Bochspwn: Identifying 0-days via system-wide memory access pattern analysis

Mateusz “j00ru” Jurczyk
Gynvael Coldwind
Google Inc.
Agenda

• Software instrumentation in security
  – what? why? how?
  – what about kernels?
• Bochspwn and double fetches
  – Microsoft Windows
  – Linux
  – BSD
• Other projects
  – Bochspwn:msan sneak peek
• Future work
DEMO

WINDOWS 7 SP1 32-BIT
ELEVATION OF PRIVILEGES
SOFTWARE INSTRUMENTATION TRIVIA
The basics of instrumentation

Program execution flow

Program state
- CPU context
- Memory address space
- Resources

Program state
- CPU context
- Memory address space
- Resources

monitor()

Program state
- CPU context
- Memory address space
- Resources

examine()

Program state
- CPU context
- Memory address space
- Resources

change()
Important points

• Instrumentation only operates on existing program states.
  – doesn’t generate new ones, we can only „feed” it.
  – **the pros:** can reason about real software behavior and identify real bugs.
  – **the cons:** limited to actual code coverage.

• We end up with two separate problems
  – the more different states the more knowledge about program logic
  – the better instrumentation
Existing technology

- User-mode instrumentation widely spread nowadays
  - DBI
    - extensible frameworks: Intel Pin, DynamoRIO
    - run-time program verification projects: the valgrind suite, Microsoft Application Verifier
  - compile-time solutions
    - \{Address, Memory, Thread\} Sanitizer
    - IOC
    - gprof
    - stack / heap protectors are kind of, too.
Known applications in security

• Code coverage analysis
  – corpus distillation, exploring program state tree, various “smart fuzzing” techniques

• Detection of security-relevant conditions
  – memory corruption, out-of-bounds access (ASan, valgrind)
  – dynamic allocator issues, e.g. double free (ASan, valgrind)
  – use of uninitialized memory (MSan, valgrind)
  – data races (TSan, valgrind)
  – integer overflows (IOC)
  – API misuse (AppVerifier)
Known applications in security

• Fault injection
  – stability testing, e.g. failing every $n^{th}$ allocation (AppVerifier)
  – in-memory fuzzing

• Detection of active exploitation (malware pipelines)
  – running code outside of executable images and JIT regions → shellcode indicator
  – no CALL before RETN → ROP indicator
  – etc.
• Coverage-based corpus distillation helped Google find tons of bugs
  – proprietary: Adobe Flash, Adobe Reader, Chrome PDF Reader, ...
  – open-source: FFmpeg, FreeType2, libexif, libtiff, ...

• AddressSanitizer contributed, too:
  – 1000+ vulnerabilities in Chromium, WebKit, Mozilla, webrtc, Perl, PHP, ...

• Too many to list them all.
If it works so well....

... why not apply it to whole operating system kernels instead of individual programs?

e.g. for vulnerability discovery.
Well, there is something...

Driver Verifier

- Microsoft tool.
- Tests device drivers for common mistakes.
- Limited subset of detectable bad states.
  - mostly API misuse.

Not much beyond it, though (for Windows at least).
NOT ENOUGH.

(feels like a highly underestimated potential)
Motivation

• If there is a buffer overflow or double free(), the kernel will crash anyway...

  but

• It turns out there are a number of vulnerability classes which don’t explicitly manifest themselves in the kernel.
  – even though they’re triggered all the time.

• We could detect them!
Motivation (cont’d)

• Kernels **do** have vulnerabilities.
  – mostly local (elevation of privileges)
  – these are becoming an important component in remote exploit chains (sandbox escapes)

• Kernel-wide instrumentation is a largely unexplored area.

• Vulnerability hunting automation is cost effective.
  – especially for bugs otherwise difficult to find with manual auditing.
Approach: extending fuzzing

examine() \rightarrow \text{is\_violation}()
Approach: extending fuzzing

\[
\text{examine}(\text{Program state}) \rightarrow \text{is\_violation}(\text{Program state})
\]

- For a kernel, program execution flow includes:
  - booting up
  - execution of all active device drivers in addition to the kernel
  - coverage from normal operation (running services, shell etc.)
  - artificially provoked code paths
  - system termination
Detection: cross-platform kernel bugs

• Memory corruption
  – {stack, heap, static} oob {reads, writes}, use-after-free

• Double memory fetch from ring-3
  – (a.k.a. time-of-check-to-time-of-use, tocttou)

• Use of uninitialized memory
  – stack variables, pool/heap allocations and so forth.

• Copying uninitialized memory to user-mode
  – disclosure of potentially sensitive information processed by the kernel.
Detection: system-specific bugs

Breaking core security assumptions being part of the operating system design or making incorrect ones.

**Windows examples:**

- Referencing user-mode pointers while “Previous Mode” is *KernelMode*.
- Calling `ObReferenceObjectByHandle` with `Type=NULL` in non handle type-agnostic contexts.
In theory, we could target any *wrong* behavior or system state, as long as there is a simple model we can use to detect it.

