Early Defenses and More Attacks

CS-576 Systems Security

Instructor: Georgios Portokalidis

Spring 2018

Topics

Recap: Control-flow hijacking and code injection attacks

Non executable stack (and heap)

Early code-reuse attacks/return-to-libc

ASCII armored space

Stackguard & Stackshield

Heap protections

ASLR

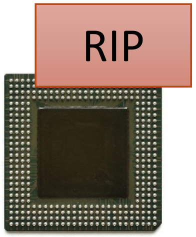
Bypassing ASLR

Recap: Control-flow Hijacking Attacks

Attacks that take over control flow…

…by leveraging bugs like…

- Stack and heap overflows
- Format string
- Use-after-free
- Type confusion
- Integer overflows



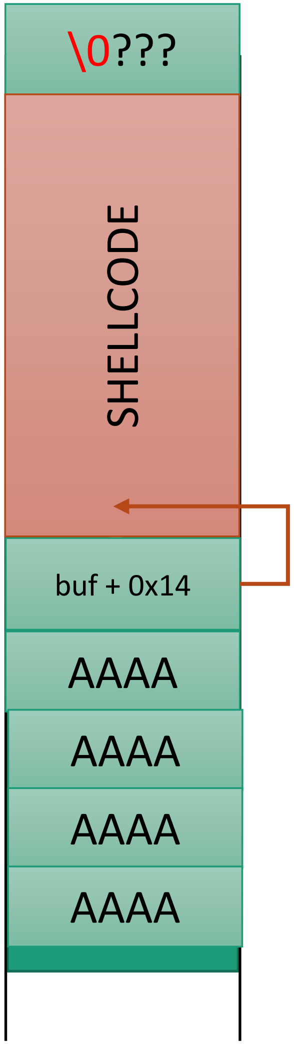
…to corrupt a pointer in memory

- Function pointers on the heap or stack
- Return addresses on the stack
- Virtual table pointers

Recap: Code Injection

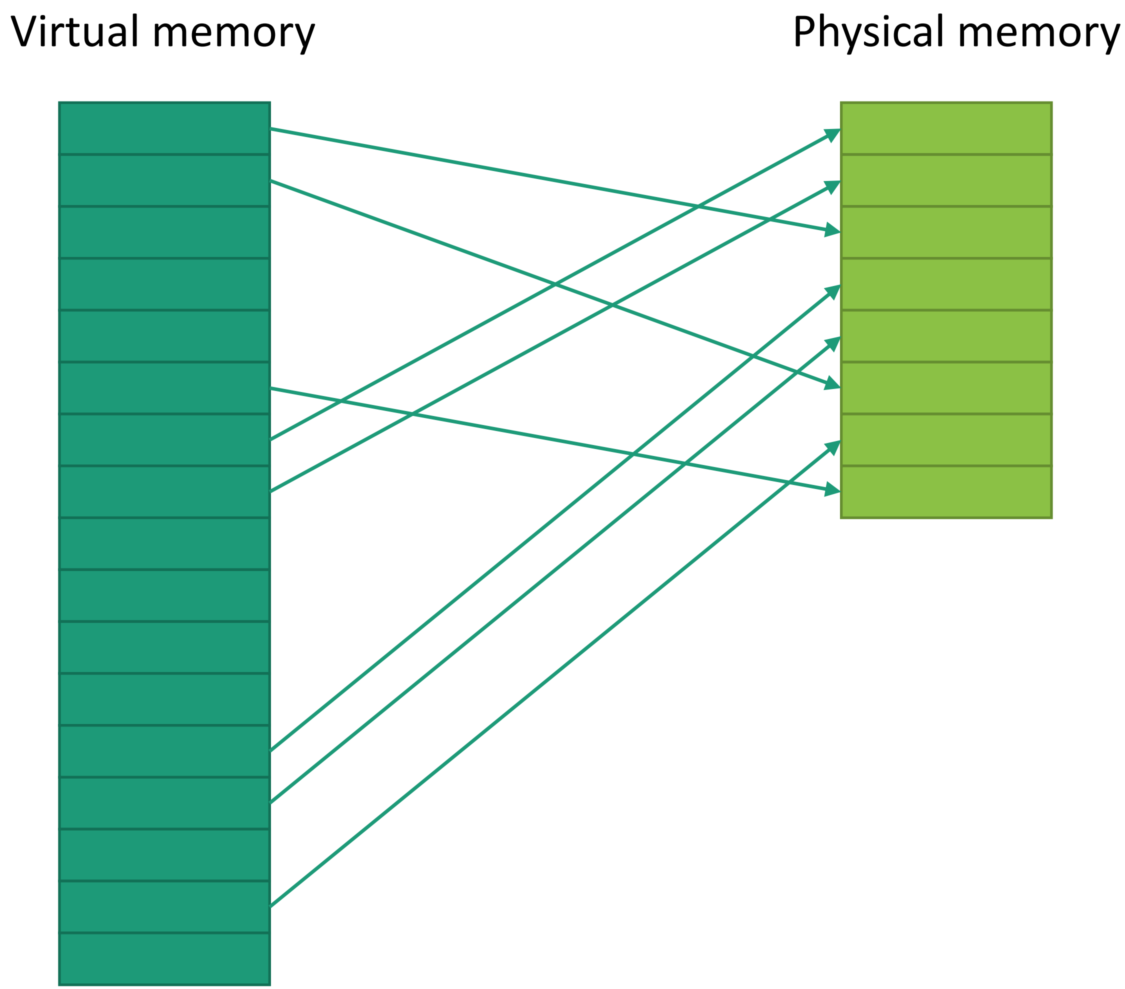
Malicious code (shellcode) is injected into attacker controlled, executable memory

The controlled instruction pointer is directed to injected code

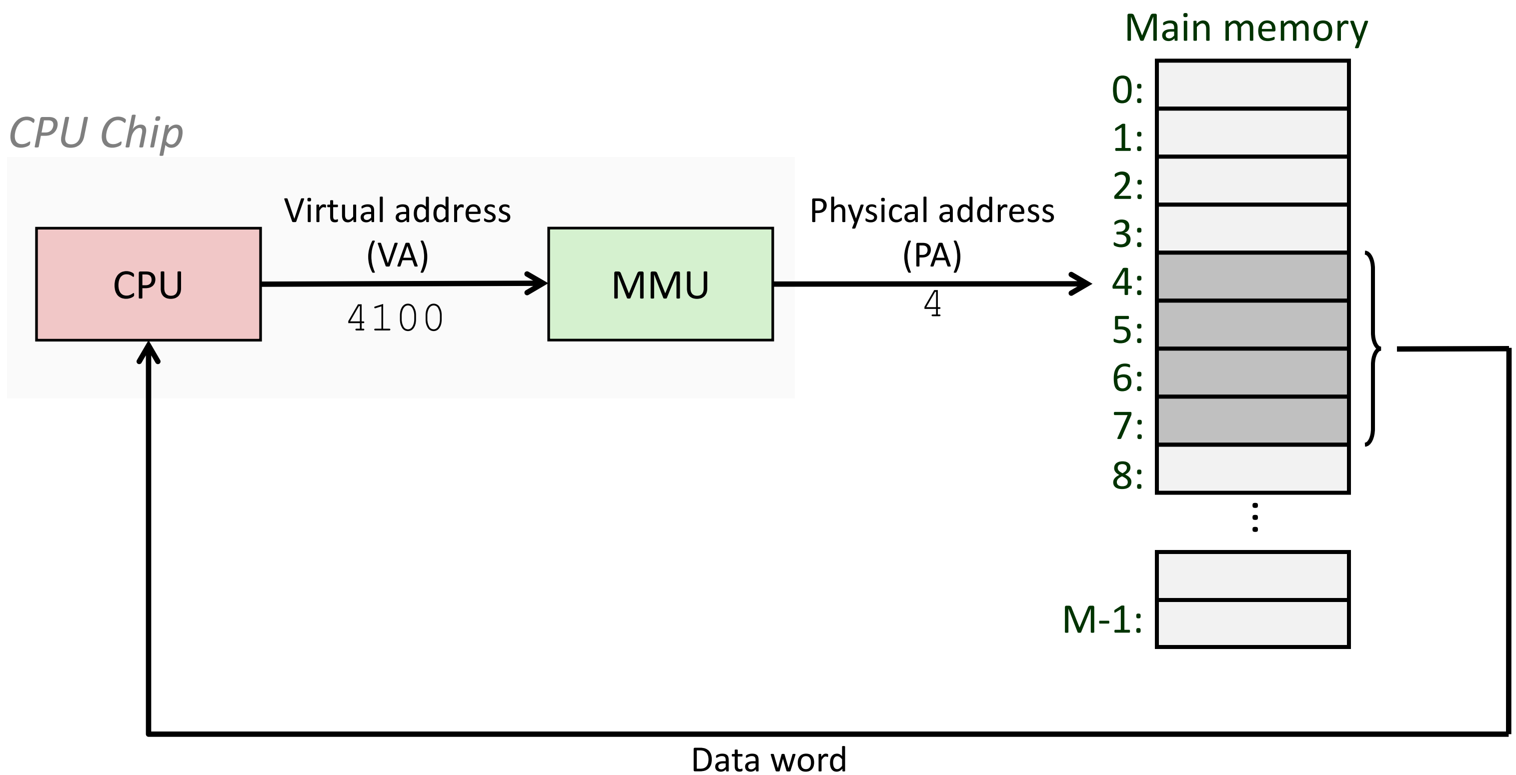


Non-Executable Stack (and data segments)

Virtual Memory



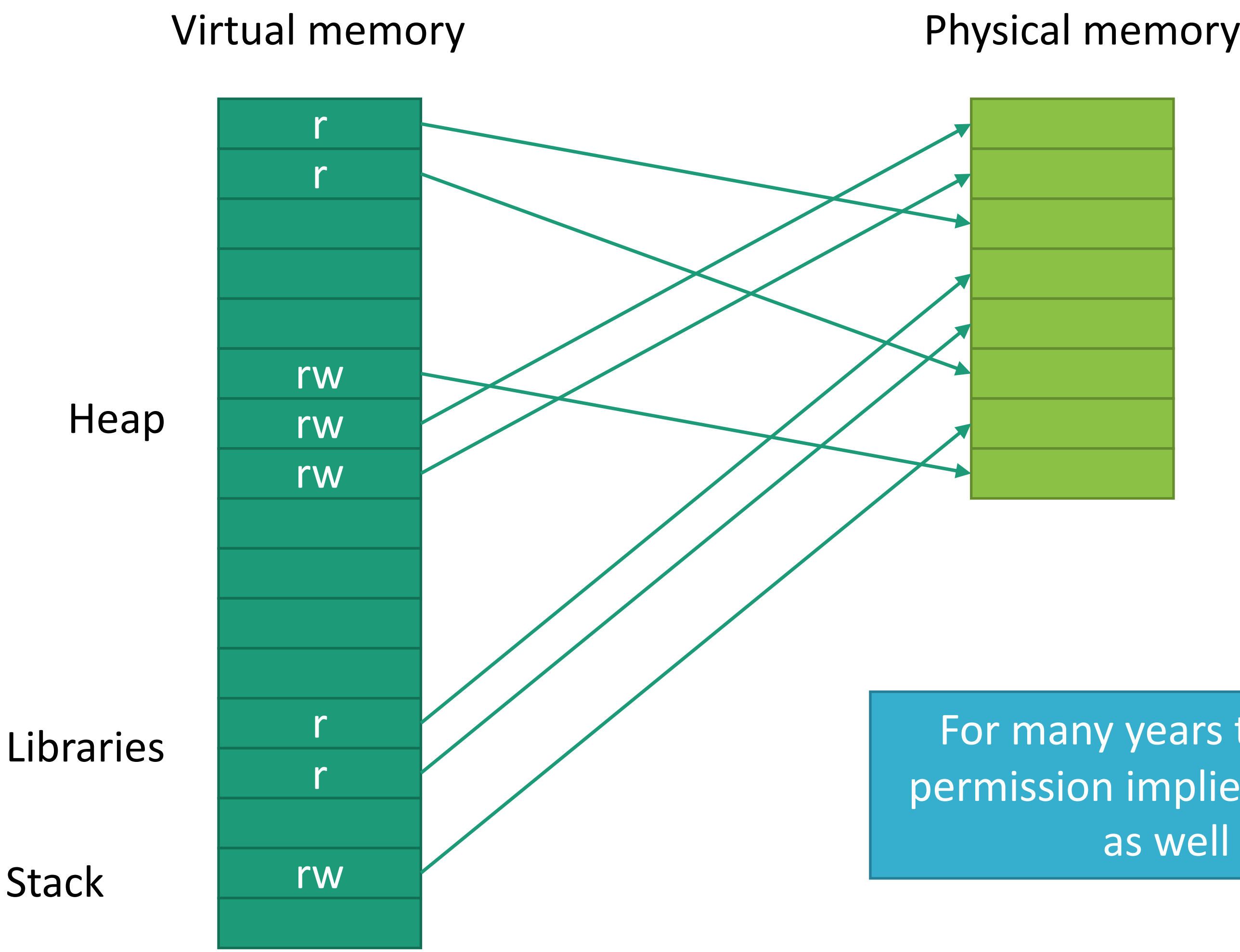
The Memory Management Unit



Used in all modern servers, laptops, and smart phones

One of the great ideas in computer science

Page Permissions



For many years the **read** permission implied **execute** as well

Non-executable Memory (PaX)

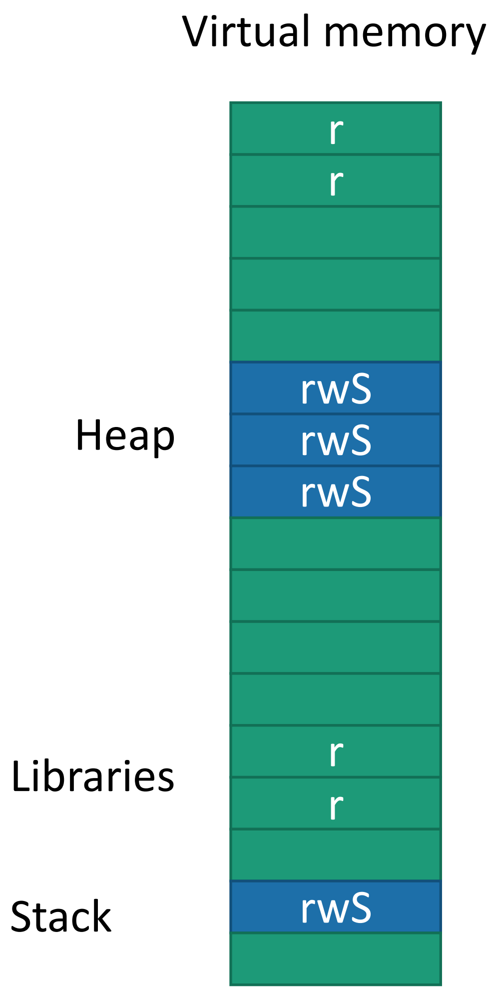
PaX stands for PageEXec

Introduced in 2000

A Linux kernel patch protection emulating Non-Executable memory

PaX refused code execution on writable pages

Emulating Non-Executable Memory



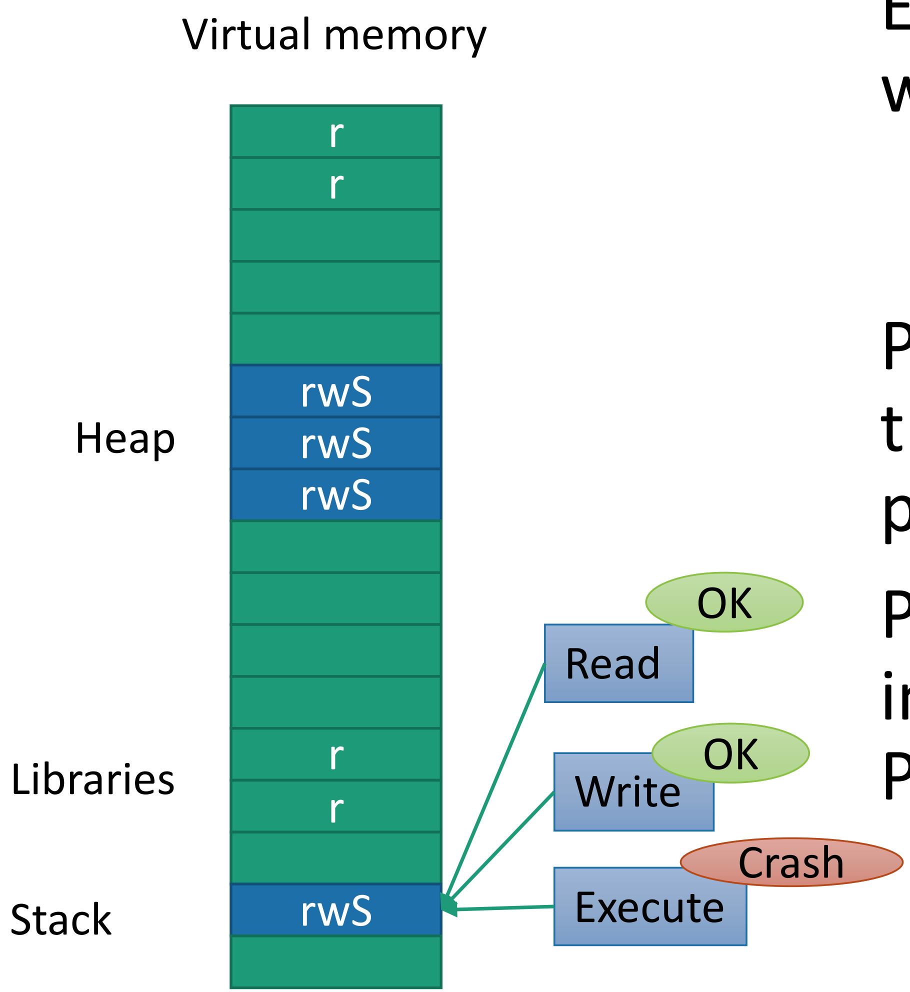
Each page is associated with a supervisor bit

- Access only allowed from the kernel

PaX set that bit and kept track of PaX-protected pages

Page-fault handler intercepted to check for PaX-protected pages

Emulating Non-Executable Memory



Each page is associated with a supervisor bit

- Access only allowed from the kernel

PaX set that bit and kept track of PaX-protected pages

Page-fault handler intercepted to check for PaX-protected pages

NX-bit

Processor manufacturers introduced a new bit in page permissions to prevents code injections

Coined **N**o-e**X**ecute or **E**xecute **N**ever

The NX-bit (No-eXecute) was introduced first by AMD to resolve such issues in 2001

- Asserting NX, makes a readable page non-executable
- Frequently referred to as Data Execution Prevention (DEP) on Windows

Marketed as antivirus technology

Enhanced virus protection

Costin Raiu *Kaspersky Lab*

download slides (PDF)

AMD Athlon 64 CPU Feature:

1. HyperTransport technology
2. Cool'n'Quiet technology
3. Enhanced Virus Protection for Microsoft Windows XP SP2

The AMD64 architecture is an affordable way of getting the power of 64-bit processing into a desktop computer. Interesting enough, AMD has not only designed an improved CPU core and longer registers, but they have also included a feature designed to significantly increase the security of modern operating systems.

