Breaking the x86 ISA

domas / @xoreaxeaxeax / Black Hat 2017
Christopher Domas
Cyber Security Researcher @ Battelle Memorial Institute

./bio
We don’t trust software.

- We audit it
- We reverse it
- We break it
- We sandbox it

Trust.
But the processor itself?

We blindly trust.

Trust.
Why?

Hardware has all the same problems as software

Secret functionality?
  Appendix H.

Bugs?
  F00F, FDIV, TSX, Hyperthreading, Ryzen

Vulnerabilities?
  SYSRET, cache poisoning, sinkhole

Trust.
We should stop blindly trusting our hardware.
What do we need to worry about?
Historical examples

- ICEBP (f1)
- LOADALL (0f07)
- apicall (0ffff0)

Hidden instructions
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>AND</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Eb</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG+ES (Prefix)</td>
</tr>
<tr>
<td>3</td>
<td>XOR</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Eb</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG+SS (Prefix)</td>
</tr>
<tr>
<td>4</td>
<td>INC[^2]** general register / REX[^2]** prefixes</td>
<td>eAX</td>
<td>REX.B</td>
<td>eBX</td>
<td>REX.X</td>
<td>eBP</td>
<td>REX.RB</td>
</tr>
<tr>
<td>5</td>
<td>AX</td>
<td>I</td>
<td>x9</td>
<td>i</td>
<td>x10</td>
<td>B</td>
<td>i</td>
</tr>
<tr>
<td>6</td>
<td>PUSHA[^4]**</td>
<td>POPA[^4]**</td>
<td>BOUND[^4]**</td>
<td>ARPL[^4]**</td>
<td>Ew, Gv</td>
<td>MOV32[^4]**</td>
<td>AX</td>
</tr>
<tr>
<td>7</td>
<td>O</td>
<td>NO</td>
<td>bind</td>
<td>NC</td>
<td>NBE/A</td>
<td>NBE/A</td>
<td>NBE/A</td>
</tr>
<tr>
<td>8</td>
<td>Immediate Grp 1**A</td>
<td>Eb, Ib</td>
<td>Ex, Iz</td>
<td>Eb, Ib</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Eb, Gb</td>
</tr>
<tr>
<td>9</td>
<td>NGIP</td>
<td>PAUSE(F3)</td>
<td>XCHG ax, rAX</td>
<td>rCX</td>
<td>rDX</td>
<td>rBX</td>
<td>rBP</td>
</tr>
<tr>
<td>A</td>
<td>MOV</td>
<td>AL, Gb</td>
<td>rAX, Gx</td>
<td>rAX, AL</td>
<td>Gv, rAX</td>
<td>MOVSB</td>
<td>MOVSB</td>
</tr>
<tr>
<td>C</td>
<td>Shift Grp 2**A</td>
<td>Eb, Ib</td>
<td>Ex, Ib</td>
<td>Eb, Ib</td>
<td>Eb, Ib</td>
<td>Eb, Ib</td>
<td>Eb, Ib</td>
</tr>
<tr>
<td>D</td>
<td>Shift Grp 2**B</td>
<td>Eb, Cl</td>
<td>Ev, Cl</td>
<td>Ev, Cl</td>
<td>Ev, Cl</td>
<td>Ev, Cl</td>
<td>Ev, Cl</td>
</tr>
<tr>
<td>E</td>
<td>LOOPNE[^4]**</td>
<td>LOOP[^4]**</td>
