Early Defenses and More Attacks

CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 2018

Recap

Vulnerabilities

- Heap overflows can be used to perform arbitrary writes
- Format string vulnerabilities can be used to leak memory and perform arbitrary writes

Defenses

- Stack canaries/cookies can be used to detect stack smashing
- Compilers and libraries (libsafe) can add some basic boundary checking to dangerous functions (strcpy, memcpy, etc.)
- Non-executable data regions prevent code injection
  - Strive for Write-XOR-Execute in programs

Modern attacks: return-to-libc

Topics

Stack overflow defenses

- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses

- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses

Topics

Stack overflow defenses

- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses

- Non executable stack (and heap)
- **Early code-reuse attacks/return-to-libc**
- ASCII armored space

ASLR and bypasses

Return-to Attacks

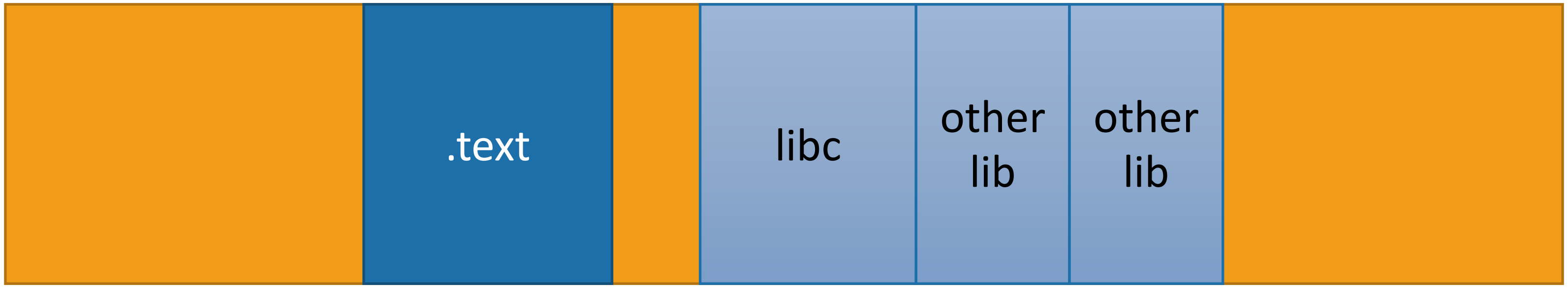
What can I do if I control the return address when I cannot inject code?

Return-to Attacks

What can I do if I control the return address when I cannot inject code?

Return to an existing function (e.g., a libc function)

Process



Application code

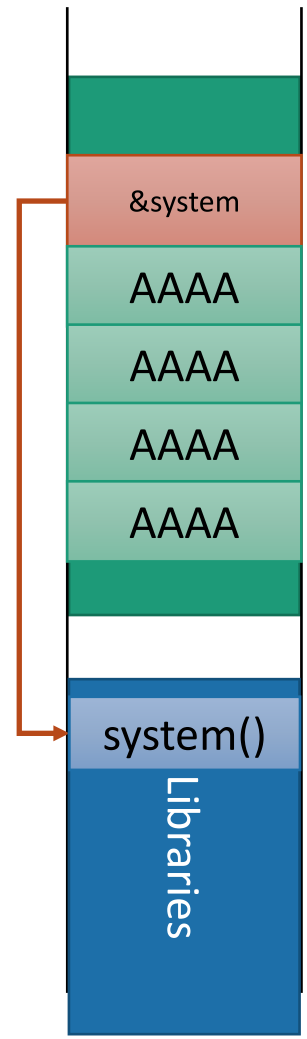
C library
(defines system call wrappers, memory management routines, and other basic facilities)

```
$ ldd /bin/ls
	linux-vdso.so.1 (0x00007ffc83b62000)
	libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edfdf1000)
	libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edfbe8000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)
	libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edf5cf000)
	libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf3cb000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f9ee0016000)
	libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf1c6000)
	libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edefa9000)
```

Return-to-libc (ret2libc) on 32-bits

Replace return address with the address of an **existing** function

Example: system() executes an a program in a new process



Shell Using ret2libc

Locate system libc call

- *int system(const char *command);*

Set return address to the address of *system()*

```
$ readelf -s /lib/i386-linux-gnu/libc-2.19.so |grep system
1442: 0003de80   56 FUNC   WEAK DEFAULT 12 system@@GLIBC_2.0
```

Prepare one argument for system()

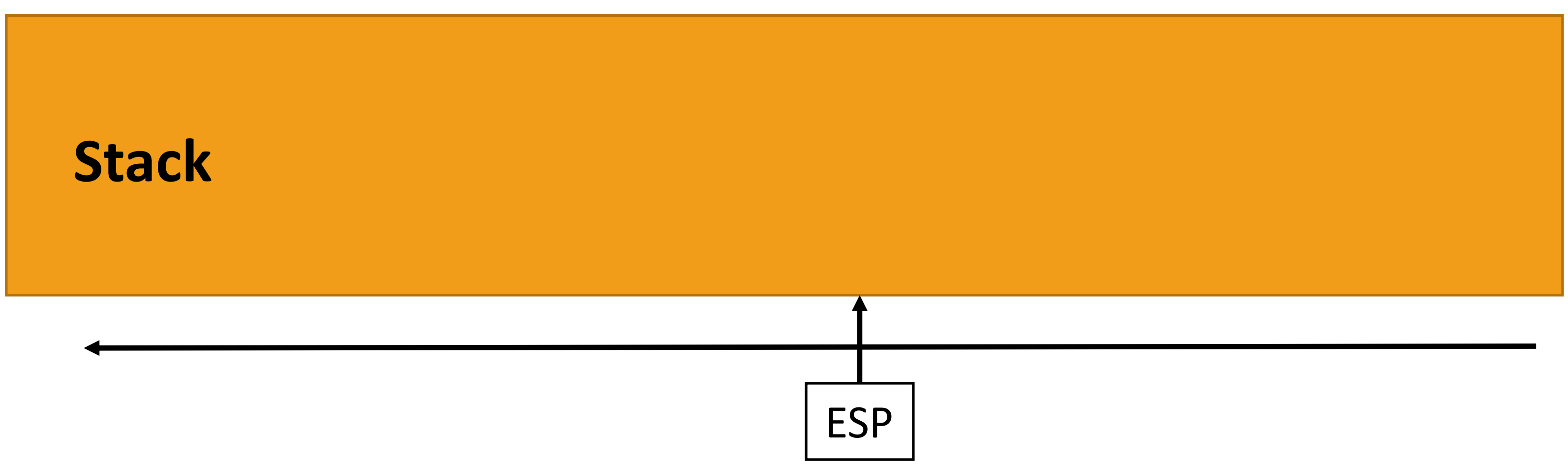
```c
int main(void)
{
        system("/bin/shell");
        return 0;
}
```

```
080483fb <main>:
 80483fb:       8d 4c 24 04             lea    0x4(%esp),%ecx
 80483ff:       83 e4 f0                and    $0xfffffff0,%esp
 8048402:       ff 71 fc                pushl  -0x4(%ecx)
 8048405:       55                      push   %ebp
 8048406:       89 e5                   mov    %esp,%ebp
 8048408:       51                      push   %ecx
 8048409:       83 ec 04                sub    $0x4,%esp
 804840c:       83 ec 0c                sub    $0xc,%esp
 804840f:       68 c0 84 04 08          push   $0x80484c0
 8048414:       e8 b7 fe ff ff          call   80482d0 <system@plt>
...
```

Preparing the Stack

EIP

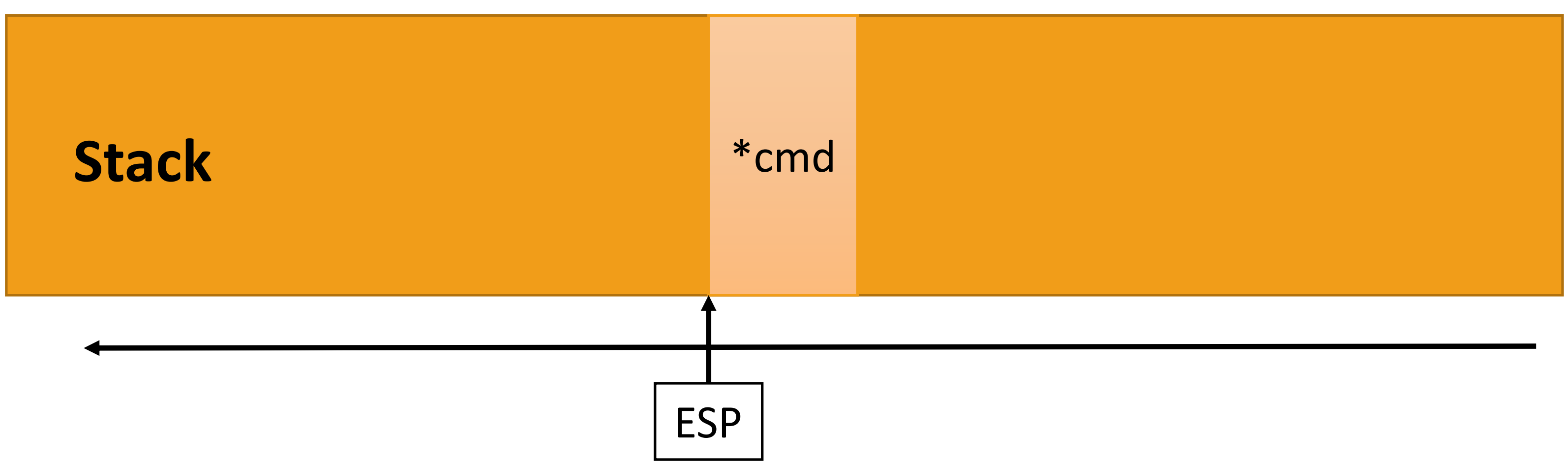
```
804840f:    68 c0 84 04 08    push   $0x80484c0
8048414:    e8 b7 fe ff ff    call   80482d0 <system@plt>
```



Preparing the Stack

EIP →

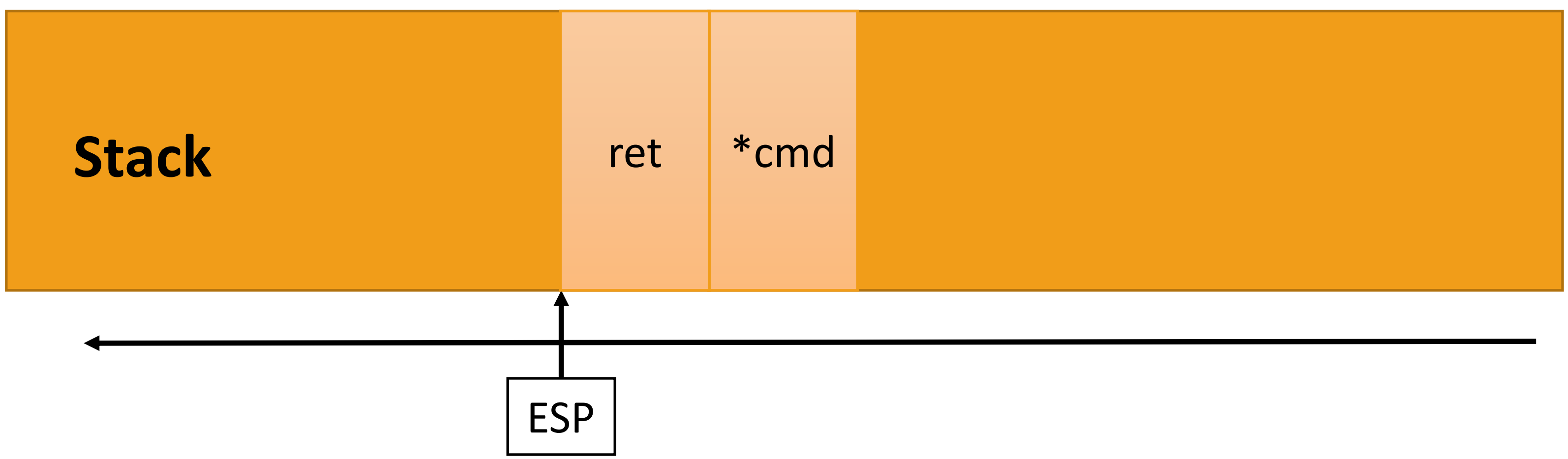
```
804840f:     68 c0 84 04 08     push   $0x80484c0
8048414:     e8 b7 fe ff ff     call   80482d0 <system@plt>
```



