New Memory Forensics Techniques to Defeat Device Monitoring Malware

Andrew Case, Gustavo Moreira, Austin Sellers,
Golden G. Richard III

andrew@dfir.org, gmoreira@volexity.com, asellers@volexity.com,
golden@cct.lsu.edu

April 4, 2022

1 Introduction

Malware that is capable of monitoring hardware devices poses a significant threat to the privacy and security of users and organizations. Common capabilities of such malware include keystroke logging, clipboard monitoring, sampling of microphone audio, and recording of web camera footage. All modern operating systems implement APIs that provide this hardware access and all of them have been abused by numerous malware samples to monitor the activity of journalists and dissidents, conduct espionage operations against corporate and government targets, and gather data that allows for blackmail of individuals.

Existing methods for detecting these malware techniques are largely confined to malware that operates within kernel space, commonly referred to as kernel rootkits. The use of such rootkits has waned in recent years as operating system vendors have sharply locked down access to kernel memory. This includes enforcement of driver signing, limiting which organizations can receive signing certificates, and adding proactive monitoring systems within the operating system that detect common rootkit tampering techniques. These limitations placed upon kernel rootkits as well as the easy-to-use APIs in userland that allow for access to hardware devices has led to a significant number of device monitoring malware samples that operate solely within process memory. Such malware is comparatively much simpler to write and is also much easier to make portable across a wide variety of operating system versions.

Unfortunately, current methods for direct detection of such userland malware are severely outdated or completely lacking. These include attempts at live forensics, which relies on system APIs to enumerate artifacts, but these APIs are often hooked by malware to hide their activity. Partial existing memory forensics techniques for Windows exist, but are outdated, and there are monitoring methods used across the major operating systems that have no memory forensic

detection support at all. Given the significant recent emphasis on memory analysis during incident response, such as in CISA directives released after the detection of ProxyLogon [1] as well the SolarWindows supply chain compromise [2], it is imperative that memory forensic techniques are able to properly detect modern threats.

In this paper, we present our effort to research and develop memory forensic algorithms capable of direct detection of userland device monitoring malware across all three major operating systems. To accomplish this goal, we first undertook a significant effort to understand and document the APIs that provide device access to userland components. This effort included binary analysis of closed source components (Windows, Mac) as well as study of open source components (Linux, Mac). These efforts in turn led to the update of existing and creation of new Volatility plugins that are capable of automatically locating and extracting all relevant information about processes that are monitoring hardware devices. These plugins quickly inform analysts of the presence of such malware as well as key in information such as the addresses of callback functions. We plan to contribute our Volatility plugins and additions to the community upon publication of this paper.

2 Research and Experimental Setup

2.1 Operating Systems and Versions Tested

During our research effort, we aimed to develop capabilities that covered all supported versions of target operating systems as well as bleeding edge ones, where possible. Given that the layout of operating system data structures often changes between versions and that many of the components we analyzed were closed source, we fully documented all of our developed plugins with the steps needed to find the correct offsets and associated information in future versions of each operating system.

The following table lists the starting and ending version tested and supported added for each operating system:

Operating System	**Earliest Version**	**Latest Version**
Windows 10	10563	22000.556
macOS	Catalina	Monterey
Linux	2.6.18	5.14

This wide range of versions covers Windows 10 starting from build 10586, released in 2015, to the latest release at the time this paper is written. We also cover Linux kernel versions going back to 2008. All versions of macOS supported by Apple at the time of writing were also covered in our research. We choose to test and include such as a wide of range of kernel versions to ensure that our effort is as widely useful to the community as possible.

2.2 Memory Sample Creation

When developing new memory forensic capabilities, it is imperative to develop proof-of-concept applications that perform the same actions as malware, but in a controlled and logged manner. By developing these POC applications, the researcher can be certain that artifacts recovered from a memory sample with a POC application active match precisely with the values recorded by the POC as it ran. As an example, a POC that places an API hook would record the process ID of its victim along with the address that was hooked and the address of the malicious handler. The memory analysis researcher can then run newly developed plugins against these memory samples and confirm that the recovered values match. These samples can also be saved to perform regression testing of future software releases.

Given the severity of the malware types discussed in this paper, our team wanted to ensure that our research process and results could be repeated and verified well into the future. To meet this need, we developed POC applications that performed each action hunted for by our plugin set. When then generated memory samples with our developed POCs active and ensured that our plugin output matched. Snippets of these POCs will be shown and discussed where relevant throughout the paper.

To create stable and valid memory samples, two methods for acquisition were used. The first was the use of Surge Collect Pro from Volexity [3]. This commercial software supports stable acquisition across Windows, Linux, and macOS systems. Besides capturing physical memory, Surge also records a significant amount of system state and metadata to json files. This extra metadata allowed us to automate a significant amount of the testing. The second approach we used for acquisition was snapshotting and suspending the VMware virtual machines that we used for testing. The system state files (.vmem, .vmss, .vmsn) created when snapshotting and/or suspending a guest VM contain a copy of all physical memory as well the metadata needed to fully perform memory analysis. The main downside to this approach is that the system state files do not contain the wide range of metadata that Surge produces.

2.3 Analysis Tools and Resources

IDA Pro was used for all binary analysis performed during our research. Source code studying of Linux kernel versions was largely performed using the excellent Exlixir cross reference website [4] and macOS source code study was performed using a self-hosted OpenGrok instance that contained source code from Apple's open source code website [5].

3 Windows Analysis - SetWindowsHookEx

There are two APIs provided by Windows systems that userland malware abuses to monitor devices. In this section, we discuss the internals of *SetWindowsHookEx* along with our updates and creation of new Volatility capabilities that

provide automated detection of abuse of this API. Before our effort, Volatility only supported detection of *SetWindowsHookEx* abuse through Windows 7 and did not recover all needed information. In the following section, we give *RegisterRawInputDevices* the same, complete treatment.

3.1 Background

The *SetWindowsHookEx* API provides the ability for applications to install *hooks* that activate when specific device or window (GUI) events occur. These hooks can target the keyboard or mouse as well as activity within the GUI environment, such as messages between sent between applications or an application changing its foreground/background state. The callbacks associated with these hooks receive the specific data of event that triggered the hook, such as the button pressed on a keyboard. Given the power and flexibility of this API, numerous malware variants and samples have abused this API for keylogging, mouse monitoring, and code injection.

Figure 1 shows the function prototype for *SetWindowsHookEx*.

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

Figure 1: SetWindowsHookEx Prototype

The first parameter, *idHook*, specifies which event the hook will monitor, such as *WH_KEYBOARD*, to monitor keystrokes. The second parameter, *lpfn* is the callback to activated upon each monitor event. The last two parameters, *hmod* and *dwThreadId*, control the behaviour of which processes are hooked and how they are hooked.

If *hmod* is NULL then the executable calling *SetWindowsHookEx* must host the callback function. Otherwise, *hmod* must reference a valid handle to the DLL that hosts the callback function. If *dwThreadID* is non-NULL then it specifies the particular thread for which the monitor should be placed. If NULL is passed, then all threads within the same desktop as the calling application will be hooked. As discussed shortly, Volatility does not currently cover all combinations of *hmod* and *dwThreadId*, which we discovered during our testing and fixed during development.

