Bochspwn Revolutions

Further Advancements in Detecting Kernel Infoleaks with x86 Emulation

Mateusz Jurczyk (@j00ru)

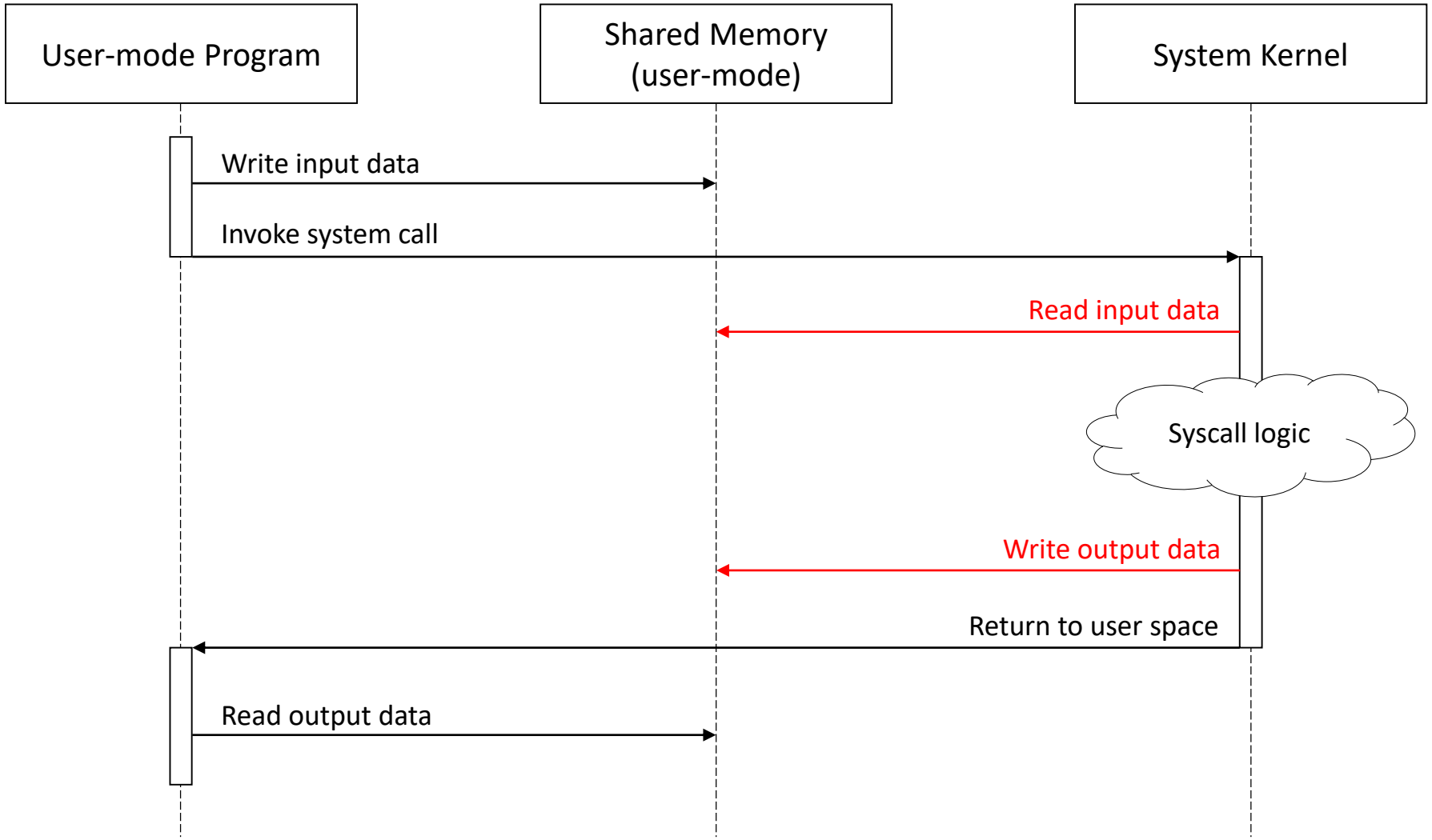
INFILTRATE 2018, Miami

Agenda

- Short recap of kernel infoleaks and Bochspwn Reloaded for x86 guests
- Implementing x64 guest support – challenges and results
- Detecting memory disclosure in file systems
- Identifying KASLR bypasses through *double-writes*
- Bonus – memory disclosure in PDB files
- Future work and conclusions

User $\leftrightarrow$ kernel communication

Life of a system call



In a perfect world...

- Within the scope of a single system call, every byte in ring 3 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 (Bochspwn 2013)
- Double writes
- Read-after-write conditions
- Unprotected accesses to user-mode pointers
- …

The subject of this talk

Written to at most once, securely,

only with data intended for user-mode

Causes of kernel memory disclosure

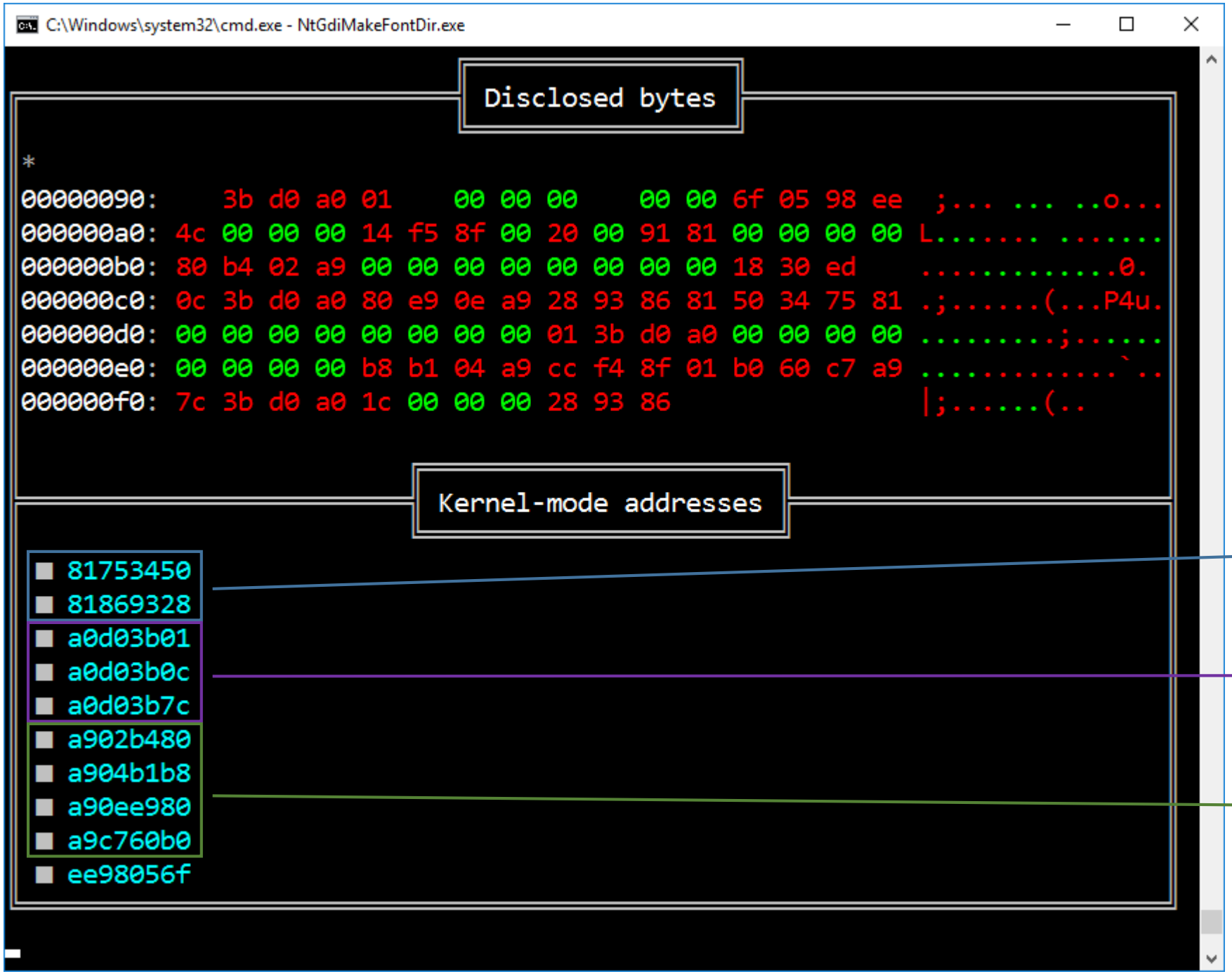
- **Uninitialized variables** of primitive types
- **Uninitialized structure fields** (e.g. *reserved*)
- **Padding bytes** in the middle and at the end of structures
- **Unused bytes** in unions due to fields of different sizes
- Partially filled **fixed-sized arrays**
- **Arbitrary** syscall **output buffer sizes**

Contributing factors

- No default initialization of dynamic and automatic objects
  - With some exceptions, mostly on Linux
- No visible consequences of leaks for developers or end users
  - Little chance to discover or learn about them by accident
- Leaks hidden behind system API
  - Discarded by system DLLs and invisible to legitimate applications

Severity

- „Just” local info leaks, no memory corruption or remote exploitation involved by nature
- Actual severity depends on what is leaked out of the kernel
- Mostly useful as single links in LPE exploit chains
  - Kernel-mode addresses and stack cookies are easiest to leak
  - Other sensitive information from the heap/pools may be potentially disclosed, too


C:\Windows\system32\cmd.exe - NtGdiMakeFontDir.exe
Disclosed bytes
*
00000090: 3b d0 a0 01 00 00 00 00 00 6f 05 98 ee ;... ... ..o...
000000a0: 4c 00 00 00 14 f5 8f 00 20 00 91 81 00 00 00 00 L..... .....
000000b0: 80 b4 02 a9 00 00 00 00 00 00 00 00 18 30 ed .....0.
000000c0: 0c 3b d0 a0 80 e9 0e a9 28 93 86 81 50 34 75 81 .;.....(...P4u.
000000d0: 00 00 00 00 00 00 00 00 01 3b d0 a0 00 00 00 00 .....;.....
000000e0: 00 00 00 00 b8 b1 04 a9 cc f4 8f 01 b0 60 c7 a9 .....`..
000000f0: 7c 3b d0 a0 1c 00 00 00 28 93 86 |;.....(..
Kernel-mode addresses
81753450
81869328
a0d03b01
a0d03b0c
a0d03b7c
a902b480
a904b1b8
a90ee980
a9c760b0
ee98056f
Kernel code addresses (ntoskrnl.exe)
Kernel stack addresses
Non-paged pool addresses

Prior work – Windows

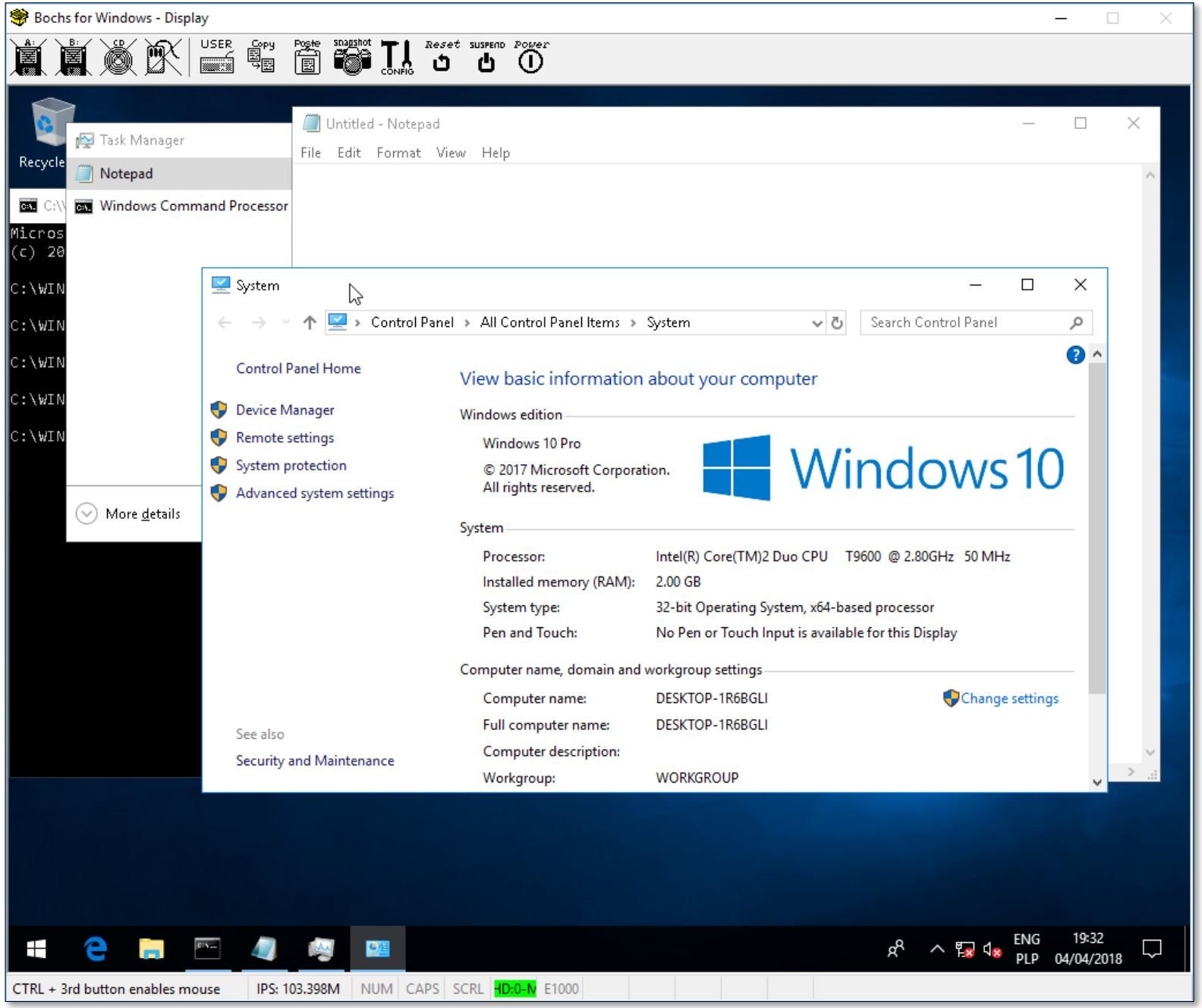
- Very little done for the 30 years of system existence, up until 2015
  1. **P0 Issue #480** (`win32k!NtGdiGetTextMetrics`, CVE-2015-2433), Matt Tait, July 2015. Collision with Hacking Team
  2. ***Leaking Windows Kernel Pointers***, Wandering Glitch, RuxCon, October 2016
     - Eight kernel uninitialized memory disclosure bugs fixed in 2015
  3. ***Automatically Discovering Windows Kernel Information Leak Vulnerabilities***, fanxiaocao and pjf of IceSword Lab (Qihoo 360), June 2017
  4. **Zeroing buffered I/O output buffer in Windows**, Joseph Bialek, June 2017

Prior work – Linux

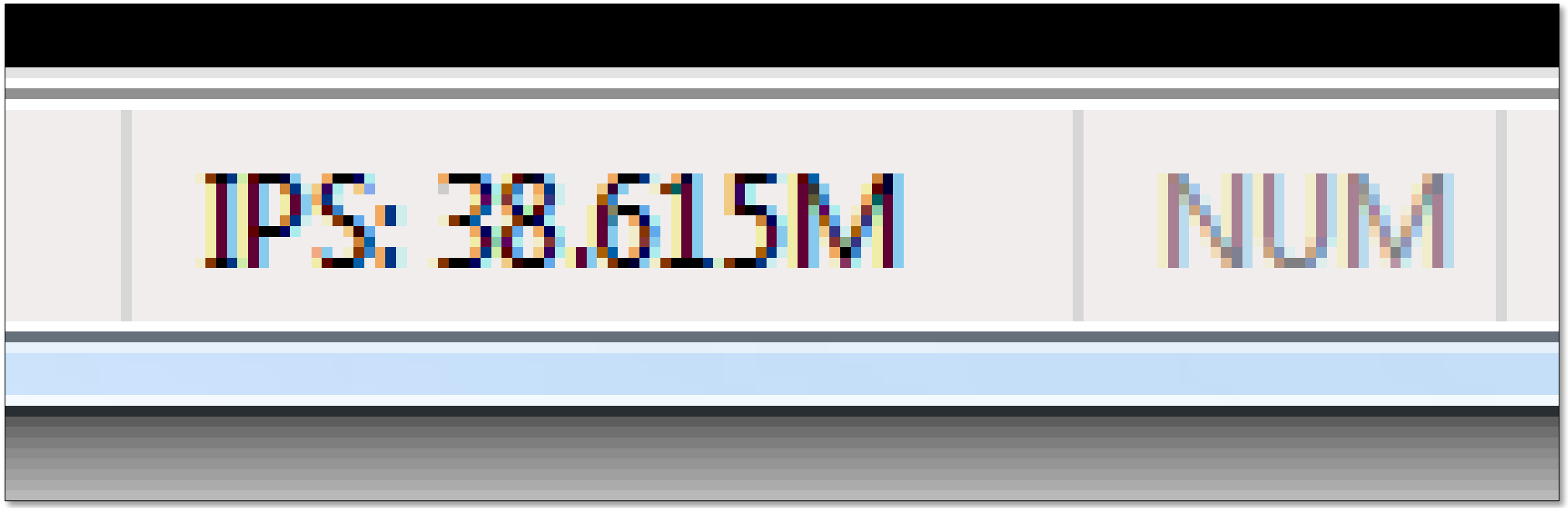
- Detection
  - Individual findings, e.g. over 25 bugs by Rosenberg and Oberheide (2009-2010), over 20 bugs by Krause (2013)
  - `-Wuninitialized`
  - kmemcheck
  - Static analysis with Coccinelle (Peiró, 2014-2016)
  - UniSan (Lu et al., 2016)
  - KernelMemorySanitizer (Potapenko, 2017–…)
- Mitigation
  - Mainline: `CONFIG_PAGE_POISONING`, `CONFIG_DEBUG_SLAB`
  - grsecurity/PaX: `PAX_MEMORY_SANITIZE`, `PAX_MEMORY_STRUCTLEAK`, `PAX_MEMORY_STACKLEAK`
  - Secure deallocation (Chow et al., 2005)
  - Split Kernel (Kurmus and Zippel, 2014)
  - UniSan (Lu et al., 2016)
  - SafeInit (Milburn et al., 2017)

Bochspwn Reloaded (2017)

- Bochs is a full IA-32 and AMD64 PC emulator
  - CPU plus all basic peripherals, i.e. a whole emulated computer
- Written in C++
- Correctly hosts all common operating systems
- Provides an extensive instrumentation API


Bochs for Windows - Display
USER
Copy
Paste
Snapshot
CONFIG
Reset
SUSPEND
Power
Recycle
Task Manager
Notepad
Windows Command Processor
More details
Micros
(c) 20
C:\WIN
C:\WIN
C:\WIN
C:\WIN
C:\WIN
Untitled - Notepad
File Edit Format View Help
System
Control Panel › All Control Panel Items › System
Search Control Panel
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
View basic information about your computer
Windows edition
Windows 10 Pro
© 2017 Microsoft Corporation.
All rights reserved.
Windows 10
System
Processor: Intel(R) Core(TM)2 Duo CPU T9600 @ 2.80GHz 50 MHz
Installed memory (RAM): 2.00 GB
System type: 32-bit Operating System, x64-based processor
Pen and Touch: No Pen or Touch Input is available for this Display
Computer name, domain and workgroup settings
Computer name: DESKTOP-1R6BGLI
Change settings
Full computer name: DESKTOP-1R6BGLI
Computer description:
Workgroup: WORKGROUP
See also
Security and Maintenance
ENG
PLP
19:32
04/04/2018
CTRL + 3rd button enables mouse
IPS: 103.398M
NUM
CAPS
SCRL
HD:0-M
E1000

Performance (short story)