The idea of hardware protection isn't new – every contemporary CPU includes at least a basic hardware mechanism for enforcing a security scheme, for instance, those from the Intel x86 family, based on

Adoption

A non-executable stack was not immediately adopted

The OS occasionally needed to place code in the stack

- For example, trampoline code for handling UNIX signals

W^X Policy

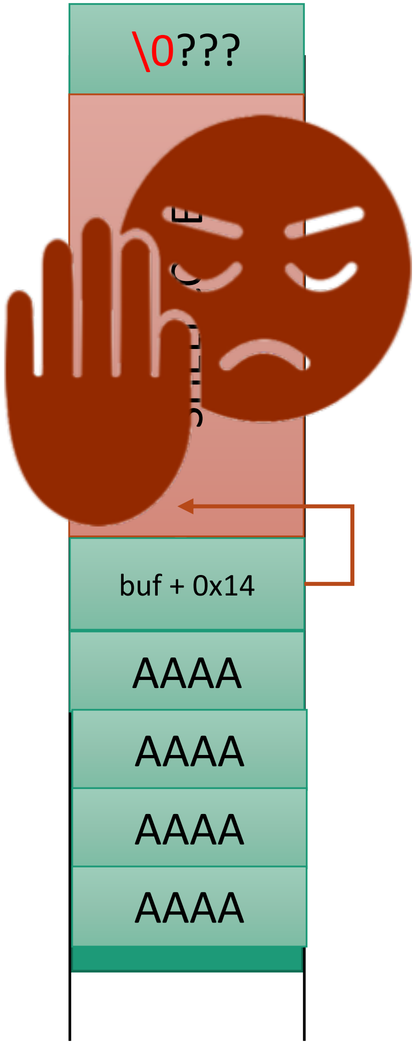
Data-execution prevention lead to a more generic security policy

The Write XOR Execute (W^X) policy mandates that in a program there are no memory pages that are both writable and executable

No More Code Injection

Malicious code (shellcode) is injected into attacker controlled, executable memory

The controlled instruction pointer is directed to injected code



Unless You Are a Browser…

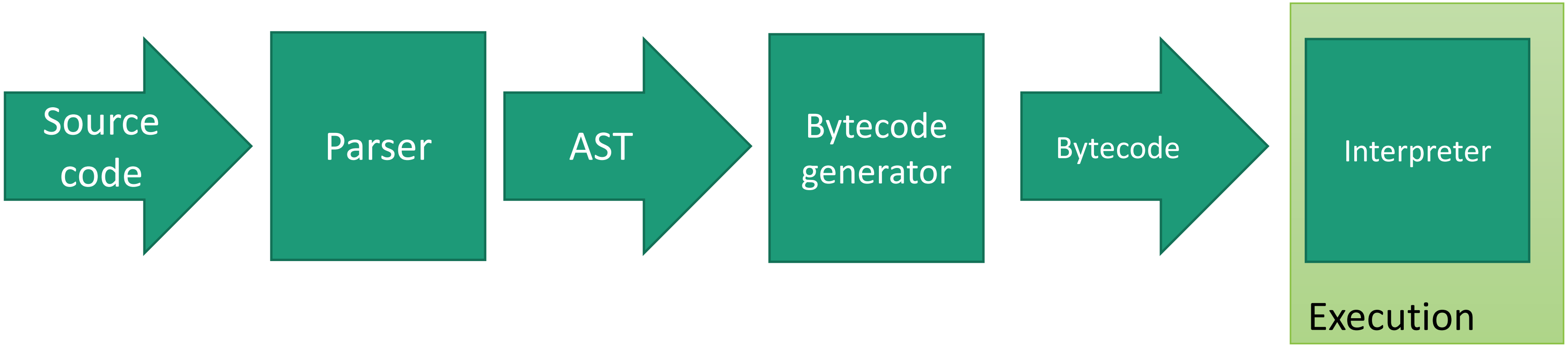
Very popular software

- Probably installed on every client device

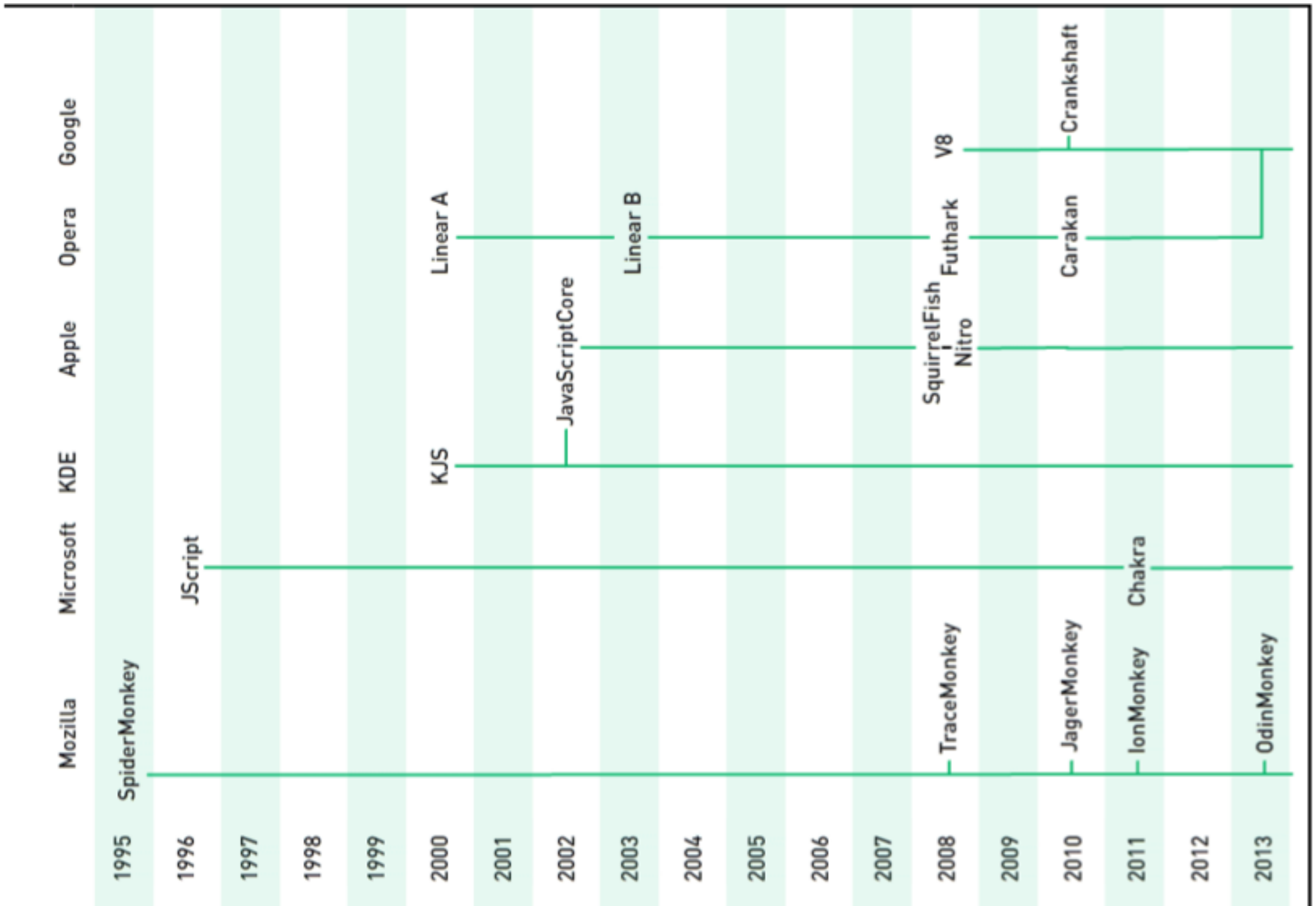
Large and complex software

Execute JavaScript

How Does JavaScript Run

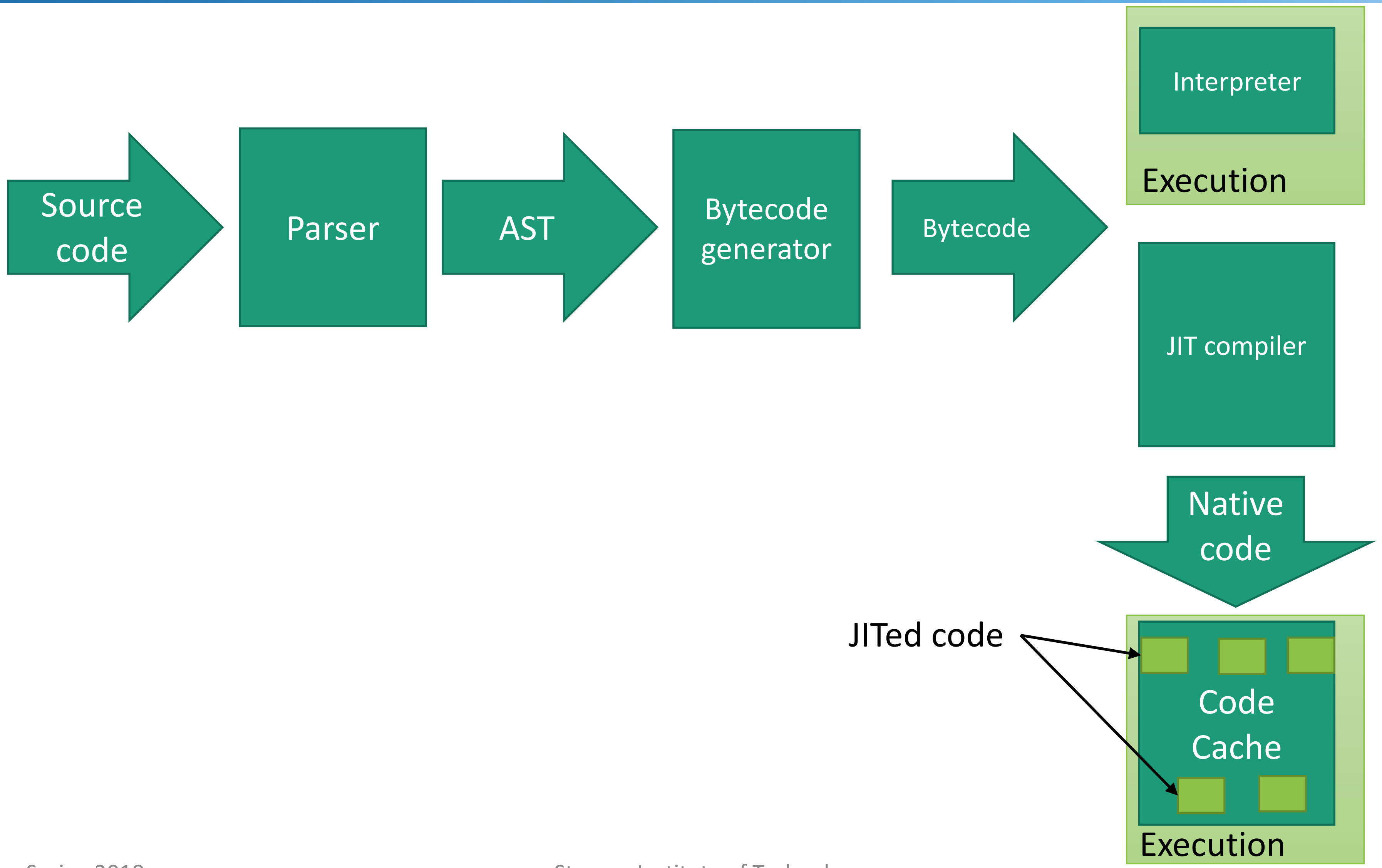


JS Engines Family Tree

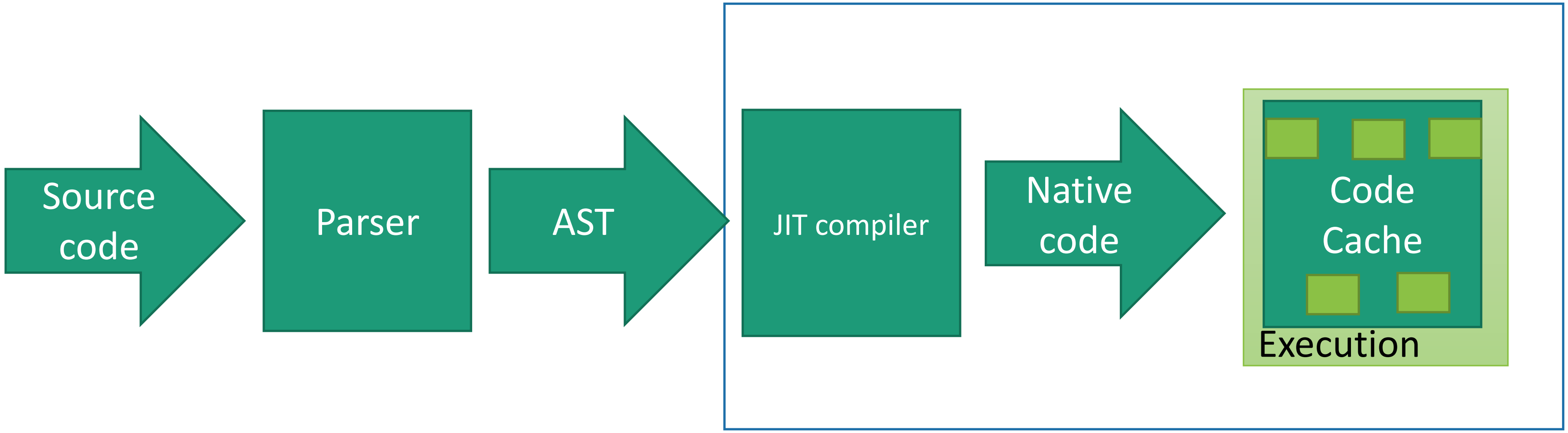


http://creativejs.com/2013/06/the-race-for-speed-part-1-the-javascript-engine-family-tree/index.html

How Does JavaScript Run



How Does JavaScript Run

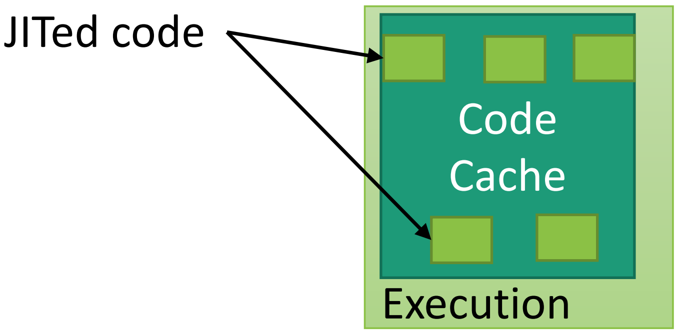


- Google V8 designed specifically to execute at speed.
- Bytecode generation skipped
- Directly emit native code
- Overall JavaScript execution improved by 150%

Code Cache

JITed code and code cache have interesting properties from the perspective of the attacker

- Code is continuously generated
- Code needs to be executable

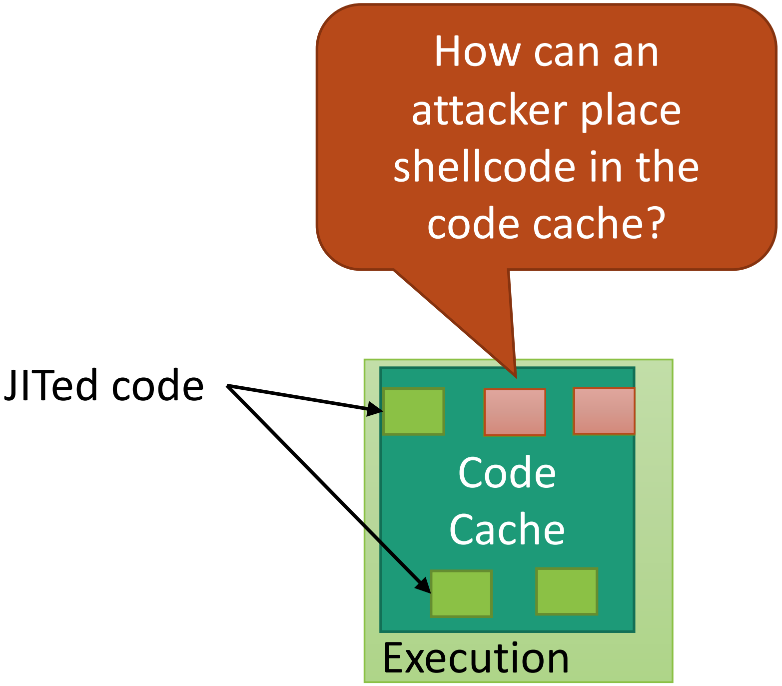


Violates the W^X policy

Code Cache

JITed code and code cache have interesting properties from the perspective of the attacker

- Code is continuously generated
- Code needs to be executable



Violates the W^X policy

From JS to Code Cache

JS code is JITed and placed in the code cache

Some JS engines do not separate data and code

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552'); // CORE
myvar += unescape('%u\414C%u\214E'); // LAN!
alert("allocation done");