The simpler the model, the better.
Or just gather information and save it instead of active state examination.
Regular code coverage techniques can be applied to kernels similarly to client applications.

Imagine:
- Instrumenting kernel file format parsing for corpus minimization.
  - `win32k.sys` with bitmaps, fonts, metafiles, ...
  - `nt` and keyboard layouts.
- Instrumenting `nt` / `win32k.sys` to find coverage improving syscall invocations (paired with a guest ring-3 fuzzer).
Sky is the limit

• All other techniques originating from user-mode also apply.
  – e.g. the implementation of different valgrind utilities could be ported to kernel-mode*

Basically, sky is the limit.

* not necessarily trivial
KERNEL INSTRUMENTATION
Initial assumptions

- Instrument software platforms
  - Microsoft Windows
  - Linux
  - FreeBSD, OpenBSD, NetBSD
  - possibly Mac OS X
- Hardware platform: x86 and/or x86-64.
- Instrumentation granularity:
  - per instruction
  - per basic block
  - per memory access
  - per execution of instruction at a specific address ("breakpoints")
How do you instrument?

There are several options.
Software emulators (bochs)

Pros

• Full access to the CPU logic
  – including the ability to change
  – 100% control over the execution environment.
• Ease of development.
• Ease of debugging.

Cons

• Extremely, painfully slow.
• Even slower with additional instrumentation running.
• Limited to virtual (emulated) hardware.
Virtualization: VT-x or SVM

- Operating system (ring 3)
- User land
- Kernel land (ring 0)
- Thin hypervisor (ring -1)
- Instrumentation
- Hardware

- Hardware
- User land
- Thin hypervisor
- Operating system
- Instrumentation
VMM - thin hypervisor

Pros

• Extremely low overhead.
  – compared to emulators.

• Running on real hardware (and their device drivers).

Cons

• Tricky implementation.
• Difficult debugging.
• Partially system-specific.
  – e.g. kernel module running the VMM
• CPU-specific.
  – might require at least model “x” from manufacturer “y”
• Limited ability to change CPU logic.
External scripted debugger (e.g. WinDbg via 1394)
External scripted debugger

Pros

- Relatively easy to implement.
  - depending on debugger scripting language. WinDbg + Python is easy.
- Relatively low overhead.
- Real hardware (in case of physical debugging).

Cons

- Slower than VMM
- System-specific (WinDbg vs kgdb vs ...)
- Limited ability to change the CPU logic.
Kernel execution flow

... mov edi, edi
push ebp
mov ebp, esp
mov eax, [ebp+arg_0]
mov edx, [ebp+arg_4]
push esi
mov esi, eax
sub esi, edx
...
## x86 trap hijacking

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Low overhead.</td>
<td>• Tricky implementation with lots of pitfalls.</td>
</tr>
<tr>
<td>• Real hardware.</td>
<td>• Not very elegant.</td>
</tr>
<tr>
<td>• Debuggable.</td>
<td>• Partially system-specific.</td>
</tr>
<tr>
<td></td>
<td>• Limited ability to change CPU logic.</td>
</tr>
</tbody>
</table>
IA-32 hardware debugger
IA-32 hardware debugger

Pros

• Nearly native speed.
• Real hardware.

Cons

• We don’t have a hardware debugger 😞
• Significantly more expensive than other solutions discussed.
• Unsure about scripting capabilities.
• Still unable to modify some internals of the CPU 😞
AND THE WINNER IS...
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
  - SSE2, SSE3, SSSE3, SSE4, SSE5, AVX, both SVM & VT-x etc.
• Correctly hosts all common operating systems.
• Provides extensive instrumentation API.
• A-W-E-S-O-M-E!
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
## Bochs instrumentation callbacks

<table>
<thead>
<tr>
<th>BX_INSTR_INIT_ENV</th>
<th>BX_INSTR_INTERRUPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX_INSTR_EXIT_ENV</td>
<td>BX_INSTR_HWINTERUPT</td>
</tr>
<tr>
<td>BX_INSTR_INITIALIZE</td>
<td>BX_INSTR_CLFLUSH</td>
</tr>
<tr>
<td>BX_INSTR_EXIT</td>
<td>BX_INSTR_CACHE_CNTRL</td>
</tr>
<tr>
<td>BX_INSTR_RESET</td>
<td>BX_INSTR_TLB_CNTRL</td>
</tr>
<tr>
<td>BX_INSTR_HLT</td>
<td>BX_INSTR_PREFETCH_HINT</td>
</tr>
<tr>
<td>BX_INSTR_MWAIT</td>
<td>BX_INSTR_BEFORE_EXECUTION</td>
</tr>
<tr>
<td>BX_INSTR_DEBUG_PROMPT</td>
<td>BX_INSTR_AFTER_EXECUTION</td>
</tr>
<tr>
<td>BX_INSTR_DEBUG_CMD</td>
<td>BX_INSTR_REPEAT_ITERATION</td>
</tr>
<tr>
<td>BX_INSTR_CNEAR_BRANCH_TAKEN</td>
<td>BX_INSTR_LIN_ACCESS</td>
</tr>
<tr>
<td>BX_INSTR_CNEAR_BRANCH_NOT_TAKEN</td>
<td>BX_INSTR_PHY_ACCESS</td>
</tr>
<tr>
<td>BX_INSTR_UCNEAR_BRANCH</td>
<td>BX_INSTR_INP</td>
</tr>
<tr>
<td>BX_INSTR_FAR_BRANCH</td>
<td>BX_INSTR_INP2</td>
</tr>
<tr>
<td>BX_INSTR_OPCODE</td>
<td>BX_INSTR_OUTP</td>
</tr>
<tr>
<td>BX_INSTR_EXCEPTION</td>
<td>BX_INSTR_WRMSR</td>
</tr>
</tbody>
</table>
## Bochs instrumentation callbacks

