Locking Down the Windows Kernel: Mitigating Null Pointer Exploitation

Tarjei Mandt

Norman Threat Research
tarjei.mandt@norman.com

Abstract. One of the most prevalent bug classes affecting Windows kernel components today is undeniably `NULL` pointer dereferences. Unlike other platforms such as Linux, Windows (in staying true to backwards compatibility) allows non-privileged users to map the null page within the context of a user process. As kernel and user-mode components share the same virtual address space, an attacker may potentially be able to exploit kernel null dereference vulnerabilities by controlling the dereferenced data. In this paper, we propose a way to generically mitigate `NULL` pointer exploitation on Windows by restricting access to the lower portion of process memory using VAD manipulation. Importantly, as the proposed method employs features already present in the memory manager and does not introduce any offending hooks, it can be introduced on a wide range of Windows platforms. Additionally, because the mitigation only introduces minor changes at process creation-time, the performance cost is minimal.

Keywords: null pointer vulnerabilities, exploitation, mitigation

1 Introduction

A null-pointer dereference takes place when a pointer with a value of `NULL` [1] is used to as though it pointed to a valid memory area. From a remote code execution perspective, vulnerabilities that operate on null pointers are rarely considered exploitable (unless you control the offset from null as well) as the program affected by the vulnerability is not likely to have the corresponding page mapped. This also applies to web browsers and other scriptable applications, despite an attacker having a large degree of control of the process memory.

From a local privilege escalation perspective, the situation is different. In this case, the attacker already has a foothold on a system and is free to run arbitrary code under the context of the current user. Because user processes and kernel mode components share the same virtual address space, a kernel `NULL` pointer deference will inadvertently operate on user-mode memory. As users are free to allocate the null page on Windows for compatibility reasons, kernel `NULL` pointer dereferences often end up being exploitable. Unfortunately, Windows does not allow users to configure the behavior of the memory manager to prevent mapping of the null page. Because many kernel vulnerabilities [8]

as well as recent exploitation techniques [7] rely on null page mapping to be leveraged by an attacker, addressing this vector becomes an important goal in mitigating kernel exploitation on Windows.

In this paper, we propose a method for mitigating exploitation of `NULL` pointer dereference vulnerabilities on Windows. Specifically, our proposed solution leverages VAD manipulation to ensure that no running process may allocate memory or load modules at the null page. As we operate completely within the scope of the memory manager without using offending hooks of any kind, it can be customized to work on any supported Windows platform. A proof of concept driver has been developed to demonstrate the proposed method on both x86 and x64 versions of Windows 7.

The rest of the paper is organized as follows. In Section 2, we review related work on addressing `NULL` pointer exploitation on both Windows and other platforms. In Section 3, we cover some of the internals regarding Windows memory management, necessary to understand the remainder of the paper and on which the proposed method is built on. In Section 4, we discuss various methods for addressing null page mappings on Windows, while in Section 5 we outline the proposed implementation. In Section 6, we discuss the advantages and disadvantages of the proposed method. Finally, in Section 7 and 8 we discuss future areas of research and provide a conclusion of the paper.

2 Related Work

On Windows, the Enhanced Mitigation Experience Toolkit (EMET) [12] is a toolkit developed by Microsoft designed to make it more difficult for an attacker to exploit software vulnerabilities. It deploys mitigations such as mandatory ASLR[1], DEP, heap spray pre-allocation, and export address table filtering. EMET also features null-page protection (by marking the null-page `NOACCESS`) to prevent exploits from referencing this area of memory. Although this mitigation may thwart remote exploitation attempts in certain scenarios, it does not in any way prevent an attacker from gaining control of the null page once on a local system or when code execution has been obtained. This is because an attacker could easily modify its protection and thus leverage it in exploiting a kernel `NULL` dereference vulnerability.

