Malicious Input

Buffer Overflow
Malicious Input

Buffer Overflow
Control Flow Hijack: Always control + computation

shellcode (aka payload) padding &buf

computation + control

Acknowledgement: Some slides from David Brumley, Ed Schwartz, Kevin Snow, and Luci Davi
Control Flow Hijack: Always control + computation

shellcode (aka payload) padding &buf

computation + control

Return-oriented programming (ROP): shellcode without code injection

Acknowledgement: Some slides from David Brumley, Ed Schwartz, Kevin Snow, and Luci Davi
The Evolution of The CFH War

Overflow return address
Execute in the stack
Return-into-libc
Return-oriented program (ROP)


Canaries
Data-execution prevention
Randomize libc address (ASLR)
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
Agenda

- ROP Overview
- Gadgets
- Disassembling code
- Hacking Blind: BROP
Motivation: Return-to-libc Attack

Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will “call” libc function

No injected code!
Motivation: Return-to-libc Attack

Overwrite return address with address of libc function
- setup fake return address and argument(s)
- ret will “call” libc function

No injected code!
Motivation: Return-to-libc Attack

Overwrite return address with address of libc function
  • setup fake return address and argument(s)
  • `ret` will "call" libc function

No injected code!
What if we don’t know the address of system?
What if we don’t know the address of system?

Use existing application logic that does!
Need to find an instruction sequence, aka *gadget*, with esp

```
buf (64 bytes)
argv[1]
argv
return addr
caller's ebp
...%ebp
%esp
```
Need to find an instruction sequence, aka *gadget*, with esp
Scorecard for ret2libc
Scorecard for ret2libc

• No injected code ➔ DEP ineffective

• Requires knowing address of system
Gadget Examples

• Save address of esp
• Execute our own shellcode
• Shacham: gadgets are Turing-complete

Our example shellcode:

```
xor ecx, ecx
mul ecx
push ecx
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
mov al, 0xb
int 0x80
```

The trick is to find the desired instruction sequences, or semantically equivalent ones.
Return Oriented Programming Techniques

Geometry of Flesh on the Bone, Shacham et al, CCS 2007
Daily Blog Tips awarded the

Last week Darren Rowse, from the famous ProBlogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog, Daniel commented that...
Daily Blog Tips awarded the

Last week Darren Rowse, from the famous ProBlogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is renowned for attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog, Daniel commented that...
Daily Blog Tips awarded the

Last week Darren House, the Daily Blog Tips is Ren
from the famous attracting a vast audience follo
Problogger blog, among of bloggers who are imp
announced the winners of looking to improve their
his latest Group Writing project called "Reviews and Predictions". Among The
blogs. When asked about The
the success of his blog that
Daniel commented that rela
ROP Programming: Key Steps

1. Disassemble code
2. Identify *useful* code sequences as gadgets
3. Assemble gadgets into desired shellcode
ROP Overview
ROP Overview

• Idea: We forge shell code out of existing application logic gadgets
ROP Overview

• Idea: We forge shell code out of existing application logic gadgets

• Requirements: vulnerability + gadgets + some unrandomized code
ROP Overview

• Idea: We forge shell code out of existing application logic gadgets

• Requirements:
  vulnerability + gadgets + some unrandomized code

• History:
  – No code randomized: Code injection
  – DEP enabled by default: ROP attacks using libc gadgets publicized ~2007
  – Libc randomized
  – ASLR library load points
  – Today: Windows 7 compiler randomizes text by default, Randomizing text on Linux not straight-forward.
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
There are many \textit{semantically equivalent} ways to achieve the same net shellcode effect
Equivalence

\[ \text{Mem}[v2] = v1 \]