<table>
<thead>
<tr>
<th>BX_INSTR_INIT_ENV</th>
<th>BX_INSTR_INTERRUPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BX_INSTR_EXIT_ENV</td>
<td>BX_INSTR_HWINTERRUPT</td>
</tr>
<tr>
<td>BX_INSTR_INITIALIZE</td>
<td>BX_INSTR_CLFLUSH</td>
</tr>
<tr>
<td>BX_INSTR_EXIT</td>
<td>BX_INSTR_CACHE_CNTRL</td>
</tr>
<tr>
<td>BX_INSTR_RESET</td>
<td>BX_INSTR_TLB_CNTRL</td>
</tr>
<tr>
<td>BX_INSTR_HLT</td>
<td>BX_INSTR_PREFETCH_HINT</td>
</tr>
<tr>
<td>BX_INSTR_MWAIT</td>
<td>BX_INSTR_BEFORE_EXECUTION</td>
</tr>
<tr>
<td>BX_INSTR_DEBUG_PROMPT</td>
<td>BX_INSTR_AFTER_EXECUTION</td>
</tr>
<tr>
<td>BX_INSTR_DEBUG_CMD</td>
<td>BX_INSTR_REPEAT_ITERATION</td>
</tr>
<tr>
<td>BX_INSTR_CNEAR_BRANCH_TAKEN</td>
<td>BX_INSTR_LIN_ACCESS</td>
</tr>
<tr>
<td>BX_INSTR_CNEAR_BRANCH_NOT_TAKEN</td>
<td>BX_INSTR_PHY_ACCESS</td>
</tr>
<tr>
<td>BX_INSTR_UCNEAR_BRANCH</td>
<td>BX_INSTR_INP</td>
</tr>
<tr>
<td>BX_INSTR_FAR_BRANCH</td>
<td>BX_INSTR_INP2</td>
</tr>
<tr>
<td>BX_INSTR_OPCODE</td>
<td>BX_INSTR_OUTP</td>
</tr>
<tr>
<td>BX_INSTR_EXCEPTION</td>
<td>BX_INSTR_WRMSR</td>
</tr>
</tbody>
</table>
**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_OUTP
- BX_INSTR_WRMSR
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
### 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
Performance (short story)
Performance (long story)

• On a modern PC (decent i7), non-instrumented guests run at up to 80MHz.
  – 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 1-40MHz.
    • still acceptable for research purposes (not regular work).
  – Simple logic and optimal implementation is a key to success.
Having the technical ability to instrument any operating system in any way... what shall we start with?
DOUBLE FETCHES
Quick introduction

**Time-of-check-to-time-of-use**

„Inconsistency between the checking of a condition and the use of the results of that check.”

• *Double fetch* is a specific case of *tocttou*
  – user address space is shared across ring0 / ring3.
  – userland memory can be modified at any time by concurrent ring3 thread.
  – if the kernel assumes consistency of a userland value between any two points in time, it’s (most likely) a bug.
Example and how it all started

win32k!SfnINOUTSTYLECHANGE 6 months ago

.text:BF8C3120 mov eax, _W32UserProbeAddress
.text:BF8C3125 cmp ecx, eax
[...]
.text:BF8C3154 cmp [ecx+8], eax
.text:BF8C3157 jnb short loc_BF8C315C
.text:BF8C3159 mov eax, [ecx+8]

• 27 instances identified in win32k.sys in Q4 2012.
• Fixed in February 2013.
• Allowed for disclosure of arbitrary kernel memory to ring-3.
Trivia

• Double fetches occur within consistent code blocks.
  – single system call, single IOCTL handler.

• Only local vulnerabilities (code execution required)
  – Elevation of Privileges
    • primarily buffer overflows and write-what-where conditions.
  – Information Disclosure
    • arbitrary reads and under-filled buffers.
  – all sorts of Denial of Service
    • due to failed exploitation of the two previous items.
They are race conditions after all – exploitation takes some advanced CPU-delaying and scheduler feng shui.

