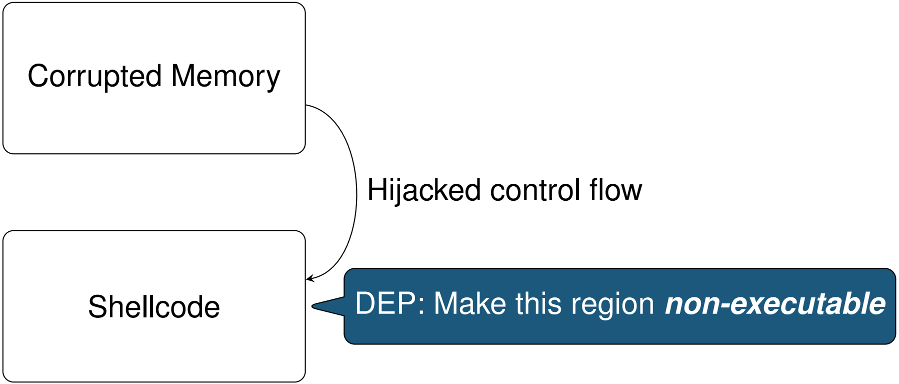
Lec 9: ASLR

IS561: Binary Code Analysis and Secure Software Systems

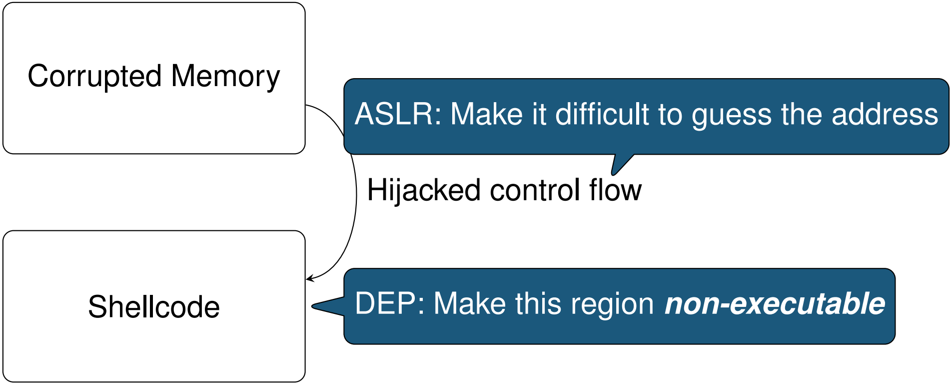
Sang Kil Cha

Address Space Layout Randomization (ASLR)

DEP vs. ASLR


Corrupted Memory
Hijacked control flow
Shellcode
DEP: Make this region ***non-executable***

DEP vs. ASLR


Corrupted Memory
ASLR: Make it difficult to guess the address
Hijacked control flow
Shellcode
DEP: Make this region *non-executable*

World without ASLR

Use the same address space over and over again.
(Previously, the only thing that matters was environment variables.)

Printing out RSP

```c
#include <stdio.h>

int main(void)
{
  int x = 42;
  return printf("%p\n", &x);
}
```

Turning on ASLR

You can enable ASLR by:

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

But, why 2? What's the meaning of it?

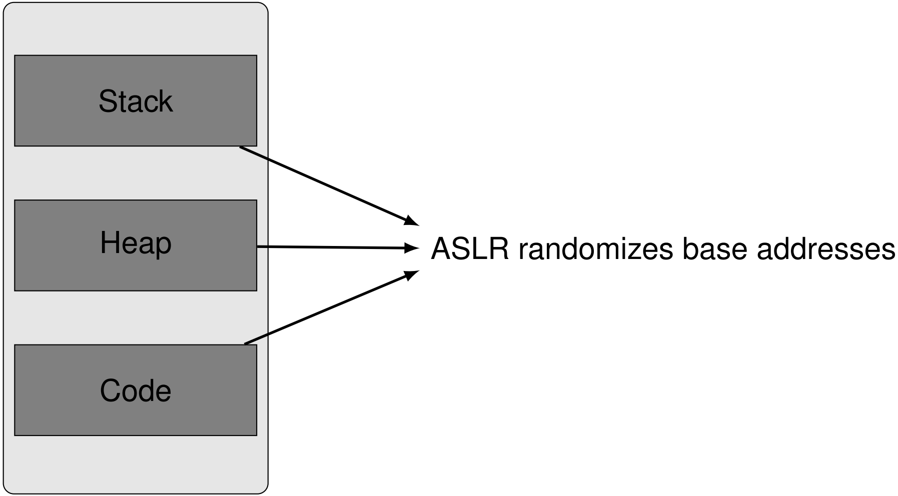
⇒ Read the manual: `man proc`[1]

[1] Always remember: manual is your best friend.