Desired Logic

Stack

...  
\( v_2 \)  
...  
\( v_1 \)  
esp
Equivalence

Mem[v2] = v1

Desired Logic

\[ \text{Stack} \]

\[ \text{esp} \]

a₁: mov eax, [esp]
a₂: mov ebx, [esp+8]
a₃: mov [ebx], eax

Implementation 1
Gadgets

A gadget is any instruction sequence ending with `ret`
Gadgets

Mem[v2] = v1

Desired Logic

Suppose a2 and a3 on stack

Stack

\[
\begin{align*}
  &a_5 \\
  &v_2 \\
  &a_3 \\
  &v_1
\end{align*}
\]

esp

eip

Implementation 2

<table>
<thead>
<tr>
<th>eax</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>a1</td>
</tr>
</tbody>
</table>

\begin{align*}
a_1 & : \text{pop eax;} \\
a_2 & : \text{ret} \\
a_3 & : \text{pop ebx;} \\
a_4 & : \text{ret} \\
a_5 & : \text{mov [ebx], eax}
\end{align*}
Gadgets

Desired Logic

Mem[v2] = v1

Suppose a2 and a3 on stack

Stack

<table>
<thead>
<tr>
<th>esp</th>
<th>v1</th>
<th>a3</th>
<th>v2</th>
<th>a5</th>
</tr>
</thead>
</table>

Implementation 2

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>a1</td>
</tr>
</tbody>
</table>

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
Gadgets

Mem[v2] = v1

Desired Logic

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>a1</td>
</tr>
</tbody>
</table>

Stack

- a5
- v2
- a3
- v1

Implementation 2

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
Gadgets

Mem[v2] = v1

Desired Logic

Stack

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a5</td>
<td>esp</td>
</tr>
<tr>
<td>v2</td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td></td>
</tr>
<tr>
<td>v1</td>
<td></td>
</tr>
</tbody>
</table>

eax | v1
ebx |
eip | a3

Implementation 2

a₁: pop eax;
a₂: ret
a₃: pop ebx;
a₄: ret
a₅: mov [ebx], eax
**Gadgets**

**Desired Logic**

\[ \text{Mem[v2]} = \text{v1} \]

<table>
<thead>
<tr>
<th>Stack</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td></td>
</tr>
<tr>
<td>v2</td>
<td></td>
</tr>
<tr>
<td>a5</td>
<td>esp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eax</th>
<th>\text{v1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>\text{a3}</td>
</tr>
</tbody>
</table>

**Implementation 2**

- \text{a1: pop eax;}
- \text{a2: ret}
- \text{a3: pop ebx;}
- \text{a4: ret}
- \text{a5: mov [ebx], eax}
Gadgets

Desired Logic

Mem[v2] = v1

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a3</td>
</tr>
</tbody>
</table>

Stack

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax

Implementation 2
Gadgets

Desired Logic

Mem[v2] = v1

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a4</td>
</tr>
</tbody>
</table>

Stack

esp

a5
v2
a3
v1

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax

Implementation 2
Gadgets

**Mem[v2] = v1**

Desired Logic

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a5</td>
</tr>
</tbody>
</table>

Stack

- a5
- v2
- a3
- v1

Implementation 2

- a1: pop eax;
- a2: ret
- a3: pop ebx;
- a4: ret
- a5: mov [ebx], eax
Gadgets

Mem[v2] = v1

Desired Logic

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a5</td>
</tr>
</tbody>
</table>

Stack

esp

| a5 |
| v2 |
| a3 |
| v1 |

Implementation 2

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
Equivalence

Mem[v2] = v1

Desired Logic

... ...

-------------

Implementation 1

\[\begin{array}{c}
\text{esp} \\
v_1 \\
a_2 \\
v_2 \\
a_3
\end{array}\]

Implementation 2

\begin{align*}
a_1: & \text{ pop eax; ret} \\
a_2: & \text{ pop ebx; ret} \\
a_3: & \text{ mov [ebx], eax}
\end{align*}
Desired Logic

\[
\text{Mem}[v2] = v1
\]

Semantically equivalent

\[
\begin{align*}
&\text{a}_1: \text{mov eax, [esp]} \leftrightarrow \text{a}_1: \text{pop eax; ret} \\
&\text{a}_2: \text{mov ebx, [esp+8]} \leftrightarrow \text{a}_2: \text{pop ebx; ret} \\
&\text{a}_3: \text{mov [ebx], eax} \leftrightarrow \text{a}_3: \text{mov [ebx], eax}
\end{align*}
\]