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

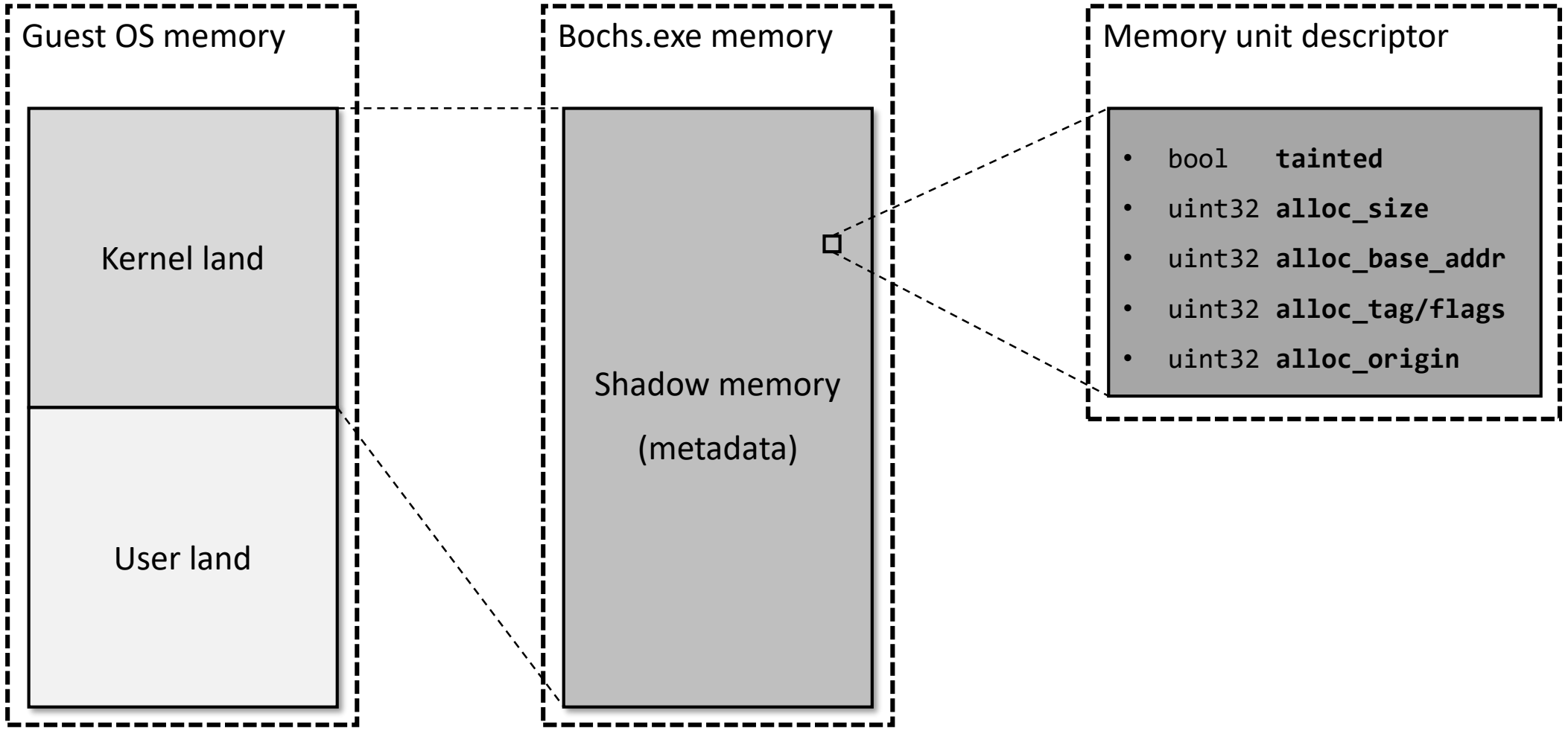
Bochs instrumentation callbacks

- BX_INSTR_INIT_ENV
- BX_INSTR_EXIT_ENV
- **BX_INSTR_INITIALIZE**
- **BX_INSTR_EXIT**
- BX_INSTR_RESET
- BX_INSTR_HLT
- BX_INSTR_MWAIT
- BX_INSTR_DEBUG_PROMPT
- BX_INSTR_DEBUG_CMD
- BX_INSTR_CNEAR_BRANCH_TAKEN
- BX_INSTR_CNEAR_BRANCH_NOT_TAKEN
- BX_INSTR_UCNEAR_BRANCH
- BX_INSTR_FAR_BRANCH
- BX_INSTR_OPCODE
- BX_INSTR_EXCEPTION
- **BX_INSTR_INTERRUPT**
- BX_INSTR_HWINTERRUPT
- BX_INSTR_CLFLUSH
- BX_INSTR_CACHE_CNTRL
- BX_INSTR_TLB_CNTRL
- BX_INSTR_PREFETCH_HINT
- **BX_INSTR_BEFORE_EXECUTION**
- **BX_INSTR_AFTER_EXECUTION**
- BX_INSTR_REPEAT_ITERATION
- **BX_INSTR_LIN_ACCESS**
- BX_INSTR_PHY_ACCESS
- BX_INSTR_INP
- BX_INSTR_INP2
- BX_INSTR_OUTP
- **BX_INSTR_WRMSR**
- BX_INSTR_VMEXIT

Core logic

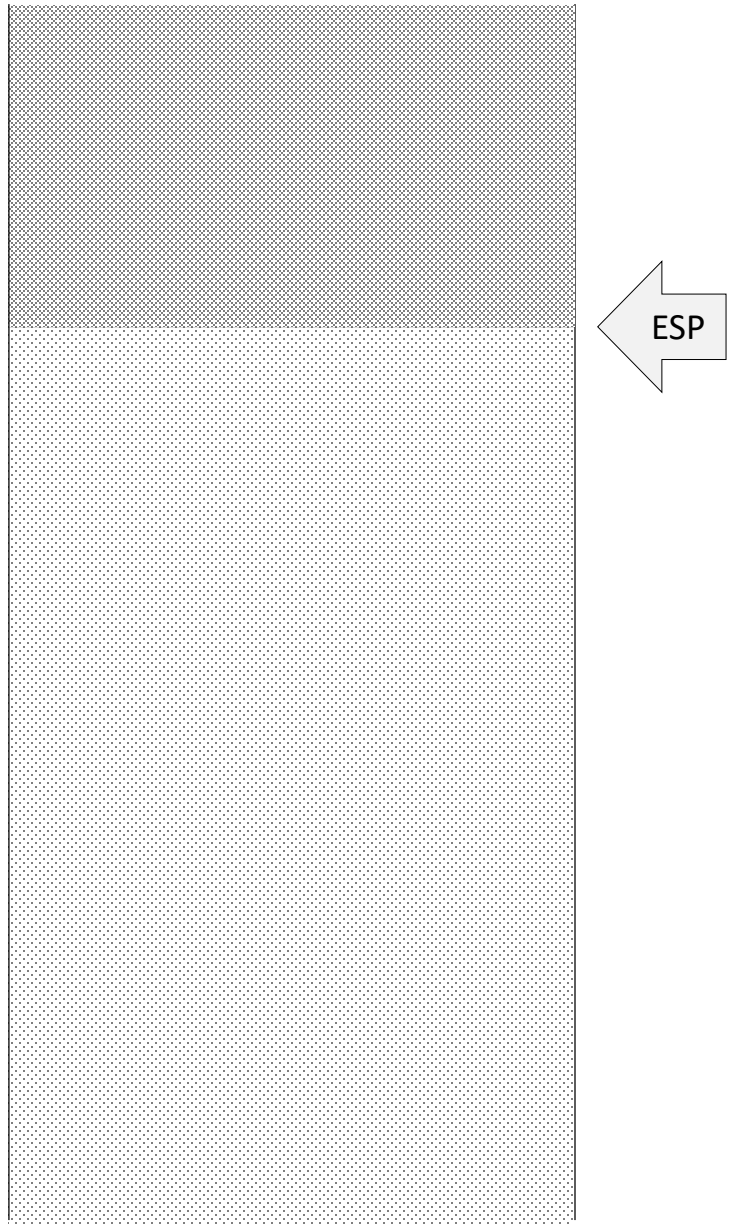
- Taint tracking for the entire kernel address space
- Primary functionality:
    1. Set taint on new allocations
    2. Remove taint on all memory writes
    3. Propagate taint in memory on `memcpy()`
    4. Detect copying of tainted memory to user-mode

Shadow memory (32-bit)



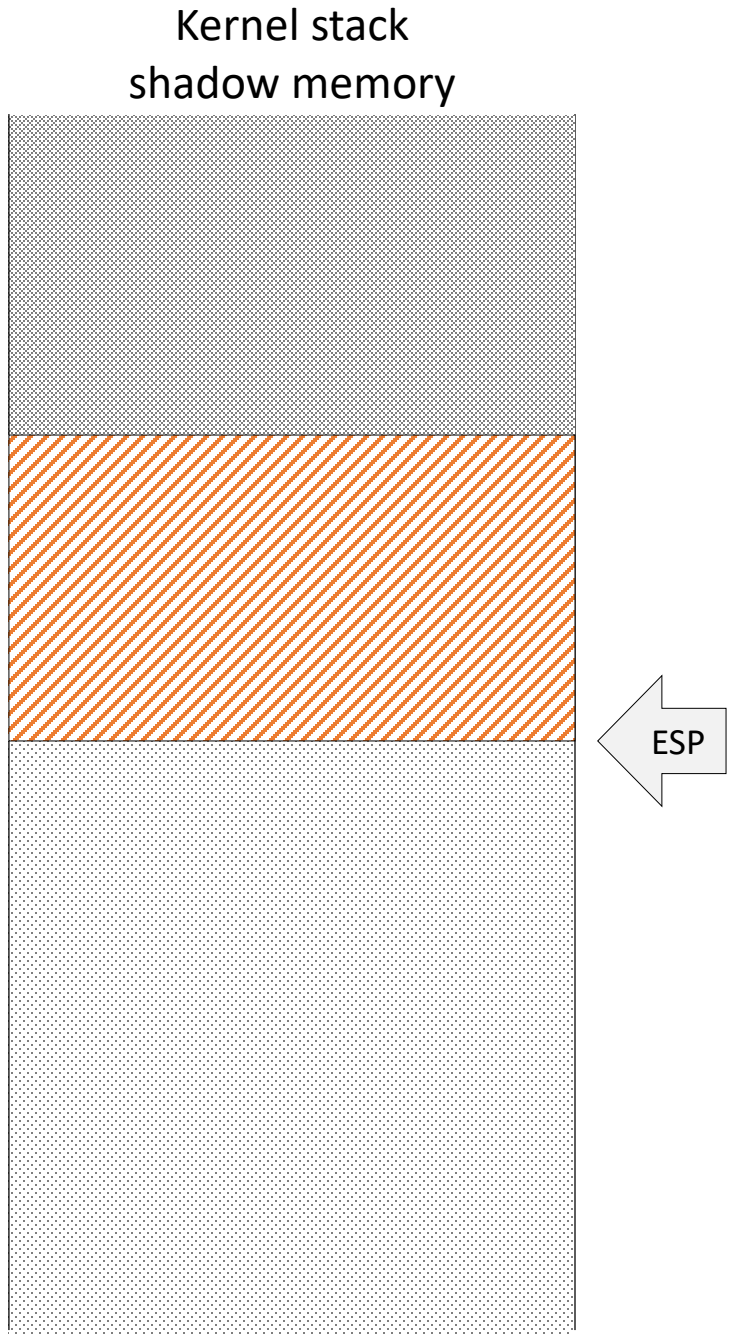
Taint tracking in action

```
1  typedef struct _SYSCALL_OUTPUT {
2    DWORD Sum;
3    QWORD LargeSum;
4  } SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;
5
6  NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
7    SYSCALL_OUTPUT OutputStruct;
8
9    OutputStruct.Sum = InputValue + 2;
10   OutputStruct.LargeSum = 0;
11
12   RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
13   return STATUS_SUCCESS;
14 }
```



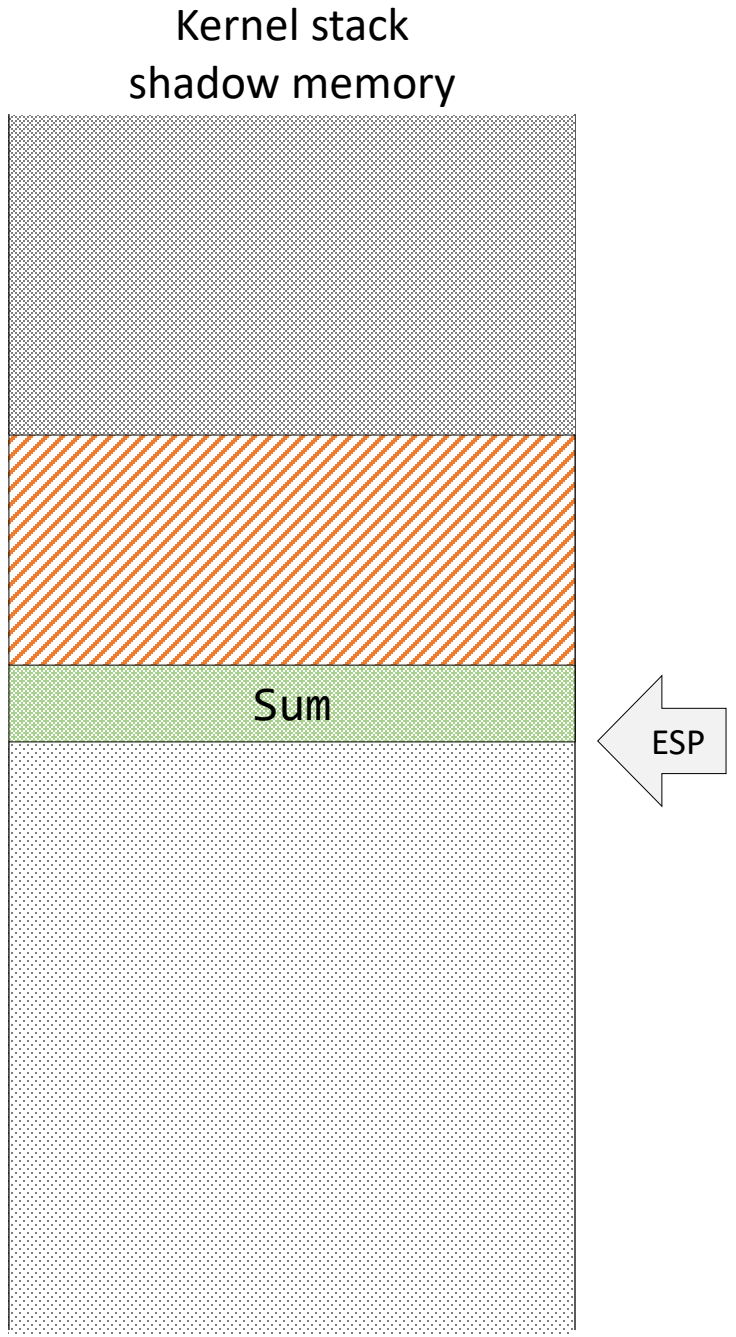
Taint tracking in action

```
1  typedef struct _SYSCALL_OUTPUT {
2    DWORD Sum;
3    QWORD LargeSum;
4  } SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;
5
6  NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
7    SYSCALL_OUTPUT OutputStruct;
8
9    OutputStruct.Sum = InputValue + 2;
10   OutputStruct.LargeSum = 0;
11
12   RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
13   return STATUS_SUCCESS;
14 }
```



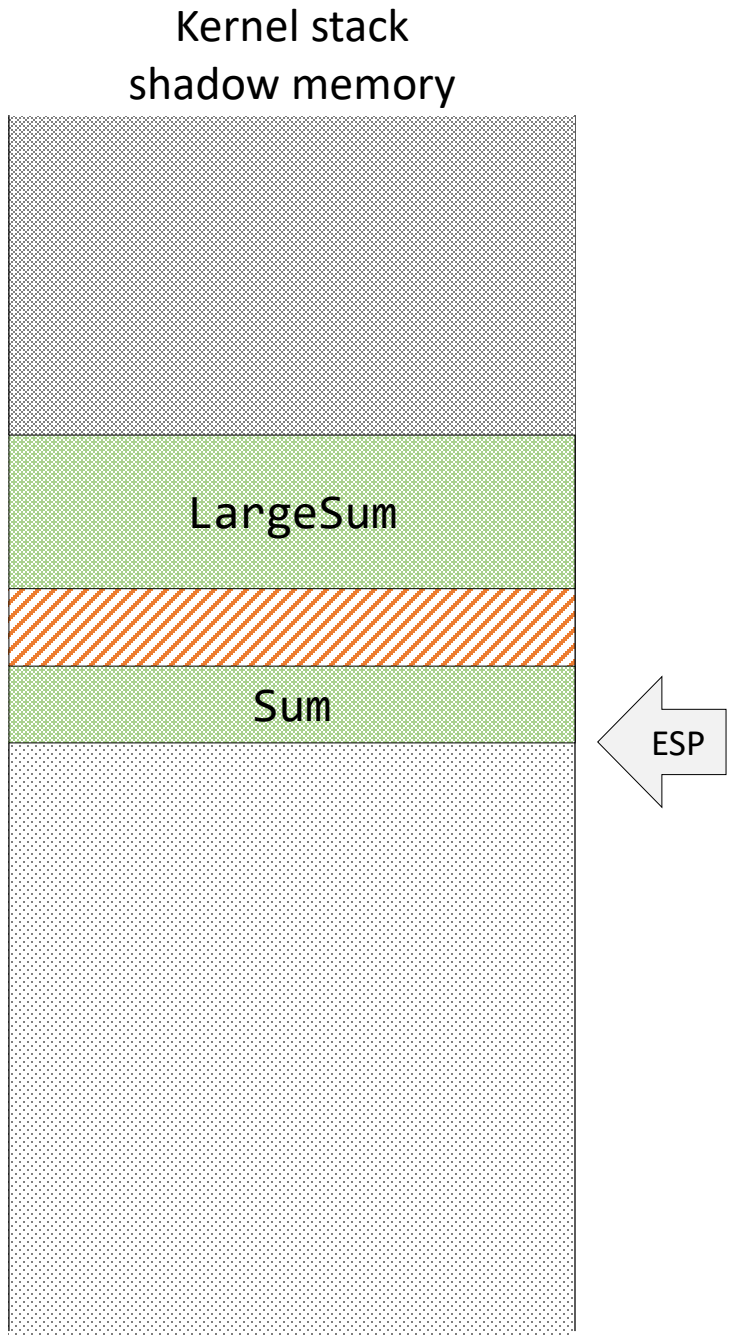
Taint tracking in action

```
1  typedef struct _SYSCALL_OUTPUT {
2    DWORD Sum;
3    QWORD LargeSum;
4  } SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;
5
6  NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
7    SYSCALL_OUTPUT OutputStruct;
8
9    OutputStruct.Sum = InputValue + 2;
10   OutputStruct.LargeSum = 0;
11
12   RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
13   return STATUS_SUCCESS;
14 }
```



