



CSE 127: Computer Security

Stack Buffer Overflows

Nadia Heninger and Deian Stefan

Some slides adopted from Kirill Levchenko and Stefan Savage

When is a program secure?

- When it does exactly what it should?
 - Not more
 - Not less

When is a program secure?

- When it does exactly what it should?
 - Not more
 - Not less
- But how do we know what a program is supposed to do?
 - Somebody tells us? (Do we trust them?)
 - We write the code ourselves? (What fraction of the software you use have you written?)

When is a program secure?

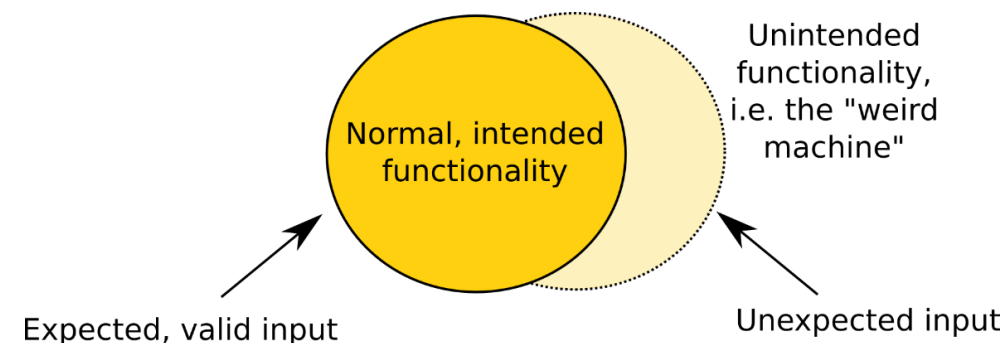
- Try 2: When it doesn't do bad things
- Easier to specify a list of “bad” things:
 - Delete or corrupt important files
 - Crash my system
 - Send my password over the Internet
 - Send threatening email to the professor

When is a program secure?

But ... what if most of the time the program doesn't do bad things, but occasionally it does? Or could?
Is it secure?

Weird machines

- Complex systems almost always contain unintended functionality
 - “Weird machines”
- An **exploit** is a mechanism by which an attacker triggers unintended functionality in the system
 - Programming of the weird machine



Weird machines

- Security requires understanding not just the intended but also the unintended functionality present in the implementation
 - Developers' blind spot
 - Attackers' strength

What is a software vulnerability?

What is a software vulnerability?

- A bug in a program that allows an unprivileged user capabilities that should be denied to them

What is a software vulnerability?

- A bug in a program that allows an unprivileged user capabilities that should be denied to them
- There are a lot of types of vulns, but among the most classic and important are vulnerabilities that violate “control flow integrity”
 - Why? Lets attacker run code on your computer!

What is a software vulnerability?

- A bug in a program that allows an unprivileged user capabilities that should be denied to them
- There are a lot of types of vulns, but among the most classic and important are vulnerabilities that violate “control flow integrity”
 - Why? Lets attacker run code on your computer!
- Typically these involve violating assumptions of the programming language or its run-time

Starting exploits

- Dive into low level details of how exploits work
 - How can a remote attacker get **your** machine to execute **their** code?
- Threat model
 - Victim code is **handling input** that comes from across a security boundary
 - What are some examples of this?
 - Want to protect **integrity of execution** and **confidentiality of data** from being compromised by malicious and highly skilled users of our system.

Today: (stack) buffer overflows

Lecture objectives:

- Understand how buffer overflow vulnerabilities can be exploited
- Identify buffer overflow vulnerabilities in code and assess their impact
- Avoid introducing buffer overflow vulnerabilities during implementation
- Correctly fix buffer overflow vulnerabilities

Buffer overflows

- Defn: an anomaly that occurs when a program writes data beyond the boundary of a buffer.
- Archetypal software vulnerability
 - Ubiquitous in system software (C/C++)
 - OSes, web servers, web browsers, etc.
 - If your program crashes with memory faults, you probably have a buffer overflow vulnerability.

Why are they interesting?

- A basic core concept that enables a broad range of possible attacks
 - Sometimes a single byte is all the attacker needs
- Ongoing arms race between defenders and attackers
 - Co-evolution of defenses and exploitation techniques

How are they introduced?

How are they introduced?

- No automatic bounds checking in C/C++

How are they introduced?

- No automatic bounds checking in C/C++
- The problem is made more acute by the fact many C stdlib functions make it easy to go past bounds
- String manipulation functions like `gets()`, `strcpy()`, and `strcat()` all write to the destination buffer until they encounter a terminating `'\0'` byte in the input

How are they introduced?

- No automatic bounds checking in C/C++
- The problem is made more acute by the fact many C stdlib functions make it easy to go past bounds
- String manipulation functions like `gets()`, `strcpy()`, and `strcat()` all write to the destination buffer until they encounter a terminating `'\0'` byte in the input
 - Whoever is providing the input (often from the other side of a security boundary) controls how much gets written


Example 1: spot the vuln!

```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

<http://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c>

Example 1: spot the vuln!


- What does gets() do?



```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

Example 1: spot the vuln!

- What does gets() do?
 - How many characters does it read in?
 - Who decides how much input to provide?



```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

Example 1: spot the vuln!

- What does gets() do?
 - How many characters does it read in?
 - Who decides how much input to provide?
- How large is line[]?

```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```

Example 1: spot the vuln!

- What does gets() do?
 - How many characters does it read in?
 - Who decides how much input to provide?
- How large is line[]?
 - Implicit assumption about input length

```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```


Example 1: spot the vuln!

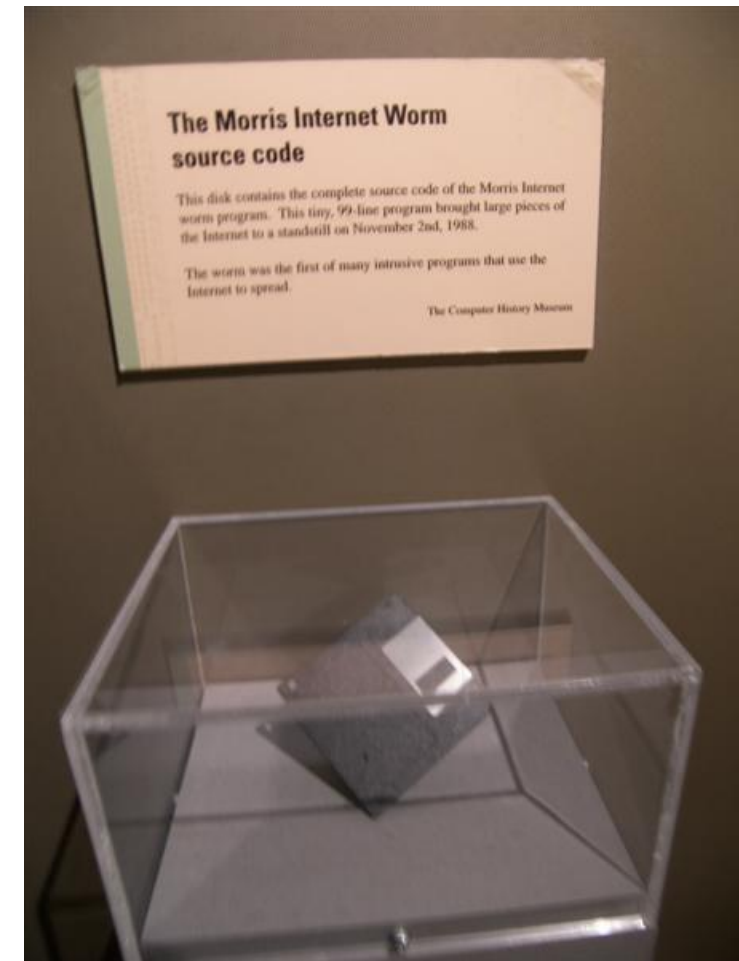
- What does gets() do?
 - How many characters does it read in?
 - Who decides how much input to provide?
- How large is line[]?
 - Implicit assumption about input length
- What happens if, say 536, characters are provided as input?

