CSE 127: Computer Security

Buffer overflow defenses

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

Can we just avoid writing C code that has buffer overflow bugs?

Yes! Avoid unsafe functions!

- strcpy, strcat, gets, etc.
- This is a good idea in general...

Yes! Avoid unsafe functions!

- strcpy, strcat, gets, etc.
- This is a good idea in general…
- But…
  - Requires manual code rewrite
  - Non-library functions may be vulnerable
    - E.g. user creates their own strcpy
  - No guarantee you found everything
  - Alternatives are also error-prone!

Even printf is tricky

If `buf` is under control of attacker

is: `printf(buf)` safe?

Even printf is tricky

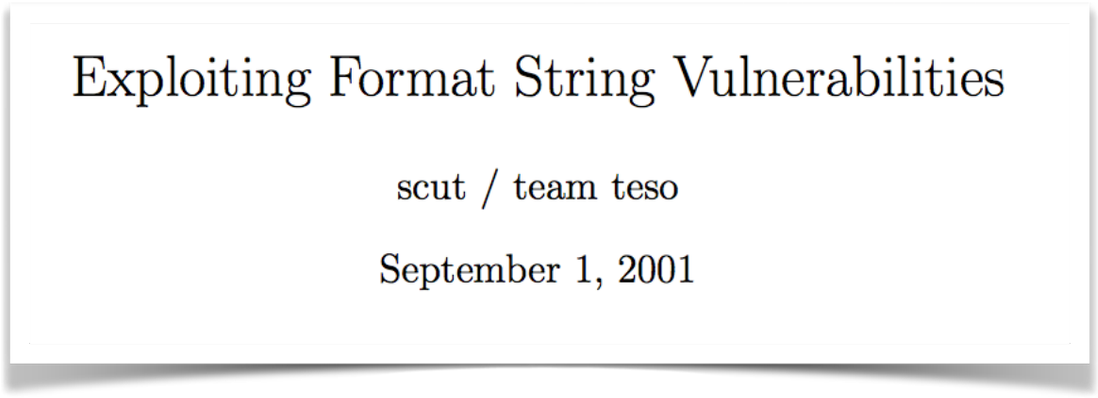
If `buf` is under control of attacker is: `printf("%s\n", buf)` safe?

Even printf is tricky

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

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

If we can’t avoid writing buggy C code...
can we mitigate their exploitation?

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]

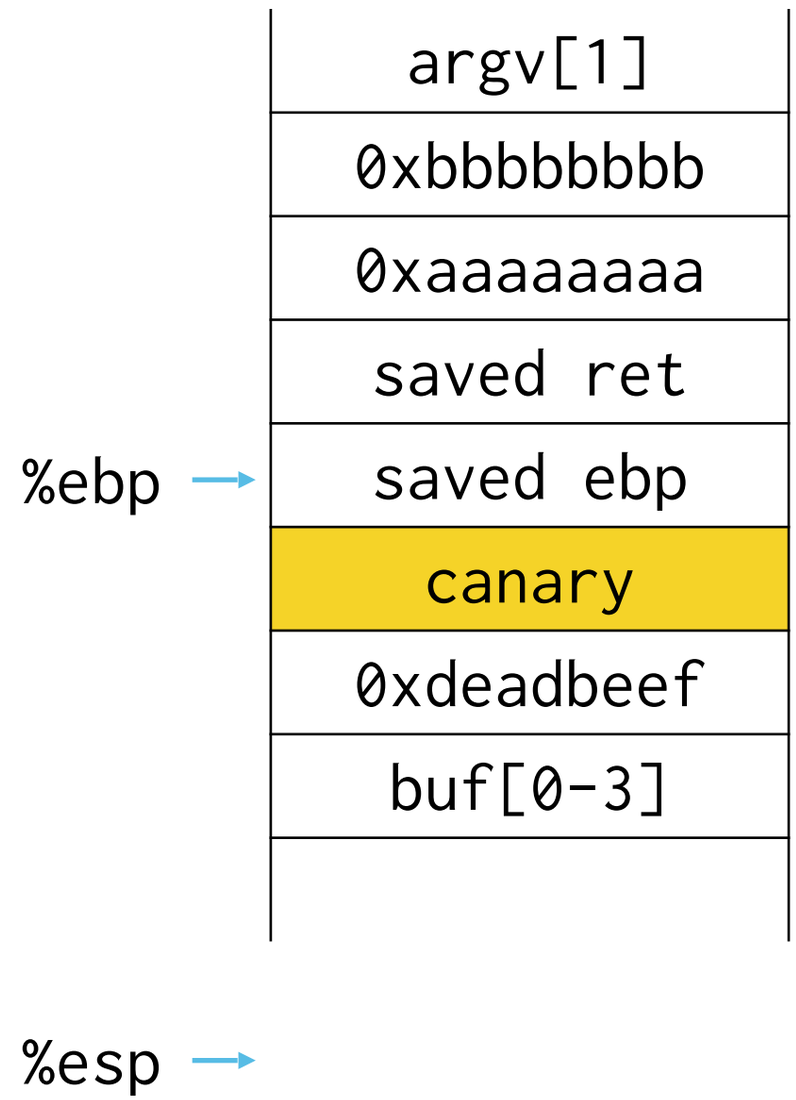
The phrase "canary in a coal mine" is frequently used to refer to a person or thing which serves as an early warning of a coming crisis. By analogy, the term "climate canary" is used to refer to a species (called an indicator species) that is affected by an environmental danger prior to other species,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

Stack canaries

- Prevent control flow hijacking by detecting 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
```

Can we defeat canaries?

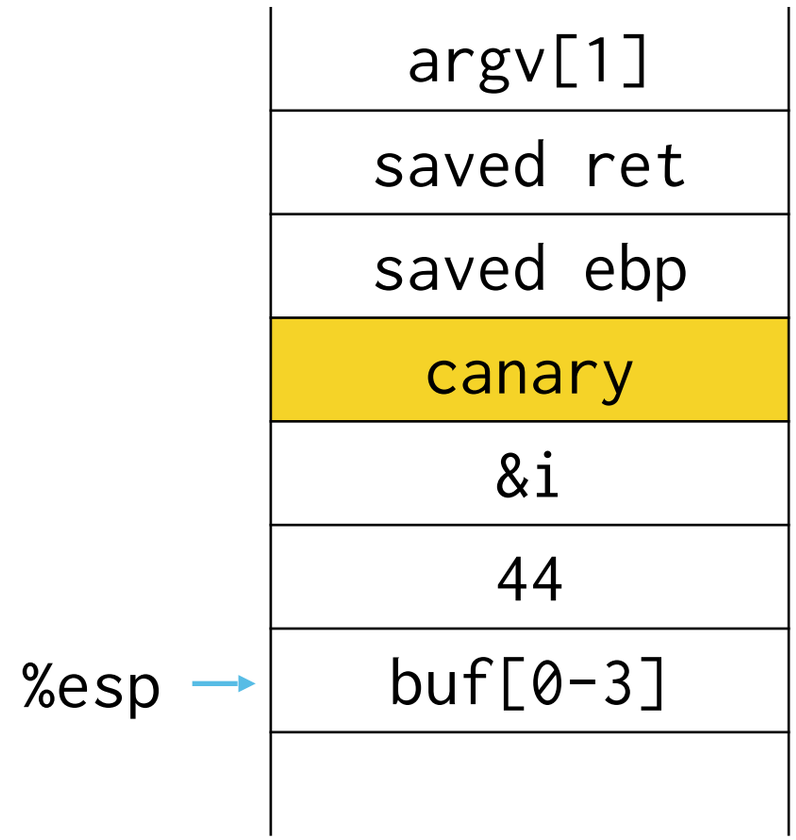
- **Assumption:** impossible to subvert control flow without corrupting the canary
- Think outside the box

Can we defeat canaries?

- **Assumption:** impossible to subvert control flow without corrupting the canary
- Think outside the box
  - Overwrite function pointer elsewhere on the stack/heap
  - Pointer subterfuge
  - memcpy buffer overflow with fixed canary
  - Learn the canary

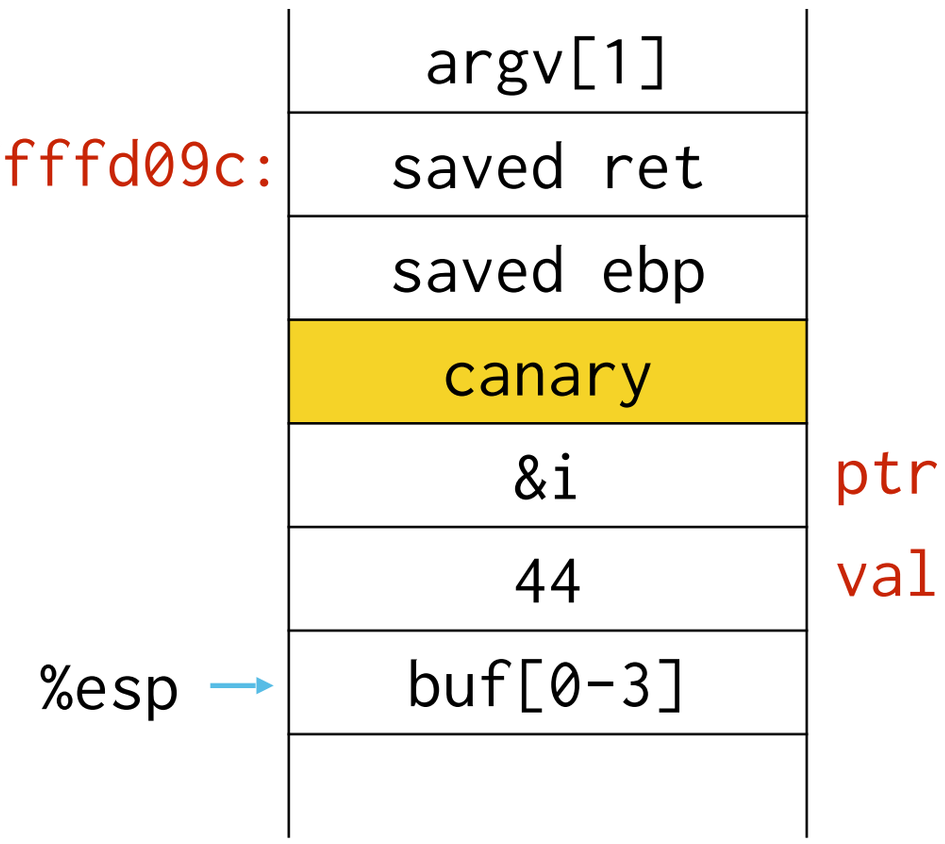
Pointer subterfuge

```
#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

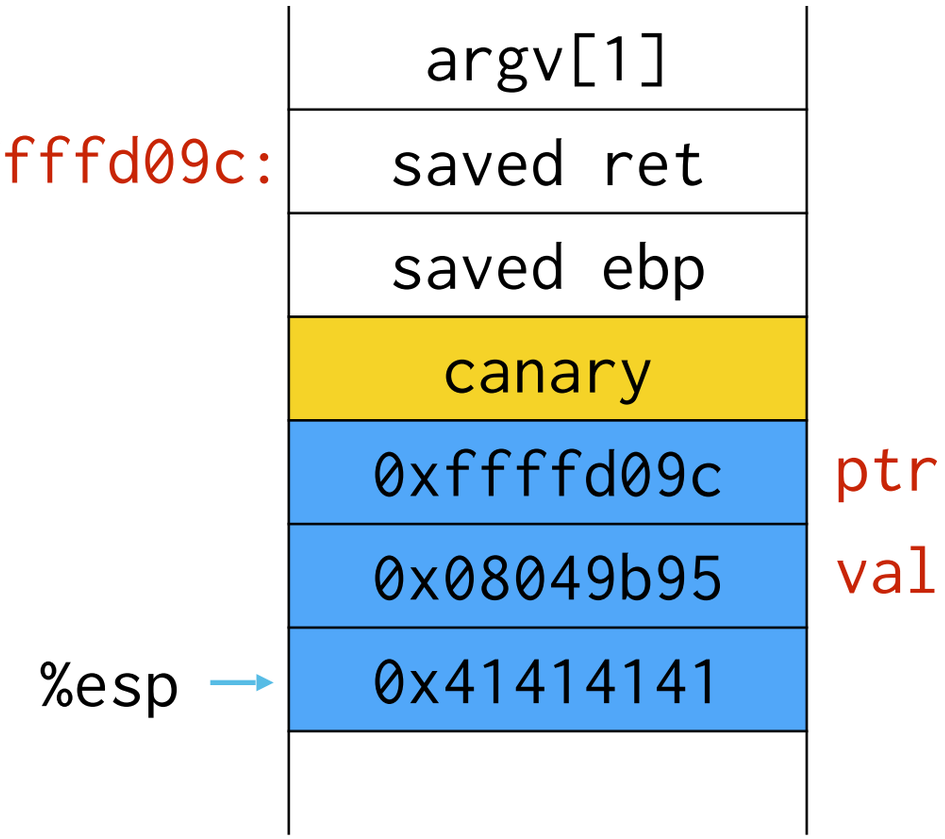
```
#include <stdio.h>
#include <string.h>

0x08049b95:  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;
}
```

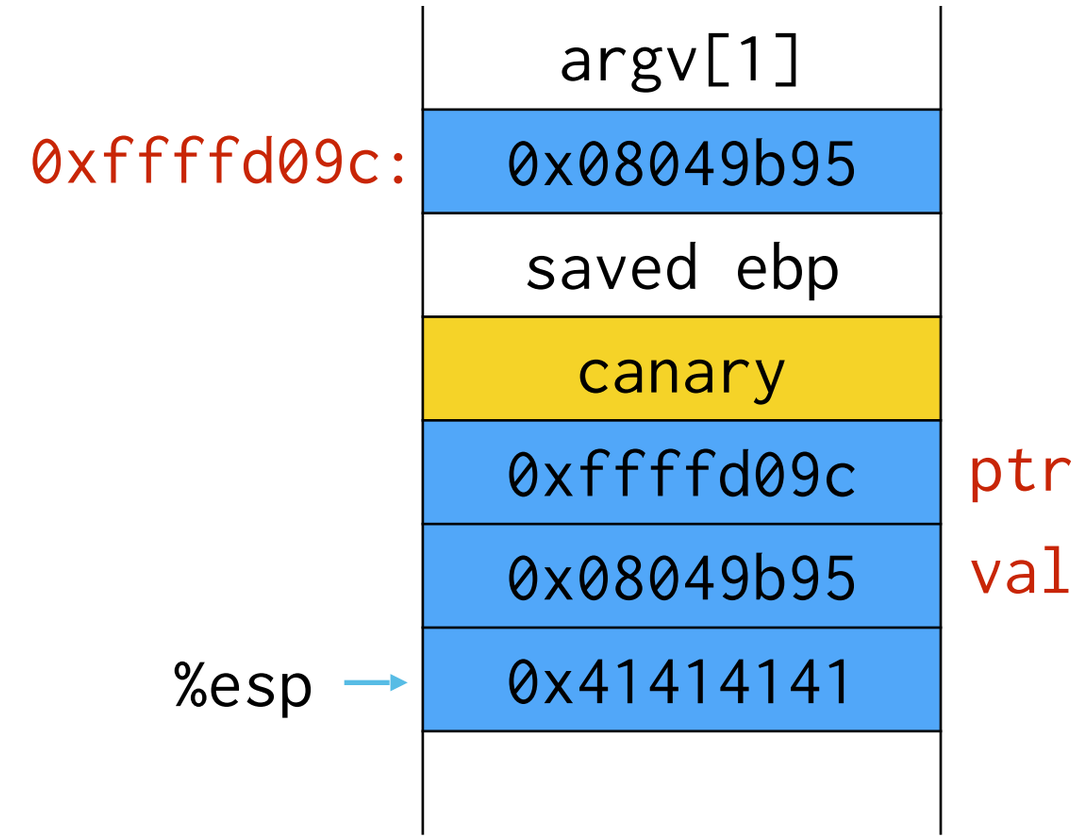
0x08049b95:



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;
}
```

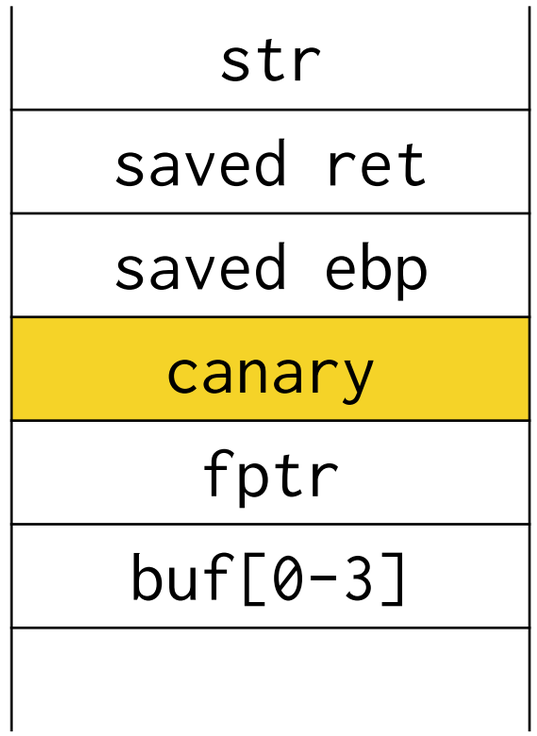


Overwrite function pointer on stack

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

Overwrite function pointer on stack

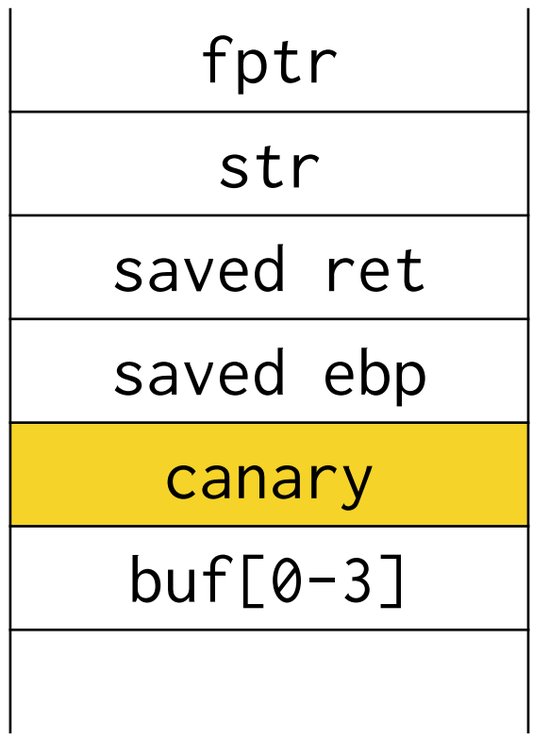
```
void func(char *str) {
  void (*fptr)() = &bar;
  char buf[4];
  strcpy(buf,str);
  fptr()
}
```



Or a function pointer argument

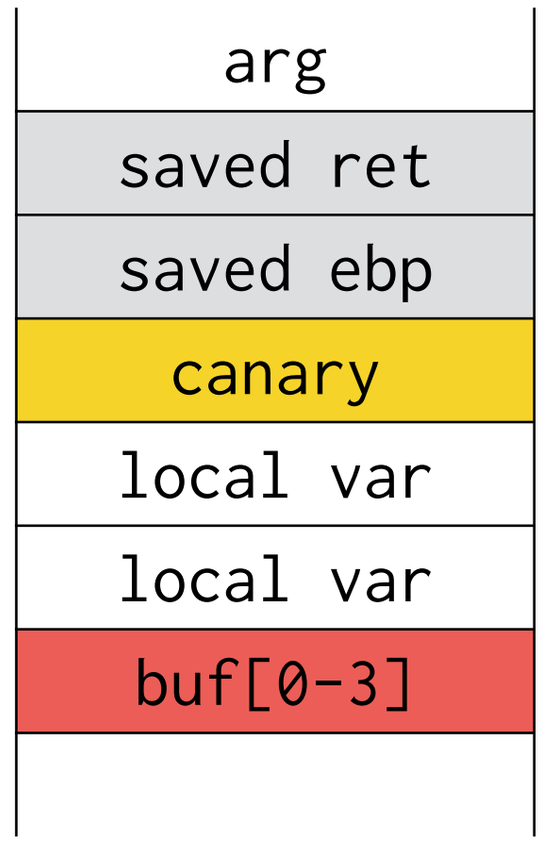
Or a function pointer argument

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



What can we do about this?

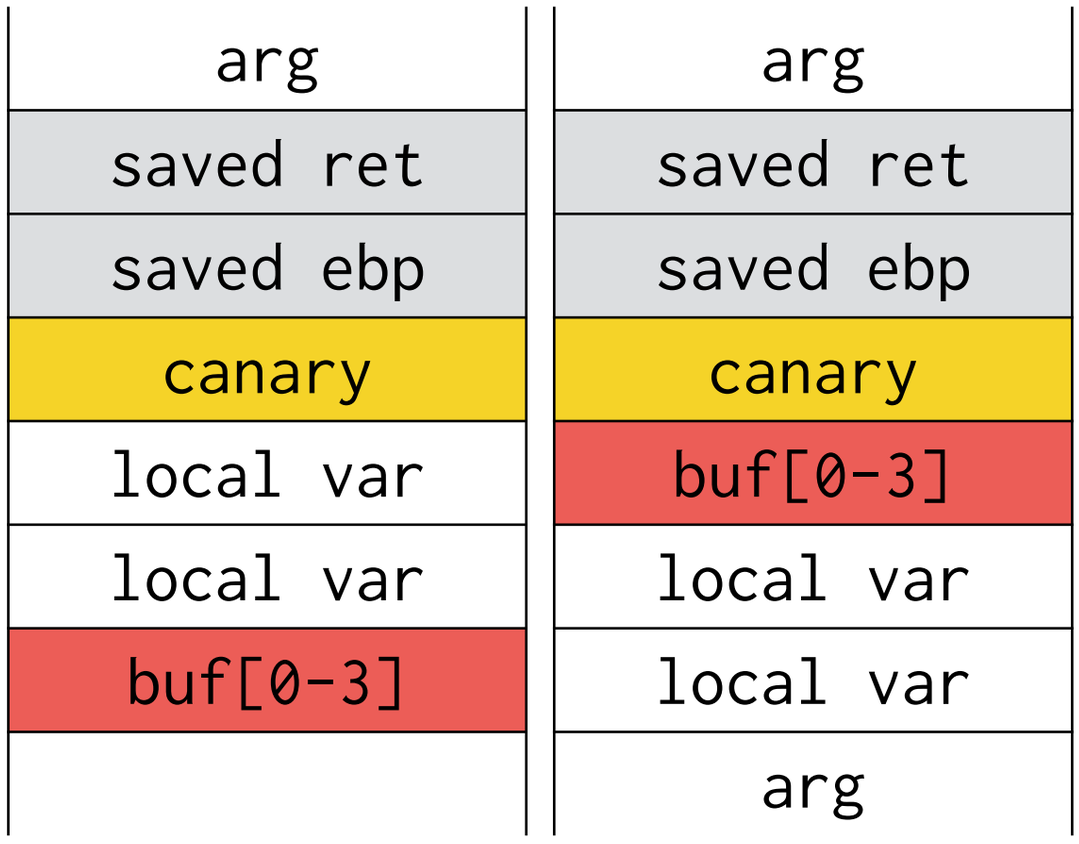
- **Problem:** Overflowing locals and arguments can allow attacker to hijack control flow



What can we do about this?

- **Problem:** Overflowing locals and arguments can allow attacker to hijack control flow

- **Solution:**
  - Move buffers closer to canaries vs. lexical order
  - Copy args to top of stack



Your compiler does this already

Your compiler does this already

-fstack-protector

- Functions with char bufs ≥ ssp-buffer-size (default=8)
- Functions with variable sized `alloca()`s

Your compiler does this already

-fstack-protector

- Functions with char bufs ≥ ssp-buffer-size (default=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

Your compiler does this already

-fstack-protector

- Functions with char bufs ≥ ssp-buffer-size (default=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!

If we zoom in...

