented in recipe 13-2 of the Malware Analyst's Cookbook [43].

6.2.3. Post-Infection Actions

After using *rundll32* to execute the malware, we launched Notepad and typed in a sentence followed by opening Internet Explorer and browsing to Google. The purpose of these actions was to trigger events monitored by Turla in processes whose activity we controlled. We then waited 30 seconds and suspended the virtual machine. When the virtual machine was suspended, a complete copy of RAM at the time of suspension was written to disk. This provided a near-instantaneous capture of memory, and is a standard method of collecting memory captures infected with a particular malware sample [43, 24].

6.3. Hook Handler Analysis

When run in its default mode, *hooktracer_messagehooks* lists only the message hooks that it determines to be malicious based on the criteria previously listed. This allows an investigator to quickly determine, with confidence, if malware utilizing malicious hooks is present on the system. If an investigator wishes to observe the behaviour of a hook in detail, they can then run the plugin with the *–list-apis* option set. This will instruct the plugin to list, in order, all exported functions called by an emulated hook. The plugin can also be run with *–list-apis-condensed* option set, which instructs it to only list functions that match the built-in filter of suspicious and malicious functions. In both modes, functions whose parameters are of interest to investigators are listed. Figure 7 shows the output of *hooktracer_messagehooks*'s condensed API listing mode against an instance of Turla's hook. Note that line numbers have been added in the figure to aid the discussion.

By simply reading the output, an investigator can determine the hook's functionality and make the same determination as the plugin's automated engine, namely, that the hook is malicious. Lines 1-5 illustrate building a string that includes the handle of the current window and the system time. HookTracer reports *sprintf* related functions by showing both the format specifier sent to the function as well as the buffer filled in by the function. Lines 6-7 show gathering of the victim process' process ID (1084) and associated formatting. Lines 8-10 show the filename of the process being gathered and saved. Line 11-12 shows the Window Text as returned by HookTracer. All of the gathered values are then concatenated together by the *memmove* call on line 13. Lines 14-17 show the common API sequence used to convert a keyboard input to a Unicode character (*ToUnicodeEx*). On line 18, our fake keystroke ('A') is then sent to *swprintf*. Lines 19-26 show the file path of the keylogger file being built, and line 27 shows the *CreateFile* call to open a handle to the file. Note that the *41424344* in the output is the fake handle value assigned by HookTracer. After the file is opened, successive calls to *memmove* are used to concatenate a header-type

```
1.  GetForegroundWindow
2.  GetSystemTime
3.  SystemTimeToFileTime
4.  swprintf: [%02d.%02d.%04d %02d:%02d:%02d.%03d] \
              | [24.1.2019 20:43:38.0]
5.  swprintf: [h]:%d | [h]:1234
6.  GetWindowThreadProcessId
7.  swprintf: [pid]:%d | [pid]:1084
8.  OpenProcess: 1084
9.  GetProcessImageFileNameW
10. swprintf:[pn]:%s | [pn]:turla.dll
11. GetWindowTextW
12. swprintf: [t]:%s | [t]:MyWindowName
13. memmove: [24.1.2019 20:43:38.0] [h]:1234  \
             [pid]:1084 [pn]:turla.dll [t]:MyWindowName
14. GetKeyboardState
15. GetKeyboardLayout
16. MapVirtualKeyExW
17. ToUnicodeEx
18. swprintf: %c | A
19. GetModuleFileNameW
20. lstrcatw: C:\Users\bob\Desktop\
21. lstrcatw: SPUNINST
22. lstrcatw: C:\Users\bob\Desktop\
23. FindFirstFileW
24. lstrcatw: SPUNINST
25. lstrcatw: msimm.dat
26. FindClose
27. CreateFileW: 41424344 -> C:\Users\bob\Desktop\SPUNINST\msimm.dat
28. memmove: KSL0T Ver=21.0
29. memmove: [u]:
30. memmove: WIN-948O8I1DO91\bob
31. memmove: [24.1.2019 20:43:38.0] [h]:1234 \
             [pid]:1084 [pn]:turla.exe [t]:MyWindowName
32. memmove: [24.1.2019 20:43:38.0] [h]:1234 \
             [pid]:1084 [pn]:turla.exe [t]:MyWindowName A
33. WriteFileW: 41424344 -> 278
34. CloseHandle: 41424344
35. CallNextHookEx
```

Figure 7: hooktracer_messagehooks against Turla.

value (*KSL0T Ver=21.0*), the Windows computer or domain name, username of the logged on user, the previously gathered environmental data, and finally the 'A' keystroke. The hook then writes to the previously opened file, closes the file handle, and calls *CallNextHookEx*, as its work is completed for the current keystroke.

All of the previous analysis was accomplished without any reverse engineering effort by the plugin's user. Results are also obtained quickly as the plugin's analysis time for each hook is less than a minute in our test Debian virtual machine, to which we assigned a mere 2 CPU cores and 2GB of RAM.

6.4. Automatic Component Extraction

For functions that interact with the filesystem, registry, or network, *hooktracer_messagehooks* can be instructed to automatically save the referenced components to the investigator's system. In the previous Turla analysis, this leads to extraction of the file written to by the keylogger:

C:/Users/bob/Desktop/SPUNINST/msimm.dat.

6.5. Logfile Decoding

Turla encrypts recorded data using an array of XOR keys. The array is indexed based on the position in the file of the byte being written. Since the XOR key is not static, the plugin cannot automatically produce a script capable of decoding any log file produced by the keylogger. It does isolate the encryption loop, however, which saves a tremendous amount of analysis time. When used in conjunction with automatic component extraction, the malicious DLL will also be automatically written to disk. The analyst can then perform any additional reverse engineering, as needed, beginning analysis at the exact address where logging data is transformed.