- Some exploitation techniques detailed by sgrakkyu and twiz in 2007 [1].
- Check our SyScan 2013 slides [2], white-paper [3] and follow-up post [4].
Detection via instrumentation – general idea

**Step 1**
Collect information about all memory accesses throughout the operating system lifespan.

**Step 2**
Find pairs of kernel→user references such that both:
- are within the same thread.
- are within the same system call invocation.
- access the same memory location.

**Step 3**
Filter out known false positives and manually inspect remaining reports in search of actual bugs.
Memory access characteristics

- Quite a lot of information is required to describe each access.
  - linear address of accessed memory
  - length of access $\in \{1, 2, 4, 8, 10, 16, 32\}$
  - access type $\in \{\text{read, write, read+write, execute}\}$
  - linear address of accessing instruction
  - unique identifier of syscall invocation
  - system call number
  - process name
  - unique thread id: (pid, tid, creation_time)
  - complete callstack
    - module name
    - module base
    - offset from base
  - instruction disassembly (or opcode bytes)
Quite a lot of information is required to describe each access.

- linear address of accessed memory
- length of access ∈ \{1, 2, 4, 8, 10, 16, 32\}
- access type ∈ \{read, write, read+write, execute\}
- linear address of accessing instruction
- unique identifier of syscall invocation
- system call number
- process name
- unique thread id: (pid, tid, creation_time)
- complete callstack
  - module name
  - module base
  - offset from base
- instruction disassembly (or opcode bytes)
Quite a lot of information is required to describe each access.

- linear address of accessed memory
- length of access $\in \{1, 2, 4, 8, 10, 16, 32\}$
- access type $\in \{\text{read, write, read+write, execute}\}$
- linear address of accessing instruction
- unique identifier of syscall invocation
- system call number
- process name
- unique thread id: (pid, tid, creation_time)
- complete callstack
  - module name
  - module base
  - offset from base
- instruction disassembly (or opcode bytes)
Our implementation

• Create a „memlog.bin” database of all memory accesses by running OS through instrumented Bochs for a few days.
• Split the file into thread-specific logs.
• Run the doublefetch utility over each of them.
• Symbolize the resulting reports.

Ready for manual examination.
Bochspwn report

[pid/tid/ct: 00000049/00000049/0028fc5d3be580] {init}
00000003, 0000000b: READ of 950226c (7 * 4 bytes),
pc = c12d89d1 [ mov edx, dword ptr ds:[eax-3] ]

#0 0xc12d89d1 (kernel+002d89d1)
   __get_user_4   arch/x86/lib/getuser.S:69

#1 0xc115a910 (kernel+0015a910)
   do_execve_common   fs/exec.c:1553

#2 0xc115aa27 (kernel+0015aa27)
   do_execve   fs/exec.c:1621

#3 0xc1019517 (kernel+00019517)
   sys_execve   arch/x86/kernel/process.c:356

...
Platform differences

• Generic idea, largely system-specific implementation
  – different distinction between „user” and „kernel” address space
  – different system structures to traverse
  – different ways to generate code coverage
  – different false positives in reports
  – different results 😊

• Let’s look into each of them separately.
Memory boundaries

- Virtual address space divided in two (user / kernel)
  - simple “less than” and “greater than” can be applied to Eip.
- Windows x86
  - boundary \(0x80000000\), user land below, kernel land above
  - can be \(0xc0000000\) for /3G switch, we didn’t use it
- Windows x86-64
  - non-continuous address space
  - below \(0x0000007ff00000000\) user land.
  - above \(0xfffff80000000000\) kernel land.
Process/thread structure traversal

- **Kernel**
  - x86
    - FS.base
  - x86-64
    - GS.base

- **KPCR**
  - Irql

- **KPCR**
  - Current Thread

- **ETHREAD**
  - KTHREAD
    - Process
    - CreateTime
  - Cid
    - Unique Process
    - Unique Thread

- **EPROCESS**
  - Image FileName
Device driver list traversal

**Kernel FS.base**

- **KPCR**
  - KdVersion Block
- **DBGKD_GET_VERSION64**
  - PsLoaded ModuleList

**x86**

- **ntoskrnl.exe**
  - PsLoaded ModuleList

**x86-64**

- **LDR_MODULE**
  - BaseAddress
  - SizeOfImage
  - BaseDllName
  - Flink
  - Blink
- **LDR_MODULE**
  - BaseAddress
  - SizeOfImage
  - BaseDllName
  - Flink
  - Blink
- **LDR_MODULE**
  - BaseAddress
  - SizeOfImage
  - BaseDllName
  - Flink
  - Blink
Common false positives

• Reports originating from the „System” process during early boot-up
  – the user/kernel boundary doesn’t apply yet.
  – neutralized by ignoring the process entirely.
    • i.e. discard memory accesses from pid=0 and pid=4
    • also speeds up the guest significantly

• Reports from APC-related kernel routines
  – Neutralized by reading Irql from KPCR and ignoring all \texttt{Irql=APC\_LEVEL} references.
Common false positives

• Reports originating from the CI.dll kernel module (digital executable signatures)
  – filtered out by removing all database entries with “CI.dll” somewhere in the callstack in post-processing.