```
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
```

If we zoom in...

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
```

If we zoom in...

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
```

If we zoom in...

copy arg1

copy arg2

copy arg3

```
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
```

If we zoom in…

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
```

Can we defeat canaries?

- **Assumption:** impossible to subvert control flow without corrupting the canary
- Think outside the box
  - Overwrite function pointer elsewhere on the stack/heap
  - Pointer subterfuge
  - memcpy buffer overflow with fixed canary
  - Learn the canary

How do we pick canaries?

- Pick a clever value!
  - E.g., `0x000d0aff` (0, CR, NL, -1) to terminate string ops like `strcpy` and `gets`
  - Even if attacker knows value, can't overwrite past canary!

Not all overflows are due to strings

Many other functions handle buffers

- E.g., memcpy, memmove, read
- These are also error-prone!

```
void func(char *str) {
  char buf[1024];
  memcpy(buf,str, strlen(str));
}
```

How do we pick canaries?

- Pick a random value!
  - When?

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

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
  - E.g.,

CVE-2020-15999: FreeType Heap Buffer Overflow in Load_SBit_Png

Sergei Glazunov, Project Zero (Originally posted on Project Zero blog 2021-02-04)

The Basics

Disclosure or Patch Date: 19 October 2020

Product: Google Chrome/ Freetype

• • •

The vulnerability was used by the actor in two exploit chains:

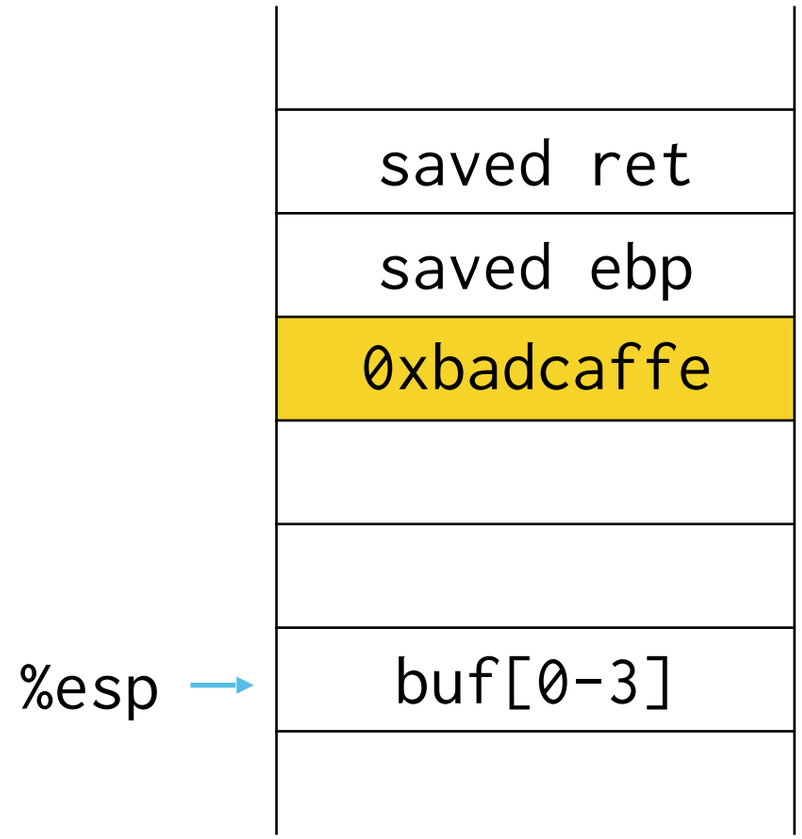
- together with a OS kernel issue (CVE-2020-17087) on Windows,
- together with a Chrome-specific UAF (CVE-2020-16010) in the browser process on Android.

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

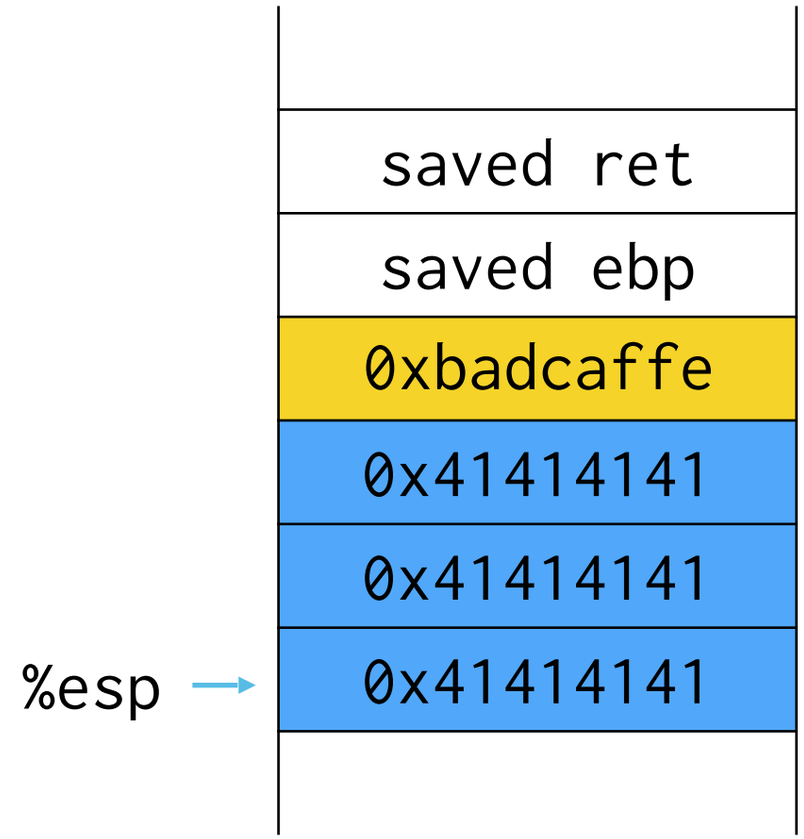
Perfect for brute forcing

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



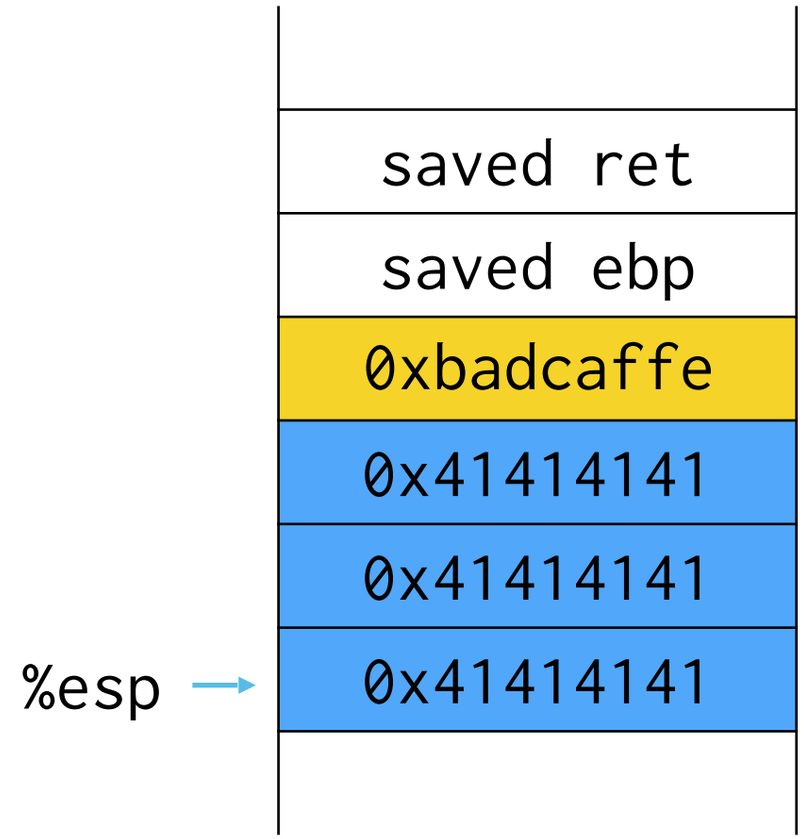
Perfect for brute forcing

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



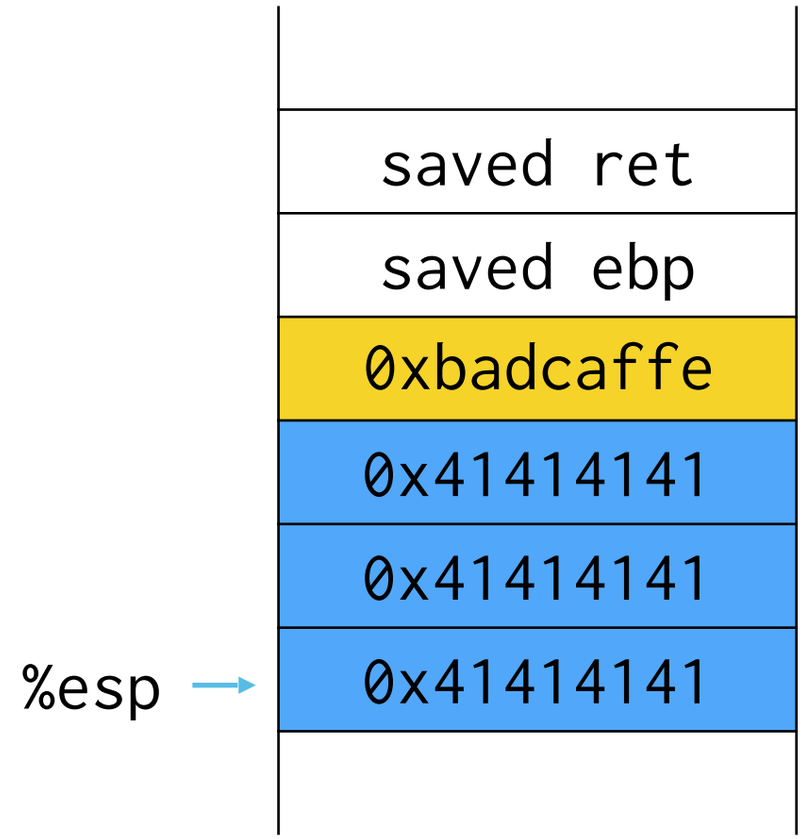
Perfect for brute forcing

- 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

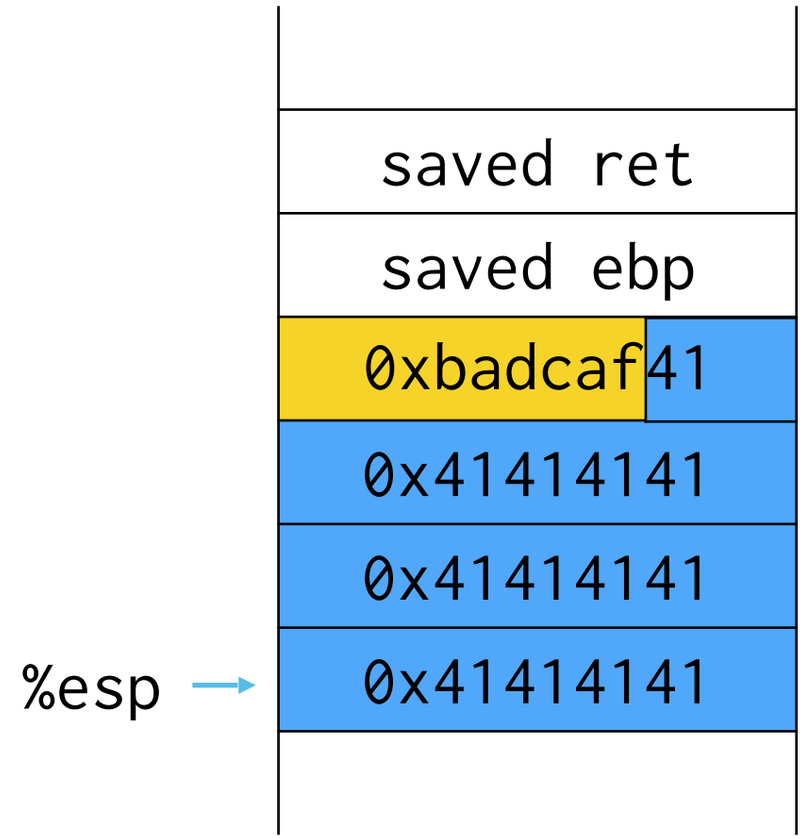
- Forked process has same memory layout and contents as parent, including canary values!
- The fork on crash lets us try different canary values



Figured out size of frame!

