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

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

#### Assembly Code

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

#### Stack Layout

<table>
<thead>
<tr>
<th>Address</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 &quot;erase&quot;</td>
</tr>
<tr>
<td>2A 2E 2A 20</td>
<td>.data &quot;<em>.</em>&quot;</td>
</tr>
<tr>
<td>61 (x24)</td>
<td>.data &quot;aaaaa...&quot;</td>
</tr>
<tr>
<td>61 61 61 61</td>
<td>.data &quot;aaaa&quot;</td>
</tr>
<tr>
<td>30 FB 1F 00</td>
<td>&lt;addr of buf&gt;</td>
</tr>
</tbody>
</table>

- Top of stack (lower addresses)
- Buf (64 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 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 “aaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 00 <addr of buf>
```

---

**Diagram:**

- **Top of stack (lower addresses):**
  - `lea eax,[ebp-48h]`
  - `push eax`
  - `call <system>`
  - `.data “erase”`
  - `.data “*..*”`
  - `.data “aaaa...”`
  - `.data “aaaa”`
  - `<addr of buf>`

- **Bottom of stack (higher addresses):**
  - `argv (4 bytes)`
  - `argc (4 bytes)`
  - `<addr of buf>`

---

**Description:**

The image illustrates a code-injection example in both C code and assembly. The C code defines a `main` function that takes an integer `argc` and a pointer to a string `argv[]`. Inside the function, a buffer `buf` is created and populated with a string copied from the first argument `argv[1]`. Then, the `leas` instruction is used to load the address of `buf` into `eax`, followed by a `push` and a `call` to a function named `<system>`. The assembly code shows the corresponding instructions for these actions, including the use of `.data` directives to define strings and the use of `lea`, `push`, and `call` instructions in assembly language.
Code-injection Example

```c
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 “aaaaaaa...”
61 61 61 61         .data “aaaa”
30 FB 1F 00         <addr of buf>

top of stack (lower addresses)

bottom of stack (higher addresses)

lea eax,[ebp-48h]
push eax
call <system>
f9 15 bc 82 2f 01
8d 45 b8
lea eax,[ebp-48h]
push eax
```

erase *.* aaaaaaaa
aaaaaaaaaaaaaaaa
Code-injection Example

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

Hex Code:

- `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>`

Assembly:

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

Stack Layout:

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

- **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;
}
```

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

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

Data:
- `.data “erase”`
- `.data “*. *”`
- `.data “aaaaa...”`
- `.data “aaaa”`
- `<addr of buf>`

**Stack Diagram:**

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

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

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

**Stack Layout**

- **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>
```

- `erase *.*
  aaaaaaa...`
- `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;
}
```
Return-to-libc Example

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

```
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 *.*
aaaaaaa...
```

```
       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 ret</td>
</tr>
<tr>
<td>bottom of stack (higher addresses)</td>
<td>address of system</td>
</tr>
<tr>
<td></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>
Where’s the FEEB?
The Effectiveness of Instruction Set Randomization

Ana Nora Sovarel, David Evans, Nathanael Paul
University of Virginia
USENIX 2005
Instruction Set Randomization

- **Idea:** Randomize the opcode encodings
  - Secure CPU has privileged 8-bit KEY register
  - CPU xor’s each fetched instruction byte with KEY before interpreting (decrypting it)
  - OS xor’s entire program text with KEY at load-time (encrypting it in memory)

- **Better implementation:**
  - Key is a length-n byte sequence
  - CPU xor’s code at address i with KEY[i mod n]
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ... return 0;
}

8D 45 B8 <random instructions> 50 <random instructions>
FF 15 BC 82 2F 01 <random instructions> 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>

top of stack (lower addresses)
<random instructions>
<random instructions>
<random instructions>
<random instructions>
erase *.* aaaaaaaa aaaaaaaaaaaaaaaaaaaaa

bottom of stack (higher addresses)
aaaa
<addr of buf>
argv (4 bytes)
argc (4 bytes)
Attacking ISR

• Goal: Discover the KEY (or at least some of it)
• Four-phase attack:
  – Phase 1: discover 1 or 2 bytes of the KEY
  – Phase 2: discover 4 bytes of the KEY
  – Phase 3: discover 100 bytes of the KEY
  – Phase 4: inject full-sized malicious payloads
# Phase 1: Return-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

---

**Stack Layout**

- **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)**

**Hex Dump**

- `XX`  `ret?`
- `61 (x63)`  `.data “aaaaa...”`
- `61 61 61 61`  `.data “aaaa”`
- `30 FB 1F 10`  `<addr of buf>`
- `61 (x8)`  `.data “aaaaaaaa”`
- `03 14 DF 01`  `<original return addr>`
Phase 1: Return-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
Phase 1: Return-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

- `ret?` at the top of stack
- `.data "aaaaa..."`
- `.data "aaaaa"`
- `<addr of buf>`
- `<original return addr>`
Phase 1: Return-attack

```
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phase 1: Return-attack

int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}

XX ret?
61 (x63) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>
61 (x8) .data “aaaaaaaaa”
03 14 DF 01 <original return addr>
Phase 1: Jump-attack

```c
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

loop: jump loop?

