New Memory Forensics Techniques to Defeat Device Monitoring Malware

Andrew Case, Gustavo Moreira, Austin Sellers, Golden Richard

Motivation

- Malware that is capable of monitoring hardware devices (keyboards, microphones, web cameras, etc.) is now commonly deployed against human targets
- This type of malware poses a serious threat to privacy and security
- Existing memory forensic algorithms against this type of malware are outdated, incomplete, or non-existent

Research Goals

- For the major operating systems (Windows, Linux, macOS):

1. Study the methods used by userland (process) malware to monitor hardware devices
2. Research (source code review, binary analysis) how the abused APIs are implemented
3. Determine if current memory forensics tools could detect each abuse
4. For ones not currently detected, develop capabilities to automatically detect the abuse

Why Memory Forensics is Needed

- Across platforms, memory-only payloads are often used by malware that monitors hardware devices
- Disk and live forensics generally can find no traces of this malware
- Volatile memory is the ***only*** place to determine that such malware is present and to fully investigate it

Windows Research Setup

- Focused on Windows 10
- Analyzed all major builds starting with 10563 (2015) through 22000.556 (March 2022)
- Developed POC software that used the APIs abused by real-world malware
- Memory collection with VMware suspend states for initial work
- Used Surge Collect Pro and its file collection capabilities for long term automated testing

Windows Research - SetWindowsHookEx

- Historically, the most widely abused API by userland keyloggers
- Allows for registering for hooks (callbacks) for hardware events of interest in all threads in a desktop or a specific thread
- The most common use of the API leads to the malicious DLL being injected into *every* process where a hook triggers (keystroke, mouse movement, etc.)
- Volatility's *messagehooks* plugin aims to recover abuse of this API
  - Never properly updated for Windows 10
  - Testing showed it did not support all hook variations

