Bochspwn Reloaded

Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking

Mateusz “j00ru” Jurczyk

Black Hat USA 2017, Las Vegas
Agenda

• User ↔ kernel communication pitfalls in modern operating systems

• Introduction to Bochspwn Reloaded
  • Detecting kernel information disclosure with software x86 emulation

• System-specific approaches and results in Windows and Linux

• Future work and conclusions
Bio

- Project Zero @ Google
- CTF Player @ Dragon Sector
- Low-level security researcher with interest in all sorts of vulnerability research and software exploitation.
- http://j00ru.vexillium.org/
- @j00ru
User ↔ kernel communication
OS design fundamentals

• User applications run independently of other programs / the kernel.

• Whenever they want to interact with the system, they call into the kernel.

• Ring-3 memory is the i/o data exchange channel.
  • Also registers to a small extent.
Life of a system call

User-mode Program
- Write input data
- Invoke system call

Shared Memory (user-mode)
- Read input data
- Syscall logic
- Write output data
- Return to user space

System Kernel
- Read output data
Life of a system call

- User-mode Program
  - Write input data
  - Invoke system call
- Shared Memory (user-mode)
- System Kernel
  - Read input data
  - Syscall logic
  - Write output data
  - Return to user space
  - Read output data
In a perfect world...

• Within the scope of a single system call, each memory unit is:
  1. Read from at most once, securely.
     ... then ...
  2. Written to at most once, securely, only with data intended for user-mode.
In reality (double fetches)

Read from **at most once**, securely.

- Subject of the original *Bochspwn* study in 2013 with Gynvael Coldwind.

- Possible violation: *double (or multiple) fetches*, may allow race conditions to break code assumptions → buffer overflows, write-what-where conditions, arbitrary reads, other badness.

- Dozens (40+) vulnerabilities reported and fixed in Windows.
Kernel double fetches

Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns

- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1258)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1259)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1260)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1261)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1262)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1263)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1264)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1265)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1266)
- Mateusz "J00ru" Jurczyk of Google Inc for reporting the Win32k Race Condition Vulnerability (CVE-2013-1267)

Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns

Bochspwn: Identifying 0-days via system-wide memory access pattern analysis
In reality – various other problem indicators

• Unprotected accesses to user-mode pointers.

• User-mode accesses while PreviousMode=KernelMode.

• Multiple writes to a single memory area.

• Reading from a user-mode address after already having written to it.

• Accessing ring-3 memory:
  • within deeply nested call stacks.
  • with the first enabled exception handler very high up the call stack.
The subject of this talk

Written to at most once, securely,

only with data intended for user-mode
Writing data to ring-3

• System calls
  • Almost every single one on any system.

• IOCTLs
  • A special case of syscalls, but often have dedicated output mechanisms.

• User-mode callbacks
  • Specific to the graphical win32k.sys subsystem on Windows.

• Exception handling
  • Building exception records on the user-mode stack.
The easy problem – primitive types

NTSTATUS NtMultiplyByTwo(DWORD InputValue, LPDWORD OutputPointer) {
    DWORD OutputValue;

    if (InputValue != 0) {
        OutputValue = InputValue * 2;
    }

    *OutputPointer = OutputValue;
    return STATUS_SUCCESS;
}

Uninitialized if InputValue == 0
The easy problem – primitive types

- Disclosure of uninitialized data via basic types can and will occur, but:
  - is not a trivial bug for developers to make,
  - compilers will often warn about instances of such issues,
  - leaks only a limited amount of data at once (max 4 or 8 bytes on x86),
  - may be detected during development or testing, since they can be functional bugs.

- Not an inherent problem to kernel security.
The hard problem – structures and unions

```c
typedef struct _SYSCALL_OUTPUT {
    DWORD Sum;
    DWORD Product;
    DWORD Reserved;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NtArithOperations(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
    SYSCALL_OUTPUT OutputStruct;

    OutputStruct.Sum = InputValue + 2;
    OutputStruct.Product = InputValue * 2;

    RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
    return STATUS_SUCCESS;
}
```

Never initialized because „reserved”
The hard problem – structures and unions

typedef union _SYSCALL_OUTPUT {
    DWORD Sum;
    QWORD LargeSum;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
    SYSCALL_OUTPUT OutputUnion;

    OutputUnion.Sum = InputValue + 2;

    RtlCopyMemory(OutputPointer, &OutputUnion, sizeof(SYSCALL_OUTPUT));
    return STATUS_SUCCESS;
}
The hard problem – structures and unions

typedef struct _SYSCALL_OUTPUT { 
    DWORD Sum;
    QWORD LargeSum;
} SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;

NTSTATUS NtSmallSum(DWORD inputValue, PSYSCALL_OUTPUT OutputPointer) {
    SYSCALL_OUTPUT OutputStruct;
    OutputStruct.Sum = inputValue + 2;
    OutputStruct.LargeSum = 0;

    RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
    return STATUS_SUCCESS;
}
The hard problem – structures and unions

• Structures and unions are almost always copied in memory entirely.

• With many fields, it’s easy to forget to set some of them.
  • or they could be uninitialized by design.

• Unions introduce holes for data types of different sizes.

• Compilers introduce padding holes to align fields in memory properly.

• Compilers have little insight into structures (essentially data blobs):
  • dynamically allocated from heap / pools.
  • copied in memory with memcpy() etc.
The hard problem – fixed-size arrays

The code snippet demonstrates a function `NtGetSystemPath` which returns the path to the System32 directory.
The hard problem – fixed-size arrays

• Many instances of long fixed-size buffers used in user ↔ kernel data exchange.
  • Paths, names, identifiers etc.
  • While container size is fixed, the content length is usually variable, and most storage ends up unused.

• Frequently part of structures, which makes it even harder to only copy the relevant part to user-mode.

• May disclose huge continuous portions of uninitialized memory at once.
The hard problem – arbitrary request sizes

NTSTATUS NtMagicValues(LPDWORD OutputPointer, DWORD OutputLength) {
    if (OutputLength < 3 * sizeof(DWORD)) {
        return STATUS_BUFFER_TOO_SMALL;
    }

    LPDWORD KernelBuffer = Allocate(OutputLength);
    KernelBuffer[0] = 0xdeadbeef;
    KernelBuffer[1] = 0xbadc0ffe;
    KernelBuffer[2] = 0xcafed00d;

    RtlCopyMemory(OutputPointer, KernelBuffer, OutputLength);
    Free(KernelBuffer);

    return STATUS_SUCCESS;
}
The hard problem – arbitrary request sizes

- Common scheme in Windows – making allocations with user-controlled size and passing them back fully regardless of the amount of relevant data inside.

- May enable disclosure from both stack/heap in the same affected code.
  - Kernel often relies on stack memory for small buffers and falls back to pools for large ones.

