Binary Code Analysis and Secure Software Systems

06 – ASLR
Sang Kil Cha
Recap
Control Flow Hijack Attack

Corrupted Memory

Hijacked Control Flow

Attacker’s Code (Shellcode)

DEP makes this region non-executable!
Different Perspective: ASLR

Corrupted Memory

Attacker’s Code (Shellcode)

Hijacked Control Flow

ASLR makes it difficult to guess the address of the shellcode
Address Space Layout Randomization (ASLR)
World without ASLR

Same address space over and over again.
(The only thing that matters was environment variable)
Printing out ESP

#include <stdio.h>

int main(void)
{
    int x = 42;
    return printf("%08x\n", &x);
}
World with ASLR

Enable ASLR by:

$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space

Why 2? How would you figure out the meaning of this parameter?
ASLR

$2^{32}$

Stack

Heap

Code

Randomized based addresses
Randomness of ASLR on Linux x86

2^32

Stack
24 bits of randomness

Heap
16 bits of randomness

Code
16 bits of randomness

Q: Why not fully utilize 32 bits for randomization?
Previous Exploits Will *Not* Work w/ ASLR

- Memory layout will be **randomized** with ASLR.
  - Randomizes the **base address** of the stack, heap, and code segments

- We cannot know the address of shellcode nor library functions.

Are we safe now?
Attacking ASLR
Part 1. Entropy
ASLR Entropy is Small on x86

• Just 16 bits (heap, libraries) on x86

• Brute-forcing is possible for server applications that use `forking`.
  - Forked process has the same address space layout as its parent
  - Once we know the address of a function in LIBC, we can deduce the addresses of all functions in LIBC!

• Reference:
  On the Effectiveness of Address-Space Randomization, 
  *CCS 2004*
The Attack

• Target: Apache web server
  - Forks children on requests

• Vulnerability: Buffer Overflow

• Method: Return to LIIBC (usleep)
  - Try to brute-force the address of usleep
  - The fake parameter of usleep is 16,000,000 (waiting for 16 sec.)

• Once we know the address of usleep, we can determine the address of exec or system
How Often Should Randomization Happen?

• On *Windows*: every time the machine starts
  - Each module will get a random address once per boot
    (but, stack and heap will be randomized per execution)

• On *Linux*: every time a process loads
  - Each module will get a random address for every execution

Which one is better?
Windows is Faster than Linux

• On Windows: every time the machine starts
  – Each module will get a random address once per boot
  – Faster: relocation once at boot time

• On Linux: every time a process loads
  – Each module will get a random address for every execution
  – Slower: relocation fixups for every execution

But Linux is safer than Windows
How Much Safe against Brute-Forcing?

What is the expected number of trials to correctly guess the base address for each case?

Case 1: no randomization for each trial (*Windows*)
Case 2: re-randomization for each trial (*Linux*)
What is the probability of selecting the red ball? (When ASLR has $N$-bit randomness)

Case 1: Select balls without replacement (*Windows*)
Case 2: Select balls with replacement (*Linux*)
Selecting Balls W/O Replacement

(Windows)

\[ Pr[\text{success on 1st trial}] = \frac{1}{2^N} \]

\[ Pr[\text{success only on 2nd trial}] = \left( 1 - \frac{1}{2^N} \right) \left( \frac{1}{2^N - 1} \right) \]

\[ = \left( \frac{2^N - 1}{2^N} \right) \left( \frac{1}{2^N - 1} \right) = \frac{1}{2^N} \]
Selecting Balls W/O Replacement (Windows)

\[ Pr[\text{success only on 3rd trial}] = \left( \frac{2^N - 1}{2^N} \right) \left( \frac{2^N - 2}{2^N - 1} \right) \left( \frac{1}{2^N - 2} \right) = \frac{1}{2^N} \]

\[ Pr[\text{success only on kth trial}] = \left( \frac{2^N - 1}{2^N} \right) \times \cdots \times \left( \frac{2^N - k + 1}{2^N - k} \right) \times \left( \frac{1}{2^N - k + 1} \right) = \frac{1}{2^N} \]
Selecting Balls W/O Replacement (Windows)

