CAPTURING 0DAY EXPLOITS WITH PERFECTLY PLACED HARDWARE TRAPS

Cody Pierce . Matt Spisak . Kenneth Fitch
INTRODUCTION
EXPLOIT DETECTION IS A MOVING TARGET

- Exploitation is increasingly more sophisticated
- Creativity in exploitation is hard to plan for in the Security Development Lifecycle (SDL)
- A well financed attacker armed with 0days has the advantage
# Early Prevention to Maintain the Advantage

## Pre-Exploitation
- Software Identification
- Version Identification
- OS Identification

## Exploitation
- Memory Organization
- Vulnerability Preparation
- Vulnerability Trigger

## Post-Exploitation
- Return Oriented Programming
- Payload Execution
- COE
- Code Execution
- Persistence
EARLY PREVENTION TO MAINTAIN THE ADVANTAGE

Pre-Exploitation

- Software Identification
- Version Identification
- OS Identification

Exploitation

- Memory Organization
- Vulnerability Preparation
- Vulnerability Trigger

Post-Exploitation

- Return Oriented Programming
- Payload Execution
- COE
- Code Execution
- Persistence
INTRODUCING HARDWARE ASSISTANCE
INTRODUCING - HARDWARE ASSISTANCE

CPU CORE

PERFORMANCE MONITORING UNIT

BRANCH PREDICTION UNIT
INTRODUCING - PERFORMANCE MONITORING UNIT

CPU CORE

PERFORMANCE MONITORING UNIT

BRANCH PREDICTION UNIT
PERFORMANCE MONITORING UNIT

- A special unit in microprocessor architectures to enable hardware level performance and system information. Often used to optimize hardware and software

- The PMU can be programmed to record dozens of different hardware “events”

- Traditionally reserved for developers and system architects
INTRODUCING - PERFORMANCE MONITORING UNIT

CPU CORE

0x03 LD_BLOCKS
0x0D INT_MISC
0x3C CPU_CLK_UNHALTED
0x48 L1D_PEND_MISS
0x85 ITLB_MISSES
0x89 BR_MISP_EXEC
0xA2 RESOURCESTALLS
0xAE ITLB.ITLB_FLUSH
0xC0 INST RETIRED

PERFORMANCE MONITORING UNIT
PERFORMANCE MONITORING UNIT FOR SECURITY

- “Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters”, Yuan et al., 2011
- “CFIMon: Detecting Violation of Control Flow Integrity using Performance Counters”, Xia et al., 2012
- “kBouncer: Efficient and Transparent ROP Mitigation”, Pappas, 2012
- “ Transparent ROP Detection using CPU Performance Counters”, Li & Crouse, 2014
INTRODUCING - BRANCH PREDICTION UNIT

CPU CORE

PERFORMANCE MONITORING UNIT

BRANCH PREDICTION UNIT

BLACKHAT 2016
BRANCH PREDICTION UNIT

- A unit in microprocessor architectures dedicated to improving the prediction of branch destinations to increase instruction pipeline efficiency
- Better branch prediction can have a large effect on processor performance
- Misprediction penalties can be many clock cycles due to flushing and filling the correct branch into the instruction pipeline
- Indirect branches can be common in C++ applications and predicting them is crucial to performance
INTRODUCING - HARDWARE ASSISTANCE

CPU CORE

BRANCH PREDICTION UNIT

PERFORMANCE MONITORING UNIT

INTERRUPT 0xFE
INTRODUCING
CONTROL FLOW INTEGRITY
CONTROL FLOW INTEGRITY

- Enforcement of legitimate control flow in a program
- Traditionally done with compiler generated instrumentation
- Many different implementation of policy enforcement exist but the basic idea is to validate each indirect control flow transfer against a static list of trusted functions
INTRODUCING - CONTROL FLOW INTEGRITY

CONTROL BRANCH

CFI POLICY ENFORCEMENT

DESTINATION

TERMINATE
ALTERNATIVE CFI IMPLEMENTATIONS

- Control Flow Guard (CFG), Microsoft, 2014
- Control-flow Enforcement Technology (CET), Intel, TBD?
- Return Address Protection/Indirect Control Transfer Protection (RAP/ICTP), PaX Team, 2015
ALTERNATIVE CFI IMPLEMENTATIONS

