The story of CVE-2011-2018 exploitation

Mateusz "j00ru" Jurczyk

February - April 2012

Abstract

Exploitation of Windows kernel vulnerabilities is recently drawing more and more attention, as observed in both monthly Microsoft advisories and technical talks presented on public security events. One of the most recent security flaws fixed in the Windows kernel was CVE-2011-2018[1], a vulnerability which could potentially allow a local attacker to execute arbitrary code with system privileges. The problem affected all - and only - 32-bit editions of the Windows NT-family line, up to Windows 8 Developer Preview[2]. In this article, I present how certain novel exploitation techniques can be used on different Windows platforms to reach an elevation of privileges through this specific kernel vulnerability.

1 General information

Although the original name assigned by Microsoft might imply that the vulnerability was directly related to exception handling and the vulnerability FAQ refers to some kind of objects, I consider the information to be rather misleading, as the bug doesn't have anything to do with Windows Object Manager or any other type of objects in the common meaning. Alike, exception handling is only one of the affected mechanisms, while the bug resides in a completely different part of the kernel – a generic dispatcher of transitions between user- and kernel-mode.

Due to the nature of the vulnerability, which is strictly related to custom *Local Descriptor Table* entries only possible to be created locally (through the *NtSetLdtEntries* or *NtSetInformationThread* system services), I believe the scope of the bug is limited to local attacks. Considering that the X86-64 architecture almost entirely abandons the usage of segments, 64-bit Windows editions are not affected by the bug by definition.

As a matter of fact, the issue was found accidentally during the development of a CrackMe program with Gynvael Coldwind. The project was an entry to the *Pimp My CrackMe* competition [2], and in itself was meant to become a Proof

[1]The vulnerability was officially titled "Windows Kernel Exception Handler Vulnerability" in the Microsoft Security Bulletin.

[2]Windows 8 Developer Preview was released on September 13th, 2011, roughly three months before the official patch release date.

of Concept presenting how IA-32 segmentation could be used for the purpose of execution flow obfuscation. Interestingly, the application began to crash my Windows Vista machine at early stages of the project development. After the contest finished, I started to investigate the crash dumps and soon found out that the manifested kernel bug was exploitable on all modern NT-family operating systems. This paper attempts to document the efforts I originally made to create a reliable exploit for the Windows XP and Windows Vista / 7 platforms.

2 Initial crash

The concept presented in the *Pimp CrackMe* challenge relied on creating numerous ring-3 code segments in a per-process LDT structure. According to experimental tests performed with the most commonly used debugging software, making extensive use of IA-32 segmentation might cause substantial difficulty during run-time analysis of the target program's execution flow [6]. I believe this phenomenon is primarily motivated by the fact that even though custom segments are still present and supported by CPU vendors, they are almost never observed in common software[3], as the popular flat memory model meets most requirements of a modern execution environment. Detailed information on creating custom LDT entries on Windows has been publicly available since early years of the last decade [11].

Our CrackMe implemented a simplistic virtual machine supporting around 10 instructions with a trivial CPU context and encoding scheme. Every instruction handler had its own code segment assigned to it, so that each of them could be invoked through a far call instruction. Given n virtual instructions, I intuitively decided to use the $\{0, \cdots, n-1\}$ range of LDT indexes. Once the segment-switching code worked correctly, I began to randomly encounter Blue Screens of Death while running the program for testing purposes. Listing 1 presents an excerpt from the crash log generated upon the occurrence of an unexpected bugcheck.

Listing 1: Initial system crash

```
TRAP_FRAME:  f572acf0 -- (.trap 0xfffffffff572acf0)
ErrCode = 00000002
eax=c0000005 ebx=fffffff4 ecx=00010101 edx=fffffff esi=00000202 edi=f572ad20
eip=8053d861 esp=f572ad64 ebp=f572ad64 iopl=0         nv up di ng nz ac po cy
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010093
nt!KiSystemCallExit2+0x84:
8053d861 897308          mov     dword ptr [ebx+8],esi ds:0023:fffffffc=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 804f7bad to 80527c0c

STACK_TEXT:
```

[3] There are several exceptions to the rule, such as the Google Chrome NaCl project, which uses segmentation to facilitate its security model on 32-bit platforms.

```
f572a82c 804f7bad 00000003 fffffffc 00000000 nt!RtlpBreakWithStatusInstruction
f572a878 804f879a 00000003 00000000 c07ffff8 nt!KiBugCheckDebugBreak+0x19
f572ac58 804f8cc5 00000050 fffffffc 00000001 nt!KeBugCheck2+0x574
f572ac78 8051cc7f 00000050 fffffffc 00000001 nt!KeBugCheckEx+0x1b
f572acd8 805405d4 00000001 fffffffc 00000000 nt!MmAccessFault+0x8e7
f572acd8 8053d861 00000001 fffffffc 00000000 nt!KiTrap0E+0xcc
f572ad64 00000005 badb0d00 00000101 00000000 nt!KiSystemCallExit2+0x84
WARNING: Frame IP not in any known module. Following frames may be wrong.
0022fef4 00000000 0000001b 77c50000 ffffffff 0x5
```

3 Vulnerability analysis

Windows trap frame is an internal structure responsible for the storage of various parts of the execution context such as general-purpose, debug and segment registers, flags or other information regarding the CPU state previous to an interrupt, exception or privilege switch. Although the structure is opaque and not officially documented, it is possible to obtain its definition with WinDbg and debug symbols available through Microsoft symbol server (`http://msdl.microsoft.com/download/symbols`). The structure used on a 32-bit version of Windows XP, Vista and 7 is presented in Listing 2.

Listing 2: KTRAP_FRAME structure definition

```
kd> dt _KTRAP_FRAME
nt!_KTRAP_FRAME
  +0x000 DbgEbp           : Uint4B \
  +0x004 DbgEip           : Uint4B |
  +0x008 DbgArgMark       : Uint4B |
  +0x00c DbgArgPointer    : Uint4B |
  +0x010 TempSegCs        : Uint4B |
  +0x014 TempEsp          : Uint4B |
  +0x018 Dr0              : Uint4B |
  +0x01c Dr1              : Uint4B |
  +0x020 Dr2              : Uint4B |
  +0x024 Dr3              : Uint4B |
  +0x028 Dr6              : Uint4B |
  +0x02c Dr7              : Uint4B |
  +0x030 SegGs            : Uint4B |
  +0x034 SegEs            : Uint4B | Initialized by Windows
  +0x038 SegDs            : Uint4B |
  +0x03c Edx              : Uint4B |
  +0x040 Ecx              : Uint4B |
  +0x044 Eax              : Uint4B |
  +0x048 PreviousPreviousMode : Uint4B
  +0x04c ExceptionList    : Ptr32 _EXCEPTION_REGISTRATION_RECORD
  +0x050 SegFs            : Uint4B |
  +0x054 Edi              : Uint4B |
  +0x058 Esi              : Uint4B |
  +0x05c Ebx              : Uint4B |
  +0x060 Ebp              : Uint4B /
  +0x064 ErrCode          : Uint4B > Initialized by CPU or Windows
  +0x068 Eip              : Uint4B \
  +0x06c SegCs            : Uint4B |
```

```
+0x070 EFlags           : Uint4B | Initialized by CPU
+0x074 HardwareEsp      : Uint4B |
+0x078 HardwareSegSs    : Uint4B /
+0x07c V86Es            : Uint4B \
+0x080 V86Ds            : Uint4B | Optionally initialized by CPU
+0x084 V86Fs            : Uint4B |
+0x088 V86Gs            : Uint4B /
```

The structure is formed on the kernel stack once an exception or interrupt is generated or delivered to the processor. After one of these conditions takes place, the CPU saves the most sensitive pieces of the execution context on the stack. The number of words pushed on the stack may differ, depending on whether a privilege switch was involved in the event (see Figures 1 and 2). More details on how IA-32 processors handle interrupts and exceptions can be found in "Intel 64 and IA-32 Architectures Software Developer's Manual", Volume 3A, section "Exception- or Interrupt-Handler Procedures" [4].