```
1 main(argc, argv)
2     char *argv[];
3 {
4     register char *sp;
5     char line[512];
6     struct sockaddr_in sin;
7     int i, p[2], pid, status;
8     FILE *fp;
9     char *av[4];
10
11     i = sizeof (sin);
12     if (getpeername(0, &sin, &i) < 0)
13         fatal(argv[0], "getpeername");
14     line[0] = '\0';
15     gets(line);
16     //...
17     return(0);
18 }
```


<http://minnie.tuhs.org/cgi-bin/utree.pl?file=4.3BSD/usr/src/etc/fingerd.c>

Morris worm


- This fingerd vulnerability was one of several exploited by the Morris Worm in 1988
 - Created by Robert Morris graduate student at Cornell
- One of the first Internet worms
 - Devastating effect on the Internet at the time
 - Took over hundreds of computers and shut down large chunks of the Internet
- Aside: First use of the US CFAA



But it's 2019... still a problem?

 chromium ▾ [New issue](#)

[Open issues](#) ▾

 [Sign in](#)

☆ Starred by 3 users

Owner: mcasas@chromium.org

CC: abdulsyed@chromium.org
adetaylor@chromium.org
m...@chromium.org
pabrai@chromium.org
mmoroz@chromium.org
guidou@chromium.org
webrtc-dev@chromium.org

Status: Fixed (Closed)

Components: [Blink>MediaRecording](#)

Modified: Jun 6, 2019

Editors: ----

EstimatedDays: ----

NextAction: ----

Notes: ----


OKR_link: ----

OS: [Linux](#), [Android](#), [Windows](#),
[Chrome](#), [Mac](#), [Fuchsia](#)

Pri: [2](#)

Type: [Bug-Security](#)

Hotlist-Merge-Review
reward-500
Security_Severity-Medium

Issue 930035: Security: Stack out-of-bounds writes in WebmMuxer::AddAudioTrack  [Code](#)

Reported by mlfbr...@stanford.edu on Thu, Feb 7, 2019, 9:30 PM PST

VULNERABILITY DETAILS
Security: Stack out-of-bounds writes in WebmMuxer::AddAudioTrack
I'm not certain that its triggerable, but it looks as if it should be fixed

REPRODUCTION CASE

- Stack allocated opus header in AddAudioTrack:

uint8_t opus_header[OPUS_EXTRADATA_SIZE]; // and OPUS_EXTRADATA_SIZE is 19
(see: https://cs.chromium.org/chromium/src/media/muxers/webm_muxer.cc?type=cs&q=WebmMuxer::AddAudioTrack&g=0&l=303)
- Function calls WriteOpusHeader(params, opus_header);
(function https://cs.chromium.org/chromium/src/media/muxers/webm_muxer.cc?l=20&gsn=WriteOpusHeader)
- WriteOpusHeader does:

header[OPUS_EXTRADATA_NUM_COUPLED_OFFSET] = 0;

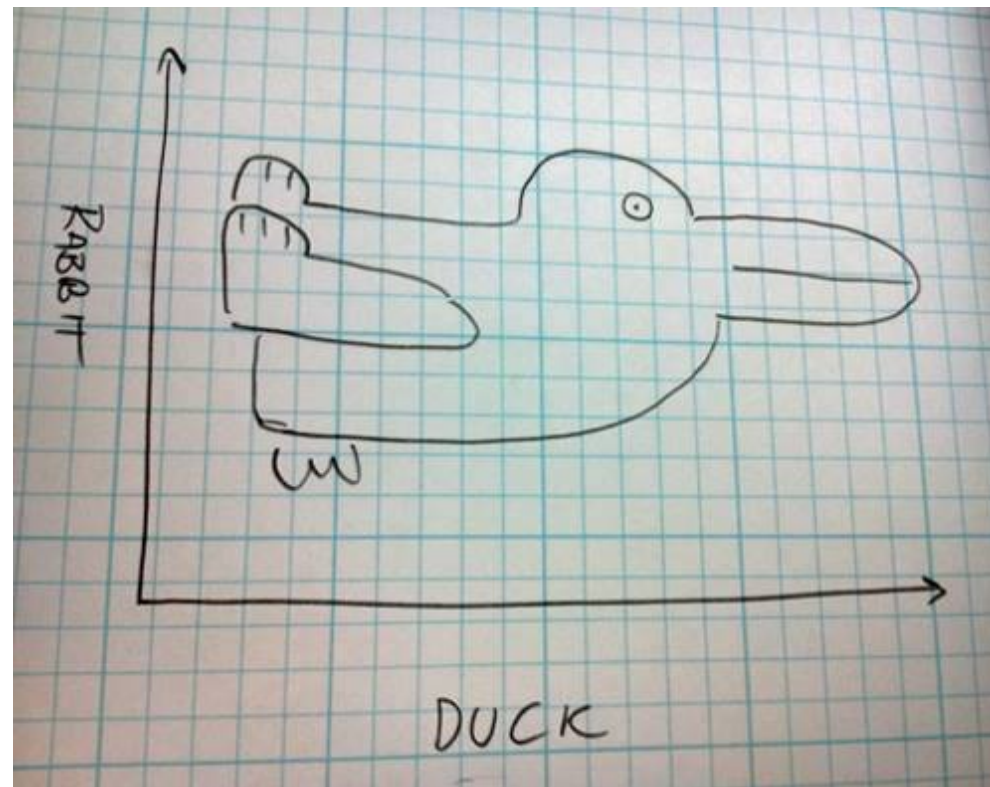
where
OPUS_EXTRADATA_NUM_STREAMS_OFFSET = OPUS_EXTRADATA_SIZE,
OPUS_EXTRADATA_NUM_COUPLED_OFFSET = OPUS_EXTRADATA_NUM_STREAMS_OFFSET + 1,
OPUS_EXTRADATA_STREAM_MAP_OFFSET = OPUS_EXTRADATA_NUM_STREAMS_OFFSET + 2
(allocated size was OPUS_EXTRADATA_SIZE)
- It continues to write out of bounds:
for (int i = 0; i < params.channels(); ++i) {
 header[OPUS_EXTRADATA_STREAM_MAP_OFFSET + i] =
 kOpusVorbisChannelMap[params.channels() - 1][i];
}

OK but...

- Why does overflowing a buffer let you take over a machine?
- That seems crazy no?

Changing perspectives

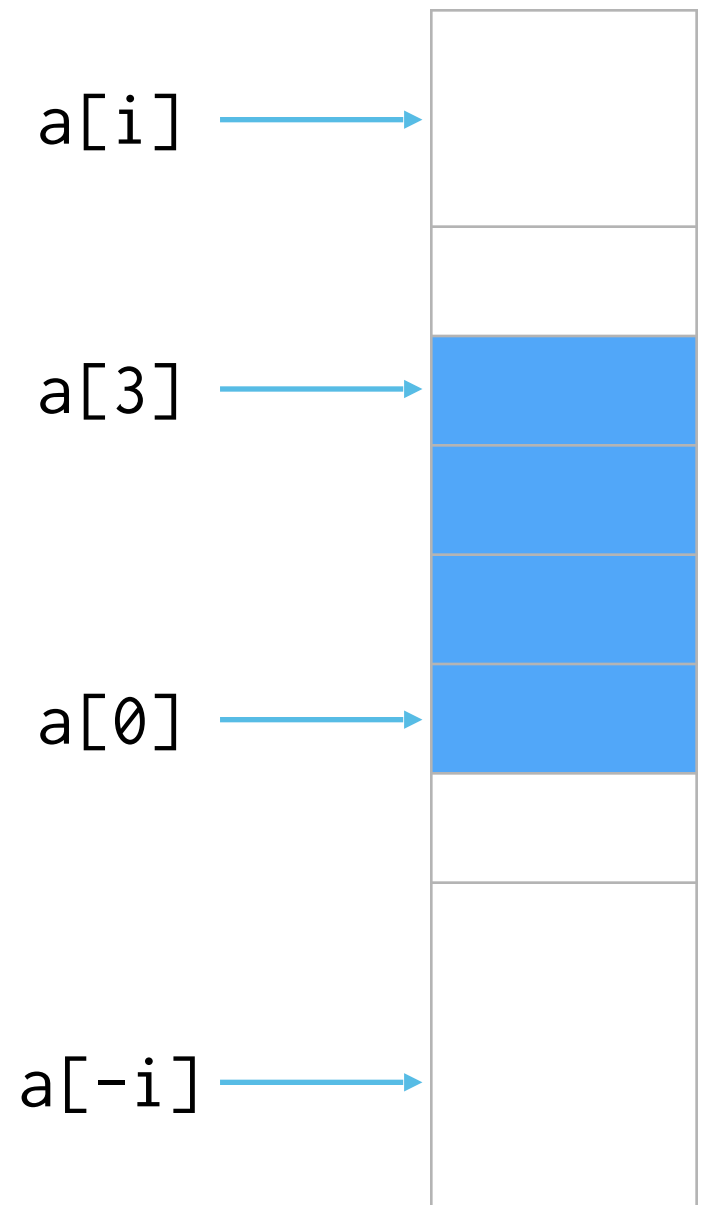
- Your program manipulates data
- Data manipulates your program



What we need to know

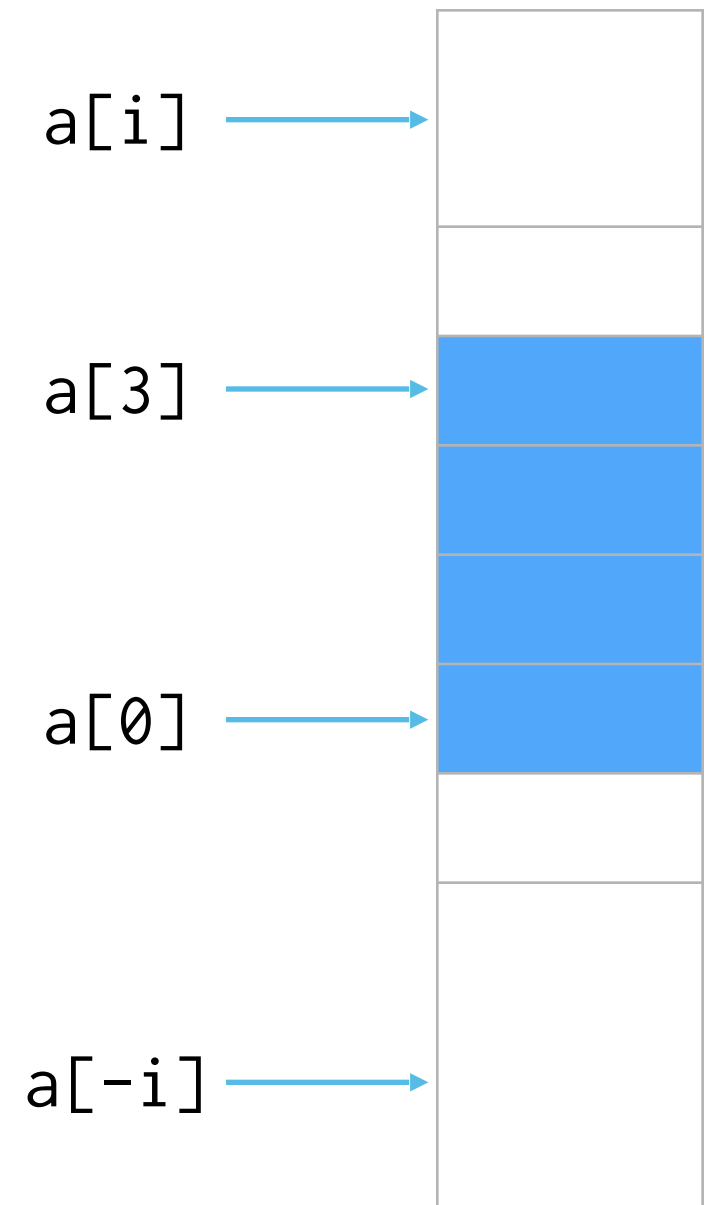
- How C arrays work
- How memory is laid out
- How function calls work
- How to turn an array overflow into an exploit

How does an array work?



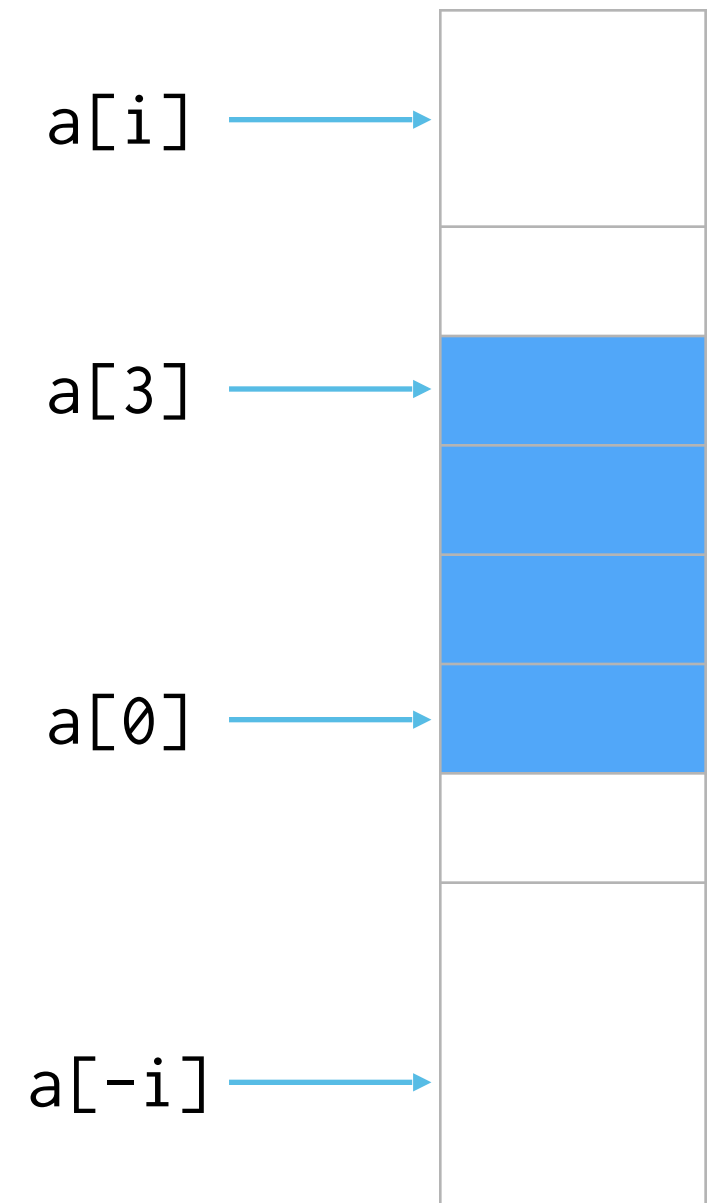
How does an array work?

- What's the abstraction?



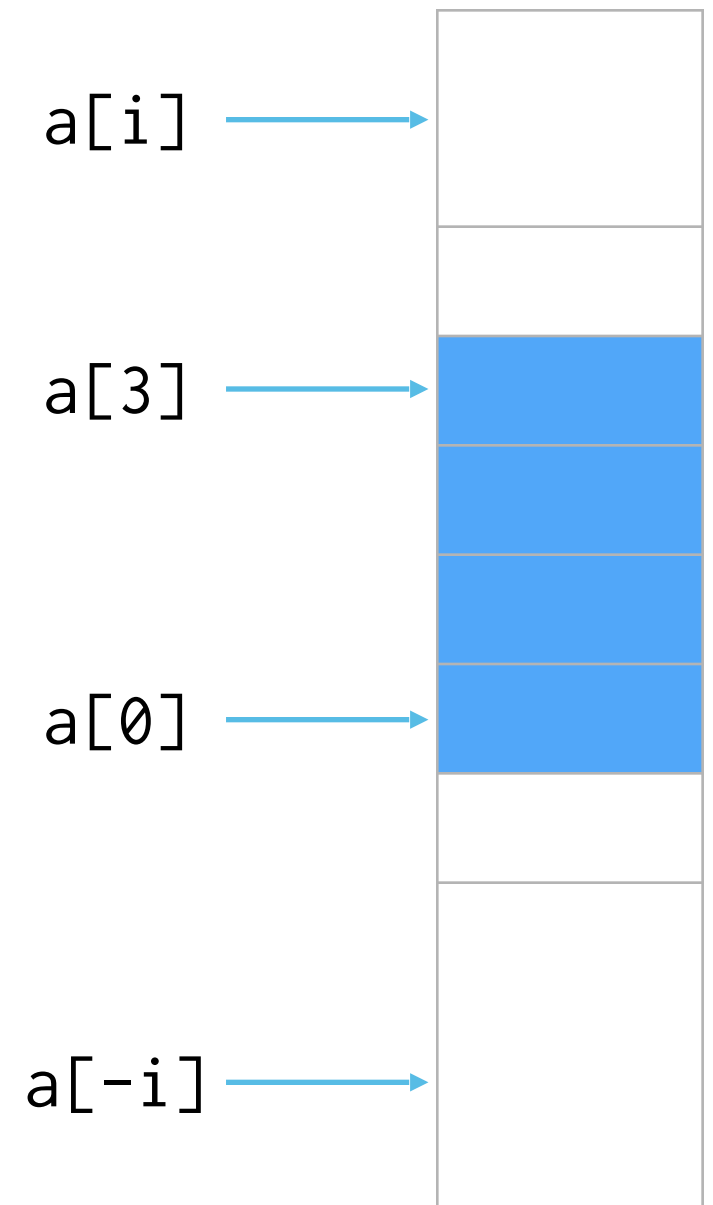
How does an array work?

- What's the abstraction?
- What's the reality?
 - What happens if you try to write past the end of an array in C/C++?



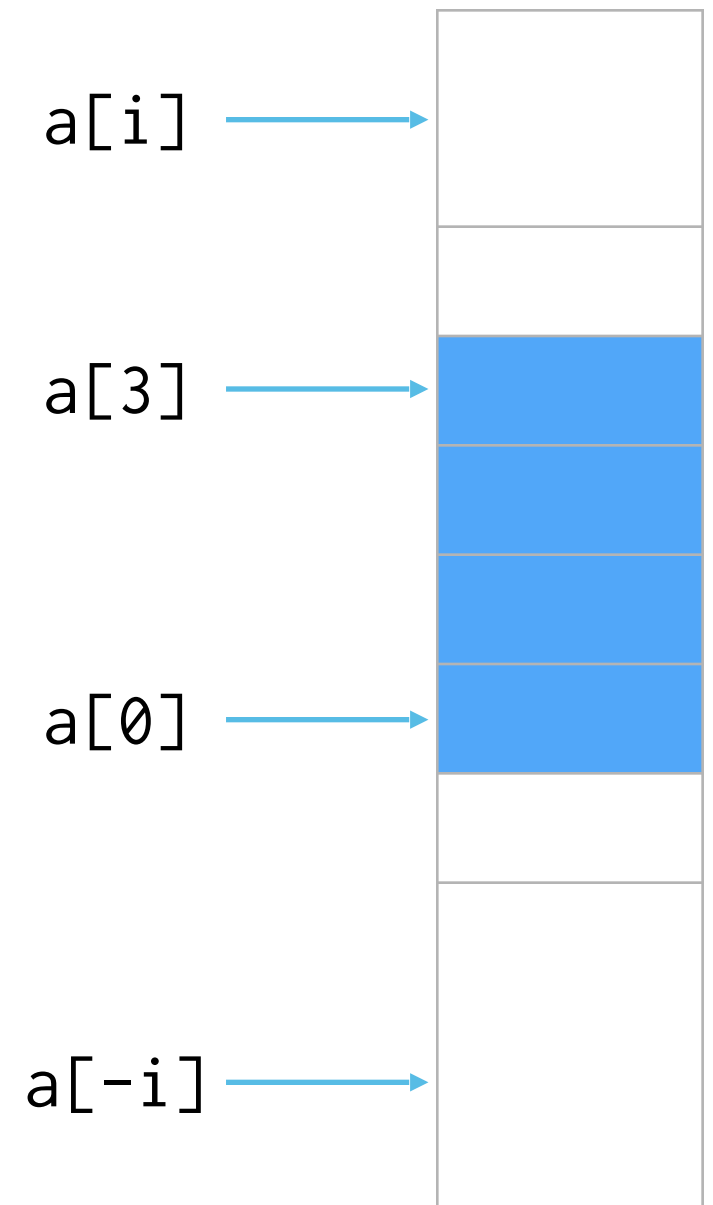
How does an array work?

- What's the abstraction?
- What's the reality?
 - What happens if you try to write past the end of an array in C/C++?
 - What does the language spec say?



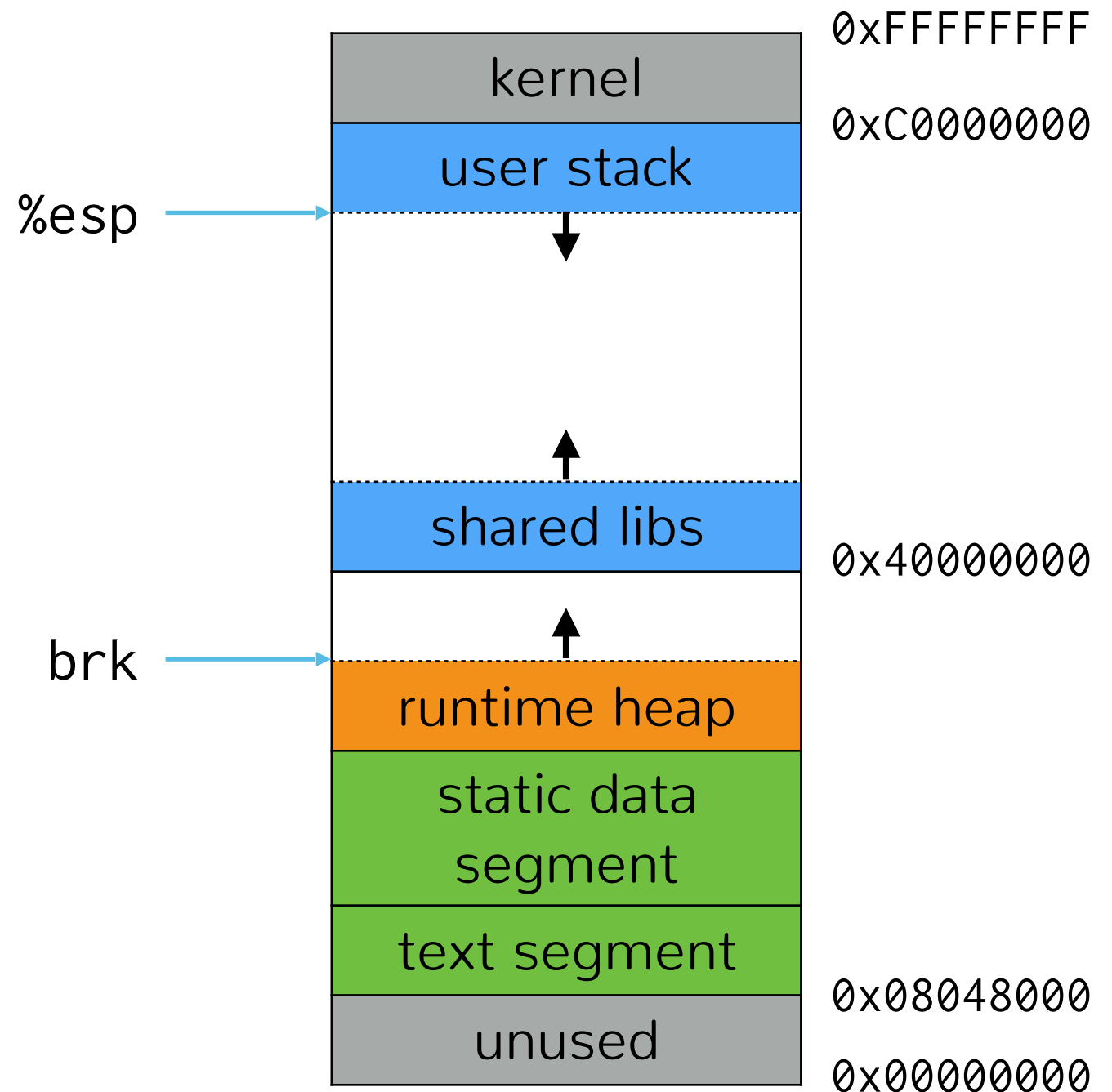
How does an array work?

- What's the abstraction?
- What's the reality?
 - What happens if you try to write past the end of an array in C/C++?
 - What does the language spec say?
 - What happens in most implementations?



Linux process memory layout

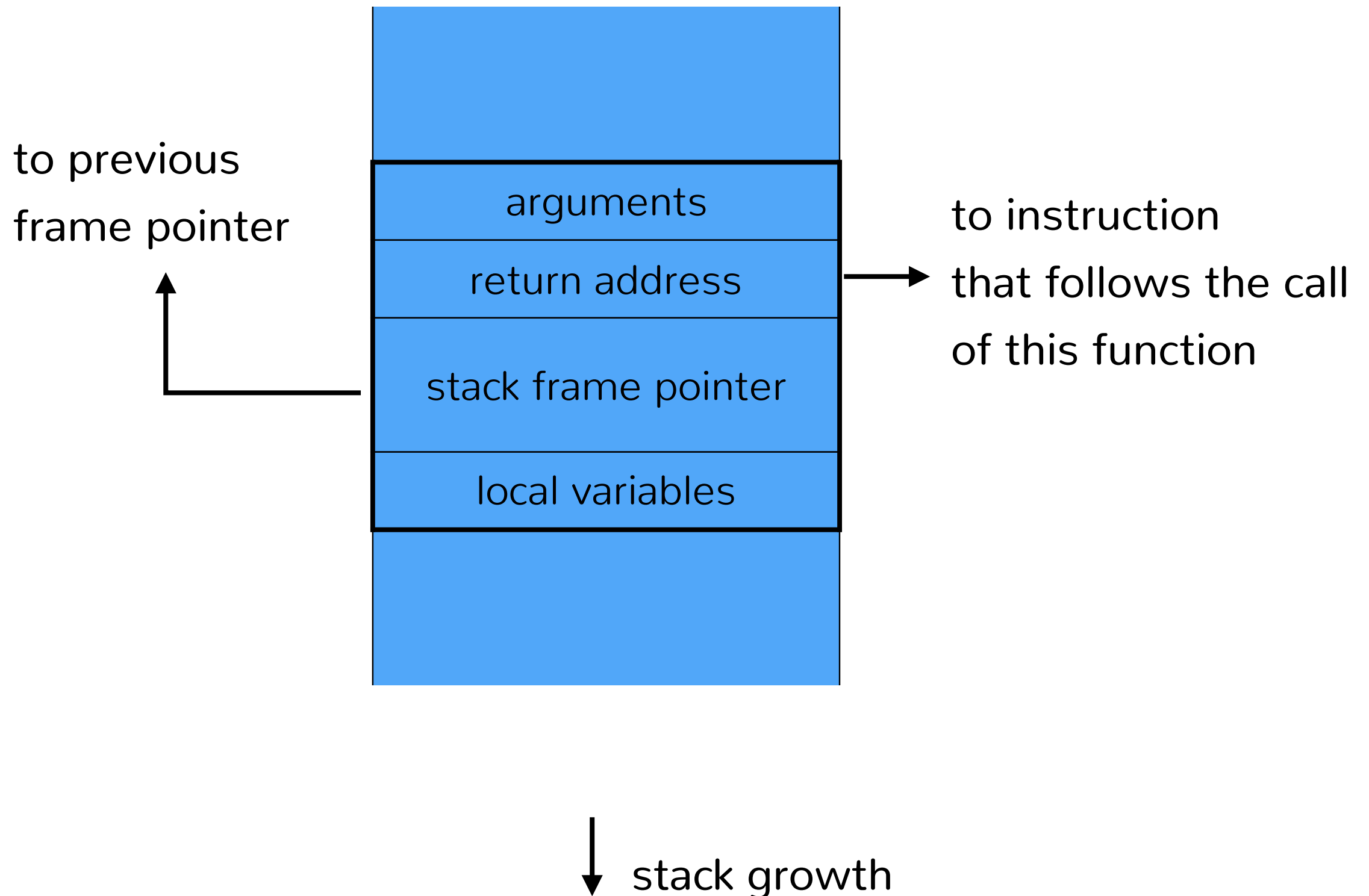
- Stack
 -
- Heap
 -
- Data segment
 - .data, .bss
- Text segment
 - Executable code



The Stack

- Stack divided into frames
 - Frame stores locals and args to called functions
- **Stack pointer** points to top of stack
 - x86: Stack grows down (from high to low addresses)
 - x86: Stored in %esp register
- **Frame pointer** points to caller's stack frame
 - Also called base pointer
 - x86: Stored in %ebp register

Stack frame



Example 0

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;

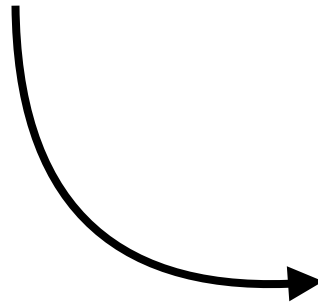
    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

<https://godbolt.org/z/3iFhjy>

Compiled to x86

