CSE543 - Introduction to Computer and Network Security

Module: Return-oriented Programming

Professor Trent Jaeger
Anatomy of Control-Flow Exploits
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
• **First** -- attacker gets control of program flow (return address, function pointer)
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
• **First** -- attacker gets control of program flow (return address, function pointer)
  ▸ Stack (buffer), heap, format string vulnerability, …
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
• **First** -- attacker gets control of program flow (return address, function pointer)
  ▸ Stack (buffer), heap, format string vulnerability, …
• **Second** -- attacker uses control of program flow to launch attacks
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
  • **First** -- attacker gets control of program flow (return address, function pointer)
    ▸ Stack (buffer), heap, format string vulnerability, …
  • **Second** -- attacker uses control of program flow to launch attacks
    ▸ E.g., Code injection
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
  • **First** -- attacker gets control of program flow (return address, function pointer)
    ‣ Stack (buffer), heap, format string vulnerability, …
  • **Second** -- attacker uses control of program flow to launch attacks
    ‣ E.g., Code injection
      • Adversary injects malcode into victim
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
  • First -- attacker gets control of program flow (return address, function pointer)
    ‣ Stack (buffer), heap, format string vulnerability, …
  • Second -- attacker uses control of program flow to launch attacks
    ‣ E.g., Code injection
      • Adversary injects malcode into victim
      • E.g., onto stack or into other data region
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
• **First** -- attacker gets control of program flow (return address, function pointer)
  ▸ Stack (buffer), heap, format string vulnerability, ...
• **Second** -- attacker uses control of program flow to launch attacks
  ▸ E.g., Code injection
    • Adversary injects malcode into victim
    • E.g., onto stack or into other data region
Anatomy of Control-Flow Exploits

• Two steps in control-flow exploitation
  
  • **First** -- attacker gets control of program flow (return address, function pointer)
    ‣ Stack (buffer), heap, format string vulnerability, …
  
  • **Second** -- attacker uses control of program flow to launch attacks
    ‣ E.g., Code injection
      • Adversary injects malcode into victim
      • E.g., onto stack or into other data region

  ‣ How is code injection done?
Code Injection
Code Injection

• Advantage
Code Injection

• Advantage
  • Adversary can install any code they want
Code Injection

• Advantage
  • Adversary can install any code they want
    • What code do adversaries want?
Code Injection

• Advantage
  • Adversary can install any code they want
    • What code do adversaries want?

  ▸ Defenses
Code Injection

• Advantage
  • Adversary can install any code they want
    • What code do adversaries want?
  ▸ Defenses
    • NX bit - set memory as non-executable (stack)
Code Injection

• Advantage
  • Adversary can install any code they want
    • What code do adversaries want?
  ‣ Defenses
  • **NX bit** - set memory as non-executable (stack)
  • **W (xor) X** - set memory as either writeable or executable, but not both
Code Injection

• Advantage
  • Adversary can install any code they want
    • What code do adversaries want?
  ‣ Defenses
    • NX bit - set memory as non-executable (stack)
    • W (xor) X - set memory as either writeable or executable, but not both
Code Injection

• Advantage
  • Adversary can install any code they want
  • What code do adversaries want?

› Defenses
  • **NX bit** - set memory as non-executable (stack)
  • **W (xor) X** - set memory as either writeable or executable, but not both

• What can adversary do to circumvent these defenses and still execute useful code (for them)?
Return-to-libc Attacks
Return-to-libc Attacks

• Method
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, …
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, ...

• Advantage
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, …

• Advantage
  • Get useful function without code injection
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, …

• Advantage
  • Get useful function without code injection

• Defenses
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, …

• Advantage
  • Get useful function without code injection

• Defenses
  • Remove unwanted library functions
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, …

• Advantage
  • Get useful function without code injection

• Defenses
  • Remove unwanted library functions

• How to overcome this defense???
Return-to-libc Attacks

• Method
  • Overwrite target of indirect call/jmp target to a library routine (e.g., system)
    • Return address, function pointer, ...

• Advantage
  • Get useful function without code injection

• Defenses
  • Remove unwanted library functions

• How to overcome this defense???
  • Topic of today’s lecture
Return-Oriented Programming

• Arbitrary exploitation **without code injection or whole-function reuse** (return-to-libc)
Return-Oriented Programming

Bad code versus bad behavior

“Bad” behavior

“Good” behavior

Attacker code

Application code

Problem: this implication is false!
any sufficiently large program codebase

arbitrary attacker computation and behavior, without code injection

(in the absence of control-flow integrity)
Return-to-libc

- Divert control flow of exploited program into libc code
  - system(), printf(),
- No code injection required

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc — remove system()?

