Early Defenses and More Attacks

CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 2018

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

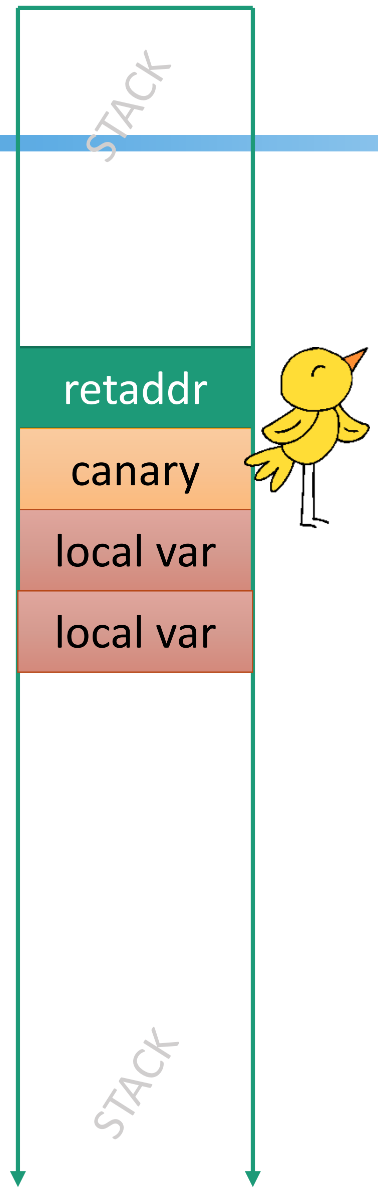
StackGuard

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

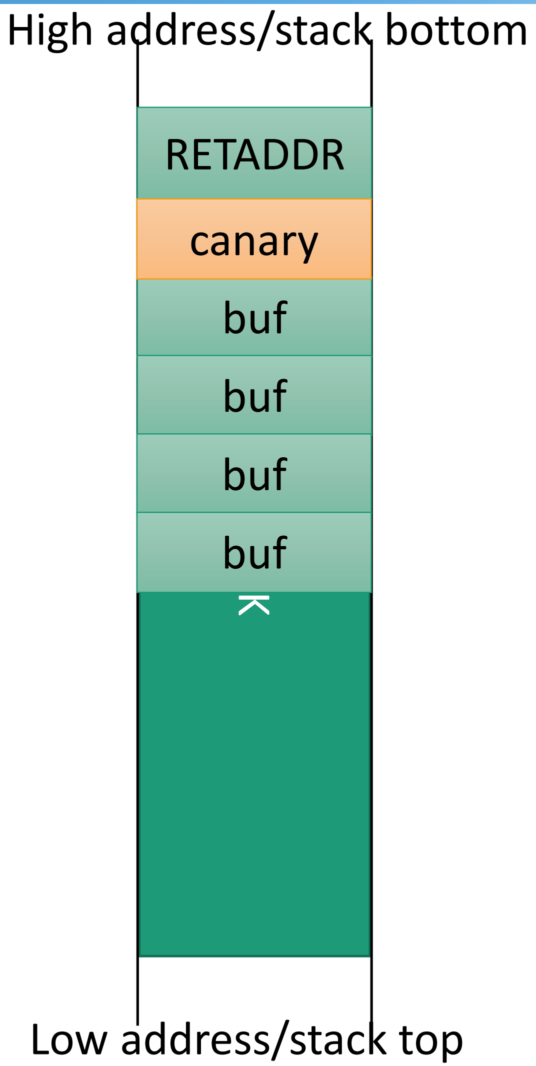
- Program stops if the canary has changed

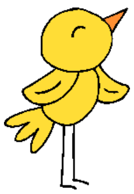


Stack Overflow With Canary

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

        strcpy(buf, str);

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

        return strlen(buf);
}
```

```
./mytest AAAAA
```

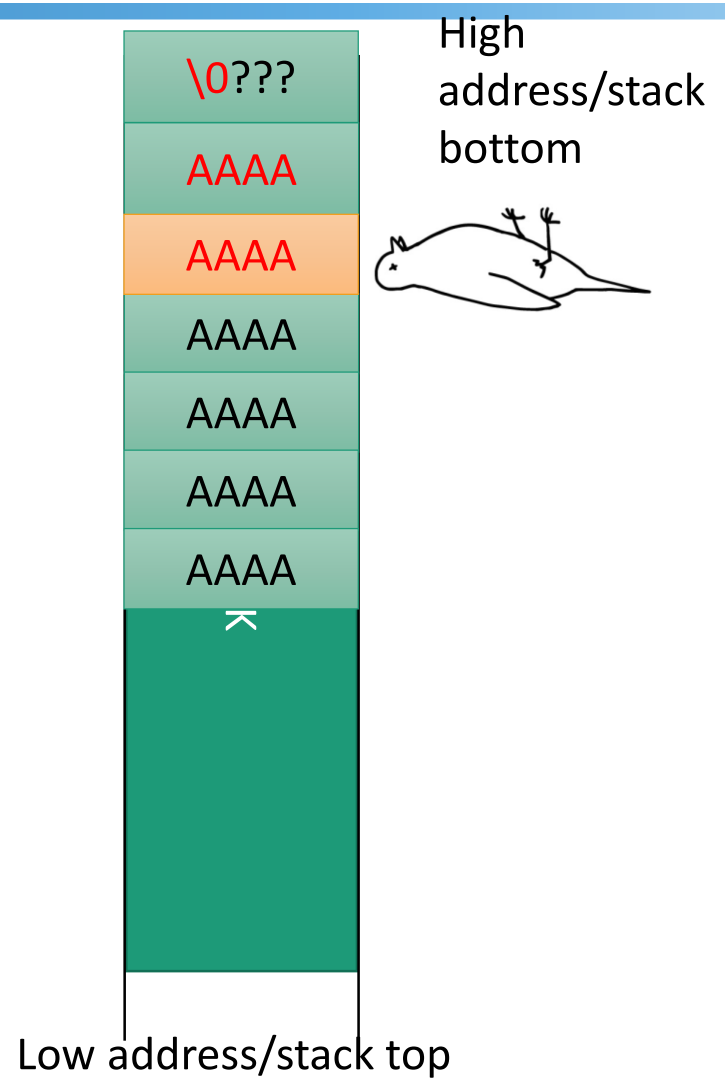




Stack Overflow with Canary

```
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.

