Q: Exploit Hardening Made Easy


CS 6301-002: Language-based Security
Dr. Kevin Hamlen
Attacker’s Dilemma

• Problem Scenario
  – Attack target is a server running some known native code software (e.g., Apache web server).
  – Attacker knows exact software version, but has no physical access or remote privileges.
  – Attacker wishes to “take control” of process (e.g., make it divulge or delete private files).

• Significant assumption: Attacker knows a vulnerability (e.g., buffer overflow bug).
  – Defender doesn’t know it (vulnerability is zero-day).

• How can the attacker leverage this vulnerability to do more than just crash the process?
Anatomy of a Software Hack

- Usually two parts
  - “Exploit” – Maneuver process into executing bug
    - Example: Provide a long input string to overflow the buffer.
    - Let’s assume we already know how to do that part.
  - “Payload” – Leverage bug to convince process to execute attacker-supplied code

- Three kinds of payloads (in order of increasing sophistication):
  - direct code injection
  - jump-to-libc
  - return-oriented programming (ROP)
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

---

```
lea eax,[ebp-48h]
push eax
call <system>
.data "erase"
.data "*.*"
.data "aaaaa..."
.data "aaaa"
<addr of buf>
```

---

- Top of stack (lower addresses)
  - Arguments
  - Local variables
  - Saved registers

- Bottom of stack (higher addresses)
  - Arguments
  - Local variables
  - Saved registers
Code-injection Example

void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    ...
    return;
}
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ... return;
}
```

8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>

lea eax,[ebp-48h]
push eax
call <system>
.data “erase ”
.data “*.* ”
.data “aaaaa...”
.data “aaaa”
<addr of buf>

bottom of stack (higher addresses)
taget addresses

erase *.* aaaaaaaaa
aaaaaaaaaaaaaaaaaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)

```
# Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    ...
    return;
}
```

**Assembly Code:**
```
8D 45 B8  lea eax,[ebp-48h]
50          push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20.data “erase”
2A 2E 2A 20 .data “*.*”
61 (x24)    .data “aaaaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

**Stack Overview:**
- **Top of stack (lower addresses):**
  - `lea eax,[ebp-48h]`
  - `push eax`
  - `call <system>`
  - `.data “erase”`
  - `.data “*.*”`
  - `.data “aaaaaaa...”`
  - `.data “aaaa”`
  - `<addr of buf>`
- **Bottom of stack (higher addresses):**
  - `aaa`
  - `<addr of buf>`
  - `argv (4 bytes)`
  - `argc (4 bytes)`
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

Assembly Code:

```
8D 45 B8  lea eax,[ebp-48h]
50          push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data "erase"
2A 2E 2A 20 .data ".*.*"
61 (x24) .data "aaaaa..."
61 61 61 61 .data "aaaa"
30 FB 1F 00 <addr of buf>
```

Stack Diagram:

- **Top of stack (lower addresses):**
  - lea eax,[ebp-48h]
  - push eax
  - call <system>

- **Bottom of stack (higher addresses):**
  - argv (4 bytes)
  - <addr of buf>
  - <addr of "erase *.* ...">
Code-injection Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

8D 45 B8 lea eax,[ebp-48h]
50 push eax
FF 15 BC 82 2F 01 call <system>
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*.* ”
61 (x24) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>

lea eax,[ebp-48h]
push eax
call <system>
.data “erase ”
.data “*.* ”
.data “aaaaa...”
.data “aaaa”
<addr of buf>

top of stack (lower addresses)

bottom of stack (higher addresses)
Defense: \(\mathbb{W} \oplus \mathbb{X}\) Pages

- **Data Execution Prevention (DEP)**
  - disallow writable & executable permission on any one page of process memory
  - stack is writable but non-executable by default
  - now default on most Windows & Linux systems

- **Counter-attack**
  - don’t insert any code onto the stack
  - jump *directly to existing dangerous code*
    - usually library code, since there are many dangerous things there, and libraries are common to many applications
  - called “jump-to-libc”
Return-to-libc Example

```c
void main(int argc, char *argv[])
{
  char buf[64];
  strcpy(buf,argv[1]);
  ...
  return;
}
```