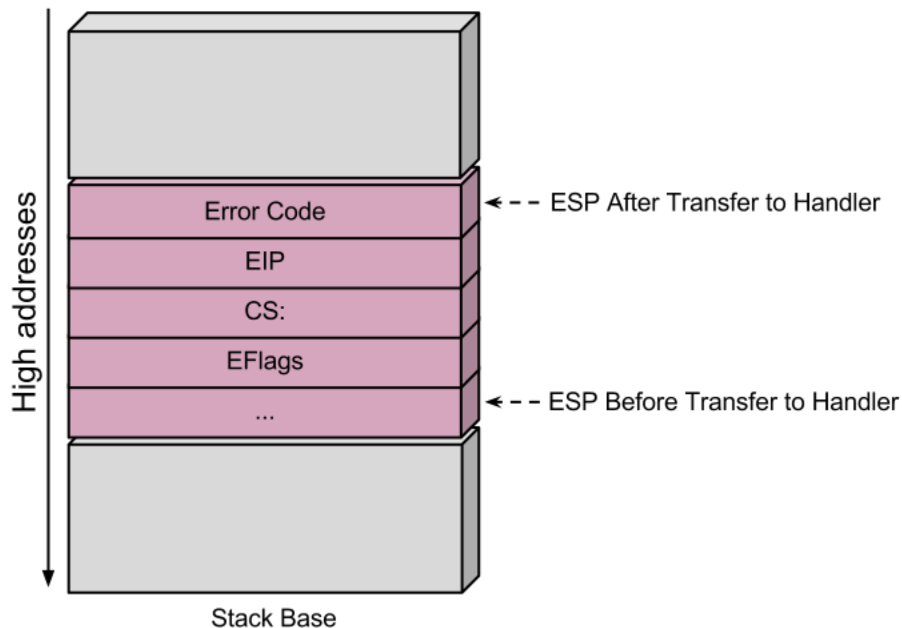

Figure 1: Stack Usage with no Privilege-Level Change

The upper part of the trap frame is filled by Windows, by manually pushing the registers and other context characteristics on the stack. The structure resides in memory thorough the execution of an interrupt handler, and is afterwards used to restore the original context of the interrupted task, so that the breakout from regular code execution is fully transparent to the underlying software.

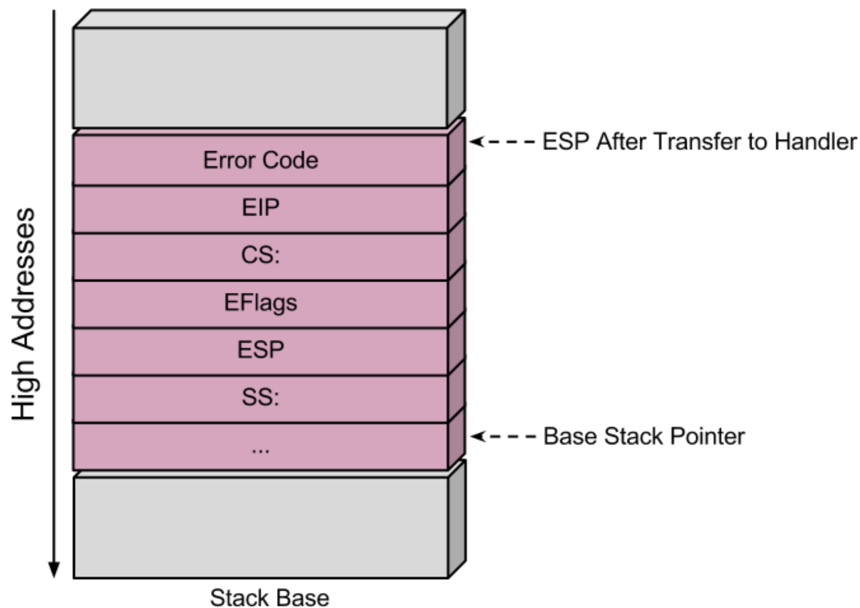

Figure 2: Stack Usage with a Privilege-Level Change

By just looking at the KTRAP_FRAME structure definition, one can deduce it can be used to store more information than just rough values of the processor registers. Specifically, the fields starting with "Dbg" and "Temp" prefixes (*DbgEbp*, *DbgArgMark*, *TempSegCs*, *TempEsp*) seem to be most interesting. As it turns out, the *SegCs* field not only serves as a container to back up the cs: selector, but is also occasionally used as a marker, indicating that the interrupt exit routine should use the *TempSegCs* / *TempEsp* pair instead of *SegCs* / *HardwareEsp* when returning to the previous task. Exemplary snippets of the Windows kernel code[4] making use of this specific *SegCs* property are shown in Listings 3 and 4.

Listing 3: Setting *SegCs* marker

```
VOID
KiEspToTrapFrame(
   IN PKTRAP_FRAME TrapFrame,
   IN ULONG Esp
   )

   (...)

       //
       //  Edit frame, setting edit marker as needed.
       //
```

[4]The presented code listings are part of the Windows Research Kernel project.

```
if ((TrapFrame->SegCs & FRAME_EDITED) == 0) {

    //  Kernel frame that has already been edited,
    //  store value in TempEsp.

    TrapFrame->TempEsp = Esp;

} else {

    //  Kernel frame for which Esp is being edited first time.
    //  Save real SegCs, set marked in SegCs, save Esp value.

    if (OldEsp != Esp) {

        TrapFrame->TempSegCs = TrapFrame->SegCs;
        TrapFrame->SegCs = TrapFrame->SegCs & ~FRAME_EDITED;
        TrapFrame->TempEsp = Esp;
    }
}
```

Listing 4: Examining *SegCs* against a marker while returning from interrupt

```
test    word ptr [esp]+TsSegCs,FRAME_EDITED
jz      b                           ; Edited frame pop out.

(...)
```

Considering that the numeric value of the FRAME_EDITED constant is defined as 0xFFF8, we get a clear picture of what is going on here. The kernel assumes that it is normally impossible to have *SegCs* inside a trap frame set to a value with the highest 13 bits cleared, and consequently uses such state to indicate the presence of some special condition. The structure of a segment selector on X86 platforms is presented in Figure 3.

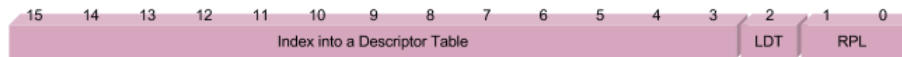

Figure 3: Intel X86 segment selector format

The Intel X86 manuals state that the first GDT entry (index=0) is architecturally reserved, as all segment selectors pointing at GDT[0] (i.e. with the high 14 bits cleared) are treated as special NUL selectors regardless of the first *Global Descriptor Table* entry contents. However, there is no corresponding rule in regards to *Local Descriptor Table*, hence making it feasible to set up a valid segment at LDT[0] and use it to execute code (i.e. with cs: set to a numeric value of 0007h).

As a consequence, it is possible to trick the kernel into thinking that *SegCs* value has a special, reserved meaning while it really is just a valid code selector.

Such effect can be achieved by creating an LDT entry with index=0, switching cs: and triggering a software interrupt (or waiting for a hardware one to occur). As shown in Listing 5, the kernel would then use *TempEsp* as a new stack pointer and execute an IRETD instruction with the *TempSegCs* value as its parameter. As we consider the fact that none of the fields are initialized prior to being mistakenly used, it becomes apparent that we have just faced a stack-based uninitialized variable reference vulnerability.

