HARDWARE-ASSISTED ROOTKITS:
ABUSING PERFORMANCE COUNTERS ON THE ARM AND X86 ARCHITECTURES

Matt Spisak
@matspisak

USENIX WOOT ‘16, Austin TX
OUTLINE

- Motivation
- Performance Monitoring Unit
- ARM PMU-Assisted Rootkit
- Intel x64 PMU-Assisted Rootkit
- Analysis
Kernel Patch Protection

- Mitigations such as Kernel Patch Protection complicate rootkit development

- Examples of KPP:
  - Microsoft PatchGuard - x64
  - Samsung TIMA-RKP - TrustZone based kernel Monitor
  - Apple “WatchTower” - iOS 9+
PMU-ASSISTED SECURITY RESEARCH

- Prior art in (x86) PMU for debugging or defensive applications:
  - ROP detection with mispredicted branches
  - Control flow integrity (CFI) using Intel BTS and PMU
  - Rootkit detection (NumChecker) using perf counters
- What about offensive applications for the PMU?
- And what about ARM?
PMU BACKGROUND

- Found in most modern CPUs
- Typically consists of the following components:
  - 1 or more counters (PMCs) for counting events
  - Set of events that can be counted
  - Interrupt (PMI) to signal a counter overflow == sampling period
NORMAL USAGE OF PMU

- Provides real-time feedback on system
- Useful for software/hardware engineers
- Tools:
  - Intel VTune
  - ARM DS-5 Streamline
  - Linux perf / oprofile
  - Apple Xcode Instruments
## PMU Comparison

<table>
<thead>
<tr>
<th></th>
<th>ARMv6</th>
<th>Original Pentium</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Introduced</strong></td>
<td>CP15 system control coprocessor</td>
<td>Model Specific Registers (MSRs)</td>
</tr>
<tr>
<td></td>
<td>memory-mapped (optional)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>external interface (optional)</td>
<td></td>
</tr>
<tr>
<td><strong>Interfaces</strong></td>
<td><strong>Interrupt Delivery</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IRQ</td>
<td>Non-maskable Interrupt (NMI)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interrupt Descriptor Table (IDT)</td>
</tr>
<tr>
<td><strong>Number of Counters</strong></td>
<td>1 cycle, up to 31 general purpose</td>
<td>3 fixed, 4+ general purpose</td>
</tr>
<tr>
<td><strong>Events</strong></td>
<td>Extensible</td>
<td>Fixed (Intel Manual)</td>
</tr>
<tr>
<td><strong>User mode Access</strong></td>
<td>Yes*</td>
<td>No</td>
</tr>
</tbody>
</table>

* *PMUSERENR.EN bit must first be set from PL1/EL1 or higher*
### PMU WORKFLOW

**PMC1: 0xFFFFFFFFFD (-3)**  
Event: 0x08 (Instruction Retired)

<table>
<thead>
<tr>
<th>PMC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>PUSH.W {R4-R11, LR}</td>
</tr>
<tr>
<td>-2</td>
<td>SUB SP, SP, #0x1C</td>
</tr>
<tr>
<td>-1</td>
<td>LDR.W R8, [SP, #0x40]</td>
</tr>
<tr>
<td>0</td>
<td>LDR R4, [R0, #0x18]</td>
</tr>
<tr>
<td>-2</td>
<td>MOV R7, R0</td>
</tr>
<tr>
<td>-1</td>
<td>MOV R6, R1</td>
</tr>
<tr>
<td>0</td>
<td>MOV R0, R4</td>
</tr>
<tr>
<td>-2</td>
<td>MOV R1, R8</td>
</tr>
<tr>
<td>-1</td>
<td>MOV R5, R2</td>
</tr>
<tr>
<td>0</td>
<td>MOV R9, R3</td>
</tr>
<tr>
<td>-2</td>
<td>BL sub_xyz</td>
</tr>
</tbody>
</table>

Overflow occurrences:
- 0
- 1
- 2

<< PMI
PMU-ASSISTED ROOTKIT APPROACH

- Identify candidate rootkit PMU events:
  - event is superset of all system calls
  - overhead of non-syscalls is low
- Trap all occurrences of “rootkit” event
- Attacker controlled ISR can optionally redirect execution
ARM ROOTKIT
## Table C-1 PMU IMPLEMENTATION DEFINED event numbers (continued)

