Modern Exploitation and Defenses

CS-576 Systems Security

Instructor: Georgios Portokalidis

Fall 2018

Recap: Broadly Deployed Security Mechanisms

NX-bit → Prevent arbitrary code execution

Stack canaries → Detect and prevent stack overflows

ASLR → Introduce uncertainty on the location of injected shellcode and existing code in a running program

They have raised the bar for attackers

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

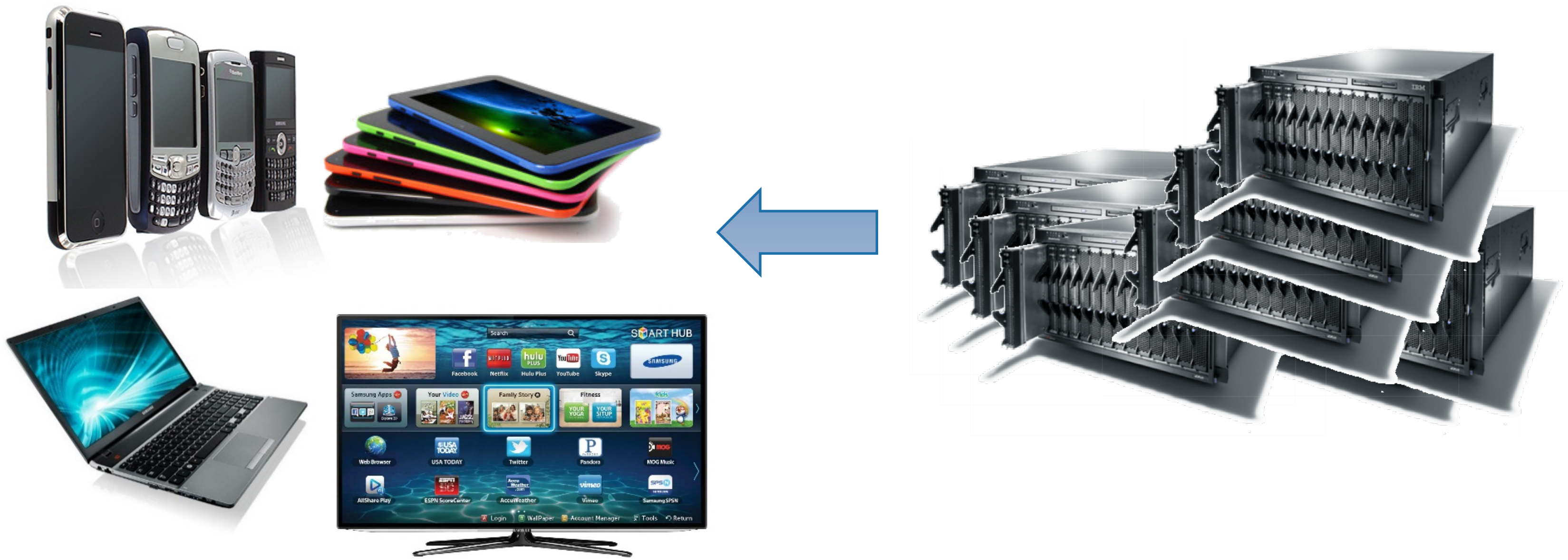
JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

Shift in Target Selection

Clients **Servers**



Shift in Target Selection

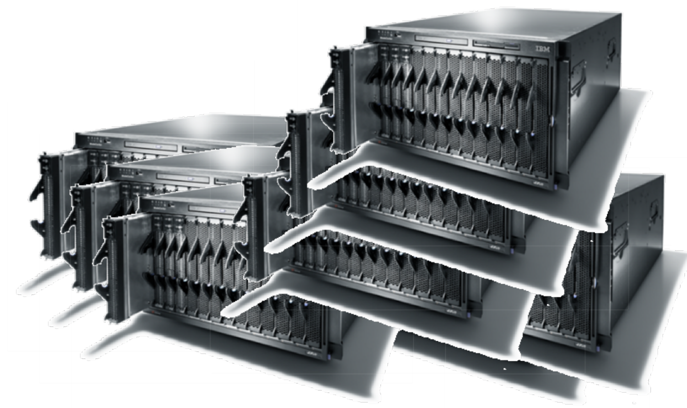
Clients

Web browsers

Flash

Acrobat Reader

Servers



Shift in Target Selection

Clients

Web browsers

Flash

Acrobat Reader

Why?

Software popularity

Large and complex software

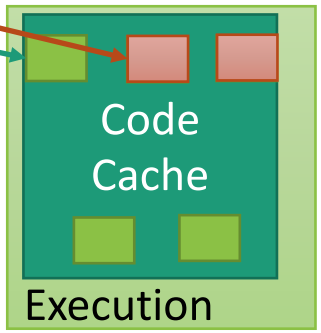
- More buggy

Dynamically translates and executes Javascript

- Attackers can run code on target (even if in isolation)

Recap: Code Injection in the Code Cache

```
<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>
```



ASLR → Code cache location **unknown**

Heap Spraying

Attempt to place shellcode at a predictable location

Mechanisms:

Dynamically expand buffer by appending copies of the shellcode

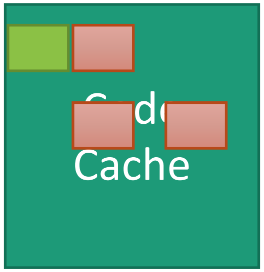
On the fly generate variables

https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/

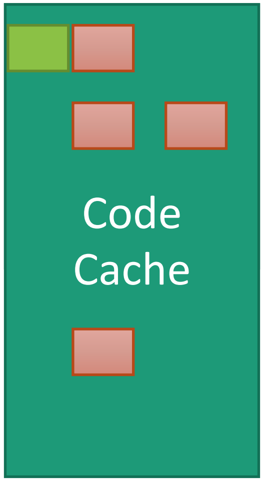
var v1 = “myshellcode”;

var v2 = “myshellcode”;

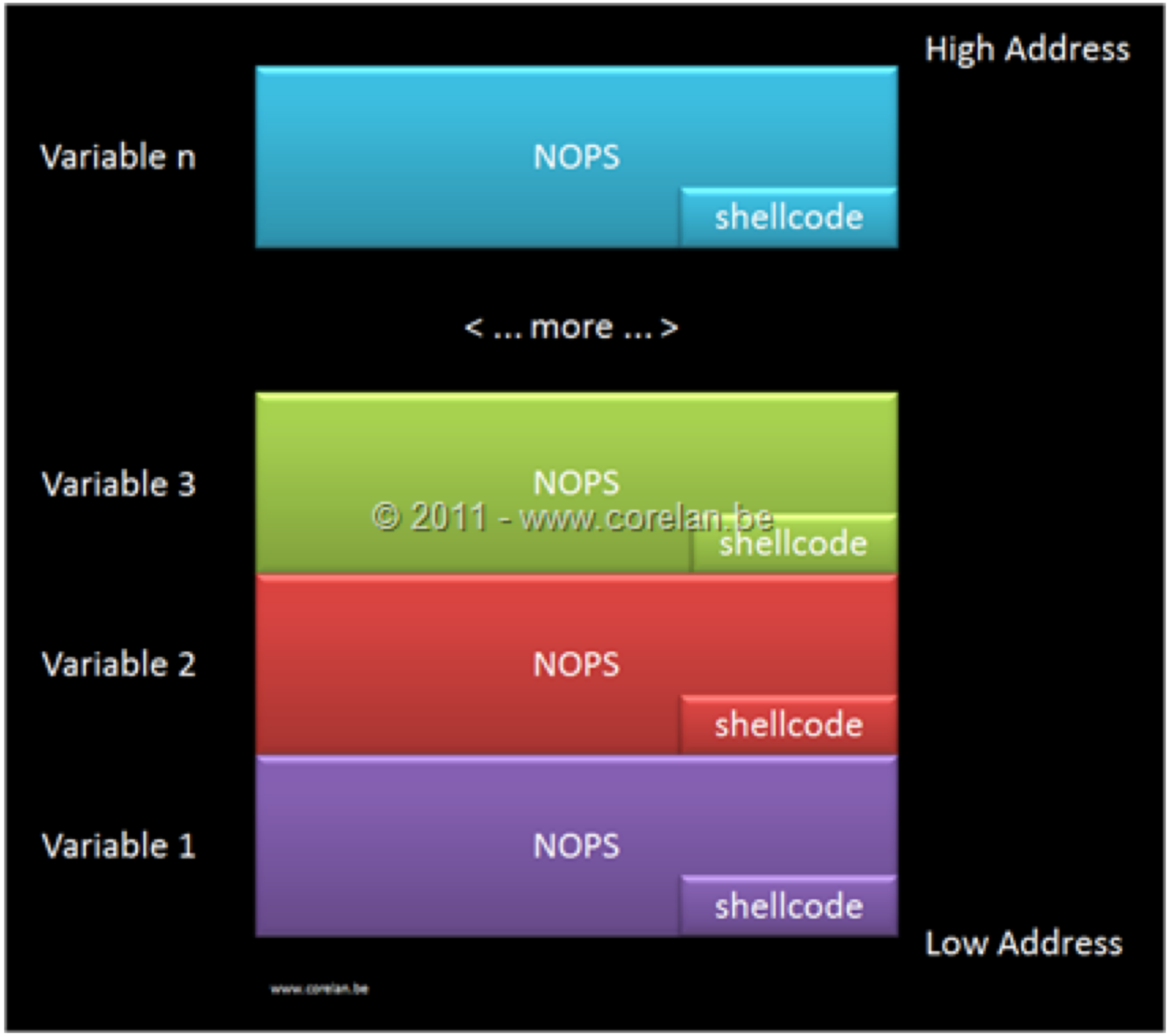
var v3 = “myshellcode”;

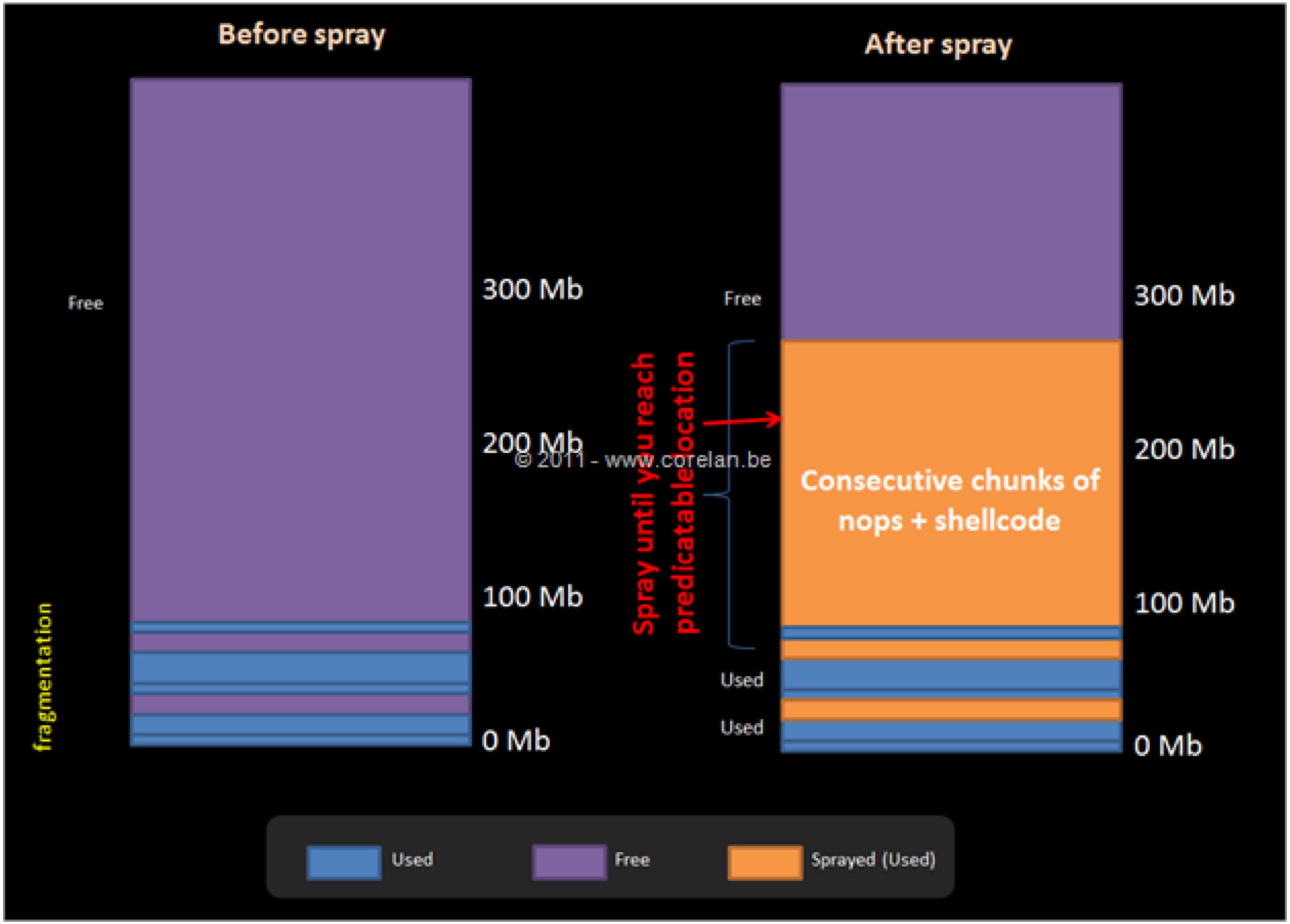


```
var v1 = “myshellcode”;
var v2 = “myshellcode”;
var v3 = “myshellcode”;
var v4 = “myshellcode”;
```



Large NOP Sleds




Before spray
After spray
Free
300 Mb
200 Mb
100 Mb
0 Mb
fragmentation
© 2011 - www.corelan.be
Spray until you reach predicatable location
Free
Used
Used
300 Mb
200 Mb
100 Mb
0 Mb
Consecutive chunks of nops + shellcode
Used
Free
Sprayed (Used)