Although new to Windows, mitigations against `NULL` pointer vulnerabilities have already been introduced on many other widely used platforms. Linux traditionally allowed null pages to be mapped, but have since the 2.6.23 kernel enforced memory mapping restrictions using `mmap_min_addr`[2]. Specifically, this tunable sets the smallest virtual memory address that can be allocated (e.g. via the `mmap()` function) and thus acts as a mitigation towards kernel `NULL` pointer dereferences. Although introducing the required checks may seem straightforward, Linux has yet to prove that its implementation is a proper one. This stems from the fact that the mitigation is not added in a centralized way, but rather

[1] Mandatory ASLR uses module rebasing on ASLR incompatible systems
[2] FreeBSD uses the sysctl(8) variable security.bsd.map_at_zero

in all places where the null page could be allocated or mapped. This in turn led to a number of interesting cases in which the mitigation could be bypassed, either by leveraging setuid applications with the `MMAP_PAGE_ZERO` personality to map the null page (CVE-2009-1895 [14]), or by leveraging functions that did not call the appropriate security hooks (CVE-2007-6434 [10], CVE-2010-4346 [9]). Because of the problems faced in Linux, it clearly becomes a goal to enforce the null page mapping restriction in a more centralized manner.

A more aggressive way of preventing user-mode dereferences from the kernel is using x86 segmentation. In ensuring that the kernel data segment (`__KERNEL_DS`) does not extend into the user-mode address space, any attempts at referencing user-mode memory results in an access violation. This is exactly what the *PaX project* does with a feature called `PAX_UDEREF` [13] in order to mitigate kernel exploitation attempts involving user-mode dereferences. The challenge with using segmentation is that it requires significant changes to how data is passed between user and kernel-mode (in order to deal with the non-overlapping segments). `PAX_UDEREF` solves this by reloading the proper segment register with `__USER_DS` within `copy_from_user` and `copy_to_user` for the duration of the copy. Unfortunately, Windows lacks consistent APIs for passing data between rings, hence the approach would be very difficult to implement without making significant changes to the kernel. Another big drawback is the lack of proper segmentation support on x64, which makes segmentation unsuitable for generically addressing `NULL` pointer exploitation on Windows.

3 Memory Management

In order to understand how null page mappings can be addressed on Windows, we will briefly review some internals regarding process memory management. We begin by describing how the OS and the CPU manages virtual addresses, and move on to more operating system specific details such as virtual address descriptors (VADs).

3.1 Page Tables

In order for the CPU to translate virtual addresses to physical addresses, the memory manager creates and maintains page tables. In Windows, each process has its own page table, and stores the pointer to the physical base address of the page directory in the CR3 register. When a context switch occurs, Windows loads this register with the value held by the `DirectoryTableBase` field in the `KPROCESS` structure, effectively granting each process its own process space. Upon a memory read or write, the CPU uses the virtual address to look up the corresponding page table entry (using a two, three, or four-level table lookup depending on the running architecture) in order to determine whether the page is present. Because page table address lookups are inefficient, the CPU uses a translation lookaside buffer (TLB) to store recently translated entries.

As PTEs are 32 bit (Figure 1) and pages on the Intel platform are 4KB, only 20 bits need to be used to describe a physical page. The remaining 12 bits are used by the hardware and the operating system to maintain control information about the virtual page represented by this particular entry.

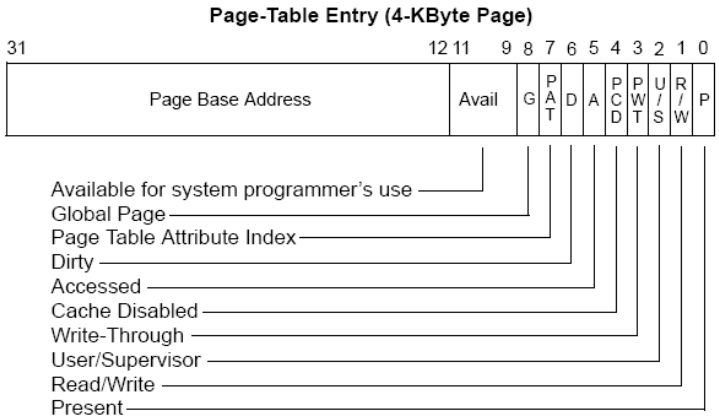

Fig. 1. Page table entry

One of the most important bits is the **valid** bit. If the bit is set, the memory management unit (MMU) of the CPU can perform the address resolution on its own and locate the corresponding physical page. If the bit is cleared, the OS is invoked to take action (calls the page fault handler) and is free to use the remaining bits of the PTE arbitrarily. Also, of particular interest to restricting page access is the **owner** or U/S bit. When cleared, only kernel mode (supervisor) may access the page referenced by the PTE. We look more into using information held in the PTE to restrict page access in Section 4.2.