- Often leads to large leaks of a controlled number of bytes.
  - Facilitates aligning heap allocation sizes to trigger collisions with specific objects in memory.
  - Gives significantly more power to the attacker in comparison to other bugs.
Extra factors: no automatic initialization

- Neither Windows nor Linux pre-initialize allocations (stack or heap) by default.
  - Exceptions from the rule mostly found in Linux: `kzalloc()`, `__GFP_ZERO`, `PAX_MEMORY_STACKLEAK` etc.
  - Buffered IOCTL I/O buffer is now always cleared in Windows since June 2017 (new!)
    - Resulting regions have old, leftover garbage bytes set by their last user.
- From MSDN:

  **Note** Memory that `ExAllocatePoolWithTag` allocates is uninitialized. A kernel-mode driver must first zero this memory if it is going to make it visible to user-mode software (to avoid leaking potentially privileged contents).
Extra factors: no visible consequences

- C/C++ don’t make it easy to copy data securely between different security domains, but there’s also hardly any punishment.
  - If the kernel discloses a few uninitialized bytes here and there, nothing will crash and likely no one will ever know (until now 😊).
- If a kernel developer is not aware of the bug class and not actively trying to prevent it, they’ll probably never find out by accident.
Extra factors: leaks hidden behind system API

User-mode Program

Call API function

User-Mode System API

Convert arguments and invoke syscall

Syscall logic

Write output with leaks and return

Extract meaningful data

Disclosed memory lost here

System Kernel

Return the specific requested values
Severity and considerations

• “Just” local info leaks, no memory corruption or remote exploitation involved by nature.

• Actual severity depends on what we manage to leak out of the kernel.

• On the upside, most disclosures are silent / transparent, so we can trigger the bugs indefinitely without ever worrying about system stability.
Severity and considerations

• Mostly useful as a single link in a LPE exploit chain.
  • Especially with the amount of effort put into KASLR and protecting information about the kernel address space.

• One real-life example is a Windows kernel exploit found in the HackingTeam dump in July 2015 (CVE-2015-2433, MS15-080).
  • Pool memory disclosure leaking base address of win32k.sys.
  • Independently discovered by Matt Tait at P0, Issue #480.

Kernel-mode ASLR leak via uninitialized memory returned to usermode by NtGdiGetTextMetrics
Reported by mattrait@google.com, Jul 10 2015
Stack disclosure benefits

• Consistent, immediately useful values, but with limited variety and potential to leak anything else:
  • Addresses of kernel stack, heap (pools), and executable images.
  • /GS stack cookies.
  • Syscall-specific data used by services previously invoked in the same thread.
  • Potentially data of interrupt handlers, if they so happen to trigger in the context of the exploit thread.
Heap disclosure benefits

• Less obvious memory, but with more potential to collide with miscellaneous sensitive information:
  • Addresses of heap, potentially executable images.
  • Possibly data of any active kernel module (disk, network, video, peripheral drivers).
    • Depending on heap type, allocation size and system activity.
Prior work (Windows)

- **P0 Issue #480** (win32k!NtGdiGetTextMetrics, CVE-2015-2433), Matt Tait, July 2015

- **Leaking Windows Kernel Pointers**, Wandering Glitch, RuxCon, October 2016
  - Eight kernel uninitialized memory disclosure bugs fixed in 2015.

- **Win32k Dark Composition: Attacking the Shadow Part of Graphic Subsystem**, Peng Qiu and SheFang Zhong, CanSecWest, March 2017
  - Hints about multiple infoleaks in win32k.sys user-mode callbacks, no specific details.

- **Automatically Discovering Windows Kernel Information Leak Vulnerabilities**, fanxiaocao and pjf of IceSword Lab (Qihoo 360), June 2017
Prior work (Linux)

• In 2010, **Dan Rosenberg** went on a rampage and killed 20+ info leaks in various subsystems.
  
  • Some of the work mentioned in *Stackjacking and Other Kernel Nonsense*, presented by Dan Rosenberg and Jon Oberheide in 2011.

• A number of patches submitted throughout the years by various researchers: **Salva Peiró, Clément Lecigne, Marcel Holtmann, Kees Cook, Jeff Mahoney**, to name a few.

• The problem seems to be known and well understood in Linux.
Bochspwn Reloaded design
• Bochs is a full IA-32 and AMD64 PC emulator.
  • CPU plus all basic peripherals, i.e. a whole emulated computer.
• Written in C++.
• Supports all latest CPUs and their advanced features.
  • SSE, SSE2, SSE3, SSSE3, SSE4, AVX, AVX2, AVX512, SVM / VT-x etc.
• Correctly hosts all common operating systems.
• Provides an extensive instrumentation API.
Performance (short story)

IPS: 38.615M  NUM
Performance (long story)

- On a modern PC, non-instrumented guests run at up to **80-100M IPS**.
  - Sufficient to boot up a system in reasonable time (<5 minutes).
  - Environment fairly responsive, at between 1-5 frames per second.

- Instrumentation incurs a severe overhead.
  - Performance can drop to **30-40M IPS**.
    - still acceptable for research purposes.
  - Simple logic and optimal implementation is the key to success.
Bochs instrumentation support

• Instrumentation written in the form of callback functions plugged into Bochs through BX_INSTR macros, statically built into bochs.exe.

• Rich variety of event callbacks:
  • init, shutdown, before/after instruction, linear/physical memory access, exception, interrupt, ...

• Enables developing virtually any logic to examine or steer the whole operating system execution.
  • counting statistics, tracing instructions or memory accesses, adding metadata, altering instruction behavior, adding new instructions, ...
Bochs instrumentation callbacks

- BX_INTR_INIT_ENV
- BX_INTR_EXIT_ENV
- **BX_INTR_INITIALIZE**
- **BX_INTR_EXIT**
- BX_INTR_RESET
- BX_INTR_HLT
- BX_INTR_MWAIT
- BX_INTR_DEBUG_PROMPT
- BX_INTR_DEBUG_CMD
- BX_INTR_CNEAR_BRANCH_TAKEN
- BX_INTR_CNEAR_BRANCH_NOT_TAKEN
- BX_INTR_UCNEAR_BRANCH
- BX_INTR_FAR_BRANCH
- BX_INTR_OPCODE
- BX_INTR_EXCEPTION
- BX_INTR_INTERRUPT

- BX_INTR_HWINTERRUPT
- BX_INTR_CLFLUSH
- BX_INTR_CACHE_CNTRL
- BX_INTR_TLB_CNTRL
- BX_INTR_PREFETCH_HINT
- **BX_INTR_BEFORE_EXECUTION**
- **BX_INTR_AFTER_EXECUTION**
- BX_INTR_REPEAT_ITERATION
- **BX_INTR_LIN_ACCESS**
- BX_INTR_PHY_ACCESS
- BX_INTR_INP
- BX_INTR_INP2
- BX_INTR_OUTP
- BX_INTR_WRMSR
- BX_INTR_VMEXIT
Core logic

• Taint tracking for the entire kernel address space.