</script>
</body>
</html>
```

Bypassing PaX and NX

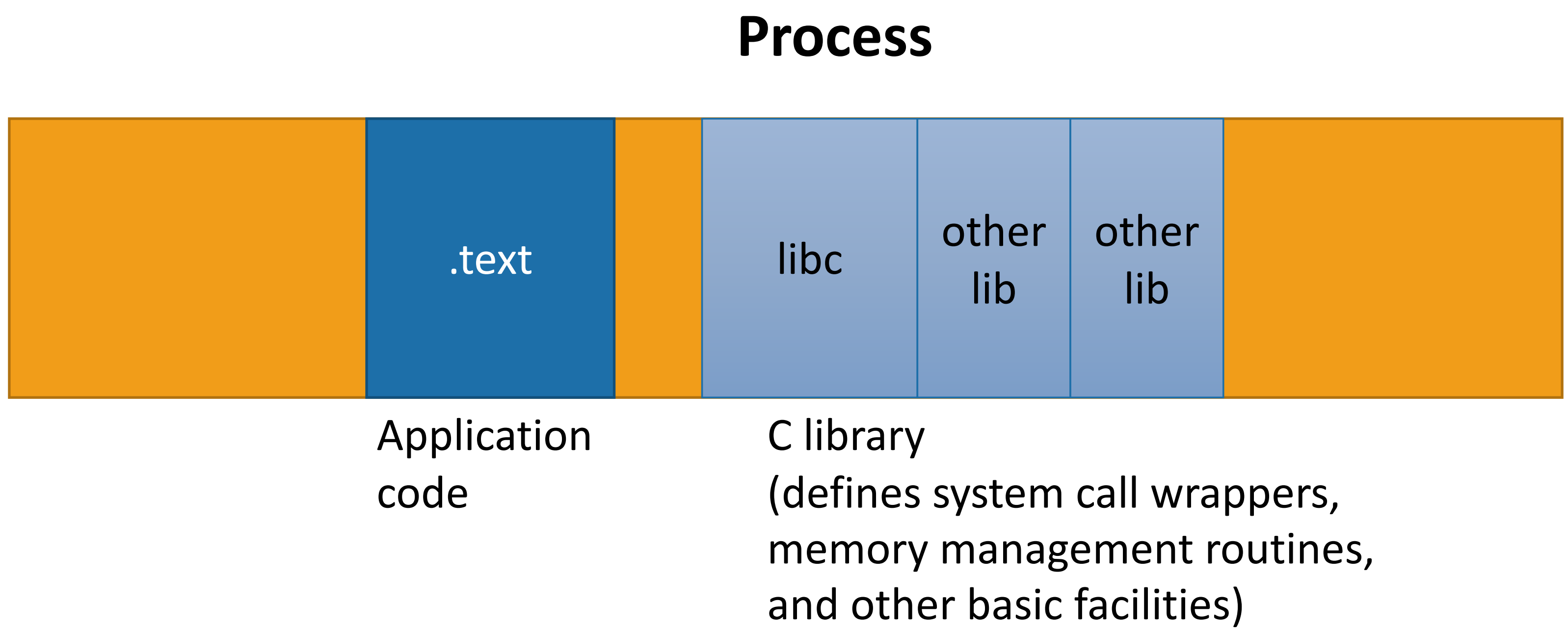
Return-to Attacks

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)

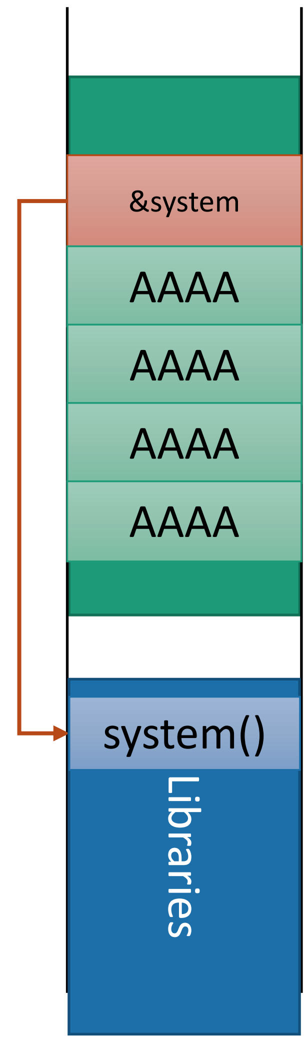


```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffc83b62000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edfdf1000)
        libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edfbe8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)
        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()

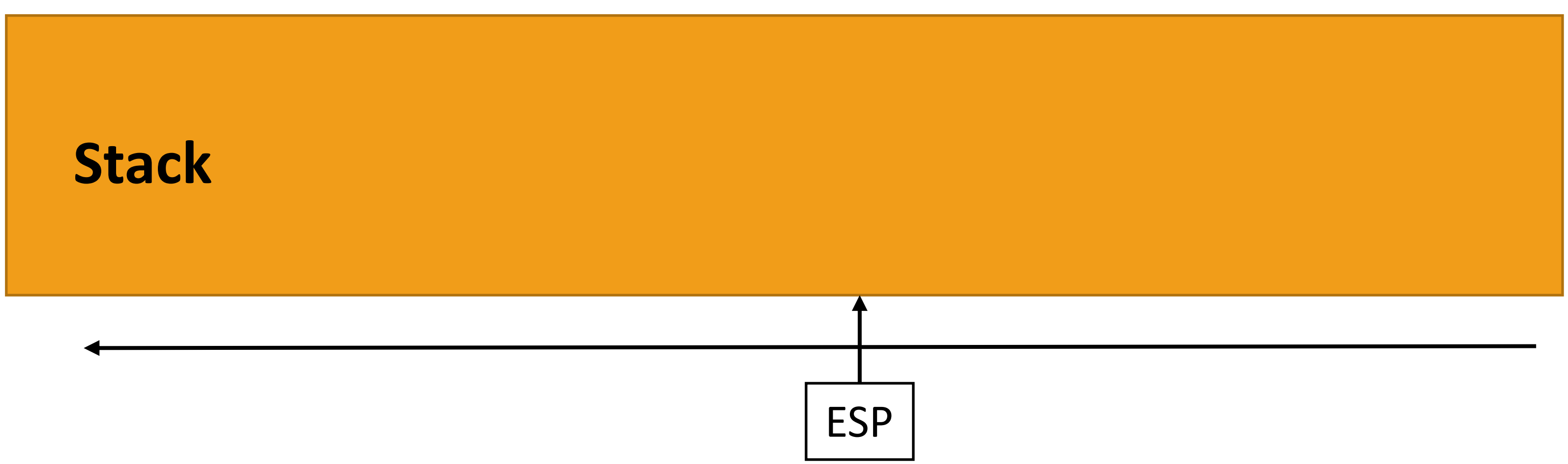
```c
int main(void)
{
        system("/bin/shell");
        return 0;
}
```

```
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 →

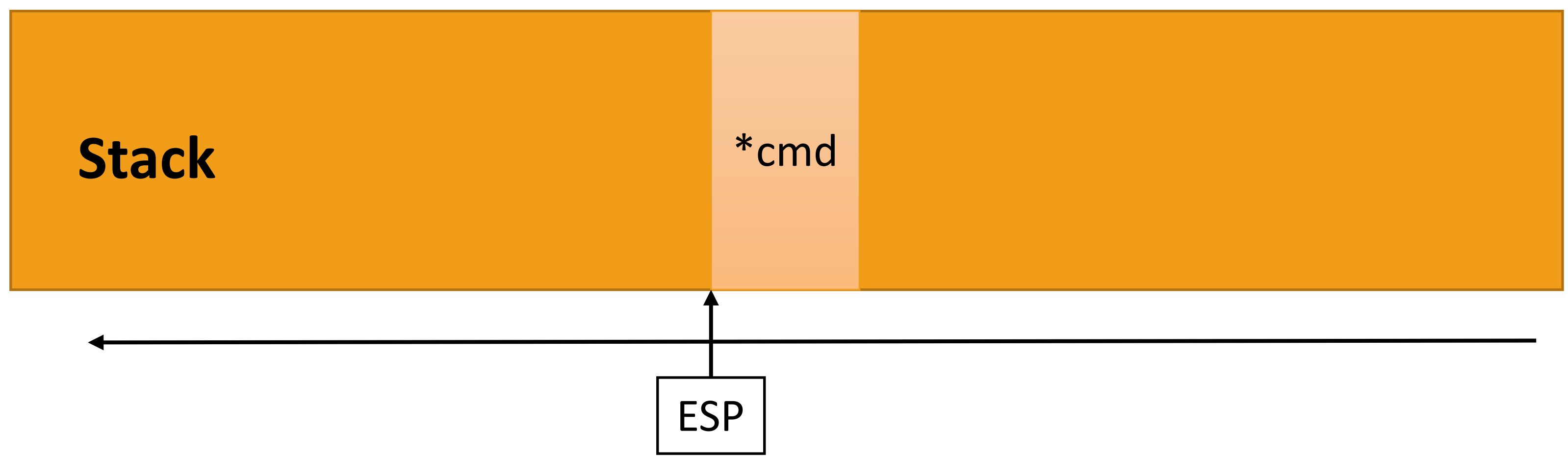
```
804840f:    68 c0 84 04 08    push   $0x80484c0
8048414:    e8 b7 fe ff ff    call   80482d0 <system@plt>
```



Preparing the Stack

EIP →

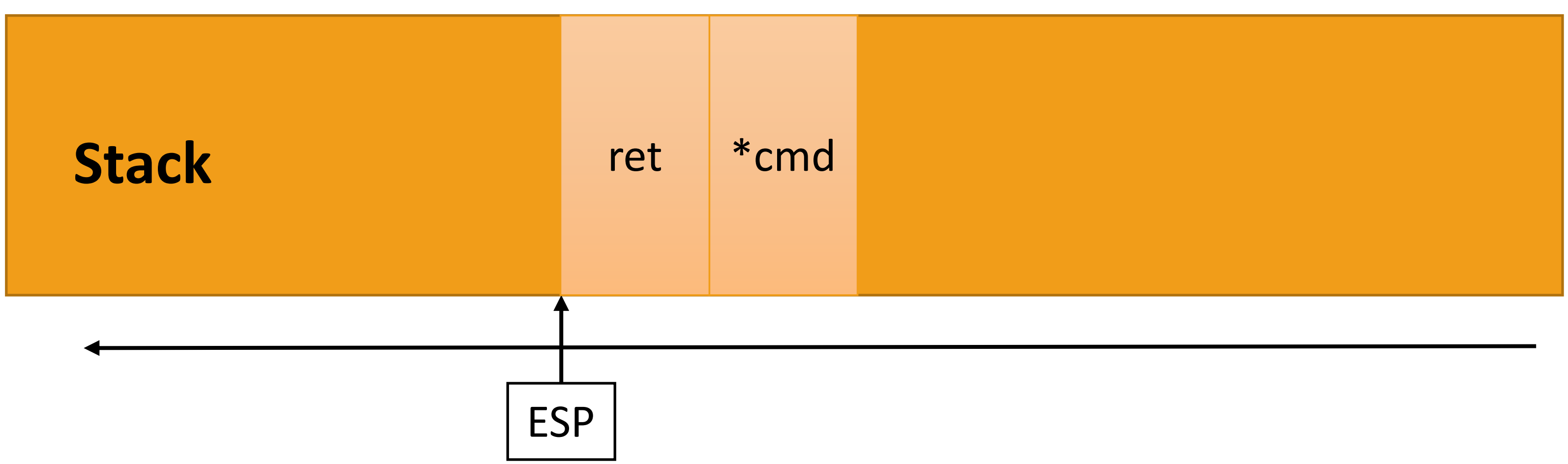
```
804840f:     68 c0 84 04 08     push   $0x80484c0
8048414:     e8 b7 fe ff ff     call   80482d0 <system@plt>
```



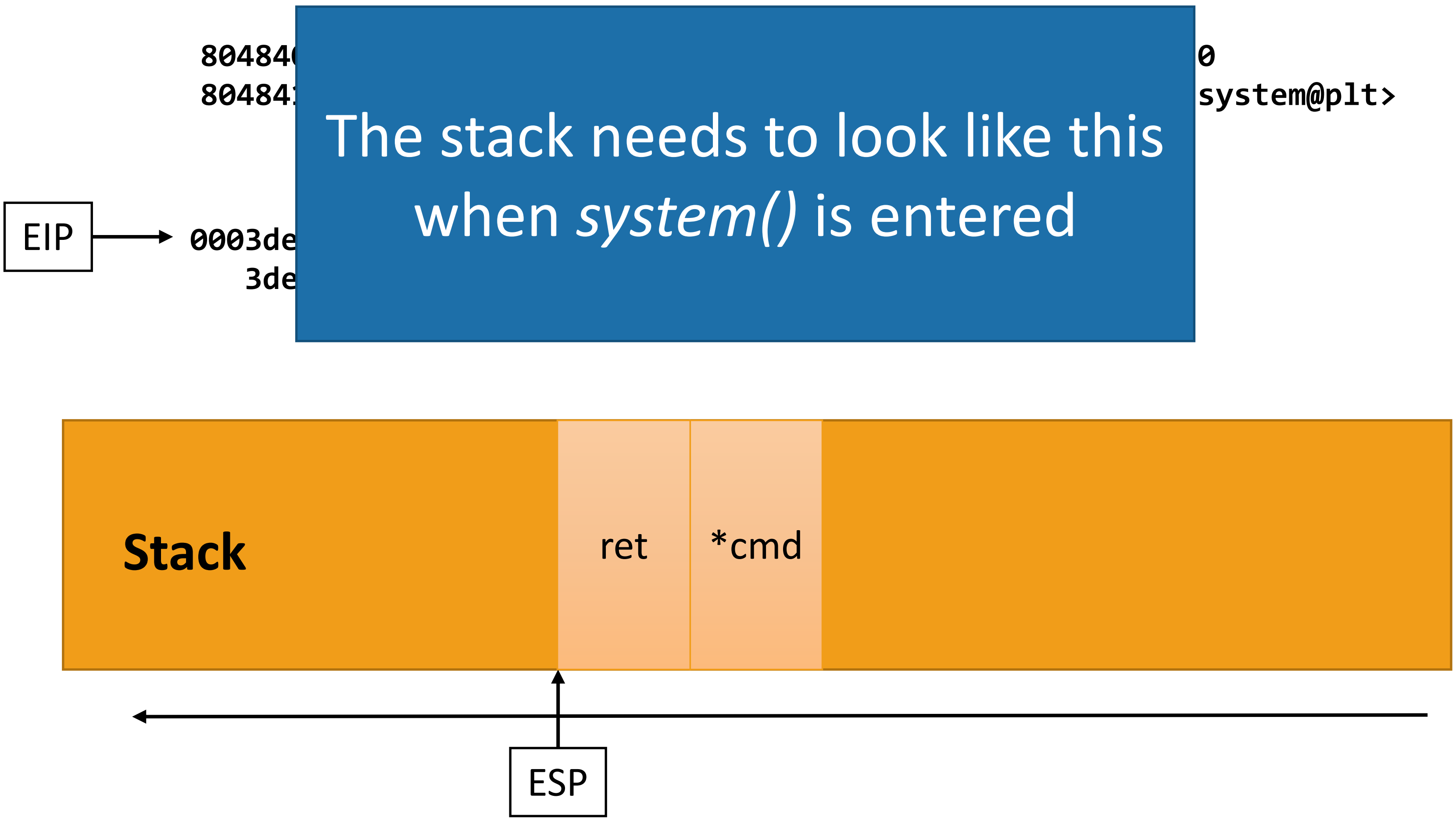
Preparing the Stack