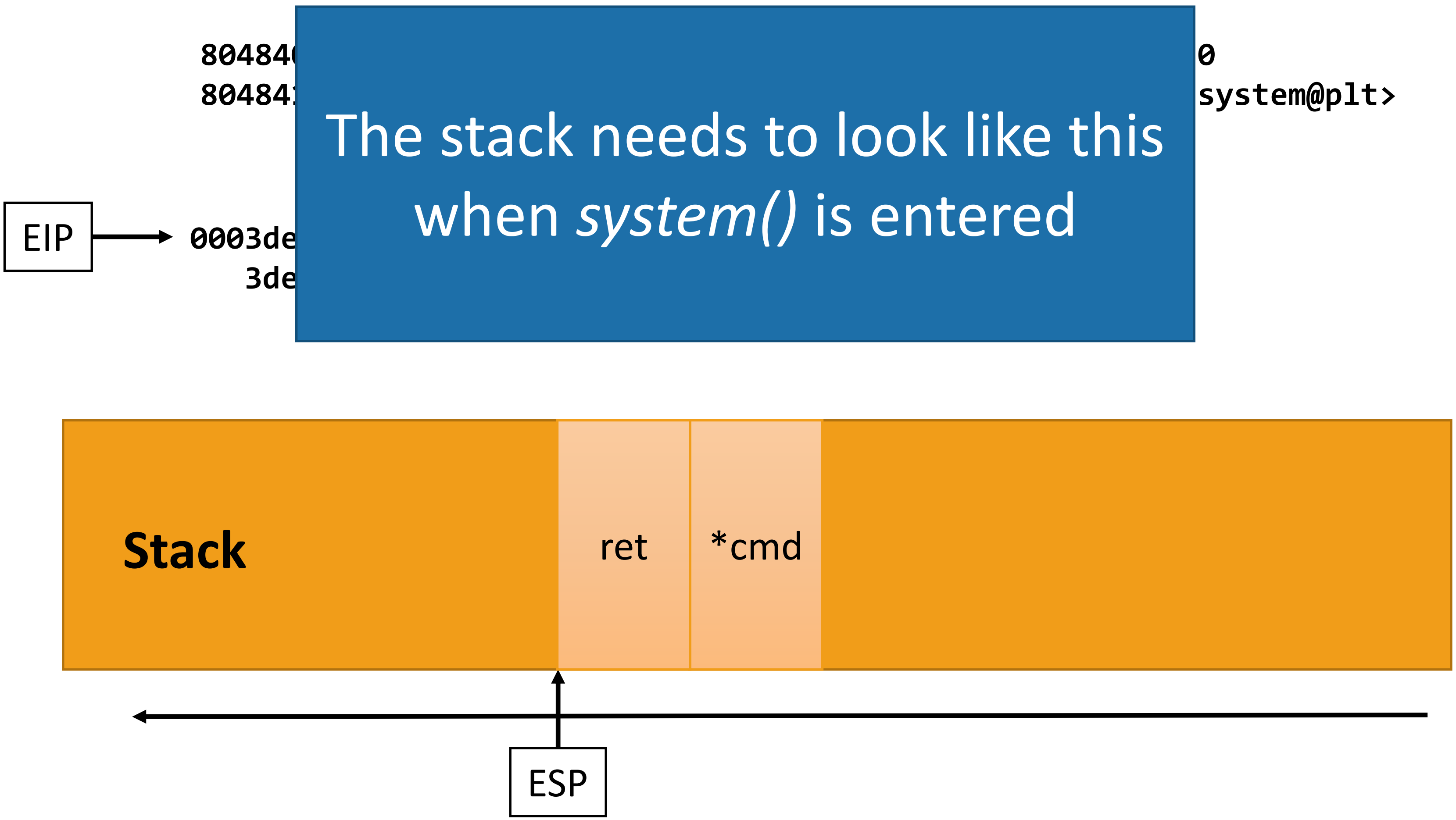
Preparing the Stack

```
804840f:      68 c0 84 04 08            push   $0x80484c0
8048414:      e8 b7 fe ff ff            call   80482d0 <system@plt>
```

```
EIP ──► 0003de80 <__libc_system>:
           3de80:       53                       push   %ebx
```

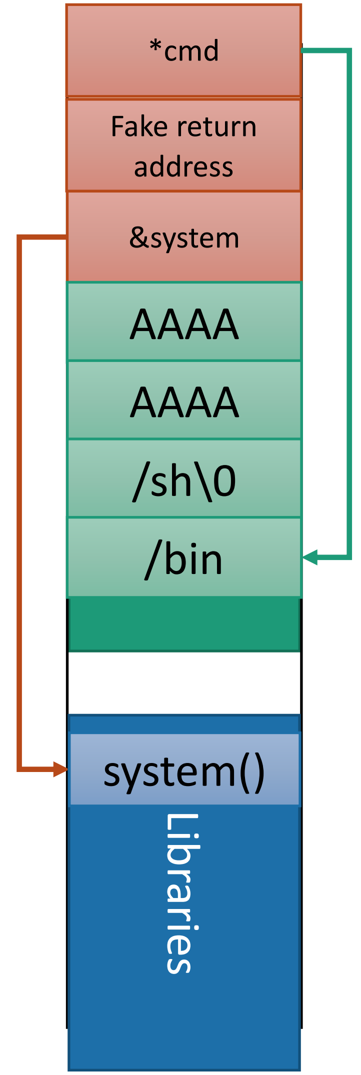


Preparing the Stack



Preparing the Stack

Add a fake return address and a pointer to the command we want to execute on the stack



Return-to-libc on 64-bits

Arguments are passed using registers

- First 6 integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9

RBP, RBX, and R12–R15 are callee saved

RAX used for function return

```c
int main(void)
{
        system("/bin/shell");
        return 0;
}
```

```
0000000000400506 <main>:
  400506:       55                      push   %rbp
  400507:       48 89 e5                mov    %rsp,%rbp
  40050a:       bf a4 05 40 00          mov    $0x4005a4,%edi
  40050f:       e8 cc fe ff ff          callq  4003e0 <system@plt>
...
```

How to load an argument to a register (e.g., rdi)?

Code-reuse Attacks

Any code that already exists in the process can be executed

For example, the following sequence

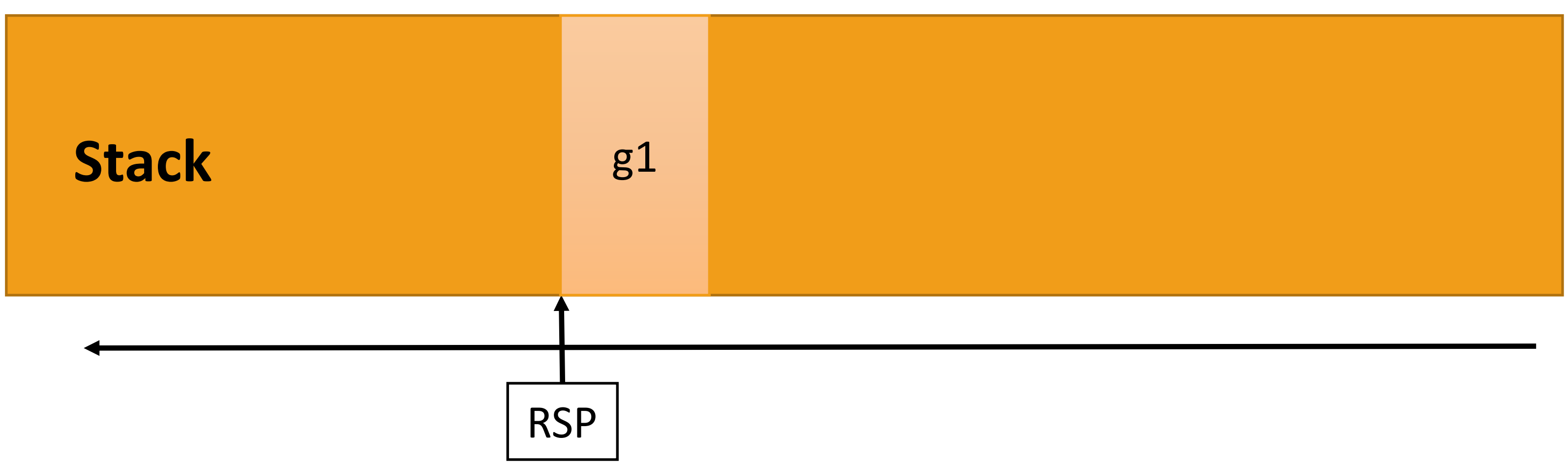
```
0x0000000000405255 : pop rdi ; ret
```

Such short instructions sequences are referred to as **gadgets**

Return-to-libc on 64-bit

Redirect control to gadget

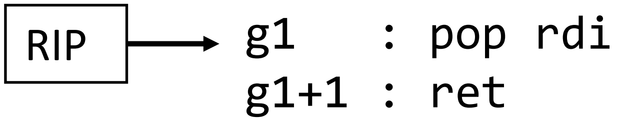
```
g1   : pop rdi
g1+1 : ret
```

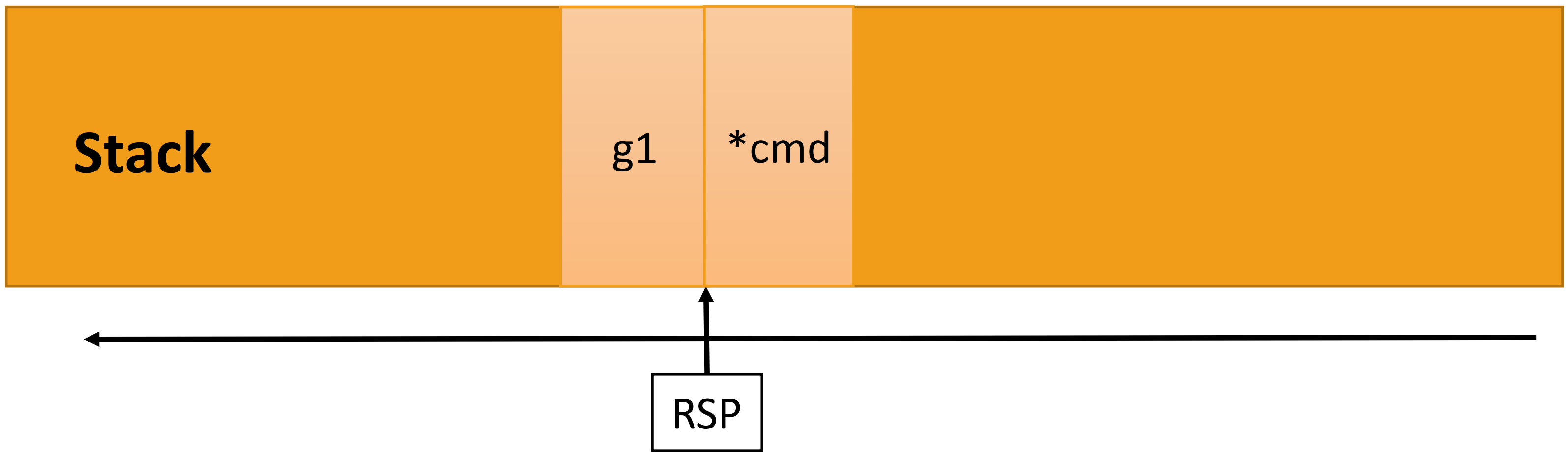


Return-to-libc on 64-bit

Redirect control to gadget

Load argument on register



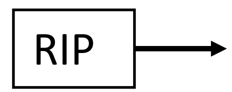


Return-to-libc on 64-bit

Redirect control to gadget

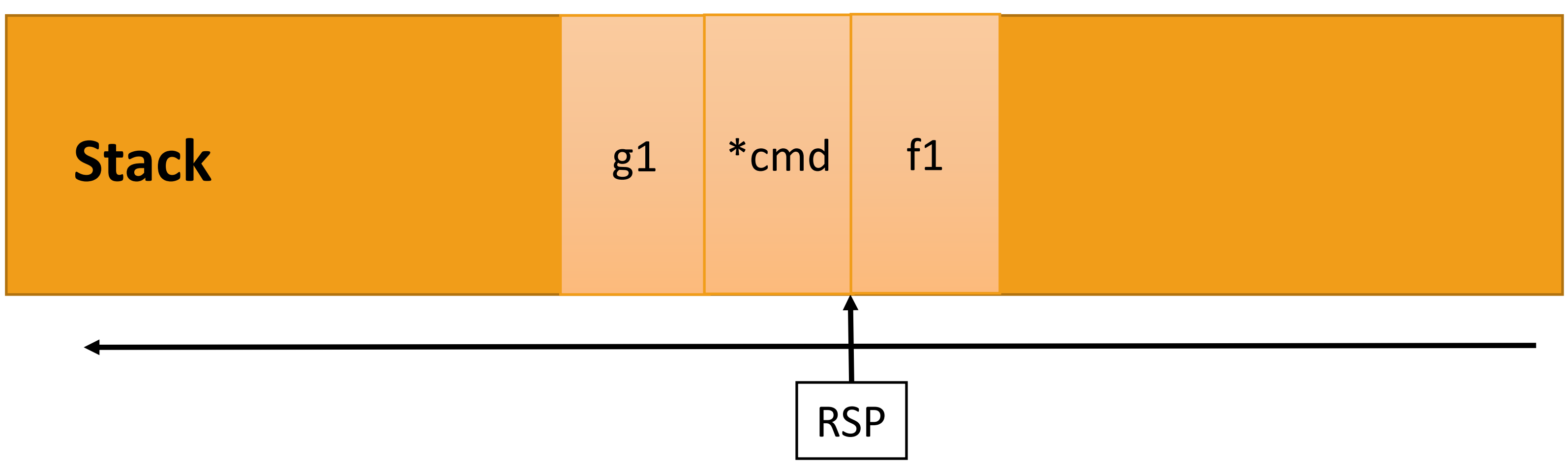
Load argument on register

Redirect control to libc function




```
g1   : pop rdi
g1+1 : ret
```

```
f1 <__libc_system>:
f1 : push rbp
```