- Control Flow Guard (CFG), Microsoft, 2014
- Control-flow Enforcement Technology (CET), Intel, TBD?
- Return Address Protection/Indirect Control Transfer Protection (RAP/ICTP), PaX Team, 2015
- While these are very strong implementations they require recompilation, updated software/kernel/OS, or aren’t cross platform
SCOPING OUR RESEARCH TO FILL THE GAP

- No source code access
- Cross-Platform OS support
- 32 and 64 bit support
- No pre-processing of binaries or CFG reconstruction
- Not specific to a single bug-class or exploit technique such as Use-After-Free (UAF) or Return-oriented Programming (ROP)
- Overhead must be acceptable in benchmarks and subjective user experience
REAL-WORLD VERIFICATION

- Approach must be verified using “real” exploits and “real” software
- Cyber Grand Challenge samples
- Research community PoCs
- Metasploit modules
- Exploit Kit samples including previous 0days
- Internally developed exploits
CFI APPROACH
HARDWARE-ASSISTED CONTROL FLOW INTEGRITY (HA-CFI)

- Hijacked indirect branches almost always mispredicted by BPU
- HA-CFI Approach:
  - Use Intel PMU to trap all mispredicted indirect branches
  - Requires setting counter to -1
  - Use ISR for CFI policy: validate **indirect branch** destinations in real-time
  - Initial prototype in Linux
OUR INITIAL APPROACH

PERFORMANCE MONITORING UNIT

CORE 1

IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

CORE 2

IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

...

CORE N

IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

KERNEL

PMU INTERRUPT SERVICE ROUTINE

USER MODE

DATA COLLECTION CLIENT

PMU INTERRUPT SERVICE ROUTINE

PYTHON POST-PROCESSING

PMU INTERRUPT SERVICE ROUTINE
BUT WONT ALL THOSE INTERRUPTS BE EXPENSIVE?
# INDIRECT BRANCH CFI COMPARISON

<table>
<thead>
<tr>
<th></th>
<th>Source Code Required</th>
<th>Patching Required</th>
<th>Overhead</th>
<th>CFI Logic Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary Rewriting</strong></td>
<td>NO</td>
<td>YES</td>
<td>LOW</td>
<td>100% FOR PROTECTED CALLS</td>
</tr>
<tr>
<td><strong>Compiler Transformation</strong></td>
<td>YES</td>
<td>NO</td>
<td>LOW</td>
<td>100% FOR PROTECTED CALLS</td>
</tr>
<tr>
<td><strong>PMU-Assisted</strong></td>
<td>NO</td>
<td>NO</td>
<td>MEDIUM</td>
<td>ONLY WHEN MISPREDICTED 1%-20%</td>
</tr>
</tbody>
</table>
PROGRAMMING THE PMU

PERFORMANCE MONITORING UNIT

CORE 1
- IA32_PMC0 = -1
- IA32_PERFEVTSEL0 = MISP INDIRECT CALL

CORE 2
- IA32_PMC0 = -1
- IA32_PERFEVTSEL0 = MISP INDIRECT CALL

... (same pattern for CORE N)

KERNEL
- PMU INTERRUPT SERVICE ROUTINE
  - PMU INTERRUPT SERVICE ROUTINE
  - PMU INTERRUPT SERVICE ROUTINE

USER MODE
- DATA COLLECTION CLIENT
- PYTHON POST-PROCESSING
PROGRAMMING THE PMU

- Controlled by several Model Specific Registers (MSRs)
  - IA32_PERF_GLOBAL_CTRL : global enable/disable of counters
  - IA32_PERFEVTSELx : event to count, mode inclusion bits, interrupt bit
  - IA32_PMCx : counter value
  - IA32_PERF_GLOBAL_STATUS / IA32_PERF_GLOBAL_OVF_CTRL
    - counter overflow status and clear registers

- Additional references: Threads 2014 [Li et al], BH USA 2015 [Herath, Fogh]
INDIRECT BRANCH – INTEL PMU EVENTS