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

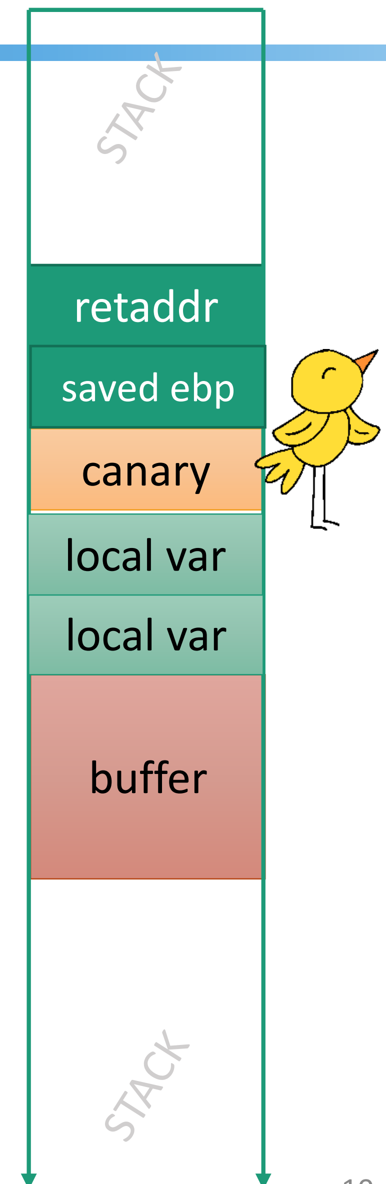
```
0000000000400606 <mytest>:
  400606:       55                      push   %rbp
  400607:       48 89 e5                mov    %rsp,%rbp
  40060a:       48 83 ec 30             sub    $0x30,%rsp
  40060e:       48 89 7d d8             mov    %rdi,-0x28(%rbp)
  400612:       64 48 8b 04 25 28 00    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

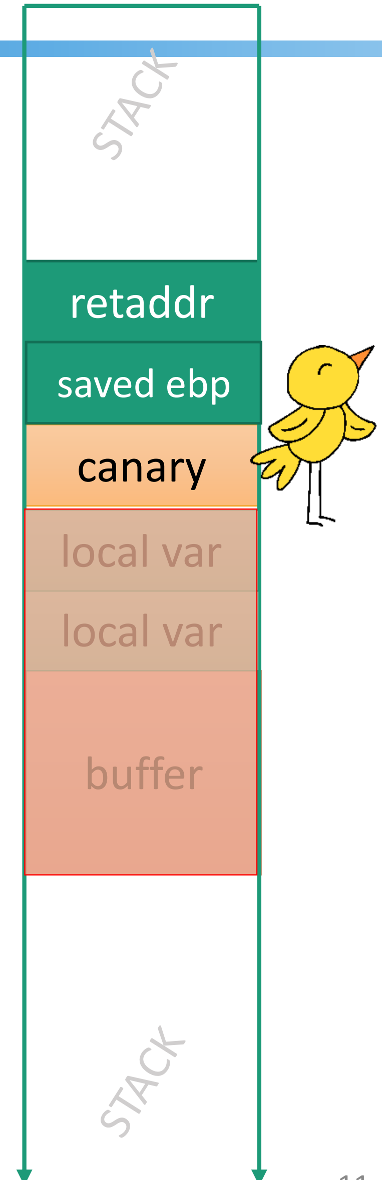
The order of local variables may be important



Alignment of Stack Buffers and Canaries

The order of local variables may be important

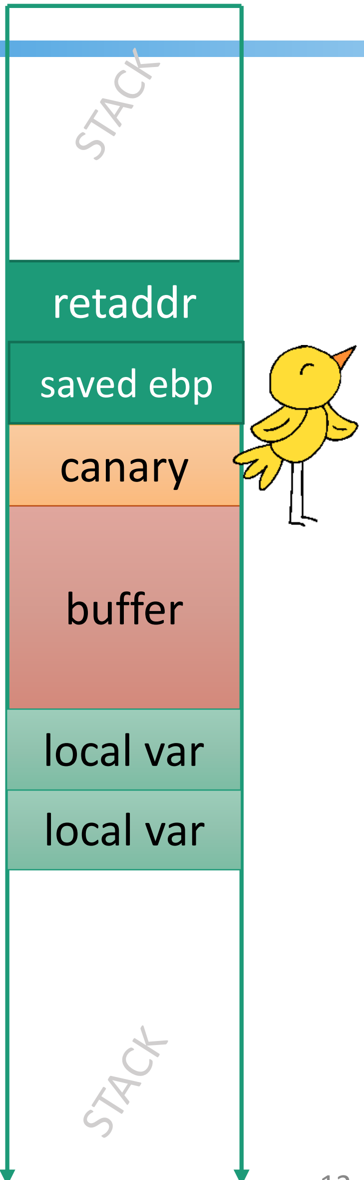
Buffer overflows could allow important local variables to be controlled



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”



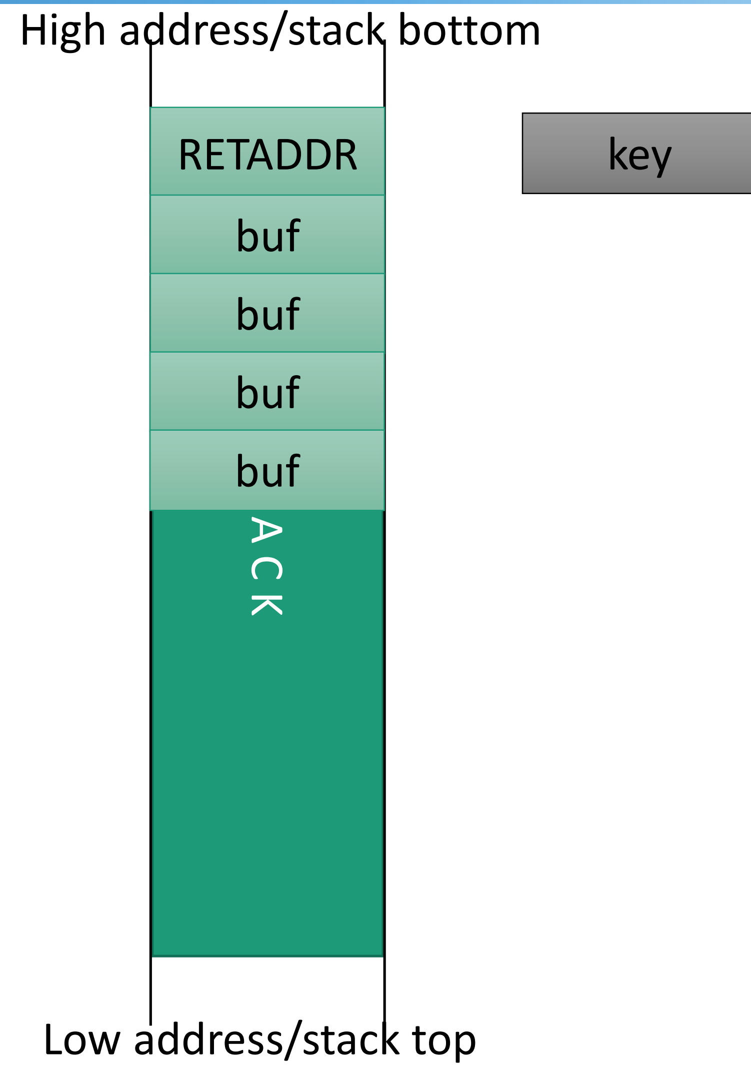
StackShield

Address obfuscation instead of canary

Encrypt return address on stack by XORing with random string

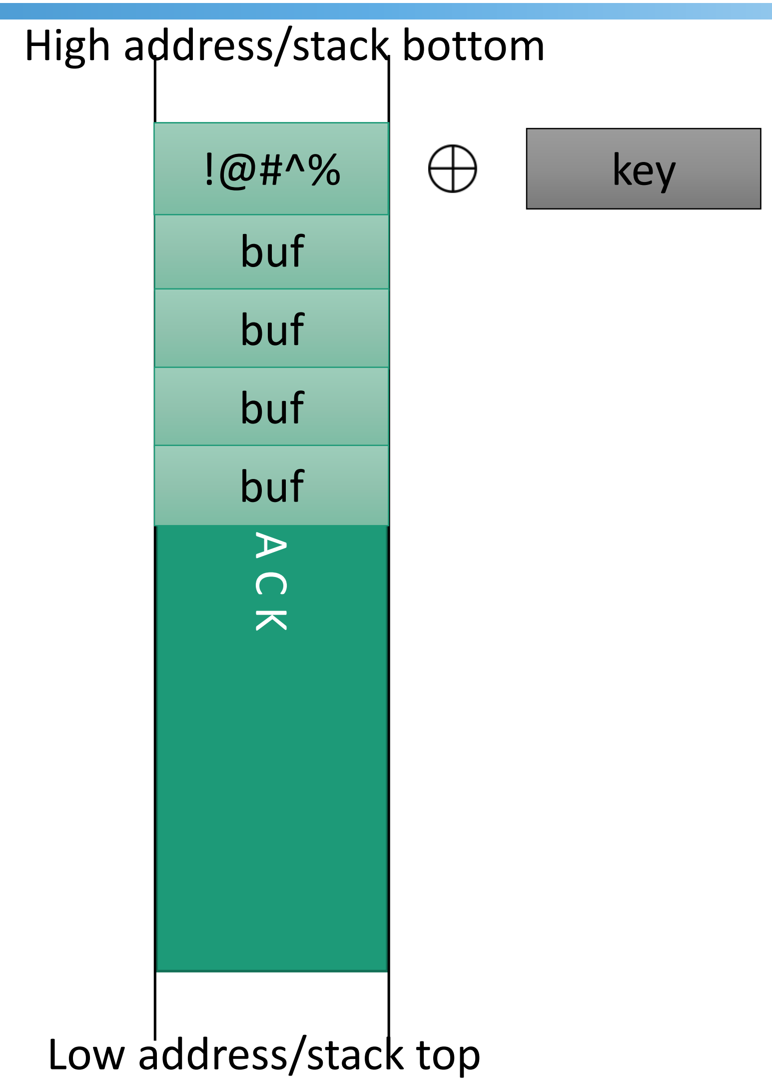
Decrypt just before returning from function

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



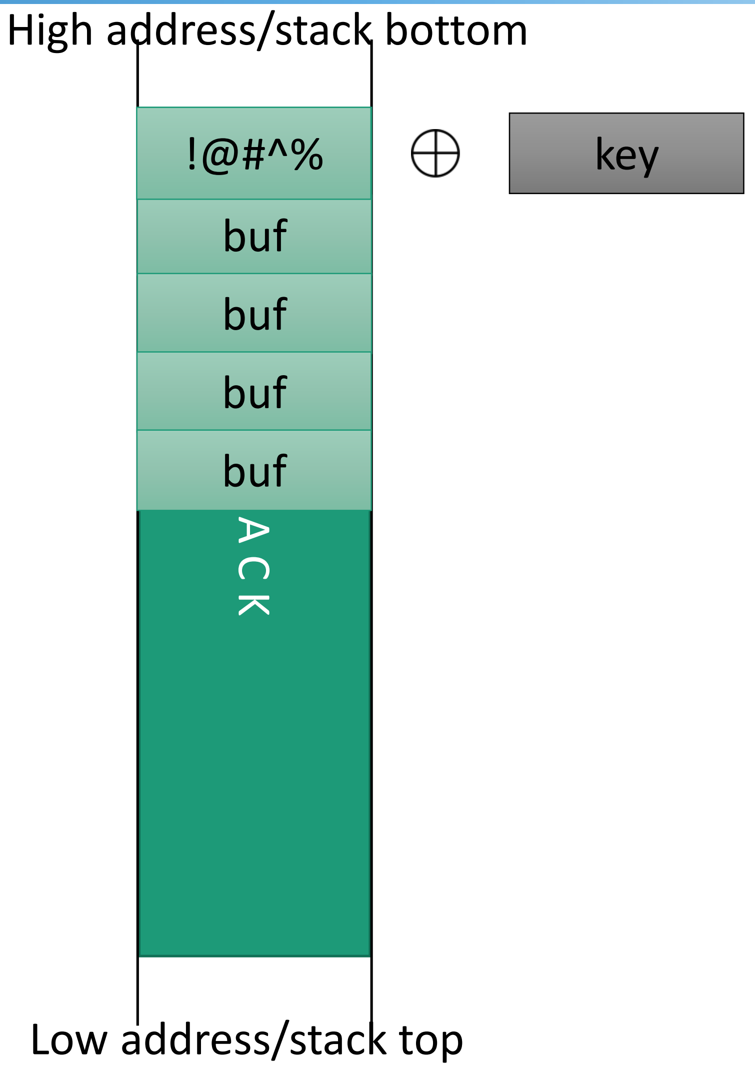
Example: StackShield