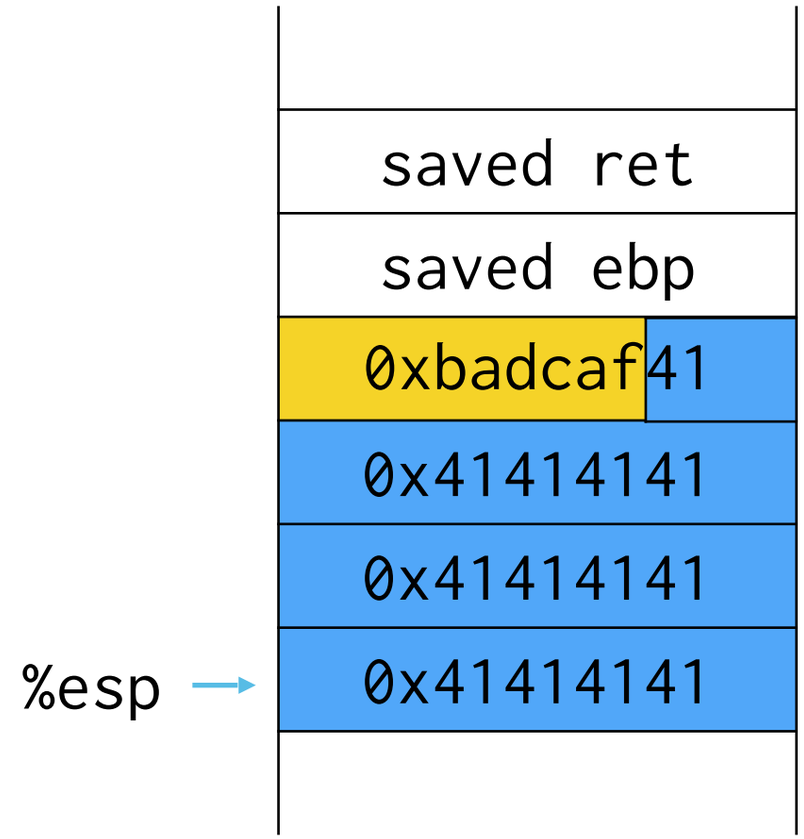
Perfect for brute forcing

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



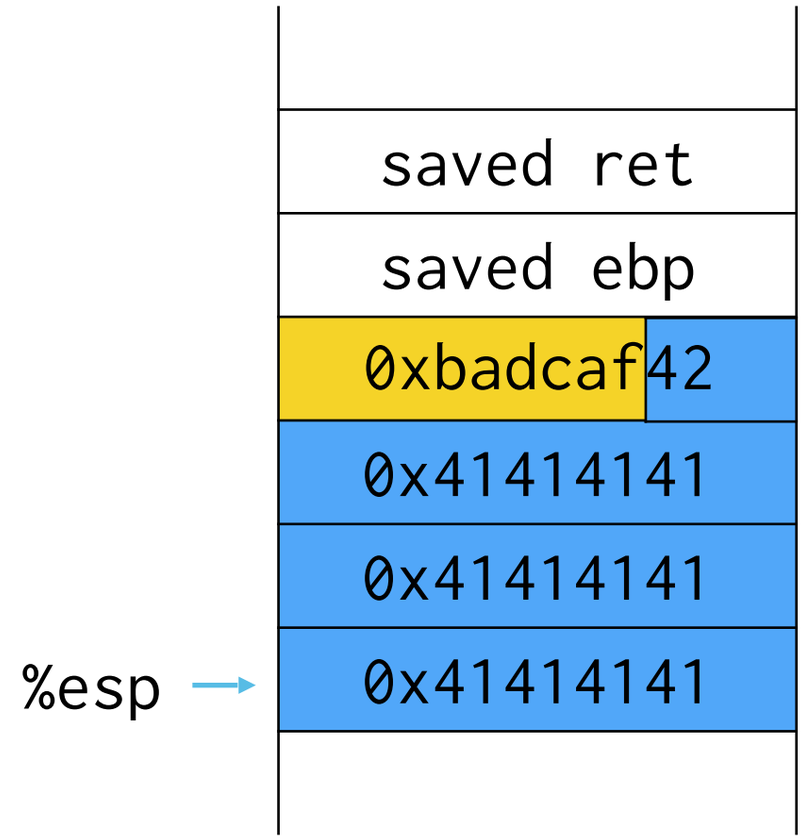
Perfect for brute forcing

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



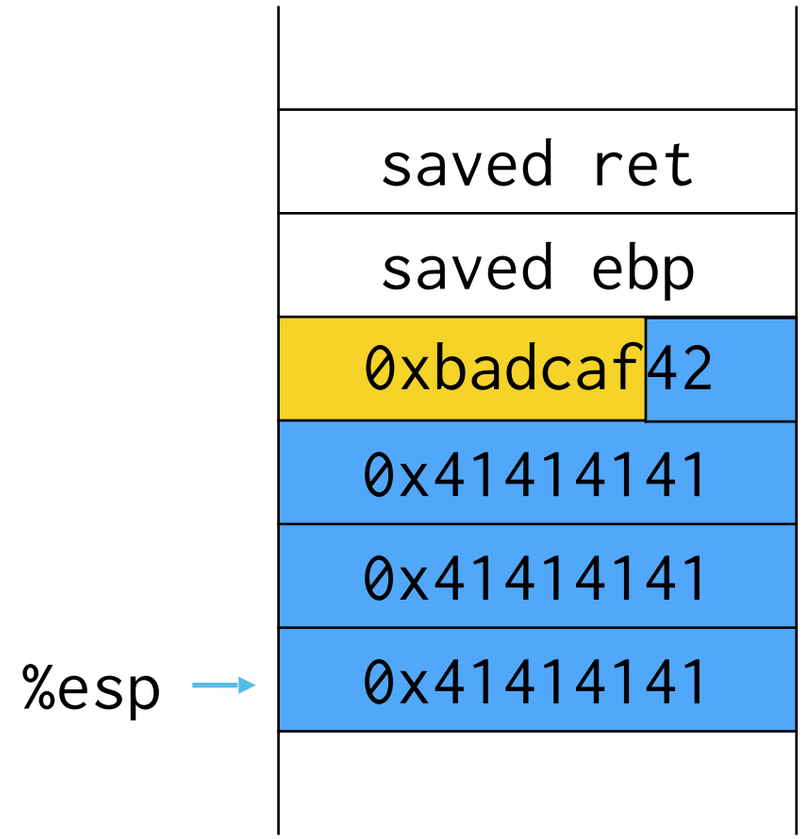
Perfect for brute forcing

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



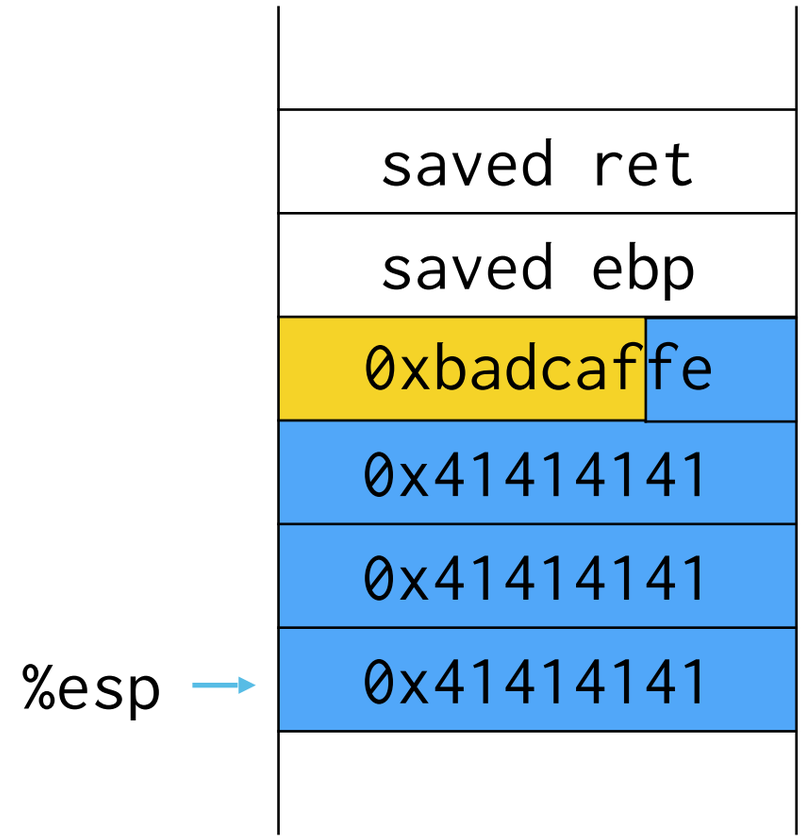
Perfect for brute forcing

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



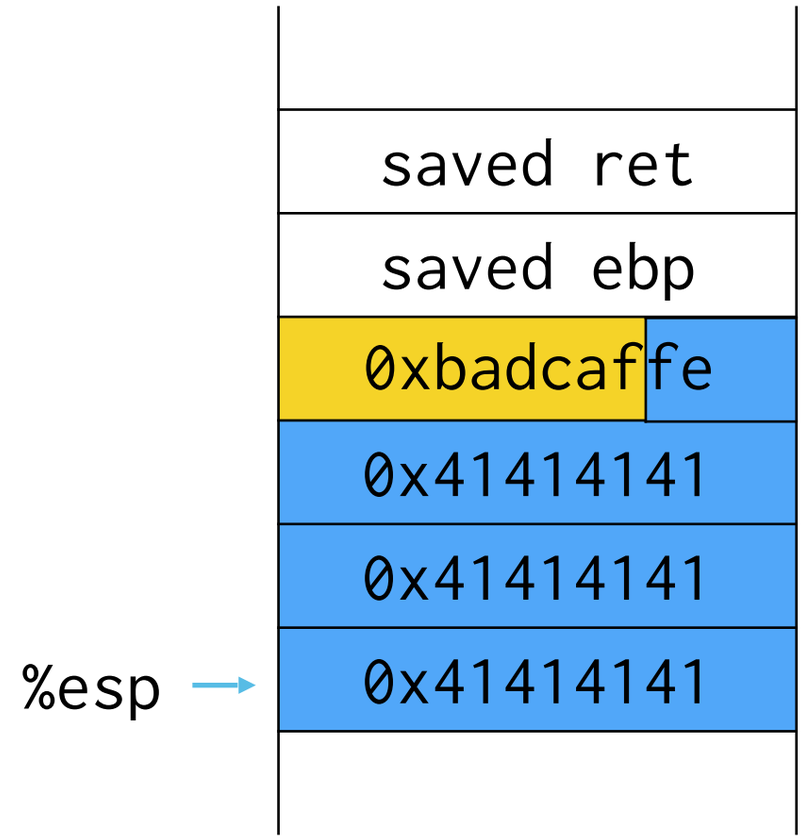
Perfect for brute forcing

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



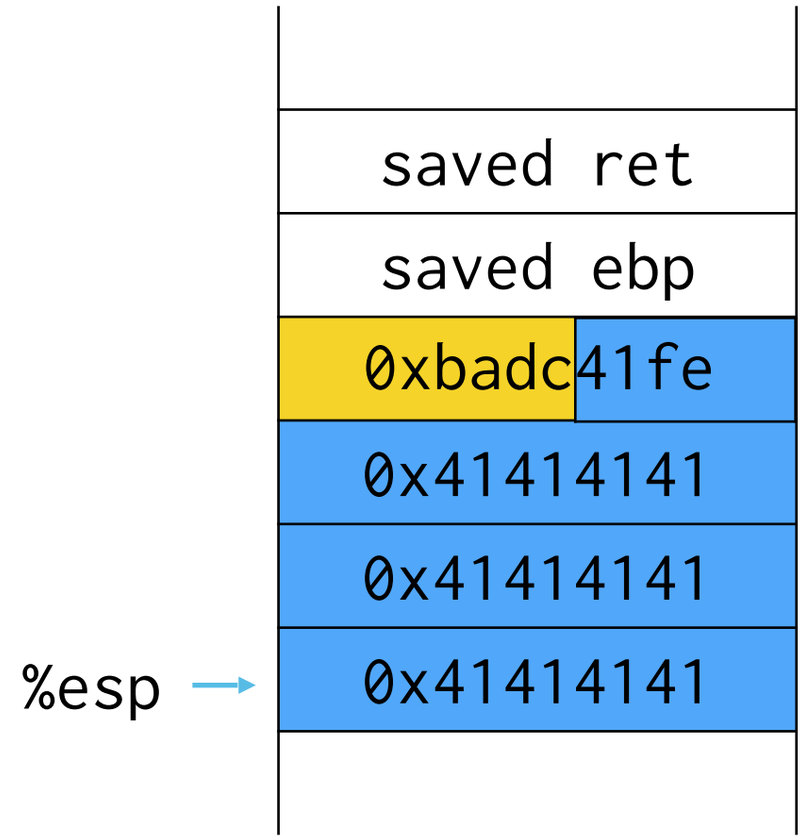
Perfect for brute forcing

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



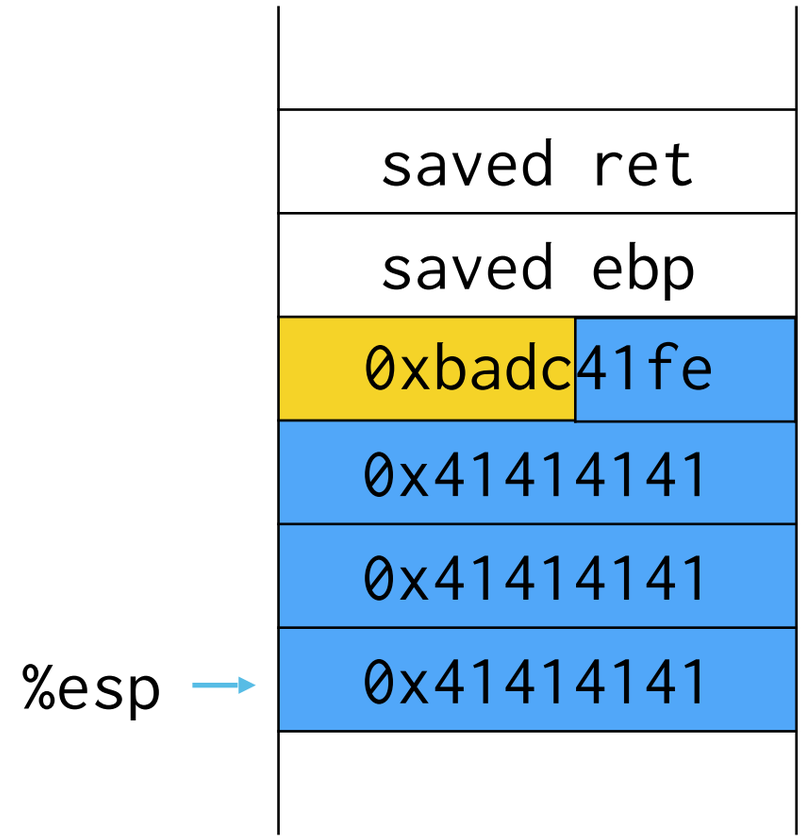
Perfect for brute forcing

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



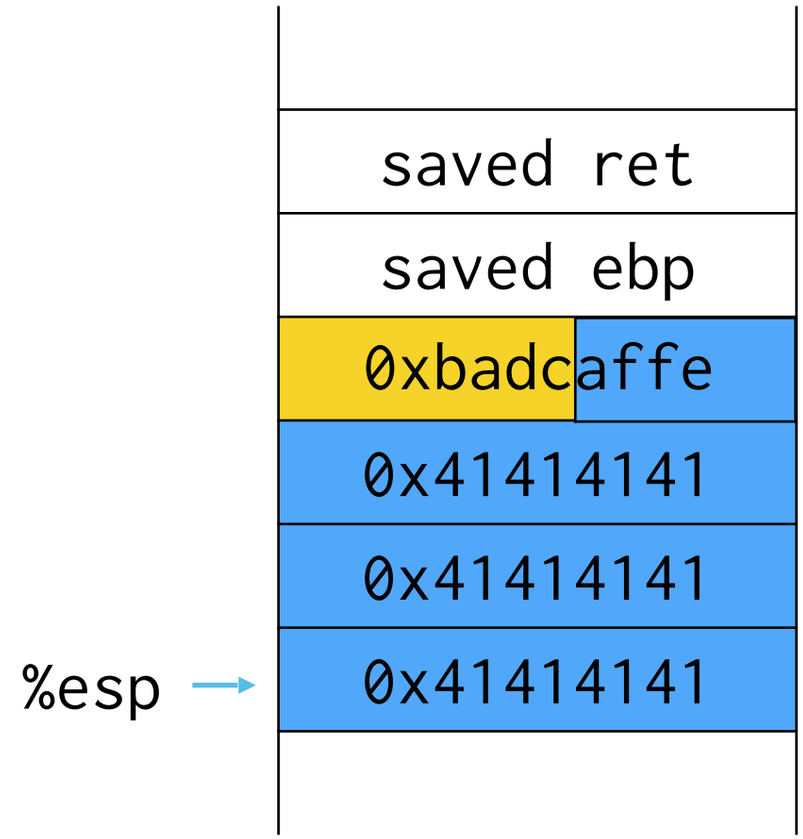
Perfect for brute forcing

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



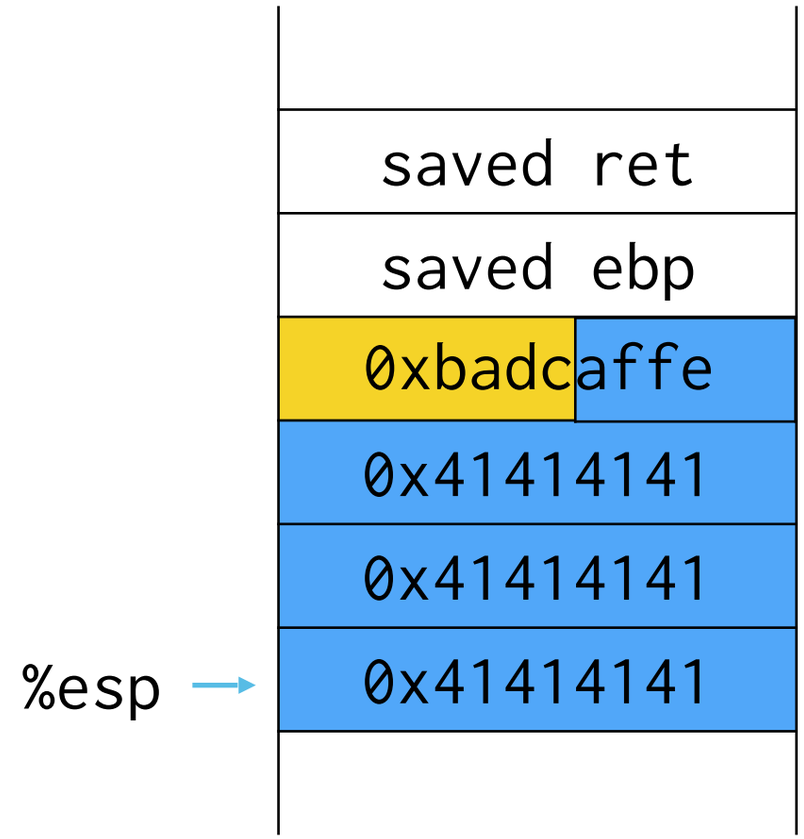
Perfect for brute forcing

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



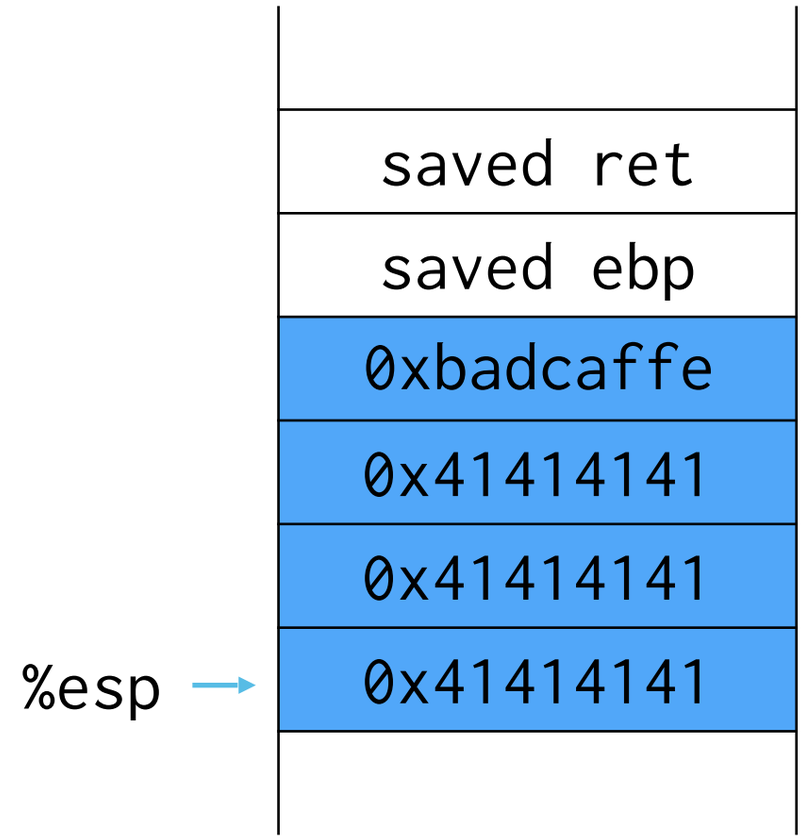
Perfect for brute forcing

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



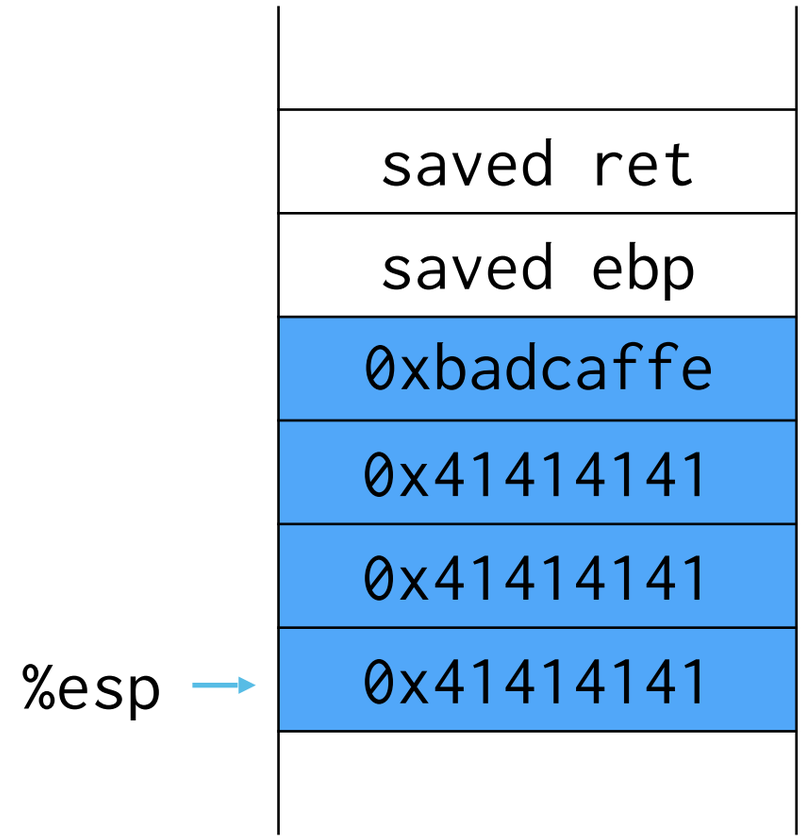
Perfect for brute forcing

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



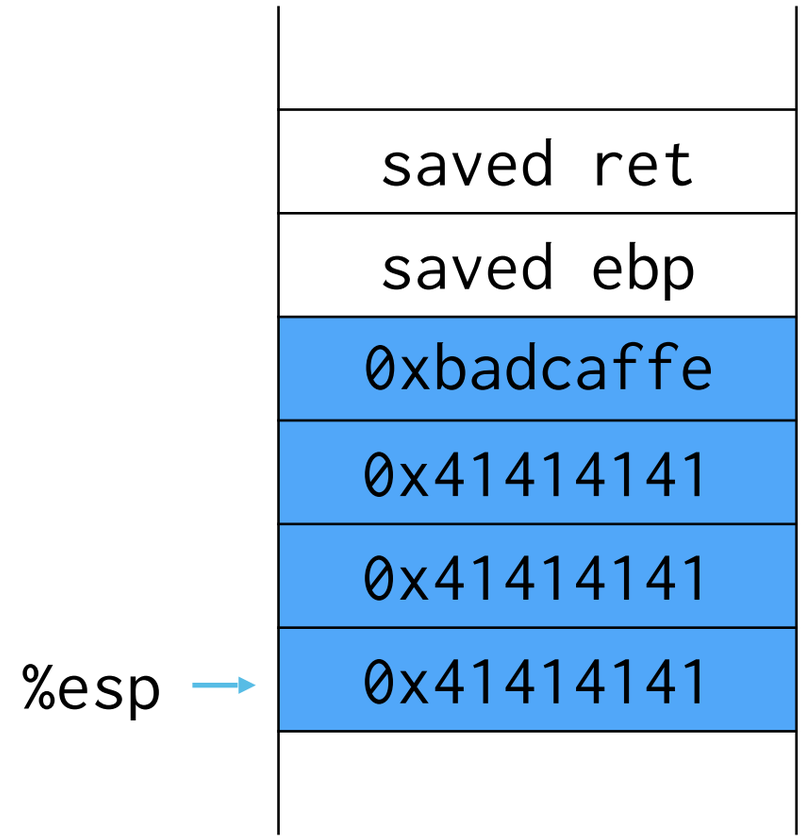
Perfect for brute forcing

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



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



Figured out the canary!

