CS165 – Computer Security

Control Flow Defense and ROP

Oct 19, 2021

Agenda

Canary / Stack Cookies

Data Execution Prevention (DEP) /No eXecute (NX)

Address Space Layout Randomization (ASLR)

Agenda

Canary / Stack Cookies

Data Execution Prevention (DEP) /No eXecute (NX)

Address Space Layout Randomization (ASLR)

NEXT

Other Non-randomized Sections

- Dynamically linked libraries are loaded at runtime. This is called ***lazy binding***.

- Two important data structures
  - Global Offset Table
  - Procedure Linkage Table

commonly positioned statically at compile-time

Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

LIBC

```
<dynamic_printf_addr>:
...
```

Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

LIBC

```
<dynamic_printf_addr>:
...
```

Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

Transfer control to PLT entry of printf

LIBC

```
<dynamic_printf_addr>:
...
```

Dynamic Linking

```
...
printf(“hello ”);
...
printf(“world\n”);
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

Linker

LIBC

```
<dynamic_printf_addr>:
...
```

Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

Linker fills in the actual addresses of library functions

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

Dynamic Linking

```
...
printf("hello ");
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_printf_addr
```

Subsequent calls to printf do not require the linker

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

Exploiting the linking process

- GOT entries are really function pointers positioned at known addresses

- **Idea:** use other vulnerabilities to take control (e.g., format string)

GOT Hijacking

```
...
printf(usr_input);
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

GOT Hijacking

```
...
printf(usr_input);
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: dynamic_linker_addr
```

Use the format string to overwrite a GOT entry

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

GOT Hijacking

```
...
printf(usr_input);
...
printf(“world\n”);
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: any_attacker_addr
```

Use the format string to overwrite a GOT entry

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

GOT Hijacking

```
...
printf(usr_input);
...
printf("world\n");
...
```

```
<printf@plt>: jmp GOT[printf]
```

```
GOT
...
<printf>: any_attacker_addr
```

The next invocation transfers control wherever the attacker wants (e.g., system, pop-ret, etc)

LIBC

```
<dynamic_printf_addr>:
...
```

Linker

Quiz

- What defenses can defeat the GOT Hijacking attack?
  - Canary, DEP/NX, ASLR?

```
#include <unistd.h>
void main(int argc, char ** argv){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    system(name[0]);
    exit(0);
}
```

```
80483c4 <main>:
80483c4:   55                      push   %ebp
80483c5:   89 e5                   mov    %esp,%ebp
80483c7:   83 e4 f0                and    $0xfffffff0,%esp
80483ca:   83 ec 20                sub    $0x20,%esp
80483cd:   c7 44 24 18 b0 84 04    movl   $0x80484b0,0x18(%esp)
80483d4:   08
80483d5:   c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483dc:   00
80483dd:   8b 44 24 18             mov    0x18(%esp),%eax
80483e1:   89 04 24                mov    %eax,(%esp)
80483e4:   e8 03 ff ff ff          call   80482ec <system@plt>
80483e9:   c9                      leave
80483ea:   c3                      ret


80482dc <__gmon_start__@plt>:
80482dc:    ff 25 b0 95 04 08      jmp    *0x80495b0
80482e2:    68 00 00 00 00         push   $0x0
80482e7:    e9 e0 ff ff ff         jmp    80482cc <_init+0x30>

80482ec <system@plt>:
80482ec:    ff 25 b4 95 04 08      jmp    *0x80495b4
80482f2:    68 08 00 00 00         push   $0x8
80482f7:    e9 d0 ff ff ff         jmp    80482cc <_init+0x30>
```

```
#include <unistd.h>
void main(int argc, char ** argv){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    system(name[0]);
    exit(0);
}
```

```
root@debian:~# readelf -x 23 a.out
Hex dump of section '.got.plt':
  0x080495a4   d0940408 00000000 00000000 e2820408
  0x080495b4   f2820408 02830408
```

```
80483c4 <main>:
80483c4:   55                        push   %ebp
80483c5:   89 e5                     mov    %esp,%ebp
80483c7:   83 e4 f0                  and    $0xfffffff0,%esp
80483ca:   83 ec 20                  sub    $0x20,%esp
80483cd:   c7 44 24 18 b0 84 04      movl   $0x80484b0,0x18(%esp)
80483d4:   08
80483d5:   c7 44 24 1c 00 00 00      movl   $0x0,0x1c(%esp)
80483dc:   00
80483dd:   8b 44 24 18               mov    0x18(%esp),%eax
80483e1:   89 04 24                  mov    %eax,(%esp)
80483e4:   e8 03 ff ff ff            call   80482ec <system@plt>
80483e9:   c9                        leave
80483ea:   c3                        ret


80482dc <__gmon_start__@plt>:
80482dc:    ff 25 b0 95 04 08        jmp    *0x80495b0
80482e2:    68 00 00 00 00           push   $0x0
80482e7:    e9 e0 ff ff ff           jmp    80482cc <_init+0x30>

80482ec <system@plt>:
80482ec:    ff 25 b4 95 04 08        jmp    *0x80495b4
80482f2:    68 08 00 00 00           push   $0x8
80482f7:    e9 d0 ff ff ff           jmp    80482cc <_init+0x30>
```

```c
#include <unistd.h>
void main(int argc, char ** argv){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    system(name[0]);
    exit(0);
}
```

```
root@debian:~# readelf -x 23 a.out
Hex dump of section '.got.plt':
  0x080495a4   d0940408 00000000 00000000 e2820408
  0x080495b4   f2820408 02830408
```

```
80483c4 <main>:
80483c4:   55                      push   %ebp
80483c5:   89 e5                   mov    %esp,%ebp
80483c7:   83 e4 f0                and    $0xfffffff0,%esp
80483ca:   83 ec 20                sub    $0x20,%esp
80483cd:   c7 44 24 18 b0 84 04    movl   $0x80484b0,0x18(%esp)
80483d4:   08
80483d5:   c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483dc:   00
80483dd:   8b 44 24 18             mov    0x18(%esp),%eax
80483e1:   89 04 24                mov    %eax,(%esp)
80483e4:   e8 03 ff ff ff          call   80482ec <system@plt>
80483e9:   c9                      leave
80483ea:   c3                      ret


80482dc <__gmon_start__@plt>:
80482dc:    ff 25 b0 95 04 08      jmp    *0x80495b0
80482e2:    68 00 00 00 00         push   $0x0
80482e7:    e9 e0 ff ff ff         jmp    80482cc <_init+0x30>

80482ec <system@plt>:
80482ec:    ff 25 b4 95 04 08      jmp    *0x80495b4
80482f2:    68 08 00 00 00         push   $0x8
80482f7:    e9 d0 ff ff ff         jmp    80482cc <_init+0x30>
```

```c
#include <unistd.h>
void main(int argc, char ** argv){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    system(name[0]);
    exit(0);
}
```

```
root@debian:~# readelf -x 23 a.out
Hex dump of section '.got.plt':
  0x080495a4   d0940408 00000000 00000000 e2820408
  0x080495b4   f2820408 02830408
```

```
80483c4 <main>:
80483c4:   55                        push   %ebp
80483c5:   89 e5                     mov    %esp,%ebp
80483c7:   83 e4 f0                  and    $0xfffffff0,%esp
80483ca:   83 ec 20                  sub    $0x20,%esp
80483cd:   c7 44 24 18 b0 84 04      movl   $0x80484b0,0x18(%esp)
80483d4:   08
80483d5:   c7 44 24 1c 00 00 00      movl   $0x0,0x1c(%esp)
80483dc:   00
80483dd:   8b 44 24 18               mov    0x18(%esp),%eax
80483e1:   89 04 24                  mov    %eax,(%esp)
80483e4:   e8 03 ff ff ff            call   80482ec <system@plt>
80483e9:   c9                        leave
80483ea:   c3                        ret


80482dc <__gmon_start__@plt>:
80482dc:    ff 25 b0 95 04 08        jmp    *0x80495b0
80482e2:    68 00 00 00 00           push   $0x0
80482e7:    e9 e0 ff ff ff           jmp    80482cc <_init+0x30>

80482ec <system@plt>:
80482ec:    ff 25 b4 95 04 08        jmp    *0x80495b4
80482f2:    68 08 00 00 00           push   $0x8
80482f7:    e9 d0 ff ff ff           jmp    80482cc <_init+0x30>
```

```c
#include <unistd.h>
void main(int argc, char ** argv){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    system(name[0]);
    exit(0);
}
```

```
root@debian:~# readelf -x 23 a.out
Hex dump of section '.got.plt':
  0x080495a4  d0940408 00000000 00000000 e2820408
  0x080495b4  f2820408 02830408
```

80482f2

```
80483c4 <main>:
80483c4:   55                      push   %ebp
80483c5:   89 e5                   mov    %esp,%ebp
80483c7:   83 e4 f0                and    $0xfffffff0,%esp
80483ca:   83 ec 20                sub    $0x20,%esp
80483cd:   c7 44 24 18 b0 84 04    movl   $0x80484b0,0x18(%esp)
80483d4:   08
80483d5:   c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483dc:   00
80483dd:   8b 44 24 18             mov    0x18(%esp),%eax
80483e1:   89 04 24                mov    %eax,(%esp)
80483e4:   e8 03 ff ff ff          call   80482ec <system@plt>
80483e9:   c9                      leave
80483ea:   c3                      ret


80482dc <__gmon_start__@plt>:
80482dc:    ff 25 b0 95 04 08      jmp    *0x80495b0
80482e2:    68 00 00 00 00         push   $0x0
80482e7:    e9 e0 ff ff ff         jmp    80482cc <_init+0x30>

80482ec <system@plt>:
80482ec:    ff 25 b4 95 04 08      jmp    *0x80495b4
80482f2:    68 08 00 00 00         push   $0x8
80482f7:    e9 d0 ff ff ff         jmp    80482cc <_init+0x30>
```

```
#include <unistd.h>
void main(int argc, char ** argv){
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    system(name[0]);
    exit(0);
}
```

```
root@debian:~# readelf -x 23 a.out
Hex dump of section '.got.plt':
  0x080495a4   d0940408 00000000 00000000 e2820408
  0x080495b4   f2820408 02830408
```

80482f2

```
80483c4 <main>:
80483c4:   55                      push   %ebp
80483c5:   89 e5                   mov    %esp,%ebp
80483c7:   83 e4 f0                and    $0xfffffff0,%esp
80483ca:   83 ec 20                sub    $0x20,%esp
80483cd:   c7 44 24 18 b0 84 04    movl   $0x80484b0,0x18(%esp)
80483d4:   08
80483d5:   c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483dc:   00
80483dd:   8b 44 24 18             mov    0x18(%esp),%eax
80483e1:   89 04 24                mov    %eax,(%esp)
80483e4:   e8 03 ff ff ff          call   80482ec <system@plt>
80483e9:   c9                      leave
80483ea:   c3                      ret


80482dc <__gmon_start__@plt>:
80482dc:    ff 25 b0 95 04 08      jmp    *0x80495b0
80482e2:    68 00 00 00 00         push   $0x0
80482e7:    e9 e0 ff ff ff         jmp    80482cc <_init+0x30>

80482ec <system@plt>:
80482ec:    ff 25 b4 95 04 08      jmp    *0x80495b4
80482f2:    68 08 00 00 00         push   $0x8
80482f7:    e9 d0 ff ff ff         jmp    80482cc <_init+0x30>
```

Many other techniques

- ret2bss, ret2data, ret2heap, ret2eax
- string pointer
- ret2dtors
  - overwriting dtors section

How to attack with ASLR?

Attack

- **Brute Force**
- **Non-randomized memory**
- **Stack Juggling**
- **GOT Hijacking**

All assume some memory non-randomized

How to attack with ASLR?