Return-to-libc on 64-bit

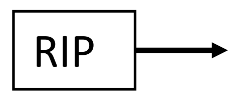
Redirect control to gadget

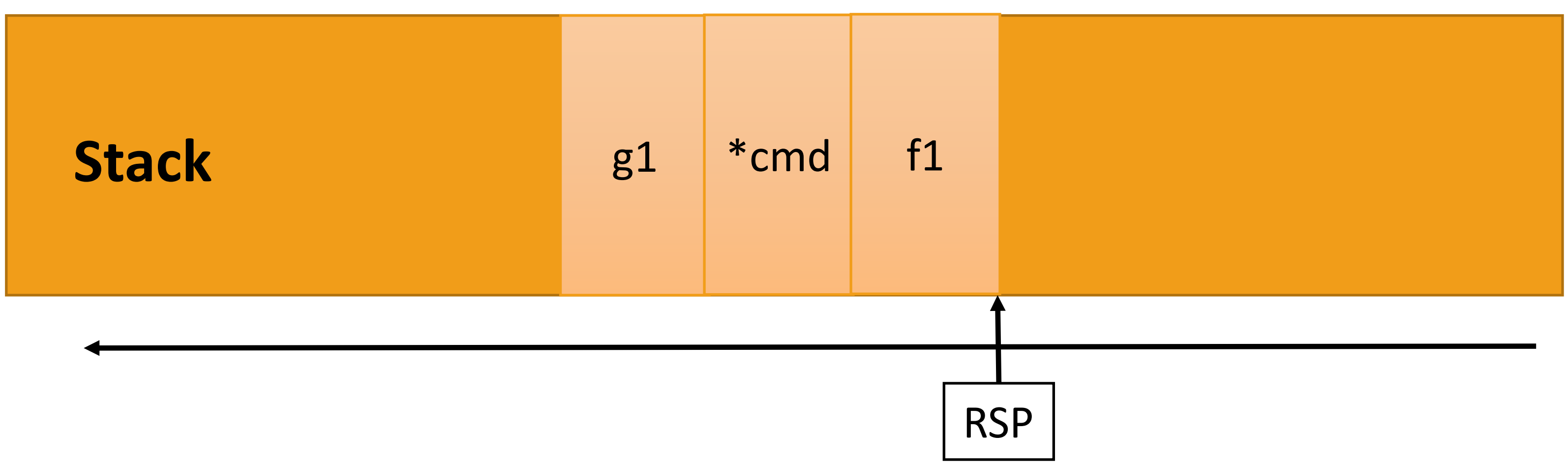
Load argument on register

Redirect control to libc function

```
g1   : pop rdi
g1+1 : ret
```

```
f1 <__libc_system>:
f1 : push rbp
```





Return-to-libc on 64-bit

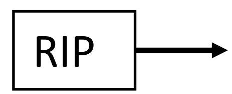
Redirect control to gadget

Load argument on register

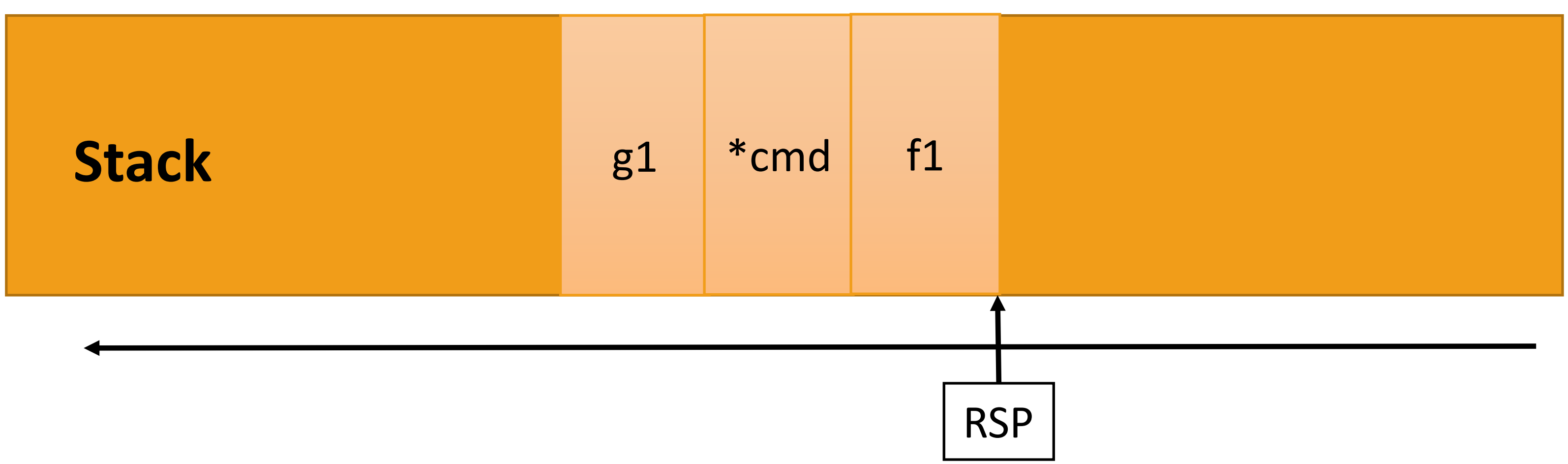
Redirect control to libc function

Get shell!!

```
g1   : pop rdi
g1+1 : ret
```




```
f1 <__libc_system>:
f1 : push rbp
```