<table>
<thead>
<tr>
<th>EVENT NAME</th>
<th>UMASK</th>
<th>CODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR_MISP RETIRED.NEAR_CALL</td>
<td>0x02</td>
<td>0xC5</td>
<td>Direct and indirect mispredicted near call instructions retired</td>
</tr>
<tr>
<td>BR_MISP_EXEC.TAKEN_INDIRECT_NEAR_CALL</td>
<td>0xA0</td>
<td>0x89</td>
<td>Taken speculative and retired mispredicted indirect calls</td>
</tr>
</tbody>
</table>

- BR_MISP RETIRED (PEBS) counts retired only, includes direct and indirect
- BR_MISP_EXEC includes speculative events == branches falsely labeled as mispredicted
- Opted to use BR_MISP RETIRED.NEAR_CALL since more precise and fewer Interrupts
THE INTERRUPT SERVICE ROUTINE

PERFORMANCE MONITORING UNIT

<table>
<thead>
<tr>
<th>CORE 1</th>
<th>CORE 2</th>
<th>...</th>
<th>CORE N</th>
</tr>
</thead>
<tbody>
<tr>
<td>IA32_PMCO = -1</td>
<td>IA32_PERFEVTSEL0 = MISP INDIRECT CALL</td>
<td>...</td>
<td>IA32_PMCO = -1</td>
</tr>
</tbody>
</table>

KERNEL

PMU INTERRUPT SERVICE ROUTINE

DATA COLLECTION CLIENT

USER MODE

PMU INTERRUPT SERVICE ROUTINE

PYTHON POST-PROCESSING
PMU TRAPS “ON PAPER”

IA32_PMC0: 0xFFFFFFFF (-1)   Event: 0x5102C5

<table>
<thead>
<tr>
<th>PMC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0x1000: MOV rax, [rsi]</td>
</tr>
<tr>
<td>-1</td>
<td>0x1003: MOV rdi, [rax+0x78]</td>
</tr>
<tr>
<td>-1</td>
<td>0x1007: CALL rdi</td>
</tr>
<tr>
<td>0</td>
<td>0xB890: MOV rax, rsp</td>
</tr>
<tr>
<td></td>
<td>0xB893: MOV [rax+0x20], r9d</td>
</tr>
</tbody>
</table>

RIP: 0xB890

PMU ISR

```c
#ifdef WINDOWS
   ip = KTRAP_FRAME.RIP;
#else
   ip = pt_regs.rip;
#endif

//Apply CFI policy to RIP
do_cfi(ip);
```
# PMU TRAPS IN THE REAL WORLD

**IA32_PMC0**: 0xFFFFFFFF (-1)  
**Event**: 0x5102C5

<table>
<thead>
<tr>
<th>PMC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0x1000: MOV rax, [rsi]</td>
</tr>
<tr>
<td>-1</td>
<td>0x1003: MOV rdi, [rax+0x78]</td>
</tr>
<tr>
<td>-1</td>
<td>0x1007: CALL rdi</td>
</tr>
</tbody>
</table>

**foo:**

<table>
<thead>
<tr>
<th></th>
<th>0x890: MOV rax, rsp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x893: MOV [rax+0x20], r9d</td>
</tr>
</tbody>
</table>

**RIP**: 0xB893

---

**Skid = 1 Instruction**

PMI

PMU ISR

?
PMU TRAPS IN THE REAL WORLD

- Due to instruction skid after overflow, no guarantee saved IP is address of branch destination
- AMD docs state skid could be up to 72 instructions
- We found 1 instruction skid (or none) to be most common on Intel
- Need a more precise way to get branch target address on PMU overflow
LBR TO THE RESCUE

- Intel Last Branch Record (LBR) can provide us precise branch addresses
- Configured and accessed via MSRs:
  - IA32_DEBUGCTL : Enable/Disable bit, Freeze on PMI bit
  - LBR_SELECT : filter types of branches
  - LASTBRANCH_x_FROM_IP / LASTBRANCH_x_TO_IP : LBR stack entries
  - LBR_TOS : Offset that points to current top of LBR stack
LBR_to_the_Rescue