```
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

Problems

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

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

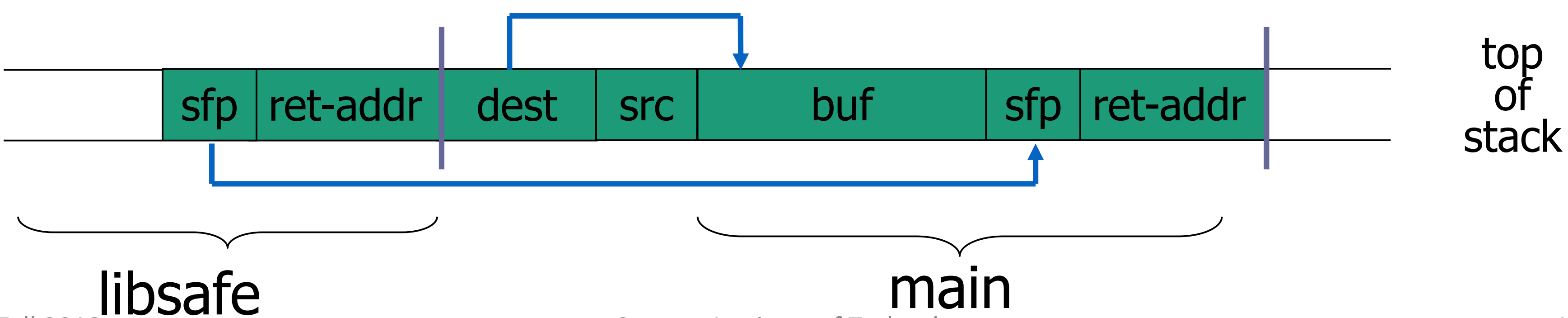
ASLR and bypasses

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



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

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!

* **glibc detected** * ./load: double free or corruption (!prev): 0x0000000000c6ed50 ***

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!


```
* glibc detected * ./load: double free or corruption (!prev): 0x0000000000c6ed50 ***
```

Other Protections

Separating metadata from chunks

Adding canary type values

Topics

Stack overflow defenses

- Stackguard & Stackshield
- Boundary checking

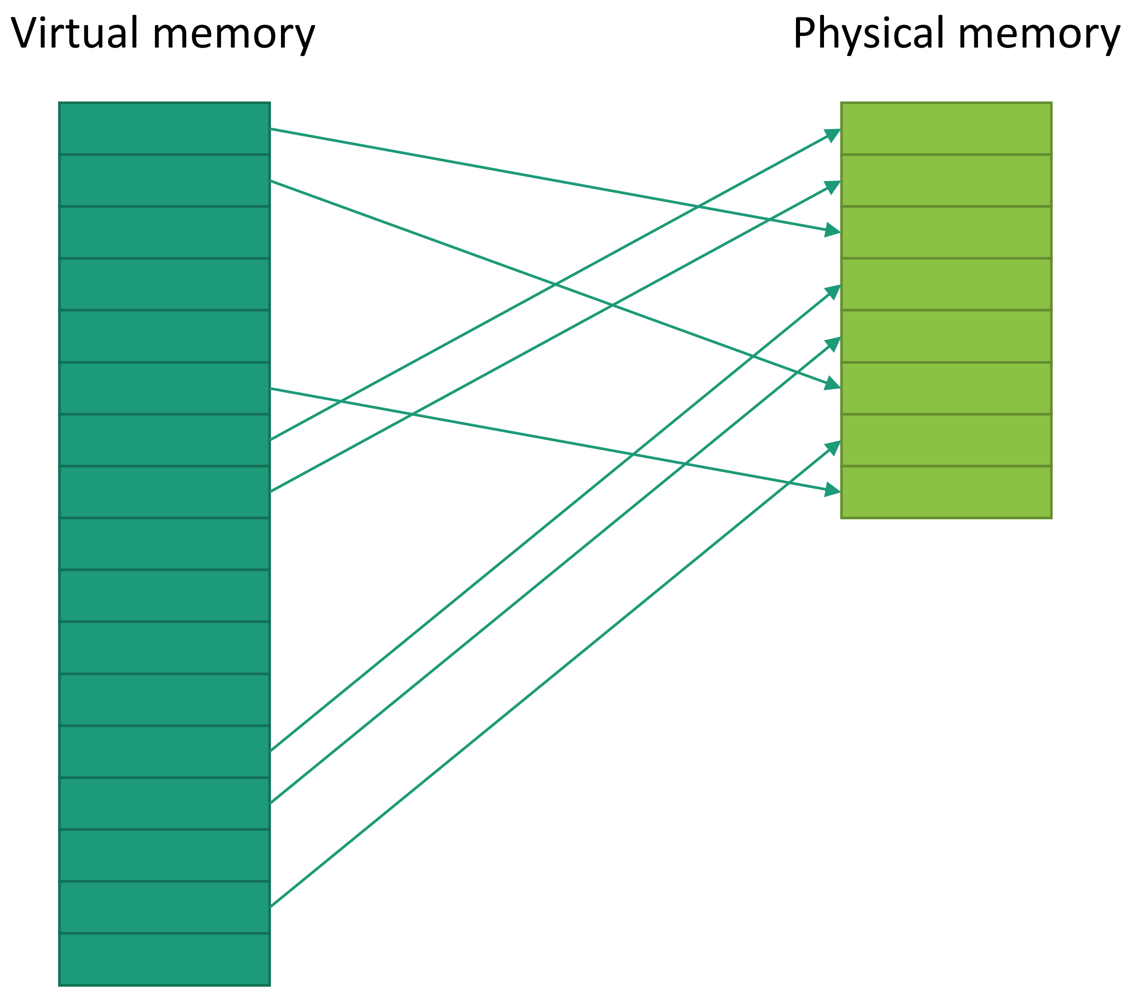
Heap corruption defenses

Code-injection defenses and bypasses

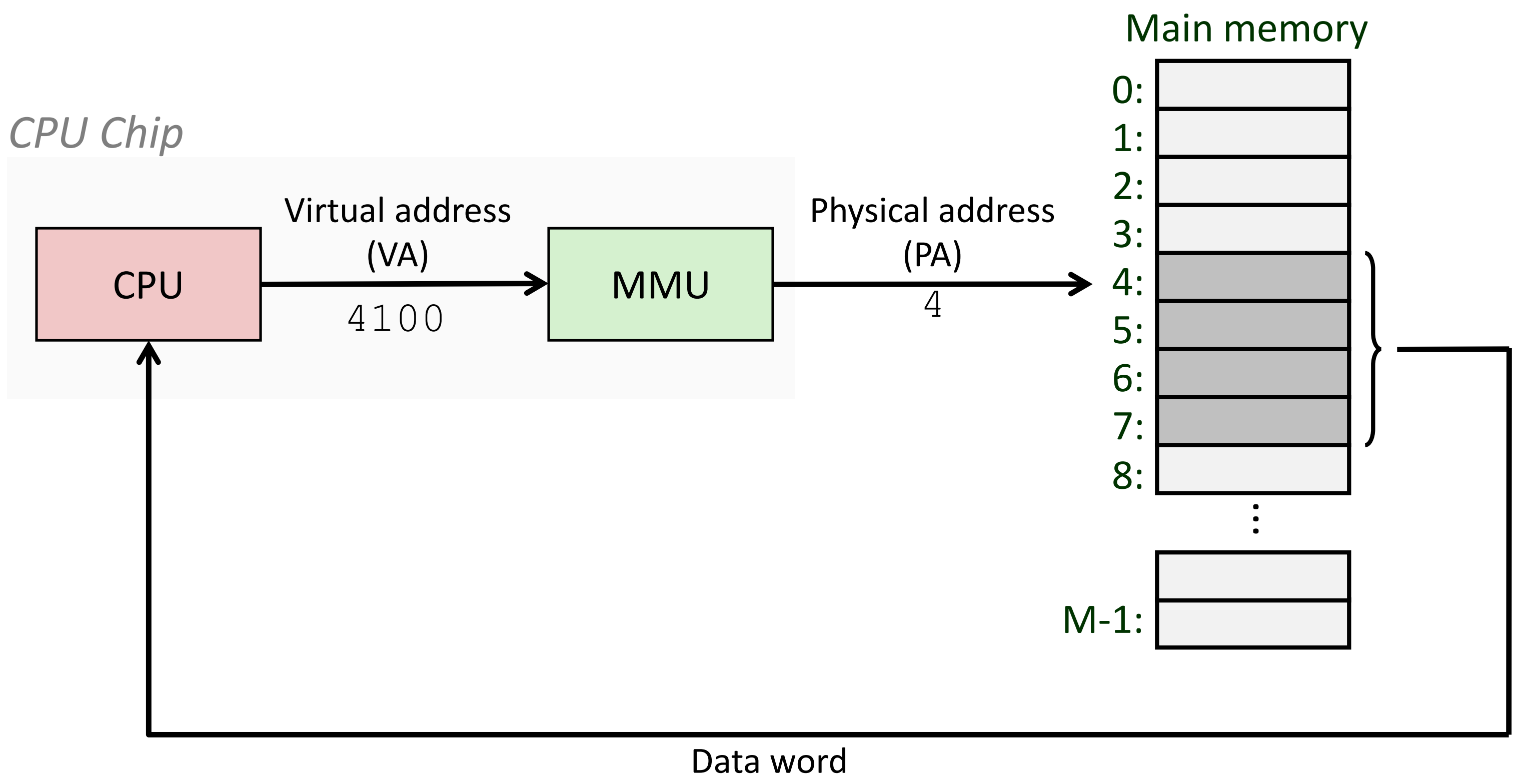
- **Non executable stack (and heap)**
- Early code-reuse attacks/return-to-libc
- ASCII armored space