Topics

Stack overflow defenses

- Stackguard & Stackshield
- Boundary checking

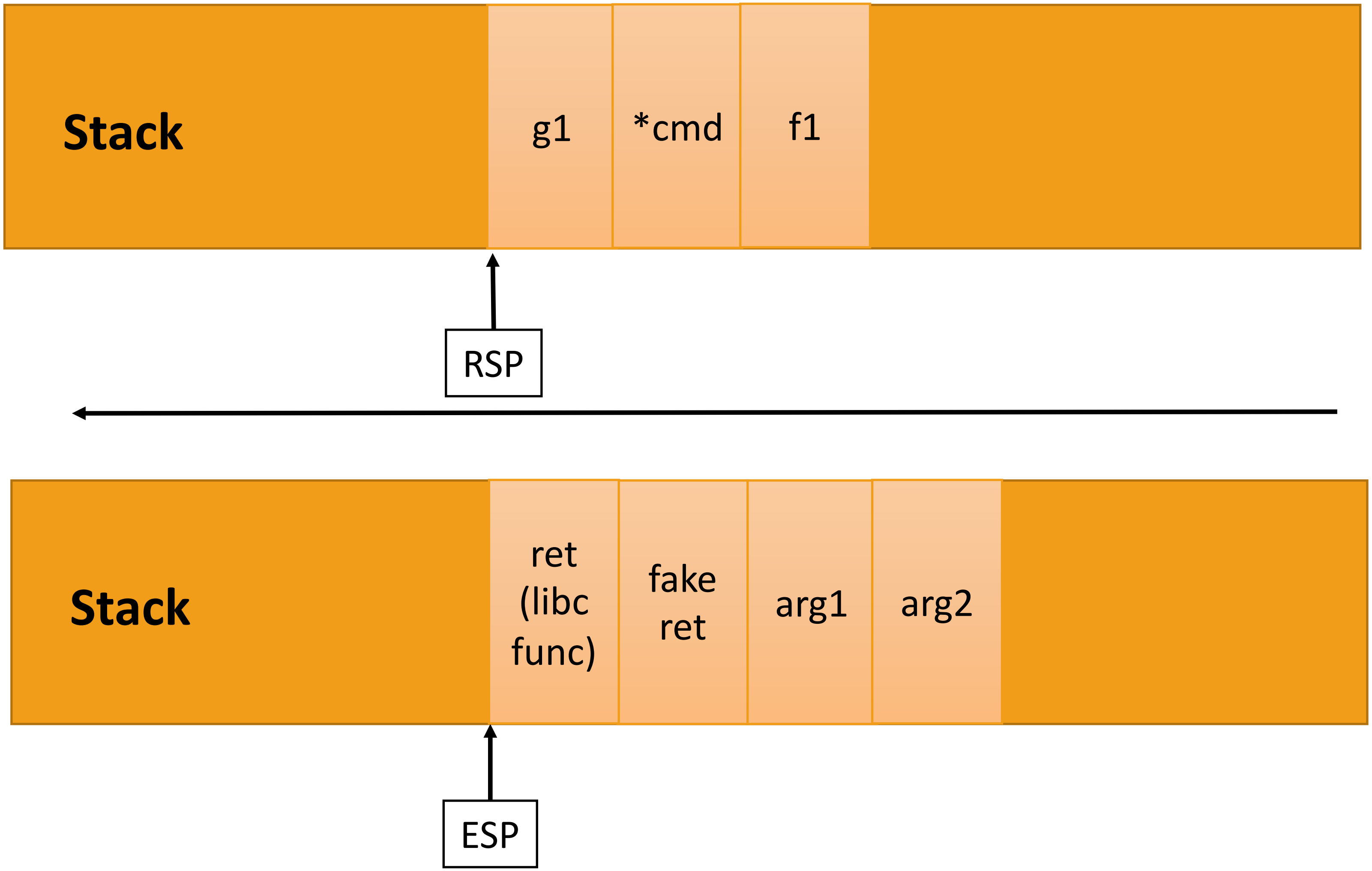
Heap corruption defenses

Code-injection defenses and bypasses

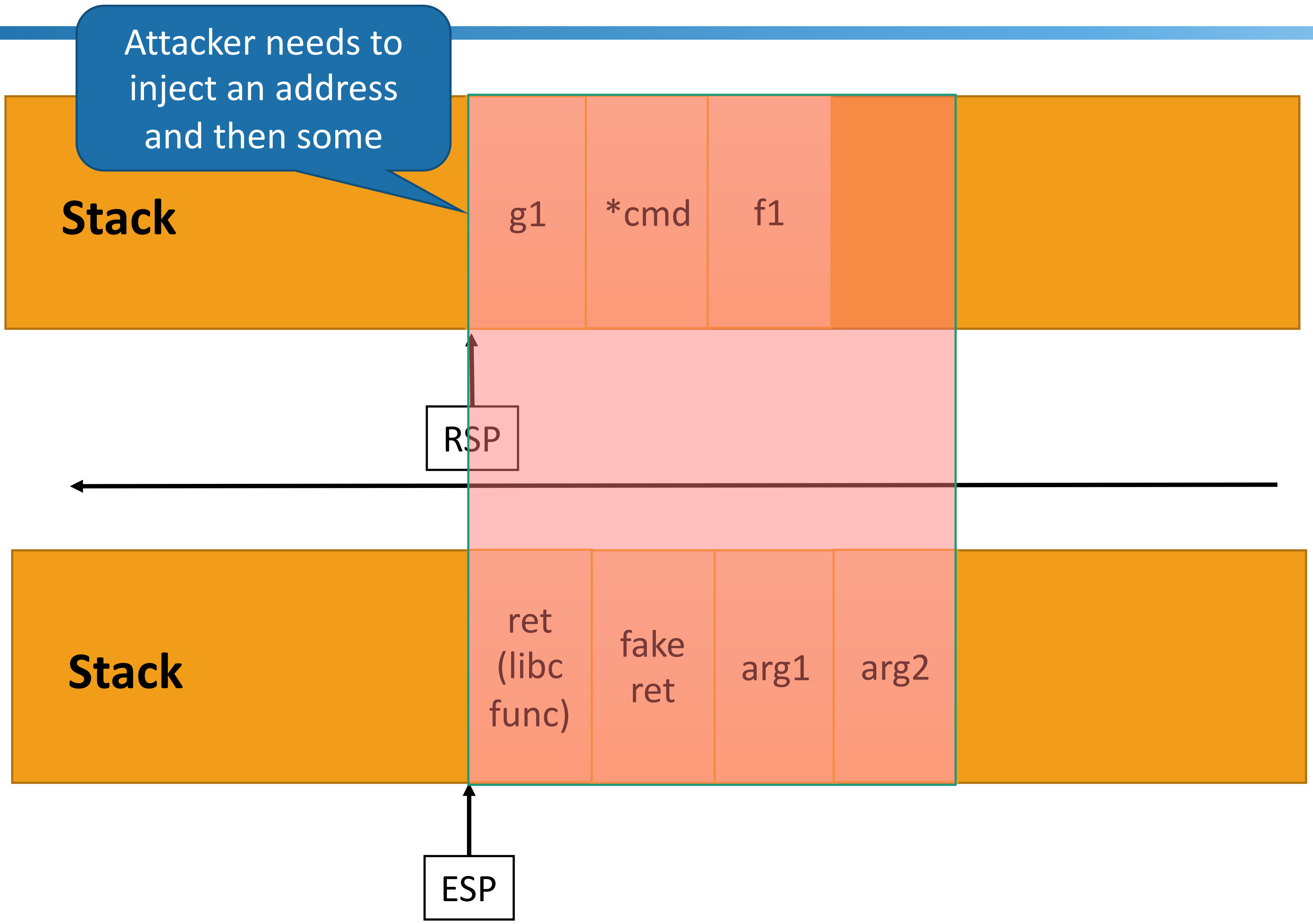
- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- **ASCII armored space**

ASLR and bypasses

ASCII Armored Address Space

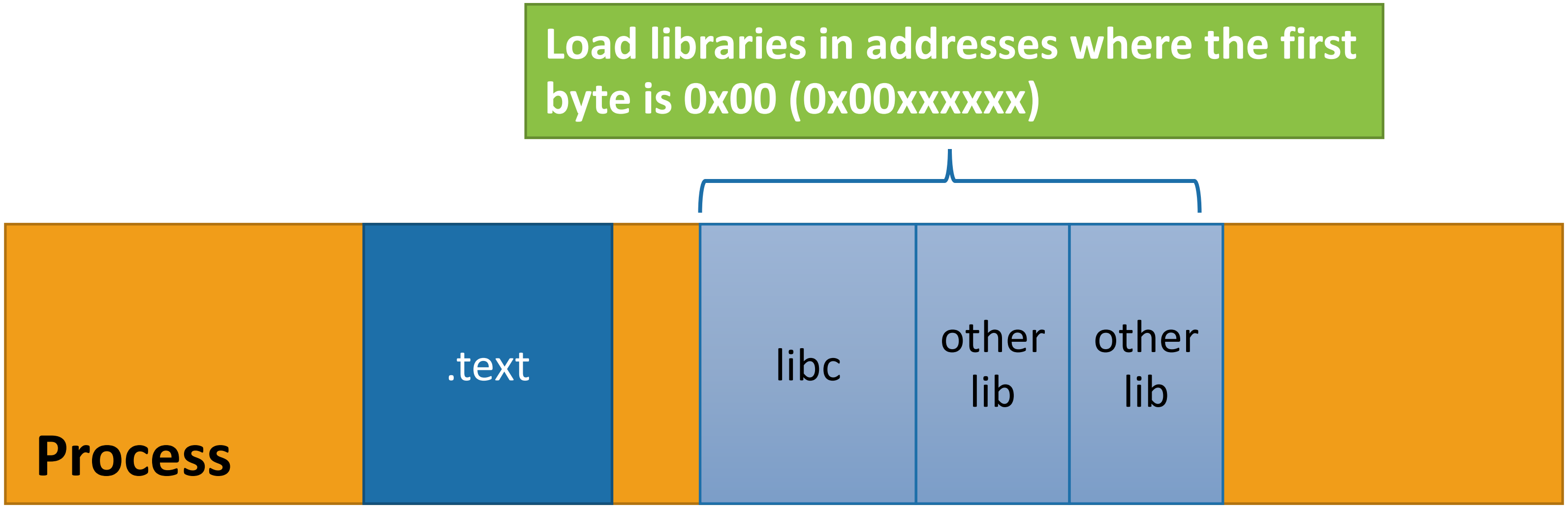


ASCII Armored Address Space

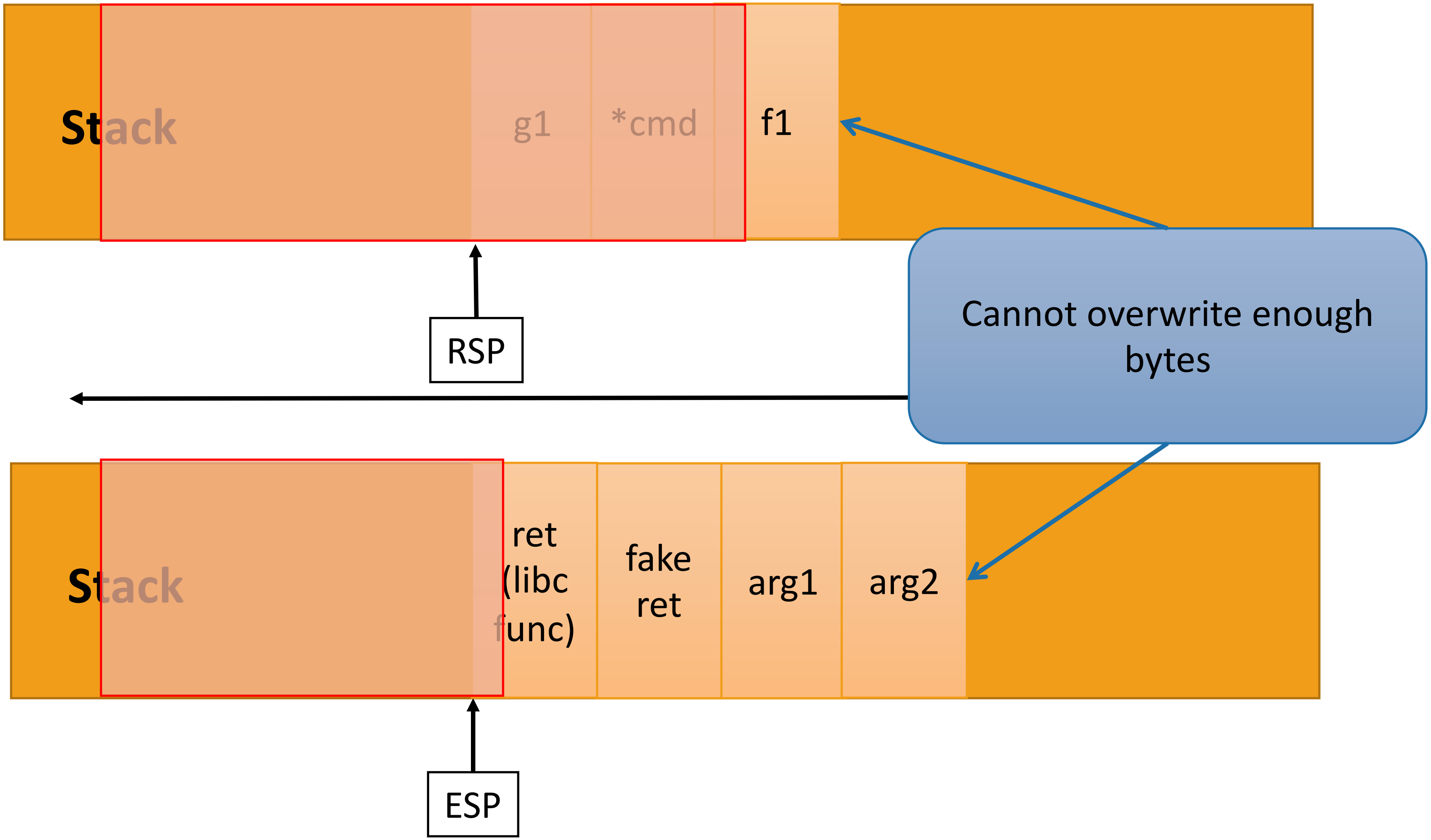


ASCII Armored Address Space

Observation: strcpy() stops copying on the first null byte!



ASCII Armored Address Space



Problems

Other methods of copying data may not have the same limitation: memcpy(), gets(), read(), fread(), custom copy routines, etc.

Topics

Stack overflow defenses

- Stackguard & Stackshield
- Boundary checking

Heap corruption defenses

Code-injection defenses and bypasses

- Non executable stack (and heap)
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses

Fixed Process Layout

Layout is fixed across all instances of a specific system version → ret2libc attack are possible

```
$ ldd /bin/ls
         linux-vdso.so.1 (0x00007ffc83b62000)
         libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edfdf1000)
         libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edfbe8000)
         libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)
         libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edf5cf000)
         libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf3cb000)
         /lib64/ld-linux-x86-64.so.2 (0x00007f9ee0016000)
         libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf1c6000)
         libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edefa9000)
```

One Attack Fits All

Layout is fixed across all instances of a specific system version → ret2libc attack are possible

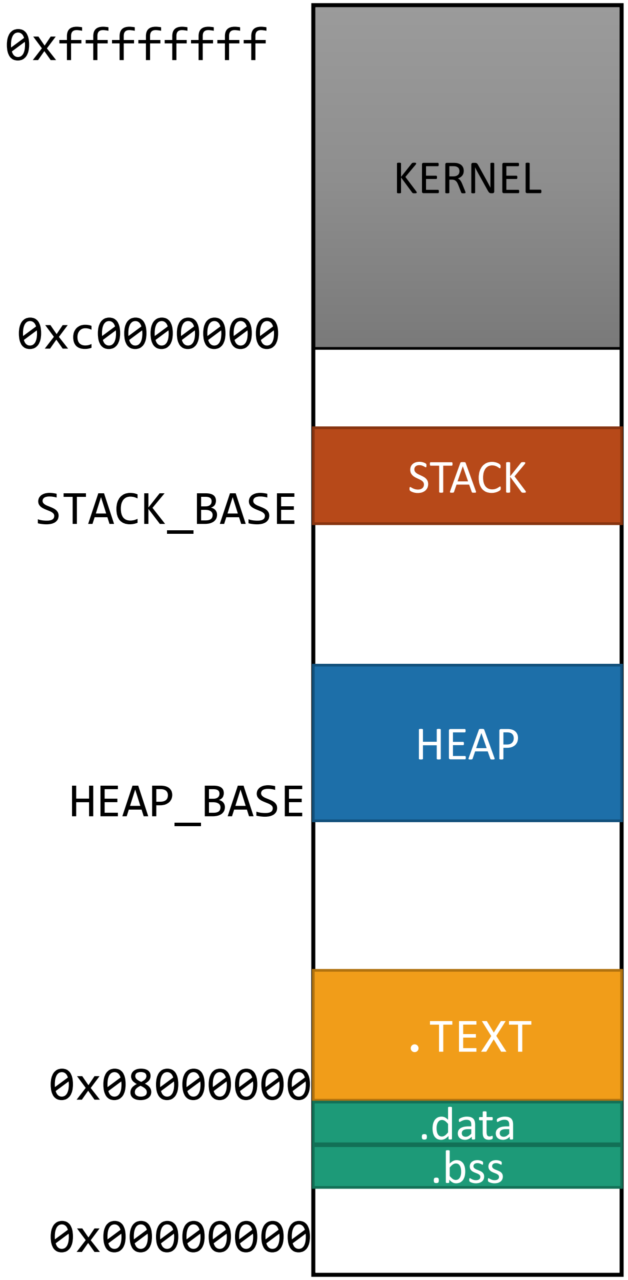
An exploit developed on one system will work on all other systems running the same software