Listing 5: Using TempSegCs and TempEsp to set up a return frame

```
        jz      b                           ; Edited frame pop out.

        (...)

b:       mov     ebx,[esp]+TsTempSegCs
        mov     [esp]+TsSegCs,ebx

        (...)

        mov     ebx,[esp]+TsTempEsp
        sub     ebx,12
        mov     [esp]+TsErrCode,ebx

;
;   Copy eip,cs,eflags to new stack.  note we do this high to low
;

        mov     esi,[esp]+TsEflags
        mov     [ebx+8],esi
        mov     esi,[esp]+TsSegCs
        mov     [ebx+4],esi
        mov     esi,[esp]+TsEip
        mov     [ebx],esi
```

In almost all practical scenarios, neither *TempSegCs* nor *TempEsp* are ever filled with any data at all; the structure fields usually remain zero-ed out during the lifespan of a given process. This explains the appearance of the initial crash, including the attempt to write to the 0xfffffffc address (calculated as *TempEsp* - 4). In the current state, the flaw could only be used to trigger a Blue Screen of Death and crash the machine. Successful elevation-of-privileges exploitation relies on one's ability to control the values of *TempSegCs* and *TempEsp*; if it were possible, turning the security flaw into an Administrators command prompt would be a matter of writing the desired payload.

During the course of several weeks after encountering the first crash, I have developed methods to successfully exploit the issue on Windows XP SP3, and later on Windows Vista and 7; the latter part turned out to be considerably harder. Let's proceed to the juicy part.

4 Exploitation - initial notes

Given that the only possible way to exploit the flaw is to fill the two crucial fields in `KTRAP_FRAME` with non-zero (possibly controlled) values, I initially focused on looking for ways to achieve this goal. One of the most important characteristics of a trap frame is that it is almost always allocated at exactly the same place on the kernel stack. The underlying reason for this behavior is the management algorithm of the stack - when in user-mode, the kernel stack pointer is set to the top of the stack (or somewhere close to the top). Since the trap frame is the first structure allocated on the stack upon an interrupt, it is always mapped to the very same virtual address for a specific thread.

The main advantage of the above property is the fact that once filled, the values of uninitialized structure fields reside there for a really long time. On the other hand, this also means that it is not possible to write to the memory area assigned to the targeted fields in any way other than through an explicit reference to `KTRAP_FRAME`.

Personally, I was able to think of two potential approaches to the problem of controlling *TempSegCs* and *TempEsp*:

- Get the kernel to fill the fields legitimately (triggering the *SegCs*-marking kernel mechanism), and then re-use those values in a malicious way.
- Spray a region of the kernel stack below the trap frame with controlled data, and have the trap frame mapped to that lower area of the stack, so that *TempSegCs* and *TempEsp* are allocated in memory previously filled with arbitrary bytes.

As later turned out, the first idea was not applicable in real-life conditions, as the *SegCs*-marking mechanism could only be used on a trap frame describing kernel-mode code interruption, whereas our exploit would be only able to produce user-mode frames. On the other hand, the second concept proved to work on all modern Windows versions (although the technical details of how to accomplish it were different between them). Let's see how the task can be accomplished on a Windows XP / 2003 platform.

5 Windows XP exploitation

As mentioned in previous sections, the assembly presented in Listing 7 is executed after making a wrong assumption that the saved cs: selector has a a special meaning reserved only for kernel mode use-cases. The following trap frame fields are involved in the operation:

- *TsTempEsp*: Unitialized value,
- *TsErrCode*: Irrelevant, used to back up *TsTempEsp*,
- *TsEflags*: The original EFlags of the interrupted code,

- *TsSegCs*: Unitialized value,
- *TsEip*: The original Eip of the interrupted code.

As a result, having the two undefined fields initialized with valid values, the faulty `KiSystemCallExit2` (also known as `Kei386EoiHelper`) routine should be able to seamlessly return to the interrupted code, the only difference being a potentially modified cs: selector and Esp register.

During regular ring-3 thread execution, the kernel stack pointer points to a specific address, usually very close to the stack base. When a trap-frame is built, the original stack pointer is decremented by an adequate number of bytes[5]. The most common kernel stack layout observed during an interrupt or system call invocation is presented in Figure 4.

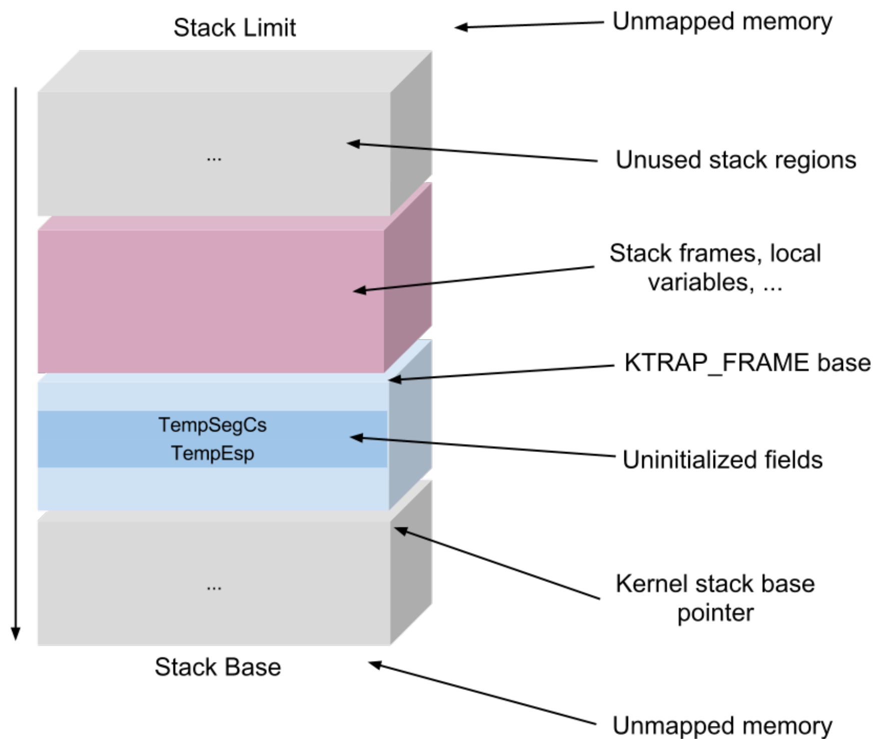

Figure 4: Typical kernel-mode stack layout

The ultimate objective is to move the kernel stack base pointer towards the bottom of the stack, so that the structure is remapped into better controlled memory areas. Let's find out about possible ways to do it.

[5]Usually 124 (7Ch) bytes, being a typical `KTRAP_FRAME` structure size.

5.1 Trap frame allocation

Shifting the kernel stack base address is definitely not something people do purposely on a daily basis. On the other hand, it turns out that the operation is an essential part of the GUI-process management in kernel mode. Specifically, the Windows kernel provides an undocumented functionality making it possible for `win32k.sys` and potentially other device drivers to "call-back" into user-mode. The exported kernel function implementing the feature is called `KeUserModeCallback`, and has been thoroughly examined and described by a Norwegian security researcher Tarjei Mandt, who showed that incorrect usage of the mechanism did lead to over 40 Privilege Escalation vulnerabilities in all Windows NT-family systems [9].

Every time a user-mode callback is invoked (which happens fairly frequently for every GUI thread), the kernel saves current context information (i.e. the kernel-mode return address) on the current stack and performs a return to the less-privileged execution mode. Since the callback return context consumes some memory at the top of the stack, respective interrupts invoked from within a nested user-mode callback result in having the new trap frame allocated in the lower portions of the stack (see Figure 5).

According to my personal experiments, the delta between the original and a post-callback stack base is around 2608 (0A30h) bytes[6]. As the callbacks can be used in a recursive fashion, it is possible to decrease the stack base by any multiplicity of that number by triggering an adequate number of nested callbacks. The mechanism itself works by returning to a constant `ntdll!KiUserModeCallbackDispatcher` function, which invokes the proper handler within user32.dll, based on a parameter passed through the user-mode stack (see Listing 6).