LBR_SELECT = 0x1ED (Indirect Calls in ring > 0)

easy first check in ISR
PMU TRAPS WITH LBR PRECISION

IA32_PMC0: 0xFFFFFFFF (-1) Event: 0x5102C5

<table>
<thead>
<tr>
<th>PMC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0x1000: MOV rax, [rsi]</td>
</tr>
<tr>
<td>-1</td>
<td>0x1003: MOV rdi, [rax+0x78]</td>
</tr>
<tr>
<td>-1</td>
<td>0x1007: CALL rdi</td>
</tr>
</tbody>
</table>

foo:

| 0   | 0xB890: MOV rax, rsp |
| 0   | 0xB893: MOV [rax+0x20], r9d |

RIP: 0xB893

PMU ISR

//Get LBR TO
tos = rdmsr(LBR_TOS);
lbr_to = rdmsr(LBR_TO + tos);

//Apply CFI policy to LBR to
do_cfi(lbr_to);
COLLECTING ALL THE DATA

PERFORMANCE MONITORING UNIT

CORE 1
IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

CORE 2
IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

... 

CORE N
IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

KERNEL

PMU INTERRUPT SERVICE ROUTINE

USER MODE

DATA COLLECTION CLIENT

PYTHON POST-PROCESSING

LAST BRANCH RECORD
VALIDATING APPROACH W/ CYBER GRAND CHALLENGE SAMPLES

vrp@ubuntu:~$ miniperf -p 8491 -i 1 -e 0x51a089
Monitoring process: CROMU_00044 (8491)

80007F51FCF6DDBD 7F51FCF9D5F0
80007F51FCF9F62B 7F51FCF9E570
80007F51FCF9E734 7F51FCF9E7C0
80007F51FCF9E4D9 7F51FCF9FC0
80007F51FCF9E69D 7F51FCF9D50
80007F51FCF6DDBD 7F51FCF9D5F0
80007F51FCF9F62B 7F51FCF9E570
80007F51FCF9E734 7F51FCF9E7C0

... 80007F51FCF9D69E 7F51FCF9E7C0
80007F51FCF9E4D9 7F51FCF9FC0
80000000004032B2 41414141

403294:    callq  <_ZN10CUserEntry20GetLastUnreadMessageEv>
403299:    mov    %rax,-0x30(%rbp)
40329d:    mov    -0x30(%rbp),%rax
4032a1:    mov    (%rax),%rax
4032a4:    add    $0x10,%rax
4032a8:    mov    (%rax),%rax
4032ab:    mov    -0x30(%rbp),%rdx
4032af:    mov    %rdx,%rdi
4032b2:    callq  *%rax

// Display last unread message
pCur = pUser->GetLastUnreadMessage();
printf( "From: @s\n", pCur->GetFrom().c_str() );
### INDIRECT BRANCH ANALYSIS - CVE-2014-0556

- **CFI APPROACH**
- **BLACkHAT 2016**

<table>
<thead>
<tr>
<th>ACTIONSCRIPT TRIGGER</th>
<th>HIJACKED CALL SITE</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ByteArray.readBytes()</code></td>
<td>0x33D438: jmp rax</td>
</tr>
<tr>
<td><code>ByteArray.readBytes()</code></td>
<td>0x33D3BC: call qword ptr [rax]</td>
</tr>
<tr>
<td><code>ByteArray.readMultiByte()</code></td>
<td>0x33D1D6: call qword ptr [rax]</td>
</tr>
<tr>
<td><code>ByteArray.readMultiByte()</code></td>
<td>0x33D343: call qword ptr [rax+0x10]</td>
</tr>
<tr>
<td><code>ByteArray.readMultiByte()</code></td>
<td>0x33D1A7: call qword ptr [rax+0x10]</td>
</tr>
<tr>
<td><code>ByteArray.readMultiByte()</code></td>
<td>0x405358: call qword ptr [rax+0x8]</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x33D4A8: jmp rax</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x33D0E7: call qword ptr [rax+0x10]</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x33CFFB: call qword ptr [rax+0x10]</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x40805A: call qword ptr [rcx]</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x33CE48: call qword ptr [rax]</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x33D0B8: call qword ptr [rax]</td>
</tr>
<tr>
<td><code>ByteArray.writeObject()</code></td>
<td>0x33D05E: call qword ptr [rax+0x10]</td>
</tr>
<tr>
<td><code>ByteArray.writeBytes()</code></td>
<td>0x33D4A8: jmp rax</td>
</tr>
<tr>
<td><code>ByteArray.writeUTF()</code></td>
<td>0x33CFFB: call qword ptr [rax+0x10]</td>
</tr>
<tr>
<td><code>ByteArray.writeUTFBytes()</code></td>
<td>0x40805A: call qword ptr [rcx]</td>
</tr>
</tbody>
</table>