<table>
<thead>
<tr>
<th>Event number</th>
<th>Event mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7F-0x80</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x81</td>
<td>EXC_UNDEF</td>
<td>Exception taken, Undefined Instruction</td>
</tr>
<tr>
<td>0x82</td>
<td>EXC_SVC</td>
<td>Exception taken, Supervisor Call</td>
</tr>
<tr>
<td>0x83</td>
<td>EXC_PABORT</td>
<td>Exception taken, Prefetch Abort</td>
</tr>
<tr>
<td>0x84</td>
<td>EXC_DABORT</td>
<td>Exception taken, Data Abort</td>
</tr>
<tr>
<td>0x85</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x86</td>
<td>EXC_IRQ</td>
<td>Exception taken, IRQ</td>
</tr>
<tr>
<td>0x87</td>
<td>EXC_FIQ</td>
<td>Exception taken, FIQ</td>
</tr>
<tr>
<td>0x88</td>
<td>EXC_SMC</td>
<td>Exception taken, Secure Monitor Call</td>
</tr>
<tr>
<td>0x89</td>
<td>-</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x8A</td>
<td>EXC_HVC</td>
<td>Exception taken, Hypervisor Call</td>
</tr>
</tbody>
</table>

*ARM Architecture Manual ARMv7-A&R - Appendix C*
## Counting the Exception Vector Table

<table>
<thead>
<tr>
<th>EVENT</th>
<th>ARM Design</th>
<th>Custom ARM-based Design</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cortex-A7</td>
<td>Cortex-A53</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SVC</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Data Abort</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>IRQ</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>FIQ</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>SMC</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
ARM ROOTKIT

APPROACH

- Count and trap SVC instructions

Motorola Nexus 6
Qualcomm APQ8084 (Krait) CPU
Android 5.0
CHALLENGES

- Finding the PMU Interrupt
  - Device tree source
  - Brute force: Register unused PPI/SPI’s; trigger PMIs; diff /proc/interrupts
  - Registration: request_percpu_irq(), request_threaded_irq() on Android

- CPU Hot-Plugging
  - Linux/Android provides a callback: register_hotcpu_notifier()

- Interrupt instruction skid
**Challenge: Delayed Instruction Skid**

- PMI triggered at some point after IRQs enabled in `vector_sw i`

- 3 cases we must deal with:
  1. **PMI before branch to syscall routine within vector_sw i**
  2. **PMI at entry point of syscall routine**
  3. **PMI in middle of syscall routine**

```
vector_sw i:

......
MCR p15, 0, R12, c1, c0, 0
CPSIE i
MOV R9, SP, LSR#13
MOV R9, R9, LSL#13
ADR R8, sys_call_table
LDR R10, [R9]
STMFD SPI!, {R4,R5}
TST R10, #0xF00
BNE __sys_trace
CMP R7, #0x17C
ADR LR, ret_fast_syscall
LDRCC PC, [R8, R7, LSL#2]
```

```sys_read```

```
STMFD SPI!, {R0-R2,R4-R9,LR}
MOV R8, R1
MOV R1, SP
MOV R9, R2
BL fg et_light
SUBS R6, R0, #0
......
```
CASE 1: INTERRUPT BEFORE BRANCH TO SYSCALL ROUTINE

```c
#define CPSIE_ADDR 0xC01064D0

... irq_regs = get_irq_regs();
pregs = task_pt_regs(current);
...
if (pregs->ARM_r7 == 0x3 //sys_read
{
    switch (irq_regs->ARM_pc - CPSIE_ADDR) //offset after CPSIE
    {
        //emulate remaining instructions up to LDRCC
        //can skip those involved in resolving syscall routine
        case 0x0
        case 0x4
            irq_regs->ARM_r9 = irq_regs->ARM_sp & 0xFFFFE000
        ...
        case 0x14
        case 0x18
        case 0x1C
        case 0x20
            irq_regs->ARM_lr = ret_fast_syscall;
        case 0x24
            irq_regs->ARM_pc = (uint32_t)hook_sysread;
    }
}
```
Replace saved PC with address of hook

```c
irq_regs = get_irq_regs();
pregs = task_pt_regs(current);
...
if (pregs->ARM_r7 == 0x3) //sys_read
{
  //Check if PMU interrupted at entry point addr of sys_read
  if (pregs->ARM_pc == orig_sys_read)
  {
    preg->ARM_pc = (uint32_t)hook_sys_read;
```

![Diagram showing ARM instructions for sys_read]