<td>LOOP[^4]**</td>
<td>LOOP[^4]**</td>
<td>LOOP[^4]**</td>
<td>EB</td>
<td>AL, Ib</td>
</tr>
<tr>
<td>F</td>
<td>LOCK (Prefix)</td>
<td>REP XACQUIRE (Prefix)</td>
<td>REP XACQUIRE (Prefix)</td>
<td>REP XPREP</td>
<td>XPREP</td>
<td>HLT</td>
<td>CMC</td>
</tr>
<tr>
<td>Table A-2. One-byte Opcode Map: (00H — F7H) *</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>PUSH EDX</td>
</tr>
<tr>
<td>1</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>PUSH SS</td>
</tr>
<tr>
<td>2</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG=ES</td>
</tr>
<tr>
<td>3</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, Ib</td>
<td>rAX, Iz</td>
<td>SEG=SS</td>
</tr>
<tr>
<td>4</td>
<td>eAX REX</td>
<td>eCX REX, B</td>
<td>eDX REX, X</td>
<td>eAX REX</td>
<td>eBP REX R</td>
<td>eDI REX, RX</td>
<td>INC general register / REX Prefixes</td>
</tr>
<tr>
<td>5</td>
<td>rAX:8</td>
<td>rCX:10</td>
<td>rDX:10</td>
<td>rBX:11</td>
<td>rSP:12</td>
<td>rBP:13</td>
<td>rSI:14</td>
</tr>
<tr>
<td>6</td>
<td>PUSHA REX</td>
<td>POPA REX</td>
<td>BOUND REX</td>
<td>ARPL Ew, Gw</td>
<td>MOV/SCOX</td>
<td>MOV/SSOX</td>
<td>SEG=FS (Prefix)</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>NO</td>
<td>B/NAE/NC</td>
<td>NB/NAE/NC</td>
<td>ZIE</td>
<td>NZ/NE</td>
<td>B/E/NA</td>
</tr>
<tr>
<td>8</td>
<td>Eb, Ib</td>
<td>Ex, Iz</td>
<td>Eb, Ib</td>
<td>Eb, Gb</td>
<td>Eb, Gv</td>
<td>TEST</td>
<td>XCHG</td>
</tr>
<tr>
<td>9</td>
<td>NDIP PAUSE(F3)</td>
<td>XCHG d, rAX</td>
<td>rCX:9</td>
<td>rDX:10</td>
<td>rBX:11</td>
<td>rBP:13</td>
<td>rSI:14</td>
</tr>
<tr>
<td>A</td>
<td>AL, Gb</td>
<td>rAX, Cx</td>
<td>Gv, AL</td>
<td>Gv, rAX</td>
<td>MOV</td>
<td>MOVSB</td>
<td>MOVSW/DQ</td>
</tr>
<tr>
<td>C</td>
<td>Eb, Ib</td>
<td>Eb, Ib</td>
<td>Ex, Ib</td>
<td>Ex, Ib</td>
<td>Ex, Ib</td>
<td>Ex, Ib</td>
<td>Ex, Ib</td>
</tr>
<tr>
<td>D</td>
<td>Eb, 1</td>
<td>Eb, 1</td>
<td>Eb, CL</td>
<td>Eb, CL</td>
<td>AAM</td>
<td>AAD</td>
<td>XLAT</td>
</tr>
<tr>
<td>E</td>
<td>LOOPNE, LOOPNZ</td>
<td>LOOPNZ</td>
<td>LOOPNZ</td>
<td>LOOPNZ</td>
<td>LOOPNZ</td>
<td>HLT</td>
<td>CMC</td>
</tr>
<tr>
<td>F</td>
<td>LOCK (Prefix)</td>
<td>LOOPNZ</td>
<td>LOOPNZ</td>
<td>LOOPNZ</td>
<td>LOOPNZ</td>
<td>LOCKNZ</td>
<td>LOCKNZ</td>
</tr>
</tbody>
</table>
Table A-2. One-byte Opcode Map: (00H – F7H) *