```
data "aaaaa..."
data "aaaa"
<addr of buf>
```

XX XX
61 (x62)
61 61 61 61
30 FB 1F 10

**top of stack (lower addresses)**

- saved EBP (4 bytes)
- saved EIP (4 bytes)
- argv (4 bytes)
- argc (4 bytes)

**buf (64 bytes)**

**bottom of stack (higher addresses)**
### Phase 1: Jump-attack

```c
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

---

<table>
<thead>
<tr>
<th>XX XX</th>
<th>loop: jump loop?</th>
</tr>
</thead>
<tbody>
<tr>
<td>61 (x62)</td>
<td>.data “aaaaa…”</td>
</tr>
<tr>
<td>61 61 61 61</td>
<td>.data “aaaa”</td>
</tr>
<tr>
<td>30 FB 1F 10</td>
<td>&lt;addr of buf&gt;</td>
</tr>
</tbody>
</table>

---

```
loop: jump loop?

aaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaa
aaaaa
<addr of buf>
```

---

<table>
<thead>
<tr>
<th>argv (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc (4 bytes)</td>
</tr>
</tbody>
</table>

---

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

---

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

---

```
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phase 1: Jump-attack

int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
Phase 1: Jump-attack

```
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phase 2: Jump-attack

```
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf,argv[1]);
    ...
    return 0;
}
```
Phase 3: Jump-attack

```c
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

EB 03 14 DF XX  jump <original ret addr?>
61 (x59)  .data “aaaaa…”
61 61 61 61 .data “aaaa”
30 FB 1F 10  <addr of buf>
Phase 3: Jump-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```

EB 03 14 DF XX jump <original ret addr?>
61 (x59) .data “aaaaa…”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>
Phase 3: Jump-attack

```c
int main(int argc, char *argv[])
{
    char buf[64];
    strcpy(buf, argv[1]);
    ...return 0;
}
```

EB 03 14 DF XX jump <original ret addr?>
61 (x59) .data “aaaaa...”
61 61 61 61 .data “aaaa”
30 FB 1F 10 <addr of buf>

top of stack (lower addresses)

jump <original ret addr?>
aaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaa
aaaaaaaaaaaa

<addr of buf>

bottom of stack (higher addresses)

```
argv (4 bytes)
argc (4 bytes)
```
Phase 3: Jump-attack

```
int main(int argc, char *argv[]) {
    char buf[64];
    strcpy(buf, argv[1]);
    ...
    return 0;
}
```
Phases 4: Full-size Payloads

• Learn ~100 bytes of KEY using Phases 1-3
• Goal: Construct a payload such that...
  – execution of payload never steps IP outside the 100-byte window of known KEY’s
  – payload can be much larger than 100 bytes
• Solution: inject a virtual machine!
  – main engine of VM confined to 100-byte window
  – VM copies small chunks of payload into window
  – copying process encrypts using known KEY bytes
  – chunk returns back to main engine when next chunk required
Phase 4: MicroVM

**start:**
- save worm address in ebp
- move stack frame pointer
- WormIP = 0
- copy (and encrypt) worm code
- update WormIP
- save VM registers
- load worm registers

**read_more_worm:**
- 22-byte worm execution buffer
- save worm registers
- load VM registers
- jmp read_more_worm

- worm code

- other worm data
Technical Issues

• False positives
  – probabilistic analysis and mitigation strategies
  – (see Section 3 of paper)

• Payloads that contain null bytes
  – compute them dynamically (e.g., “xor eax,eax” instead of “mov eax,0x00000000”)

• ISR’s that re-randomize after crashes
  – only an issue when children crash parent process
  – questionable ISR design choice
  – no easy workaround suggested, though…
Experimental Results

• Jump-attack
  – cracked 100-byte key in ~6 min. average
  – success rate: 95-100%
  – ~9 infinite loops on average
Improving ISR

• Larger instruction encodings
  – RISC: all instructions 32-bits long

• Better encryption
  – AES instead of XOR
  – (too expensive to be practical)

• Non-uniform remapping of instructions
  – introduce P[255], a random permutation of 0..255
  – to decrypt byte b at address i, compute (P[b] xor KEY[i])
  – encryption uses inverse P table
  – Why does this defeat the attack?
References