Summary: Heap Spraying

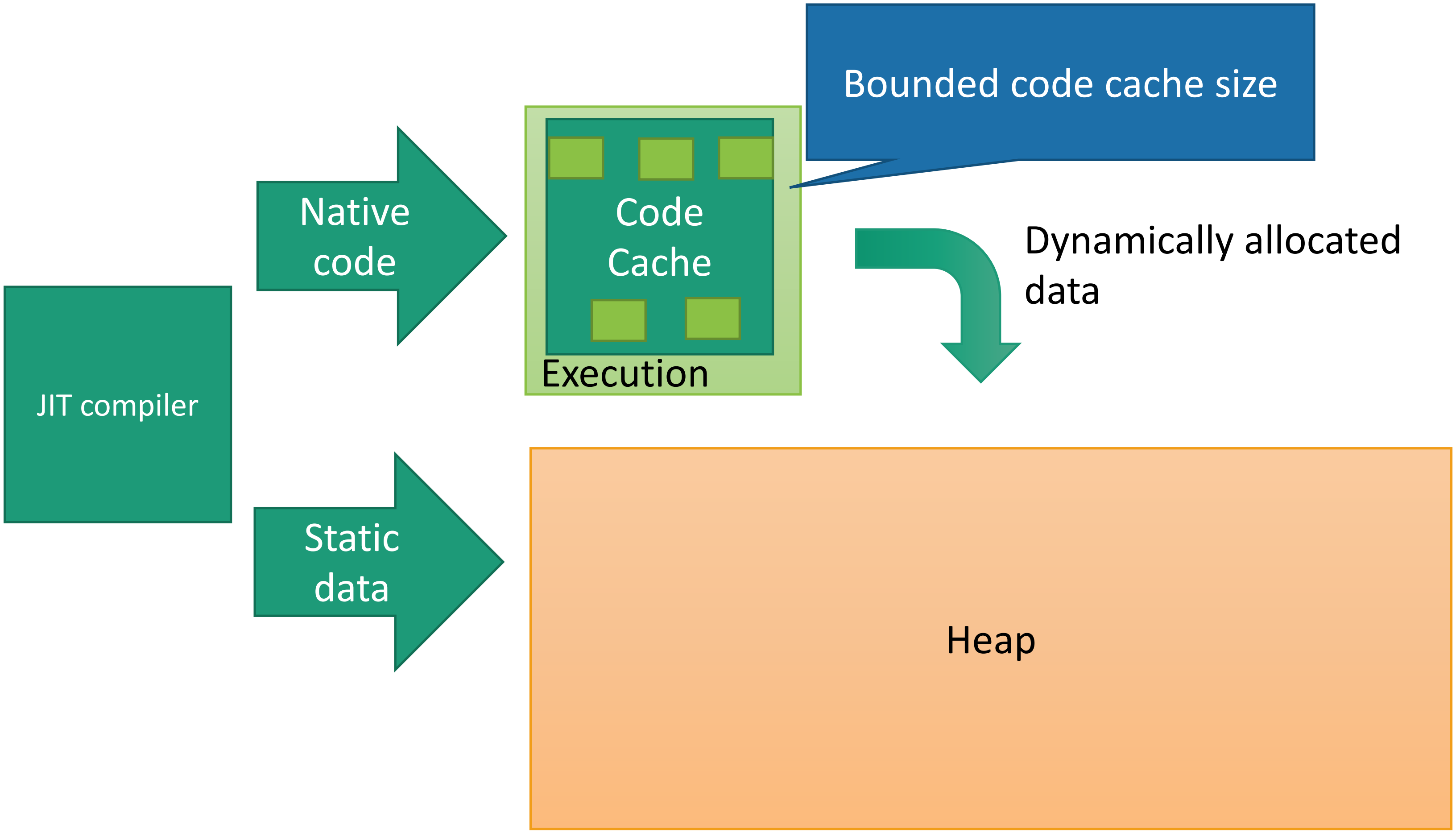
May require multiple attempts

Can possibly defeat ASLR

Heap fragmentation is in play

- May be worse in concurrent systems

Code/Data Separation in the Code Cache



ASLR + Code/data Separation + Finite Code Cache

No More Code Injection

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

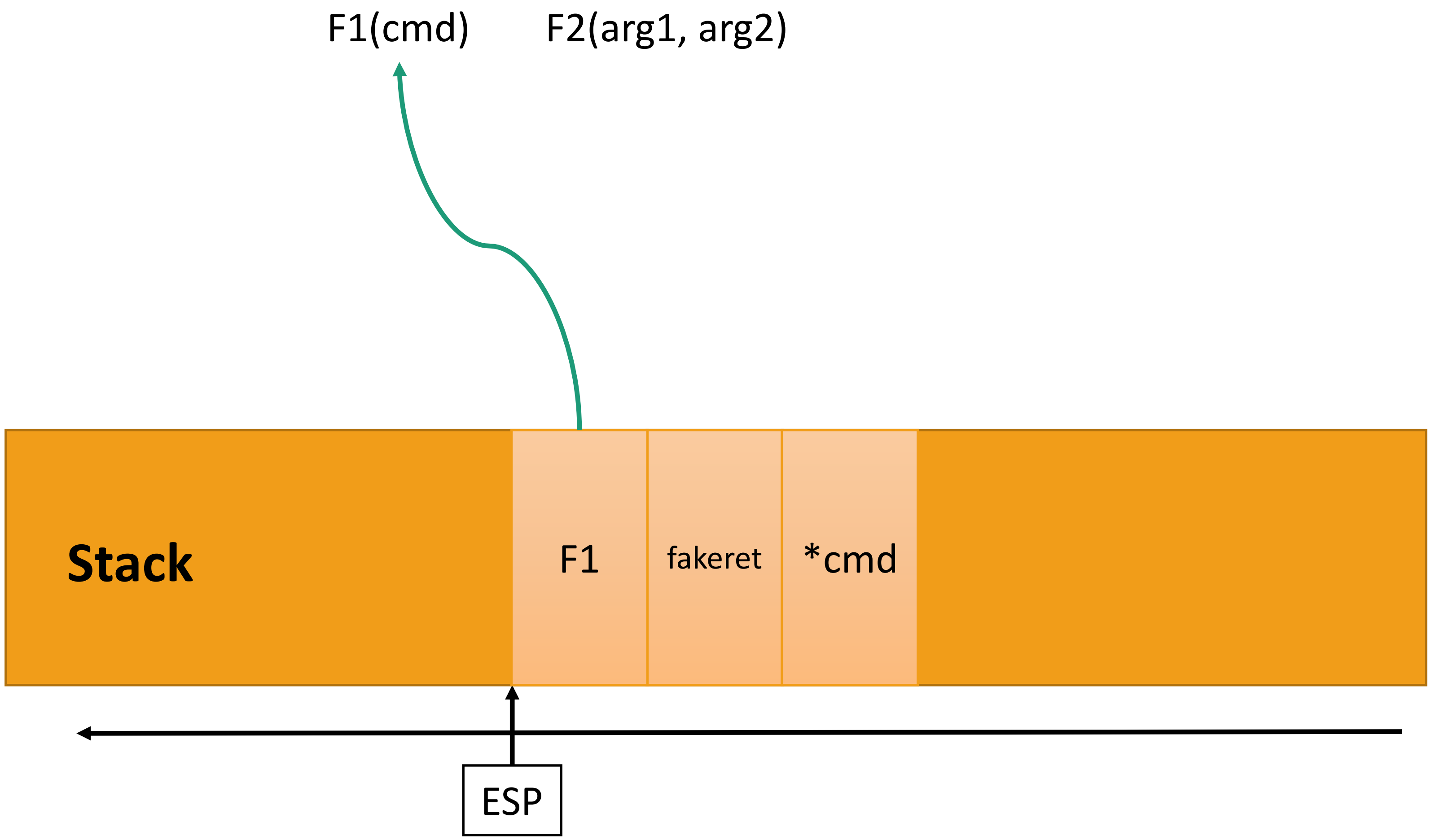
Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