ASLR and bypasses

Virtual Memory



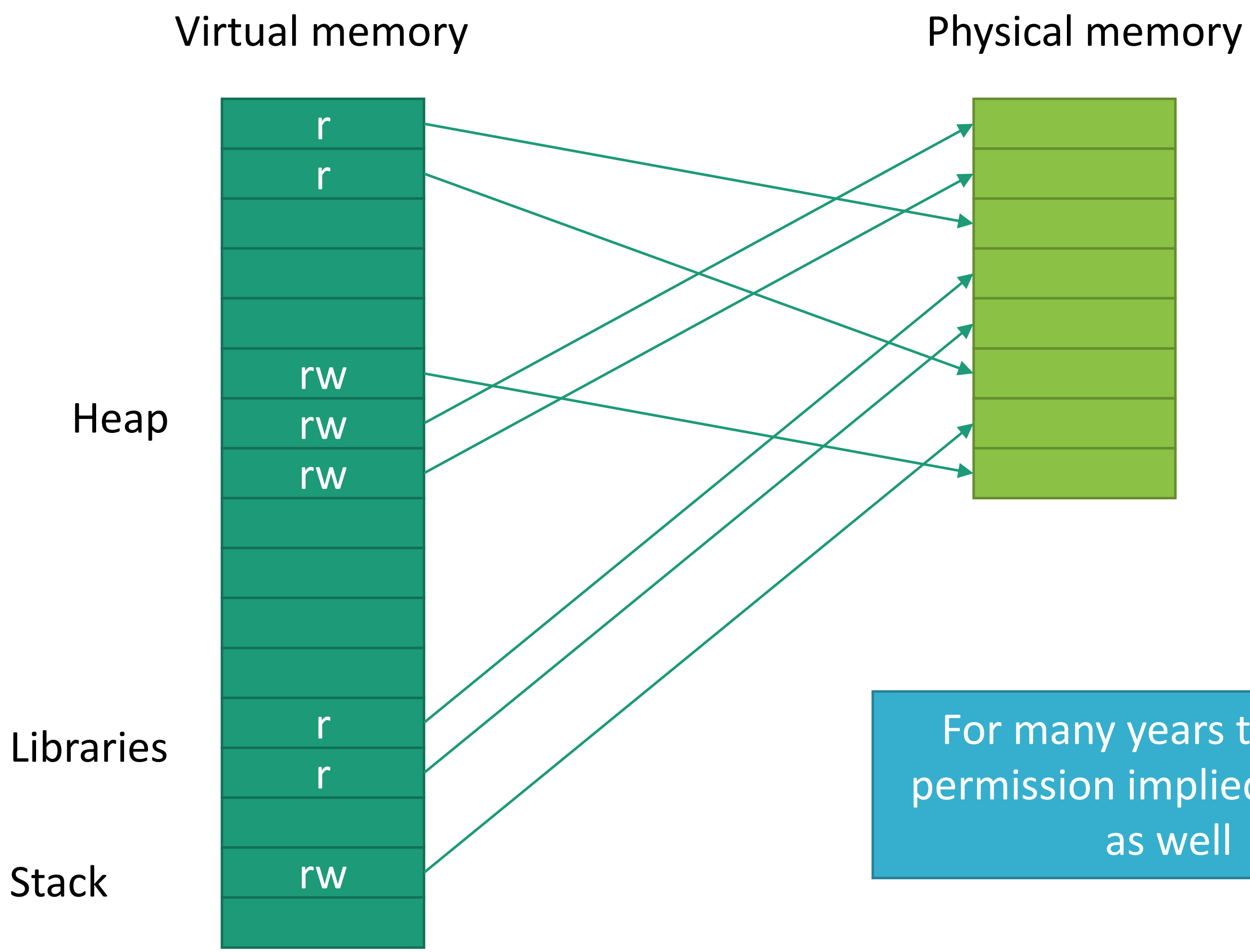
The Memory Management Unit



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)

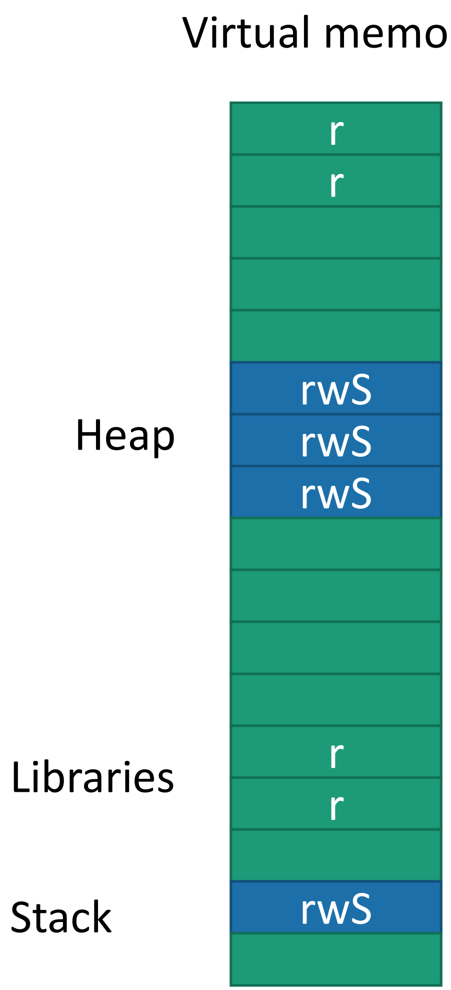
PaX stands for PageEXec

Introduced in 2000

A Linux kernel patch protection emulating non-executable memory

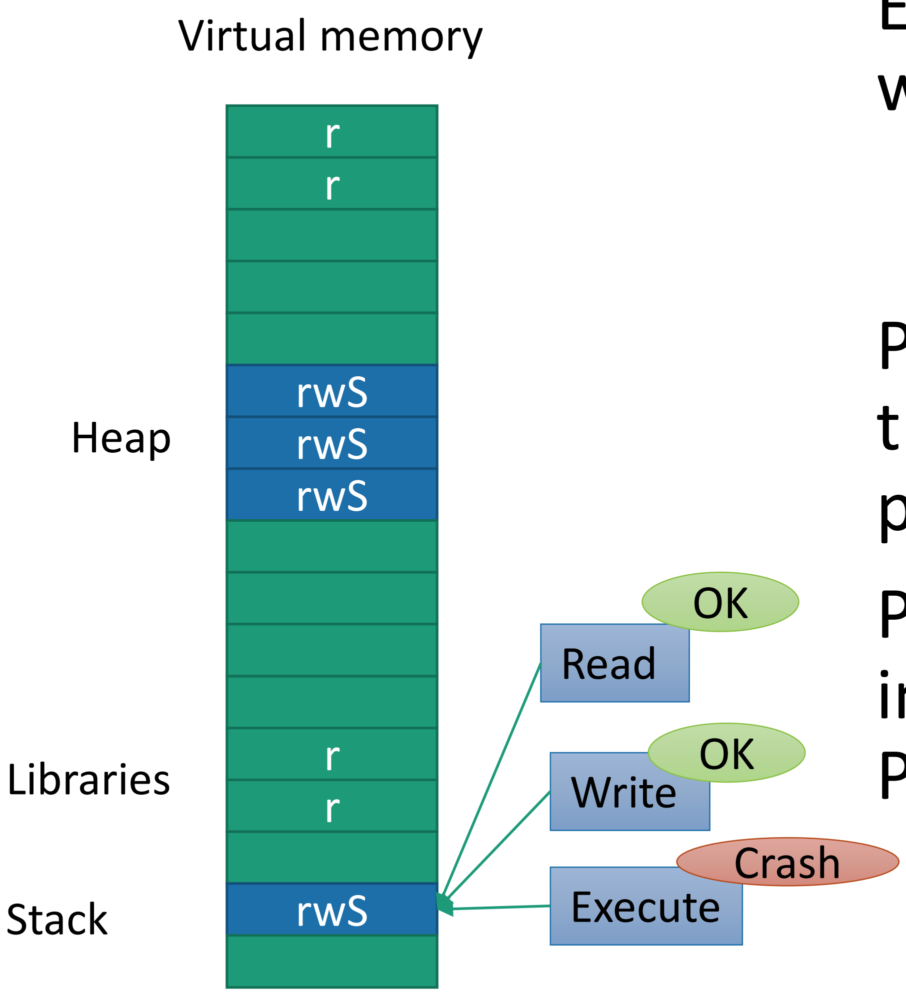
PaX refused code execution on writable pages