- **Ubuntu 14.04.3 LTS x64** Pepper Flash 14.0.0.177
- **Moved to real-world Linux x64 POC, but missed hijack due to JMP**
- **Tweaked ActionScript POC from Chris Evans to generate additional data:**
  - 16 unique hijack points
  - Call / JMP Analysis
- Hijackable indirect JMP slightly more common in Linux binaries
- Indirect JMPs often used for switch statements
- For this talk we will focus exclusively on indirect CALLs
WHAT IS A VALID INDIRECT BRANCH?

Firefox
167,755,264
Branches collected

Dromaeo JavaScript Benchmark
dromaeo.com
CFI APPROACH

OUR FINAL DESIGN

PERFORMANCE MONITORING UNIT

CORE 1

IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

CORE 2

IA32_PMC0 = -1
IA32_PERFEVTSEL0 = MISP INDIRECT CALL

KERNEL

PMU INTERRUPT SERVICE ROUTINE

PMU INTERRUPT SERVICE ROUTINE

PMU INTERRUPT SERVICE ROUTINE

WHITELIST OF VALID BRANCH TARGETS

hacfi.sys / hacfi.ko
WHITELIST GENERATION

- Only after we were able to validate all 160M branches from ELF did we even explore real-time whitelist query
- Generate list on each image load in protected processes
- Overall approach is same on ELF and PE:
  - Find all code pointer addresses present in loaded image
  - Code pointer considered if relative or absolute address points to .text
  - Primarily focus on Exports, Relocations, and “Callbacks”
KEY CHALLENGES

- Receiving PMU Interrupts
- Clearing PMU Interrupts
- Thread Tracking
RECEIVING PMU INTERRUPTS ON WINDOWS

- Modifying the Interrupt Descriptor Table (IDT) for the PMU interrupt will not work for x64 due to PatchGuard
- While investigating how Windows handles PMI, we discovered a non-exported kernel routine in hal.dll
  - \texttt{HalpSetSystemInformation()}
    - \texttt{InformationClass} of \texttt{HalProfileSourceInterruptHandler}
    - Reachable through \texttt{HalDispatchTable} export
RECEIVING PMU INTERRUPTS ON WINDOWS

```c
NT_STATUS _HalpSetSystemInformation(HAL_SET_INFORMATION_CLASS InformationClass, ULONG BufferSize, PVOID *Buffer) {
    // ...

    if (InformationClass == HalProfileSourceInterruptHandler) {
        if (BufferSize != 4)
            return STATUS_INFO_LENGTH_MISMATCH;

        if (_HalpFeatureBits & 1 == 0)
            return STATUS_INVALID_DEVICE_REQUEST;

        if (ProfilingProcessId == 0) {
            _HalpPerfInterruptHandler = Buffer[0];
            if (Buffer[0] != NULL)
                ProfilingProcessId = PsGetCurrentProcessId();
        } else {
            if (PsGetCurrentProcessId() != ProfilingProcessId)
                return STATUS_INVALID_DEVICE_REQUEST;
            _HalpPerfInterruptHandler = Buffer[0];
            ProfilingProcessId = (Buffer[0] ? ProfilingProcessId : 0);
        }
    }

    return STATUS_SUCCESS;
}
// ...
```
IMPLEMENTATION CHALLENGES

RECEIVING PMU INTERRUPTS ON WINDOWS

- Pass in the interrupt handler function and it will be called when a PMI occurs

```c
NTSTATUS status;
PVOID buffer[1];

buffer[0] = profileSourceInterruptHandler;
status = HalpSetSystemInformation(HalProfileSourceInterruptHandler,
   sizeof(PVOID),
   buffer);
```

- Calling (from the same process) with a NULL pointer deregisters the handler

```c
NTSTATUS status;
PVOID buffer[1];

buffer[0] = NULL;
status = HalpSetSystemInformation(HalProfileSourceInterruptHandler,
   sizeof(PVOID),
   buffer);
```
Another issue encountered involved unmasking PMU interrupts from the handler.