Listing 6: `KiUserCallbackDispatcher` assembly snippet

```
.text:7C90E440 ; __stdcall KiUserCallbackDispatcher(x, x, x)
.text:7C90E440                 public _KiUserCallbackDispatcher@12
.text:7C90E440 _KiUserCallbackDispatcher@12 proc near
.text:7C90E440                 add     esp, 4
.text:7C90E443                 pop     edx
.text:7C90E444                 mov     eax, large fs:18h
.text:7C90E44A                 mov     eax, [eax+30h]
.text:7C90E44D                 mov     eax, [eax+2Ch]
.text:7C90E450                 call    dword ptr [eax+edx*4]
.text:7C90E453                 xor     ecx, ecx
.text:7C90E455                 xor     edx, edx
.text:7C90E457                 int     2Bh
.text:7C90E459                 int     3
.text:7C90E45A                 mov     edi, edi
```

The routine obtains a list of the callback handlers from `[[fs:18]+30h]+2Ch`[7],

[6]The number includes initial trap frame, a local kernel-mode context and the user-mode callback return frame.

[7]The fs: segment register typically points to the Thread Environment Block structure, while fs:[18h] is supposed to store the address of the local Process Environment Block. The

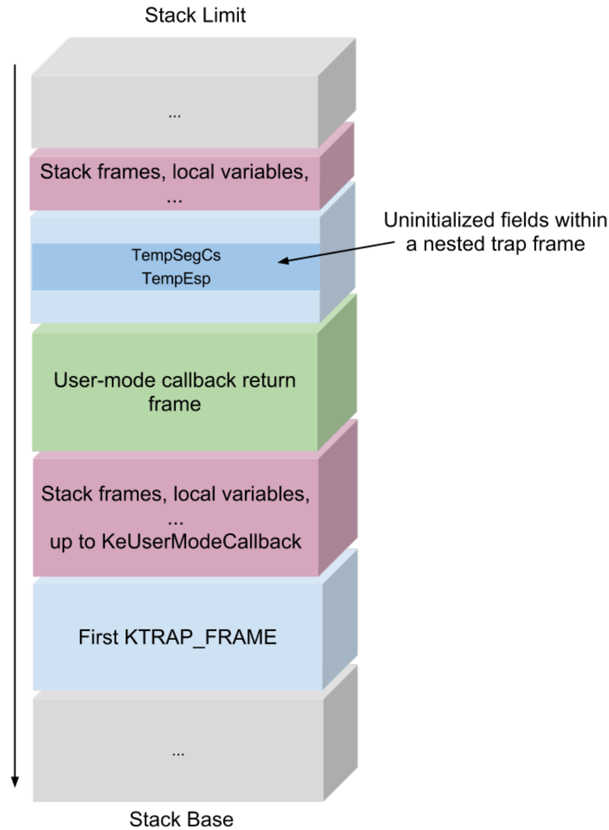

Figure 5: Kernel stack layout after invoking a nested interrupt

and invokes a corresponding routine. After the handler returns, the dispatcher uses interrupt 2Bh to resume kernel-mode execution. It is possible to *hijack* the user32.dll dispatch table and intercept the execution when a user-mode callback is triggered from ring-0 by replacing the default dispatch table pointer with a list of attacker-controlled functions. As a result of being able to execute arbitrary code in the context of a user-mode callback, we can easily craft trap frames at lower portions of the kernel stack.

Being able to move the trap frame around, the last remaining problem is how the kernel stack can be filled with controlled data, prior to mapping `KTRAP_FRAME` to that memory and having the kernel use the custom values as *TempSegCs* and *TempEsp*. An ideal solution would be to get a system service to copy some controlled bytes into a large enough local buffer stored on the stack. Since the delta between typical and callback-adjusted stack bases is

overall expression translates to `PEB.KernelCallbackTable`.

around 0A00h, it would be safe to control as much as 1000h (4kB, roughly one memory page) bytes of the stack.

As it turns out, the desired effect can be successfully achieved by taking advantage of the `nt!NtMapUserPhysicalPages` system service. The routine's internal stack frame is 1100h bytes large, primarily influenced by a local array of 400h items of type `ULONG`. The function prologue is presented in Listing 7.

Listing 7: `nt!NtMapUserPhysicalPages` syscall prologue

```
...

#define COPY_STACK_SIZE             1024

...

NTSTATUS
NtMapUserPhysicalPages (
    __in PVOID VirtualAddress,
    __in ULONG_PTR NumberOfPages,
    __in_ecount_opt(NumberOfPages) PULONG_PTR UserPfnArray
    )

...

   ULONG_PTR StackArray[COPY_STACK_SIZE];

...

   PoolArea = (PVOID)&StackArray[0];

...

       if (NumberOfPages > COPY_STACK_SIZE) {
           PoolArea = ExAllocatePoolWithTag (NonPagedPool,
                                             NumberOfBytes,
                                             'wRmM');

           if (PoolArea == NULL) {
               return STATUS_INSUFFICIENT_RESOURCES;
           }
       }

       //
       // Capture the specified page frame numbers.
       //

       Status = MiCaptureUlongPtrArray (PoolArea,
                                        UserPfnArray,
                                        NumberOfPages);

...
```

As the listing shows, the service is capable of copying up to 4096 user-controlled bytes into a local buffer. When called with specially crafted parameters, this behavior allows an attacker to entirely cover a `KTRAP_FRAME` structure

(which can be later allocated within the boundaries of the local buffer) and consequently control all uninitialized fields therein. For a more detailed description of the spraying technique, see ”nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques” [5].

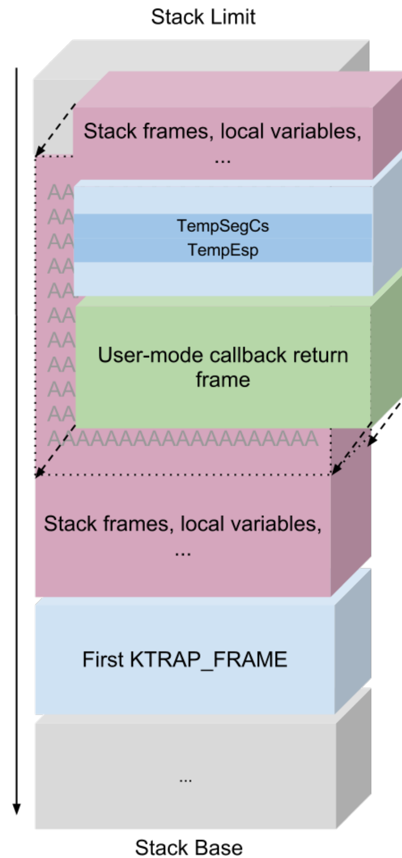

Figure 6: Stack spraying illustrated

To sum up, the following steps need to be performed in order to complete the first exploitation stage:

1. Load user32.dll
2. Hook the user32.dll callback table using a PEB array pointer
3. Call `nt!NtMapUserPhysicalPages` to spray 4kB of kernel stack with arbitrary data
4. Trigger a user-mode callback (e.g. through a `MessageBox` API call)

... from within intercepted callback handler:

6. Create a code segment at index=0 in *Local Descriptor Table*
7. Trigger the vulnerability through a jump into cs:=7

Interestingly, Step 6 can be alternatively achieved with three lines of assembly shown in Listing 8. During the execution of such an expensive loop, a hardware interrupt will likely occur in the context of the thread, having the same effect as directly invoking a software interrupt.

Listing 8: Loop waiting for an elevated CPL

```
@@:
  mov ax, cs
  and ax, 3
  jnz @@
```

After spraying the stack with a block of 41414141 values and performing the rest of the outlined steps, one should be able to achieve the effect presented in Listing 9.