Implementation 1

Implementation 2

Stack

<table>
<thead>
<tr>
<th>\text{a}_3</th>
<th>\text{v}_2</th>
<th>\text{a}_2</th>
<th>\text{v}_1</th>
</tr>
</thead>
</table>

"Gadgets"
Return-Oriented Programming (ROP)

Desired Shellcode

1. Find needed instruction gadgets at addresses $a_1$, $a_2$, and $a_3$ in *existing* code
2. Overwrite stack to execute $a_1$, $a_2$, and then $a_3$
Return-Oriented Programming (ROP)

Mem[v2] = v1

Desired Shellcode

a₁: pop eax; ret
a₂: pop ebx; ret
a₃: mov [ebx], eax
Return-Oriented Programming (ROP)

Mem[v2] = v1

Desired Shellcode

\[ \text{argv[1]} = \text{buf} \]

\[
\begin{align*}
\text{a}_1 &: \text{pop eax; ret} \\
\text{a}_2 &: \text{pop ebx; ret} \\
\text{a}_3 &: \text{mov [ebx], eax}
\end{align*}
\]
Return-Oriented Programming (ROP)

Mem[v2] = v1

Desired Shellcode

\[ a_1: \text{pop~eax};\text{ret} \]
\[ a_2: \text{pop~ebx};\text{ret} \]
\[ a_3: \text{mov~[ebx]},\text{eax} \]

Desired store executed!
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
a₁: add eax, 0x80; pop %ebp; ret
a₂: pop %eax; ret

Draw a stack diagram and ROP exploit to pop a value 0xBBBBBBBBBBBB into eax and add 80.

Known Gadgets
Quiz

```c
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

- `a1`: add eax, 0x80; pop %ebp; ret
- `a2`: pop %eax; ret