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, lb</td>
<td>rAX, Iz</td>
<td>POP ES&lt;sup&gt;24&lt;/sup&gt;</td>
<td>PUSH ES&lt;sup&gt;24&lt;/sup&gt;</td>
</tr>
<tr>
<td>1</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, lb</td>
<td>rAX, Iz</td>
<td>PUSH SS&lt;sup&gt;24&lt;/sup&gt;</td>
<td>POP SS&lt;sup&gt;24&lt;/sup&gt;</td>
</tr>
<tr>
<td>2</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, lb</td>
<td>rAX, Iz</td>
<td>SEG=ES (Prefix)</td>
<td>DAA&lt;sup&gt;64&lt;/sup&gt;</td>
</tr>
<tr>
<td>3</td>
<td>Eb, Gb</td>
<td>Ev, Gv</td>
<td>Gb, Eb</td>
<td>Gx, Ev</td>
<td>AL, lb</td>
<td>rAX, Iz</td>
<td>SEG=ES (Prefix)</td>
<td>DAA&lt;sup&gt;64&lt;/sup&gt;</td>
</tr>
<tr>
<td>4</td>
<td>rAX&lt;sup&gt;8&lt;/sup&gt;</td>
<td>rCX&lt;sup&gt;19&lt;/sup&gt;</td>
<td>rDX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>rBX&lt;sup&gt;11&lt;/sup&gt;</td>
<td>rSP&lt;sup&gt;12&lt;/sup&gt;</td>
<td>rDI&lt;sup&gt;15&lt;/sup&gt;</td>
<td>rDI&lt;sup&gt;15&lt;/sup&gt;</td>
<td>rDI&lt;sup&gt;15&lt;/sup&gt;</td>
</tr>
<tr>
<td>5</td>
<td>PUSH&lt;sup&gt;24&lt;/sup&gt; general register</td>
<td>POPA&lt;sup&gt;24&lt;/sup&gt;</td>
<td>POPAD&lt;sup&gt;24&lt;/sup&gt;</td>
<td>BOUND&lt;sup&gt;24&lt;/sup&gt;</td>
<td>Gv, Ma</td>
<td>rAX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>rAX&lt;sup&gt;10&lt;/sup&gt;</td>
<td>rAX&lt;sup&gt;10&lt;/sup&gt;</td>
</tr>
<tr>
<td>6</td>
<td>ELSE</td>
<td>Gv, Ev</td>
<td>MOV/32&lt;sup&gt;32&lt;/sup&gt;</td>
<td>MOV/32&lt;sup&gt;32&lt;/sup&gt;</td>
<td>Gx, Ev</td>
<td>Gx, Ev</td>
<td>Gx, Ev</td>
<td>Gx, Ev</td>
</tr>
<tr>
<td>7</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>8</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>9</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>A</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>B</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>C</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>D</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>E</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
<tr>
<td>F</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
<td>ELSE</td>
</tr>
</tbody>
</table>

---

So... what's this??

- **LOCK** (Prefix)
- **REPNE XACQUIRE** (Prefix)

---

[Table Image]
Find out what’s really there

Goal: Audit the Processor
How to find hidden instructions?

The challenge
Instructions can be one byte …

- inc eax
- 40

… or 15 bytes …

- lock add qword cs:[eax + 4 * eax + 07e06df23h], 0efcdab89h
- 2e 67 f0 48 818480 23df067e 89abcdef

Somewhere on the order of

1,329,227,995,784,915,872,903,807,060,280,344,576 possible instructions

The challenge

https://code.google.com/archive/p/corkami.wikis/x86oddities.wiki
The obvious approaches don’t work:

- Try them all?
  - Only works for RISC
- Try random instructions?
  - Exceptionally poor coverage
- Guided based on documentation?
  - Documentation can’t be trusted (that’s the point)
  - Poor coverage of gaps in the search space

The challenge
Goal:
- Quickly skip over bytes that don’t matter

The challenge
Observation:

The meaningful bytes of an x86 instruction impact either its length or its exception behavior.

The challenge
A depth-first-search algorithm
Guess an instruction:

Tunneling
Execute the instruction:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Tunneling
Observe its length:

Tunneling
Increment the last byte:

00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Execute the instruction:

00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Observe its length:

```
00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```
Increment the last byte:

00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling
Execute the instruction:

00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Observe its length:

00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Increment the last byte:

00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Execute the instruction:

00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Observe its length:

Tunneling
Increment the last byte:

00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Execute the instruction:

00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Observe its length:

Tunneling
Increment the last byte:

00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling
Execute the instruction:

```
00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Tunneling
Observe its length:

```
00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```
Increment the last byte:

00 04 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling
When the last byte is FF...

C7 04 05 00 00 00 00 00 00 00 00 FF 00 00 00 00

Tunneling
& … roll over …

Tunneling

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00
... and move to the previous byte

Tunneling

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00
This byte becomes the marker

0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

Tunneling
Increment the marker

Tunneling

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00
Execute the instruction

C7 04 05 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00
Observe its length

C7 04 05 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00

Tunneling
If the length has not changed...

C7 04 05 00 00 00 00 00 00 00 00 01 00 00 00 00 00

Tunneling
Increment the marker

C7 04 05 00 00 00 00 00 00 00 00 02 00 00 00 00 00

Tunneling
And repeat.

Tunneling
Continue the process...

Tunneling

C7 04 05 00 00 00 00 00 00 00 FF 00 00 00 00 00
… moving back on each rollover

Tunneling

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
… moving back on each rollover

C7 04 05 00 00 00 00 00 FF 00 00 00 00 00 00 00

Tunneling
moving back on each rollover

C7 04 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling
Tunneling
& \ldots

Tunneling
& ...  

C7 04 05 00 00 00 FF 00 00 00 00 00 00 00 00 00 00

Tunneling
Tunneling
Tunneling
Tunneling
Tunneling
Tunneling
Tunneling
When you increment a marker...

Tunneling
… execute the instruction …

C7 04 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Tunneling
and the length changes ...

Tunneling

07 04 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
… move the marker to the end of the new instruction …

Tunneling
and resume the process.

Tunneling
Tunneling through the instruction space lets us quickly skip over the bytes that don’t matter, and exhaustively search the bytes that do...
... reducing the search space from $1.3 \times 10^{36}$ instructions to $\sim 100,000,000$ (one day of scanning)

Tunneling
Instructions lengths

Catch: requires knowing the instruction length
Simple approach: trap flag

- Fails to resolve the length of faulting instructions
- Necessary to search privileged instructions:
  - ring 0 only: mov cr0, eax
  - ring -1 only: vmenter
  - ring -2 only: rsm

Instruction lengths
Solution: page fault analysis

Instruction lengths
Choose a candidate instruction
(we don’t know how long this instruction is)

0F 6A 60 6A 79 6D C6 02 6E AA D2 39 0B B7 52

Page fault analysis
Configure two consecutive pages in memory

- The first with read, write, and **execute** permissions
- The second with read, write permissions only

Page fault analysis
Place the candidate instruction in memory
- Place the first byte at the end of the first page
- Place the remaining bytes at the start of the second

Page fault analysis

0F 6A 60 6A 79 6D C6 02 ...
Page fault analysis

Execute (jump to) the instruction.

0F 6A 60 6A 79 6D C6 02 ...
The processor’s instruction decoder checks the first byte of the instruction.

Page fault analysis
If the decoder determines that another byte is necessary, it attempts to fetch it.
This byte is on a non-executable page, so the processor generates a page fault.

Page fault analysis
The #PF exception provides a fault address in the CR2 register.

Page fault analysis
If we receive a #PF, with **CR2** set to the address of the **second** page, we know the instruction continues.

**Page fault analysis**

```asm
0F 6A 60 6A 79 6D C6 02 ...
```
Move the instruction back one byte.

Page fault analysis

OF 6A 60 6A 79 6D C6 02 ...
 Execute the instruction.

Page fault analysis

0F 6A 60 6A 79 6D C6 02 ...

The processor’s instruction decoder checks the first byte of the instruction.

Page fault analysis
If the decoder determines that another byte is necessary, it attempts to fetch it.
Since this byte is in an executable page, decoding continues.

Page fault analysis
If the decoder determines that another byte is necessary, it attempts to fetch it.

Page fault analysis
This byte is on a non-executable page, so the processor generates a page fault.

Page fault analysis
Move the instruction back one byte.

Page fault analysis