Attack

Brute Force	Non-randomized memory	Stack Juggling	GOT Hijacking

Try to infer the randomized offset

Recover addresses

Read vulnerability

Side channels

Example: using format string vulnerability

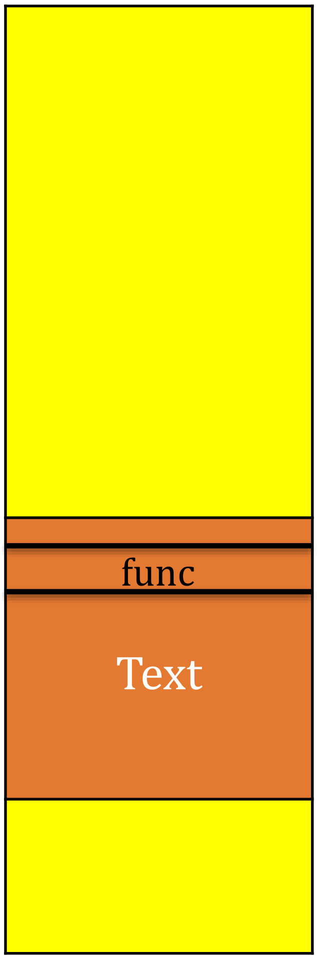
- View (a, a+64k) to locate specific sequence of instructions
  - Infer $\mathbf{r_1}$

a **+ 16 bit rand** $\mathbf{r_1}$

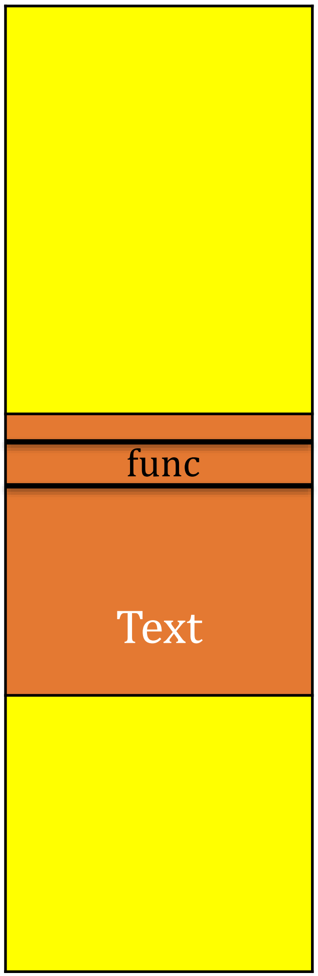
Program

- Code
- Uninitialized data
- Initialized data

Example: using CPU cache side channel

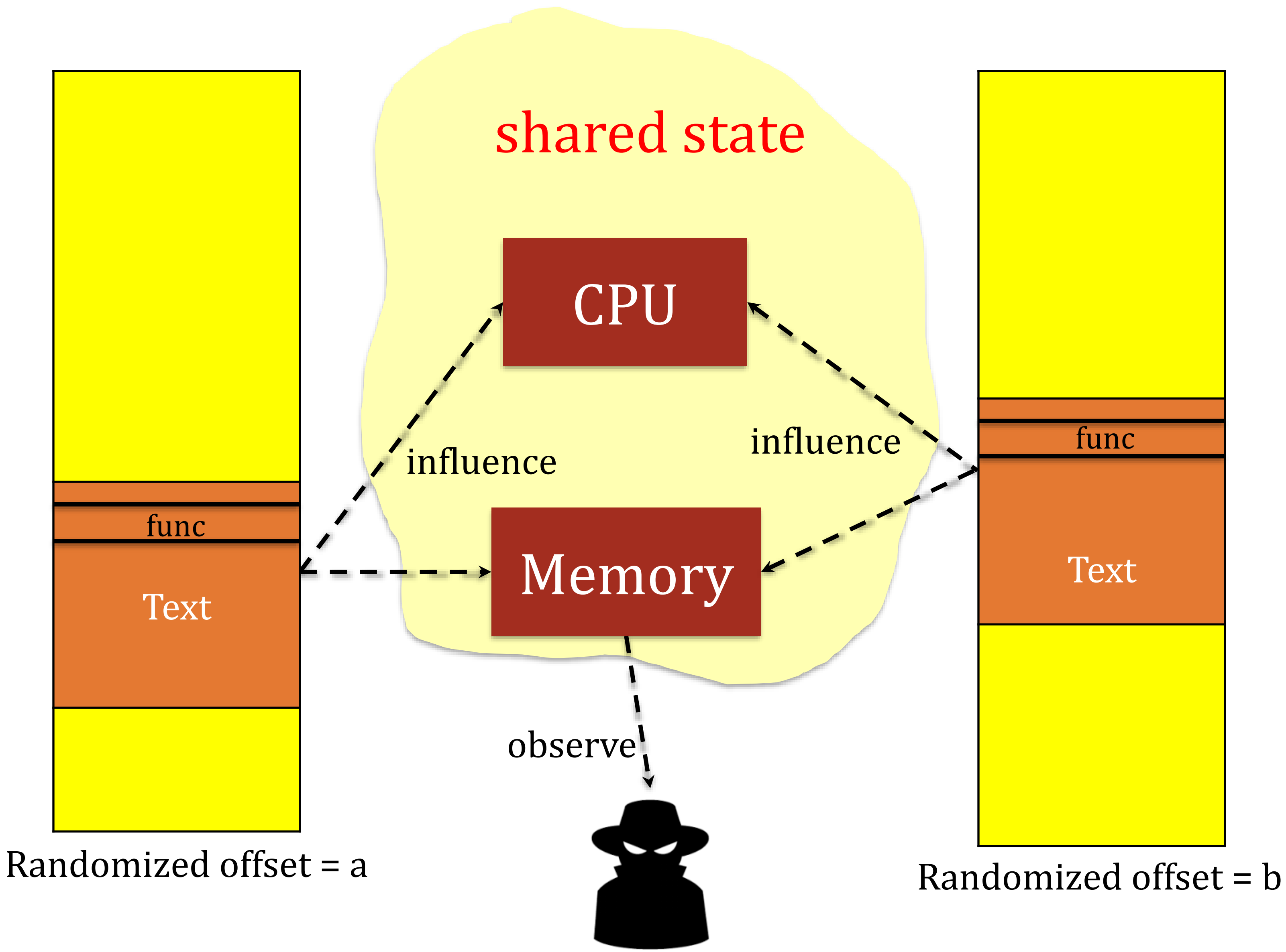


Randomized offset = a

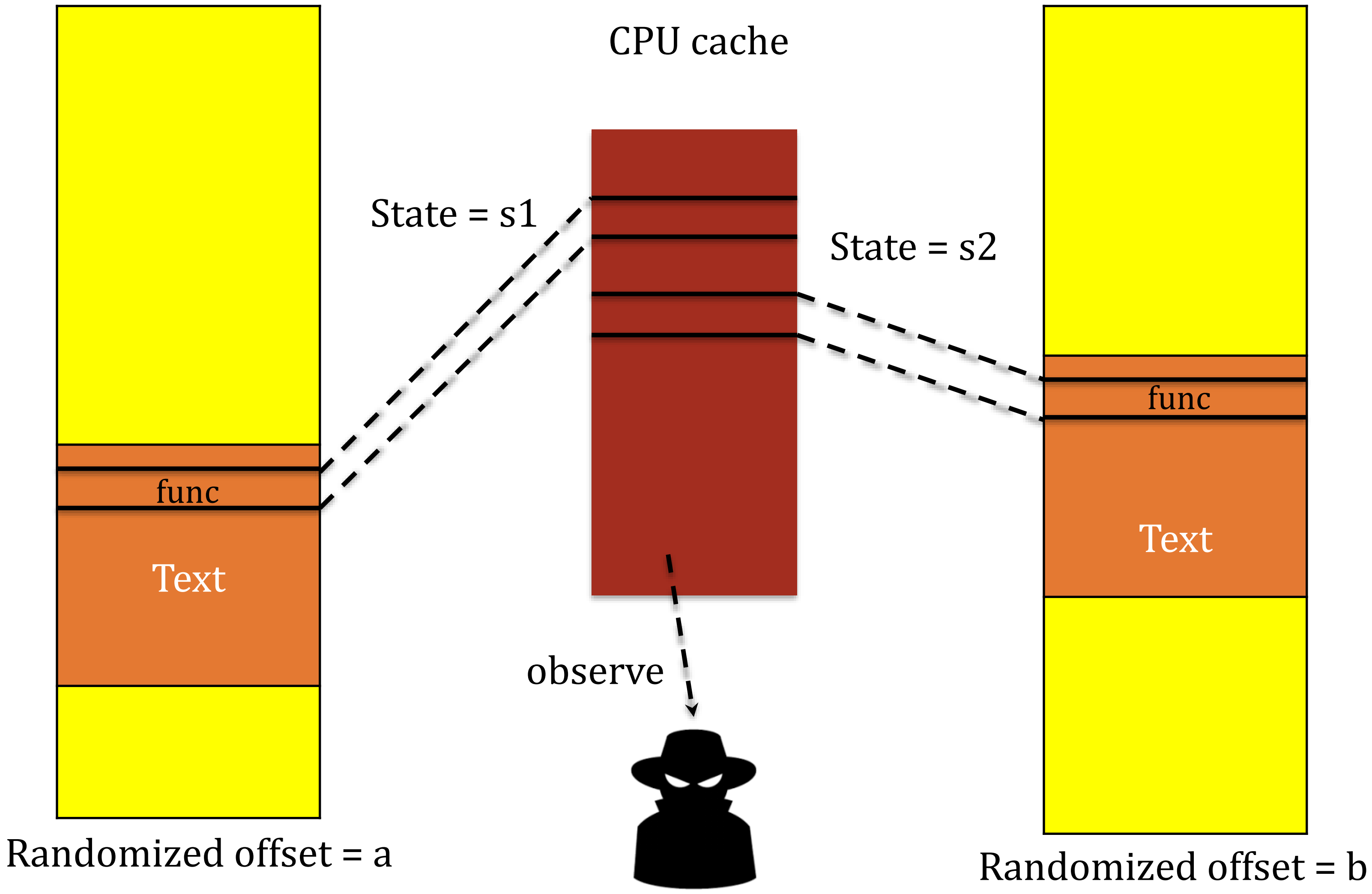


Randomized offset = b

Example: using CPU cache side channel



Example: using CPU cache side channel



The Security of ASLR

Optional Reading:

On the Effectiveness of Address-Space Randomization
by Shacham et al, ACM CCS 2004