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

Virus Bulletin :: Enhance

virusbulletin.com/conference/vb2005/abstracts/enhanced-virus-protection

Covering the global threat landscape

Blog Bulletin VB

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

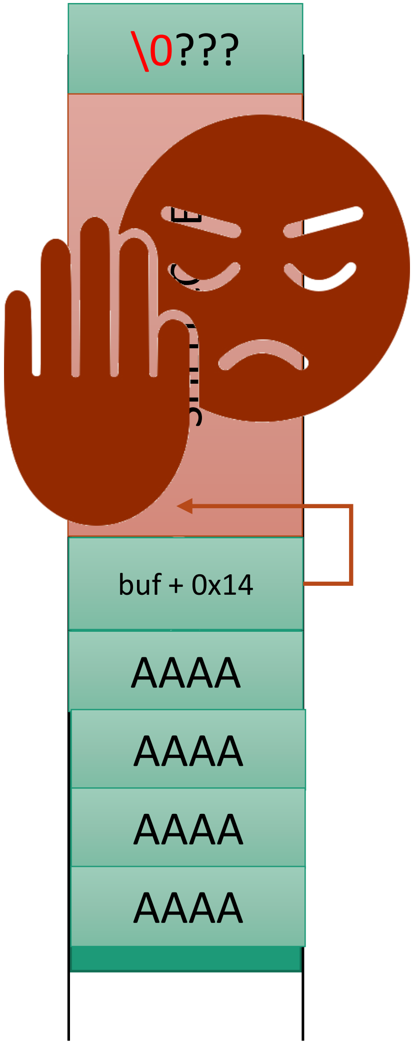
Data-execution prevention lead to a more generic security policy

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…

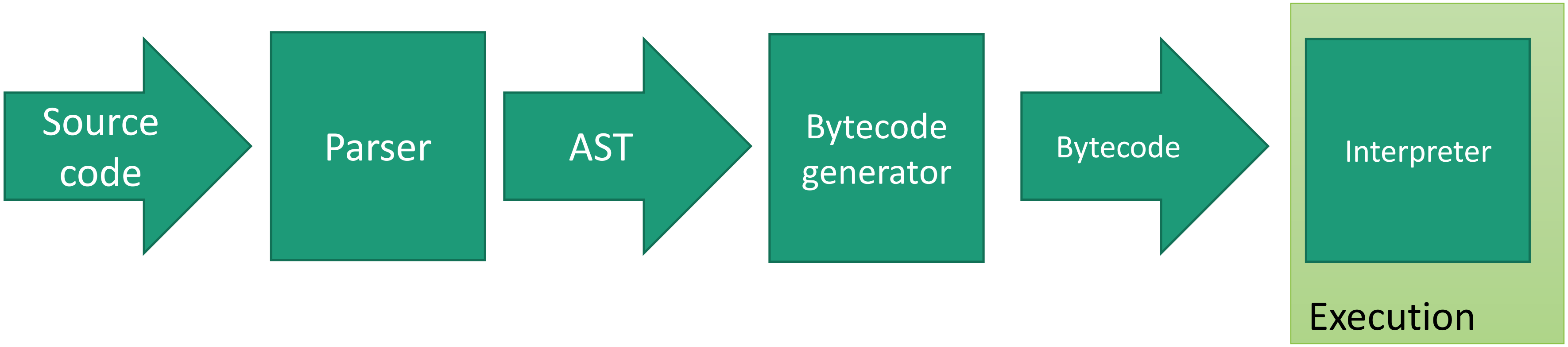
Very popular software

- Probably installed on every client device

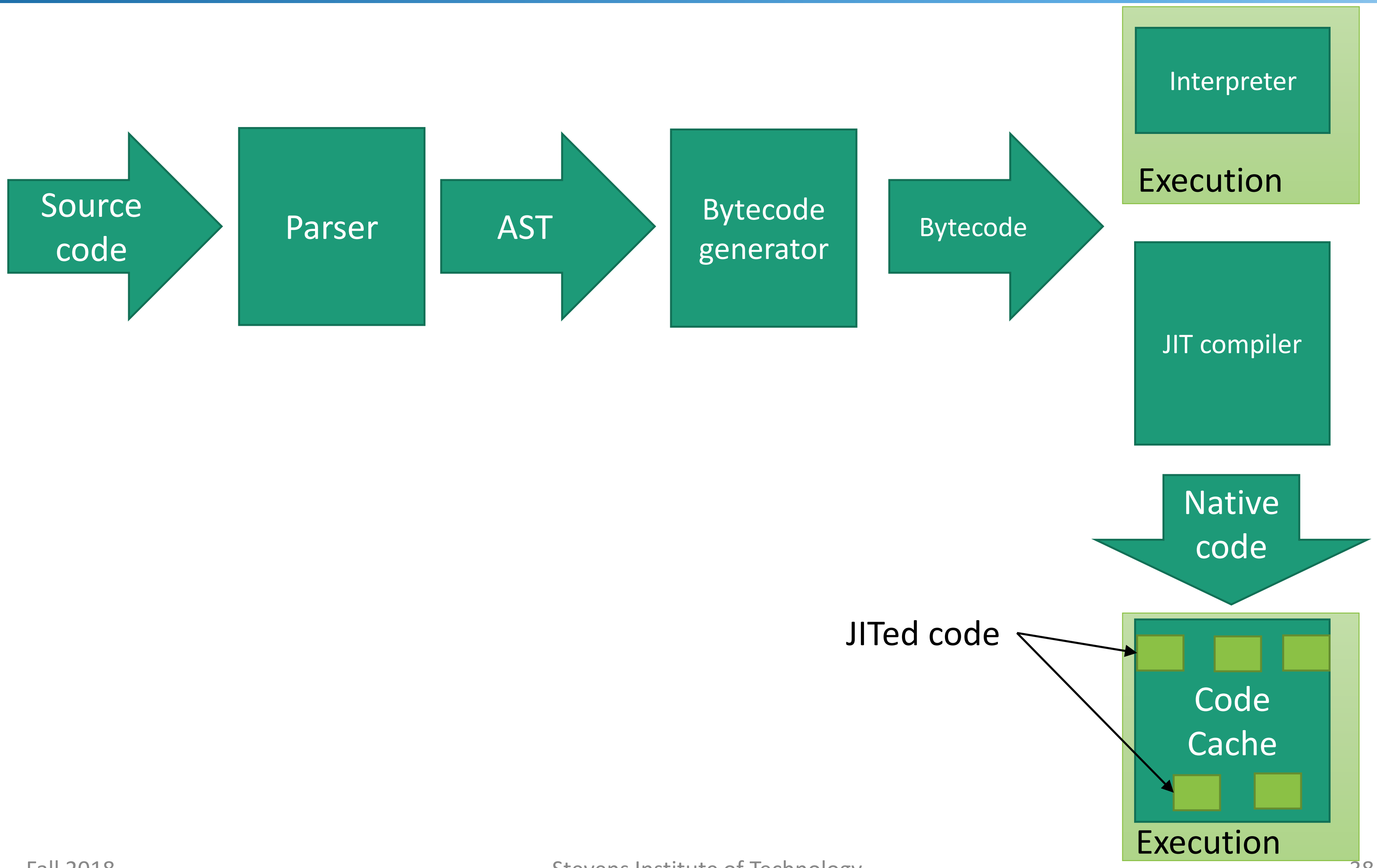
Large and complex software

Execute JavaScript

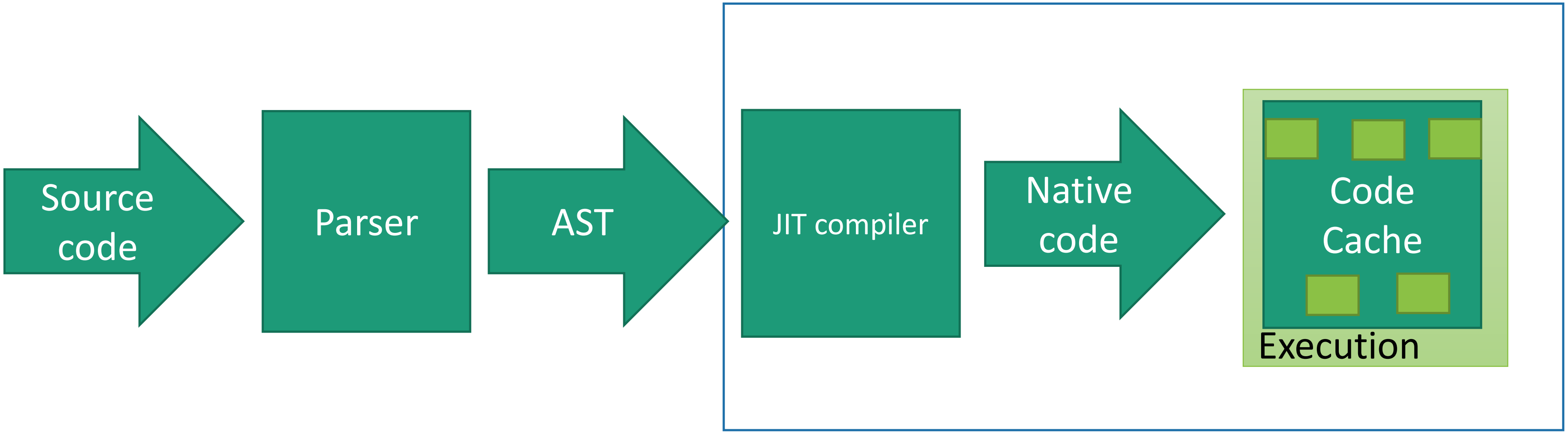
How Does JavaScript Run



How Does JavaScript Run



How Does JavaScript Run

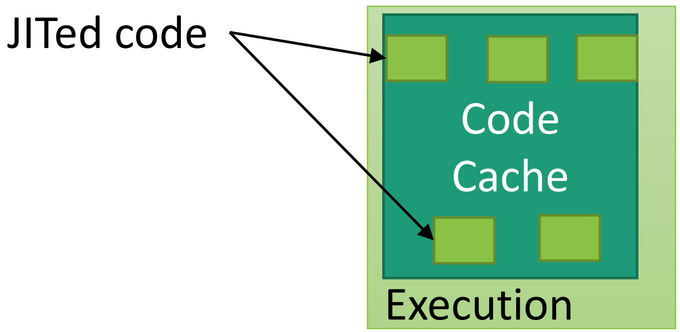


- Google V8 designed specifically to execute at speed.
- Bytecode generation skipped
- Directly emit native code
- Overall JavaScript execution improved by 150%