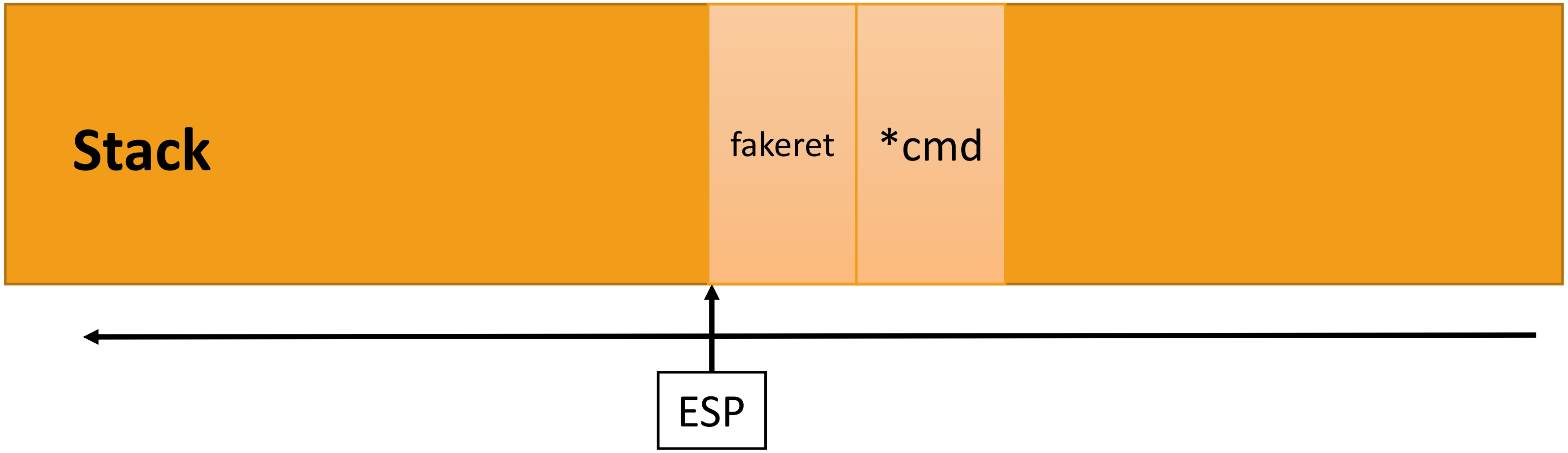
Attacks against CFI and more defenses

Chaining Functions with ret2libc

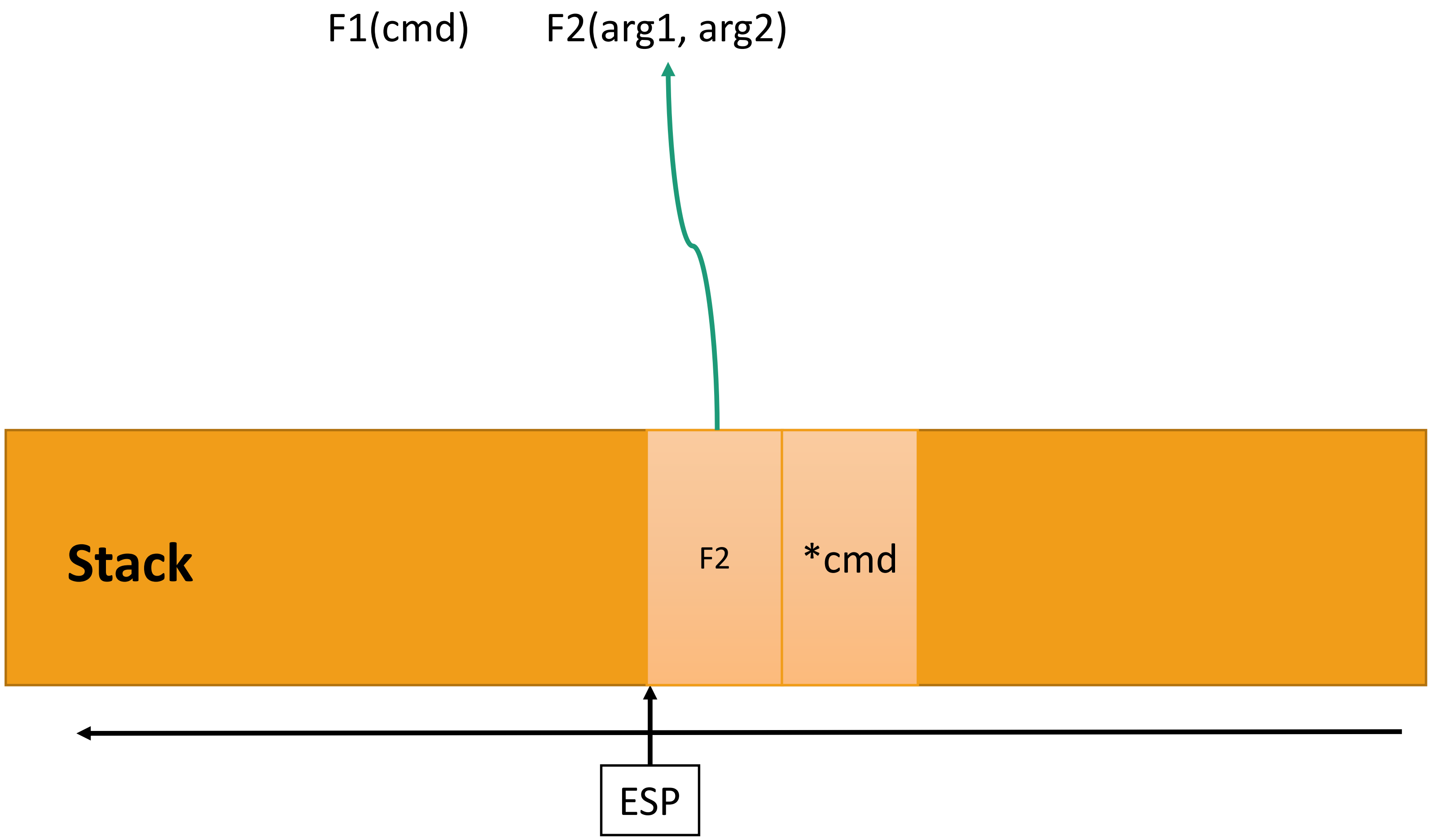


Chaining Functions with ret2libc

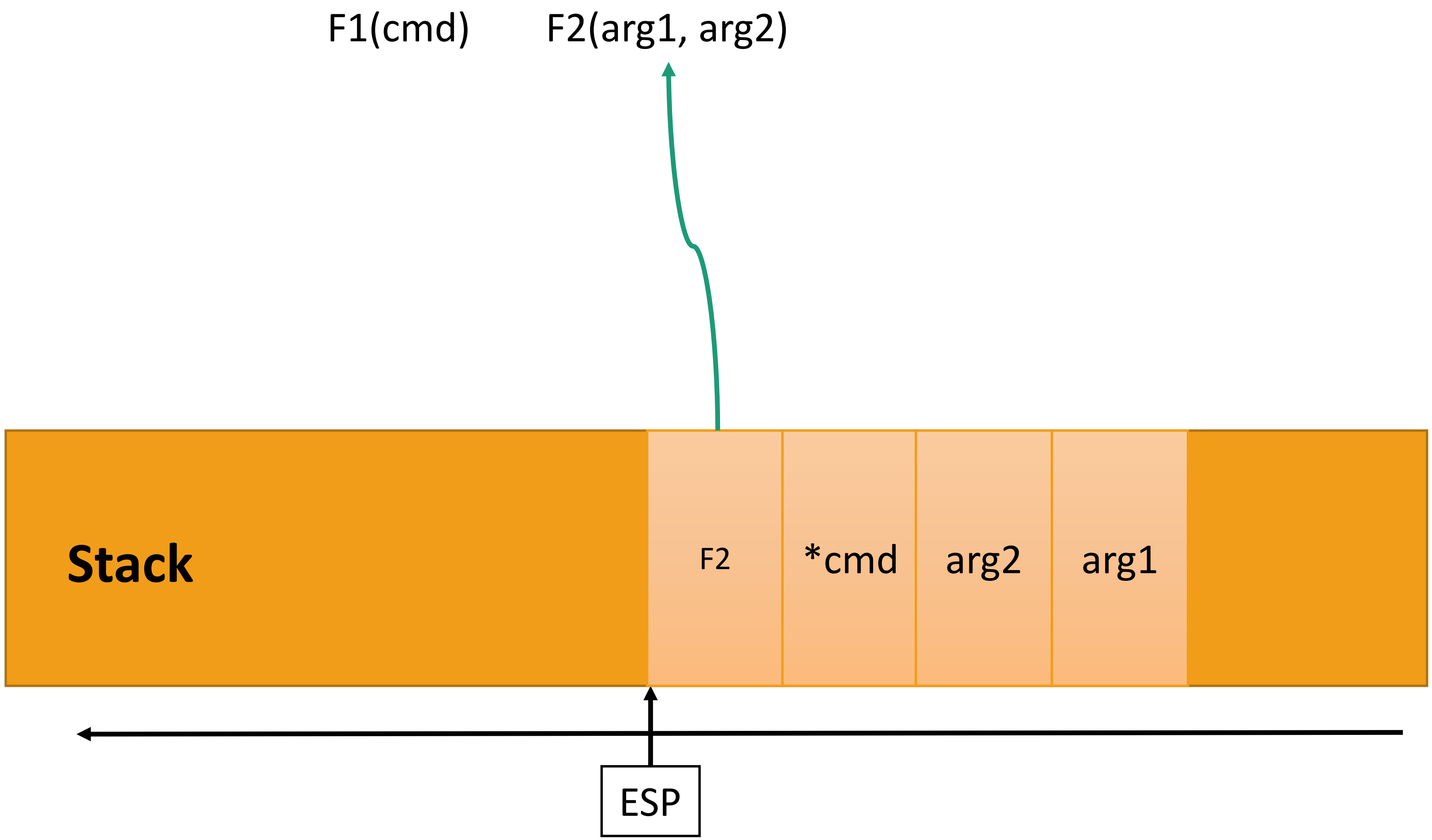
F1(cmd)   F2(arg1, arg2)



Chaining Functions with ret2libc

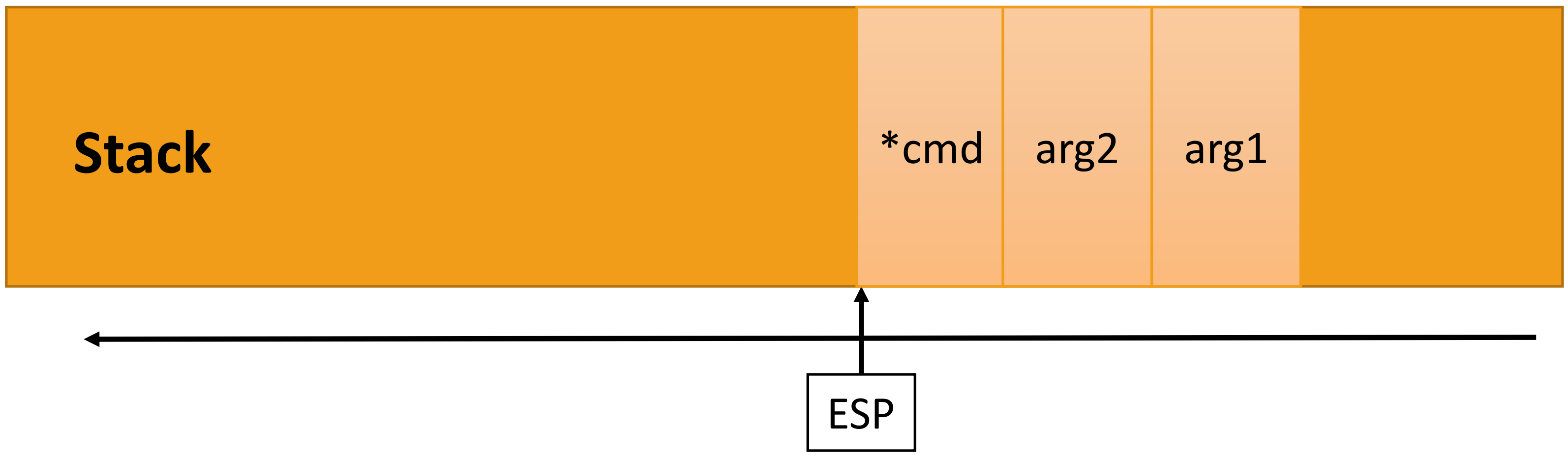


Chaining Functions with ret2libc



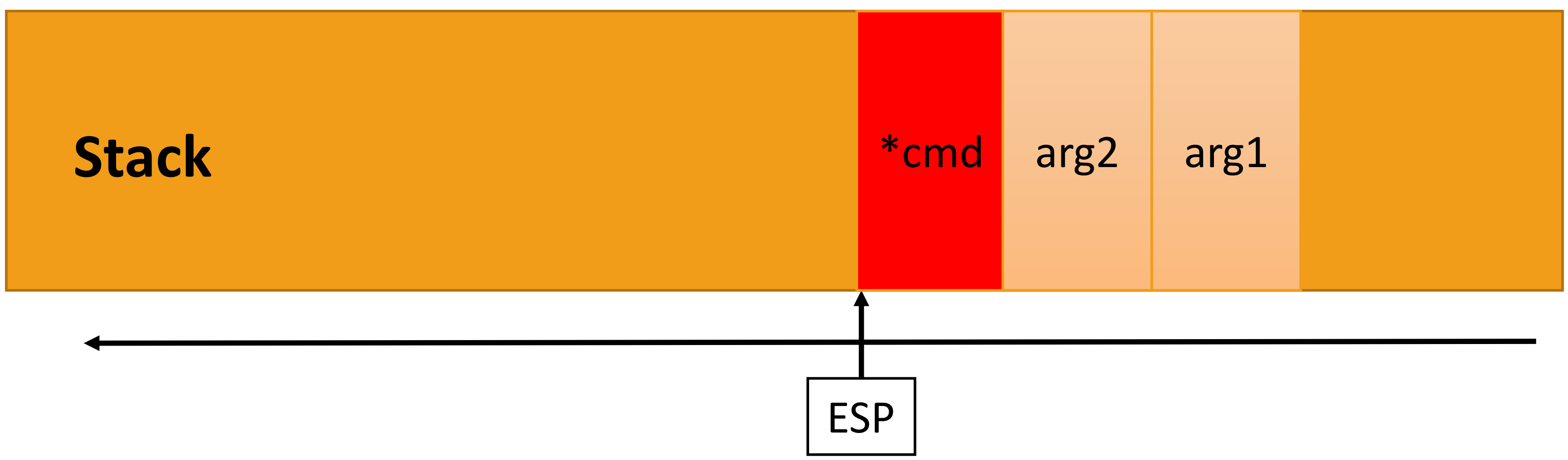
Chaining Functions with ret2libc

F1(cmd)　　F2(arg1, arg2)　　F3(arg3)



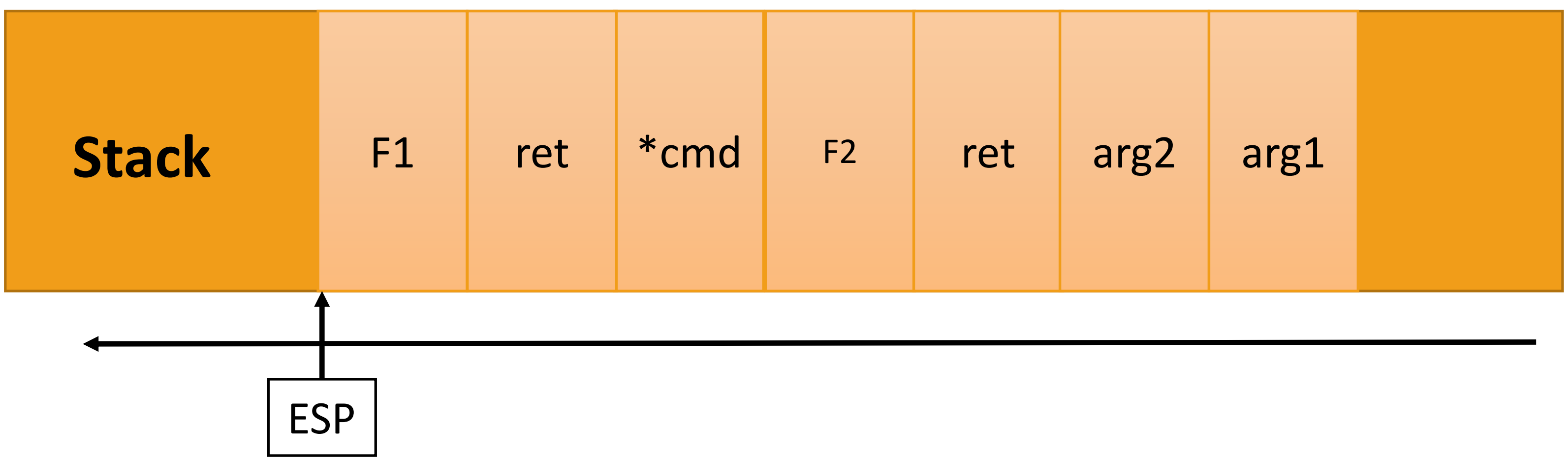
Chaining Functions with ret2libc

F1(cmd) F2(arg1, arg2) F3(arg3)



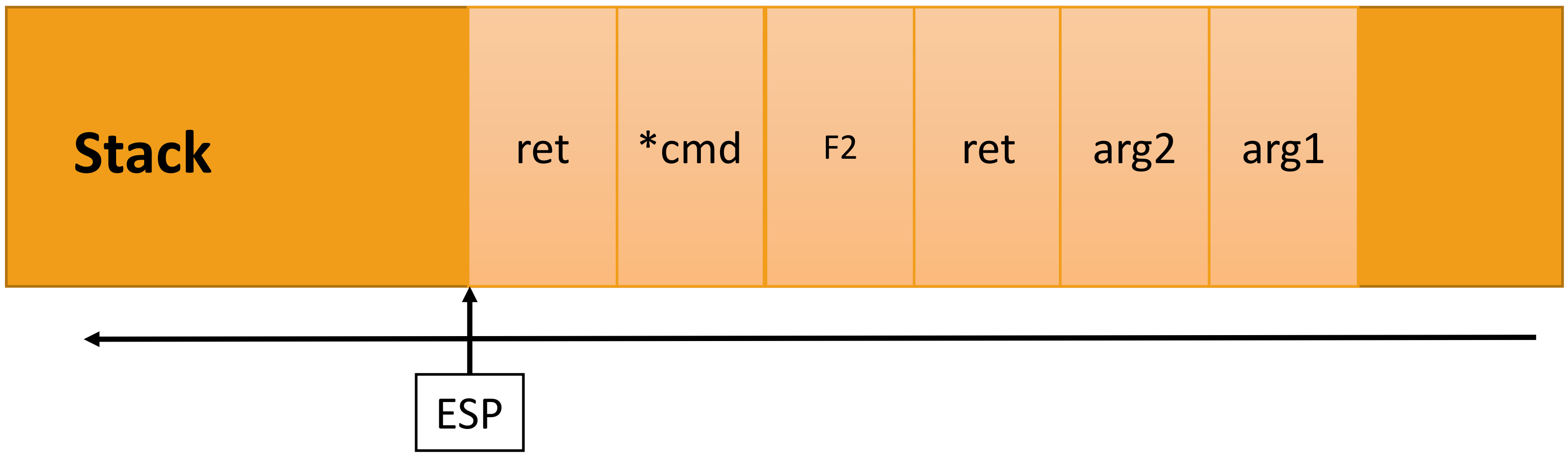
Chaining Functions with ret2libc

We need small gadgets to unwind the stack pointer in a controlled way



Chaining Functions with ret2libc

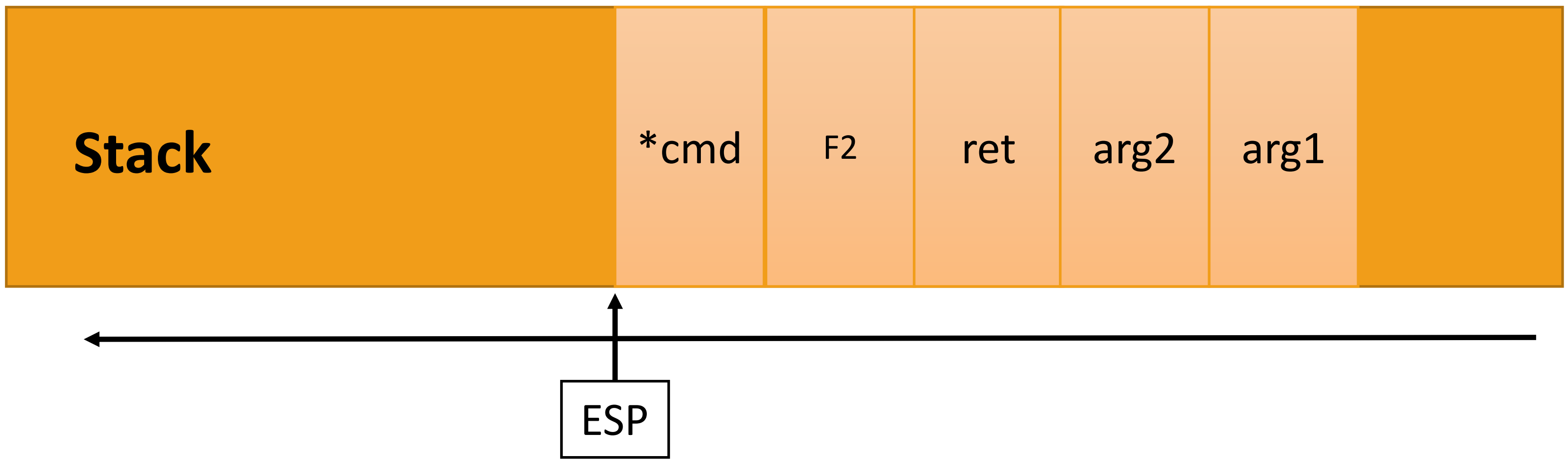
F1(cmd)