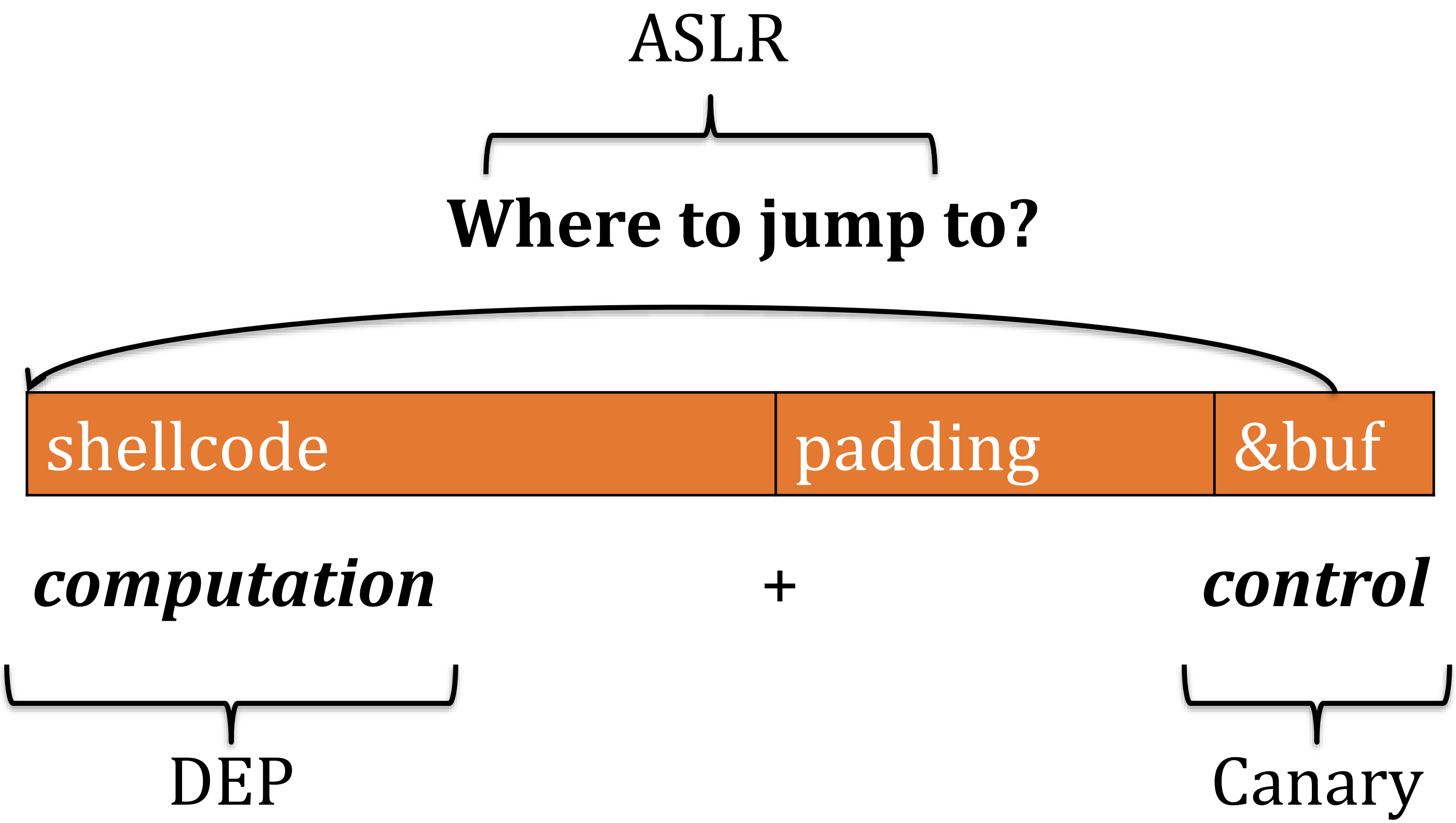
Agenda

Canary / Stack Cookies ✔

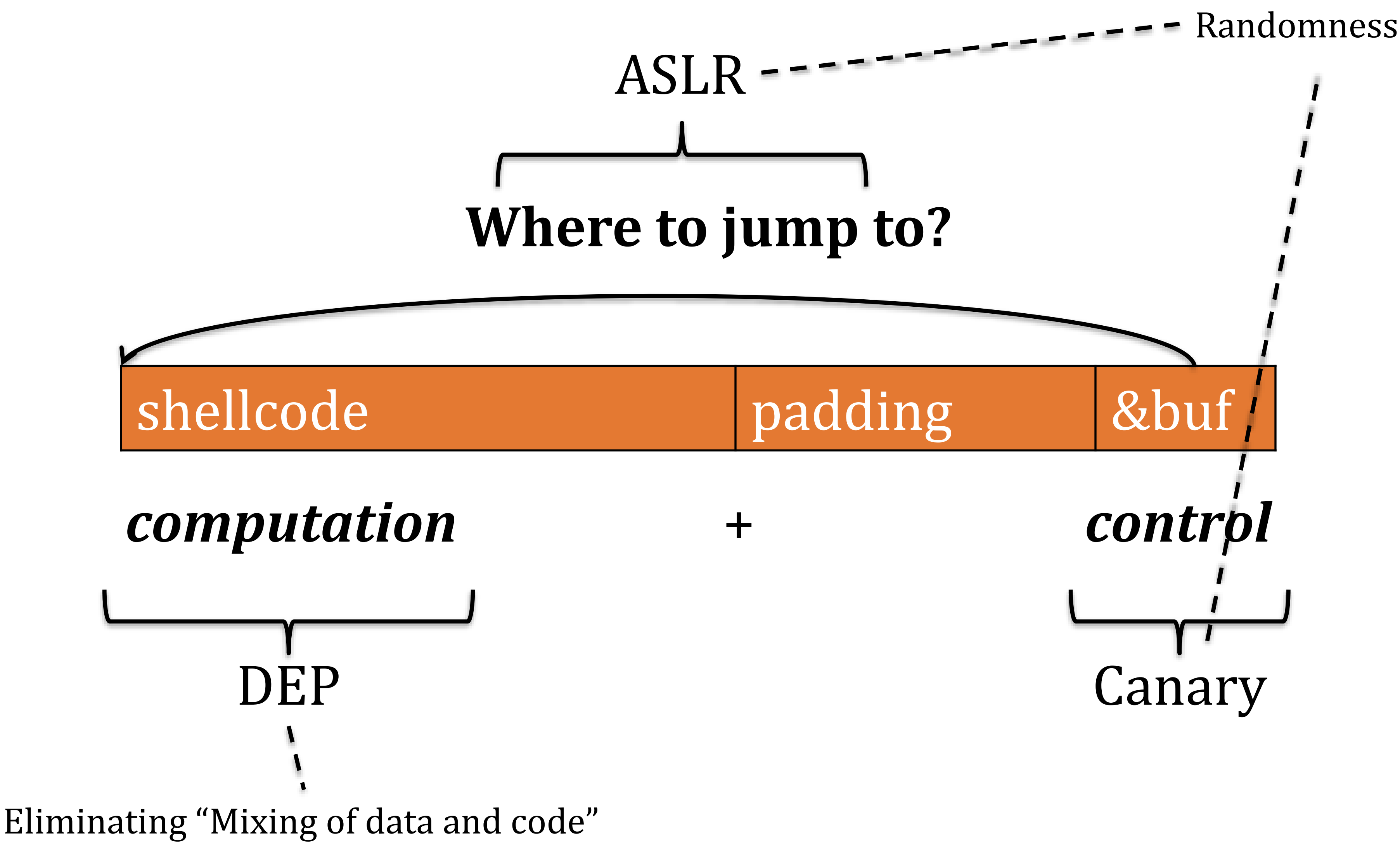
Data Execution Prevention (DEP) /No eXecute (NX) ✔

Address Space Layout Randomization (ASLR) ✔

Summary


ASLR
Where to jump to?
shellcode
padding
&buf
computation
+
control
DEP
Canary

Summary


Randomness
ASLR
Where to jump to?
shellcode
padding
&buf
computation
+
control
DEP
Canary
Eliminating “Mixing of data and code”

Return-Oriented Programming (ROP)

Control Flow Hijack: Always control + computation

shellcode (aka payload)	padding	&buf

computation + ***control***

Return-oriented programming (ROP): shellcode without code injection

Acknowledgement: Some slides from David Brumley , Ed Schwartz, Kevin Snow, and Luci Davi

Agenda

ROP Overview

Gadgets

Disassembling code

Agenda

ROP Overview

NEXT

Gadgets

Disassembling code

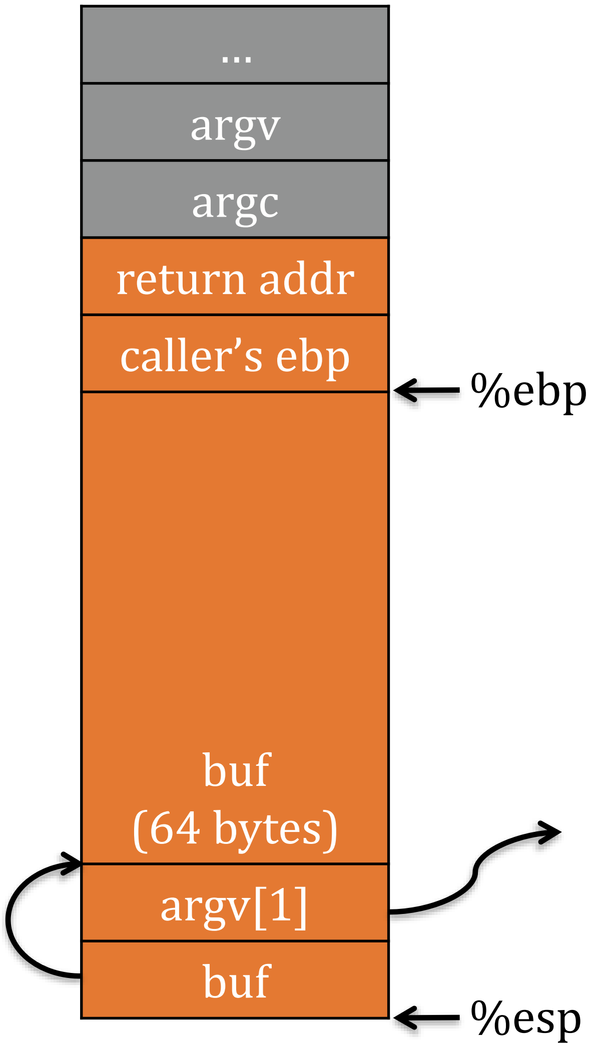
Motivation: Return-to-libc Attack

Bypassing DEP!

Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!

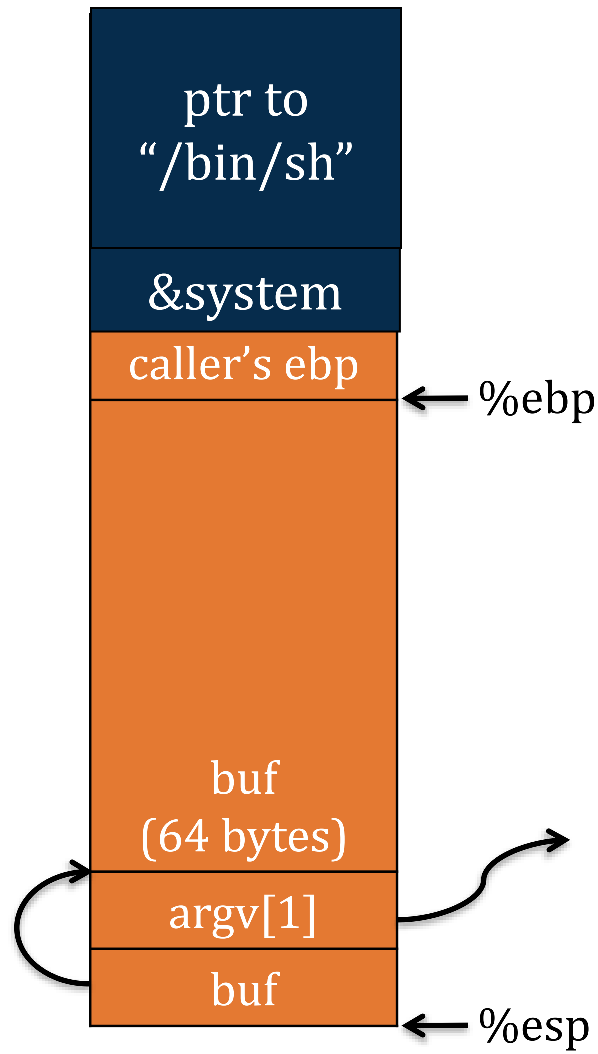


Motivation: Return-to-libc Attack

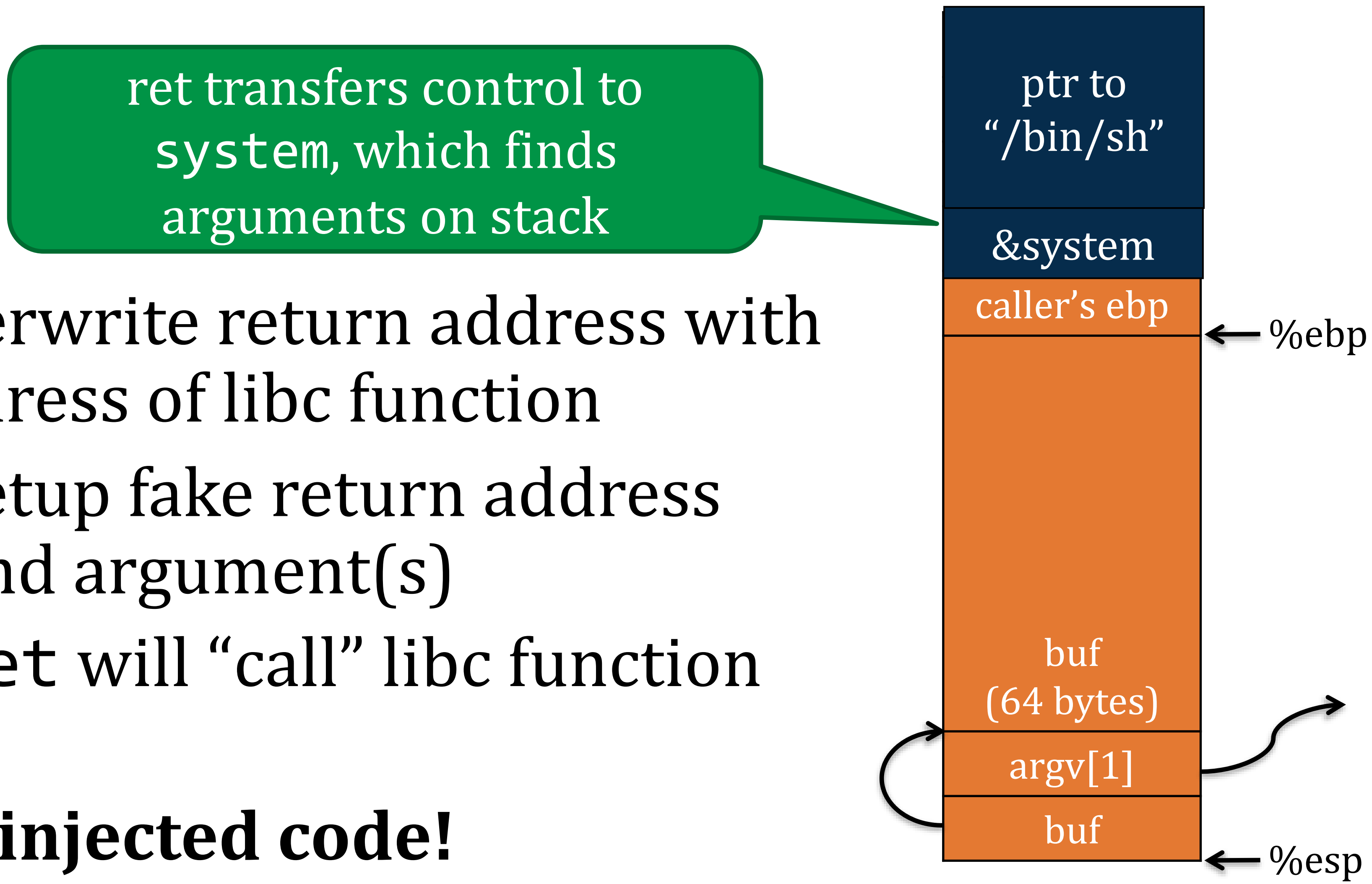
Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!



Motivation: Return-to-libc Attack



Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!

What if we don't know the address of `system()`?

What if we don't know the address of `system()`?

Have to defeat ASLR

What if we don't know the address of `system()`?

Have to defeat ASLR

But `system()` may not be linked,
or
perhaps we need more than just `system()`

What if we don't know the address of `system()`?