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

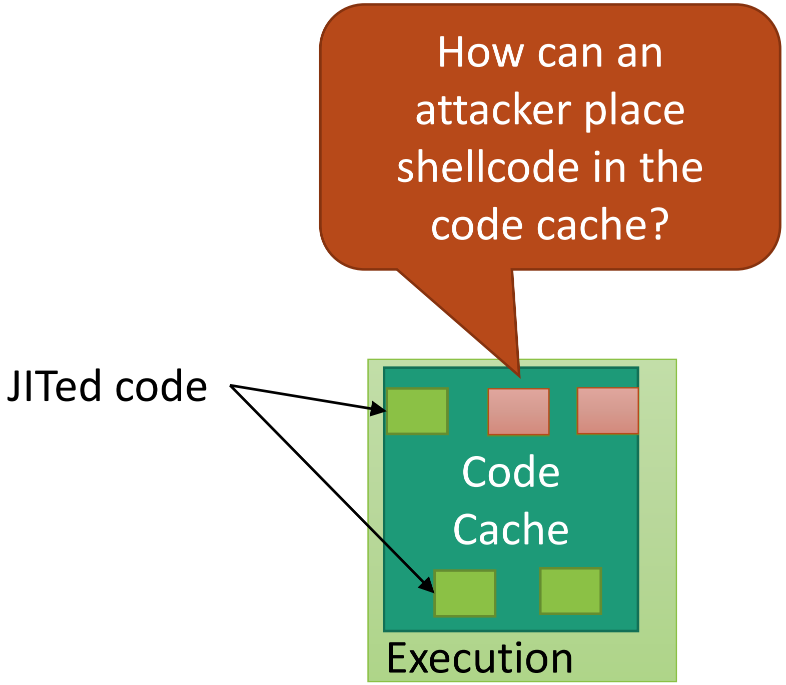


Violates the W^X policy

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>
```

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

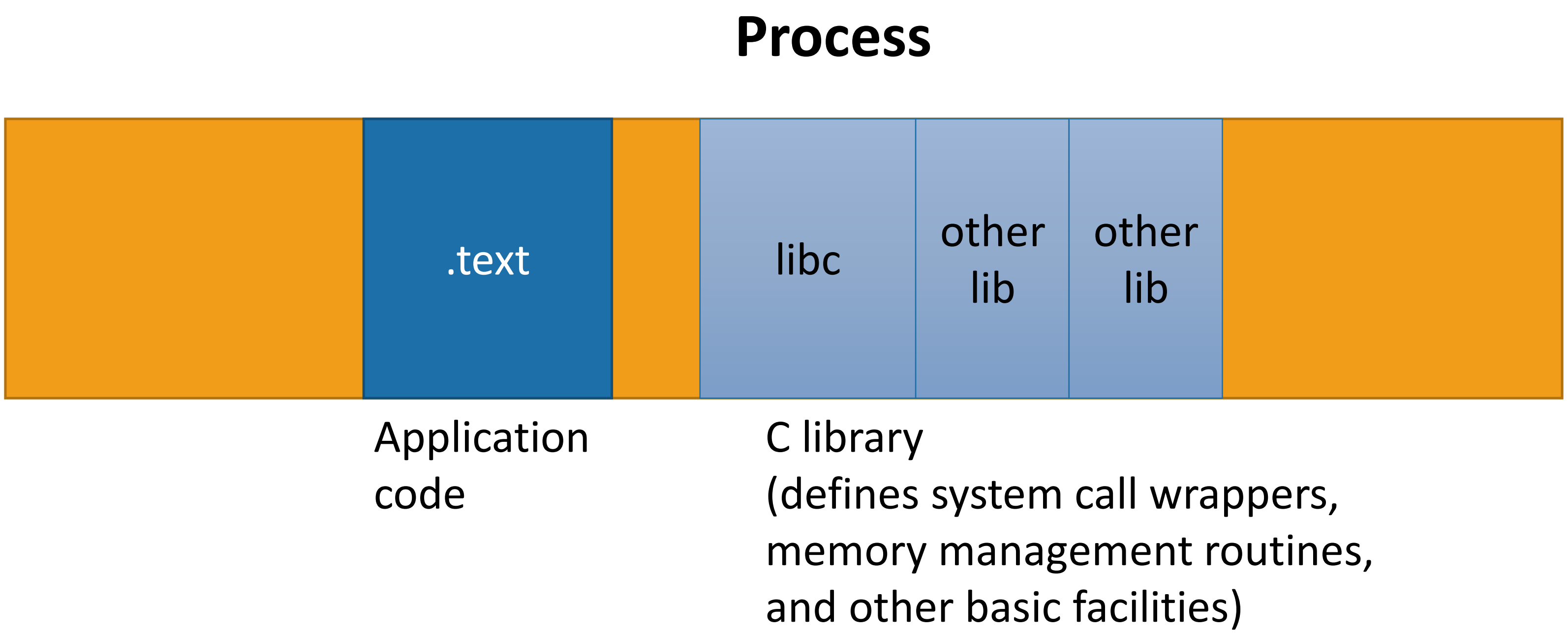
Return-to Attacks

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)

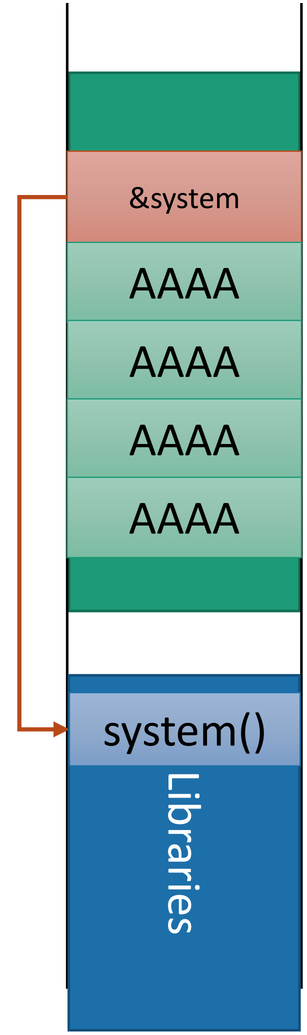


```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00007ffc83b62000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f9edfdf1000)
        libacl.so.1 => /lib/x86_64-linux-gnu/libacl.so.1 (0x00007f9edfbe8000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9edf83d000)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