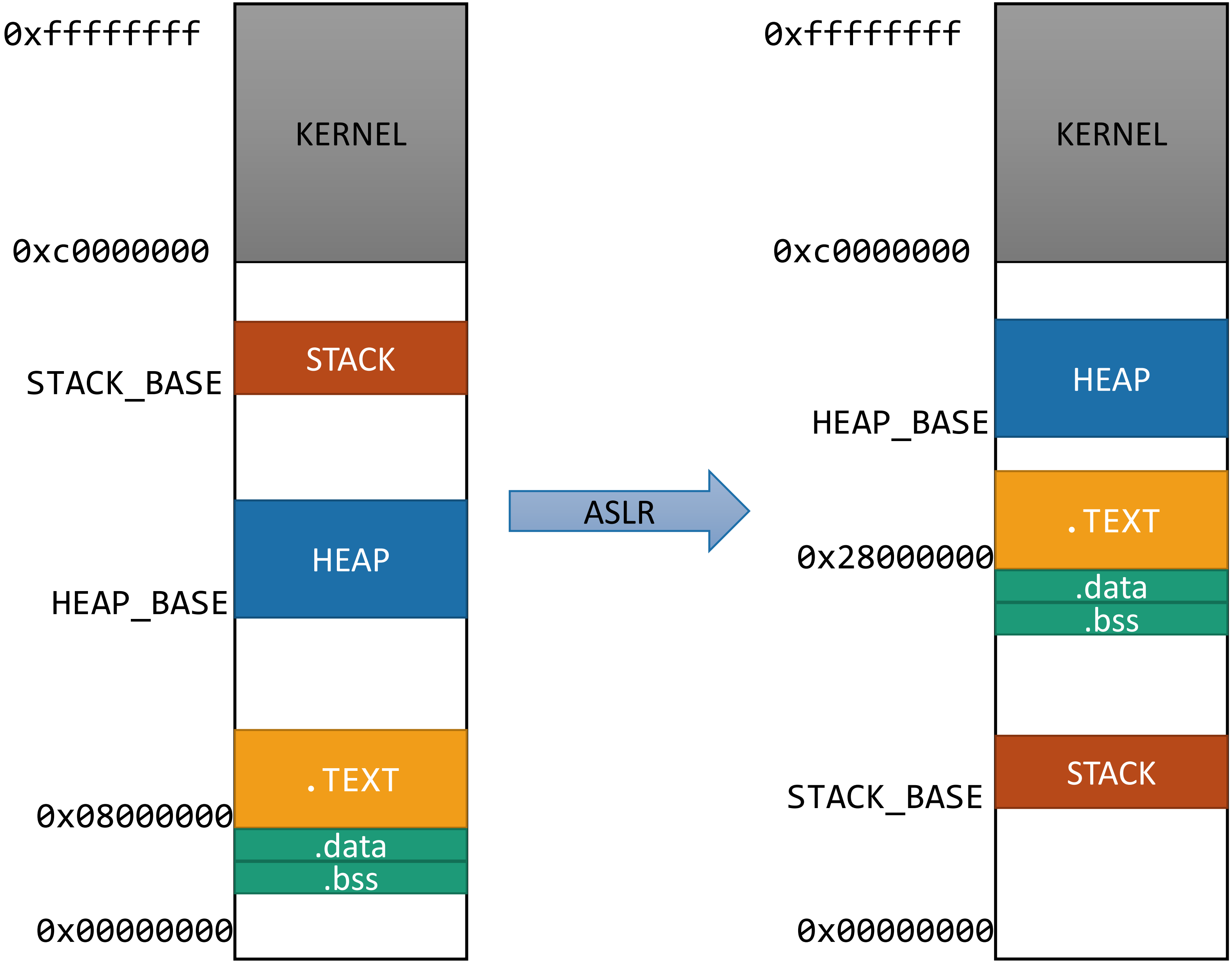
```
$ ldd /bin/ls
	linux-vdso.so.1 (0x00007ffc83b62000)
	libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edfdf1000)
	libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edfbe8000)
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)
	libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f9edf5cf000)
	libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9edf3cb000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f9ee0016000)
	libattr.so.1 => /lib/x86_64-linux-gnu/libattr.so.1 (0x00007f9edf1c6000)
	libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9edefa9000)
```

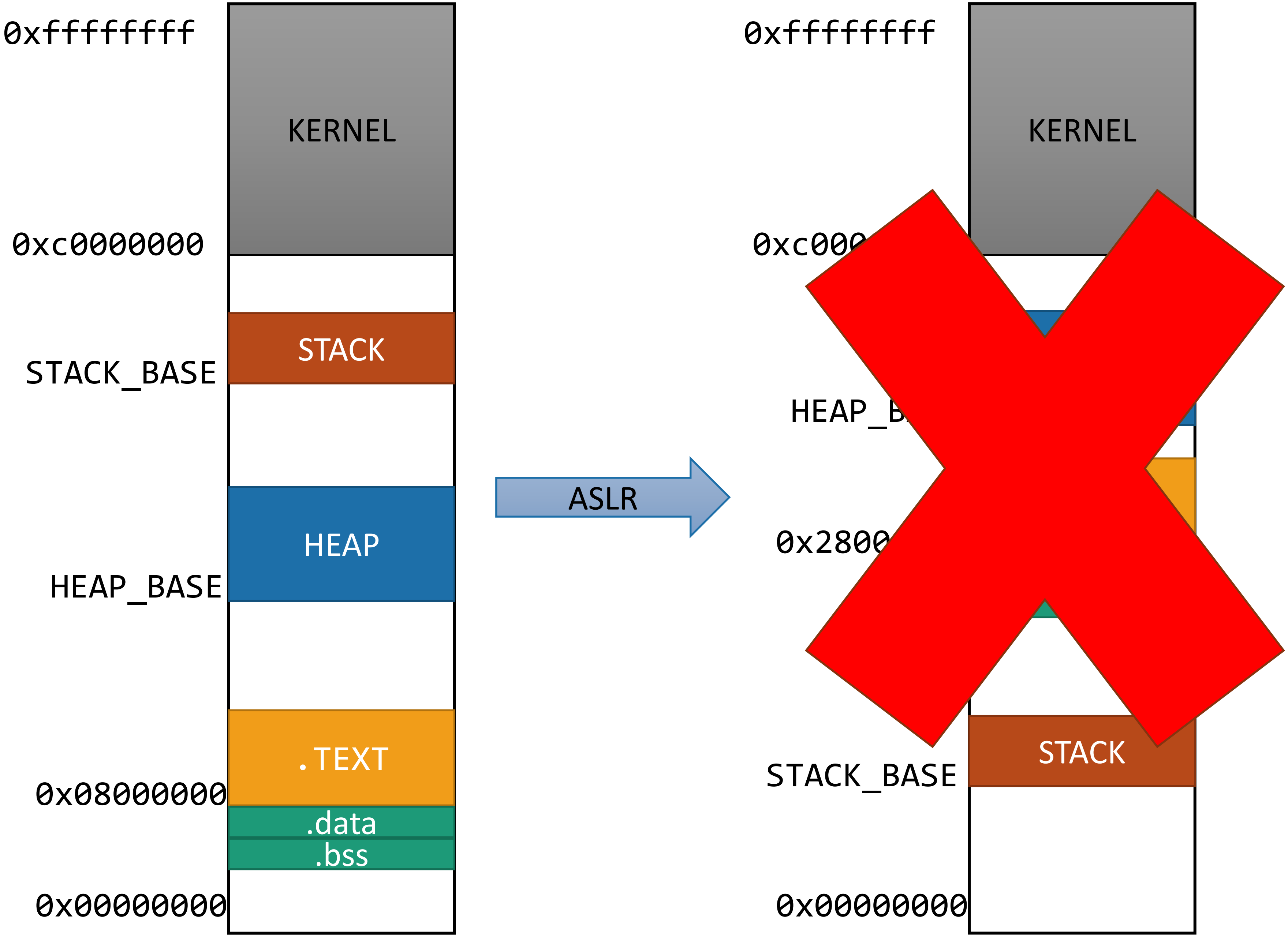
Enter Address Space Layout Randomization

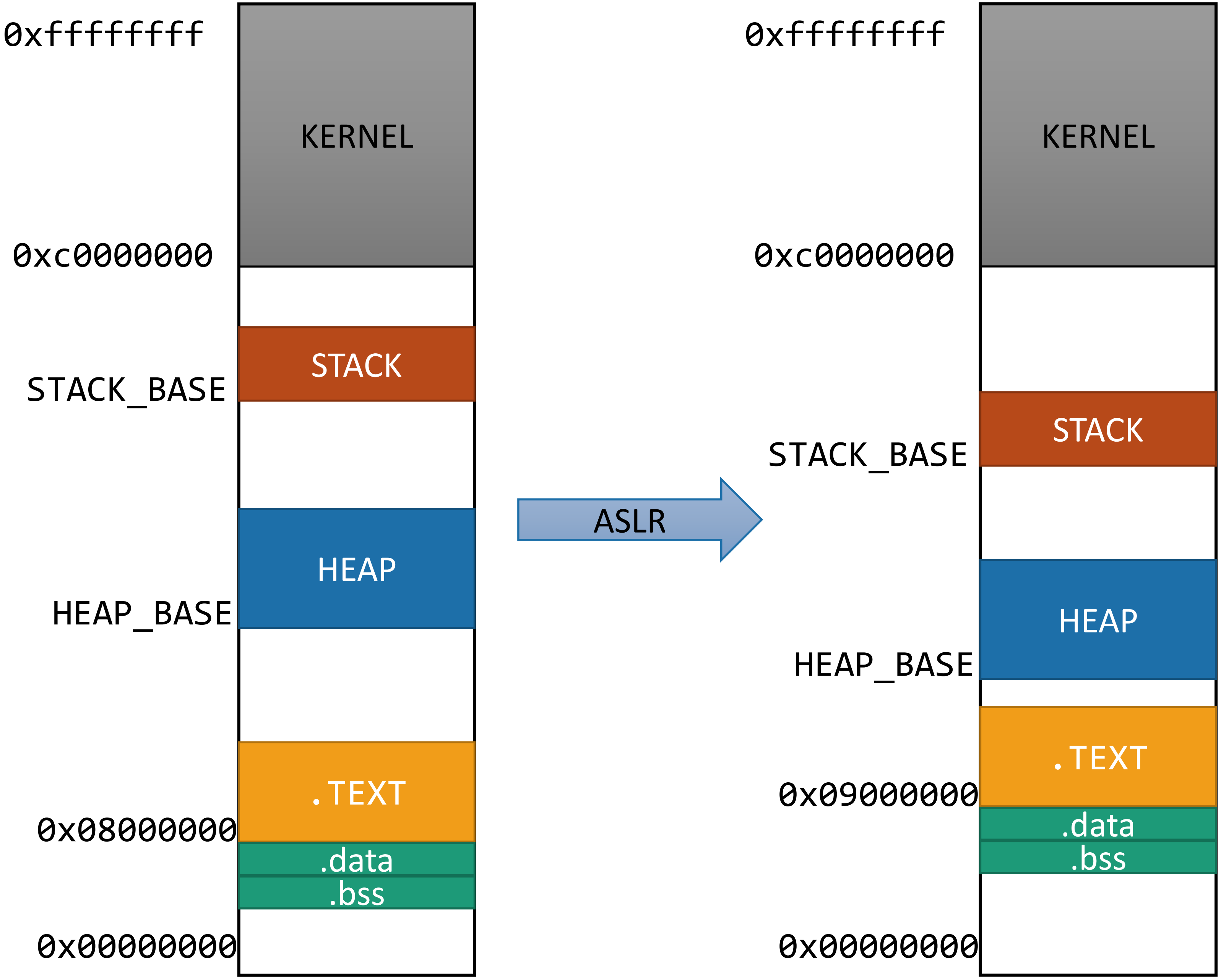
Disrupt exploits by:

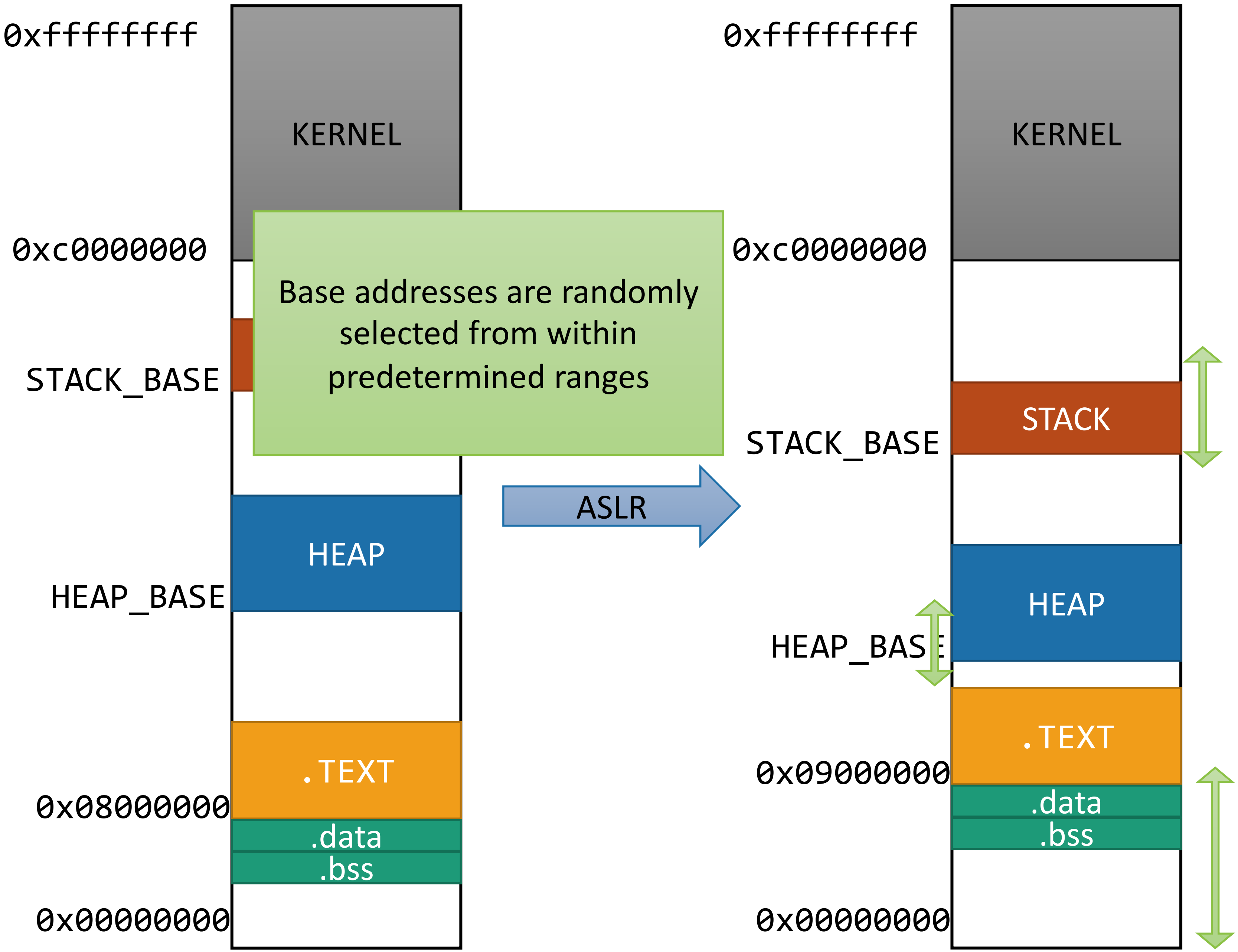
- Randomly choosing the base address of stack, heap, and code segments
- Randomize location of Global Offset Table
- Contains pointers to all functions/globals exported by a library

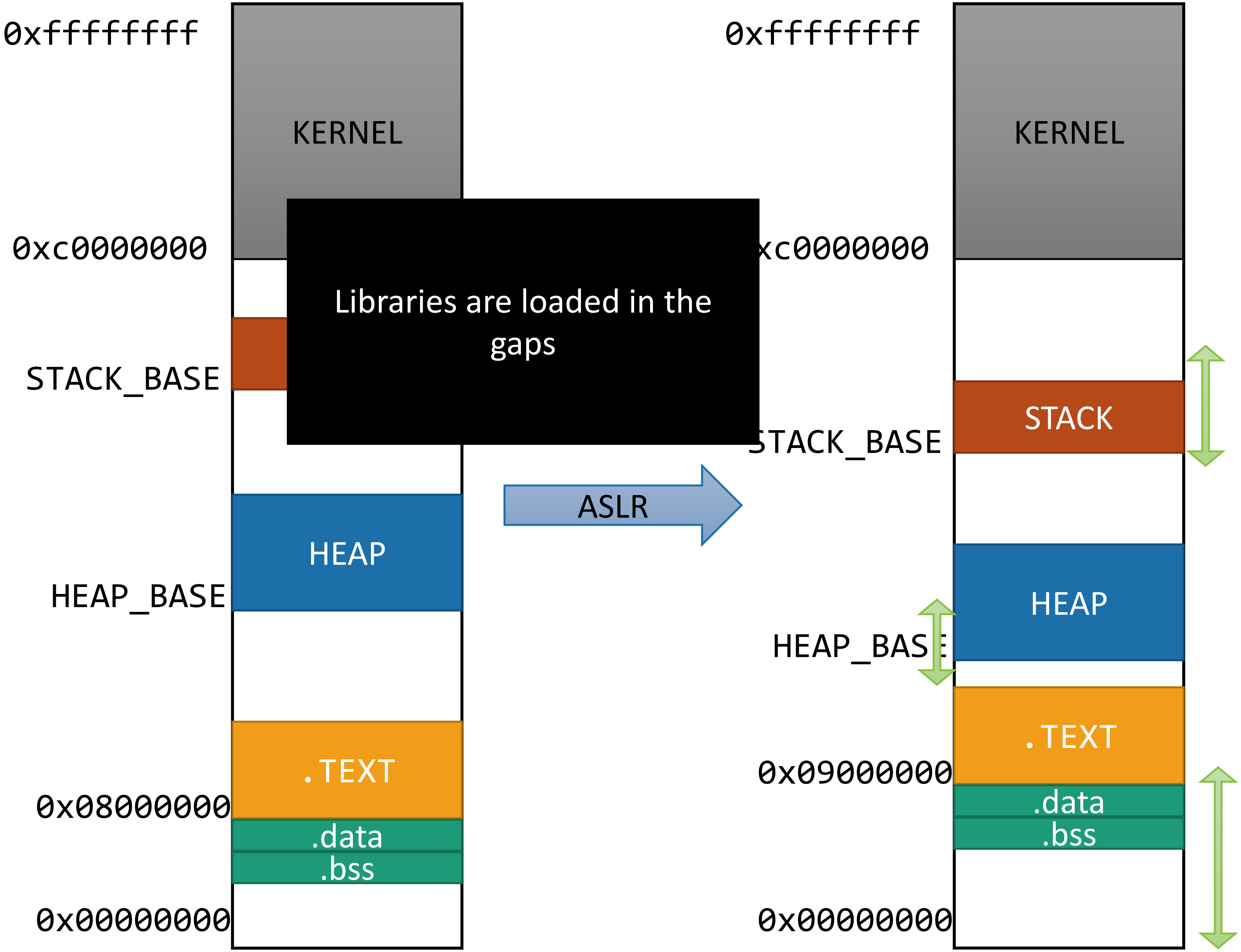

0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
0xc0000000
HEAP
HEAP_BASE
.TEXT
0x28000000
.data
.bss
STACK
STACK_BASE
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
0xc000
HEAP_B
0x2800
STACK
STACK_BASE
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x09000000
.data
.bss
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
Base addresses are randomly selected from within predetermined ranges
ASLR
0xffffffff
KERNEL
0xc0000000
STACK
STACK_BASE
HEAP
HEAP_BASE
.TEXT
0x09000000
.data
.bss
0x00000000


0xffffffff
KERNEL
0xc0000000
STACK_BASE
Libraries are loaded in the gaps
HEAP
HEAP_BASE
.TEXT
0x08000000
.data
.bss
0x00000000
ASLR
0xffffffff
KERNEL
xc0000000
STACK
TACK_BASE
HEAP
HEAP_BAS
.TEXT
0x09000000
.data
.bss
0x00000000

Example

```
unsigned long getEBP (void) {
        __asm ( ”movl %ebp ,%eax ” );
}

int main(void) {
    printf(”EBP: %x\n”, getEBP());
}
```

No ASLR

```
> ./getEBP
EBP:bffff3b8

> ./getEBP
EBP:bffff3b8
```

With ASLR

```
> ./getEBP
EBP:bfaa2e58

> ./getEBP
EBP:bf9114c8
```

ASLR in Linux

First implementation from the PaX project

- https://pax.grsecurity.net/

Now part of the vanilla kernel

ASLR in Linux

Rs: number of bits randomized in the stack area

Rm: number of bits randomized in the mmap() area

Rx: number of bits randomized in the main executable area

Ls: least significant randomized bit position in the stack area

Lm: least significant randomized bit position in the mmap() area

Lx: least significant randomized bit position in the main executable area

32-bit Linux

Rs = 24, Rm = 16, Rx = 16,
Ls = 4, Lm = 12, Lx = 12

64-bit Linux

Much larger entropy

ASLR in Windows

Vista and Server 2008

Stack randomization

- Find Nth hole of suitable size (N is a 5-bit random value), then random word-aligned offset (9 bits of randomness)

Heap randomization: 5 bits

- Linear search for base + random 64K-aligned offset

EXE randomization: 8 bits

- Preferred base + random 64K-aligned offset

DLL randomization: 8 bits

- Random offset in DLL area; random loading order

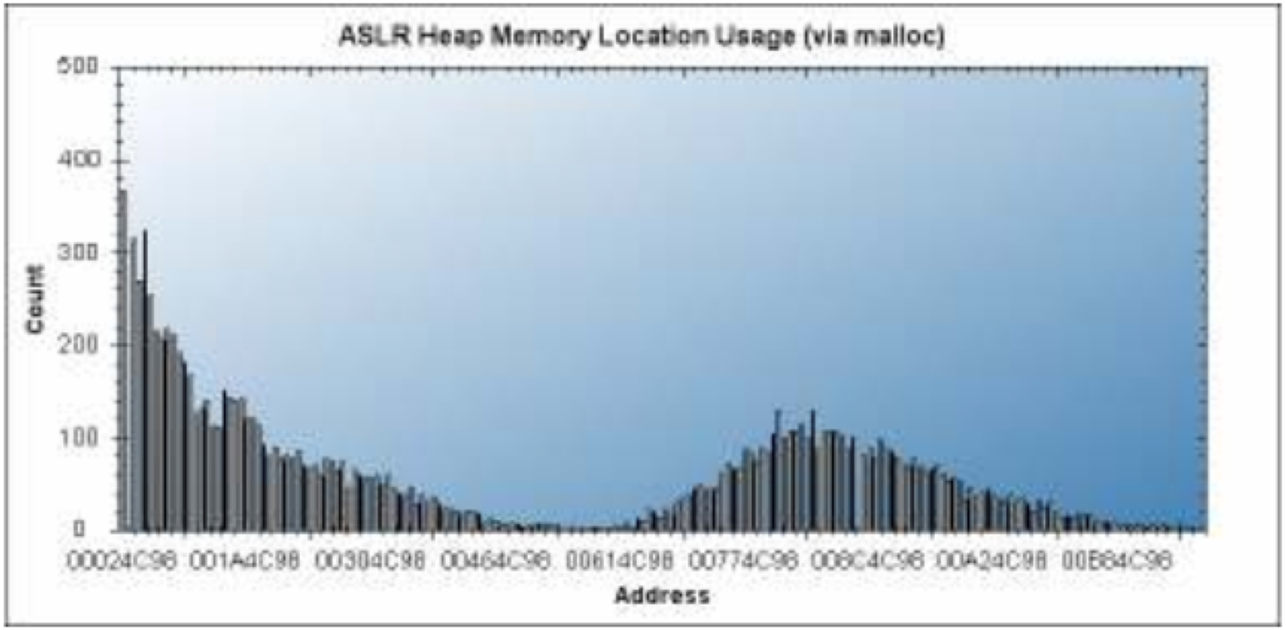
Weak Randomization

Weak random number generators, implementation bugs, etc.