SetWindowsHookEx - Global Hooks in a DLL

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

WH_KEYBOARD_LL
, WH_MOUSE, …

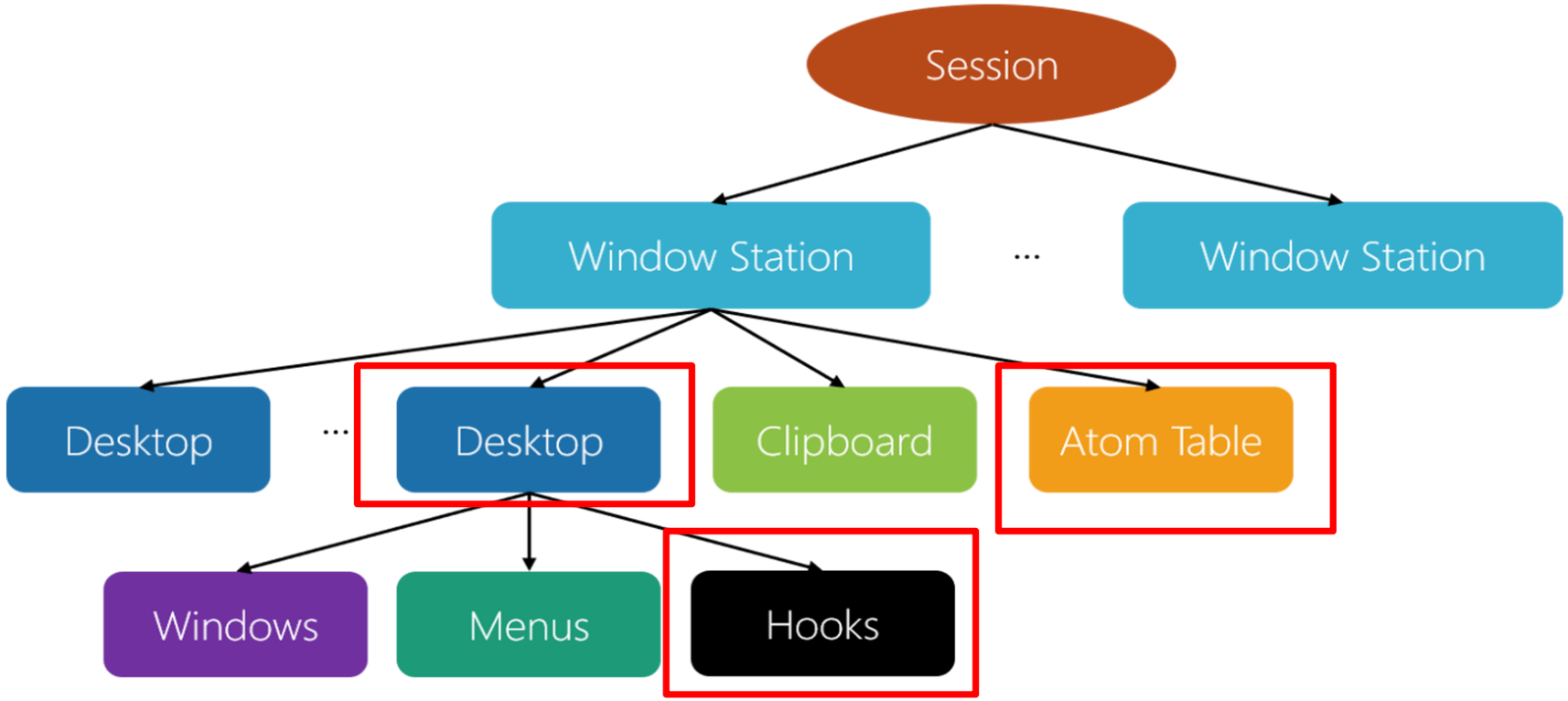
C:\keylogger.dll

NULL

```
ExpandEnvironmentStringsA("%userprofile%\\Desktop\\FakeDll.dll", fakeDLL_fName, MAX_PATH - 1);
HMODULE dll = LoadLibraryA(fakeDLL_fName);

HOOKPROC addr = (HOOKPROC)GetProcAddress(dll, "mouse_hook_procedure");

mouse_hook = SetWindowsHookExA(
        WH_MOUSE, // mouse input
        addr, // pointer to the hook procedure
        dll, // A handle to the DLL containing the hook procedure
        0 //process_info.dwThreadId //desktop apps, if this parameter is zero
);
```

```
$ python vol.py ... dlllist -p 2184

notepad.exe pid:   2184

Base             Size     LoadCount LoadTime                      Path
---------------- -------- --------- --------- -------------------- ------------
0x7ff6bee30000 0x41000  0xffff    2022-06-29 19:00:31 UTC+0000 C:\Windows\notepad.exe
0x7ffe83510000 0x1c1000 0xffff    2022-06-29 19:00:31 UTC+0000 C:\Windows\SYSTEM32\ntdll.dll
0x7ffe813e0000 0xad000  0xffff    2022-06-29 19:00:31 UTC+0000 C:\Windows\system32\KERNEL32.DLL
<snip>
0x7ffe74090000 0x8000   0x6       2022-06-29 19:00:31 UTC+0000 C:\Users\Administrator\Desktop\FakeDll.dll
<snip>
```

Enumerating Global Message Hooks



Enumeration Algorithm

1) Enumerate the **Desktops** of each of Session -> Window Station

2) Enumerate the hooks (**tagHOOK**) of each **Desktop**

3) Gather the **full path to the DLL** hosting each hook through the (new) **Atom Table**

Image Source: [2], Full Technical Details: [1]

```
Offset(V)   : 0xfffff90146001e10 0xfffff90146001e10
Session     : 1
Desktop     : WinSta0\Vol_GUI-DesktopHidden
Thread      : 5572 (notepad.exe 2184) False
Filter      : WH_MOUSE
Flags       : HF_ANSI|HF_GLOBAL
Procedure   : 0x10f0
ihmod       : 7
Module      : C:\Users\Administrator\Desktop\FakeDll.dll
```

```
if ( v1
  && ((v7 = UserGetAtomNameFromAtomTable(
             UserLibmgmtAtomTableHandle,
             *((unsigned __int16 *)&aatomSysLoaded + v2),
             v12,
             260i64),
       RtlInitUnicodeString(&String1, v12),
```