3.2 Internals

An entry on the Volatility Labs blog [6] and the Art of Memory Forensics [7] discuss a majority of the internals related to *SetWindowsHookEx* and the kernel data structures that it populates. In particular, these cover the data structures created when hooks are populated as well as the recovery of global hooks placed inside a DLL. We highly suggest reviewing these resources for readers new to analysis of the Windows GUI subsystem.

The information specific to each hook is placed within a *tagHOOK* data structure. This information includes the following:

1. The desktop where the hook is active
2. The thread targeted by the hook
3. The event monitored
4. The location of the callback function
5. The module hosting the callback function

To support the variety of data structures needed for this analysis across many Windows 10 versions, a significant reverse engineering effort was undertaken. To allow for quick support of future Windows versions to be added to our plugins, we fully documented each function inside of win32k.sys, win32kbase.sys, and win32kfull.sys needed to uncover the variety of data structure and offsets.

Besides having the correct data structure layouts, fully recovering all variations of hook placement requires treating each hook in one of three ways, each of which alters the meaning of the data stored in *tagHOOK*. These three will be discussed separately as having complete support in Volatility requires special handling of each one.

3.3 Global Hooks in a DLL

The first form of *SetWindowsHookEx* abuse is when *hmod* is set to a DLL handle and *dwThreadId* is set to NULL. This tells the operating system that the caller wants to monitor all threads within the same desktop using a function defined in the DLL. The effect of this choice is that the DLL hosting the callback will be loaded (injected) into each process that triggers the callback, such as after a keystroke is entered.

This mechanism provides a built-in code injection technique that removes the need for malware to use heavily monitored APIs, such as *WriteProcessMemory*, to introduce code into a victim process. Several malware samples, such as the Laqma malware discussed in [6], abuse *SetWindowsHookEx* solely as a mechanism to get their DLL loaded into victim processes, but then discard all future event data (keystrokes, mouse movements, etc.).

When a global hook in a DLL is used, the *ihmod* member of *tagHOOK* is set to the index within the global atom table of the element holding the path to the DLL hosting the hook, and the *offPfn* member is set to the relative offset of the callback function from the DLL base address.

3.3.1 Volatility Support

Volatility's *messagehooks* plugin recovers global hooks in a DLL by first enumerating each desktop along with each thread running inside each desktop. For each desktop or thread found, it uses the *pDeskInfo* member of the structure to find the *tagDESKTOPINFO* instance. This structure contains a member named *aphkStart*, which holds an array of pointers to *tagHOOK* instances. Each array index corresponds to the a hardcoded hook type, and each *tagHOOK* instance holds a pointer to the next one in its list. For each hook found, its relevant metadata is computed and reported. Figure 2 shows the output of the current version of the plugin against a single hook found in a memory sample. In this particular instance, the system from which memory was acquired was infected with the infamous Turla malware [8].

```
Offset(V)  : 0xfffff900c122f9f0
Session    : 2
Desktop    : WinSta0\Default
Thread     : 1816 (explorer.exe 2568)
Filter     : WH_KEYBOARD_LL
Flags      : HF_GLOBAL
Procedure  : 0x22c0
ihmod      : 1
Module     : C:\Users\JohnSmith\Desktop\tll.dll
```

Figure 2: Recovery of a global keyboard hook

The figure illustrates that Volatility has detected that the explore.exe process with a PID of 2568 has its thread with ID 1816 hooked. The event being monitored (*Filter*) is the keyboard, and the callback function (*Procedure*) is located at offset 0x22c0 from the beginning of tll.dll. Existing Volatility plugins, such as dlldump or dumpfiles, can be used to extract the malicious DLL to disk and begin binary analysis of the callback function.

3.4 Global Hooks in an Application Executable

The second form of *SetWindowsHookEx* abuse is the placement of a hook inside of an application executable (.exe file). In this form, the *hmod* parameter to *SetWindowsHookEx* is set to NULL and *dwThreadId* can either be NULL or the thread ID to target. The effect of setting *hmod* to NULL is quite significant as it means the calling executable will still have its callback activated for monitored events, but the executable itself will *not* be injected into other processes. Instead, as described by Microsoft [9], the created event information will be sent in a message to the calling process.

Figure 3 shows how our POC set a global application executable hook.

```
keyboard_hook = SetWindowsHookExA(
    WH_KEYBOARD_LL,
    keyboard_hook_procedure,
    GetModuleHandle(NULL),
    NULL
);
```

Figure 3: SetWindowsHookEx with an Application Hook

In the call it can be seen that the both the module and thread ID parameters are NULL. This means every thread within the desktop will be hooked and that the per-event data will be placed into the message input queue of the calling application. This allowed us to ensure Volatility correctly handles this case going forward.

3.4.1 Volatility Support

Volatility's existing *messagehooks* plugin was able to detect hooks registered in this way, but did not inform analysts of which process was hosting the hook. This meant that, in a real-world situation where dozens of processes are running on a victim desktop, the same hook will be reported for each of them, but without any indication of which process actually placed it. The output is also a bit confusing as it reports a callback address that is not even mapped into all of the processes, or if it is mapped, would have a different meaning in each process address space. This lack of information forces analysts to manually work backwards to trace the hook origin, which is time consuming, error prone, and requires an investigator to have previous reverse engineering experience.

We aimed to resolve this issue by researching a method to detect which application (process) actually placed the hook. Since the application is not directly mapped into each victim process, the hooking application must remain running for as long as the hook is active. With this in mind, we analyzed the internal implementation of *SetWindowsHookEx* to see how such processes are treated. We then determined that threads which set global hooks will have the *TIF_GLOBALHOOKER* flag set in the *TIF_FLAGS* member of the thread's *tagTHREADINFO* structure. We then updated *messagehooks* to print True or False at the end of each Thread line to indicate if the particular thread has set a global hook.

Figure 4 shows three blocks of output from *messagehooks* when ran against the sample with our POC active.

```
Offset(V)  : 0xfffff90146002870
Session    : 1
Desktop    : WinSta0\Vol_GUI-DesktopHidden
Thread     : <any>
Filter     : WH_KEYBOARD_LL
Flags      : HF_ANSI|HF_GLOBAL
Procedure  : 0x7ff782002300
ihmod      : -1
Module     : (Current Module)

Offset(V)  : 0xfffff90146002870
Session    : 1
Desktop    : WinSta0\Vol_GUI-DesktopHidden
Thread     : 5920 (powershell.exe 5916) False
Filter     : WH_KEYBOARD_LL
Flags      : HF_ANSI|HF_GLOBAL
Procedure  : 0x7ff782002300
ihmod      : -1
Module     : (Current Module)

Offset(V)  : 0xfffff90146002870
Session    : 1
Desktop    : WinSta0\Vol_GUI-DesktopHidden
Thread     : 400 (GUITesterAll.e 1332) True
Filter     : WH_KEYBOARD_LL
Flags      : HF_ANSI|HF_GLOBAL
Procedure  : 0x7ff782002300
ihmod      : -1
Module     : (Current Module)
```

Figure 4: SetWindowsHookEx with an Application Hook

The first block shows the global hook registered inside the *Vol_GUI-Desktop Hidden* desktop, which is created by our POC. This hook is denoted as the global one by marking the thread as $< any >$ to indicate that this hook will apply to all threads within the particular desktop. Also note that the procedure address is the full runtime address of the callback function for the keyboard hook, but no information is given here on which process is hosting the hook as the module index is -1, which marks the index as invalid. This also leads Volatility to mark the hosting module as $(CurrentModule)$, since it cannot automatically infer the

hosting module from the *tagHOOK* structure.