Biased Selection of Heap Base Address

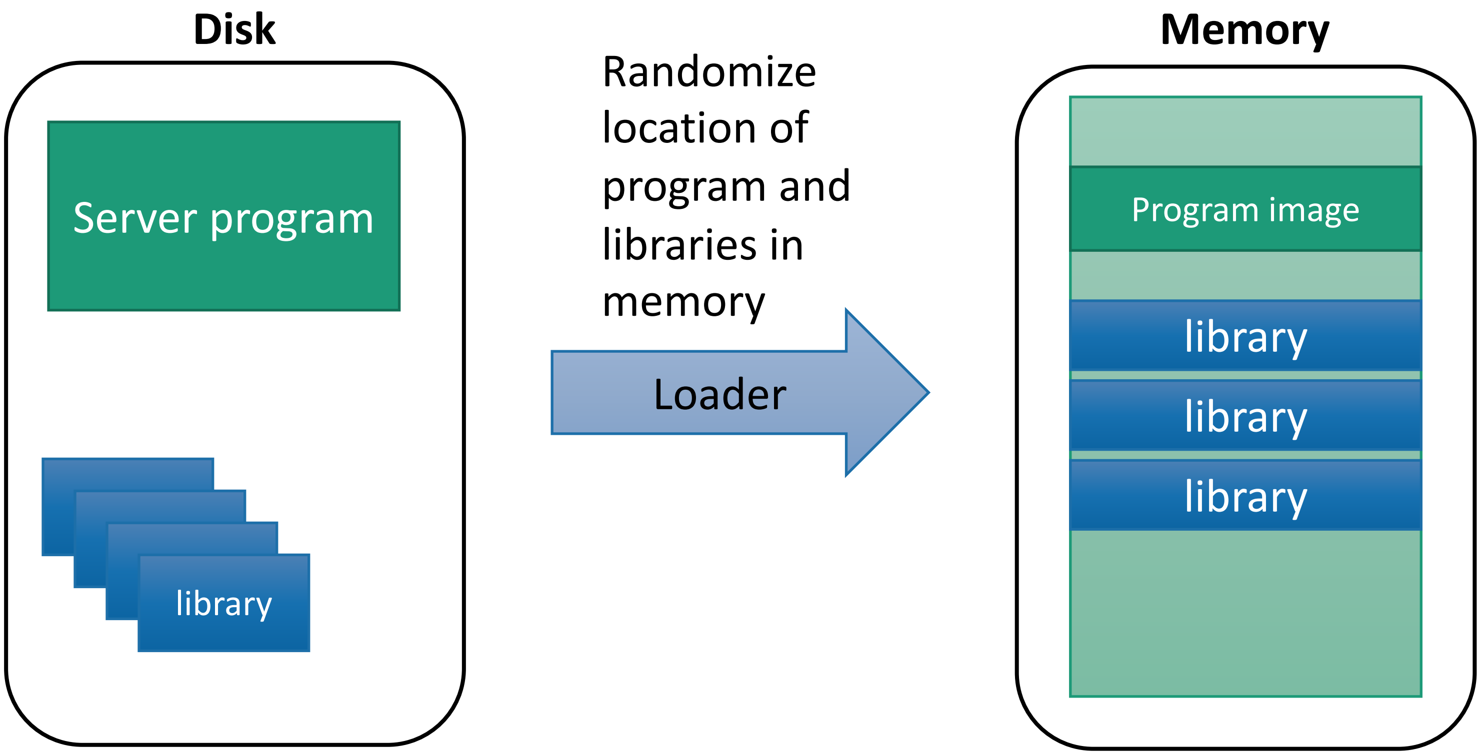
"An Analysis of Address Space Layout Randomization on Windows Vista", Ollie Whitehouse, BlackHat 2007

- https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf



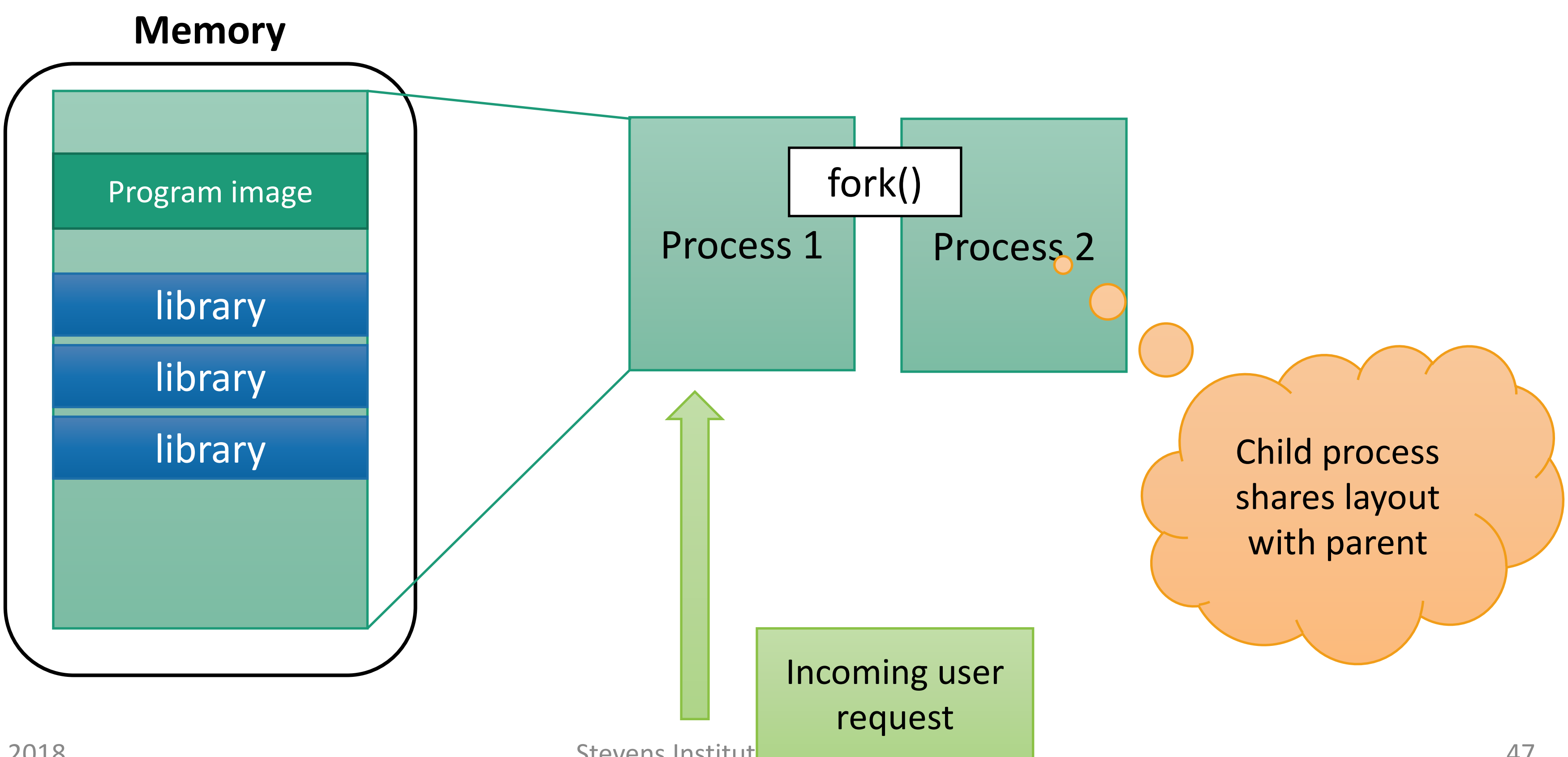
Brute-forcing ASLR

Exploiting server software using fork()



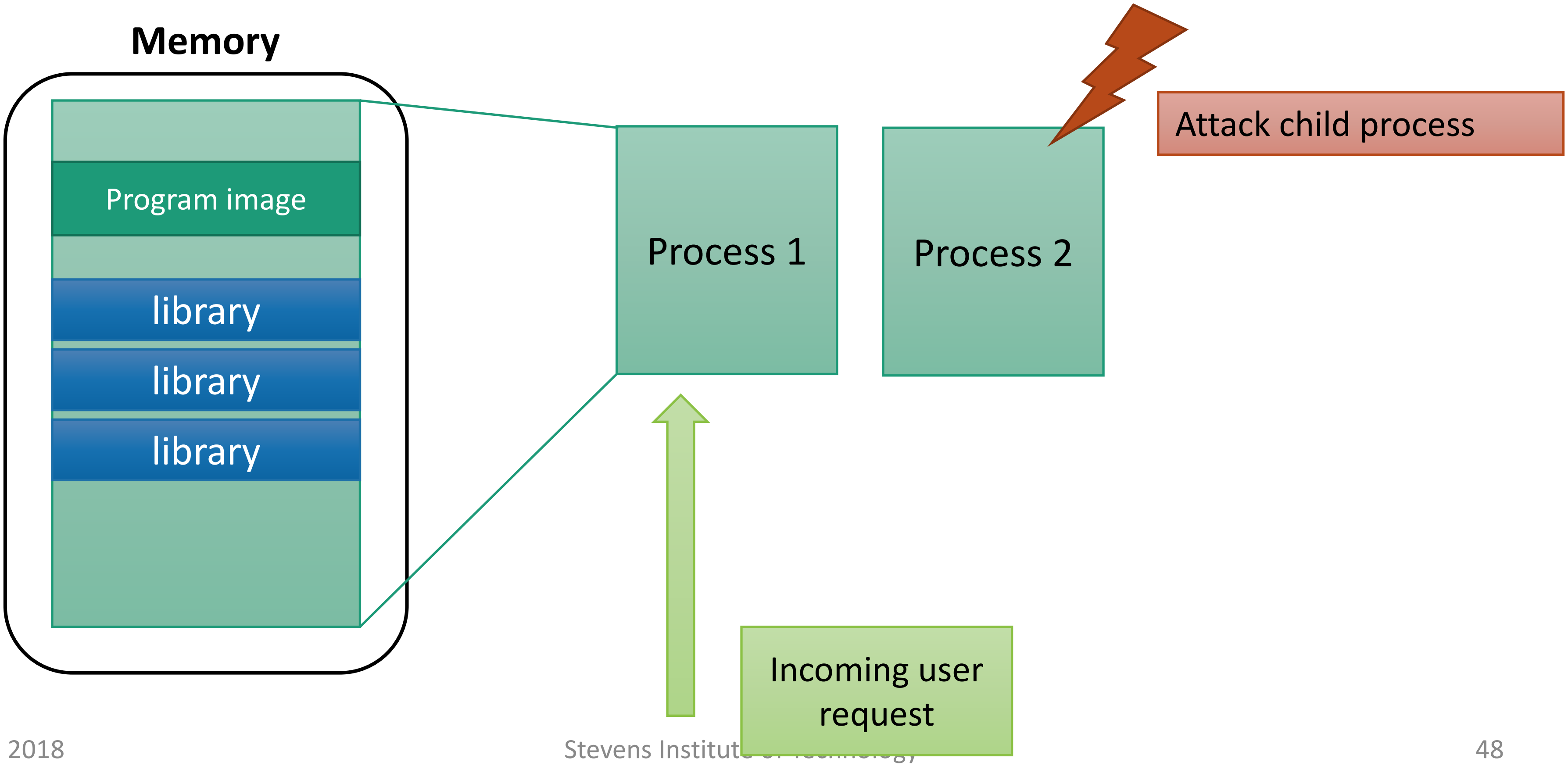
Brute-forcing ASLR

Exploiting server software using fork()



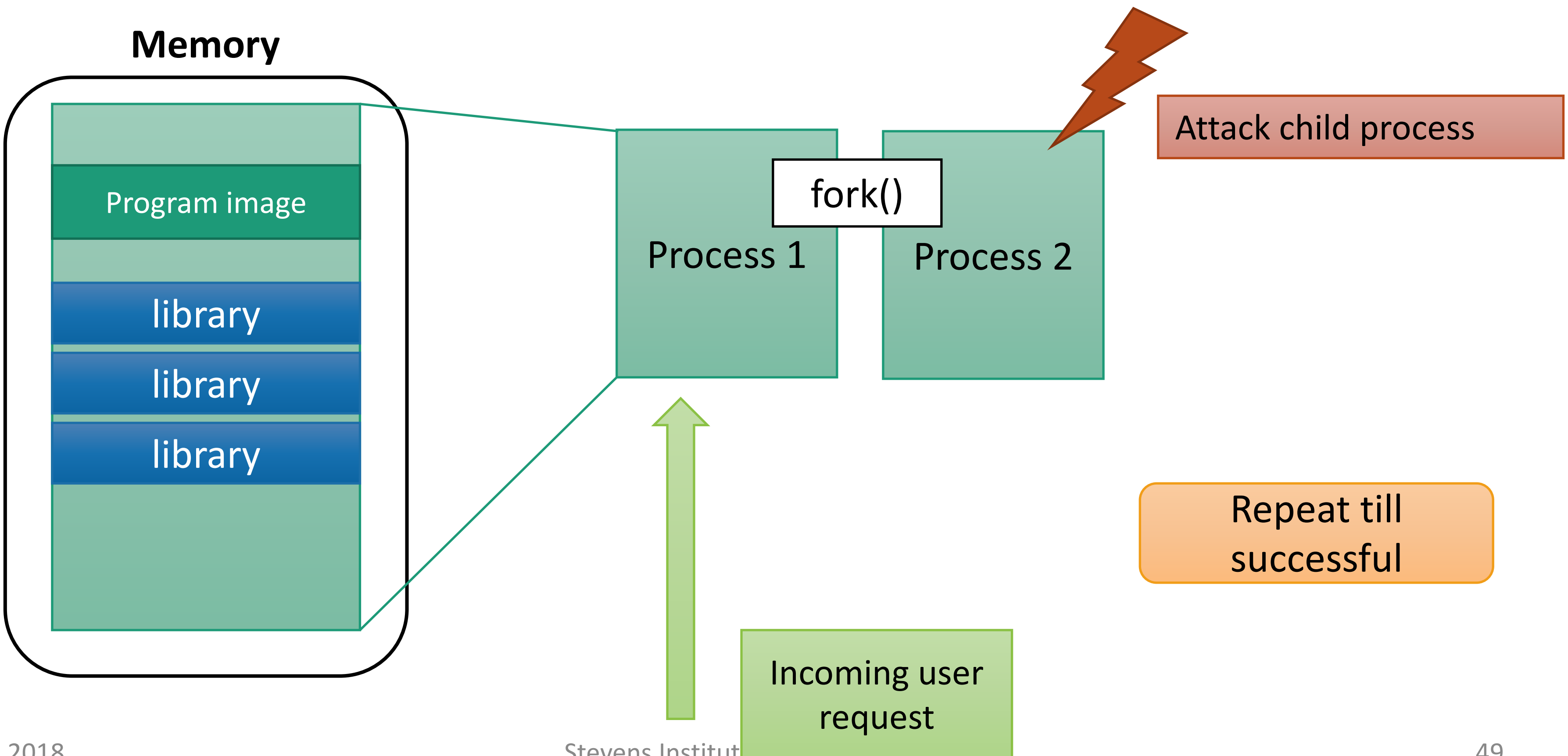
Brute-forcing ASLR

Exploiting server software using fork()