3.2 Virtual Address Descriptors

In order to conserve memory and enhance performance, Windows uses a lazy evaluation algorithm in which a PTE is not actually created until the reserved or committed memory is first accessed. This method significantly improves performance for processes that allocate large amounts of memory which may only be scarcely accessed. In order to support the lazy evaluation algorithm, Windows needs to keep track of the memory that has not yet been accessed (and for which PTEs have not yet been created). As such, each process maintains a set of data structures known as virtual address descriptors (VADs). VADs are structured in self-balancing AVL trees and hold all the information on the memory ranges allocated in a process, necessary to set up PTEs correctly. This includes information such as the page protection, whether memory has been committed,

pointers to associated file objects in memory mapped files, and so on. A diagram of an example VAD tree is shown in Figure 2.

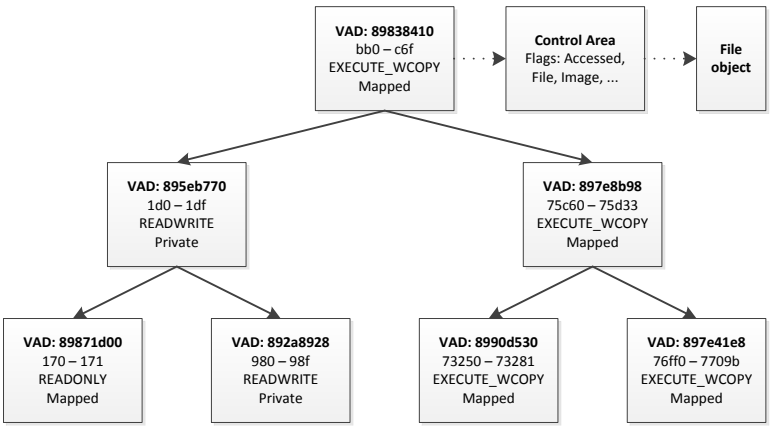

Fig. 2. Root of VAD tree for calc.exe

When a thread first accesses a memory region, the page fault handler notices that the PTE has not been created (marked invalid) and inspects the associated VAD in order to set it up correctly (in `nt!MmAccessFault`). If the memory region does not have a corresponding VAD, the handler knows that memory was not allocated by the process and therefore generates an access violation. The memory manager also uses VADs to implement the `PAGE_NOACCESS` protection, as valid PTEs in the x86 architecture have no flags that restrict memory reads[3]. When a VAD is marked `NO_ACCESS`, the memory access fault handler simply returns with an access violation without taking any further action.

The `NO_ACCESS` property of a VAD provides an ideal entry point to addressing null page mappings on Windows. In the next section, we will show that despite the lack of proper support for denying page mappings in user-space, such restrictions can be enforced generically using VAD manipulation.

4 Denying Null Page Mapping

In this section, we discuss ways of denying null page mappings on Windows. We begin by looking at traditional system call hooking and modification of page table entries, before going more into what can be done using VAD manipulation.

[3] In clearing the U/S flag, the page will only become accessible to the kernel

4.1 System Call Hooking

Historically, hooking system calls have been a popular way for security solutions as well as rootkits to filter the information flow between user and kernel-mode components. For preventing null page mappings on Windows, there appears to be only two system calls that we're concerned with: `NtAllocateVirtualMemory` and `NtMapViewOfSection`. Both functions allow the null page to be mapped in specifying a base address less than the size of a page, but above zero. Hence, validating the `BaseAddress` parameter before passing the request to the kernel could be used in preventing null page mappings.

It should be noted that system calls can also be called internally by the kernel using the `Zw` prefixed functions[4]. For instance, in creating a VDM (16-bit) process on 32-bit versions of Windows, the kernel maps the null page by calling `ZwAllocateVirtualMemory` (with a `BaseAddress` parameter of 1) to support the 16-bit execution environment. Moreover, memory management functions exported by the kernel may allow other kernel components such as third party drivers to map the null page, e.g. via calls to `MmMapViewofSection`.