The second block, which is repeated for each victim thread in the full plugin output, shows that the same hook has targeted the powershell.exe process with PID 5916. This output can be automatically matched to the first block as the hook address is the same since the per-thread's hook structure is pointed to the same one as the globally registered hook attached to the desktop. We also note that the *False* at the end of the Thread line is our addition that prints if the *TIF_GLOBALHOOKER* flag is set. This per-thread output is how analysts can know which processes and threads were infected by malicious hooks, and, as mentioned previously, there are typically dozens of these lines when hooks are active on real-world systems.

The third and final block of output in the figure shows that the *GUITesterAll* program (our POC) is the one that actually placed the hook as its *TIF_GLOBALHOOKER* flag is set (the *True* at the end of the Thread line). With this new information, an analyst immediately knows 1) which process actually placed the hook 2) which address to look at in process memory for the hook. Furthermore, since the main application executable is hosting the hook, the analyst can use the existing procdump plugin's -p option with the process ID (1332) to automatically extract this malicious executable to disk. This allows analysis in static analysis tools, such as IDA Pro or Ghidra. As we just demonstrated, our addition of analysis of the per-thread *TIF_FLAGS* has transformed an error-prone, manual process into a more automated solution.

3.5 Thread-Specific Hooks

The last type of hook covered in this section are thread-specific hooks. These are created when the *dwThreadId* is set to a specific thread ID instead of NULL to target all threads. To create one of these, we added the code shown in Figure 5 to our POC application.

```
mouse_hook = SetWindowsHookExA(
      WH_MOUSE,
      mouse_hook_procedure,
      dll_handle,
      process_info.dwThreadId
);
```

Figure 5: SetWindowsHookEx with a Thread-Specific Hook

In the code it can be seen that the mouse monitoring hook is registered against a particular process, which in our POC is a notepad.exe process spawned previously by the POC. The module parameter is a handle to the DLL implementing *mouse_hook_procedure*. Since this hook is process specific, Windows will

load the referenced DLL into the victim process.

3.5.1 Volatility Support

Our testing showed that the existing Volatility *messagehooks* plugin had no support for thread-specific hooks, so we needed to research why these were missed. Analysis of the kernel functions that activate hooks showed that thread-specific hooks were placed in a *aphkStart* array inside the *tagTHREADINFO* structure, and not in the one referenced from the *tagDESKTOPINFO* structure. Based on this discovery, we updated the *messagehooks* plugin to enumerate both *aphkStart* arrays and then our previous keyboard hook was enumerated along with the addition of our thread-specific mouse hook. Figure 6 shows the output of our updated *messagehooks* plugin against a memory sample with our mouse hook active. As can be seen, the plugin is now capable of recovering thread-specific hooks.

```
Offset(V)  : 0xfffff90146007630
Session    : 1
Desktop    : WinSta0\Vol_GUI-DesktopHidden
Thread     : 4192 (notepad.exe 2688) False
Filter     : WH_MOUSE
Flags      : HF_ANSI
Procedure  : 0x10f0
ihmod      : 7
Module     : 0x7L
```

Figure 6: *messagehooks* Recovering our Thread-Specific Hook

We then proceeded to test this new support across Windows versions and noticed that the DLL path was not recovered for a single memory sample. We then manually ran Volatility's *atomscan* plugin across the memory samples, which scans Windows atom tables and outputs the individual atoms. We determined that the DLL path for the hooking DLL was not present in the atom table for any of the samples. This indicated that something might be wrong with the plugin, as we expected the *ihmod* value of *tagHOOK* to reference a global atom containing the path of the hooking DLL as it does for the global hooks described previously.

This led us to re-examine the internal implementation of *SetWindowsHookEx*, and we determined that thread-specific hooks do not populate the global atom table at all. Instead, the *ihmod* value of thread-specific hooks reference an index into the *ahmodLibLoaded* array stored inside the *tagPROCESSINFO* structure referenced from the hooked thread. This array holds the base addresses of DLLs associated with the process. With this information, we were able to update

messagehooks to conditionally retrieve the DLL path based on whether the hook being global or local. With this change, Volatility can now retrieve the DLL path, if present, for all hook types and variations. Figure 7 shows the output of our fully updated plugin that is aware of both DLL path sources.

```
Offset(V)  : 0xfffff90146007630
Session    : 1
Desktop    : WinSta0\Vol_GUI-DesktopHidden
Thread     : 4192 (notepad.exe 2688) False
Filter     : WH_MOUSE
Flags      : HF_ANSI
Procedure  : 0x10f0
ihmod      : 7
Module     : C:\Users\Administrator\Desktop\hook.dll
```

Figure 7: Properly Recovering the DLL Path

As illustrated, the full path is retrieved for the DLL hosting the mouse hook as opposed to simply printing out an *ihmod* value. By extracting the DLL to disk, the investigator can then begin analysis of the hook procedure beginning at offset *0x10f0* of the DLL.

With all of the updates described in this section, the *messagehooks* plugin of Volatility is now able to successfully recover complete information of all message hook variations through the latest version of Windows 10.

4 Windows Analysis - RegisterRawInputDevices

4.1 Background

Abuse of the *RegisterRawInputDevices* API is the second popular method that Windows userland malware employs to monitor device activity. Many samples used in high profile attacks and by APT groups have abused this feature, including PlugX variants, the Dexter Point-of-Sale malware, HawkEye, FIN7, and APT27 [10, 11, 12, 13, 14, 15]. An article on Code Project by Mike Mee provides a very accessible read on how these hooks are registered by a programmer [16]. We strongly suggest reading this article before proceeding if you are new to this type of keylogger.

The normal method for abusing this API is to first register an invisible window that will be used to attach the malware's callback function. This is performed by registering a custom class through the use of the *RegisterClass* API. Once registered, the class can then be used to create the hidden window. Figure 8 shows how this is performed in our POC application.

```
WNDCLASS wc = { 0 };

wc.lpfnWndProc = WndProc;
wc.hInstance = hInstance;
wc.lpszClassName = L"Vol_GUI-kl";

RegisterClass(&wc);

WriteOutputFile(FormatStringOutput("WndClassName", "Vol_GUI-kl"));
WriteOutputFile(FuncAddrFunc("WndProc", &WndProc));

hWnd = CreateWindow(wc.lpszClassName, "My Hidden Window", ...);
```

Figure 8: *RegisterRawInputDevices* Prototype

As shown, we define a callback function named *WndProc* that receives GUI messages and we named our custom class *Vol_GUI-kl*. Next, we write information to the output file for future verification and then we create our hidden window. The first parameter to *CreateWindow* must be the name a registered class, which we set to our previously registered instance. We then give our window a pre-defined name so that we can verify it during future plugins runs.