Figure 8 shows where this process leads an investigator for Turla's hook. A simple XOR loop is used by the malware, which then calls *WriteFile* followed by *CloseHandle*. This same API call sequence was listed in Figure 7. In the IDA Pro screenshot, we have annotated some of the local variables based on their parameter position to *WriteFile*. Making this annotation is a very simple operation since *WriteFile* and its parameters are documented on the MSDN. After annotating the *logged_buffer* and *buffer_length* parameters, it becomes obvious where they are used in the XOR loop. We then see where the current value to XOR comes out of a large array that we renamed to *encryption_keys*. From here, it would be a very straightforward operation to write a custom decoder for strings encrypted with the malware. Given our extensive experience in scoping malware analysis projects, we estimate the entire reversing task described here would take a novice reverse engineer around 30 minutes and around 10 minutes for an expert.

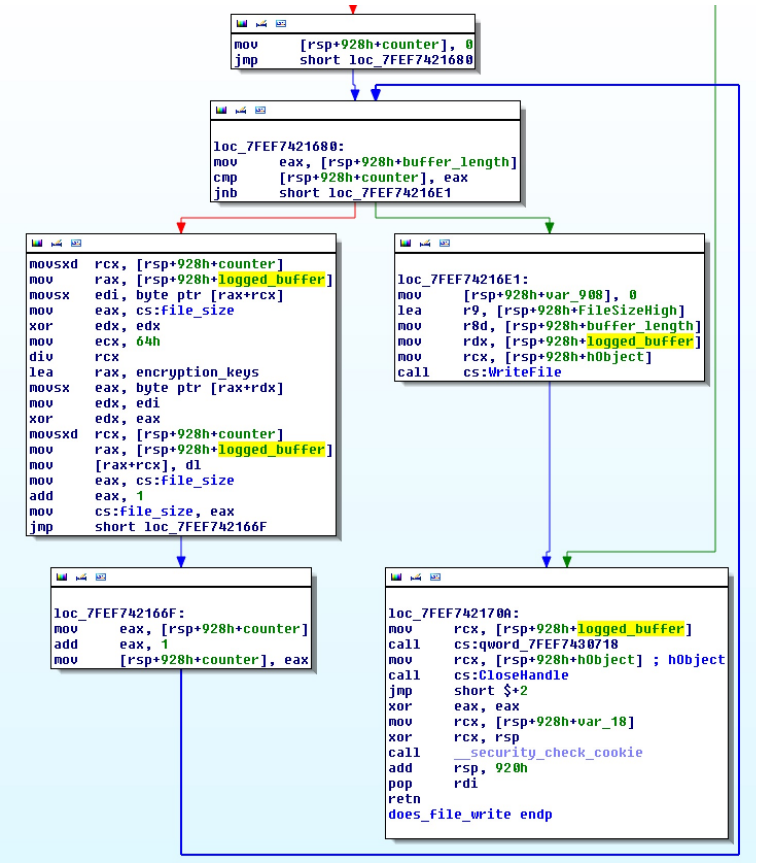

Figure 8: Analysis of Turla XOR loop in IDA Pro.

6.6. Creating a Trace Record

Figure 9 shows the trace record for Turla's hook, which includes all of the functions shown in Figure 7 except for those related to string and buffer manipulation. In our experience with malware analysis, we have observed malware samples use new string manipulation functions, such as changing *sprintf* to *snprintf*, in updated versions. This was generally done when the malware updated its on-disk format of logged data or when it attempted to fix vulnerabilities in its code. To avoid missing instances of such variations, we developed HookTracer to only record use of core Windows APIs whose functionality cannot be easily or directly replaced by calling a different function.

6.7. Generating IOCs

6.7.1. IOC Creation and Use

As mentioned previously, HookTracer can generate indicators of compromise files to support existing workflows used by incident response teams. For Turla's

```
[GetForegroundWindow,GetSystemTime,SystemTimeToFileTime
,GetWindowThreadProcessId,OpenProcess,
GetProcessImageFileNameW,GetWindowTextW,GetKeyboardState,
GetKeyboardLayout,MapVirtualKeyExW,ToUnicodeEx,
GetModuleFileNameW,CreateFileW,WriteFileW,CloseHandle]
```

Figure 9: Trace record for Turla.

hook, this would include three records: 1) the full path to the log file, 2) the *SPUNINST* sub-directory, and 3) the *msimm.dat* file. HookTracer generates separate records for several reasons. First, the full path can be highly specific to our emulator. Turla is an example of malware that writes its log file to the directory that the malware is launched from, which obviously changes per infection. Other keyloggers write to hardcoded paths though, which is why we still include the full path. Second, we include any sub-directories created by the malware as experience tells us that some malware will create a hardcoded directory name, but then vary the name of the file inside of it. Finally, we keep the name of the keylogger file itself for malware, such as Turla, that use hardcoded names. By keeping all components, the generated IOCs are as flexible and broad as possible. Once generated, an incident response team member can then feed the IOC into any enterprise-level endpoint security monitoring (EDR) product to determine every system in the environment that contains file(s) matching the IOC. When using industry-standard EDRs, thousands of endpoints can be checked in an under a minute, and the use of IOCs in this manner is standard practice in the industry [18, 37, 68].

7. Hooktracer Testing and Limitations

To test HookTracer's correctness and viability, we tested it against a number of other real-world malware samples to ensure that the reported hook behaviour matched that of each sample. In this section, we document the results of this work for several prolific malware samples. For 64-bit samples, we leveraged the same Windows 7 64-bit virtual machine used for the Turla case study. For 32-bit samples, we used a 32-bit version of Windows 7 inside of a virtual machine.