65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “*. *”
61 (x58) .data “aaaa…”
BC 82 2F 01 .data <system>
61 (x8) .data “aaaa…”
30 FB 1F 00 .data <buf>

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf (64 bytes)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bottom of stack (higher addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>saved EBP (4 bytes)</td>
</tr>
<tr>
<td>saved EIP (4 bytes)</td>
</tr>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
</tbody>
</table>
Return-to-libc Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

```
65 72 61 73 65 20 .data "erase"
2A 2E 2A 20 .data "*.*"
61 (x58) .data "aaaa..."
BC 82 2F 01 .data <system>
61 (x8) .data "aaaa..."
30 FB 1F 00 .data <buf>
```

```
top of stack (lower addresses)

erase *.*

aaaaaaa...
```

```
   aaaa
addr of <system>
   aaaa
   aaaa
addr of <buf>
```
Return-to-libc Example

```c
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

top of stack (lower addresses)

```
data “erase”
data “.*”
data “aaaa…”
data <system>
data “aaaa…”
data <buf>
```

```
65 72 61 73 65 20 .data “erase”
2A 2E 2A 20 .data “.*”
61 (x58) .data “aaaa…”
BC 82 2F 01 .data <system>
61 (x8) .data “aaaa…”
30 FB 1F 00 .data <buf>
```
Return-to-libc Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase <em>.</em></td>
</tr>
<tr>
<td>aaaaaaaaaa...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>addr of &lt;system&gt;</th>
<th>aaaa</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr of &lt;buf&gt;</td>
<td>aaaa</td>
</tr>
<tr>
<td></td>
<td>aaaa</td>
</tr>
</tbody>
</table>

Return-to-libc Example

```cpp
void libc::system(char *cmd) {
    // <passes cmd to the shell!>
}
```

```
.data "erase "
.data ".*.*"
.data "aaaa..."
.data <system>
.数据 <buf>
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 72 61 73 65 20</td>
<td>.data &quot;erase &quot;</td>
</tr>
<tr>
<td>2A 2E 2A 20</td>
<td>.data &quot;.<em>.</em>&quot;</td>
</tr>
<tr>
<td>61 (x58)</td>
<td>.data &quot;aaaa...&quot;</td>
</tr>
<tr>
<td>BC 82 2F 01</td>
<td>.data &lt;system&gt;</td>
</tr>
<tr>
<td>61 (x8)</td>
<td>.data &quot;aaaa...&quot;</td>
</tr>
<tr>
<td>30 FB 1F 00</td>
<td>.data &lt;buf&gt;</td>
</tr>
</tbody>
</table>
```

- `erase *.*
- `aaaaaaa...`
Defense: Hide the Libraries

• **Address Space Layout Randomization (ASLR)**
  – Loader chooses starting address of each library *at load-time* (not compile-time)
    • Libraries already compiled with this capability, so that loader can avoid address space conflicts
    • Note that application main modules do NOT typically have this capability!
  – Tweak the loader to choose the address semi-randomly
  – Result: Attacker cannot reliably predict where libraries are, so cannot reliably jump to any particular code!

• **Counter-attack: Return-Oriented Programming**
  – Payload jumps to main module code instead of libraries.
  – Challenge: Far less dangerous code there (typically).
  – Can the attacker really do much damage?
Return-Oriented Programming

• Key insight: Exploit the “ret” instruction
  – Semantics of ret: Pop the address atop the stack and jump there.
  – Attacker controls the stack...
  – So attacker can control where ALL ret instructions jump henceforth!

• Can string together ret-ending code fragments already present in the main module to implement an attack payload!
ROP Example

void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
ROP Example

```c
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
```

top of stack (lower addresses)
- erase *
- aaaaaa...