• Numerous false positives in messaging related routines in win32k.sys
  – filtered out by filtering the final logs against a black-list of known bad functions.
Memory probing

• The Windows kernel has several ways to probe user memory
  – public ProbeForRead, ProbeForWrite API
  – internal functions and macros (inlined in the code)

• Two most prevalent patterns

```assembly
; ecx = user-provided address
mov eax, [ecx]
mov [ecx], eax

; ecx = user-provided address
mov al, [ecx]
```
Memory probing

• First pattern mitigated by:
  – logging all 4-byte “write” accesses in addition to “read”
  – implementing an anti-probe mechanism in `doublefetch.cc`
    • if a “write” of the same (address, size) immediately follows a “read”,
      discard the “read”.

• Second pattern mitigated by ignoring all reads of less than two 2 bytes.
  – extremely rare, ~99% of 1-byte reads is probing.
Symbolization

- Microsoft supports a “Debug Help” DLL – DbgHelp.dll
  - has API for symbol resolving
  - `SymInitialize`, `SymLoadModule64`, `SymFromAddr`

- Required PDB to be downloaded from Microsoft Symbol Server

- Trivial to implement one’s own resolver.
89 potential new issues discovered
  - part of the initial 27 bugs were also rediscovered
  - all reported to Microsoft (November 2012 – January 2013)

37 EoPs officially addressed by MS13-016, MS13-017, MS13-031, MS13-036, MS13-046

13 issues were classified as “Local DoS” only

One big problem is still being worked on, and three cases are under re-investigation.

The rest were non-exploitable / non-issues / etc.
The less official results

• Microsoft were very receptive to the reports.
• There is evidence that extensive variant analysis was performed.
  – `nt!ApphelpCacheQuery`, `win32k!NtUserDisplayConfigGetDeviceInfo`, examples are all around.
  – also three of our original reports were fixed as variants with no CVE.
  – we have no idea how many internal discoveries were fixed, but probably a few dozens.
• We also shared Bochspwn with MSFT, but have no official confirmation on whether they use the code or concept.
The logs are out

- We are releasing all valid Bochspwn reports from our runs against Windows, Linux, FreeBSD.
  - MSFT assessed a majority of the reports as DoS or non-issue.
    - we don’t have resources to investigate them all.
    - let the larger collective confirm.
  - Some Windows issues have not been fixed for 9 months after the original reports. This is by far too long.
  - Logs from other systems are released for reference, and again, verification.
Final thoughts

• Windows kernel is designed/written poorly with regards to reading user land data
  – no pointer annotations (in contrast to Linux __user)
  – no dedicated fetch functions (in contrast to copyin / copyout)
  – no strict data-fetching policies; everyone do as they will.

• Bugs are bound to occur.

• The only problem: generating coverage.
  – imagine: we found ~40 and motivated the discovery of dozens of further bugs by not much more than just booting the system up.
Improving code coverage

• What we did:
  – system boot up
  – typical navigation in the system: Internet Explorer, Wordpad, Notepad, Registry Editor, Control Panel, builtin games
  – playing multimedia (video, audio)
  – starting Starcraft 1
  – running the Wine Conformance Tests

• Far too little.
Improving code coverage

• All further ideas are extremely welcome.
• We currently believe a moderately-smart system call fuzzer should dramatically improve the coverage.
  – in the works. new Windows double-fetch iterations will follow soon. 😊
Coverage: instructions in base images

- **ntoskrnl.exe percentage coverage**
  - Boot + terminate: 35.53%
  - All tests: 38.53%

- **win32k.sys percentage coverage**
  - Boot + terminate: 29.46%
  - All tests: 48.93%
Coverage: system calls invoked

- **ntoskrnl.exe**
  - boot + terminate: 196
  - all tests: 227

- **win32k.sys**
  - boot + terminate: 291
  - all tests: 460
Coverage: fetch instructions executed

- ntoskrnl.exe
  - boot + terminate: 947
  - all tests: 1093

- win32k.sys
  - boot + terminate: 764
  - all tests: 1683

- all modules
  - boot + terminate: 2836
  - all tests: 4064
LINUX
Bochspwn vs Linux

- And by Linux we mean: Ubuntu Server 13.04 64-bit
- Stock kernel: Linux 3.5.0-23-generic
Linux: process and thread information

- Getting to thread-specific data.
  - Step 1. Get kernel-mode stack pointer.

![Diagram showing Linux process and thread information](image)
Linux: process and thread information

- Step 2: Getting to thread_info.

```
(struct thread_info *)
(RSP & ~(THREAD_SIZE - 1))

FFFFFFFFFFFFFFFFFE000
```
Linux: process and thread information

- Step 3: Diving deeper.
Linux: module information

• Getting to the modules

modules
Linux: callstack

Stock kernel is compiled with frame pointers (RBP)
Linux: callstack

• Callstack
  – Sometimes missing second frame?