7.1. Loki Bot

Loki Bot is an information stealer sold on underground online markets that targets credentials and keystrokes, and that has been used in a variety of real-world attack campaigns [51, 70]. Loki logs keystrokes through the use of SetWindowsHookEx, and its message hook handler gathers the last pressed key, the contents of the clipboard, and the name of the active window.

7.1.1. Setup

Our analysis was performed against a 32-bit executable sample of Loki that has a MD5 hash value of:

eccad903b4c27d149e159338f58481a9

In order to activate the keylogger functionality, we followed the steps as described in Rob Pantazopoulos' research paper [58]. Once activated, we then typed letters into Notepad, Wordpad, and Internet Explorer, waited thirty seconds, and then suspended the virtual machine.

7.1.2. Analysis

Figure 10 shows the output of our plugin against the message hooks handler of Loki, and the malicious nature of the handler is clear. After the handler locates and calls APIs to retrieve the keystroke value (lines 1-16), it then calls *GetWindowTextW* on line 18. Next it opens a handle to a strangely named file (line 22) and writes the return value of *GetWindowTextW*, whose value was faked by HookTracer (*MyWindowName*), to the file. It then reads the contents of the clipboard (line 30), opens a a new handle the same file (line 35), and writes out the clipboard contents to the file (line 38). Loki prefixes clipboard contents with *CB:* and HookTracer fakes *GetClipboardData* to return a value of *BBBBCCCCDDDD*. Finally, on lines 43-46 we see the handler writing the faked keystroke *A* to the file.

Loki's message hook handler triggers several of our malicious criteria, including clipboard access and writing clipboard and keystroke data to a file, and is automatically flagged as malicious by the plugin.

7.2. Turla Keylogger Module

The Turla group previously described has developed several distinct toolkits over the years. In its description of "The Epic Turla Operation", Kaspersky researchers documented a keylogger deployed by Turla that relied on *SetWindowsHookEx*, but that is distinct from the previously analyzed sample [35].

7.2.1. Setup

Our analysis was performed against a 32-bit Epic Turla keylogger sample that has a MD5 hash value of:

a3cbf6179d437909eb532b7319b3dafe

In order to gather a sample for analysis, we executed the malware to activate it, typed into Notepad, Wordpad, and Internet Explorer, waited thirty seconds, and then suspended the virtual machine.

7.2.2. Analysis

Figure 11 shows the output of our plugin against the message hook handler of the Epic Turla keylogger. Unlike the previously discussed malware, this handler performs only one task, which is to append the current keystroke to the key log file. For unknown reasons, the keylogger exports its message hook function *_LowLevelKeyboardProc@12*, which causes it to appear as line 1 in the output.

```
1.  LoadLibraryW: user32 -> 77200000
2.  GetForegroundWindow
3.  LoadLibraryW: user32 -> 77200000
4.  GetWindowThreadProcessId
5.  LoadLibraryW: user32 -> 77200000
6.  GetKeyboardState
7.  LoadLibraryW: user32 -> 77200000
8.  GetKeyboardLayout
9.  LoadLibraryW: user32 -> 77200000
10. ToUnicodeEx
11. LoadLibraryW: user32 -> 77200000
12. GetAsyncKeyState
13. LoadLibraryW: user32 -> 77200000
14. GetKeyState
15. LoadLibraryW: user32 -> 77200000
16. GetForegroundWindow
17. LoadLibraryW: user32 -> 77200000
18. GetWindowTextW
19. LoadLibraryW: user32 -> 77200000
20. wsprintfW
21. wvsprintfW
22. CreateFileW: C:\Users\Administrator\AppData\Roaming\4C0383\34A037.kdb
23. SetFilePointer
24. GetProcessIdOfThread
25. WriteFile: 41424344 -> 48 -> Window: MyWindowName
26. CloseHandle
27. LoadLibraryW: user32 -> 77200000
28. OpenClipboard
29. LoadLibraryW: user32 -> 77200000
30. GetClipboardData
31. GlobalLock
32. LoadLibraryW: user32 -> 77200000
33. wsprintfW
34. wvsprintfW
35. CreateFileW: C:\Users\Administrator\AppData\Roaming\4C0383\34A037.kdb
36. SetFilePointer
37. GetProcessIdOfThread
38. WriteFile: 41424344 -> 28 -> CB: BBBBCCCCDDDD
39. CloseHandle
40. GlobalUnlock
41. LoadLibraryW: user32 -> 77200000
42. CloseClipboard
43. CreateFileW: C:\Users\Administrator\AppData\Roaming\4C0383\34A037.kdb
44. SetFilePointer
45. GetProcessIdOfThread
46. WriteFile: 41424344 -> 2 -> A
47. CloseHandle
48. LoadLibraryW: user32 -> 77200000
49. CallNextHookEx
```

Figure 10: hooktracer_messagehooks against Loki Bot.

```
1.  _LowLevelKeyboardProc@12
2.  memset: cfffffe8 - 0 - 20 - 20
3.  strlen: cfffffe8 - 0 -
4.  sprintf: cfffffe8 | %c | a
5.  vfprintf:  %s | a
6.  fflush
7.  CallNextHookEx
```

Figure 11: hooktracer_messagehooks against Epic Turla.

```
$ python vol.py -f infected.mem  --profile=Win7SP1x86 handles -p 2724 -t File
Volatility Foundation Volatility Framework 2.6.1
Offset(V)  Pid  Handle Access   Type Details
---------- ---- ------ -------- ---- -------
0x854ff5b0 2724 0x8    0x100020 File <snip>\Users\Administrator\Desktop
0x8713dba8 2724 0x44   0x12019f File <snip>\Users\ADMINI~1\AppData\Local\Temp\~DFD3O8.tmp
```

Figure 12: Finding the Log File.