PMU interrupts are delivered by the APIC.

In order to acknowledge an interrupt has been handled and to receive future interrupts, a register in the APIC needs to be written.

How this is accomplished depends on the APIC interface used, which differs between Windows versions.
CLEARING PMU INTERRUPTS ON WINDOWS

**xAPIC**
- Existed since Pentium 4
- Windows 7
- APIC Registers are accessed through mapped physical memory
- Register access accomplished using physical memory mapped into kernel virtual memory via `MmMapIoSpace`

**x2APIC**
- Introduced in Nehalem microarch
- Windows 8/8.1
- APIC Registers are accessed via MSRs
- Interface can be accessed with a single `__writemsr` intrinsic

```c
__writemsr(LVT_x2APIC_PMI, 0xFE)
```
RECEIVING PMU INTERRUPTS ON LINUX

- Setting this up on Linux is even simpler
- Register for a Non-Maskable Interrupt (NMI) handler

```c
register_nmi_handler(NMI_LOCAL,
               our_nmi_handler,
               NMI_FLAG_FIRST,
               "hacfi_pmi");

unregister_nmi_handler(NMI_LOCAL, "hacfi_pmi");
```
THREAD TRACKING

- We don’t want to monitor the entire system
- Monitoring can be restricted to a few “high threat” executables
- The PMU doesn’t know anything about thread or process context
THREAD TRACKING ON WINDOWS

- Not so straightforward

- Windows has no (explicit) mechanism for executing arbitrary code at thread context switches

- Without some sort of callback when a thread quantum starts execution, we don’t know when to turn on the PMU counters

- This is a problem
ASYNCHRONOUS PROCEDURE CALLS TO THE RESCUE

> “When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function.” - Microsoft

> Perfect! We could just use APCs to get callbacks, and re-queue a new one whenever we finish the previous

> Not quite that simple, since we don’t track all threads and don’t know when a monitored quantum has ended

> Also, scheduling an APC for the current thread, from an APC handler, leads to an endless APC loop due to the software interrupt
OUR APC SOLUTION

1. Schedule a kernel APC for every thread we want to track
2. Configure PMU to trap all mispredicted branches
3. When we see an interrupt for the wrong thread, schedule a new APC for the previous thread on the processor (or all tracked threads that don’t have one currently queued)
4. Repeat
THREAD TRACKING ON LINUX

- Very straightforward
- `preempt_notifier_init` gives us a simple callback registration for when a thread is preempted

```c
static struct preempt_notifier notifier;
static struct preempt_ops hacfi_preempt_ops = {
    .sched_in = hacfi_notifier_sched_in,
    .sched_out = hacfi_notifier_sched_out
};

static void hacfi_notifier_sched_in(struct preempt_notifier *notifier, int cpu);
static void hacfi_notifier_sched_out(struct preempt_notifier *notifier, struct task_struct *next);
preempt_notifier_init(&notifier, &hacfi_preempt_ops);
```
RESULTS
ANALYSIS OF RESULTS

- Performance Overhead
- Exploit Detection efficacy testing
PERFORMANCE

- We expect there to be a lot of mispredicted branches leading to excessive interrupts
- There is also a minor fixed overhead for each quantum from the APC
- We need to test and see if this is feasible...
How many PMU interrupts are we talking about?