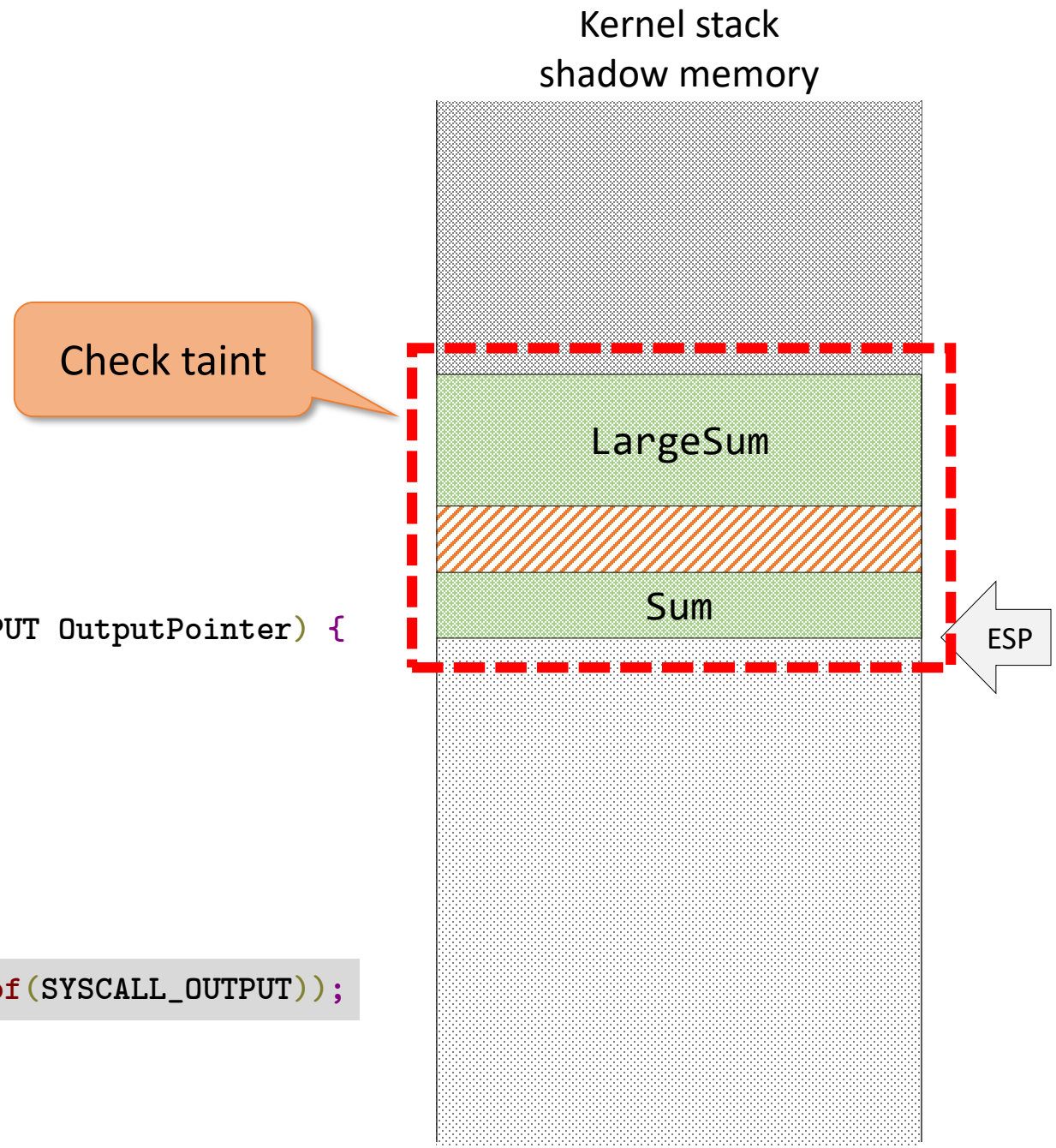
Taint tracking in action

```
1  typedef struct _SYSCALL_OUTPUT {
2    DWORD Sum;
3    QWORD LargeSum;
4  } SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;
5
6  NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
7    SYSCALL_OUTPUT OutputStruct;
8
9    OutputStruct.Sum = InputValue + 2;
10   OutputStruct.LargeSum = 0;
11
12   RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
13   return STATUS_SUCCESS;
14 }
```



Taint tracking in action

```
1  typedef struct _SYSCALL_OUTPUT {
2    DWORD Sum;
3    QWORD LargeSum;
4  } SYSCALL_OUTPUT, *PSYSCALL_OUTPUT;
5
6  NTSTATUS NtSmallSum(DWORD InputValue, PSYSCALL_OUTPUT OutputPointer) {
7    SYSCALL_OUTPUT OutputStruct;
8
9    OutputStruct.Sum = InputValue + 2;
10   OutputStruct.LargeSum = 0;
11
12   RtlCopyMemory(OutputPointer, &OutputStruct, sizeof(SYSCALL_OUTPUT));
13   return STATUS_SUCCESS;
14 }
```



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

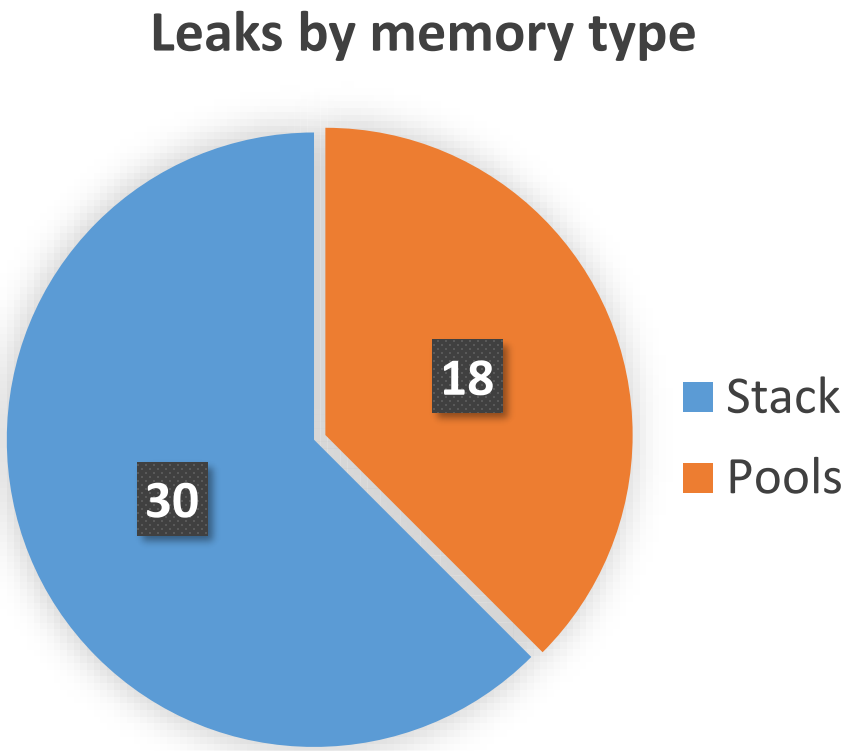
Windows bug report

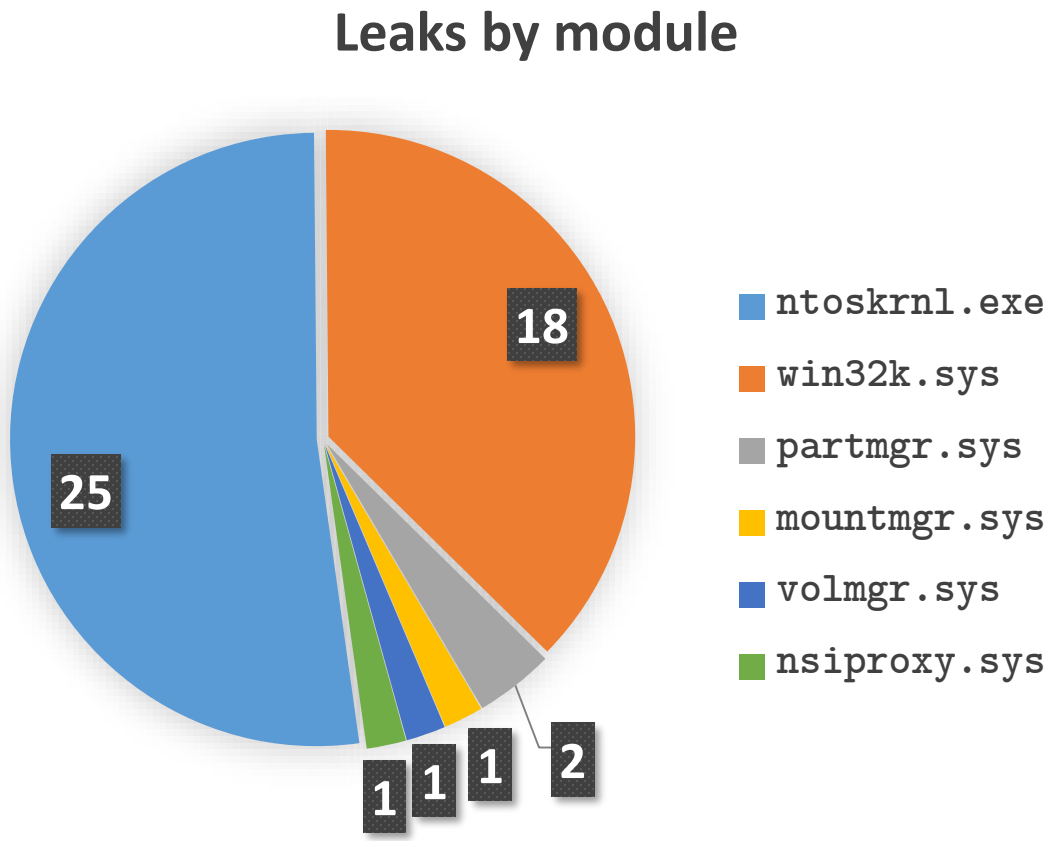
```
------------------------------ found uninit-access of address 94447d04
[pid/tid: 000006f0/00000740] {    explorer.exe}
       READ of 94447d04 (4 bytes, kernel--->user), pc = 902df30f
       [ rep movsd dword ptr es:[edi], dword ptr ds:[esi] ]
[Pool allocation not recognized]
Allocation origin: 0x90334988 ((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)
```

Testing Windows

- 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**
  - Running ~30 tests from my custom `NtQuery` test suite

Windows 32-bit results summary: 48 CVE





Windows 32-bit pool disclosures

CVE	Component	Fix date	Number of leaked bytes
CVE-2017-0258	nt!SepInitSystemDacls	May 2017	8
CVE-2017-0259	nt!NtTraceControl (EtwpSetProviderTraits)	May 2017	60
CVE-2017-8462	nt!NtQueryVolumeInformationFile (FileFsVolumeInformation)	June 2017	1
CVE-2017-8469	partmgr, IOCTL_DISK_GET_DRIVE_LAYOUT_EX	June 2017	484
CVE-2017-8484	win32k!NtGdiGetOutlineTextMetricsInternalW	June 2017	5
CVE-2017-8488	mountmgr, IOCTL_MOUNTMGR_QUERY_POINTS	June 2017	14
CVE-2017-8489	WMIDataDevice, IOCTL 0x224000 (WmiQueryAllData)	June 2017	72
CVE-2017-8490	win32k!NtGdiEnumFonts	June 2017	6672
CVE-2017-8491	volmgr, IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS	June 2017	8
CVE-2017-8492	partmgr, IOCTL_DISK_GET_DRIVE_GEOMETRY_EX	June 2017	4
CVE-2017-8564	nsiproxy, IOCTL 0x120007 (NsiGetParameter)	July 2017	13
CVE-2017-0299	nt!NtNotifyChangeDirectoryFile	August 2017	2
CVE-2017-8680	win32k!NtGdiGetGlyphOutline	September 2017	Arbitrary
CVE-2017-11784	nt!RtlpCopyLegacyContextX86	October 2017	192
CVE-2017-11785	nt!NtQueryObject (ObjectNameInformation)	October 2017	56
CVE-2017-11831	nt!NtQueryDirectoryFile (luafv!LuafvCopyDirectoryEntry)	November 2017	25
CVE-2018-0746	nt!NtQuerySystemInformation (MemoryTopologyInformation)	January 2018	12
CVE-2018-0972	nt!NtQueryInformationTransactionManager (TransactionManagerRecoveryInformation)	April 2018	8

Windows 32-bit stack disclosures

CVE	Component	Fix date	Number of leaked bytes
CVE-2017-0167	`win32kfull!SfnINLPUAHDRAWMENUITEM`	April 2017	20
CVE-2017-0245	`win32k!xxxClientLpkDrawTextEx`	May 2017	4
CVE-2017-0300	`nt!NtQueryInformationWorkerFactory (WorkerFactoryBasicInformation)`	June 2017	5
CVE-2017-8470	`win32k!NtGdiExtGetObjectW`	June 2017	50
CVE-2017-8471	`win32k!NtGdiGetOutlineTextMetricsInternalW`	June 2017	4
CVE-2017-8472	`win32k!NtGdiGetTextMetricsW`	June 2017	7
CVE-2017-8473	`win32k!NtGdiGetRealizationInfo`	June 2017	8
CVE-2017-8474	`DeviceApi (nt!PiDqIrpQueryGetResult, nt!PiDqIrpQueryCreate, nt!PiDqQueryCompletePendedIrp)`	June 2017	8
CVE-2017-8475	`win32k!ClientPrinterThunk`	June 2017	20
CVE-2017-8476	`nt!NtQueryInformationProcess (ProcessVmCounters)`	June 2017	4
CVE-2017-8477	`win32k!NtGdiMakeFontDir`	June 2017	104
CVE-2017-8478	`nt!NtQueryInformationJobObject (JobObjectNotificationLimitInformation)`	June 2017	4
CVE-2017-8479	`nt!NtQueryInformationJobObject (JobObjectMemoryUsageInformation)`	June 2017	16
CVE-2017-8480	`nt!NtQueryInformationTransaction (TransactionPropertiesInformation)`	June 2017	6
CVE-2017-8481	`nt!NtQueryInformationResourceManager (ResourceManagerBasicInformation)`	June 2017	2
CVE-2017-8482	`nt!KiDispatchException`	June 2017	32
CVE-2017-8485	`nt!NtQueryInformationJobObject (JobObjectBasicLimitInformation, JobObjectExtendedLimitInformation)`	June 2017	8

Windows 32-bit stack disclosures (cont'd)

CVE	Component	Fix date	Number of leaked bytes
CVE-2017-8677	`win32k!NtGdiHLSurfGetInformation (information class 3)`	September 2017	8
CVE-2017-8678	`win32k!NtQueryCompositionSurfaceBinding`	September 2017	4
CVE-2017-8681	`win32k!NtGdiGetPhysicalMonitorDescription`	September 2017	128
CVE-2017-8684	`win32k!NtGdiGetFontResourceInfoInternalW`	September 2017	88
CVE-2017-8685	`win32k!NtGdiEngCreatePalette`	September 2017	1024
CVE-2017-8687	`win32k!NtGdiDoBanding`	September 2017	8
CVE-2017-11853	`win32k!xxxSendMenuSelect`	November 2017	12
CVE-2018-0745	`nt!NtQueryInformationProcess (ProcessEnergyValues)`	January 2018	4
CVE-2018-0747	`nt!RawMountVolume`	January 2018	4
CVE-2018-0832	`nt!RtlpCopyLegacyContextX86`	February 2018	4
CVE-2018-0969	`nt!NtQueryAttributesFile`	April 2018	4
CVE-2018-0970	`nt!NtQueryVolumeInformationFile`	April 2018	4/16
CVE-2018-0975	`nt!NtQueryFullAttributesFile`	April 2018	4/56