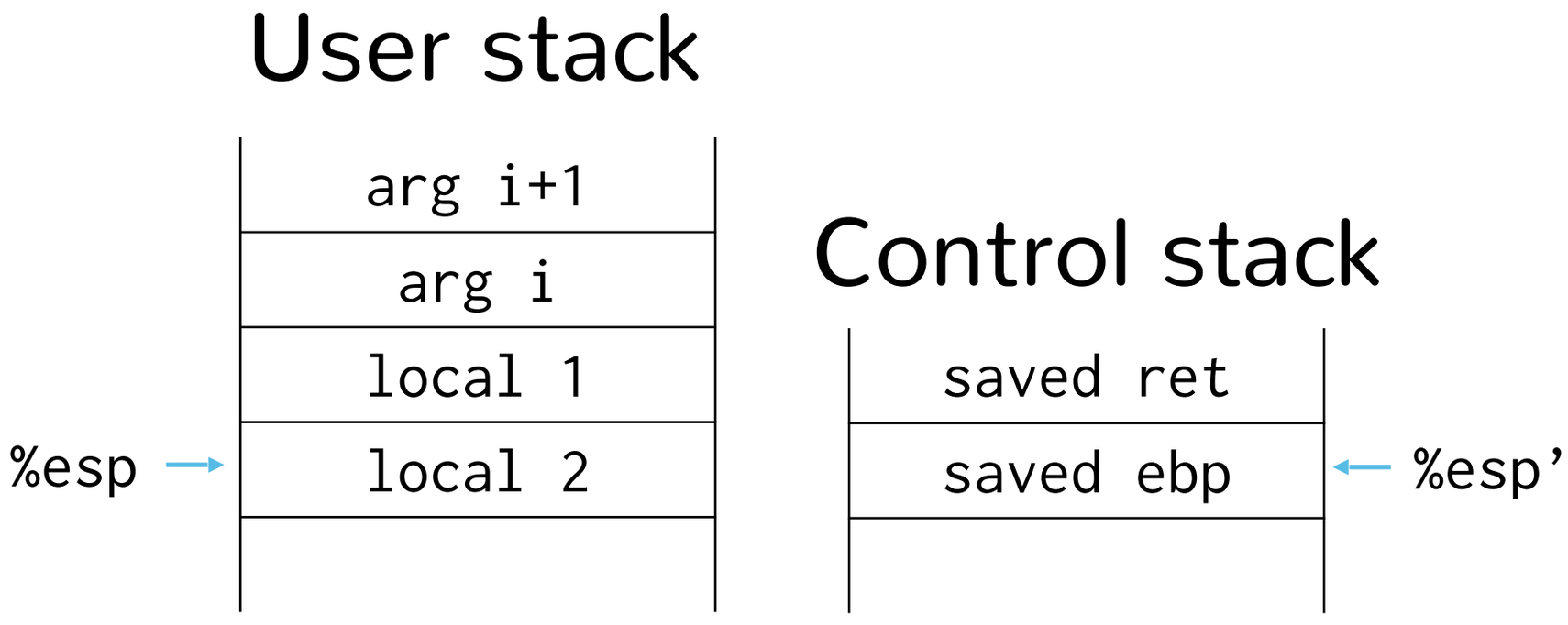
Buffer overflow mitigations

- 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: Control data is stored next to data

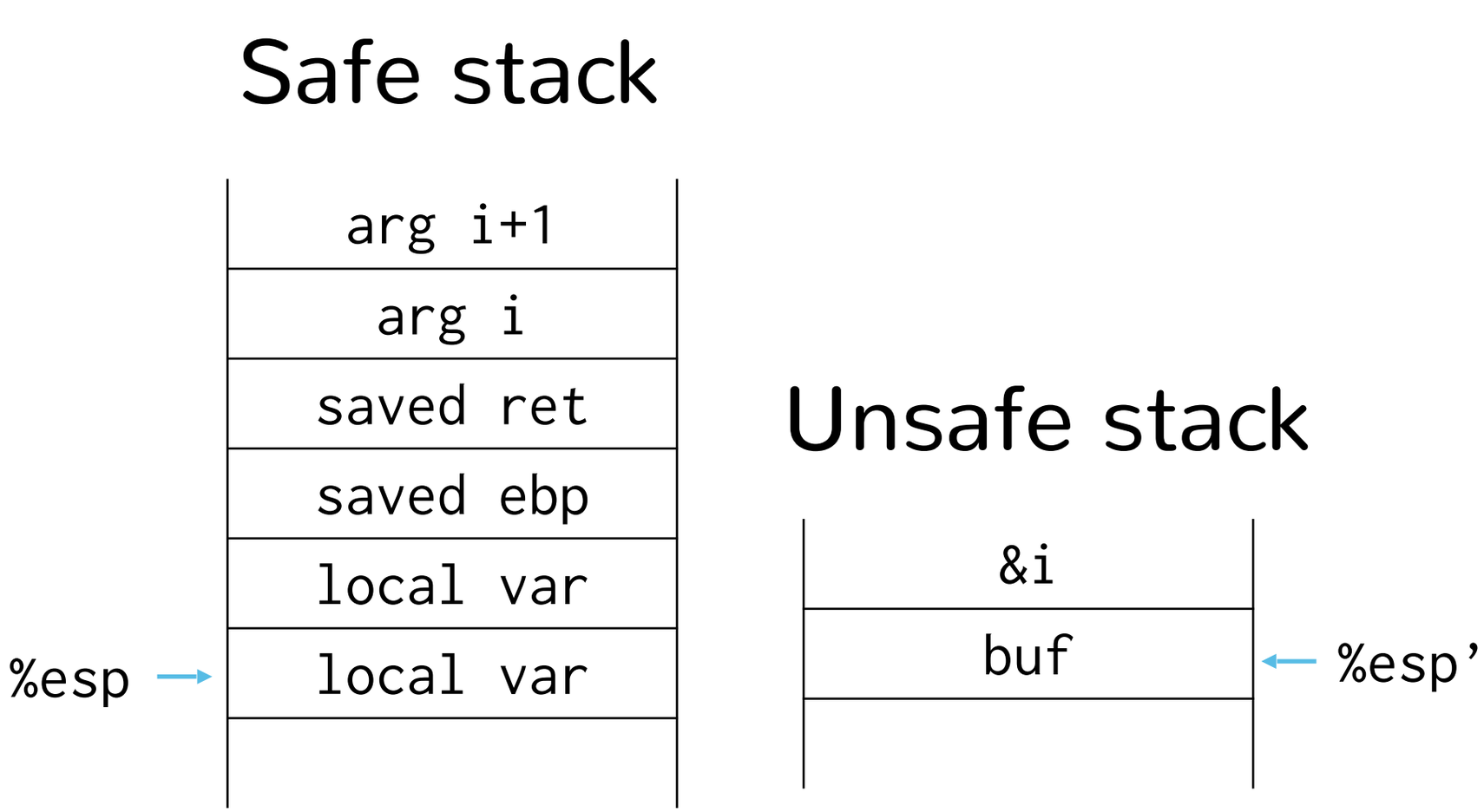
Solution: Bridge the implementation and abstraction gap: separate the control stack



Safe stack

Problem: Unsafe data structures stored next to control

Solution: Move unsafe data structures to separate 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
  - Location of control/stack stack is secret

How do we defeat this?

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

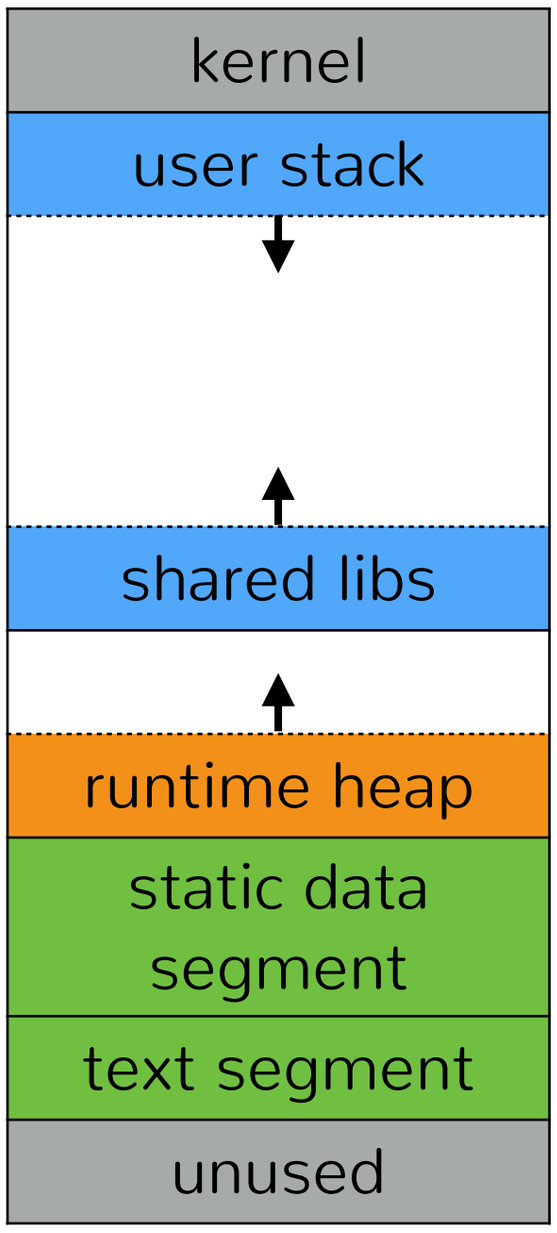
Buffer overflow mitigations

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

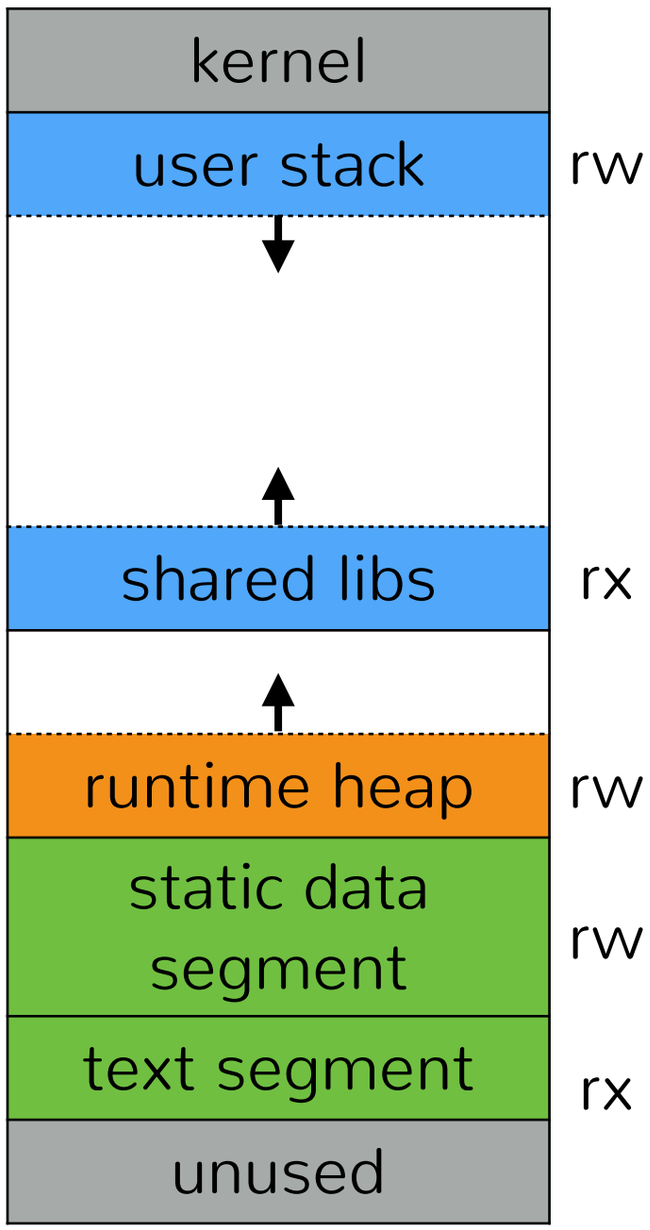
W^X: write XOR execute

- **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

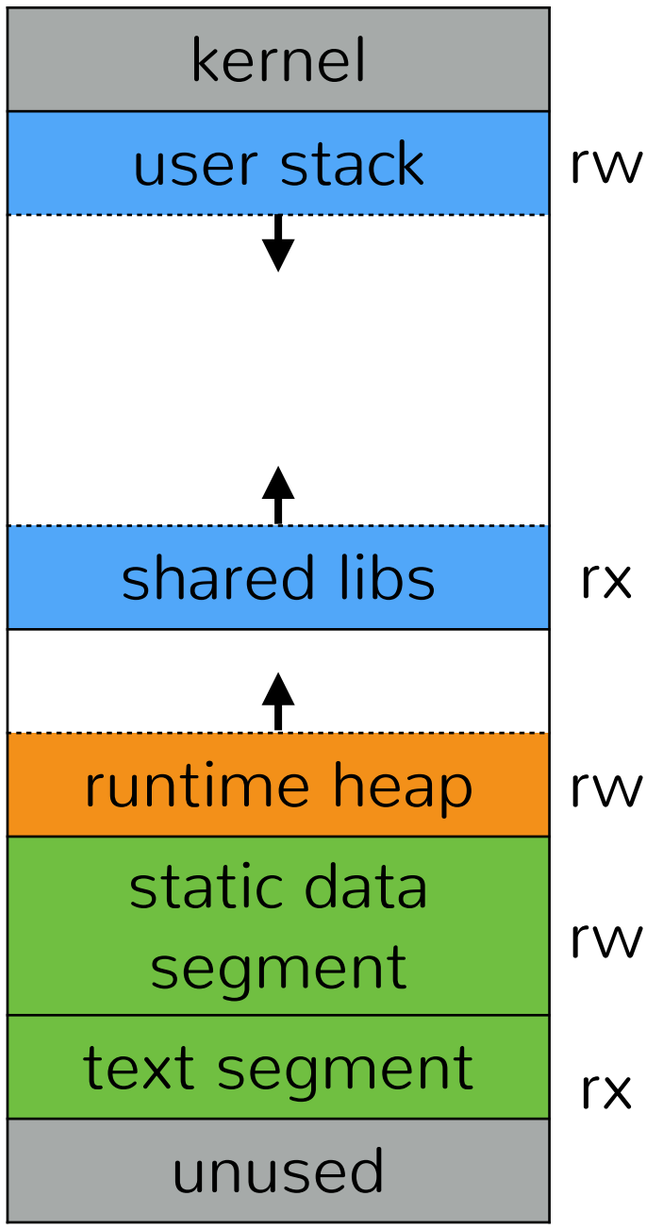
Recall our memory layout

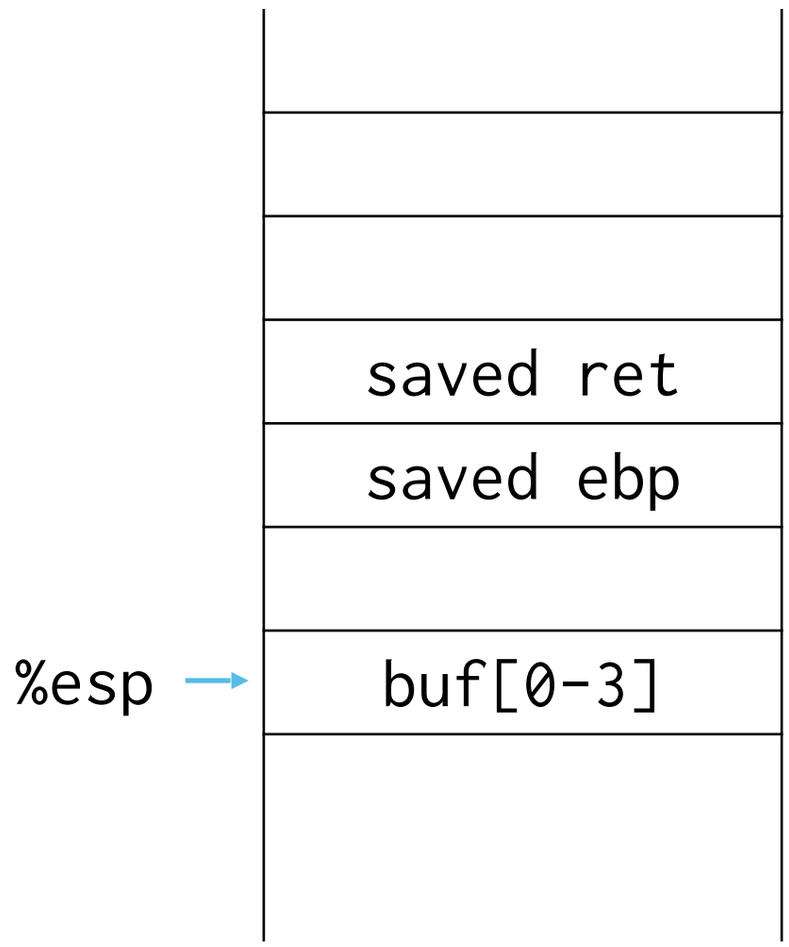


Recall our memory layout

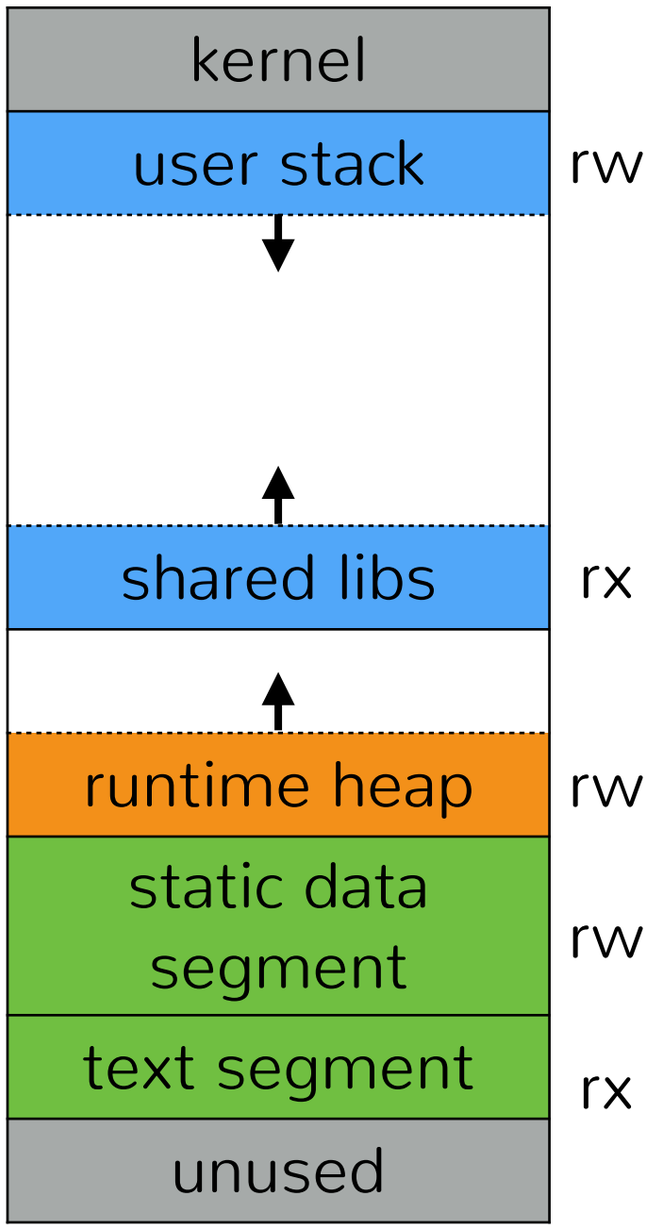


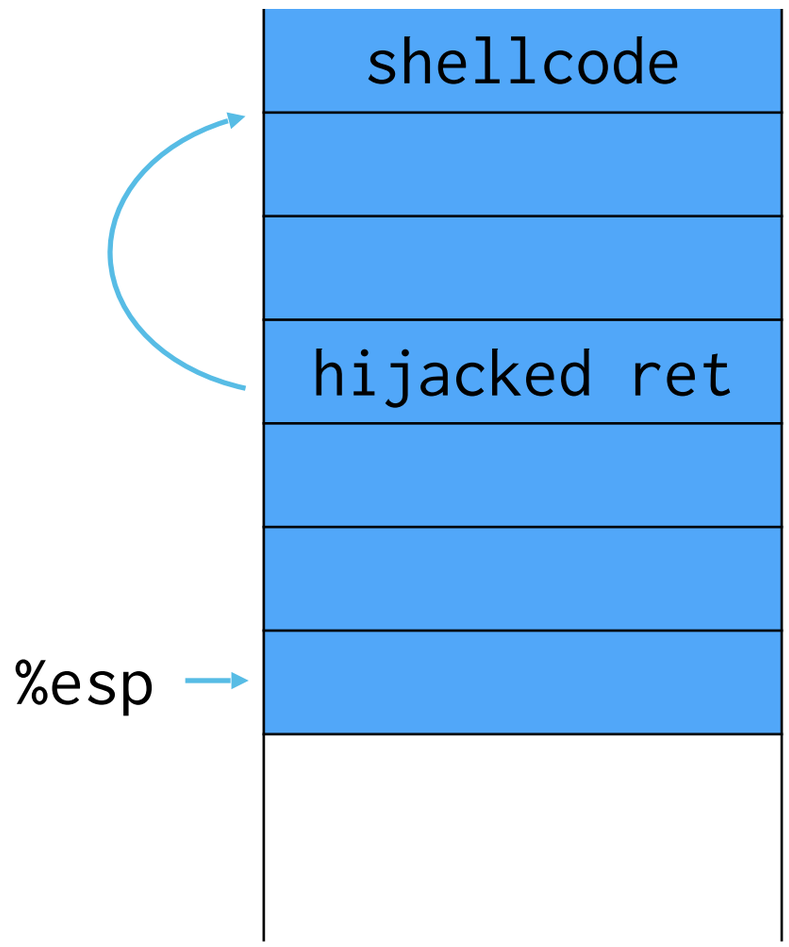
Recall our memory layout



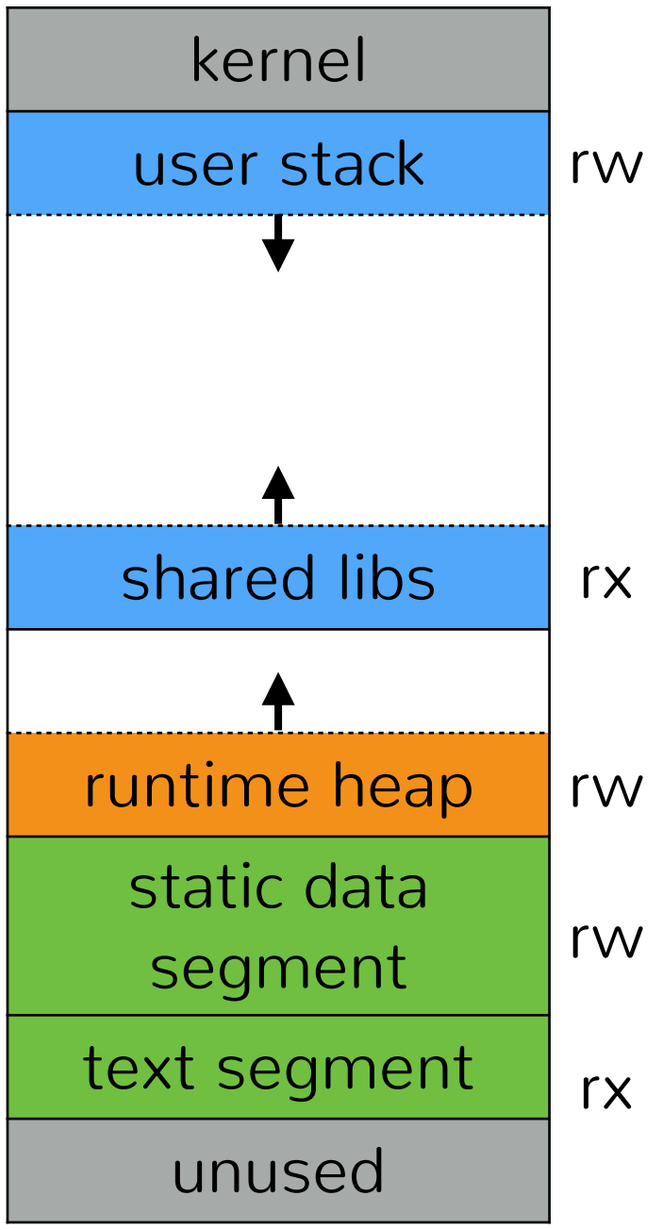


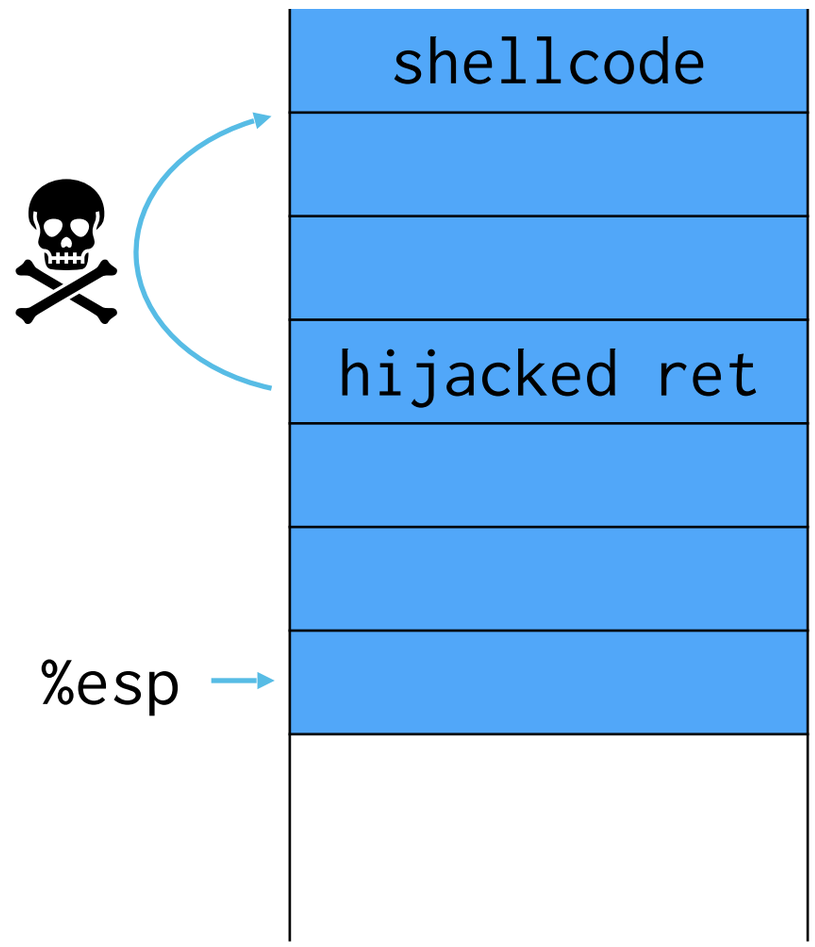
Recall our memory layout





Recall our memory layout





W^X tradeoffs

- **Easy to deploy:** No code changes or recompilation
- **Fast:** Enforced in hardware
  - Downside: what do you do on embedded devices?
- Some pages need to be both writeable and executable
  - Why?