Mispredicted indirect calls during Octane:

- Sandy Bridge: 13,557,147
- Ivy Bridge: 14,678,764
- Haswell: 8,058,444
## PERFORMANCE OVERHEAD

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Baseline</th>
<th>HA-CFI</th>
<th>EMET</th>
</tr>
</thead>
<tbody>
<tr>
<td>PassMark PerformanceTest</td>
<td>score: 940</td>
<td>9% score: 855</td>
<td>3% score: 910</td>
</tr>
<tr>
<td>Dromaeo JavaScript w/ Internet Explorer</td>
<td>325 runs/s</td>
<td>22% 253 runs/s</td>
<td>32% 220 runs/s</td>
</tr>
</tbody>
</table>

*TESTING PERFORMED ON AN INTEL HASWELL CPU*
EXPLOIT DETECTION TESTING

- We needed exploits to test....
- We wanted exploits of recent CVEs for Adobe Flash, Internet Explorer, and Microsoft Office
- To Metasploit!
## EXPLOIT DETECTION TESTING – METASPLOIT

<table>
<thead>
<tr>
<th>VULNERABILITY</th>
<th>TARGET</th>
<th>DETECTION RATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVE-2014-0497</td>
<td>Flash Player 11.7.700.202</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2014-0515</td>
<td>Flash Player 11.7.700.275</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2014-0556</td>
<td>Flash Player 14.0.0.145</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2014-0569</td>
<td>Flash Player 15.0.0.167</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2014-8440</td>
<td>Flash Player 15.0.0.189</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-0311</td>
<td>Flash Player 16.0.0.235</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-0313</td>
<td>Flash Player 16.0.0.296</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-0359</td>
<td>Flash Player 17.0.0.134</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-3090</td>
<td>Flash Player 17.0.0.169</td>
<td>90%</td>
</tr>
<tr>
<td>CVE-2015-3105</td>
<td>Flash Player 17.0.0.188</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-3113</td>
<td>Flash Player 18.0.0.160</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-5119</td>
<td>Flash Player 15.0.0.189</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2015-5122</td>
<td>Flash Player 18.0.0.194</td>
<td>100%</td>
</tr>
<tr>
<td>CVE-2014-1761</td>
<td>Microsoft Word 2010</td>
<td>100%</td>
</tr>
</tbody>
</table>
EXPLOIT DETECTION TESTING

- Metasploit results were great, but what about the bad guys?
- The techniques used in an exploit matter as much or more than the actual vulnerability itself
- We don’t think Metasploit is a great testbed for HA-CFI, due to lack of diversity in exploitation approach
- So we turned to VirusTotal and Exploit Kit samples collected in the wild
EXPLOIT DETECTION TESTING – VIRUSTOTAL

▸ VirusTotal enabled us to test on real-world malware including previously 0day exploits

▸ Decided that samples from some of the more popular exploit kits would be a good basis for testing

▸ Using actual exploits from ‘the wild’ should provide a good sample of exploitation techniques

▸ We chose 48 unique samples for our testbed

7 Exploit Kits
48 Samples
20 Unique CVEs
We analyzed each sample and bucketed them into three separate categories according to exploitation technique:

- **ROP Technique** - Uses standard Return Oriented Programming techniques.

- **ROPless Technique A** - Flash exploitation technique invoking a wrapper routine of `VirtualProtect` to make shellcode executable.

- **ROPless Technique B** - Similar to A, but via hijacking `Method.apply()` in ActionScript to find and invoke `VirtualProtect` directly (Vitaly Toropov).
## Exploit Kit Detection - HA-CFI vs EMET

<table>
<thead>
<tr>
<th>Code Execution Technique</th>
<th># Samples</th>
<th>HA-CFI Detection Rate</th>
<th>EMET Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROP</td>
<td>37</td>
<td>95%</td>
<td>100%</td>
</tr>
<tr>
<td>ROPless Technique A</td>
<td>1</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>ROPless Technique B</td>
<td>10</td>
<td>100%</td>
<td>0%</td>
</tr>
</tbody>
</table>
## Exploit Kit Detection - By Bug Class