The biggest drawback of using system call hooking is that the mitigation would not be introduced in a centralized way; hence any missing check could result in a bypass. Moreover, hooking system calls is discouraged by Microsoft and could very well introduce vulnerabilities on its own [4]. For instance, failing to properly capture arguments (such as the `BaseAddress` pointer) could lead to race condition issues between the hook and the real implementation and allow an attacker to bypass the mitigation. Hooking also becomes measurably harder on 64-bit platforms due to the integrity checks enforced by Kernel Patch Protection (aka PatchGuard) [3].

4.2 PTE Modification

In Section 3, we briefly discussed page table entries and their importance in virtual to physical address translation. As PTEs hold several bits describing the state and permissions of a page, PTE modification can also provide us with some useful results. Specifically, in leveraging the U/S bit of a page table entry, user-mode processes can be restricted from accessing certain pages in user-mode memory, such as the null page. When the U/S bit is cleared, the memory access fault handler (`nt!MmAccessFault`) sees that the page is owned by the kernel, hence returns an access violation if the fault originated from user-mode. Moreover, as the kernel also allows user-mode pages to be secured in memory (e.g. using `MmSecureVirtualMemory`[5]), users may not always be allowed to unmap or free a mapped page. This is also how special regions of memory such as the `KUSER_SHARED_DATA` section and thread and process environment blocks are mapped into a user-mode process. The drawback of this approach is that the null page, although inaccessible to users, will still be accessible from the kernel.

[4] In updating the `PreviousMode`, these functions let the kernel bypass checks typically enforced on user provided buffers.

[5] http://msdn.microsoft.com/en-us/library/ff556374(v=vs.85).aspx

This could very well lead to exploitable conditions depending on how the kernel uses the null page data.

4.3 VAD Manipulation

Having looked at both system call hooking and page table entries, we turn to VADs. As these structures are used to describe the process memory space in Windows, it becomes an ideal candidate for preventing null page mappings. Our goal in this respect is to create an entry that prevents any attempt at null page allocation. It is also important that we work within the confines of the memory manager, as any low-level kernel modification could potentially affect system stability and yield unpredictable results. In order to achieve our goal, we will study how PEBs and TEBs are created, as these special structures use specifically crafted VADs to remain secured in memory. This is important because any private allocation made in user-space (regardless of the U/S bit) can be deleted by the process owner, unless secured in this way. We also want to avoid using the `MmSecureVirtualMemory` API exposed by the kernel as it requires the pages to be present, hence defeats the purpose of preventing null page mappings.

Specifically, the function we are interested in is `nt!MiCreatePebOrTeb`. With some help from ReactOS[6] (or the WRK[7]), we can make out how the VAD entry is created.

```
NTSTATUS
NTAPI
MiCreatePebOrTeb(  IN PEPROCESS Process,
                   IN ULONG Size,
                   OUT PULONG_PTR Base )
PMMVAD_LONG Vad;

/* ... */

/* Allocate a VAD */
Vad = ExAllocatePoolWithTag(NonPagedPool, sizeof(MMVAD_LONG),
     'ldaV' );

if (!Vad) return STATUS_NO_MEMORY;

/* Setup the primary flags with the size, and make it
    commited, private, RW */
Vad->u.LongFlags = 0;
Vad->u.VadFlags.CommitCharge = BYTES_TO_PAGES(Size);
Vad->u.VadFlags.MemCommit = TRUE;
Vad->u.VadFlags.PrivateMemory = TRUE;
Vad->u.VadFlags.Protection = MM_READWRITE;
Vad->u.VadFlags.NoChange = TRUE;
```