Expected number of trials before success:

\[
\sum_{k=1}^{2^N} k \cdot Pr[\text{success only on } k\text{th trial}] = \sum_{k=1}^{2^N} \frac{k}{2^N}
\]

\[
= \frac{1}{2^N} \sum_{k=1}^{2^N} k
\]

\[
= \frac{1}{2^N} \cdot \frac{2^N (2^N + 1)}{2}
\]

\[
= \frac{2^N + 1}{2}
\]
Selecting Balls W/ Replacement (Linux)

\[ P_r[\text{success on 1st trial}] = \frac{1}{2^N} \]

\[ P_r[\text{success only on 2nd trial}] = \left( 1 - \frac{1}{2^N} \right) \frac{1}{2^N} \]

\[ P_r[\text{success only on } k\text{th trial}] = \left( 1 - \frac{1}{2^N} \right)^{k-1} \frac{1}{2^N} \]

Classic Geometric Distribution where \( p = 1 / 2^N \)
Selecting Balls W/ Replacement 
(*Linux*)

Expected number of trials before success:

\[
E[x] = \frac{1}{p} \text{ (for geometric distribution)}
\]

\[= 2^N\]

Classic Geometric Distribution where \(p = \frac{1}{2^N}\)
ASLR Comparison: Windows vs. Linux

Brute-force attack will success in

\[
\frac{2^N + 1}{2} \approx 2^N - 1 \quad \text{vs.} \quad 2^N
\]

trials on Windows vs. trials on Linux

Linux is \(\approx 2\) times safer than Windows against a brute-force attack
Attacking ASLR
Part 2. Fixed Address
Not All VMAs are Randomized

**Code** (.text) section is typically *not* randomized, because randomizing binary code is not trivial without source code

- Function calls with *absolute addresses*
- Accessing global data with *absolute addresses*

Q: Why is it difficult to replace absolute addresses with relative addresses?
Position-Independent Executable (PIE)

By default, on Debian/Ubuntu, gcc will create a PIE.

Run gcc with -fno-pic --no-pie option to get a non-PIE.

Let’s check the difference
Legacy Binaries are Not PIE

• 93%* of Linux binaries are NOT PIE-enabled.
• Thus, code section is not randomized.
• Thus, we can still use code reuse attacks (e.g., ROP)

* Surgically Returning to Randomized lib(c), ACSAC 2009
Performance vs. Security

• Relative-addressing instructions are slower than absolute-addressing instructions.

• Performance overhead of PIE on x86 is 10% on average.
  – Too much PIE is bad for performance, ETH Techreport, 2012

• Most applications on current x86 are still not PIEs.
Attack #1: Exploiting Leaked Pointers

• At the moment of exploitation, some registers can contain a useful address pointer.
  - E.g., `strcpy` returns a pointer to the target buffer in `eax`  
    ⇒ Jump to an instruction: `call [eax]`

• The stack often contains pointers to local buffers.
  - Return-to-ret
  - Return-to-pop-ret

• More on memory leaks in the next lecture.
Attack #2: ROP

Some code fragments are not randomized
⇒ Why not use ROP on them?

Fact: *relative offsets* between LIBC functions are the same regardless of ASLR

Use ROP to read a GOT entry, and compute the address of `system` by using relative offsets
Example*

Suppose we can get the address of open function from the GOT

\[(\text{base addr. of libc}) = (\text{addr. of open}) - (\text{offset of open from libc})\]
\[(\text{addr. of system}) = (\text{base addr. of libc}) + (\text{offset of system from libc})\]

* Surgically Returning to Randomized lib(c), ACSAC 2009
Example (ROP)

\[(\text{addr. of } \text{system}) = (\text{addr. of open}) - (\text{offset of open from libc}) + (\text{offset of system from libc})\]
Defenses?

• Using 64-bit CPU
  − Lots of entropy

• Detecting brute-force attempts is possible
  − Many crashes in a short amount of time

• Non-forking server

• Code randomization (a.k.a. fine-grained ASLR)
Code Randomization
Motivation

ASLR only changes base addresses of VMAs
- A single pointer leak can reveal the entire memory layout of a VMA
Fine-Grained ASLR

Classic ASLR

Function-level ASLR

Instruction-level ASLR
Fine-grained ASLR typically requires debugging information or source code in order to perform relocation (as in PIE).