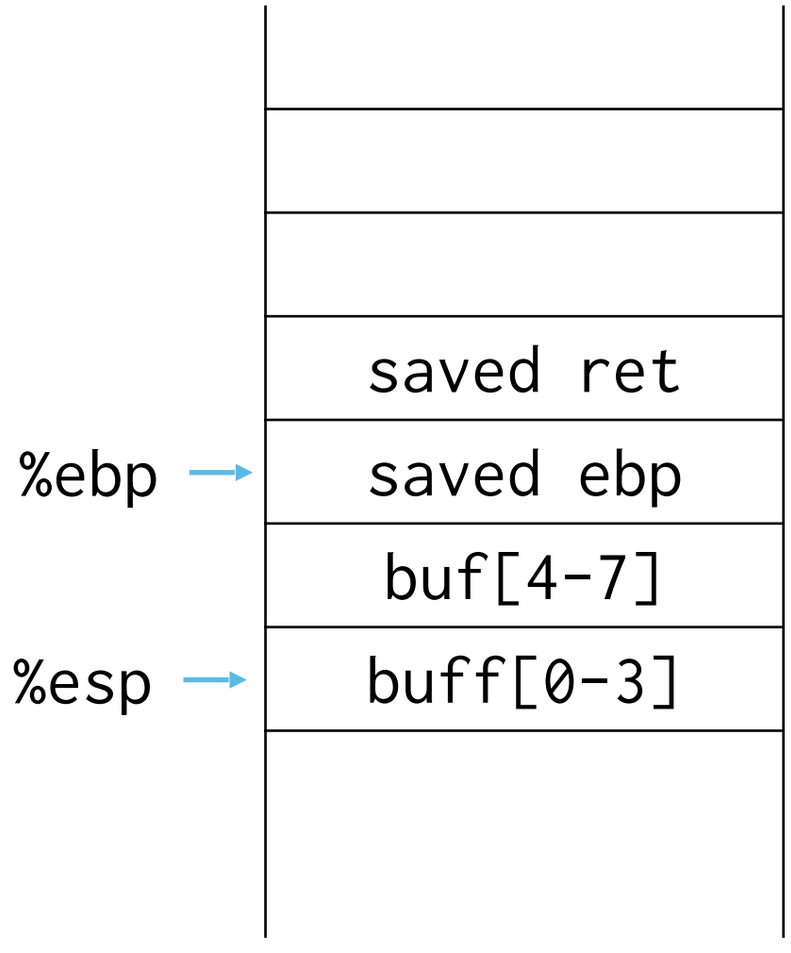
How can we defeat W^X?

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

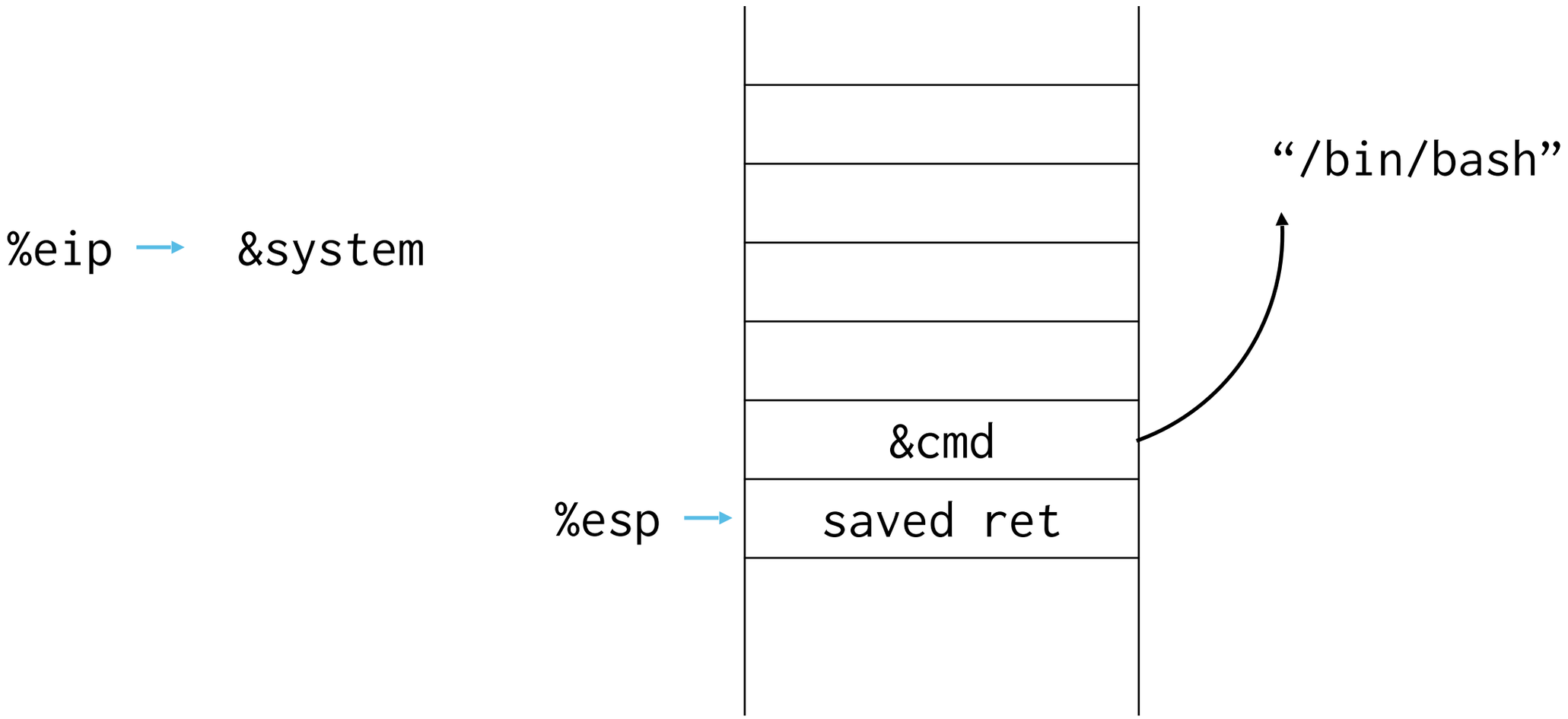
Employees must
wash hands before
returning to libc

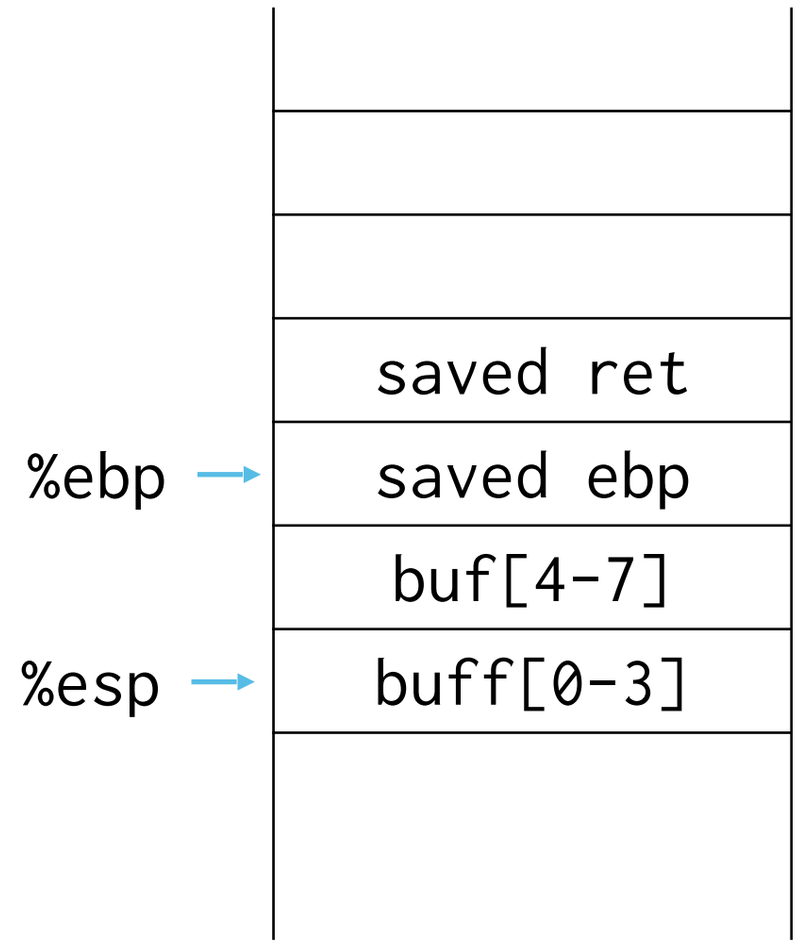
Redirecting control flow to `system()`

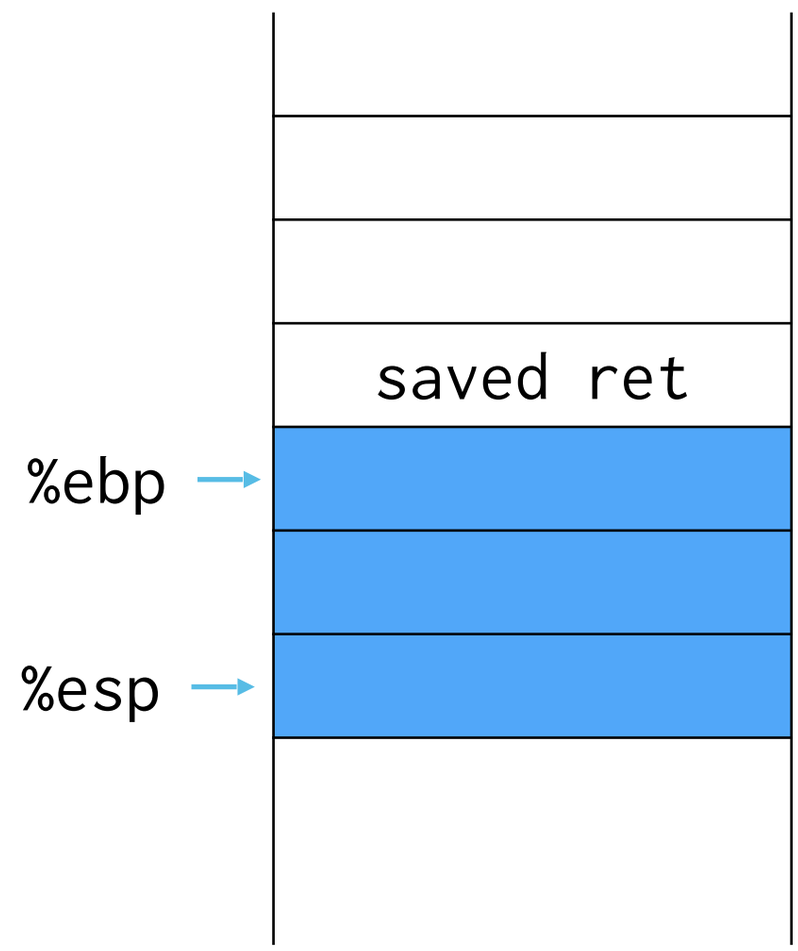
- Last lecture: redirected control flow to `bar()`
- Calling `system()` is the same, but need to have argument to string "`/bin/sh`" on stack

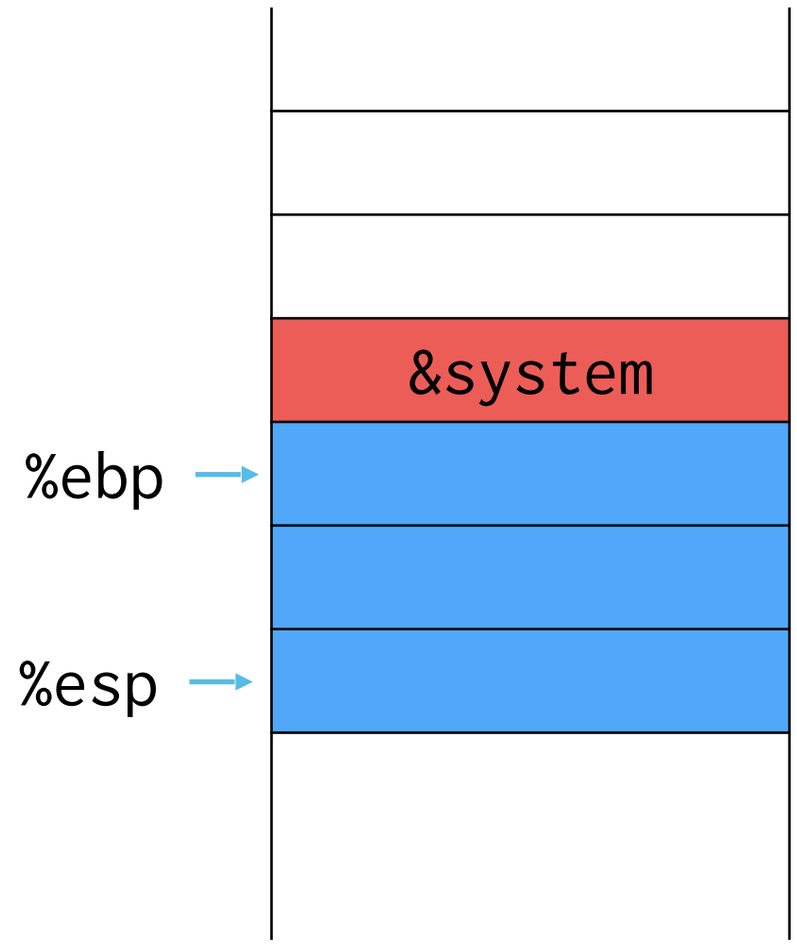


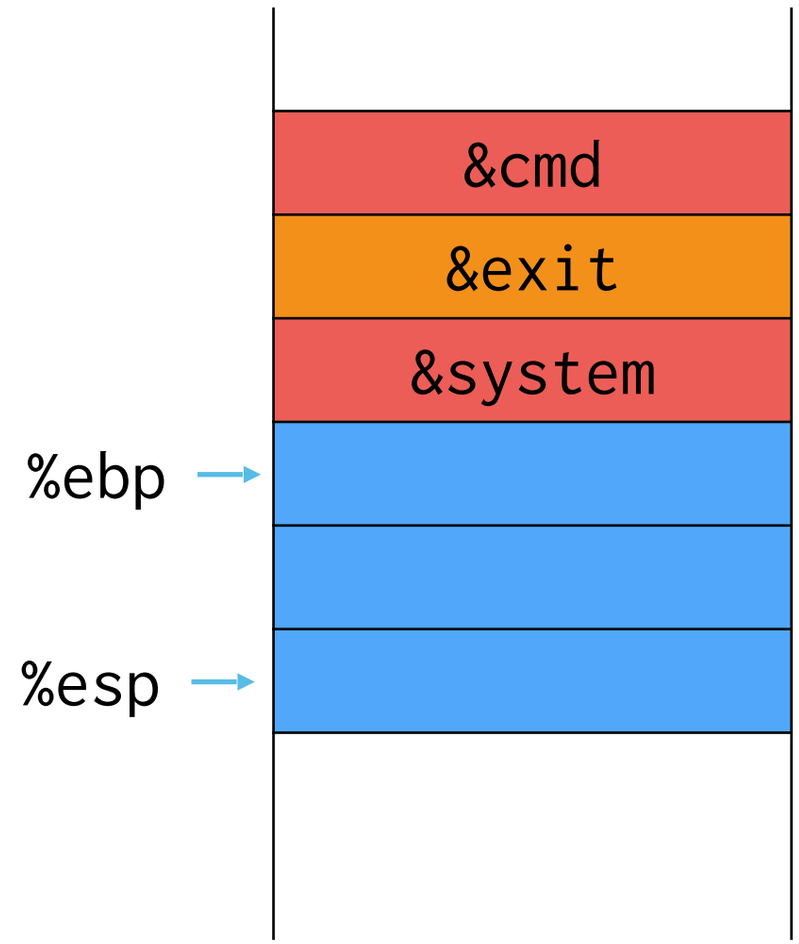
Normal system("/bin/bash") call

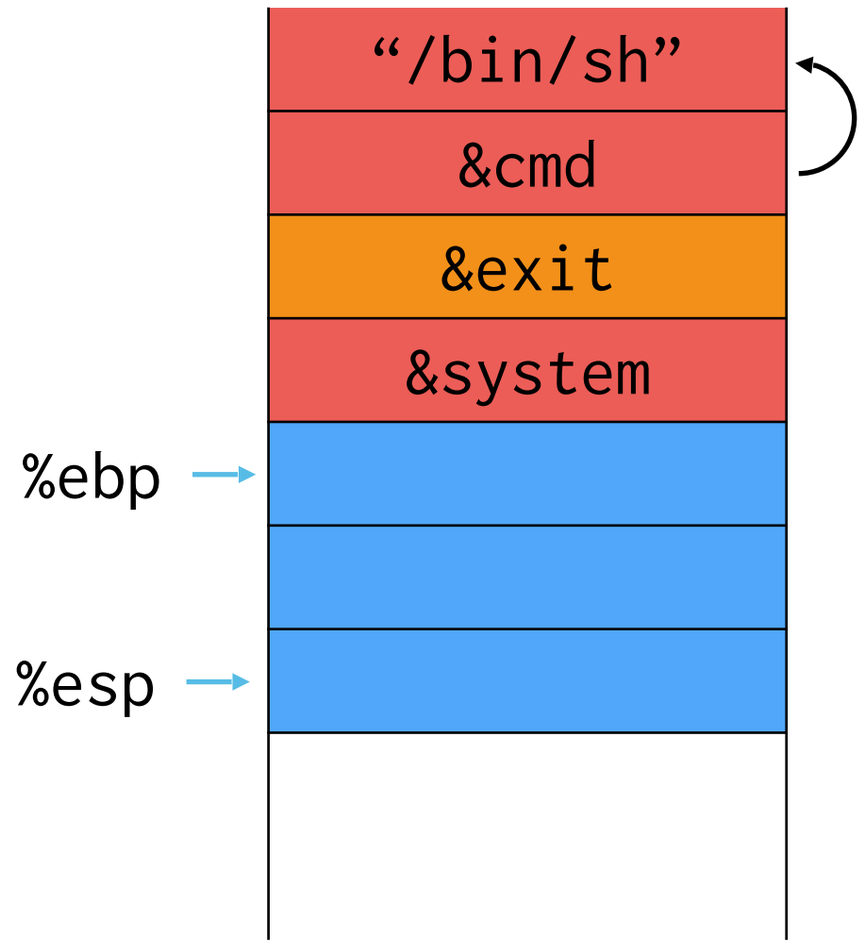


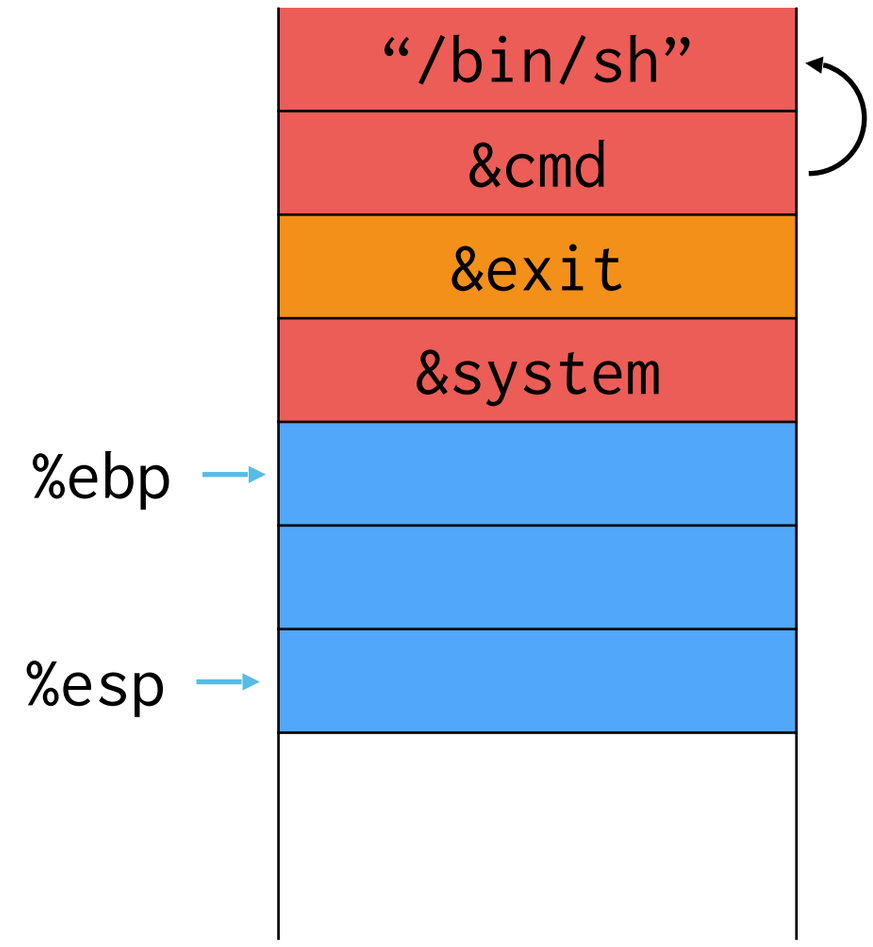

saved ret
%ebp
saved ebp
buf[4-7]
%esp
buff[0-3]