```
AAAAA ... a3 a2 0xBBBBBBBBBBB a1
```

- **Overwrite buf**
- **Start rop chain**
- **gadget 1 + data**
- **gadget 2**

<table>
<thead>
<tr>
<th>&lt;data for pop ebp&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
</tr>
<tr>
<td>0xBBBB</td>
</tr>
<tr>
<td>a2</td>
</tr>
<tr>
<td>a3</td>
</tr>
<tr>
<td>saved ebp</td>
</tr>
<tr>
<td>buf</td>
</tr>
</tbody>
</table>
Attack Surface: Linux

Unrandomized

Randomized
Attack Surface: Linux

Unrandomized

Program Image

Randomized
Attack Surface: Linux

Unrandomized

Program Image

Randomized

Libc
Attack Surface: Linux

Unrandomized

- Program Image

Randomized

- Libc
- Stack
Attack Surface: Linux

Unrandomized

Randomized

Program Image

Libc

Stack

Heap
Attack Surface: Windows

Unrandomized

Randomized
Attack Surface: Windows

Unrandomized  Randomized

Program Image
Attack Surface: Windows

Unrandomized

Program Image

Libc

Randomized
Attack Surface: Windows

Unrandomized

Randomized

Program Image

Libc

Stack
Attack Surface: Windows

Unrandomized

Randomized

Program Image

Libc

Stack

Heap
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
Recall: Execution Model

- Code
- Stack
- Heap

Process Memory

EIP Processor

Fetch, decode, execute

read and write
Disassembly

user@box:~/l2$ objdump -d ./file
...
00000000 <even_sum>:
0: 55         push   %ebp
1: 89 e5      mov    %esp,%ebp
3: 83 ec 10   sub    $0x10,%esp
6: 8b 45 0c   mov    0xc(%ebp),%eax
9: 03 45 08   add    0x8(%ebp),%eax
C: 03 45 10   add    0x10(%ebp),%eax
F: 89 45 fc   mov    %eax,0xfffffffc(%ebp)
12: 8b 45 fc  mov    0xfffffffc(%ebp),%eax
15: 83 e0 01   and    $0x1,%eax
18: 84 c0      test   %al,%al
1a: 74 03      je     1f <even_sum+0x1f>
1c: ff 45 fc  incl   0xfffffffffc(%ebp)
1f: 8b 45 fc  mov    0xfffffffffc(%ebp),%eax
22: c9         leave
23: c3         ret

Address
Executable instructions
Disassemble
Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Algorithm:
1. Decode Instruction
2. Advance EIP by len
Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Algorithm:
1. Decode Instruction
2. Advance EIP by len

PUSH—Push Word, Doubleword or Quadword Onto the Stack

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF 16</td>
<td>PUSH r/m16</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF 16</td>
<td>PUSH r/m32</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>FF 16</td>
<td>PUSH r/m64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r/m64. Default operand size 64-bits.</td>
</tr>
<tr>
<td>S0+rw</td>
<td>PUSH r16</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r16.</td>
</tr>
<tr>
<td>S0+rd</td>
<td>PUSH r32</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r32.</td>
</tr>
<tr>
<td>S0+rd</td>
<td>PUSH r64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r64. Default operand size 64-bits.</td>
</tr>
</tbody>
</table>
Linear-Sweep Disassembly

Executable Instructions

0x55  0x89  0xe5  0x83  0xec  0x10  ...  0xc9

PUSH—Push Word, Doubleword or Quadword Onto the Stack

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF /6</td>
<td>PUSH r/m16</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m32</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r/m64. Default operand size 64-bits.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r64. Default operand size 64-bits.</td>
</tr>
</tbody>
</table>

Algorithm:
1. Decode Instruction
2. Advance EIP by len

<table>
<thead>
<tr>
<th>Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte register</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>Register</td>
</tr>
<tr>
<td>AL</td>
</tr>
<tr>
<td>CL</td>
</tr>
<tr>
<td>DL</td>
</tr>
<tr>
<td>BPL</td>
</tr>
</tbody>
</table>
Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

PUSH—Push Word, Doubleword or Quadword Onto the Stack

<table>
<thead>
<tr>
<th>Opcode*</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF /6</td>
<td>PUSH r/m16</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r/m16.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m32</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r/m32.</td>
</tr>
<tr>
<td>FF /6</td>
<td>PUSH r/m64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r/m64. Default operand size 64-bits.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r16</td>
<td>Valid</td>
<td>Valid</td>
<td>Push r16.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r32</td>
<td>N.E.</td>
<td>Valid</td>
<td>Push r32.</td>
</tr>
<tr>
<td>50+rd</td>
<td>PUSH r64</td>
<td>Valid</td>
<td>N.E.</td>
<td>Push r64. Default operand size 64-bits.</td>
</tr>
</tbody>
</table>

Algorithm:
1. Decode Instruction
2. Advance EIP by len

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

<table>
<thead>
<tr>
<th>byte register</th>
<th>word register</th>
<th>dword register</th>
<th>quadword register (64-Bit Mode only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>RX.B</td>
<td>Reg Field</td>
<td>Register</td>
</tr>
<tr>
<td>AL</td>
<td>None</td>
<td>0</td>
<td>AX</td>
</tr>
<tr>
<td>CL</td>
<td>None</td>
<td>1</td>
<td>CX</td>
</tr>
<tr>
<td>DL</td>
<td>None</td>
<td>2</td>
<td>DX</td>
</tr>
<tr>
<td>BPL</td>
<td>Yes</td>
<td>5</td>
<td>BP</td>
</tr>
</tbody>
</table>
Linear-Sweep Disassembly

Executable Instructions

0x55  0x89  0xe5  0x83  0xec  0x10  ...  0xc9

Disassembler
EIP

push ebp
Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ...

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Compat/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>88 /r</td>
<td>MOV r/m8,r8</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>REX + 88 /r</td>
<td>MOV r/m8*** r8***</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>89 /r</td>
<td>MOV r/m16,r16</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r16 to r/m16.</td>
</tr>
<tr>
<td>89 /r</td>
<td>MOV r/m32,r32</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r32 to r/m32.</td>
</tr>
</tbody>
</table>

push ebp
## Linear-Sweep Disassembly

### Executable Instructions