```asm
STMFD SP!, {R0-R2,R4-R9,LR}
MOV R8, R1
MOV R1, SP
MOV R9, R2
BL fgets_light
SUBS R6, R0, #0
....
```
CASE 3: MIDDLE OF SYSCALL ROUTINE

- Let syscall routine complete
- Find address of ret_fast_syscall on the stack and replace with address of trampoline
- Trampoline loads LR with ret_fast_syscall, and branches to a post_hook function
- post_hook can retrieve original params, and modify as necessary
Hook `sys_read` in context of `qmuxd` in order to intercept all QMI comms from modem to Android (using only the PMU).
INTEL X64 ROOTKIT
FINDING A SYSCALL COUNTING EVENT

- No obvious PMU event like SVC for ARM
- SYSCALL instruction effectively takes a far branch to address in IA32_LSTAR (e.g. KiSystemCall64 on Windows)
- We can capture Far branches two ways with Intel PMU
  - BR_INST_RETIRED.FAR_BRANCH
  - ROB_MISCEVENTS.LBR_INSERTED (requires LBR filtered to FAR only)
- Using this approach, we can now trap SYSCALLs on x64
**PMU Configuration**

- We can count Far Branches occurring in ring-0 only to reduce additional interrupts branches to user mode

<table>
<thead>
<tr>
<th>PERFEVTSELx</th>
<th>LBR_FILTER</th>
<th>PMCx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event: BR_INST_RETIRED</td>
<td>N/A</td>
<td>-2</td>
</tr>
<tr>
<td>Umask: FAR_BRANCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR encoding: 0x5240C4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Event: ROB_MISC_EVENTS</td>
<td>0xFE</td>
<td>-2</td>
</tr>
<tr>
<td>Umask: LBR_INSERTS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSR encoding: 0x5320CC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**SAMPLING PERIOD**

- Resetting counter value to -1 will result in an interrupt loop due to the iretq in ISR returning to interrupted kernel code
- Choose counter value of -2

<table>
<thead>
<tr>
<th>LBR_SELECT: 0xFE (FAR_BRANCH in RING0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LBR FROM IP</strong></td>
</tr>
<tr>
<td>userspace addr</td>
</tr>
<tr>
<td>IRET from Perf ISR</td>
</tr>
<tr>
<td>userspace addr</td>
</tr>
<tr>
<td>IRET from Perf ISR</td>
</tr>
<tr>
<td>userspace addr</td>
</tr>
<tr>
<td>IRET from Perf ISR</td>
</tr>
</tbody>
</table>
## Implementation with LBR

### KiSystemCall64:

<table>
<thead>
<tr>
<th>PMC</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>MOV  r10, rcx</td>
</tr>
<tr>
<td>-1</td>
<td>MOV  eax, 0x4B</td>
</tr>
<tr>
<td>-1</td>
<td>SYSCALL</td>
</tr>
<tr>
<td>0</td>
<td>MOV  [rbx+1E0h], rcx</td>
</tr>
</tbody>
</table>

### PMU ISR

//Get Last Branch Recorded
tos = rdmsr(LBR_TOS);
lbr_to = rdmsr(LBR_TO + tos)

//check if its a syscall
if lbr_to == rdmsr(IA32_LSTAR)
{
    //This was a syscall
}
Supporting Virtualized Environments (No LBR)

KiSystemCall64:

```
KiSystemCall64: STI
               MOV [rbx+1E0h], rcx
               SYSCALL
```

PMU ISR

```
//Get saved IP
ip = KTRAP_FRAME.RIP;

//check if IP is in KiSystemCall64
if rdmsr(IA32_LSTAR) < ip &&
   ip < (IA32_LSTAR + 0x290)
{
   //This was a syscall
}
```
CHALLENGES

- Finding the PMU Interrupt
- CPU Hot-Plugging
- Interrupt instruction skid
  - 99.9999% of trapped Win7 syscalls caught before jump to resolved routine from SSDT
- In other words, implementation on Windows/x64 is much easier than ARM
LIMITATIONS

- PMU Registers are not persistent to a core reset
- PMU registers could be modified by other kernel code
  - PMU Watchdog may be necessary, a cloaked thread that monitors for someone changing PMU and changing back
- Detection could be based on increase in PMU interrupts serviced, or just presence of particular values in PMU registers
- That said, this is still a practical approach towards rootkits
Evades Kernel Patch Protection, including PatchGuard

Could be extended to hook other IDT/EVT entries beyond SYSCALL/SVC

Overhead is quite low:

- Benchmarking “real-world” usage is tough
- PassMark and JavaScript benchmarks used (2-6% Android, <10% Windows)
- Not noticeable at all with subjective testing and analysis
ACKNOWLEDGEMENTS

- Cody Pierce, Endgame
- Kenny Fitch, Endgame
- Eric Miller, Endgame
- Jamie Butler, Endgame
- Several others at Endgame
- Researchers that paved the way for PMU assisted security research
QUESTIONS?