[6] http://www.reactos.org/
[7] http://www.microsoft.com/resources/sharedsource/windowsacademic/researchkernelkit.mspx

```
/* Setup the secondary flags to make it a secured, writable,
    long VAD */
Vad->u2.LongFlags2 = 0;
Vad->u2.VadFlags2.OneSecured = TRUE;
Vad->u2.VadFlags2.LongVad = TRUE;
Vad->u2.VadFlags2.ReadOnly = FALSE;

/* Validate VAD range */
/* ... */

/* Build the rest of the VAD now */
Vad->StartingVpn = (*Base) >> PAGE_SHIFT;
Vad->EndingVpn = ((*Base) + Size - 1) >> PAGE_SHIFT;
Vad->u3.Secured.StartVpn = *Base;
Vad->u3.Secured.EndVpn = (Vad->EndingVpn << PAGE_SHIFT) | (
    PAGE_SIZE - 1);
Vad->u1.Parent = NULL;
```

Listing 1. Function creating VAD for PEB/TEBs

The key to preventing deletion in Listing 1 is the `NoChange` flag as well as the `OneSecured` flag used to secure the address range and prevent changes to the memory protection. In order to further lock down the VAD and deny read and write access, we also need to set the `Protection` flag to `MM_NOACCESS` (0x18). This protection is essentially equivalent to a guard page, but is not cleared on access. Additionally, as we don't want the memory manager to think that any memory is actually committed (this will actually allow `NtProtectVirtualMemory` to alter the page protection), we need to set the `MemCommit` flag to `FALSE`.

Although this gets us well on our way, there is still a fundamental issue we need to address. It turns out that a user could still commit memory for a range that has been reserved, hence be able to map the null page (create a valid PTE) and circumvent the mitigation. Fortunately, the memory manager has a special flag that prevents memory from being committed. If `VadFlags.CommitCharge` is set to `MM_MAX_COMMIT` (0x7ffff on x86 or 0x7fffffffffff on x64), any attempt at committing memory in the range will result in a `STATUS_CONFLICTING_ADDRESSES` error. This effectively allows us to create a custom VAD that denies access to the null page from both user and kernel-space.

5 Implementation

In order to implement the method described in Section 4.3, we create a kernel-mode driver and add a process-creation callback routine using the exported `PsSetCreateProcessNotifyRoutine` function[8]. The callback is invoked before any threads are executed in a process (from within `nt!PspInsertThread`) and

[8] http://msdn.microsoft.com/en-us/library/ff559951(v=vs.85).aspx

before the process terminates in `nt!PspExitProcess`. Within the callback, we manipulate the VAD tree by inserting a crafted VAD entry to prevent subsequent code from mapping the null page.

Before we can insert a VAD into the VAD tree, we need to check if the null page has already been mapped by the process before the process-creation callback was invoked. This may happen in 16-bit processes where the null page is used to store information specific to the 16-bit execution environment, such as the interrupt vector table at address 0000h. The null page allocated by `nt!PspSetupUserProcessAddressSpace` is using `ZwAllocateVirtualMemory`, hence we could simply check for the presence of this allocation in our callback and free it if needed. In turn, this would allow us to insert our crafted VAD entry. Inspired by the PEB/TEB VAD, we initialize the crafted VAD as shown in Listing 2.

```
Vad = (PMMVAD_LONG) ExAllocatePoolWithTag(NonPagedPool,
    sizeof(MMVAD_LONG), 'ldaV');

// we probably don't want this to happen (terminate process?)
if (Vad == NULL)
   goto ExitCleanup;

RtlZeroMemory(Vad,sizeof(MMVAD_LONG));

Vad->StartingVpn = 0 >> PAGE_SHIFT;
Vad->EndingVpn = 0xffff >> PAGE_SHIFT;

Vad->u.VadFlags.CommitCharge = MM_MAX_COMMIT; // special
    value to prevent commit!
Vad->u.VadFlags.MemCommit = FALSE;
Vad->u.VadFlags.PrivateMemory = TRUE;
Vad->u.VadFlags.Protection = MM_NOACCESS;
Vad->u.VadFlags.NoChange = TRUE; // mark as non-deleteable

Vad->u2.VadFlags2.OneSecured = TRUE;
Vad->u2.VadFlags2.LongVad = TRUE;

Vad->u3.Secured.u1.StartVa = 0;
Vad->u3.Secured.EndVa = (PVOID) 0xffff;

// insert into process VAD tree
InsertVad(Vad,Process);
```