• Required functionality:

  1. Set taint on new allocations (stack and heap).
  2. Remove taint on free (heap-only).
  3. Propagate taint in memory.
  4. Detect copying of tainted memory to user-mode.
Ancillary functionality

• Keep track of loaded guest kernel modules.
• Read stack traces on error to deduplicate bugs.
• Symbolize callstacks to prettify reports.
• Break into kernel debugger (attached to guest) on error.
Shadow memory representation

- Guest OS memory
  - Kernel land
  - User land
- Bochs.exe memory
  - Shadow memory (metadata)
- Memory unit descriptor
  - bool tainted
  - uint32 alloc_size
  - uint32 alloc_base_addr
  - uint32 alloc_tag/flags
  - uint32 alloc_origin
Shadow memory representation

• Linear in relation to the size of the guest kernel address space.
  • Only 32-bit guests supported at the moment.
  • Some information stored at 1-byte granularity, some at 8-byte granularity.

• Stores extra metadata useful for bug reports in addition to taint.

• Max shadow memory consumption:
  • Windows (2 GB kernel space) – **6 GB**
  • Linux (1 GB kernel space) – **3 GB**
  • Easily manageable with sufficient RAM on the host.
Double-tainting

• Every time a region is tainted, corresponding guest memory is also padded with a special marker byte.
  • 0xAA for heap and 0xBB for stack areas.

• May trigger use-of-uninit-memory bugs other than just info leaks.

• Provides evidence that a bug indicated by shadow memory is real.

• Eliminates all false-positives, guarantees ~100% true-positive ratio.
Setting taint on stack

• Cross-platform, universal.

• Detect instructions modifying the ESP register:

  \[
  \text{ADD~ESP,~...} \quad \text{SUB~ESP,~...} \quad \text{AND~ESP,~...}
  \]

• After execution, if ESP decreased, call:

  \[
  \text{set\_taint(ESP}\text{\textsubscript{old}, ESP}\text{\textsubscript{new}})
  \]

• Relies on the guest behaving properly, but both Windows and Linux do.
Setting taint on heap/pools (simplified)

• Very system specific.

• Requires knowledge of both the allocated address and request (size, tag, flags, origin etc.) at the same time.

• Then:

```
set_taint(address, address + size)
```
Removing taint on heap free

• Break on `free()` function prologue.

• Look up allocation size from shadow memory.

• Clear all taint and metadata for the whole region.
  • Alternatively: re-taint to detect UAF and leaks of freed memory.
Taint propagation

• The hard part – detecting data transfers.

• Bochspwn only propagates taint for `<REP> MOVS{B, D}` instructions.
  • Typically used by `memcpy()` and its inlined versions across drivers.
  • Both source (`ESI`) and destination (`EDI`) addresses conveniently known at the same time.
  • We mostly care about copies of large memory blobs, anyway.

• Best effort approach
  • Moving taint across registers would require instrumenting dozens or hundreds of instructions instead of one, incurring a very significant CPU overhead for arguably little benefit.
Taint propagation

• If a memory access is not a result of `<REP> MOVSB, D`:
  • On write, clear the taint on the memory area (mark initialized).
  • On read, check taint. If shadow memory indicates uninitialized read, verify it with guest memory.
    • In case of mismatch (byte is not equal to the marker for whatever reason), clear taint.
    • If it’s a real uninitialized read, we may report it as a bug if running in „strict mode“.
Bug detection

• Activated on `<REP> MOVSB, EDI` when ESI is in kernel-mode and EDI is in user-mode.
  
  • Copying an output data blob to user land.

  • If there is any tainted byte in the source memory region, report a bug.
Let’s run it against some real systems
Bochspwn vs. Windows
(Un)tainting pool allocations

• A number of pool allocation routines in the kernel:
  • ExAllocatePool, ExAllocatePoolEx, ExAllocatePoolWithTag,
    ExAllocatePoolWithTagPriority,
  ExAllocatePoolWithTagQuotaTag

• All eventually call into one: **ExAllocatePoolWithTag**.

• STDCALL calling convention: arguments on stack, return value in EAX.
  • Both request (origin, size, tag) and output (allocated address) available at the same time.

• Similar for untainting freed regions.

• Extremely convenient for instrumentation.
ExAllocatePoolWithTag

Callers

ExAllocatePoolWithQuotaTag

ExAllocatePoolWithPriority

ExAllocatePoolEx

ExAllocatePool

ExAllocatePoolWithTag

EAX allocated address
[ESP] allocation origin
[ESP+4] requested size
[ESP+8] allocation tag

loc_529AAC:
    pop edi
    pop esi
    pop ebx
    mov esp, ebp
    pop ebp
    ret 0ch

__stdcall ExAllocatePoolWithTag(x, x, x) endp
Exported entry 229. ExFreePoolWithTag

Attributes: bp-based frame

\textbf{ExFreePoolWithTag} (VOID P, ULONG Tag)
\textbf{ExFreePoolEx} (VOID P, ULONG Tag)
\textbf{ExFreePool} (VOID P)

\textbf{ExFreePoolWithTag} (VOID P, ULONG Tag)
\textbf{ExFreePoolEx} (VOID P, ULONG Tag)
\textbf{ExFreePool} (VOID P)

\textbf{mov} edi, edi
\textbf{push} ebp
\textbf{mov} ebp, esp
\textbf{and} esp, 0FFFFFFF0h
\textbf{mov} eax, _ExpSpecialAllocations
\textbf{sub} esp, 4Ch
\textbf{push} ebx
\textbf{push} esi
\textbf{mov} esi, [ebp+P]
\textbf{push} edi
\textbf{test} eax, eax
\textbf{jz} loc_528095

\textbf{ESP+4} freed region

\textbf{Callers}

\textbf{ExFreePool}

\textbf{ExFreePoolEx}

\textbf{ExFreePoolWithTag}
Optimized, specialized allocators

- `win32k!AllocFreeTmpBuffer` first tries to return a cached memory region (`win32k!gpTmpGlobalFree`) for allocations of $\leq 4096$ bytes.
  - Called from $\sim 55$ locations, many syscall handlers.
  - Can be easily patched out to always use the system allocator.

```c
PUVOID __stdcall AllocFreeTmpBuffer(unsigned int a1)
{
    PVOID result; // eax@2

    if ( a1 > 0x1000 || (result = InterlockedExchange(gpTmpGlobalFree, 0)) == 0 )
        result = AllocThreadBufferWithTag(a1, 'pMTG');
    return result;
}
```
Propagating taint and detecting bugs

• The standalone `memcpy()` function in drivers is implemented mostly as `rep movs`.
  
  • Still some optimizations left which transfer data through registers.
  