- We show: this perception is false.
ROP vs return-to-libc

attacker control of stack

arbitrary attacker computation and behavior via return-into-libc techniques

(given any sufficiently large codebase to draw on)
ROP Attacks

- Need control of memory around %esp
- Rewrite stack:
  - Buffer overflow on stack
  - Format string vuln to rewrite stack contents
- Move stack:
  - Overwrite saved frame pointer on stack; on leave/ret, move %esp to area under attacker control
  - Overflow function pointer to a register spring for %esp:
    - set or modify %esp from an attacker-controlled register
    - then return
Machine Instructions

- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip
ROP Execution

- Stack pointer (%esp) determines which instruction sequence to fetch & execute
- Processor doesn’t automatically increment %esp; — but the “ret” at end of each instruction sequence does
Building ROP Functionality

No-ops

- No-op instruction does nothing but advance `%eip`
- Return-oriented equivalent:
  - point to return instruction
  - advances `%esp`
- Useful in nop sled
Building ROP Functionality

Immediate constants

- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use
Building ROP Functionality

Control flow

- Ordinary programming:
  - (Conditionally) set %eip to new value

- Return-oriented equivalent:
  - (Conditionally) set %esp to new value
Gadgets: multiple instruction sequences

- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register:
  - Load address of source word into %eax
  - Load memory at (%eax) into %ebx
Finding Gadgets

Finding instruction sequences

- Any instruction sequence ending in “ret” is useful — could be part of a gadget

- **Algorithmic problem**: recover all sequences of valid instructions from libc that end in a “ret” insn

- Idea: at each ret (c3 byte) look back:
  - are preceding $i$ bytes a valid length-$i$ insn?
  - recurse from found instructions

- Collect instruction sequences in a trie
Conclusions

- Code injection is not necessary for arbitrary exploitation
- Defenses that distinguish “good code” from “bad code” are useless
- Return-oriented programming likely possible on every architecture, not just x86
- Compilers make sophisticated return-oriented exploits easy to write
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
%eax = 0x8048000
%ebx =

Code
pop %eax
ret
pop %ebx
ret
movl %eax, (%ebx)
ret
```

```
G1
5
jmp G2
0x8048000
jmp G3
...
```

```
Return Address
buf
```

```
Registers
%eax =
%ebx =
```

```
Memory
0x8048000 =
```
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
%eax = %ebx = 0x8048000
```

- Code
  - pop %eax
  - ret
  - pop %ebx
  - ret
  - movl %eax, (%ebx)
  - ret

- Stack

```
G1
5
jmp G2
0x8048000
jump G3
...
```

- Return Address
  - buf

- Registers
  - %eax =
  - %ebx =

- Memory
  - 0x8048000 =
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret
```

Code

```
%eax =
%ebx =
```

Registers

```
G1
5
jmp G2
0x8048000
jump G3
...
```

Stack

```
Return Address
buf
```

Memory

```
0x8048000 =
```
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret

%eax = 5
%ebx =
```

```
Code

Stack

G1

5

jmp G2

0x8048000

jump G3

...

Memory

Return Address

buf

0x8048000 =
```
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```assembly
pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax = 5</td>
<td>0x8048000 =</td>
</tr>
<tr>
<td>%ebx =</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop %eax</td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>pop %ebx</td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>movl %eax, (%ebx)</td>
</tr>
<tr>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>jmp G2</td>
</tr>
<tr>
<td>0x8048000</td>
</tr>
<tr>
<td>jump G3</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>

Return Address

buf
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
%eax = \text{5}
%ebx = 0x8048000
```

![Diagram showing code and stack for ROP example](image)
ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```asm
%eax = 5
%ebx = 0x8048000
```

```
pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret
```

```
G1

5

jmp G2

0x8048000

jump G3

...
```

```
buf
```

```
Return Address
```

```
Registers
```

```
Memory
```

```
Code
```

```
Stack
```

```
```
ROP Example

• Use ESP as program counter
  – E.g., Store 5 at address 0x8048000 (without introducing new code)

```
%eax = 5
%ebx = 0x8048000
```

```
pop %eax
ret
pop %ebx
ret
movl %eax, (%ebx)
ret
```
Advanced Defenses
Advanced Defenses