Listing 9: A result of triggering CVE-2011-2018 with a sprayed stack

```
FAULTING_IP:
nt!KiSystemCallExit2+84
8053d861 897308          mov     dword ptr [ebx+8],esi

TRAP_FRAME:  f5deb2c0 -- (.trap 0xfffffffff5deb2c0)
ErrCode = 00000002
eax=c0000005 ebx=41414135 ecx=00010101 edx=f5deb634 esi=00000202 edi=f5deb2f0
eip=8053d861 esp=f5deb334 ebp=f5deb334 iopl=0         nv up di pl nz ac pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010016
nt!KiSystemCallExit2+0x84:
8053d861 897308          mov     dword ptr [ebx+8],esi ds:0023:4141413d=????????
Resetting default scope
```

5.2 What's next?

Controlling the *TempSegCs* and *TempEsp* fields enables us to get the kernel to create the following return frame at a chosen virtual memory address and invoke an IRETD instruction:

```
+0x00 Eip from the original trap frame
+0x04 TempSegCs (controlled)
+0x08 EFlags from the original trap frame
```

In other words, the kernel will attempt to return to the previous execution context, only difference being a fully controlled cs: selector. In order to perform an elevation of privileges, we need to point it to a code segment with RPL=0 and DPL=0. The only available option is to use the default kernel-mode code

segment, initialized in GDT[1] and represented by cs:=0008h (index=1, ldt=0, rpl=0).

Notably, the `KiSystemCallExit2` routine executes with the Interrupt Request Level (IRQL) equal to `DISPATCH_LEVEL`, thus pointing *TempEsp* to a pageable memory region (for example, user-mode area) might and likely would cause a `IRQL_NOT_LESS_OR_EQUAL` bugcheck. Consequently, it is required to find a non-pageable and writable memory (e.g. *NonPaged* pool or part of a device driver's image) within the kernel virtual address space, to use it for the fake exit frame storage. Neither of those address types are hard to obtain, thanks to numerous kernel communication channels revealing lots of information regarding the ring-0 address space [7]. Due to my personal preferences, I chose to use a non-pageable region of the ntoskrnl.exe executable image.

Furthermore, since the user-mode callback stack delta can be potentially subject to future modifications, it would be most desirable to build an offset-resilient exploit. As the only two fields initialized through stack spraying are *TempSegCs* and *TempEsp*, setting them both to a valid kernel stack pointer ending with 0008h prevents the exploit from failing upon different offsets. The technique works only due to the IRETD instruction implementation - given a `xxxx0008` parameter as the target code selector, it will always ignore the upper 16 bits of the argument.

For testing purposes, I decided to use the exported `nt!HalDispatchTable` symbol to calculate the final 32-bit spraying operand:

```
(&HalDispatchTable & 0FFFF0000h) + 0008h
```

After filling the kernel stack with the above DWORD value and having the bug triggered, we should expect the kernel to return back to the previous execution address, only difference being the newly acquired ring-0 privileges - note the cs: register value in Listing 10.

Listing 10: Payload running with escalated ring-0 privileges

```
kd> r
eax=67500000 ebx=0120e4c4 ecx=675135a8 edx=00000001 esi=92f7bdb0 edi=67501b9b
eip=010e000e esp=badb0d00 ebp=0120e4d0 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
010e000e cc              int     3

kd> u
010e0000 bbc4e42001      mov     ebx,120E4C4h
010e0005 668cc8          mov     ax,cs
010e0008 66250300        and     ax,3
010e000c 75f7            jne     010e0005
010e000e cc              int     3
010e000f 0f20c2          mov     edx,cr0
010e0012 81e2fffffeff    and     edx,0FFFEFFFFh
010e0018 0f22c2          mov     cr0,edx
```

5.3 Writing a kernel-mode payload

Although the primary goal of escalating code execution privileges to ring-0 has been accomplished, it is still required to fix the broken operating system state and use the acquired rights to fully compromise the system in a clean fashion (e.g. load a custom kernel device driver, or create a command shell with `NT AUTHORITY\SYSTEM` privileges).

In order to reliably execute ring-0 payload, the exploit will take the following steps after returning from the faulty `KiSystemCallExit2`:

1. Overwrite the `nt!HalDispatchTable+4` function pointer with a user-mode shellcode address,
2. Perform a regular kernel-to-user return using a minimal trap frame set up on the kernel stack.

After that, we should end up with a stable operating system state and a redirected kernel-mode pointer, which can be invoked via the `NtQueryInterval-Profile` service at any convenient time [8]. A pseudo-code of an exemplary stage-one assembly payload is shown in Listing 11.

Listing 11: Stage-one payload pseudo-code

```
While (SegCs & 3) != 0:
  Nop;

Turn off memory protection through CR0;

[nt!HalDispatchTable + 4] = &Stage2Payload;

Push the following values on kernel-mode stack:
  +0x00: 0x0023 (KGDT_R3_DATA, data segment selector)
  +0x04: Address of user-mode stack
  +0x08: 0x001B (KGDT_R3_CODE, code segment selector)
  +0x0C: Address of user-mode routine

Restore memory protection through CR0;

Invoke IRETD;
```

Having an opportunity to execute a high-level function as stage-two payload, we can implement the routine to make use of the documented kernel API interface. The approach guarantees correct performance of the code on all modern Windows editions, and doesnt put the attacker at risk of using obscure solutions (such as relying on `EPROCESS` structure offsets). The pseudo-code of a payload elevating the privileges of a chosen process can be found in Listing 12.

Listing 12: Exemplary stage-two payload pseudo-code

```
Open handle to a process with PID=4 (SYSTEM process) via ZwOpenProcess;

Open the process' security token via ZwOpenProcessToken;
```

```
Duplicate the token via ZwDuplicateToken;

Assign the token to a chosen process (e.g. GetCurrentProcess()) via
ZwSetInformationProcess;
```

Addresses of the required kernel API functions referenced in the payload can be easily obtained from within user-mode, by making use of the `LoadLibrary` and `GetProcAddress` APIs, and pieces of information revealed by `EnumDeviceDrivers`. More information regarding the implementation of a custom `GetKernelProcAddress` function can be found in the "Windows Security Hardening Through Kernel Address Protection" article [7].

When all of the discussed steps are successfully completed, one should be able to see the result shown in Figure 7. That's it for Windows XP.

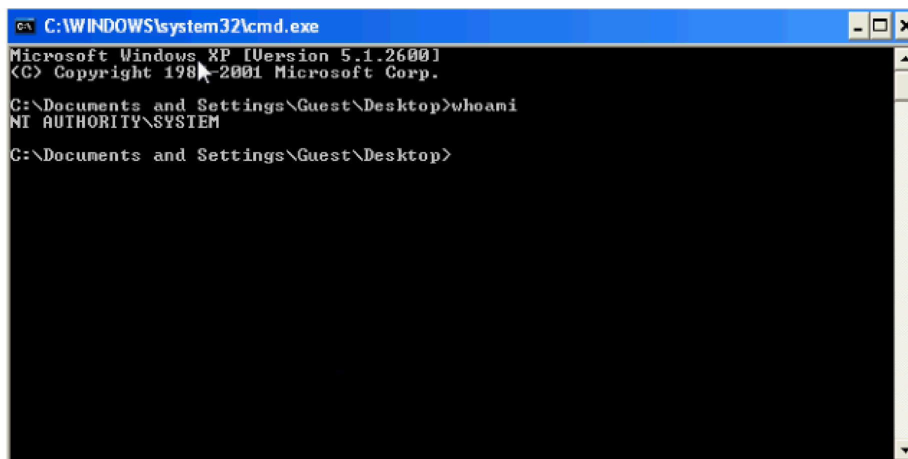

Figure 7: Result of successful exploitation on the Windows XP platform.