#0 0xc12d89d1 __get_user_4 getuser.S:69
#? ?????????? ?????????????????? ?????????????????:???
#1 0xc115a910 do_execve_common exec.c:1553
#2 0xc115aa27 do_execve exec.c:1621
#3 0xc1019517 sys_execve process.c:356
#4 0xc15e9eee ptregs_execve entry_32.S:730
Linux: callstack

• Callstack
  – Sometimes missing second frame?
  • Functions in .S do not preserve frame pointers.
  • What can we do about it?
    – Save (per thread) call stack – it’s slow.
    – Limit it only to .S functions which trigger events.
    – Record only the last one (.S is always #0).
Linux: callstack

- Callstack
  - Sometimes missing a frame?
  - Inline functions.
    - Compile with ignoring inline requests.
    - Actually a symbolization problem.
Linux: symbolization

• Stock kernel symbols available in repositories.

• GNU `addr2line` tool + a short python script.
Linux: coverage

• Getting decent coverage
  – Fuzzers:
    • iknowthis - https://code.google.com/p/iknowthis/
    • Trinity - http://codemonkey.org.uk/projects/trinity/
    • fsfuzzer
    • other
  – Tests
    • Linux Test Project (ltp) - http://ltp.sourceforge.net/
Coverage: instructions in base images

- 21.21% (boot + terminate)
- 28.47% (all tests)

Kernel image coverage.
Coverage: system calls invoked

 Kernel image

- boot + terminate
- all tests

Coverage: system calls invoked

169

301
Linux: coverage statistics

• Log sizes: 78 MB to 189 GB

• Total unique threads in all runs: c.a. 50k

• Double fetch logs (unfiltered): 70 KB to 200 KB
Linux: results

Bochspwn vs GNU/Linux

Final result:

??? bugs found
Linux: results

Bochspwn vs GNU/Linux

Final result:

0?? bugs found
Linux: results

Bochspwn vs GNU/Linux

Final result:

00? bugs found
Linux: results

Bochs pwn vs GNU/Linux

Final result:

000 bugs found
• Results:
  – Nothing found.

• Why is that?! (a.k.a. documenting failure)
  – Copy functions.
  – Annotations.
  – Overall design.
Linux: results

- User-to-kernel copy functions
  - `do_strncpy_from_user` + `do_strnlen_user`
  - `__get_user_{1,2,4,8}` and `__get_user` macro
  - `__copy_user_{zeroing | intel | nocache}`
  - `copy_user_generic_unrolled`,
    `copy_user_generic_string`,
    `copy_user_enhanced_fast_string`
  - other? (`copy_page`)
Linux: results

• Annotations
  
  ```c
  #define __user
  
  __attribute__((
    noderef,
    address_space(1)
  ))
  ```

Sponsored by **sparse**: https://sparse.wiki.kernel.org/index.php/Main_Page
Linux: results

• Overall design
  – No (or not too many) deep structures.
  – Need to call a function to dereference user pointer.
Double fetch conditions (non-issues)

1. Counting the argv [] list length (do_execve_common)
2. strlen () + memcpy ()
3. -ESTALE path resolving retry
4. writing to file in a ext4 file system
5. seeding the blocking/nonblocking random pools
6. usage of XSAVE / XRSTOR instructions
7. *__getsockopt implementations
Example of a non-issue (getsockopt)

```c
static int do_tcp_getsockopt(struct sock *sk, int level,
   int optname, char __user *optval, int __user *
   *optlen)
{
   [...]
   int val, len;

   if (get_user(len, optlen))
       return -EFAULT;

   len = min_t(unsigned int, len, sizeof(int));

   if (len < 0)
       return -EINVAL;
```
Example of a non-issue (getsockopt)

switch (optname) {

[...]

case TCP_INFO: {
    struct tcp_info info;

    if (get_user(len, optlen))
        return -EFAULT;

[...]

len = min_t(unsigned int, len, sizeof(info));
if (put_user(len, optlen))
    return -EFAULT;
if (copy_to_user(optval, &info, len))
    return -EFAULT;
case TCP_CONGESTION:
    if (get_user(len, optlen))
        return -EFAULT;
    len = min_t(unsigned int, len, TCP_CA_NAME_MAX);

[...]

case TCP_COOKIE_TRANSACTIONS: {
    struct tcp_cookie_transactions ctd;
    struct tcp_cookie_values *cvp = tp->cookie_values;

    if (get_user(len, optlen))
        return -EFAULT;
    if (len < sizeof(ctd))
        return -EINVAL;

[...]
Bochspwn vs. FreeBSD

- FreeBSD 9.1 64-bit
- Stock kernel (GENERIC)
FreeBSD: process and thread

• Getting to thread-specific data.

  **Ring 3**

  MSR C0000102H
  (MSR_KERNELGSBase)

  pcpu->msr.kernelgsbase

  **Ring 0**

  GS.base

  pcpu->get_segment_base(BX_SEG_REG_GS)
FreeBSD: process and thread

• Getting to thread-specific data.
  – Step 2. Diving deeper.
Getting to the modules

FreeBSD: modules

- struct module
  - queue
  - file
- struct linker_file
  - name
  - pathname
  - address
  - size
- pathname[]
FreeBSD: modules

• Getting to the modules?
  – by default there were 477 (sic!) registered modules.
  – ... and all of them were in the kernel image.
  – perhaps we can just ignore them? Yes.