```
1 int foobar(int a, int b, int c)
2 {
3     int xx = a + 2;
4     int yy = b + 3;
5     int zz = c + 4;
6     int sum = xx + yy + zz;
7
8     return xx * yy * zz + sum;
9 }
10
11 int main()
12 {
13     return foobar(77, 88, 99);
14 }
```



```
1 foobar(int, int, int):
2     pushl   %ebp
3     movl    %esp, %ebp
4     subl    $16, %esp
5     movl    8(%ebp), %eax
6     addl    $2, %eax
7     movl    %eax, -4(%ebp)
8     movl    12(%ebp), %eax
9     addl    $3, %eax
10    movl    %eax, -8(%ebp)
11    movl    16(%ebp), %eax
12    addl    $4, %eax
13    movl    %eax, -12(%ebp)
14    movl    -4(%ebp), %edx
15    movl    -8(%ebp), %eax
16    addl    %eax, %edx
17    movl    -12(%ebp), %eax
18    addl    %edx, %eax
19    movl    %eax, -16(%ebp)
20    movl    -4(%ebp), %eax
21    imull    -8(%ebp), %eax
22    imull    -12(%ebp), %eax
23    movl    %eax, %edx
24    movl    -16(%ebp), %eax
25    addl    %edx, %eax
26    leave
27    ret
28
29 main:
30     pushl   %ebp
31     movl    %esp, %ebp
32     pushl   $99
33     pushl   $88
34     pushl   $77
35     call    foobar(int, int, int)
36     addl    $12, %esp
37     nop
38     leave
39     ret
```

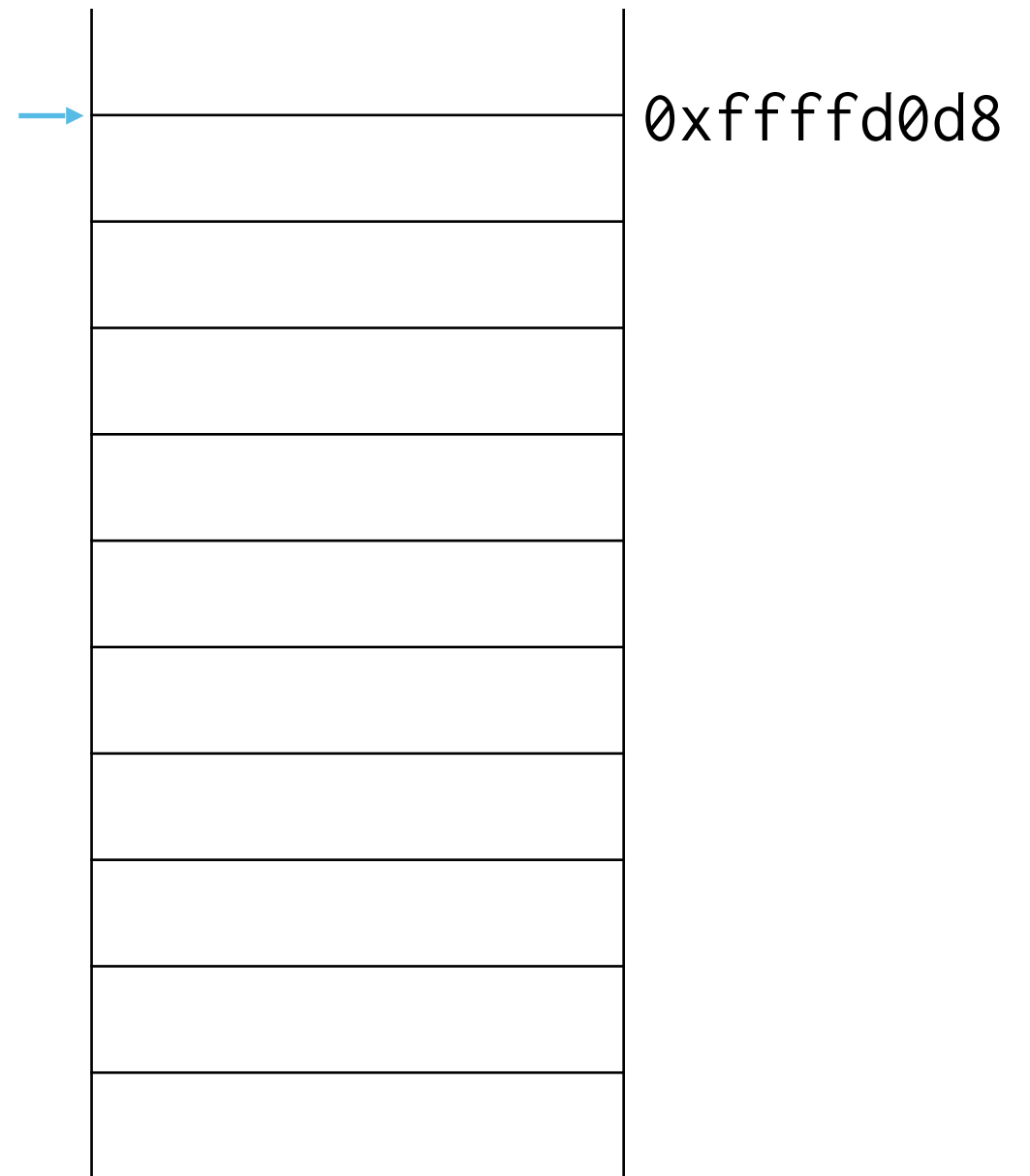


```

1  foobar(int, int, int):
2      pushl    %ebp
3      movl     %esp, %ebp
4      subl     $16, %esp
5      movl     8(%ebp), %eax
6      addl     $2, %eax
7      movl     %eax, -4(%ebp)
8      movl     12(%ebp), %eax
9      addl     $3, %eax
10     movl     %eax, -8(%ebp)
11     movl     16(%ebp), %eax
12     addl     $4, %eax
13     movl     %eax, -12(%ebp)
14     movl     -4(%ebp), %edx
15     movl     -8(%ebp), %eax
16     addl     %eax, %edx
17     movl     -12(%ebp), %eax
18     addl     %edx, %eax
19     movl     %eax, -16(%ebp)
20     movl     -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl     %eax, %edx
24     movl     -16(%ebp), %eax
25     addl     %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

```

%esp, %ebp →

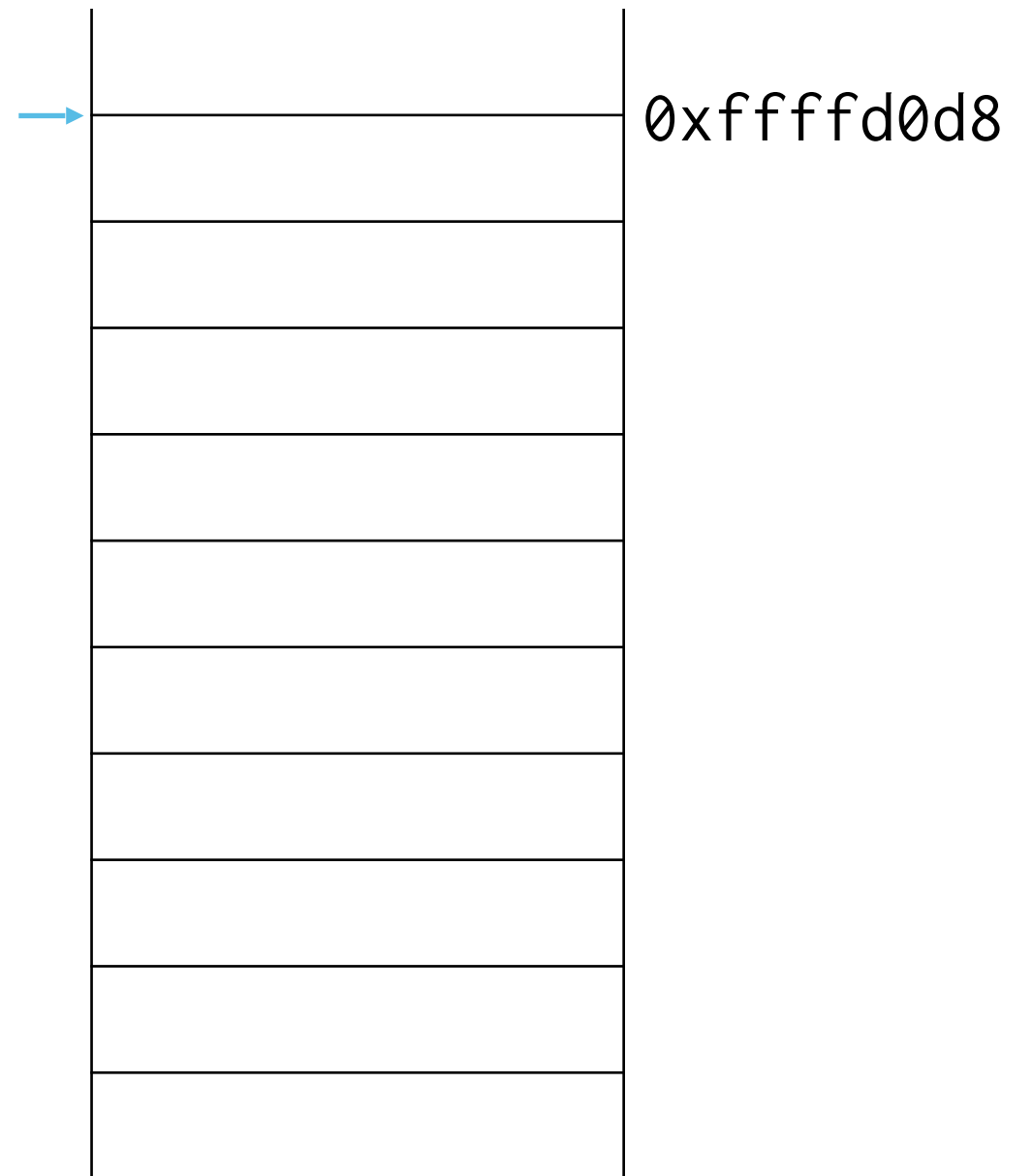


```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29      pushl   %ebp
30      movl    %esp, %ebp
31      → pushl   $99
32      pushl   $88
33      pushl   $77
34      call    foobar(int, int, int)
35      addl    $12, %esp
36      nop
37      leave
38      ret

```

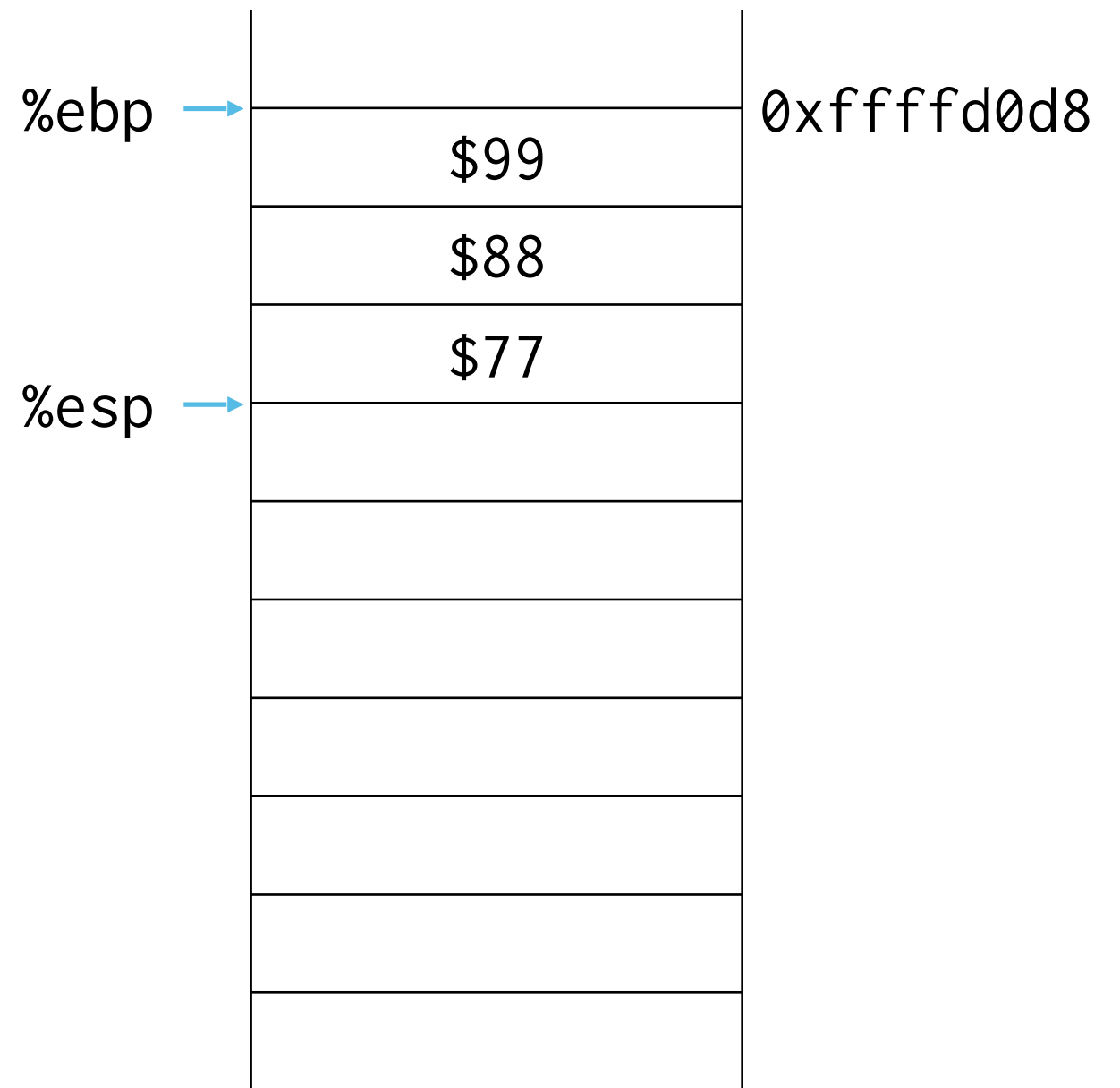
%esp, %ebp →




```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29      pushl   %ebp
30      movl    %esp, %ebp
31      pushl   $99
32      pushl   $88
33      → pushl   $77
34      call    foobar(int, int, int)
35      addl    $12, %esp
36      nop
37      leave
38      ret

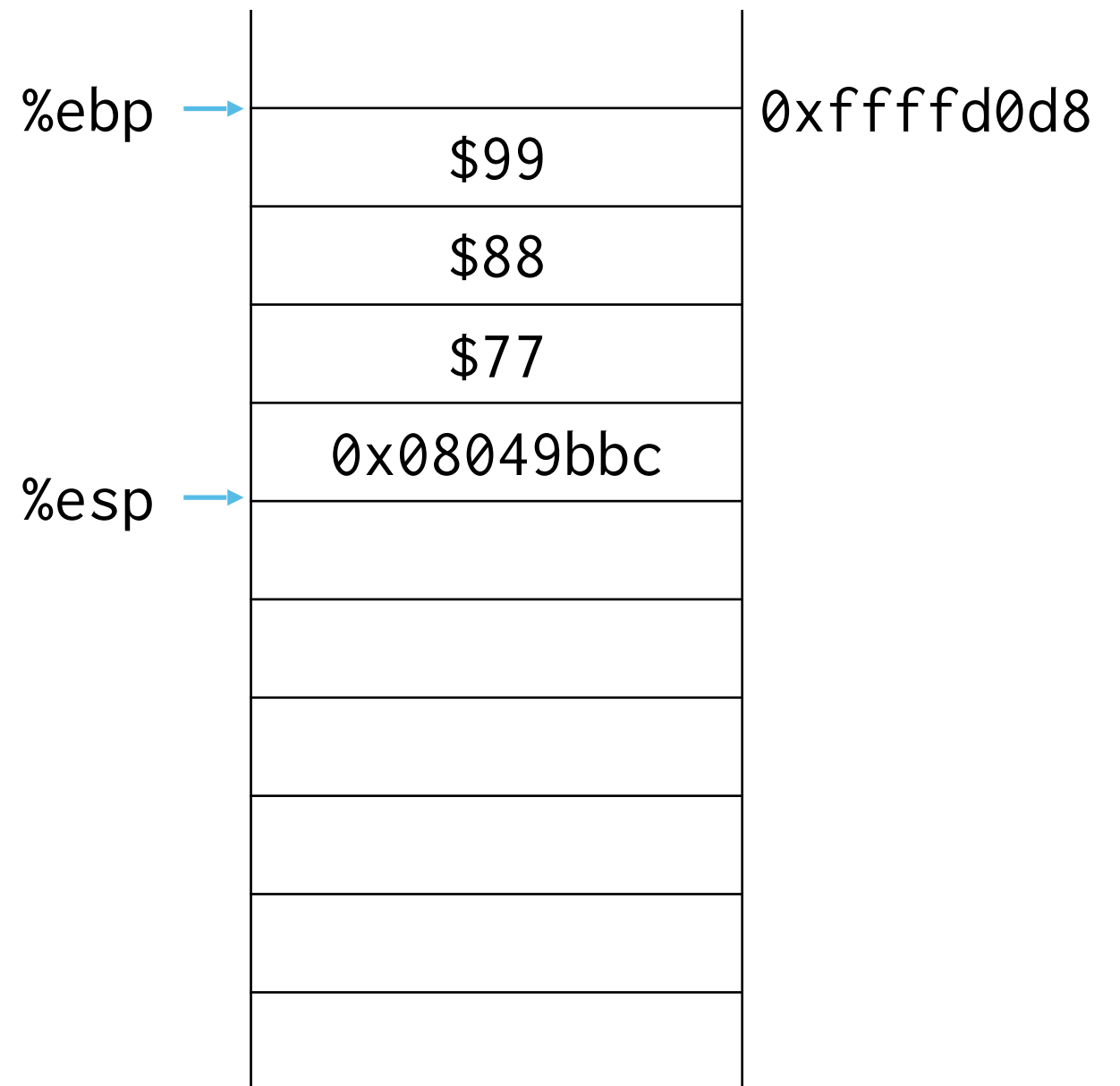
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29      pushl   %ebp
30      movl    %esp, %ebp
31      pushl   $99
32      pushl   $88
33      pushl   $77
34      → call   foobar(int, int, int)
35      addl    $12, %esp
36      nop
37      leave
38      ret

```



%eip = 0x08049ba7