```
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
```

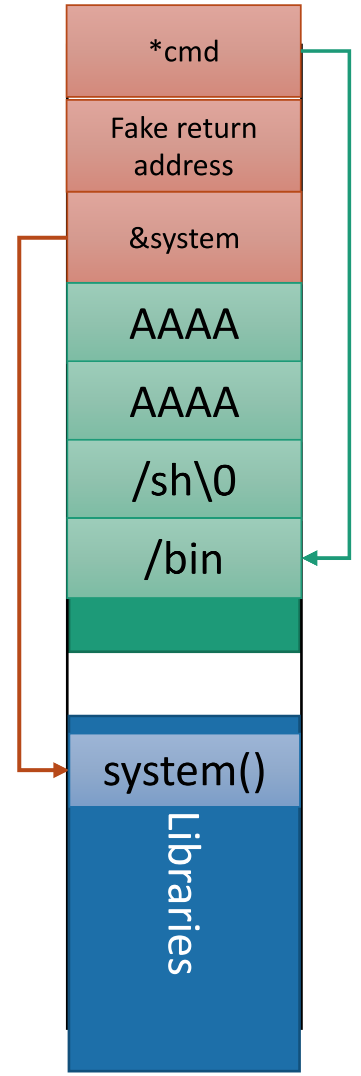


Preparing the Stack



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

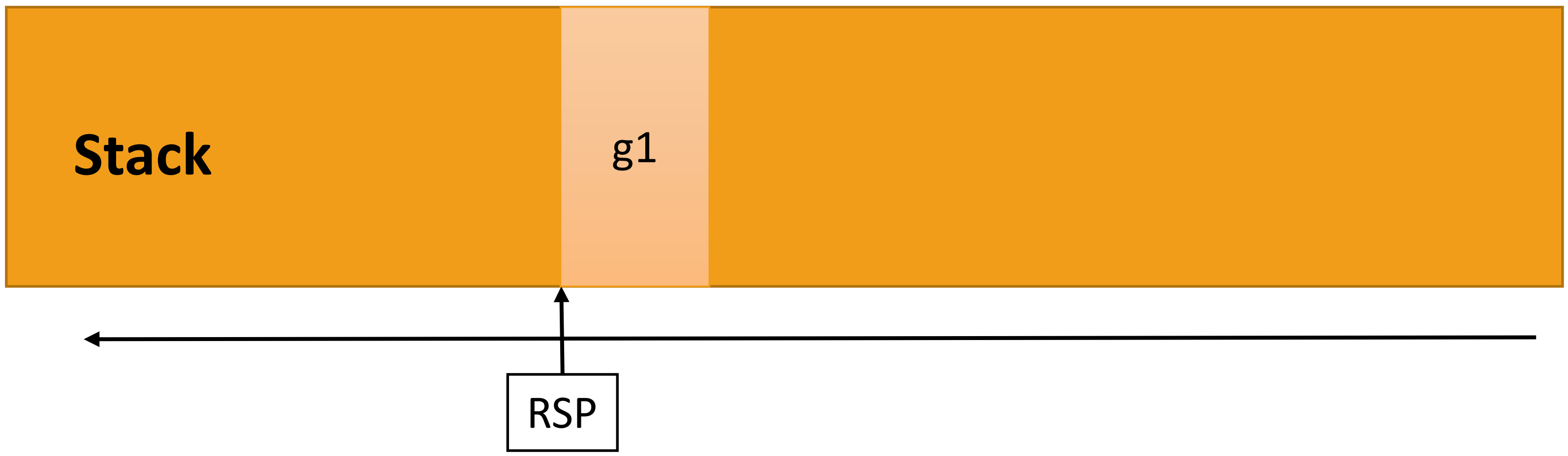
```
0x0000000000405255 : pop rdi ; ret
```

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

Return-to-libc on 64-bit

Redirect control to gadget

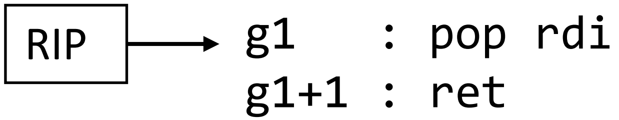
```
g1   : pop rdi
g1+1 : ret
```

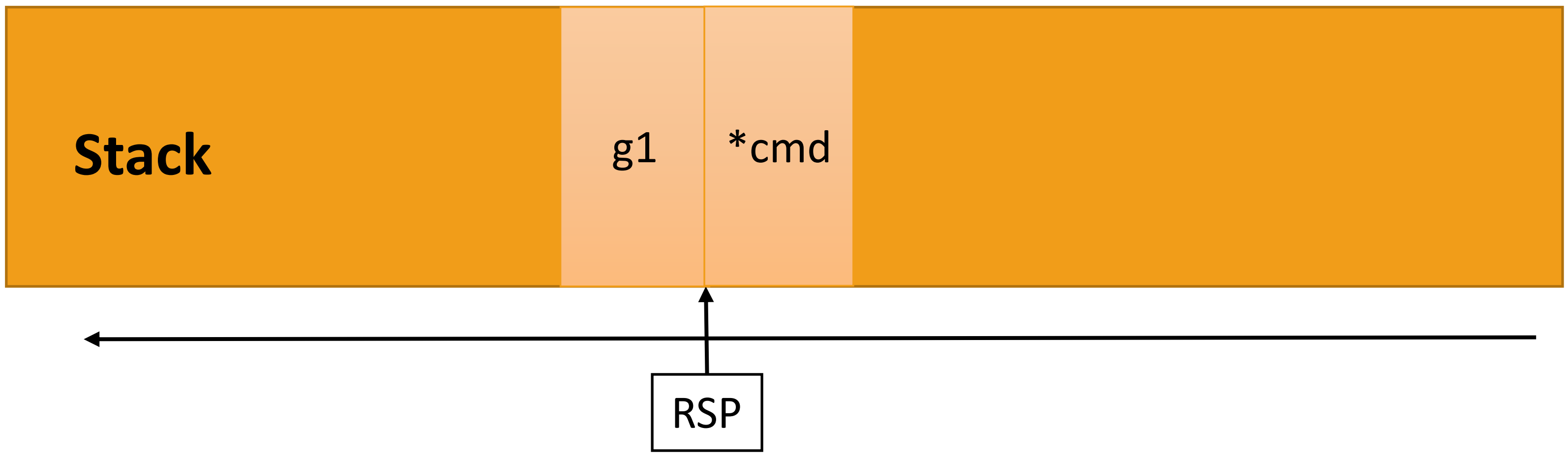


Return-to-libc on 64-bit

Redirect control to gadget

Load argument on register



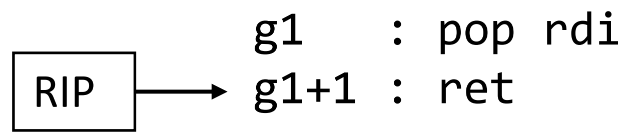


Return-to-libc on 64-bit

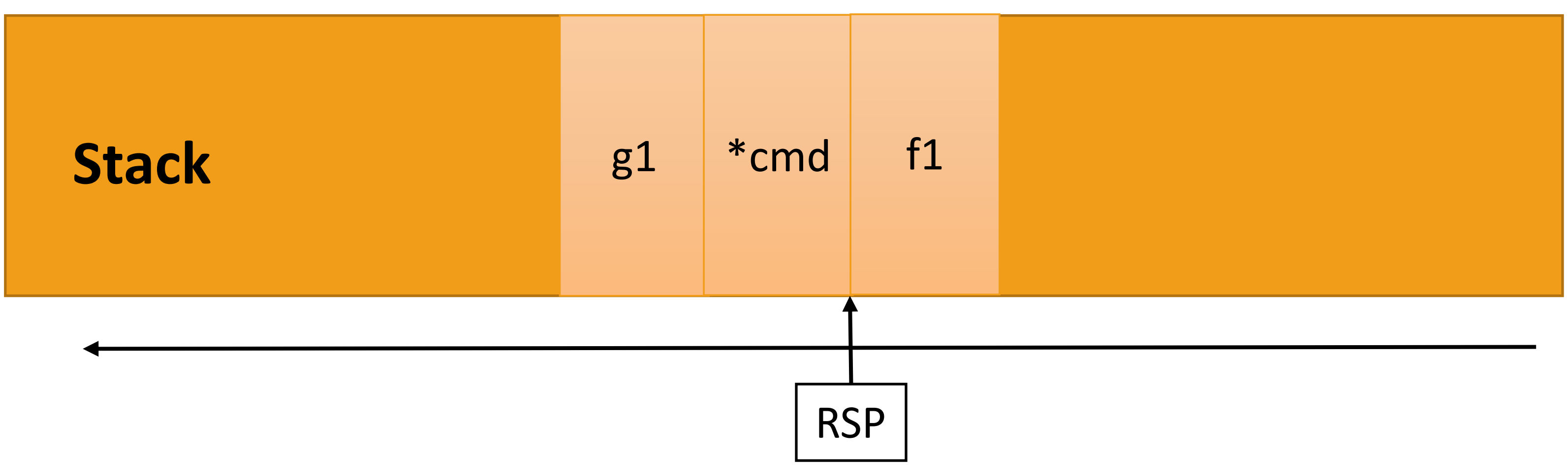
Redirect control to gadget

Load argument on register

Redirect control to libc function




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



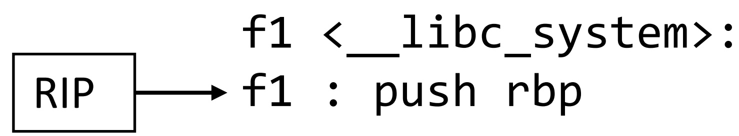
Return-to-libc on 64-bit

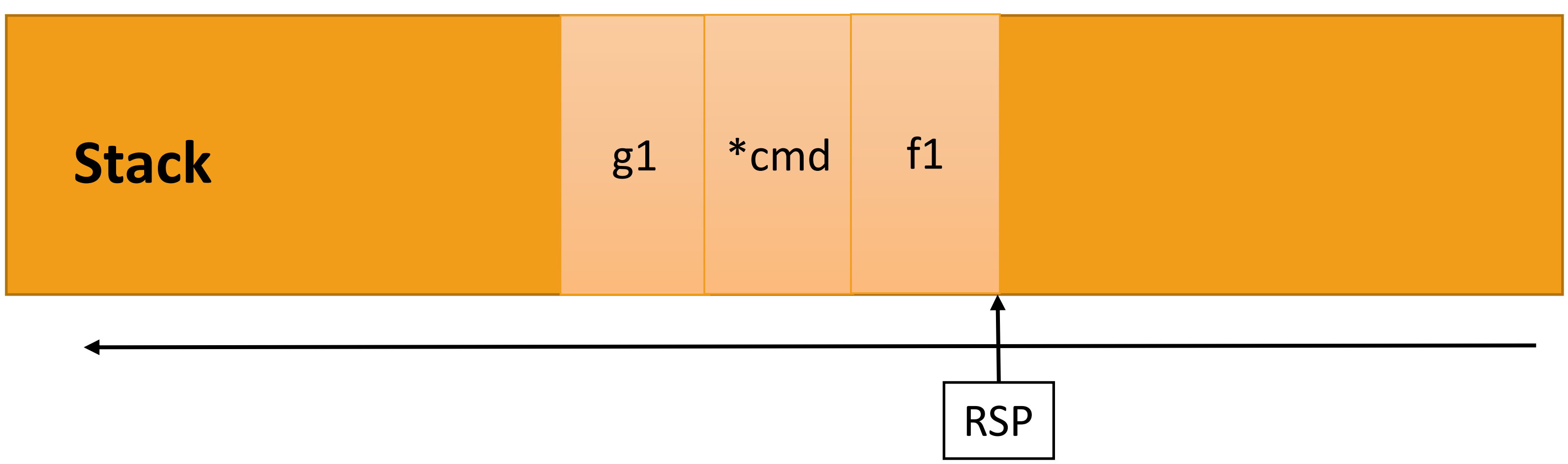
Redirect control to gadget

Load argument on register

Redirect control to libc function

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





Return-to-libc on 64-bit

Redirect control to gadget

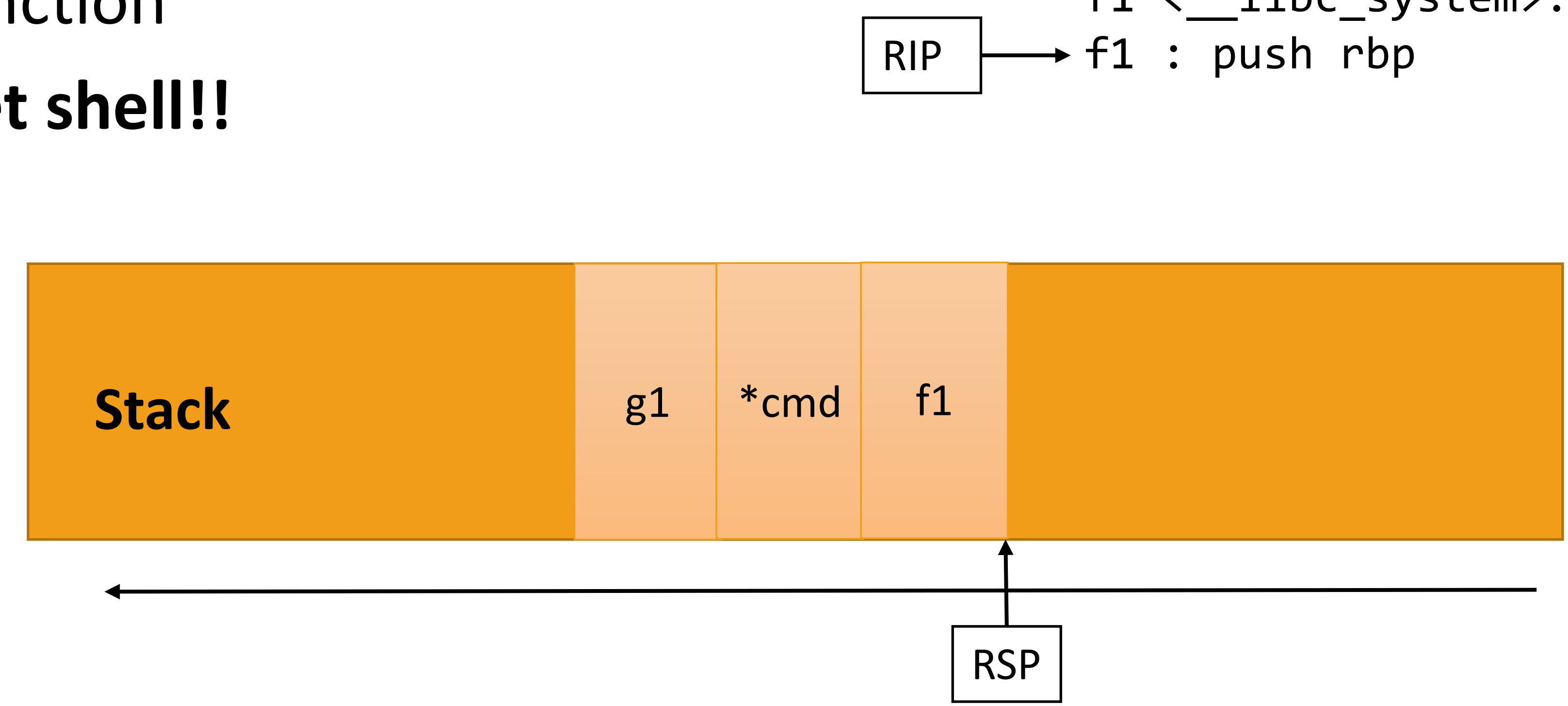
Load argument on register

Redirect control to libc function

Get shell!!

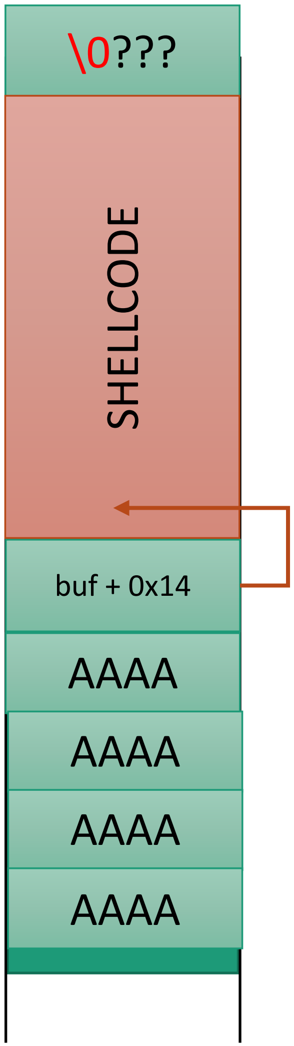
```
g1   : pop rdi
g1+1 : ret
```

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



ASCII Armored Address Space

Shellcode Limitations

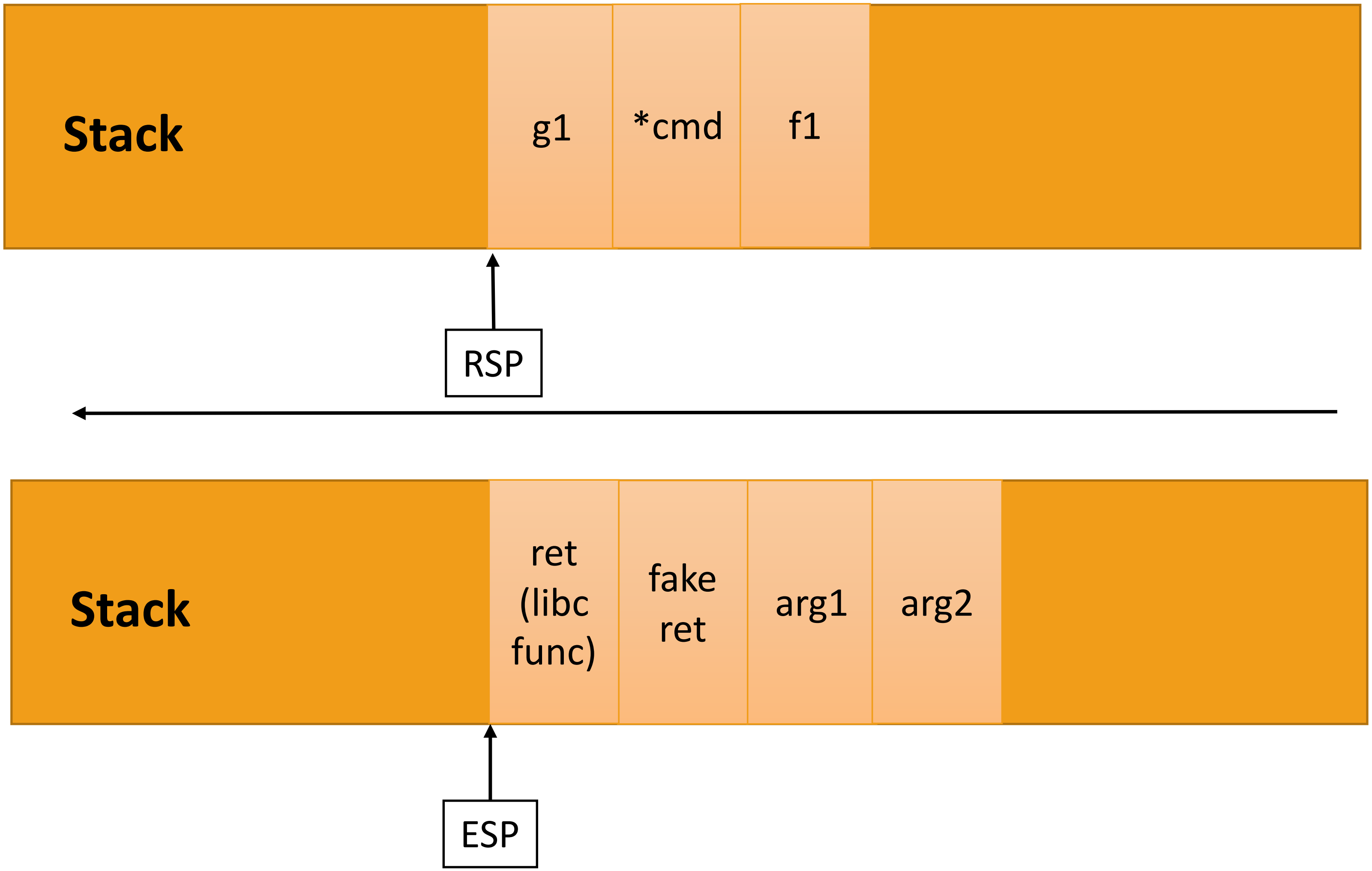


Injected shellcode cannot include a null byte because of strcpy()

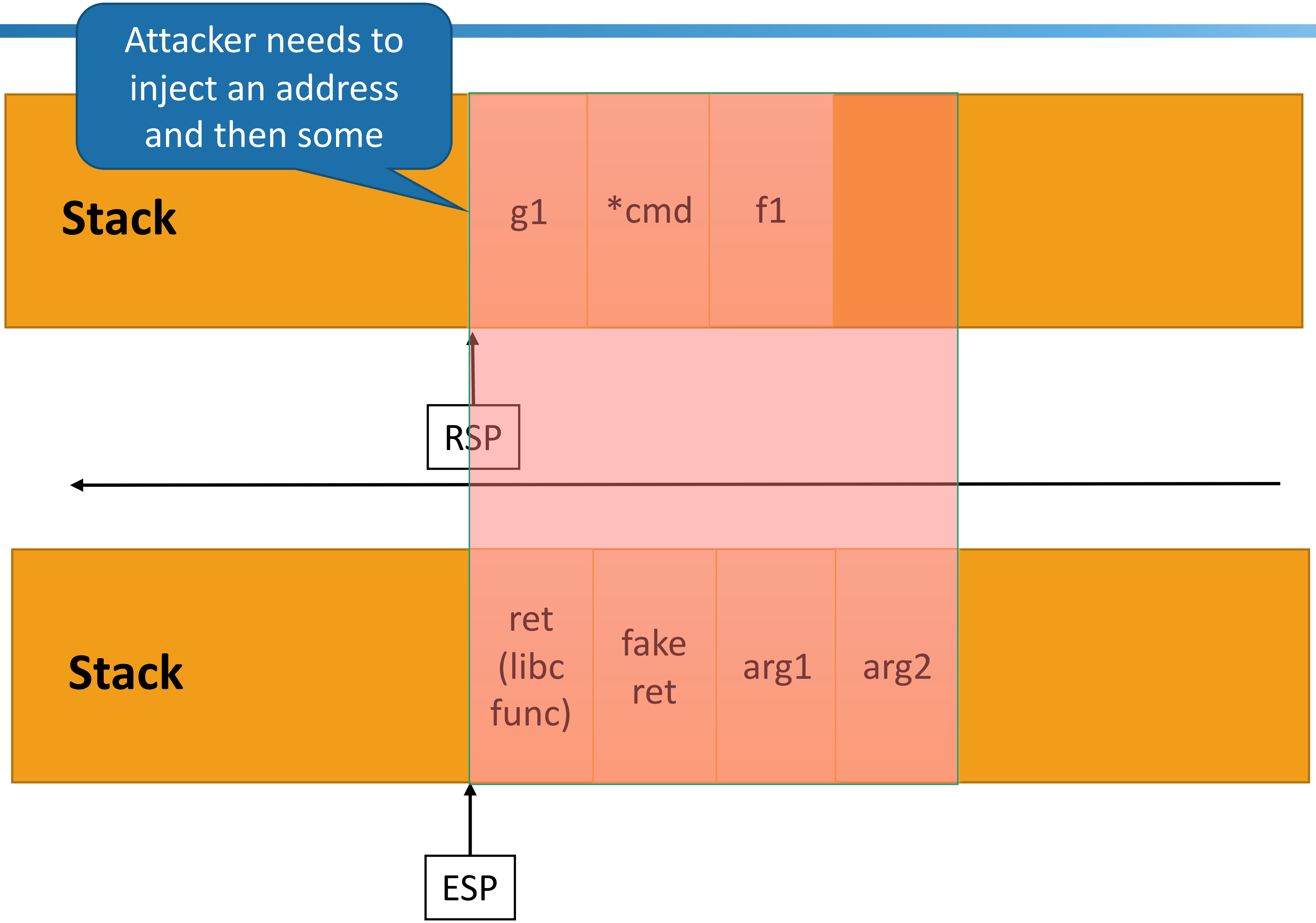
Shellcode needs to be carefully crafted to avoid disallowed bytes

The injected return address cannot contain a zero byte!

ASCII Armored Address Space

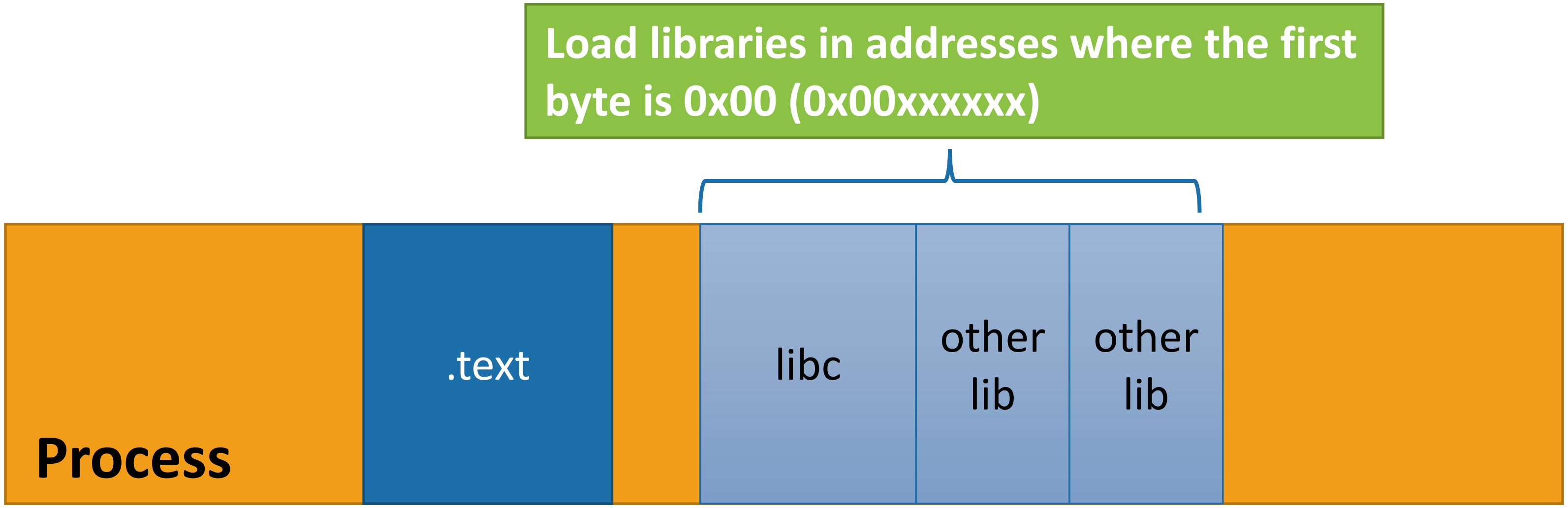


ASCII Armored Address Space

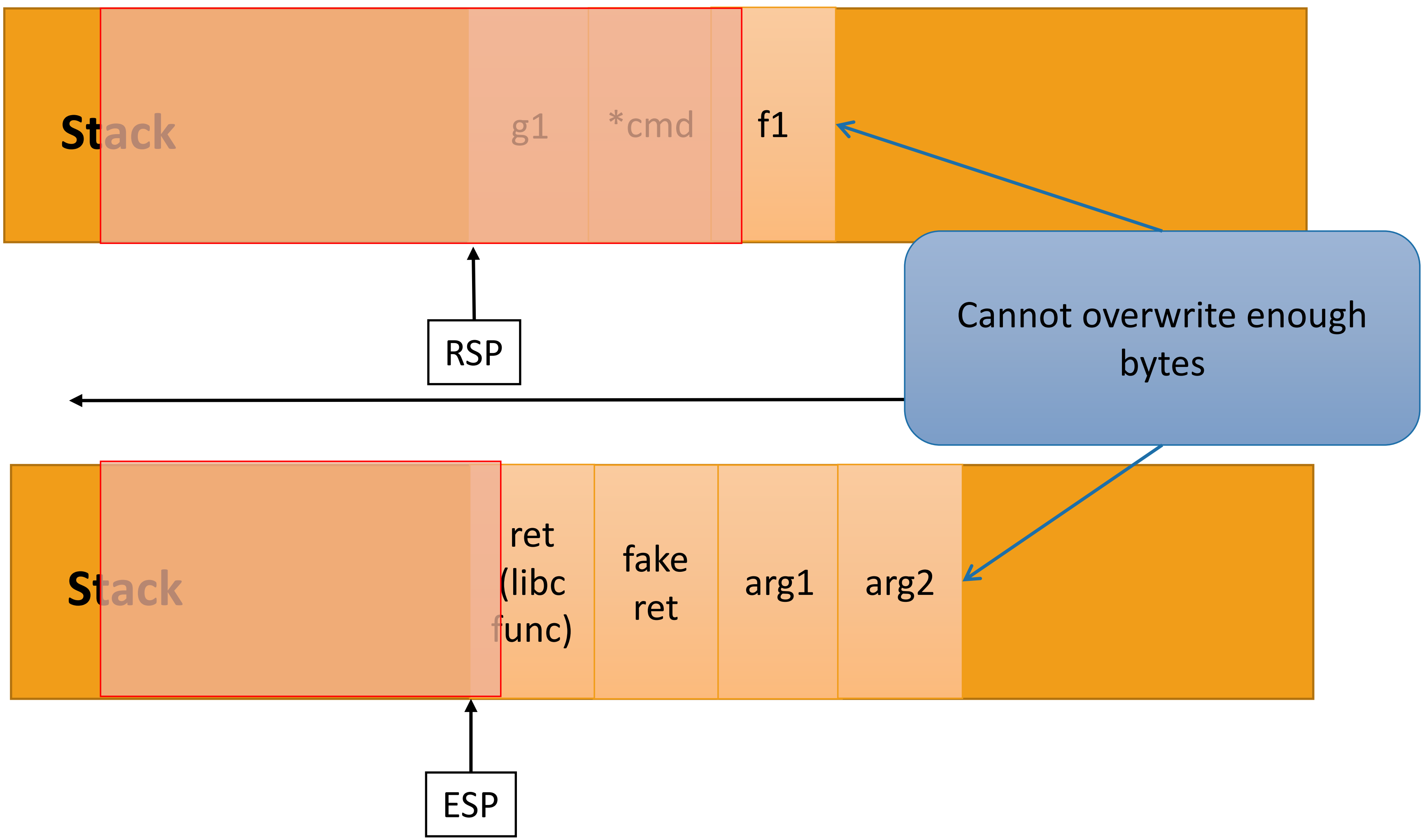


ASCII Armored Address Space

strcpy() stops copying on the first null byte!



ASCII Armored Address Space



Problems

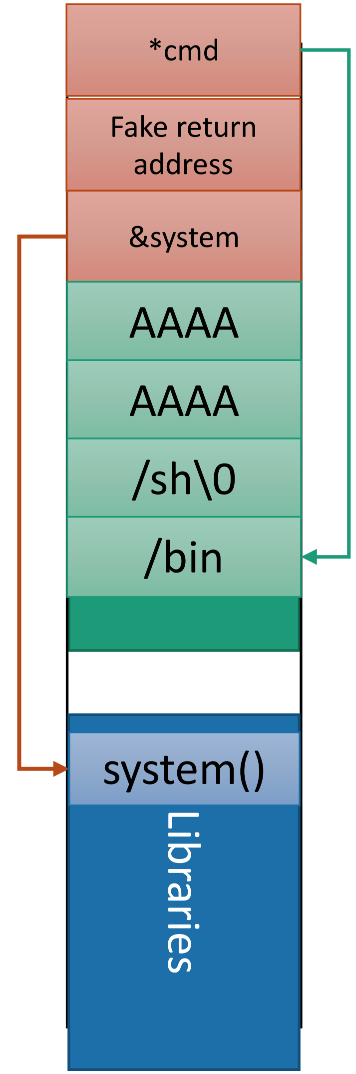
Other methods of copying data may not have the same limitation: memcpy(), gets(), read(), fread(), custom copy routines, etc.

Stackguard & Stackshield

Detecting Corrupted Return Addresses

Attacks can reuse existing code

How about preventing the use of corrupted data to influence RIP?



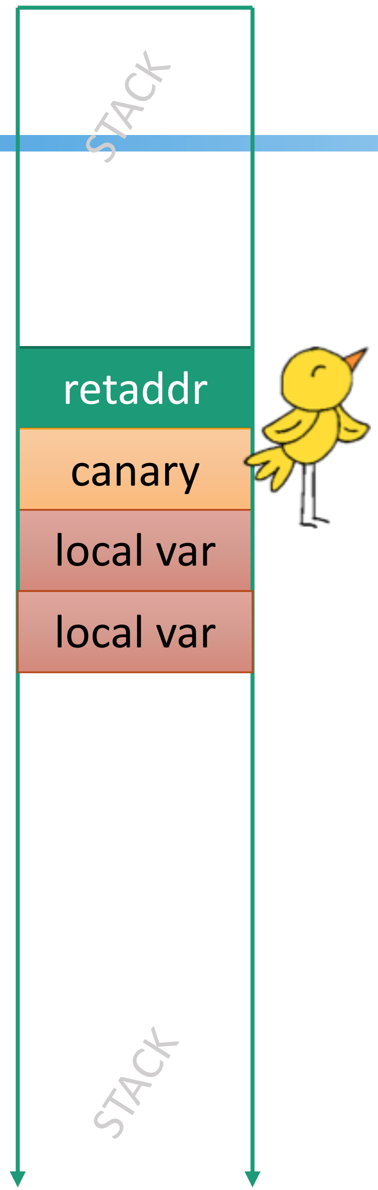
StackGuard

Insert special values, called canaries, between local variables and function return address

Canary values are inserted on function entry

Canaries are verified before a function returns

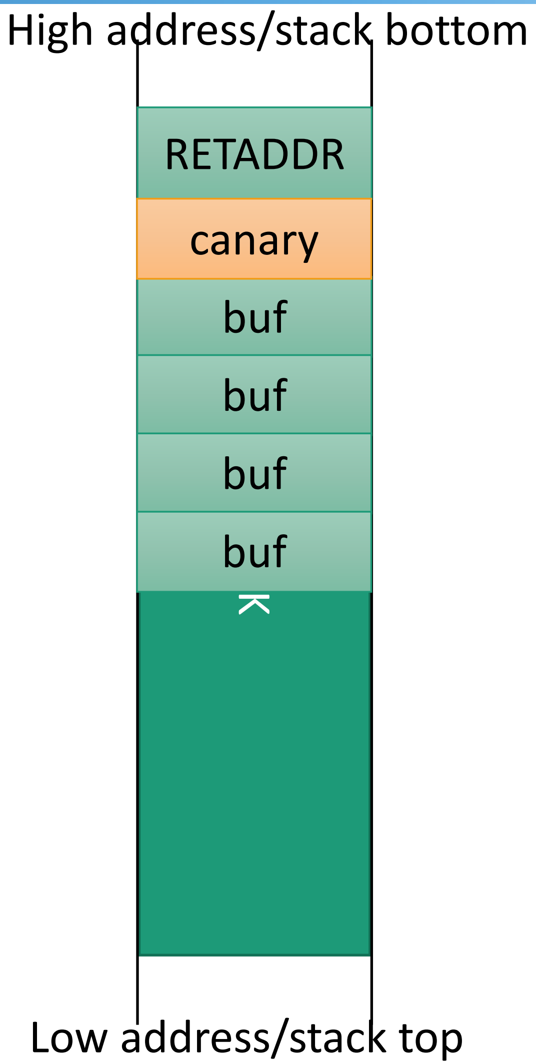
- Program stops if the canary has changed

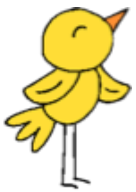


Stack Overflow With Canary

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

./mytest AAAAA

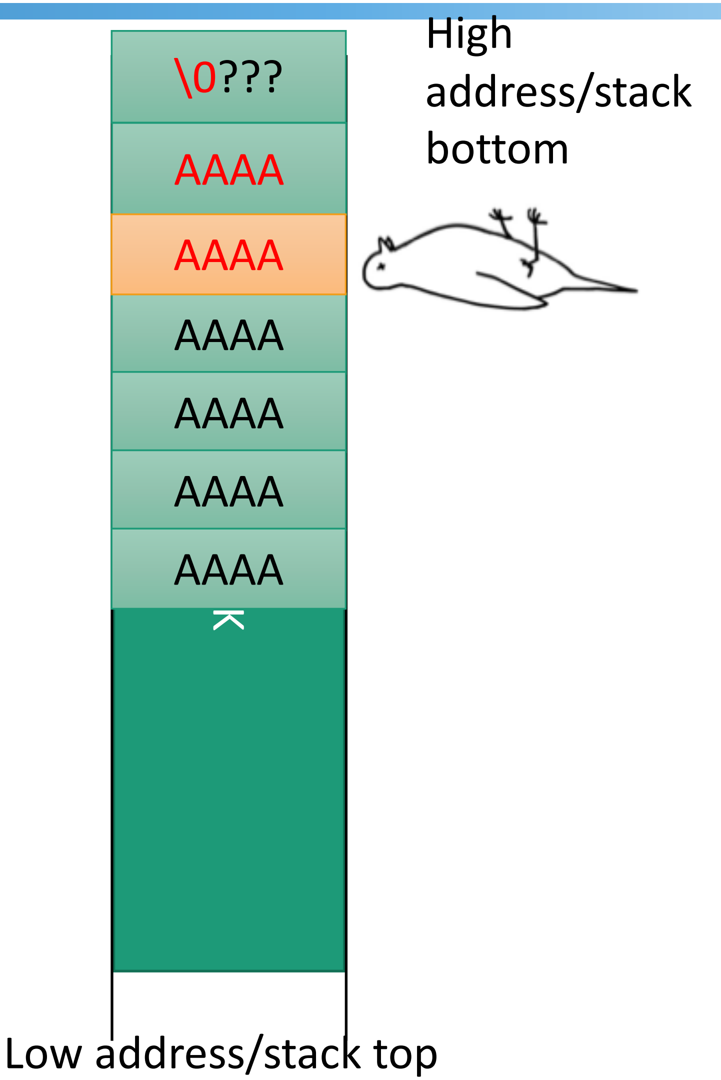




Stack Overflow with Canary

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```

./mytest AAAAAAAAAAAAAAAAAAAAAAAA



Canary Types

Random canary: (used in Visual Studio, gcc, etc.)

- Choose random string at program startup
- Insert canary string into every stack frame
- Verify canary before returning from function
- To corrupt random canary, attacker must learn current random string

Terminator canary:

Canary = 0 (null), newline, linefeed, EOF

- String functions will not copy beyond terminator
- Hence, attacker cannot use string functions to corrupt stack.

From GCC's documentation

-fstack-protector

Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions **with buffers larger than 8 bytes**. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits

Can be disabled with **-fno-stack-protector** flag

Example: C code

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("len: %ld\n", strlen(buf));
        return strlen(buf);
}
```