Lines 2-4 show the zeroing out of a buffer at address 0xcfffffe8 followed by the copying of a lowercase *a* into it.

This *a* is actually our fake keystroke of *A* in lowercase form. The case is inverted as, instead of using the system APIs to translate the keyboard code to a character, the hook does an unusual combination of checking for special keys (shift, cap locks, etc.). Since these are not faked by HookTracer, the keylogger calculates the keystroke as being in lowercase. The keystroke is then written to a file on line 5 using an already opened file handle, the file buffer flushed to disk on line 6, and the hook terminates on line 7. This behaviour meets our criteria of not writing keystrokes to disk and is automatically flagged as malicious.

7.2.3. Finding the Log File

Since the malware uses an already opened file handle to write to the key log file, the name of the file is not immediately obvious from the plugin's output. This straightforward for analysts to determine, however, as *hooktracer_messagehooks* can be configured to report the process ID of analyzed hooks, which for this sample is *2724*.

Volatility's *handles* plugin can then be run with filters set for that PID and to only show file handles. Figure 12 shows this invocation of Volatility and the subsequent output. As can be seen, only two file handles are opened by the malware. The first is a reference to the Desktop, and the second to a strangely named file under the user's temporary folder. Volatility's *dumpfiles* plugin could then be used to extract the file and verify its contents.

7.3. Gozi

Gozi, which also goes by Ursnif or ISFB, is a banking trojan that has been around since the mid-2000s [64] and is still actively used in attack campaigns today [67]. It has undergone significant changes during this period and also

inspired related malware, such as GozNym [65]. Combined, the Gozi family of malware is responsible for the theft of hundreds of millions of dollars.

7.3.1. Setup

Our analysis was performed against a 32-bit Gozi sample that has a MD5 hash value of:

e6d118192fc848797e15dc0600834783

In order to gather a sample for analysis, we executed the malware to activate it, typed into Notepad, Wordpad, and Internet Explorer, waited thirty seconds, and then suspended the virtual machine.

7.3.2. Analysis

Figure 13 shows the output of *hooktracer_messagehooks* against the system infected with Gozi. This hook operates by attaching the infected threads input queue to its own (lines 6, 10, and 44), gathering the name of the module it is executing inside of (lines 4, 11, 14, 16, 38, and 39), gathering the system time (line 41), and gathering the name of the window in which is executing (line 42).

The use of the debug APIs by this handler meets our criteria and is automatically flagged as suspicious. Manual review of the output also shows behaviour very consistent with a keylogger and would trigger an analyst to perform further analysis of the malware.

7.4. Telebot Keylogger

Telebots is an APT group believed to be based out of Russia. There are previously attributed to attacks against the Ukrainian power-grid as well as the NotPetya ransomware outbreak [52, 21]. The keylogger analyzed in this section was used in the second wave of attacks against the Ukrainian infrastructure. It was part of a toolchain that ended with the KillDisk malware, which deletes important user and system files and renders victim systems unbootable.

7.4.1. Setup

Our analysis was performed against a 64-bit sample that has a MD5 hash value of:

4919569cd19164c1f123f97c5b44b03b

In order to gather a sample for analysis, we executed the malware to activate it, typed into Notepad, Wordpad, and Internet Explorer, waited thirty seconds, and then suspended the virtual machine.

```
1.  GetCurrentThreadId
2.  GetProcAddress: GetForegroundWindow -> 7721335d
3.  GetForegroundWindow
4.  GetWindowThreadProcessId
5.  GetProcAddress: AttachThreadInput -> 77236b54
6.  AttachThreadInput
7.  GetProcAddress: GetFocus -> 77213a34
8.  GetFocus
9.  GetProcAddress: AttachThreadInput -> 77236b54
10. AttachThreadInput
11. GetWindowThreadProcessId
12. GetProcAddress: AttachThreadInput -> 77236b54
13. AttachThreadInput
14. OpenProcess: pid = 1084
15. GetProcAddress: GetModuleBaseNameA -> 774515a4
16. GetModuleBaseNameA
17. _strupr
18. lstrlenA
19. lstrlenA
20. CloseHandle
21. memset: cffffe80 - 0 - 38 - 38
22. memset: cffffd80 - 0 - 256 - 256
23. GetAncestor
24. GetProcAddress: GetKeyboardState -> 77236946
25. GetKeyboardState
26. GetProcAddress: GetKeyboardLayout -> 77213800
27. GetKeyboardLayout
28. GetProcAddress: GetAsyncKeyState -> 7720a256
29. GetAsyncKeyState
30. GetProcAddress: GetAsyncKeyState -> 7720a256
31. GetAsyncKeyState
32. GetProcAddress: GetAsyncKeyState -> 7720a256
33. GetAsyncKeyState
34. GetProcAddress: ToUnicodeEx -> 772221b2
35. ToUnicodeEx
36. RtlAllocateHeap: 19012 | 4a44 || 20480 | 5000 -> b0200000
37. memset: b0200000 - 0 - 19012 - 19012
38. OpenProcess: pid = 1084
39. GetModuleFileNameExW
40. CloseHandle
41. GetSystemTimeAsFileTime
42. GetWindowTextW
43. GetProcAddress: AttachThreadInput -> 77236b54
44. AttachThreadInput
45. GetProcAddress: CallNextHookEx -> 7720abe1
46. CallNextHookEx
```

Figure 13: hooktracer_messagehooks against Gozi.