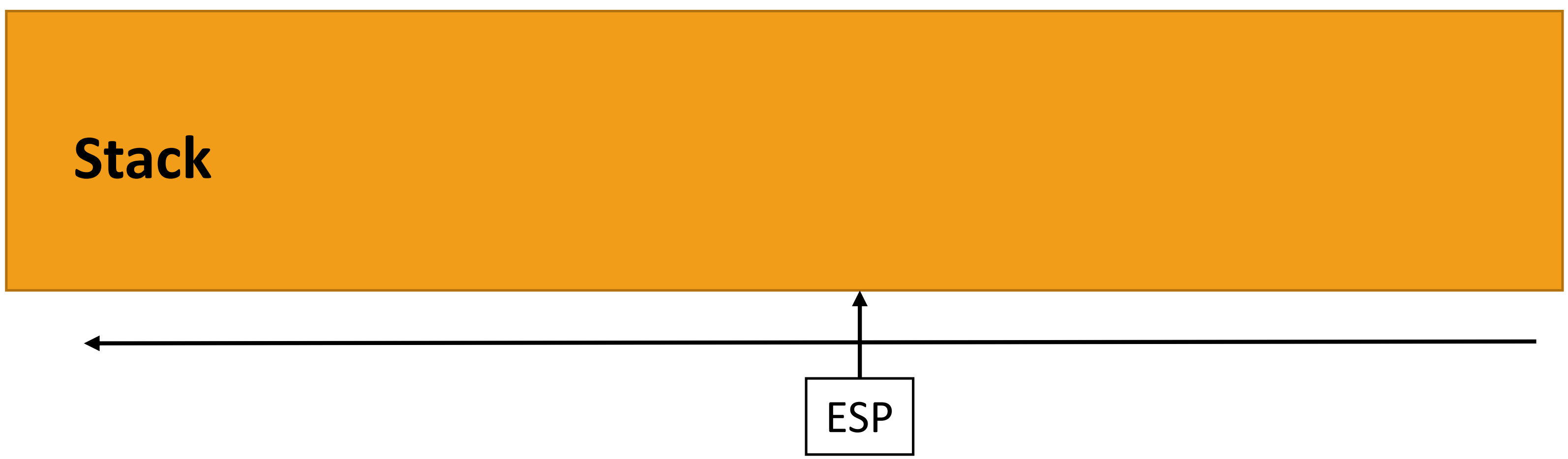
```
int main(void)
{
        system("/bin/shell");
        return 0;
}
```

```
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

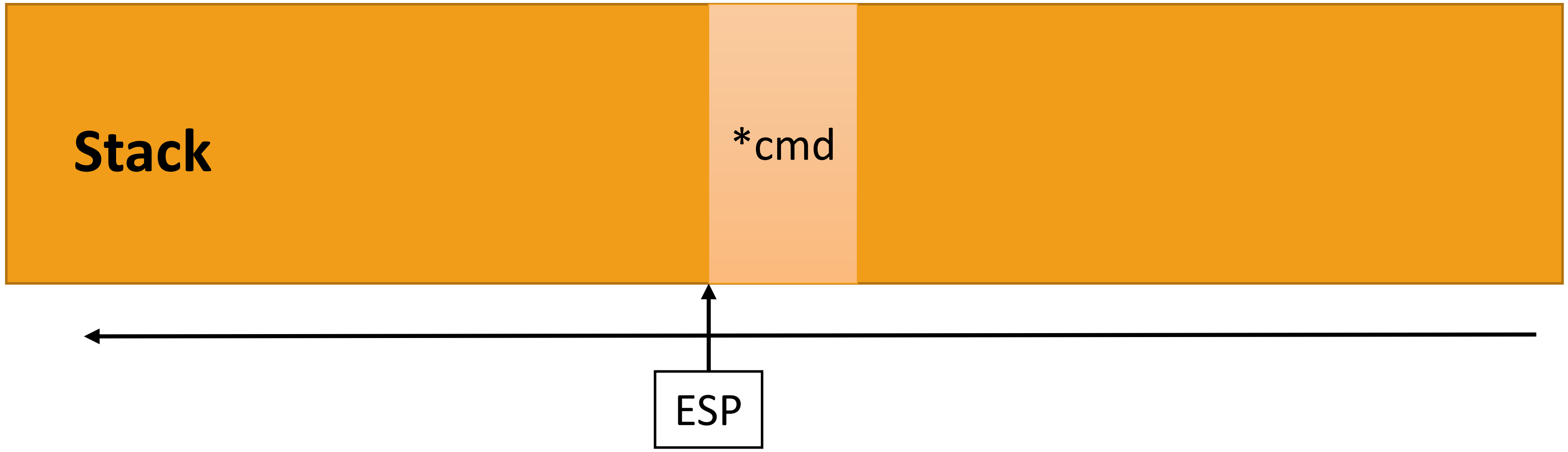
```
804840f:    68 c0 84 04 08    push   $0x80484c0
8048414:    e8 b7 fe ff ff    call   80482d0 <system@plt>
```



Preparing the Stack

EIP →

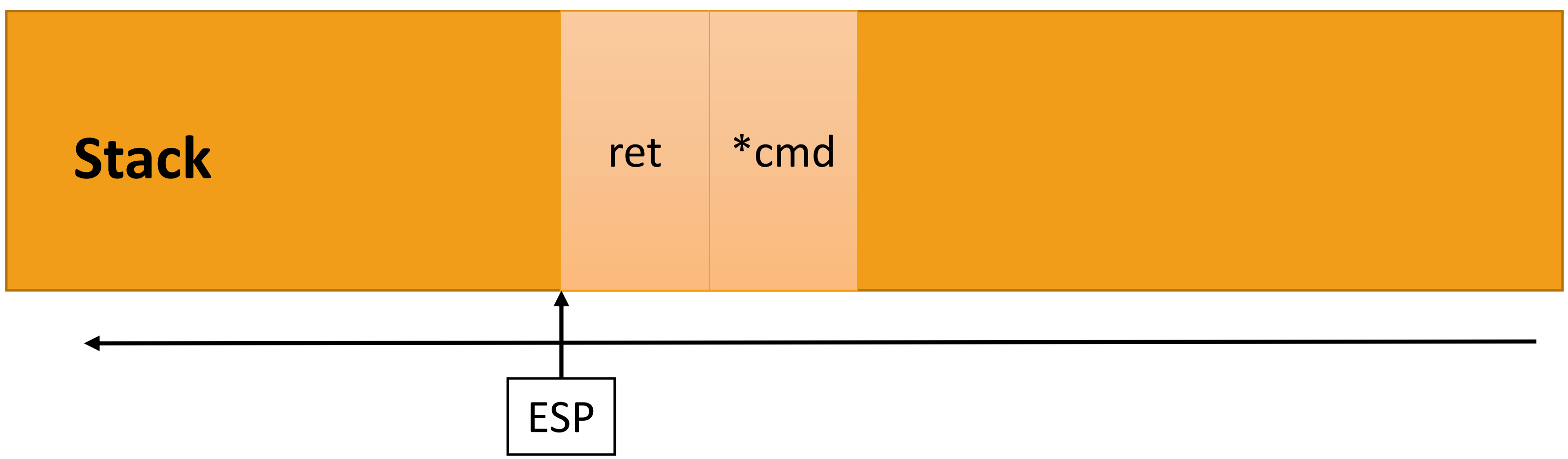
```
804840f:    68 c0 84 04 08    push   $0x80484c0
8048414:    e8 b7 fe ff ff    call   80482d0 <system@plt>
```



Preparing the Stack

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

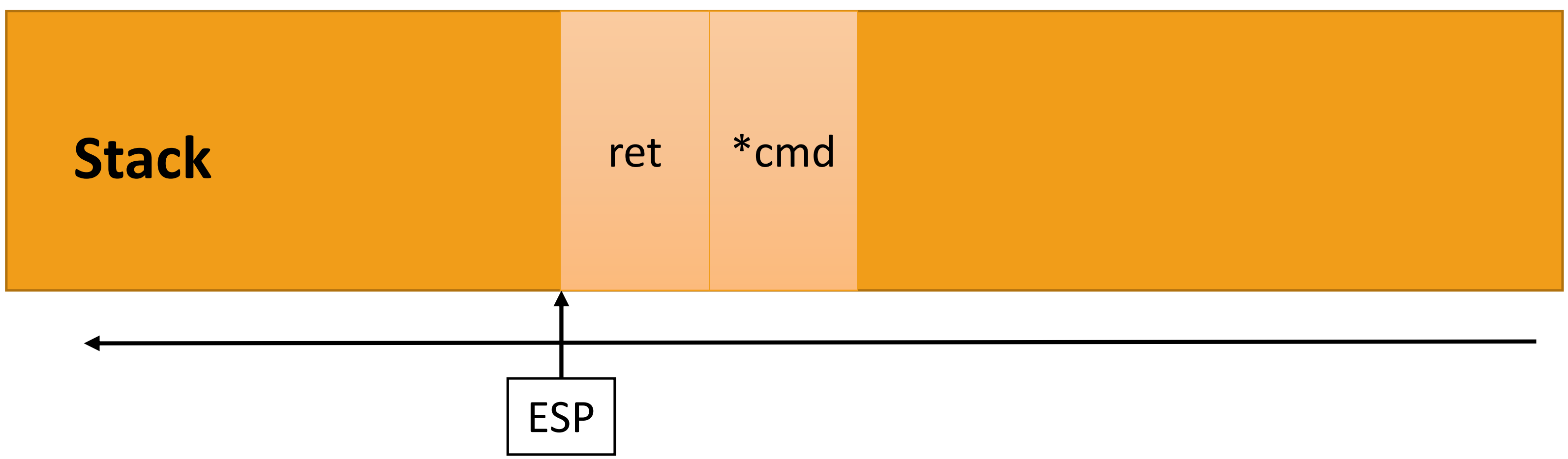
EIP →
```
0003de80 <__libc_system>:
   3de80:      53              push   %ebx
```



Preparing the Stack

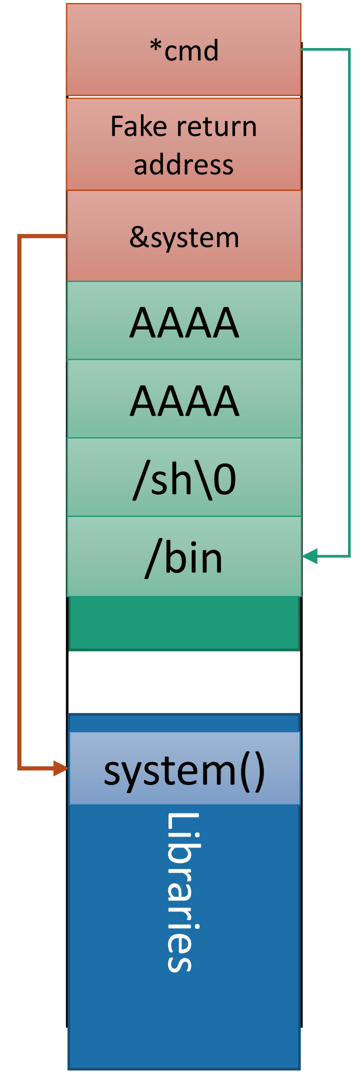
```
80484(                                              0
80484:                                              system@plt>

EIP → 0003de
         3de
```

The stack needs to look like this when *system()* is entered



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;
}
```

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

```
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>
...
```