```

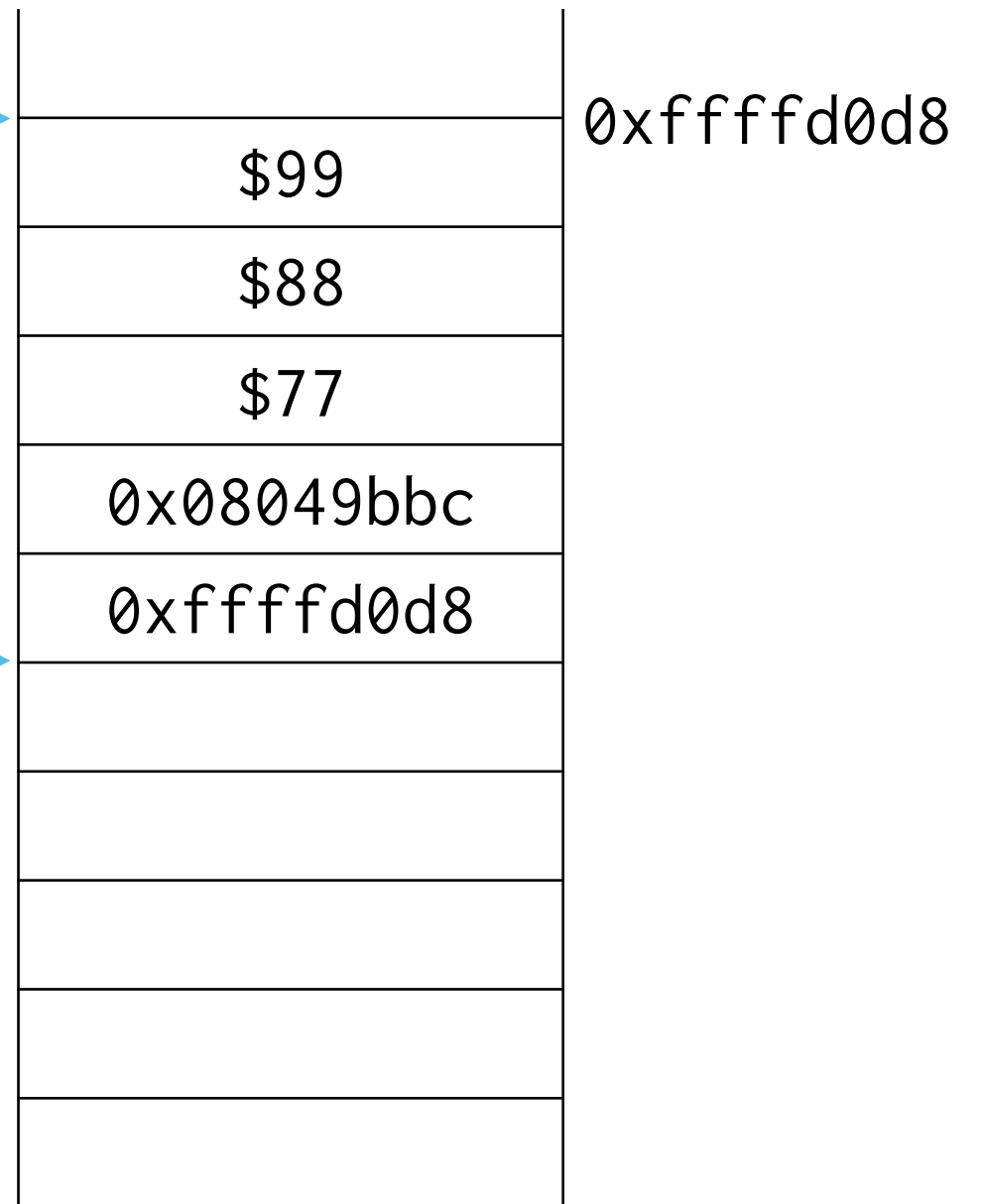
1  foobar(int, int, int):
2  → pushl   %ebp
3     movl   %esp, %ebp
4     subl   $16, %esp
5     movl   8(%ebp), %eax
6     addl   $2, %eax
7     movl   %eax, -4(%ebp)
8     movl   12(%ebp), %eax
9     addl   $3, %eax
10    movl   %eax, -8(%ebp)
11    movl   16(%ebp), %eax
12    addl   $4, %eax
13    movl   %eax, -12(%ebp)
14    movl   -4(%ebp), %edx
15    movl   -8(%ebp), %eax
16    addl   %eax, %edx
17    movl   -12(%ebp), %eax
18    addl   %edx, %eax
19    movl   %eax, -16(%ebp)
20    movl   -4(%ebp), %eax
21    imull   -8(%ebp), %eax
22    imull   -12(%ebp), %eax
23    movl   %eax, %edx
24    movl   -16(%ebp), %eax
25    addl   %edx, %eax
26    leave
27    ret
28  main:
29     pushl   %ebp
30     movl   %esp, %ebp
31     pushl   $99
32     pushl   $88
33     pushl   $77
34     call    foobar(int, int, int)
35     addl    $12, %esp
36     nop
37     leave
38     ret

```

%ebp →



%esp →

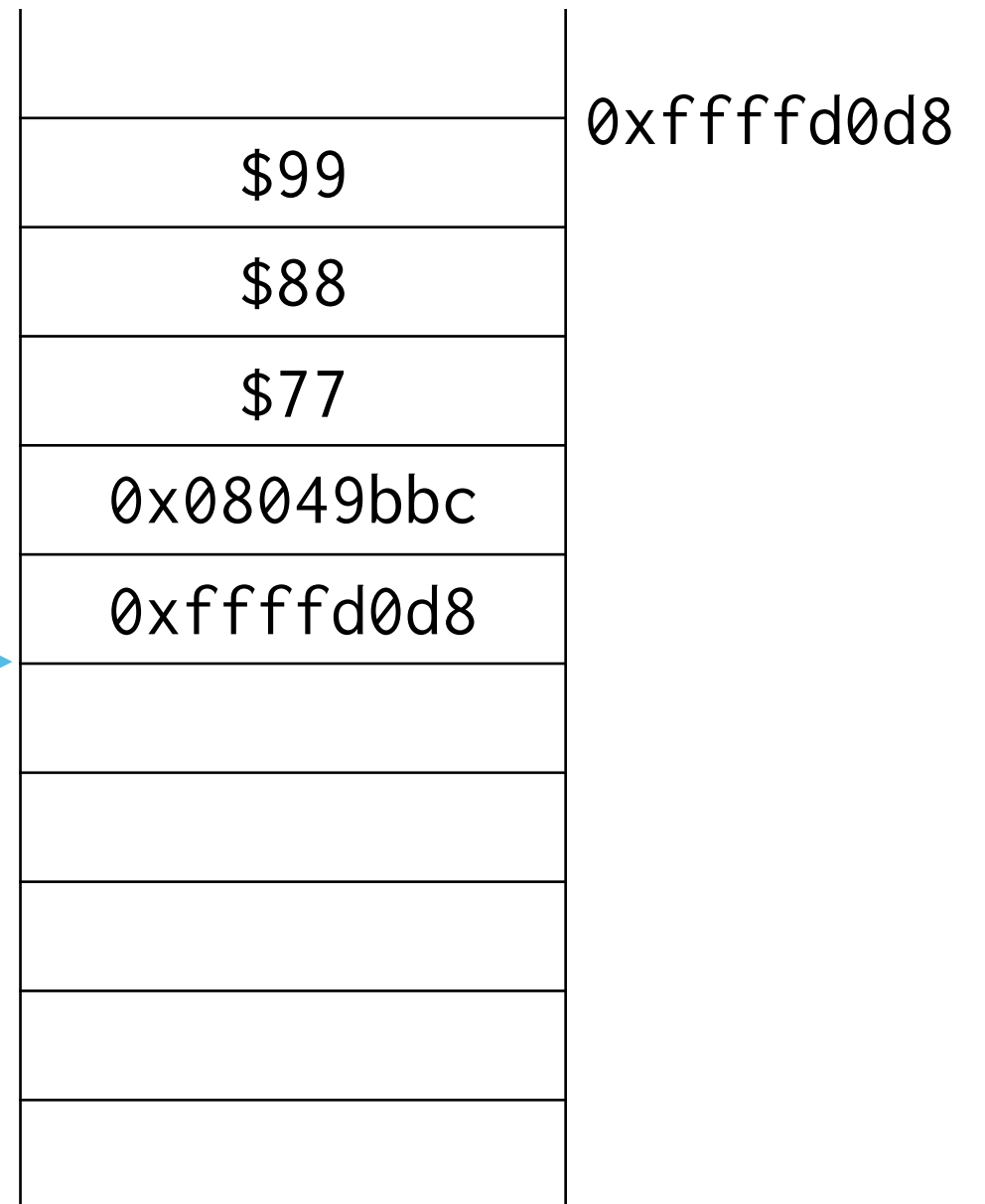