Listing 2. Function crafting custom VAD to prevent null page mapping

`InsertVad()` inserts the VAD into the AVL tree (at the leftmost branch) and increments the `MM_AVL_TABLE.NumberGenericTableElements` counter representing the number of VAD entries in the AVL tree. This function is functionally equivalent to `MiInsertVad()` in the NT executive, but doesn't include the

rebalancing algorithm[9] as this is performed by the kernel on subsequent VAD insertions. When the process terminates, `nt!MmCleanProcessAddressSpace` iterates over all the VAD entries and frees them as required, hence no special care is needed for cleaning up the crafted VAD. In the output in Example 1, we have inserted a crafted VAD entry that prevents users from accessing memory below 0x10000.

```
kd> !process fffffa800d122060
PROCESS fffffa800d122060
    SessionId: 1  Cid: 063c    Peb: 7fffffd5000  ParentCid: 05ac
    DirBase: 322ec000  ObjectTable: fffff8a006b2e890  HandleCount:  71.
    Image: calc.exe
    VadRoot fffffa800d4b88c0 Vads 154 Clone 0 Private 1328. Modified 0.

kd> !vad fffffa800d4b88c0
VAD             level      start      end    commit
fffffa800d478f70 ( 7)          0        f        -1 Private      NO_ACCESS
fffffa800d2131b0 ( 6)         10       1f         0 Mapped       READWRITE
fffffa800d3ec160 ( 7)         20       22         0 Mapped       READONLY
fffffa800d1b5360 ( 5)         30       33         0 Mapped       READONLY
...

kd> !pte 0
                            VA 0000000000000000
... PDE at FFFFF6FB40000000    PTE at FFFFF68000000000
... contains 0090000000B84867  contains 0000000000000000
... pfn b84       ---DA--UWEV  not valid
```

Example 1: WinDbg output showing the crafted VAD entry in the AVL tree

6 Discussion

In order to circumvent the proposed mitigation, an attacker must find a way to map the null page before the process creation callback, be able to remove or modify the VAD entry, or find a code path that does not honor the `NO_ACCESS` restriction. In the former, even if the attacker somehow managed to load the executable image at virtual address null (or below 0x10000), the driver could still check for the presence of this memory mapping and terminate the affected process on notice. However, it is worth noting that creating the logic necessary to perform such verification may have slight impact on performance.

[9] AVL trees rebalance after each insertion or deletion to optimize subsequent lookups.

Function	Addr	Type	Protection	Result
NtAllocateVirtualMemory	1	MEM_RESERVE	READONLY	0xC0000018
NtAllocateVirtualMemory	1	MEM_COMMIT	READONLY	0xC0000018
NtMapViewOfSection	1	MEM_DOS_LIM	READONLY	0xC0000018
NtProtectVirtualMemory	0		READWRITE	0xC000002D
NtProtectVirtualMemory	0		READONLY	0xC0000045
NtFreeVirtualMemory	0	MEM_RELEASE		0xC0000045

Table 1. Results of null page operations on Windows 7

In order to test the robustness of the implementation, a tool was developed to perform several test cases. Specifically, paths to functions that operate on virtual memory were exercised extensively to ensure that the introduced modifications did not affect system stability or cause any unexpected behavior. Moreover, in order to attempt to circumvent the mitigation, test cases were targeted on functions that alter the state of VAD and PTE entries. If the crafted VAD entry could somehow be removed using legitimate system calls, an attacker could easily gain control of the null page. The results of these tests are summarized in Table 1, and conclude that an attacker cannot modify the VAD though any traditional means.

One disadvantage of the proposed mitigation is having to rely on fixed offsets in parsing the VAD tree. Specifically, the `VadRoot` pointer in the process object structure (`nt!_EPROCESS`) has a tendency to move around between versions of Windows. Moreover, internal structures such as `MMVAD` and `MMADDRESS_NODE` seem to have been updated slightly between versions 5.x (2000/XP/2003) and 6.x (Vista and later). Unfortunately, there's no easy way of addressing this problem other than maintaining a database of offsets as none of the internal memory management APIs (Mi*) for operating on VADs and address nodes are exposed by the NT executive.