<table>
<thead>
<tr>
<th>Bug Class</th>
<th># CVE'S</th>
<th># Samples</th>
<th>HA-CFI Detection Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-of-bounds Write</td>
<td>3</td>
<td>6</td>
<td>83.3%</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>3</td>
<td>6</td>
<td>83.3%</td>
</tr>
<tr>
<td>Integer Overflow</td>
<td>2</td>
<td>6</td>
<td>100%</td>
</tr>
<tr>
<td>Use-After-Free</td>
<td>4</td>
<td>14</td>
<td>100%</td>
</tr>
<tr>
<td>Double Free</td>
<td>2</td>
<td>4</td>
<td>100%</td>
</tr>
<tr>
<td>Type Confusion</td>
<td>3</td>
<td>6</td>
<td>100%</td>
</tr>
<tr>
<td>Race Condition</td>
<td>1</td>
<td>4</td>
<td>100%</td>
</tr>
<tr>
<td>Uninitialized Memory</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
</tbody>
</table>
CASE STUDIES
CLASSIC ROP TECHNIQUE

- **CVE-2015-2419**: Double-free in jscript9 (MS15-065)

- Magnitude EK Sample

  HA-CFI blocks the initial hijack.
  RIP in ISR = jscript9 + 3BE32

  ```
  jscript9 + A7541 ; JavascriptOperators::OP_SetElementI
  call edi

  jscript9 + 3BE32
  xchg eax,esp ; stack pivot gadget
  ret

  jscript9 + 4B0B5
  mov [ecx+0xC],ax ; CoE help
  ret

  jscript9 + 3BE33
  ret

  kernel32 + 42C15
  VirtualProtectStub ; mark shellcode +X
  ```

StackPivot detected on VirtualProtect
ROPLESS TECHNIQUE #1

- **CVE-2014-0515**: Heap overflow in Adobe Flash (patch in 13.0.0.206)
- Found in many Exploit kits and watering hole attacks
- ROPless technique re-uses VirtualProtect wrapper function in Flash image
- 2 control flow hijacks: one to VP wrapper, second one to shellcode
- Bypasses anti-ROP checks since VP invoked somewhat legitimately
**ROPLESS TECHNIQUE #1**

- **CVE-2014-0515**: Heap overflow in Adobe Flash (patch in 13.0.0.206)

Flash32_12_0_0_77 + 25783D

```
call dword ptr [eax+0x14]
```

HA-CFI detects and blocks the initial hijack.

IP at time of interrupt = Flash32_12_0_0_77 + 3BD636

Flash32_12_0_0_77 + 3BD636

```
push 1
push dword ptr [eax-8]
push dword ptr [eax-4]
call virtual_protect_wrapper
add esp,0xC
ret
```

This branch is also mispredicted

FileReference.cancel()
FUTURE WORK

- Hypervisor support to enable hardware features in virtual machines
- Last Branch Record (LBR) is not fully supported in popular hypervisors
- Performance Monitoring Interrupts (PMI) on overflow is supported in many hypervisors
- We wrote a patch for Xen to enable HA-CFI but it crashes randomly, anyone want to help?
- Just-In-Time code pages are hard to validate with our current whitelist approach
CONCLUSION
EXPLOIT DEFENSE

- Exploit defense needs to detect and prevent exploitation at the earliest phase.
- Compile-time solutions are powerful, but there is room for run-time defense too.
- Defenses focused exclusively on techniques such as Return-oriented Programming can be easily circumvented as new methods get adopted.
- Exploits will continue to “look normal” to bypass prevention checks.
HARDWARE ASSISTED CONTROL FLOW INTEGRITY

- CFI is a powerful first step in ensuring only trusted code paths can be executed.
- Many vulnerabilities must hijack control-flow to achieve code execution.
- Hardware can be leveraged for strong CFI policy enforcement of applications at run-time.
- Many new hardware features are emerging that can be used for exploit defense.
CFI ENFORCEMENT

- CFI policies can be more complex
- Powerful features of the PMU interrupt on branches for prevention are the high IRQL and complete access to context information
- We have more ideas in the works to detect additional events and apply policies to detect abnormal read, writes, and cases where attackers stay within our whitelist
SPECIAL THANKS

- Aaron Lamb, Endgame
- Gabriel Landau, Endgame
- Andrea Limbago, Endgame
- Kafeine, malware.dontneedcoffee.com
- Fellow researchers and vendors working on exploit defense
QUESTIONS?