```
0F 6A 60 6A 79 6D C6 02 ...
```
Execute the instruction.

Page fault analysis

0F 6A 60  6A 79 6D C6 02 ...

Diagram showing hexadecimal code.
Continue the process while we receive #PF exceptions with CR2 = second page address.

Page fault analysis
Move the instruction back one byte.

Page fault analysis

0F 6A 60 6A 79 6D C6 02 ...

Execute.

Page fault analysis

0F 6A 60 6A 79 6D C6 02 ...
Eventually, the entire instruction will reside in the executable page.

Page fault analysis
The instruction could run.
The instruction could throw a different fault.
The instruction could throw a \#PF, but with a different CR2.

Page fault analysis

\textbf{0F 6A 60 6A} \textbf{79 6D C6 02 ...}
In all cases, we know the instruction has been successfully decoded, so must reside entirely in the executable page.
With this, we know the instruction’s length.

| 0F 6A 60 6A | 79 6D C6 02 ... |

**Page fault analysis**
We now know how many bytes the instruction decoder consumed.

But just because the bytes were *decoded* does not mean the instruction *exists*.

If the instruction does not exist, the processor generates the `#UD` exception after the instruction decode (invalid opcode exception).

Page fault analysis
If we don’t receive a #UD, the instruction exists.

Page fault analysis
Resolves lengths for:

- Successfully executing instructions
- Faulting instructions
- Privileged instructions:
  - ring 0 only: mov cr0, eax
  - ring -1 only: vmenter
  - ring -2 only: rsm

Page fault analysis
The "injector" process performs the page fault analysis and tunneling instruction generation.
We’re fuzzing the same device that we’re running on
How do we make sure we don’t crash?

Surviving
Step 1:
- Limit ourselves to ring 3
- We can still resolve instructions living in deeper rings
- This prevents accidental total system failure (except in the case of serious processor bugs)
Step 2:
- Hook all exceptions the instruction might generate
- In Linux:
  - SIGSEGV
  - SIGILL
  - SIGFPE
  - SIGBUS
  - SIGTRAP
- Process will clean up after itself when possible
Step 3:

- Initialize general purpose registers to 0
- Arbitrary memory write instructions like \( \text{add [eax + 4 * ecx], 0x9102} \) will not hit the injecting process’s address space

Surviving
Step 3 (continued):

- Memory calculations using an offset:
  ```assembly
  add [eax + 4 * ecx + 0xf98102cd6], 0x9102
  ```
  would still result in non-zero accesses

- Could lead to process corruption
  if the offset falls into the injector’s address space

Surviving
Step 3 (continued):

- The tunneling approach ensures offsets are constrained
  - 0x0000002F
  - 0x0000A900
  - 0x00420000
  - 0x1E000000
- The tunneled offsets will not fall into the injector’s address space
- They will seg fault, but seg faults are caught
- The process still won’t corrupt itself

Surviving
We’ve handled faulting instructions
What about non-faulting instructions?
The analysis needs to continue after an instruction executes
Set the trap flag prior to executing the candidate instruction.
On trap, reload the registers to a known state.
With these...
  - Ring 3
  - Exception handling
  - Register initialization
  - Register maintenance
  - Execution trapping

... the injector survives.

Surviving
So we now have a way to search the instructions space.

How do we make sense of the instructions we execute?
The “sifter” process parses the executions from the injector, and pulls out the anomalies.
We need a “ground truth”

Use a disassembler

- It was written based on the documentation
- Capstone
Undocumented instruction:
- Disassembler doesn’t recognize byte sequence and ...
- Instruction generates anything but a #UD

Software bug:
- Disassembler recognizes instruction but ...
- Processor says the length is different

Hardware bug:
- ???
- No consistent heuristic, investigate when something fails

