CSE 127: Computer Security

Low-level mitigations

Deian Stefan

Some slides adopted from Nadia Heninger, Kirill Levchenko, Stefan Savage, and Stephen Checkoway

Today: mitigating buffer overflows

Lecture objectives:

- Understand how to mitigate buffer overflow attacks
- Understand the trade-offs of different mitigations
- Understand how mitigations can be bypassed

Buffer overflow mitigations

→ Avoid unsafe functions

- Stack canaries
- Separate control stack
- Memory writable or executable, not both (W^X)
- Address space layout randomization (ASLR)

Avoiding Unsafe Functions

- strcpy, strcat, gets, etc.
- **Plus:** Good idea in general
- **Minus:** Requires manual code rewrite
- **Minus:** Non-library functions may be vulnerable
  - E.g. user creates their own strcpy
- **Minus:** No guarantee you found everything
- **Minus:** alternatives are also error-prone

Even printf is tricky

If `buf` is under control of attacker is: `printf("%s\n", buf)` safe?

Even printf is tricky

If `buf` is under control of attacker

is: `printf(buf)` safe?

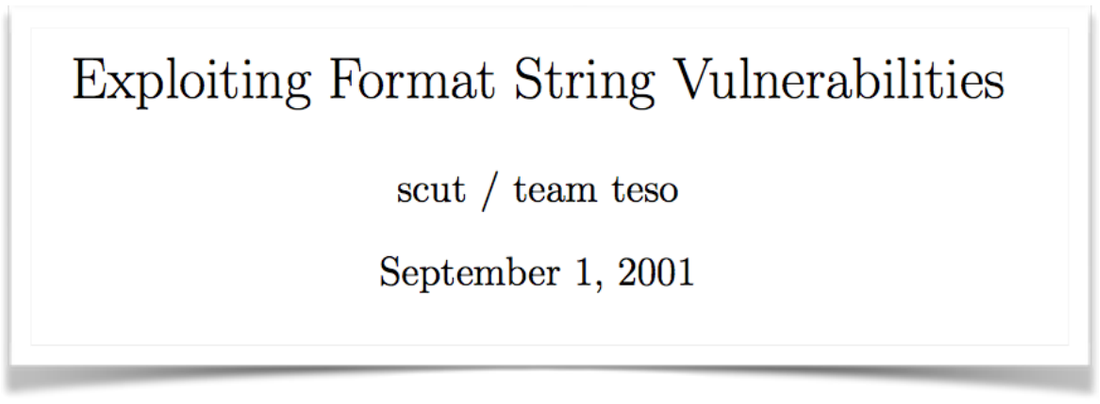
Even printf is tricky

Is `printf("%s\n")` safe?

Even printf is tricky

printf can be used to read and write memory

➠ control flow hijacking!



Exploiting Format String Vulnerabilities

scut / team teso

September 1, 2001

https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Buffer overflow mitigations

- Avoid unsafe functions
- → Stack canaries
- Separate control stack
- Memory writable or executable, not both (W^X)
- Address space layout randomization (ASLR)

Miner's canary [edit]

Canaries were used as sentinel species for use in detecting carbon monoxide in coal mining from around 1913 when the idea was suggested by John Scott Haldane.[14] Toxic gases such as carbon monoxide or asphyxiant gases such as methane[15] in the mine would affect the bird before affecting the miners. Signs of distress from the bird indicated to the miners that conditions were unsafe. The birds were generally kept in carriers which had small oxygen bottles attached to revive the birds, so that they could be used multiple times within the mine.[16] The use of miners' canaries in British mines was phased out in 1986.[17][18]

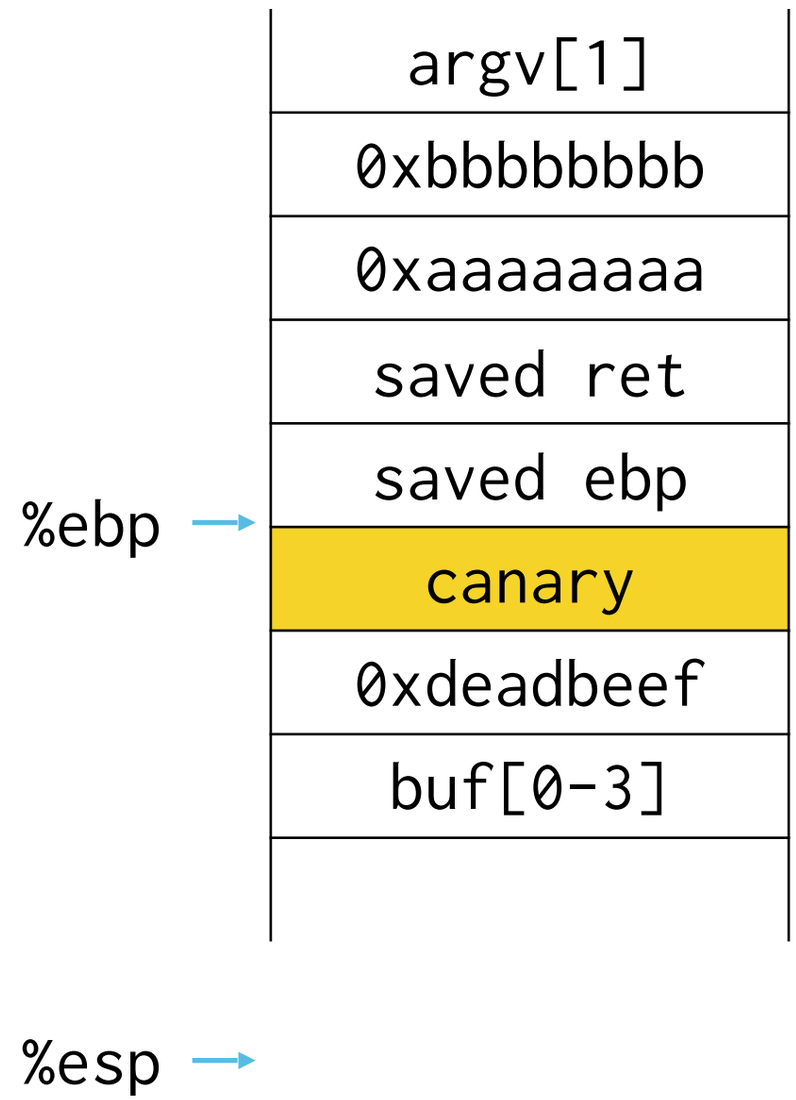
The phrase "canary in a coal mine" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis. By analogy, the term "climate canary" is used to refer to a species (called an indicator species) that is affected by an environmental danger prior to other species, thus serving as an early warning system for the other species with regard to the danger.[19]

Stack canaries

- **Goal:** Prevent control flow hijacking by detecting stack-buffer overflows
- **Idea:**
  - Place canary between local variables and saved frame pointer (and return address)
  - Check canary before jumping to return address
- **Approach:**
  - Modify function prologues and epilogues

Example (at a high level)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```



Compiled, without canaries

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

With -fstack-protector-strong

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

void func(int a, int b, char *str) {
  int c = 0xdeadbeef;
  char buf[4];
  strcpy(buf,str);
}

int main(int argc, char**argv) {
  func(0xaaaaaaaa,0xbbbbbbbb,argv[1]);
  return 0;
}
```

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

With -fstack-protector-strong

write canary from `%gs:20` to stack `-12(%ebp)`

compare canary in `%gs:20` to that on stack `-12(%ebp)`

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

Trade-offs

- **Easy to deploy:** Can implement mitigation as compiler pass (i.e., don't need to change your code)
- **Performance:** Every protected function is more expensive

No stack protection

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

-fstack-protector-strong

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

When do we add canaries?

When do we add canaries?

- -fstack-protector
  - Functions with character buffers $\geq$ ssp-buffer-size (default is 8)
  - Functions with variable sized `alloca()`s

When do we add canaries?

- -fstack-protector
  - Functions with character buffers ≥ ssp-buffer-size (default is 8)
  - Functions with variable sized `alloca()`s
- -fstack-protector-strong
  - Functions with local arrays of any size/type
  - Functions that have references to local stack variables

When do we add canaries?

- -fstack-protector
  - Functions with character buffers ≥ ssp-buffer-size (default is 8)
  - Functions with variable sized `alloca()`s
- -fstack-protector-strong
  - Functions with local arrays of any size/type
  - Functions that have references to local stack variables
- -fstack-protector-all:
  - All functions!

There is a cost even for same func:

No stack protection

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

-fstack-protector-strong

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

-fstack-protector-all

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

(we'll see why in just a bit)

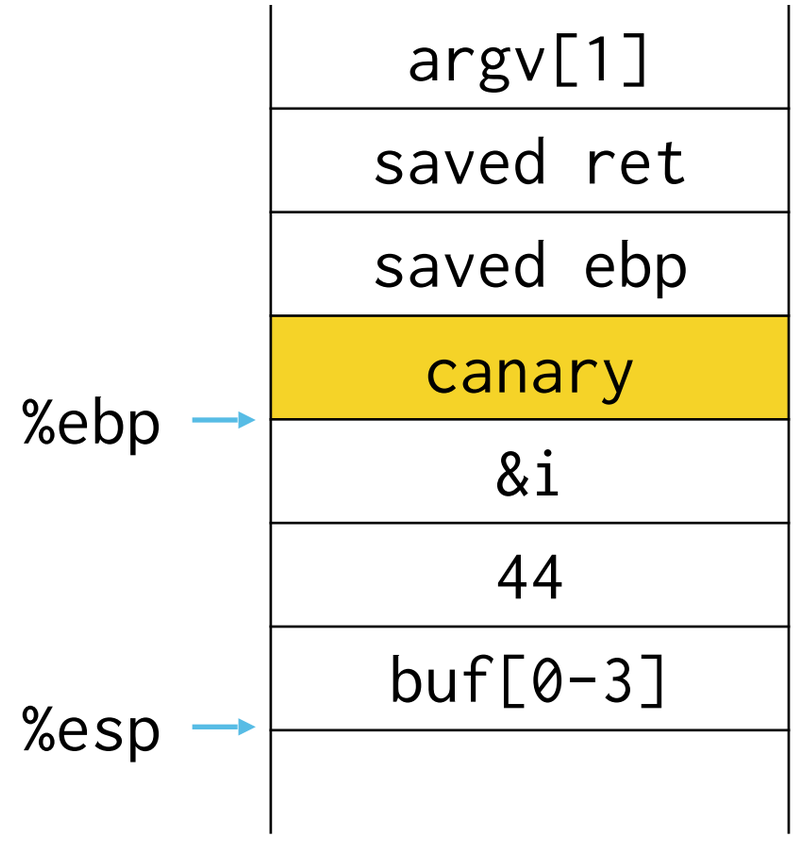
How can we defeat canaries?

How can we defeat canaries?

- Assumption: impossible to subvert control flow without corrupting the canary
- Attack vectors
  - Use targeted write gadget (e.g., with format strings)
  - Pointer subterfuge
  - Overwrite function pointer elsewhere on the stack/heap
  - memcpy buffer overflow with fixed canary
  - Learn the canary

Pointer subterfuge

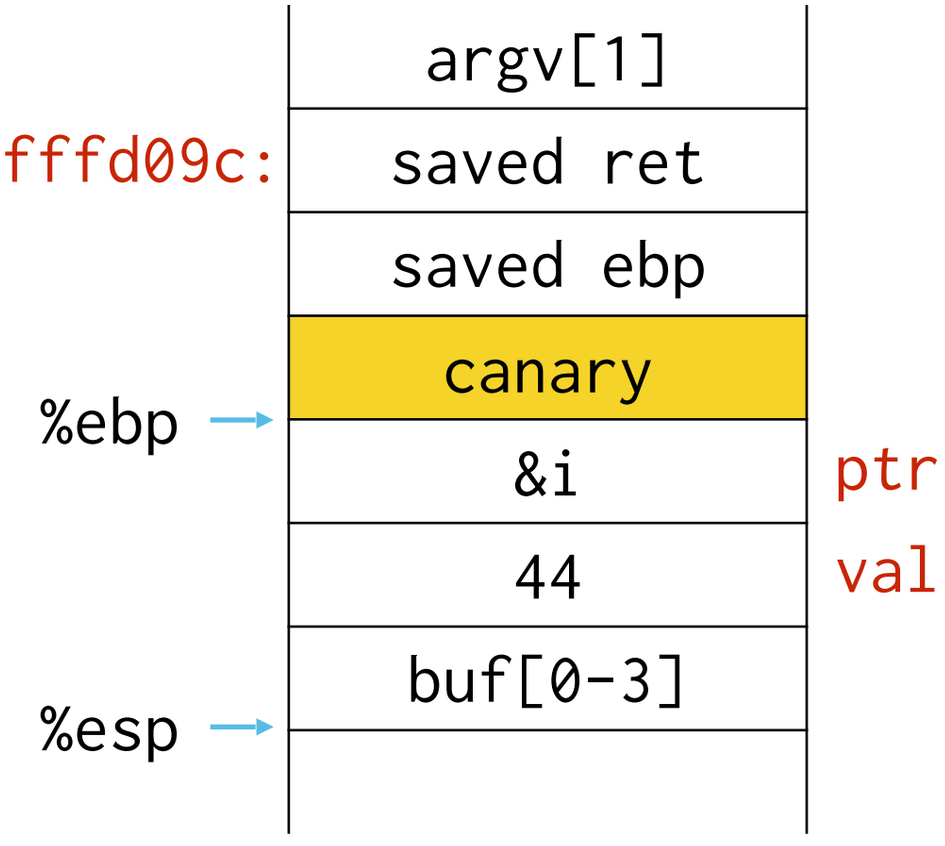
```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

int i = 42;

void func(char *str) {
  int *ptr = &i;
  int val = 44;
  char buf[4];
  strcpy(buf,str);
  *ptr = val;
}

int main(int argc, char**argv) {
  func(argv[1]);
  return 0;
}
```



Pointer subterfuge

```
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

int i = 42;

void func(char *str) {
  int *ptr = &i;
  int val = 44;
  char buf[4];
  strcpy(buf,str);
  *ptr = val;
}

int main(int argc, char**argv) {
  func(argv[1]);
  return 0;
}
```

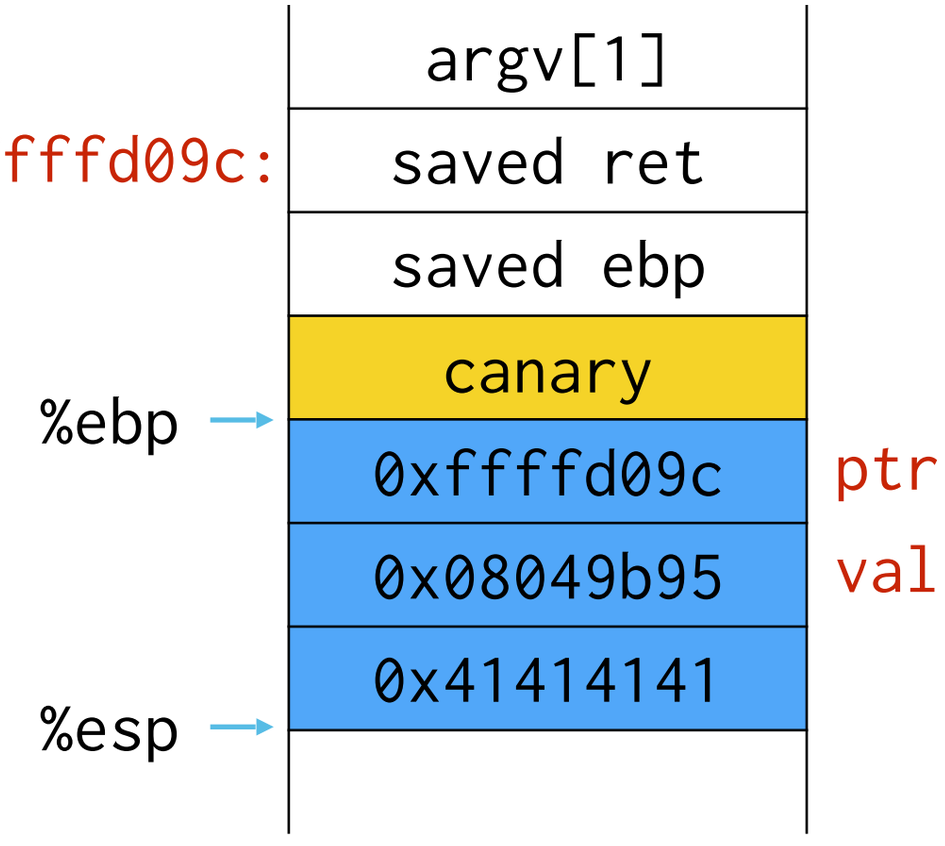
0x08049b95:



Pointer subterfuge

```
0x08049b95:
```

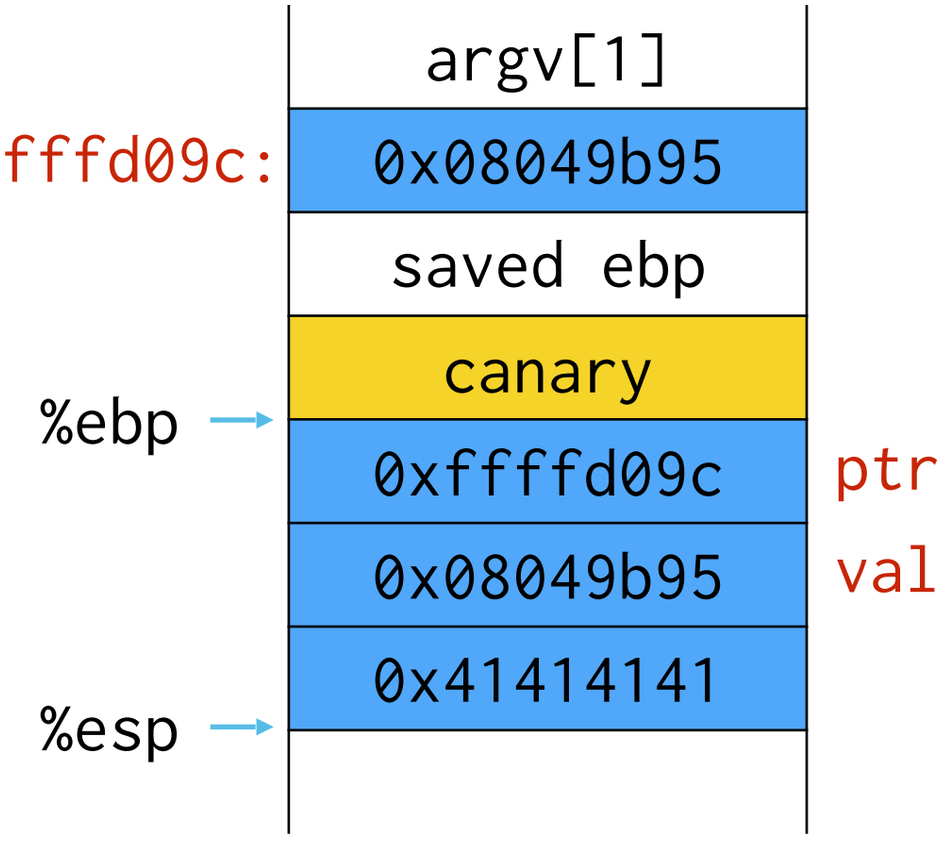
```c
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

int i = 42;

void func(char *str) {
  int *ptr = &i;
  int val = 44;
  char buf[4];
  strcpy(buf,str);
  *ptr = val;
}

int main(int argc, char**argv) {
  func(argv[1]);
  return 0;
}
```



Pointer subterfuge

```
0x08049b95:
```

```c
#include <stdio.h>
#include <string.h>

void foo() {
  printf("hello all!!\n");
  exit(0);
}

int i = 42;

void func(char *str) {
  int *ptr = &i;
  int val = 44;
  char buf[4];
  strcpy(buf,str);
  *ptr = val;
}

int main(int argc, char**argv) {
  func(argv[1]);
  return 0;
}
```



example3.c

Overwrite function pointer on stack

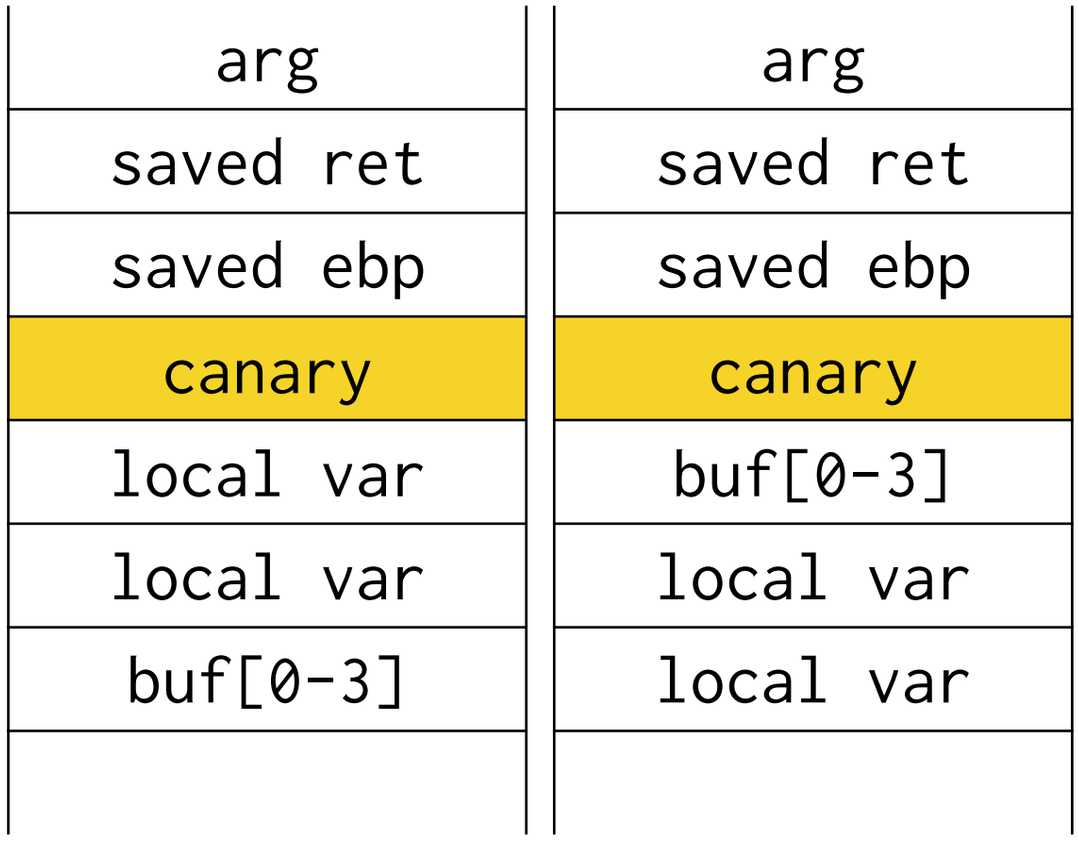
- Similar to previous example, but overwrite function pointer on stack
  - Tricky: compiler can load it into register before `strcpy()`

```
void func(char *str) {
  void (*fptr)() = &bar;
  char buf[4];
  strcpy(buf,str);
  fptr()
}
```

example4.c

Can we do anything about this?

- **Problem:** overflowing local variables can allow attacker to hijack control flow
- **Solution:** some implementations reorder local variables, place buffers closer to canaries vs. lexical order

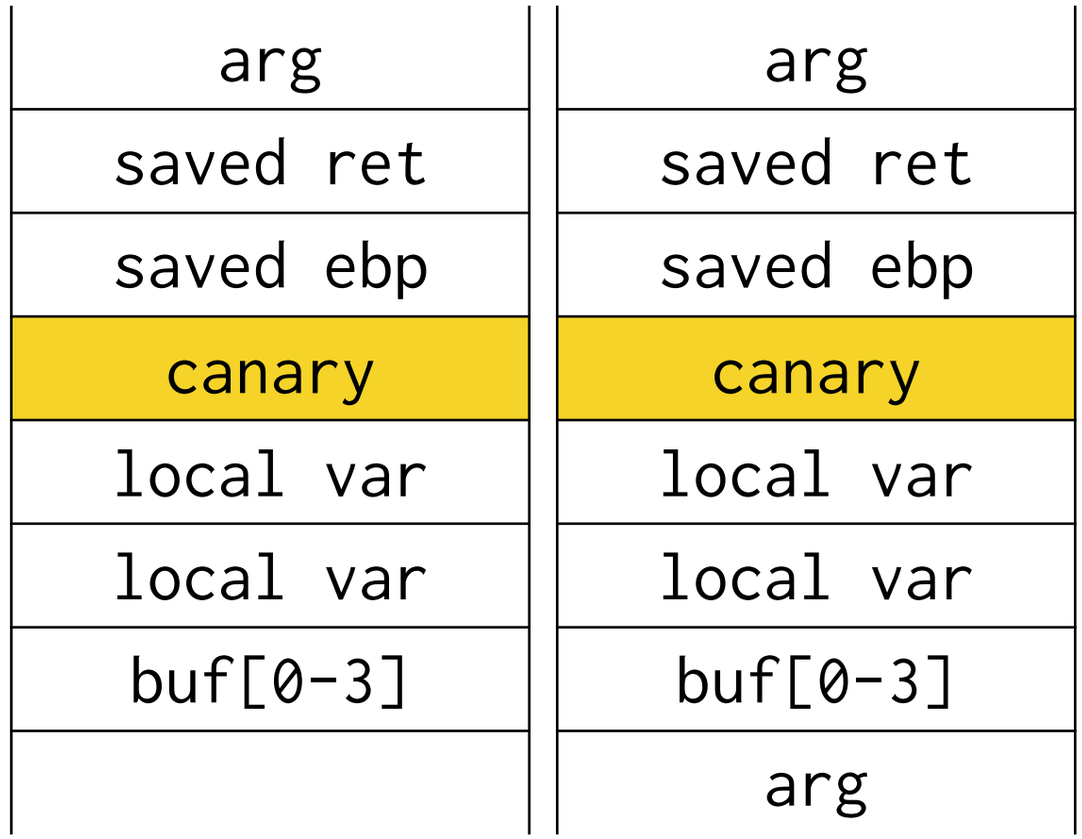


What about function arguments?

What about function arguments?

- Same problem!