Example: Compiled Code

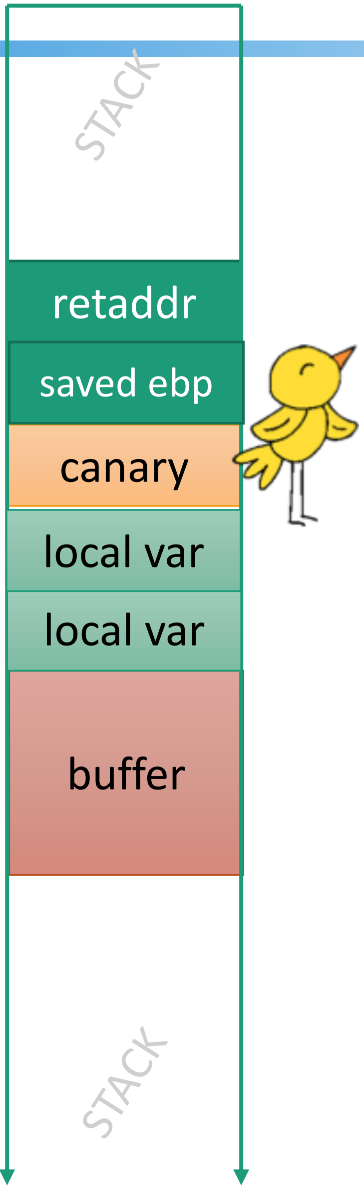
```
0000000000400606 <mytest>:
  400606:       55                      push   %rbp
  400607:       48 89 e5                mov    %rsp,%rbp
  40060a:       48 83 ec 30             sub    $0x30,%rsp
  40060e:       48 89 7d d8             mov    %rdi,-0x28(%rbp)
  400612:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  400619:       00 00
  40061b:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  ...
  40065e:       48 8b 4d f8             mov    -0x8(%rbp),%rcx
  400662:       64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
  400669:       00 00
  40066b:       74 05                   je     400672 <mytest+0x6c>
  40066d:       e8 5e fe ff ff          callq  4004d0 <__stack_chk_fail@plt>
  400672:       c9                      leaveq
  400673:       c3                      retq
```

Store canary

Verify canary

Alignment of Stack Buffers and Canaries

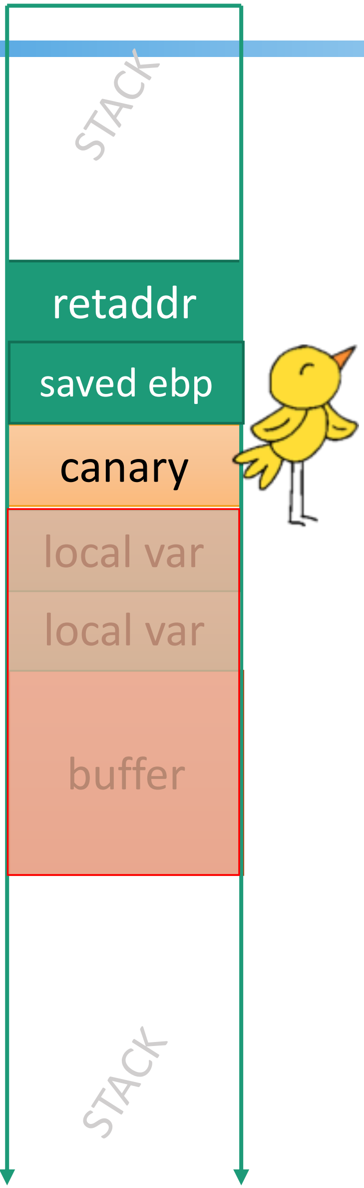
The order of local variables may be important