  • All instances of `memcpy()` have the same signature – they can be patched to only use `rep movs` on disk or at run time in kernel debugger.

• Inlined memory copy is typically also compiled to `rep movs`.

• As a result, tracking most transfers of large data blobs works with Bochspwn’s universal approach.
Windows 7 memory taint layout

0x80000000

0xffffffff

40 minutes of run time, 20s. interval, boot + initial ReactOS tests
Windows 10 memory taint layout

120 minutes of run time, 60s. interval, boot + initial ReactOS tests
Keeping track of processes/threads

• Simple traversal of a kernel linked-list in guest virtual memory.

• Unchanged since original Bochspwn from 2013.
Keeping track of loaded kernel modules

- Simple traversal of a kernel linked-list in guest virtual memory.
- Unchanged since original Bochspwn from 2013.
found uninit-access of address 94447d04

[pid/tid: 000006f0/00000740] { explorer.exe}

READ of 94447d04 (4 bytes, kernel-->user), pc = 902df30f

[ rep movsd dword ptr es:[edi], dword ptr ds:[esi] ]

[Pool allocation not recognized]

Allocation origin: 0x90334988 ((000c4988) win32k.sys!__SEH_prolog4+00000018)

Destination address: 1b9d380

Shadow bytes: 00 ff ff ff Guest bytes: 00 bb bb bb

Stack trace:

#0 0x902df30f ((0006f30f) win32k.sys!NtGdiGetRealizationInfo+0000005e)
#1 0x8288cdb6 ((0003ddb6) ntoskrnl.exe!KiSystemServicePostCall+00000000)
Kernel debugger support

- Textual Bochspwn reports are quite verbose, but not always sufficient to reproduce bugs.
  - Especially for IOCTL / other complex cases, where function arguments need to be deeply inspected, kernel objects examined etc.

- Solution – attach WinDbg to the emulated guest kernel!
  - Easily configured, Bochs has support for redirecting COM ports to Windows pipes.
  - Of course slow, as everything working on top of Bochs, but workable. 😊
Breaking on bugs

• Attached debugger is not of much use if we can’t debug the system at the very moment of the infoleak.

• Hence: after the bug is logged to file, Bochspwn injects an INT3 exception in the emulator.
  • WinDbg stops directly after the offending `rep movs` instruction.

• Overall feels quite magical. 😊
Testing performed

• Instrumentation run on both Windows 7 and 10.

• Executed actions:
  • System boot up.
  • Starting a few default apps – Internet Explorer, Wordpad, Registry Editor, Control Panel, games etc.
  • Generating some network traffic.
  • Running ~800 ReactOS unit tests (largely improved since 2013).

• Kernel code coverage still a major roadblock for effective usage of full-system instrumentation.
Results!
<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>CVE</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win32k Information Disclosure Vulnerability</td>
<td>CVE-2017-0299</td>
<td>Mateusz Jurczyk of Google Project Zero</td>
</tr>
</tbody>
</table>
Summary of the results so far

- A total of **30 vulnerabilities** fixed by Microsoft in the last months (mostly June).
## Summary – pool disclosures

<table>
<thead>
<tr>
<th>Issue #</th>
<th>CVE</th>
<th>Component</th>
<th>Fixed in</th>
<th>Root cause</th>
<th>Number of leaked bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1144</td>
<td>CVE-2017-8484</td>
<td>win32k!NtGdiGetOutlineTextMetricsInternalW</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>5</td>
</tr>
<tr>
<td>1145</td>
<td>CVE-2017-0258</td>
<td>nt!SepInitSystemDacs</td>
<td>May 2017</td>
<td>Structure size miscalculation</td>
<td>8</td>
</tr>
<tr>
<td>1147</td>
<td>CVE-2017-8487</td>
<td>\Device\KsecDD, IOCTL 0x390400</td>
<td>June 2017</td>
<td>Unicode string alignment</td>
<td>6</td>
</tr>
<tr>
<td>1150</td>
<td>CVE-2017-8488</td>
<td>MountMgr, IOCTL_MOUNTMGR_QUERY_POINTS</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>14</td>
</tr>
<tr>
<td>1152</td>
<td>CVE-2017-8490</td>
<td>WMIDataDevice, IOCTL 0x224000 (WmiQueryAllData)</td>
<td>June 2017</td>
<td>Structure alignment, Uninitialized fields</td>
<td>72</td>
</tr>
<tr>
<td>1153</td>
<td>CVE-2017-8491</td>
<td>win32k!NtGdiEnumFonts</td>
<td>June 2017</td>
<td>Fixed-size string buffers, Structure alignment, Uninitialized fields</td>
<td>6672</td>
</tr>
<tr>
<td>1154</td>
<td>CVE-2017-8492</td>
<td>VolMgr, IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>8</td>
</tr>
<tr>
<td>1156</td>
<td>CVE-2017-8493</td>
<td>Partmgr, IOCTL_DISK_GET_DRIVE_GEOMETRY_EX</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>4</td>
</tr>
<tr>
<td>1159</td>
<td>CVE-2017-8496</td>
<td>Partmgr, IOCTL_DISK_GET_DRIVE_LAYOUT_EX</td>
<td>June 2017</td>
<td>Structure alignment, Different-size union overlap</td>
<td>484</td>
</tr>
<tr>
<td>1161</td>
<td>CVE-2017-0259</td>
<td>nt!NtTraceControl (EtwpSetProviderTraits)</td>
<td>May 2017</td>
<td>?</td>
<td>60</td>
</tr>
<tr>
<td>1166</td>
<td>CVE-2017-8462</td>
<td>nt!NtQueryVolumeInformationFile (FileFsVolumeInformation)</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>1</td>
</tr>
<tr>
<td>1169</td>
<td>CVE-2017-0299</td>
<td>nt!NtNotifyChangeDirectoryFile</td>
<td>June 2017</td>
<td>Unicode string alignment</td>
<td>2</td>
</tr>
<tr>
<td>1238</td>
<td>CVE-2017-8564</td>
<td>Nsiproxy/netio, IOCTL 0x120007 (NsGetParameter)</td>
<td>July 2017</td>
<td>Structure alignment</td>
<td>13</td>
</tr>
</tbody>
</table>
Summary – stack disclosures