7.4.2. Analysis

Figure 14 shows the output of *hooktracer_messagehooks* against a system infected with the Telebots keylogger. On lines 3-10, the output shows that process ID of the host process is gathered and written to a log file. This log file is stored in a suspiciously named file under the user's temp folder, which is a common location for malware to store data. Lines 11-20 show the malware writing the window name to the log file.

Lines 18-27 show the malware gathering the name of the executable it as running as and writing it to the log file. This is accomplished through the use of *CreateToolhelp32Snapshot* and *Process32FirstW*. These functions are used to begin walking the process list. The malware walks the list in order to find the process that it is running as so that it can extract the name. HookTracer fakes the name of the first process returned as *fake_process.exe*, which can be seen in the output on line 25. HookTracer also returns the same PID in calls to *GetWindowThreadProcessID* as it does for the fake process it returns from *Process32FirstW*. The combination of these two functions used together occurs often in malware so this increases the chances that malware will find "itself" during emulation. Lines 31-36 show the malware converting the faked *A* key to a Unicode character (*ToUnicodeEx*) and then writing it to the log file.

In summary, this hook gathers to name and PID of the process it is running as, the name of the active window when the latest key was pressed, and the Unicode value for the last key pressed. It then writes these to a log file. These actions violate several of our criteria, including the use of debug APIs and writing keystroke data to disk. This automatically triggers the hook being marked as suspicious by the plugin.

7.5. Limitations of Automated Message Hook Analysis

In order for *hooktracer_messagehooks* to identify a hook as malicious, the hook must violate at least one of the criteria described previously as being suspicious. The yty malware framework leveraged by the Donot Team APT group is an example of keylogger malware that leverages *SetWindowsHookEx* for keylogging, but does not violate any of the criteria [69].

Figure 15 shows the output of *hooktracer_messagehooks* against the yty keylogger. As can be seen, the hook's only operations are to convert the pressed key to its character equivalent and to allocate and de-allocate a few memory regions. Reverse engineering of the handler showed that it was storing the converted keypress inside of a custom data structure, and only once a certain number of keys were pressed did the hook write the stored keys to disk. Since HookTracer emulates only one key press, it did not trigger this extended behaviour.

Although our plugin does miss the malicious activity of this particular hook, we still strongly believe that our research is highly practical and of great real-world use. To start, the approach taken by yty to only record keystrokes once a certain number is reached is very rarely seen in real world malware as it can lead to lost keystrokes. For example, if the hosting process is terminated between a set of keystrokes and the threshold being reached, then none of them

```
1.  GetForegroundWindow
2.  GetWindowTextW
3.  GetWindowThreadProcessId
4.  CreateFileW: C:\Users\JOHNSM~1\AppData\Local\Temp\_klg2249768.~tmp
5.  GetFileType
6.  RtlAllocateHeap: b0200000
7.  SetFilePointer
8.  WriteFile: 41424344 -> 34 -> [*]Window PID >
9.  HeapFree
10. CloseHandle
11. CreateFileW: C:\Users\JOHNSM~1\AppData\Local\Temp\_klg2249768.~tmp
12. GetFileType
13. RtlAllocateHeap: b0202000
14. SetFilePointer
15. WriteFile: 41424344 -> 26 -> MyWindowName
16. HeapFree
17. CloseHandle
18. CreateToolhelp32Snapshot
19. Process32FirstW
20. CloseHandle
21. CreateFileW: C:\Users\JOHNSM~1\AppData\Local\Temp\_klg2249768.~tmp
22. GetFileType
23. RtlAllocateHeap: b0204000
24. SetFilePointer
25. WriteFile: 41424344 -> 42 -> [*] IMAGE : fake_process.exe
26. HeapFree
27. CloseHandle
28. GetForegroundWindow
29. GetWindowThreadProcessId
30. GetKeyboardLayout
31. ToUnicodeEx
32. CreateFileW: C:\Users\JOHNSM~1\AppData\Local\Temp\_klg2249768.~tmp
33. GetFileType
34. RtlAllocateHeap: b0206000
35. SetFilePointer
36. WriteFile: 41424344 -> 2 -> A
37. HeapFree
38. CloseHandle
39. CallNextHookEx
```

Figure 14: hooktracer_messagehooks against Telebots Keylogger.

```
1.  GetKeyNameTextW
2.  RtlAllocateHeap
3.  RtlAllocateHeap
4.  RtlAllocateHeap
5.  GetKeyState
6.  GetKeyState
7.  RtlAllocateHeap
8.  HeapValidate
9.  HeapFree
10. RtlAllocateHeap
11. HeapValidate
12. HeapFree
13. RtlAllocateHeap
14. HeapValidate
15. HeapFree
16. HeapValidate
17. HeapFree
18. HeapValidate
19. HeapFree
20. CallNextHookEx
```

Figure 15: hooktracer_messagehooks against Donot Team's Keylogger.

will be logged. Similarly, if the user logs off or shuts down the system in the gap time, keystrokes will again be lost. Further driving our belief is that, even in rare occurrences such as yty's approach, our plugin still tells an analyst that the executable processes the keystroke, and that further reversing is needed to figure out what is done with it. This itself is a clue that points investigators in the right direction. We believe the powerful automation provided by *hooktracer_messagehooks* to accurately describe the actions of message hooks is a significant and novel memory forensic capability.