SetWindowsHookEx - Global Hooks in an EXE

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

WH_KEYBOARD_LL

NULL

NULL

Global Exe Hooks – WH_KEYBOARD_LL Only

```
keyboard_hook = SetWindowsHookExA(
      WH_KEYBOARD_LL, // low-level keyboard input events
      keyboard_hook_procedure, // pointer to the hook procedure
      GetModuleHandle(NULL), // A handle to the DLL containing the hook procedure
      NULL
);
```

This hook is called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

The keyboard input can come from the local keyboard driver or from calls to the keybd_event function. If the input comes from a call to **keybd_event**, the input was "injected". However, the WH_KEYBOARD_LL hook is not injected into another process. Instead, the context switches back to the process that installed the hook and it is called in its original context. Then the context switches back to the application that generated the event.

```
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)
```

First output block with “<any>” denotes that this hook applies to all threads in the desktop

Volatility knows the address of the hook, but not which process is hosting it

```
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)
```

During our research, we discovered that the *TIF_GLOBALHOOKER* flag denotes if a thread has placed a hook. Volatility now parses this flag.

```
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)
```

“True” here tells us that *GUITesterAll* placed the hook

SetWindowsHookEx – Thread Specific Hooks

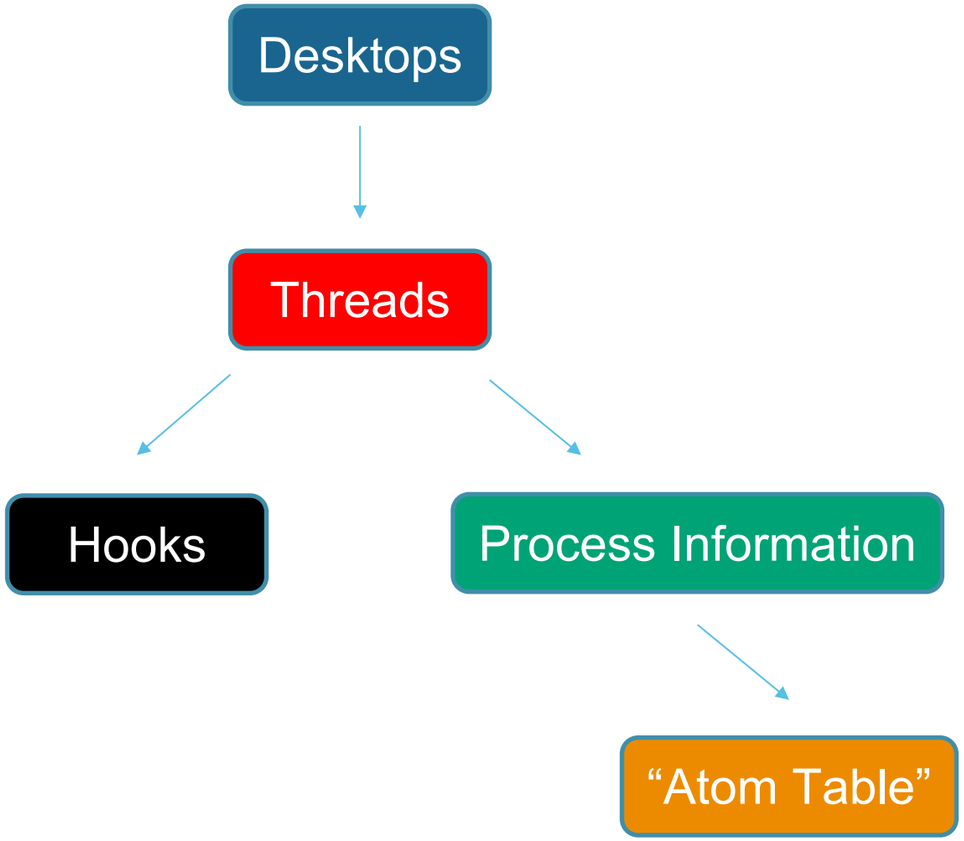
```
HHOOK SetWindowsHookExA(
    [in] int          idHook,
    [in] HOOKPROC     lpfn,
    [in] HINSTANCE    hmod,
    [in] DWORD        dwThreadId
);
```