Brute-forcing ASLR

Exploiting server software using fork()



ASLR and Code

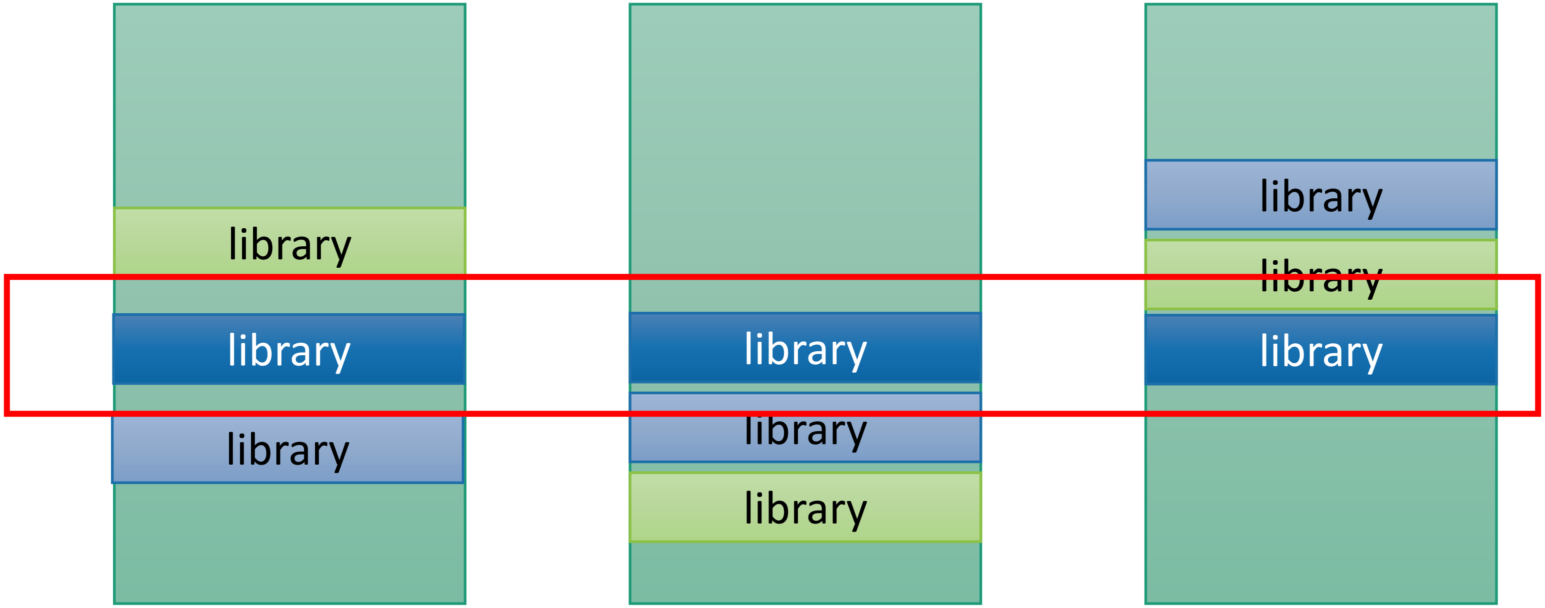
For ASLR to be applied to code it needs to be position independent

Libraries → Position Independent Code (PIC)

Executables → Position Independent (PIE)

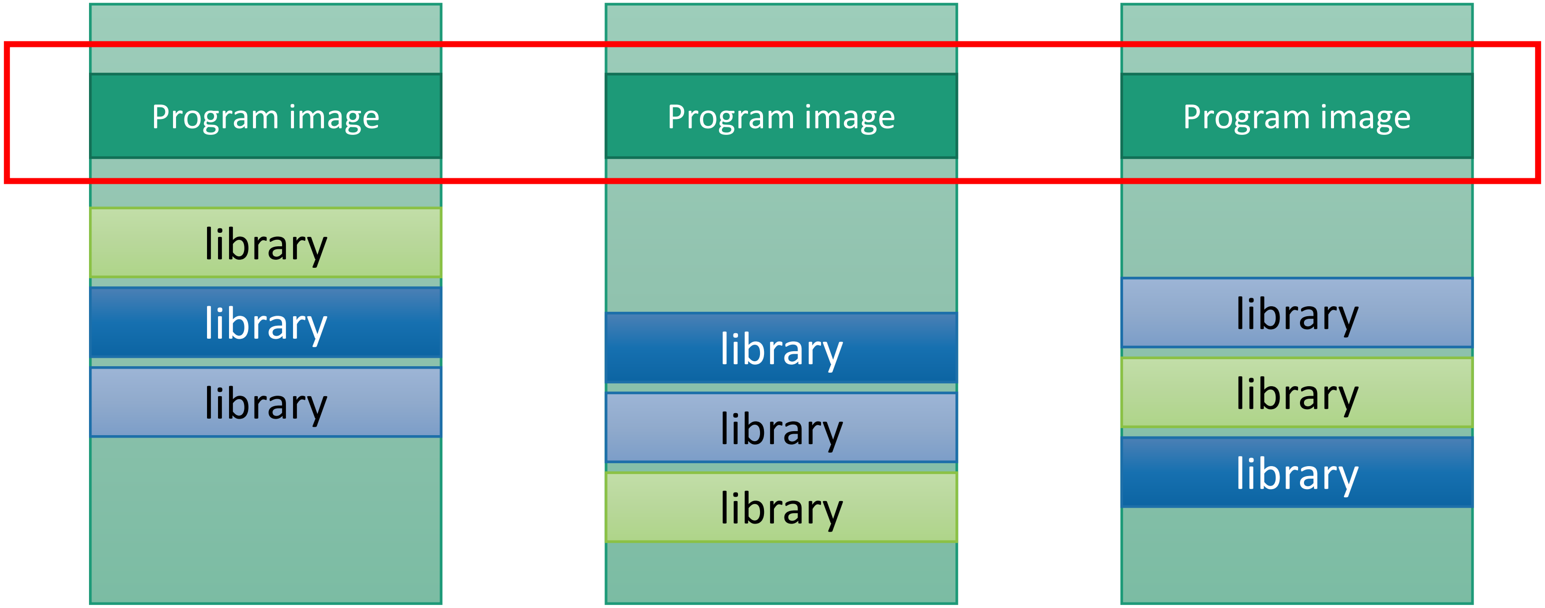
Exploit the Weakest Link

A single non-randomized library may be enough



Exploit the Weakest Link

Do not forget the program image



Exploit the Weakest Link

Executables became PIE recently

Distribution	Tested Binaries	PIE Enabled	Not PIE
Ubuntu 12.10	646	111 (17.18%)	535
Debian 6	592	61 (10.30%)	531
CentOS 6.3	1340	217 (16.19%)	1123

Percentage of PIE binaries in different Linux distributions

Return-to-PLT

PLT

```
00000000004004a0 <puts@plt>:
  4004a0:       ff 25 3a 06 20 00       jmpq   *0x20063a(%rip)        # 600ae0 <_GLOBAL_OFFSET_TABLE_+0x20>
  4004a6:       68 01 00 00 00          pushq  $0x1
  4004ab:       e9 d0 ff ff ff          jmpq   400480 <_init+0x28>

00000000004004b0 <printf@plt>:
  4004b0:       ff 25 32 06 20 00       jmpq   *0x200632(%rip)        # 600ae8 <_GLOBAL_OFFSET_TABLE_+0x28>
  4004b6:       68 02 00 00 00          pushq  $0x2
  4004bb:       e9 c0 ff ff ff          jmpq   400480 <_init+0x28>
```

```
0000000000600ac0 <_GLOBAL_OFFSET_TABLE_>:
  600ae0:       a6 04 40 00 00 00 00 00
  600ae8:       b6 04 40 00 00 00 00 00
```

PLT entry consists of 3 instructions

- First jumps to address contained in the GOT
- Initially pointing to the linker →will resolve the function and update the GOT

Functions are bound lazily →on first call

Information Leaks

An information leak is caused by exploiting a bug that discloses the memory layout and/or contents of a program

Main idea:

- Corrupting (partially) data that affect what or how much is read from memory
- Receive the output of the read

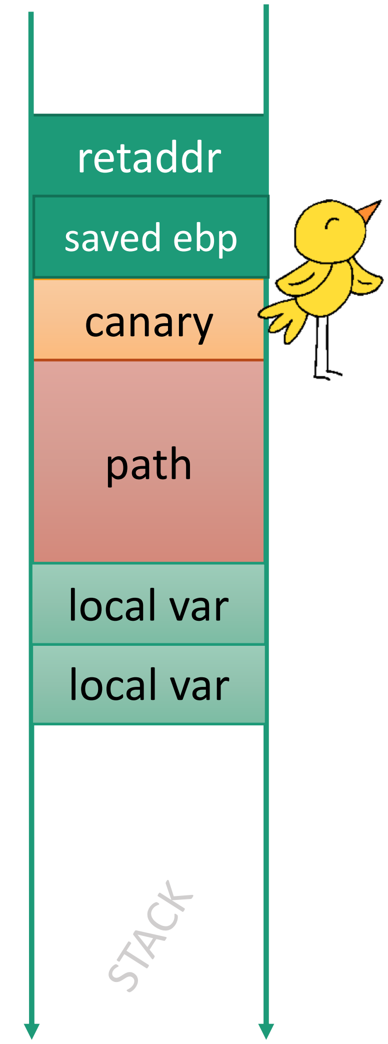
Leak Can Occur in the Stack

```
void func(char *filename, int len)
{
        char path[128] = "/tmp/";

        memcpy(path, filename, len);

        ...
        fprintf(logfl, "Opened %s\n", path);
        ...
}
```

Omitting or overwriting the terminating '\0' character and reading a string can leak data



Or the Heap

```
void string::copy(string *src)
{
            ...
            memcpy(this->data, src->data, src->len);
            ...
}

outputfile->copy(userinput);
...
logfl << “user entered ” << userinput << endl;
```

```
class string
{
...
private:
            size_t len;
            char *data;
...
};
```

Or the Heap

```
void string::copy(string *src)
{
            ...
            memcpy(this->data, src->data, src->len);
            ...
}

outputfile->copy(userinput);
...
logfl << “user entered ” << userinput << endl;
```

```
class string
{
...
private:
            size_t len;
            char *data;
...
};
```

Summary of ASLR Weaknesses

Memory leaks

- Combine memory leaks with control-flow hijacking
- Repeatable arbitrary memory leaks are better

Insufficient entropy

Incompatible binaries

…

Attacks in the Information Leak Era

Many of the other bugs we have already seen can be used to leak information

- Overflow
- Use-after-free
- Type confusion

JavaScript is frequently used as it allows dynamically triggering the exploit multiple times to

- Leak data
- Hijack control flow

https://media.blackhat.com/bh-us-12/Briefings/Serna/BH_US_12_Serna_Leak_Era_Slides.pdf