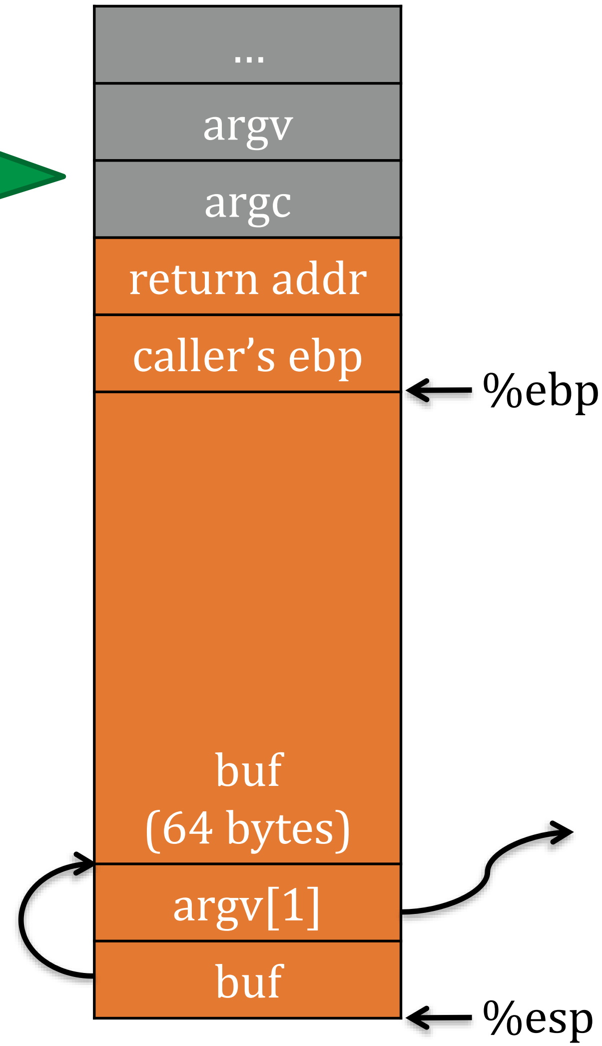
Have to defeat ASLR

But `system()` may not be linked,
or
perhaps we need more than just `system()`

ROP to the rescue:
A generalization of ret2libc

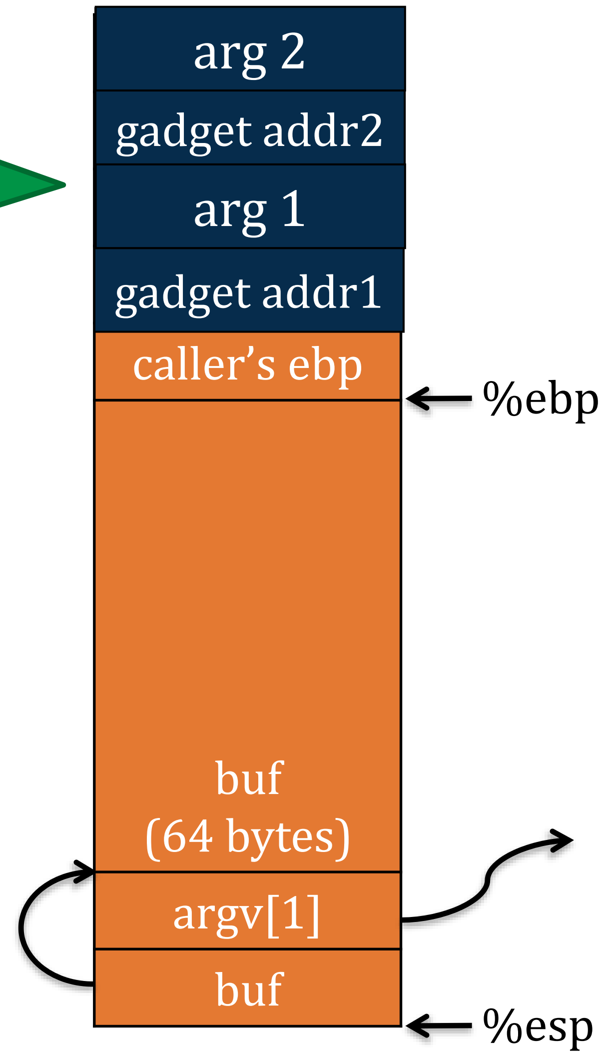
Return-Oriented Programming (ROP)

Need to find an instruction sequence, aka ***gadget***, with esp



Return-Oriented Programming (ROP)

Need to find an instruction sequence, aka *gadget*, with esp



Return Oriented Programming Techniques

Geometry of Flesh on the Bone, Shacham et al, CCS 2007

The New Yo

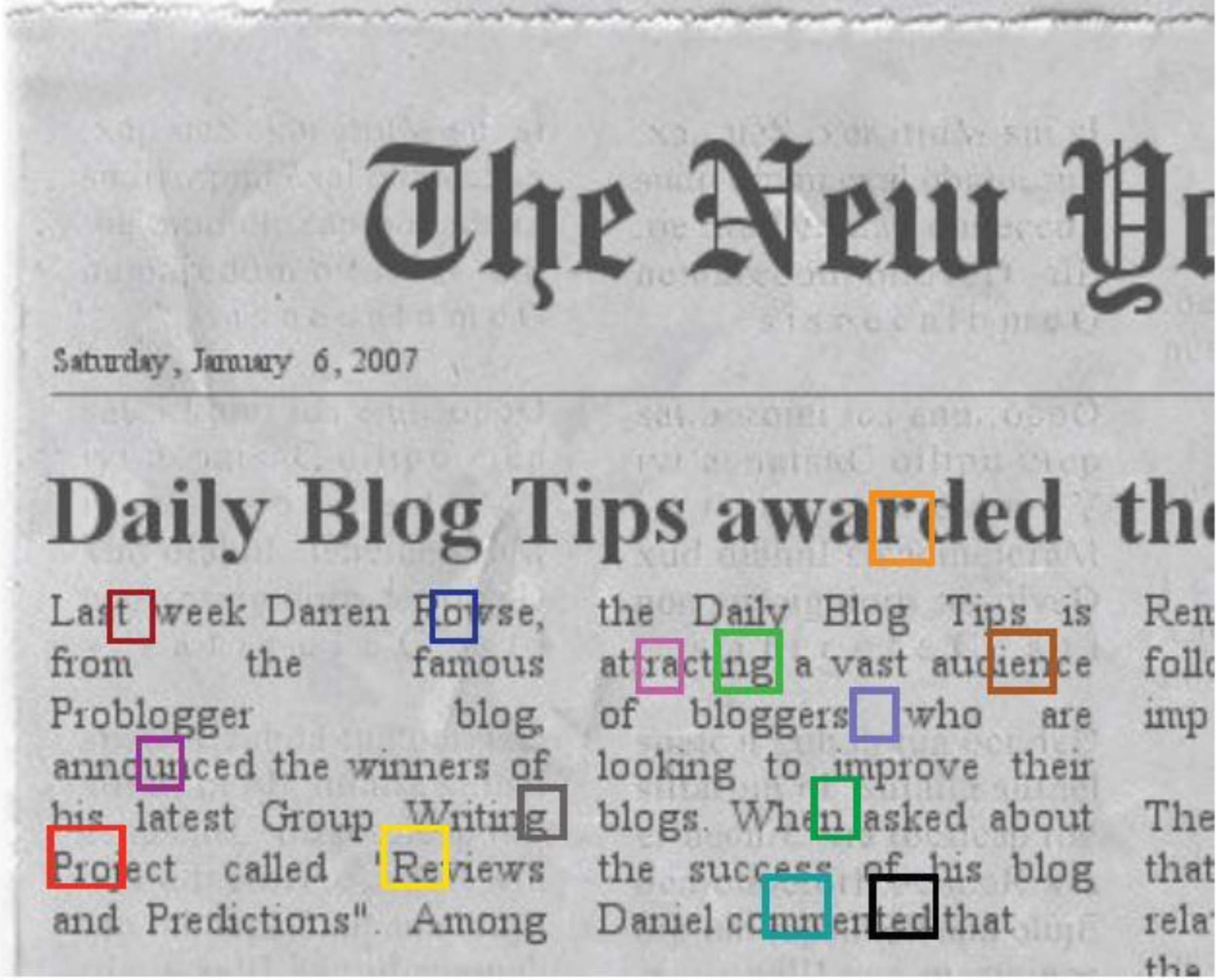
Saturday, January 6, 2007

Daily Blog Tips awarded th

Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

Ren
follc
imp

The
that
rela
the



The New Y

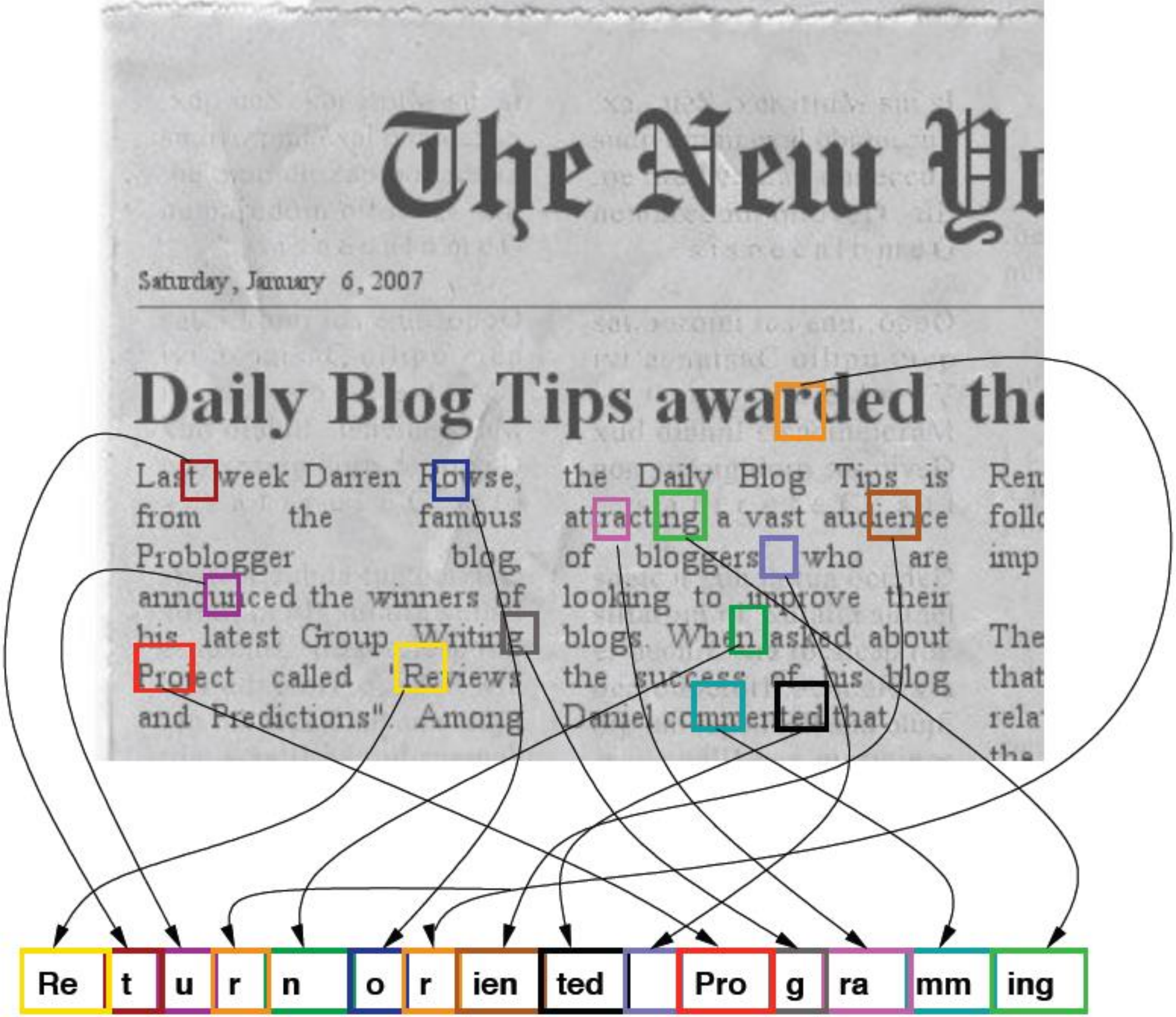
Saturday, January 6, 2007

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

Ren
follo
imp

The
that
rela


The New Yo
Saturday, January 6, 2007
Daily Blog Tips awarded the
Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among
the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that
Re
t
u
r
n
o
r
ien
ted
Pro
g
ra
mm
ing

ROP Programming: Key Steps

1. Disassemble code
2. Identify ***useful*** code sequences as gadgets
3. Assemble gadgets into desired shellcode

ROP Overview

ROP Overview

- Idea: We forge shell code out of existing application logic gadgets

ROP Overview

- Idea: We forge shell code out of existing application logic gadgets

- Requirements:
vulnerability + gadgets + some unrandomized code

ROP Overview

- Idea: We forge shell code out of existing application logic gadgets
- Requirements:
  vulnerability + gadgets + some unrandomized code
- History:
  - No code randomized: Code injection

    DEP enabled by default: ROP attacks using libc gadgets publicized ~2007
  - Libc randomized

    ASLR library load points

    Today: Windows 7/10 compiler randomizes text by default, Randomizing text on Linux not straightforward.