saved ret
%ebp
%esp

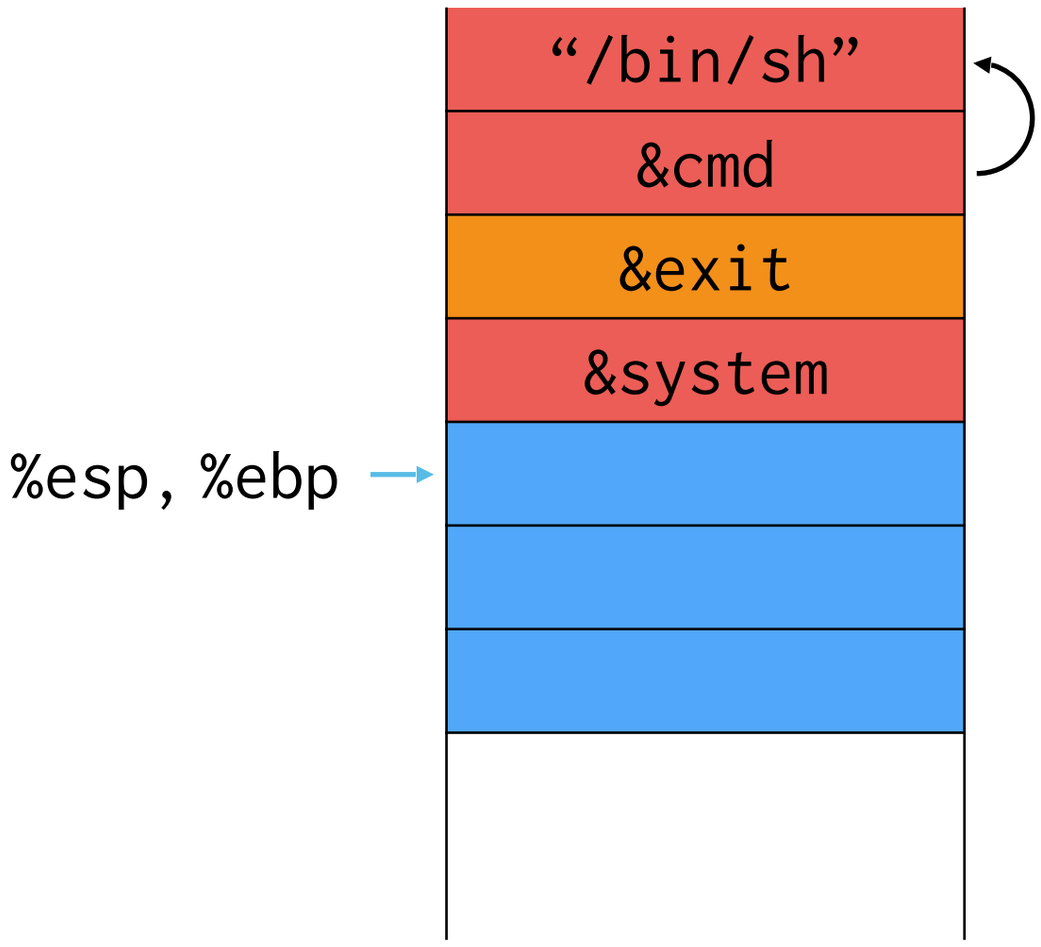

&system
%ebp
%esp


&cmd
&exit
&system
%ebp
%esp

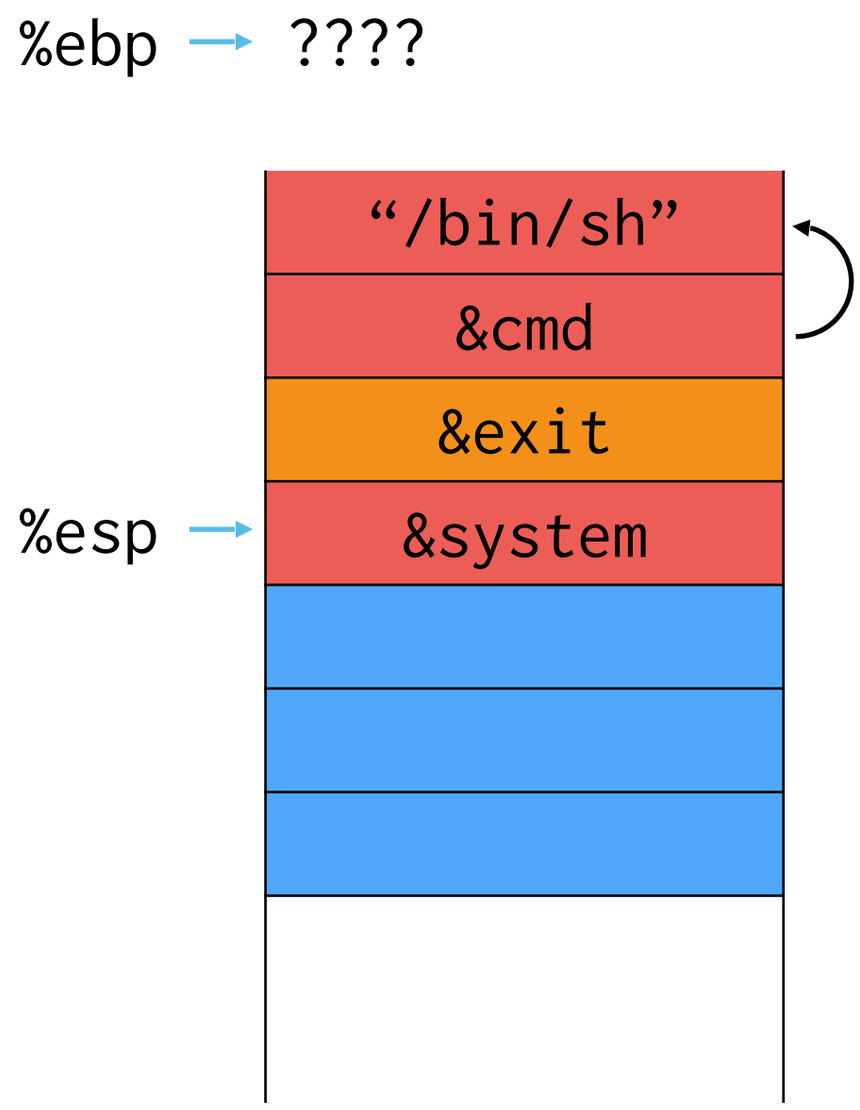

“/bin/sh”
&cmd
&exit
&system
%ebp
%esp



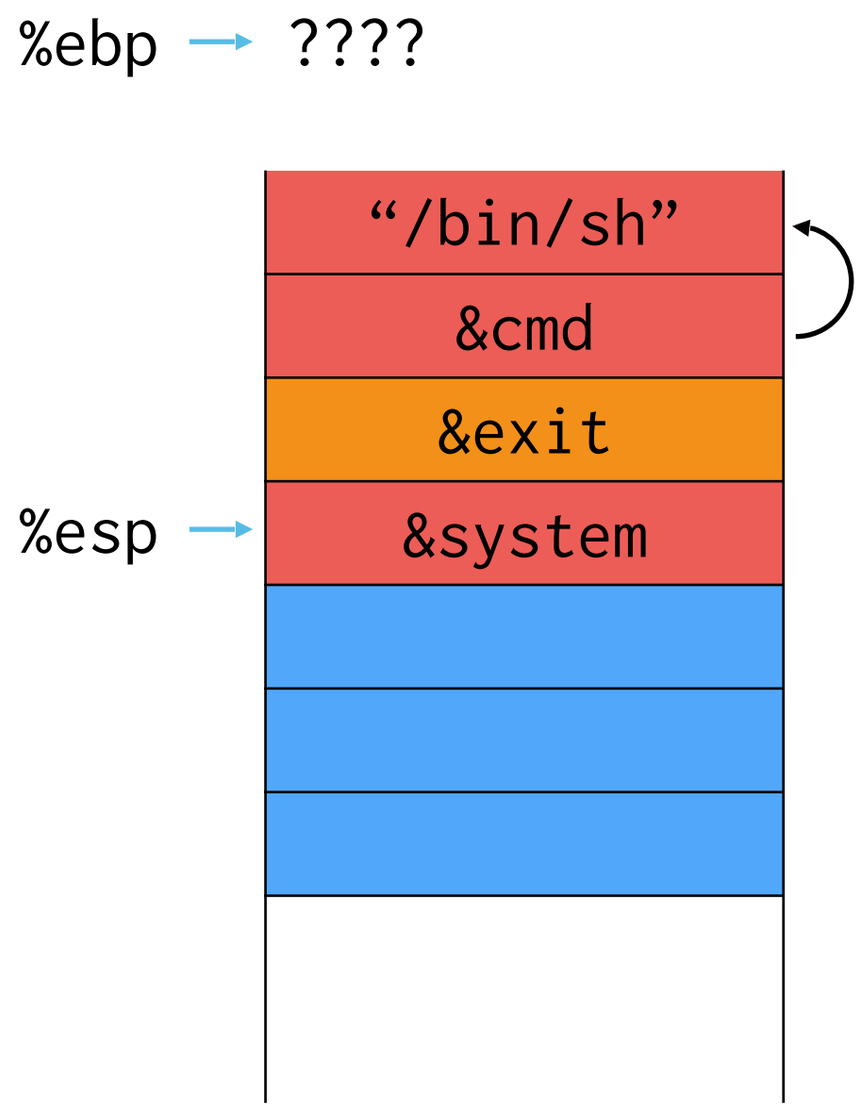

```
leave = mov %ebp, %esp
        pop %ebp
```



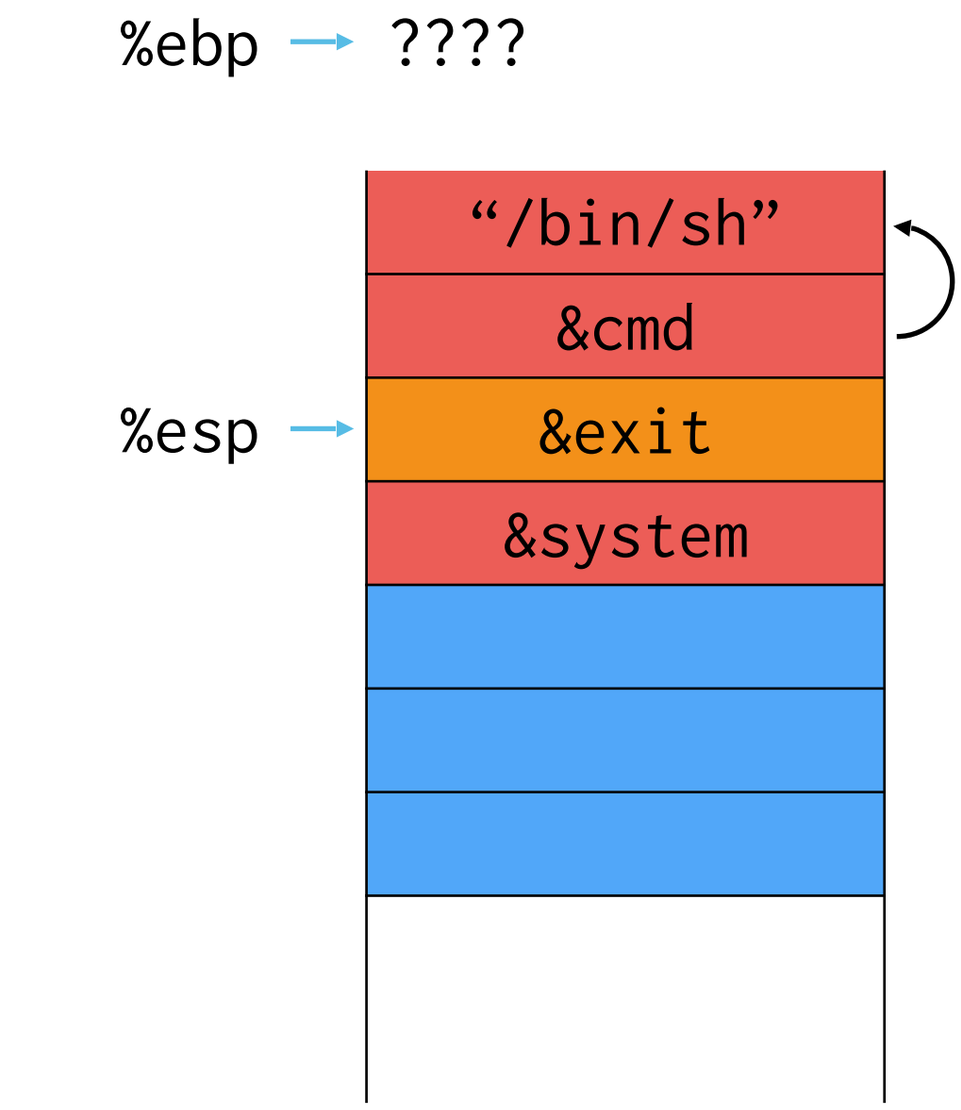

```
leave = mov %ebp, %esp
        pop %ebp
```




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

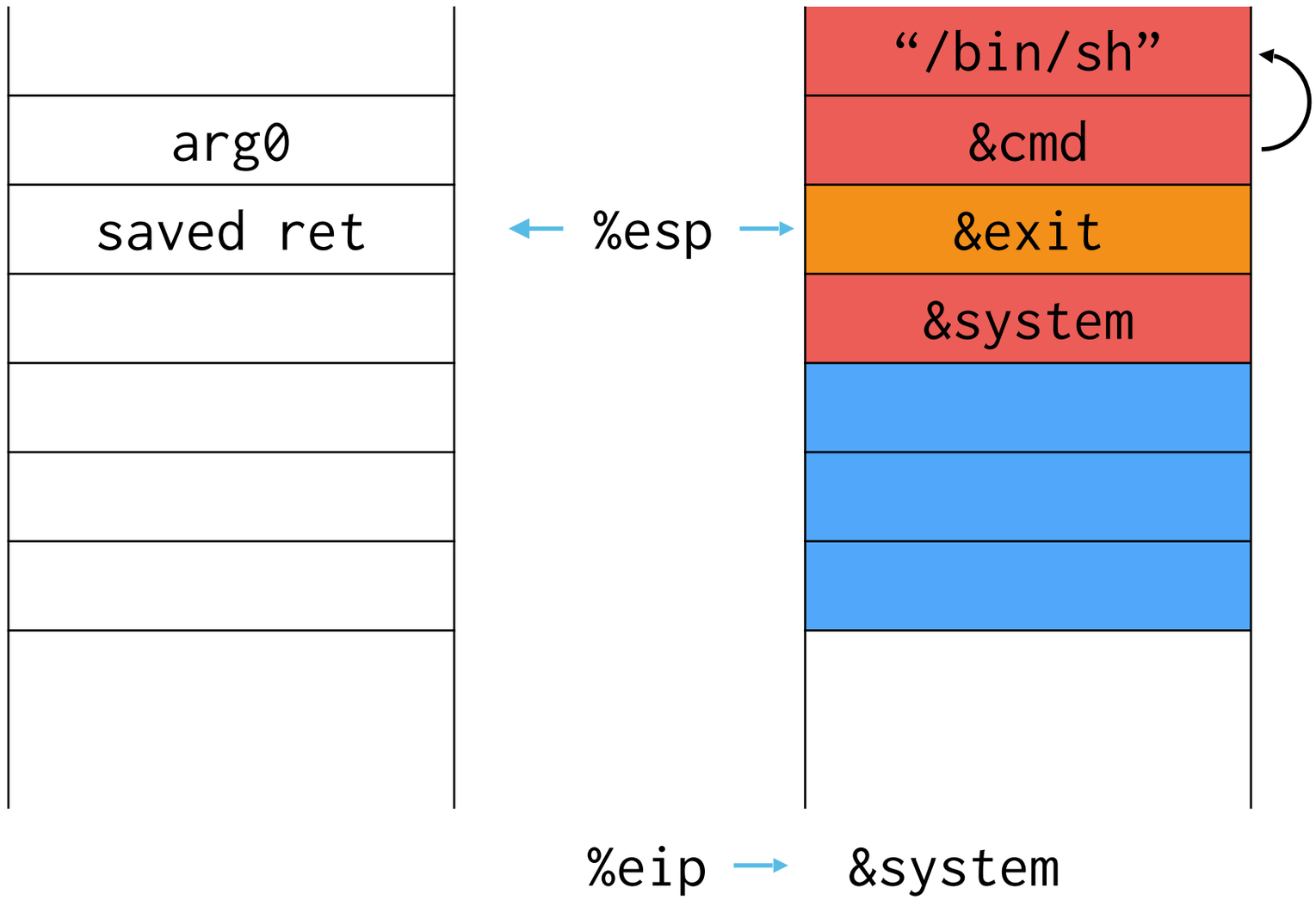



```
ret =   pop %eip
```



ret = pop %eip

This looks like a normal call!



But I want to execute shellcode, not just call `system()`!

Can we inject code?

Can we inject code?

MPROTECT(2) **Linux Programmer's Manual** **MPROTECT(2)**

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

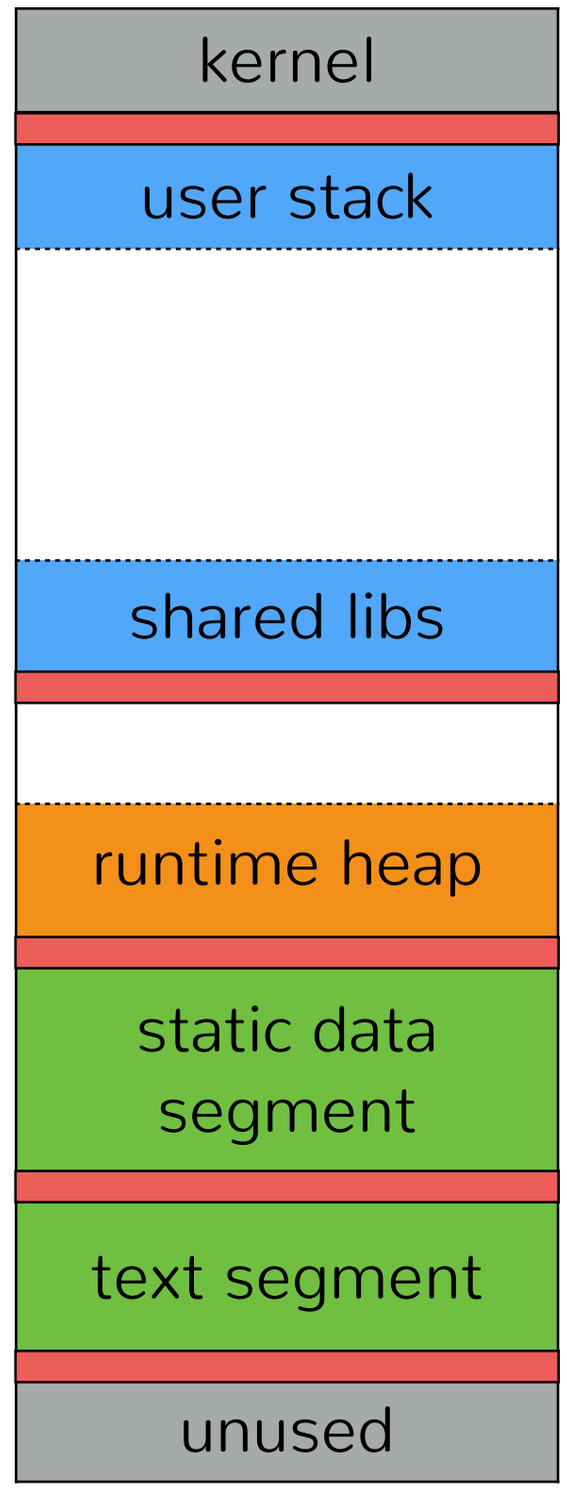
Michalis Athanasakis, FORTH, Greece, michath@ics.forth.gr
Elias Athanasopoulos, FORTH, Greece, elathan@ics.forth.gr
Michalis Polychronakis, Stony Brook University, mikepo@cs.stonybrook.edu
Georgios Portokalidis, Stevens Institute of Tech., gportoka@stevens.edu
Sotiris Ioannidis, FORTH, Greece, sotiris@ics.forth.gr

Buffer overflow mitigations

- Avoid unsafe functions
- 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: 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

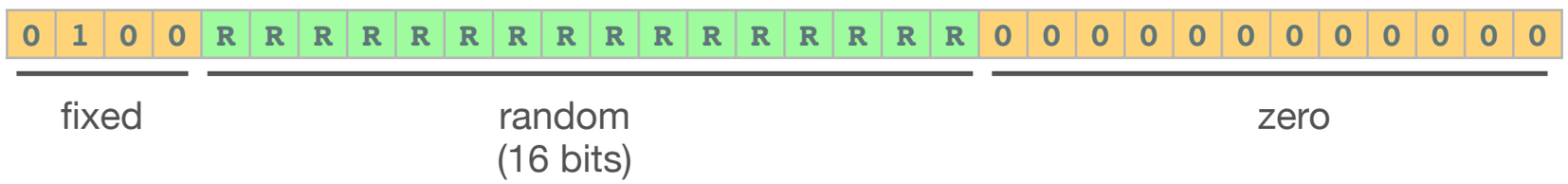


How much do we randomize?