Linux bug 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
```

Testing Linux

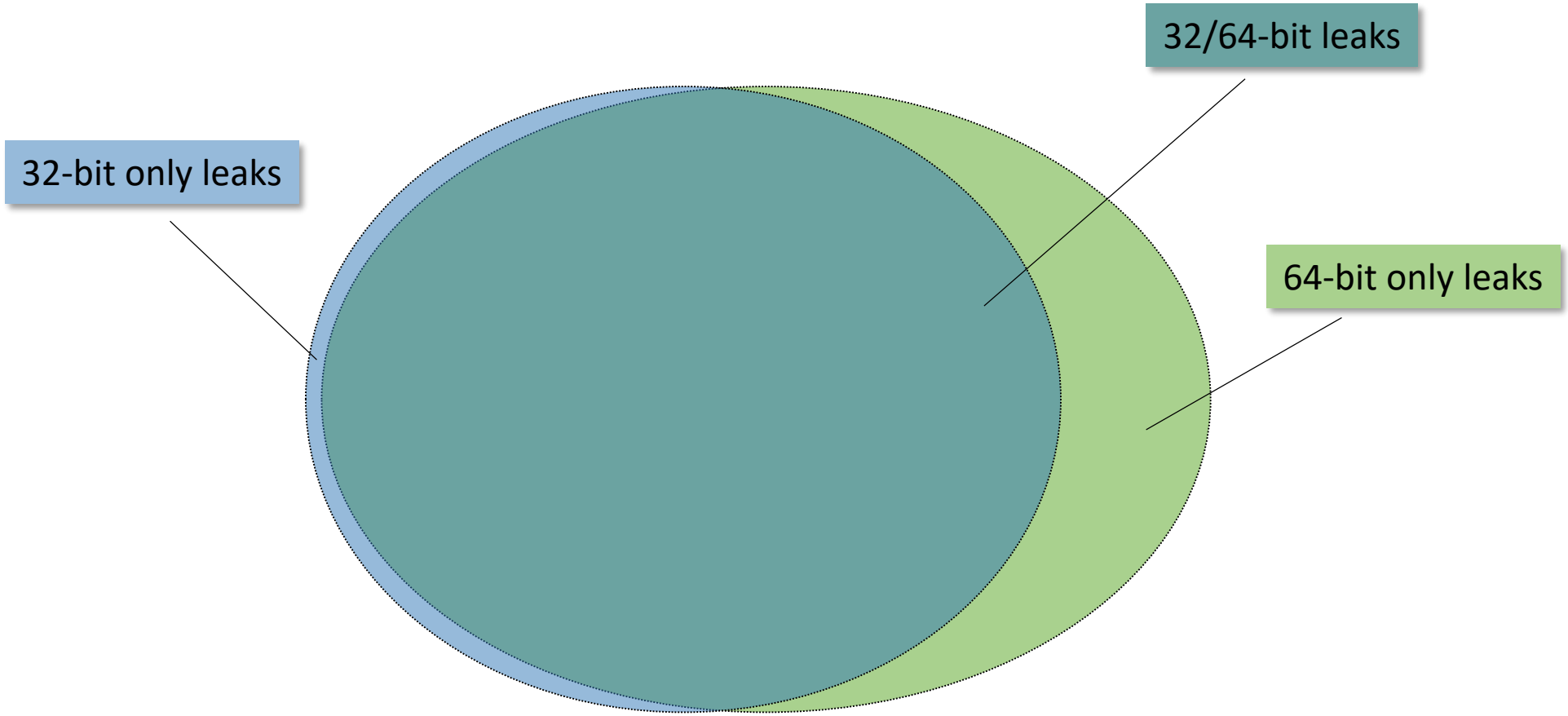
- Instrumentation run on **Ubuntu 16.10 32-bit** (**kernel 4.8**)
- Executed actions:
  - System boot up
  - Logging in via SSH
  - Starting a few command-line programs and reading from **/dev** and **/proc** pseudo-files
  - Running **Linux Test Project** (LTP) unit tests
  - Running the **Trinity** + **iknowthis** system call fuzzers

Use of uninitialized memory on Linux

Location	Infoleak	Patch sent	Found externally	Fix commit	Memory
net/nfc/llcp_sock.c	✔ (NFC-only)	✔	After Bochspwn	608c4adfca	Stack
drivers/md/dm-ioctl.c	✔ (root-only)		Before Bochspwn	4617f564c0	Stack
net/bluetooth/l2cap_sock.c net/bluetooth/rfcomm/sock.c net/bluetooth/sco.c		✔		d2ecfa765d	Stack
net/caif/caif_socket.c		✔		20a3d5bf5e	Stack
net/iucv/af_iucv.c		✔		e3c42b61ff	Stack
net/nfc/llcp_sock.c		✔		f6a5885fc4	Stack
net/unix/af_unix.c		✔		defbcf2dec	Stack
kernel/sysctl_binary.c		✔	After Bochspwn	9380fa60b1	Stack
fs/eventpoll.c			Before Bochspwn	c857ab640c	Stack
kernel/printk/printk.c			Before Bochspwn (code refactor)	5aa068ea40	Heap
net/decnet/netfilter/dn_rtmsg.c		✔		dd0da17b20	Heap
net/netfilter/nfnetlink.c		✔		f55ce7b024	Heap
fs/ext4/inode.c			Before Bochspwn	2a527d6858	Stack
net/ipv4/fib_frontend.c			Before Bochspwn	c64c0b3cac	Heap
fs/fuse/file.c		✔		68227c03cb	Heap
arch/x86/kernel/alternative.c		✔		fc152d22d6	Stack

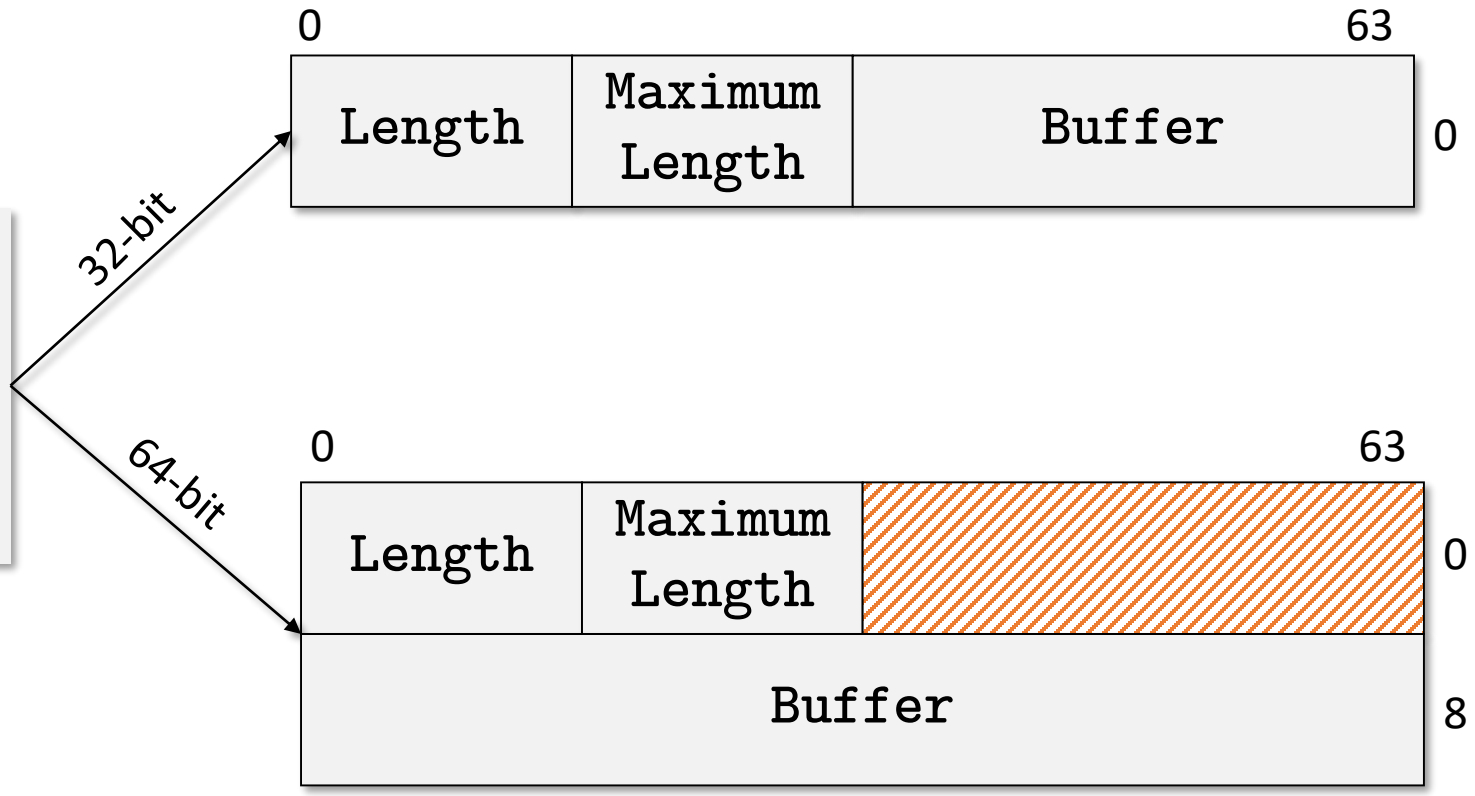
64-bit Windows guest support

Motivation



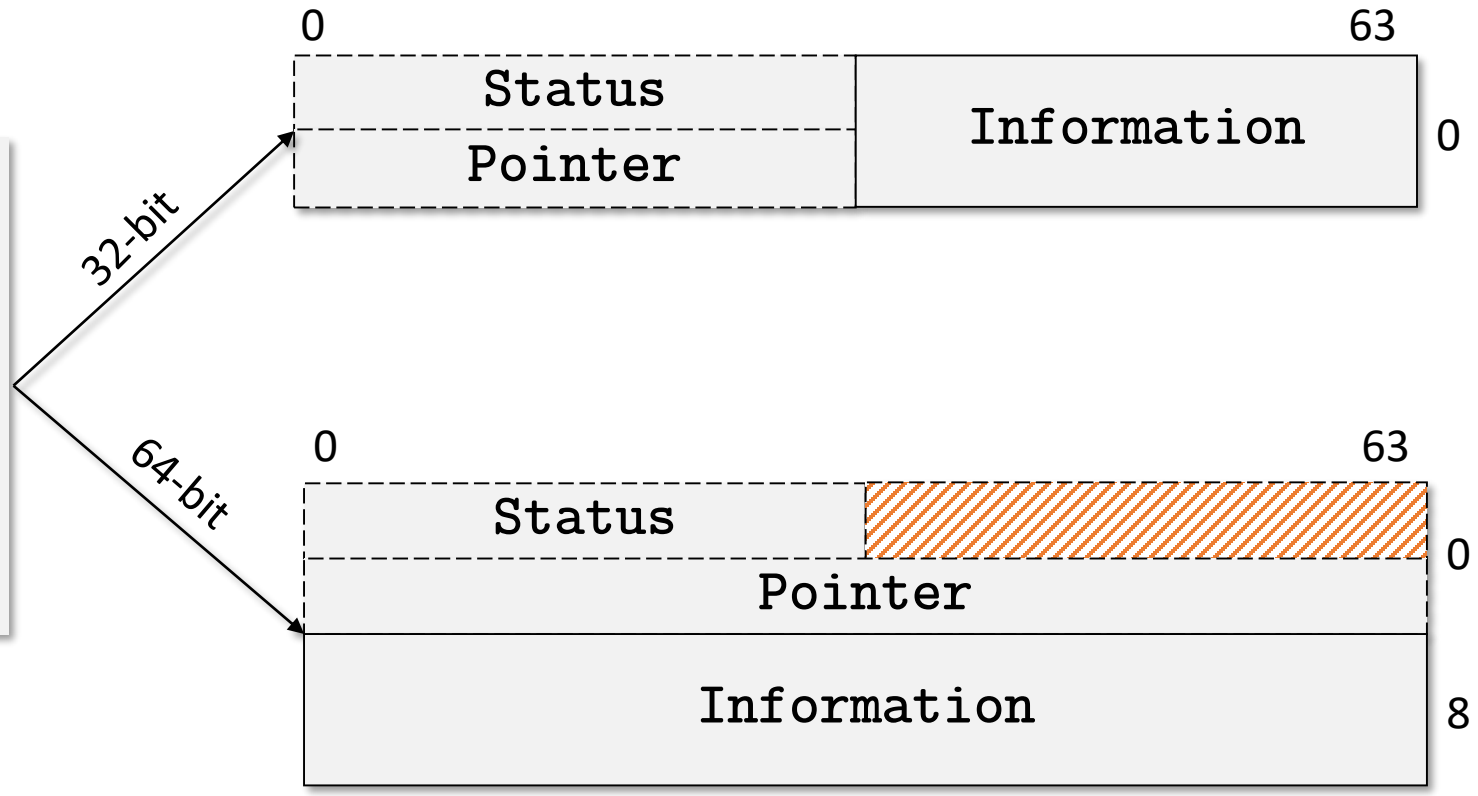
What's new – increased alignment

```
struct UNICODE_STRING {
  USHORT Length;
  USHORT MaximumLength;
  PWSTR  Buffer;
};
```



What's new – increased field sizes

```
struct IO_STATUS_BLOCK {
  union {
    NTSTATUS Status;
    PVOID Pointer;
  };
  ULONG_PTR Information;
};
```



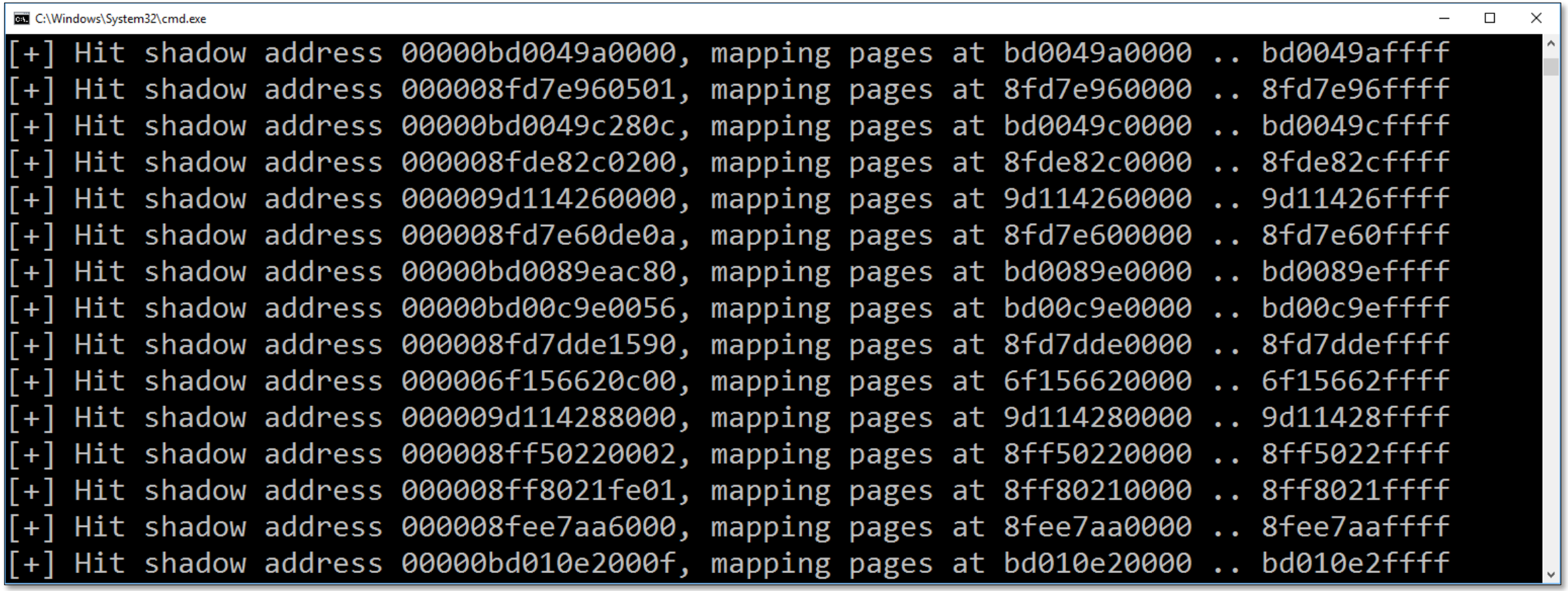
Problem #1 – shadow memory representation

- For x86 guest systems, shadow memory is allocated statically
  - In our case: **3X** memory overhead
  - Windows: 2 GB kernel space → 6 GB of metadata
  - Linux: 1 GB kernel space → 3 GB of metadata
- On x64, kernel address space ranges from `0xfff800000000000` to `0xffffffffffffffff`, a total of 128 terabytes
  - Impossible to allocate, equals the size of the user-mode address space

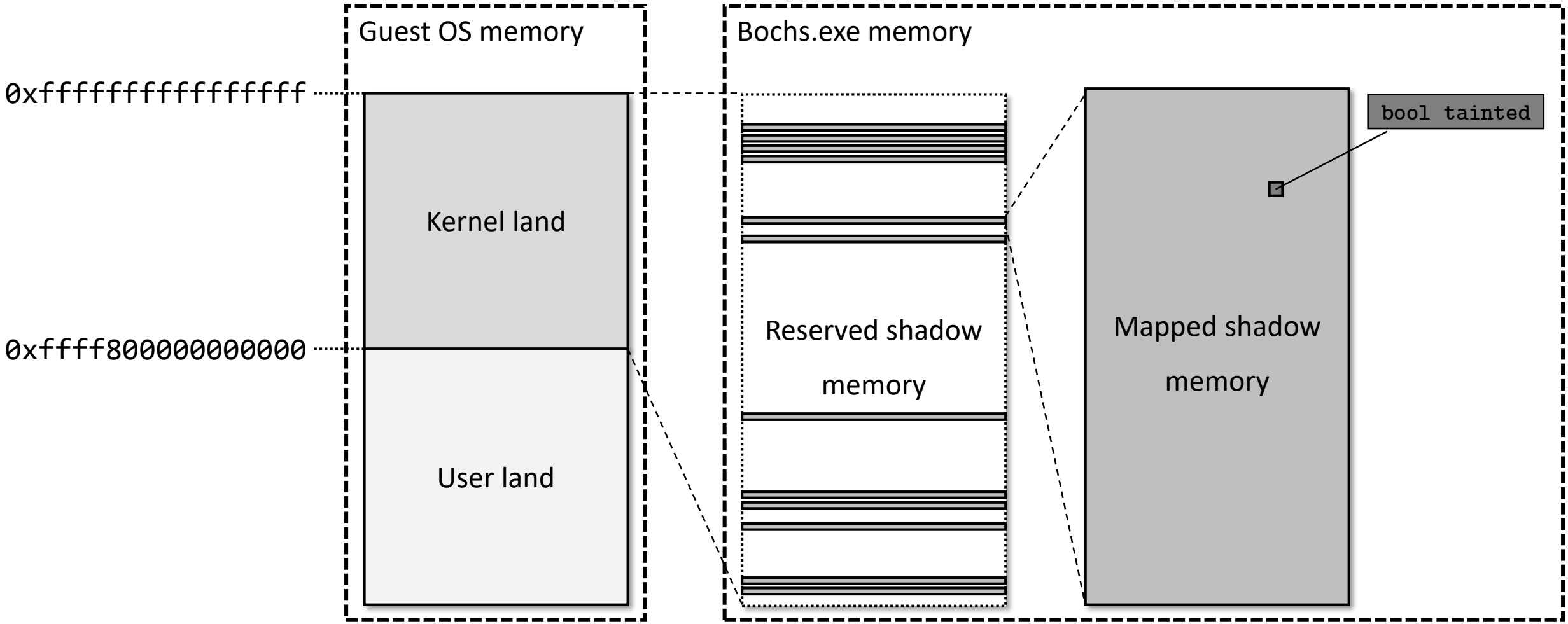
New strategy

- Eliminate some of the less critical metadata classes
- Convert the allocation origins container from static array to hash map
- Optimize taint representation by using bitmasks
  - Reserve a region of 16 terabytes – ⅛ of the user address space; back only the required pages with physical memory
  - Windows doesn’t support overcommit ☹
  - Implement custom overcommit using *vectored exception handling*

Custom overcommit



Shadow memory representation



Problem #2 – detecting data transfers

- x86 – standard library `memcpy()` is *mostly* implemented with `rep movs`
  - Easy to intercept in the Bochs instrumentation
  - Source, destination and copy size are all known at the same time
- x64 – more difficult, `memcpy()` is optimized to use `mov` and SSE instructions
  - Registers are not tainted – previous logic doesn't work
  - Patching not a feasible option
    - PatchGuard
    - Each driver has its own copy of the function

Implementation changes

Windows 7 x86

```
.text:00431231                  jb      short CopyUnwindUp_0
.text:00431233                  rep movsd                       ; jumptable 0043125C default case
.text:00431235                  jmp     ds:off_43134C[edx*4]
```

Windows 7 x64

```
.text:0000000140095720          mov     r9, [rdx+rcx]
.text:0000000140095724          mov     r10, [rdx+rcx+8]
.text:0000000140095729          movnti  qword ptr [rcx], r9
.text:000000014009572D          movnti  qword ptr [rcx+8], r10
.text:0000000140095732          mov     r9, [rdx+rcx+10h]
.text:0000000140095737          mov     r10, [rdx+rcx+18h]
.text:000000014009573C          movnti  qword ptr [rcx+10h], r9
.text:0000000140095741          movnti  qword ptr [rcx+18h], r10
.text:0000000140095746          mov     r9, [rdx+rcx+20h]
.text:000000014009574B          mov     r10, [rdx+rcx+28h]
.text:0000000140095750          add     rcx, 40h ; '@'
.text:0000000140095754          movnti  qword ptr [rcx-20h], r9
.text:0000000140095759          movnti  qword ptr [rcx-18h], r10
.text:000000014009575E          mov     r9, [rdx+rcx-10h]
.text:0000000140095763          mov     r10, [rdx+rcx-8]
.text:0000000140095768          dec     eax
.text:000000014009576A          movnti  qword ptr [rcx-10h], r9
.text:000000014009576F          movnti  qword ptr [rcx-8], r10
.text:0000000140095774          jnz     short mcpy90
.text:0000000140095776          sub     r8, 1000h
.text:000000014009577D          cmp     r8, 1000h
.text:0000000140095784          jnb     mcpy81
.text:000000014009578A          lock or byte ptr [rsp+0], 0
.text:000000014009578F          jmp     mcpy20
```

Windows 10 x64

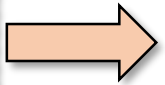
```
.text:0000000140189700          movdqu  xmm0, xmmword ptr [rdx+rcx]
.text:0000000140189705          movdqu  xmm1, xmmword ptr [rdx+rcx+10h]
.text:000000014018970B          movntdq xmmword ptr [rcx], xmm0
.text:000000014018970F          movntdq xmmword ptr [rcx+10h], xmm1
.text:0000000140189714          add     rcx, 40h ; '@'
.text:0000000140189718          movdqu  xmm0, xmmword ptr [rdx+rcx-20h]
.text:000000014018971E          movdqu  xmm1, xmmword ptr [rdx+rcx-10h]
.text:0000000140189724          movntdq xmmword ptr [rcx-20h], xmm0
.text:0000000140189729          movntdq xmmword ptr [rcx-10h], xmm1
.text:000000014018972E          dec     eax
.text:0000000140189730          jnz     short lcpy40
.text:0000000140189732          sub     r8, 200h
.text:0000000140189739          cmp     r8, 200h
.text:0000000140189740          jnb     short lcpy20
.text:0000000140189742          lock or byte ptr [rsp+0], 0
.text:0000000140189747          mov     r9, r8
.text:000000014018974A          shr     r9, 5
.text:000000014018974E          jnz     xcpy40
.text:0000000140189754          jmp     mcpy10
```

Solution

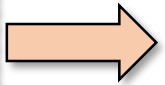
- Identify `memcpy()` function prologues with binary signatures

config.ini

```
[win7_64]

memcpy_signature = 4C8BD9482BD10F829E0100004983F808

…

[win10_64]

memcpy_signature = 4C8BD9482BD10F82A20100004983F84F
```



```
memmove         proc near

                mov     r11, rcx


                sub     rdx, rcx
                jb      mmov10
                cmp     r8, 8
```



```
memmove         proc near

                mov     r11, rcx


                sub     rdx, rcx
                jb      mmov10
                cmp     r8, 4Fh ; 'O'
```

Unsolved problem – aggressive inlining

- With each new version of Windows, more `memcpy()` calls with constant size are unrolled

	Windows 7	Windows 10
`ntoskrnl.exe`	1641	1626
`win32k.sys`	1133	696

Number of explicit `memcpy()` calls in the kernel

win32k!NtGdiCreateColorSpace

Windows 7 x64

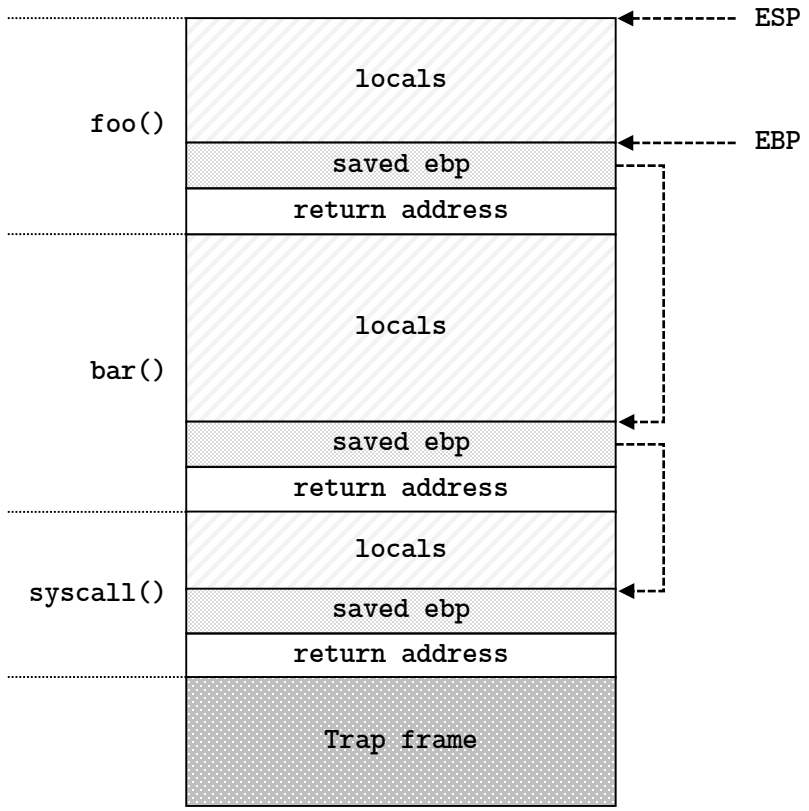
```
.text:FFFFF97FFF21C2C2                  lea     rcx, [rsp+298h+Dst] ; Dst
.text:FFFFF97FFF21C2C7                  mov     r8d, 250h       ; Size
.text:FFFFF97FFF21C2CD                  call    memmove
```

Windows 10 x64