Chaining Functions with ret2libc

F1(cmd)

```
pop eax; ret
```

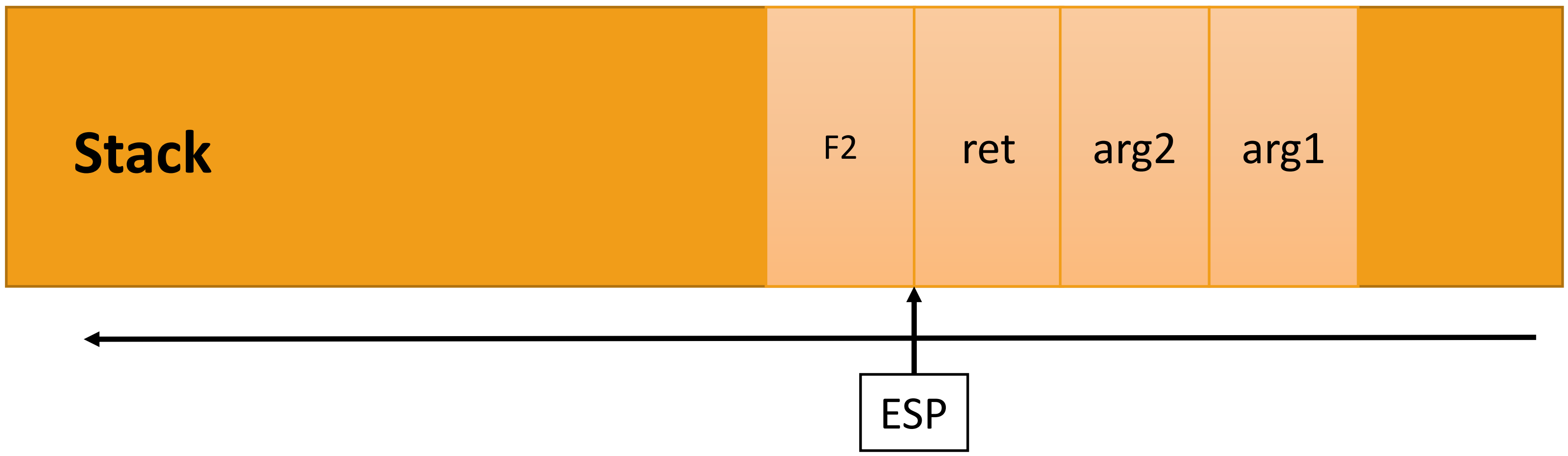


Chaining Functions with ret2libc

F1(cmd)

`pop eax; ret`

F2(arg1, arg2)



Chaining Functions with ret2libc

F1(cmd)

```
pop eax; ret
```

F2(arg1, arg2)

```
add 0x8,esp; ret
```

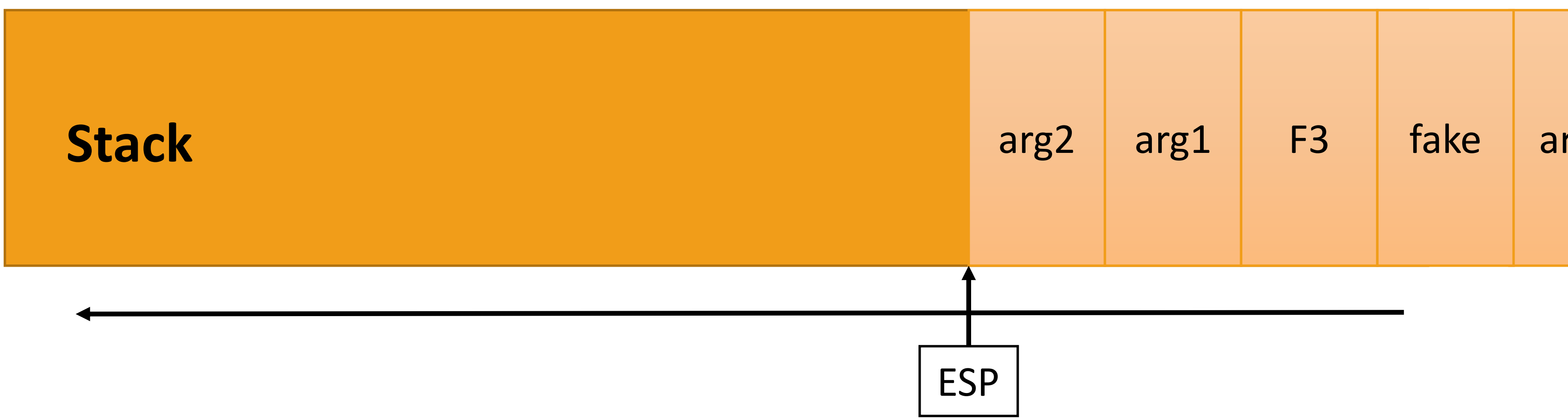
Chaining Functions with ret2libc

F1(cmd)

```
pop eax; ret
```

F3(arg1, arg2)

```
add 0x8,esp; ret
```



Chaining Functions with ret2libc

F1(cmd)

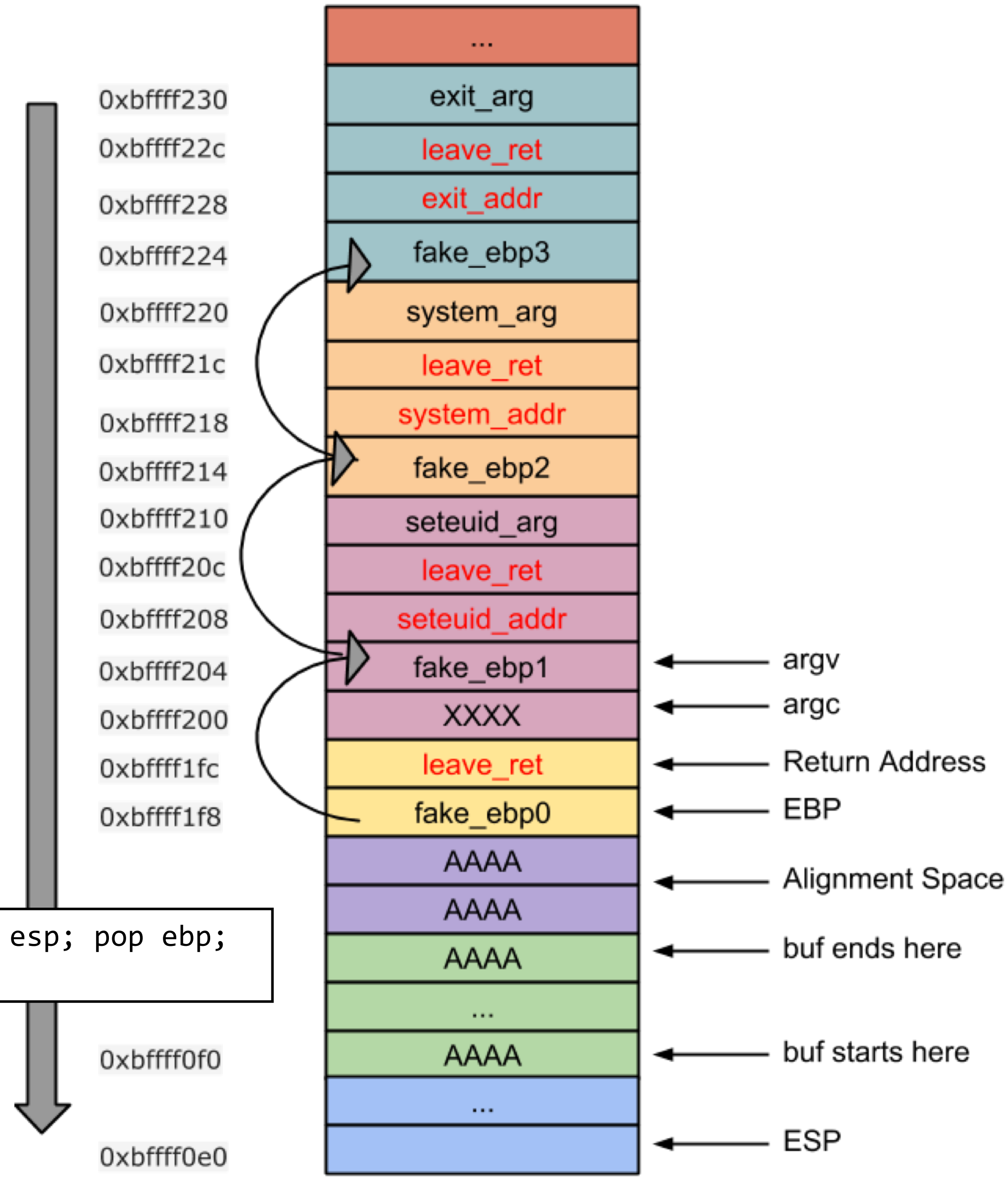
`pop eax; ret`

F2(arg1, arg2)

`add 0x8,esp; ret`

F3(arg3)

https://sploitfun.wordpress.com/2015/05/08/bypassing-nx-bit-using-chained-return-to-libc/




```
0x0804851c <+88>: leave //mov ebp, esp; pop ebp;
0x0804851d <+89>: ret   //return
```

main() Stack Layout - Chained with multiple libc functions

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

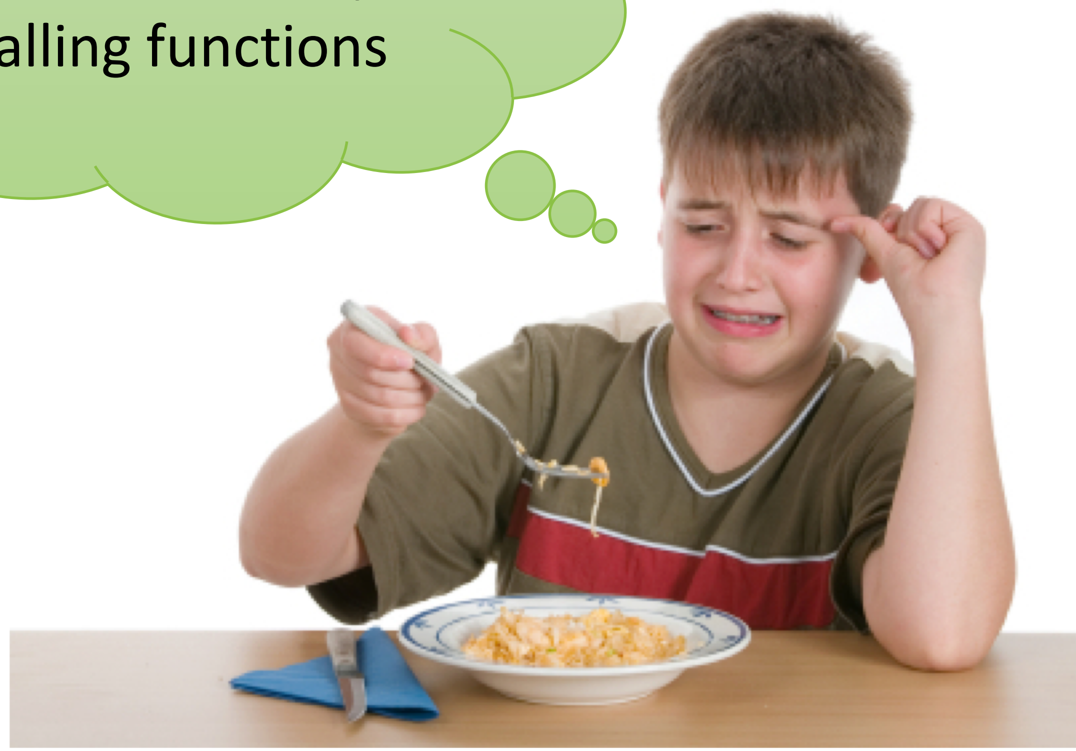
Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