• Control-flow attack defenses operate at two stages
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ‣ Prevent attacker from getting control
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ‣ Prevent attacker from getting control
    • StackGuard, heap sanity checks, ASLR, shadow stacks, ...
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ▸ Prevent attacker from getting control
    • StackGuard, heap sanity checks, ASLR, shadow stacks, ...
  ▸ Prevent attacker from using control for malice
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ‣ Prevent attacker from getting control
    • StackGuard, heap sanity checks, ASLR, shadow stacks, ...
  ‣ Prevent attacker from using control for malice
    • NX, W (xor) X, ASLR, Control Flow Integrity (CFI), ...
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ‣ Prevent attacker from getting control
    • StackGuard, heap sanity checks, ASLR, shadow stacks, ...
  ‣ Prevent attacker from using control for malice
    • NX, W (xor) X, ASLR, Control Flow Integrity (CFI), ...

• For maximum security, a system should use a combination of these defenses
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ‣ Prevent attacker from getting control
    • StackGuard, heap sanity checks, ASLR, shadow stacks, ...
  ‣ Prevent attacker from using control for malice
    • NX, W (xor) X, ASLR, Control Flow Integrity (CFI), ...

• For maximum security, a system should use a combination of these defenses

• Q. Is subverting control-flow the only goal of an attacker?
Control-Flow Integrity
Control-Flow Integrity

• Goal: Ensure that process control follows source code
  ‣ Adversary can only choose authorized control-flow sequences
Control-Flow Integrity

• Goal: Ensure that process control follows source code
  ‣ Adversary can only choose authorized control-flow sequences

• Build a model from source code that describes control flow
  ‣ E.g., control-flow graph
Control-Flow Integrity

• Goal: Ensure that process control follows source code
  ‣ Adversary can only choose authorized control-flow sequences

• Build a model from source code that describes control flow
  ‣ E.g., control-flow graph

• Enforce the model on program execution
  ‣ Instrument control-flow code
    • Jumps, calls, returns, ...
Goal: Ensure that process control follows source code
  ‣ Adversary can only choose authorized control-flow sequences

Build a model from source code that describes control flow
  ‣ E.g., control-flow graph

Enforce the model on program execution
  ‣ Instrument control-flow code
    • Jumps, calls, returns, ...

Challenges
  ‣ Building accurate model
  ‣ Efficient enforcement
Software Control Flow Integrity
Techniques, Proofs, & Security Applications

Jay Ligatti summer 2004 intern work with:
Úlfar Erlingsson and Martín Abadi
Our Mechanism

call fp

return
Our Mechanism

`return` call fp

```
A_{call}   B_1
A_{call+1} B_{ret}
```
Our Mechanism

\[ \text{call fp} \xrightarrow{nop \ IMM_1} \text{return} \]

\[ \text{if(*fp != nop IMM_1) halt} \]

CFG excerpt

\[ \text{A}_{\text{call}} \rightarrow \text{B}_1 \]

\[ \text{A}_{\text{call+1}} \xleftarrow{} \text{B}_{\text{ret}} \]
Our Mechanism

\[
\begin{align*}
F_A & \quad \text{call fp} \\
& \quad \text{nop IMM}_2 \\
& \quad \text{if}(*fp \neq \text{nop IMM}_1) \quad \text{halt} \\
F_B & \quad \text{nop IMM}_1 \\
& \quad \text{if}(**\text{esp} \neq \text{nop IMM}_2) \quad \text{halt} \\
& \quad \text{return}
\end{align*}
\]

NB: Need to ensure bit patterns for nops appear nowhere else in code memory
More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int → int function

$A_{call}$ $\rightarrow$ $B_1$
$B_1$ $\rightarrow$ $C_1$

$\text{succ}(A_{call}) = \{B_1, C_1\}$
More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int $\rightarrow$ int function

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

CFG excerpt

$succ(A_{\text{call}}) = \{B_1, C_1\}$
Imprecise Return Information

Q: What if $F_B$ can return to many functions?

CFG excerpt:

$succ(B_{ret}) = \{A_{call+1}, D_{call+1}\}$
Imprecise Return Information

Q: What if $F_B$ can return to many functions?