8. Related Work

The use of emulation to evaluate malware has a long history in the field of computer security. The most common form of emulation for this purpose is *whole system emulation*. In this model, the entire operating system as well as all running applications are emulated. This allows fine grained inspection and control of the running system by monitoring applications. QEMU and Bochs [41] are the most commonly used emulators for this purpose. TEMU [77], built on top of QEMU, is one of the first mature security analysis projects to use whole system emulation. HookFinder [76] was built on top of TEMU to monitor for malicious hooks installed by rootkits in kernel memory. Panaroma [78], MAVMM [55], Lares [61], and Ether [20] are other foundational projects in this area. Besides direct emulation, there are also other areas of significant research aimed at allowing analysis and monitoring of malware outside of the environment the malware is executing in. Virtual machine introspection (VMI)

is a widely-use technology for this goal as it allows monitoring of guest virtual machines from the host. This has the benefit of the security monitor executing from a "safe" environment where the malware being observed would ideally not be able to attack it. Due to this advantage, there has been significant virtual machine introspection research in both academia [54, 25, 11, 34, 7, 6, 5, 8] and industry [60], including the libvmi project that allows running Volatility plugins against live virtual machine guests [59]. The use of malware sandboxes is also very popular and driven by virtual machine technology. Cuckoo Sandbox is the most widely used of these [17] and can produce detailed reports of system activity by malware.

While whole system emulation, virtual machine introspection, and sandboxes are mature technologies that are widely used for malware analysis, they do not fully meet the needs of real-world incident response teams nor do they fit in well within memory analysis-based workflows. To use these technologies during incident response, an analyst must first accomplish two tasks. The first is actually locating the malicious code in memory. As documented previously, this is a labor intensive task when using currently available Volatility plugins. Second, the analyst must then extract the module (DLL or EXE file) hosting the code. This presents a few problems in itself. For Volatility to automatically extract an executable module, it needs the metadata contained in the file's header. Since anti-virus engines and EDRs also process this metadata on live systems, malware will often zero out their modules' header after initialization. This forces the analyst to then perform a very manual process of rebuilding the PE header from scratch to make the file understandable by analysis tools, such as IDA Pro [30]. Furthermore, only in extremely rare circumstances can a file extracted from process memory be later executed on a different system. This occurs due to the substantial changes that occur during loading, including global variable initialization, selective section loading, and IAT patching [46]. This means that executables extracted from memory cannot be reliably executed in a virtual machine and the analyst must attempt to recover the module from filesystem of the infected system, assuming it is present.

Memory-only loading of DLLs, commonly referred to as reflective injection, is an extremely popular attack technique, and as the name implies, the loaded DLL is never written to disk at any point [66, 72]. This technique is generally accomplished by malware reading encrypted DLLs over the network or from encrypted stores within the main executable already in memory. The buffer containing the DLL is then decrypted and directly initialized by the malware's loader. The buffer is then usually zeroed out to prevent direct recovery. While Volatility can find and extract the sections of such DLLs, they will never be directly executable on any other system.

HookTracer solves all of these issues and makes powerful, automatic emulation of memory resident code accessible to even novice incident response team members. Instead of requiring the analyst to reverse engineer in-memory code, HookTracer can automatically determine if a hook is malicious, locate the hosting code, and extract it to disk. For situations where malicious code is not backed by a file on disk, HookTracer will determine the memory region hosting

the code and extract it. HookTracer also eliminates the need to attempt to make malicious code executable on a separate system to then leverage technologies such as whole system emulation. Instead, in-memory code can be directly emulated and detailed reports can be produced of the code's behaviour. This greatly streamlines memory analysis processing and allows full automation of the entire workflow. Besides HookTracer, there have been two recent projects that leverage unicorn in conjunction with Volatility. The first, ROPEMU [27, 26], automatically detects ROP chains [50] 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 [29] also hunts for ROP chains and was specifically developed to detect the "Gargoyle" attack [47] 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 the Windows API functions leveraged by the Gargoyle attack. Although neither of these projects overlap our efforts, we consider them to be important related work, as they both leverage *unicorn* in conjunction with Volatility and help showcase the growing realization by the memory forensics community that current incident response workflows are incompatible with traditional techniques and technology.

9. Conclusion

The rise of memory-only malware and attack payloads has led to significant research and development efforts in the field of memory forensics. The largest downside of these efforts has been the general inaccessibility of many of the techniques to all but expert investigators. This causes significant bottlenecks within the incident response workflow of organizations and leads to inconsistent analysis results that are heavily dependent on the skill level of the investigator. Our research efforts with HookTracer and *hooktracer_messagehooks* have bridged this gap in a key area of incident response - the detection and analysis of userland keyloggers. When keyloggers are active on a system, a wide range of data, including keystrokes, clipboard contents, and more, are vulnerable to recording and exfiltration. By leveraging *hooktracer_messagehooks*, even novice investigators can automatically determine the presence of keyloggers as well as generate detailed records of the keylogger's behaviour. These records can then used in automated detection and remediation at enterprise scale.

10. Acknowledgements

This work is supported by the National Science Foundation under Grant Number 1703683.

11. References

[1] 2016. Powershell Empire. `https://www.powershellempire.com`.

[2] 2017. The Volatility Framework: Volatile Memory Artifact Extraction Utility Framework. https://github.com/volatilityfoundation/volatility.

[3] 2018. Unicorn Showcase. http://www.unicorn-engine.org/showcase/.

[4] 2019. ZwQueryInformationFile. https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntifs/nf-ntifs-ntqueryinformationfile.

[5] Irfan Ahmed and Golden G. Richard III. 2013. Live Forensic Analysis of Kernel Code for Malware Detection in Cloud Computing Environments. *Proceedings of the 65th Annual Meeting of the American Academy of Forensic Sciences (AAFS)* (2013).