```
61 72 61 73 65 20  .data “erase”
2A 2E 2A 20         .data “.*.*”
61 (x58)            .data “aaaa...”
BC 82 2F 04         .data <addr1>
61 61 61 61         .data “aaaa”
82 8C 2E 04         .data <addr2>
82 8C 2E 04         .data <addr2>
7F 22 30 04         .data <addr3>
```
ROP Example

- `init_display`: ...
  - `<... 1024 bytes ...>`
- `system`: ...
- `addr2`: `add eax, 512`
  - `ret`
- `addr1`: `mov eax, [init_display]`
  - `call eax`
  - `pop ebx`
  - `ret`
- `addr3`: `call eax`
  - `ret`

```
<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase <em>.</em></td>
</tr>
<tr>
<td>aaaaaaaa...</td>
</tr>
</tbody>
</table>
```
```
| <addr1>                       |
| aaaa                         |
| <addr2>                       |
| <addr2>                       |
| <addr3>                       |
```
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret

addr1: mov eax, [init_display]
call eax
pop ebx
ret

addr3: call eax
ret

eax = init_display

top of stack (lower addresses)

erase *.*
aaaaaaaa...

aaaa
<addr1>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...

< ... 1024 bytes ... >

system: ...

addr2:
  add eax, 512
  ret
  ...

addr1:
  mov eax, [init_display]
  call eax
  pop ebx
  ret
  ...

addr3:
  call eax
  ret

eax = init_display

Erasing addresses:
- *.*
- aaaaaaa...

Top of stack (lower addresses):
- aaaa
- <addr1+5>
- aaaa
- <addr2>
- <addr2>
- <addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr1: mov eax, [init_display]
call eax
pop ebx
ret

addr2: add eax, 512
ret

addr3: call eax
ret

top of stack (lower addresses)
erase *.*
aaaaaaaa...

...
ROP Example

**init_display**: ...
< ... 1024 bytes ... >

**system**: ...

**addr2**: add eax, 512
ret
...

**addr1**: mov eax, [init_display]
call eax
pop ebx
ret
...

**addr3**: call eax
ret

---

eax = init_display

---

top of stack (lower addresses)
erase *.*

aaaaaaa...

aaaa

<addr1+5>

aaaa

<addr2>

<addr2>

<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
pop ebx
ret
...
addr3: call eax
ret
eax = init_display

top of stack (lower addresses)
erase *.*
aaaaaaa...

aaaa
<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
  < ... 1024 bytes ... >
system: ...

addr2: add eax, 512
  ret
  ...
addr1: mov eax, [init_display]
  call eax
  pop ebx
  ret
  ...
addr3: call eax
  ret

eax = init_display+512

top of stack (lower addresses)
  erase *.*
  aaaaaaaa...

  aaaa
  <addr1+5>
  aaaa
  <addr2>
  <addr2>
  <addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
pop ebx
ret
...
addr3: call eax
ret

eax = init_display+512

top of stack (lower addresses)
erase *.*
aaaaaa...

<addr1+5>
aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >

system: ...

addr2: add eax, 512
ret
...

addr1: mov eax, [init_display]
call eax
call eax
pop ebx
ret
...

addr3: call eax
ret

eax = init_display+1024 = system !!!

top of stack (lower addresses)
erase *.*

aaaaaaa...

aaaa
<addr1+5>

aaaa
<addr2>
<addr2>
<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >

system: ...

addr2: add eax, 512
ret
...

addr1: mov eax, [init_display]
call eax
pop ebx
ret
...

addr3: call eax
ret

eax = init_display+1024 = system !!!

top of stack (lower addresses)

erase *.*

aaaaaaa...

aaaa

<addr1+5>

aaaa

<addr2>

<addr2>

<addr3>
ROP Example

init_display: ...
< ... 1024 bytes ... >
system: ...

addr2: add eax, 512
ret
...
addr1: mov eax, [init_display]
call eax
call eax
pop ebx
ret
...
addr3: call eax
ret

eax = init_display+1024 = system !!!
top of stack (lower addresses)
erase *.*
aaaaaaa...

<addr2>
<addr2>
<addr3>
ROP Attack Surface

• Gadgets: Every \texttt{ret}-ending byte sequence at a known location is available to attacker