<table>
<thead>
<tr>
<th>Issue #</th>
<th>CVE</th>
<th>Component</th>
<th>Fixed in</th>
<th>Root cause</th>
<th>Number of leaked bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1177</td>
<td>CVE-2017-8482</td>
<td>nt!KiDispatchException</td>
<td>June 2017</td>
<td>Uninitialized fields</td>
<td>32</td>
</tr>
<tr>
<td>1178</td>
<td>CVE-2017-8470</td>
<td>win32k!NtGdiExtGetObjectW</td>
<td>June 2017</td>
<td>Fixed-size string buffer</td>
<td>50</td>
</tr>
<tr>
<td>1179</td>
<td>CVE-2017-8471</td>
<td>win32k!NtGdiGetOutlineTextMetricsInternalW</td>
<td>June 2017</td>
<td>Uninitialized field</td>
<td>4</td>
</tr>
<tr>
<td>1180</td>
<td>CVE-2017-8472</td>
<td>win32k!NtGdiGetTextMetricsW</td>
<td>June 2017</td>
<td>Structure alignment,</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Uninitialized field</td>
<td></td>
</tr>
<tr>
<td>1181</td>
<td>CVE-2017-8473</td>
<td>win32k!NtGdiGetRealizationInfo</td>
<td>June 2017</td>
<td>Uninitialized fields</td>
<td>8</td>
</tr>
<tr>
<td>1182</td>
<td>CVE-2017-0245</td>
<td>win32k!xxxClientLpkDrawTextEx</td>
<td>May 2017</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1183</td>
<td>CVE-2017-8474</td>
<td>DeviceApi (PiDqIrpQueryGetResult,</td>
<td>June 2017</td>
<td>Uninitialized fields</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PiDqIrpQueryCreate, PiDqQueryCompletePendedIrp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1186</td>
<td>CVE-2017-8475</td>
<td>win32k!ClientPrinterThunk</td>
<td>June 2017</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>1189</td>
<td>CVE-2017-8485</td>
<td>nt!NtQueryInformationJobObject (BasicLimitInformation, ExtendedLimitInformation)</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>8</td>
</tr>
<tr>
<td>1190</td>
<td>CVE-2017-8476</td>
<td>nt!NtQueryInformationProcess (ProcessVmCounters)</td>
<td>June 2017</td>
<td>Structure alignment</td>
<td>4</td>
</tr>
<tr>
<td>1191</td>
<td>CVE-2017-8477</td>
<td>win32k!NtGdiMakeFontDir</td>
<td>June 2017</td>
<td>Uninitialized fields</td>
<td>104</td>
</tr>
<tr>
<td>1192</td>
<td>CVE-2017-0167</td>
<td>win32kfull!SfnINLPUAHDRAWMENUITEM</td>
<td>April 2017</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>1193</td>
<td>CVE-2017-8478</td>
<td>nt!NtQueryInformationJobObject (information class 12)</td>
<td>June 2017</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1194</td>
<td>CVE-2017-8479</td>
<td>nt!NtQueryInformationJobObject (information class 28)</td>
<td>June 2017</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>1196</td>
<td>CVE-2017-8480</td>
<td>nt!NtQueryInformationTransaction (information class 1)</td>
<td>June 2017</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>1207</td>
<td>CVE-2017-8481</td>
<td>nt!NtQueryInformationResourceManager (information class 0)</td>
<td>June 2017</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1214</td>
<td>CVE-2017-0300</td>
<td>nt!NtQueryInformationWorkerFactory (WorkerFactoryBasicInformation)</td>
<td>June 2017</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
Pool infoleak reproduction

• Use a regular VM with guest Windows.

• Find out which driver makes the allocation leaked to user-mode (e.g. win32k.sys).

• Enable **Special Pools** for that module, reboot.

• Start PoC twice, observe a repeated marker byte where data is leaked (changes between runs).
D:\>VolumeDiskExtents.exe

00000000: 01 00 00 00 00 39 39 39 39 ....9999
00000008: 00 00 00 00 00 39 39 39 39 ....9999
00000010: 00 00 50 06 00 00 00 00 ..P.....
00000018: 00 00 a0 f9 09 00 00 00 ........
D:\>VolumeDiskExtents.exe

00000000: 01 00 00 00 00 2f 2f 2f 2f ....\\
00000008: 00 00 00 00 00 2f 2f 2f 2f ....\\
00000010: 00 00 50 06 00 00 00 00 ..P.....
00000018: 00 00 a0 f9 09 00 00 00 ........
Stack infoleak reproduction

• More difficult, there is no official / documented way of padding stack allocations with marker bytes.

• In a typical scenario, it may not be obvious that/which specific bytes are leaked.
  • Non-volatile, non-interesting values (e.g. zeros) often occupy a large portion of the stack.
  • Observations could differ in Microsoft’s test environment.

• Reliable proof of concept programs are highly desired.
  • To fully ensure that a bug is real also outside of Bochspwn environment.
  • To make the vendor’s life easier with analysis.
Stack spraying to the rescue

• A number of primitives exist in the Windows kernel to fill the kernel stack with controlled data.
  • Thanks to optimizations – local buffers used for „small” requests in many syscalls.
• Easy to identify: look for Nt* functions with large stack frames in IDA.
• My favorite: `nt!NtMapUserPhysicalPages`
  • Sprays up to 4096 bytes on x86 and 8192 bytes on x86-64.
  • Documented in „nt!NtMapUserPhysicalPages and Kernel Stack-Spraying Techniques” blog post in 2011.
1. Spray the kernel stack with an easily recognizable pattern.

2. Trigger the bug directly after, and observe the marker bytes at uninitialized offsets.
D:\>NtGdiGetRealizationInfo.exe

00000000: 10 00 00 00 03 01 00 00 ........
00000008: 2e 00 00 00 69 00 00 46 ....i..F
00000010: 41 41 41 41 41 41 41 41 AAAAAAAA
Quick digression: bugs without Bochspwn

• If memory marking can be used for bug demonstration, it can be used for discovery too.

• Basic idea:
  • Enable Special Pools for all common kernel modules.
  • Invoke tested system call twice, pre-spraying the kernel stack with a different byte each time.
  • Compare output in search of repeated patterns of differing bytes at common offsets.
NTSTATUS
NTAPI
NtQueryInformationProcess (  
   IN HANDLE ProcessHandle,  
   IN PROCESSINFOCLASS ProcessInformationClass,  
   OUT PVOID ProcessInformation,  
   IN ULONG ProcessInformationLength,  
   OUT PULONG ReturnLength OPTIONAL  
);
Fruitful idea

Windows Kernel stack memory disclosure in nt!NtQueryInformationJobObject (information class 12)
Project Member Reported by mjurczyk@gmail.com, Mar 17

Windows Kernel stack memory disclosure in nt!NtQueryInformationJobObject (information class 28)
Project Member Reported by mjurczyk@gmail.com, Mar 17

Windows Kernel stack memory disclosure in nt!NtQueryInformationTransaction (information class 1)
Project Member Reported by mjurczyk@gmail.com, Mar 17

Windows Kernel stack memory disclosure in nt!NtQueryInformationResourceManager (information class 0)
Project Member Reported by mjurczyk@gmail.com, Mar 20

Windows Kernel stack memory disclosure in nt!NtQueryInformationWorkerFactory (WorkerFactoryBasicInformation)
Project Member Reported by mjurczyk@gmail.com, Mar 21
Windows infoleak summary

• The problem seems to have remained almost completely unrecognized until just now (with a few exceptions).
  • The *invisibility* and non-obviousness of this bug class and no notion of privilege separation in C/C++ doesn’t really help.
  • It’s a fundamental issue, trivial to overlook but very difficult to get right in the code.
Windows infoleak summary

• Windows has a very loose approach to kernel→user data transfers.