WH_KEYBOARD_LL
, WH_MOUSE, etc.

DLL Handle | NULL

<TID of target thread>

Enumerating Thread-Specific Hooks



- Thread-specific hooks are stored within the thread data structure
- A per-process data structure holds the “atom table” equivalent list of DLLs
- Volatility was previously unable to enumerate these hooks

Adding Initial Support

```
Offset(V)   : 0xffff90146007630
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
```

Incorporating Per-Process “Atom Table”

```
Module      : C:\Users\Administrator\Desktop\FakeDll.dll
```

Windows Research - RegisterRawInputDevices

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

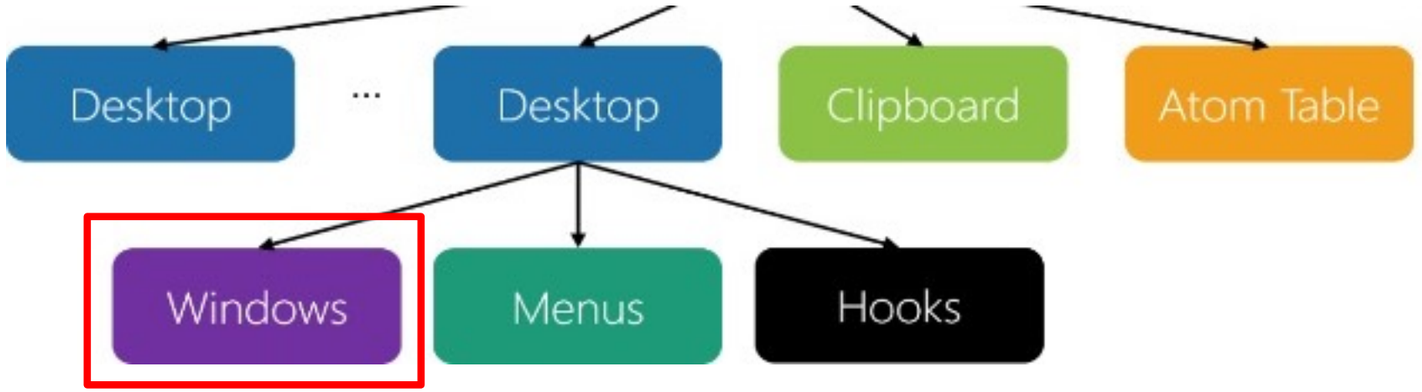
```
typedef struct tagRAWINPUTDEVICE {
  USHORT usUsagePage;
  USHORT usUsage;
  DWORD  dwFlags;
  HWND   hwndTarget;
} RAWINPUTDEVICE, *PRAWINPUTDEVICE,
```