Alignment of Stack Buffers and Canaries

The order of local variables may be important

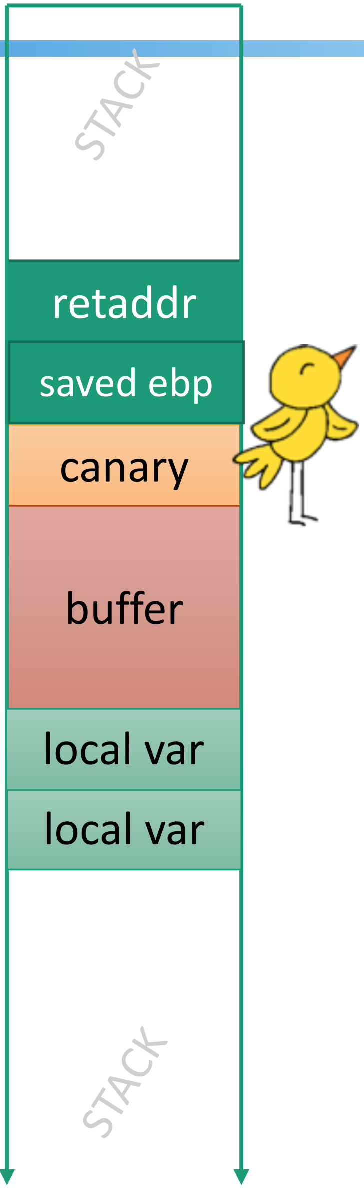
Buffer overflows could allow important local variables to be controlled



Alignment of Stack Buffers and Canaries

Place canary between buffer and saved ebp/return address

The compiler may not always be able to align stack variables "ideally"



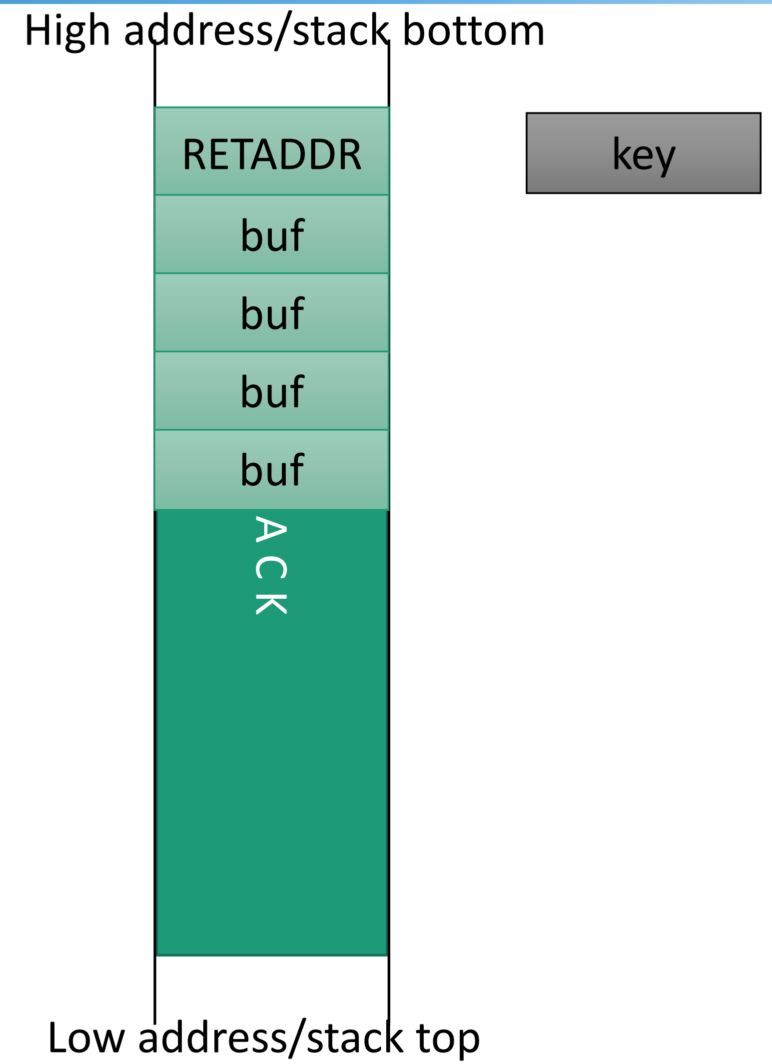
StackShield

Address obfuscation instead of canary

Encrypt return address on stack by XORing with random string

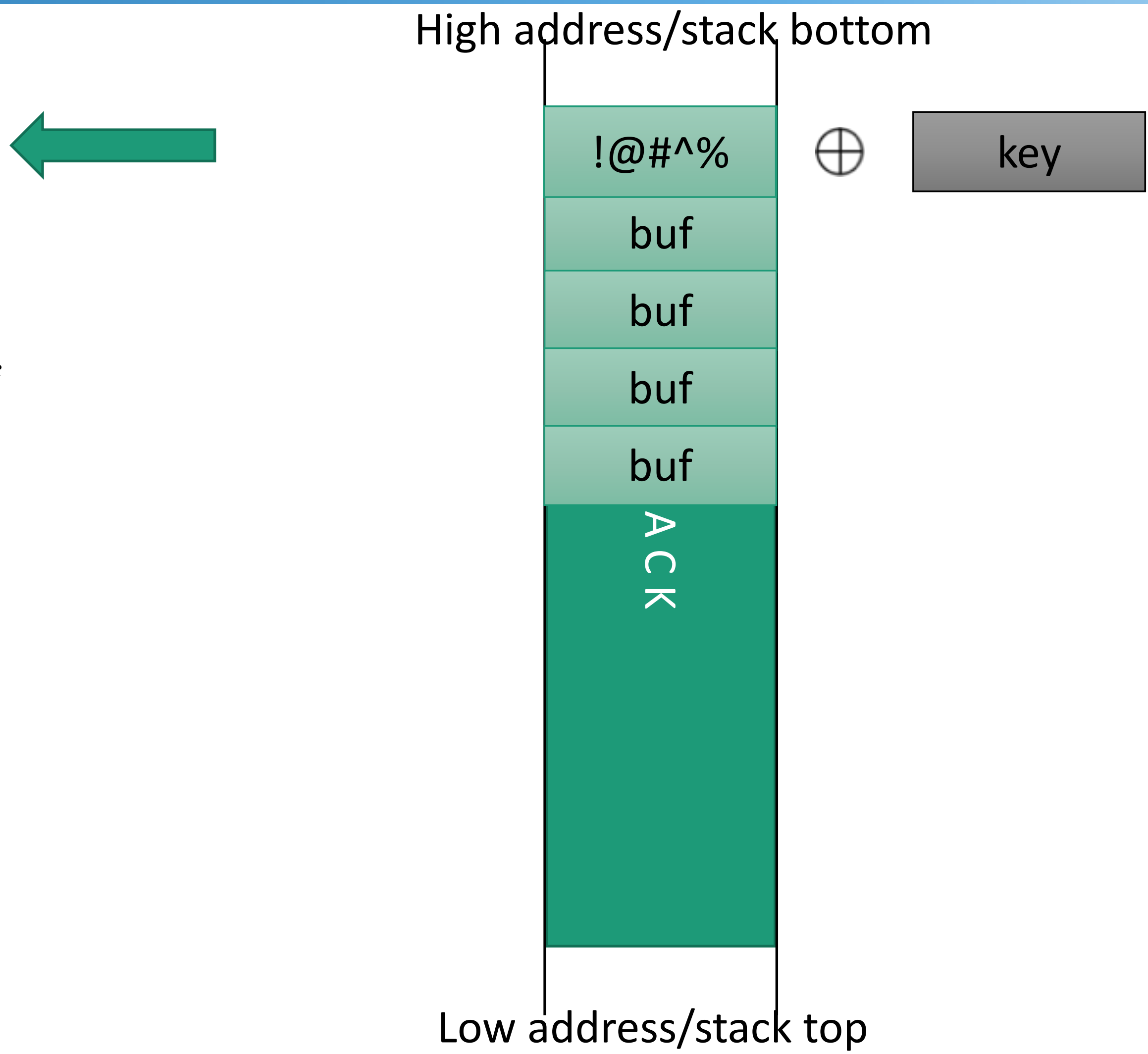
Decrypt just before returning from function

Attacker needs decryption key to set return address to desired value.



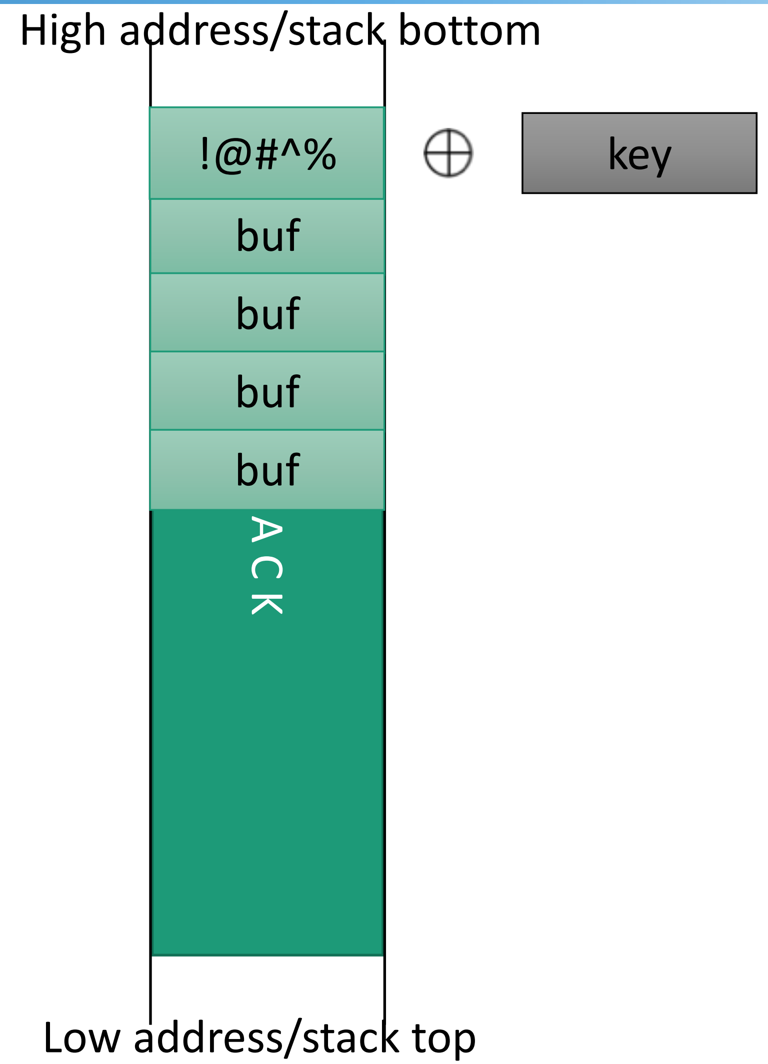
Example: StackShield

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```



Example: StackShield

```
int mytest(char *str)
{
        char buf[16];

        strcpy(buf, str);

        printf("%s\n", buf);

        return strlen(buf);
}
```



Problems

Canaries can be omitted in small functions or non-string buffers

Canaries/keys can be leaked

Bugs may leave canaries untouched

Heap Protections

Heap Protections

Heap Arbitrary Writes

n->next->prev = n->prev;

n->prev->next = n->next;

Facts About DLinked Lists

n->prev->next == n

n->next->prev == n

If these are violated a corruption has occurred!

Other Protections

Separating metadata from chunks

Adding canary type values

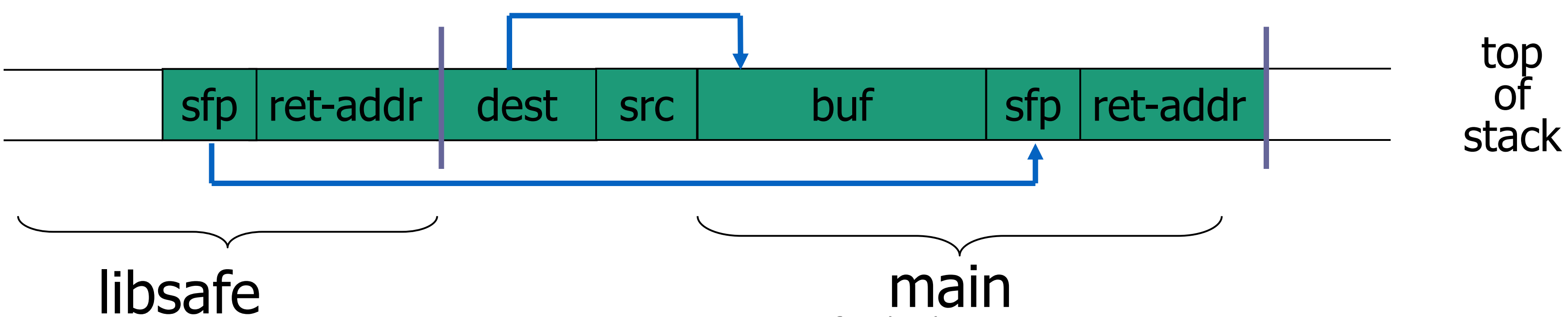
Boundary Checking

Run time checking: Libsafe

Dynamically loaded library

Intercepts calls to strcpy (dest, src)

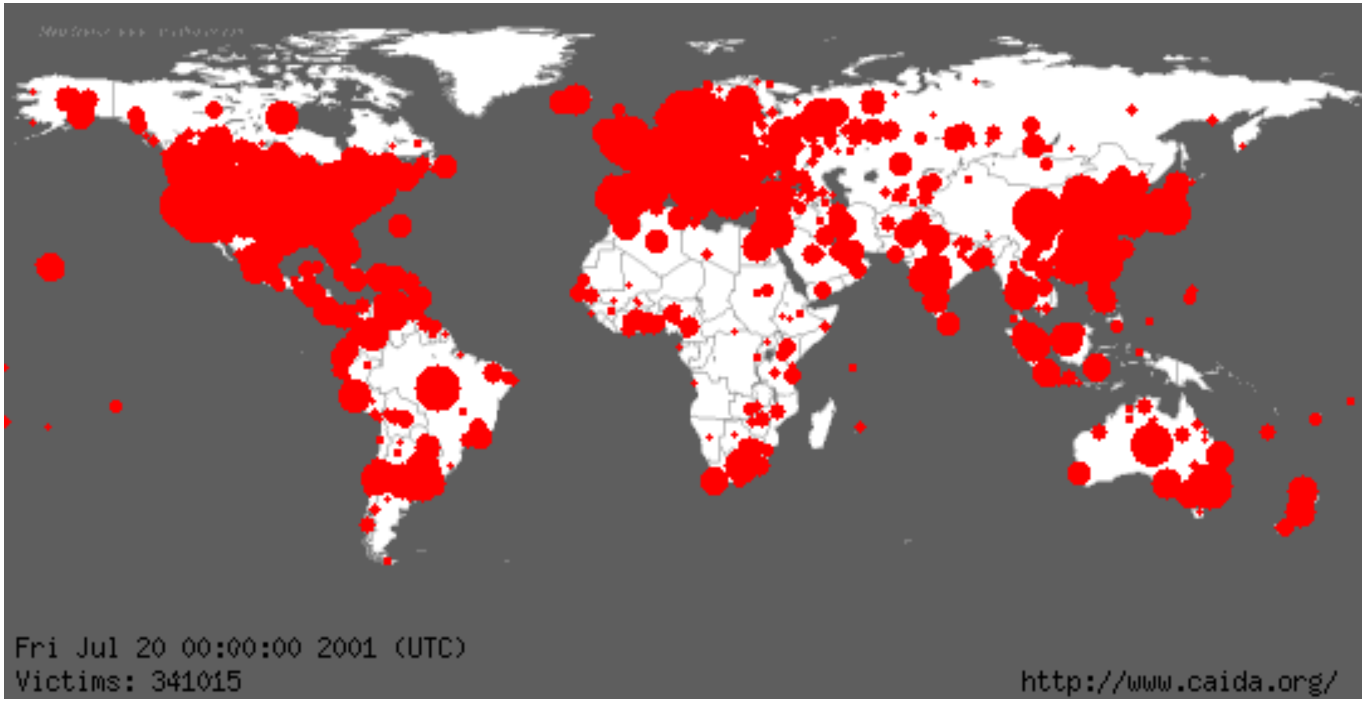
- Validates sufficient space in current stack frame:
  |frame-pointer – dest| > strlen(src)
- If so, does strcpy.
  Otherwise, terminates application.



Address-space Layout Randomization (ASLR)

One Attack Fits All (Lack of Diversity)

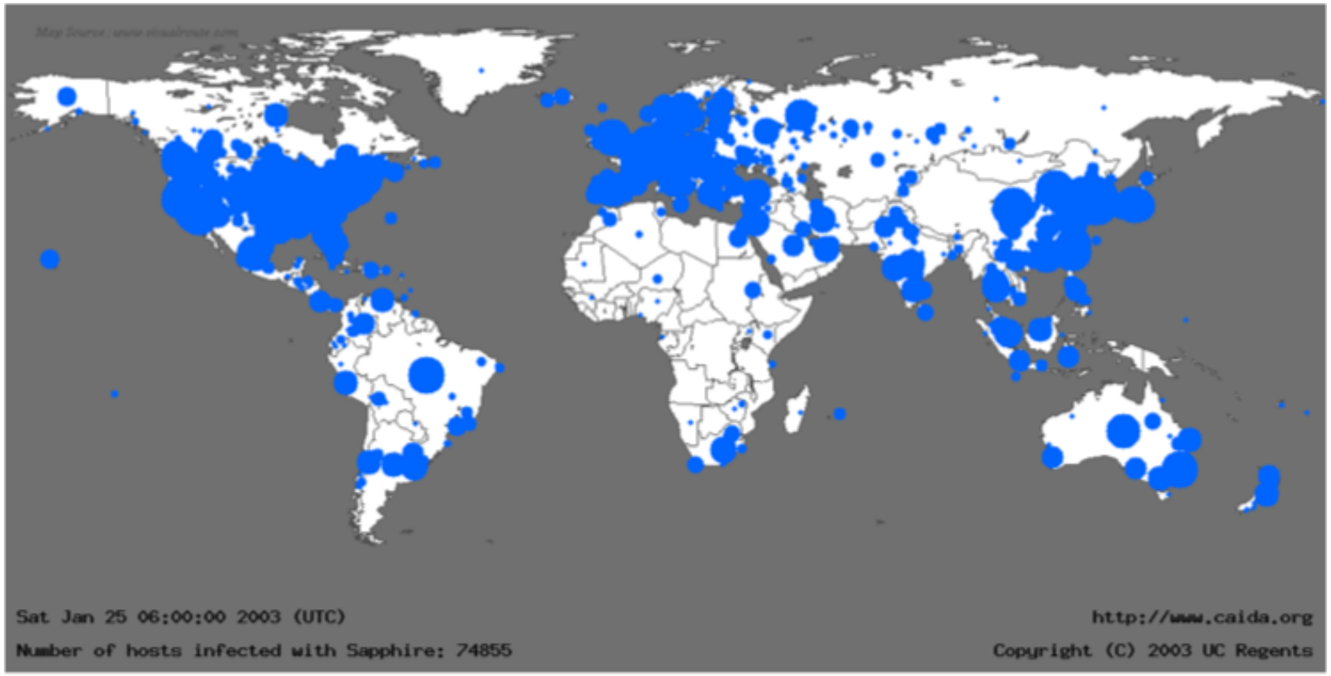
CodeRed worm exploits an MS IIS web server buffer overflow on July 2001



Infections after 24 hours

One Attack Fits All (Lack of Diversity)