```

1  foobar(int, int, int):
2      pushl    %ebp
3      → movl    %esp, %ebp
4      subl    $16, %esp
5      movl     8(%ebp), %eax
6      addl     $2, %eax
7      movl     %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl     $3, %eax
10     movl     %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl     $4, %eax
13     movl     %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl     %eax, %edx
17     movl    -12(%ebp), %eax
18     addl     %edx, %eax
19     movl     %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl     %eax, %edx
24     movl    -16(%ebp), %eax
25     addl     %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

```

%esp, %ebp →



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      → subl   $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29      pushl   %ebp
30      movl    %esp, %ebp
31      pushl   $99
32      pushl   $88
33      pushl   $77
34      call    foobar(int, int, int)
35      addl    $12, %esp
36      nop
37      leave
38      ret

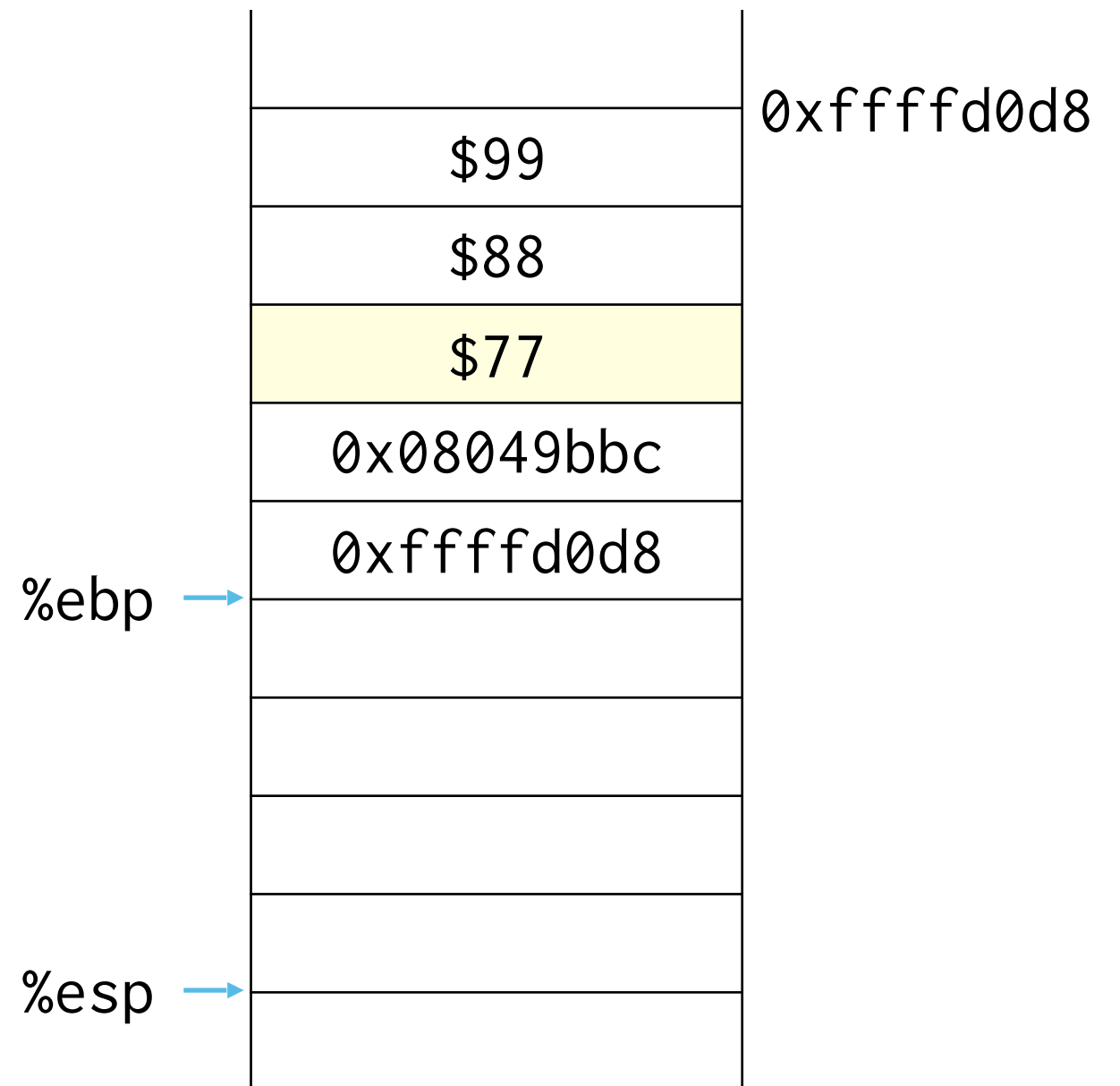
```




```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      → movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

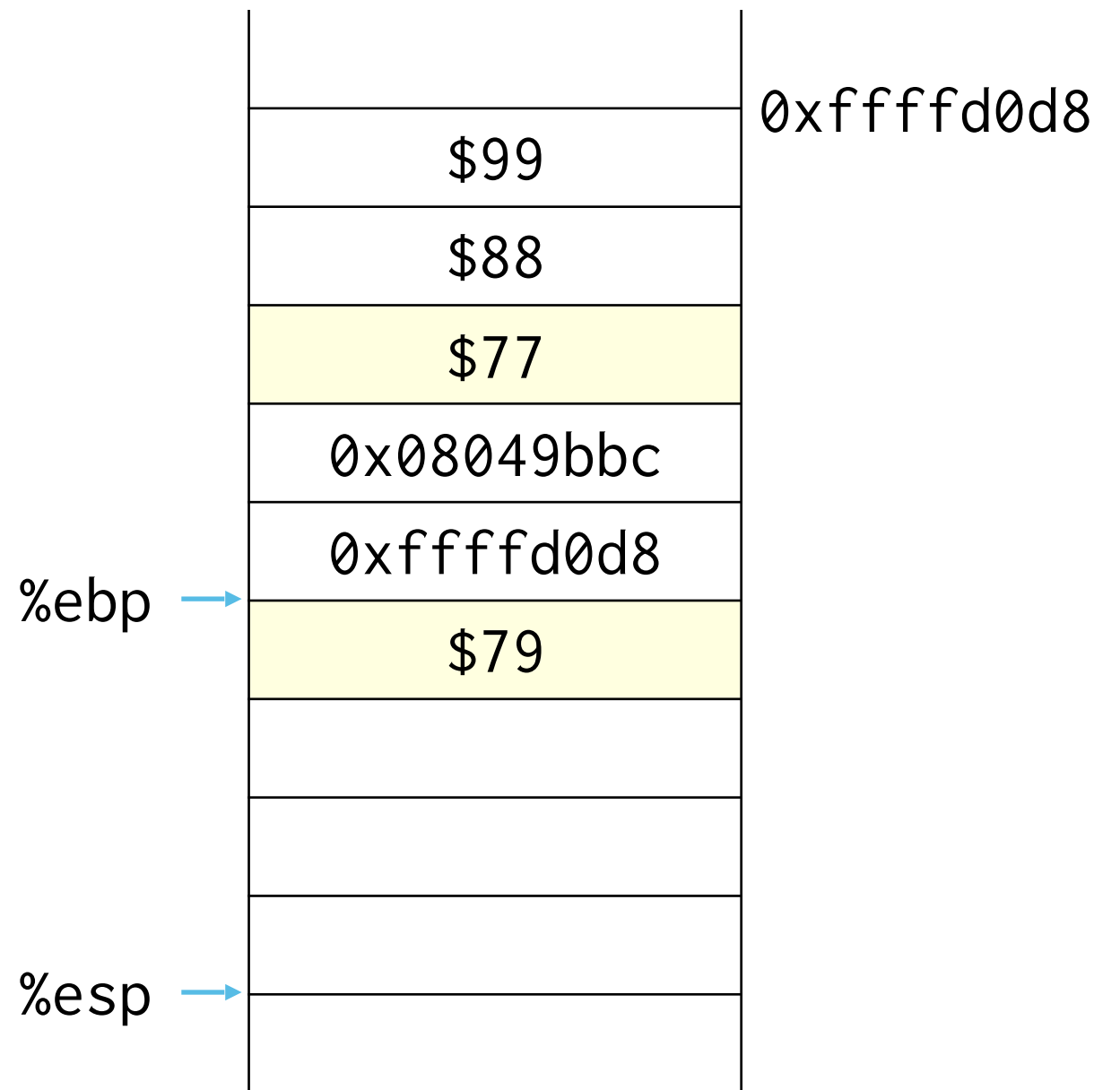
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      → movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

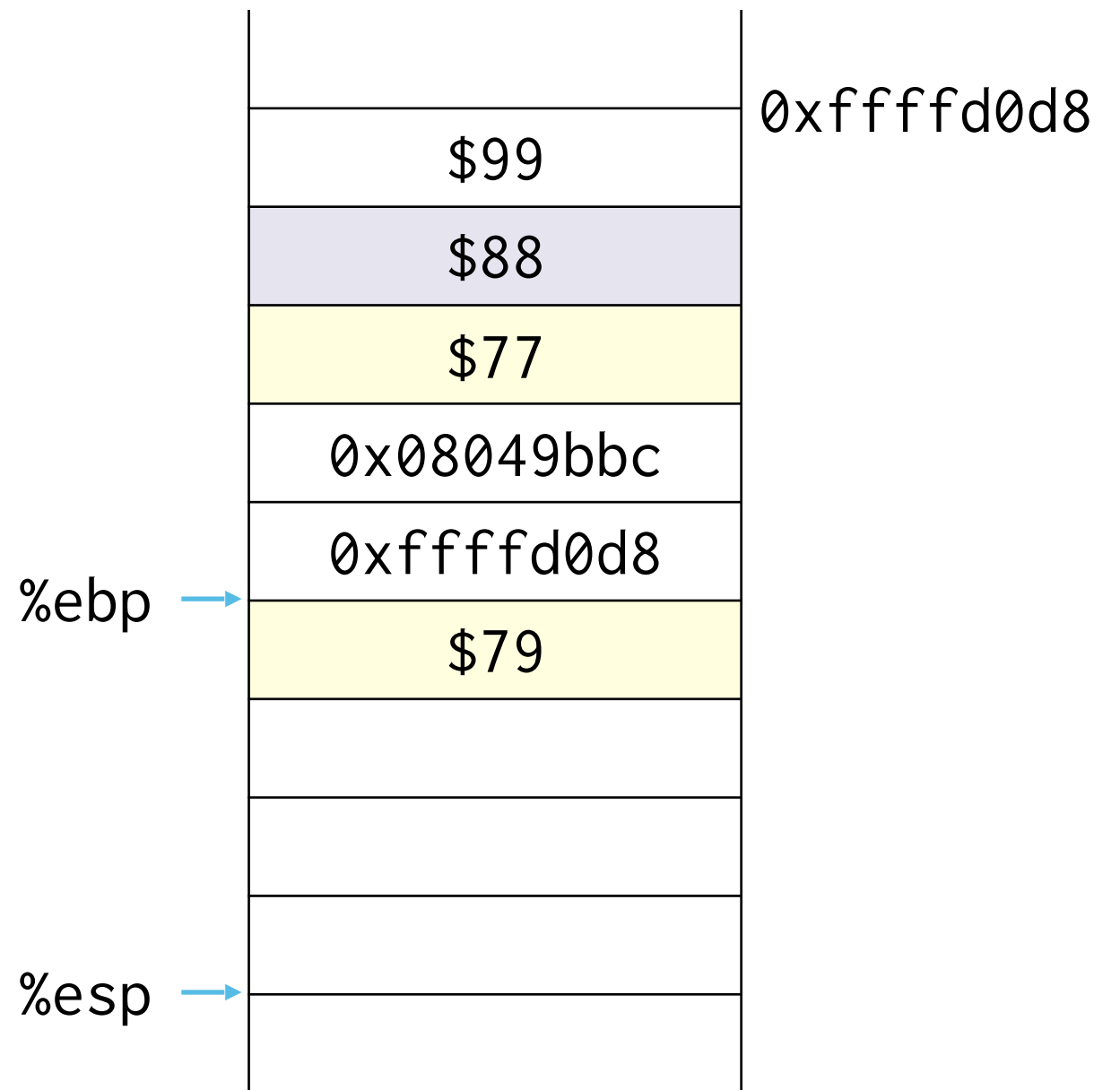
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      → movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

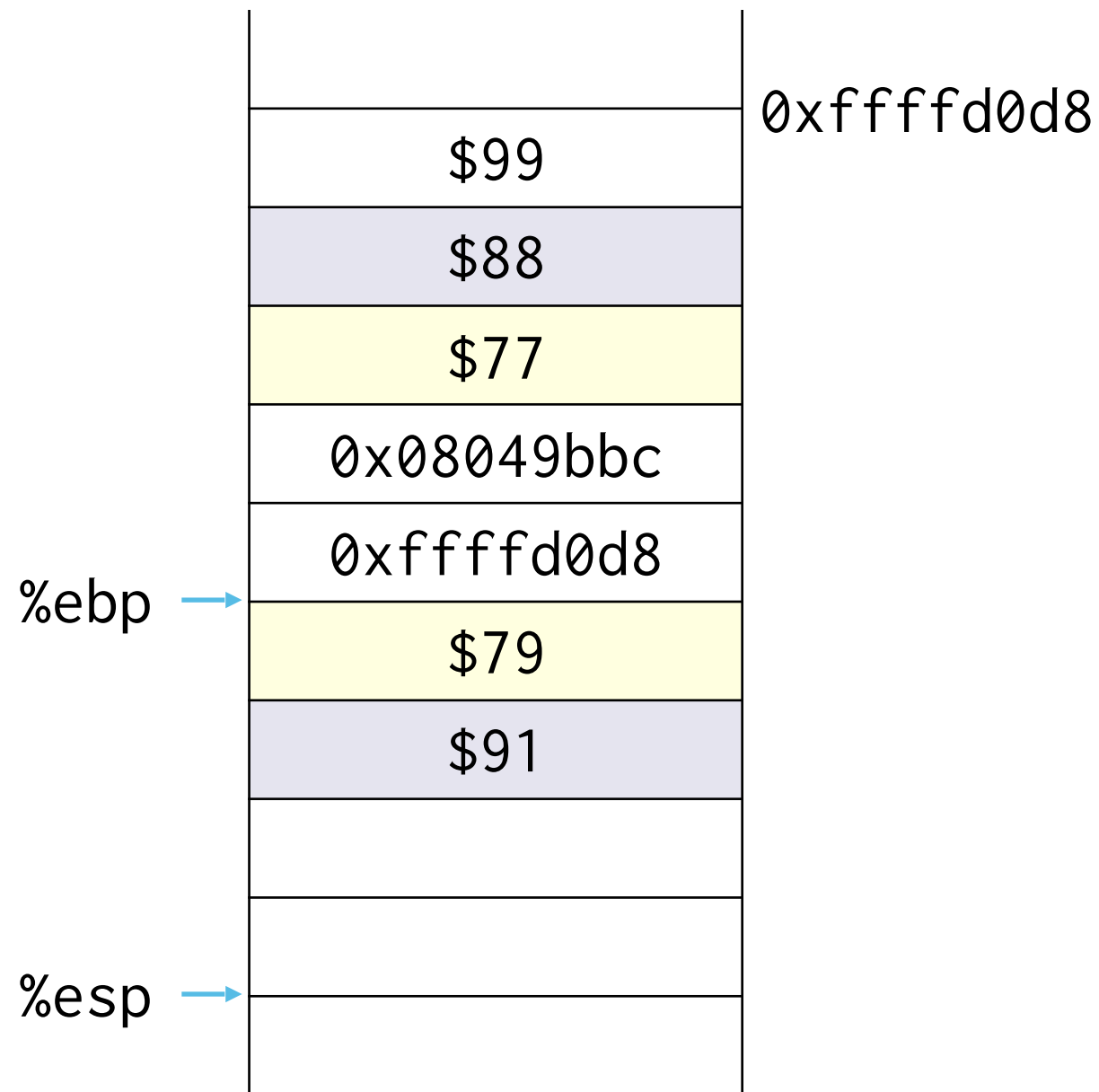
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     → movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29      pushl   %ebp
30      movl    %esp, %ebp
31      pushl   $99
32      pushl   $88
33      pushl   $77
34      call    foobar(int, int, int)
35      addl    $12, %esp
36      nop
37      leave
38      ret

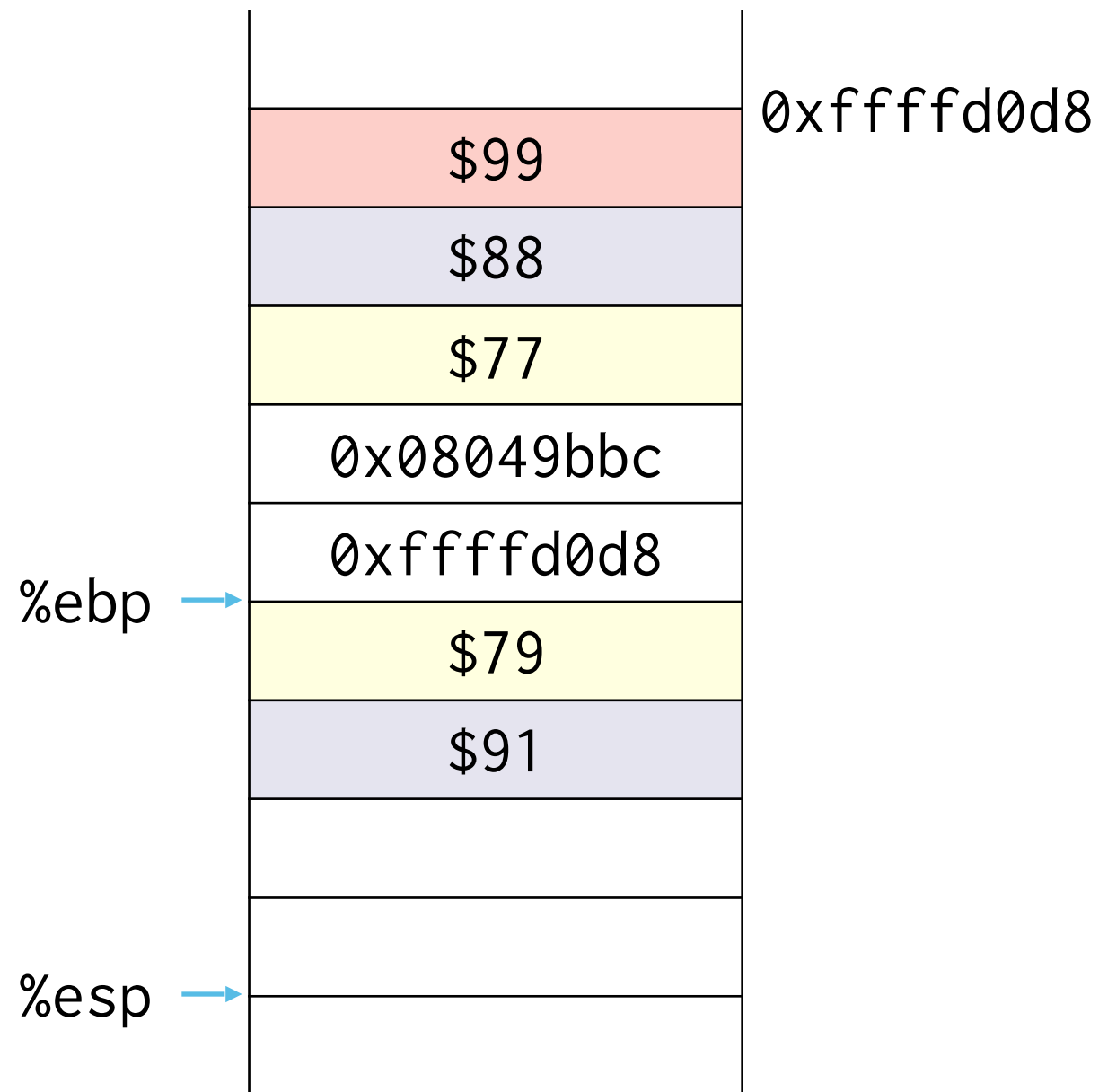
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     → movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

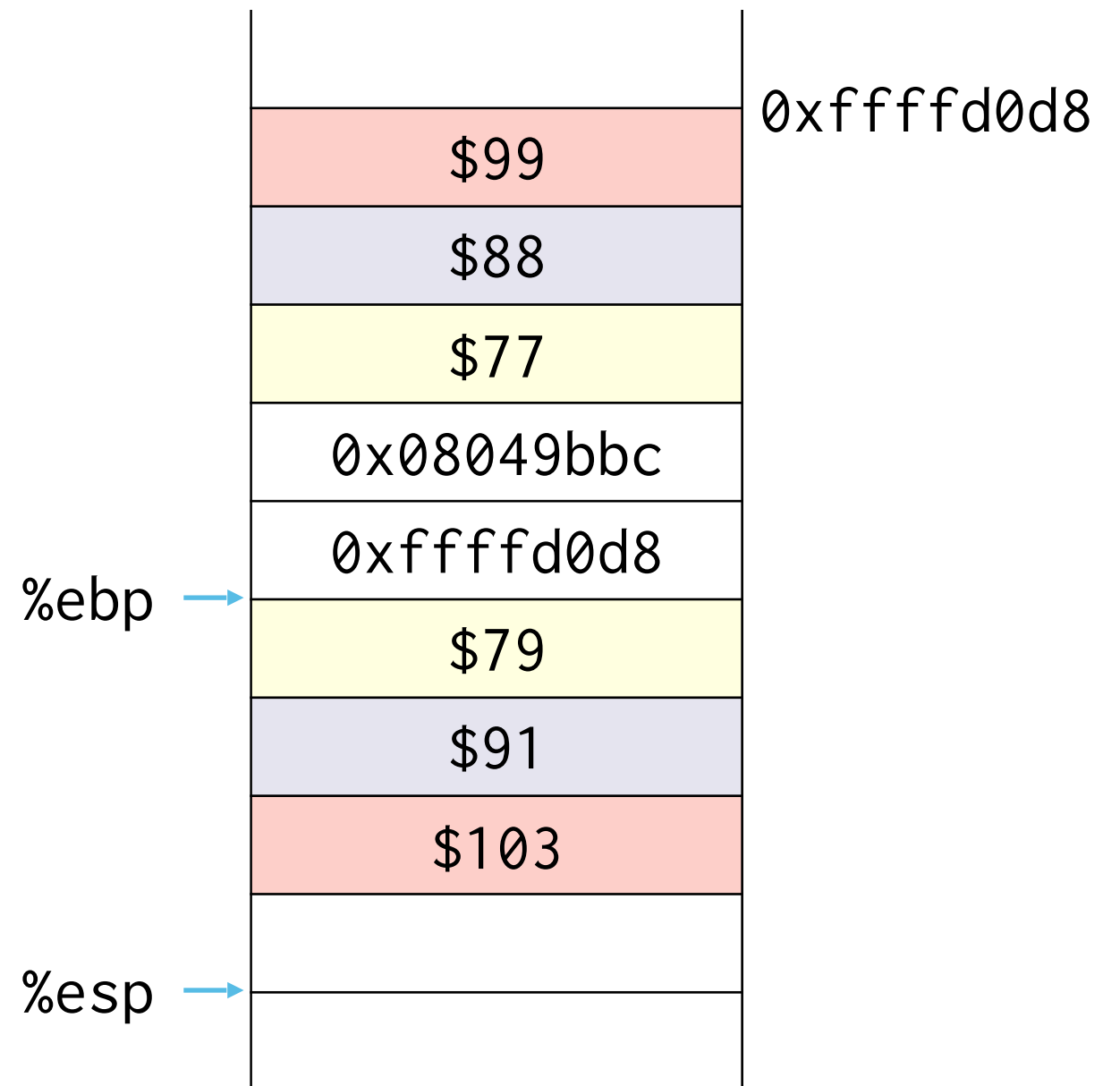
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     → movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

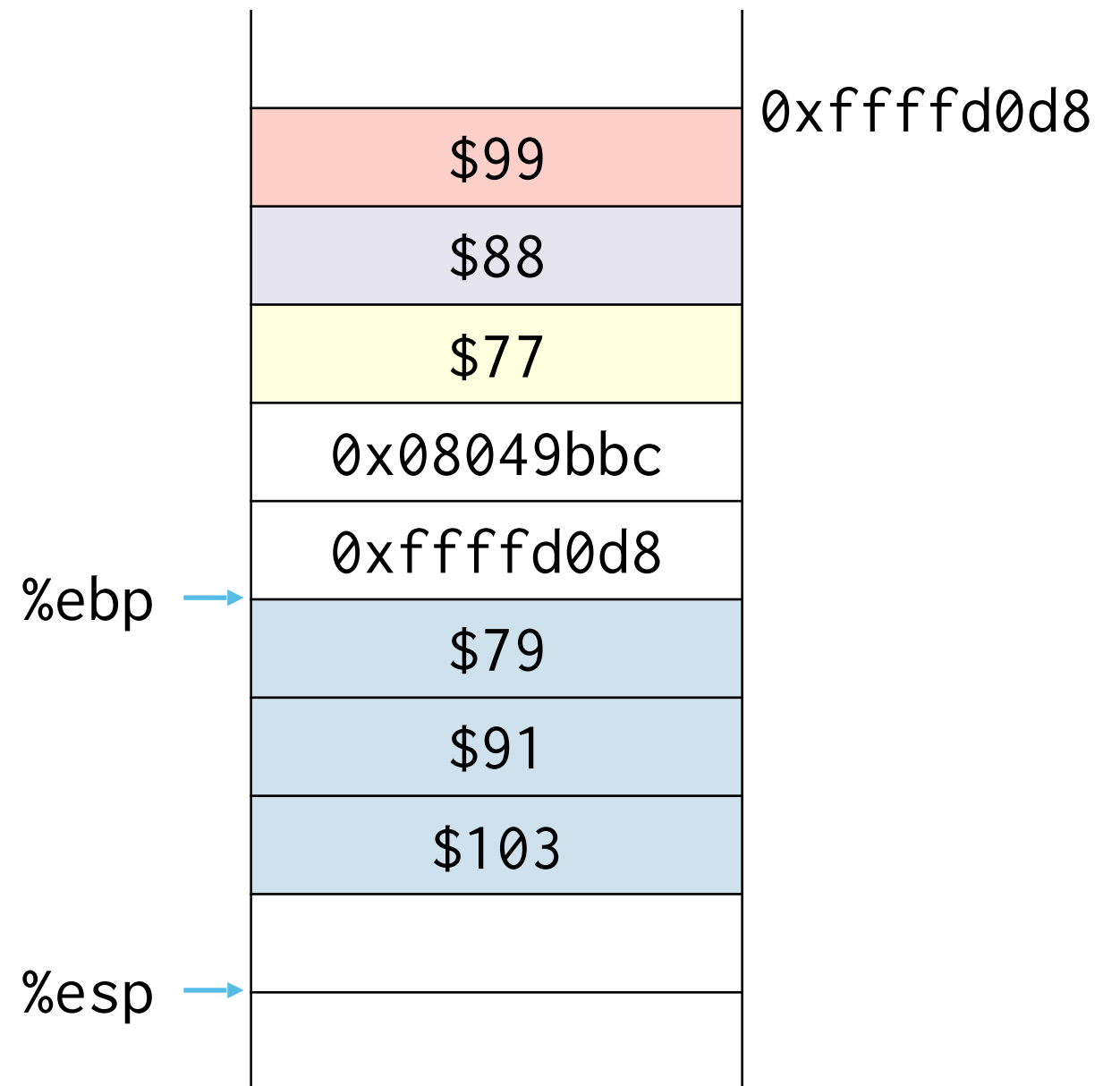
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     → movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

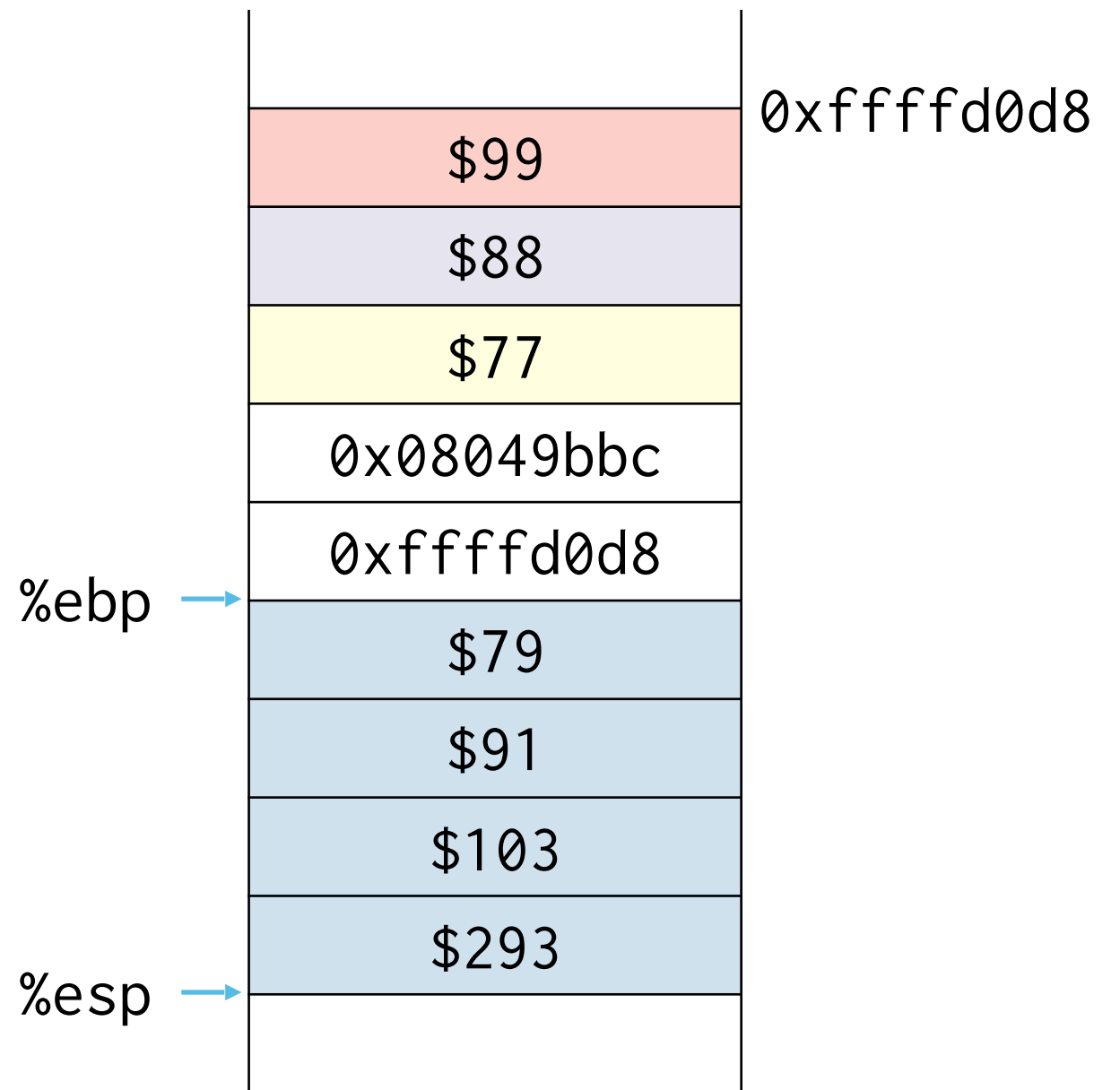
```