I don't like only calling functions



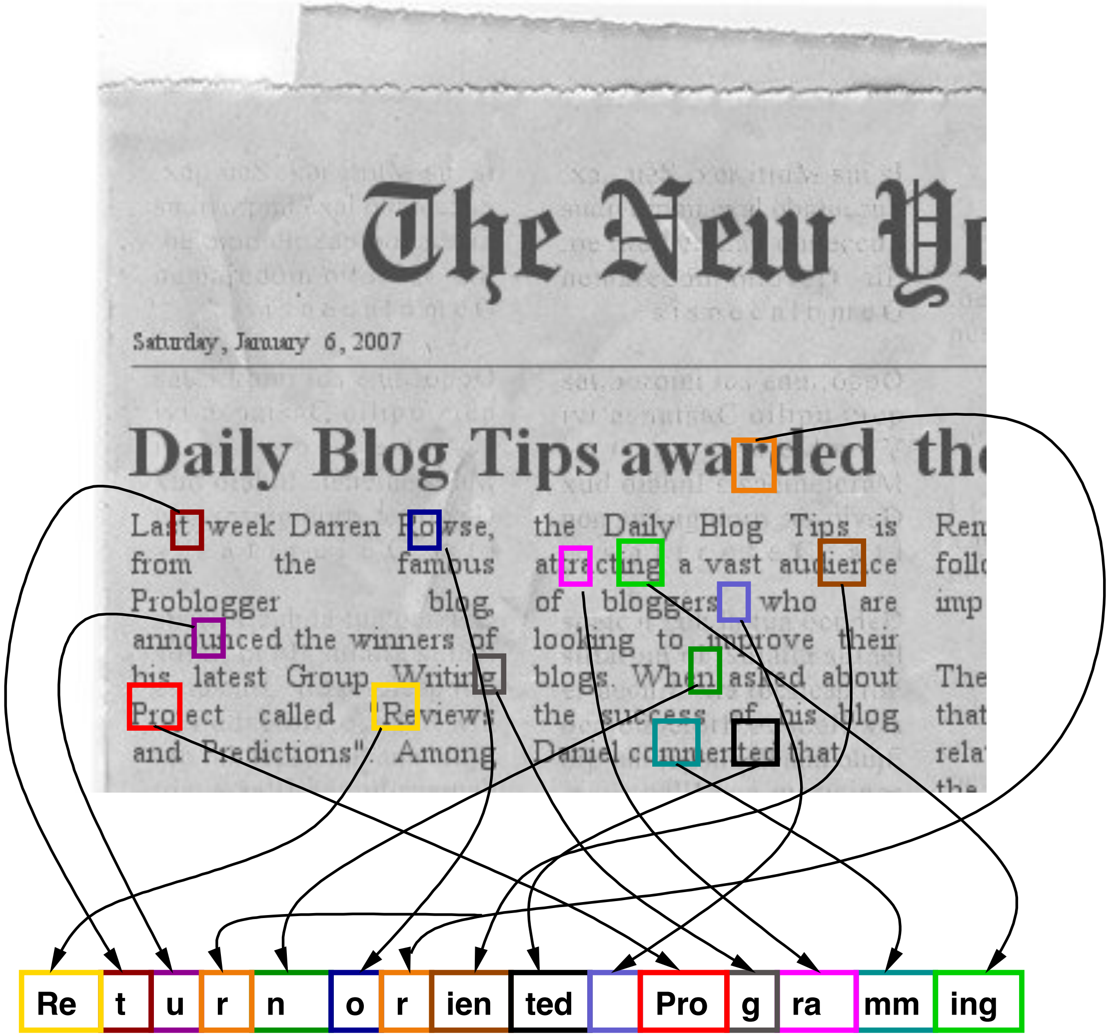
Enter Return-Oriented Programming

Re-use parts of the application's code (gadget) to perform arbitrary computations

A Turing complete machine

Use the stack like a tape providing the data for the computation and the instruction pointer

A Code Collage



https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

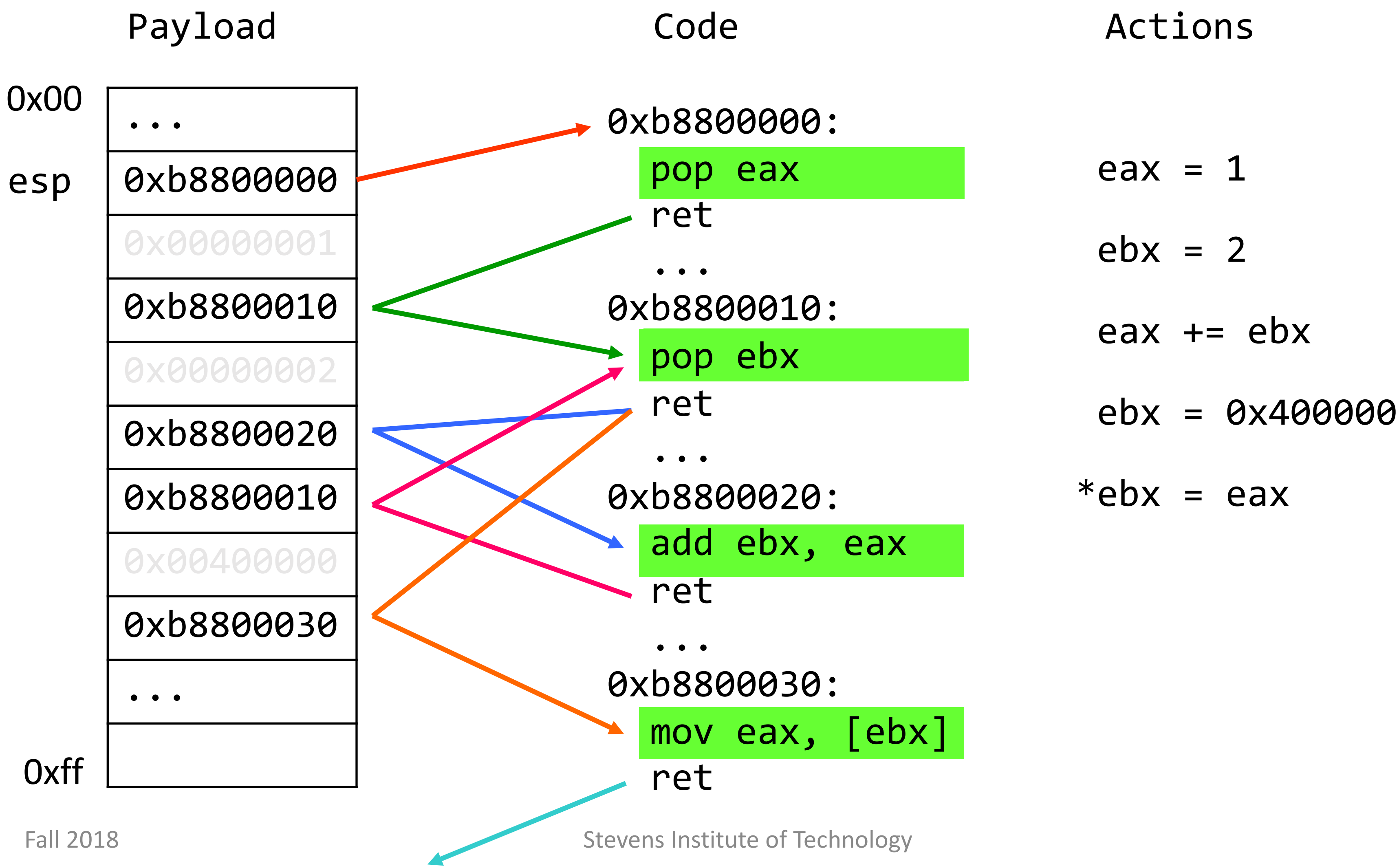
Gadgets

```
mov (%rcx),%rbx
test %rbx,%rbx
je 41c523 <main+0x803>
mov %rbx,%rdi
callq 42ab00
mov %rax,0x2cda9d(%rip)
cmpb $0x2d,(%rbx)
je 41c4ac <main+0x78c>
mov 0x2cda8d(%rip),%rax
ret
test %rbx,%rbx
mov $0x4ab054,%eax
cmove %rax,%rbx
mov %rbx,0x2cda6a(%rip)
test %rdi,%rdi
je 41c0c2 <main+0x3a2>
mov $0x63b,%edx
mov $0x4ab01d,%esi
callq 46cab0 <sh_xfree>
ret
```

```
mov %rax,0x2d2945(%rip)
mov 0x2cda16(%rip),%rax
test %rax,%rax
je 41c112 <
movzbl (%ra
callq 41b64
mov 0xb8(%r
cmp 0xc(%rsp),
mov %rax,0x2d2670(%rip)
je 41c214 <main+0x4f4>
xchg %ax,%ax
mov (%rsp),%rdx
movslq %r15d,%rax
mov (%rdx,%rax,8),%r14
ret
je 41c214 <main+0x4f4>
cmpb $0x2d,(%r14)
jne 41c214 <main+0x4f4>
movzbl 0x1(%r14),%r12d
movl $0x0,0x18(%rsp)
cmp $0x2d,%r12b
```

```
je 41c440 <main+0x720>
xor %ebp,%ebp
mov $0x4c223a,%ebx
add $0x1,%r14
jmp 41c1a3 <main+0x483>
cmp (%rbx),%r12b
mov %ebp,%r13d
jne 41c188 <main+0x468>
mov %rbx,%rsi
test %eax,%eax
xchg %ax,%ax
jne 41c188 <main+0x468>
movslq %ebp,%rax
ret
cmpl $0x1,0x4ab3c8(%rax)
je 41c461 <main+0x741>
mov (%rsp),%rcx
add $0x1,%r15d
movslq %r15d,%rdx
mov (%rcx,%rdx,8),%rdx
test %rdx,%rdx
je 41cefd <main+0x11dd>
```

An Example


Payload
Code
Actions
0x00
esp
0xff
...
0xb8800000
0x00000001
0xb8800010
0x00000002
0xb8800020
0xb8800010
0x00400000
0xb8800030
...
0xb8800000:
pop eax
ret
...
0xb8800010:
pop ebx
ret
...
0xb8800020:
add ebx, eax
ret
...
0xb8800030:
mov eax, [ebx]
ret
eax = 1
ebx = 2
eax += ebx
ebx = 0x400000
*ebx = eax

Current State of the Art

First-stage ROP code for bypassing NX

- Allocate/set W+X memory (VirtualAlloc, VirtualProtect, …)
- Copy embedded shellcode into the newly allocated area

Second stage jumps to injected code

Pure-ROP exploits

- In-the-wild exploit against Adobe Reader XI
- CVE-2013-0640

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

Fine-Grained Code Randomization

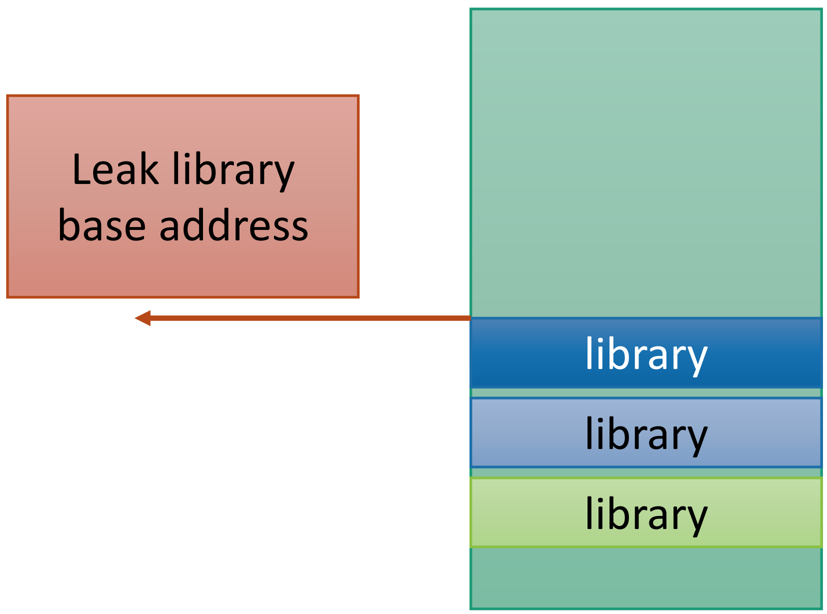
Randomize the layout of the code within a library/executable

Aims to defeat ROP-style attacks that rely on a memory leak to de-randomize the base address of a code segment

- This allows using the gadgets within

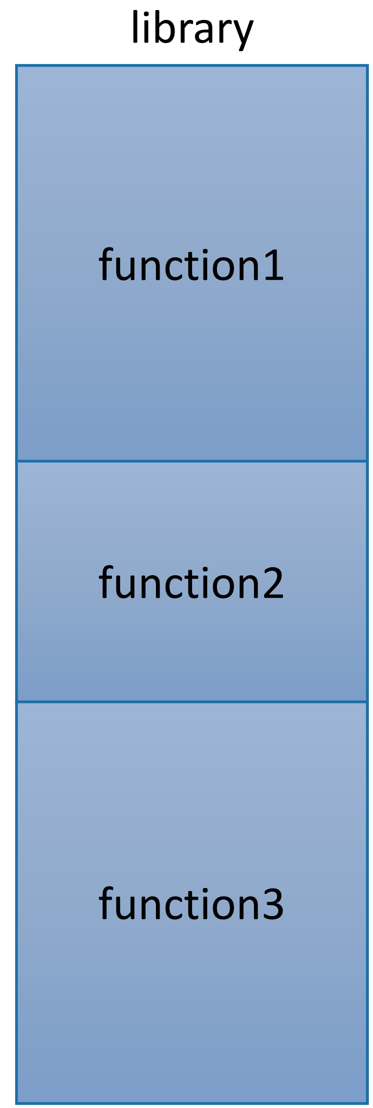
Can be applied at different levels with increasing overheads