HID_USAGE_PAGE_GENERIC

HID_USAGE_GENERIC_KEYBOARD

<Handle to the target window>

Registering to Monitor

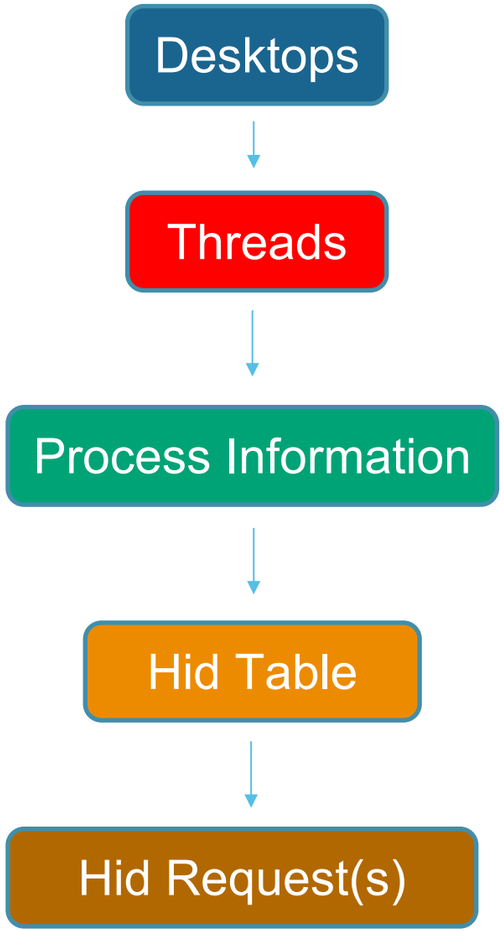



```
WNDCLASS wc = { 0 };

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", ...);
```

Malicious Window Callback Procedure

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        switch (message) {
                case WM_CREATE: {
                        // register interest in raw data
                        // ignore legacy messages and receive system wide keystrokes
                        rid.dwFlags = RIDEV_NOLEGACY | RIDEV_INPUTSINK;

                        rid.usUsagePage = HID_USAGE_PAGE_GENERIC;
                        rid.usUsage = HID_USAGE_GENERIC_KEYBOARD;
                        rid.hwndTarget = hWnd;
                        RegisterRawInputDevices(&rid, 1, sizeof(rid));
                        break;

                case WM_INPUT: {
                        if (GetRawInputData((HRAWINPUT)lParam, RID_INPUT, ... {
                                break;
                        }
                        LPBYTE lpb = new BYTE[dwSize];
                        if (lpb == NULL) {
                                break;
                        }
                        if (GetRawInputData (HRAWINPUT)lParam, RID_INPUT, ... {
                                delete[] lpb;
                                break;
                        }
```

Enumerating Input Monitors



- Per-process data structure stores a HID table
- This table stores a list of monitoring requests
- Each request tracks its target window and usage

Detecting the Device Monitor

```
$ python vol.py ... rawinputdevicemonitors

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
```

Linux Research – strace and ptrace

- *ptrace* is the debugging facility of Linux
- *strace* is a popular tool that relies on *ptrace* to monitor system calls made by other processes
- Allows for monitoring of buffers sent to hardware devices (keyboards, mics, …)
- Can be completely locked down, even to root users – but not universally applied

Detecting Direct Debugging

```
# 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
```

```
$ 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
```

```
# 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


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
```

Logging in with “secretpassword!” password

Typing ‘netstat’ once character at a time

Detecting Child Process Debugging

```
$ 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
```

NOTE: SEIZED means that a process began being debugged after it was already started or that a child process was automatically debugged as a result of its parent process being debugged. See the *ptrace(2)* manual page for complete information.

Linux Research – Input Events

The Input Event subsystem can be abused by userland malware to monitor keystrokes on physically attached keyboards

```
ls -l /dev/input/by-path
total 0
[snip] pci-0000:02:00.0-usb-0:1:1.0-event-mouse -> ../event3
[snip] pci-0000:02:00.0-usb-0:1:1.0-mouse -> ../mouse2
[snip] platform-i8042-serio-0-event-kbd -> ../event0
[snip] platform-i8042-serio-1-event-mouse -> ../event2
[snip] platform-i8042-serio-1-mouse -> ../mouse0
[snip] platform-pcspkr-event-spkr -> ../event5
```