Code-reuse Attacks

Any code that already exists in the process can be executed

For example, the following sequence

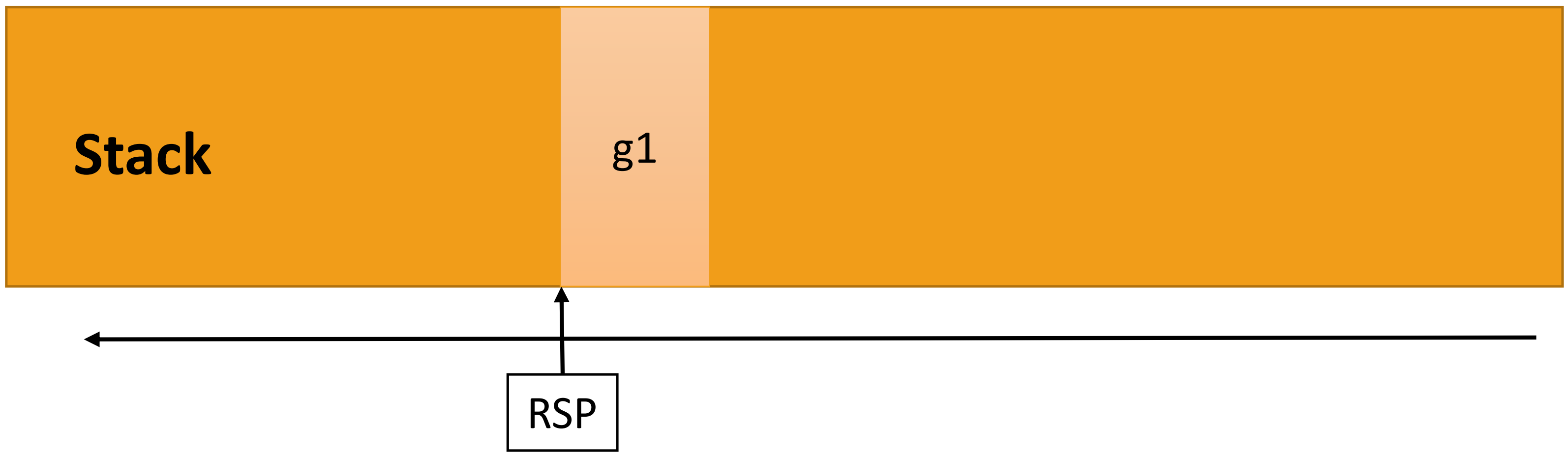
```
0x0000000000405255 : pop rdi ; ret
```

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

Return-to-libc on 64-bit

Redirect control to gadget

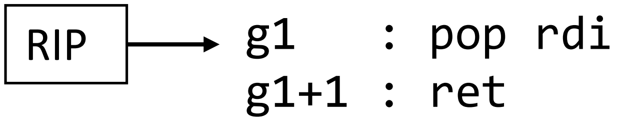
```
g1   : pop rdi
g1+1 : ret
```

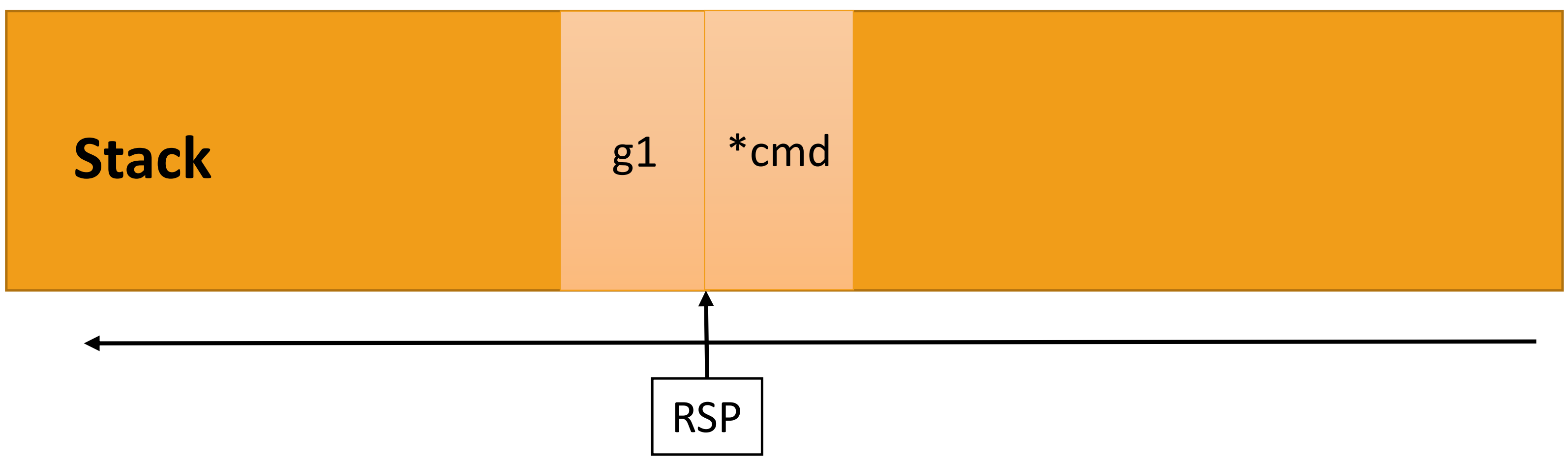


Return-to-libc on 64-bit

Redirect control to gadget

Load argument on register





Return-to-libc on 64-bit

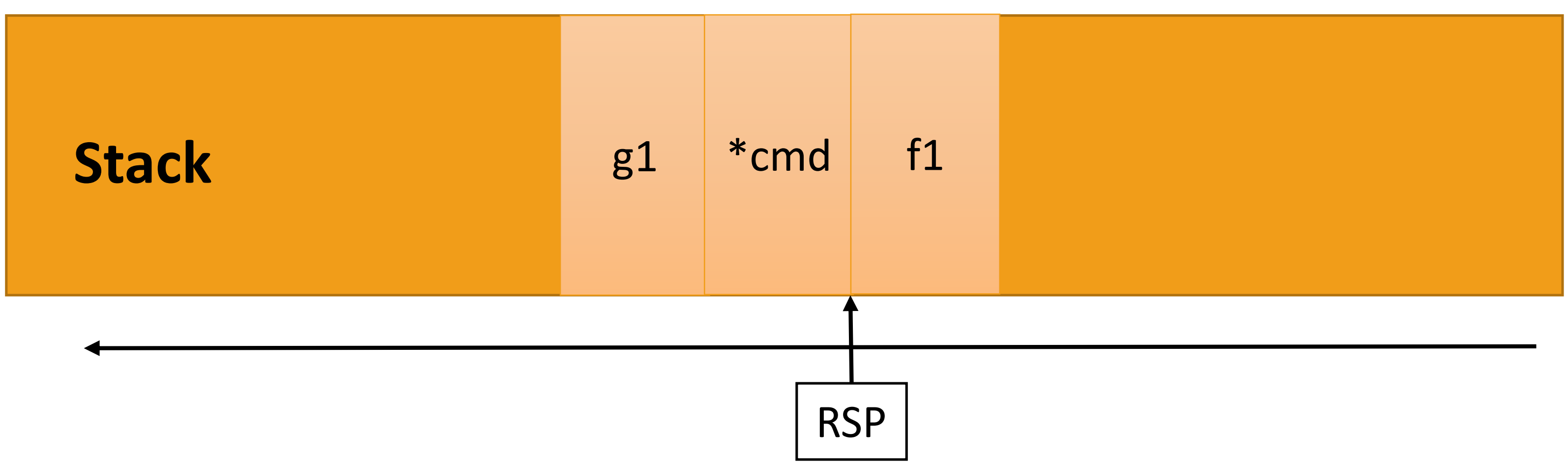
Redirect control to gadget

Load argument on register

Redirect control to libc function

RIP

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

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



Return-to-libc on 64-bit

Redirect control to gadget

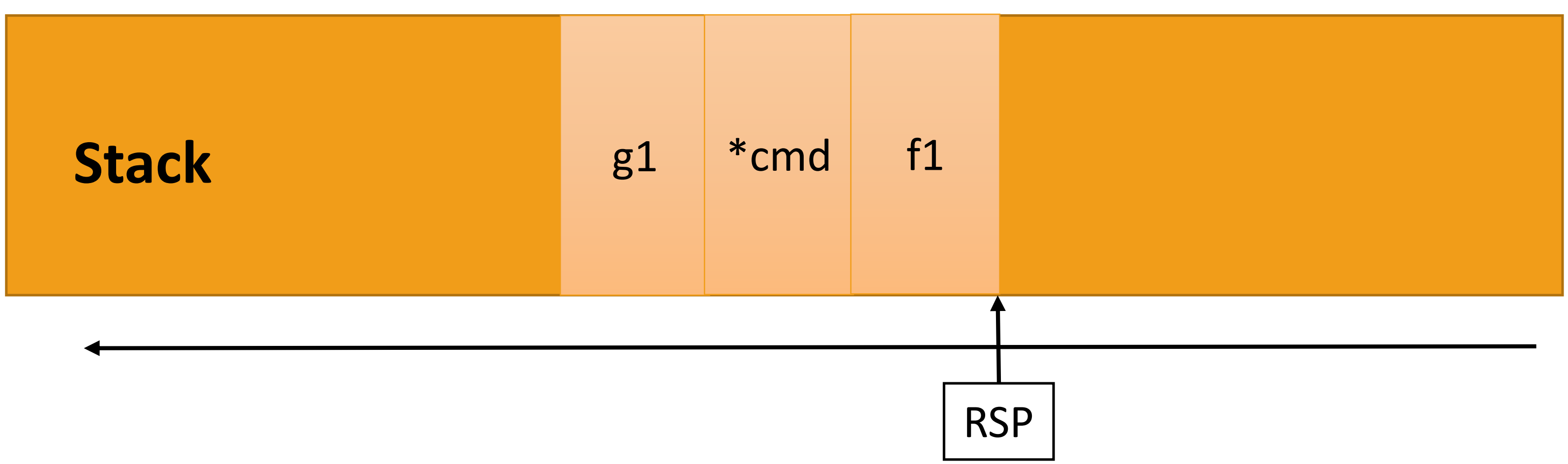
Load argument on register

Redirect control to libc function

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

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

RIP → 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>:
RIP → f1 : push rbp
```