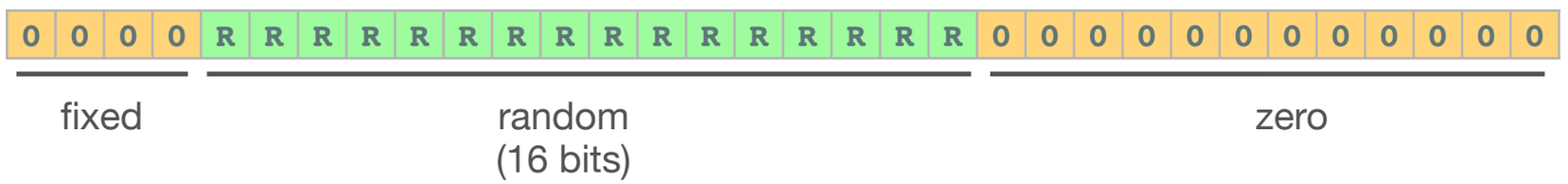
32-bit PaX ASLR (x86)

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
- **But! Helps mitigate heap-based overflow attacks**

When do we randomize?

- Many options
  - At boot?
  - At compile/link time?
  - At run/load time?
  - On fork?
- What's the tradeoff?

How can we defeat ASLR?

- -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()`

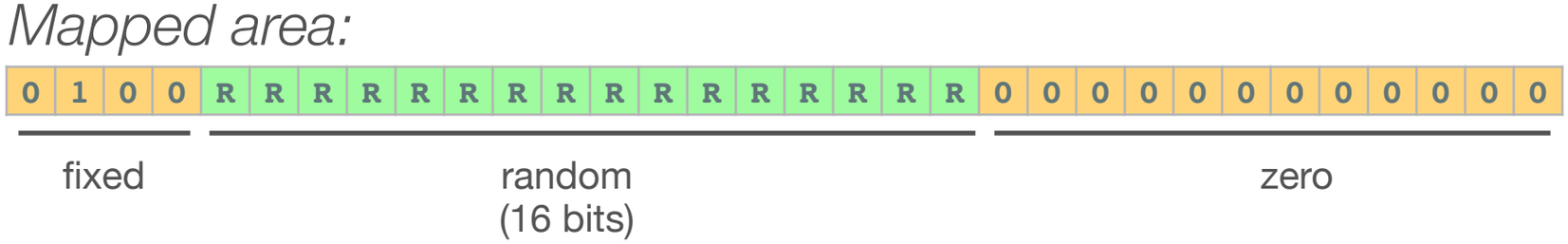
```
char buf[64];
…
strcpy(buf, s); // overflow
```

Assumptions

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

Attack steps

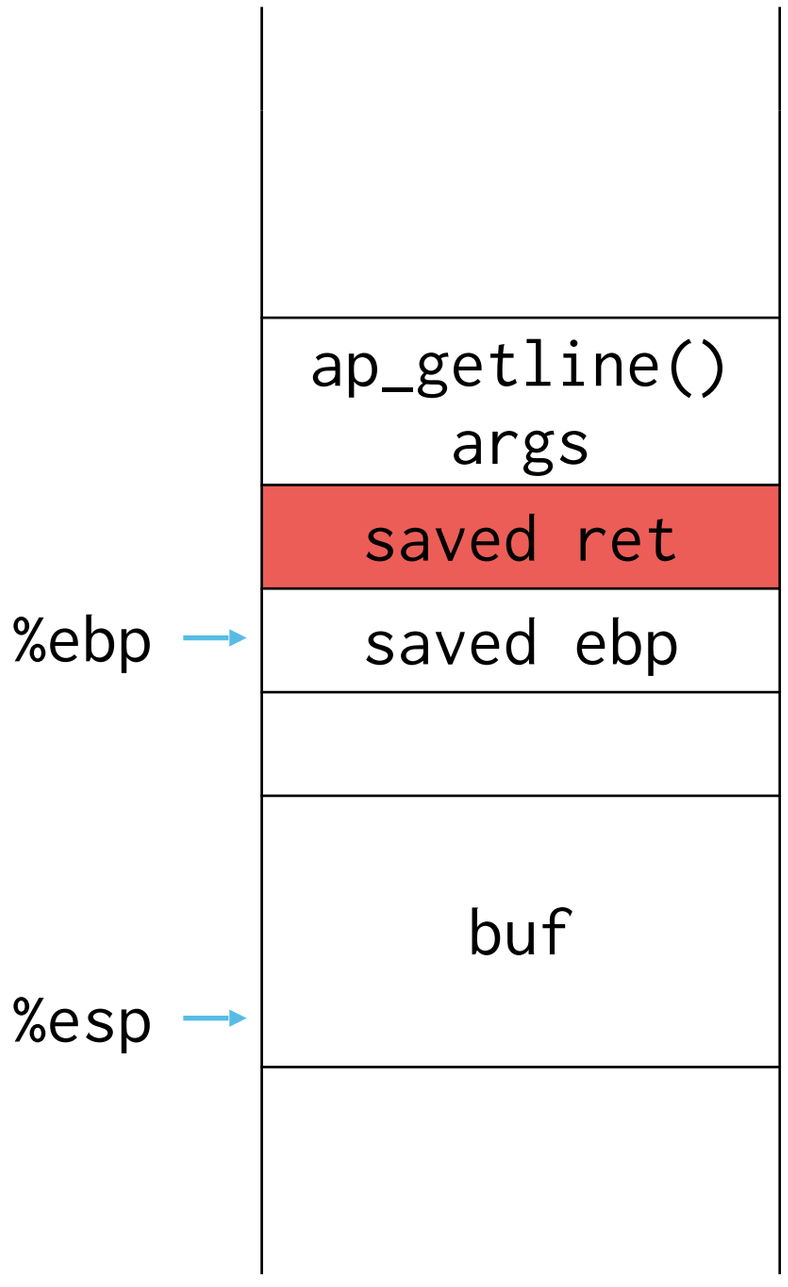
- **Stage 1**: Find base of mapped region



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

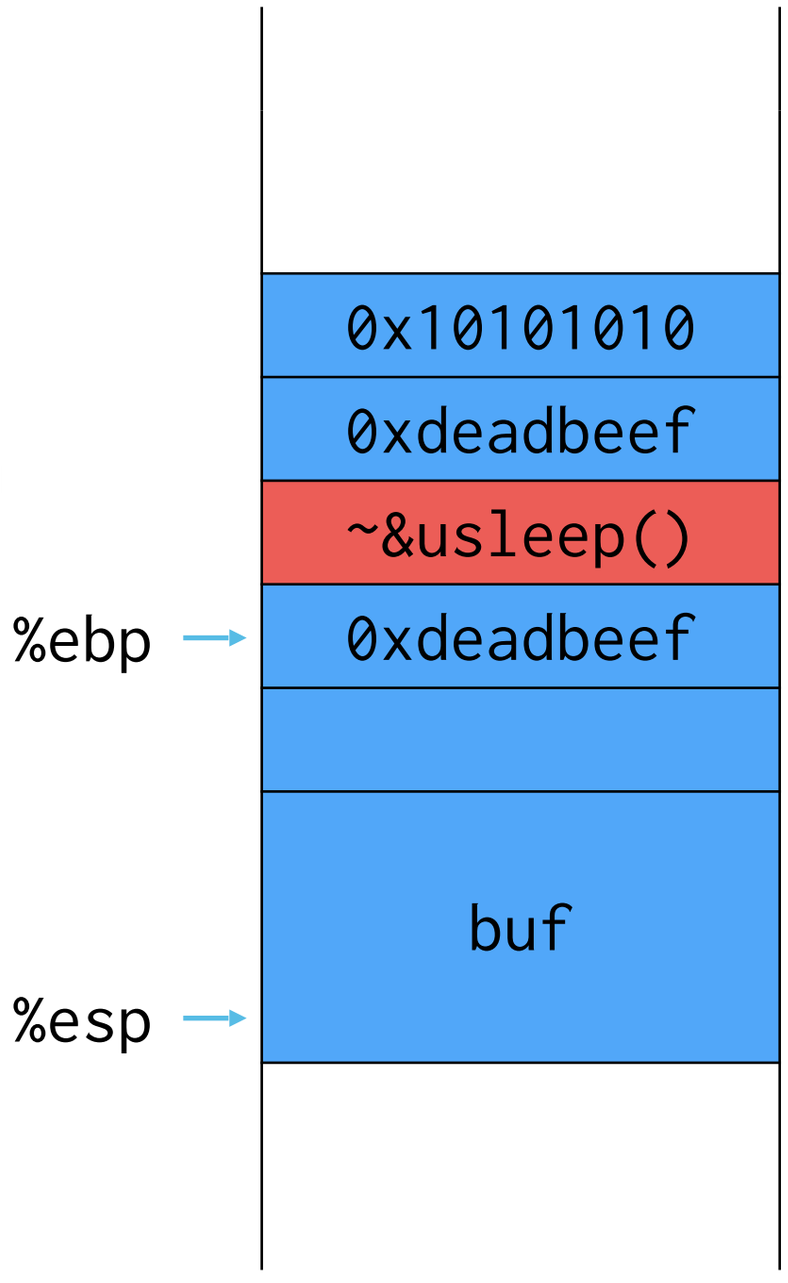
How do we find the mapped region?

- 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

- What is the success probability?
  - ➤
- Do we need to derandomize the stack base?
  - ➤

Derandomizing ASLR

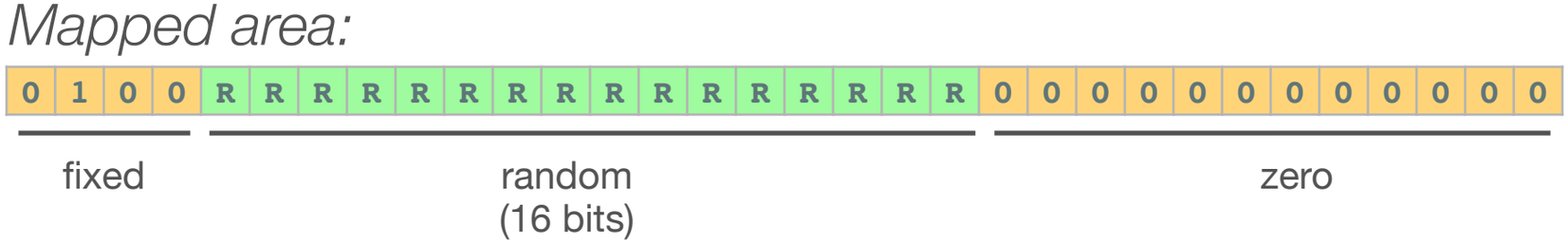
- What is the success probability?
  - $1/2^{16}$ — 65,536 tries maximum
- Do we need to derandomize the stack base?
  -

Derandomizing ASLR

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

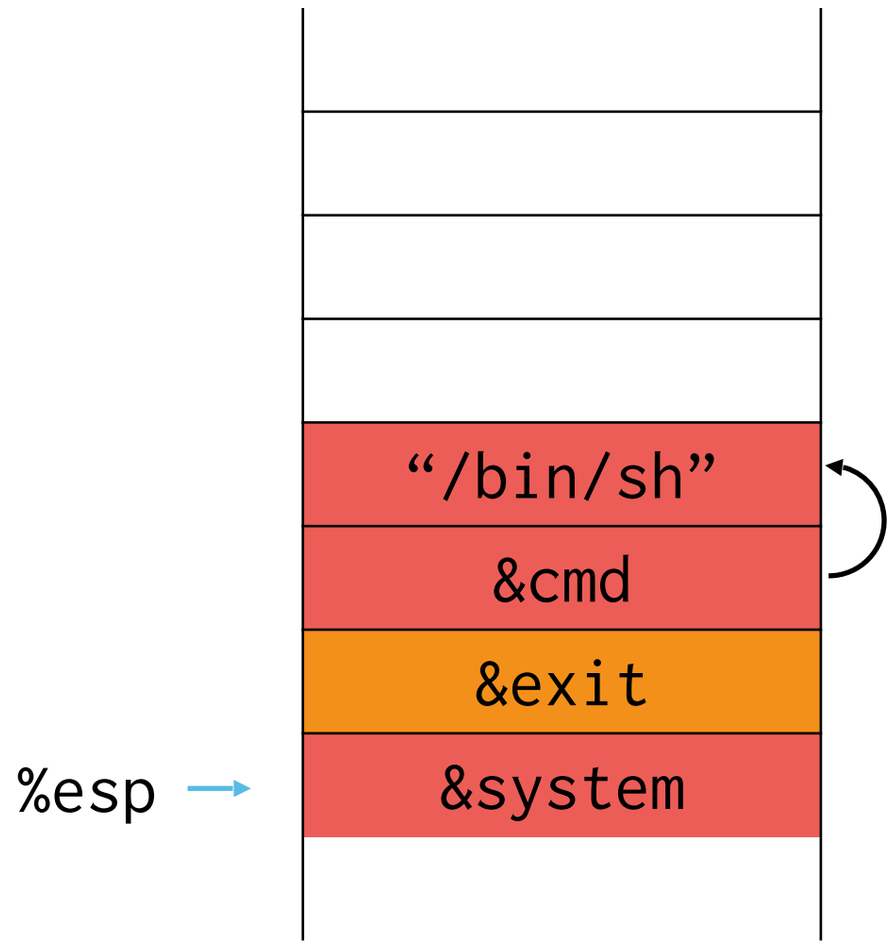
Attack steps

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



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

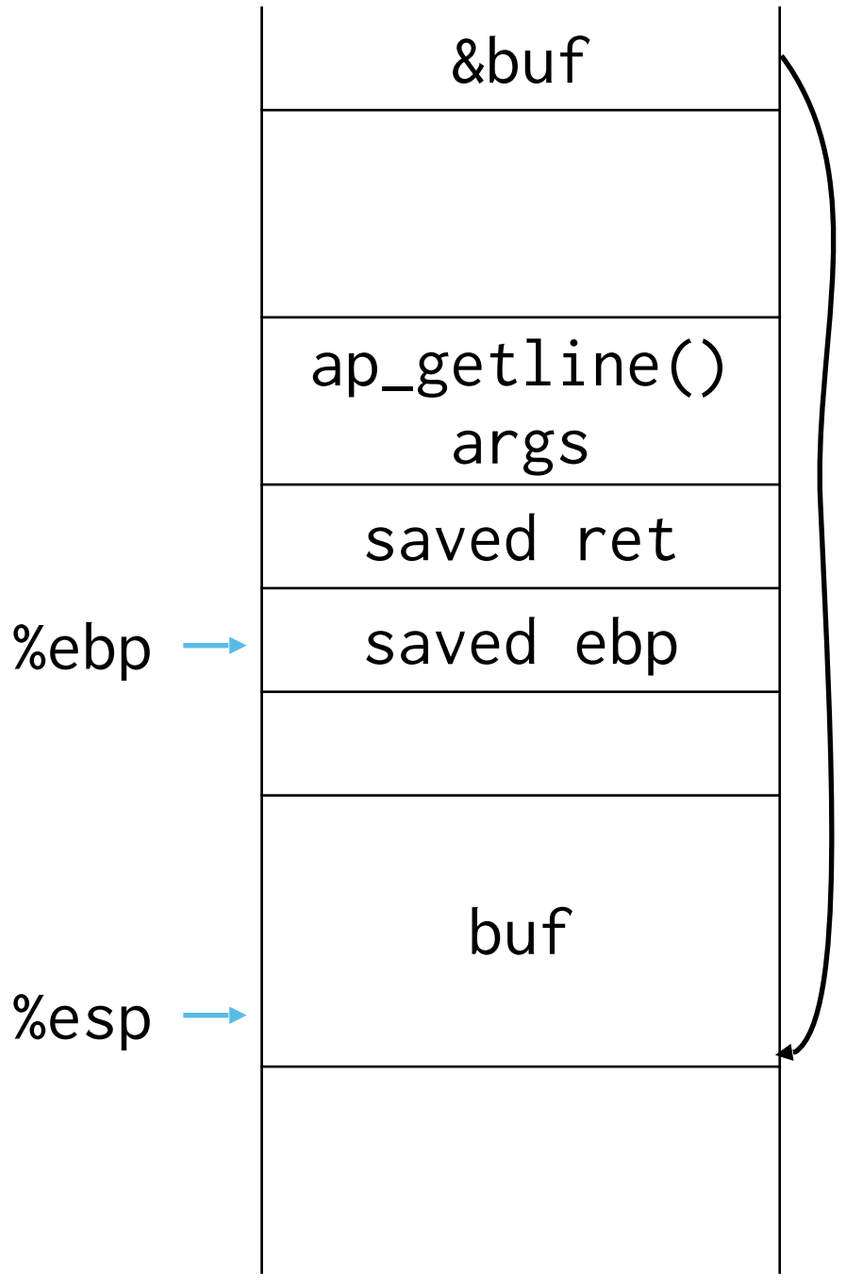
How do we call system?



How do we call system?

In the paper...

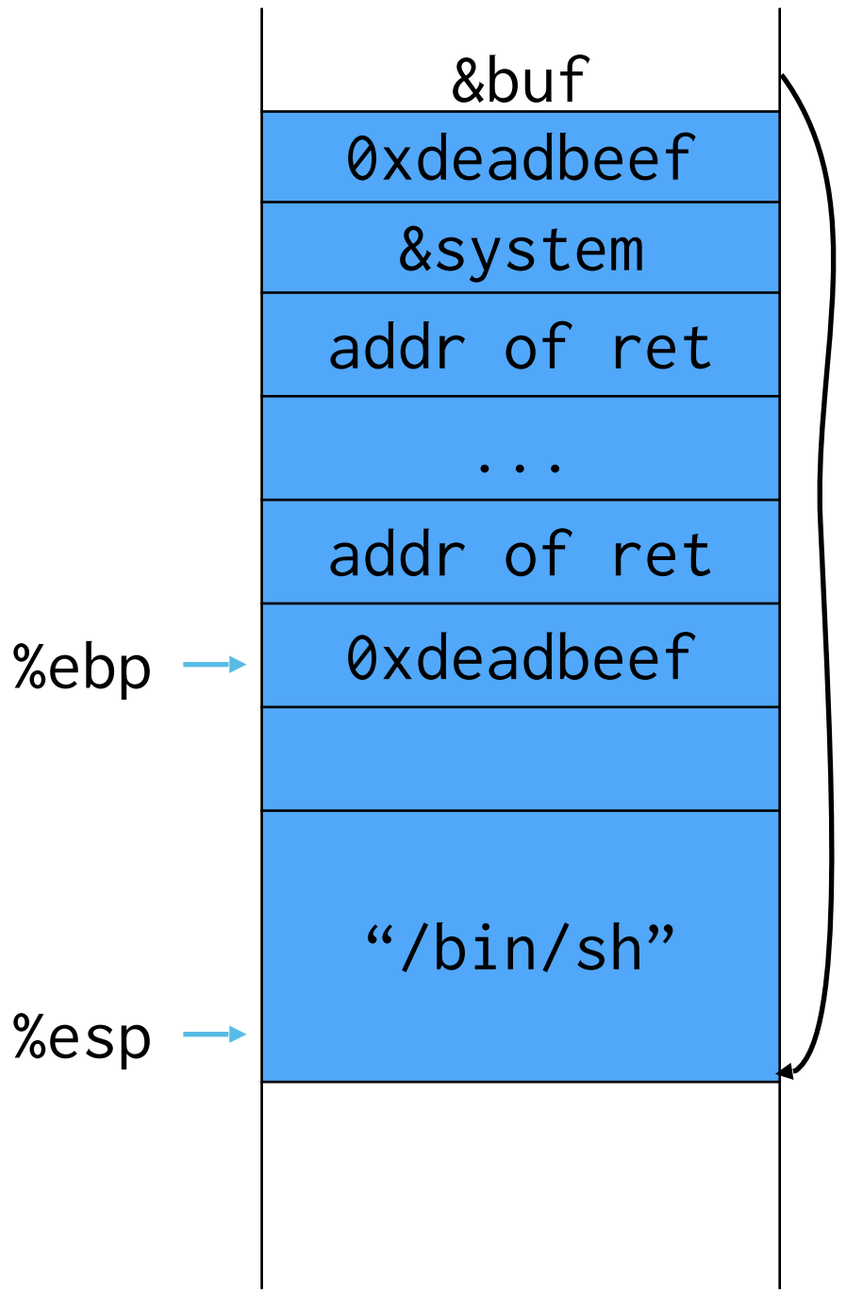
- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`



How do we call system?