```
void func(char *str, void (*fptr)()) {
  char buf[4];
  strcpy(buf,str);
  fptr()
}
```



- **Solution:** also copy args to the top of the stack to make overwriting them via local variables less likely

That’s what we were seeing before

No stack protection

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $-559038737, -12(%ebp)
        subl    $8, %esp
        pushl   16(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        leave
        ret
```

-fstack-protector-strong

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    16(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -28(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

-fstack-protector-all

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

-fstack-protector-strong

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

-fstack-protector-strong

copy arg1

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

-fstack-protector-strong

copy arg1

copy arg2

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

-fstack-protector-strong

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

copy arg1

copy arg2

copy arg3

-fstack-protector-strong

copy arg1

copy arg2

copy arg3

write canary

```
func(int, int, char*):
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        movl    %eax, -28(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -32(%ebp)
        movl    16(%ebp), %eax
        movl    %eax, -36(%ebp)
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $-559038737, -20(%ebp)
        subl    $8, %esp
        pushl   -36(%ebp)
        leal    -16(%ebp), %eax
        pushl   %eax
        call    strcpy
        addl    $16, %esp
        nop
        movl    -12(%ebp), %eax
        xorl    %gs:20, %eax
        je      .L4
        call    __stack_chk_fail
.L4:
        leave
        ret
```

How can we defeat canaries?

- Assumption: impossible to subvert control flow without corrupting the canary
- Ideas?
  - Use targeted write (e.g., with format strings)
  - Pointer subterfuge
  - Overwrite function pointer elsewhere on the stack/heap
  - → memcpy buffer overflow with fixed canary
  - Learn the canary

memcpy with fixed canary

- Canary values like `0x000d0aff` (0, CR, NL, -1) are designed to terminate string ops like `strcpy` and `gets`
- Even random canaries have null bytes
- How do we defeat this?
  - Find memcpy/memmove/read vulnerability

How can we defeat canaries?

- Assumption: impossible to subvert control flow without corrupting the canary
- Ideas?
  - Use targeted write (e.g., with format strings)
  - Pointer subterfuge
  - Overwrite function pointer elsewhere on the stack/heap
  - memcpy buffer overflow with fixed canary
  - → Learn the canary

Learn the canary

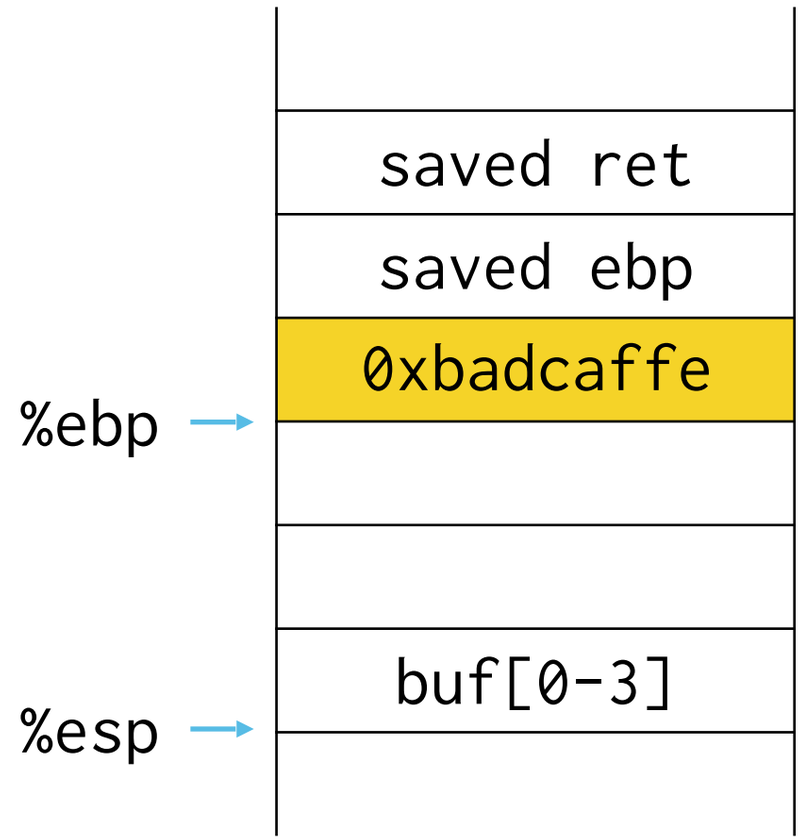
- Approach 1: chained vulnerabilities
  - Exploit one vulnerability to read the value of the canary
  - Exploit a second to perform stack buffer overflow
- Modern exploits chain multiple vulnerabilities
  - Recent Chinese gov iPhone exploit: 14 vulns!

Learn the canary

- Approach 2: brute force servers (e.g., Apache2)
  - Main server process:
    - Establish listening socket
    - Fork several workers: if any die, fork new one!
  - Worker process:
    - Accept connection on listening socket & process request

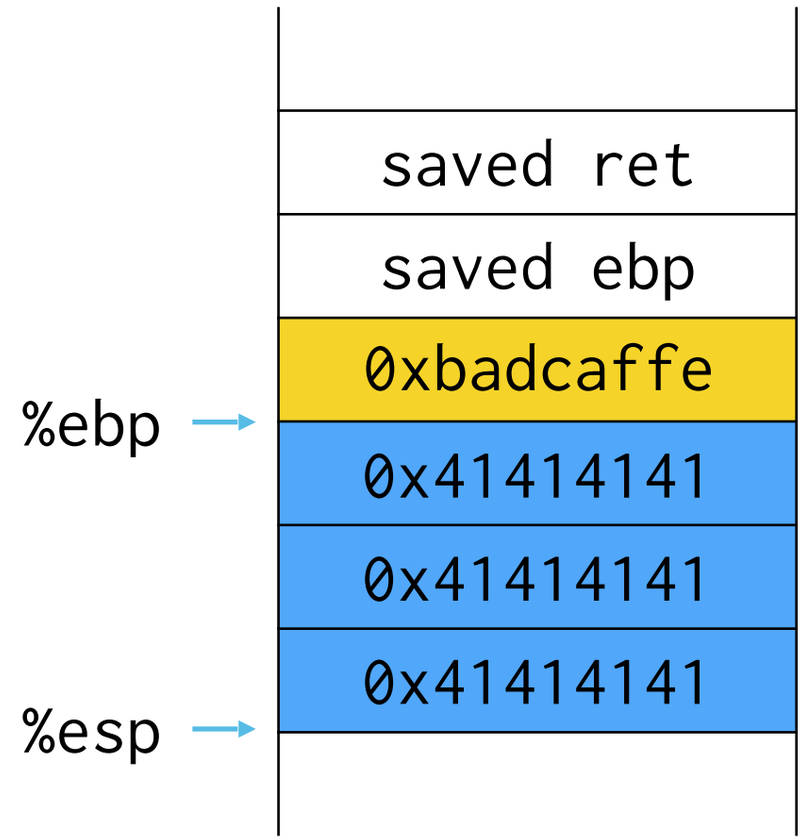
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



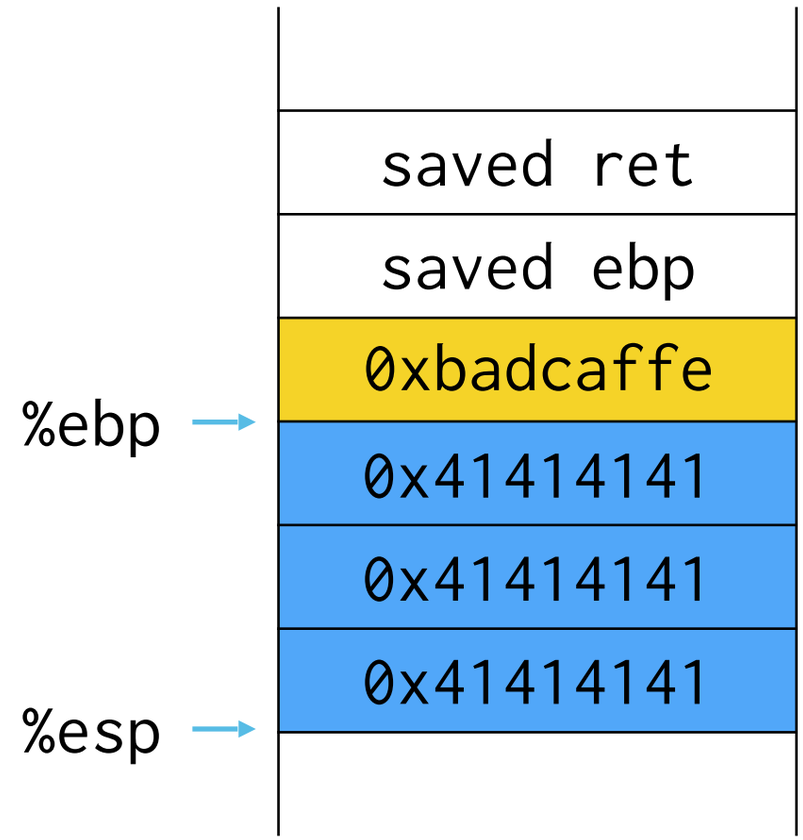
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



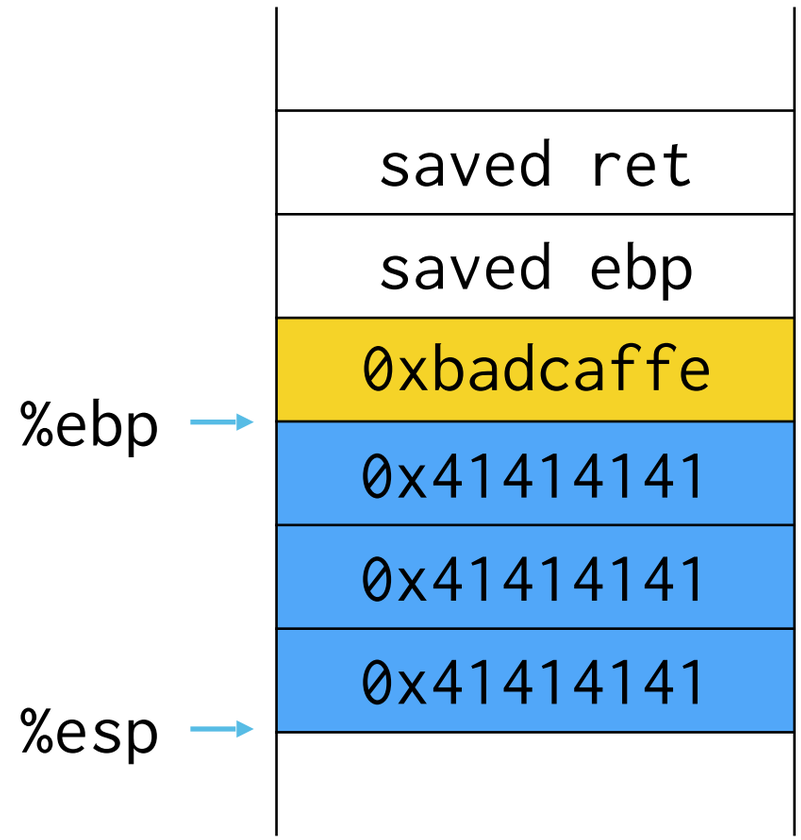
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



Perfect for brute forcing

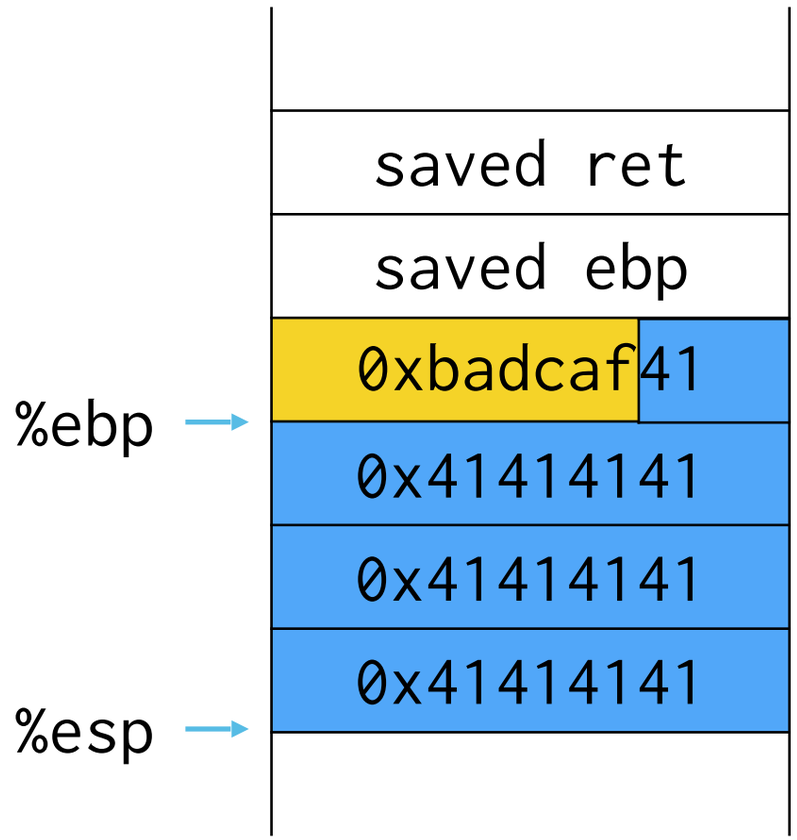
- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



we know size of buffer!

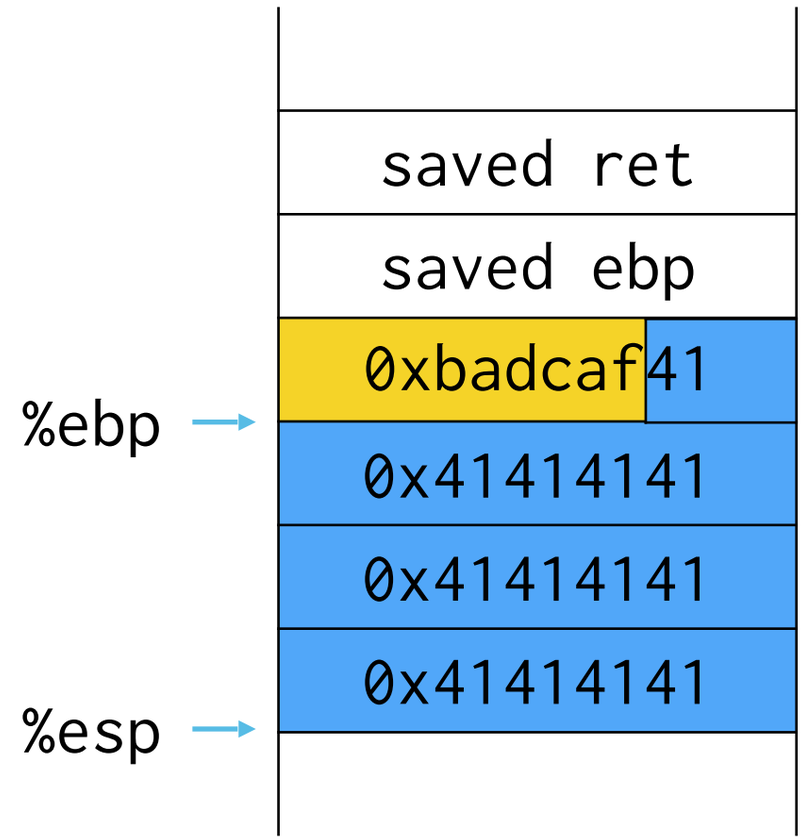
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



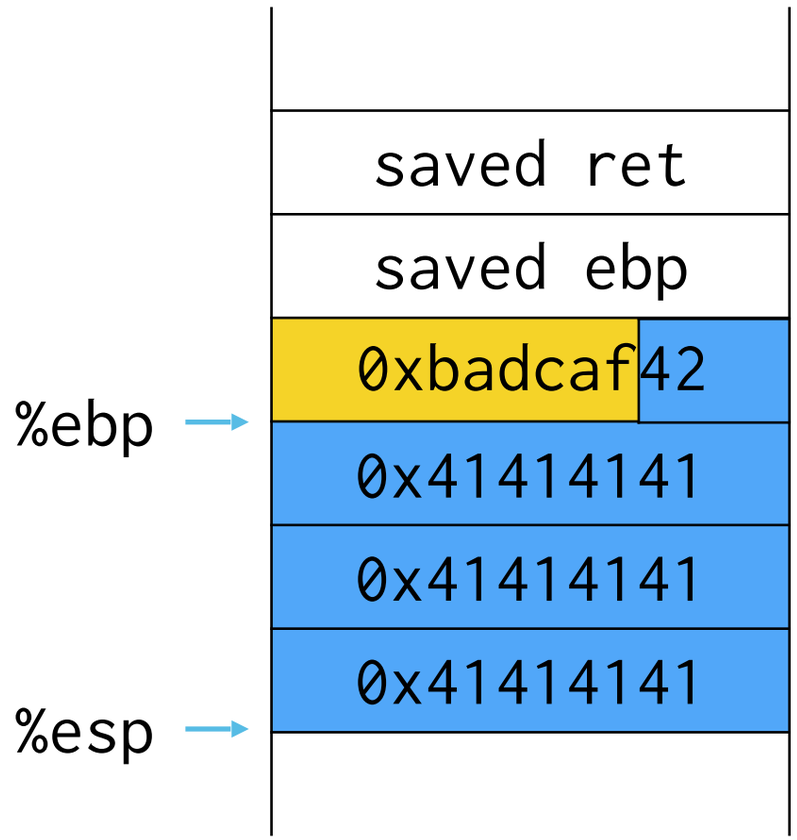
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



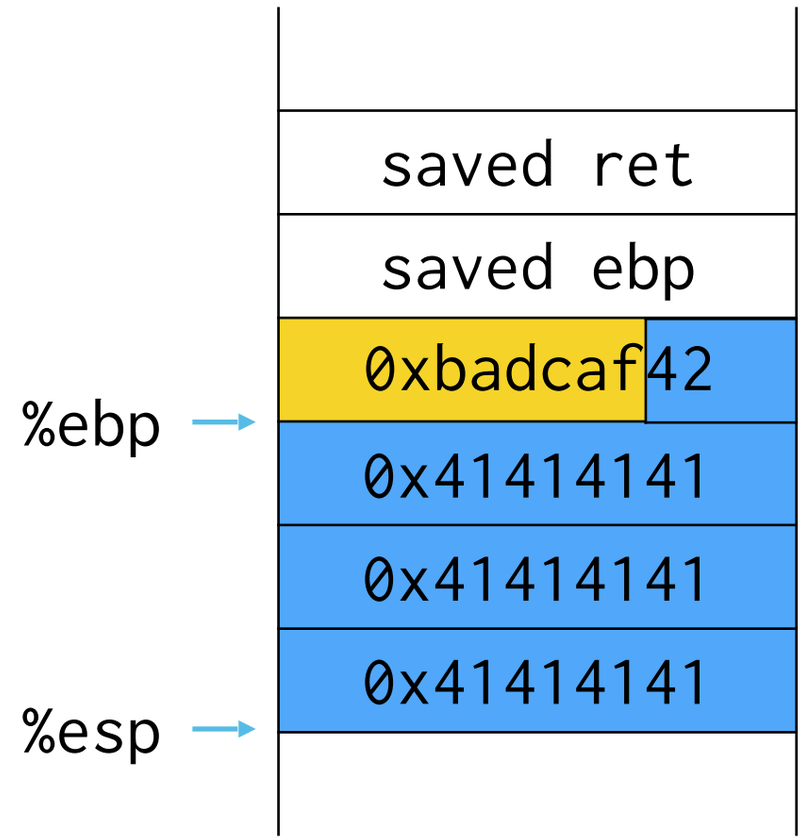
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



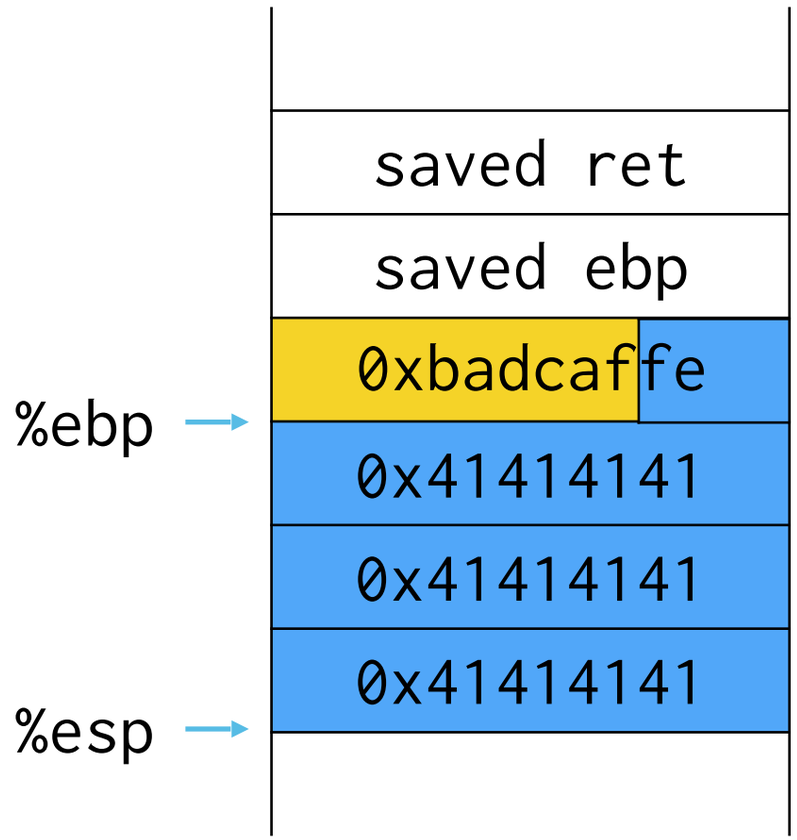
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



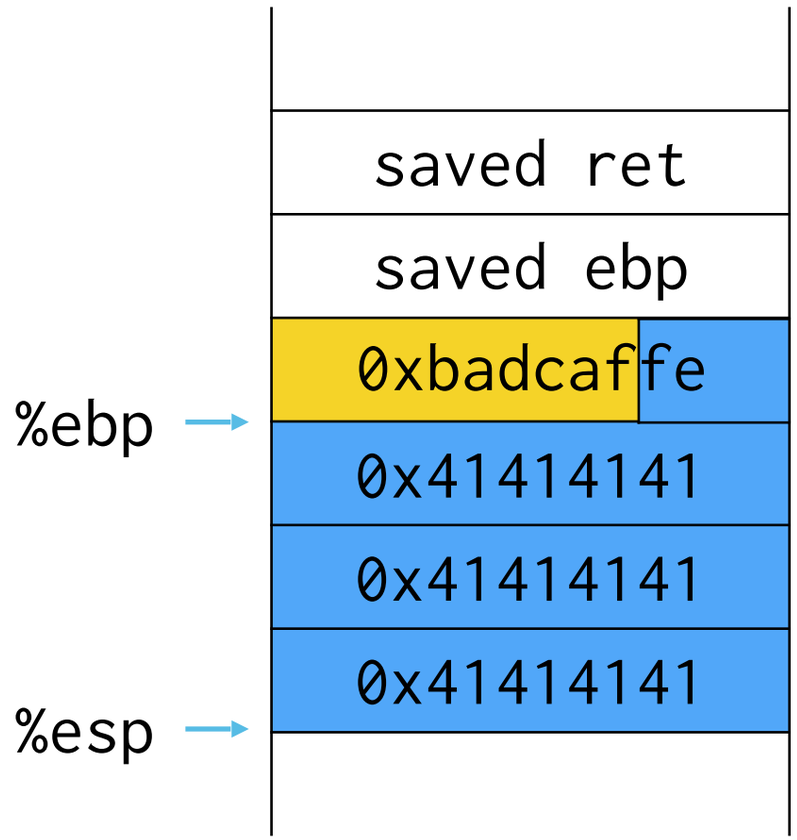
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



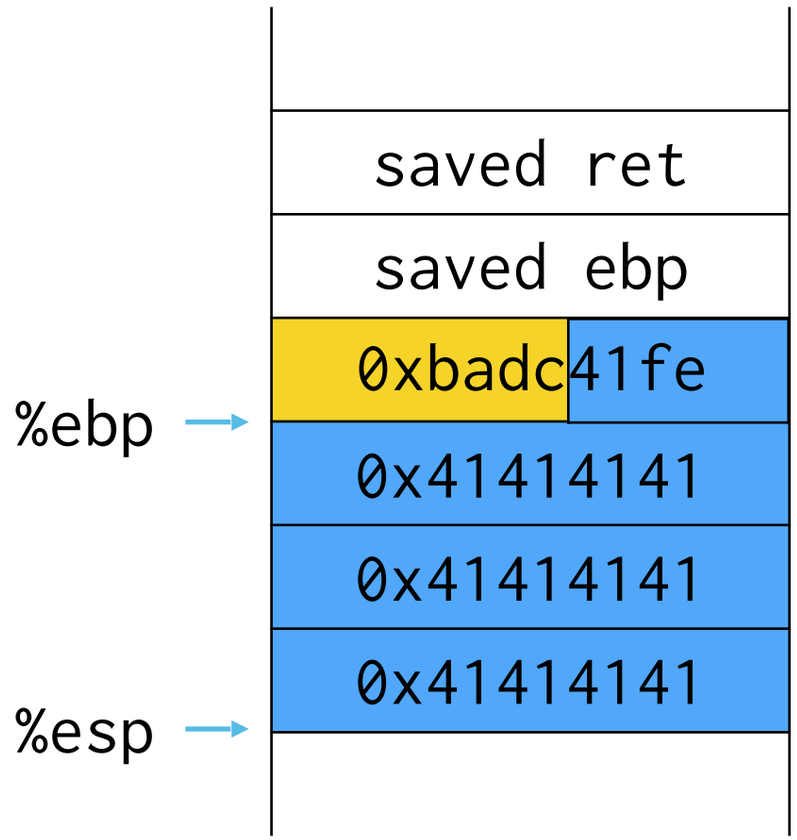
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