To receive all input events, the callback function must properly handle *WM_CREATE* and *WM_INPUT* messages. *WM_CREATE* is sent upon creation of a window of the given class, and this is when a keylogger must use *RegisterRawInputDevices* to receive future *WM_INPUT* messages that contain each keystroke. Figure 9 shows the prototype for *RegisterRawInputDevices* and Figure 10 displays the input structure (*tagRAWINPUTDEVICE*).

```
BOOL RegisterRawInputDevices(
  [in] PCRAWINPUTDEVICE pRawInputDevices,
  [in] UINT             uiNumDevices,
  [in] UINT             cbSize
);
```

Figure 9: *RegisterRawInputDevices* Prototype

```
typedef struct tagRAWINPUTDEVICE {
  USHORT usUsagePage;
  USHORT usUsage;
  DWORD  dwFlags;
  HWND   hwndTarget;
} RAWINPUTDEVICE, *PRAWINPUTDEVICE, *LPRAWINPUTDEVICE;
```

Figure 10: *tagRAWINPUTDEVICE* Definition

The first parameter, *pRawInputDevices*, specifies one or more *tagRAWINPUTDEVICE* structures that describe the devices to be monitored. The second parameter specifies how many devices are described and the last parameter specifies how large each input structure is.

To properly monitor the keyboard or other desired device, the *usUsagePage* and *usUsage* of the *tagRAWINPUTDEVICE* structure must be set correctly. For malware that targets common devices (keyboard, mouse), the *usUsagePage* must be set to *HID_USAGE_PAGE_GENERIC*, which has a constant value of 1. Figure 11 shows the possible options for the *usUsage* member.

Usage ID	Usage Name	*hidusage.h* constant
0x01	Pointer	HID_USAGE_GENERIC_POINTER
0x02	Mouse	HID_USAGE_GENERIC_MOUSE
0x04	Joystick	HID_USAGE_GENERIC_JOYSTICK
0x05	Game Pad	HID_USAGE_GENERIC_GAMEPAD
0x06	Keyboard	HID_USAGE_GENERIC_KEYBOARD
0x07	Keypad	HID_USAGE_GENERIC_KEYPAD
0x08	Multi-axis Controller	HID_USAGE_GENERIC_MULTI_AXIS_CONTROLLER

Figure 11: *messagehooks* Recovering our Thread-Specific Hook

For malware that wants to perform keylogging, it will set this value to *HID_USAGE_GENERIC_KEYBOARD*, which has a constant value of 6.

Once registered, the window callback procedure will then receive *WM_INPUT* messages after each keystroke, and the *GetRawInputData* API can be used to

translate these messages to the keys typed on the monitored keyboard.

4.2 Internals

A blog post on the "Eye of Ra" blog provides a good overview of the data structures that Windows 7 uses to track raw input device monitors [17]. This includes a high level view of determining which processes called *RegisterRawInputDevices* and most of the data structures involved. We used this blog post as a starting point for our research, but given that it only targeted Windows 7 and many of the functions and implementation details were modified in Windows 10, there was still a substantial amount of work to do. We also aimed to determine not only which process(es) were monitoring input devices, but also the addresses of the input handlers. Finally, this blog post is able to use WinDBG to parse the types of the GUI subsystem as Microsoft released full type information for several versions of Windows 7. Unfortunately, this was stopped and later versions of Windows 7 and all versions of Windows 10 must be reverse engineered to discover type and algorithm changes inside the GUI subsystem. This meant we needed to perform a significant reverse engineering effort to understand the modern versions of the handling code.

The use of *RegisterRawInputDevices* leads to a *tagPROCESS_HID_TABLE* instance being tracked from the process' *tagPROCESSINFO* structure. This hid table structure then tracks the monitors associated with each process in *tagPROCESS_HID_REQUEST* structures that contain the *usUsagePage* and *usUsage* values described previously as well a handle to the window that the hook is associated with. Inside of each window, which is tracked by a *tagWND* structure, there is a member named *lpfnWndProc* that holds the address of the window callback procedure, if any. This per-monitor request data is what we used to build our new Volatility plugin that is capable of uncovering *RegisterRawInputDevices* abuse.

4.3 Adding Volatility Support

Before our research effort, Volatility had no plugin to report instances of *RegisterRawInputDevices* usage. We addressed this issue through the development of a new *rawinputdevicemonitors* plugin. Our previous work to make *messagehooks* function properly across Windows 10 versions meant that we had a head start in this effort. In particular, *messagehooks* already enumerates *tagPROCESSINFO* instances, which is what we need to start our analysis of input device monitors.

For each *tagPROCESSINFO* found, we check if its HID table pointer is set. If so, that means the process has registered at least one monitor. We then proceed from the *tagPROCESS_HID_TABLE* to the list of valid *tagPROCESS_HID_REQUEST* structures. This allows us to determine which requests are monitoring devices of interest (keyboard and mouse) and to report the process ID and name, window name, and window procedure callback address.

Figure 12 shows the output block out our new plugin when run against a memory sample with our POC input device monitor active.

```
Offset (V)       : 0xfffff90141c5b160
Session          : 1
Desktop          : WinSta0\Vol_GUI-DesktopHidden
Process          : GUITesterAll.e 3812
Window Name      : My Hidden Window
Window Procedure : 0x7ff6e3b63220
Monitor usUsage  : 6
```

Figure 12: *rawinputdevicemonitors* Recovering our POC

As shown, the monitoring application (GUITesterAll) is reported along with the name and address of the window procedure. The usUsage value is reported as 6, which corresponds to keyboarding monitoring. With this information available, an investigator can immediately begin static analysis of the malicious application to determine which action(s) it takes upon each keystroke.

5 Linux Analysis - strace and ptrace

After finishing our research on Windows, we turned our attention to Linux. Linux has three main methods for userland keylogging: 1) strace (debugging APIs) 2) Input Events 3) TIOCSTI. We will now cover these in order.

5.1 Background

strace is a built-in Linux utility that leverages the Linux debug API (*ptrace*) to record system calls made by applications. strace is extremely popular with system administrators and malware analysts as it not only records the names of system calls made but also the parameters, such as file names, network addresses, and process IDs. The power afforded by strace and the *ptrace* system call that it relies on has resulted in widespread abuse by malware and attackers to spy on victim processes. Many resources show how it trivial it is to spy on victim users with strace, including the theft of SSH credentials and recording of commands (e.g., [18]).

The abuse of these debugging facilities sometimes leads systems administrators to lock down ptrace, even for root users [19, 20]. As with most security controls, however, these settings are not enabled by default on common distributions and are not universally applied to production servers. Given the threat posed by these interfaces, we investigated the debugging API internals with a goal of detecting processes being debugged.

5.2 Internals

Each Linux process is tracked by a *task_struct* structure that serves the same purpose as *_EPROCESS* on Windows. Inside each *task_struct* is a member

named *ptraced* that holds a linked list of all processes that the examined process is tracing. This allows us to directly enumerate all processes being traced.

ptrace supports a variety of options, such as *PTRACE_O_TRACEFORK* and *PTRACE_O_TRACECLONE*, that allow tracing programs to automatically trace (debug) child processes spawned by the original. strace supports this capability by following children processes if the *-f* option is specified in the command line invocation. To determine all instances of programs being debugged and to discover the process debugging them, we must compare the *parent* member of *task_struct* to the *real_parent* member. These will not be the same when a process is being debugged by a process other than its direct parent, such as a debugger, and we can use this discrepancy to determine the real tracing process even if it is several parents up the chain.