• Tip of the iceberg, there may be many more instances of the bug lurking in the codebase.
  • Hundreds of `memcpy()` calls to user-mode exist, every one of them is a potential disclosure.
  • Especially those where size is user-controlled, but the amount of relevant data is fixed or otherwise limited.
Mitigation ideas (generic)

• Fully bug-proof: memset all stack and pool allocations when they are made/requested.
  • Would pretty much make the problem go away without any actual bug-fixing.
  • Easily implemented, but the overhead is probably too large?
  • Most kernel allocations don’t end up copied to user-mode, anyway.
That was fast!

Anyone notice my change to the Windows IO Manager to generically kill a class of info disclosure? BufferedIO output buffer is always zero'd.
Mitigation ideas (generic)

• More realistic:
  • Clear the kernel stack post-syscall (a.k.a. **PAX_MEMORY_STACKLEAK**).
    • Prevents cross-syscall leaks, which are probably the majority.
  • Add a new allocator function clearing returned memory regions.
  • Detect which allocations end up copied to user-mode and clear only those
    (automatically or by adding `memset()` calls in code manually).
Mitigation ideas (bug-specific)

• With Windows source code, Microsoft could take the whole Bochspwn idea to the next level:
  • Adding instrumentation at compile time → access to much more semantic information, e.g. better taint propagation (full vs. just `memcpy`).
  • More code coverage → more bugs found.
  • Static analysis easier to use to guide dynamic approaches and vice versa.
Closing remarks

• The Bochspwn approach can be also used to detect *regular* use of uninitialized memory, but the results are much harder to triage:
  • LOTS of false positives.
  • Lack of source code makes it very difficult to determine if an access is a bug and what its impact is.

• Leaking specific sensitive data from pool disclosures seems like an interesting subject and still needs research. 😊
Bochspwn vs. Linux
Tainting heap allocations

• MUCH more complex than on Windows:
  • A number of allocators, public and internal, with many variants: kmalloc, vmalloc, kmem_cache_alloc.
  • Allocator functions have different declarations.
  • Passing arguments via registers (regparm=3) means request information is not available on RET instruction.
  • kmem_cache’s have allocation sizes specified during cache creation.
  • kmem_cache’s may have constructors (tainting at a different time then returning region to caller).
  • Allocators may return pointers \( \leq 0x10 \) (not just NULL).
Variety of allocators (kmalloc/kmem_cache)

```c
void  *kmalloc(size_t, gfp_t);
void  *__kmalloc(size_t, gfp_t);
void  *kmalloc_order(size_t, gfp_t, unsigned int);
void  *kmalloc_order_trace(size_t, gfp_t, unsigned int);
void  *kmalloc_large(size_t, gfp_t);
void  *kzalloc(size_t, gfp_t);
struct kmem_cache  *kmem_cache_create(const char *, size_t, size_t,
                  unsigned long, void (*)(void *)));
void  *kmem_cache_alloc(struct kmem_cache *, gfp_t);
void  *kmem_cache_alloc_trace(struct kmem_cache *, gfp_t, size_t);
```
Variety of allocators (vmalloc)

```c
void *vmalloc(unsigned long);
void *vzalloc(unsigned long);
void *vmalloc_user(unsigned long);
void *vmalloc_node(unsigned long, int);
void *vzalloc_node(unsigned long, int);
void *vmalloc_exec(unsigned long);
void *vmalloc_32(unsigned long);
void *vmalloc_32_user(unsigned long);
void *__vmalloc(unsigned long, gfp_t, pgprot_t);
void *__vmalloc_node_range(unsigned long, unsigned long, unsigned long, unsigned long, gfp_t,
                           pgprot_t, unsigned long, int, const void *);
```
Variety of allocators

• Of course many of them call into each other, but in the end, we still had to hook into:
  • __kmalloc
  • kmalloc_order
  • __kmalloc_track_caller
  • __vmalloc_node
  • kmem_cache_create
  • kmem_cache_alloc
  • kmem_cache_alloc_trace

• ... and the corresponding free() routines, too.
• First three arguments to functions are passed through EAX, EDX, ECX.
  • Tried compiling the kernel without the option, but failed to boot. 😞

• Information about the allocation request and result is not available at the same time.

• Necessary to intercept execution twice: in the prologue and epilogue of the allocator.
; Attributes: bp-based frame

public __kmalloc
__kmalloc proc near

var 20= dword ptr -20h
var 1C= dword ptr -1Ch
var 18= dword ptr -18h
var 14= dword ptr -14h
var 10= dword ptr -10h

push ebp
mov ebp, esp
push edi
push esi
push ebx
sub esp, 20h

requests[ESP]["size"] = EAX
requests[ESP]["flags"] = ECX

set_taint(EAX, EAX + requests[ESP]["size"])
kmem_cache_{create,alloc}

- Dedicated mechanism for quick allocation of fixed-sized memory regions (e.g. structs).
  - `kmem_cache_create` creates a cache object (receives size, flags, constructor).
  - `kmem_cache_alloc` allocates memory from cache.
  - `kmem_cache_free` frees a memory region from cache.
  - `kmem_cache_destroy` destroys the cache object.

- We need to:
  - Maintain an up-to-date list of currently active caches.
  - Break on cache constructors to set taint on memory.
  - Break on allocators to set other metadata (e.g. caller’s EIP).
Propagating taint

- \texttt{CONFIG\_X86\_GENERIC=y} and \texttt{CONFIG\_X86\_USE\_3DNOW=n} sufficient to compile \texttt{memcpy()} into a combination of \texttt{rep movs\{d,b\}}.

```
.text:C13CC438  mov     ebx, ecx
.text:C13CC43D  mov     edi, eax
.text:C13CC43F  shr     ecx, 2
.text:C13CC442  mov     esi, edx
.text:C13CC444  rep movsd
.text:C13CC446  mov     ecx, ebx
.text:C13CC448  and     ecx, 3
.text:C13CC448  jz      short loc_C13CC44F
.text:C13CC44D  rep movsb
.text:C13CC44F  loc_C13CC44F:
.text:C13CC44F  pop     ebx
.text:C13CC450  pop     esi
.text:C13CC451  pop     edi
.text:C13CC452  pop     ebp
.text:C13CC453  ret
.text:C13CC453  memcpy  endp
```
Ubuntu 16.04 memory taint layout