Another disadvantage is that applications and legacy components that currently rely on null page access are likely to break. In the VDM subsystem case, 16-bit applications fail to initialize properly on execution. However, aside from this minor inconvenience, no significant side effects have been observed while testing the proposed mitigation on various system configuration. Due to the significant number of exploitable `NULL` pointer vulnerabilities recently affecting Windows kernel components [8][2], the added protection provided by the proposed mitigation is believed to be far more important to the average user than maintaining backwards compatibility. It should also be noted that the VDM subsystem is not present on 64-bit Windows, hence the introduced changes will have a less noticeable impact on such systems.

One of the big concerns in employing new exploit mitigations is the potential impact on performance. In our case, the proposed mitigation only introduces minor changes at process creation-time, hence has very little impact on performance.

7 Future Work

There is still much work to be done in order to address kernel exploitation on Windows. Although future versions of the operating system are likely to harden frequently targeted areas such as the kernel pool [6][7] and add support for recent hardware mitigations such as SMEP [11][5], an attacker will still have a lot of opportunity at hand in exploiting kernel vulnerabilities. As privilege escalation attacks require knowledge about the kernel address space to be successful, randomizing modules and limiting information accessible to users is key to mitigating kernel exploitation. Unfortunately, the Intel architecture was never designed to provide the clear separation between user and kernel-mode that is needed to fully address this problem. However, as mitigations make exploitation considerably harder, continued work in this field will make the risk less severe over time.

Acknowledgements

Thanks to Thomas Garnier and Dan Rosenberg for providing valuable feedback and helping out with the paper.

8 Conclusion

In this paper, we have proposed a way to generically mitigate exploitation of `NULL` pointer vulnerabilities in Windows by restricting access to the lower portion of process memory using VAD manipulation. Importantly, as the proposed method employs features already present in the memory manager and does not introduce any offending hooks, it can be introduced on a wide range of Windows platforms without significant portability or compatibility issues. Additionally, because the mitigation only introduces minor changes at process creation-time, the performance cost is minimal.

References

[1] Gynvael Coldwind: Why NULL points to 0? `http://gynvael.coldwind.pl/?id=399`
[2] Nicolas A. Economou: MS10-048: Win32k Window Creation Vulnerability (CVE-2010-1897) `http://www.ekoparty.org/archive/2010/ekoparty_2010-Economou-2x1_Microsoft_Bug.pdf`
[3] Scott Field: An Introduction to Kernel Patch Protection. `http://blogs.msdn.com/b/windowsvistasecurity/archive/2006/08/11/695993.aspx`
[4] Ken Johnson: Why hooking system services is more difficult (and difficult) than it looks. `http://www.nynaeve.net/?p=210`
[5] Mateusz Jurczyk, Gynvael Coldwind: SMEP: What is it, and how to beat it on Windows. `http://j00ru.vexillium.org/?p=783`
[6] Kostya Kortchinsky: Real World Kernel Pool Exploitation. SyScan 2008.

[7] Tarjei Mandt: Kernel Pool Exploitation on Windows 7. Black Hat Briefings DC 2011. https://media.blackhat.com/bh-dc-11/Mandt/BlackHat_DC_2011_Mandt_kernelpool-wp.pdf

[8] Microsoft Security Bulletin MS11-034: Vulnerabilities in Windows Kernel-Mode Drivers Could Allow Elevation of Privilege. http://www.microsoft.com/technet/security/bulletin/ms11-034.mspx

[9] Tavis Ormandy: install_special_mapping skips security_file_mmap check. http://article.gmane.org/gmane.linux.kernel/1074552

[10] Eric Paris: VM/Security: add security hook to do_brk. http://lkml.indiana.edu/hypermail/linux/kernel/0712.0/0874.html

[11] Dan Rosenberg: SMEP: What is It, and How to Beat It on Linux. http://vulnfactory.org/blog/2011/06/05/smep-what-is-it-and-how-to-beat-it-on-linux/

[12] Fermin J. Serna, Andrew Roths: Enhanced Mitigation Experience Toolkit 2.0. http://technet.microsoft.com/en-us/security/video/gg469855

[13] Brad Spender: UDEREF. http://grsecurity.net/~spender/uderef.txt

[14] Julien Tinnes: Bypassing Linux' NULL pointer dereference exploit prevention (mmap_min_addr) http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html