Sifting
sandsifter - demo
sandsifter

shl ebx, 0x6b
(unk)
    and edx, esi
imul edx, dword ptr [rbx], 0x58112d43
movabs dword ptr [0x82d917b0fbb1eb5b], eax
    (unk)
push rsp
    or eax, 0x13753778
fstt
    jbe 0xfffffffffffffb9
l: 2
    jle 0xffffffffffffffdb
    and esi, esp
    and byte ptr [rax], al
push -0x33da2f5b
    in eax, dx
    mov esi, 0xe44908d6
pop rsp
    mov eax, dword ptr [rdi + rax*4 - 0x2f5561f1]
    (unk)
    and dword ptr [rdx], edx

# 2,259,724
39800/s
# 112

dbe1102a3eb94b7a436193c6c73b60be
dbe06ea350976600ebe93210563a5f39b
0f1bd311bb6376398c8cc1ab20ccc2af
0df01de21248565a6838e8f5ce4353f4
dfc37eff85e9ca82c485c523ba4b201e
dbe1f2552633814af7441c7cf3f0dce
dfc0ab37538a7f3035f10e740311891
0f1ae471537e81fe974e61c20ae9c91
0f0d97a9c3f2542c1047a092b1f5b66f
dfc207323fcb7c7e8b88320f3c2587b18

(sandsifter)
VIA Nano U3500@1000MHz
arch: 32 / processor: 0 / vendor: CentaurHauls / family: 6 / model: n/a / stepping: 8 / ucode: n/a

instruction:
0fa7c2

prefixes:  (_)
valids:  (1)
lengths:  (3)
signums:  (5)
signals:  (sigtrap)
sicodes:  (2)

analysis:
capstone:
  (unk)
n/a
ndisasm:
  (unknown)
n/a
objdump:
  (unknown)
n/a

j: down, J: DOWN
k: up, K: UP
l: expand L: all
h: collapse H: all
g: start G: end
{: previous }: next
q: quit and print
We now have a way to systematically scan our processor for secrets and bugs.
I scanned eight systems in my test library.
Results

- Hidden instructions
- Ubiquitous software bugs
- Hypervisor flaws
- Hardware bugs
Hidden instructions
Scanned: Intel Core i7-4650U CPU

Intel hidden instructions
Intel hidden instructions

- 0f0dxx
  - Undocumented for non-/1 reg fields
- 0f18xx, 0f{1a-1f}xx
  - Undocumented until December 2016
- 0fae{e9-ef, f1-f7, f9-ff}
  - Undocumented for non-0 r/m fields until June 2014
Intel hidden instructions

- dbe0, dbe1
- df\{c0-c7\}
- f1
- \{c0-c1\}\{30-37, 70-77, b0-b7, f0-f7\}
- \{d0-d1\}\{30-37, 70-77, b0-b7, f0-f7\}
- \{d2-d3\}\{30-37, 70-77, b0-b7, f0-f7\}
- f6 /1, f7 /1
Scanned: AMD Athlon (Geode NX1500)

AMD hidden instructions
- 0f0f{40-7f}{80-ff}{xx}
  - Undocumented for range of xx
- dbe0, dbe1
- df{c0-c7}

AMD hidden instructions
VIA hidden instructions
- `0f0dxx`
  - Undocumented by Intel for non-`/1` reg fields

- `0f18xx, 0f{1a-1f}xx`
  - Undocumented by Intel until December 2016

- `0fa7{c1-c7}`

- `0fae{e9-ef, f1-f7, f9-ff}`
  - Undocumented by Intel for non-`0 r/m` fields until June 2014

- `dbe0, dbe1`

- `df{c0-c7}`

VIA hidden instructions
What do these *do?*

- Some have been reverse engineered
- Some have no record at all.

Hidden instructions
Software bugs
Issue:

- The sifter is forced to use a disassembler as its “ground truth”
- Every disassembler we tried as the “ground truth” was littered with bugs.

Software bugs
Most bugs only appear in a few tools, and are not especially interesting.

Some bugs appeared in all tools.

These can be used to an attacker’s advantage.

Software bugs
Software bugs

- \texttt{66e9xxxxxxx} (jmp)
- \texttt{66e8xxxxxxx} (call)
In x86_64

Theoretically, a jmp (e9) or call (e8), with a data size override prefix (66)

- Changes operand size from default of 32
  - Does that mean 16 bit or 64 bit?
  - Neither. 66 is ignored by the processor here.

Software bugs
Everyone parses this wrong.

Software bugs
Software bugs (IDA)
Software bugs (VS)
An attacker can use this to mask malicious behavior

- Throw off disassembly and jump targets to cause analysis tools to miss the real behavior
Software bugs (objdump)
66 jmp
- Why does everyone get this wrong?
- AMD: override changes operand to 16 bits, instruction pointer truncated
- Intel: override ignored.

Software bugs
- Issues when we can’t agree on a standard
  - sysret bugs
- Either Intel or AMD is going to be vulnerable when there is a difference
- Impractically complex architecture
  - Tools cannot parse a jump instruction