0xc0000000

0xffffffff

60 minutes of run time, 20s. interval, boot + trinity fuzzer + linux test project

stack pages    heap pages
Other useful CONFIG options

- `CONFIG_DEBUG_INFO=y` to enable debugging symbols.
- `CONFIG_VMSPLIT_3G=y` to use the 3G/1G user/kernel split.
- `CONFIG_RANDOMIZE_BASE=n` to disable kernel ASLR.
- `CONFIG_X86_SMAP=n` to disable SMAP.
- `CONFIG_HARDENED_USERCOPY=n` to disable sanity checks unnecessary during instrumentation.
Detecting bugs – copy_to_user

Set `CONFIG_X86_INTEL_USERCOPY=n` to have `copy_to_user()` compiled to use `rep movs{d,b} instead of a sequence of mov.

```
.text:C13CCA2B  mov  ebx, ecx
.text:C13CCA2D  mov  edi, eax
.text:C13CCA2F  mov  esi, edx
.text:C13CCA31  cmp  ecx, 7
.text:C13CCA34  jbe  short loc:C13CCA4E
.text:C13CCA36  mov  ecx, edi
.text:C13CCA38  neg  ecx
.text:C13CCA3A  and  ecx, 7
.text:C13CCA3D  sub  ebx, ecx
.text:C13CCA3F  rep movsb
.text:C13CCA41  mov  ecx, ebx
.text:C13CCA43  shr  ecx, 2
.text:C13CCA46  and  ebx, 3
.text:C13CCA49  nop
.text:C13CCA4A  rep movsd
.text:C13CCA4C  mov  ecx, ebx
.text:C13CCA4E  loc:C13CCA4E:  ; CODE XREF: __copy_from_user_11_nocache_nozero+14!)
.text:C13CCA4E  rep movsb
.text:C13CCA50  pop  ebx
.text:C13CCA51  mov  eax, ecx
.text:C13CCA53  pop  esi
.text:C13CCA54  pop  edi
.text:C13CCA55  pop  ebp
.text:C13CCA56  ret
.text:C13CCA56  __copy_from_user_11_nocache_nozero endp
```
Detecting bugs – put_user

• Linux has a macro to write values of primitive types to userland memory.

• No internal `memcpy()`, so such leaks wouldn’t normally get detected.

• Each architecture has its own version of the macro, x86 too.

• Very difficult to modify the source to convert it to Bochspwn-compatible `rep movs`.

  • Various constructs passed as argument: constants, variables, structure fields, function return values etc.
The solution – temporary strict mode

```c
#define __put_user(x, ptr) \
({ \
   __typeof__(*(ptr)) __x; \
   ... \
   __asm("prefetcht1 (%eax)"); \
   __x = (x); \
   __asm("prefetcht2 (%eax)"); \
   ... 
})
```

1. Enable *strict mode* (for current ESP)
2. Evaluate expression written to userland
3. Disable *strict mode*
Strict mode

• **PREFETCH**\{1, 2\} instructions are effectively NOPs in Bochs.
  • Can be used as markers in the code, or „hypercalls”.

• In between **PREFETCH1** and **PREFETCH2**, all reads of uninitialized memory are reported as kernel→user leaks, if ESP is unchanged.
  • The code block only contains evaluation of the expression being written to ring-3.
  • Verifying ESP prevents polluting logs with reports from function calls, thread preemptions etc.

• **365** such constructs added to the vmlinux used by Bochspwn.
Strict mode as seen in IDA

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:C1027F72</td>
<td>prefetcht1 byte ptr [eax]</td>
</tr>
<tr>
<td>.text:C1027F75</td>
<td>mov eax, [ebp+var_B4]</td>
</tr>
<tr>
<td>.text:C1027F78</td>
<td>mov [ebp+var_AC], eax</td>
</tr>
<tr>
<td>.text:C1027F81</td>
<td>prefetcht2 byte ptr [eax]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:C1035910</td>
<td>prefetcht1 byte ptr [eax]</td>
</tr>
<tr>
<td>.text:C1035913</td>
<td>mov eax, [ebp+var_14]</td>
</tr>
<tr>
<td>.text:C1035916</td>
<td>mov edx, edi</td>
</tr>
<tr>
<td>.text:C1035918</td>
<td>call getreg</td>
</tr>
<tr>
<td>.text:C103591D</td>
<td>mov [ebp+var_10], eax</td>
</tr>
<tr>
<td>.text:C1035920</td>
<td>prefetcht2 byte ptr [eax]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text:C11ED784</td>
<td>prefetcht1 byte ptr [eax]</td>
</tr>
<tr>
<td>.text:C11ED787</td>
<td>mov eax, [ebp+var_18]</td>
</tr>
<tr>
<td>.text:C11ED78A</td>
<td>mov edx, [ebp+var_14]</td>
</tr>
<tr>
<td>.text:C11ED78D</td>
<td>mov [ebp+var_10], eax</td>
</tr>
<tr>
<td>.text:C11ED790</td>
<td>mov [ebp+var_C], edx</td>
</tr>
<tr>
<td>.text:C11ED793</td>
<td>prefetcht2 byte ptr [eax]</td>
</tr>
</tbody>
</table>
Keeping track of modules, symbolization etc.

Again, simple logic unchanged since the 2013 Bochspwn.
Bochspwn report

-----------------------------
found uninit-access of address f5733f38

======== READ of f5733f38 (4 bytes, kernel-->kernel), pc = f8aaf5c5
        [mov edi, dword ptr ds:[ebx+84]]

[Heap allocation not recognized]
Allocation origin: 0xc16b40bc: SYSC_connect at net/socket.c:1524
Shadow bytes: ff ff ff ff Guest bytes: bb bb bb bb

Stack trace:
#0 0xf8aaf5c5: llcp_sock_connect at net/nfc/llcp_sock.c:668
#1 0xc16b4141: SYSC_connect at net/socket.c:1536
#2 0xc16b4b26: SyS_connect at net/socket.c:1517
#3 0xc100375d: do_syscall_32_irqs_on at arch/x86/entry/common.c:330
        (inlined by) do_fast_syscall_32 at arch/x86/entry/common.c:392
Kernel debugging
Testing performed

• Instrumentation run on **Ubuntu 16.10 32-bit (kernel 4.8)**.

• Executed actions:
  • System boot up.
  • Logging in via SSH.
  • Starting a few command-line programs and reading from `/dev` and `/proc` pseudo-files.
  • Running **Linux Test Project** (LTP) unit tests.
  • Running the **Trinity + iknowthis** system call fuzzers.

• Coverage-guided fuzzing with **Syzkaller** sounds like a perfect fit, but it doesn’t actively support the x86 platform (currently only x86-64 and arm64).
Results!
Direct kernel→user disclosures

• Just **one** (1) minor bug!

• Disclosure of 7 uninitialized kernel stack bytes in the handling of specific IOCTLS in **ctl_ioctl** (drivers/md/dm-ioctl.c).