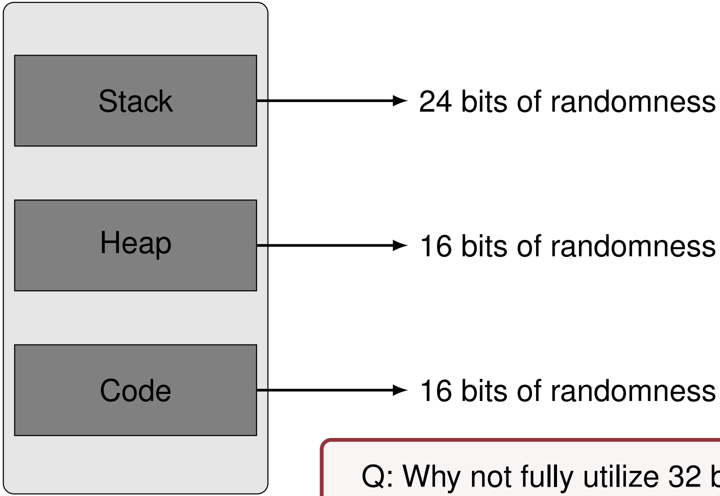
Manual Says ...

Value	Description
0	Turn ASLR off. This is the default for architectures that don't support ASLR, and when the kernel is booted with the norandmaps parameter.
1	Make the addresses of mmap(2) allocations, the stack, and the VDSO page randomized. Among other things, this means that shared libraries will be loaded at randomized addresses. The text segment of PIE-linked binaries will also be loaded at a randomized address.
2	Also support heap randomization.

ASLR randomizes VMAs



Randomness of Linux x86 ASLR



Q: Why not fully utilize 32 bits?

Previous Exploits Will Stop Working w/ ASLR

- ASLR will randomize the base addresses of the stack, heap, and code.
- We cannot know the address of our shellcode nor library functions.
  - Thus, no return-to-stack nor return-to-LIBC.

Previous Exploits Will Stop Working w/ ASLR

- ASLR will randomize the base addresses of the stack, heap, and code.
- We cannot know the address of our shellcode nor library functions.
  - Thus, no return-to-stack nor return-to-LIBC.

Are we safe now?

Attacking ASLR

Attack #1: Entropy is Small on x86

- Just 16 bits are used for heap and libraries on x86.
- Brute-forcing is possible when a server application uses `forking`[2].
  - A forked process has the same address space layout as its parent. Thus, every time we connect to the server, we will see the same memory space layout for the server process.
  - Once we know the address of a single function in LIBC, we can deduce the addresses of all functions in LIBC.

[2] On the Effectiveness of Address-Space Randomization, ***CCS 2004***.

Attack Example

- Target: Apache web server
  - Forks off children on requests.
- Vulnerability: Buffer overflow vulnerability.
- Method: Return-to-LIBC (`usleep`)
  1. Try to brute-force the address of `usleep` with a fake parameter of 16,000,000 (waiting for 16 sec.).
  2. Once we found the address, we can determine the address of `exec` or `system`, assuming that we have the LIBC binary on the server[3].

[3] If we know the OS distribution as well as the LIBC version, then we can get the same binary.

Randomization Frequency on Two Major OSes

- On Windows: every time the machine starts (like a forking server).
  - 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.

Randomization Frequency on Two Major OSes

- On Windows: every time the machine starts (like a forking server).
  - 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

No need to randomize the base address per execution.

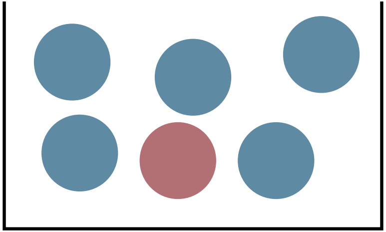
But, Linux is safer than Windows!

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***).

$2^N - 1$ Blue Balls and 1 Red Ball in a Jar



Let N be the number of randomized bits; so there are a total of 2^N possible base addresses. There is only one red ball in a jar, which corresponds to the expected base address. Then, what is the probability of selecting the red ball?

- Case 1: Select balls without replacement (***Windows***).
- Case 2: Select balls with replacement (***Linux***).

Case 1: Selecting Balls w/o Replacement

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

.

Case 1: Selecting Balls w/o Replacement

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

.