6 Windows Vista / 7 exploitation

Beginning with Windows Vista and 2008, Microsoft introduced fundamental changes in how user-mode callbacks work internally. In the previous system editions, context information regarding all recursive callbacks was stored in the scope of a single kernel stack, allowing user-mode applications to manipulate the location of the trap frame. As previously discussed, the latter behavior was the key to successful exploitation of the considered vulnerability.

Newer operating systems no longer use a single stack for multiple callbacks. Instead, every time a user-mode callback is invoked, a completely new kernel stack is spawned and the base stack pointer is moved to the top of the new memory area. The overall functionality is implemented by an internal `KiMigrateToNewKernelStack` routine, as shown in Listing 13.

Listing 13: New user-mode calback implementation

```
.text:00465738 ; __stdcall KiCallUserMode(x, x, x)
.text:00465738 _KiCallUserMode@12 proc near
.text:00465738
.text:00465738
.text:00465738 var_18          = byte ptr -18h
.text:00465738 arg_8           = dword ptr  0Ch
.text:00465738
.text:00465738                 push    ebp
.text:00465739                 push    ebx

(...)

.text:00465761                 mov     ecx, [esp+10h+arg_8]
.text:00465765                 xor     edx, edx
.text:00465767                 lea     eax, [esp+10h+var_18]
.text:0046576B                 push    eax
.text:0046576C                 call    @KiMigrateToNewKernelStack@12
```

Unfortunately, this simple change renders our previous technique completely useless in the context of the affected systems, since it prevents us from controlling the *TempSegCs* and *TempEsp* fields, again. In order to escalate privileges on Windows Vista or 7, the only way around is to come up with another way of shifting the stack base address to achieve a trap frame mapping different from the default one. At first, I believed that the problem was hopeless; it took over two months to realize there might be a way to turn the security flaw into a privilege escalation; the concept is, however, incomparably more complex than in Windows XP.

6.1 Segment update faults

Whenever user- or kernel- code attempts to modify one of the six segment registers, the CPU performs basic verification to ensure that the operation makes sense (i.e. the target selector points to a valid GDT/LDT entry) and is allowed from a security perspective. In case a failure occurs while loading a new segment selector into a register, the CPU generates Interrupt 11 - Segment Not Present (#NP)[8]. This fact is going to be particularly useful later in the paper.

As a matter of fact, the Windows kernel often loads cs:, ds: and other segment registers on behalf of user-mode code; three notable examples of this behavior are listed below:

1. The usage of `SetThreadContext` documented API results in having the `CONTEXT` structure fields copied into a remote thread's trap frame and later loaded to actual registers.

2. The usage of the undocumented `NtContinue` service has the same effect, but it only affects the context of the current thread.

[8]One exception to the rule is the ss: register, which has its own Stack Faults (#SS) exception.

3. Windows VDM (Virtual DOS Machine) - in order to invoke execution of arbitrary 16-bit code in a controlled environment, it is required to call the `NtVdmControl(VdmStartExecution)` service from within the NTVDM.EXE subsystem process, which also results in having the CPU context loaded from a pre-defined location in *Process Environment Block*.

Since the `KiSystemCallExit2` routine doesn't perform an in-depth verification of the *SegCs*, *SegDs*, ..., *SegSs* fields before using them, it is possible to provide the kernel with a bogus selector and have it used as an (implicit) operand in an instruction such as `POP DS` or `IRETD`. As a consequence of the design allowing user-mode applications to generate kernel #NP exceptions, we should expect the kernel to handle such events properly - and that is precisely the case. If we take a look at the \base\ntos\ke\i386\trap.asm file, lines 4236 - 4346, we will see that the kernel performs analysis of the faulting instruction's opcode and responds accordingly (see Listing 14).

Listing 14: Windows #NP exception handler implementation

```
align dword
      public  _KiTrap0B
_KiTrap0B       proc

(...)

Kt0b30:

      (...)

      mov     eax, [ebp]+TsEip        ; (eax)->faulted Instruction
      mov     eax, [eax]              ; (eax)= opcode of faulted instruction
      mov     edx, [ebp]+TsEbp        ; (edx)->previous trap exit trapframe

      add     edx, TsSegDs            ; [edx] = prev trapframe + TsSegDs
      cmp     al, POP_DS              ; Is it pop ds instruction?
      jz      Kt0b90                  ; if z, yes, go Kt0b90

      add     edx, TsSegEs - TsSegDs  ; [edx] = prev trapframe + TsSegEs
      cmp     al, POP_ES              ; Is it pop es instruction?
      jz      Kt0b90                  ; if z, yes, go Kt0b90

      add     edx, TsSegFs - TsSegEs  ; [edx] = prev trapframe + TsSegFs
      cmp     ax, POP_FS              ; Is it pop fs (2-byte) instruction?
      jz      Kt0b90                  ; If z, yes, go Kt0b90

      add     edx, TsSegGs - TsSegFs  ; [edx] = prev trapframe + TsSegGs
      cmp     ax, POP_GS              ; Is it pop gs (2-byte) instruction?
      jz      Kt0b90                  ; If z, yes, go Kt0b90

;
; The exception is not caused by pop instruction.  We still need to check
; if it is caused by iret (to user mode.)  Because user may have a NP
; cs and we will trap at iret in trap exit code.
;
```

```
        cmp     al, IRET_OP              ; Is it an iret instruction?
        jne     Kt0b199                  ; if ne, not iret, go bugcheck
```

What is even more, it turns out that causing an IRETD instruction to fail upon an invalid *SegCs* value can have a very desirable impact on the layout of the kernel stack. Let's analyze the situation in more detail - the layout of the stack right before the execution of IRETD is shown in Figure 8.

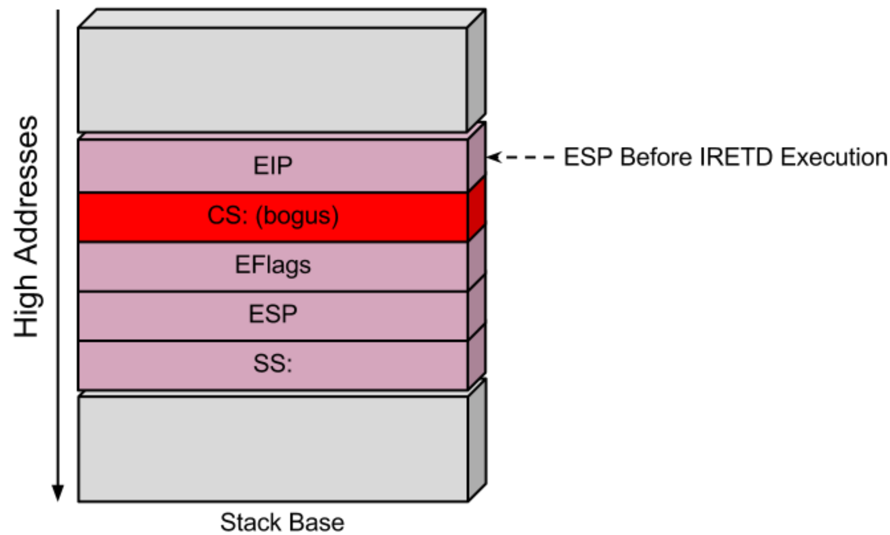

Figure 8: Kernel stack layout before IRETD execution

When IRETD loads the controlled (and intentionally bogus) *SegCs* value from stack, the selector verification fails causing an #NP exception to be generated on top of the current stack layout (Image 9).

As a consequence of a nested interrupt, the new trap frame is shifted by 20 bytes (5 fields, each four-byte long). After the CPU passes the execution to `nt!KiTrap0B` (the #NP handler), the execution path shown in Listing 15 is taken.