CFG excerpt

$succ(B_{ret}) = \{A_{call+1}, D_{call+1}\}$

if(**esp != nop IMM$_2$) halt

return
Imprecise Return Information

Q: What if \( F_B \) can return to many functions?
A: Imprecise CFG

\[
\begin{align*}
\text{succ}(B_{\text{ret}}) &= \{A_{\text{call}+1}, D_{\text{call}+1}\} \\

\text{if}(**\text{esp} \neq \text{nop IMM}_2) &\text{ halt}
\end{align*}
\]
Imprecise Return Information

Q: What if $F_B$ can return to many functions?
A: Imprecise CFG

CFG excerpt

$$\text{succ}(B_{\text{ret}}) = \{A_{\text{call+1}}, D_{\text{call+1}}\}$$

CFG Integrity:
Changes to the PC are only to valid successor PCs, per succ().
No “Zig-Zag” Imprecision

CFG excerpt

A\text{\_call} \rightarrow B_1

E\text{\_call} \rightarrow C_1
No “Zig-Zag” Imprecision

CFG excerpt

A_{call} \rightarrow B_1

E_{call} \rightarrow C_1
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

CFG excerpt

A_{call} \rightarrow B_1 \rightarrow C_1

E_{call} \rightarrow B_1 \rightarrow C_1
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags
More Challenges
More Challenges

• Returns used as jumps
  ‣ E.g., signal handling
More Challenges

• Returns used as jumps
  ‣ E.g., signal handling
• Exceptions
More Challenges

• Returns used as jumps
  ‣ E.g., signal handling
• Exceptions
• Runtime generation of indirect jumps
  ‣ E.g., dynamic shared libraries
More Challenges

- Returns used as jumps
  - E.g., signal handling
- Exceptions
- Runtime generation of indirect jumps
  - E.g., dynamic shared libraries
- Indirect jumps using arithmetic operators
  - E.g., assembly
More Challenges

- Returns used as jumps
  - E.g., signal handling
- Exceptions
- Runtime generation of indirect jumps
  - E.g., dynamic shared libraries
- Indirect jumps using arithmetic operators
  - E.g., assembly
- **Take away:** CFI is a principled approach to stop control flow attacks, but challenges remain
Alternatives to CFI?
Alternatives to CFI?

• What are the fundamental enablers of ROP attacks?
  • CFI: violate control flow
  • Adversary can choose gadgets
Alternatives to CFI?

• What are the fundamental enablers of ROP attacks?
  • **CFI**: violate control flow
  • Adversary can choose gadgets
• Can we prevent adversaries from choosing useful gadgets?
  • In general, adversaries can create/obtain the same binary as is run by the victim
  • But, that need not be the case
Apply Crypto to Code?
Apply Crypto to Code?

- Can we **randomize** the program’s execution in such a way that an adversary cannot select gadgets?
Apply Crypto to Code?

• Can we *randomize* the program’s execution in such a way that an adversary cannot select gadgets?

• Given a *secret key* and a *program address space*, encrypt the address space such that
  • the probability that an adversary can locate a particular instruction (start of gadget) is sufficiently low
  • and the program still runs correctly and efficiently
Apply Crypto to Code?

- Can we **randomize** the program’s execution in such a way that an adversary cannot select gadgets?
- Given a **secret key** and a **program address space**, encrypt the address space such that
  - the probability that an adversary can locate a particular instruction (start of gadget) is sufficiently low
  - and the program still runs correctly and efficiently
- Called **address space randomization**
Prevent Injection on Stack
Prevent Injection on Stack

• One idea applied in practice
Prevent Injection on Stack

• One idea applied in practice
• Suppose an adversary wants to inject code onto the stack
  ‣ Write onto the stack (buffer overflow)
  ‣ Jump to that malcode (return address)
Prevent Injection on Stack

• One idea applied in practice
• Suppose an adversary wants to inject code onto the stack
  ‣ Write onto the stack (buffer overflow)
  ‣ Jump to that malcode (return address)
• Randomize the base address of the stack on each execution
  ‣ Prevents adversary from predicting malicious return address
Prevent Injection on Stack

• One idea applied in practice
• Suppose an adversary wants to inject code onto the stack
  ‣ Write onto the stack (buffer overflow)
  ‣ Jump to that malcode (return address)
• Randomize the base address of the stack on each execution
  ‣ Prevents adversary from predicting malicious return address
• Can we apply this idea more generally?
ASLR
ASLR

- For control-flow attacks, attacker needs absolute addresses
ASLR

- For control-flow attacks, attacker needs absolute addresses
- Address-space Layout Randomization (ASLR) randomizes base addresses of memory segments on each invocation of the program
  - Attacker cannot predict absolute addresses
ASLR

- For control-flow attacks, attacker needs absolute addresses
- Address-space Layout Randomization (ASLR) randomizes base addresses of memory segments on each invocation of the program
  - Attacker cannot predict absolute addresses
- Heap, stack, data, text, mmap, ...