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

Case 1: Selecting Balls w/o Replacement

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

$$Pr[\text{success only on } k\text{th trial}] = \left(\frac{2^N - 1}{2^N}\right) \times \cdots \times \left(\frac{2^N - k + 1}{2^N - k}\right)\left(\frac{1}{2^N - k + 1}\right)$$
$$= \frac{1}{2^N}$$

Case 1: Selecting Balls w/o Replacement

Expected number of trials before success:

$$\sum_{k=1}^{2^N} k.Pr[\text{success only on } k\text{th trial}] = \sum_{k=1}^{2^N} \frac{k}{2^N}$$
$$= \frac{1}{2^N} \sum_{k=1}^{2^N} k$$
$$= \frac{1}{2^N} \frac{2^N(2^N+1)}{2}$$
$$= \frac{2^N+1}{2}$$

Case 2: Selecting Balls w/ Replacement

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

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

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

The classic geometric distribution where $p = \frac{1}{2^N}$.

Case 2: Selecting Balls w/ Replacement

Expected number of trials before success:

$$
\begin{aligned}
E[x] &= \frac{1}{p} \text{ (for geometric distribution)} \\
&= 2^N
\end{aligned}
$$

Comparison: Windows vs. Linux

Brute-force attack will succeed in

- $\frac{2^N+1}{2} \approx 2^{N-1}$ trials on Windows.
- 2^N trials on Linux.

Hence, Linux is $\approx 2\times$ safer than Windows against a brute-force attack.

Security vs. Performance?

Windows's ASLR is faster, but less secure than it of Linux.

Attack #2: Exploiting Fixed Addresses

Most binaries (before 2016) had non-randomized segments (VMAs).

Before 2016, compilers created ***non-PIE***[4] executables by default.

[4] Non Position-Independent Executable.

Position-Independent Executable (PIE)

Position-Independent Code (PIC) or PIE is code that runs regardless of its location (e.g., shellcode).

- "`gcc`" will produce a PIE by default.
- "`gcc -fno-pic -no-pie`" will produce a non-PIE.

PIE vs. non-PIE

```
...
lea RAX, qword ptr [RIP - 0x25]
mov qword ptr [RBP-0x8], RAX
mov RAX, qword ptr [RBP-0x8]
mov EDI, 0x2a
call RAX
...
```

```
...
mov qword ptr [RBP-0x8], 0x401106
mov RAX, qword ptr [RBP-0x8]
mov EDI, 0x2a
call RAX
...
```

Any Libraries Must Be Position-Independent

Any shared objects (.so), such as LIBC, are position-independent.

Why?

Legacy Binaries Are Not a PIE

- 93% of Linux binaries were not a PIE (in 2009).
- Thus, the code sections were not randomized.
- Thus, code reuse attacks (e.g., ROP) are still possible on legacy binaries.

But, why?

Security vs. Performance

- Relative-addressing instructions are slower than absolute-addressing instructions.
- Performance overhead of PIE on x86 is 10% on average[5].
- Most applications on current x86 are still not PIEs.

[5] Too much PIE is bad for performance, ETH Techreport, 2012.

ROP-based Attack on Legacy Binaries

- Code sections are not randomized, hence we can use ROP.
- But, LIBC address is randomized! Cannot directly return to LIBC functions.

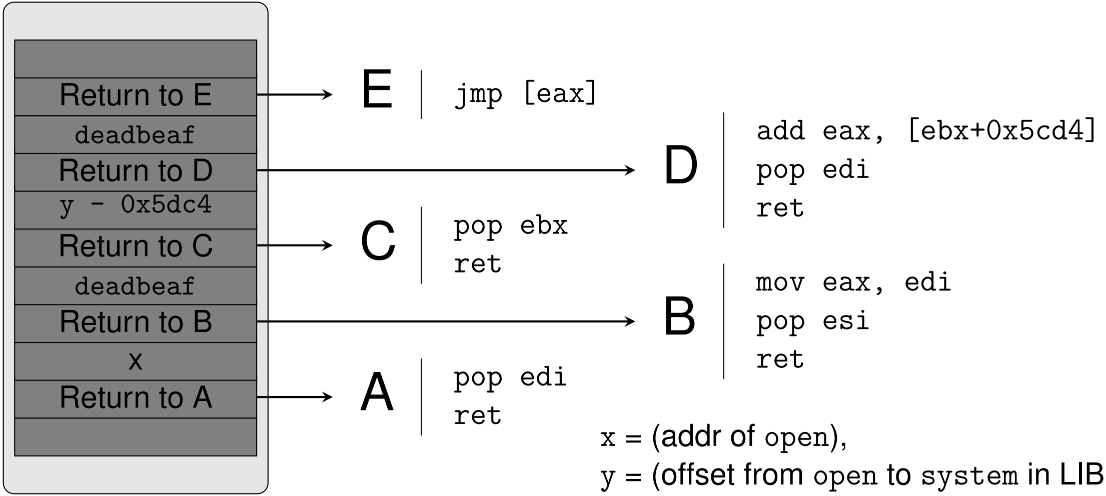
> But, still, relative offsets between LIBC functions are the same regardless of ASLR.