5.3 Volatility Support

Before our research, Volatility had no existing plugin to report processes that were being debugged. To remedy this, we developed the *linux_process_ptrace* plugin, which reports on all processes being debugged, the process IDs of tracing processes, and the tracing state of each process. We have two sets of figures that demonstrate this plugin.

5.3.1 Detecting gdb Usage

The first, Figure 13, shows us loading the *cat* executable into gdb followed by executing it inside the debugger.

```
# gdb -q /bin/cat
Reading symbols from /bin/cat...(no debugging symbols found)...done.
(gdb) r
Starting program: /usr/bin/cat

# ps aux | grep cat
root        778  2.0  9.7  59108 45688 pts/0    T    09:49   0:00 gdb /bin/cat
root        780  0.0  0.1   5400   828 pts/0    t    09:49   0:00 /usr/bin/cat
```

Figure 13: Debugging the *cat* command with gdb

We then ran the ps command to determine the process IDs of the created *cat* process (780) as well as the PID of the controlling gdb process (778). Figure 14 shows the output of *linux_process_ptrace* when run against the memory sample with these processes active.

```
$ python vol.py ... linux_process_ptrace
Volatility Foundation Volatility Framework 2.6
Name  Pid  PPid  Flags   Traced by Tracing
----- ---- ----- ------- --------- -------
gdb   778  763                     780
cat   780  778   PTRACED
```

Figure 14: Our new plugin detecting cat being debugged

As shown, Volatility correctly reports that *gdb* is tracing PID 780 and that *cat* has the PTRACED flag set. The *Traced by* column of *cat* is empty since it is being directly traced by its parent *gdb* process, so its *parent* and *real_parent* members have the same value.

5.3.2 Detecting SSH Daemon Monitoring

Our second demonstration of *linux_process_ptrace* focuses on detection of an strace instance being used to keylog SSH sessions. Figure 15 shows our invocation of strace to attach to the running SSH daemon and to follow all future children processes.

```
# ps aux | grep sshd
root        436  0.0  1.1  15852  5216 ?  Ss   09:42   0:00 /usr/sbin/sshd -D

# strace -fp 436
strace: Process 436 attached
```

Figure 15: Monitoring SSHD with strace

After attaching strace to the SSH daemon, we then logged into the system through SSH from a remote computer. After successfully logging in, we then ran the *netstat* command and observed the output. Figure 16 shows select portions of this output as generated by our strace invocation.

```
1042  write(4, "\0\0\0\24\f", 5)        = 5
1042  write(4, "\0\0\0\17secretpassword!", 19) = 19
1042  read(4,  <unfinished ...>
1041  <... poll resumed> )              = 1 ([{fd=6, revents=POLLIN}])
1041  read(6, "\0\0\0\24", 4)           = 4
1041  read(6, "\f\0\0\0\17secretpassword!", 20) = 20

1047  read(12, "n", 16384)              = 1
1047  read(12, "e", 16384)              = 1
1047  read(12, "t", 16384)              = 1
1047  read(12, "s", 16384)              = 1
1047  read(12, "t", 16384)              = 1
1047  read(12, "a", 16384)              = 1
1047  read(12, "t", 16384)              = 1

1051  execve("/usr/bin/netstat", ["netstat"], 0x559ee66538e0 /* 20 vars */) = 0

1051  openat(AT_FDCWD, "/proc/net/udp6", O_RDONLY) = 3
1051  read(3, "  sl  local_address              "..., 4096) = 497
1051  read(3, "", 4096)                 = 0
1051  close(3)                          = 0
```

Figure 16: strace output During SSH login and Session

In the first block of output, the string *secretpassword!* can be observed, which is the password of the user account that was logged in with remotely. This theft of plaintext passwords is one of the main reasons that SSHD is a common target of malware, and it also highlights why the use of SSH keys can provide a significant security boost. The second block of output shows our *netstat* command being read one character at a time, leading to the 3rd and final blocks of output showing *netstat* being executed and then network connection information being read from files under */proc/net/*.

Figure 17 shows the output of *linux_process_ptrace* against the memory sample taken after our SSH activity.

```
$ python vol.py ... linux_process_ptrace
Volatility Foundation Volatility Framework 2.6
Name      Pid     PPid  Flags          Traced by  Tracing
--------- ------- ----  -------------- ---------- -------
sshd      436     1     PTRACED|SEIZED 1127
strace    1127    1124                             1140,1139,1131,436
sshd      1131    436   PTRACED|SEIZED 1127
sshd      1139    1131  PTRACED|SEIZED 1127
bash      1140    1139  PTRACED|SEIZED 1127
```

Figure 17: Volatility detecting the SSH Daemon Monitoring

To start, it can be seen that our strace process has a PID of 1127, and that four processes (3 sshd instances, plus one bash instance) report PID 1127 as their *Traced by* value. Furthermore, they all have the *SEIZED* flag set. *SEIZED* indicates that strace either attached to an already running process or was attached to a child automatically as a result of tracing the parent. As we know,

we manually attached strace to the running sshd instance with PID 436 and then our supplied *-f* option to strace told it to follow all future children processes. This information combined with the *Tracing* column of strace matching the four PIDs of the other processes confirms to us that strace is truly the process responsible for tracing the others.

As demonstrated in this section, our new *linux_process_ptrace* plugin can successfully detect all debugged processes in a memory sample, which can lead to automated detection of a wide variety of malware and attacker abuses.

6 Linux Analysis - Input Events

The next Linux keylogging method that we examined was the abuse of input events [21].

6.1 Background

The */dev/input* directory of Linux systems is powered by the input subsystem that exposes a wide range of local devices in a uniform manner. By reading from a specific device's */dev/input* file, applications can receive event data as it is generated. In the case of keyboard devices, this data includes each keystroke typed on the physical keyboard. Figure 18 shows hows devices are exported on a live system.

```
$ ls /dev/input
by-path  event0  event1  event2  event3  mice  mouse0  mouse1

$ ls -l /dev/input/by-path/
total 0
[snip] platform-i8042-serio-0-event-kbd -> ../event1
[snip] platform-i8042-serio-1-event-mouse -> ../event2
[snip] platform-i8042-serio-1-mouse -> ../mouse0
```

Figure 18: Viewing Input Devices

In this output, in it can be seen that the physical keyboard is mapped to the *event1* file while *event2* and *mouse0* correspond to the physical mouse. Given the power of the input subsystem, malware frequently abuses this interface to perform keylogging. This is accomplished by opening a file handle to the device of interest, and then calling the *read* system call in a loop to obtain event data as it is generated.

6.2 Internals

The input subsystem matches devices to input handlers that pass generated events back to the userspace components waiting for them to populate. Documentation maintained by the Linux kernel developers discuss the internals of the kernel portion and userland API of this interface [22]. Next, we show that it is possible

to automatically detect malware abusing this interface solely from its file handle activity, so a deep understanding of the kernel internals is not relevant to our Volatility plugin in this instance.