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     → movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

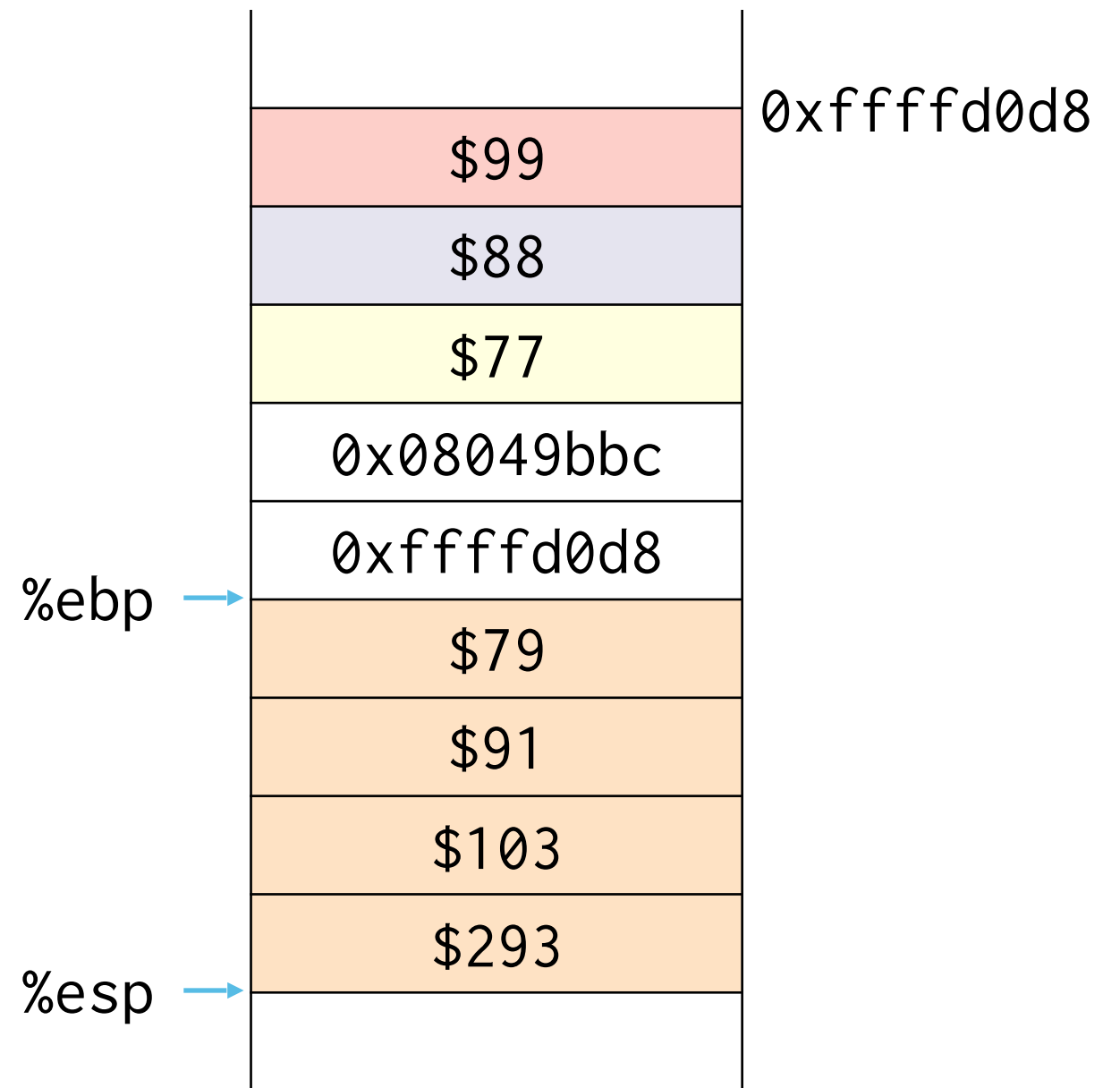
```




```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl   %esp, %ebp
4      subl   $16, %esp
5      movl   8(%ebp), %eax
6      addl   $2, %eax
7      movl   %eax, -4(%ebp)
8      movl   12(%ebp), %eax
9      addl   $3, %eax
10     movl   %eax, -8(%ebp)
11     movl   16(%ebp), %eax
12     addl   $4, %eax
13     movl   %eax, -12(%ebp)
14     movl   -4(%ebp), %edx
15     movl   -8(%ebp), %eax
16     addl   %eax, %edx
17     movl   -12(%ebp), %eax
18     addl   %edx, %eax
19     movl   %eax, -16(%ebp)
20     movl   -4(%ebp), %eax
21     imull   -8(%ebp), %eax
22     imull   -12(%ebp), %eax
23     movl   %eax, %edx
24     movl   -16(%ebp), %eax
25     → addl   %edx, %eax
26     leave
27     ret
28 main:
29     pushl   %ebp
30     movl   %esp, %ebp
31     pushl   $99
32     pushl   $88
33     pushl   $77
34     call    foobar(int, int, int)
35     addl   $12, %esp
36     nop
37     leave
38     ret

```

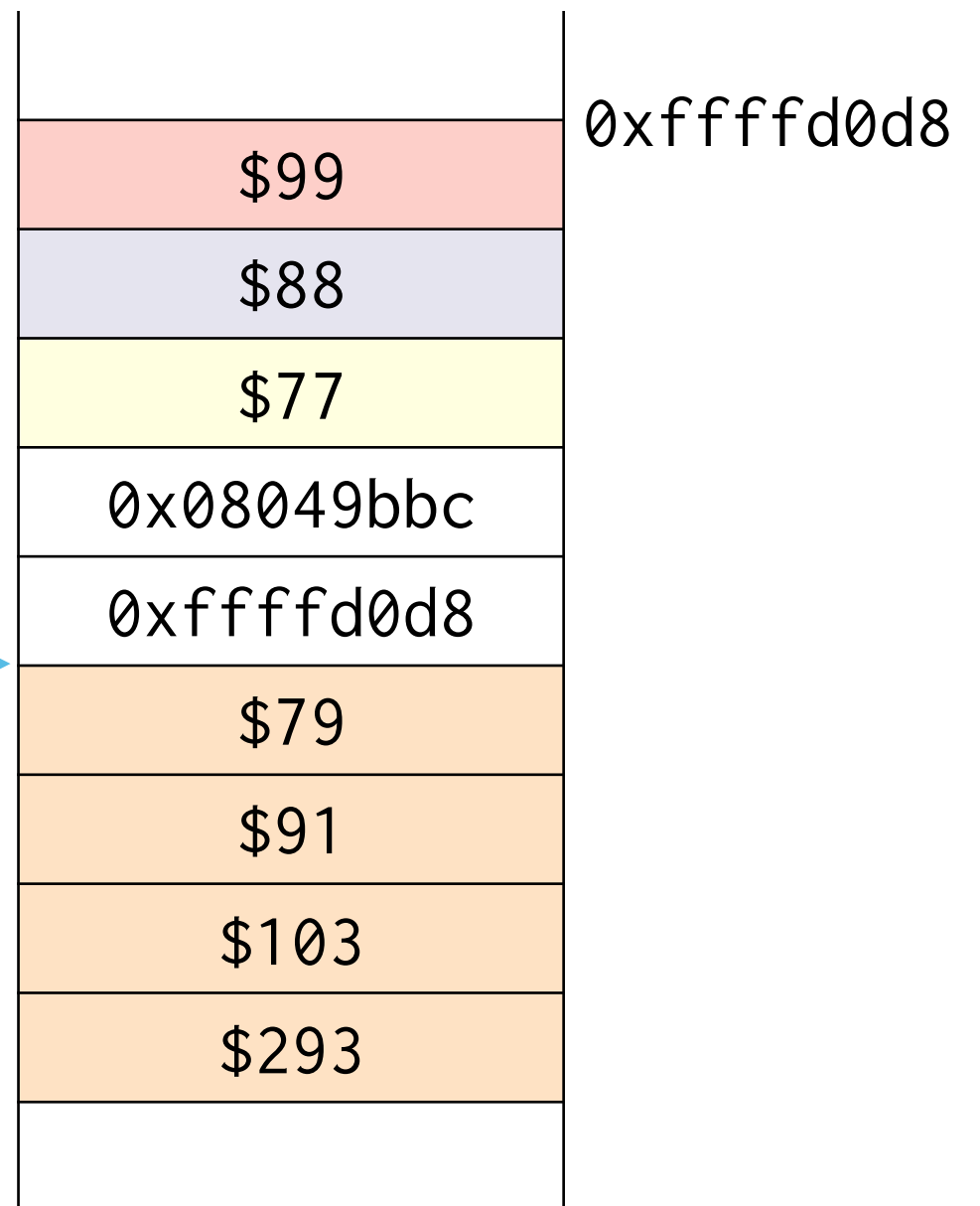


```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull   -8(%ebp), %eax
22     imull   -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     → leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

```

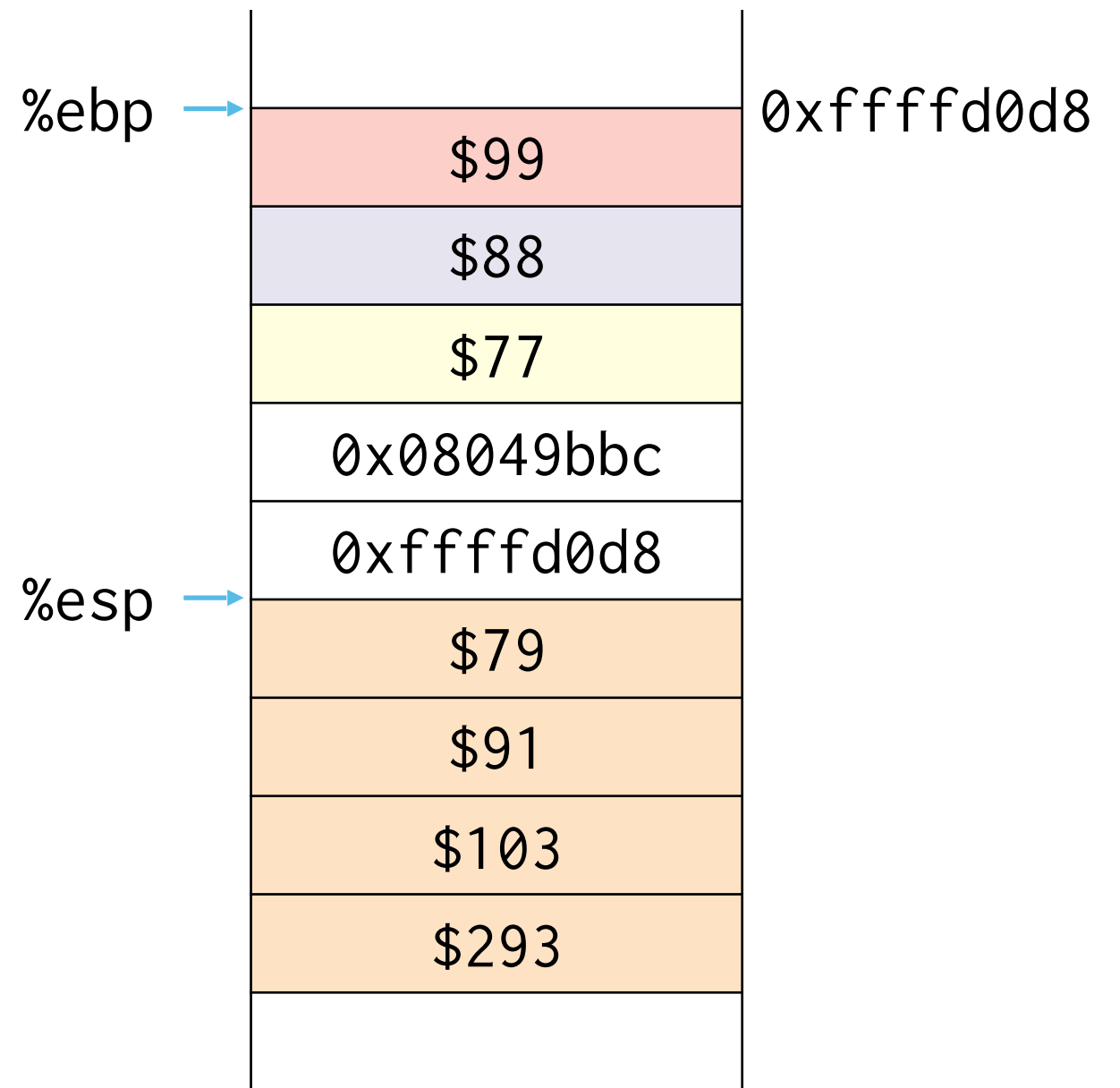
%esp, %ebp →