Agenda

ROP Overview

Gadgets

Disassembling code

Agenda

ROP Overview

Gadgets

NEXT

Disassembling code

There are many

semantically equivalent

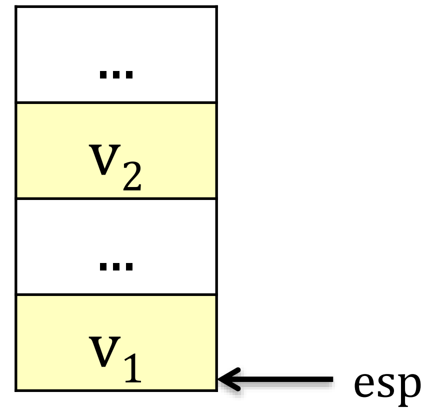
ways to achieve the same net

shellcode effect

Equivalence

```
Mem[v2] = v1
```

Desired Logic

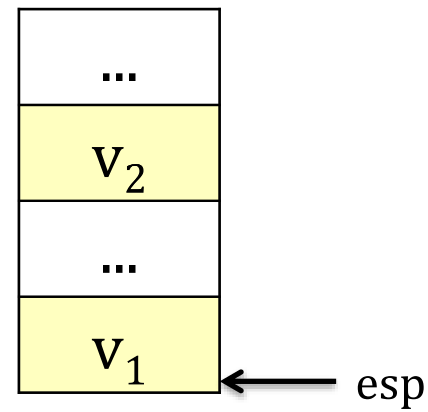


Stack

Equivalence

```
Mem[v2] = v1
```

Desired Logic



Stack

a_1: `mov eax, [esp]`

a_2: `mov ebx, [esp+8]`

a_3: `mov [ebx], eax`

Implementation 1

Gadgets

A gadget is any instruction sequence ending with `ret`

Gadgets

`Mem[v2] = v1`

Desired Logic

Stack
a_5
v_2
a_3
v_1 ← esp

Stack

Suppose a_3 and a_5 on stack

eax	
ebx	
eip	a_1

```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic

Suppose a_3 and a_5 on stack

a_5
v_2
a_3
v_1

← esp

Stack

eax	v_1
ebx	
eip	a_1

a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic

a_5
v_2
a_3
v_1

esp

Stack

eax	v_1
ebx	
eip	a_1

a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic

a_5	
v_2	
a_3	← esp
v_1	

Stack

eax	v_1
ebx	
eip	a_3

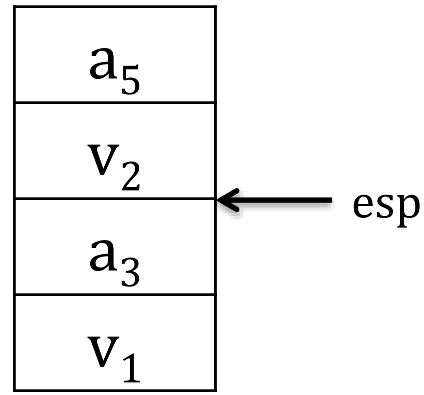
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

eax	v_1
ebx	
eip	a_3

a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic

a_5	
v_2	← esp
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_3

a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic

a_5	
v_2	← esp
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_4

a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic

a_5	← esp
v_2	
a_3	
v_1	

Stack

eax	v_1
ebx	v_2
eip	a_5

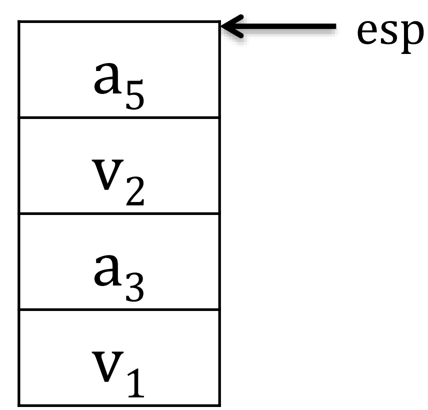
```
a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
```

Implementation 2

Gadgets

`Mem[v2] = v1`

Desired Logic



Stack

eax	v_1
ebx	v_2
eip	a_5

a_1: `pop eax;`
a_2: `ret`
a_3: `pop ebx;`
a_4: `ret`
a_5: `mov [ebx], eax`

Implementation 2

Equivalence

```
Mem[v2] = v1
```

Desired Logic

a_3
v_2
a_2
v_1 ← esp

Stack

"Gadgets"

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

Implementation 2

Equivalence

`Mem[v2] = v1`

Desired Logic

Stack
a_3
v_2
a_2
v_1 ← esp

Stack

semantically equivalent

"Gadgets"

Implementation 1		Implementation 2
a_1: mov eax, [esp]	⟷	a_1: pop eax; ret
a_2: mov ebx, [esp+8]	⟷	a_2: pop ebx; ret
a_3: mov [ebx], eax	⟷	a_3: mov [ebx], eax

Implementation 1 **Implementation 2**

Equivalence

```
Mem[v2] = v1
```

Desired Logic

a_3
v_2
a_2
v_1

Stack

No need to be contiuous!

```
a1: pop eax; ret
...
a3: mov [ebx], eax
...
a2: pop ebx; ret
```

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```

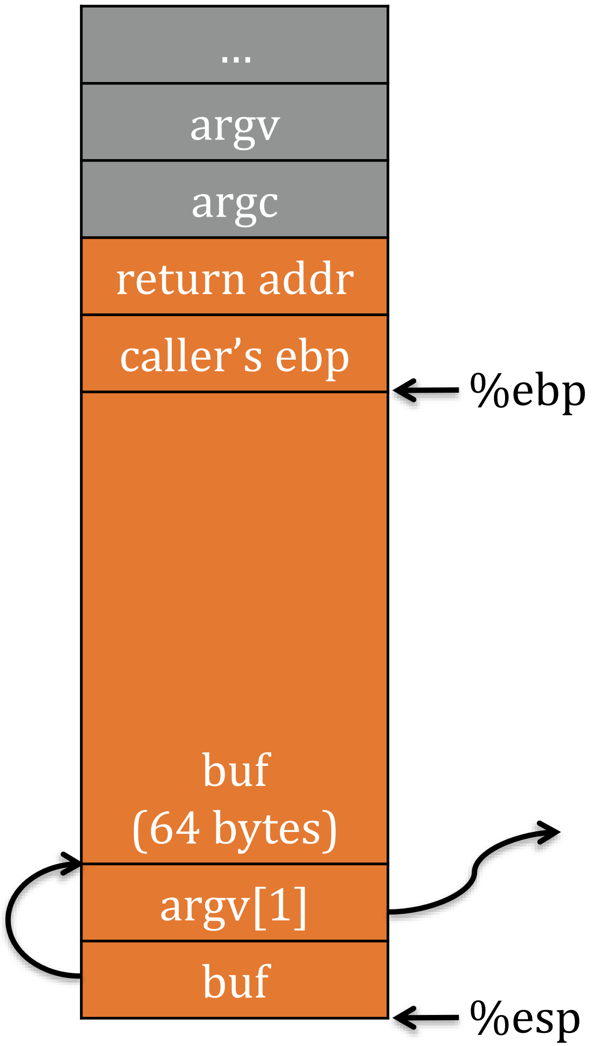
Implementation 2

Return-Oriented Programming (ROP)

```
Mem[v2] = v1
```

Desired *Shellcode*

- Find needed instruction gadgets at addresses a_1, a_2, and a_3 in *existing* code
- Overwrite stack to execute a_1, a_2, and then a_3

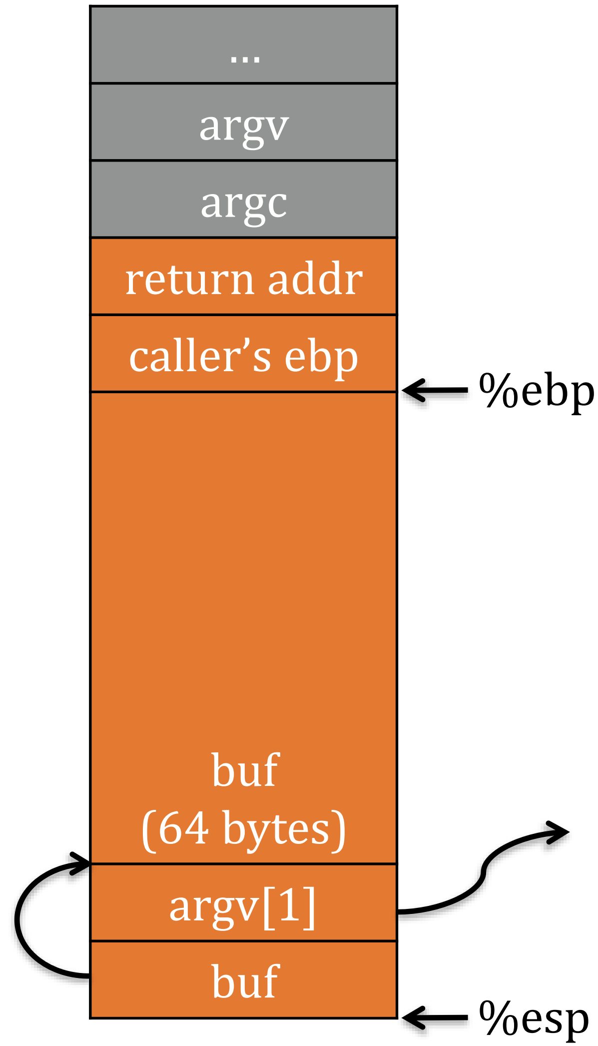


Return-Oriented Programming (ROP)

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```



Return-Oriented Programming (ROP)

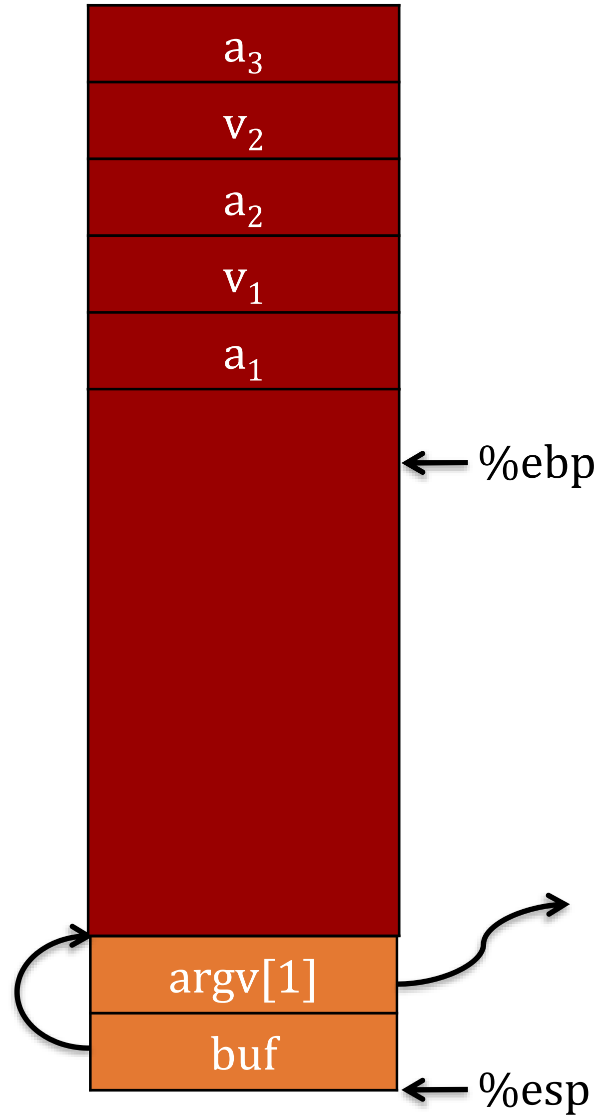
`Mem[v2] = v1`

Desired *Shellcode*

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`



Return-Oriented Programming (ROP)

```
Mem[v2] = v1
```

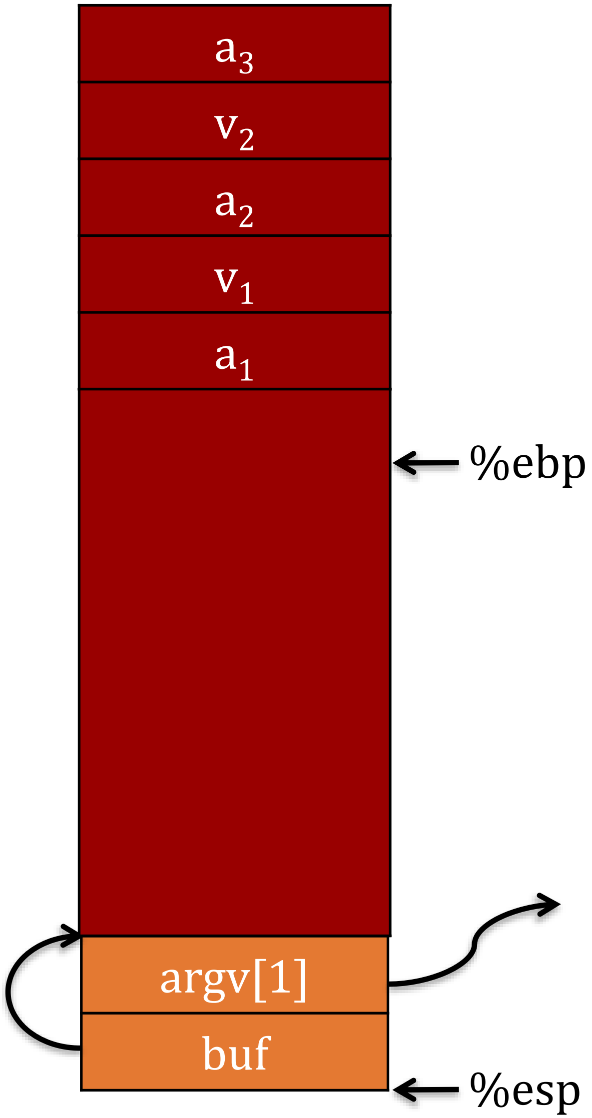
Desired *Shellcode*

a_1: `pop eax; ret`

a_2: `pop ebx; ret`

a_3: `mov [ebx], eax`

Desired store executed!



Quiz