6.3 Volatility Support

We created the *linux_input_events* plugin to detect processes that are monitoring input events. This plugin operates by enumerating the open file descriptors (handles) of each process and reporting any that reference a path under the */dev/input/* directory. Figure 19 shows the output of our test sample with the *logkeys* keylogger running [23]. This keylogger uses input device monitoring to record keystrokes typed on a physical keyboard attached to a system.

```
$ python vol.py ... linux_input_events
Volatility Foundation Volatility Framework 2.6
Process          Pid  FD Path
---------------- ---- -- ----
systemd-logind   399  10 /dev/input/event0
systemd-logind   399  17 /dev/input/event4
logkeys          4020  0 /dev/input/event0
```

Figure 19: Detecting Device Input Monitoring Processes

As shown, three entries are reported by the plugin. The first two belong to the *system-logind* process, which is a legitimate component of systemd. The process of the third entry, *logkeys*, is not part of systemd and immediately informs the investigator that the process is suspicious and requires further investigation.

7 Linux Analysis - TIOCSTI

The last keylogging approach that was researched for Linux is the abuse of the *TIOCSTI* ioctl [24].

7.1 Background

The *TIOCSTI* ioctl command simulates input to a terminal and allows the caller to inject a specified character into the terminal's input stream. This allows malware to write characters into a victim's terminal window(s), essentially "faking" input typed by the user. This capability has led to a number of vulnerabilities and malware opportunities, such as hijacking su/sudo sessions to run commands as root [25]. The wide ranging threat posed by this IOCTL command led to OpenBSD making it a NOOP [26] and many attempts to lock it down on Linux and within sandboxes.

7.2 Internals

Internally, the handler for this IOCTL simply calls the same code path as if a user had actually typed the given character directly from a keyboard or ssh session. The *tiocsti* function in the Linux kernel implements this and the comment defining the function states: "Fake input to a tty device". Figure 21 shows our POC written in Python that performs keylogging by abusing TIOCSTI.

```
fd = os.open(pty, os.O_RDWR)
tty.setraw(fd)

while True:
      c = os.read(fd, 1024)
      # Insert the given byte in the input queue
      fcntl.ioctl(fd, termios.TIOCSTI, c)

      # Wait until all output written to file descriptor fd has been transmitted.
      termios.tcdrain(fd)

      print("Read: %r" % (c.decode()))
```

Figure 20: TIOCSTI abuse for keylogging

Our POC must first open a handle to the desired terminal device, such as "/dev/pts/0", hosting an SSH session or physical keyboard. Next, it continually reads from the device, and for each set of characters read, it immediately sends them back to the input stream with TIOCSTI. This is required as the *read* performed by our keylogger removes the bytes from the same input queue as used by the victim's terminal. To ensure that characters are reflected back to the terminal and that the user does see any suspicious behavior when our keylogger is active, all characters are immediately reinserted back into the queue this way. *tcdrain* is then used to ensure that all characters are written. Our POC then writes the captured characters to the screen.

Given that the fake character is inserted in exactly the same manner as a real keypress on the keyboard, there is no specific marker in the kernel to tell us that this occurred or is still occurring. Due to this lack of artifacts, we chose to base our detection of this abuse on the open file handle, as discussed next.

7.3 Volatility Support

To detect keyloggers abusing TIOCSTI, we developed the *linux_tiocsti* plugin. The plugin operates by detecting when a process has a file descriptor (handle) open to a terminal device that is not its own. As shown in the POC, a handle to the particular terminal of interest must be opened to read data from it as well as re-insert stolen characters. A naive approach the plugin could have taken would be to compare the terminal device of a process' stdin/stdout/stderr to that of its other descriptors, but malicious processes often close these upon startup or dup() them to network sockets. This would lead to our plugin missing the malicious process.

To ensure that our plugin can detect the full range of TIOCSTI abuse, we instead compare the terminal device associated with the process' signal structure to that of its open file handles. This gives us a direct map between the original terminal where a process was spawned compared to the one it is targeting. Figure 21 shows the output of *linux_tiocsti* against a memory sample with our TIOCSTI-based keylogger active.

```
$ python vol.py ... linux_tty_handles
Volatility Foundation Volatility Framework 2.6
Name    Pid   FD  My Console Handle Console
------- ----- --- ---------- --------------
python  7997  3   pts0       /dev/pts/1


$ python vol.py ... linux_psaux -p 7997
Volatility Foundation Volatility Framework 2.6
Pid  Uid Gid  Arguments
---  --- ---  ---------
7997 0   0    python ssh_keylogger.py
```

Figure 21: Detecting Device Input Monitoring Processes

As shown, the Python process running the keylogger (PID 7997) was spawned on */dev/pts/0* but its 3rd file descriptor is open to */dev/pts/1*, which was the terminal of our victim ssh session. We also show the output of *linux_psaux* as that lists the name of the script and not just the Python interpreter. With this new plugin, investigators can automatically discover the abuse of TIOCSTI within Linux memory samples.

8 macOS - CGEventCreate

The most popular method for keylogging on macOS systems is through creating Event Taps [27].

8.1 Background

macOS Event Taps allow an application to receive a callback notification whenever a monitored hardware device is used, such as a keystroke on a keyboard or a mouse click. Given the power these have on the system, they are commonly abused by malware for keylogging and other malicious purposes. ReiKey from Objective See is a free tool to detect when event taps are registered on a live system and is highly recommend for use on production macOS systems given how frequently this API is abused by malware [28].

Event taps are registered through the *CGEventTapCreate* API [29]. The API requires several arguments, but the most important are the *callback* location, which specifies the callback to activated when events of interest fire, and *eventsOfInterest*, which specifies which events the particular callback is interested. These include a number of methods to monitor the keyboard, mouse, and touchpad.

8.2 Internals

The mechanism for tracking device events and monitors is much different in macOS than Linux and Windows. This is because the macOS the kernel, *xnu*, was originally designed as a microkernel, which puts many of the core subsystems that would be in the kernel on Windows/Linux into userland processes on macOS. It also means that the kernel performs a significant amount of time passing data back and forth between processes (interprocess communication, IPC) since processes cannot directly read/write to each other. The kernel also enforces security boundaries at this layer to ensure processes have the correct privileges to perform requested operations. While analyzing modern versions of macOS, we determined that the subsystem that controls event taps is the SkyLight Framework.

A process that wishes to register event taps must load SkyLight into its address space before it can call *CGEventTapCreate* and related functions. This call first leads to *SLEventTapCreate* being called. The *CG* of *CGEventTapCreate* stands for *Core Graphics* as this was the predecessor framework to SkyLight and the original function name is kept for backwards compatibility. In our testing, we discovered that nearly all *CG** functions have a counterpart *SL** functions inside of SkyLight and the CG functions are now just wrappers.

SLEventTapCreate leads to *event_tap_create* being executed, which performs the real work of creating the tap. To our knowledge, there is *no* online documentation that explains how taps work internally, so we set out to understand this implementation. After sanity checking arguments, *event_tap_create* registers a Mach port with *eventTapMessageHandler* as the callback [30].