Slammer worm exploits an MS SQL server buffer overflow on January 2003

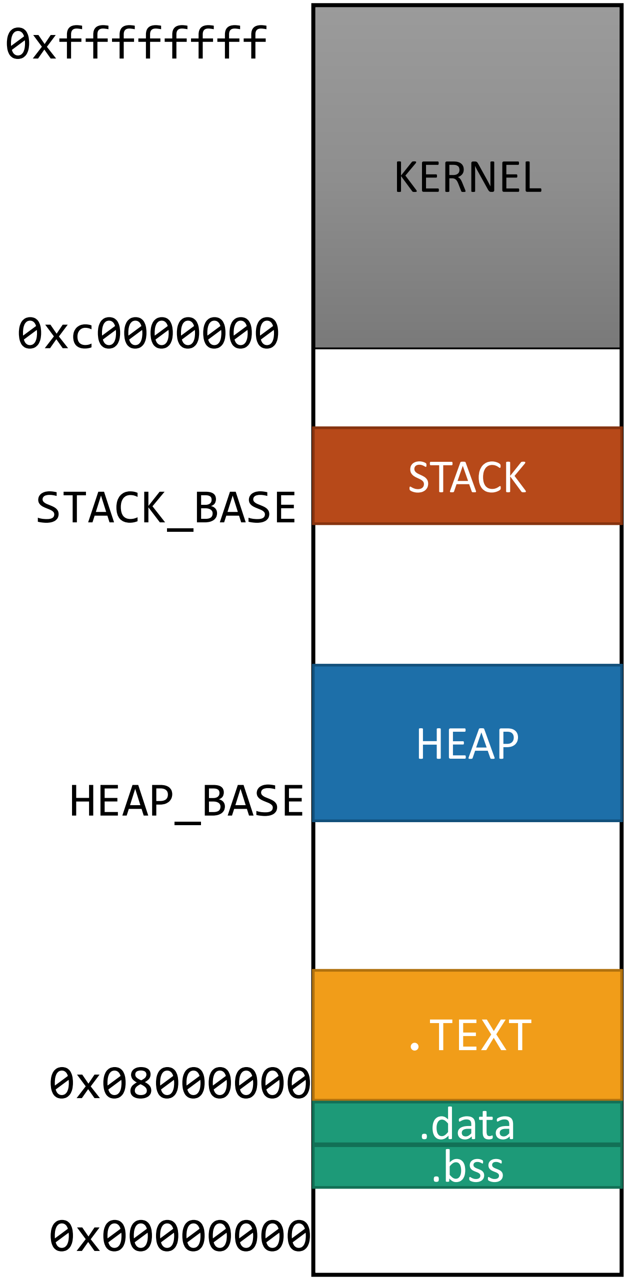


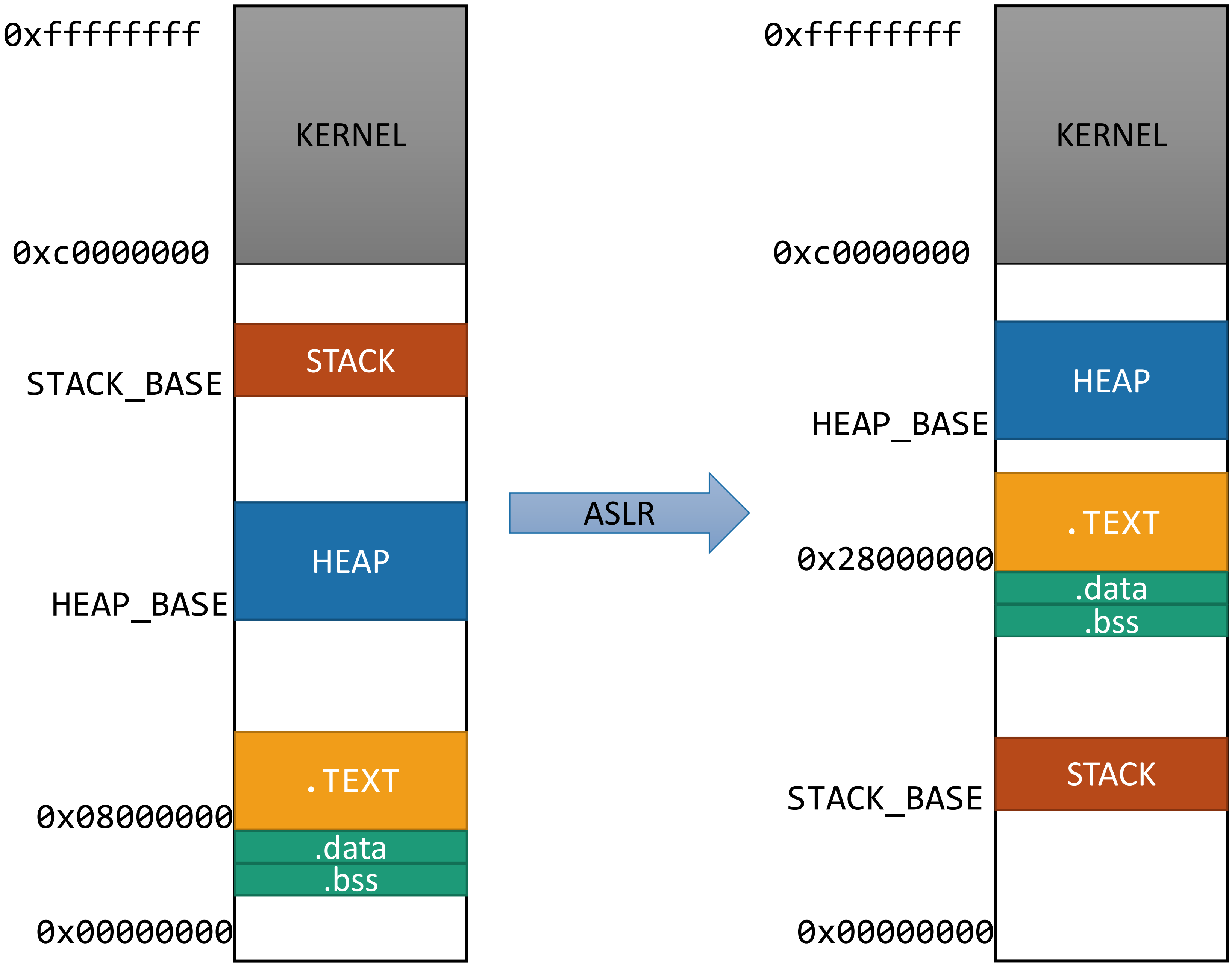
Infections after 30 minutes

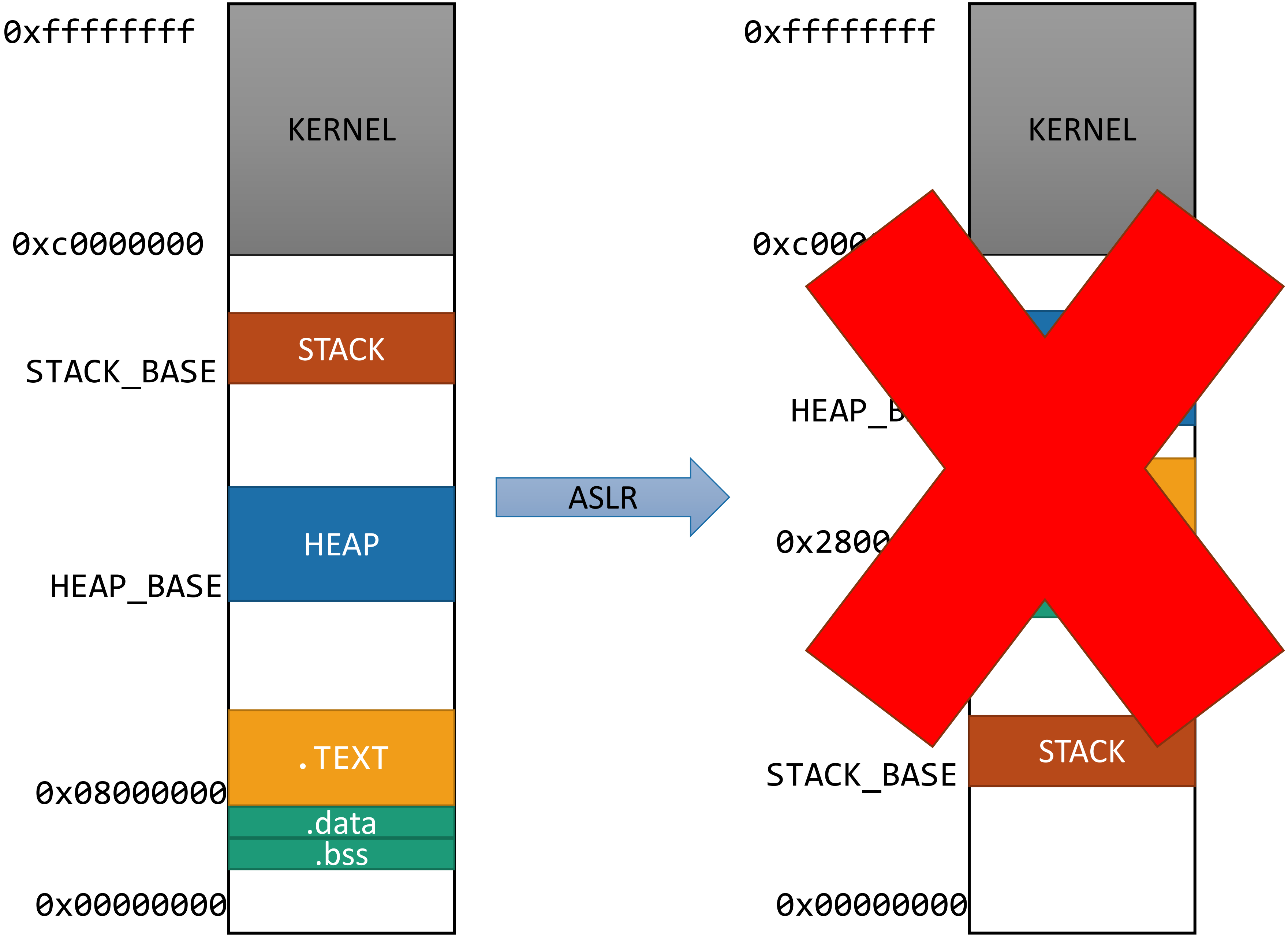
Enter Address Space Layout Randomization

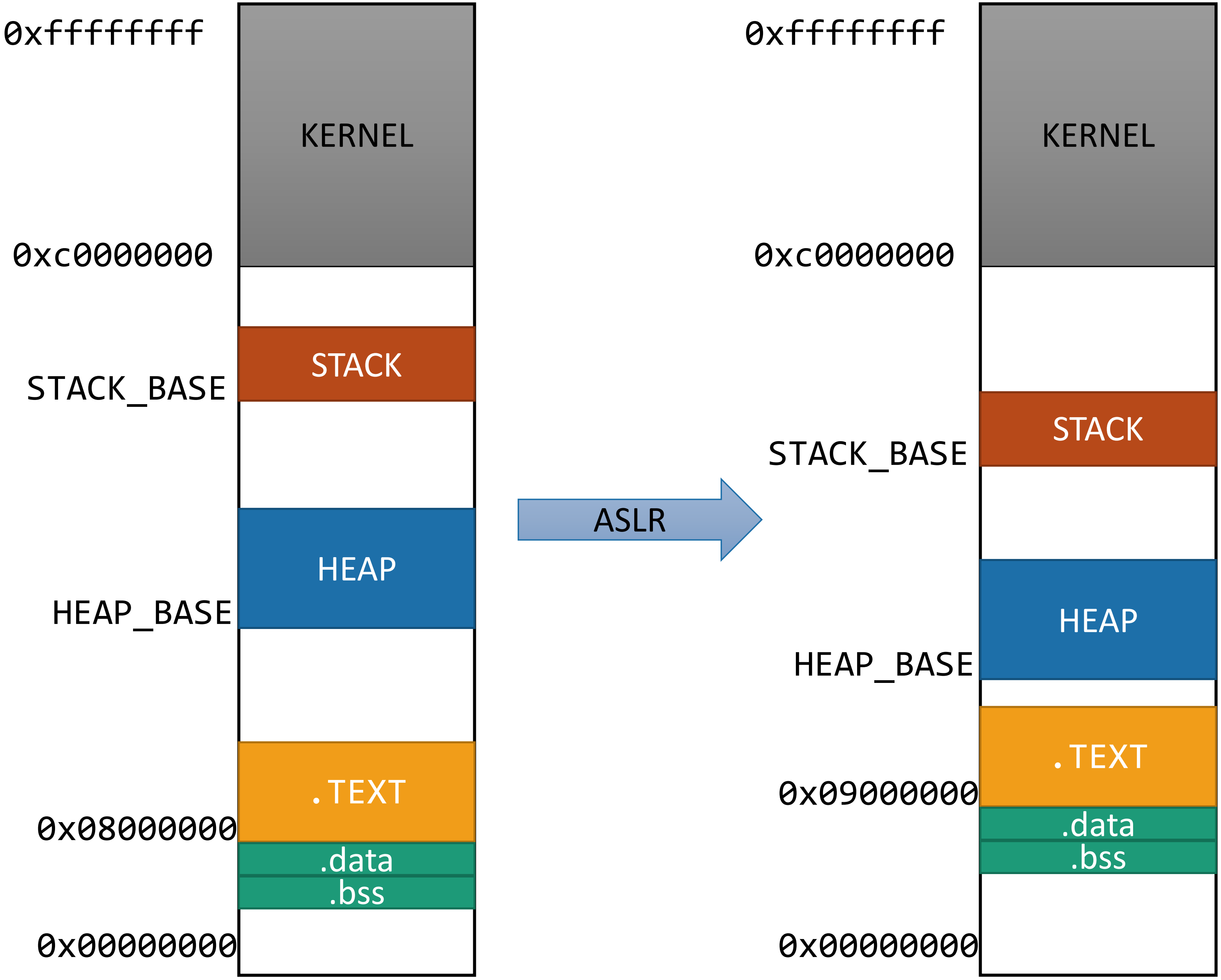
Disrupt exploits by:

- Randomly choose base address of stack, heap, and code segments
- Randomize location of Global Offset Table


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


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


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

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

Brute-forcing ASLR

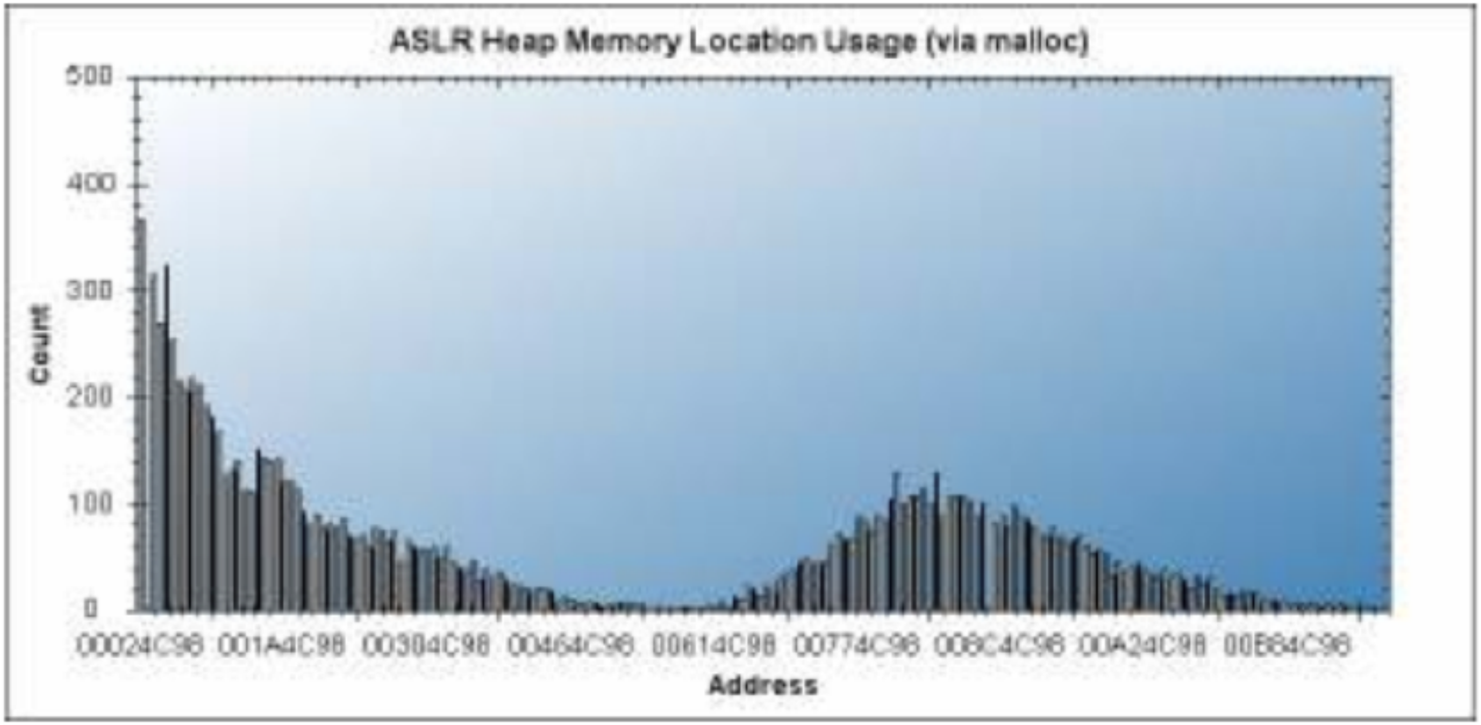
Sometimes only some of the bits in randomization are effective