  – Gadgets need not be intended, reachable code! Any bytes will do!
  – Can string gadgets together in any sequence
  – Can encode loops (because gadgets can push new addresses)

• Research questions:
  – What payloads are possible from gadget-sequencing?
  – Given a victim program and desired payload, is there a way to systematically discover a gadget-implementation?
Q: An ROP Payload Compiler

Figure 2: An overview of Q’s design.
Q Stages

• Gadget Discovery
  – find gadgets of various “types” in victim program

• Gadget Arrangement
  – infer general gadget sequences that suffice to implement payload
  – not all inferred sequences may be present in victim

• Gadget Assignment
  – match discovered gadgets to inferred arrangements

• Payload Printing
  – output a complete, working assignment
  – usable as malicious input to victim program
### Gadget “Types”

<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Parameters</th>
<th>Semantic Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoOpG</td>
<td>—</td>
<td>—</td>
<td>Does not change memory or registers</td>
</tr>
<tr>
<td>JumpG</td>
<td>AddrReg</td>
<td>Offset</td>
<td>EIP ← AddrReg + Offset</td>
</tr>
<tr>
<td>MoveRegG</td>
<td>InReg, OutReg</td>
<td>—</td>
<td>OutReg ← InReg</td>
</tr>
<tr>
<td>LoadConstG</td>
<td>OutReg, Value</td>
<td>—</td>
<td>OutReg ← Value</td>
</tr>
<tr>
<td>ArithmeticG</td>
<td>InReg1, InReg2, OutReg</td>
<td>◊b</td>
<td>OutReg ← InReg1 ◊b InReg2</td>
</tr>
<tr>
<td>LoadMemG</td>
<td>AddrReg, OutReg</td>
<td># Bytes, Offset</td>
<td>OutReg ← M[AddrReg + Offset]</td>
</tr>
<tr>
<td>StoreMemG</td>
<td>AddrReg, InReg</td>
<td># Bytes, Offset</td>
<td>M[AddrReg + Offset] ← InReg</td>
</tr>
<tr>
<td>ArithmeticLoadG</td>
<td>OutReg, AddrReg</td>
<td># Bytes, Offset, ◊b</td>
<td>OutReg ◊b ← M[AddrReg + Offset]</td>
</tr>
<tr>
<td>ArithmeticStoreG</td>
<td>InReg, AddrReg</td>
<td># Bytes, Offset, ◊b</td>
<td>M[AddrReg + Offset] ◊b ← InReg</td>
</tr>
</tbody>
</table>

- **Challenge:** Given an arbitrary gadget, how to infer its “type” from the table above?
- **Open Research Question:** Is there a better list of “types”? Why just these “types”??
Weakest Precondition

• Hoare Logic:
  – Notation “[A]C[B]” means “If the program state satisfies A, then code C eventually terminates in a program state satisfying B.
  – Example: [x=3 ∧ y=1] x:=x+y [x=4 ∧ y=1]
  – Example: [x=y] x:=x+y [x=2y]
  – Example: [true] x:=3 [x=3]
  – A = “precondition” and B = “postcondition”

• Weakest Precondition [Dijkstra, CACM’75]
  – For any C and B, there are many A satisfying [A]C[B].
  – Weakest possible precondition is “true” (no assumptions)
WP and Gadget Discovery

• Weakest Precondition Algorithm
  – known, easy algorithm for non-looping instructions
  – Example: [...?...] mov r1, r2 [r1=7]
    • A = “r2=7”
  – Generalized: [...?...] mov r1, r2 [B]
    • A = substitute “r2” for all “r1” in B

• Each gadget “type” is really a post-condition
  – MovRegG: r1=r2
  – [...?...] mov r1, r2 [r1=r2]
    • A = “r2=r2” = true

• Strategy: Gadget C has type B if WP(C,B)=true
More Nifty Science in Q

• Gadget arrangement based on every-munch (a take-all version of maximal munch)
• Various tricky register allocation problems
  – register clobbering avoidance
  – register matching
• Basically a full compiler for a very weird instruction set that it has to learn each time!
• With just 20KB of code to mine, Q is 80% successful at finding ROP payloads
• Others have found that at least 33% of all binaries contain Turing-complete gadget sets!
Next Time

• Some embarrassing failures of diversity- and obfuscation-based defenses