```
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

Draw a stack diagram and ROP exploit to **pop a value 0xBBBBBBBB into eax and add 80.**

a_1: add eax, 0x80; pop ebp; ret

a_2: pop eax; ret

Known Gadgets

Quiz

```
void foo(char *input){
    char buf[512];
    ...
    strcpy (buf, input);
    return;
}
```

a_1: add eax, 0x80; pop ebp; ret

a_2: pop eax; ret

<data for pop ebp>
a_1
0xBBBBBBBB
a_2
saved ebp
buf

$$\overbrace{\text{AAAAA ...}}^{\text{Overwrite buf}}\ \overbrace{a_2\ \text{0xBBBBBBBB}\ a_1}^{\text{gadget 1 + data}}$$

gadget 2

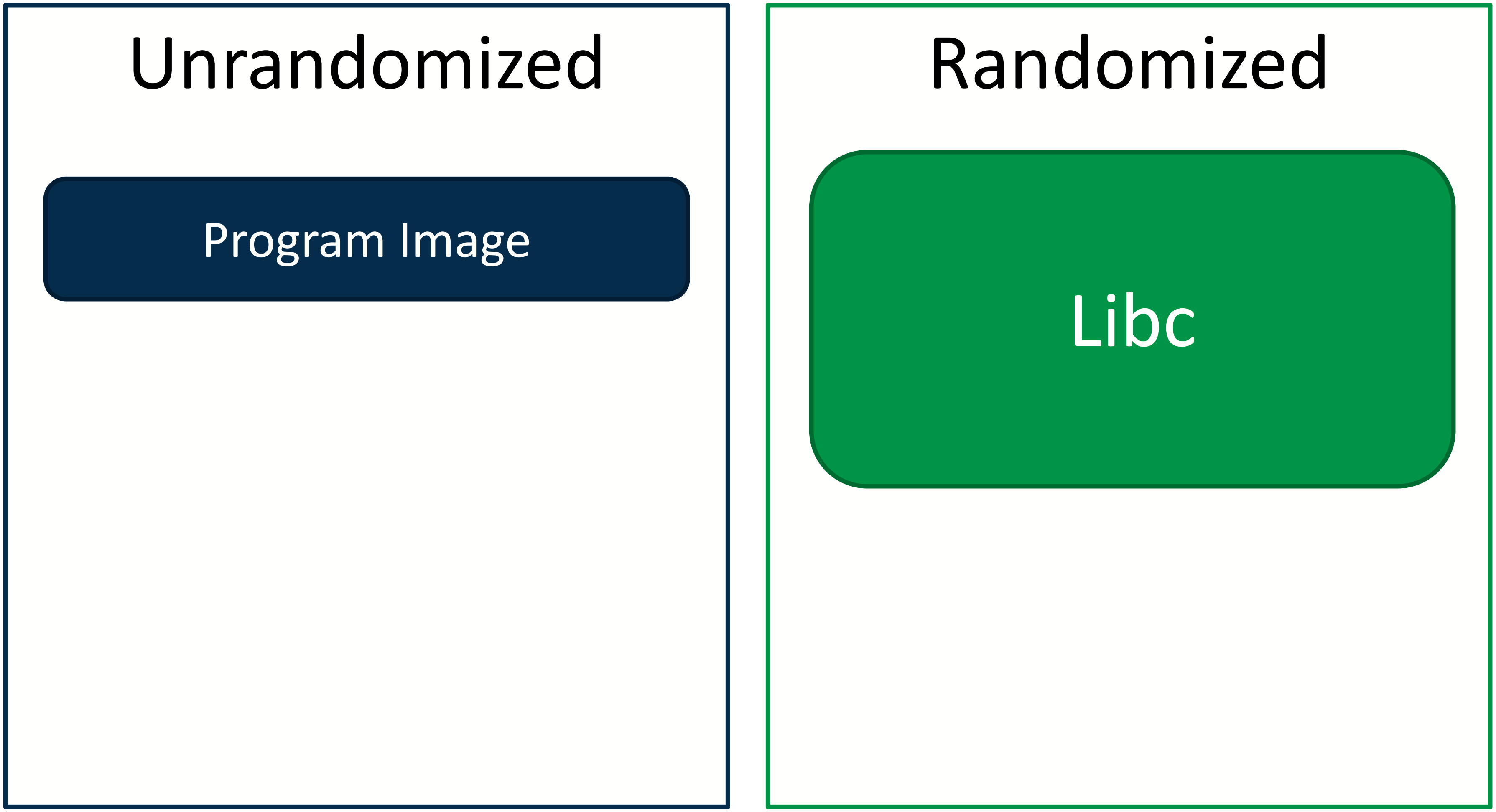
ROP address space: Linux

Unrandomized

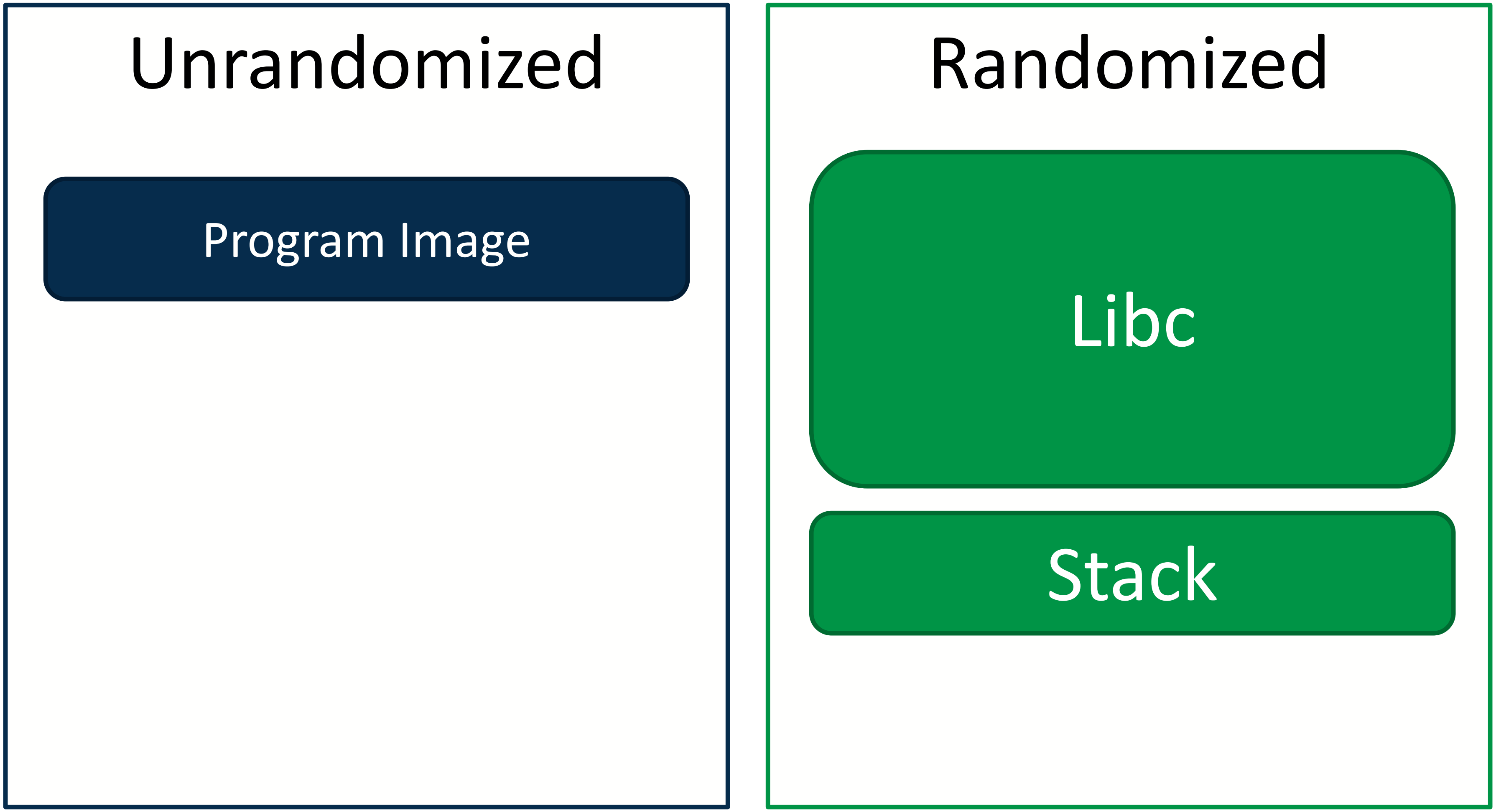
Randomized

ROP address space: Linux

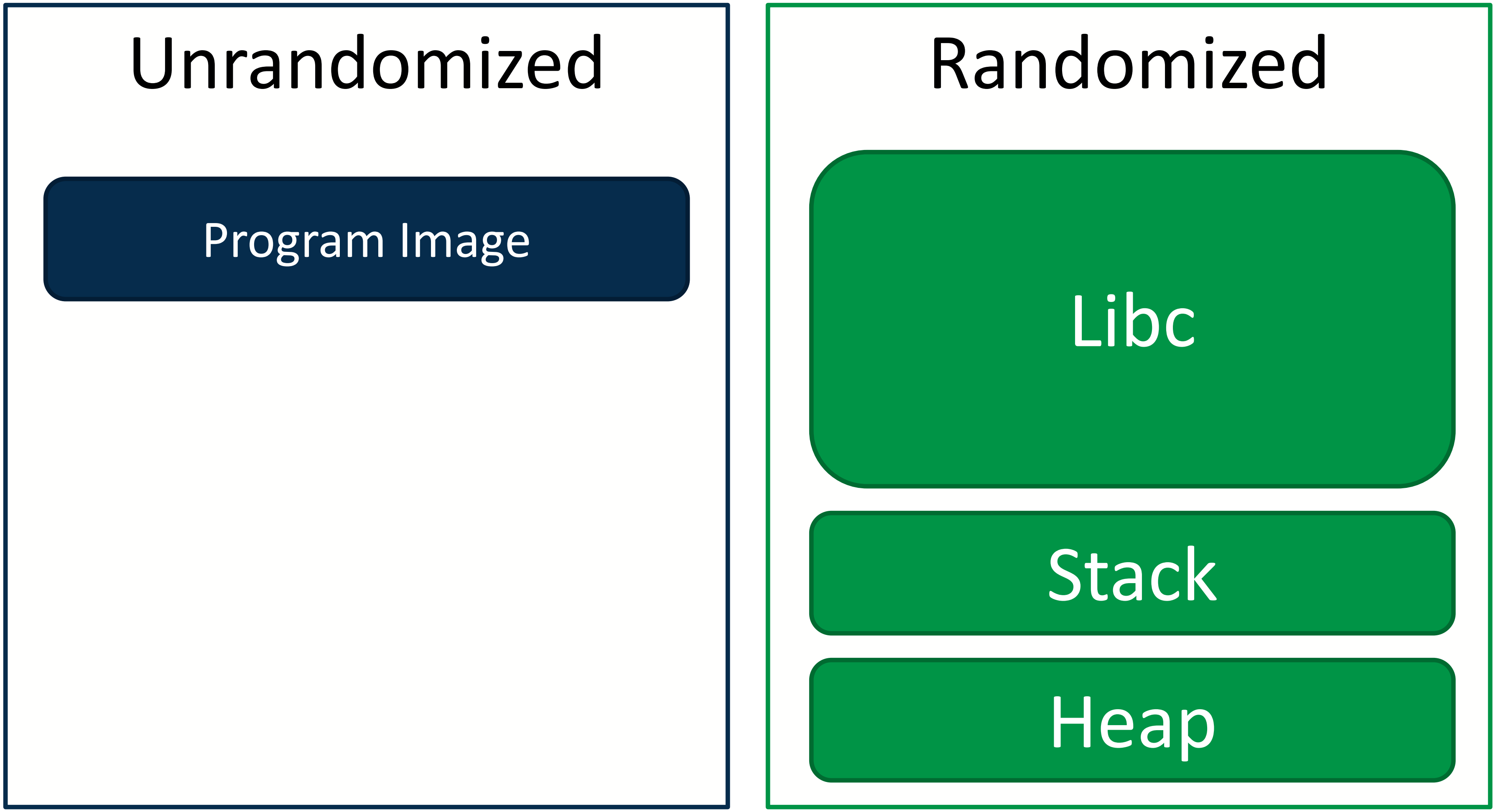
ROP address space: Linux



ROP address space: Linux



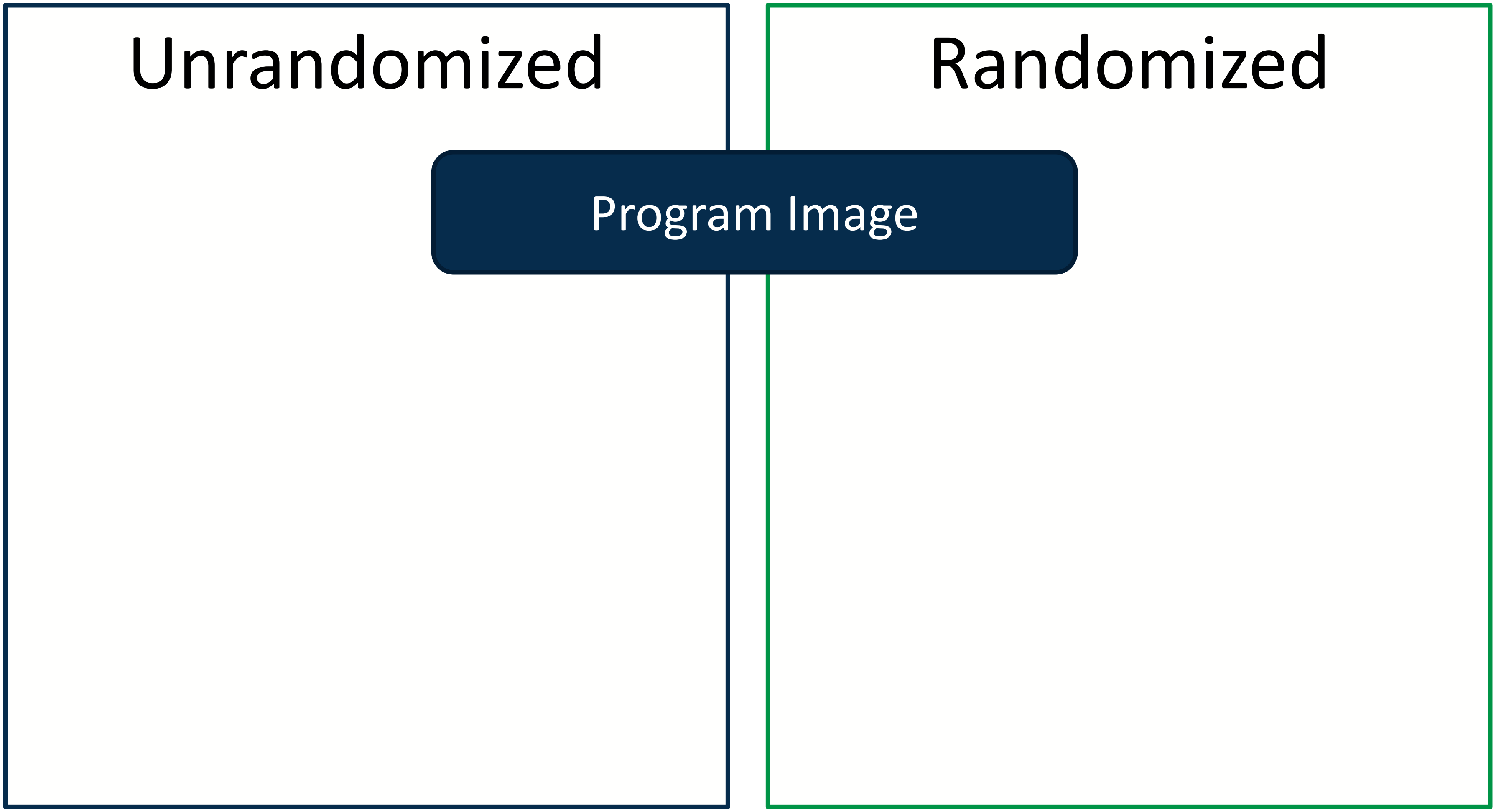
ROP address space: Linux



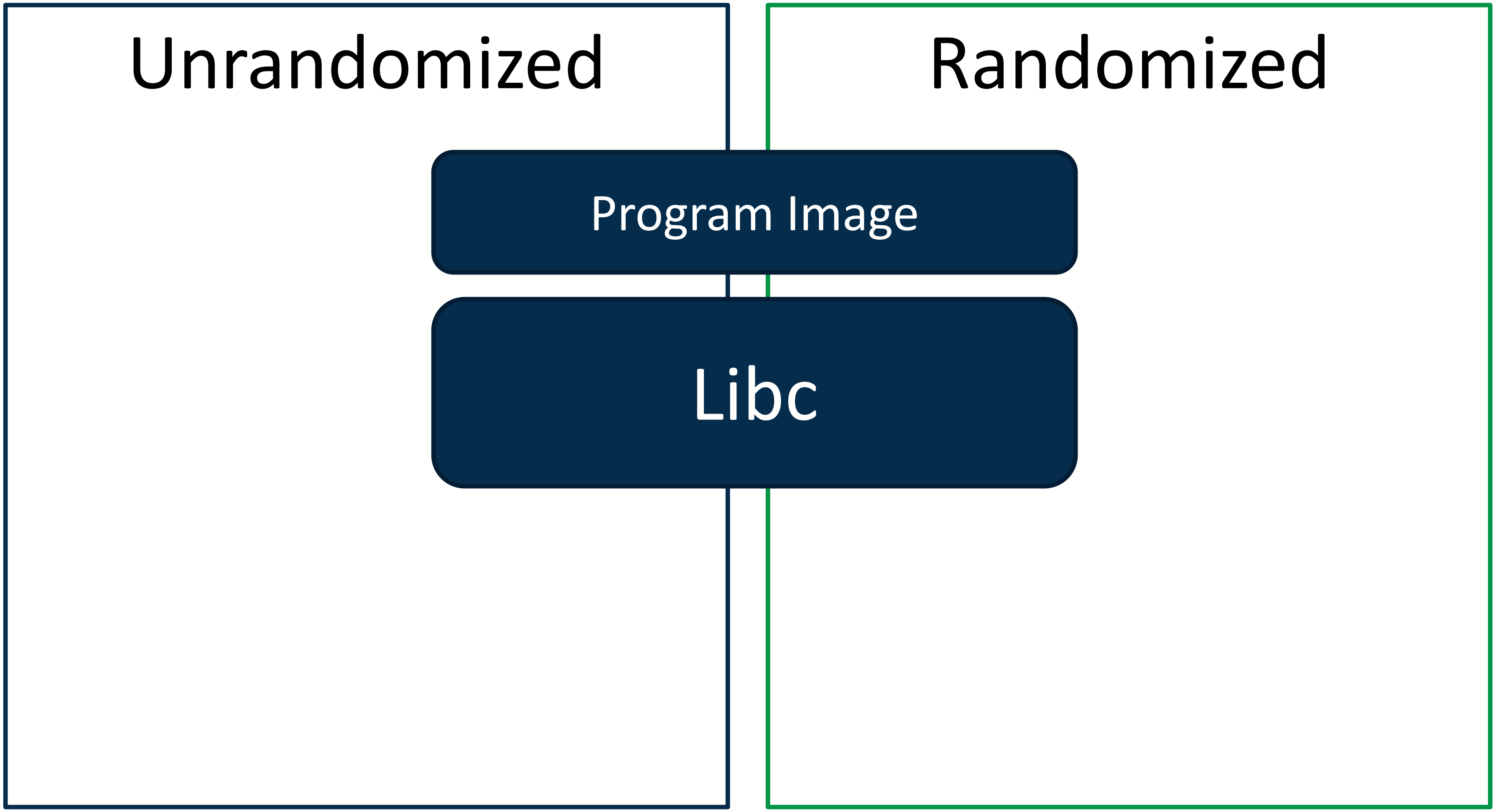
ROP address space: Windows

Unrandomized	Randomized

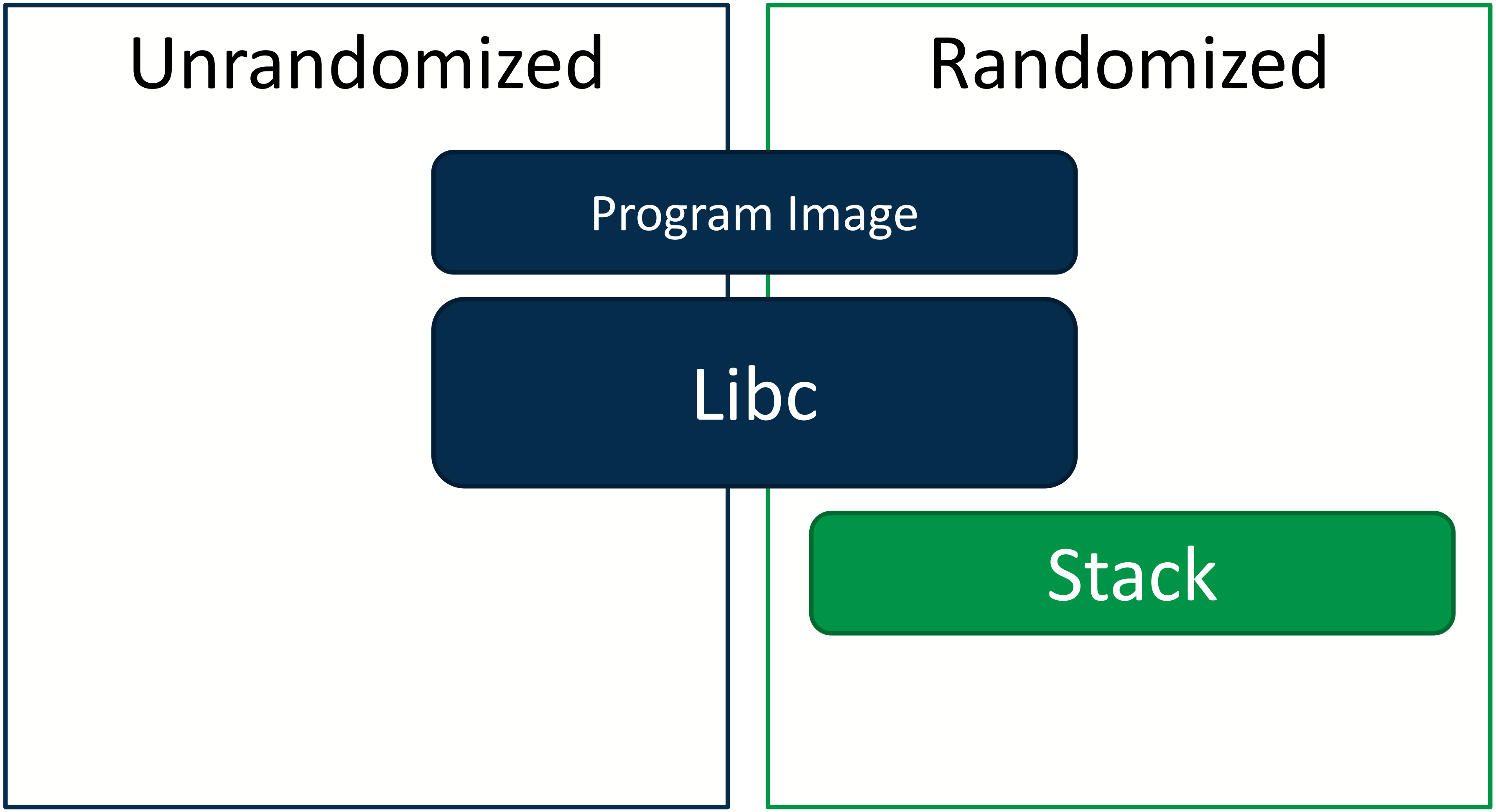
ROP address space: Windows



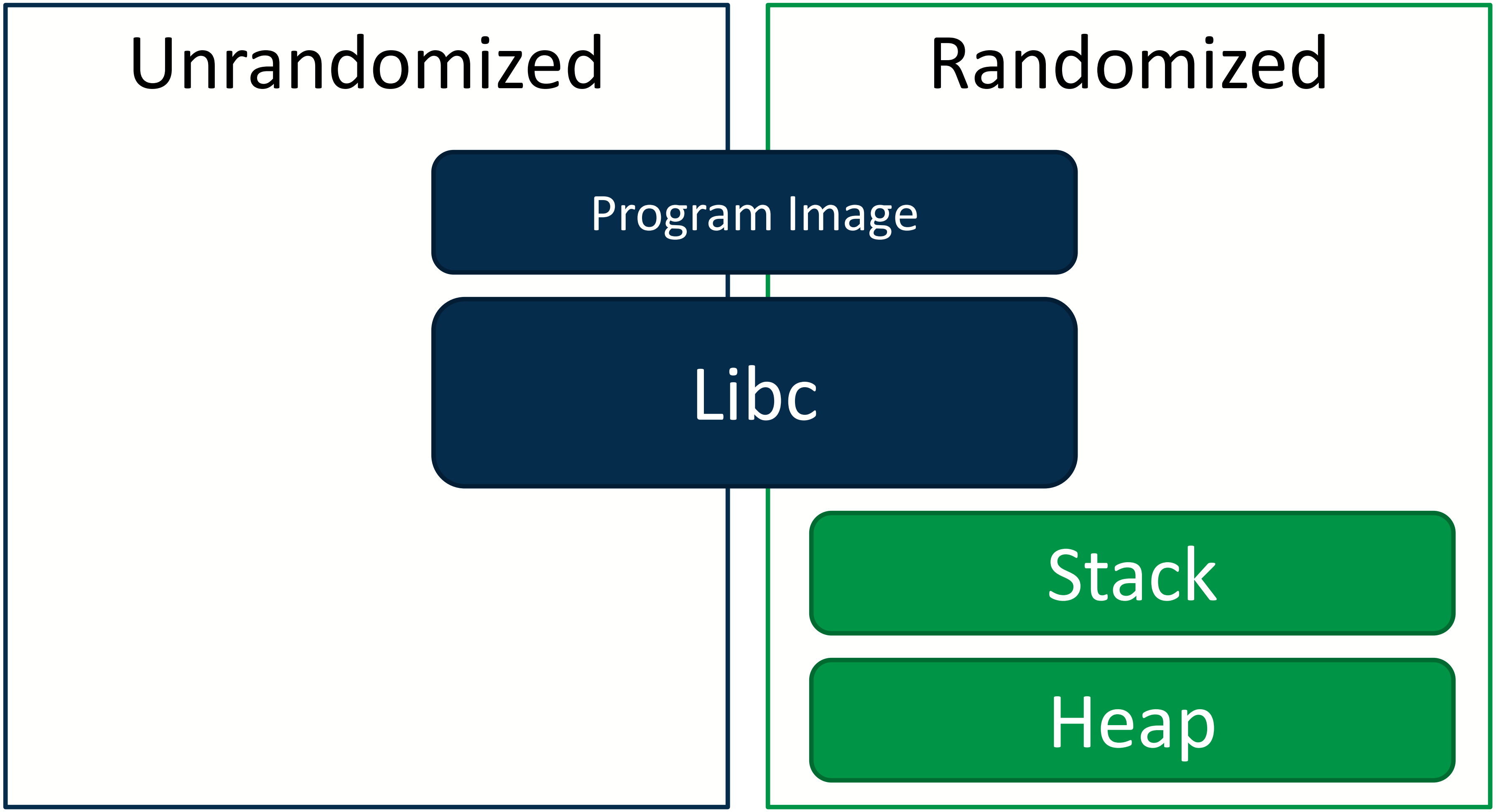
ROP address space: Windows



ROP address space: Windows



ROP address space: Windows



Agenda

ROP Overview

Gadgets

Disassembling code

Agenda

ROP Overview

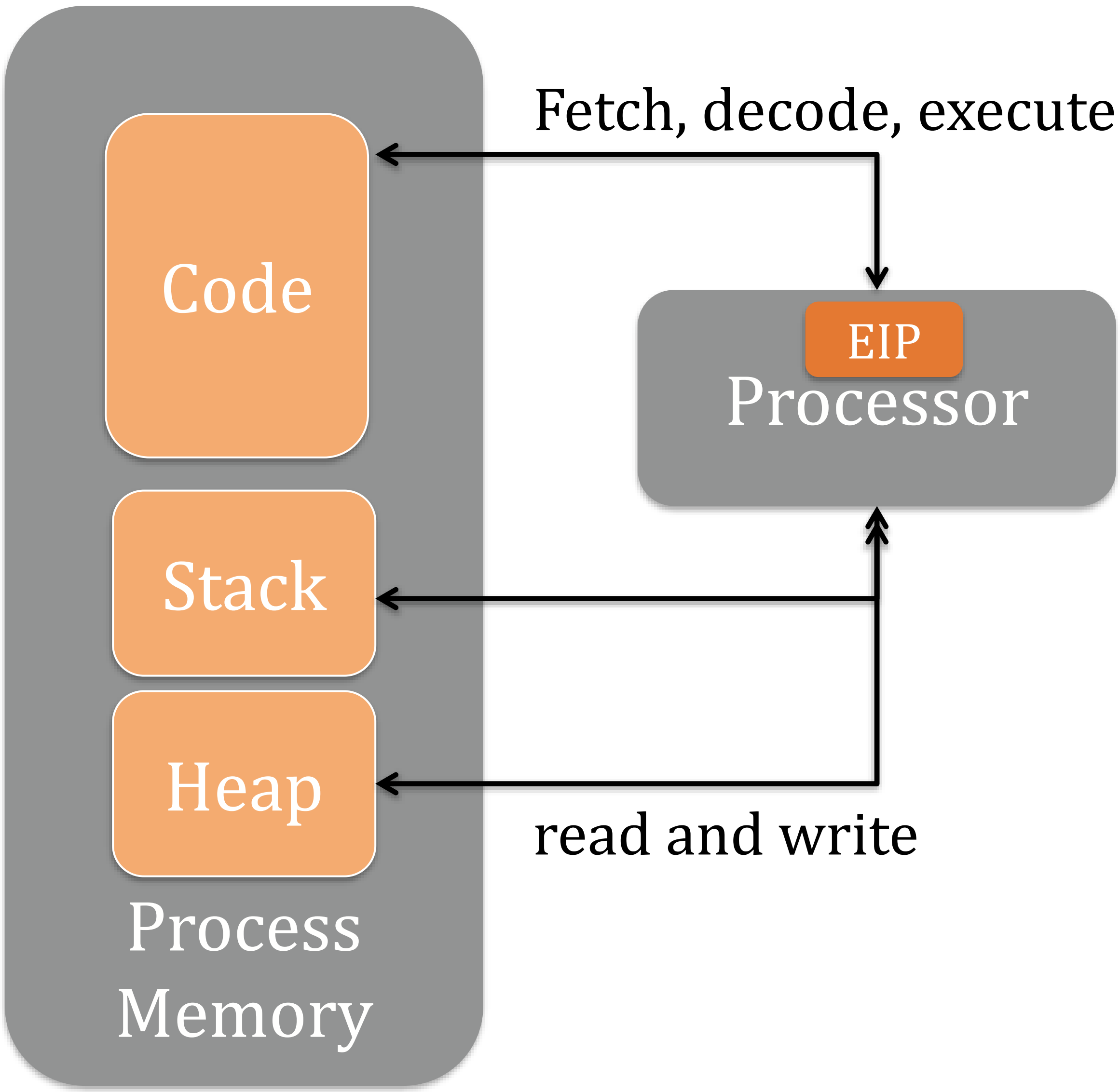
Gadgets

Disassembling code

NEXT

Or the ambiguity in decoding x86 instructions gives more potential to ROP!

Recall: Execution Model



Disassembly

```
user@box:~/l2$ objdump -d ./file
...
00000000 <even_sum>:
   0:   55          push   %ebp
   1:   89 e5       mov    %esp,%ebp
   3:   83 ec 10    sub    $0x10,%esp
   6:   8b 45 0c    mov    0xc(%ebp),%eax
   9:   03 45 08    add    0x8(%ebp),%eax
   c:   03 45 10    add    0x10(%ebp),%eax
   f:   89 45 fc    mov    %eax,0xfffffffc(%ebp)
  12:   8b 45 fc    mov    0xfffffffc(%ebp),%eax
  15:   83 e0 01    and    $0x1,%eax
  18:   84 c0       test   %al,%al
  1a:   74 03       je     1f <even_sum+0x1f>
  1c:   ff 45 fc    incl   0xfffffffc(%ebp)
  1f:   8b 45 fc    mov    0xfffffffc(%ebp),%eax
  22:   c9          leave
  23:   c3          ret
```

Disassemble

Address

Executable instructions

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 … 0xc9

Disassembler
EIP

Algorithm:

1. Decode Instruction
2. Advance EIP by len

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

Algorithm:

1. Decode Instruction
2. Advance EIP by len

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	Valid	Valid	Push *r/m16.*
FF /6	PUSH *r/m32*	N.E.	Valid	Push *r/m32.*