```
$ 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
```

Linux Research - TIOCSTI

- TIOCSTI is an IOCTL that simulates input to a specific terminal and allows the caller to *inject a character into that terminal's input stream*

su/sudo from root to another user allows TTY hijacking and arbitrary code execution

First written 2016-10-17; Last updated 2021-10-05

back

TL;DR: Don't run `su - $USER` or `sudo -u $USER` (unless `use_pty` is set) as root or `$USER` may inject arbitrary commands in your root shell. At least Linux, ~~OpenBSD~~ and FreeBSD are affected. This is not an issue when `su`-ing **to** root.

Update (2021-10-05): OpenBSD removed `TIOCSTI` in OpenBSD 6.2 making it no longer vulnerable. Linux and FreeBSD are still vulnerable.

Screenshot source: https://ruderich.org/simon/notes/su-sudo-from-root-tty-hijacking

Detecting TIOCSTI Abuse

```
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
        termios.tcdrain(fd)

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

```
$ 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
```

macOS Research - CGEventTapCreate

- *CGEventTapCreate* is the most widely abused API on macOS for hardware device monitoring

```
CFMachPortRef CGEventTapCreate(CGEventTapLocation tap, CGEventTapPlacement place, CGEventTap
Options options, CGEventMask eventsOfInterest, CGEventTapCallBack callback, void *userInfo);
```

`case keyDown`

Specifies a key down event.

`case keyUp`

Specifies a key up event.