```
.text:00000001C0102FA2                  movups  xmm0, xmmword ptr [rdx]
.text:00000001C0102FA5                  movups  xmmword ptr [rax], xmm0
.text:00000001C0102FA8                  movups  xmm1, xmmword ptr [rdx+10h]
.text:00000001C0102FAC                  movups  xmmword ptr [rax+10h], xmm1
.text:00000001C0102FB0                  movups  xmm0, xmmword ptr [rdx+20h]
.text:00000001C0102FB4                  movups  xmmword ptr [rax+20h], xmm0
.text:00000001C0102FB8                  movups  xmm1, xmmword ptr [rdx+30h]
.text:00000001C0102FBC                  movups  xmmword ptr [rax+30h], xmm1
.text:00000001C0102FC0                  movups  xmm0, xmmword ptr [rdx+40h]
.text:00000001C0102FC4                  movups  xmmword ptr [rax+40h], xmm0
.text:00000001C0102FC8                  movups  xmm1, xmmword ptr [rdx+50h]
.text:00000001C0102FCC                  movups  xmmword ptr [rax+50h], xmm1
.text:00000001C0102FD0                  movups  xmm0, xmmword ptr [rdx+60h]
.text:00000001C0102FD4                  movups  xmmword ptr [rax+60h], xmm0
.text:00000001C0102FD8                  add     rax, r10
.text:00000001C0102FDB                  movups  xmm1, xmmword ptr [rdx+70h]
.text:00000001C0102FDF                  movups  xmmword ptr [rax-10h], xmm1
.text:00000001C0102FE3                  add     rdx, r10
.text:00000001C0102FE6                  sub     rcx, 1
.text:00000001C0102FEA                  jnz     short loc_1C0102FA2
.text:00000001C0102FEC                  movups  xmm0, xmmword ptr [rdx]
.text:00000001C0102FEF                  movups  xmmword ptr [rax], xmm0
.text:00000001C0102FF2                  movups  xmm1, xmmword ptr [rdx+10h]
.text:00000001C0102FF6                  movups  xmmword ptr [rax+10h], xmm1
.text:00000001C0102FFA                  movups  xmm0, xmmword ptr [rdx+20h]
.text:00000001C0102FFE                  movups  xmmword ptr [rax+20h], xmm0
.text:00000001C0103002                  movups  xmm1, xmmword ptr [rdx+30h]
.text:00000001C0103006                  movups  xmmword ptr [rax+30h], xmm1
.text:00000001C010300A                  movups  xmm0, xmmword ptr [rdx+40h]
.text:00000001C010300E                  movups  xmmword ptr [rax+40h], xmm0
.text:00000001C0103012                  jmp     short loc_1C010301A
```

Problem #3 – unwinding callstacks

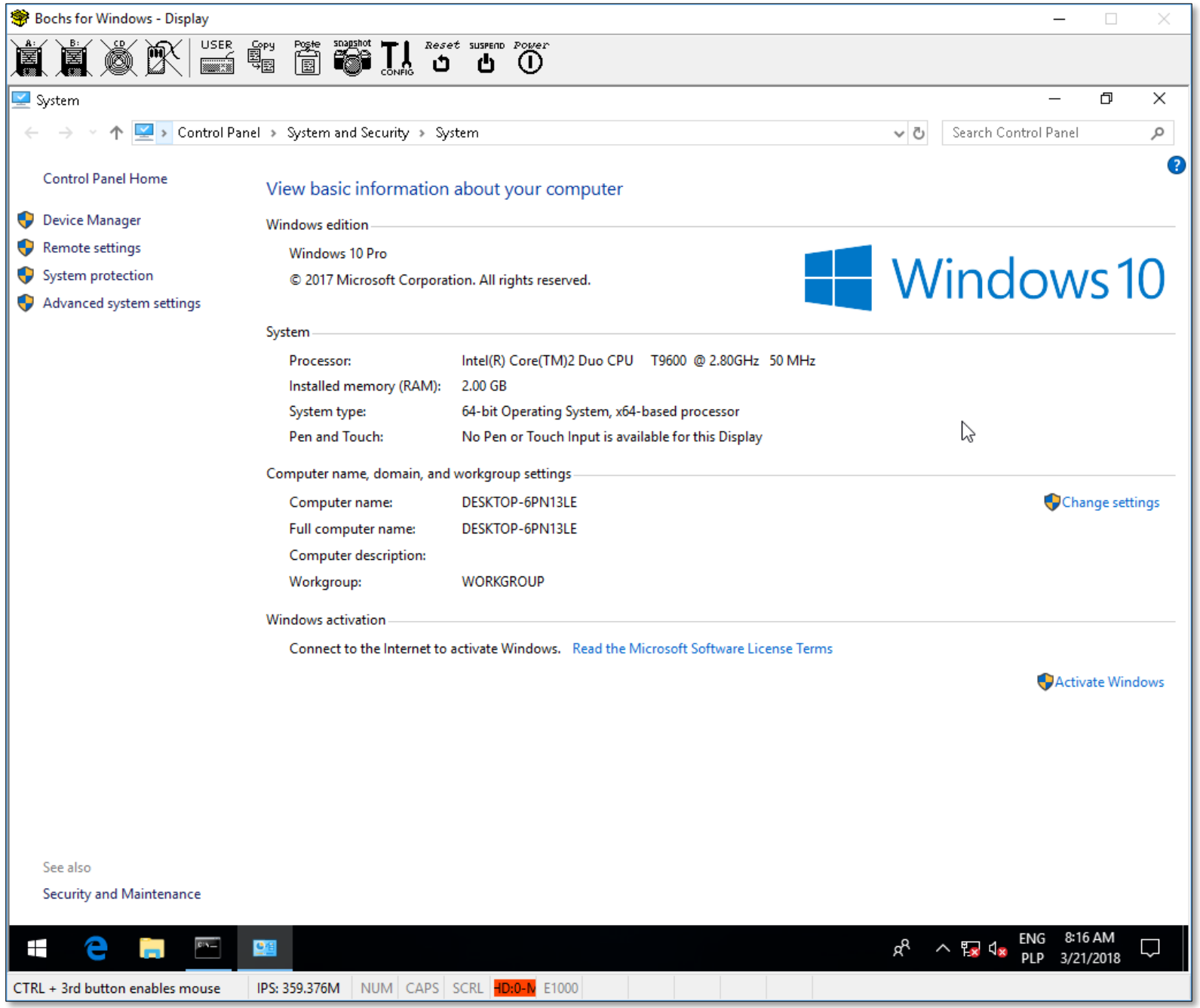
- x86: stack frames are chained together by the values of saved **EBP**
  - Call stacks can be traversed by the instrumentation without any further requirements



Unwinding callstacks (x64)

- On 64-bit builds, `RBP` no longer serves as the stack frame pointer
- Windows symbol files (`.pdb`) for each module contain the information necessary to walk the stack trace for that image
  - We load symbols in Bochspwn for symbolizing virtual addresses, anyway
- `StackWalk64` from the Debug Help Library implements the functionality
  - Requires several primitives and information regarding the target's CPU context, all easily available in Bochs

With all the problems resolved…

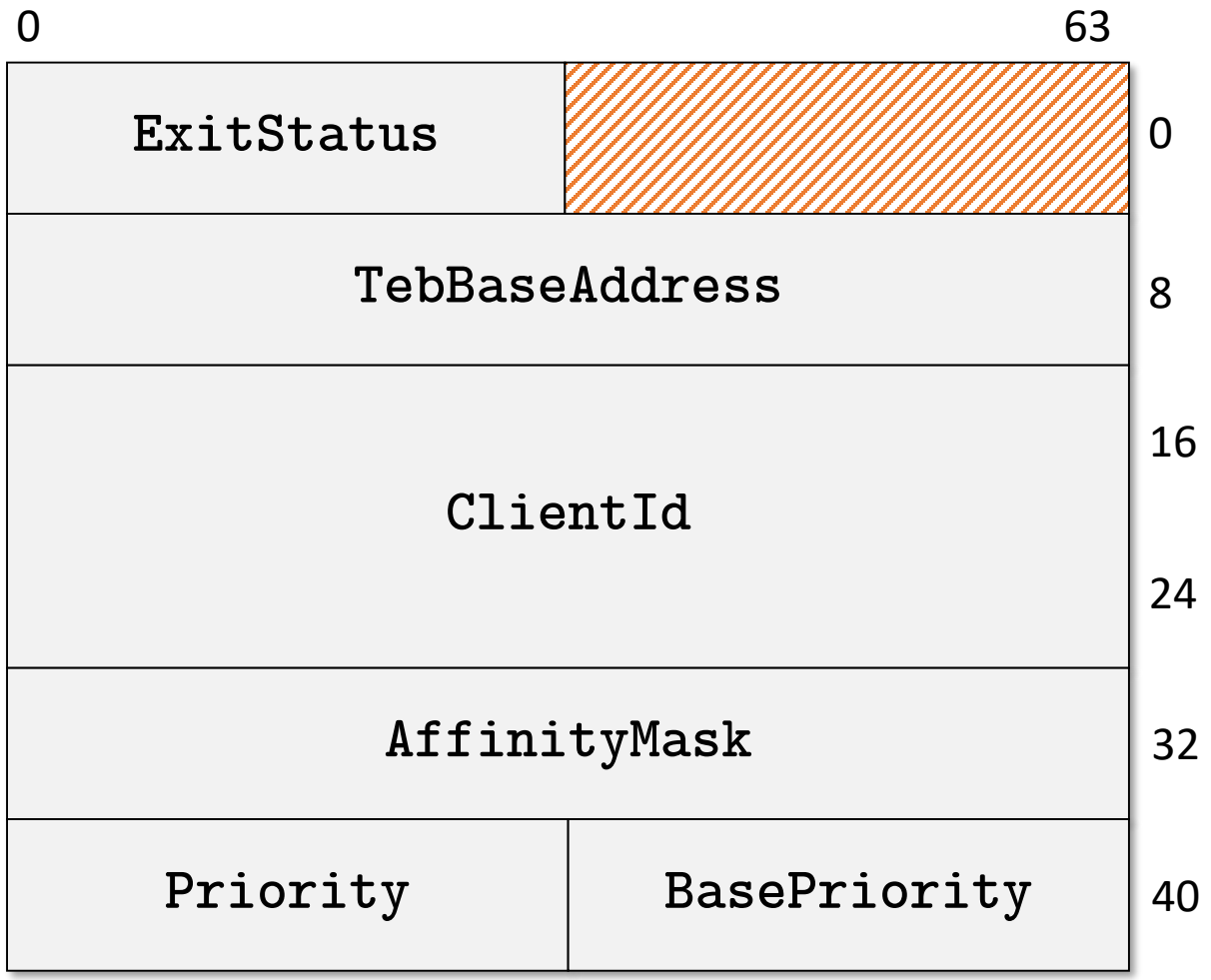

Bochs for Windows - Display
USER
Copy
Paste
snapshot
CONFIG
Reset
suspend
Power
System
Control Panel › System and Security › System
Search Control Panel
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
View basic information about your computer
Windows edition
Windows 10 Pro
© 2017 Microsoft Corporation. All rights reserved.
Windows 10
System
Processor: Intel(R) Core(TM)2 Duo CPU T9600 @ 2.80GHz 50 MHz
Installed memory (RAM): 2.00 GB
System type: 64-bit Operating System, x64-based processor
Pen and Touch: No Pen or Touch Input is available for this Display
Computer name, domain, and workgroup settings
Computer name: DESKTOP-6PN13LE
Change settings
Full computer name: DESKTOP-6PN13LE
Computer description:
Workgroup: WORKGROUP
Windows activation
Connect to the Internet to activate Windows. Read the Microsoft Software License Terms
Activate Windows
See also
Security and Maintenance
ENG
PLP
8:16 AM
3/21/2018
CTRL + 3rd button enables mouse
IPS: 359.376M
NUM
CAPS
SCRL
HD:0-M
E1000

Windows x64 bug report

```
------------------------------ found uninit-copy of address fffff8a000a63010

[pid/tid: 000001a0/000001a4] {     wininit.exe}
       COPY of fffff8a000a63010 ---> 1afab8 (64 bytes), pc = fffff80002698600
       [                             mov r11, rcx ]
Allocation origin: 0xfffff80002a11101 (ntoskrnl.exe!IopQueryNameInternal+00000071)
--- Shadow memory:
00000000: 00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00 ................
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
--- Actual memory:
00000000: 2e 00 30 00 aa aa aa aa 20 30 a6 00 a0 f8 ff ff ..0..... 0......
00000010: 5c 00 44 00 65 00 76 00 69 00 63 00 65 00 5c 00 \.D.e.v.i.c.e.\.
00000020: 48 00 61 00 72 00 64 00 64 00 69 00 73 00 6b 00 H.a.r.d.d.i.s.k.
00000030: 56 00 6f 00 6c 00 75 00 6d 00 65 00 32 00 00 00 V.o.l.u.m.e.2...
--- Stack trace:
 #0  0xfffff80002698600 ((00095600) ntoskrnl.exe!memmove+00000000)
 #1  0xfffff80002a11319 ((0040e319) ntoskrnl.exe!IopQueryNameInternal+00000289)
 #2  0xfffff800028d4426 ((002d1426) ntoskrnl.exe!IopQueryName+00000026)
 #3  0xfffff800028e8fa8 ((002e5fa8) ntoskrnl.exe!ObpQueryNameString+000000b0)
 #4  0xfffff8000291313b ((0031013b) ntoskrnl.exe!NtQueryVirtualMemory+000005fb)
 #5  0xfffff800026b9283 ((000b6283) ntoskrnl.exe!KiSystemServiceCopyEnd+00000013)
```

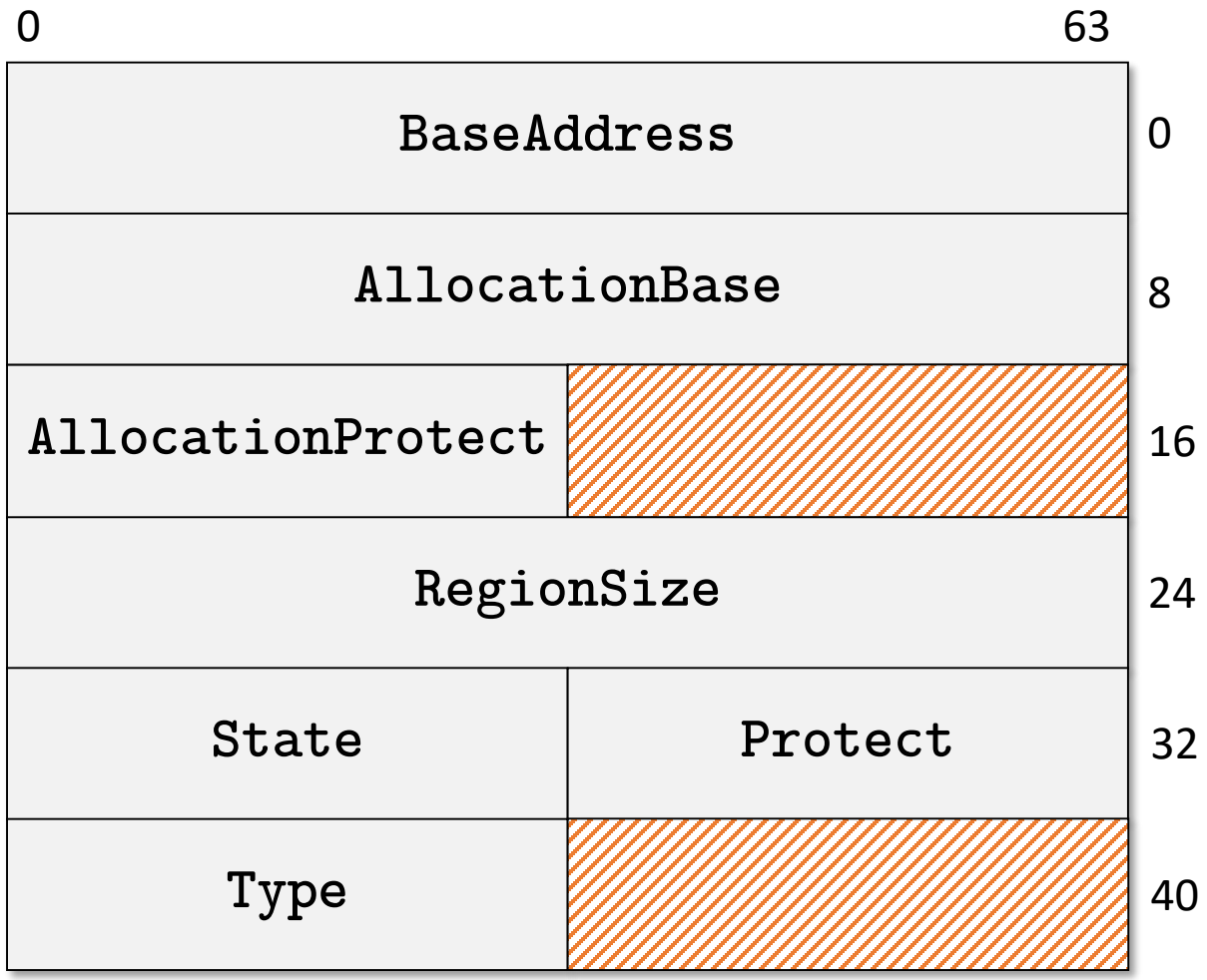
Example: THREAD_BASIC_INFORMATION

```
struct THREAD_BASIC_INFORMATION {
  NTSTATUS  ExitStatus;
  PNT_TIB   TebBaseAddress;
  CLIENT_ID ClientId;
  KAFFINITY AffinityMask;
  KPRIORITY Priority;
  KPRIORITY BasePriority;
};
```



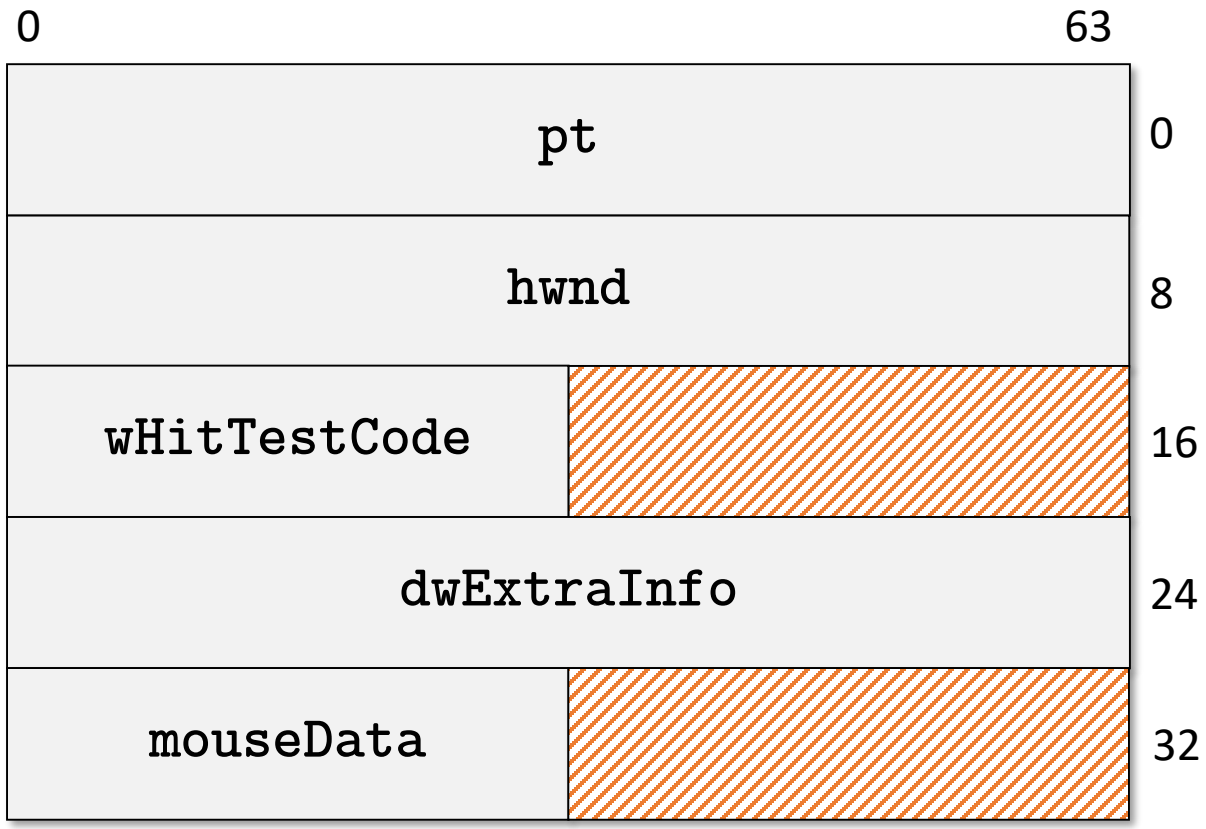
Example: MEMORY_BASIC_INFORMATION