• **/dev/control/mapper** device, only accessible to root. 😞

• Issue discovered around April 20th, I was just about to report it a few days later, but...
dm ioctl: prevent stack leak in dm ioctl call

When calling a dm ioctl that doesn't process any data (IOCTL_FLAGS_NO_PARAMS), the contents of the data field in struct dm_ioctl are left initialized. Current code is incorrectly extending the size of data copied back to user, causing the contents of kernel stack to be leaked to user. Fix by only copying contents before data and allow the functions processing the ioctl to override.

Cc: stable@vger.kernel.org
Signed-off-by: Adrian Salido <salido@google.com>
Reviewed-by: Alasdair G Kergor <cak@redhat.com>
Signed-off-by: Mike Snitzer <snitzer@redhat.com>

Diffstat

1 files changed, 1 insertions, 1 deletions

diff --git a/drivers/md/dm_ioctl.c b/drivers/md/dm_ioctl.c
index 0058b86..ddda810 100644
--- a/drivers/md/dm_ioctl.c
+++ b/drivers/md/dm_ioctl.c
@@ -1840,7 +1840,7 @@ static int ctl_ioctl(uint command, struct dm_ioctl __user *user)
 if (r)
  goto out;
- param->data_size = sizeof(*param);
+ param->data_size = offsetof(struct dm_ioctl, data);
  r = fn(param, input_param_size);
 if (unlikely(param->flags & DM_BUFFER_FULL_FLAG))
  return -EINVAL;
Global strict mode

• Looks like Linux doesn’t have any direct, trivially reachable infoleaks to user-mode...

• Bochspwn can be used to also detect use of uninitialized memory, not just leaks.
  • With source code, it’s easy to analyze and understand each report.

• Let’s try our luck there?
### Use of uninitialized memory bugs

<table>
<thead>
<tr>
<th>Location</th>
<th>Fixed</th>
<th>Patch sent</th>
<th>Found externally</th>
<th>Memory type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>llcp_sock_connect in net/nfc/llcp_sock.c</code></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (after Bochspwn)</td>
<td>Stack</td>
</tr>
<tr>
<td><code>bind()</code> and <code>connect()</code> handlers in multiple sockets (bluetooth, caif, iucv, nfc, unix)`</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Stack</td>
</tr>
<tr>
<td><code>deprecated_sysctl_warning in kernel/sysctl_binary.c</code></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes (after Bochspwn)</td>
<td>Stack</td>
</tr>
<tr>
<td><code>SYSC_epoll_ctl in fs/eventpoll.c</code></td>
<td>Yes</td>
<td>n/a</td>
<td>Yes</td>
<td>Stack</td>
</tr>
<tr>
<td><code>devkmsg_read in kernel/printk/printk.c</code></td>
<td>Yes, on 4.10+ kernels</td>
<td>n/a</td>
<td>Kind of (code area refactored)</td>
<td>Heap</td>
</tr>
<tr>
<td><code>dnrmg_receive_user_skb in net/decnet/netfilter/dn_rtmmsg.c</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Heap</td>
</tr>
<tr>
<td><code>nfnetlink_rcv in net/netfilter/nfnetlink.c</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Heap</td>
</tr>
<tr>
<td><code>ext4_update_bh_state in fs/ext4/inode.c</code></td>
<td>Yes</td>
<td>n/a</td>
<td>Yes</td>
<td>Stack</td>
</tr>
<tr>
<td><code>nl_fib_lookup in net/ipv4/fib_frontend.c</code></td>
<td>Yes</td>
<td>n/a</td>
<td>Yes</td>
<td>Heap</td>
</tr>
<tr>
<td><code>fuse_release_common in fs/fuse/file.c</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Heap</td>
</tr>
<tr>
<td><code>apply_alternatives in arch/x86/kernel/alternative.c</code></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Stack</td>
</tr>
<tr>
<td><code>__bpf_prog_run in kernel/bpf/core.c</code></td>
<td>n/a</td>
<td>n/a</td>
<td>Yes</td>
<td>Stack</td>
</tr>
<tr>
<td><code>crng_reseed in drivers/char/random.c</code></td>
<td>n/a</td>
<td>n/a</td>
<td>No</td>
<td>Stack</td>
</tr>
<tr>
<td><code>unmapped_area_topdown in mm/mmap.c</code></td>
<td>n/a</td>
<td>n/a</td>
<td>No</td>
<td>Stack</td>
</tr>
</tbody>
</table>

**Bonus:** A local kernel DoS (NULL Pointer Dereference) while experimenting with another bug.
Results summary

• Even though the list is long, the bugs are mostly insignificant.
  • For example allow to answer „is an uninitialized byte on kernel stack equal to 0?”
  • One regular memory disclosure vulnerability in AF_NFC.

• False positives are bound to happen, and sometimes they are true positives that are just „working as intended”.

• Good validation that the approach does work, but there just aren’t more obvious issues to be found.
KernelMemorySanitizer

• Linux kernel development is very rapid, bugs get fixed every day.

• Most collisions happened with **KMSAN**.
  • Currently under development by Alexander Potapenko.
  • Run-time instrumentation added by compiler to detect use of uninitialized memory.
  • Twin project of KernelAddressSanitizer, MemorySanitizer (for user-mode) and all other Sanitizers.

• The correct long-time approach to the problem in Linux.
Conclusions

• The Linux community has been on top of the problem for the last few years.

• Seemingly hardly any easy infoleaks left at all at this point.
  • Some uses of uninit memory, but even these are not trivial to find.

• Even when bugs show up, they are rather short-lived.

• Most remaining bugs should be swept off by KMSAN in the near future.
Future work
Future work for Bochspwn

• Run further iterations on Windows.
  • Triage and get a better understanding of some of the uninitialized reads detected by Bochspwn strict-mode.

• Look into improving code coverage.
  • Neverending story. Syzkaller does pretty well on Linux, no sensible equivalent for Windows.

• Improve taint propagation logic beyond just rep movs.

• Implement support for 64-bit guest systems.
  • Opens many doors – new bugs, more coverage, etc.
Future work for Bochspwn

• Taint-less approaches:
  • Poison stack and heap/pools with magic bytes, log all kernel→user writes with these bytes, review all reports for bugs.
    • Approach used (to an extent) by fanxiaocao and pjf.
  • Generalize for two or more such sessions with different marker bytes. For every write location which always has the marker at specific offset(s), that’s a bug!
• Addresses the problem of non-ideal taint propagation (for other tradeoffs).
Other (crazy) ideas

• Recompilation or binary rewriting to make the kernels transfer data exclusively with `movs{b,d}` instructions? 😊

• Apply the concept to other data sinks than just user-mode memory.
  - Outgoing network traffic.
  - File system metadata.
  - Output files saved by desktop applications.

• Other security domains? Inter-process communication, virtualization.
Thanks!

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