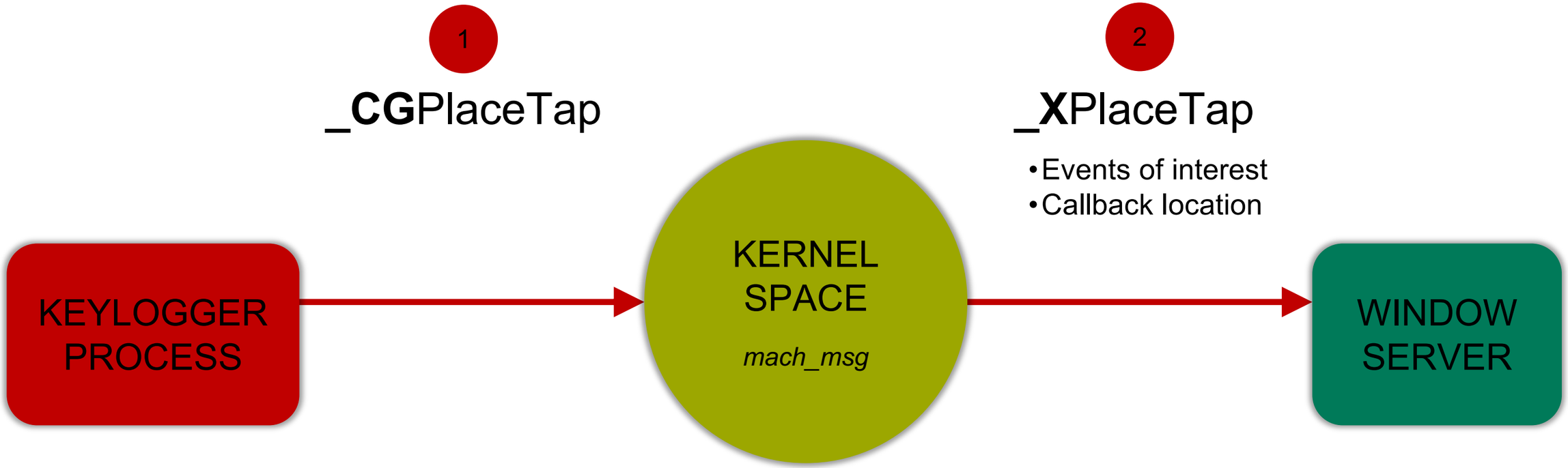
`case flagsChanged`

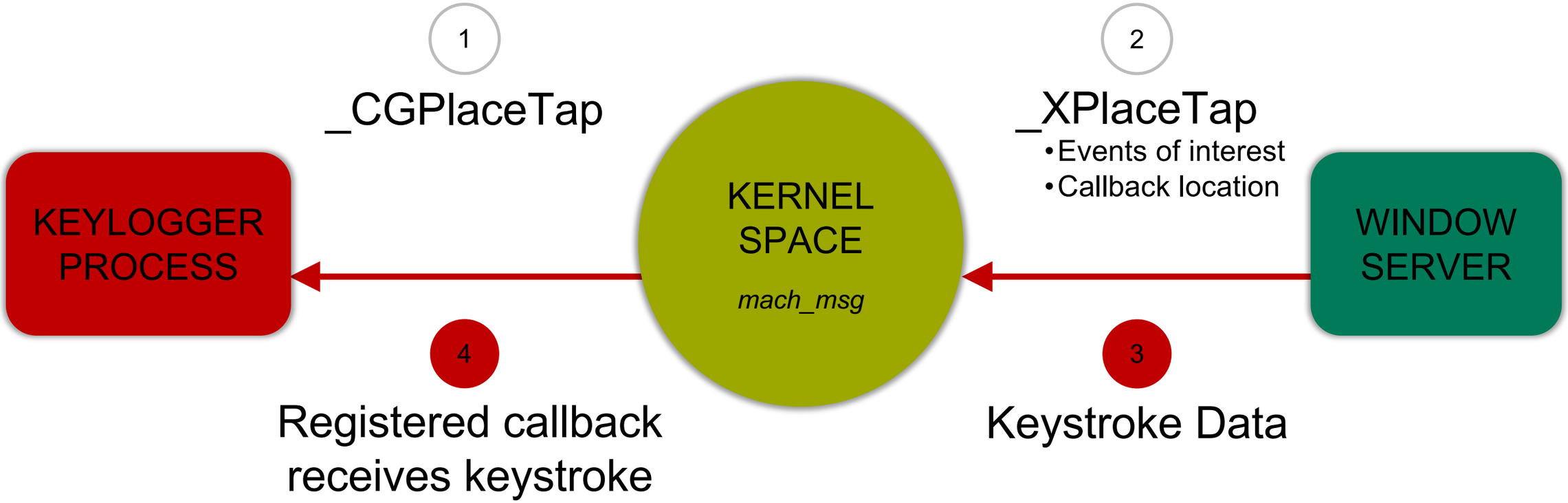
Specifies a key changed event for a modifier or status key.

CGEventTapCreate POC Code

```
// 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);
```

POC source: https://github.com/caseyscarborough/keylogger





Detecting CGEventTapCreate Abuse

```
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();
*((_BYTE *)new_CGXEventTap + 128) = v15;
*((_DWORD *)new_CGXEventTap + 43) = 0;
*((_BYTE *)new_CGXEventTap + 188) = gLastAllTapsLoggingEnabledSetting;
*((_QWORD *)new_CGXEventTap + 1) = sCGXEventTapMasterList;
sCGXEventTapMasterList = (CGXEventTap *)new_CGXEventTap;
```

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

Conclusions

- Malware that targets devices will continue to pose a serious privacy and security threat to individuals and organizations
- Our research effort enables automated detection and analysis of such malware
- Many of the data structures and subsystems analyzed previously had no public documentation
- Please see our whitepaper on the Black Hat website for complete details
  - Nearly 30 pages of code samples, IDA Pro screenshots, data structure breakdowns, and more

Questions? Comments?

Contact

andrew@dfir.org

golden@cct.lsu.edu

Social Media

@volatility, @attrc, @nolaforensix, @volexity, @lsucct

2022 Volatility Plugin Contest now open!

https://volatility-labs.blogspot.com/2022/07/the-10th-annual-volatility-plugin-contest.html

References

[1] https://volatility-labs.blogspot.com/2012/09/movp-31-detecting-malware-hooks-in.html
[2] https://scorpiosoftware.net/2019/02/17/windows-10-desktops-vs-sysinternals-desktops/