```
struct MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;
  PVOID  AllocationBase;
  DWORD  AllocationProtect;
  SIZE_T RegionSize;
  DWORD  State;
  DWORD  Protect;
  DWORD  Type;
};
```



Example: **MOUSEHOOKSTRUCTEX**

```
struct MOUSEHOOKSTRUCT {
  POINT      pt;
  HWND       hwnd;
  UINT       wHitTestCode;
  ULONG_PTR dwExtraInfo;
};

struct MOUSEHOOKSTRUCTEX {
  MOUSEHOOKSTRUCT MOUSEHOOKSTRUCT;
  DWORD           mouseData;
};
```

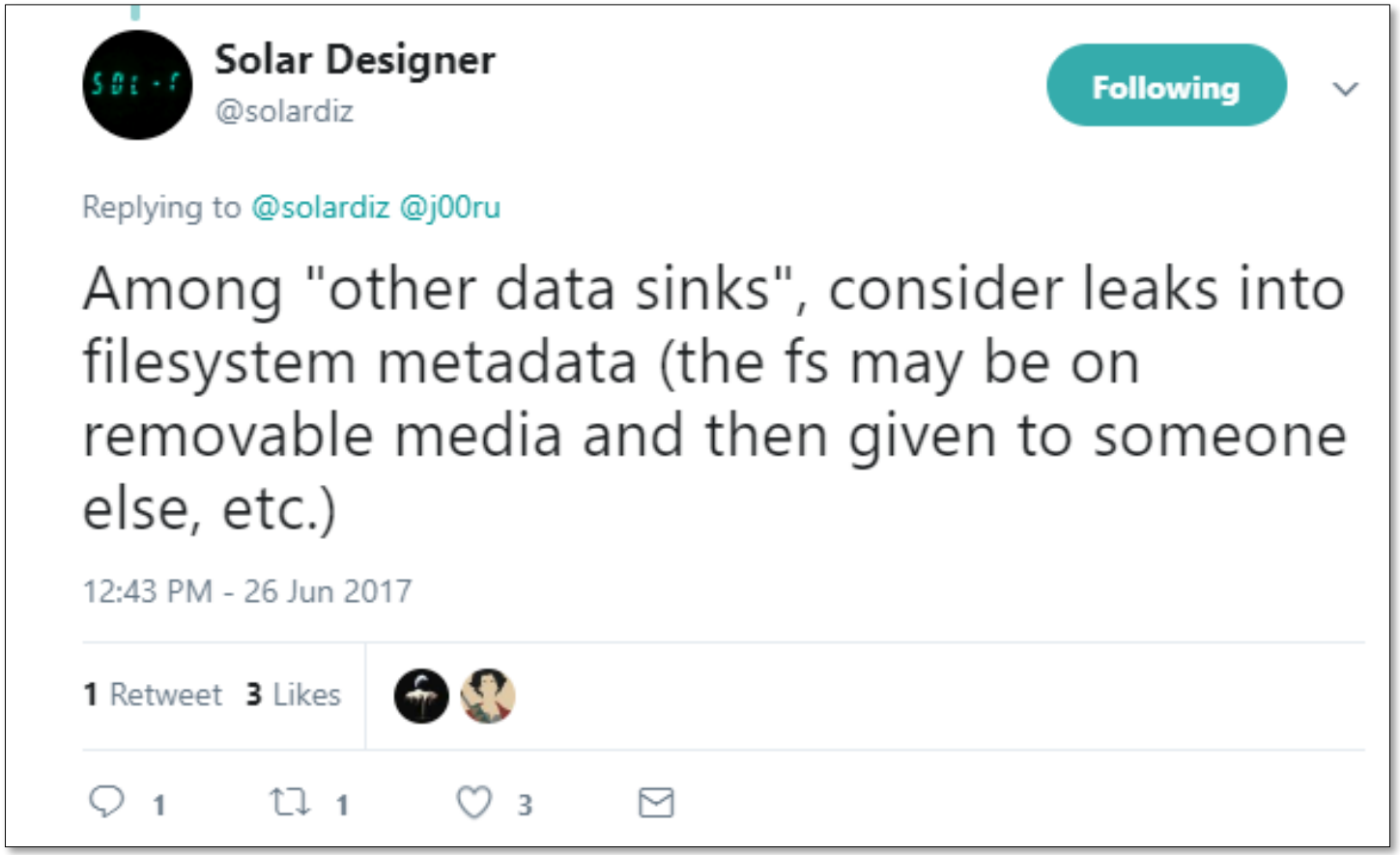


17 new x64-specific infoleaks

CVE	Component	Fix date	Memory Type	Number of leaked bytes
CVE-2018-0810	win32k!SfnINOUTLPWINDOWPOS, win32k!fnHkINLPMOUSEHOOKSTRUCTEX, win32k!fnHkINLPMSLLHOOKSTRUCT, win32k!SfnINLPHELPINFOSTRUCT	February 2018	Stack, Pool	4/8
CVE-2018-0811	win32k!XDCOBJ::RestoreAttributes	March 2018	Stack	4
CVE-2018-0813	win32k!UMPDOBJ::LockSurface	March 2018	Pool	4
CVE-2018-0814	win32k!PROXYPORT::SendRequest	March 2018	Stack	8
CVE-2018-0894	nt!NtQueryVirtualMemory (MemoryMappedFilenameInformation) nt!NtQueryObject (ObjectNameInformation)	March 2018	Pool	4
CVE-2018-0895	nt!NtQueryInformationThread (ThreadBasicInformation)	March 2018	Stack	4
CVE-2018-0896	msrpc!LRPC_CASSOCIATION::AlpcSendCancelMessage	March 2018	Stack	8
CVE-2018-0897	nt!KiDispatchException	March 2018	Stack	120
CVE-2018-0898	nt!PnpBuildCmResourceList	March 2018	Pool	8
CVE-2018-0899	videoprt!pVideoPortReportResourceList	March 2018	Pool	20
CVE-2018-0900	nt!PnpFilterResourceRequirementsList	March 2018	Pool	40
CVE-2018-0901	nt!NtWaitForDebugEvent	March 2018	Stack	4
CVE-2018-0926	win32k CoreMessagingK interface	March 2018	Pool	4
CVE-2018-0968	nt!NtQueryVirtualMemory (MemoryImageInformation)	April 2018	Stack	4
CVE-2018-0971	nt!NtQuerySystemInformation (SystemPageFileInformation[Ex])	April 2018	Stack	4
CVE-2018-0973	nt!NtQueryInformationProcess (ProcessImageFileName)	April 2018	Stack, Pool	4
CVE-2018-0974	nt!NtQueryVirtualMemory (Memory[Privileged]BasicInformation)	April 2018	Stack	8

Leaks to file systems

Other data sinks



Leaks to file system metadata

- File systems are represented with relatively complex data structures
- Is it possible that some drivers save them to storage directly from the stack/heap?
- Physical attack scenario

Detection

- In the Bochs instrumentation, fill all stack/pool allocations with a recognizable marker byte
- Change the Bochs disk operation mode in `bochsrc` from „flat” to „volatile”
- Start the OS and perform a variety of operations on the file system
- Periodically scan the disk changelog in search of marker bytes, and analyze which parts of the file system they are found in

Results (Windows)

FAT: ∅

FAT32: ∅

exFAT: ∅

Modest leaks in NTFS

Stack leaks	Pools leaks
NtfsDeleteAttributeAllocation	NtfsAddAttributeAllocation+0xb16
NtfsWriteLog	NtfsCheckpointVolume+0xdcd
NtfsAddAttributeAllocation	NtfsDeleteAttributeAllocation+0x12d
NtfsCreateAttributeWithAllocation	CreateAttributeList+0x1c
	NtfsCreateMdlAndBuffer+0x95
	NtfsInitializeReservedBuffer+0x20

Fixed collectively as CVE-2017-11880

Getting worse

Windows Kernel pool memory disclosure into NTFS metadata ($LogFile) in Ntfs!LfsRestartLogFile

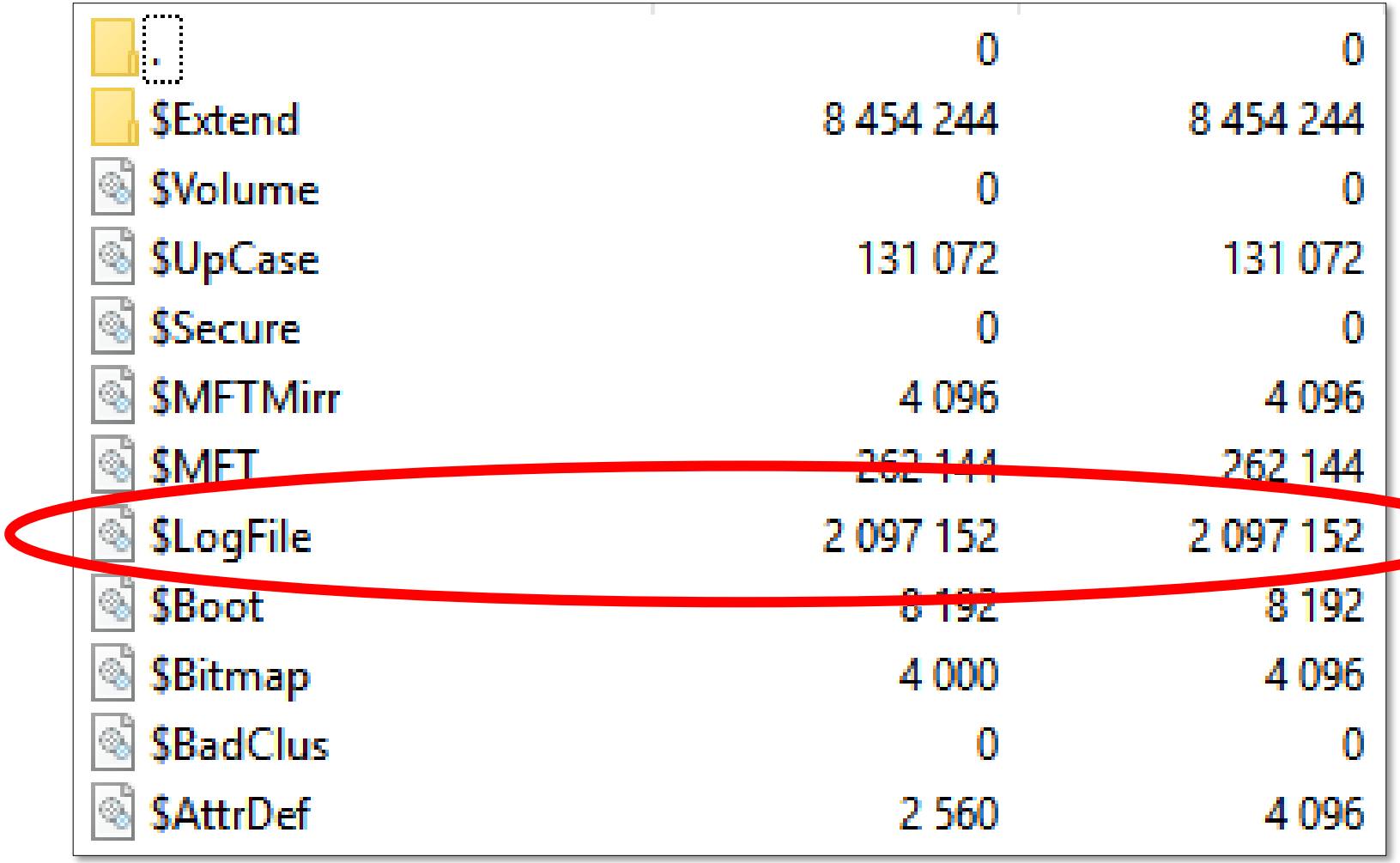
Project Member Reported by mjurczyk@google.com, Sep 4

This tracker entry is a fork of issue #1325, which this bug was reported as a part of. However, as some essential information and context was provided in issue #1325, the "Reported" date was adjusted there to account for it. The new information did not concern the vulnerability discussed here, so we are sticking to the original deadline in this case. Hence the need to create a separate entry in the tracker.

We have discovered that the NTFS.sys driver writes uninitialized kernel pool memory into the internal structures of the file system, while mounting and operating on it. This may be of a security concern in cases where, for example, users share some files with each other via USB sticks or other storage media with NTFS-formatted volumes on them. While the victim would assume that they're only sharing intended data explicitly copied to the disk, they could also unknowingly share bits and pieces of sensitive/confidential information stored in the kernel, that just happened to reside in the memory area used by NTFS.sys while constructing internal file system structures.