Listing 15: IRETD failure handling in KiTrap0B

```
        cmp     al, IRET_OP              ; Is it an iret instruction?
        jne     Kt0b199                  ; if ne, not iret, go bugcheck

        (...)

        mov     ecx, (TsErrCode+4)/4
        lea     edx, [ebp]+TsErrCode
Kt0d001:
        mov     eax, [edx]
        mov     [edx+12], eax
        sub     edx, 4
```

```
loop    Kt0d001

sti

add     esp, 12                 ; adjust esp and ebp
add     ebp, 12
mov     ebx, [ebp]+TsEip        ; (ebx)->faulting instruction
mov     esi, [ebp]+TsErrCode
and     esi, 0FFFFh
mov     eax, STATUS_ACCESS_VIOLATION
jmp     CommonDispatchException2Args0d ; Won't return
```

The assembly is responsible for fixing the trap frame, adjusting the Esp and Ebp registers and passing the execution down to a generic exception dispatcher. From the perspective of controlling *TempSegCs* and *TempEsp*, the first part of the code is particularly interesting - it basically merges the current trap-frame with the left-overs of the previous one, and does so by moving the entire new structure 12 bytes towards top of the stack. Image 10 illustrates the performance of the loop in action.

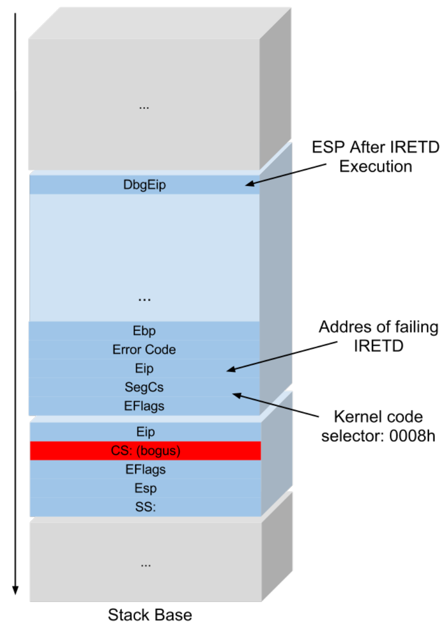

Figure 9: Kernel stack layout after IRETD execution

After one trap frame is created from the two parts, both Esp and Ebp need to be re-adjusted to point to the structure's base address. Having a clean and valid stack layout, the code proceeds to a generic exception dispatch routine. But hey... something very important has just happened!

The process of moving an entire `KTRAP_FRAME` structure forward by 12 bytes

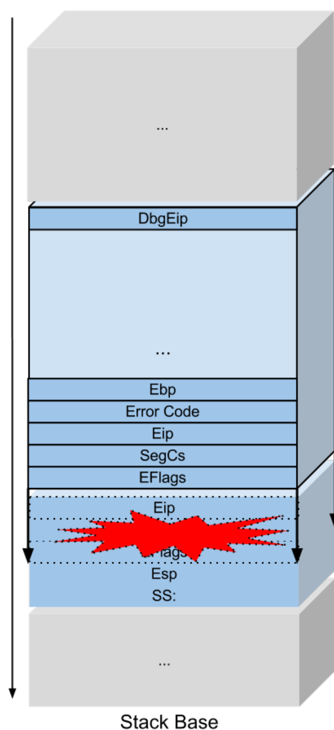

Figure 10: Merging two trap frames into a single one.

greatly affects the *TempSegCs* and *TempEsp* fields - since they were mapped lower than usual for a while, not initialized, and then copied into their usual location, they now contain whatever was present in the old, temporary location. And what was it? The *DbgEip* and *DbgArgMark* values from the very first trap frame, respectively (fields that are 12 bytes below *TempSegCs* and *TempEsp*).

That's correct - *TempSegCs* now takes the value of the old *DbgEip* field, while *TempEsp* contains bytes previously consumed by `KTRAP_FRAME.DbgArgMark`. At the time of its existence, *DbgEip* address contained the original user-mode return address, making it almost entirely controllable by a ring-3 exploit. When it comes to *DbgArgMark*, the field plays a role of a trap-frame marker and is always set to a magic value of `0BADB0D00h`, as observed in `\base\ntos\ke\i386\kimacro.inc` and shown in Listing 16.

Listing 16: The magic DbgArgMark value exposed

```
mov [ebp]+TsDbgArkMark, 0BADB0D00h
```

Unfortunately, directly after filling *TempSegCs* and *TempEsp* with non-zero values, the kernel attempts to dispatch the exception. Under typical circumstances, it is unable to handle the event, and terminates the process in emer-

gency mode without giving any chance to take advantage of the conducive stack contents. In order to intercept the IRETD exception and regain control over the process execution flow, it is necessary to attach a debugger process (through the Windows Debug API) which will receive a notification about the event, and will be able to modify the debuggee's CPU context to restore proper functioning of the process.

Specifically, when the IRETD instruction fails, the debugger receives an `EXCEPTION_DEBUG_EVENT` signal, which can be handled by redirecting the cs:eip pair to a valid location, and resuming the execution through `ContinueDebugEvent`. In result, the debuggee continues its normal execution, but having *TempSegCs* and *TempEsp* influenced by the #NP exception handler.

Before proceeding to the next section, let's summarize the steps discussed so far:

In debugger:

1. Create the core exploit process with a `DEBUG_PROCESS` flag (in my case, the NTVDM.EXE subsystem process),
2. Optionally - if using the NTVDM method of controlling IRETD parameters, inject a DLL with the exploit into the debuggee,
3. Enter a standard debugger loop,
4. When `EXCEPTION_DEBUG_EVENT` is encountered, set the debuggee's cs: to a valid value (i.e. 0x001B on most systems) and point Eip into a stage-two routine.

In debugee:

1. Optionally - if using a DLL within NTVDM, initialize a minimal VDM subsystem,
2. Craft a CONTEXT structure to contain a valid context with a bogus cs: register,
3. Use the structure to trigger an IRETD failure using one of the previously discussed techniques,
4. "Wait" until the debugger redirects the execution flow to stage-two routine.

6.2 Spraying kernel address space

The IRETD exception enables us to set the otherwise uninitialized *TempEsp* pointer to a constant value of `0BADB0D00h`, which is a step in the right direction. To make the exploit work, we need to ensure that the virtual address is mapped to non-pageable physical memory. Experimental data shows that this memory region is usually not occupied by any of the default device drivers

present on Windows 7 or dynamic pool allocations. Therefore, the virtual address can be subject to kernel address space spraying, a ring-0 equivalent of a technique most commonly used for browser vulnerability exploitation [3] [1].

Very little information regarding kernel memory spraying is publicly available on the Internet. I believe it is mostly due to a relatively small number of kernel-mode vulnerabilities, with even fewer bugs requiring any kind of address space spraying. The subject in itself is worth a separate research - in this section, I will only outline the basic concepts and tools which can be used to achieve a decent level of spraying reliability.

When trying to reach a certain kernel-mode address with non-pageable memory, the amount of physical memory available on the machine plays a key role, especially in cases where there is less RAM than the size of kernel address space (usually 2GB). For the purpose of performing controlled or semi-controlled (in terms of content) allocations from the kernel pools, a pair of `NtCreateSymbolicLinkObject` (pageable) and `NtQueueApcThread` (non-pageable memory) services is probably the simplest yet very effective choice for Windows Vista and 7.

In its great courtesy, Windows supports a great number of statistics and performance information sources, which can be easily incorporated into the spraying code in order to improve the invaluable accuracy; one example of such source is the *SystemPerformanceInformation* class, providing detailed information regarding various aspects of system memory usage. What can be even more useful, it is possible to enumerate all executive objects accessible through handles, owned by every process running in the system - together with the corresponding virtual addresses - using the *SystemHandleInformation* class. When combined with object-based spraying, both mechanisms make it feasible to reach any specific kernel address with a high degree of accuracy (depending on various conditions).

