On The Effectiveness of Address-Space Randomization

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh
Stanford University
CCS 2004
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

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

```assembly
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>
```

- **buf**: 64 bytes
- **argc**: 4 bytes
- **argv**: 4 bytes
- **(saved EBP)**: 4 bytes
- **saved EIP**: 4 bytes
- **argc**: 4 bytes
- **bottom of stack (higher addresses)**
- **top of stack (lower addresses)**

---

The diagram illustrates the code-injection example with assembly code and stack layout.
### Code-injection Example

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

---

<table>
<thead>
<tr>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8D 45 B8</td>
<td>lea eax,[ebp-48h]</td>
</tr>
<tr>
<td>50</td>
<td>push eax</td>
</tr>
<tr>
<td>FF 15 BC 82 2F 01</td>
<td>call &lt;system&gt;</td>
</tr>
<tr>
<td>65 72 61 73 65 20</td>
<td>.data “erase”</td>
</tr>
<tr>
<td>2A 2E 2A 20</td>
<td>.data “<em>.</em>”</td>
</tr>
<tr>
<td>61 (x24)</td>
<td>.data “aaaaa...”</td>
</tr>
<tr>
<td>61 61 61 61</td>
<td>.data “aaaa”</td>
</tr>
<tr>
<td>30 FB 1F 00</td>
<td>&lt;addr of buf&gt;</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Stack Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>top of stack</td>
<td>lea eax,[ebp-48h]</td>
</tr>
<tr>
<td></td>
<td>push eax</td>
</tr>
<tr>
<td></td>
<td>call &lt;system&gt;</td>
</tr>
<tr>
<td></td>
<td>erase <em>.</em> aaaaaaaa</td>
</tr>
<tr>
<td></td>
<td>aaaaaaaaaaaaaaaaaaaa</td>
</tr>
<tr>
<td></td>
<td>aaaa</td>
</tr>
<tr>
<td></td>
<td>&lt;addr of buf&gt;</td>
</tr>
<tr>
<td>bottom of stack</td>
<td>argc (4 bytes)</td>
</tr>
<tr>
<td></td>
<td>argv (4 bytes)</td>
</tr>
</tbody>
</table>

---

**Diagram:**
- The diagram illustrates the process of code injection where the input is taken from the command-line argument and concatenated with automated code to overwrite and execute harmful code on the system.
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>`

Stack Addresses:
- `argc (4 bytes)`
- `argv (4 bytes)`
- `bottom of stack (higher addresses)`
- `top of stack (lower addresses)`

Code:
- `lea eax,[ebp-48h]`
- `push eax`
- `call <system>`
- `erase *. * aaaaaaaaaa`
- `aaaaaaaaaaaaaaaa`
Code-injection Example

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

![Assembly code and stack diagram]
# 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
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 Layout:**
- **Top of stack (lower addresses):**
  - `lea eax,[ebp-48h]`
  - `push eax`
  - `call <system>`
  - `erase *.* aaaaaaaa aaaaaaaaaaaaaaaaa`

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

lea eax,[ebp-48h]
push eax
call <system>
 erase *.* aaaaaaaa
 aaaaaaaaaaaaaaaaa
<addr of buf>

argv (4 bytes)
<addr of buf>
<addr of "erase *.* ..."]>
bottom of stack (higher addresses)
top of stack (lower addresses)
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;
}
```

```
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)**

<table>
<thead>
<tr>
<th><code>buf</code> (64 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>saved EBP (4 bytes)</code></td>
</tr>
<tr>
<td><code>saved EIP (4 bytes)</code></td>
</tr>
<tr>
<td><code>argv (4 bytes)</code></td>
</tr>
<tr>
<td><code>argc (4 bytes)</code></td>
</tr>
</tbody>
</table>

**bottom of stack (higher addresses)**
Return-to-libc Example

```c
top of stack (lower addresses)
 erase *.*
  aaaaaaaaa...

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>
```

```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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</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 “<em>.</em>”</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):
  - erase *.*
  - aaaaaaa...

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

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

top of stack (lower addresses)

<table>
<thead>
<tr>
<th>erase <em>.</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaaaaaaa...</td>
</tr>
</tbody>
</table>

| aaaaa                      |
| addr of <system>           |
| aaaaa                      |
| 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><strong>buf</strong> (64 bytes)</td>
<td>erase <em>.</em></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><strong>bottom of stack (higher addresses)</strong></td>
<td>unused retaddr for system call</td>
</tr>
<tr>
<td></td>
<td>pointer to buf</td>
</tr>
<tr>
<td></td>
<td><strong>bottom of stack (higher addresses)</strong></td>
</tr>
</tbody>
</table>