Even more scary are leaks which don't require any human interaction and take place immediately when the volume is mounted. In this scenario, it could be possible to disclose kernel memory of a locked machine with physical access to a USB port, by repeatedly plugging in a flash drive (or a device which emulates one), waiting for the uninitialized memory to be written by the system, reading it back and re-mounting the disk.

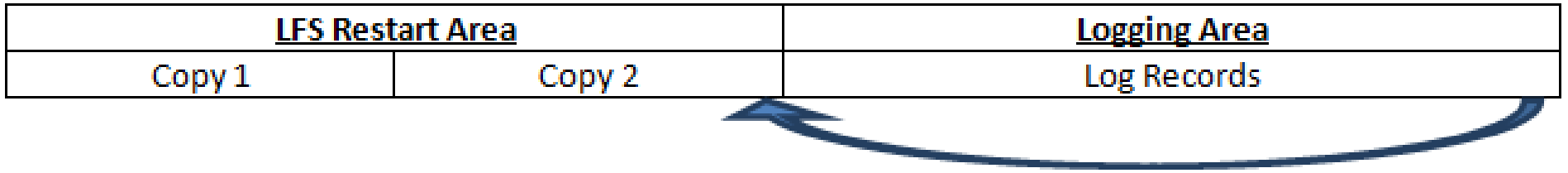
We have implemented some dedicated logic in our Bochspwn system instrumentation to detect instances of such info leaks. As a result, we have found that pool memory allocated in the Ntfs!LfsRestartLogFile function is leaked to NTFS volumes (both system and external ones). From a security perspective, the leak is quite severe, as it is triggered with no user interaction (just by mounting an NTFS volume), and it discloses ~7.5kB of kernel pool memory in 2 chunks of ~3800 consecutive bytes. This data is saved in the first 8192 bytes of the special $LogFile file, as part of the two "RSTR" restart area records. As we understand it, this means that every NTFS-formatted USB stick last accessed by Windows 7 now includes more than 7kB of junk kernel memory from that computer. However, since $LogFile is an internal file, reading from it requires raw disk access which significantly reduces the impact of the bug.



.	0	0
$Extend	8 454 244	8 454 244
$Volume	0	0
$UpCase	131 072	131 072
$Secure	0	0
$MFTMirr	4 096	4 096
$MFT	262 144	262 144
$LogFile	2 097 152	2 097 152
$Boot	8 192	8 192
$Bitmap	4 000	4 096
$BadClus	0	0
$AttrDef	2 560	4 096

CVE-2017-11817: leak in **$LogFile**

- Every NTFS volume contains an internal `$LogFile` file
  - Not accessible to user-mode programs, used by file system drivers
  - Readable through raw access to the storage device
- Initialized every time the file system is mounted



Source: `http://www.ntfs.com/transaction.htm`

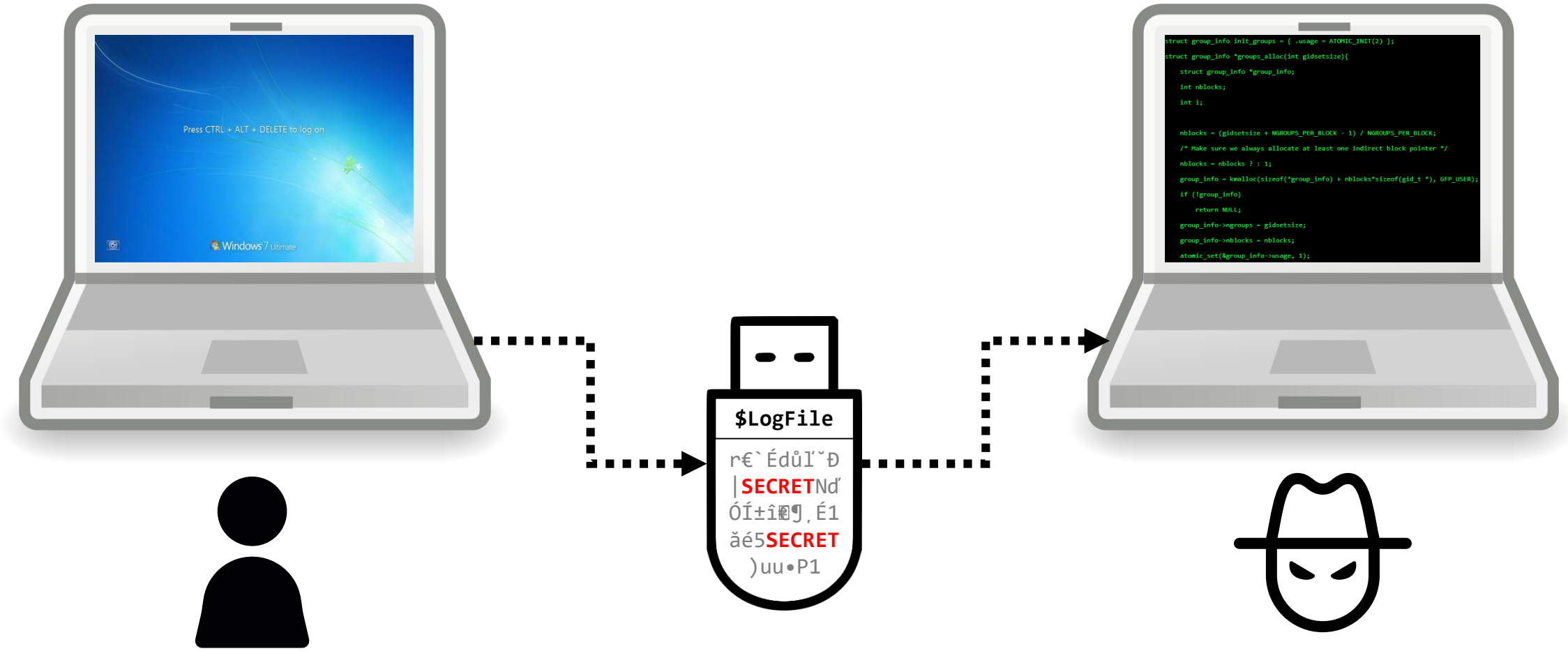
Restart areas

- Both *restart areas* are 4096 byte regions in the header of `$LogFile`
- Allocated from the pool in `Ntfs!LfsRestartLogFile`
  - The memory was not zeroed out on Windows 7 and older systems
  - Typically initialized with very little data before being written to disk
- Over 7 kB of leftover kernel memory was automatically stored on disk every time it was plugged into a machine

Attack without user interaction

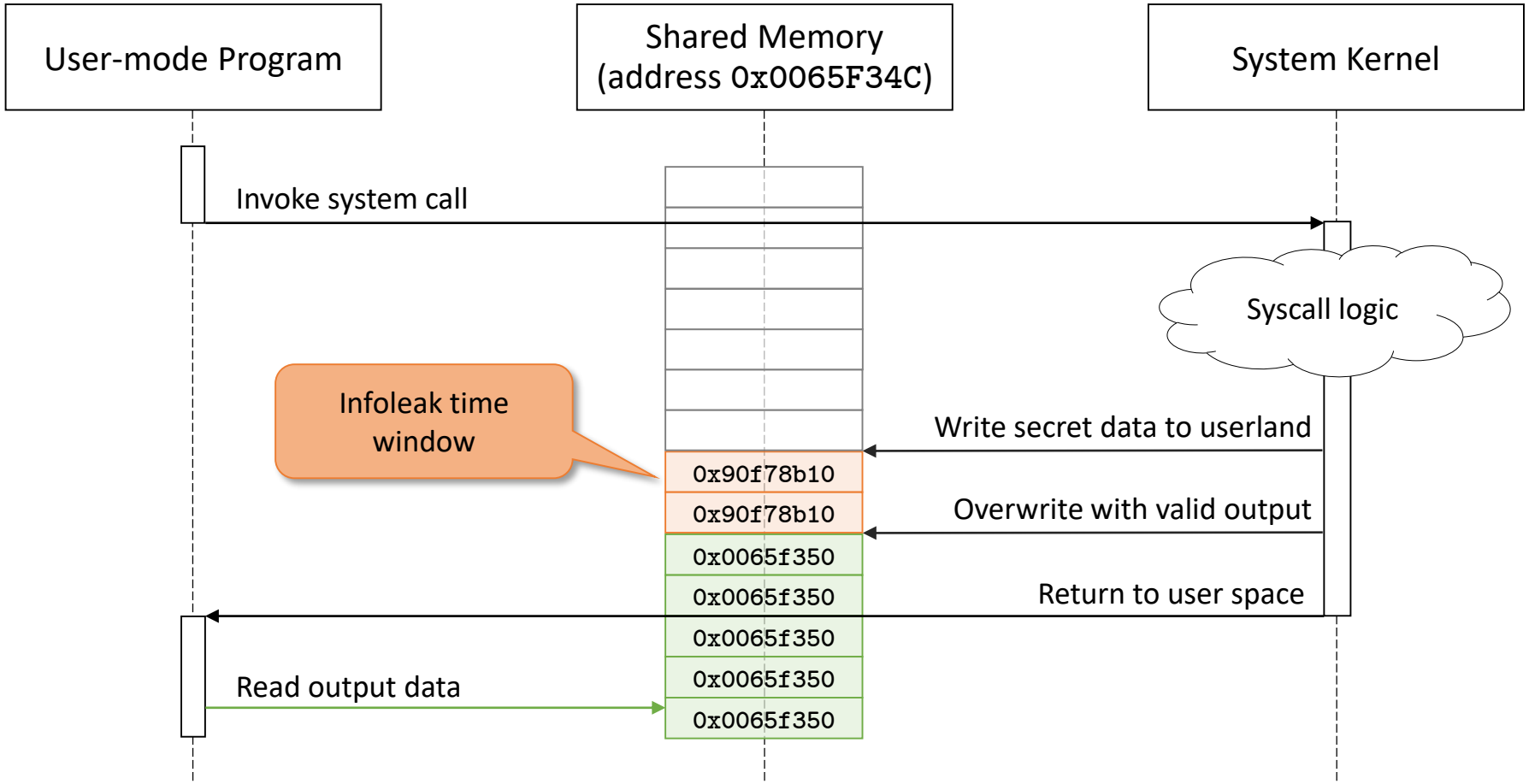
- Windows automatically mounts file systems of physically connected devices, even when the system is locked
- The vulnerability enabled extracting uninitialized kernel memory through the physical USB port

Demo

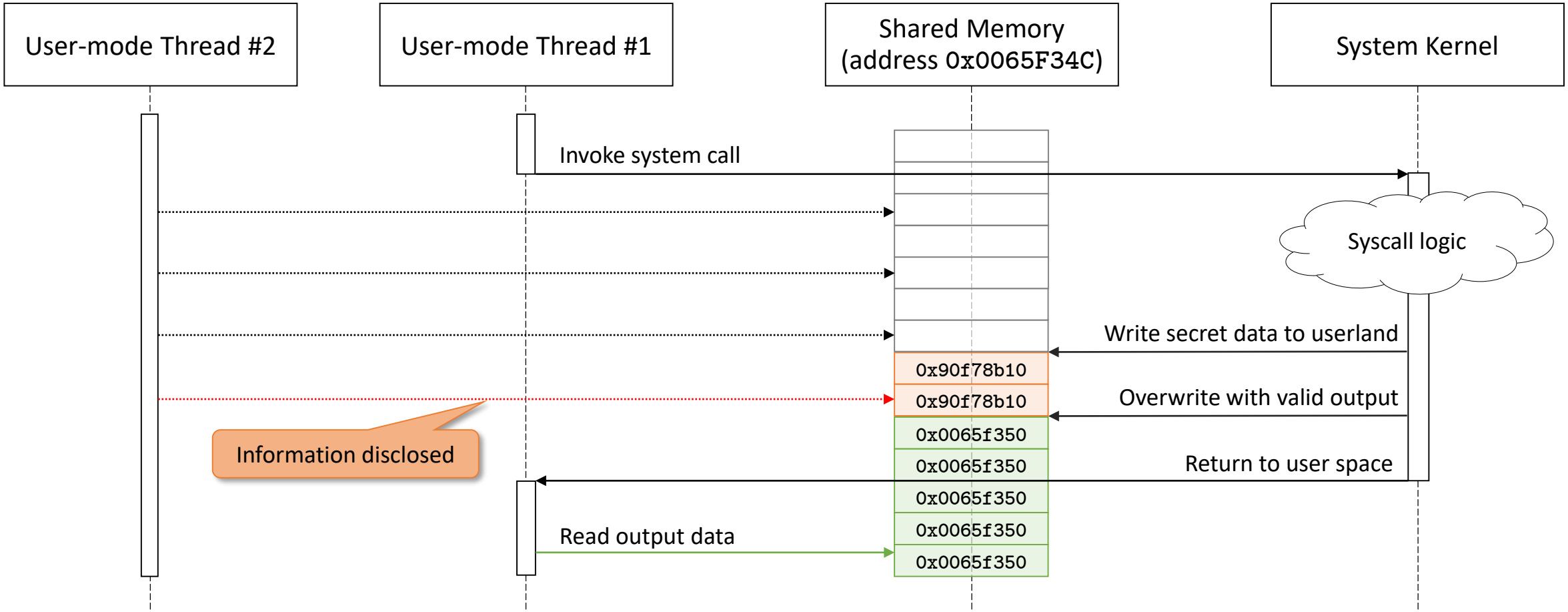

Press CTRL + ALT + DELETE to log on
Windows 7 Ultimate
struct group_info init_groups = { .usage = ATOMIC_INIT(2) };
struct group_info *groups_alloc(int gidsetsize){
struct group_info *group_info;
int nblocks;
int i;
nblocks = (gidsetsize + NGROUPS_PER_BLOCK - 1) / NGROUPS_PER_BLOCK;
/* Make sure we always allocate at least one indirect block pointer */
nblocks = nblocks ? : 1;
group_info = kmalloc(sizeof(*group_info) + nblocks*sizeof(gid_t *), GFP_USER);
if (!group_info)
return NULL;
group_info->ngroups = gidsetsize;
group_info->nblocks = nblocks;
atomic_set(&group_info->usage, 1);
$LogFile
r€`ÉdůĬ˘Ð
|SECRETNď
ÓÍ±îę¶¸É1
ăé5SECRET
)uu•P1

Leaks through *double writes*

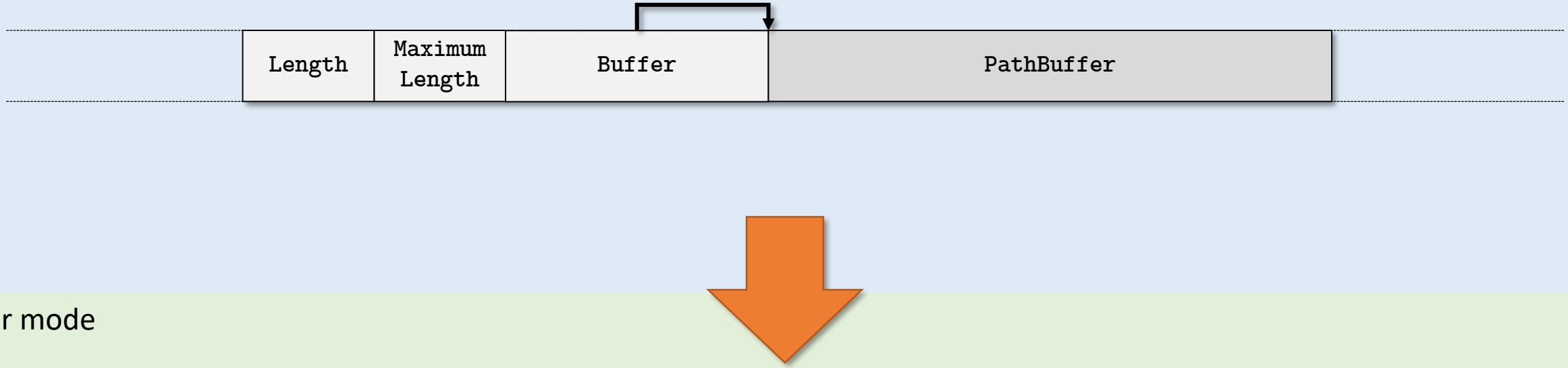
Double write race condition



Double write exploitation

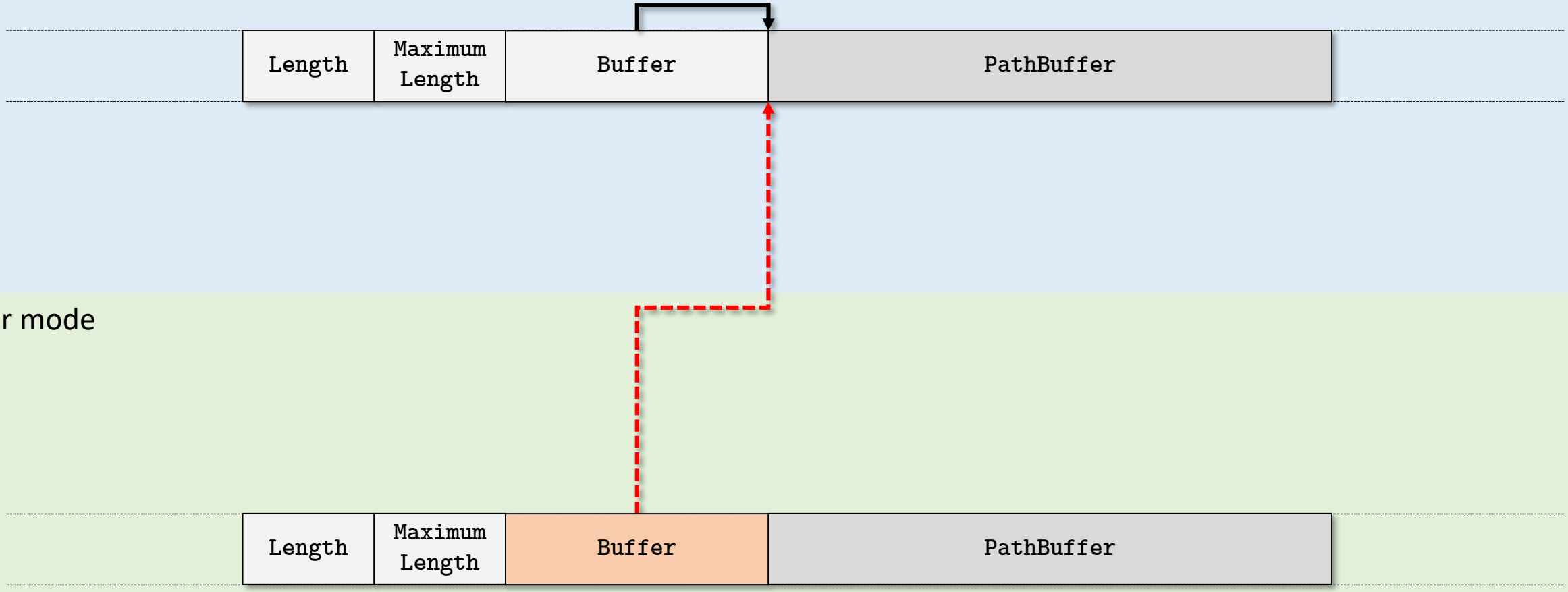


Kernel mode

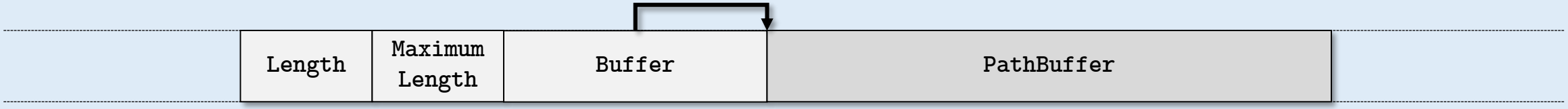


User mode

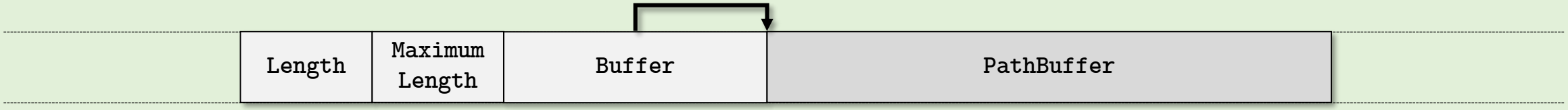
Kernel mode


Length
Maximum Length
Buffer
PathBuffer
User mode
Length
Maximum Length
Buffer
PathBuffer

Kernel mode



User mode



Circumstances

- Typical for structures with pointers passed between user/kernel mode
  - Similar to uninitialized memory disclosure – a matter of code brevity and cleanliness
  - It's easier to copy an entire structure and adjust particular fields, then copy each field separately or construct an extra local object
- `copy_to_user` creates an incentive to write more secure code
  - A local copy of the object is more likely to be created if direct pointer manipulation is not allowed

Detection with system instrumentation

- Relatively simple, similar to original Bochspwn for double fetches
- Log all kernel→user memory writes via `bx_instr_lin_access`
- If the operation overwrites non-zero data written in the scope of the same thread/system call, report a bug
  - Signal when a kernel-mode address is overwritten with a user-mode one

Double write bug report