![Diagram showing memory segments: Stack, Heap, Data, Text]
ASLR Implementations
ASLR Implementations

• Linux
ASLR Implementations

• Linux
  ‣ Introduced in Linux 2.6.12 (June 2005)
ASLR Implementations

• Linux
  ‣ Introduced in Linux 2.6.12 (June 2005)
  ‣ Shacham et al. [2004]: 16 bits of randomization defeated by a (remote) brute force attack in minutes
ASLR Implementations

• Linux
  ‣ Introduced in Linux 2.6.12 (June 2005)
  ‣ Shacham et al. [2004]: 16 bits of randomization defeated by a (remote) brute force attack in minutes
  ‣ Reality: ASLR for text segment (PIE) is rarely used
ASLR Implementations

- Linux
  - Introduced in Linux 2.6.12 (June 2005)
  - Shacham et al. [2004]: 16 bits of randomization defeated by a (remote) brute force attack in minutes
  - Reality: ASLR for text segment (PIE) is rarely used
- Only few programs in Linux use PIE
ASLR Implementations

- Linux
  - Introduced in Linux 2.6.12 (June 2005)
  - Shacham et al. [2004]: 16 bits of randomization defeated by a (remote) brute force attack in minutes
  - Reality: ASLR for text segment (PIE) is rarely used
    - Only few programs in Linux use PIE
    - Enough gadgets for ROP can be found in unrandomized code [Schwartz 2011]
ASLR Implementations
ASLR Implementations

• Windows
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
    • E.g., Oracle’s Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked ASLR-compatible
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
    • E.g., Oracle’s Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked ASLR-compatible
  ‣ From Vista study
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
    • E.g., Oracle’s Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked ASLR-compatible
  ‣ From Vista study
    • Good randomization for stack base
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
    • E.g., Oracle’s Java JRE, Adobe Reader, Mozilla Firefox,
      and Apple Quicktime (or one of their libraries) are not
      marked ASLR-compatible
  ‣ From Vista study
    • Good randomization for stack base
    • Insufficient randomization for some - e.g., heap and
      image
ASLR Implementations

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
    • E.g., Oracle’s Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked ASLR-compatible
  ‣ From Vista study
    • Good randomization for stack base
    • Insufficient randomization for some - e.g., heap and image
  • Lesson: bad crypto use will lead to vulnerabilities - again
ASLR Limitations
ASLR Limitations

• Attacks may leak randomization information
ASLR Limitations

- Attacks may leak randomization information
  - Disclosure attacks
ASLR Limitations

- Attacks may leak randomization information
  - Disclosure attacks
  - Use vulnerability to read an unauthorized program memory (extract code or randomizing state)
ASLR Limitations

• Attacks may leak randomization information
  • Disclosure attacks
  • Use vulnerability to read an unauthorized program memory (extract code or randomizing state)
• ASLR can be bypassed by information leaks about memory layout
ASLR Limitations

• Attacks may leak randomization information
  • Disclosure attacks
  • Use vulnerability to read an unauthorized program memory (extract code or randomizing state)

• ASLR can be bypassed by information leaks about memory layout
  ‣ E.g., format string vulnerabilities
ASLR Limitations

• Attacks may leak randomization information
  • Disclosure attacks
  • Use vulnerability to read an unauthorized program memory (extract code or randomizing state)
• ASLR can be bypassed by information leaks about memory layout
  ‣ E.g., format string vulnerabilities
• So, what can we do?
ASLR Limitations

• Attacks may leak randomization information
  • Disclosure attacks
  • Use vulnerability to read an unauthorized program memory (extract code or randomizing state)
• ASLR can be bypassed by information leaks about memory layout
  ‣ E.g., format string vulnerabilities
• So, what can we do?
  ‣ How do we avoid leaking the “key”?
Conclusion

- Defense against control-flow and data attacks is an ongoing arms race
- Principled approaches such as CFI and ASLR are promising
  - Significantly raised bar for attackers
  - However, they have implementation limitations
  - Active area of research