In the paper...

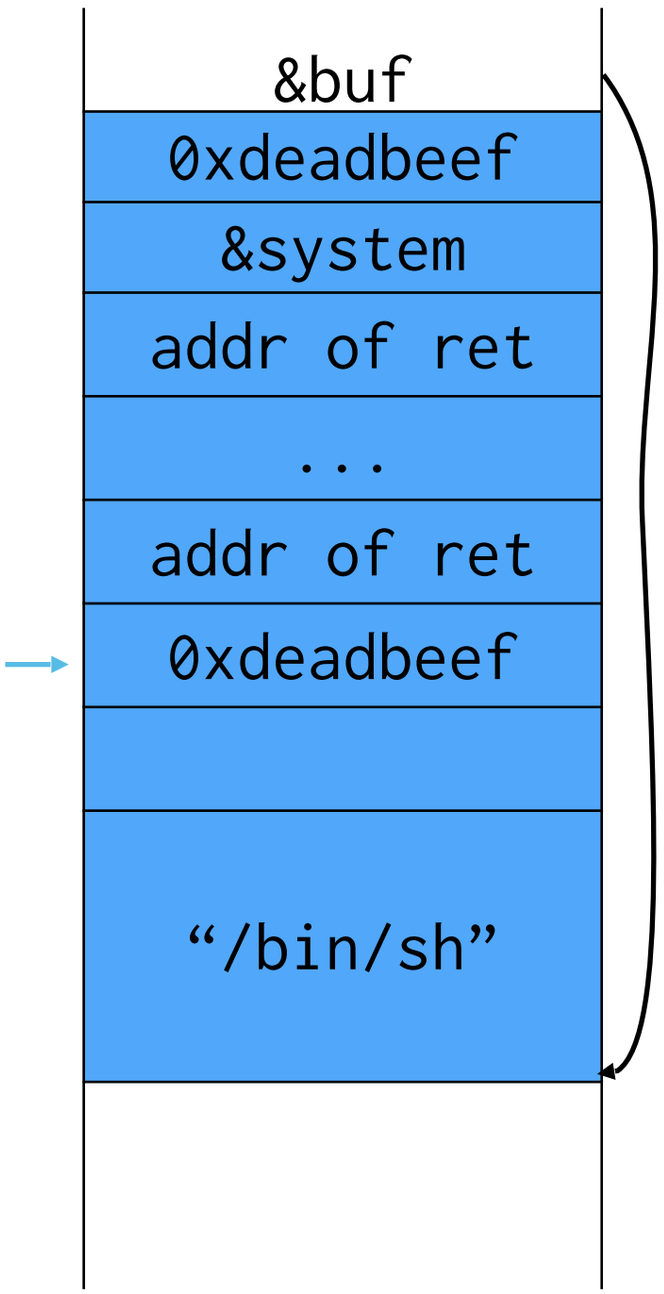
- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`



How do we call system?

In the paper...

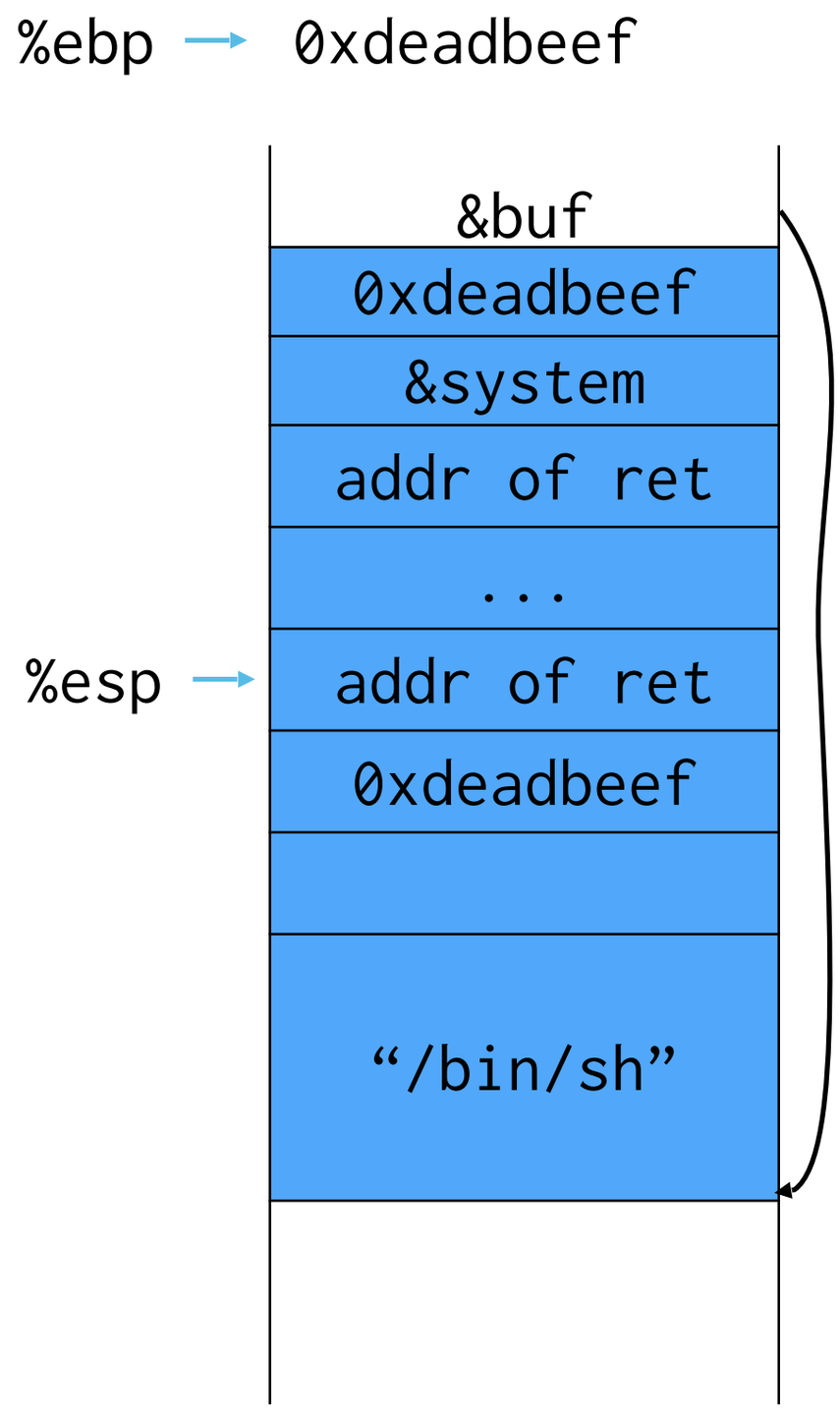
- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`



How do we call system?

In the paper...

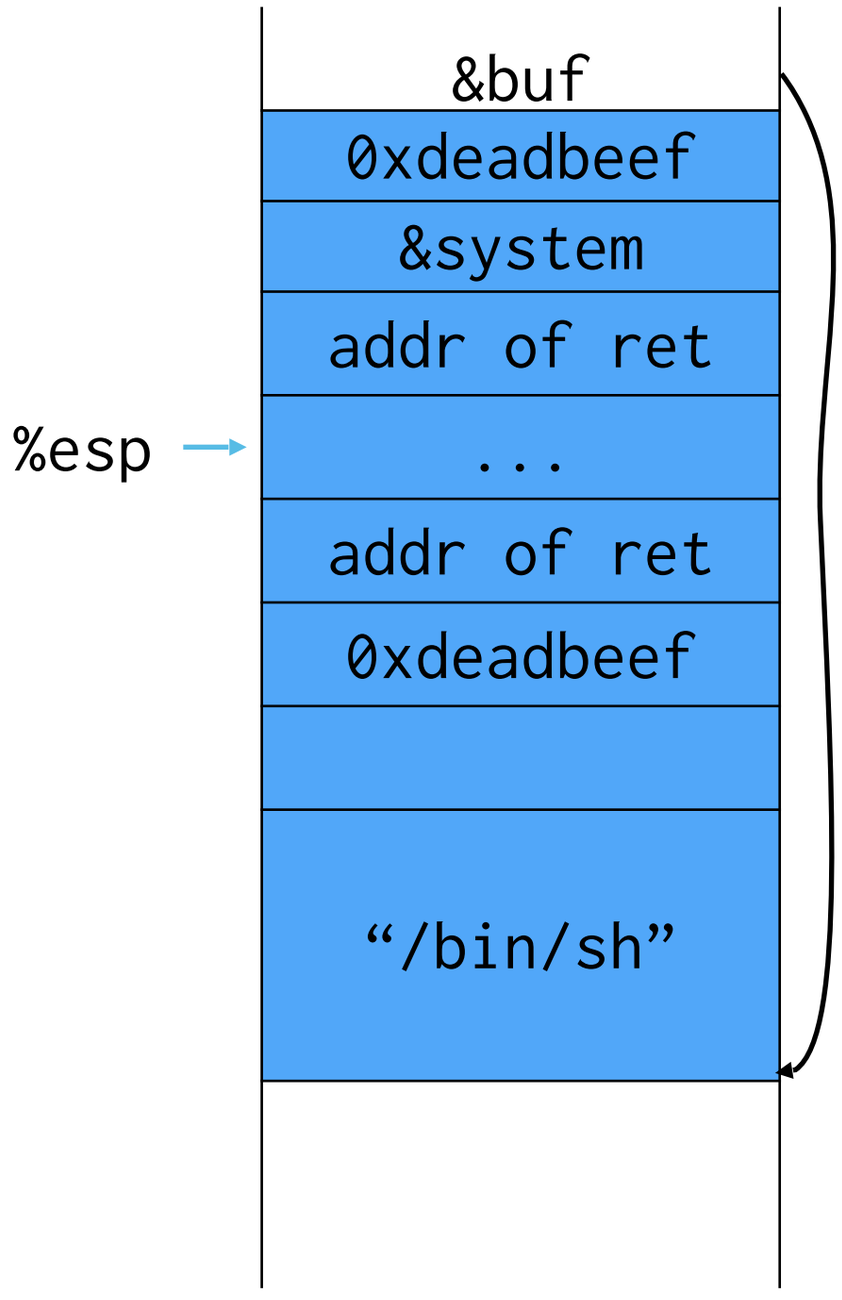
- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`



How do we call system?

In the paper...

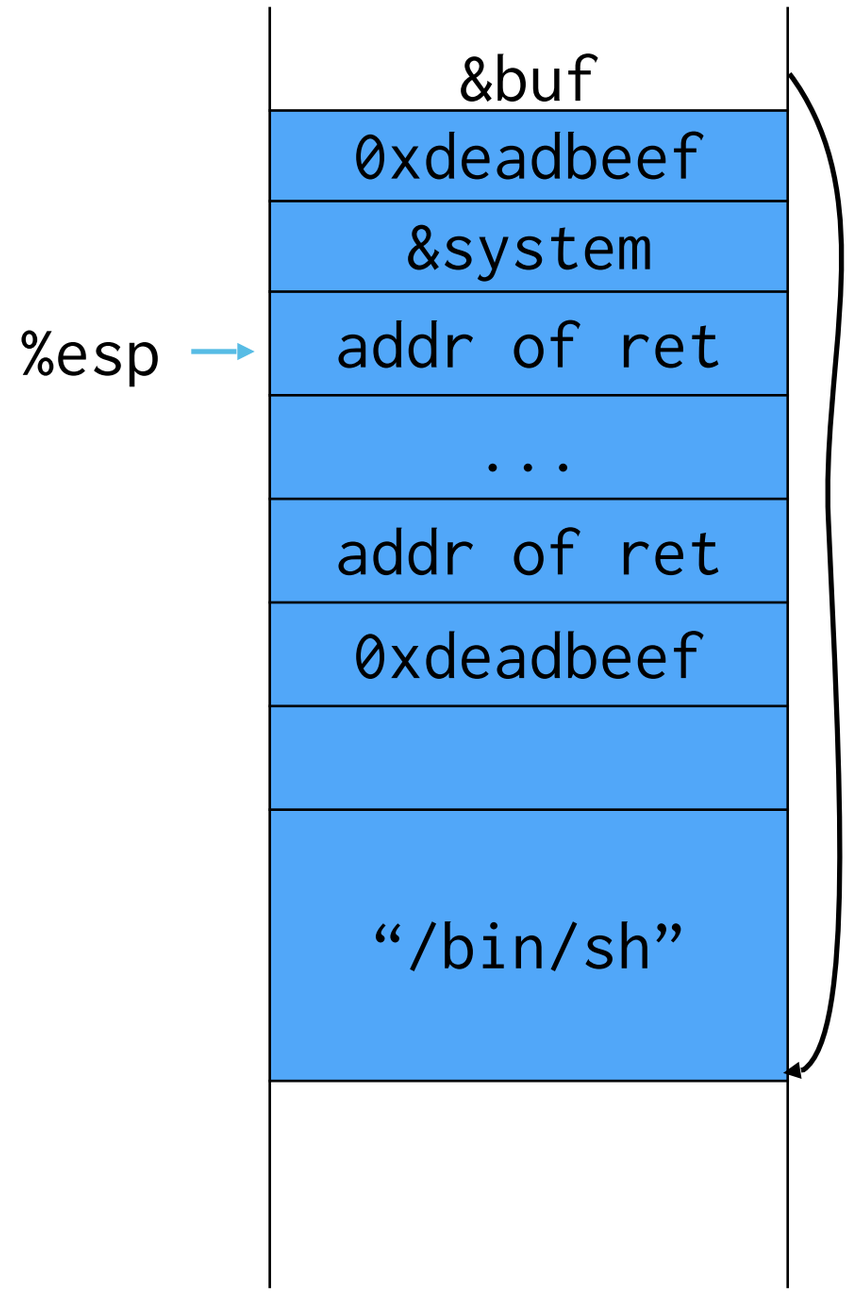
- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`



How do we call system?

In the paper...

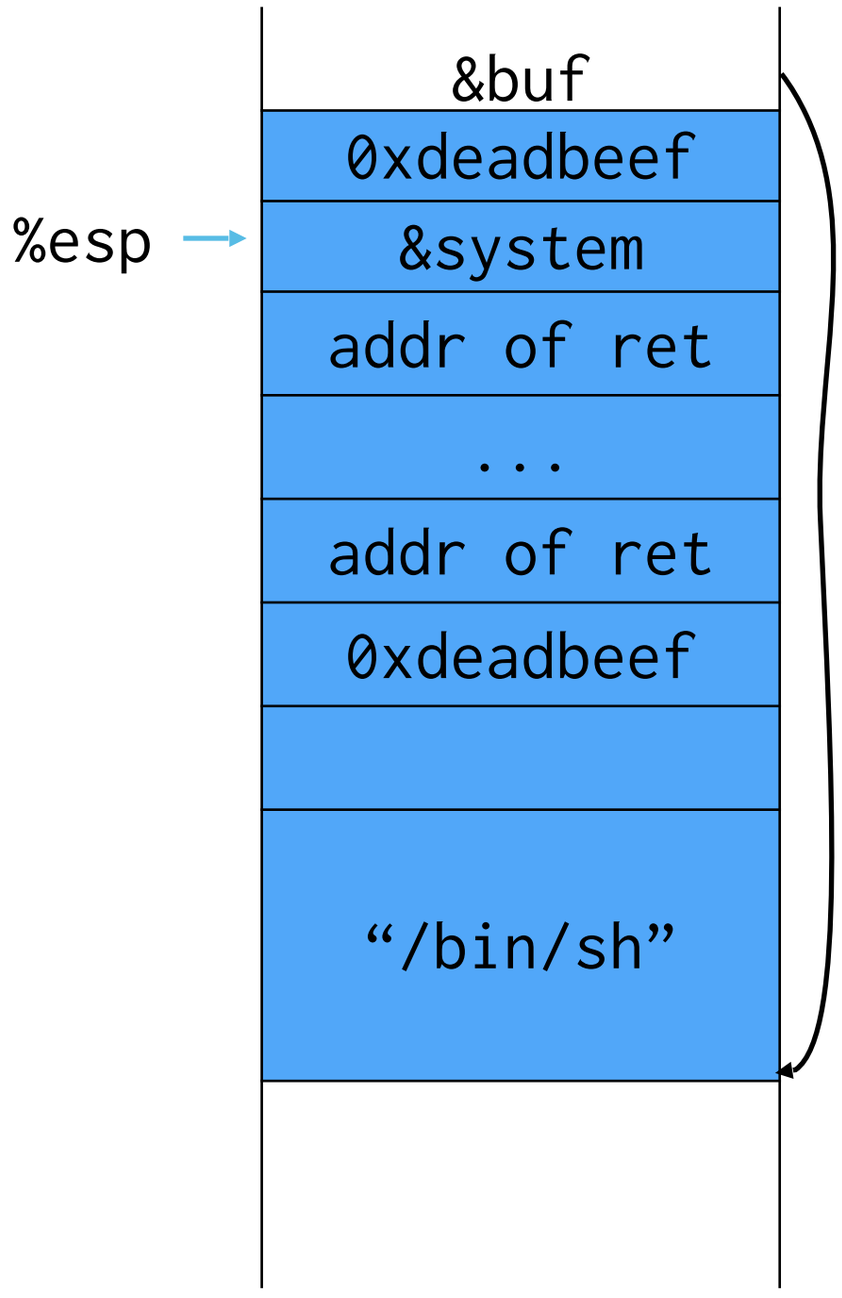
- Overwrite saved return pointer with address of `ret` instruction in libc
- Repeat until address of buf looks like argument to `system()`
- Append address of `system()`



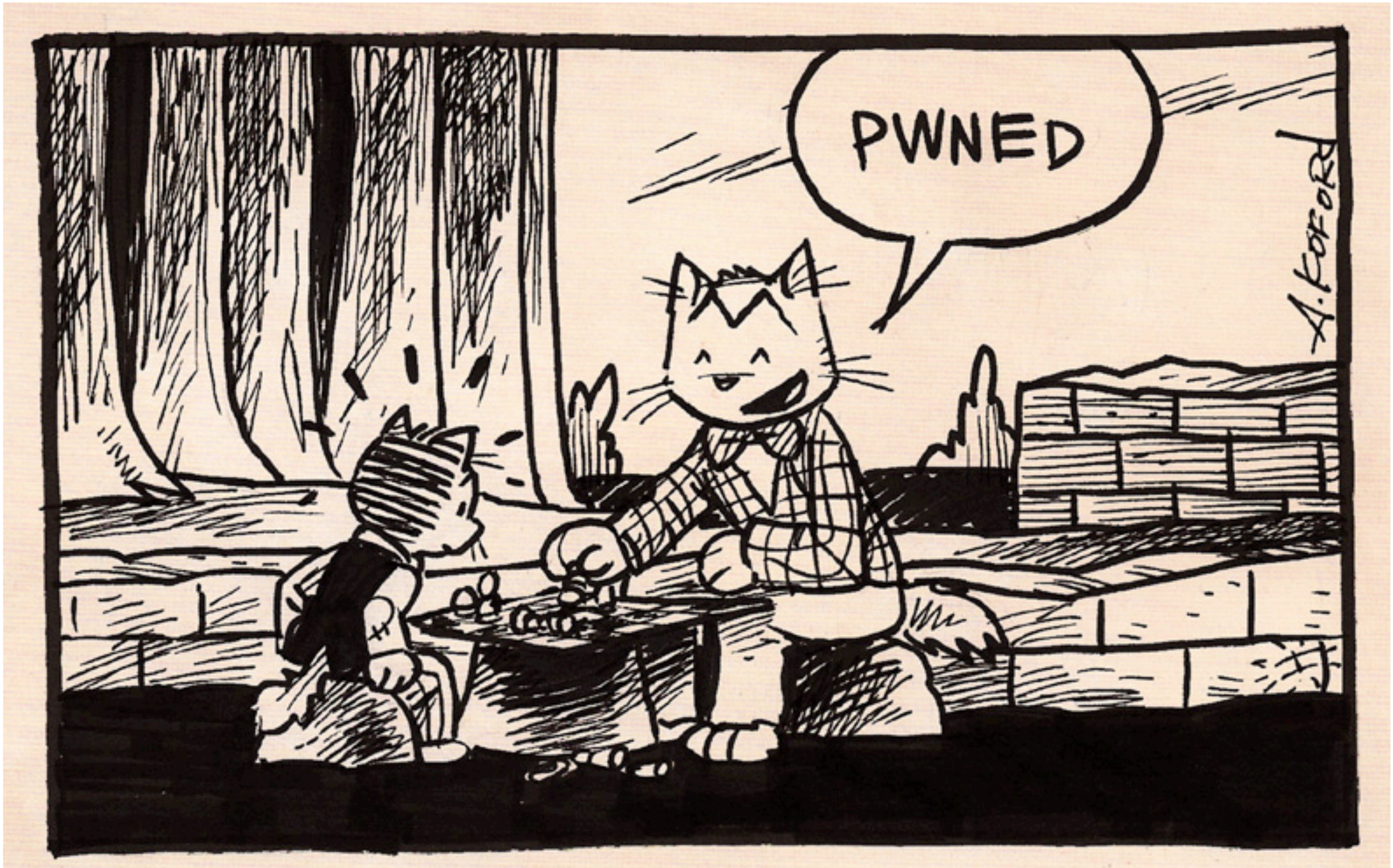
How do we call system?

In the paper...

- 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?



Buffer overflow mitigations

- Avoid unsafe functions
- Stack canaries
- 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