FF /6	PUSH *r/m64*	Valid	N.E.	Push *r/m64*. Default operand size 64-bits.
50+*rw*	PUSH *r16*	Valid	Valid	Push *r16.*
50+*rd*	PUSH *r32*	N.E.	Valid	Push *r32.*
50+*rd*	PUSH *r64*	Valid	N.E.	Push *r64*. Default operand size 64-bits.

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler EIP

Algorithm:

1. Decode Instruction
2. Advance EIP by len

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	Valid	Valid	Push *r/m16*.
FF /6	PUSH *r/m32*	N.E.	Valid	Push *r/m32*.
FF /6	PUSH *r/m64*	Valid	N.E.	Push *r/m64*. Default operand size 64-bits.
50+*rw*	PUSH *r16*	Valid	Valid	Push *r16*.
50+*rd*	PUSH *r32*	N.E.	Valid	Push *r32*.
50+*rd*	PUSH *r64*	Valid	N.E.	Push *r64*. Default operand size 64-bits.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler EIP

Algorithm:

1. Decode Instruction
2. Advance EIP by len

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	Valid	Valid	Push *r/m16*.
FF /6	PUSH *r/m32*	N.E.	Valid	Push *r/m32*.
FF /6	PUSH *r/m64*	Valid	N.E.	Push *r/m64*. Default operand size 64-bits.
50+*rw*	PUSH *r16*	Valid	Valid	Push *r16*.
50+*rd*	PUSH *r32*	N.E.	Valid	Push *r32*.
50+*rd*	PUSH *r64*	Valid	N.E.	Push *r64*. Default operand size 64-bits.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register			word register			dword register			quadword register (64-Bit Mode only)		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

```
push ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

```
push ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

MOV—Move

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8,r8	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8***,r8***	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	Valid	Valid	Move r32 to r/m32.

```
push ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler EIP

MOV—Move

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
88 /*r*	MOV *r/m8,r8*	Valid	Valid	Move *r8* to *r/m8.*
REX + 88 /*r*	MOV *r/m8****,*r8****	Valid	N.E.	Move *r8* to *r/m8.*
89 /*r*	MOV *r/m16,r16*	Valid	Valid	Move *r16* to *r/m16.*
89 /*r*	MOV *r/m32,r32*	Valid	Valid	Move *r32* to *r/m32.*

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
...										
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD

```
push ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

MOV—Move

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV r/m8,r8	Valid	Valid	Move r8 to r/m8.
REX + 88 /r	MOV r/m8***,r8***	Valid	N.E.	Move r8 to r/m8.
89 /r	MOV r/m16,r16	Valid	Valid	Move r16 to r/m16.
89 /r	MOV r/m32,r32	Valid	Valid	Move r32 to r/m32.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