Figure 22 shows this registration as well as the eventual call to *CGSPlaceTap*.

```
v12 = malloc(0x10uLL);
v13 = calloc(1uLL, 0x40uLL);
*v12 = 0x100000000LL;
v12[1] = v13;
v13[1] = a1;
v24 = a4;
v13[2] = a4;
*((_QWORD *)v13 + 3) = callback_location;
*((_QWORD *)v13 + 4) = a7;
context.version = 0LL;
context.copyDescription = 0LL;
context.info = v12;
context.retain = (const void *(__cdecl *)(const void *))retainTapProxy;
context.release = (void (__cdecl *)(const void *))releaseTapProxy;
v14 = CFMachPortCreate(0LL, eventTapMessageHandler, &context, 0LL);
if ( !v14 )
{
  free(v13);
  free(v12);
  return 0LL;
}
v7 = v14;
limit_port_queue(v14);
v23 = -1431655766;
v15 = CGSEventServerPort();
v16 = CFMachPortGetPort(v7);
v17 = v15;
v18 = v24;
v19 = _CGSPlaceTap(v17, v16, a1, v27, v26, v24, a5, v13, &v23);
```

Figure 22: *event_tap_create* registering a Mach port and task

As the figure illustrates, the callback location and other parameters to *CGEventTapCreate* are stored within a context variable. This is then passed to *CFMachPortCreate* to associate the event tap with a Mach port that can be referenced across processes. This Mach port plus the other tap information is then passed to *CGSPlaceTap* to actually install the event tap. Figure 23 shows the relevant portion of this function, in particular that it essentially exists to format its received parameters in the form that *mach_msg* expects and then sending this data to *mach_msg*.

```
msg.msgh_size = -1431655766;
v14 = 1;
v15 = a2 | 0xAAAAAAAA00000000LL;
v16 = 1288874;
v17 = NDR_record;
v18 = a3;
v19 = a4;
v20 = a5;
v21 = a6;
v22 = a7;
msg.msgh_bits = -2147478253;
msg.msgh_remote_port = a1;
v9 = mig_get_reply_port();
msg.msgh_local_port = v9;
*(_QWORD *)&msg.msgh_voucher_port = 0x748B00000000LL;
if ( &_voucher_mach_msg_set )
{
  voucher_mach_msg_set(&msg);
  v9 = msg.msgh_local_port;
}
v10 = mach_msg(&msg, 3, 0x44u, 0x34u, v9, 0, 0);
```

Figure 23: *CGSPlaceTap* calling *mach_msg*

mach_msg is a system call used to pass data between Mach ports. To continue investigating the internals of event taps, we needed to find the other end of this IPC call. Our previous knowledge of macOS combined with online searches for confirmation, which brought us to resources such as [31], led us to believe that the *WindowServer* application would be the final destination of these calls.

We then examined this binary in IDA and realized that it is a thin wrapper to daemonize many of the capabilities of SkyLight. We then reexamined the list of functions contained in the SkyLight library and saw that many of the *CG** and *SL** functions had counterparts that started with *_X* followed by the original function name. Further work in IDA confirmed that these are the server side components that handle the IPC messages from calling applications.

Figure 24 shows a portion of *_XPlaceTap*, which is the server side component of *CGSPlaceTap* used inside of the *WindowServer* process.

```
new_CGXEventTap = calloc(1uLL, 0xC0uLL);
if ( v4 )
  *((_QWORD *)new_CGXEventTap + 2) = *(_QWORD *)(__sessionControlRef + 32);
*((_DWORD *)new_CGXEventTap + 6) = v30;
*((_DWORD *)new_CGXEventTap + 7) = v31;
*((_DWORD *)new_CGXEventTap + 8) = generate_new_tap_id();
*((_DWORD *)new_CGXEventTap + 9) = v4;
*((_DWORD *)new_CGXEventTap + 10) = v32;
*((_QWORD *)new_CGXEventTap + 6) = v36;
*((_DWORD *)new_CGXEventTap + 46) = v33;
*(_DWORD *)(a2 + 40) = v36;
*((_DWORD *)new_CGXEventTap + 14) = name;
*((_QWORD *)new_CGXEventTap + 15) = (v11 << 32) | v34;
if ( (v32 & 1) != 0 )
{
  *(_OWORD *)v39.val = v29;
  *(_OWORD *)&v39.val[4] = v8;
  *(_OWORD *)&v28.val[4] = v8;
  *(_OWORD *)v28.val = v29;
  v15 = WSAuditTokenCanMonitorEvents(v28) ^ 1;
}
else
{
  v15 = 0;
}
*((_BYTE *)new_CGXEventTap + 128) = v15;
*((_DWORD *)new_CGXEventTap + 43) = 0;
*((_BYTE *)new_CGXEventTap + 188) = gLastAllTapsLoggingEnabledSetting;
*((_QWORD *)new_CGXEventTap + 1) = sCGXEventTapMasterList;
sCGXEventTapMasterList = (CGXEventTap *)new_CGXEventTap;
```

Figure 24: *_XPlaceTap* creating a new event tap structure

In this figure, a new data structure, of type *CGXEventTap* is being allocated to store the information passed from the previous *mach_msg* call. Note that we renamed the data structure to *new_CGXEventTap* so that it was easier to follow the IDA decompiler output. After the data structure is populated, it can be seen on the last line of the figure that the *sCGXEventTapMasterList* global variable is being set to the address of the new tap. Finding this function and understanding its code resulted in two benefits. First, we used this to determine the offsets of data structure members that our plugin would need to retrieve inside of *CGXEventTap*. Thankfully, the offsets of the members of interest did not change across the macOS versions we tested. The second benefit to this function was the discovery of the *sCGXEventTapMasterList* global variable, as we realized it was likely the data structure inside of the *WindowServer* process that stored all event taps for all processes.

We then used IDA's cross-referencing capabilities to see where else this variable was used. This led us to *_XGetEventTapList*, which is the server side handler for *CGGetEventTapList* [32]. This was very encouraging as we knew this API is what allows tools, such as ReiKey, to enumerate event tap handlers on a system. Analysis of this function confirmed to us that the global variable is used

to track all event taps, and that it stores pointers to all handlers contiguously in memory. With this binary analysis effort complete, we were then able to devise an algorithm for a Volatility plugin that could recover all event taps registered on a system.

8.3 Volatility Support

Before our research, Volatility had no method to enumerate event taps for macOS memory samples. After researching the internals of SkyLight, we developed the *mac_event_taps* plugin to provide this capability to memory forensic analysts. The plugin begins by finding the *WindowServer* process. It then locates the *sCGXEventTapMasterList* global variable and processes each pointer that it references. Each of these pointers leads to a *CGXEventTap* structure, and as mentioned previosuly, our binary analysis effort led us to discover the relevant offsets inside of this data structure.