  • Linux emulation layer is an external module.
FreeBSD: callstack

• Callstack & symbolization

  – Exactly the same as in Ubuntu:
    • RBP present in most functions.
    • Assembly function do not preserve RBP on stack.

  – Stock kernel symbols available in:

    /boot/kernel/kernel.symbols
FreeBSD: coverage

• Fuzzers
  – lf6 and netusse - https://code.google.com/p/netusse/
  – fsfuzzer
  – Trinity

• Tests
  – regression -
    http://svnweb.freebsd.org/base/head/tools/regression/
FreeBSD: results

Bochspwn vs FreeBSD

Final result:

??? bugs found
FreeBSD: results

Bochspwn vs FreeBSD

Final result:

0?? bugs found
FreeBSD: results

Bochspwn vs FreeBSD

Final result:

00? bugs found
FreeBSD: results

Bochspwn vs FreeBSD

Final result:

000 bugs found
FreeBSD: results

- Documenting failure again.
- Nothing found.
FreeBSD: why nothing found?

- Copying functions again: `fubyte/word/word32` and `copyin/copyinsetr`

- Historically popular bug class.
FreeBSD: a false positive

• kern_select()
  – Read no. 1: sys_generic.c:918 - fubyte
  – Read no. 2: sys_generic.c:968 - copyin

----------------- found double-read of address 0x00000008030070e8
Read no. 1:
[pid/tid/ct: 000002ee/000186c9/fffffff800050c4000] {
  sshd} 0000005e, 0000005d: READ of 0030070e8 (1 * 1 bytes), pc =
  fubyte /usr/src/sys/amd64/amd64/support.S:442
  kern_select /usr/src/sys/kern/sys_generic.c:918
  sys_select /usr/src/sys/kern/sys_generic.c:842
  syscallenter /usr/src/sys/amd64/amd64/../../kern/subr_syscall.c:135

Read no. 2:
[pid/tid/ct: 000002ee/000186c9/fffffff800050c4000] {
  sshd} 0000005e, 0000005d: READ of 0030070e8 (1 * 8 bytes), pc =
  copyin /usr/src/sys/amd64/amd64/support.S:293
  kern_select /usr/src/sys/kern/sys_generic.c:968
  sys_select /usr/src/sys/kern/sys_generic.c:842
  syscallenter /usr/src/sys/amd64/amd64/../../kern/subr_syscall.c:135
FreeBSD: a false positive

• kern_select()

Read no. 1:
error = select_check_badfd(fd_in, nd, ndu, abi_nfdbits);

static int
select_check_badfd(fd_set *fd_in, int nd, int ndu, int abi_nfdbits)
...

    res = fubyte(addr);
    if (res == -1)
        return (EFAULT);
FreeBSD: a false positive

- kern_select()

Read no. 2:

```c
#define getbits(name, x) \
    ... \
    error = copyin(name, ibits[x], ncpubytes); \
    ... \

getbits(fd_in, 0);
getbits(fd_ou, 1);
getbits(fd_ex, 2);
```
FreeBSD: a false positive

• kern_select() :
  – Yes, there is a double-fetch.
  – No, it has no security consequences in kernel-mode.
FreeBSD: other false positives

- execve and same address to binary name and argv[0]
- syslog write to file and tty
- mutex implementation (has locks)
OPENBSD
Bochspwn vs OpenBSD

- OpenBSD 5.3 64-bit
- Customly compiled kernel (DEBUG) for symbols
- Work in progress.
Bochspwn vs OpenBSD

/dev/wd0e (3562f8c0b6c5fda.e): file system is clean; not checking
setting tty flags
pf enabled
ddb.console: 0 → 1
starting network
starting early daemons: syslogd pflogd.
starting RPC daemons:.
savecore: /bsd: kvm_nlist: bad namelist
savecore: /bsd: _dumpdev not in namelist
checking quotas: done.
kvm_mkdb: can’t open /dev/ksyms
kvm_mkdb: /bsd: corrupt file: Inappropriate file type or format
kvm_mkdb: cannot determine executable type of /bsd: Device not configured
clearing /tmp
starting pre-securelevel daemons:.
setting kernel security level: kern.securelevel: 0 → 1
creating runtime link editor directory cache.
preserving editor files.
starting network daemons: sshd sendmail inetd sndiod.
starting local daemons: cron.
Sat Jul 27 11:32:04 CEST 2013

OpenBSD/amd64 (openbsd.my.domain) (ttyCO)

login:
OpenBSD: process and thread

• Getting to thread-specific data.
  – Step 1. Get kernel-mode GS base – exactly the same way as in FreeBSD.
OpenBSD: process and thread

- Getting to thread-specific data.
  - Step 2. Diving deeper.
OpenBSD: modules

Oh. Actually there are no modules on OpenBSD.
OpenBSD: callstack

• Same situation as in FreeBSD/Linux:
  – Stack pointer in RBP / preserved on stack
  – ... but not for functions implemented in assembly.