...										
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD

```
push ebp
mov %esp, %ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

Algorithm:
1. Decode Instruction
2. Advance EIP by len

```
push ebp
mov %esp, %ebp
```

Linear-Sweep Disassembly

Executable Instructions

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassembler
EIP

Algorithm:
1. Decode Instruction
2. Advance EIP by len

```
push ebp
mov %esp, %ebp
```

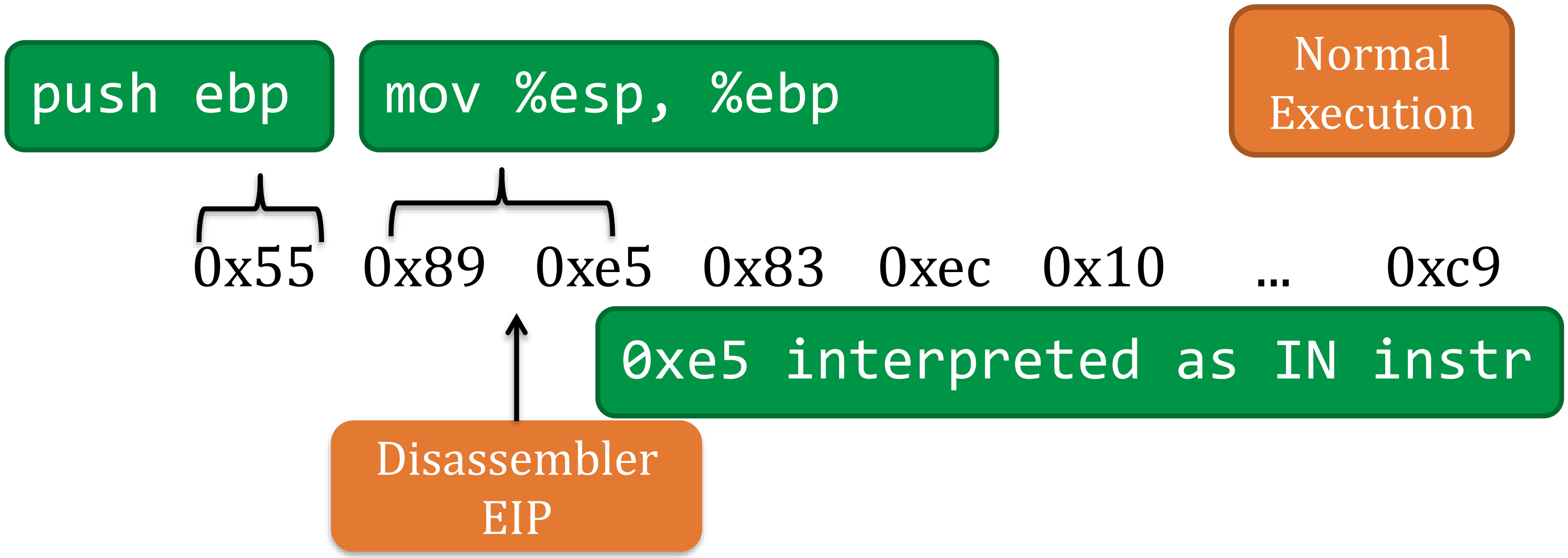
Disassemble from any address

`push ebp` `mov %esp, %ebp`

Normal Execution

0x55 0x89 0xe5 0x83 0xec 0x10 ... 0xc9

Disassemble from any address



It’s perfectly valid to start disassembling from *any* address.

All byte sequences will have a unique disassembly

Gadgets, Historically

`Mem[v2] = v1`

Semantics

a_3
v_2
a_2
v_1

a_1: `pop eax; ret`

...

a_3: `mov [ebx], eax`

...

a_2: `pop ebx; ret`

Gadgets

Gadgets, Historically

`Mem[v2] = v1`

Semantics

a_3
v_2
a_2
v_1

a_1: `pop eax; ret`

`...`

a_3: `mov [ebx], eax`

`...`

a_2: `pop ebx; ret`

Gadgets

- Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets

Gadgets, Historically

a_3
v_2
a_2
v_1

`Mem[v2] = v1`

Semantics

```
a1: pop eax; ret
...
a3: mov [ebx], eax
...
a2: pop ebx; ret
```

Gadgets

- Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets
- Common shellcode was created with these gadgets.

Gadgets, Historically

`Mem[v2] = v1`

Semantics

a_3
v_2
a_2
v_1

a_1: `pop eax; ret`

...

a_3: `mov [ebx], eax`

...

a_2: `pop ebx; ret`

Gadgets

- Shacham et al. manually identified which sequences ending in ret in libc were useful gadgets
- Common shellcode was created with these gadgets.
- Everyone used libc, so gadgets and shellcode universal

Recap: ROP [Shacham et al.]

1. Disassemble code
2. Identify *useful* code sequences as gadgets ending in ret
3. Assemble gadgets into desired shellcode

Agenda

ROP Overview

Gadgets

Disassembling code

Agenda

ROP Overview

Gadgets

Disassembling code

Looking ahead

- Still need to beat ASLR
- What about remote attacks?

Blind ROP

- Hacking Blind -- 2014
- "It is possible to write remote stack buffer overflow exploits without possessing a copy of the target binary or source code, against services that restart after a crash.
- This makes it possible to hack proprietary closed-binary services, or open-source servers manually compiled and installed from source where the binary remains unknown to the attacker"

Agenda

ROP Overview ✔

Gadgets ✔

Disassembling Code ✔

Hacking Blind: BROP ✔

Questions