Exploitation Idea

- If a LIBC function has been invoked at least once, GOT should contain a concrete address of the function in LIBC.
- Therefore, we will read the GOT entry using ROP and compute the address of `system` by using the relative offset between the LIBC function and `system`.

$$
\begin{aligned}
(\text{addr of } \texttt{system}) = {} & (\text{addr of } \texttt{open}) \\
& + (\text{offset from } \texttt{open} \text{ to } \texttt{system} \text{ in LIBC})
\end{aligned}
\tag{1}
$$

Example ROP



x = (addr of `open`),

y = (offset from `open` to `system` in LIBC)

Possible Defenses?

- Use PIEs.
- Use 64-bit CPU: lots of entropy.
- Detect brute-forcing attacks (as there should be many crashes in a short amount of time).
- Use non-forking servers.

Better ASLR?

Single pointer leakage can reveal the entire memory layout of a VMA.

Can we make it harder?

Fine-grained ASLR

Randomize code within a VMA boundary.

- Function-level randomization.
- Block-level randomization.
- Instruction-level randomization.

Fine-grained ASLR: Design Challenges

- Can we apply fine-grained ASLR without debugging information?
- How often should we apply fine-grained ASLR? Performance impact?

Example: In-Place Code Randomization

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

- Instruction reordering.

```
mov ebx, 42
mov eax, [ecx]
```
→
```
mov eax, [ecx]
mov ebx, 42
```

- Instruction substitution.

```
mov rax, 0
```
→
```
xor rax, rax
```

- Register re-allocation.

```
mov rax, [rcx]
call [rax]
```
→
```
mov rbx, [rcx]
call [rbx]
```

Example: In-Place Code Randomization

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

- Instruction reordering.

```
mov ebx, 42            mov eax, [ecx]
mov eax, [ecx]   →     mov ebx, 42
```

- Instruction substitution.

```
mov rax, 0       →     xor rax, rax
```

- Register re-allocation.

```
mov rax, [rcx]         mov rbx, [rcx]
call [rax]       →     call [rbx]
```

Does this technique mitigate return-to-LIBC attacks?

Example: Instruction Location Randomization

ILR: Where'd My Gadgets Go?, ***Oakland 2012***.

```
mov rbx, 42
mov rax, [rcx]
add rax, rcx
ret
```

→

A `add rax, rcx`

B `mov rbx, 42`

C `ret`

D `mov rax, [rcx]`

Fall-through map:

A $\mapsto$ C

B $\mapsto$ D

D $\mapsto$ A

New Challenge

Fine-grained ASLR disallows code sharing between processes! For example, sharing LIBC is not possible anymore!

[6] Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing, ***USENIX Security 2014***

New Challenge

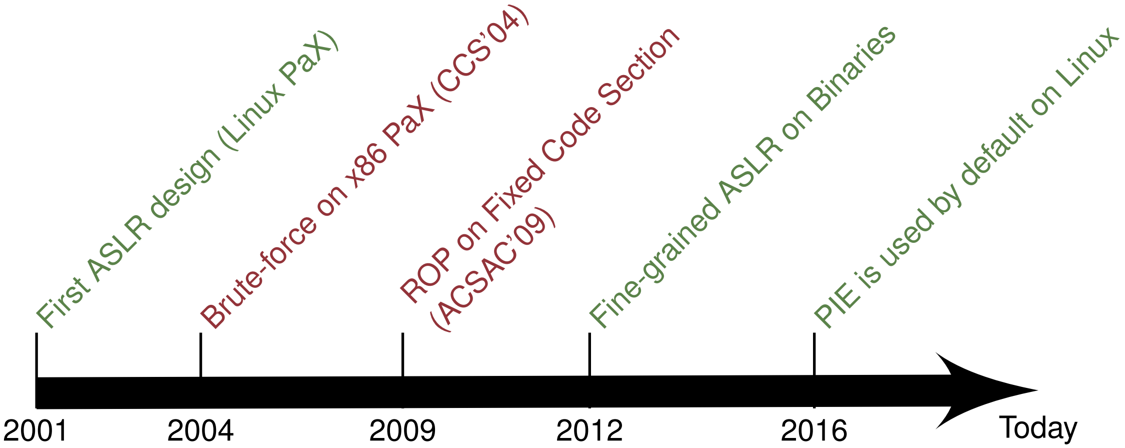
Fine-grained ASLR disallows code sharing between processes! For example, sharing LIBC is not possible anymore!

Reading: Can we apply fine-grained ASLR while still allowing code sharing?[6]

---

[6] Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing, ***USENIX Security 2014***

ASLR Attack and Defense Timeline



Question?