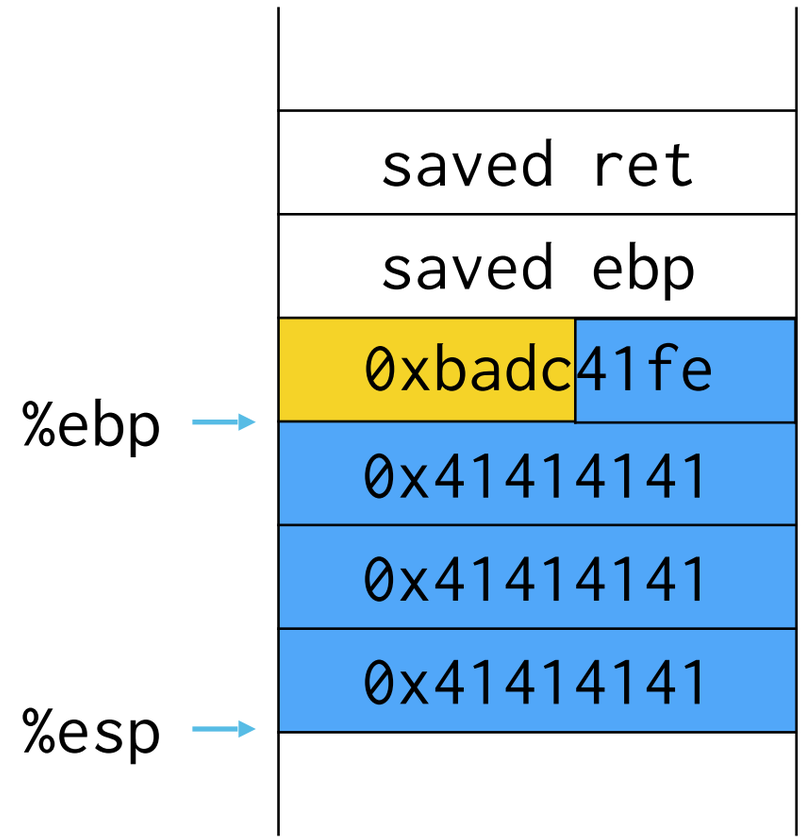
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



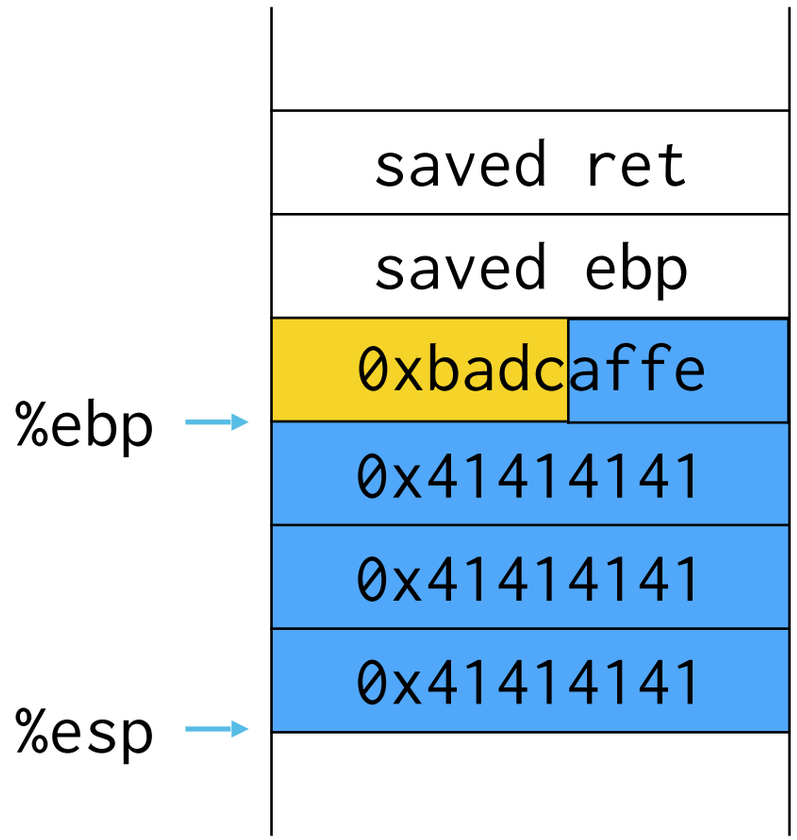
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



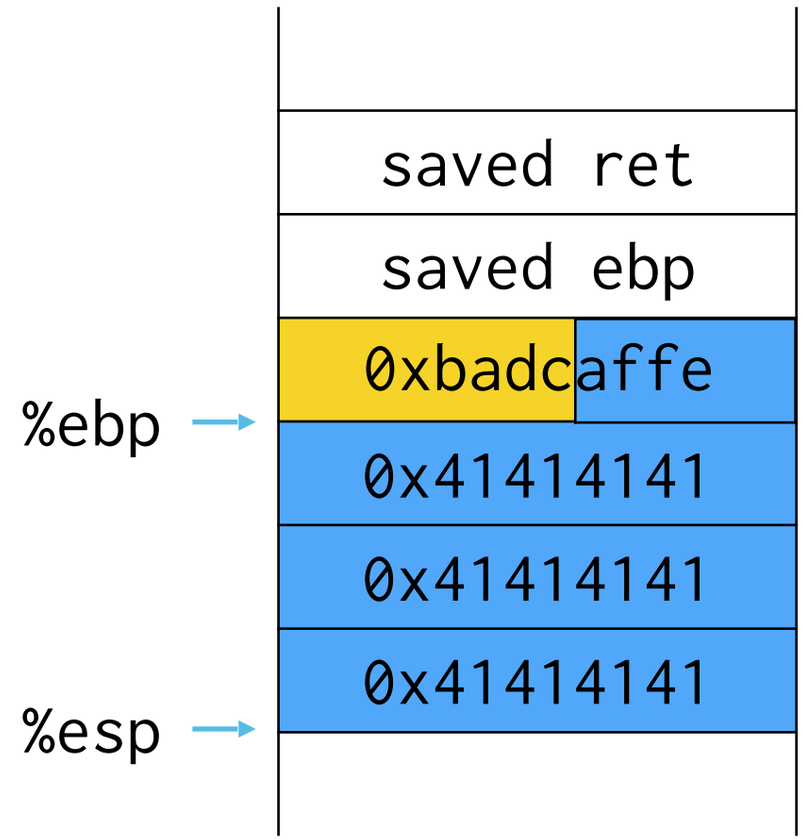
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



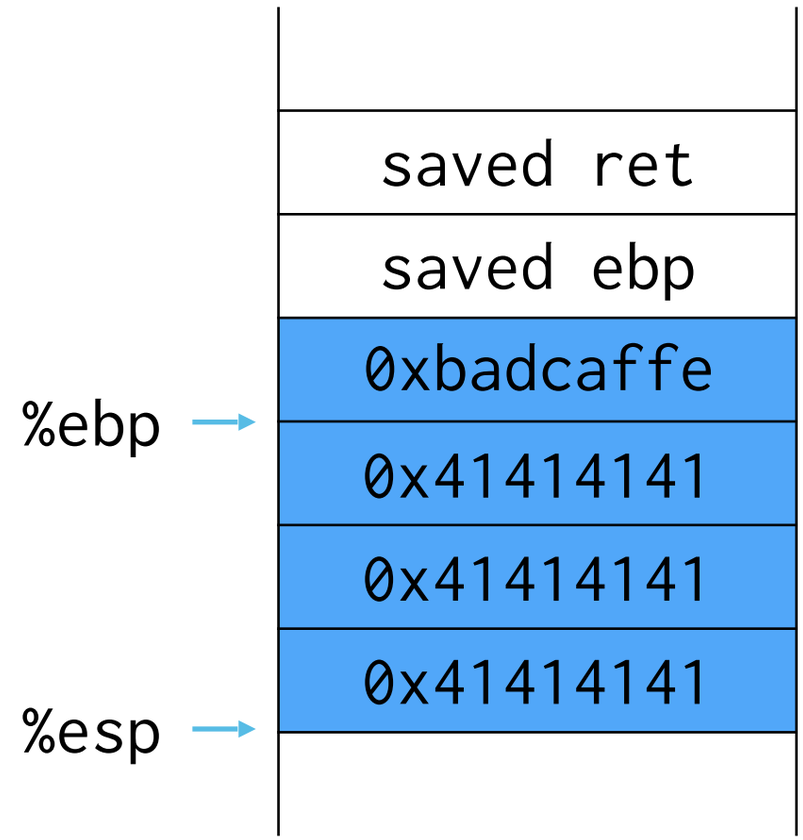
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



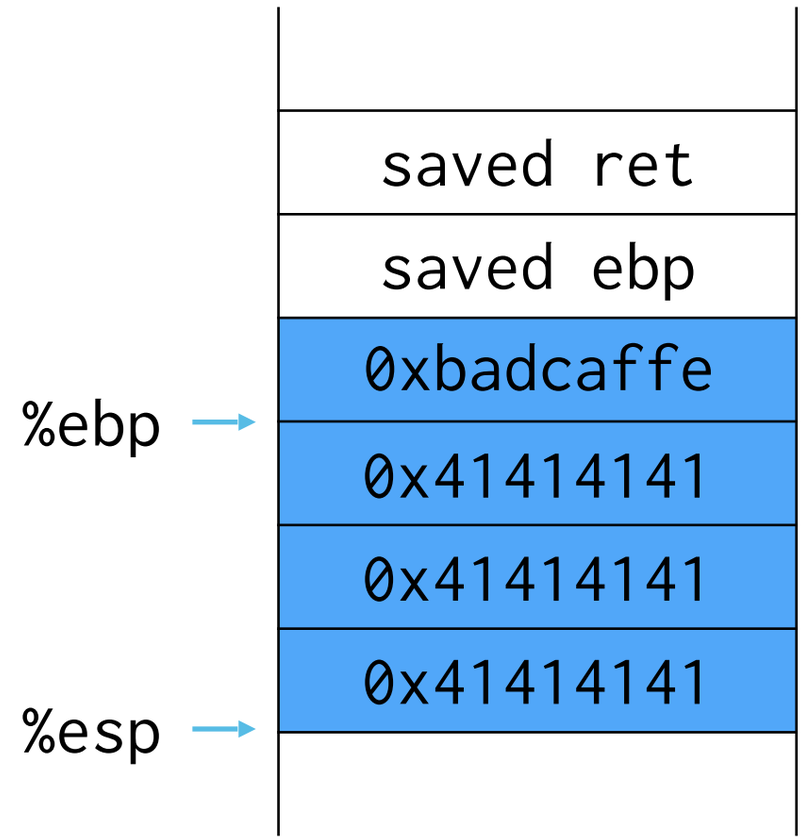
Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



Perfect for brute forcing

- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



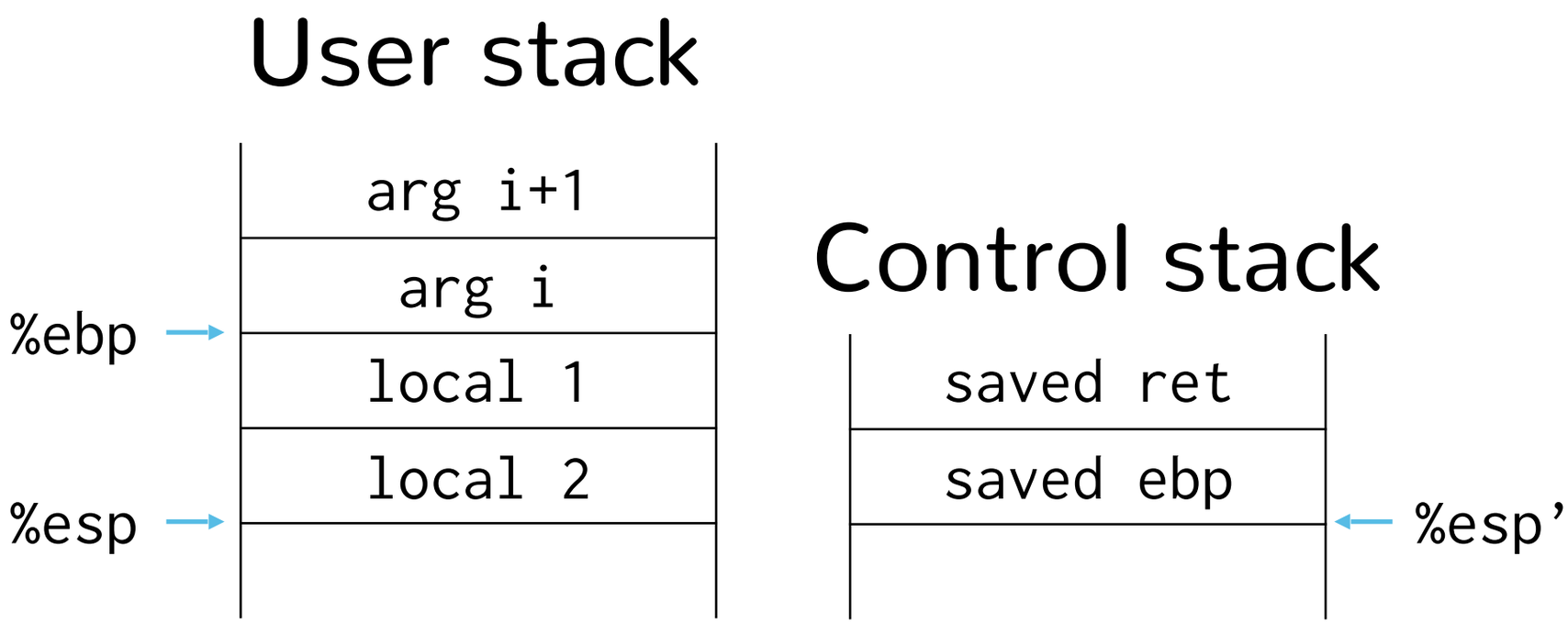
Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
- → Separate control stack
  - Memory writable or executable, not both (W^X)
  - Address space layout randomization (ASLR)

Separate control stack

Problem: The stack smashing attacks take advantage of the weird machine: control data is stored next to user data

Solution: Make it less weird by bridging the implementation and abstraction gap: separate the control stack



Separate control stack

Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can't read or manipulate control stack

Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can't read or manipulate control stack
  - How can we defeat this?

Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can't read or manipulate control stack
  - How can we defeat this?
- By construction: can't express stack smashing in Wasm

Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can't read or manipulate control stack
  - How can we defeat this?
- By construction: can't express stack smashing in Wasm
  - Challenge: we need to compile C/C++ to Wasm
  - How do we compile buffers, &var, and function ptrs?

Separate control stack

- WebAssembly (Wasm) has a separate stack
  - At the Wasm layer: can't read or manipulate control stack
  - How can we defeat this?
- By construction: can't express stack smashing in Wasm
  - Challenge: we need to compile C/C++ to Wasm
  - How do we compile buffers, &var, and function ptrs?
    - Put them on user stack!
    - So? C programs compiled to Wasm: overwrite function pointers!

Separate control stack

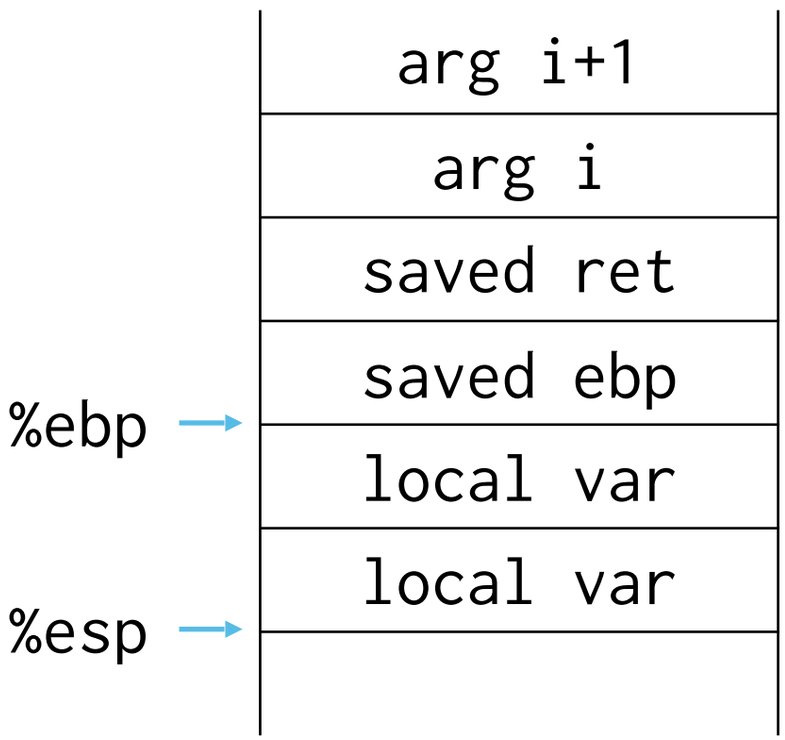
Wasm is not special.

Other byte codes and languages are similar: compiling C to X will inevitably preserve some of C's bugs.

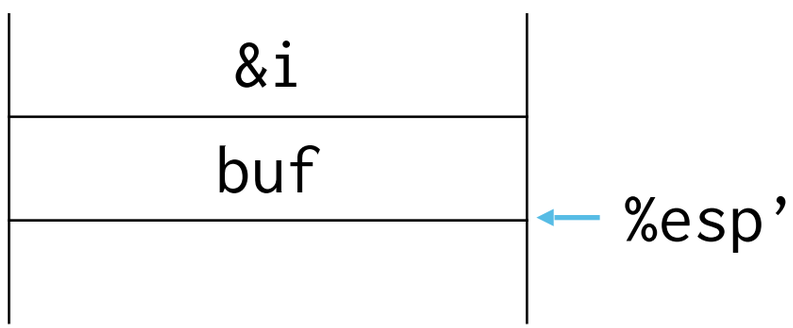
Safe stack

“SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores **return addresses**, **register spills**, and **local variables that are always accessed in a safe way**, while the unsafe stack stores everything else. This separation ensures that buffer overflows on the unsafe stack cannot be used to overwrite anything on the safe stack.”

Safe stack



Unsafe stack

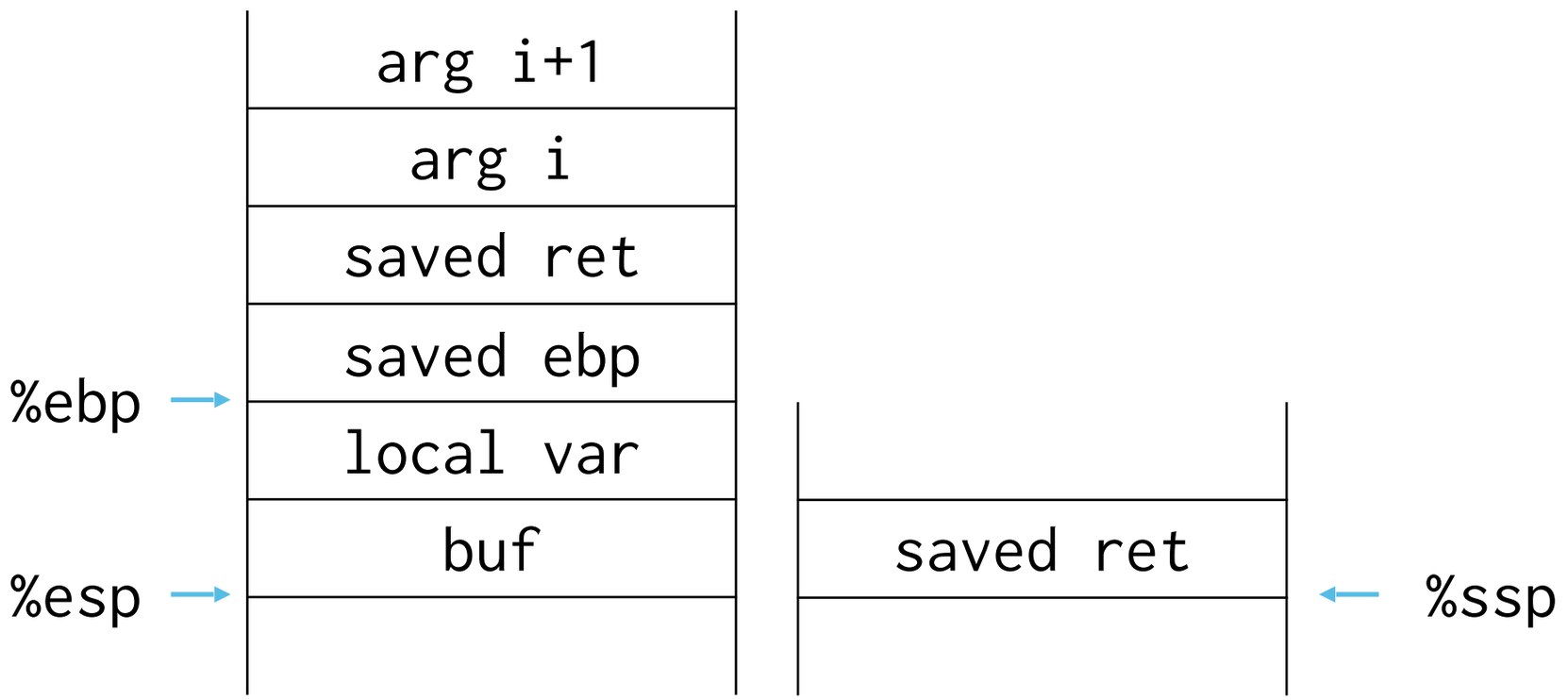


How do we implement these?

- There is no actual separate stack, we only have linear memory and loads/store instructions
- Put the safe/separate stack in a random place in the address space
  - Assumption: location of control/stack stack is secret
  - How do we defeat this?

Intel's shadow stack

- Addresses both the performance and security issues
  - New shadow stack pointer (%ssp)
    - call and ret automatically update %esp and %ssp
    - Can't update shadow stack manually
    - May need to rewrite code that manipulates stack manually



How do we defeat this?

Find a function pointer and overwrite it to point to shellcode!

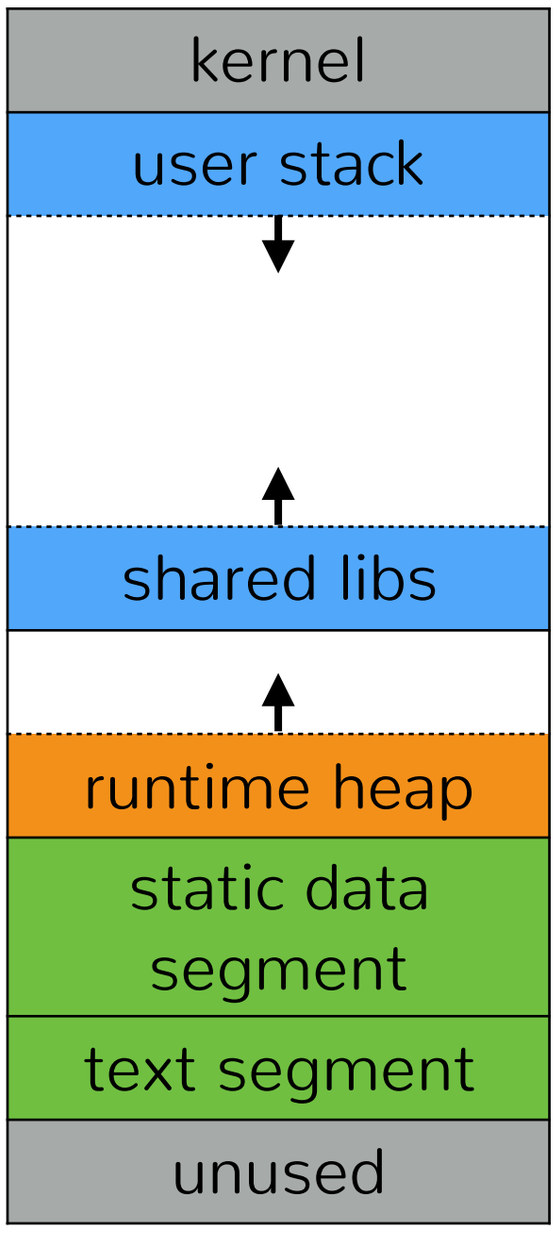
Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
- Separate control stack
- Memory writable or executable, not both (W^X)
- Address space layout randomization (ASLR)

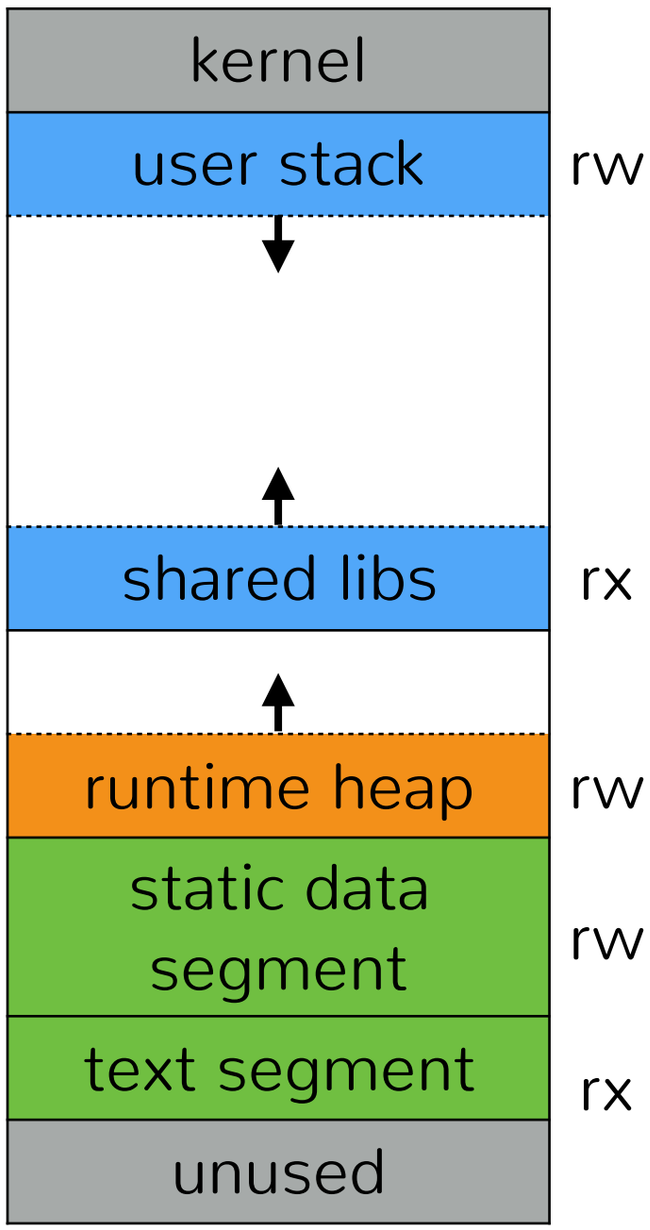
W^X: write XOR execute

- **Goal:** prevent execution of shell code from the stack
- **Insight:** use memory page permission bits
  - Use MMU to ensure memory cannot be both writeable and executable at same time
- Many names for same idea:
  - XN: eXecute Never
  - W^X: Write XOR eXecute
  - DEP: Data Execution Prevention