• Symbolizing:
  – No symbols available for stock kernel.
  – Have to recompile the kernel (DEBUG).
  – GNU addr2line-based script works with no changes.
OpenBSD: left to do

Prepare logs, look for bugs!
Bochspwn vs ...

• There are still many systems to explore.
  – NetBSD
  – OSX
  – Solaris
  – other Linux-based distributions?

• And a lot of ground to cover.
  – Dedicated tools for better coverage.
Virtualization is still a great technology for instrumenting memory accesses.

- all OS instructions execute natively.
- only instructions of desired type are intercepted.
- could be ran *seamlessly* on any (your) workstation.
  - detect bugs while you work.
  - only if you don’t use VMs, though.

Not perfect for every instrumentation.
• Start off with Joanna Rutkowska’s BluePill project [5]
  – load a driver which sets up environment, puts OS in a “jail”
    and run as a VMM.
Hyperpwn – initial concept

Source: J. Rutkowska, Black Hat USA 2006, © BlackHat
Hyperpwn – initial concept

• Instrument only 32-bit operating systems.
• Modify kernel data segment descriptor
  – LDT_ENTRY.ExpandDown = TRUE
  – LDT_ENTRY.Base = 0x00000000
  – LDT_ENTRY.Size = 0x80000000
• All kernel-mode access to 00000000-7fffffff yield a #GP exception.
• VMM intercepts the #GP, performs instrumentation, restores data segment, sets TF, continues.
• VMM intercepts #DB, sets LDT_ENTRY to instrumented, clears TF, continues.
Hyperpwn – initial concept

VMRUN

Bochspwn logic

instrument

restore ds:

set trap flag

$ds = \text{instrumented}$

clear trap flag

#GP handling

#DB handling

VMM loop
Hyperpwn – revised concept

• No VT-x / SVM in Protected Mode 😞
• No memory segmentation in Long Mode 😞
• Revised idea: tamper with Page Tables instead of GDT.
  – clear the “Present” flag for all top-level user-mode entries.
  – has the same effect, but more code required.
Hyperpwn – revised concept
Hyperpwn prototype – coming in Fall 2013
(or not)
Double fetches – what else?

• Instrument Mac OS X.

• Instrument Microsoft Windows with improved code coverage.
  – expect another flood of bugs (or not).

• Static analysis approach.
  – symbolic execution model is fairly interesting.
  – mixing static and dynamic: hint the static analyzer with known user data fetch locations.
BOCHSPWN:MSAN
Kernel memory taint tracking

• Kernel instrumentation is **really** not just about double fetches.

• Example: kernel memory taint tracking.
  – 1 to 2GB wide kernel virtual address space on x86.
  – easily up to 16 bytes of metadata per one kernel byte. that’s a lot!
  – how about tracking an “initialized” property for heaps/pools and stack?

• Possible to detect use of uninitialized memory.
  – similarly to user-mode MemorySanitizer.
Kernel memory taint tracking

- Also, feasible detection of leakage of uninitialized kernel bytes to user-mode!
- We implemented a prototype of Bochspwn:msan ...
- Ran it against Windows a week before BH USA 2013.
**It works!**

12 kernel $\rightarrow$ user pool bytes disclosure vulnerabilities found in Windows 7 and 8.1 and reported to MSFT.

By just booting up the systems.
Bochspwn:msan – more on this later this year
CONCLUSIONS
We are releasing Bochspwn as open-source today.

- official name of the project: **kfetch-toolkit**
- instrumentation + post-processing tools
  - Windows, Linux, BSD support included.
- Apache v2 license.
- [https://github.com/j00ru/kfetch-toolkit](https://github.com/j00ru/kfetch-toolkit)
- read the README for instructions.
Further research

• Kernel instrumentation potential is far from being exhausted.
  – in fact, there are hundreds* of low-hanging fruit waiting to be found.
  – so far it seems most are in Windows.

• Hack on kfetch-toolkit
  – port to other platforms (more exotic?).
  – find novel patterns, models or whole bug classes.
  – improve coverage.
  – test other presented approaches.

* personal estimate.
Final words

1. We really hope the subject will be picked up. 😊
2. If you do and have results (or problems), we’re happy to hear from you!
3. Check our blogs for slides, double-fetch reports from the past and updates.
Thanks for coming!

Questions?

Mateusz “j00ru” Jurczyk
• j00ru.vx@gmail.com
• http://j00ru.vexillium.org/
• @j00ru

Gynvael Coldwind
• gynvael@coldwind.pl
• http://gynvael.coldwind.pl/
• @gynvael

Thanks / shouts to:
Halvar Flake, Solar Designer, Clement Lecigne, Oshogbo, Martynas Venckus, Tavis Ormandy, Vasily Kulikov, Alex Ionescu, ISE
References