[6] Irfan Ahmed, Golden G. Richard III, Aleksandar Zoranic, and Vassil Roussev. 2013. Integrity Checking of Function Pointers in Kernel Pools via Virtual Machine Introspection. *Proceedings of the 16th Information Security Conference (ISC 2013)* (2013).

[7] Irfan Ahmed, Salman Javaid, Aleksandar Zoranic, and Golden G. Richard III. 2012. ModChecker: Kernel Module Integrity Checking in the Cloud Environment. *Proceedings of CloudSec 2012: The International Workshop on Security in Cloud Computing* (2012).

[8] Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden G. Richard III, and Vassil Roussev. 2013. IDTchecker: Rule-based Integrity Checking of Interrupt Descriptor Tables in Cloud Environments. *Proceedings of the 9th IFIP WG 11.9 International Conference on Digital Forensics* (2013).

[9] Fabrice Bellard. 2005. QEMU, A Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[10] Boldizsár Bencsáth and Gábor Pék and Levente Buttyán and Márk Félegyházi. 2011. Duqu: A Stuxnet-like Malware Found in the Wild. CrySyS Lab Technical Report 14.

[11] Brendan Dolan-Gavitt and Tim Leek and Michael Zhivich and Jonathon Giffin and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. 2011 IEEE Symposium on Security and Privacy. , 297–312 pages.

[12] Gianluca Brindisi. 2016. gozi-isfb. https://github.com/gbrindisi/malware/tree/master/windows/gozi-isfb.

[13] Andrew Case, Mohammad M. Jalalzai, Md Firoz-Ul-Amin, Ryan D. Maggio, Aisha Ali-Gombe, Mingxuan Sun, and Golden G. Richard III. 2019. HookTracer: A System for Automated and Accessible API Hooks Analysis. *Digital Forensics Research Conference (DFRWS)* (2019), 104–112.

[14] Catalin Cimpanu. 2018. Russia's Elite Hacking Unit Has Been Silent, But Busy. https://www.zdnet.com/article/russias-elite-hacking-unit-has-been-silent-but-busy/.

[15] The MITRE Corporation. 2018. Technique: Timestomp. https://attack.mitre.org/techniques/T1099/.

[16] The MITRE Corporation. 2018. Turla. https://attack.mitre.org/groups/G0010/.

[17] Cuckoo Foundation. 2014-2019. Cuckoo Sandbox Automated Malware Analysis. https://cuckoosandbox.org.

[18] Jessica DeCianno. 2014. IOC Security: Indicators of Attack vs. Indicators of Compromise. https://www.crowdstrike.com/blog/indicators-attack-vs-indicators-compromise/.

[19] Dell SecureWorks Counter Threat Unit Threat Intelligence. 2015. Skeleton Key Malware Analysis. https://www.secureworks.com/research/skeleton-key-malware-analysis.

[20] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 51–62.

[21] ESET. 2016. The rise of TeleBots: Analyzing disruptive KillDisk attacks. https://www.welivesecurity.com/2016/12/13/rise-telebots-analyzing-disruptive-killdisk-attacks/.

[22] FireEye. 2014. Poison Ivy: Assessing Damage and Extracting Intelligence. https://www.fireeye.com/content/dam/fireeye-www/global/en/current-threats/pdfs/rpt-poison-ivy.pdf.

[23] FireEye. 2016. FakeNet-NG-Next Generation Dynamic Network Analysis Tool. https://github.com/fireeye/flare-fakenet-ng.

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

[25] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. NDSS.

[26] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. 2016. ROPMEMU. https://github.com/Cisco-Talos/ROPMEMU/.

[27] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. 2016. ROPMEMU: A Framework for the Analysis of Complex Code-reuse Attacks. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, 47–58.

[28] Nikolay Grebennikov. 2011. Keyloggers: Implementing Keyloggers in Windows. Part Two. https://securelist.com/keyloggers-implementing-keyloggers-in-windows-part-two/36358.

[29] Aliz Hammond. 2018. Hunting for Gargoyle Memory Scanning Evasion. https://countercept.com/blog/hunting-for-gargoyle/.

[30] Hex-Rays. 2018. Hex-Rays Home. https://www.hex-rays.com.

[31] Andrew Honig, Mike Sikorski, John Laliberte, and Niles Akens. 2012. FakeNet. https://practicalmalwareanalysis.com/fakenet/.

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

[33] IOActive. 2012. *Reversal and Analysis of Zeus and SpyEye Banking Trojans.* Technical Report.

[34] Salman Javaid, Aleksandar Zoranic, Irfan Ahmed, and Golden G. Richard III. 2012. Atomizer: A Fast, Scalable and Lightweight Heap Analyzer for Virtual Machines in a Cloud Environment. *Proceedings of the 6th Layered Assurance Workshop (LAW'12)* (2012).

[35] Kaspersky. 2014. The Epic Turla Operation: Solving some of the mysteries of Snake/Uroboros. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08080105/KL_Epic_Turla_Technical_Appendix_20140806.pdf.

[36] Takashi Katsuki. 2013. Crisis: The Advanced Malware. *2013 Symantec Internet Security Threat Report.*

[37] Emmett Koen. 2017. Indicators of Compromise and Where to Find Them. https://blogs.cisco.com/security/indicators-of-compromise-and-where-to-find-them.

[38] Brian Krebs. 2013. Carberp Code Leak Stokes Copycat Fears. https://krebsonsecurity.com/2013/06/carberp-code-leak-stokes-copycat-fears/.

[39] Kaspersky Lab. 2014. Kaspersky Lab Uncovers "The Mask": One of the Most Advanced Global Cyber-espionage Operations to Date Due to the Complexity of the Toolset Used by the Attackers. https://usa.kaspersky.com/about/press-releases/2014.