Software bugs
Hypervisor bugs
In an Azure instance, the trap flag is missed on the cpuid instruction.

Azure hypervisor bugs
Azure hypervisor bugs
Hardware bugs
Hardware bugs are troubling

- A bug in hardware means you now have the same bug in all of your software.
- Difficult to find
- Difficult to fix
Intel hardware bugs
f00f bug on Pentium (anti-climactic)

Intel hardware bugs
Scanned:
- Geode NX1500, C-50

AMD hardware bugs
On several systems, receive a #UD exception prior to complete instruction fetch.

Per AMD specifications, this is incorrect.

#PF during instruction fetch takes priority.

... until ...

AMD hardware bugs
# Simultaneous Interrupt Priorities

<table>
<thead>
<tr>
<th>Interrupt Priority</th>
<th>Interrupt Condition</th>
<th>Interrupt Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>(High) 0</td>
<td>Processor Reset</td>
<td>—</td>
</tr>
<tr>
<td>0</td>
<td>Machine-Check Exception</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>External Processor Initialization (INIT)</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>SMI Interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>External Clock Stop (Stpclk)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Data, and I/O Breakpoint (Debug Register)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Single-Step Execution Instruction Trap (RFLAGS.TF=1)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Non-Maskable Interrupt</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Maskable External Interrupt (INTR)</td>
<td>32–255</td>
</tr>
<tr>
<td>5</td>
<td>Instruction Breakpoint (Debug Register)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Code-Segment-Limit Violation</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>Instruction-Fetch Page Fault</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Invalid Opcode Exception</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Device-Not-Available Exception</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Instruction-Length Violation (&gt; 15 Bytes)</td>
<td>13</td>
</tr>
</tbody>
</table>

**Note:**
This reflects the relative priority for faults encountered when fetching the first byte of an instruction. In the fetching and decoding of subsequent bytes of an instruction, an Invalid Opcode exception may be detected and raised before a fetch-related fault would be seen on a later byte. This behavior is model-dependent.
Transmeta hardware bugs
Instructions: 0f{71,72,73}xxxx
Can receive #MF exception during fetch
Example:
- Pending x87 FPU exception
- `psrad mm4, -0x50 (0f72e4b0)`
- #MF received after 0f72e4 fetched
- Correct behavior: #PF on fetch, last byte is still on invalid page

Transmeta hardware bugs
Found on one processor...

An apparent “halt and catch fire” instruction

- Single malformed instruction in ring 3 locks the processor
- Tested on 2 Windows kernels, 3 Linux kernels
- Kernel debugging, serial I/O, interrupt analysis seem to confirm

Unfortunately, not finished with responsible disclosure

No details available on chip, vendor, or instructions

hardware bugs
ring 3 processor DoS: demo
First such attack found in 20 years (since Pentium f00f)
Significant security concern: processor DoS from unprivileged user
Details (hopefully) released within the next month (stay tuned)
Open sourced:
- The sandsifter scanning tool
- github.com/xoreaxeaxeax/sandsifter

Audit your processor,
break disassemblers/emulators/hypervisors,
halt and catch fire, etc.

Conclusions
I’ve only scanned a few systems
This is a fraction of what I found on mine
Who knows what exists on yours

Conclusions
Check your system
Send us results if you can

Conclusions
Don’t blindly trust the specifications.
Sandsifter lets us introspect the black box at the heart of our systems.

Conclusions
Feedback? Ideas?

domas

@xoreaxeaxeaxeax

xoreaxeaxeaxeax@gmail.com