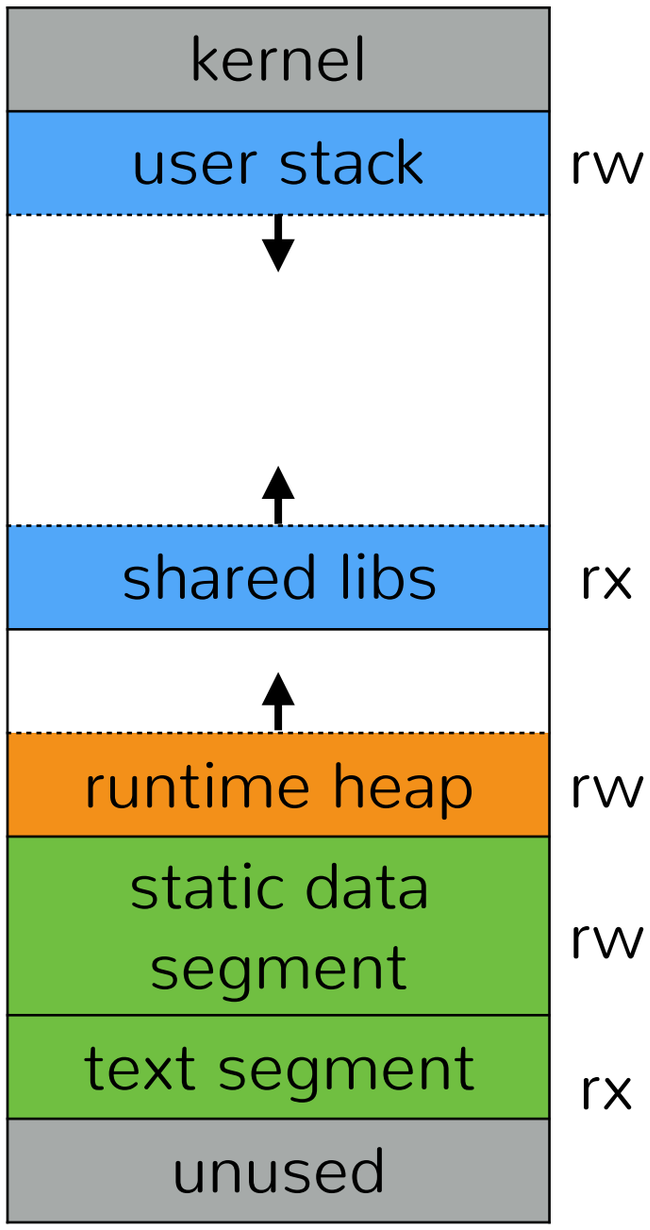
Recall our memory layout

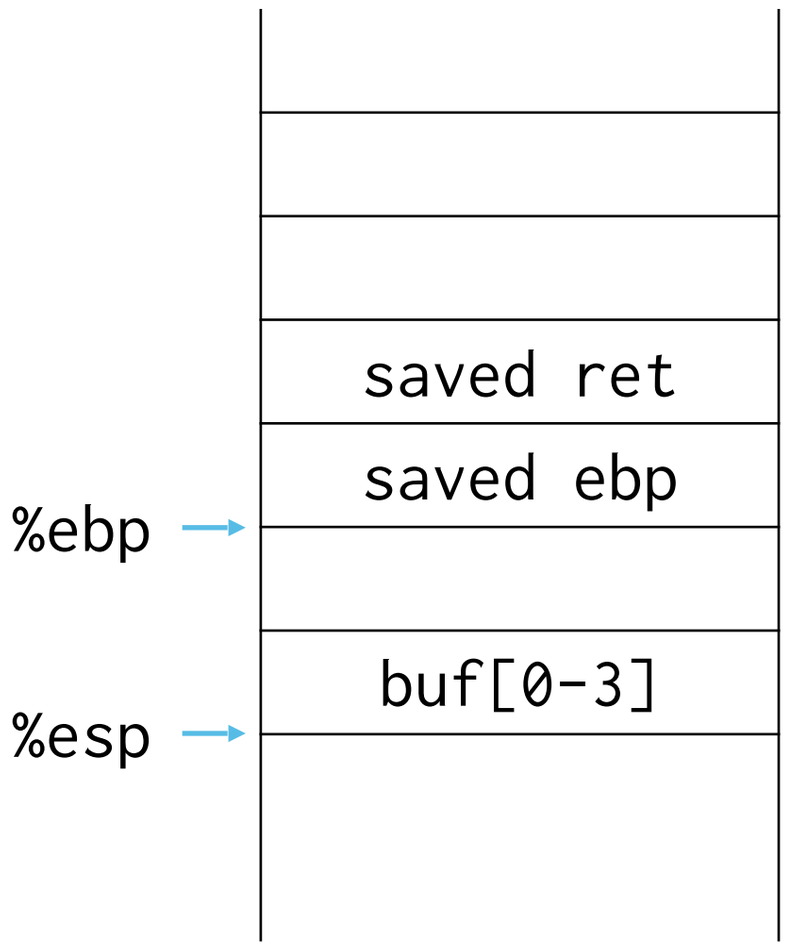


Recall our memory layout

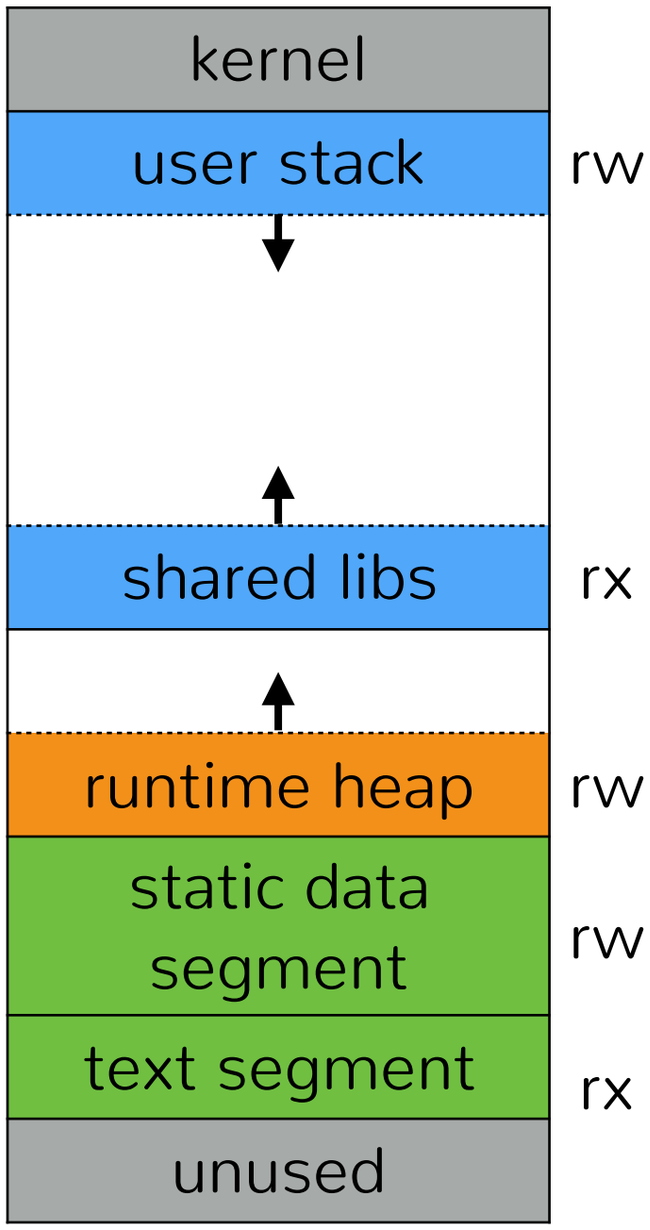


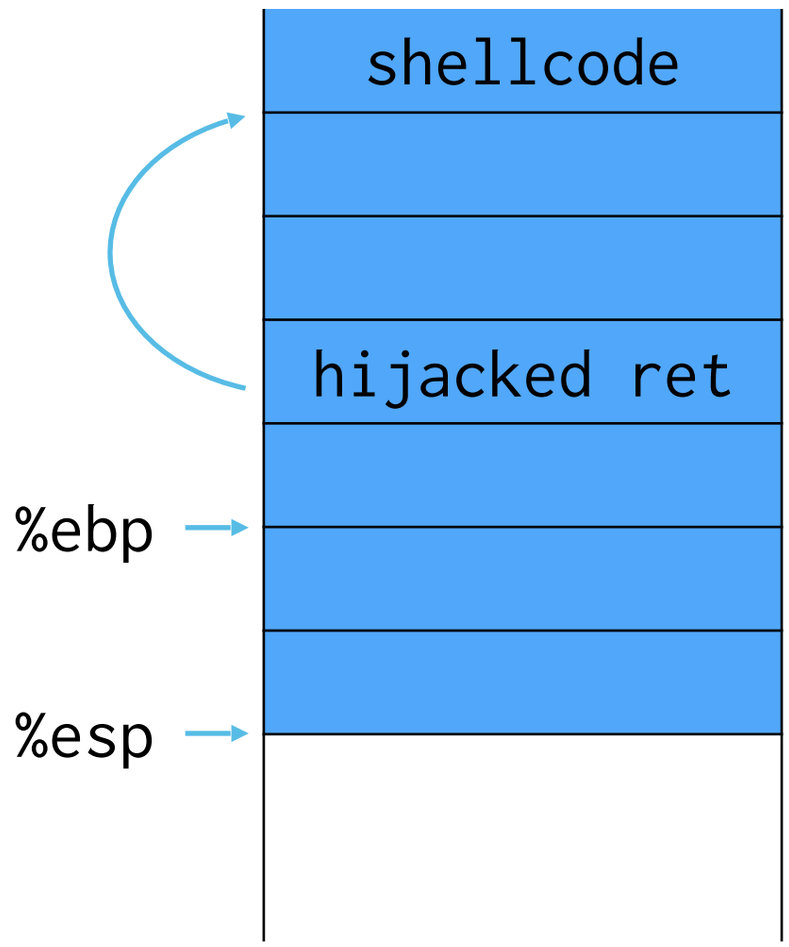
Recall our memory layout



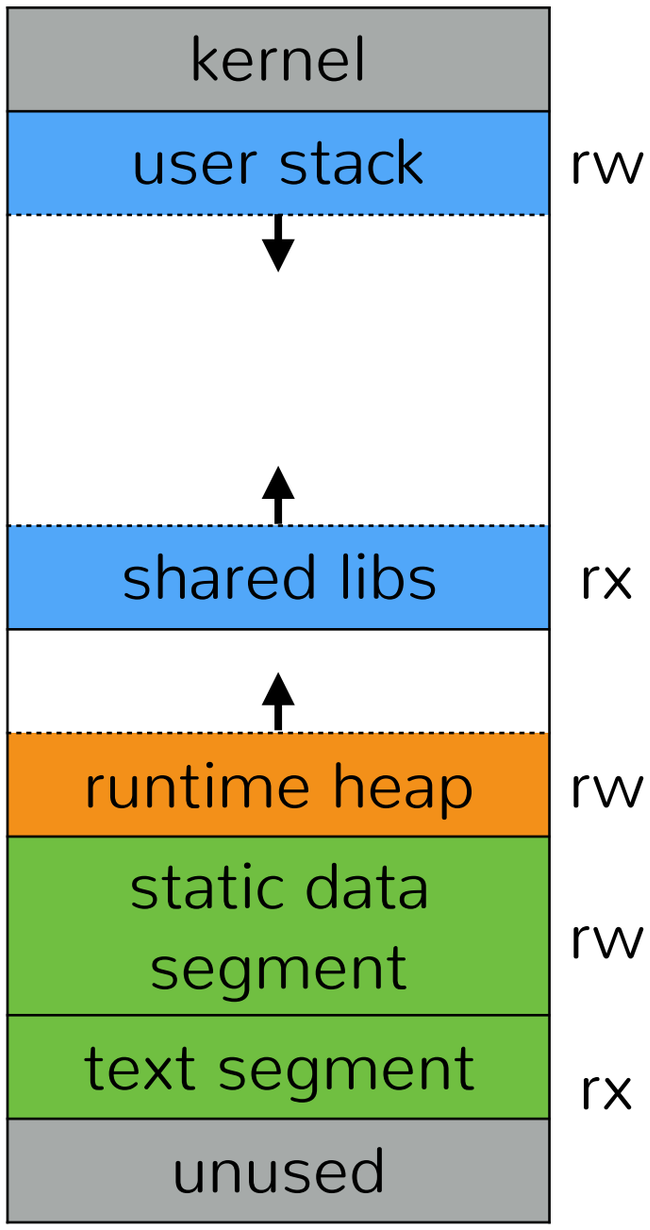


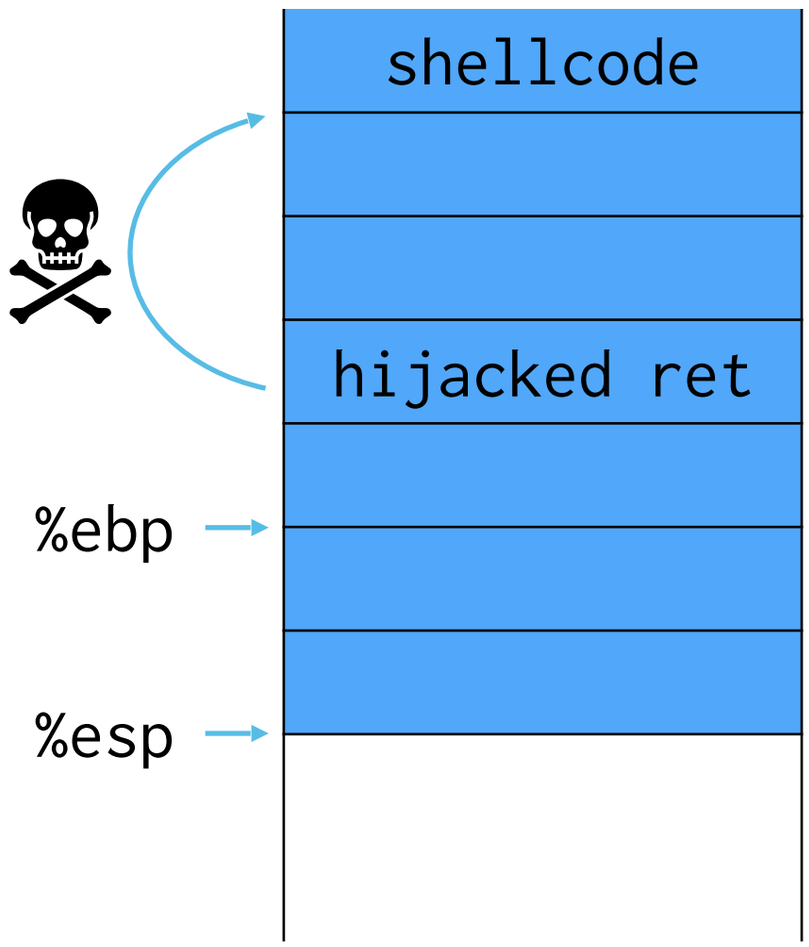
Recall our memory layout





Recall our memory layout





W^X tradeoffs

- **Easy to deploy:** No code changes or recompilation
- **Fast:** Enforced in hardware
  - Also a downside: what do you do on embedded devices?
- What if some pages need to be both writeable and executable?
  - What programs do you use that need this?

How can we defeat W^X?

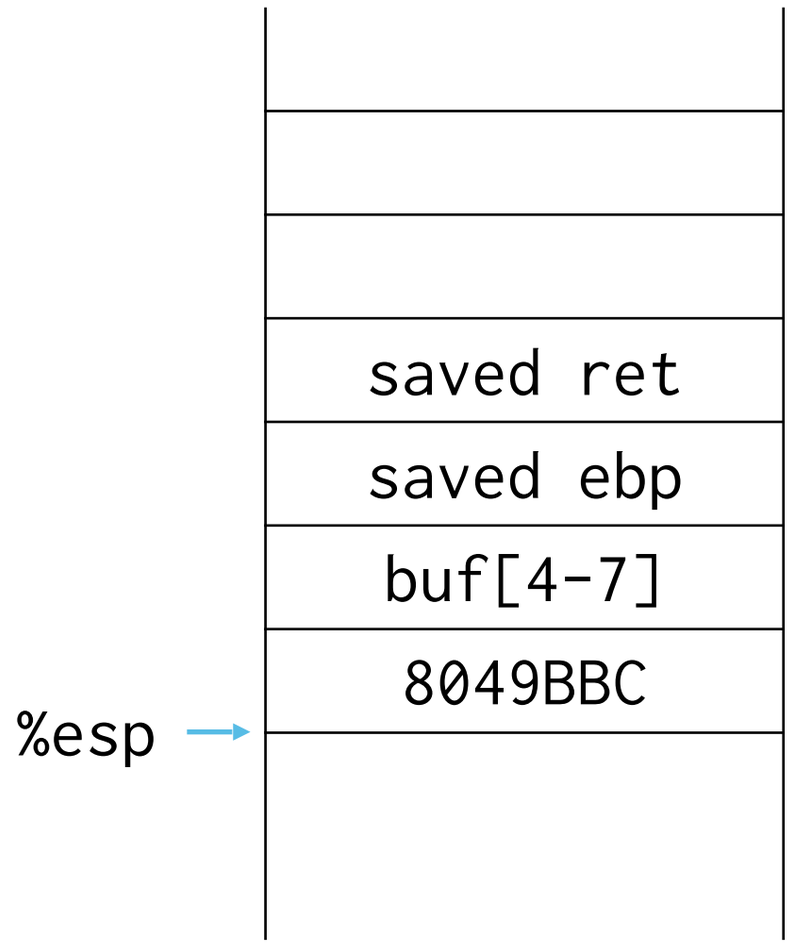
- Can still write to stack
  - Jump to existing code
- Search executable for code that does what you want
  - E.g. if program calls **`system("/bin/sh")`** you're done
  - libc is a good source of code (return-into-libc attacks)

Employees must
wash hands before
returning to libc

Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “`/bin/sh`”

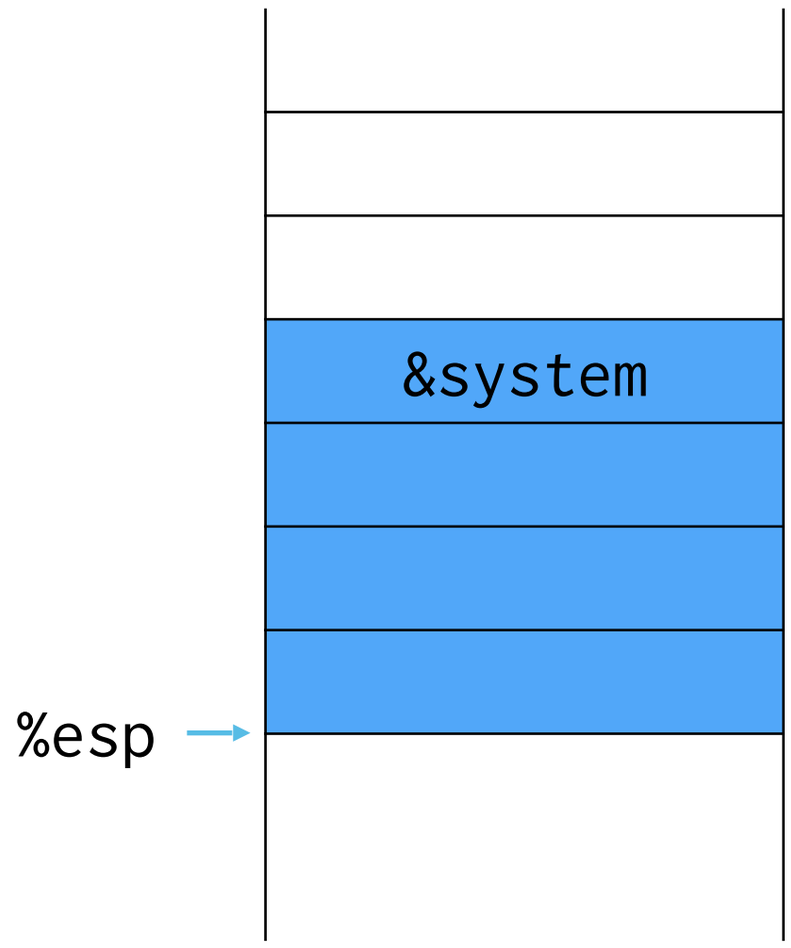
Our vulnerable function:



Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “`/bin/sh`”

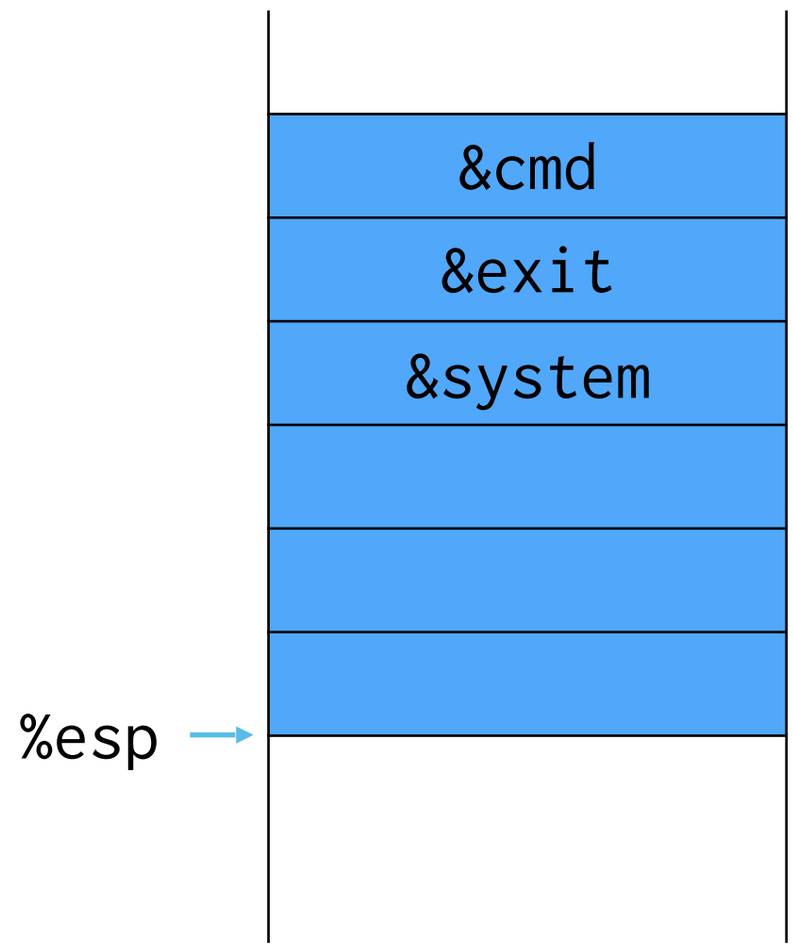
Our vulnerable function:



Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “`/bin/sh`”

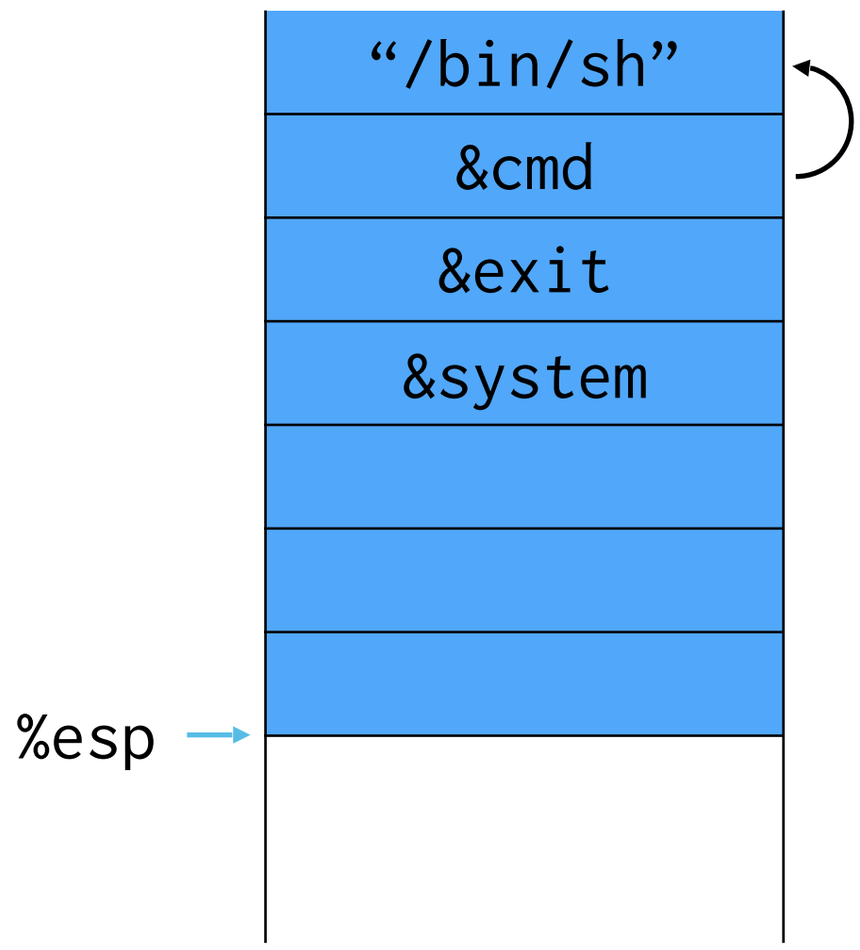
Our vulnerable function:



Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “`/bin/sh`”

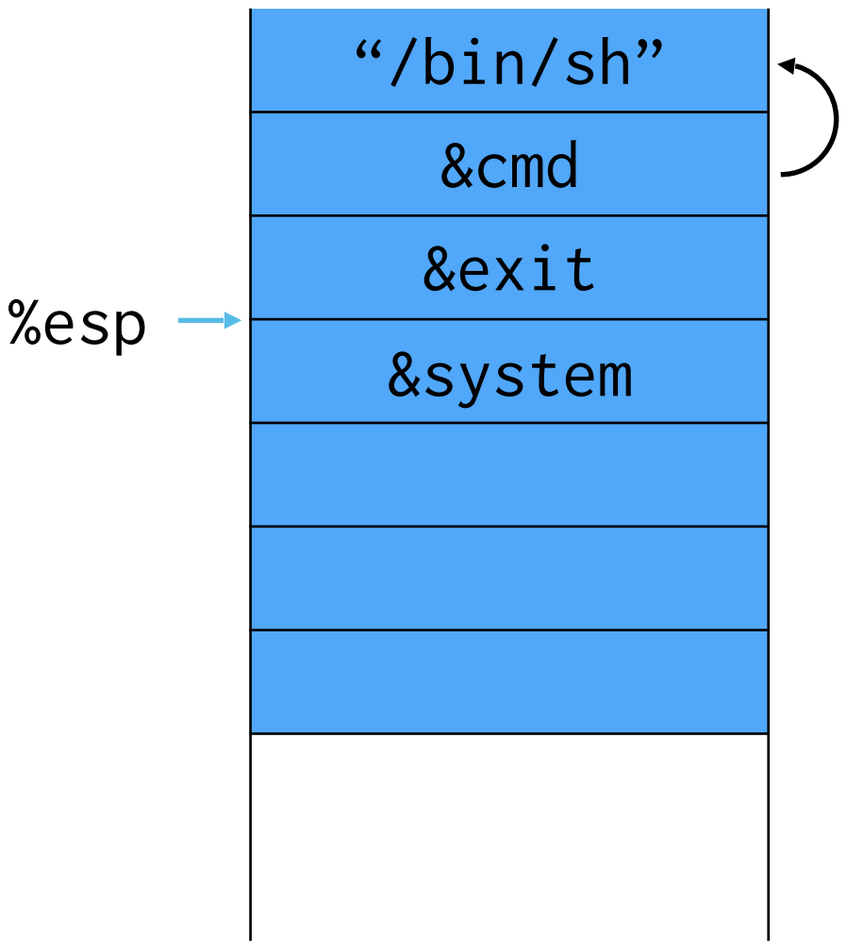
Our vulnerable function:



Calling system

- We already did this with foo
- Calling `system()` is the same, but need to argument to string “`/bin/sh`”

Our vulnerable function:



Can we inject code?