[40] Kaspersky Lab. 2019. The Epic Turla (Snake/Uroburos) Attacks. https://usa.kaspersky.com/resource-center/threats/epic-turla-snake-malware-attacks.

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

[42] Andrea Lelli. 2018. Out of Sight But Not Invisible: Defeating Fileless Malware with Behavior Monitoring, AMSI, and Next-gen AV. https://cloudblogs.microsoft.com/microsoftsecure/2018/09/27/out-of-sight-but-not-invisible-defeating-fileless-malware-with-behavior-monitoring-amsi-and-next-gen-av/.

[43] Michael Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. 2010. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. Wiley Publishing.

[44] Michael Hale Ligh. 2012. MoVP 3.1 Detecting Malware Hooks in the Windows GUI Subsystem. https://volatility-labs.blogspot.com/2012/09/movp-31-detecting-malware-hooks-in.html.

[45] Michael Hale Ligh. 2012. Reverse Engineering Poison Ivy's Injected Code Fragments. https://volatility-labs.blogspot.com/2012/10/reverse-engineering-poison-ivys.html.

[46] Michael Hale Ligh, Andrew Case, Jamie Levy, and AAron Walters. 2014. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley, New York.

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

[48] MalBot. 2018. Post 0x17.1: Analyzing Turla's Keylogger . https://malware.news/t/post-0x17-1-analyzing-turla-s-keylogger/22762.

[49] MalBot. 2018. Post 0x17.2: Analyzing Turla's Keylogger. https://malware.news/t/post-0x17-2-analyzing-turla-s-keylogger/23334.

[50] David Maloney. 2016. Return Oriented Programming (ROP) Exploits Explained. https://www.rapid7.com/resources/rop-exploit-explained/.

[51] Malpedia. 2019. Loki Password Stealer (PWS). https://malpedia.caad.fkie.fraunhofer.de/details/win.lokipws.

[52] Mike McQuade. 2018. The Untold Story of NotPetya, the Most Devastating Cyberattack in History. https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.

[53] Microsoft. 2018. SetWindowsHookExA function. https://docs.microsoft.com/en-us/windows/desktop/api/winuser/nf-winuser-setwindowshookexa.

[54] Matthew Muscat and Mark Vella. 2018. Enhancing Virtual Machine Introspection-Based Memory Analysis with Event Triggers. 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom).

[55] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. 2009. Mavmm: Lightweight and Purpose Built VMM for Malware Analysis. 2009 Annual Computer Security Applications Conference. 441–450.

[56] Martin Novak, Jonathan Grier, and Daniel Gonzales. 2018. New Approaches to Digital Evidence Acquisition and Analysis. https://www.nij.gov/journals/280/pages/new-approaches-to-digital-evidence-acquisition-and-analysis.aspx.

[57] nyx0. 2015. Carberp Banking Trojan. https://github.com/nyx0/Carberp.

[58] Rob Pantazopoulos. 2017. Loki-Bot: Information Stealer, Keylogger, & More! https://digital-forensics.sans.org/community/papers/grem/loki-bot-information-stealer-keylogger-more_4802.

[59] Bryan Payne, Steven Maresca, Tamas K. Lengyel, and Antony Saba. 2019. LibVMI Virtual Machine Introspection. http://libvmi.com/.

[60] Bryan D Payne. 2012. Simplifying Virtual Machine Introspection Using libvmi. *Sandia Report* (2012), 43–44.

[61] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An Architecture for Secure Active Monitoring Using Virtualization. IEEE Symposium on Security and Privacy. 233–247.

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

[63] Rapid7. 2019. Metasploit. https://www.metasploit.com/.

[64] SecureWorks. 2007. Gozi Trojan. https://www.secureworks.com/research/gozi.

[65] SentinelOne. 2019. GozNym Banking Malware: Gang Busted, But Is That The End? https://www.sentinelone.com/blog/goznym-banking-malware-gang-busted/.

[66] Skape and Jarkko Turkulainen. 2004. Remote Library Injection. http://www.hick.org/code/skape/papers/remote-library-injection.pdf.

[67] Sophos. 2019. Gozi V3: Tracked by Their Own Stealth. https://news.sophos.com/en-us/2019/12/24/gozi-v3-tracked-by-their-own-stealth/.

[68] Tanium. 2017. Tanium IOC Detect UserGuide. https://docs.tanium.com/.

[69] ASERT Team. 2018. Donot Team Leverages New Modular Malware Framework in South Asia. https://de.netscout.com/blog/asert/donot-team-leverages-new-modular-malware-framework-south-asia.

[70] FortiGuard SE Team. 2019. Newly Discovered Infostealer Attack Uses LokiBot. https://www.fortinet.com/blog/threat-research/new-infostealer-attack-uses-lokibot.html.

[71] Windows Defender Advanced Threat Hunting Team. 2016. Platinum: Targeted Attacks in South and Southeast Asia. https://www.microsoft.com/en-us/download/details.aspx?id=51956.

[72] Volexity. 2016. PowerDuke: Widespread Post-Election Spear Phishing Campaigns Targeting Think Tanks and NGOs. https://www.volexity.com/blog/2016/11/09/powerduke-post-election-spear-phishing-campaigns-targeting-think-tanks-and-ngos/.

[73] Kiel Wadner. 2014. An Analysis of Meterpreter During Post-Exploitation. SANS Institue Information Security Reading Room.

[74] George Waller. 2012. Keyloggers: The Most Dangerous Security Risk in Your Enterprise. https://esj.com/articles/2012/11/12/keylogger-security-risk.aspx.

[75] Forensics Wiki. 2012. List of Volatility Plugins. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference.

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

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

[78] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, 116–127.