Source-based approaches:

• Efficient Techniques for Comprehensive Protection from Memory Error Exploits, *USENIX Security 2005*

• Enhanced Operating System Security through Efficient and Fine-grained Address Space Randomization, *USENIX Security 2012*

**Binary level fine-grained ASLR?**
Binary-Level Fine-Grained ASLR

- Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization, *Oakland 2012*

- ILR: Where’d My Gadgets Go?, *Oakland 2012*

- Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code, *CCS 2012*
Intra Basic Block Randomization
(IEEE S&P 2012)

• Instruction Reordering
  mov ebx, 42
  mov eax, [ecx]
  mov eax, [ecx]
  mov eax, [ecx]
  mov ebx, 42

• Instruction Substitution
  mov eax, 0
  xor eax, eax

• Register Re-Allocation
  mov eax, [ecx]
  call [eax]
  mov ebx, [ecx]
  call [ebx]
Intra Basic Block Randomization (IEEE S&P 2012)

Does this technique mitigate return-to-libc attacks?
Binary-Level Fine-Grained ASLR

• Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization, *Oakland 2012*

• ILR: Where’d My Gadgets Go?, *Oakland 2012*

• Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code, *CCS 2012*
ILR: Instruction Location Randomization

```
mov ebx, 42
mov eax, [ecx]
add eax, ecx
ret
```

(Fall-through Map)

A ↦ C
B ↦ D
D ↦ A

A
---
add eax, ecx

B
---
mov ebx, 42

C
---
ret

D
---
mov eax, [ecx]
Binary-Level Fine-Grained ASLR

• Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization, *Oakland 2012*

• ILR: Where’d My Gadgets Go?, *Oakland 2012*

• Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code, *CCS 2012*
Reshuffling Basic Blocks
( CCS 2012 )

BBL1

mov ebx, eax
call 0x100

BBL2

add eax, 42
ret

BBL3

sub eax, 1
ret

BBL1

mov ebx, eax
call 0xc0

BBL2

add eax, 42
ret

BBL3

sub eax, 1
ret

jmp BBL2
Code Sharing Problem

- Every process will have different code
- No code sharing anymore!
Code Sharing Problem
Code Sharing Problem

Process 1
Main Code
C Library
Threading Library

Process 2
Main Code
C Library
Threading Library

Process 3
Main Code
C Library
Threading Library

Fine-graind ASLR prevents code sharing!
Position and Layout Agnostic Code

- **Process 1**
  - A
  - call B
  - B
  - xor eax,eax

- **Process 2**
  - C
    - xor eax,eax
  - D
    - call C

B and C are different
Position and Layout Agnostic Code

Process 1

0:A
4:B
...
call fs:[0x4]
xor eax,eax

A
B

Per-process jump table

C

Process 2

0:D
4:C
...
call fs:[0x4]
xor eax,eax

D

Same code
Reading

Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing,

*USENIX Security 2014*
ASLR

First ASLR design (Linux PaX)
Attack (Brute-force) on x86 PaX
Exploiting Fixed Code Section with ROP
Fine-grained ASLR on binary
Oxymoron

Similar Idea: Instruction Set Randomization (ISR)

- Encrypt the underlying system’s *instruction set*
- Decrypt each instruction at runtime

Countering Code-Injection Attacks with Instruction-Set Randomization,
*CCS 2003*

Where’s the FEEB?: The Effectiveness of Instruction Randomization,
*USENIX Security 2005*
Instruction Set Randomization (ISR)

The major issue of ISR is the performance overhead.

ASIST: Architectural Support for Instruction Set Randomization, CCS 2013

Still far from practice
History So Far ...

- Direct code injection
  - NX / DEP
  - Code-reuse attacks (e.g., ROP)
  - ASLR
  - Exploiting fixed code section with ROP
  - Not really used in practice

- ISR

- Fine-grained ASLR
Question?