```
------------------------------ found double-write of address 0x1cfca8 (base: 0x1cfca8, offset: 0x0)
[pid/tid: 000001c0/000001c4] {     wininit.exe}:
       WRITE of 4 bytes, pc = 81b9fb37 [            mov dword ptr ds:[ecx+4], edi ]

Old memory contents: |[48] 38 e0 8c|
New memory contents: |[ac] fc 1c 00|

Write no. 1 (byte 0x48):
#0  0x81957143 ((0014a143) ntoskrnl.exe!memcpy+00000033)
#1  0x81b9fb13 ((00392b13) ntoskrnl.exe!IopQueryNameInternal+000001c3)
#2  0x81b9f949 ((00392949) ntoskrnl.exe!IopQueryName+0000001b)
#3  0x81b9f869 ((00392869) ntoskrnl.exe!ObQueryNameStringMode+00000509)
#4  0x81aeb904 ((002de904) ntoskrnl.exe!MmQueryVirtualMemory+00000994)
#5  0x81aeaf5e ((002ddf5e) ntoskrnl.exe!NtQueryVirtualMemory+0000001e)
#6  0x81965d50 ((00158d50) ntoskrnl.exe!KiSystemServicePostCall+00000000)

Write no. 2 (byte 0xac):
#0  0x81b9fb37 ((00392b37) ntoskrnl.exe!IopQueryNameInternal+000001e7)
#1  0x81b9f949 ((00392949) ntoskrnl.exe!IopQueryName+0000001b)
#2  0x81b9f869 ((00392869) ntoskrnl.exe!ObQueryNameStringMode+00000509)
#3  0x81aeb904 ((002de904) ntoskrnl.exe!MmQueryVirtualMemory+00000994)
#4  0x81aeaf5e ((002ddf5e) ntoskrnl.exe!NtQueryVirtualMemory+0000001e)
#5  0x81965d50 ((00158d50) ntoskrnl.exe!KiSystemServicePostCall+00000000)
```

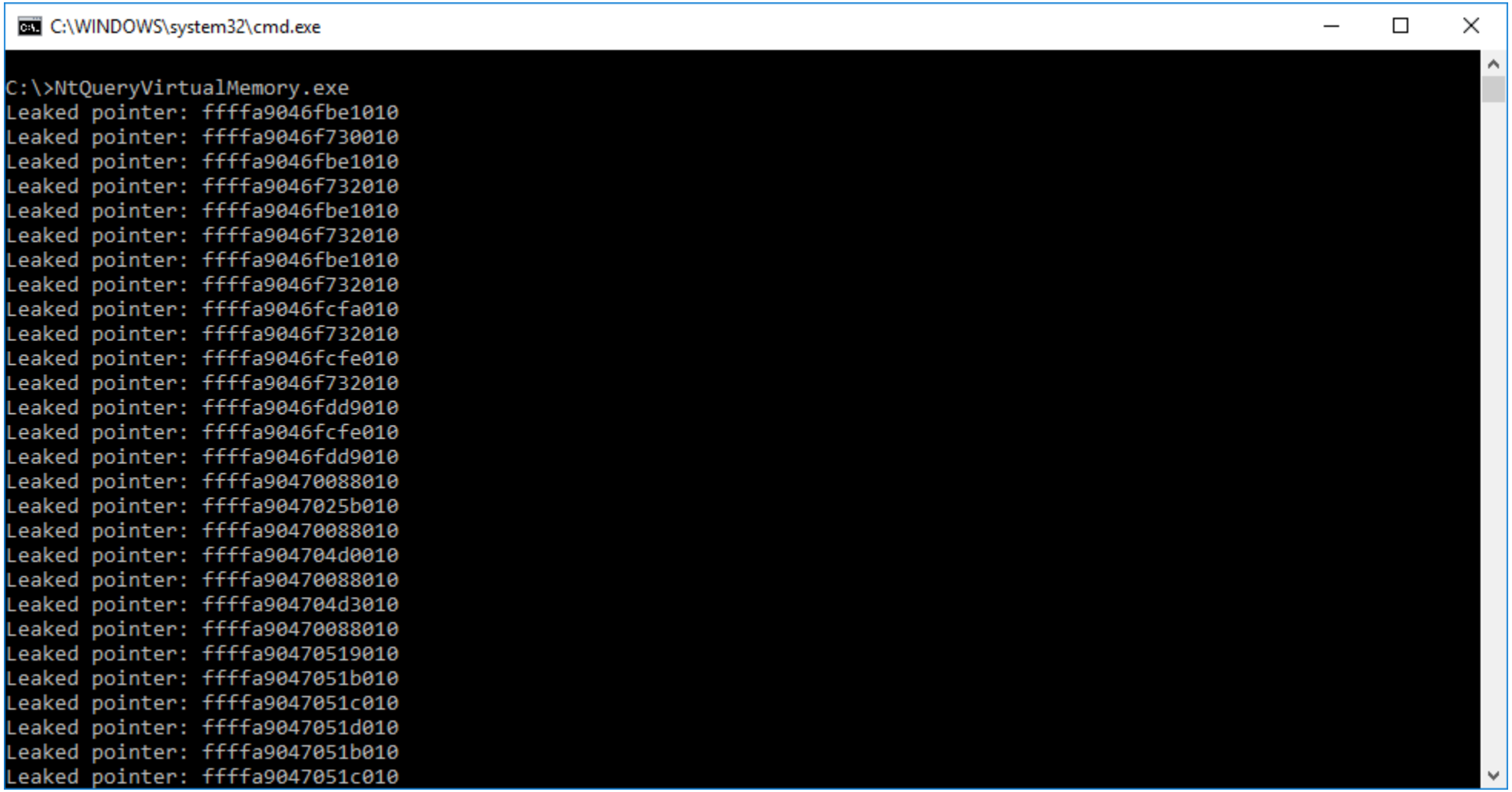
Results

1. `nt!IopQueryNameInternal`
   - Structure: `UNICODE_STRING`
   - Accessible via several entrypoints, e.g. `nt!NtQueryObject`, `nt!NtQueryVirtualMemory`
2. `nt!PspCopyAndFixupParameters`
   - Structures: `UNICODE_STRING` inside `RTL_USER_PROCESS_PARAMETERS`
3. `win32k!NtUserfnINOUTNCCALCSIZE`
   - Structure: `NCCALCSIZE_PARAMS`

MSRC response

Please note that due to some By-Design kernel pointer leaks already present in our platforms, **Information Disclosures which only disclose kernel pool pointers will only be serviced in v.Next until all by design disclosures can be resolved**. Information Disclosures of uninitialized kernel memory will continue to be serviced via Security Updates. Any leaks within privileged processes will also be considered v.Next; unless you can supply PoC which proves that you can perform the same leak - but not kernel pool pointer leaks - as an unprivileged user.

Currently 0-day

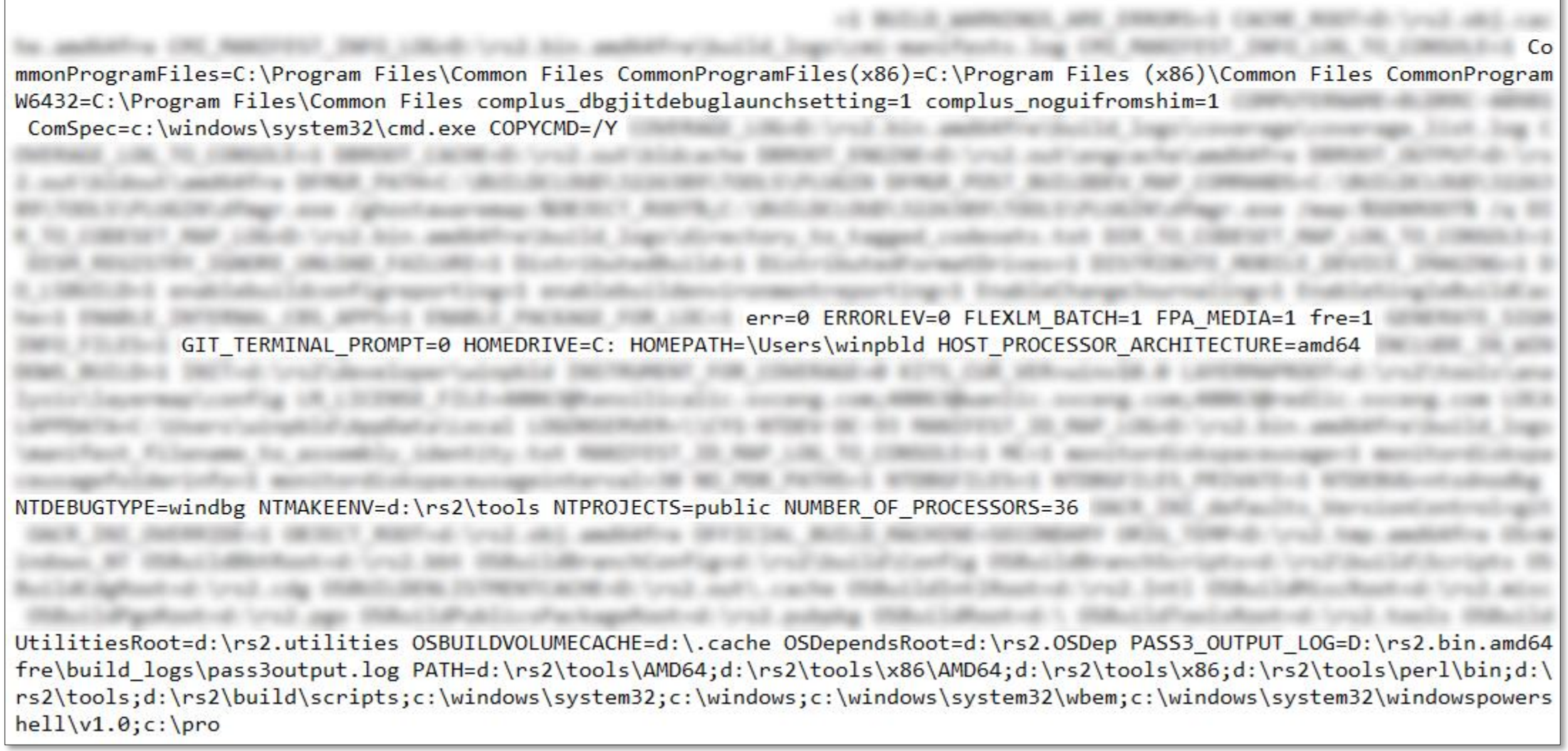


Bonus: memory disclosure in PDB (CVE-2018-1037)

Preparing Windows symbols

- Before each new Bochspwn session, I downloaded a number of `.pdb` files corresponding to the target's system files
- While randomly inspecting the contents of `combase.pdb` from Windows 10, I noticed 3 kB of strange data close to the file header

```
Microsoft C/C++ MSF 7.00...DS    .  .   {A  ä..     yA
```



```
Co
mmonProgramFiles=C:\Program Files\Common Files CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files CommonProgram
W6432=C:\Program Files\Common Files complus_dbgjitdebuglaunchsetting=1 complus_noguifromshim=1
 ComSpec=c:\windows\system32\cmd.exe COPYCMD=/Y
                                                                                                  err=0 ERRORLEV=0 FLEXLM_BATCH=1 FPA_MEDIA=1 fre=1
 GIT_TERMINAL_PROMPT=0 HOMEDRIVE=C: HOMEPATH=\Users\winpbld HOST_PROCESSOR_ARCHITECTURE=amd64
NTDEBUGTYPE=windbg NTMAKEENV=d:\rs2\tools NTPROJECTS=public NUMBER_OF_PROCESSORS=36
UtilitiesRoot=d:\rs2.utilities OSBUILDVOLUMECACHE=d:\.cache OSDependsRoot=d:\rs2.OSDep PASS3_OUTPUT_LOG=D:\rs2.bin.amd64
fre\build_logs\pass3output.log PATH=d:\rs2\tools\AMD64;d:\rs2\tools\x86\AMD64;d:\rs2\tools\x86;d:\rs2\tools\perl\bin;d:\
rs2\tools;d:\rs2\build\scripts;c:\windows\system32;c:\windows;c:\windows\system32\wbem;c:\windows\system32\windowspowers
hell\v1.0;c:\pro
```

PDB generation in Visual Studio

- In Visual Studio, `mspdbcore.dll` is used to generate symbol files
  - Hosted by an external, long-lived `mspdbsrv.exe` process
- The source code of the library was published on GitHub by Microsoft in the microsoft-pdb repository
  - Can be freely audited for bugs

Generate Debug Info	**Optimize for debugging (/DEBUG)**
Generate Program Database File	$(OutDir)$(TargetName).pdb
Generate Full Program Database File	
Strip Private Symbols	
Generate Map File	No
Map File Name	
Map Exports	No
Debuggable Assembly	

PDB structure

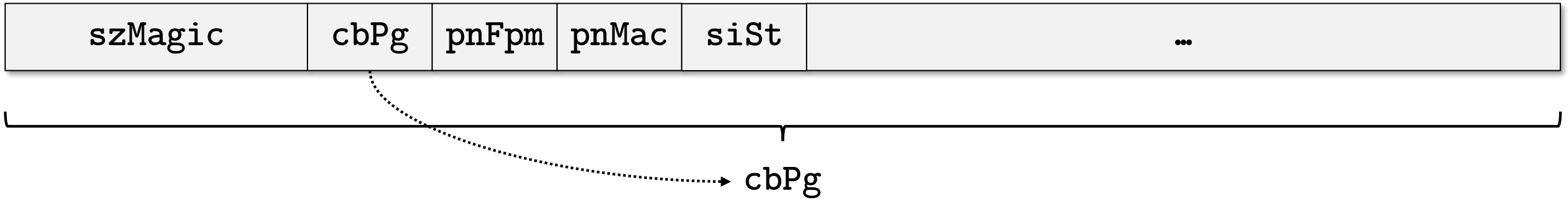
- Essentially MSF (Multi-Stream Format) files
  - Split into blocks (*pages*) of fixed size ∈ {512, 1024, 2048, 4096}
  - Typical sizes in Visual Studio are 1024 and 4096 bytes
- First block at offset 0 is the “super block” (or MSF header)

Block index	**0**	**1**	**2**	**3 – 4096**	**4096**	**4097**	**4098**	**4099 – 8191**	**…**
Meaning	The Superblock	Free Block Map 1	Free Block Map 2	Data	Data	FPM1	FPM2	Data	…

Source: `https://llvm.org/docs/PDB/MsfFile.html`

Header structure

```
union BIGMSF_HDR {  // page 0 (and more if necessary)
    struct {
        char    szMagic[0x1e];  // version string
        CB  cbPg;               // page size
        UPN pnFpm;              // page no. of valid FPM
        UPN pnMac;              // current no. of pages
        [...]
    };
    PG  pg;
};
```



Creating a PDB

Actual page size

```
1662  BOOL MSF_HB::afterCreate(MSF_EC* pec, CB cbPage) {
1663      // init hdr; when creating a new MSF, always create the BigMsf variant.
1664      memset(&bighdr, 0, sizeof bighdr);
1665      memcpy(&bighdr.szMagic, szBigHdrMagic, sizeof szBigHdrMagic);
1666      bighdr.cbPg = cbPage;
[...]
```

4096 bytes

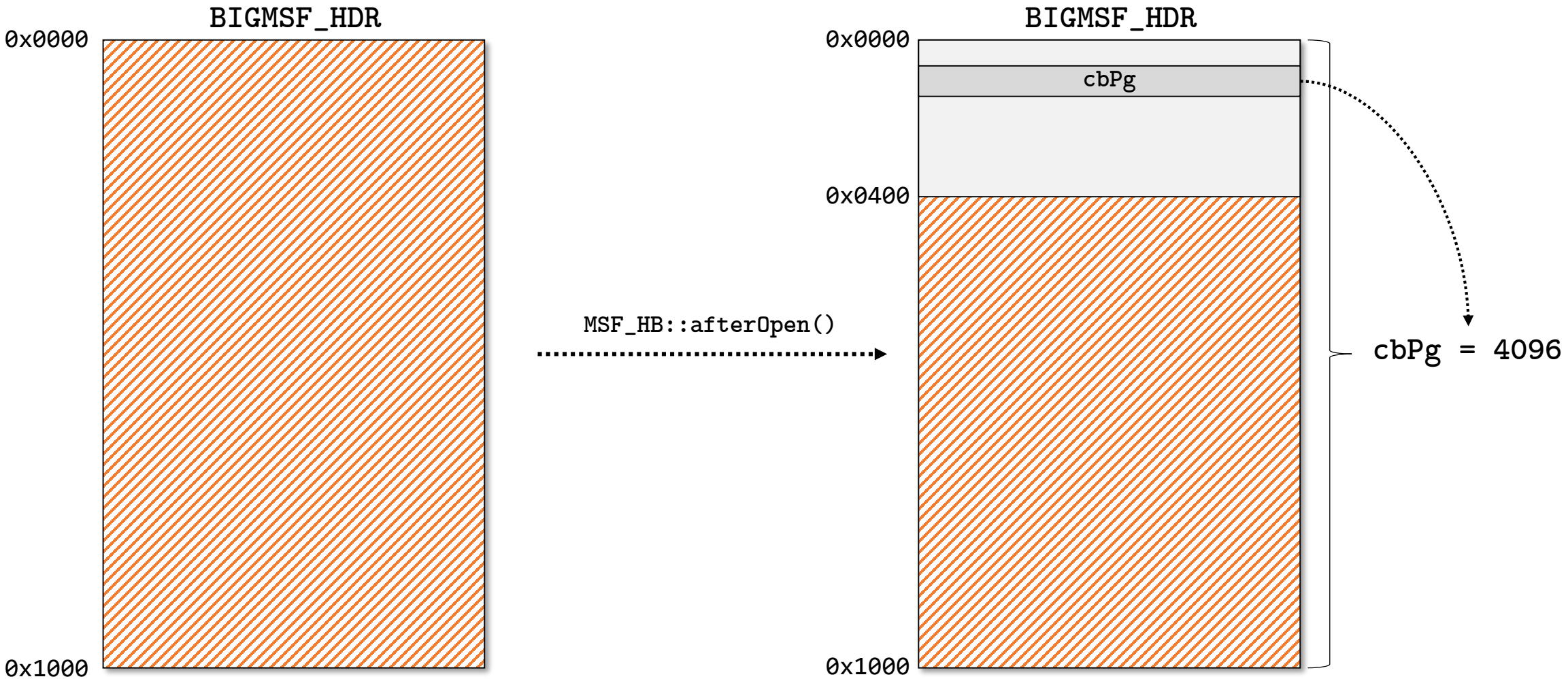
Updating an existing PDB

```
1519    BOOL MSF_HB::afterOpen( MSF_EC* pec ) {
1520                                    // VSWhidbey:600553
1521        fBigMsf = true;             // This is arbitrary, and will be overwritten in fValidHdr().
1522                                    // We do this to avoid uninitialized reads of this variable in pnMac().
1523        pnMac(1);                   // extantPn(pnHdr) must be TRUE for first readPn()!
1524        msfparms = rgmsfparms[0];   // need min page size set here for initial read.
1525
1526        if (!readPn(pnHdr, &hdr)) {
1527            if (pec) {
1528                *pec = MSF_EC_FILE_SYSTEM;
1529            }
1530            pIStream = NULL;
1531            return FALSE;
1532        }
```

```
 154  const CB    cbPgMin     = 0x400;
      [...]
 196  const MSFParms  rgmsfparms[] = {
      [...]
 200      MSF_PARMS(cbPgMin, 10, pnMaxMax, 8, 8), // gives 64meg (??)
```

1024 bytes

The bug


BIGMSF_HDR
0x0000
0x1000
MSF_HB::afterOpen()
BIGMSF_HDR
0x0000
cbPg
0x0400
0x1000
cbPg = 4096

Scope of the leak

- PDB files are not frequently exchanged over the Internet, except for the Microsoft Symbol Server
- Let's analyze the extent of the problem?
  - Only Windows 10 symbols affected
  - Only a small subset of `.pdb` files contained the leak
  - Easy to detect by checking for `cbPg = 4096`

Affected files

1. appxdeploymentclient.pdb
2. authbroker.pdb
3. biwinrt.pdb
4. combase.pdb
5. cryptowinrt.pdb
6. dllhst3g.pdb
7. mbaeapipublic.pdb
8. mbsmsapi.pdb
9. mbussdapi.pdb
10. msvideodsp.pdb
11. msxml6.pdb
12. nfccx.pdb
13. ole32.pdb
14. playtomanager.pdb
15. provcore.pdb
16. rtmediaframe.pdb
17. urlmon.pdb
18. uxtheme.pdb
19. vaultcli.pdb
20. webcamui.pdb
21. windows.applicationmodel.background.systemeventsbroker.pdb
22. windows.applicationmodel.background.timebroker.pdb
23. windows.applicationmodel.pdb
24. windows.devices.enumeration.pdb
25. windows.devices.portable.pdb
26. windows.devices.sensors.pdb
27. windows.globalization.fontgroups.pdb
28. windows.graphics.pdb
29. windows.media.streaming.pdb
30. windows.networking.backgroundtransfer.pdb
31. windows.networking.pdb
32. windows.storage.applicationdata.pdb
33. windows.storage.compression.pdb
34. windows.ui.input.inking.pdb
35. windows.ui.pdb
36. windows.ui.xaml.pdb
37. windows.web.pdb
38. wintypes.pdb
39. wpnapps.pdb
40. wwaapi.pdb

Scope of the leak

Symbol Package Set	Files total	Files with leak	Percentage	Amount of disclosed memory
Windows 10 – July 2015	30807	152	0.49%	456 kB
Windows 10 – November 2015	31712	152	0.48%	456 kB
Windows 10 – March 2016	16138	78	0.48%	234 kB
Windows 10 and Windows Server 2016 – August 2016	16238	76	0.47%	228 kB
Windows 10 – September 2016	16174	76	0.47%	228 kB
Windows 10 and Windows Server 2016 – April 2017	16755	76	0.45%	228 kB
Windows 10 and Windows Server, version 1709 – October 2017	17062	78	0.46%	234 kB
Total	**144886**	**688**	**0.47%**	**2064 kB (2.02 MB)**

Impact

- Some leaked regions contained textual strings such as environment variables on the build servers, which revealed paths, domains, command line flags etc.

Fixed just 2 weeks ago

CVE-2018-1037 | Microsoft Visual Studio Information Disclosure Vulnerability

Security Vulnerability

Published: 04/10/2018
MITRE CVE-2018-1037

An information disclosure vulnerability exists when Visual Studio improperly discloses limited contents of uninitialized memory while compiling program database (PDB) files. An attacker who took advantage of this information disclosure could view uninitialized memory from the Visual Studio instance used to compile the PDB file.

To take advantage of the vulnerability, an attacker would require access to an affected PDB file created using a vulnerable version of Visual Studio. An attacker would have no way to force a developer to produce this information disclosure.

The security update addresses the vulnerability by correcting how PDB files are generated when a project is compiled.

On this page

PDBCopy tool update

Usage: PDBCOPY.exe <*target.pdb*> <*backup.pdb*> -CVE-2018-1037 {[verbose|autofix]}

Arguments

- target.pdb: The file name of the PDB file to update.
- backup.pdb: The name to use for a backup copy of the PDB file.
- *-CVE-2018-1037*: This switch causes the tool to report on whether the PDB file is affected by the issue, and optional arguments can report the affected memory block and update the existing PDB file in place to remove the disclosed memory. This switch is exclusive from other PDBCopy switches and takes two optional arguments:
  - verbose: Dumps the memory block from the original PDB file that is removed by the switch.
  - autofix: Updates and zero-fills the affected memory block in the PDB file.

Closing words

Bochspwn Reloaded limitations

- Typical shortcomings of dynamic binary instrumentation
  - Performance
  - Dependency on kernel code coverage
  - Inability to test most device drivers
  - Accuracy of taint tracking

Future work

- For binary instrumentation:
  - Other operating systems
  - Other data sinks: file systems, network
  - Other security domains: inter-process communication (sandboxing), virtualization
- Hope to see in Windows:
  - Implementation, testing, evaluation and deployment of various detection and mitigation techniques

Thanks!

[@j00ru](https://twitter.com/j00ru)

http://j00ru.vexillium.org/

j00ru.vx@gmail.com

See upcoming whitepaper:

Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking

Mateusz Jurczyk
mjurczyk@google.com

Google, Inc.

April 2018

Abstract

One of the responsibilities of modern operating systems is to enforce privilege separation between user-mode applications and the kernel. This includes ensuring that the influence of each program on the execution environment is limited by the defined security policy, but also that programs may only access information they are authorized to read. The latter goal is especially difficult to achieve considering that the properties of C – the main programming language used in kernel development – make it highly challenging to securely pass data between different security domains. There is a significant risk of disclosing sensitive leftover kernel data hidden amidst the output of otherwise harmless system calls, unless special care is taken the prevent the problem. Issues of this kind can help bypass security mitigations such as KASLR and StackGuard, or retrieve information processed by the kernel on behalf of the system or other users, e.g. file contents, network traffic, cryptographic keys and so on.

In this paper, we introduce the concept of employing full system emulation and taint tracking to detect the disclosure of uninitialized kernel stack and heap/pool memory to user-space, and how it was successfully implemented in the *Bochspwn Reloaded* project based on the open-source Bochs IA-32 emulator. To date, the tool has been used to identify over 70 memory disclosure vulnerabilities in the Windows kernel, and more than 10 lesser bugs in Linux. Further in the document, we evaluate alternative ways of detecting such information leaks, and outline data sinks other than user-space where uninitialized memory may also leak from the kernel. Finally, we enumerate potential means of preventing and mitigating this kind of flaws, and provide suggestions on related research areas that haven't been fully explored yet. Appendix A details several further ideas for system-wide instrumentation (implemented using Bochs or otherwise), which can be used to discover other programming errors in OS kernels.