Implementation uses randomness improperly → distribution of heap bases is biased

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

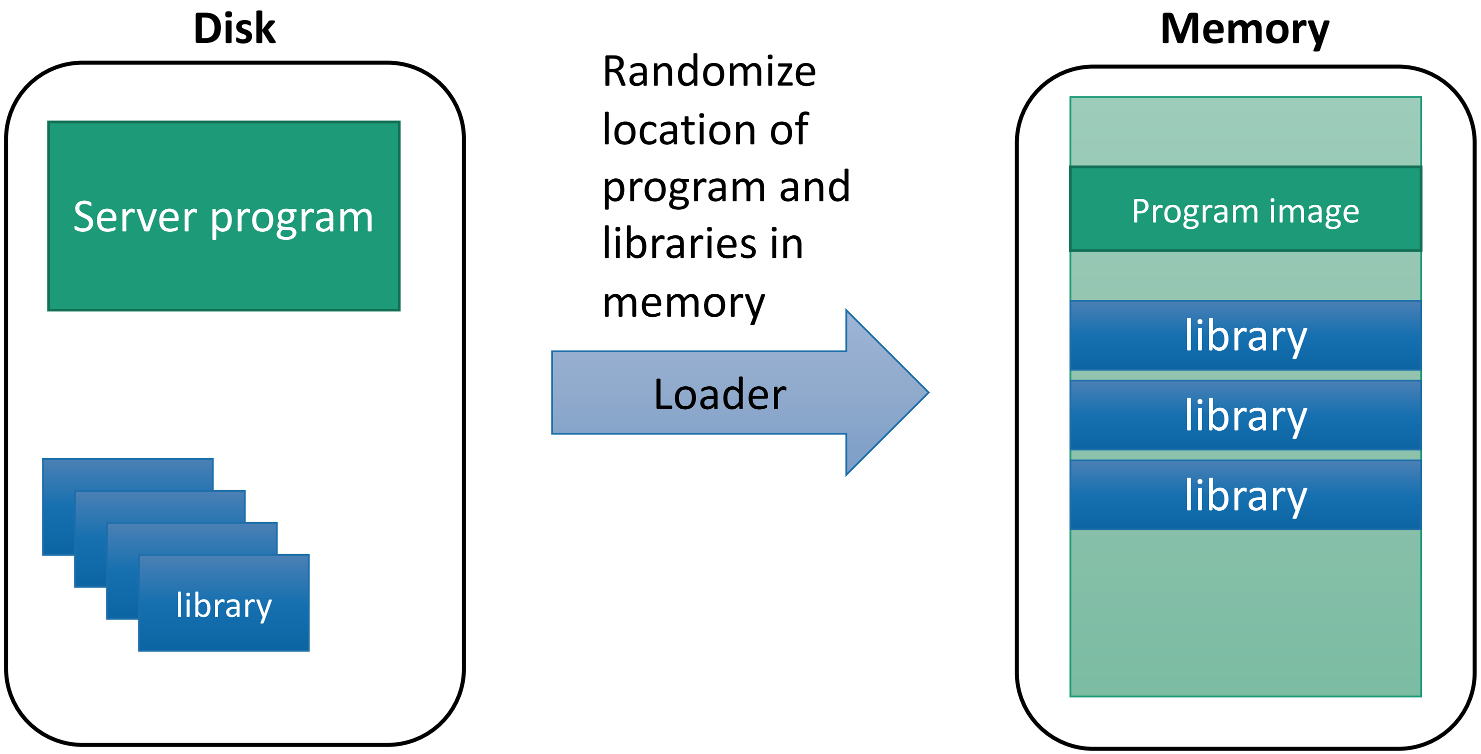
- https://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf

Biased Selection of Heap Base Address



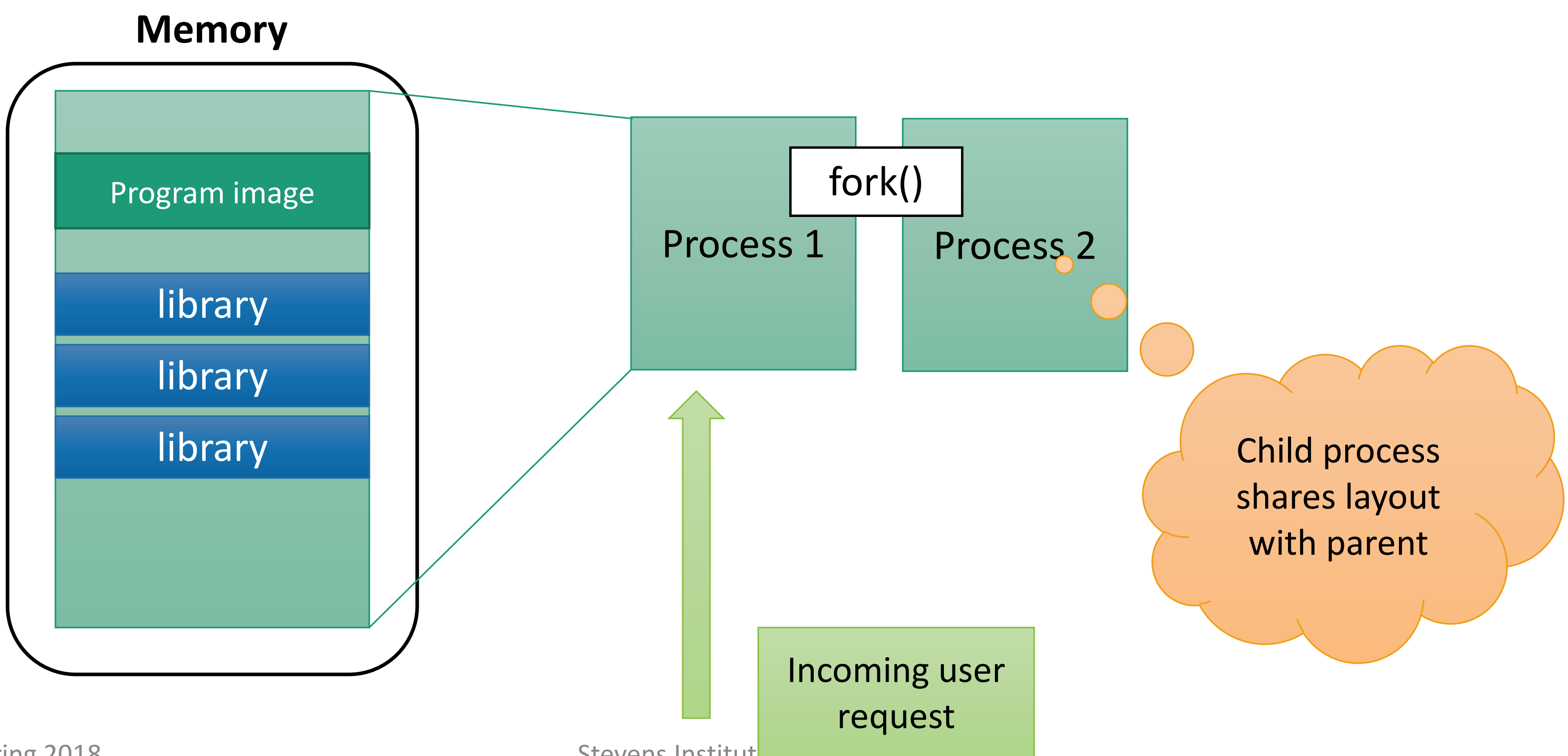
Brute-forcing ASLR

Exploiting server software using fork()



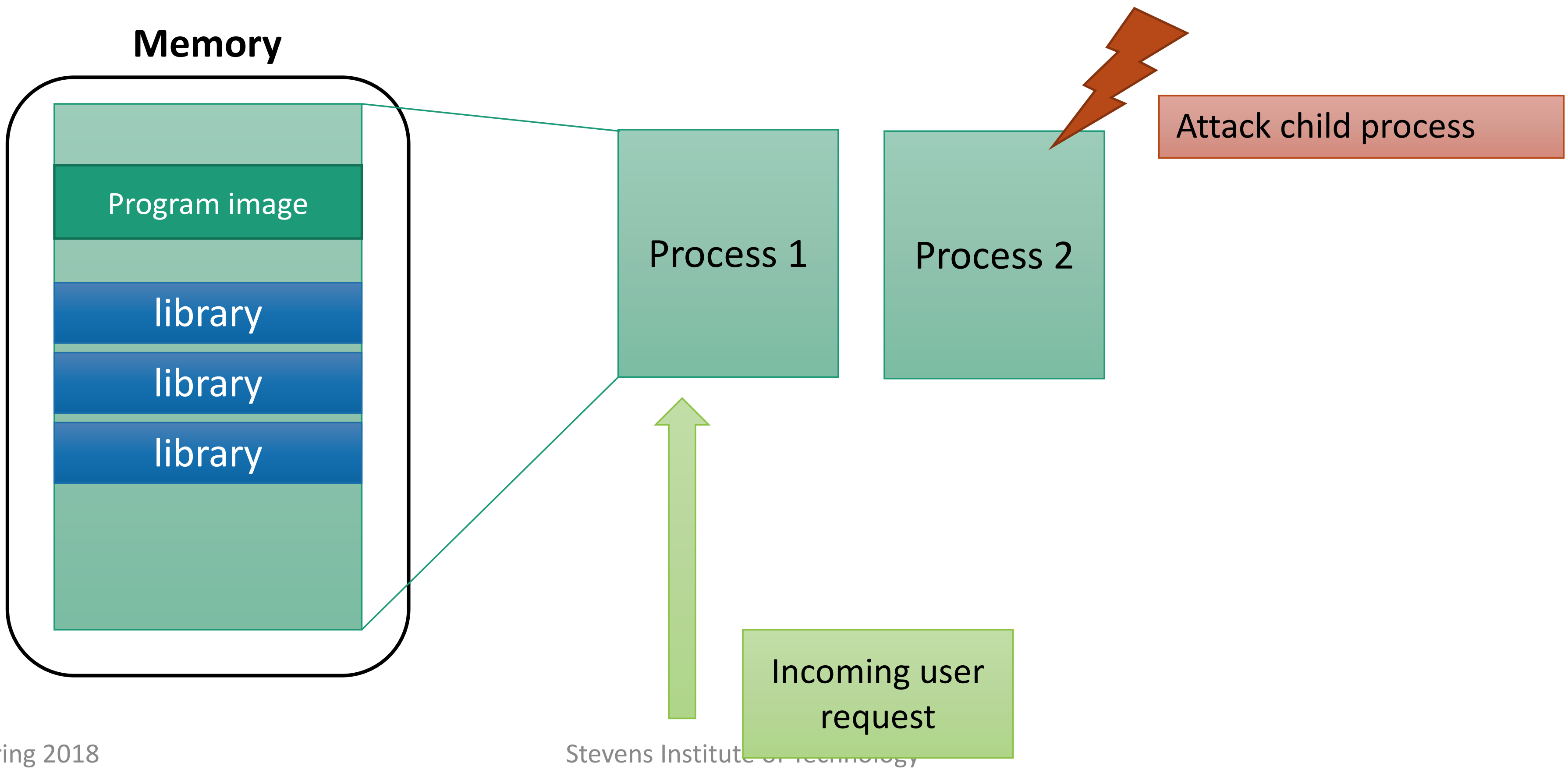
Brute-forcing ASLR

Exploiting server software using fork()



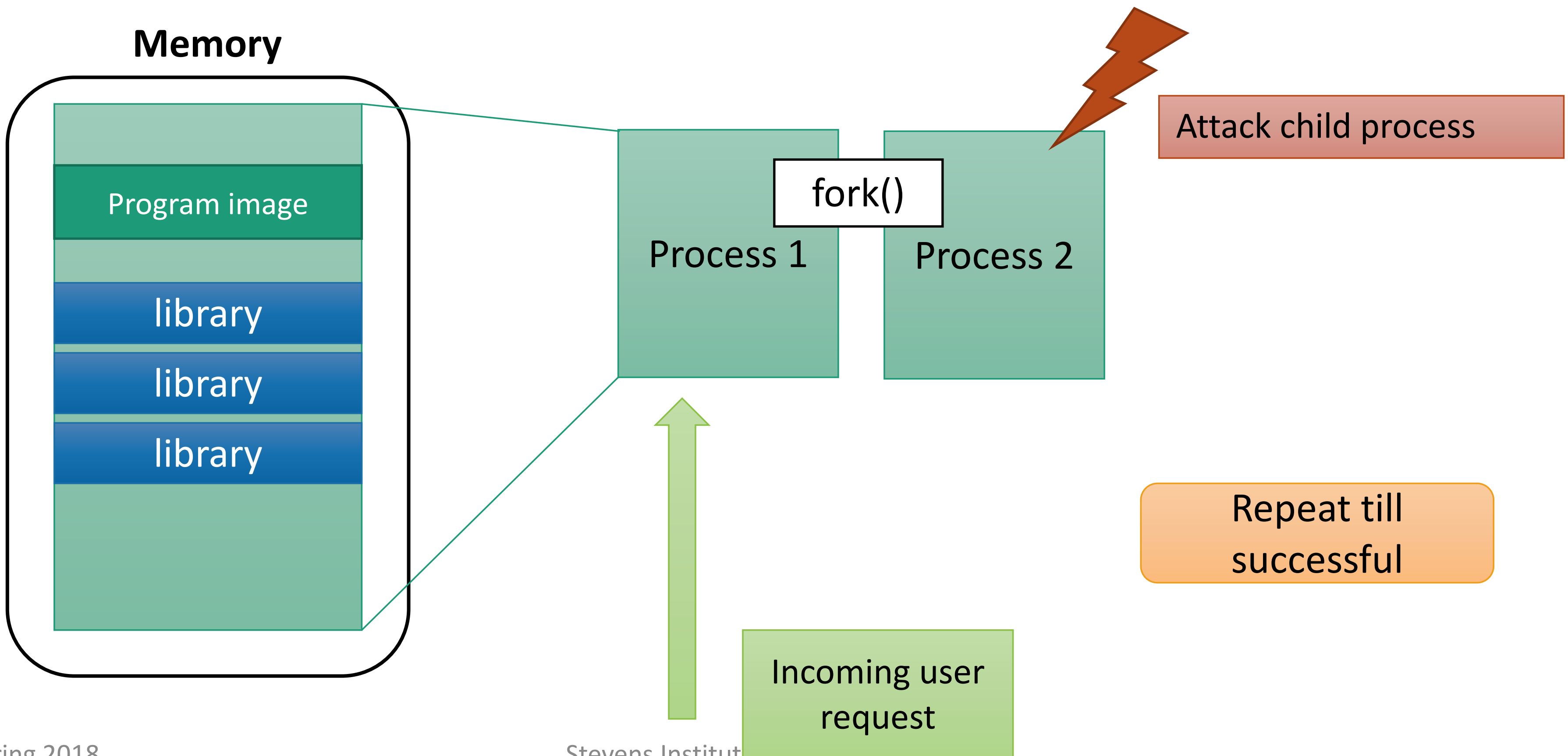
Brute-forcing ASLR

Exploiting server software using fork()



Brute-forcing ASLR

Exploiting server software using fork()



Exploit the Weakest Link

Not all program segments can be moved to a random location

ASLR-enabled programs/libraries need to be position independent (PIE)

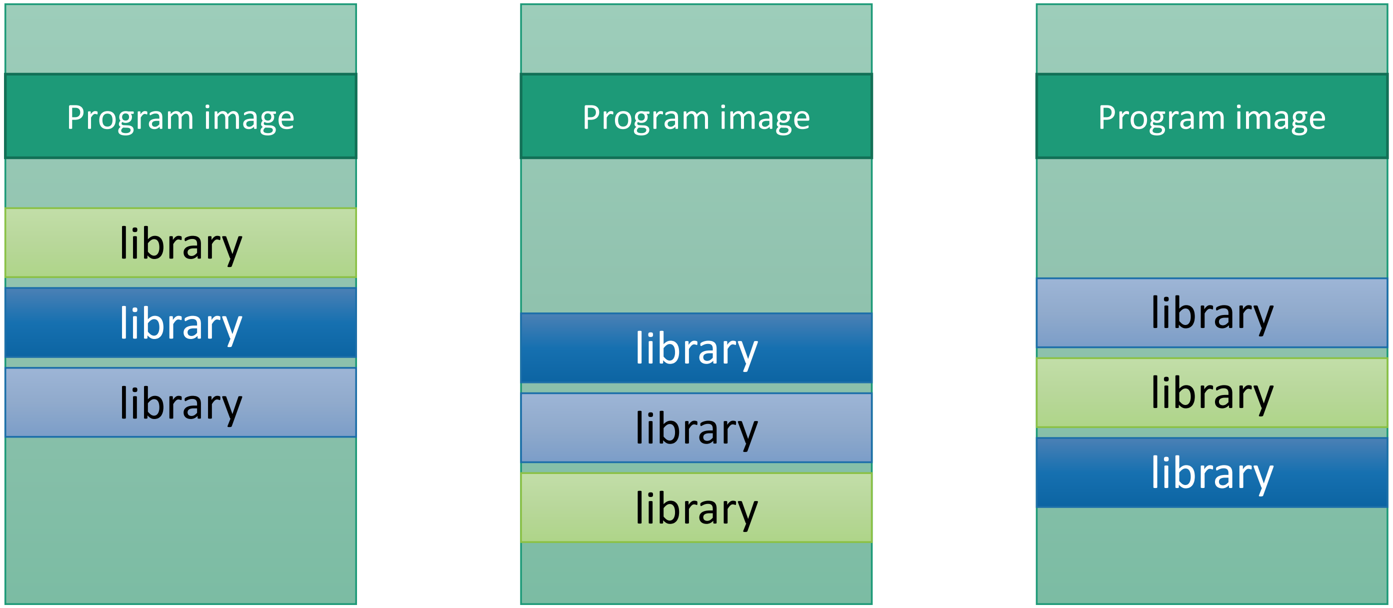
They can also opt out

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

Exploit the Weakest Link

One non-PIE may be enough



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

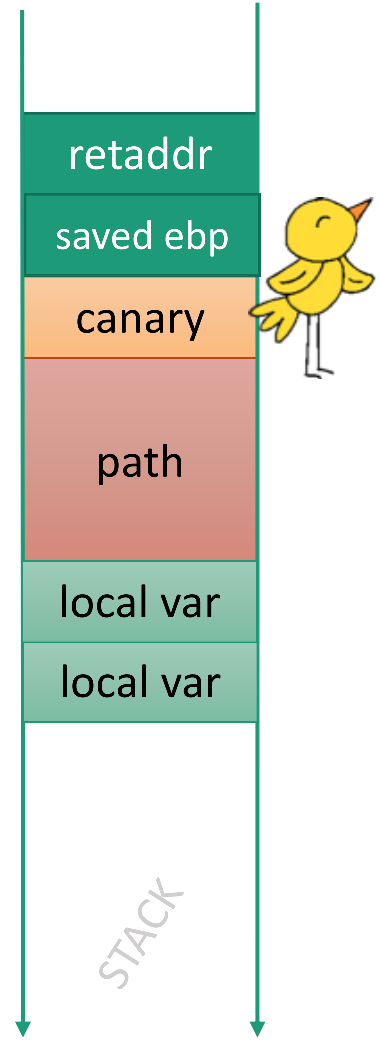
Main idea:

- 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;
...
};
```

Information Leaks Continued

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

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

MS13-037 MICROSOFT INTERNET EXPLORER DASH STYLE ARRAY INTEGER OVERFLOW

```
<html>
<head>
<script>
#{js}
</script>
<meta http-equiv="x-ua-compatible" content="IE=EmulateIE9" >
</head>
<title>
</title>
<style>v\\: * { behavior:url(#default#VML); display:inline-block }</style>
<xml:namespace ns="urn:schemas-microsoft-com:vml" prefix="v" />
<script>
#{js_trigger}
</script>
<body onload="#{create_rects_func}(); #{exploit_func}();">
<v:oval>
<v:stroke id="vml1"/>
</v:oval>
</body>
</html>
```

Summary of ASLR Weaknesses

Memory leaks

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

Insufficient entropy

Incompatible binaries

Memory spraying

- Make many copies of the attack payload
- Increase the chances of the payload being at a particular address
- Probabilistic attack

Side channels

- Infer layout based on leaks from side channels

Reading

Stackguard
ftp://gcc.gnu.org/pub/gcc/summit/2003/Stackguard.pdf

Bypassing StackGuard and StackShield
http://phrack.org/issues/56/5.html

Bypassing PaX ASLR protection
http://phrack.org/issues/59/9.html

On the Effectiveness of Address-Space Randomization

https://benpfaff.org/papers/asrandom.pdf

Low-level Software Security:Attacks and Defenses

https://trailofbits.github.io/ctf/exploits/references/tr-2007-153.pdf