- Function
- Basic block
- Instruction

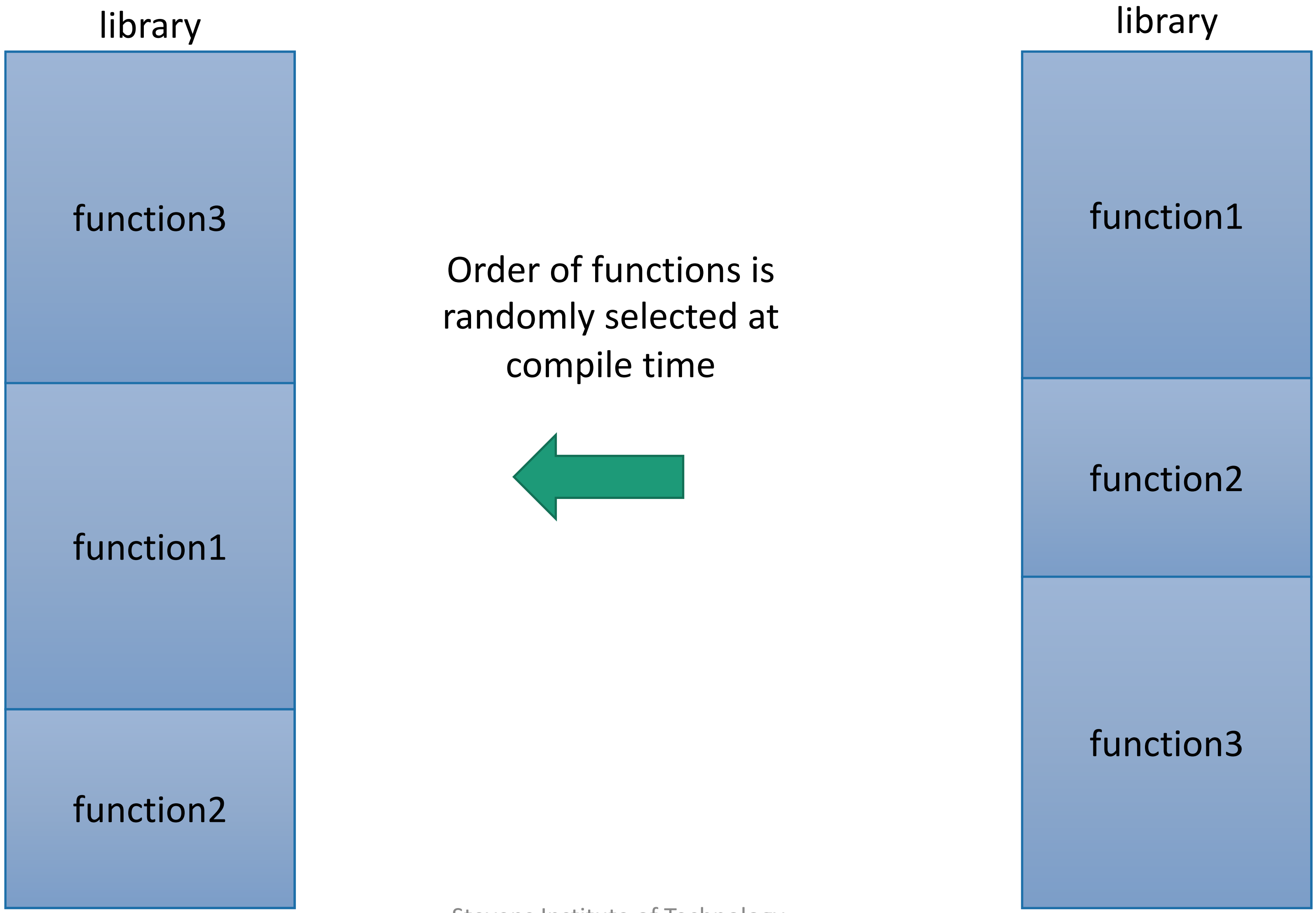

Leak library base address
library
library
library

Known library base address

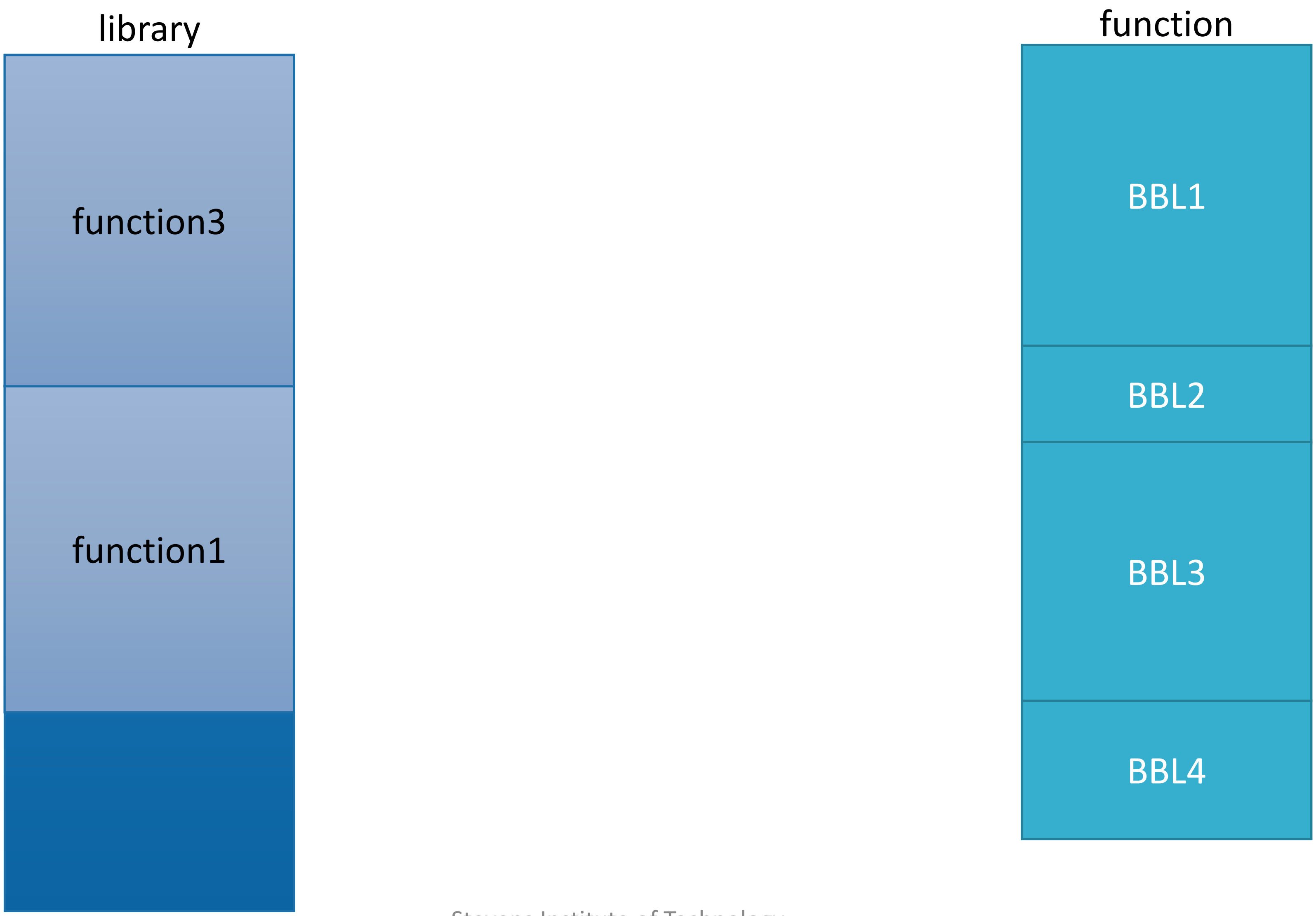
The address of every instruction is known



Function-level Randomization

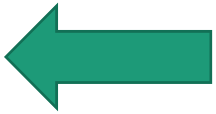


Basic Block-level Randomization



Basic Block-level Randomization

Order of basic blocks is randomly selected at compile time



Glue code may be inserted

Instruction-level Randomization

Similar concept to function and BBL-level randomization

Instruction may be

- Moved within a block (e.g., by adding random number of NOPs between them)
- Replaced with equivalent functionality
- Substituted to use different registers
- ....

Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

JIT-ROP

Just-In-Time ROP chain generation

Can bypass fine-grained randomization

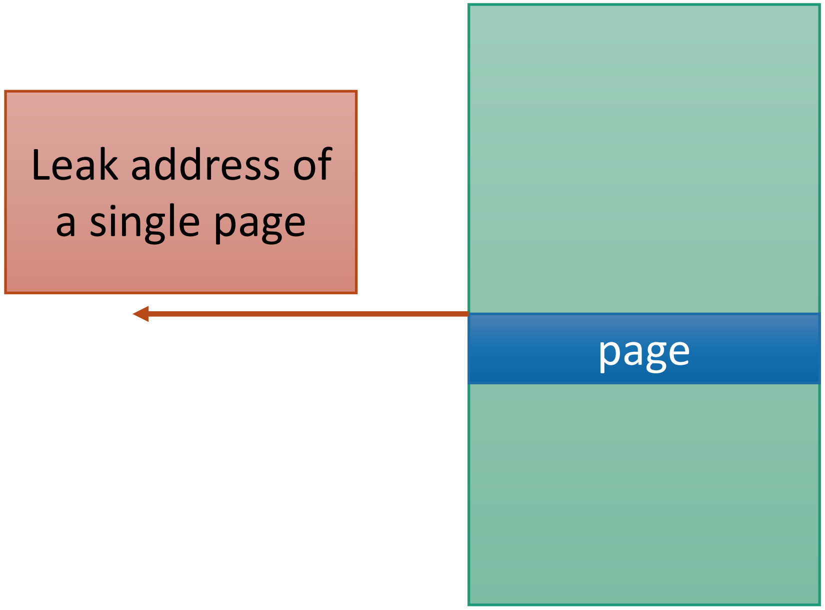
- When a memory leak can be repeatedly triggered
- Example: Leaks that can be triggered from JS

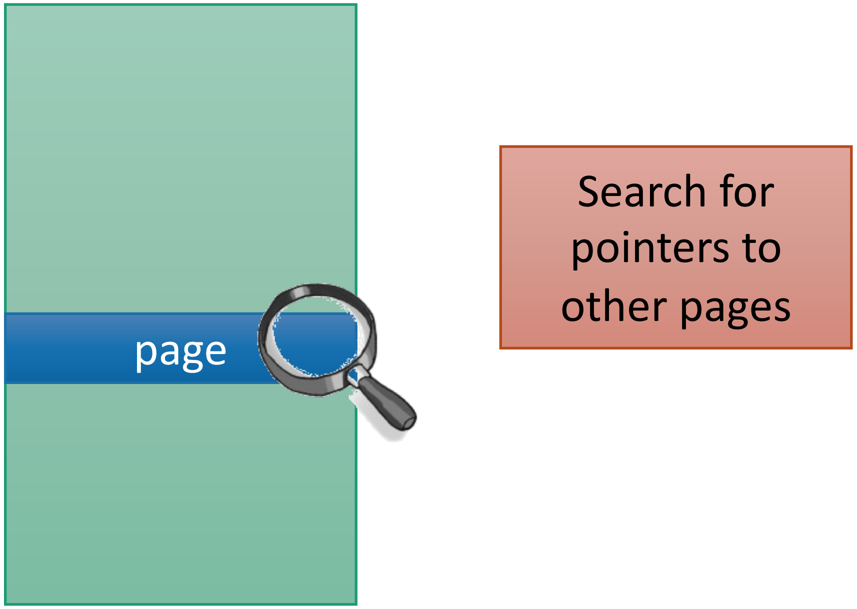
Main idea:

Dynamically leak memory and locate gadgets for ROP

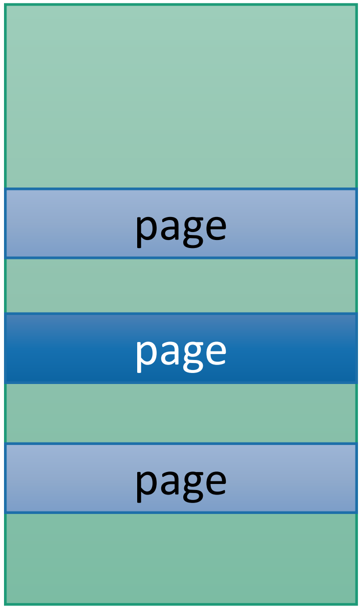
Construct ROP chain and exploit control-flow hijacking vulnerability

https://cs.unc.edu/~fabian/papers/oakland2013.pdf

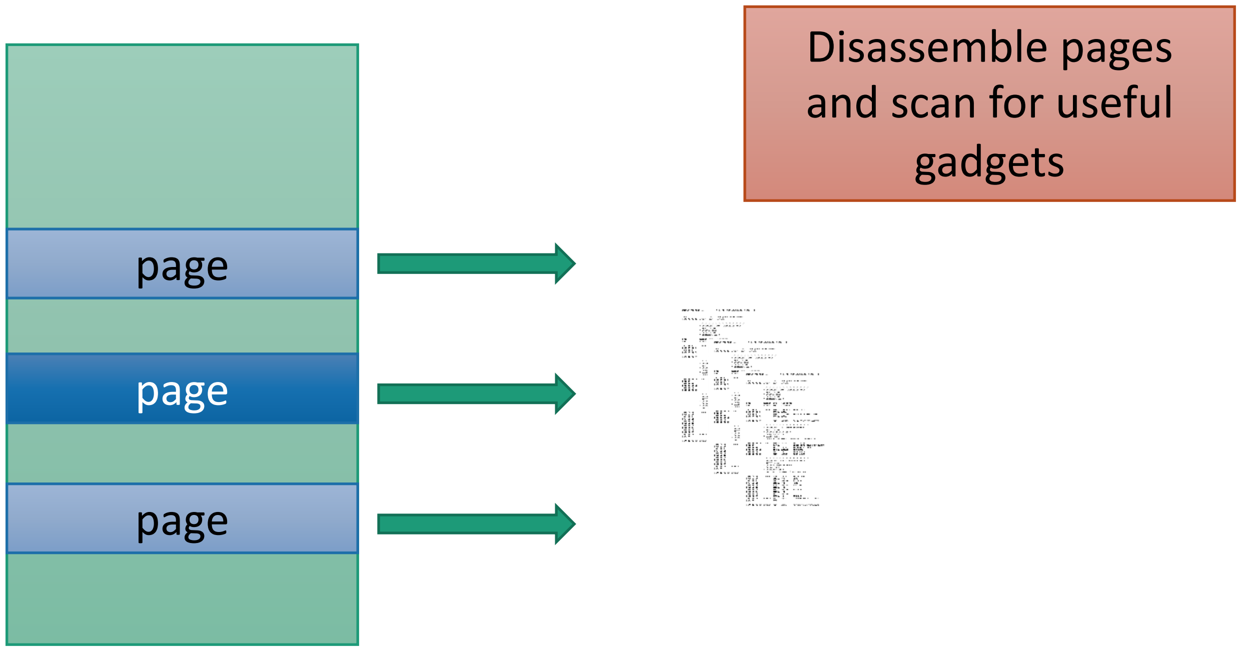

Leak address of a single page
page


page
Search for pointers to other pages

Repeat process for newly discovered pages



Just-in-time Disassembly



Topics

Attackers shift towards client programs

Back to return-to-libc

Return-oriented programming

Fine-grained code randomization

JIT-ROP

Control-flow Integrity (CFI)

Attacks against CFI and more defenses

Attacker Modus Operandi

Find memory corruption bug

- **Manipulate to take over program counter**

Find ASLR bypass

- Leak memory layout
- Spray memory
- Weakly or non-randomized sections/memory

Inject ROP payload

- Break W^X semantics

Inject code

Attacker Modus Operandi

Find memory corruption bug

- Manipulate to take over program counter

Control-flow Integrity aims to restrict the arbitrary manipulation of the program counter

Control Flow Manipulation

Function calls

```
my_function(arg1, arg2)
```

```
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```

Function returns

```
return;
```

```
return 100;
```

If statements

```
if (cond) {
} else {
}
```

Loops

```
for () { }
```

```
while { }
```

```
do { } while
```

Break/continue

```
while (true) {
     if (cond)
          break;
}
```

```
while (cond) {
     if (cond2)
          continue;
}
```

Switch statement

```
switch (cond) {
   val1: … break;
   val2: … break;
}
```

goto statement

```
goto label1;
…
Label1:
```

Control-Flow Hijacking Prone Statements

Statements where the target statement cannot be known a priori

- Indirect control-flow transfers

```
return;
```

```
return 100;
```

```
switch (cond) {
   val1: … break;
   val2: … break;
}
```

Indirect calls, returns, and some switches