The proof-of-concept code developed to demonstrate successful exploitation of the vulnerability works by raising the virtual address space consumption to 40% using paged pool and symbolic link objects (resulting in the occupation of virtual addresses up to 0B0000000h). After that, the exploit starts to spray the memory using KAPC structures allocated on NonPaged pool - when the system runs out of physical memory or a 80% address space consumption is reached, the spraying is finished.

For an in-depth analysis of the Windows kernel pool allocator, please refer to an excellent paper and slides published by Tarjei Mandt in 2011 [10].

6.3 A finishing touch

After putting all of the discussed techniques to work and triggering the vulnerability inside of the exploit child process, we should end up having ring-0 privileges after returning from the first interrupt encountered while executing code under the `LDT[0]` segment. Keep in mind that final value of the cs: register is based on the low 16 bits of the user-mode interrupt return-address at the time of invoking a syscall to pass a bogus *SegCs* value (e.g. `NtContinue`). In

order to grant elevated privileges, you might need to set up a simple assembly wrapper for calling NtContinue or NtVdmControl, and position it at the beginning of a 64kB-aligned memory block.

Listing 17: An assembly wrapper for calling a CONTEXT-switching system service

```
+0x00: NOP
+0x01: NOP
+0x02: POP AX
+0x04: MOV EDX, EBP
+0x06: INT 2Eh
+0x08: ...
```

Furthermore, you should always remember to clean up the damage made by the kernel to itself during the exploitation. In this case, the kernel arbitrarily overwrites 12 bytes residing at {0BADB0D00 - 0Ch, 0BADB0D00}, which might later manifest itself in the form of system instability.

After acquiring ring-0 privileges for your assembly payload, the rest of the steps can be duplicated from the Windows XP exploitation process: overwriting the HalDispatchTable+4 pointer and assigning the SYSTEM security token to a custom application work fine on both system platforms. The four kernel API functions used in the previous exemplary payload suffice to replace the primary token of any process on every Windows NT-family system without applying any major modifications to the code.

6.4 Putting it all in one place

Having described all techniques and concepts required to achieve a decent degree of exploitation reliability, let's summarize the major steps taken by a successful proof-of-concept exploit. Since the debugger's role has not changed since when it was last described, lets focus on the debuggee functionality.

1. Optionally - if using a DLL within NTVDM, initialize a minimal VDM subsystem,
2. Initialize a system service stub, later resulting in having a XXXX0008 return address pushed on the trap frame,
3. Craft a CONTEXT structure to contain a valid context with a bogus cs: register,
4. Use the structure to trigger an IRETD failure using one of the previously discussed techniques,
5. "Wait" until the debugger redirects the execution flow to stage-two routine,
6. Initialize pointers to kernel-mode API functions required by stage-two payload,

7. Create a code segment entry in LDT[0],
8. Spray the kernel virtual address space, in order to reach the 0BADB0D00h address with non-pageable, writable memory mapping,
9. Jump into the LDT[0] segment and trigger the vulnerability,

After returning with ring-0 privileges:

1. Fix the broken values around 0BADB0D00h,
2. Overwrite the HalDispatchTable+4 pointer with stage-two payload address,
3. Emulate a regular return to user-mode.
4. Invoke the overwritten function pointer through nt!NtQueryInterval-Profile,
5. Escalate the security token of a chosen process (e.g. a command shell),
6. Restore original HalDispatchTable+4 value and terminate.

A few minor steps such as payload initialization or spawning a command shell are not covered in the list, being either obvious or optional steps. Assuming successful completion of all the key stages of exploitation, one should be able to see his process running with the NT AUTHORITY
SYSTEM privileges, as shown in Image 11.

7 Conclusion

The number of vulnerabilities disclosed, exploited, publicly discussed and fixed in Windows user-mode client applications during the few recent years undoubtedly out-weights the quantity of kernel-mode security issues. As defense-in-depth mitigation mechanisms (such as ASLR, DEP or sandboxing) for desktop programs are becoming more and more effective, I expect to see an increase in the focus put into other promising targets, poorly secured and vulnerable kernel-mode code being the most intuitive choice. This article shows how ring-0 exploitation techniques, like stack and pool spraying combined with kernel address space information leaks and other undocumented functionalities (user-mode callbacks, specific exception handlers behavior) can prove useful for uncommon and non-trivial vulnerability exploitation. As Microsoft is going to incorporate numerous new kernel-level anti-exploitation measures in the Windows 8 build, I am really excited to see how the ring-0 security field - and specifically, offensive techniques - are going to evolve and develop in the near future.

```
C:\Windows\system32\cmd.exe
[INFO] Memory spraying progress: 0.400070% free pages => 0.800000%
[INFO] Memory spraying progress: 0.400070% free pages => 0.800000%
[INFO] Memory spraying progress: 0.405930% free pages => 0.800000%
[INFO] Memory spraying progress: 0.428391% free pages => 0.800000%
[INFO] Memory spraying progress: 0.451828% free pages => 0.800000%
[INFO] Memory spraying progress: 0.474289% free pages => 0.800000%
[INFO] Memory spraying progress: 0.496750% free pages => 0.800000%
[INFO] Memory spraying progress: 0.520187% free pages => 0.800000%
[INFO] Memory spraying progress: 0.516281% free pages => 0.800000%
[INFO] Memory spraying progress: 0.537766% free pages => 0.800000%
[WARNING] NtQueueApcThread failed, 0xc0000017

[INFO] Spraying process finished, Status=1
[DEBUG] (vdm.cpp:main 373) DBG_EVENT(EXCEPTION_DEBUG_EVENT)
[DEBUG] (vdm.cpp:main 395) Context dump:
  Eax=56565656 Ecx=56565656 Edx=56565656 Ebx=56565656
  Esi=56565656 Edi=56565656 Ebp=0113e0b0 Esp=0113e0ac
  Eip=20000005 EFlags=00000202
  CS=001b DS=0023 ES=0023 FS=003b GS=0000
[DEBUG] (vdm.cpp:main 455) DBG_EVENT(EXIT_THREAD_DEBUG_EVENT)
[DEBUG] (vdm.cpp:main 448) DBG_EVENT(EXIT_PROCESS_DEBUG_EVENT)
[INFO] Terminating the broker process

C:\Users\asdf\Desktop>
```

```
Administrator: C:\WINDOWS\SYSTEM32\CMD.EXE
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\asdf\Desktop>whoami
nt authority\system

C:\Users\asdf\Desktop>_
```

Figure 11: Escalated command shell, a result of successful exploitation on a Windows 7 platform.

References

[1] Alexander Sotirov: *Heap Feng Shui in JavaScript.* https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

[2] Bartosz Wojcik: *Konkurs Pimp My CrackMe.* http://www.secnews.pl/2011/04/28/konkurs-pimp-my-crackme/.

[3] Corelan Team (corelanc0d3r): *Exploit writing tutorial part 11 : Heap Spraying Demystified.* https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

[4] Intel: *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.* Intel Corporation, 2007.

[5] Mateusz "j00ru" Jurczyk: *nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques.* http://j00ru.vexillium.org/?p=769.

[6] Mateusz "j00ru" Jurczyk: *Protected Mode Segmentation as a powerful anti-debugging measure.* http://j00ru.vexillium.org/?p=866.

[7] Mateusz "j00ru" Jurczyk: *Windows Security Hardening Through Kernel Address Protection.* http://j00ru.vexillium.org/?p=1038.

[8] Ruben Santamarta: *Exploiting Common Flaws in Drivers.* http://reversemode.com/index.php?option=com_content&task=view&id=38&Itemid=1.

[9] Tarjei Mandt: *Kernel Attacks Through User-Mode Callbacks.* http://mista.nu/research/mandt-win32k-slides.pdf.

[10] Tarjei Mandt: *Kernel Pool Exploitation on Windows 7.* http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf.

[11] Z0mbie: *Adding LDT entries in Win2K.* http://vxheavens.com/lib/vzo13.html.