```
0x55  0x89  0xe5  0x83  0xec  0x10  ...  0xc9
```

#### Disassembler

**EIP**

---

### Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>00</td>
<td>00 08 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[ECX]</td>
<td>001</td>
<td>01</td>
<td>09 11 19 21 29 31 39</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### MOV—Move

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/ Leg Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>88 i</td>
<td>MOV r/m8,r8</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>88 i</td>
<td>MOV r/m8***'r8***</td>
<td>Valid</td>
<td>N.E.</td>
<td>Move r8 to r/m8.</td>
</tr>
<tr>
<td>89 i</td>
<td>MOV r/m16,r16</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r16 to r/m16.</td>
</tr>
<tr>
<td>89 i</td>
<td>MOV r/m32,r32</td>
<td>Valid</td>
<td>Valid</td>
<td>Move r32 to r/m32.</td>
</tr>
</tbody>
</table>
Linear-Sweep Disassembly

Executable Instructions

0x55  0x89  0xe5  0x83  0xec  0x10  ...  0xc9

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

<table>
<thead>
<tr>
<th>Effective Address</th>
<th>Mod</th>
<th>R/M</th>
<th>Value of ModR/M Byte (in Hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EAX]</td>
<td>00</td>
<td>00</td>
<td>00 0B 08 10 18 20 28 30 38</td>
</tr>
<tr>
<td>[ECX]</td>
<td>001</td>
<td>01</td>
<td>01 09 11 19 21 29 31 39</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

push ebp
mov %esp, %ebp
Linear-Sweep Disassembly

Executable Instructions
0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Algorithm:
1. Decode Instruction
2. Advance EIP by len

push ebp
mov %esp, %ebp
Linear-Sweep Disassembly

Executable Instructions
0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Algorithm:
1. Decode Instruction
2. Advance EIP by len

Disassembler
EIP

Note we don’t follow jumps: we just increment by instruction length

push ebp
mov %esp, %ebp
Disassemble from any address

push ebp  mov %esp, %ebp

0x55  0x89  0xe5  0x83  0xec  0x10  ...  0xc9

Normal Execution
Disassemble from any address

It’s perfectly valid to start disassembling from *any* address.
All byte sequences will have a unique disassembly
Recursive Descent

• Follow jumps and returns instead of linear sweep

• Undecidable: indirect jumps
  – Where does jmp *eax go?
Gadgets, Historically

Mem[v2] = v1

Semantics

a₁: pop eax; ret
...

a₃: mov [ebx], eax
...

a₂: pop ebx; ret

Gadgets
Gadgets, Historically

Mem[v2] = v1

Semantics

a₁: pop eax; ret
...

a₃: mov [ebx], eax
...

a₂: pop ebx; ret

Gadgets

• Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets
Gadgets, Historically

Semantics

\[
\text{Mem}[v2] = v1
\]

\begin{align*}
\text{a}_1 & : \text{pop eax; ret} \\
\text{...} & \\
\text{a}_3 & : \text{mov [ebx], eax} \\
\text{...} & \\
\text{a}_2 & : \text{pop ebx; ret}
\end{align*}

Gadgets

- Shacham et al. manually identified which sequences ending in `ret` in `libc` were useful gadgets.
- Common shellcode was created with these gadgets.
Gadgets, Historically

\[ \text{Mem}[v2] = v1 \]

Semantics

- \( a_1: \) pop eax; ret
- \( \ldots \)
- \( a_3: \) mov [ebx], eax
- \( \ldots \)
- \( a_2: \) pop ebx; ret

Gadgets

- Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets.
- Common shellcode was created with these gadgets.
- Everyone used libc, so gadgets and shellcode universal.
Recap: ROP[Shacham et al.]

1. Disassemble code
2. Identify *useful* code sequences as gadgets *ending in ret*
3. Assemble gadgets into desired shellcode
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
Agenda

ROP Overview

Gadgets

Disassembling code

Hacking Blind: BROP
Blind ROP

• **Hacking Blind**
  
  “It is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash.

• This makes it possible to hack proprietary closed-binary services, or open-source servers manually compiled and installed from source where the binary remains unknown to the attacker”

Agenda

ROP Overview

Gadgets

Disassembling Code

Hacking Blind: BROP
Questions