```
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```

Calls to virtual functions are indirect calls

```
Class C {
   virtual void vcall(void);
}

C obj = new C();

obj->vcall():
```

Easily Observable in Machine Code

C Code | **Machine Code**

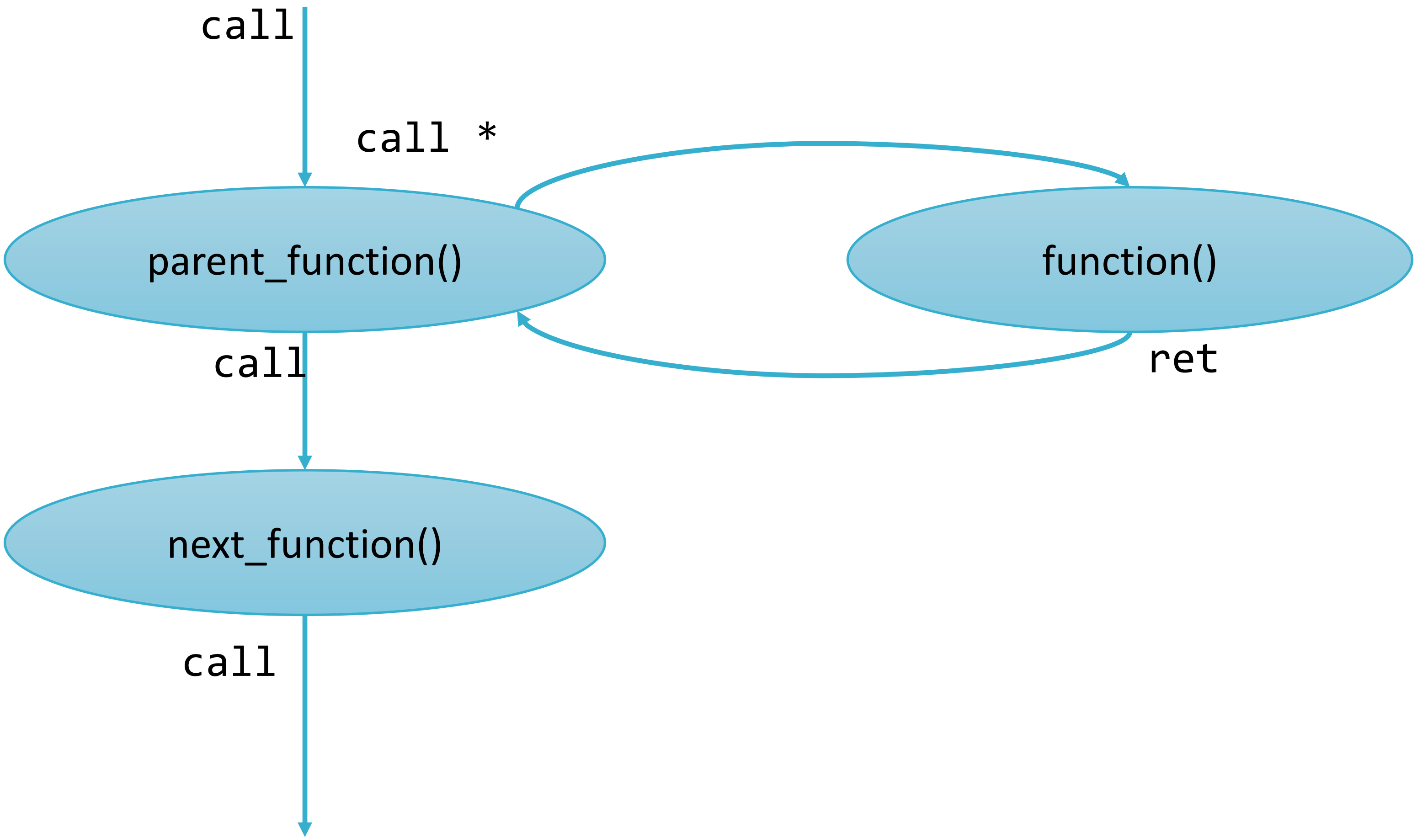
```
return;      return 100;
```
→
```
ret
```

```
switch (cond) {
   val1: … break;
   val2: … break;
}
```
→
```
jmp *(%rax)
```

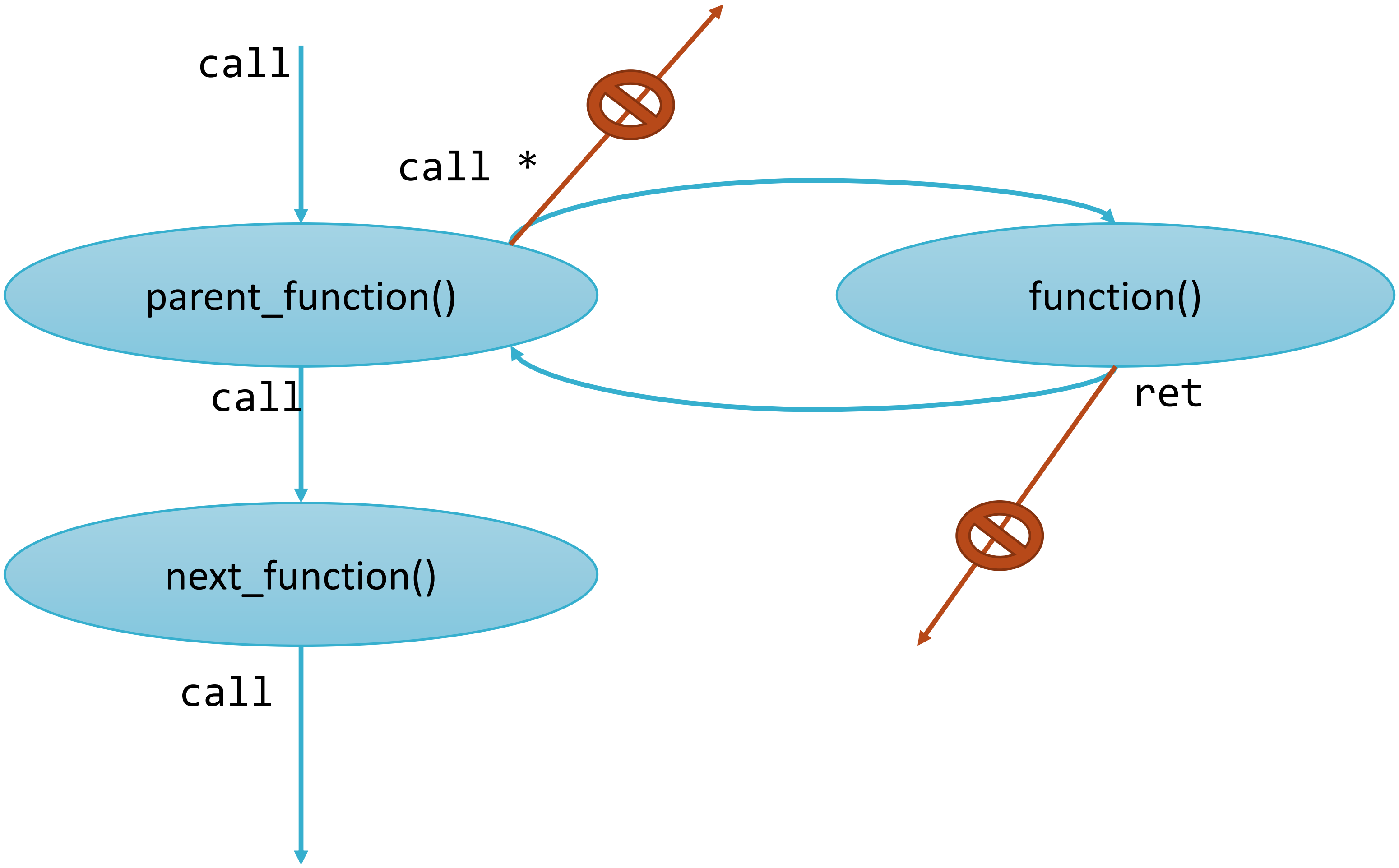
```
void (*fptr)(arg1_type, arg2_type) = &my_function;
fptr(arg1, arg2);
```
→
```
jmp *(%rax)
call *(%rax)
```

```
Class C {
   virtual void vcall(void);
}

C obj = new C();

obj->vcall():
```
→
```
call *(%rax)
```

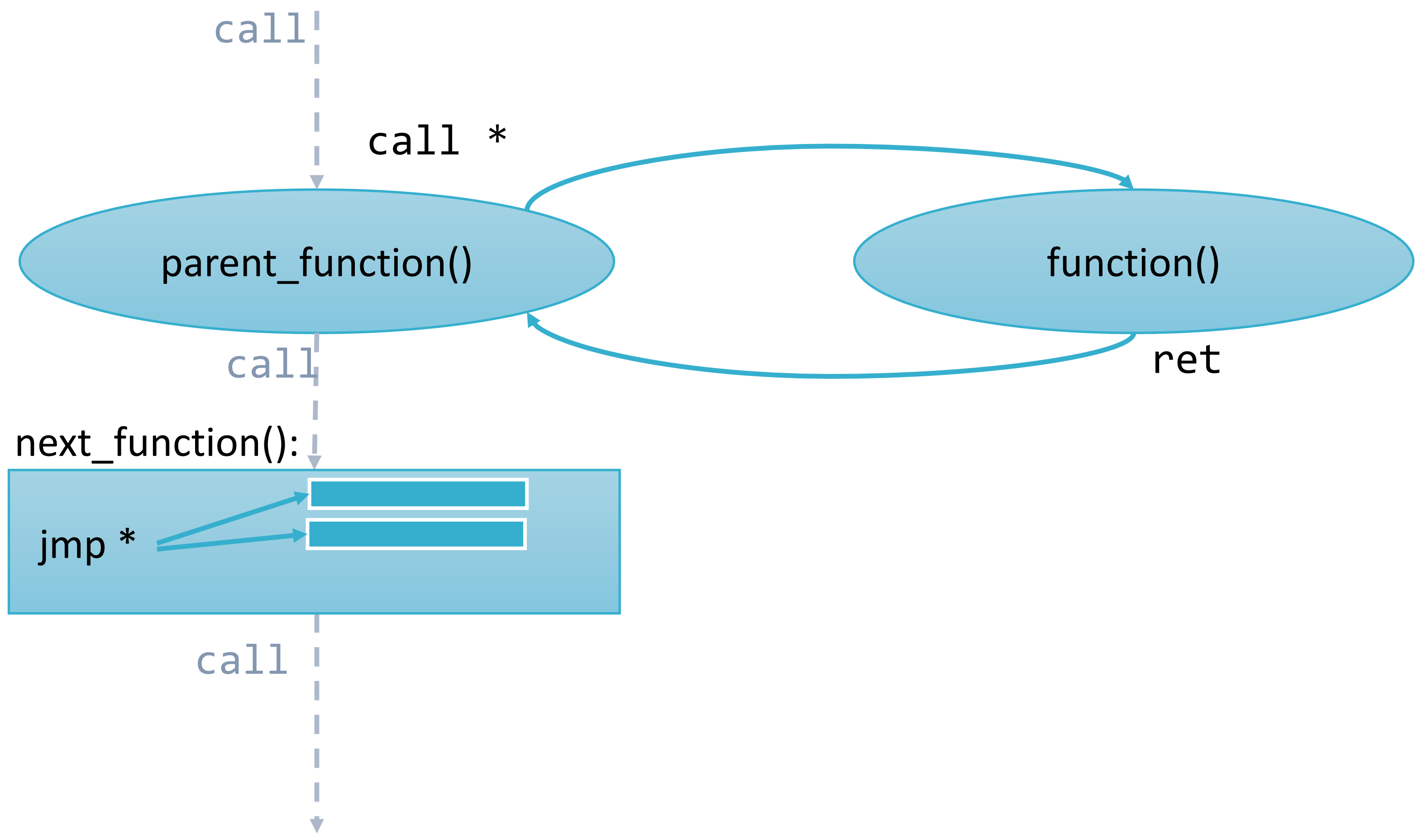
Function Call Graph (FCG)



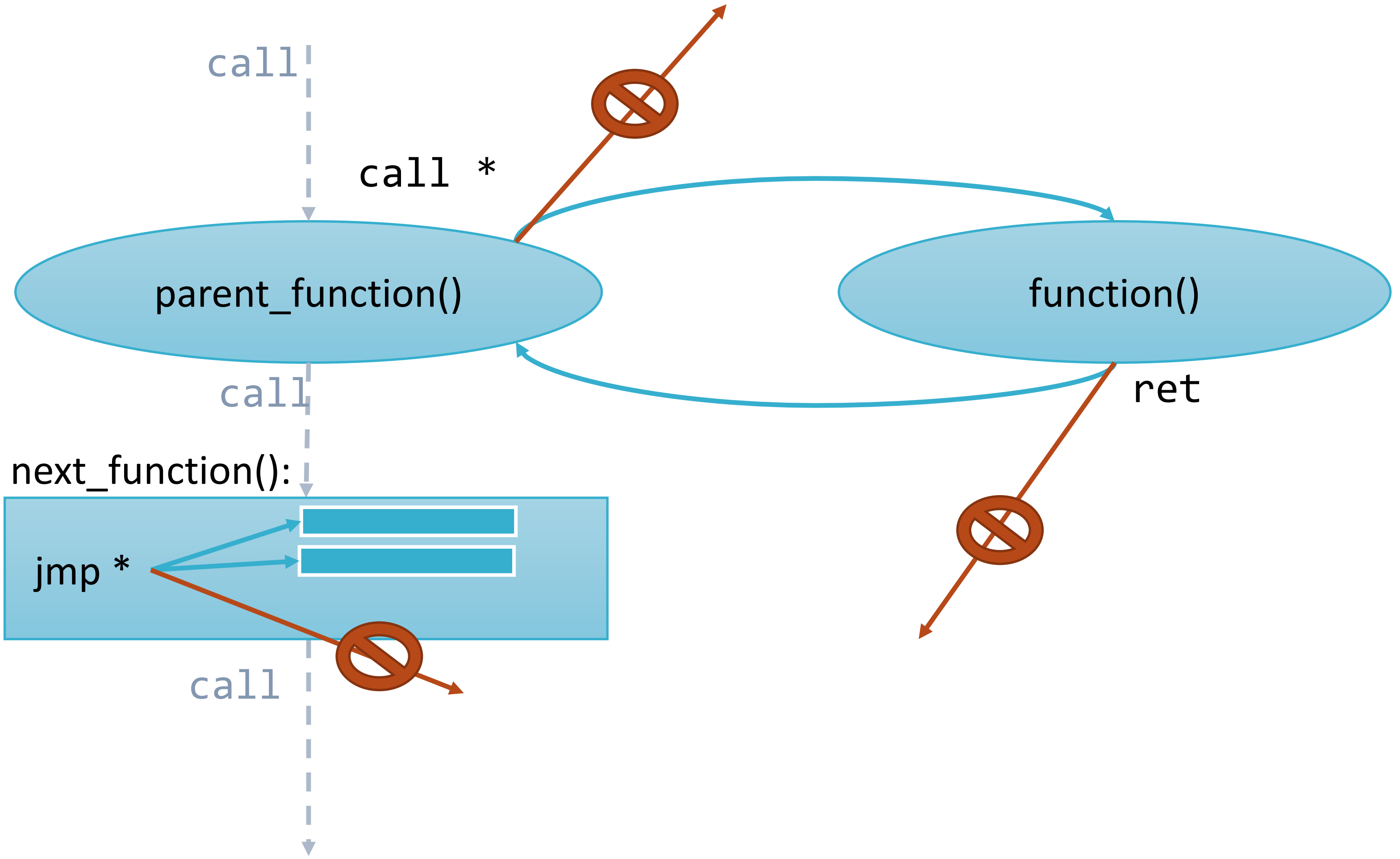
FCG Enforcement



Control-flow Graph (CFG)
Indirect flows only



CFI - CFG Enforcement



Extracting the CFG

With source code

- More reliable
- Still not perfect
- How to handle
  - Dynamically loaded libraries?
  - Callbacks