For our POC, we used the very well written *keylogger* project by Casey Scarborough [33]. This project uses *CGEventTapCreate* to monitor Key Down events on the keyboard. Figure 25 shows the portion of the keylogger that creates the event tap and then attaches it to the process' run loop.

```
// Create an event tap to retrieve keypresses.
CGEventMask eventMask = CGEventMaskBit(kCGEventKeyDown) | CGEventMaskBit(kCGEventFlagsChanged);
CFMachPortRef eventTap = CGEventTapCreate(
    kCGSessionEventTap, kCGHeadInsertEventTap, 0, eventMask, CGEventCallback, NULL
);

// Exit the program if unable to create the event tap.
if (!eventTap) {
    fprintf(stderr, "ERROR: Unable to create event tap.\n");
    exit(1);
}

// Create a run loop source and add enable the event tap.
CFRunLoopSourceRef runLoopSource = CFMachPortCreateRunLoopSource(kCFAllocatorDefault, eventTap, 0);
CFRunLoopAddSource(CFRunLoopGetCurrent(), runLoopSource, kCFRunLoopCommonModes);
CGEventTapEnable(eventTap, true);
```

Figure 25: Keylogger registering for Key Down events

Figure 26 shows the output of our new plugin against a memory sample with this keylogger active.

```
$ python vol.py ... mac_event_taps
Volatility Foundation Volatility Framework 2.6
Tapping Process Tapping Pid Events of Interest
--------------- ----------- -----------------
keylogger               958 keyDown, flagsChanged
```

Figure 26: *mac_event_taps* Detecting the Keylogger

As shown, through analysis of SkyLight and *sCGXEventTapMasterList*, our plugin automatically discovers the keylogger's tap and its registered events of interest. With this new plugin, investigators can automatically discover all event tapping malware present in macOS memory samples.

9 Conclusions

In this paper, we have presented a significant memory analysis research effort that led to the creation of algorithms and Volatility plugins capable of detecting the most widely abused userland device monitoring techniques across the three most widely used operating systems. As widely documented in technical reports, such malware techniques have been used across the world to target journalists and opposition political figures as well as in espionage campaigns aimed at nation states and corporations. Our research effort also included updating existing memory forensics algorithms to support the latest Windows versions, as well as development of completely new detection techniques for each of the three operating systems. Once included in the mainline Volatility Framework, investigators across the field will be able to automatically discover device monitoring malware across Windows, Linux, and macOS systems.

References

[1] CISA, "Mitigate Microsoft Exchange Server Vulnerabilities," https://www.cisa.gov/uscert/ncas/alerts/aa21-062a, 2021.

[2] ——, "EMERGENCY DIRECTIVE 21-01- MITIGATE SOLARWINDS ORION CODE COMPROMISE," https://www.cisa.gov/emergency-directive-21-01, 2021.

[3] Volexity, "Surge Collect Pro," https://www.volexity.com/products-overview/surge/, 2022.

[4] Elixir, "Elixir," https://elixir.bootlin.com/], 2022.

[5] Apple, "Apple Open Source," [https://opensource.apple.com], 2022.

[6] Volatility, "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, 2012.

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

[8] T. M. Corporation, "Turla," https://attack.mitre.org/groups/G0010/, 2018.

[9] Microsoft, "LowLevelKeyboardProc callback function," https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ms644985(v=vs.85), 2022.

[10] Geok Meng Ong, Chong Rong Hwa, "Pacific Ring of Fire: PlugX / Kaba," https://www.fireeye.com/blog/threat-research/2014/07/pacific-ring-of-fire-plugx-kaba.html, 2014.

[11] H. K. Chan, "VB2014 paper: Swipe away, we're watching you," https://www.virusbulletin.com/virusbulletin/2015/04/paper-swipe-away-we-re-watching-you, 2015.

[12] Hod Gavriel, "HawkEye Malware Changes Keylogging Technique," https://www.cyberbit.com/blog/endpoint-security/hawkeye-malware-keylogging-technique/, 2019.

[13] D. Web, "Study of the APT attacks on state institutions in Kazakhstan and Kyrgyzstan," https://st.drweb.com/static/new-www/news/2020/july/Study_of_the_APT_attacks_on_state_institutions_in_Kazakhstan_and_Kyrgyzstan_en.pdf, 2020.

[14] PTI TEAM, "OpBlueRaven: Unveiling Fin7/Carbanak - Part I : Tirion," https://threatintel.blog/OPBlueRaven-Part1/, 2020.

[15] Profero, "Apt27 turns to ransomware," https://shared-public-reports.s3-eu-west-1.amazonaws.com/APT27+turns+to+ransomware.pdf, 2020.

[16] Mike G. P.Mee, "Minimal Key Logger Using RAWINPUT," https://www.codeproject.com/Articles/297312/Minimal-Key-Logger-using-RAWINPUT, 2012.

[17] Eye of Ra, "Windows Keylogger Part 2: Defense against user-land," https://eyeofrablog.wordpress.com/2017/06/27/windows-keylogger-part-2-defense-against-user-land/, 2017.

[18] debojit, "Spying on ssh password using strace," https://medium.com/@deboj88/spaying-on-ssh-password-using-strace-7465ede0a5cc, 2018.

[19] RedHat, "4.15. DISABLING PTRACE()," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/sect-security-enhanced_linux-working_with_selinux-disable_ptrace, 2022.

[20] CISA, "LIMITING PTRACE ON PRODUCTION LINUX SYSTEMS," https://media.defense.gov/2019/Jul/16/2002158062/-1/-1/0/CSI-LIMITING-PTRACE-ON-PRODUCTION-LINUX-SYSTEMS.PDF, 2022.

[21] Linux, "The Linux Input Documentation," https://www.kernel.org/doc/html/v4.14/input/index.html, 2022.

[22] ——, "Linux Input Subsystem userspace API," https://www.kernel.org/doc/html/latest/input/input_uapi.html, 2022.

[23] kernc, "logkeys," https://github.com/kernc/logkeys, 2022.

[24] QNX, "TIOCSTI," https://www.qnx.com/developers/docs/7.0.0/#com.qnx.doc.neutrino.devctl/topic/tioc/tiocsti.html, 2022.

[25] Simon Ruderich, "su/sudo from root to another user allows TTY hijacking and arbitrary code execution," https://ruderich.org/simon/notes/su-sudo-from-root-tty-hijacking, 2021.

[26] brynet, "On the Insecurity of TIOCSTI," https://undeadly.org/cgi?action=article;sid=20170701132619, 2017.

[27] Apple, "Quartz Event Services," https://developer.apple.com/documentation/coregraphics/quartz_event_services, 2022.

[28] Patrick Wardle, "ReiKey," https://objective-see.com/products/reikey.html, 2022.

[29] Apple, "CGEventTapCreate," https://developer.apple.com/documentation/coregraphics/1454426-cgeventtapcreate, 2022.

[30] ——, "CFMachPortCreate," https://developer.apple.com/documentation/corefoundation/1400934-cfmachportcreate, 2022.

[31] hoakley, "WindowServer: display compositor and input event router," https://eclecticlight.co/2020/06/08/windowserver-display-compositor-and-input-event-router/, 2022.

[32] Apple, "CGGetEventTapList," https://developer.apple.com/documentation/coregraphics/1455395-cggeteventtaplist, 2022.

[33] Casey Scarborough, "macOS Keylogger," https://github.com/caseyscarborough/keylogger, 2022.