Can we inject code?

NAME top

mprotect, pkey_mprotect - set protection on a region of memory

SYNOPSIS top

```
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);

#define _GNU_SOURCE             /* See feature_test_macros(7) */
#include <sys/mman.h>

int pkey_mprotect(void *addr, size_t len, int prot, int pkey);
```

DESCRIPTION top

mprotect() changes the access protections for the calling process's memory pages containing any part of the address range in the interval [*addr*, *addr*+*len*-1]. *addr* must be aligned to a page boundary.

If the calling process tries to access memory in a manner that violates the protections, then the kernel generates a **SIGSEGV** signal for the process.

prot is a combination of the following access flags: **PROT_NONE** or a bitwise-or of the other values in the following list:

PROT_NONE The memory cannot be accessed at all.

PROT_READ The memory can be read.

PROT_WRITE The memory can be modified.

PROT_EXEC The memory can be executed.

Can we inject code?

- Just-in-time compilers produce data that becomes executable code
- JIT spraying:
  - 1. Spray heap with shellcode (and NOP slides)
  - 2. Overflow code pointer to point to spray area

What does JIT shellcode look like?

What does JIT shellcode look like?

```
var g1 = 0;
...
var g7 = 0;

for (var i=0; i<100000; ++i) {
        g1 = 50011;      \\ pop ebx; ret;
        g2 = 50009;      \\ pop ecx; ret;
        g3 = 12828721;   \\ xor eax, eax; ret;
        g4 = 12811696;   \\ mov 0x7d, al; ret;
        g5 = 12833329;   \\ xor edx, edx; ret;
        g6 = 12781490;   \\ mov 0x7,  dl; ret;
        g7 = 12812493;   \\ int 0x80; ret;
}
```

What does JIT shellcode look like?

```
var g1 = 0;
...
var g7 = 0;

for (var i=0; i<100000; ++i) {
        g1 = 50011;      \\ pop ebx; ret;
        g2 = 50009;      \\ pop ecx; ret;
        g3 = 12828721;   \\ xor eax, eax; ret;
        g4 = 12811696;   \\ mov 0x7d, al; ret;
        g5 = 12833329;   \\ xor edx, edx; ret;
        g6 = 12781490;   \\ mov 0x7,  dl; ret;
        g7 = 12812493;   \\ int 0x80; ret;
}
```

The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines

Michalis Athanasakis, FORTH, Greece, michath@ics.forth.gr
Elias Athanasopoulos, FORTH, Greece, elathan@ics.forth.gr
Michalis Polychronakis, Stony Brook University, mikepo@cs.stonybrook.edu
Georgios Portokalidis, Stevens Institute of Tech., gportoka@stevens.edu
Sotiris Ioannidis, FORTH, Greece, sotiris@ics.forth.gr

How do we defend against this?

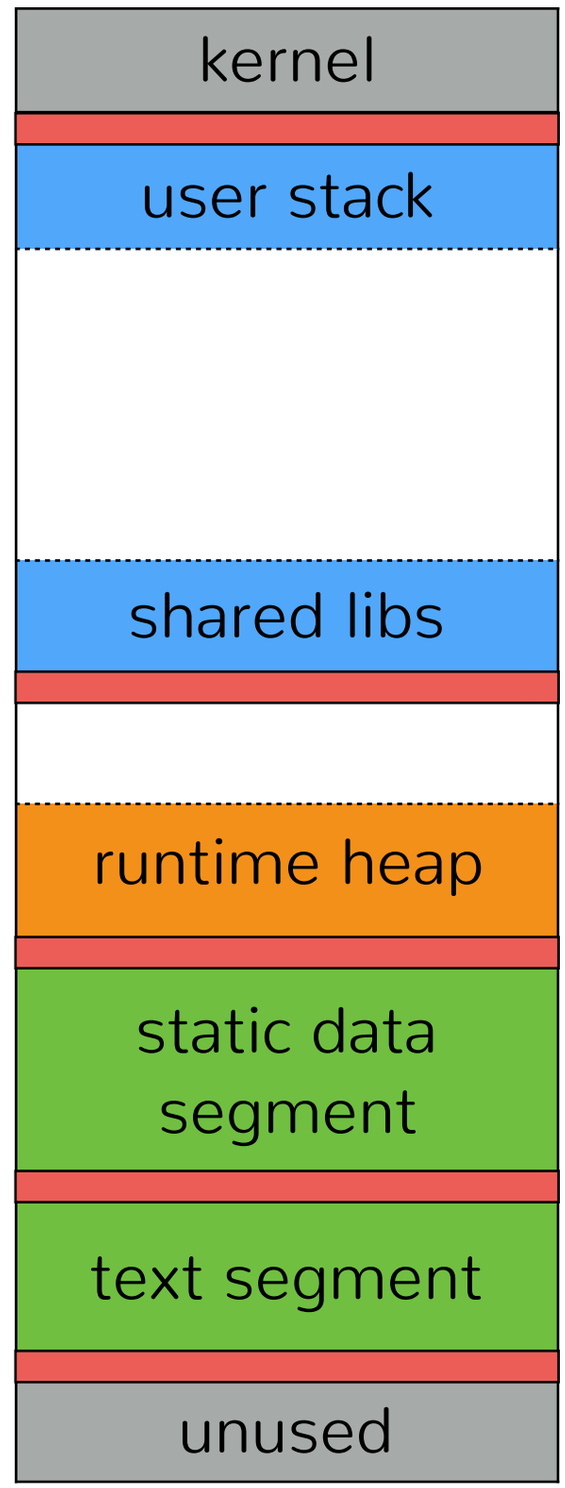
- Modify the JavaScript JIT
  - Store JavaScript strings in separate heap from rest
  - Blind constants
- Ongoing arms race
  - E.g., Wasm makes it easier for attackers: gap between Wasm and x86/ARM is much smaller than JavaScript

Buffer overflow mitigations

- Avoid unsafe functions (last lecture)
- Stack canaries
- Separate control stack
- Memory writable or executable, not both (W^X)
- → Address space layout randomization (ASLR)

ASLR

- Traditional exploits need precise addresses
  - stack-based overflows: location of shellcode
  - return-into-libc: library addresses
- **Insight:** Make it harder for attacker to guess location of shellcode/libc by randomizing the address of different memory regions



When do we randomize?

When do we randomize?

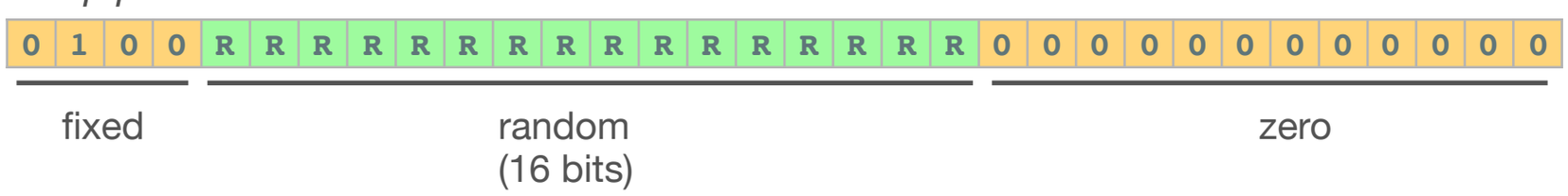
```
┌[d@bedsty code master*]
└➤ cat /proc/self/maps | egrep '(libc|heap|stack)'
55555555f000-555555580000 rw-p 00000000 00:00 0                          [heap]
7ffff7dce000-7ffff7df3000 r--p 00000000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7df3000-7ffff7f3d000 r-xp 00025000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f3d000-7ffff7f86000 r--p 0016f000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f86000-7ffff7f87000 ---p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f87000-7ffff7f8a000 r--p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f8a000-7ffff7f8d000 rw-p 001bb000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fffffffde000-7ffffffff000 rw-p 00000000 00:00 0                         [stack]
┌[d@bedsty code master*]
└➤ cat /proc/self/maps | egrep '(libc|heap|stack)'
55555555f000-555555580000 rw-p 00000000 00:00 0                          [heap]
7ffff7dce000-7ffff7df3000 r--p 00000000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7df3000-7ffff7f3d000 r-xp 00025000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f3d000-7ffff7f86000 r--p 0016f000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f86000-7ffff7f87000 ---p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f87000-7ffff7f8a000 r--p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffff7f8a000-7ffff7f8d000 rw-p 001bb000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fffffffde000-7ffffffff000 rw-p 00000000 00:00 0                         [stack]
┌[d@bedsty code master*]
└➤ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
2
┌[d@bedsty code master*]
└➤ cat /proc/self/maps | egrep '(libc|heap|stack)'
564346042000-564346063000 rw-p 00000000 00:00 0                          [heap]
7ff28472c000-7ff284751000 r--p 00000000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ff284751000-7ff28489b000 r-xp 00025000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ff28489b000-7ff2848e4000 r--p 0016f000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ff2848e4000-7ff2848e5000 ---p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ff2848e5000-7ff2848e8000 r--p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ff2848e8000-7ff2848eb000 rw-p 001bb000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fffc73cd000-7fffc73ee000 rw-p 00000000 00:00 0                          [stack]
┌[d@bedsty code master*]
└➤ cat /proc/self/maps | egrep '(libc|heap|stack)'
55945b021000-55945b042000 rw-p 00000000 00:00 0                          [heap]
7fd596208000-7fd59622d000 r--p 00000000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fd59622d000-7fd596377000 r-xp 00025000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fd596377000-7fd5963c0000 r--p 0016f000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fd5963c0000-7fd5963c1000 ---p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fd5963c1000-7fd5963c4000 r--p 001b8000 fe:02 2100102                    /usr/lib/libc-2.29.so
7fd5963c4000-7fd5963c7000 rw-p 001bb000 fe:02 2100102                    /usr/lib/libc-2.29.so
7ffe9b1b0000-7ffe9b1d1000 rw-p 00000000 00:00 0                          [stack]
```

How much randomness?

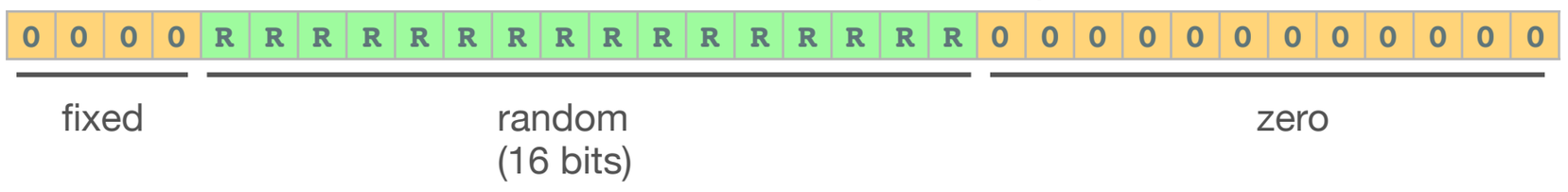
32-bit PaX ASLR (x86)

Stack:

Mapped area:



Executable code, static variables, and heap:



Tradeoff

- **Intrusive:** Need compiler, linker, loader support
  - Process layout must be randomized
  - Programs must be compiled to not have absolute jumps
- **Incurs overhead:** increases code size & perf overhead
- **Also mitigates heap-based overflow attacks**

How can we defeat ASLR?

- Older Linux would let local attacker read the stack start address from `/proc/<pid>/stat`
- -fno-pie binaries have fixed code and data addresses
  - Enough to carry out control-flow-hijacking attacks
- Each region has random offset, but layout is fixed
  - Single address in a region leaks every address in region
- Brute force for 32-bit binaries and/or pre-fork binaries
- Heap spray for 64-bit binaries

Derandomizing ALSR

- **Attack goal:** call `system()` with attacker arg
- **Target:** Apache daemon
  - **Vulnerability:** buffer overflow in `ap_getline()`

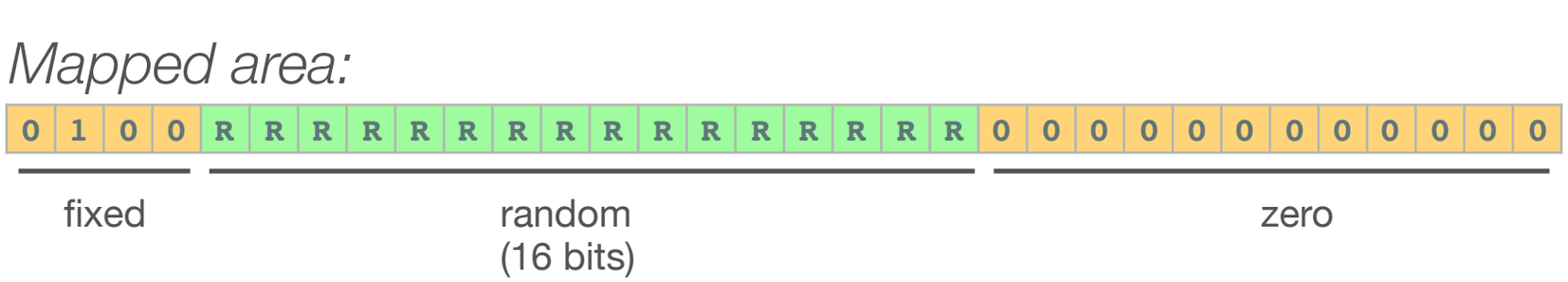
```
char buf[64];
…
strcpy(buf, s); // overflow
```

Assumptions

- W^X enabled
- PaX ASLR enabled
  - Apache forks child processes to handle client interaction
  - Recall how re-randomization works?

Attack steps

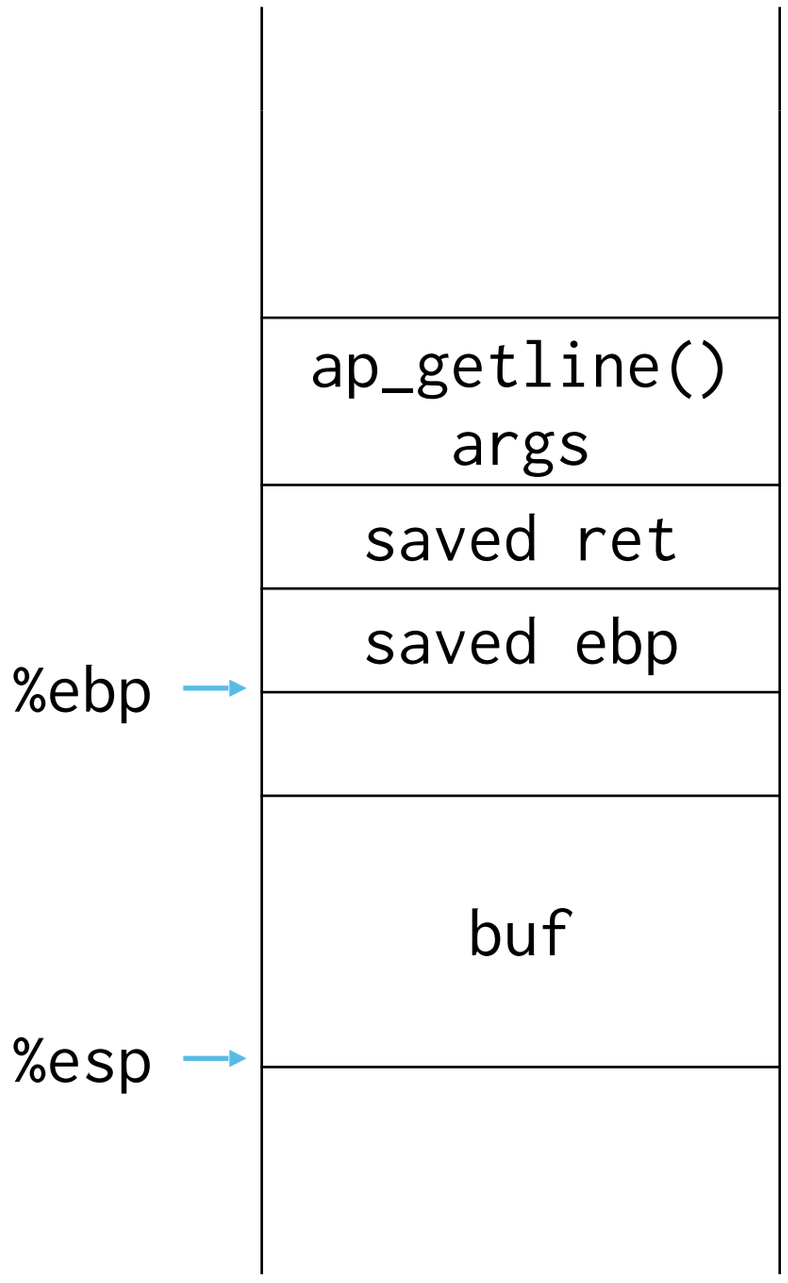
- **Stage 1**: Find base of mapped region



- **Stage 2**: Call `system()` with command string

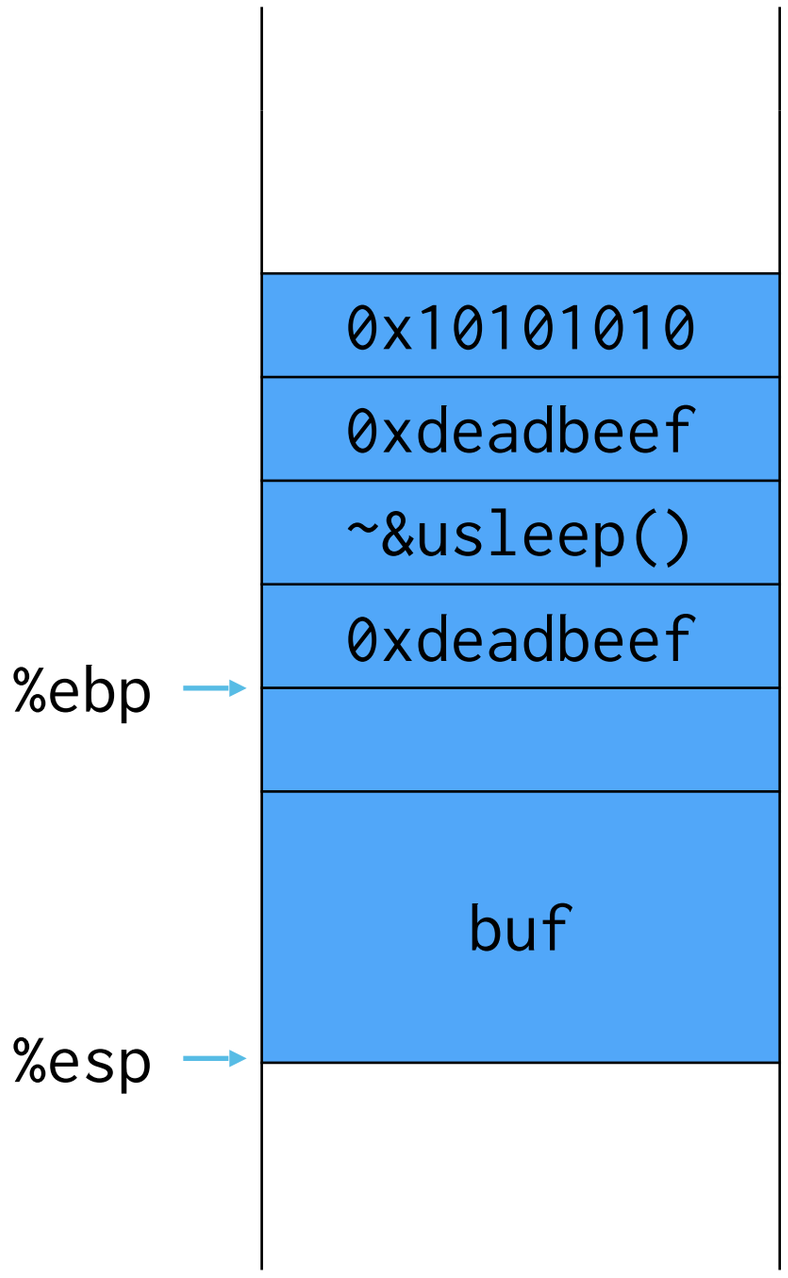
How do we find the mapped region?

- Observation: layout of mapped region (libc) is fixed
- Overwrite saved return pointer with a guess to `usleep()`
  - base + offset of usleep
  - non-negative argument



How do we find the mapped region?

- Observation: layout of mapped region (libc) is fixed
- Overwrite saved return pointer with a guess to `usleep()`
  - base + offset of usleep
  - non-negative argument



Finding base of mapped region

- If we guessed `usleep()` address right
  - ➤
- If we guessed `usleep()` address wrong
  - ➤
- Use this to tell if we guessed base of mapped region correctly

Finding base of mapped region

- If we guessed `usleep()` address right
  - Server will freeze for 16 seconds, then crash
- If we guessed `usleep()` address wrong
  -
- Use this to tell if we guessed base of mapped region correctly

Finding base of mapped region

- If we guessed `usleep()` address right
  - Server will freeze for 16 seconds, then crash
- If we guessed `usleep()` address wrong
  - Server will (likely) crash immediately
- Use this to tell if we guessed base of mapped region correctly

Derandomizing ASLR

Derandomizing ASLR

- What is the success probability?

Derandomizing ASLR

- What is the success probability?
  - $1/2^{16}$ — 65,536 tries maximum

Derandomizing ASLR

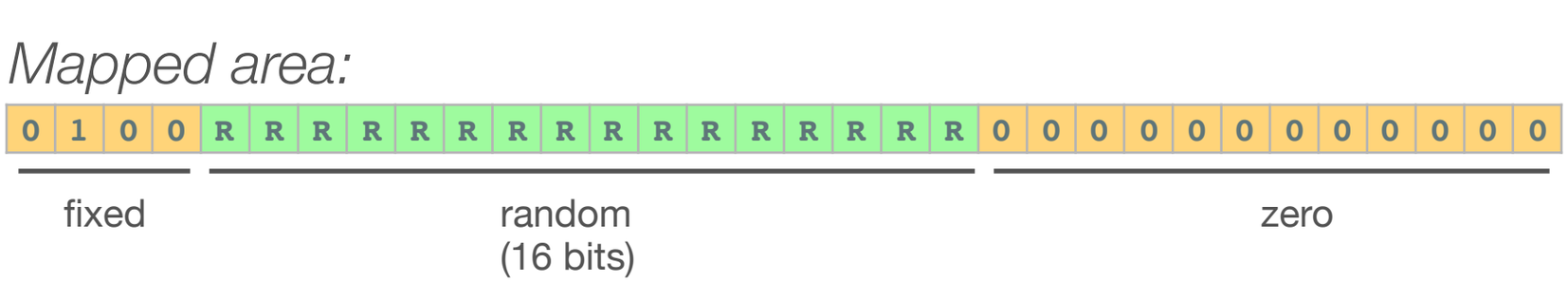
- What is the success probability?
  - $1/2^{16}$ — 65,536 tries maximum
- Do we need to derandomize the stack base?

Derandomizing ASLR

- What is the success probability?
  - $1/2^{16}$ — 65,536 tries maximum
- Do we need to derandomize the stack base?
  - No!

Attack steps

- **Stage 1**: Find base of mapped region (libc)



- **Stage 2**: Call `system()` with command string

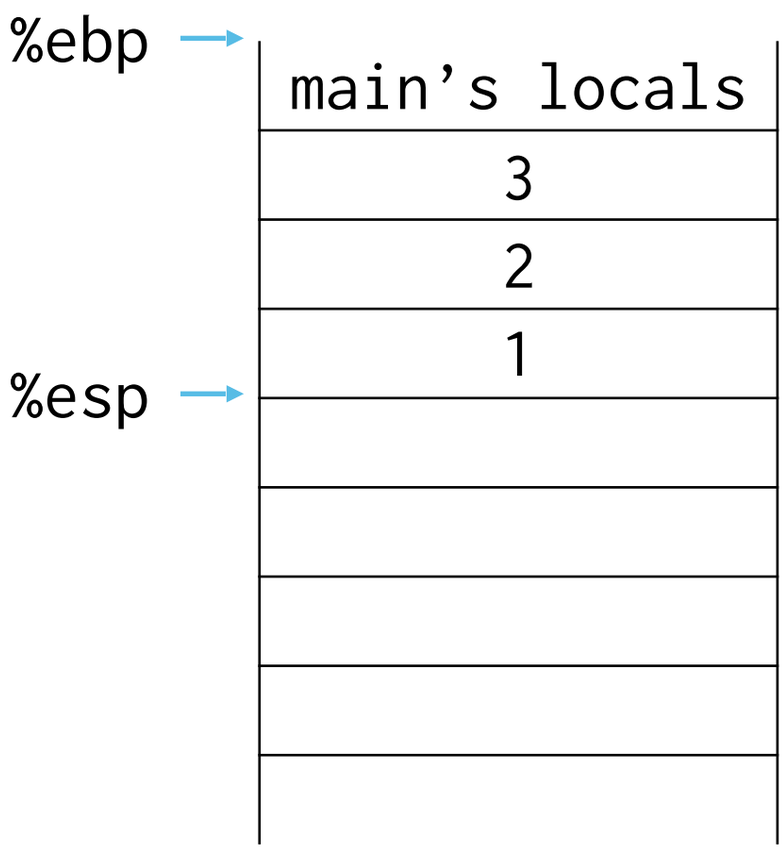
How do we call system?

- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`

How do we call system?

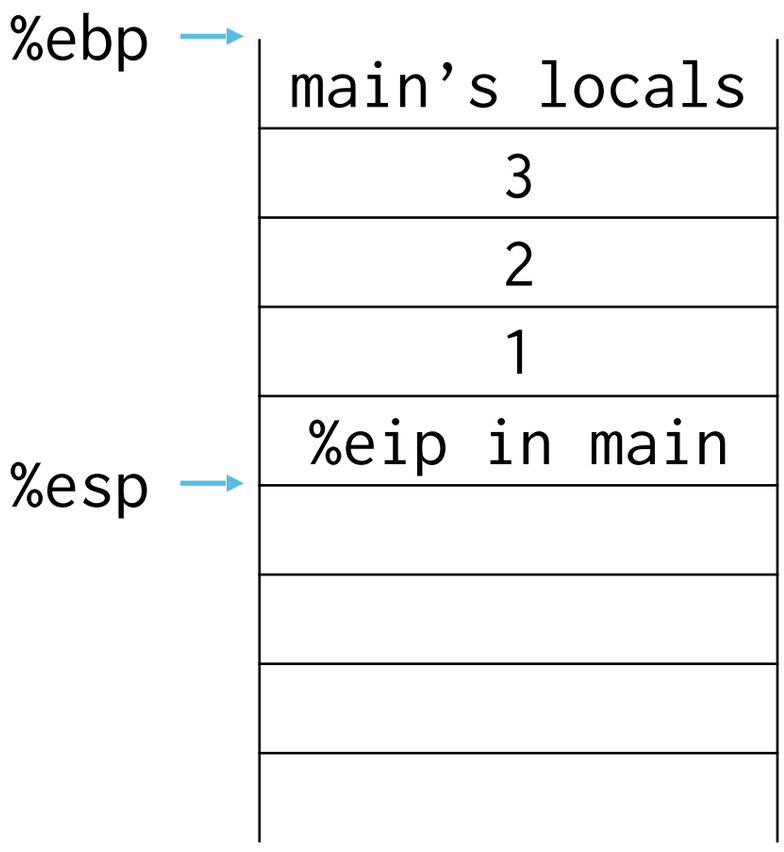
Let’s look at this outside the Apache example

Review: calling and returning



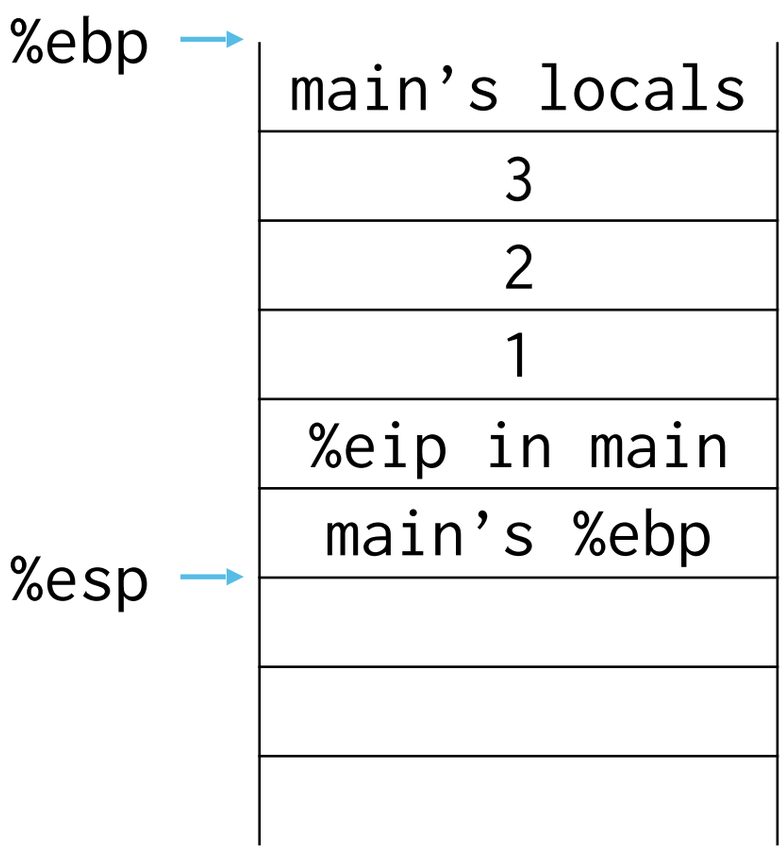

```
main()
-> foo(1,2,3)
---> bar(4)
```

Review: calling and returning



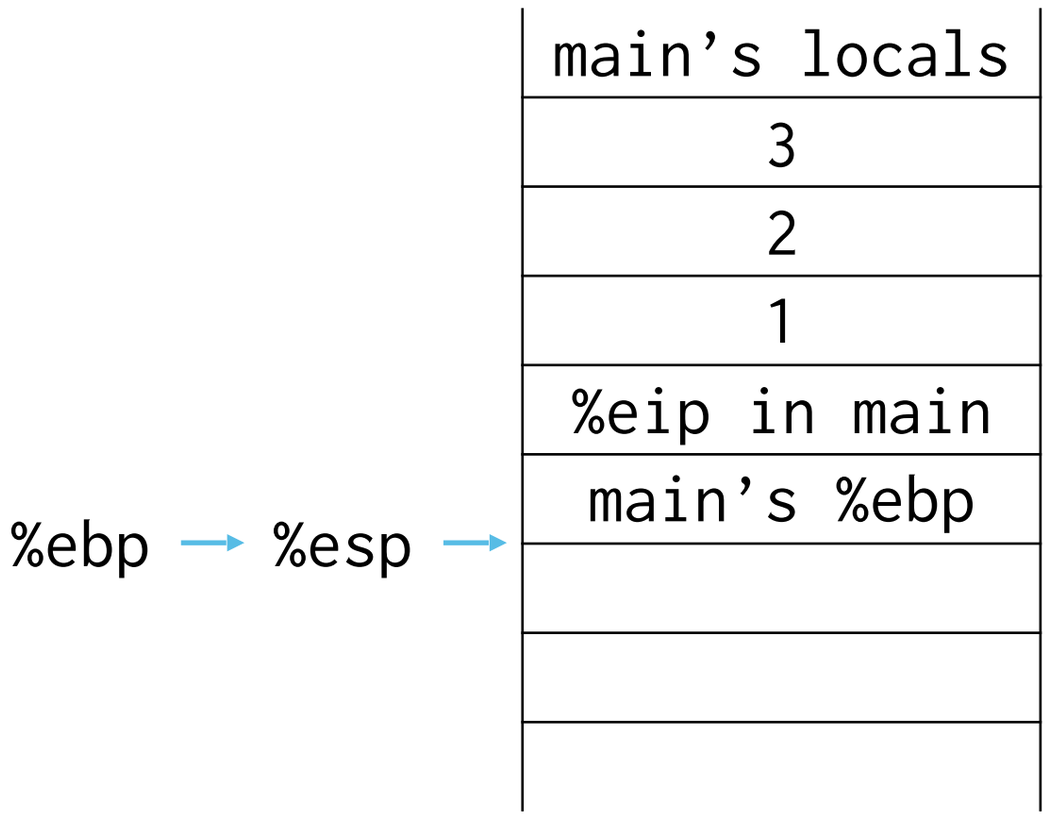

```
main()
-> foo(1,2,3)
  ---> bar(4)
```

Review: calling and returning



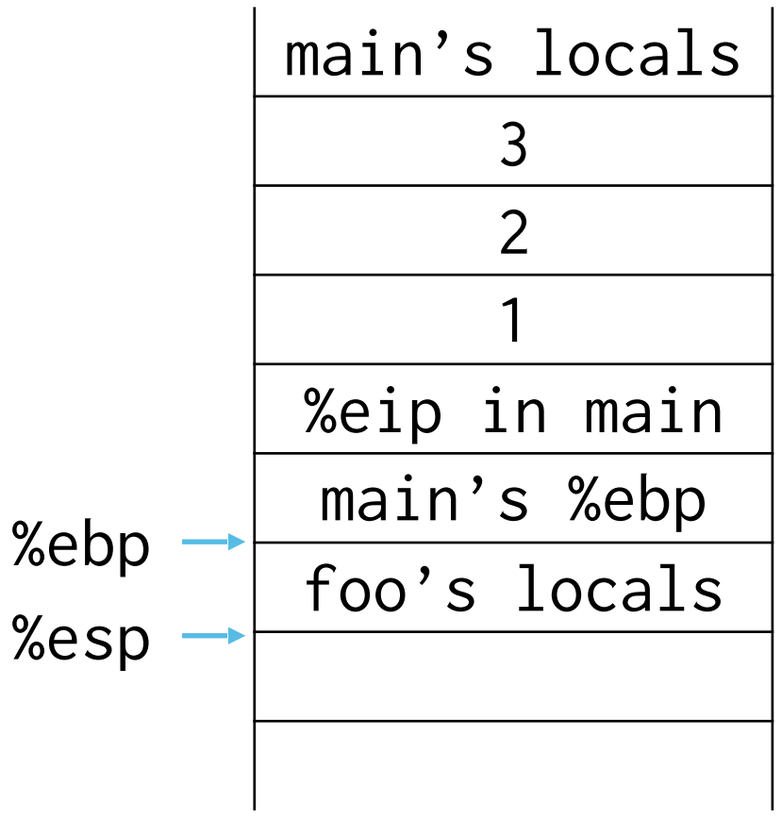

```
main()
-> foo(1,2,3)
   ---> bar(4)
```

Review: calling and returning



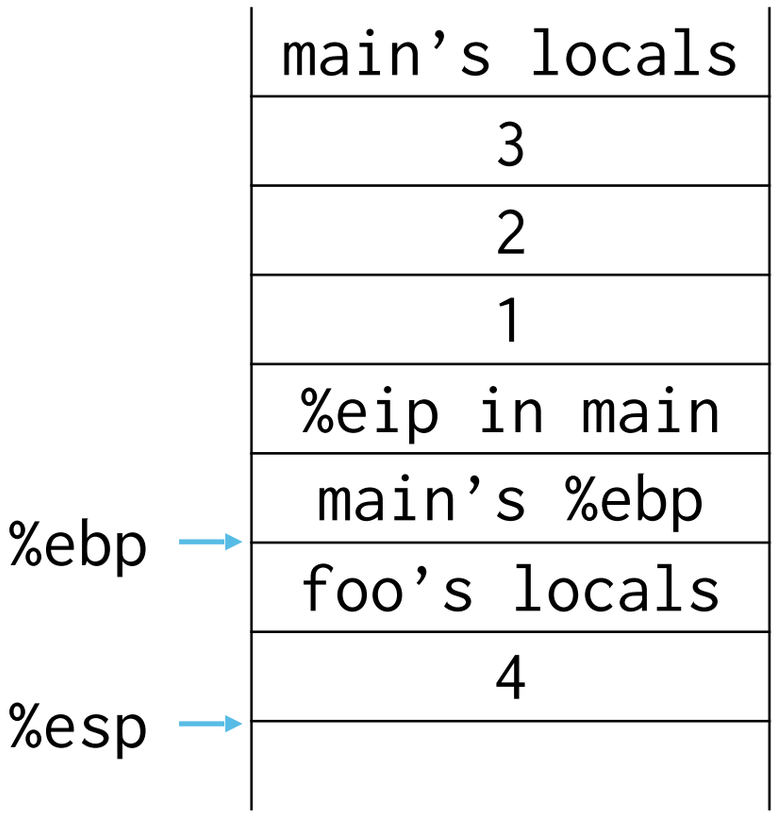

```
main()
-> foo(1,2,3)
---> bar(4)
```

Review: calling and returning



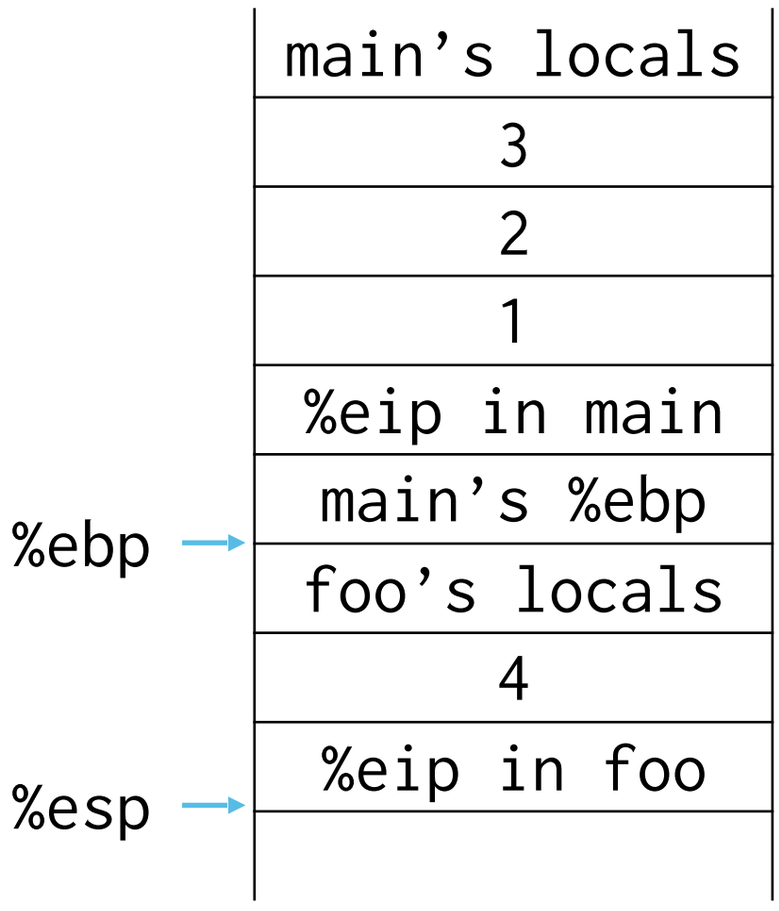

```
main()
-> foo(1,2,3)
   -> bar(4)
```

Review: calling and returning



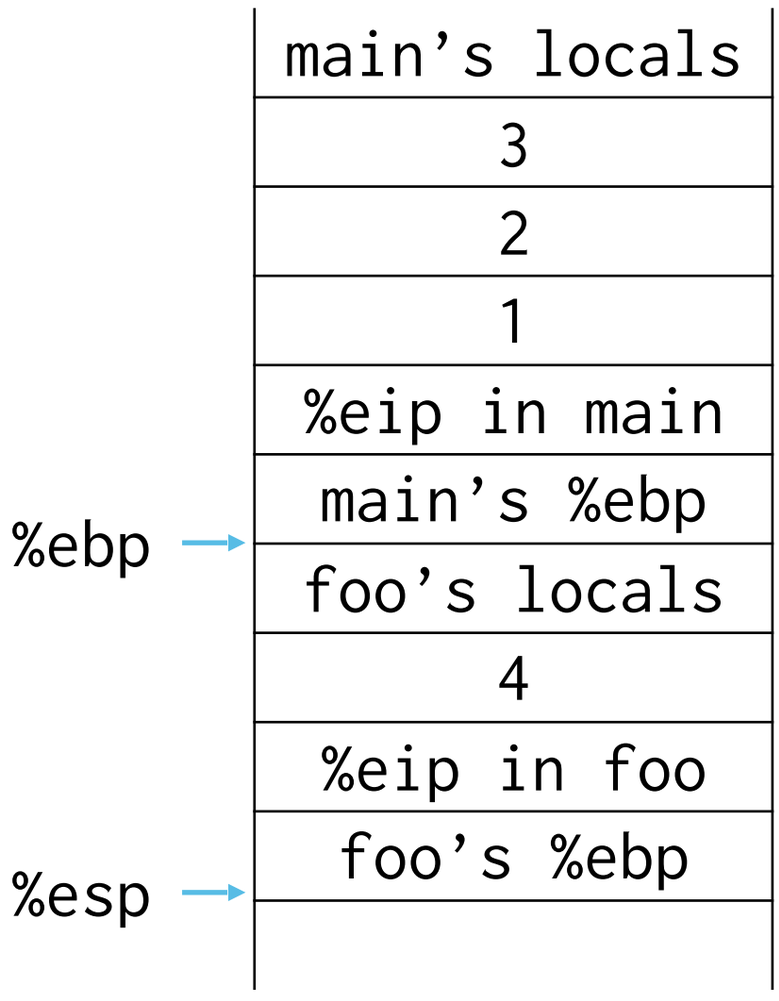

```
main()
-> foo(1,2,3)
---> bar(4)
```

Review: calling and returning



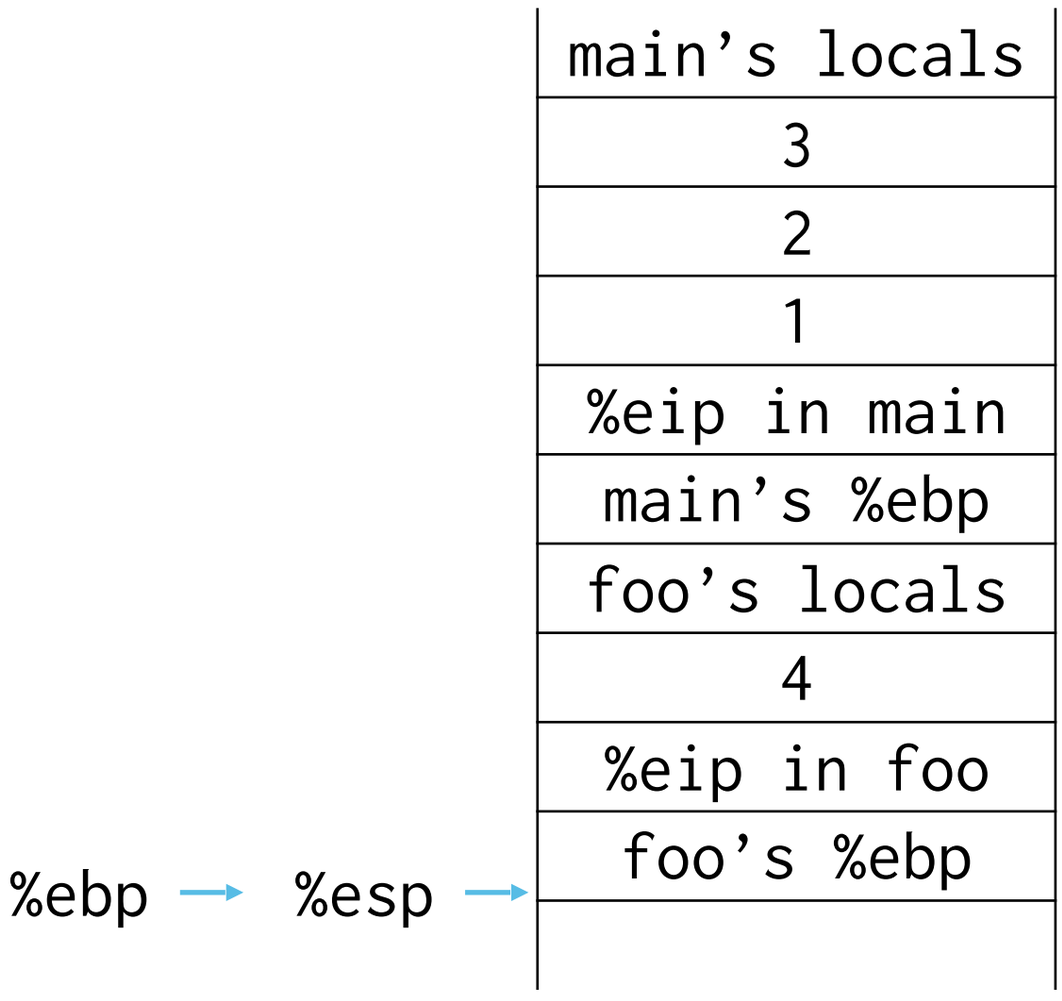

```
main()
-> foo(1,2,3)
--> bar(4)
```

Review: calling and returning



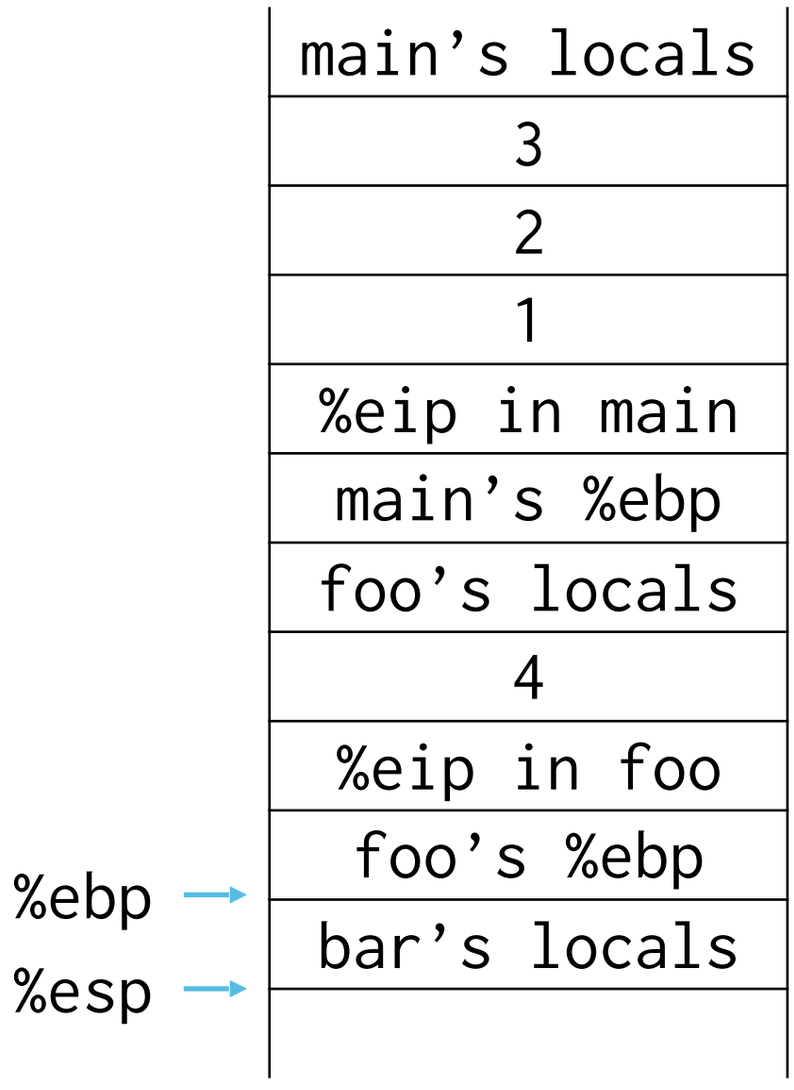

```
main()
-> foo(1,2,3)
--> bar(4)
```

Review: calling and returning



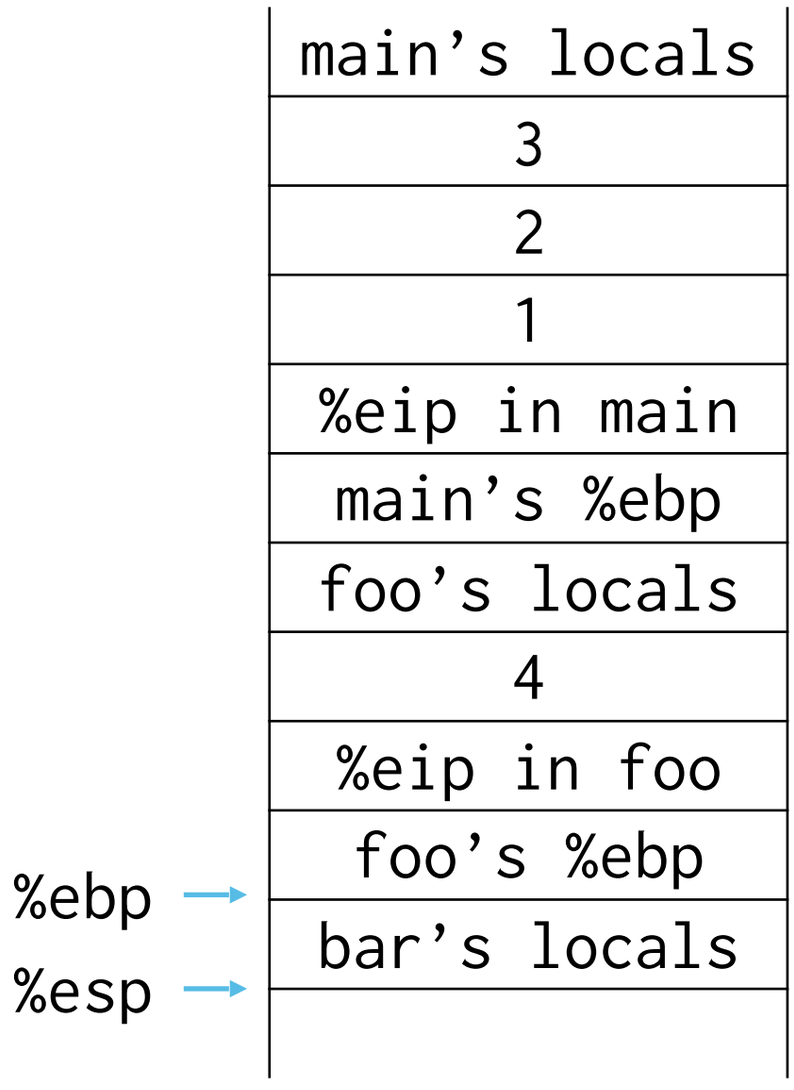

```
main()
-> foo(1,2,3)
--> bar(4)
```