Without source code

- Requires accurate disassembly
- Cannot accurately define all paths
- Shared libraries are easier to handle

```
static void (*fptr)(char *string, int len);

void set_callback(void *ptr)
{
        fptr = ptr;
}

void process_items()
{
        for (string *s : items) {
                fptr(s->c_str, s->len);
        }
}
```

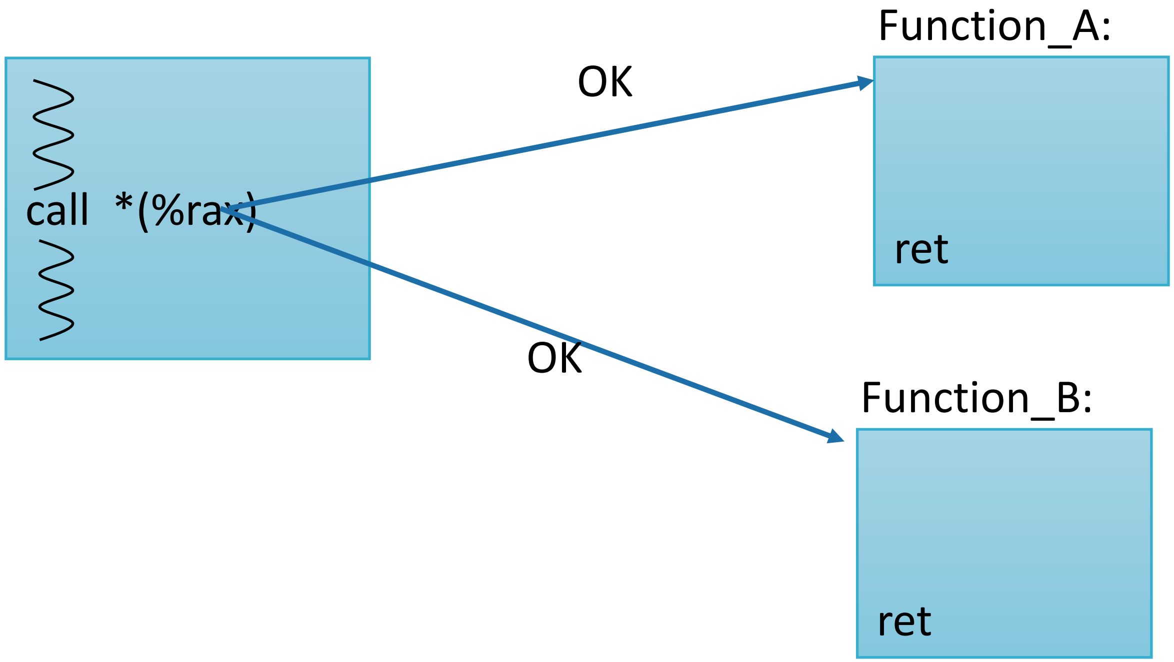
Working with an Imperfect CFG

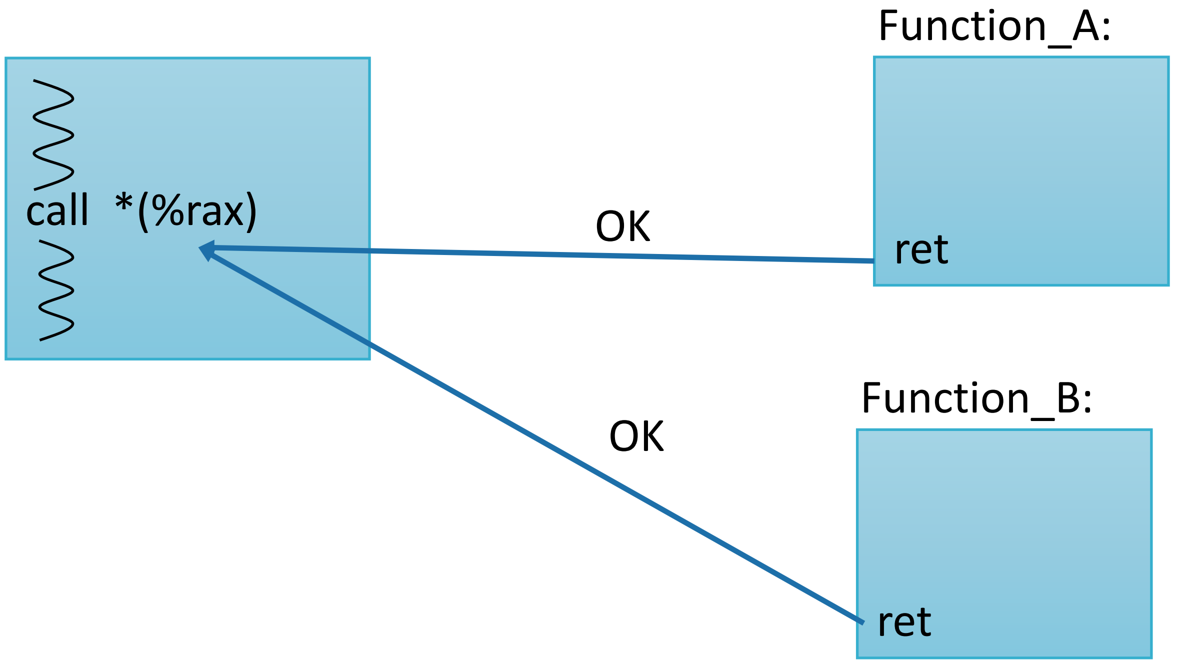
Lets assume that we know/can learn

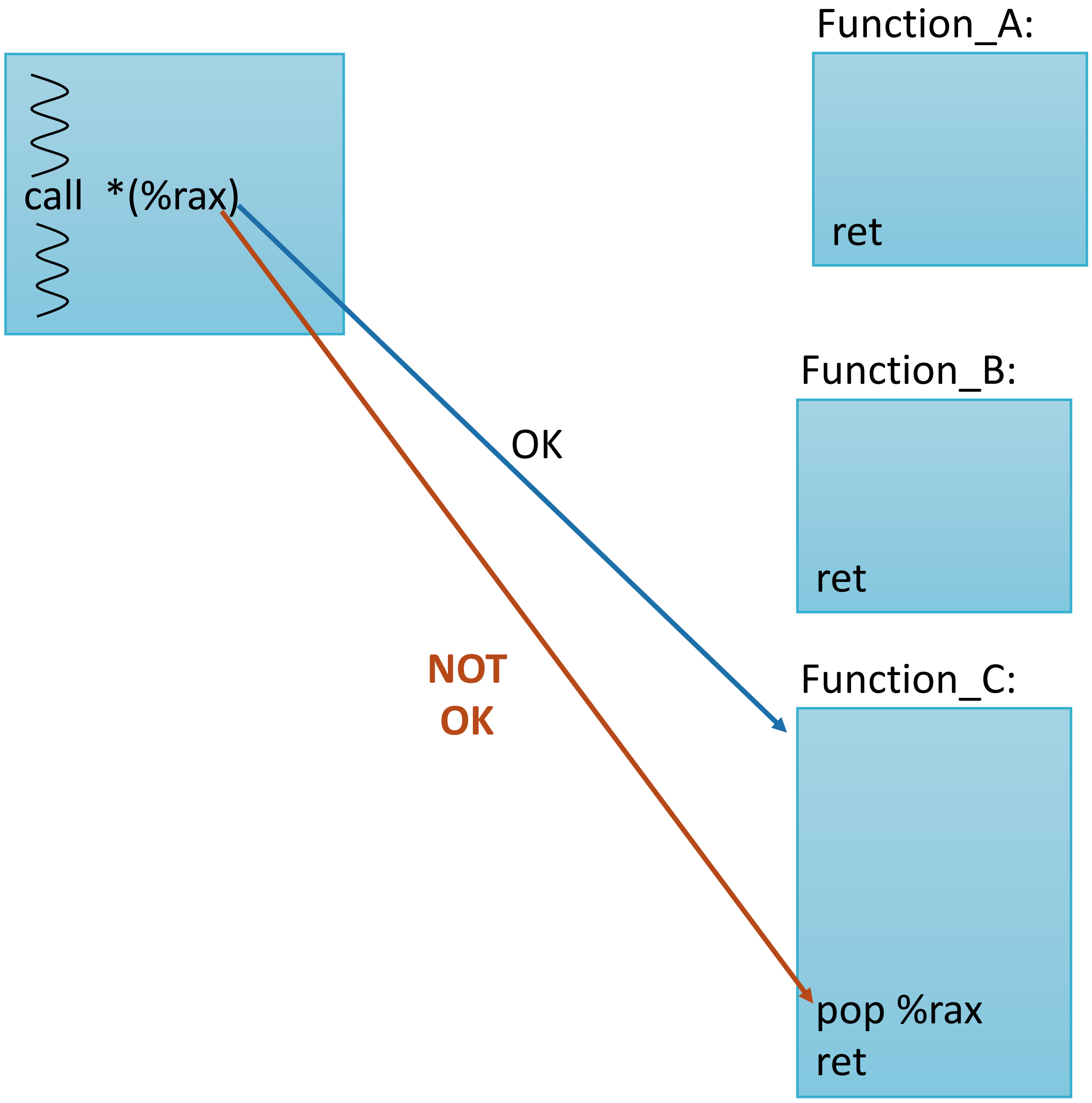
- The location of every function
- The location of every indirect branch instruction

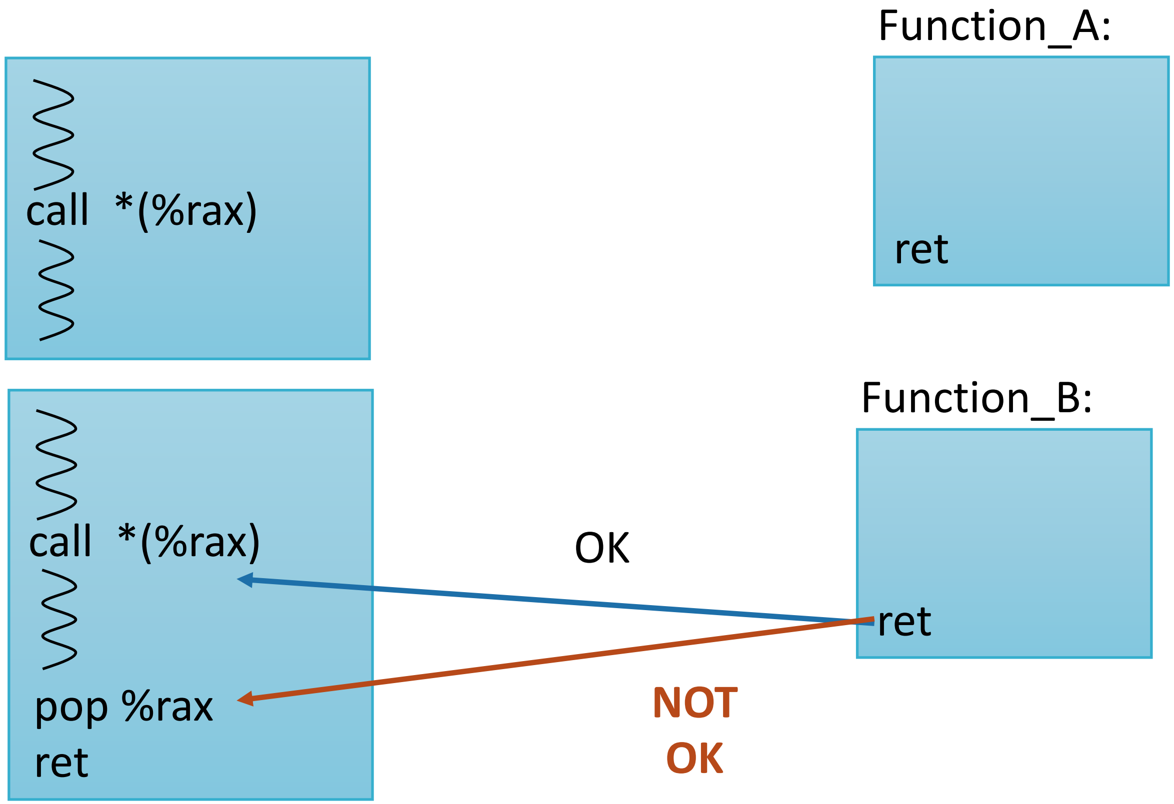
Coarse-grained CFI can enforce the following

- Indirect calls should only transfer control to functions
  - Same for most jumps
- Returns should only transfer control to instructions following a indirect call or jump


Function_A:
OK
call *(%rax)
ret
OK
Function_B:
ret


Function_A:
call *(%rax)
OK
ret
Function_B:
OK
ret


Function_A:
ret
Function_B:
ret
Function_C:
pop %rax
ret
call *(%rax)
OK
NOT
OK


call *(%rax)
Function_A:
ret
call *(%rax)
pop %rax
ret
OK
NOT OK
Function_B:
ret