Review: calling and returning




```
main()
-> foo(1,2,3)
--> bar(4)
```

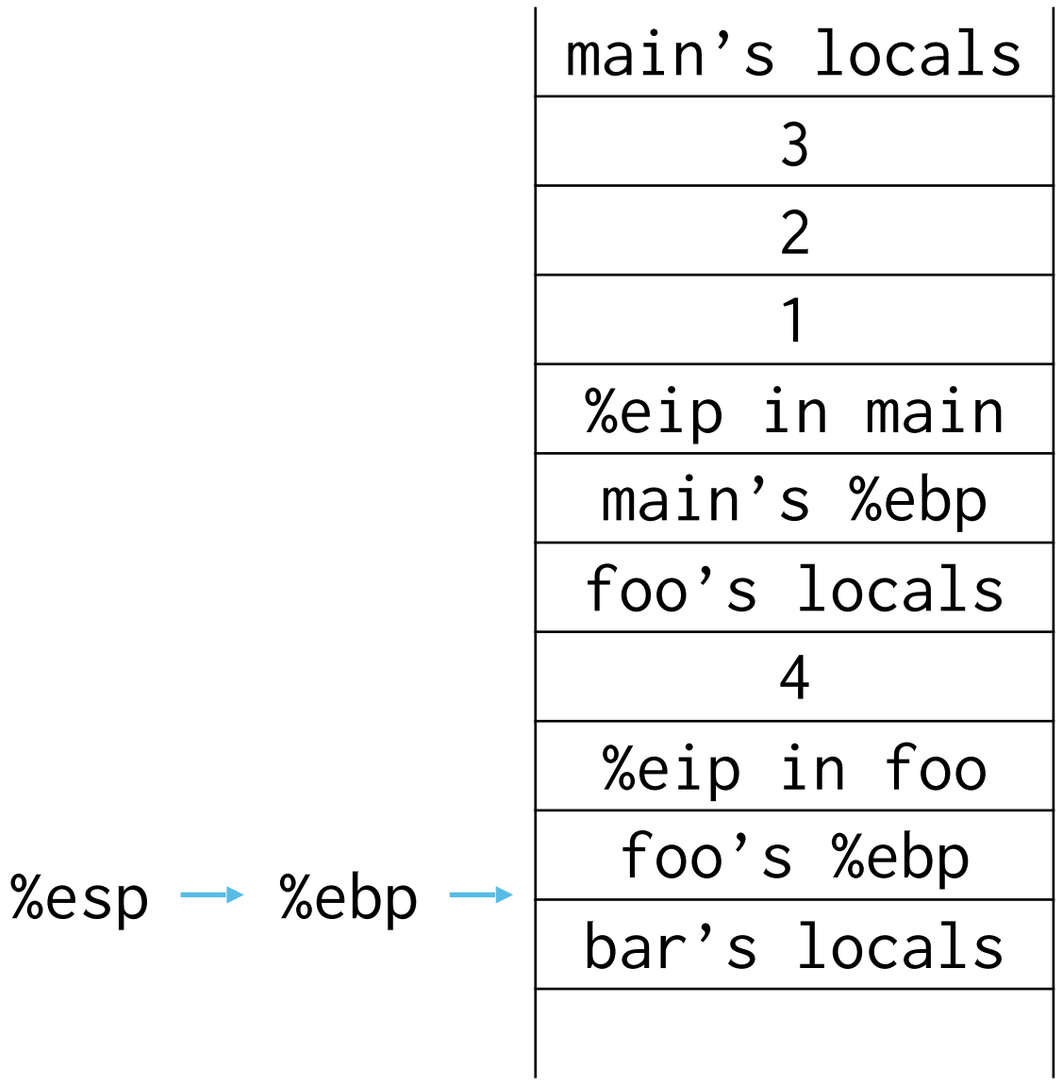
Review: calling and returning




```
main()
-> foo(1,2,3)
    --> bar(4)
```

leave = `mov %ebp, %esp`
`pop %ebp`

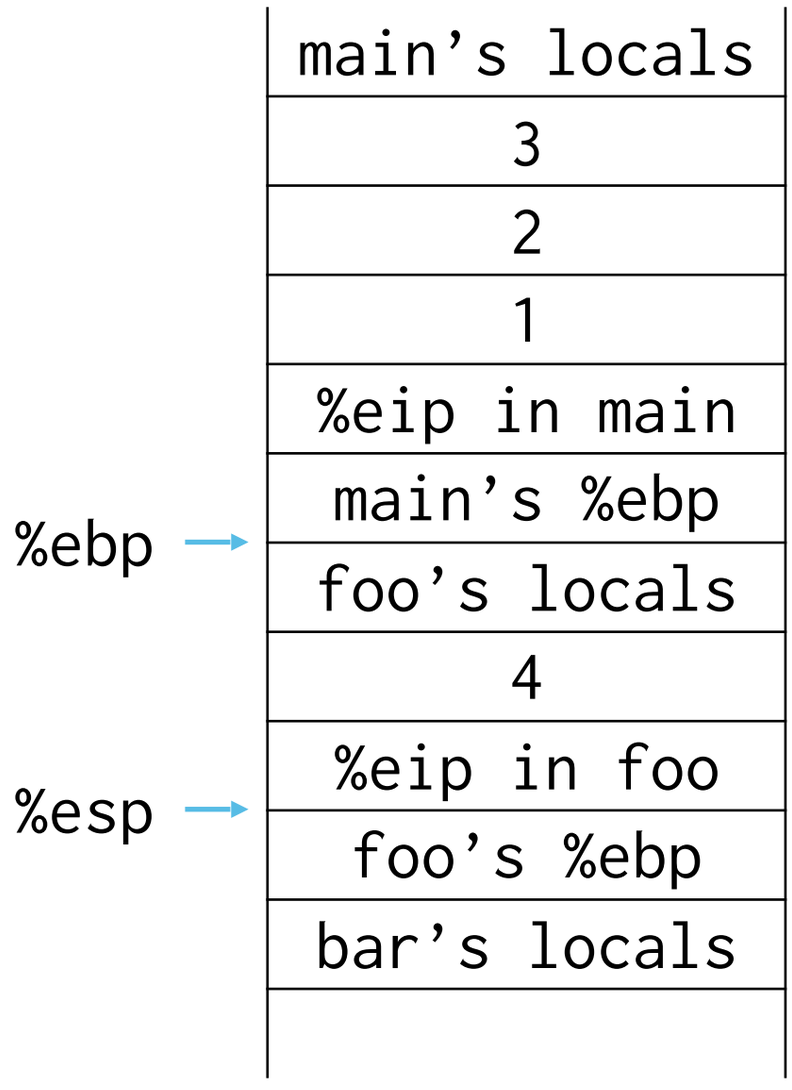
Review: calling and returning




```
main()
-> foo(1,2,3)
--> bar(4)
```

leave = `mov %ebp, %esp`
`pop %ebp`

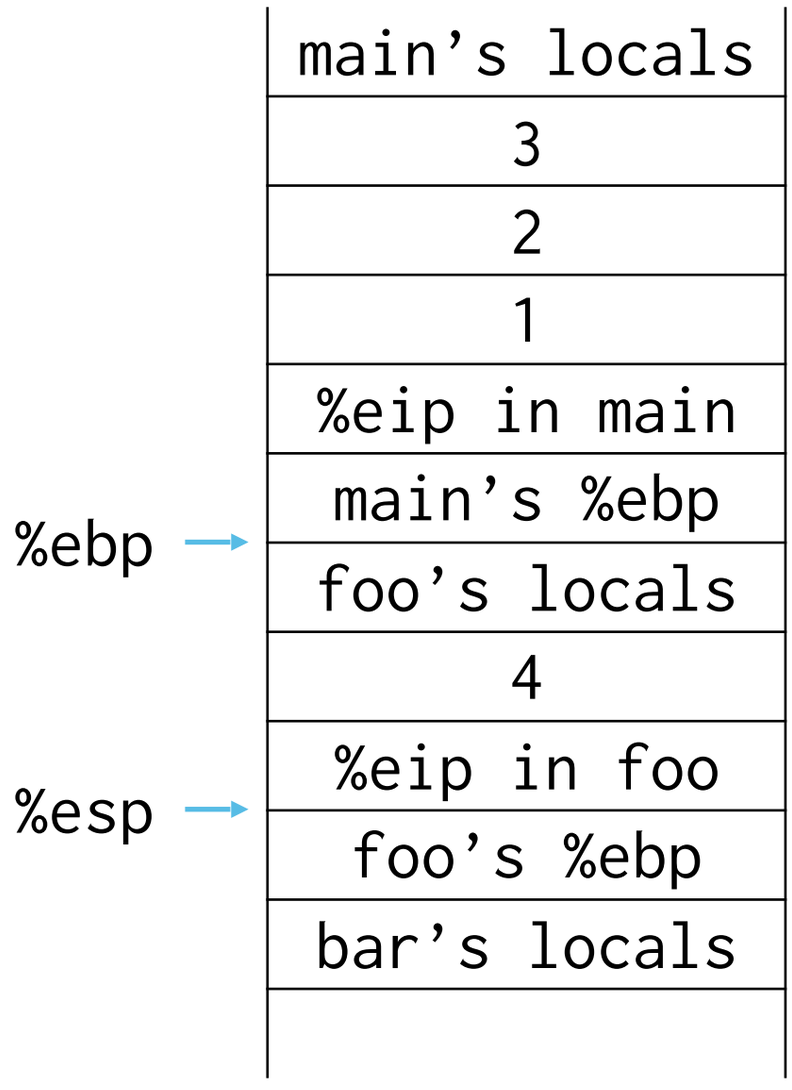
Review: calling and returning




```
main()
-> foo(1,2,3)
--> bar(4)
```

```
leave = mov %ebp, %esp
        pop %ebp
```

Review: calling and returning

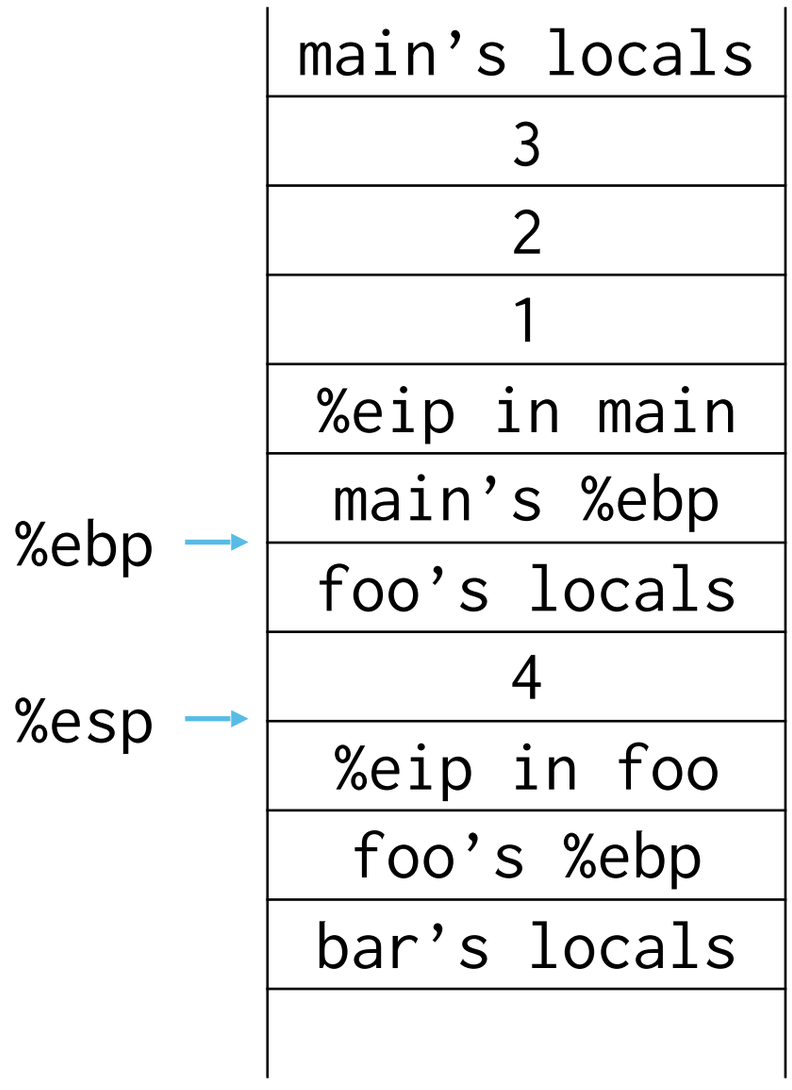



```
main()
-> foo(1,2,3)
```

```
--> bar(4)
```

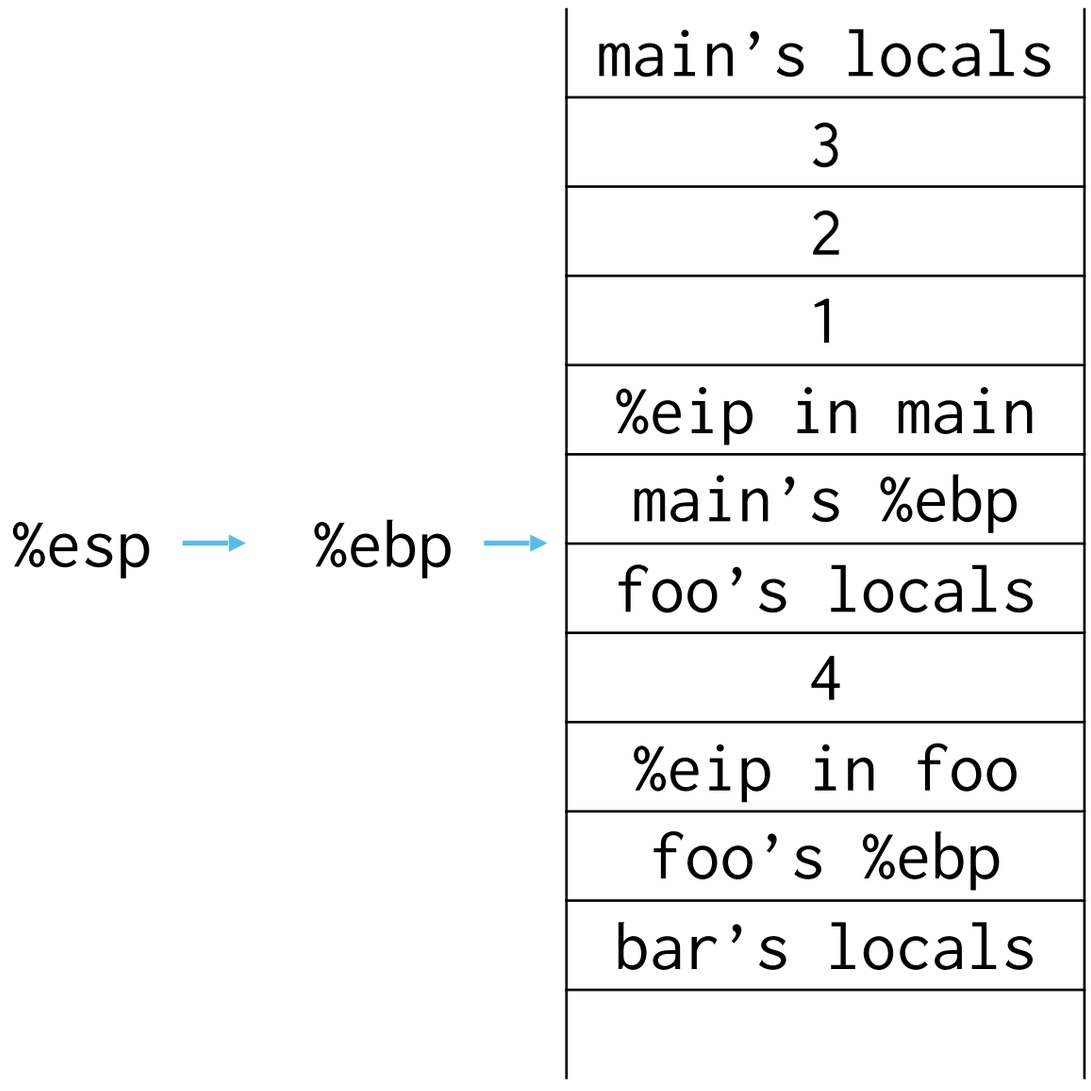
```
ret = pop %eip
```

Review: calling and returning




```
main()
-> foo(1,2,3)
---> bar(4)
```

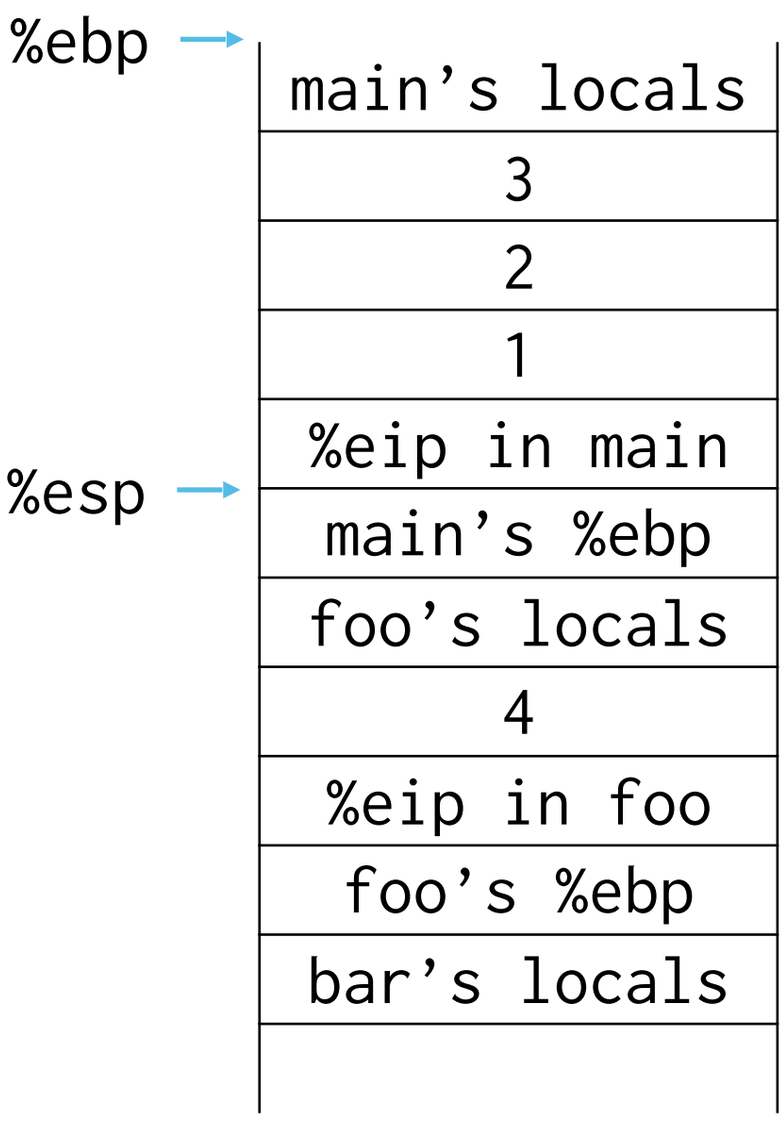
Review: calling and returning




```
main()
-> foo(1,2,3)
   --> bar(4)
```

```
leave = mov %ebp, %esp
        pop %ebp
```

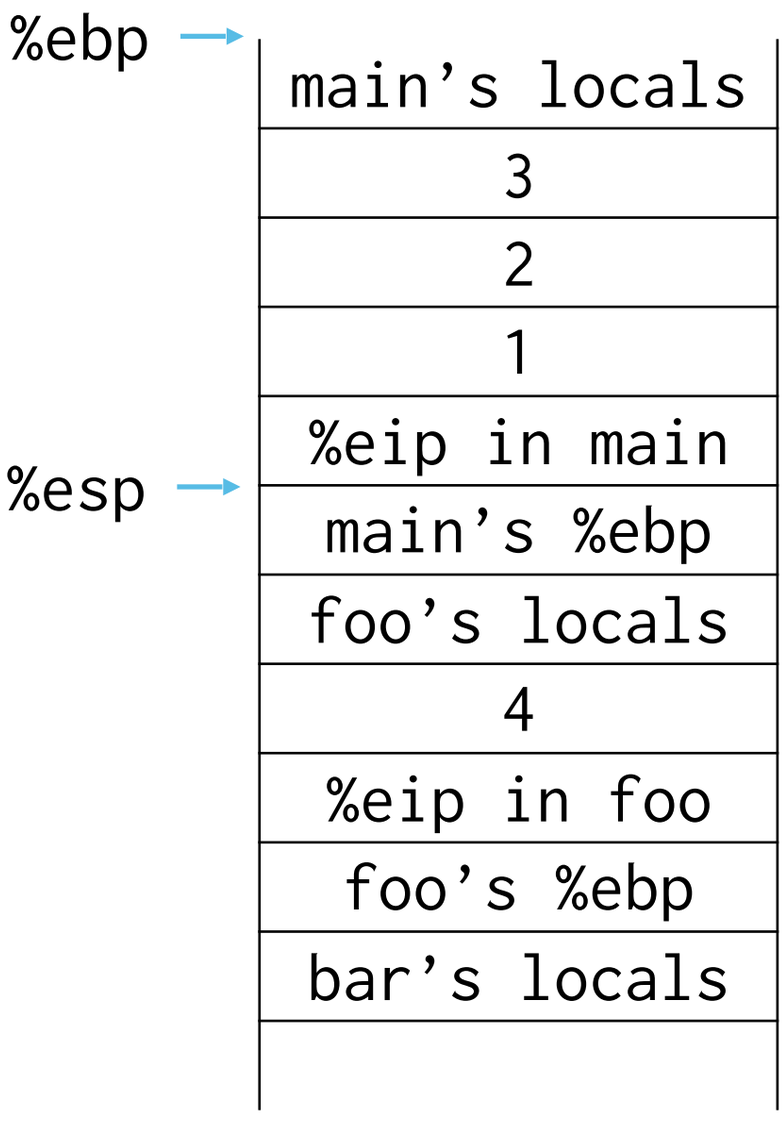
Review: calling and returning




```
main()
-> foo(1,2,3)
--> bar(4)
```

```
leave = mov %ebp, %esp
        pop %ebp
```

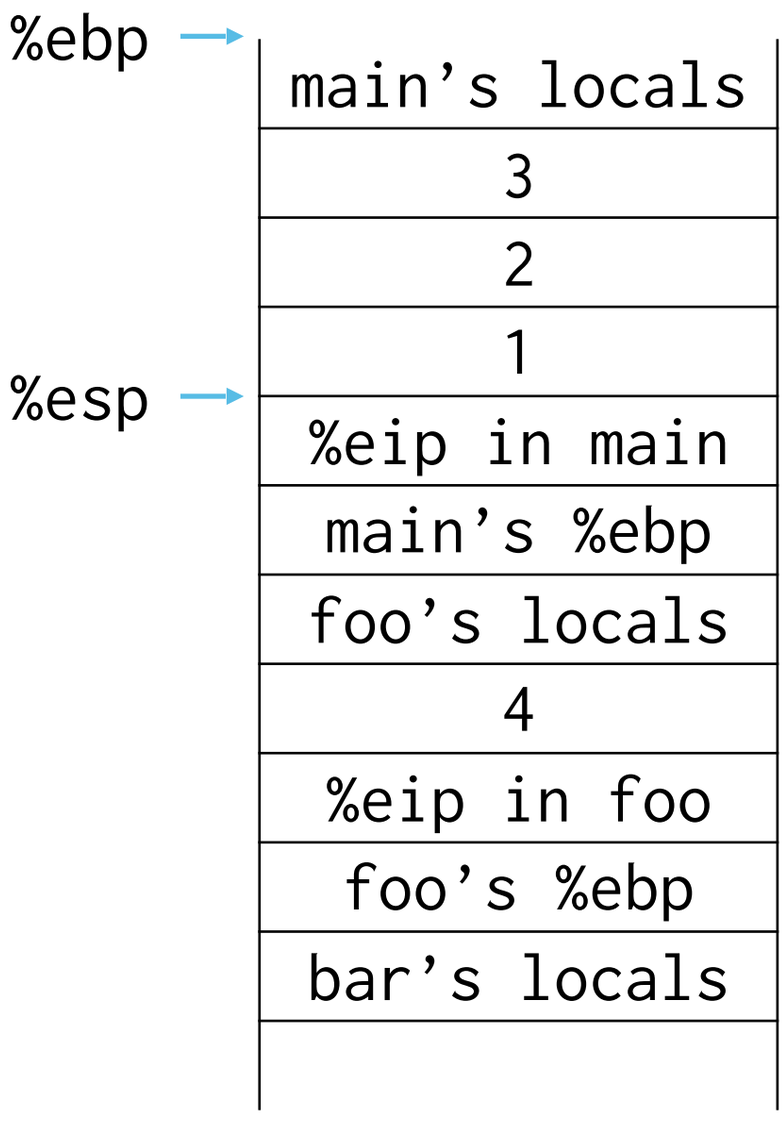
Review: calling and returning




```
main()
-> foo(1,2,3)
---> bar(4)
```

ret = pop %eip

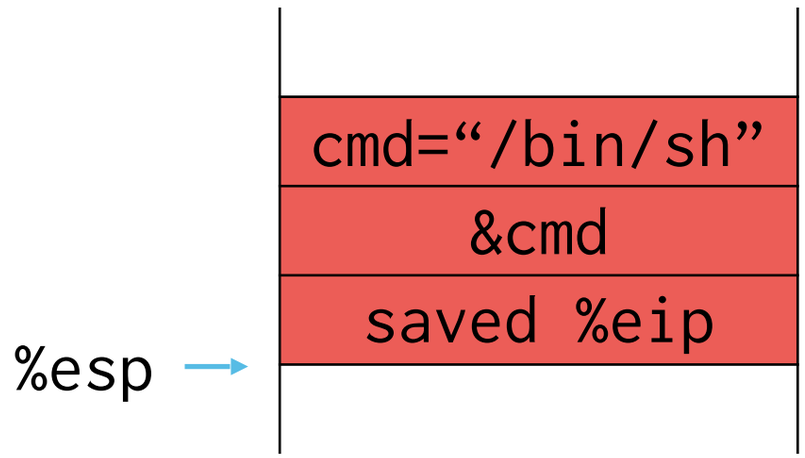
Review: calling and returning




```
main()
-> foo(1,2,3)
---> bar(4)
```

Suppose bar had overflow

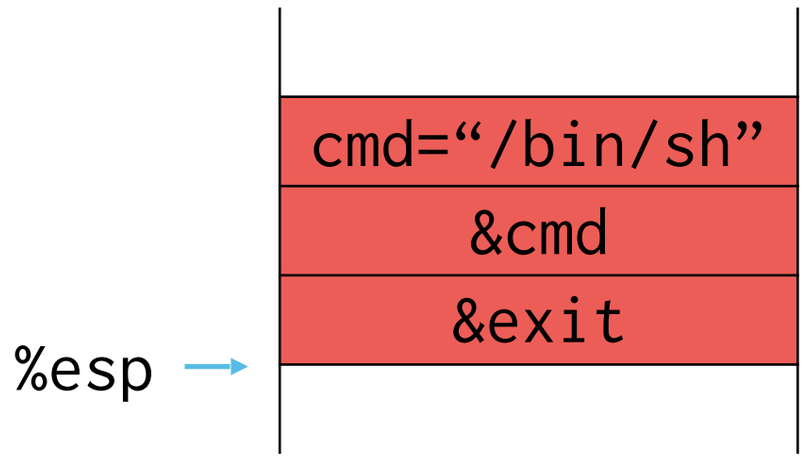
- Our goal: call `system("/bin/sh")`
- Need to set up stack frame that looks like a normal call to system:



- But we're not going to use `call` instruction to jump to system; we're going to use `ret`

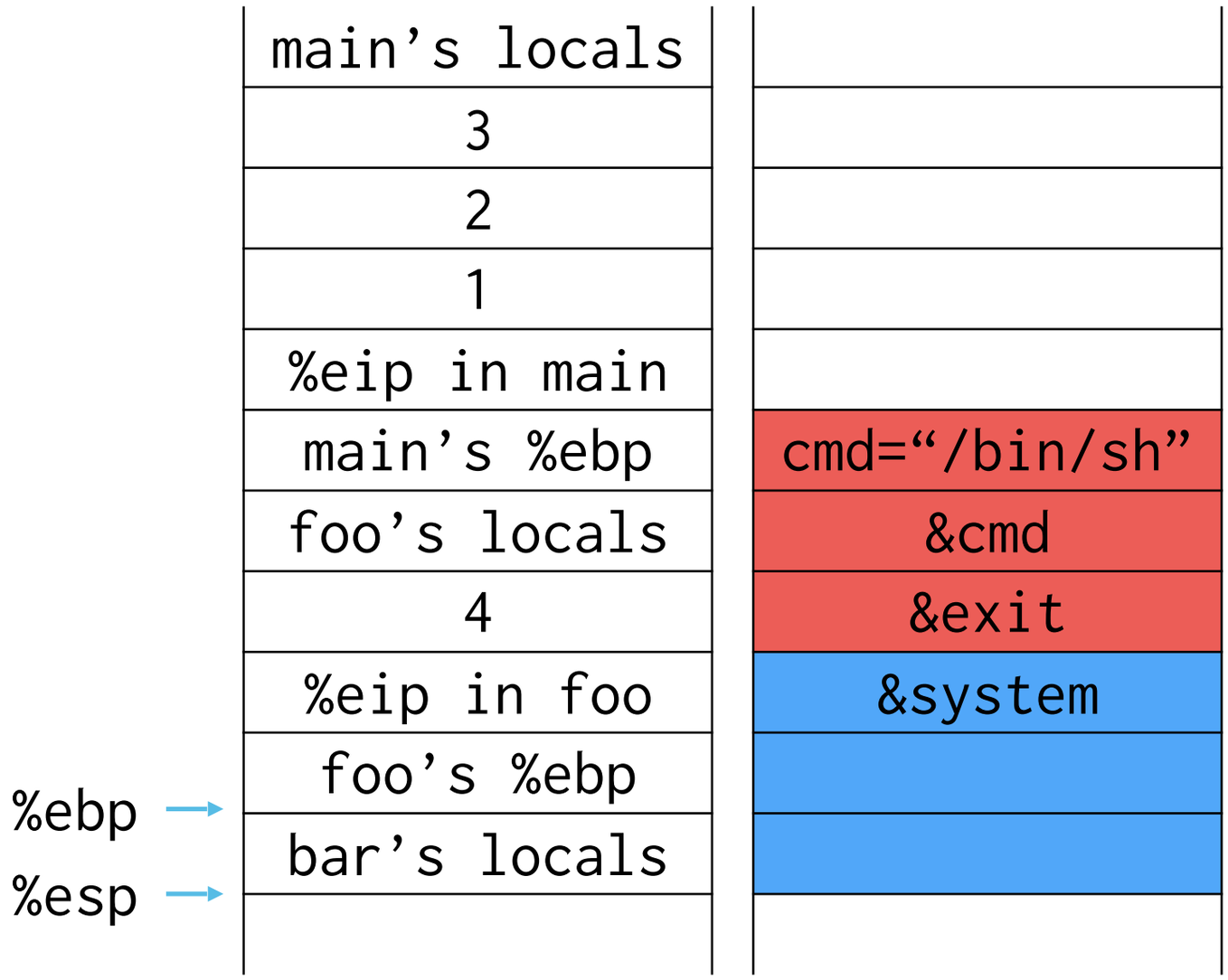
Suppose bar had overflow

- Our goal: call `system("/bin/sh")`
- Need to set up stack frame that looks like a normal call to system:

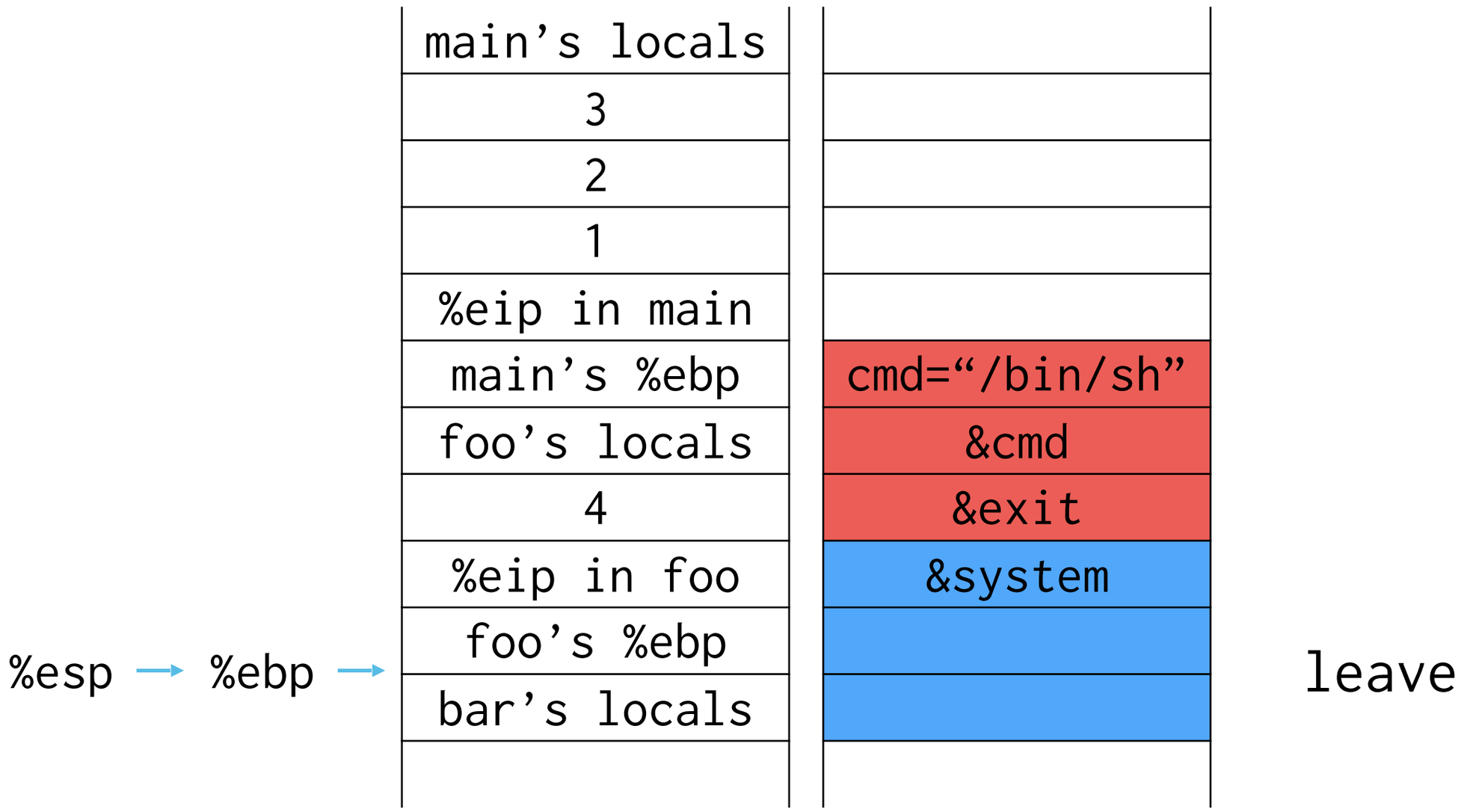


- But we're not going to use `call` instruction to jump to system; we're going to use `ret`

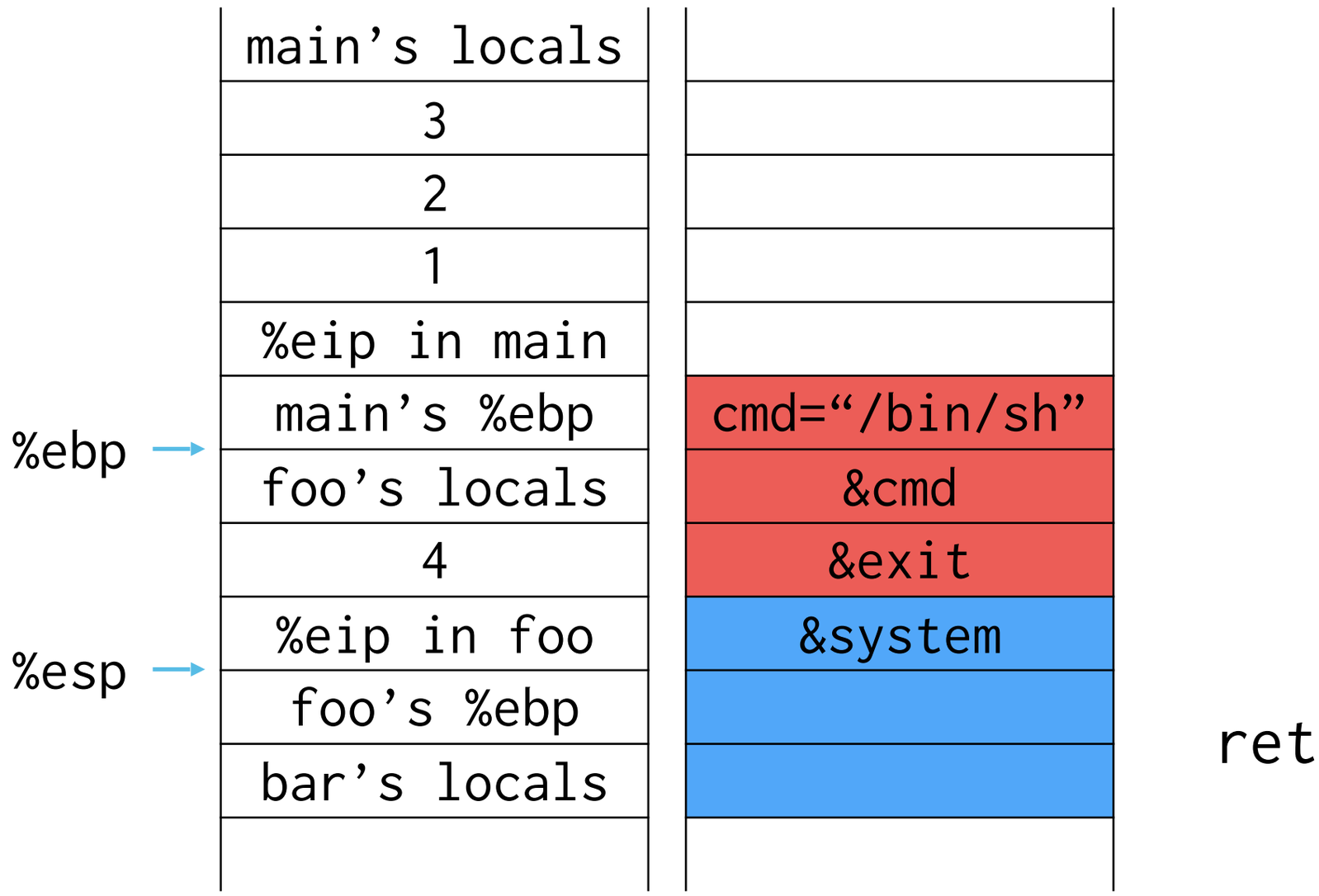
Hijacking control flow



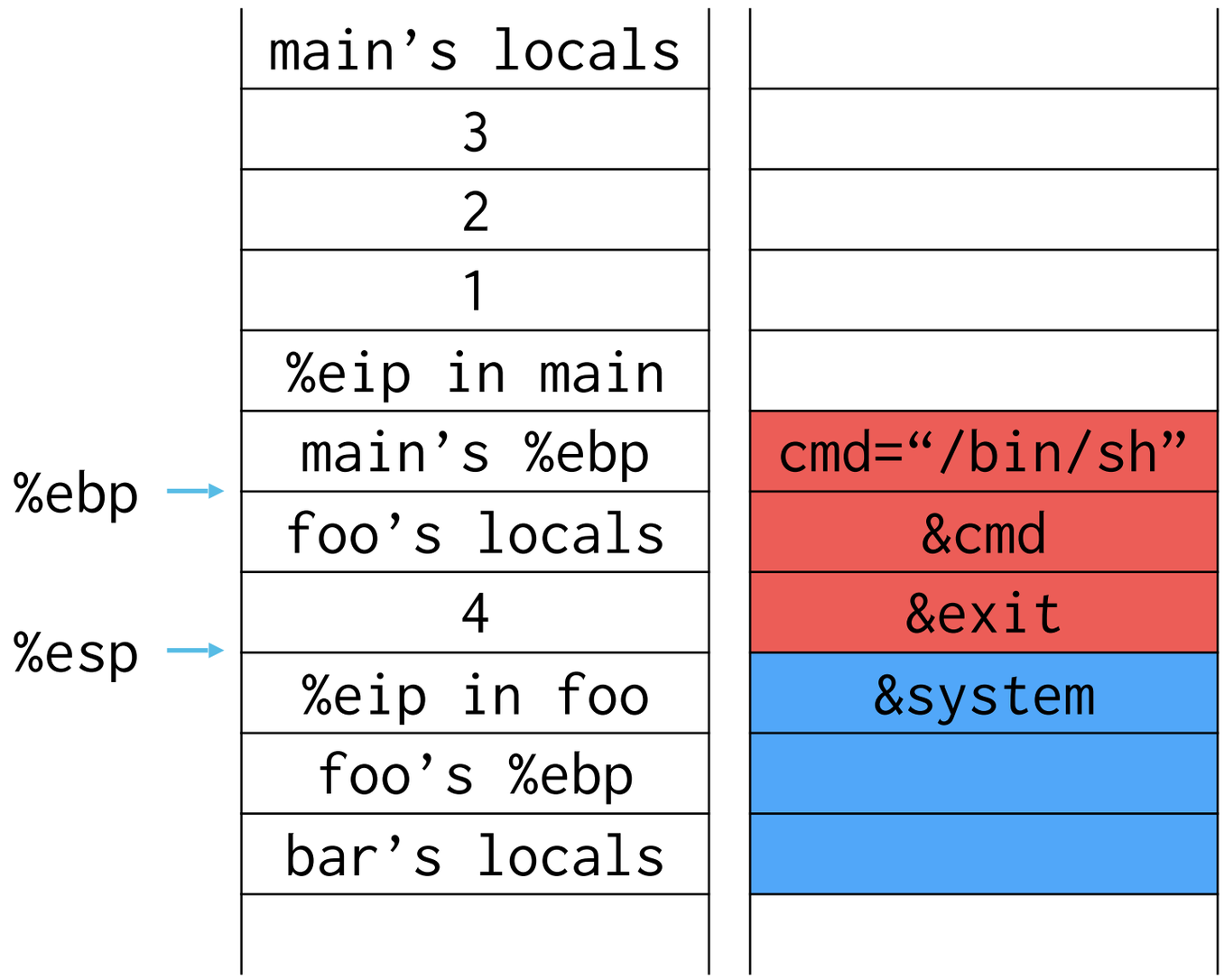
Hijacking control flow



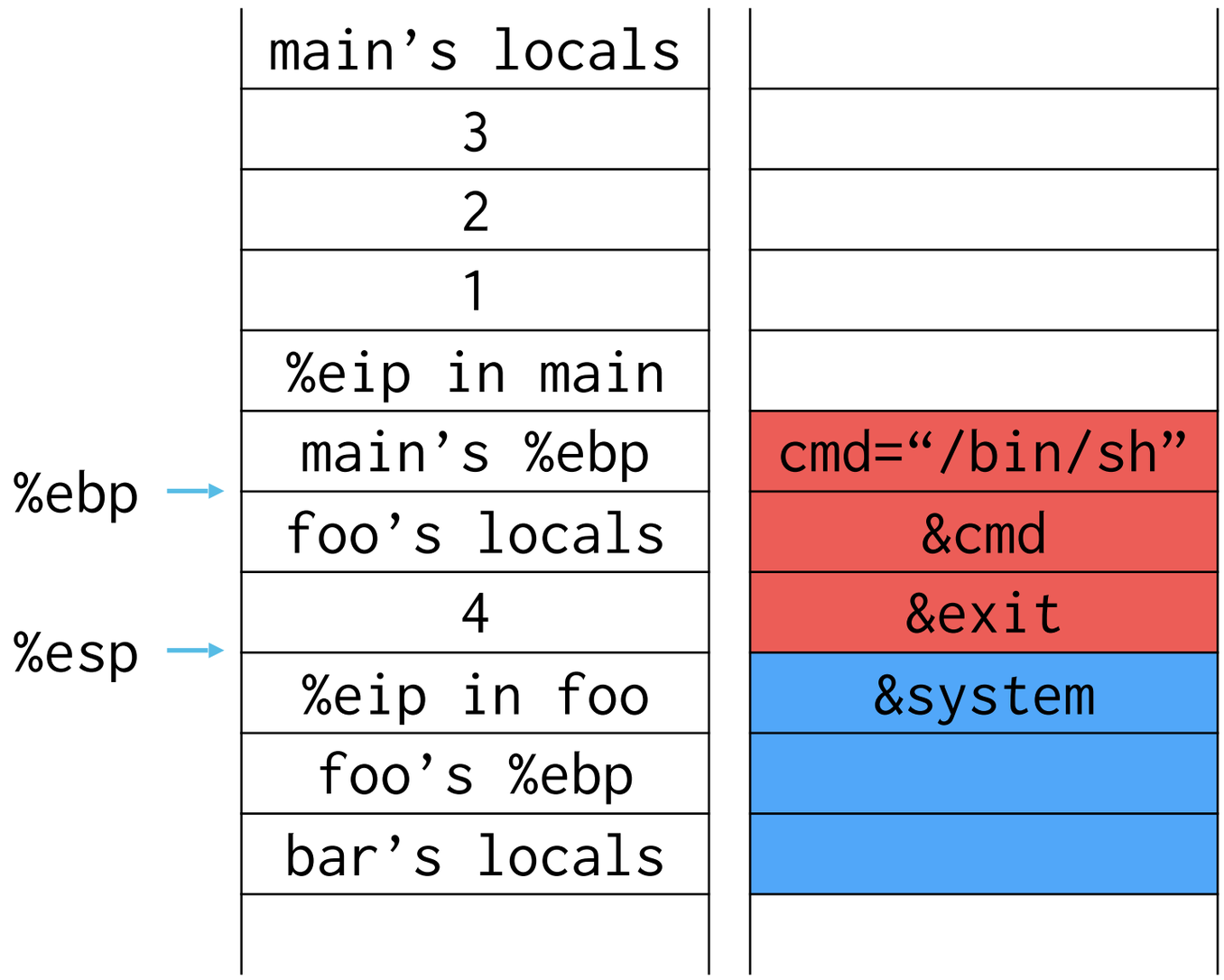
Hijacking control flow



Hijacking control flow

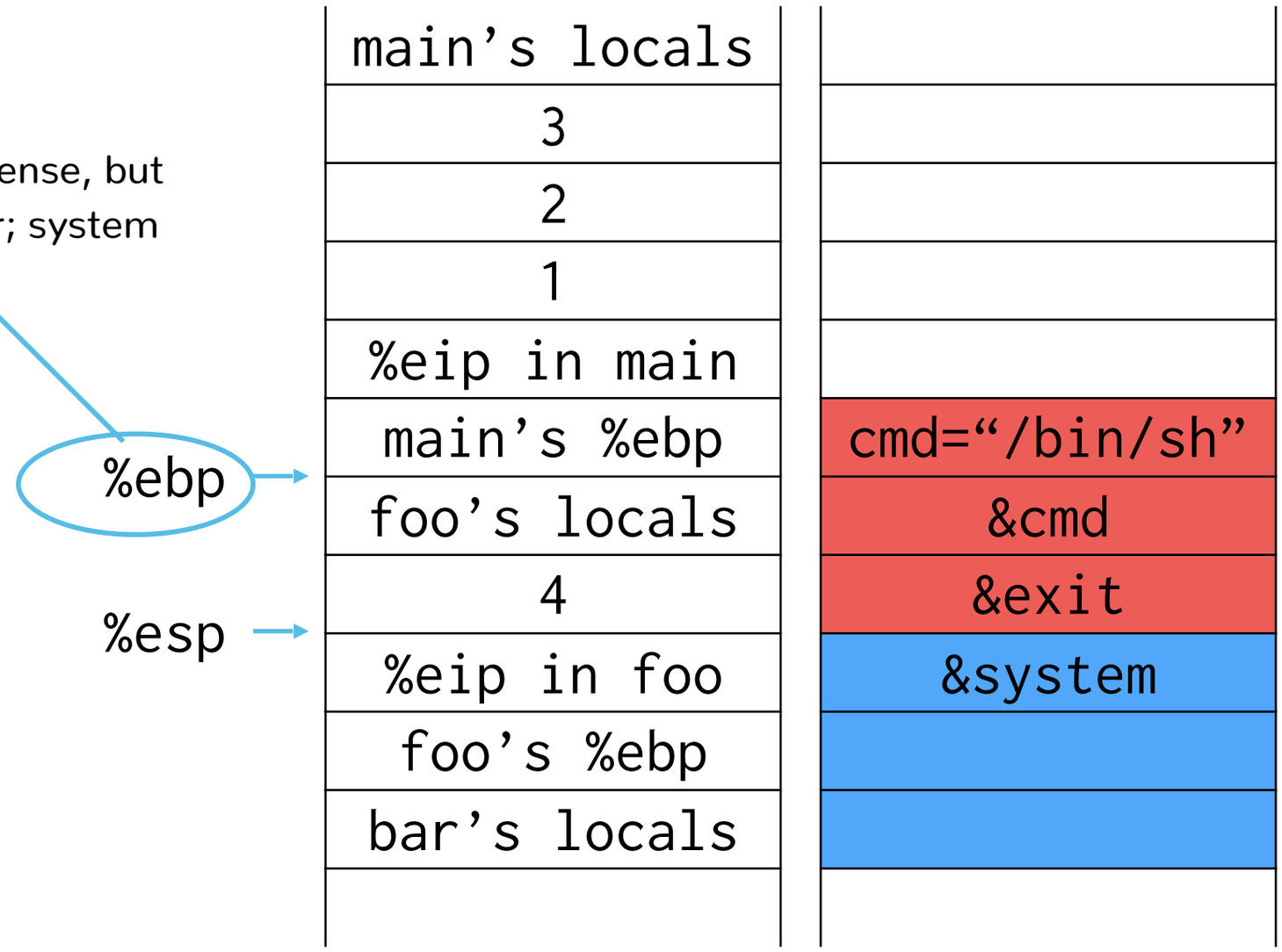


Hijacking control flow



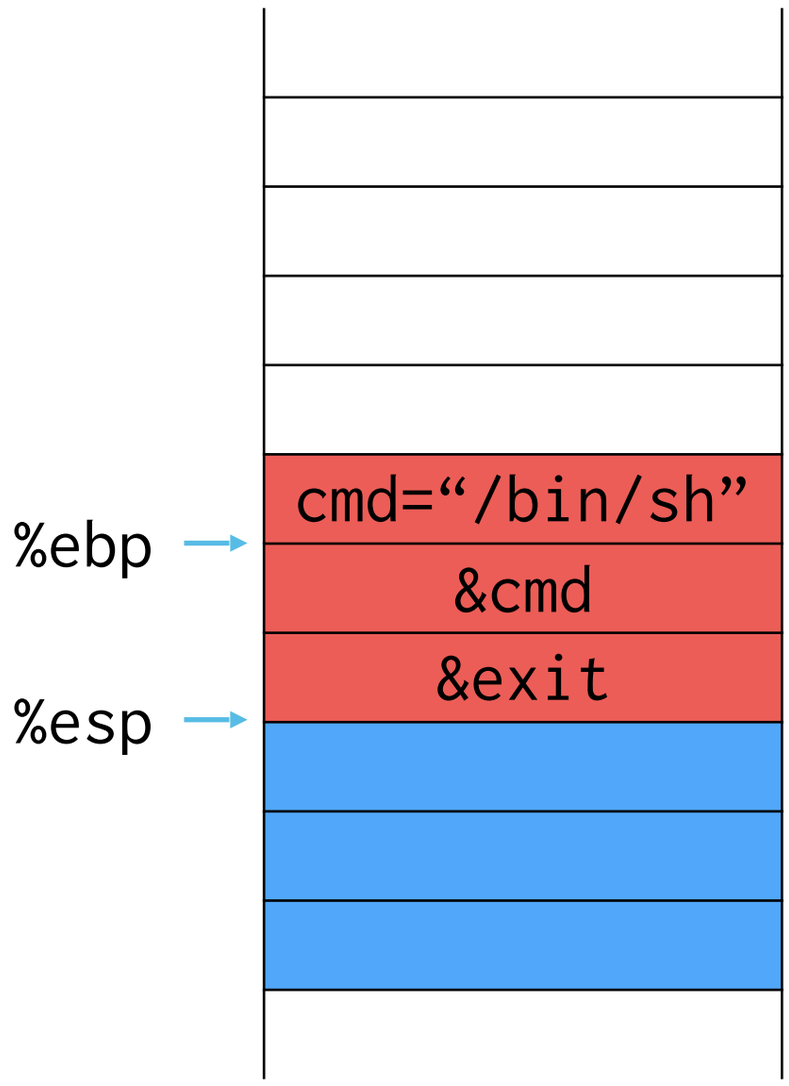
Hijacking control flow

points to nonsense, but doesn't matter; system just saves it



Hijacking control flow

- Stack frame that looks like a normal call to system:



Buffer Overflow Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Memory writable or executable, not both (W^X)
- Address Space Layout Randomization (ASLR)

None are perfect, but in practice they raise the bar dramatically