```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     → leave
27     ret
28  main:
29     pushl    %ebp
30     movl     %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call     foobar(int, int, int)
35     addl     $12, %esp
36     nop
37     leave
38     ret

```

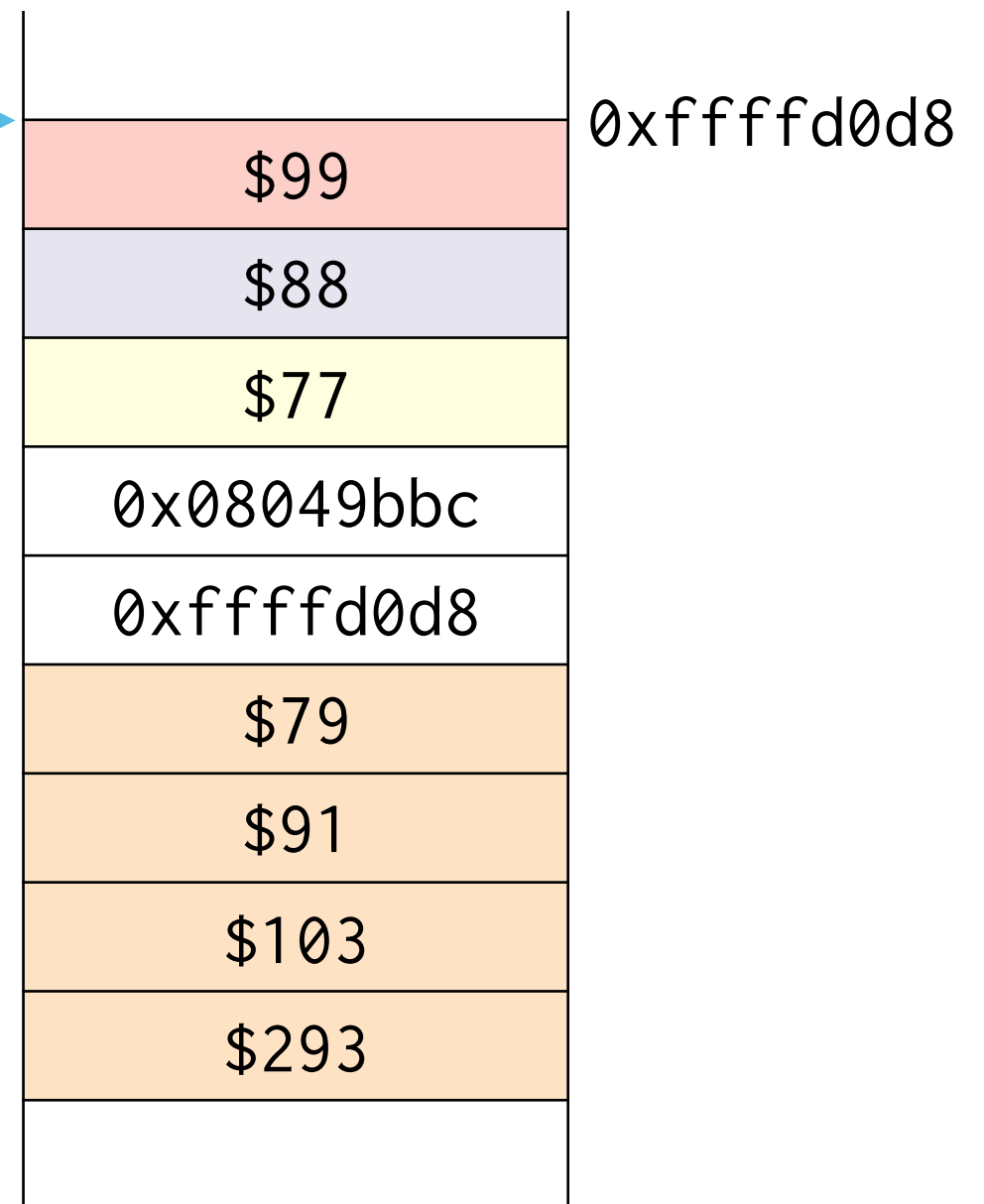


```

1  foobar(int, int, int):
2      pushl   %ebp
3      movl    %esp, %ebp
4      subl    $16, %esp
5      movl    8(%ebp), %eax
6      addl    $2, %eax
7      movl    %eax, -4(%ebp)
8      movl    12(%ebp), %eax
9      addl    $3, %eax
10     movl    %eax, -8(%ebp)
11     movl    16(%ebp), %eax
12     addl    $4, %eax
13     movl    %eax, -12(%ebp)
14     movl    -4(%ebp), %edx
15     movl    -8(%ebp), %eax
16     addl    %eax, %edx
17     movl    -12(%ebp), %eax
18     addl    %edx, %eax
19     movl    %eax, -16(%ebp)
20     movl    -4(%ebp), %eax
21     imull    -8(%ebp), %eax
22     imull    -12(%ebp), %eax
23     movl    %eax, %edx
24     movl    -16(%ebp), %eax
25     addl    %edx, %eax
26     leave
27     → ret
28 main:
29     pushl    %ebp
30     movl    %esp, %ebp
31     pushl    $99
32     pushl    $88
33     pushl    $77
34     call    foobar(int, int, int)
35     addl    $12, %esp
36     nop
37     leave
38     ret

```

%esp, %ebp →



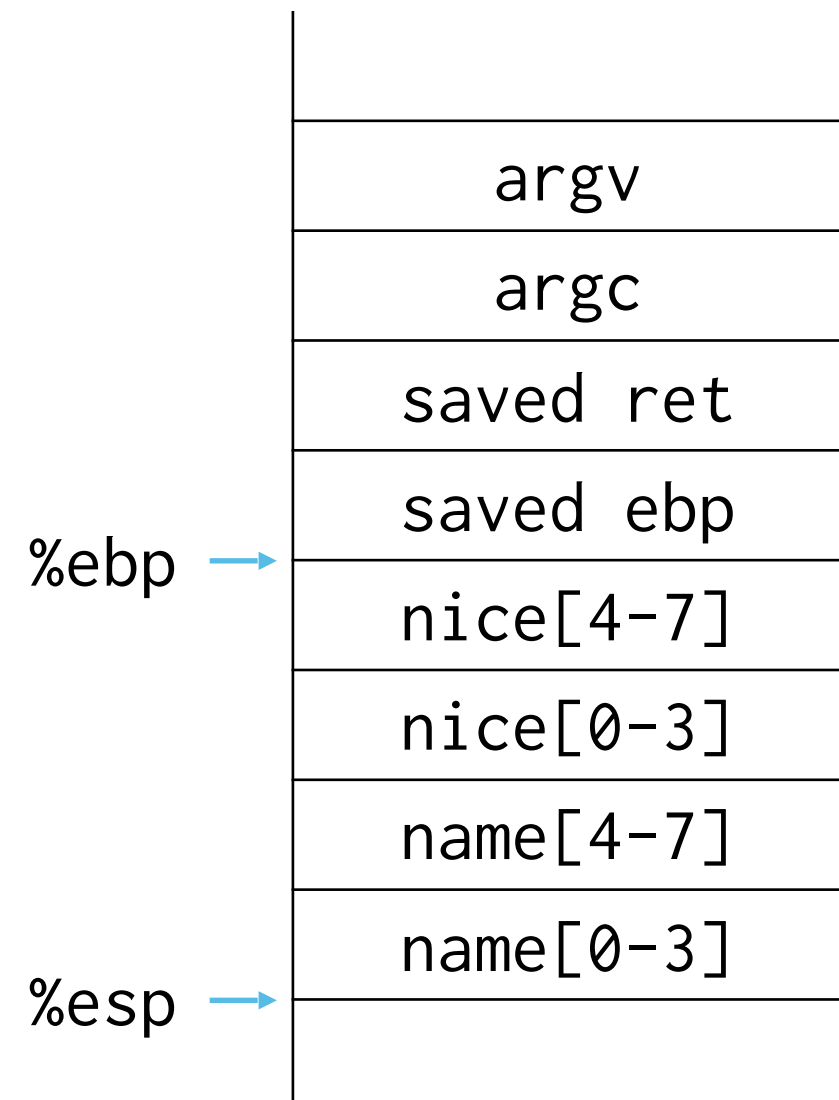
%eip = 0x08049bbc

Let's look at this in GDB (w/ GEF)

Example 1

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

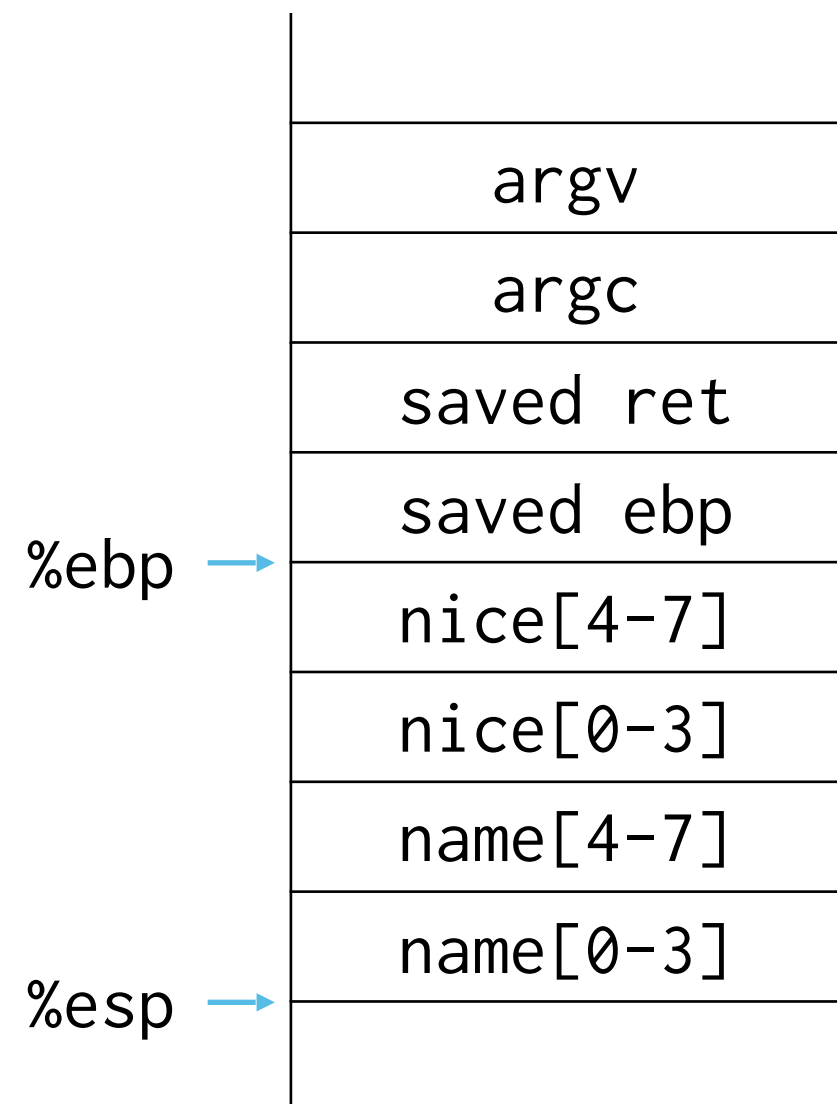
int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```



Example 1

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

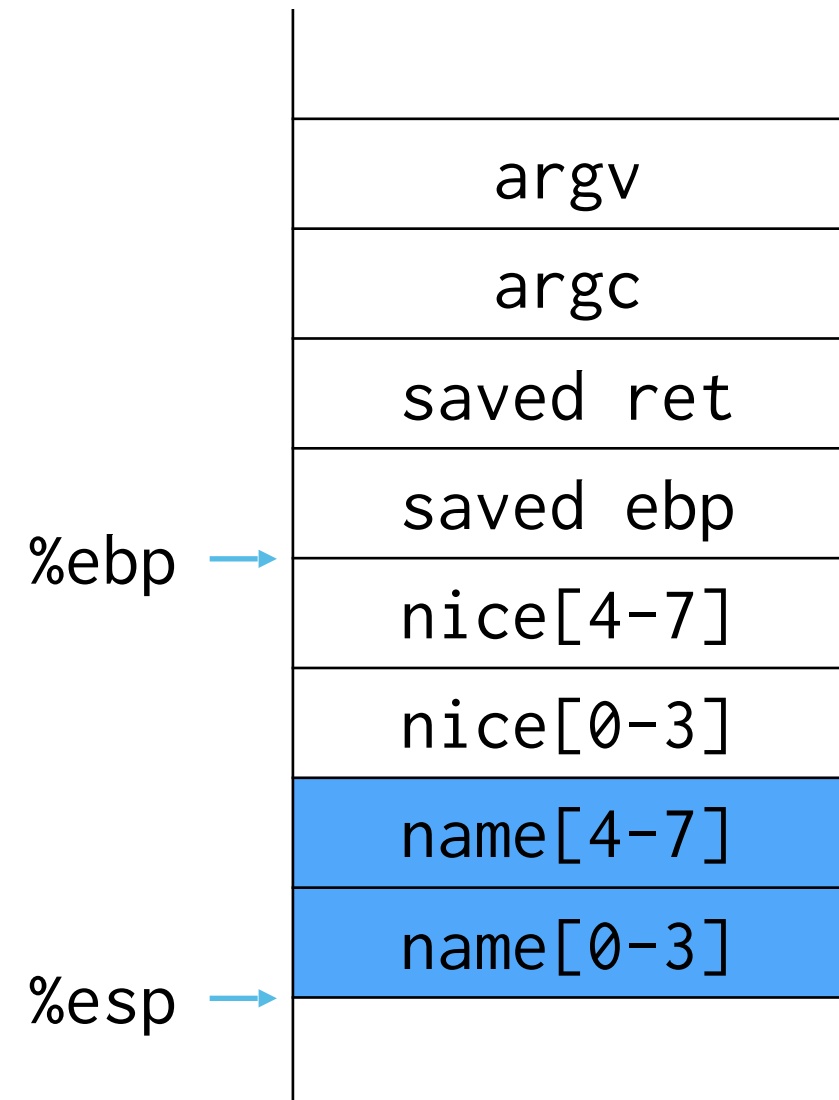
int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    → gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```



Example 1

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

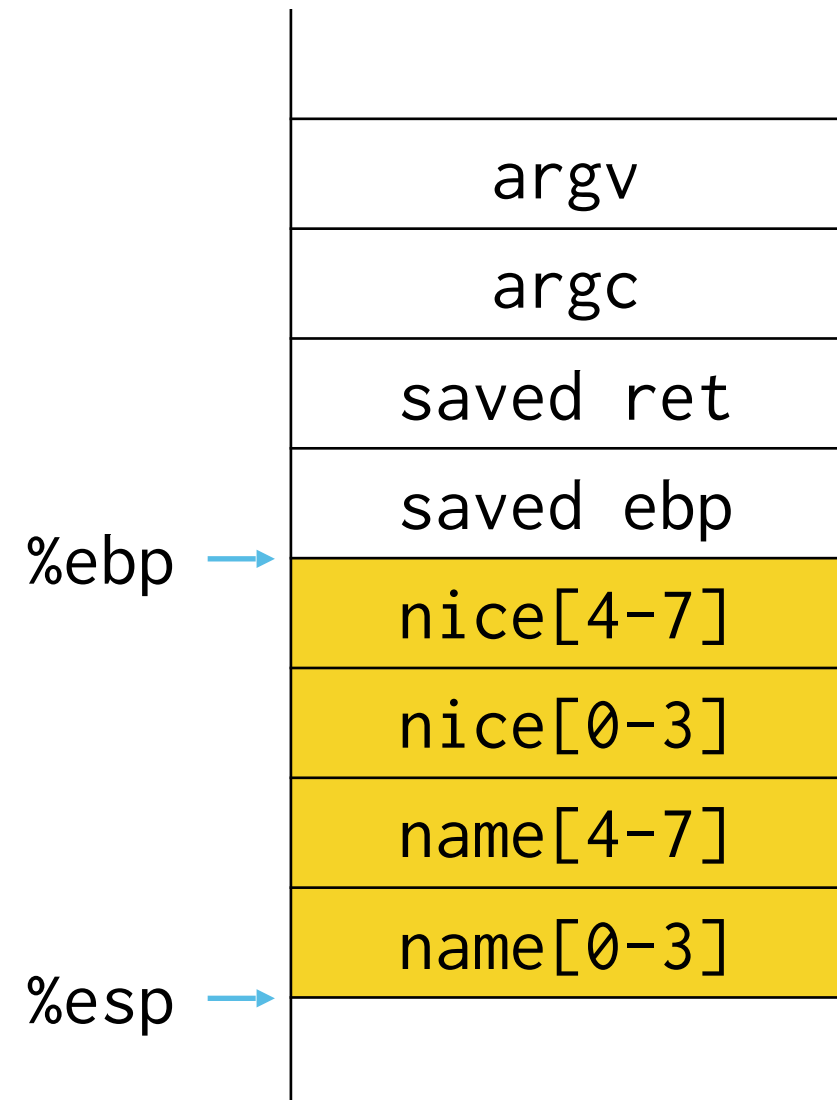
int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    → gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```



Example 1

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

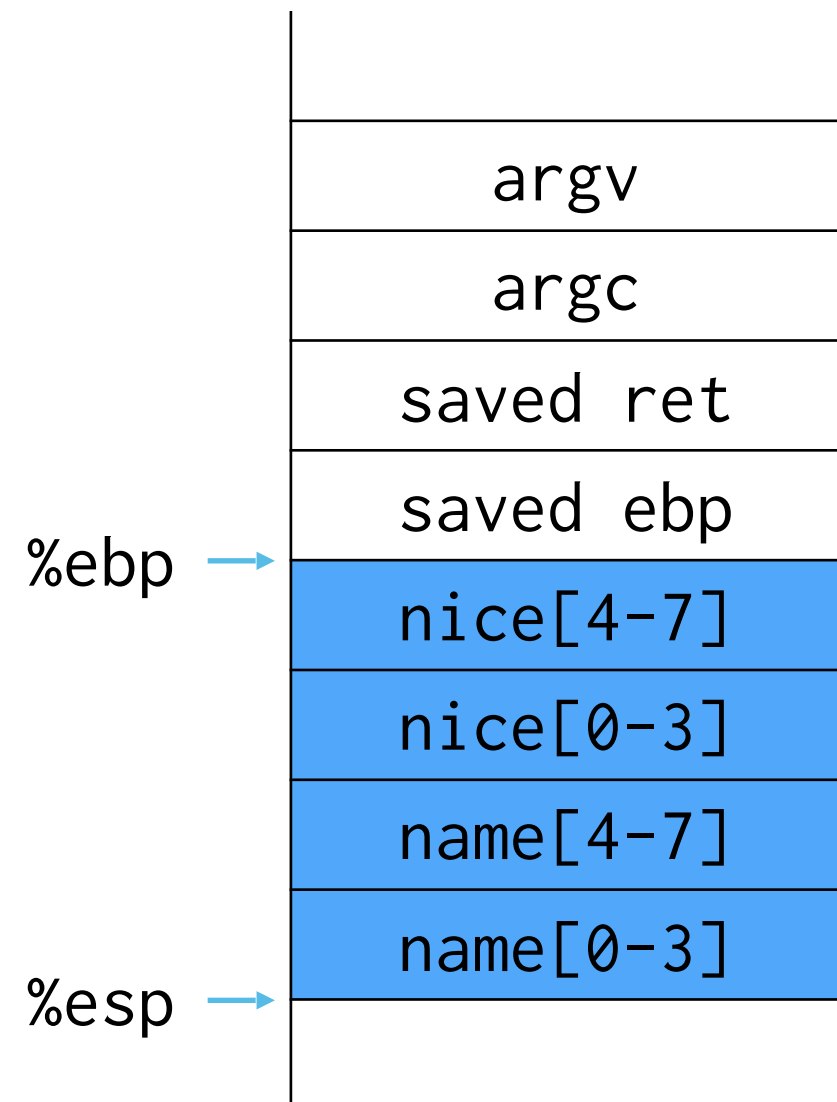
int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    gets(name);
    → printf("%s %s\n", name, nice);
    return 0;
}
```



Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    → gets(name);
    printf("%s %s\n", name, nice);
    return 0;
}
```

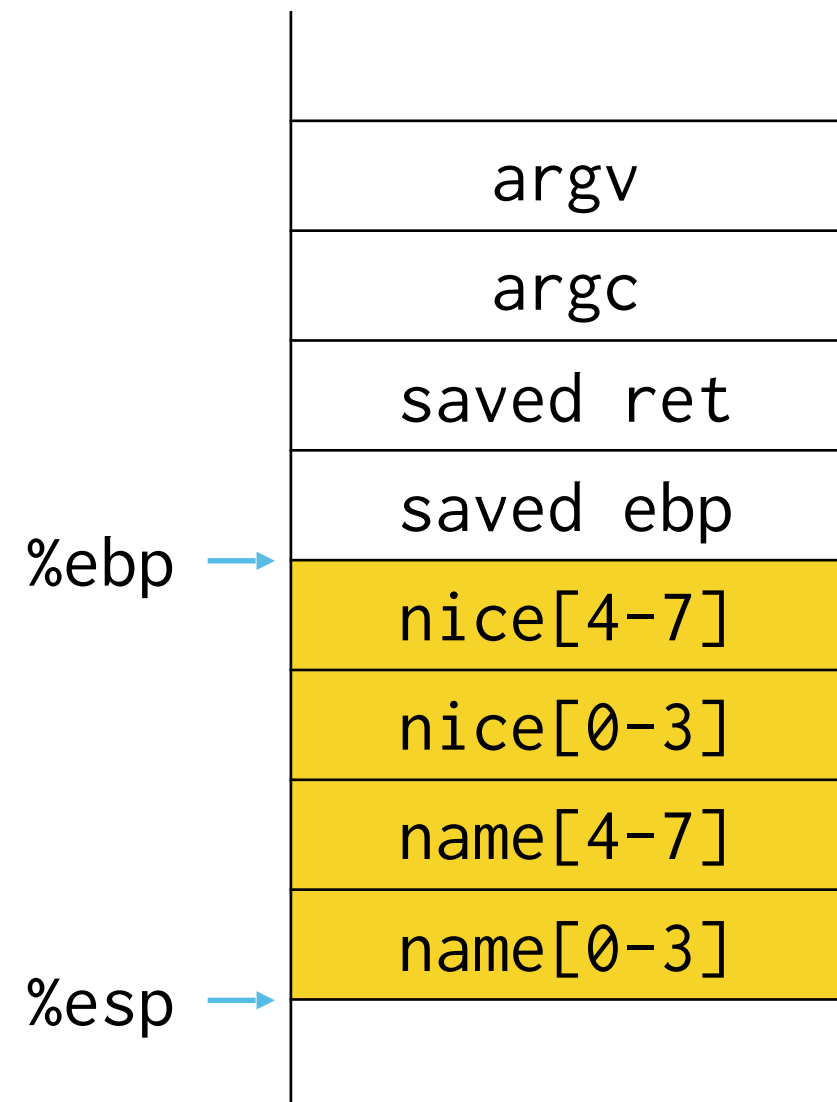


What happens if we read a long name?

Example 1

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

int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    gets(name);
    → printf("%s %s\n", name, nice);
    return 0;
}
```



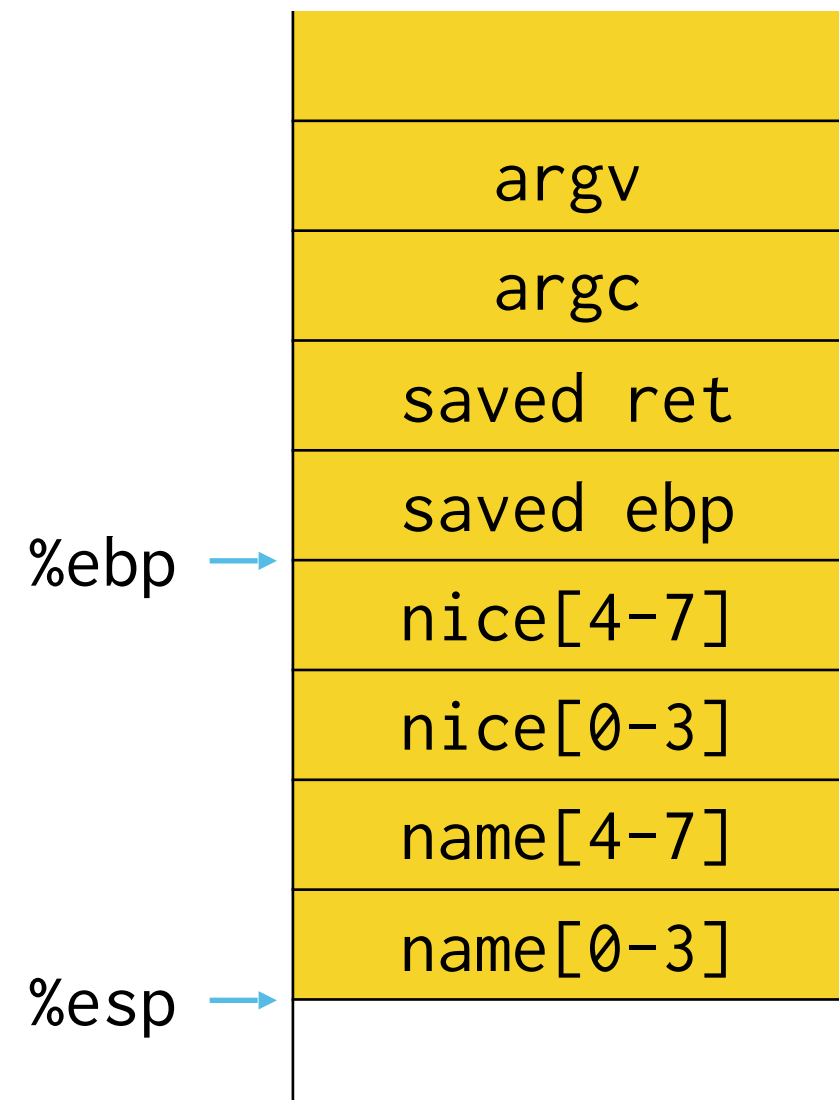
What happens if we read a long name?

Example 1



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

int main(int argc, char**argv) {
    char nice[] = "is nice.";
    char name[8];
    gets(name);
    → printf("%s %s\n", name, nice);
    return 0;
}
```



If not null terminated can read more of the stack

Let's run this program!

Example 2

```
#include <stdio.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;
}
```

Example 2

```
#include <stdio.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;
}
```

Example 2

```
#include <stdio.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;
}
```

argv[1]
0xbbbbbbbb
0xaaaaaaaa
saved ret
saved ebp
0xdeadbeef
buf[0-3]

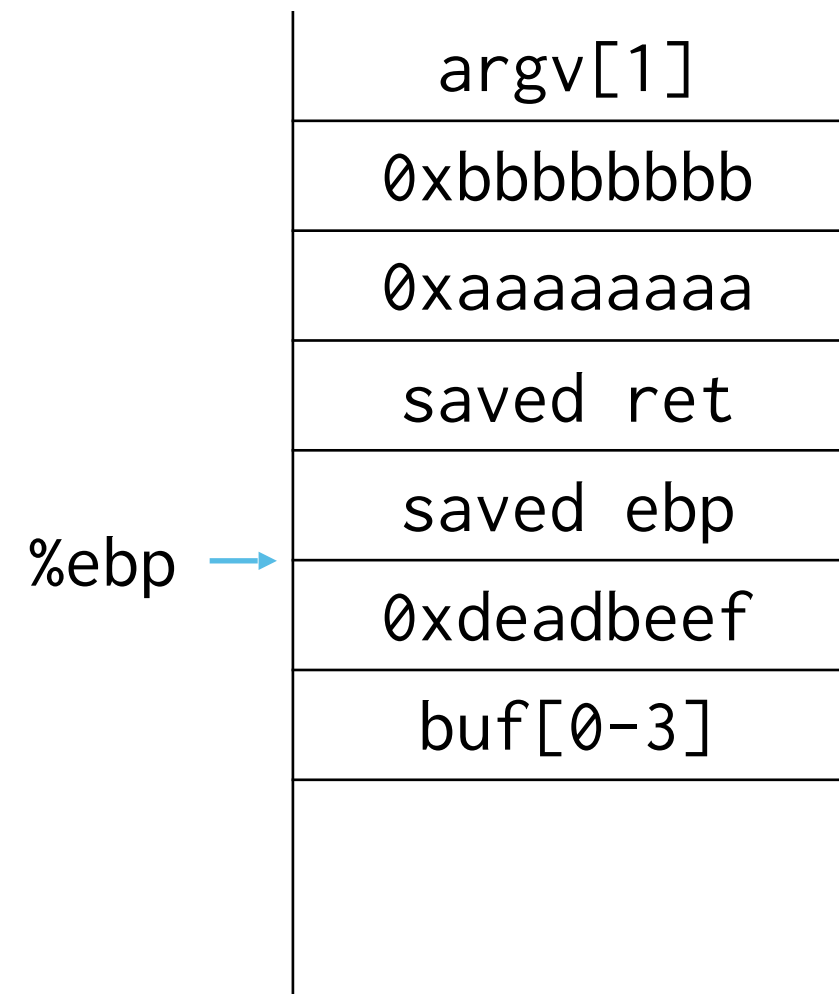
Example 2

```
#include <stdio.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;
}
```



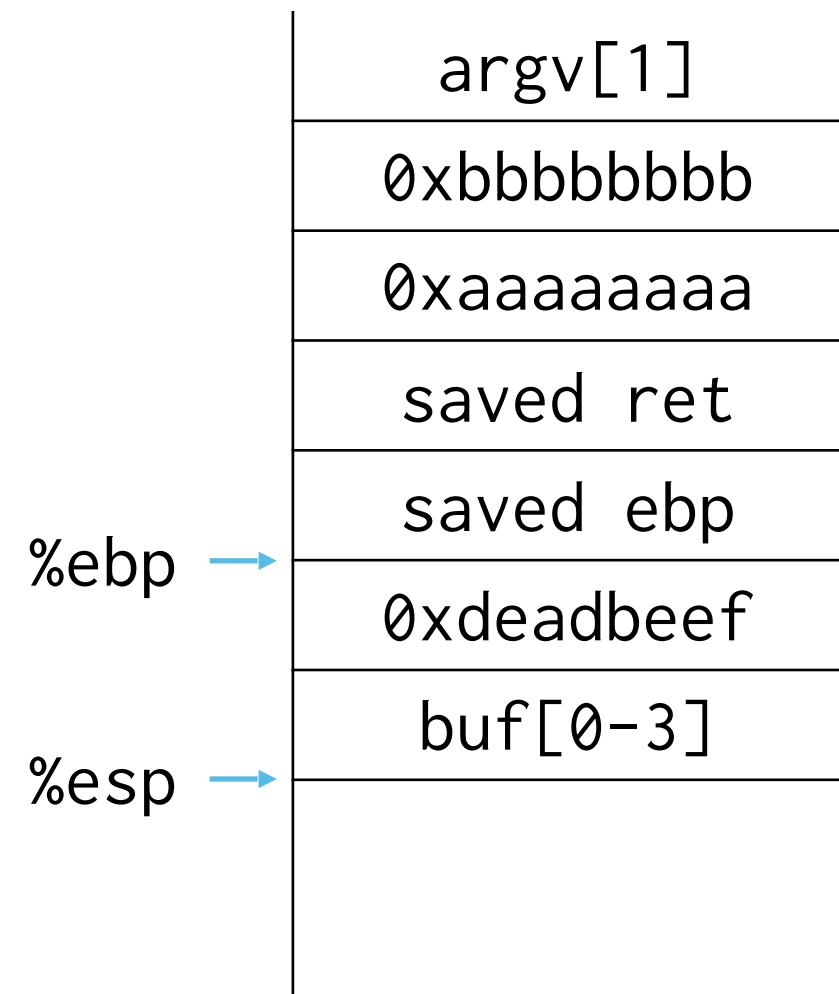
Example 2

```
#include <stdio.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;
}
```



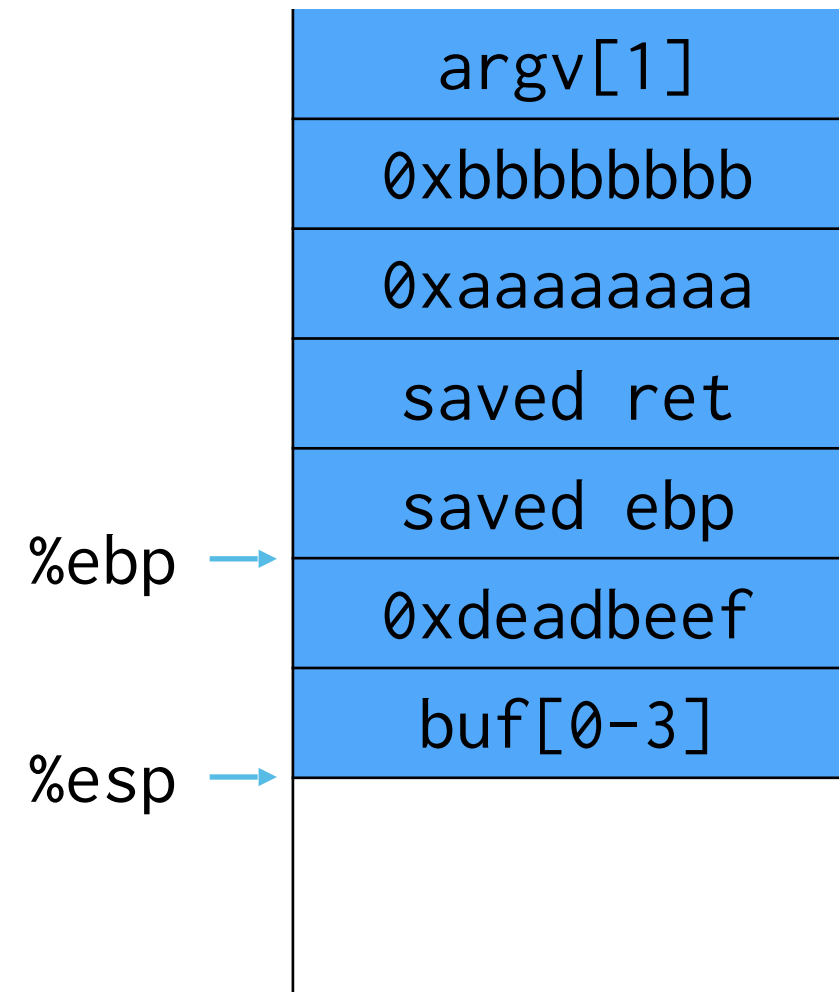
Example 2

```
#include <stdio.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;
}
```



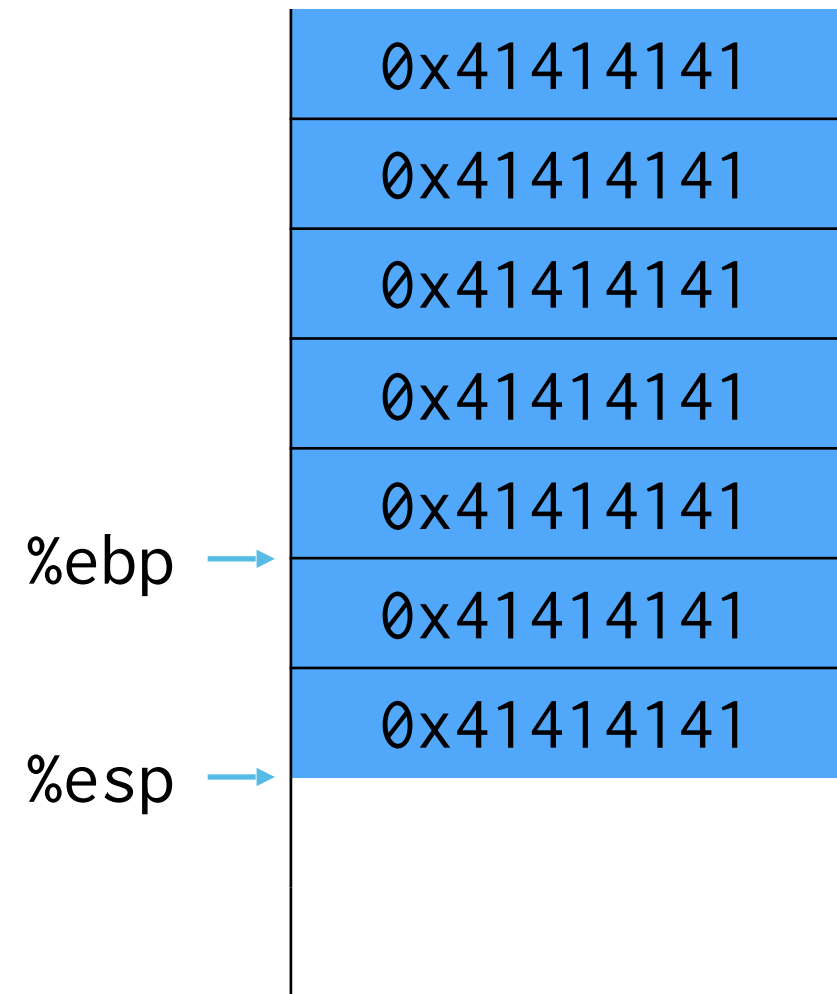
Example 2

```
#include <stdio.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;
}
```



If first argument to program is "AAAAAAAAA..."

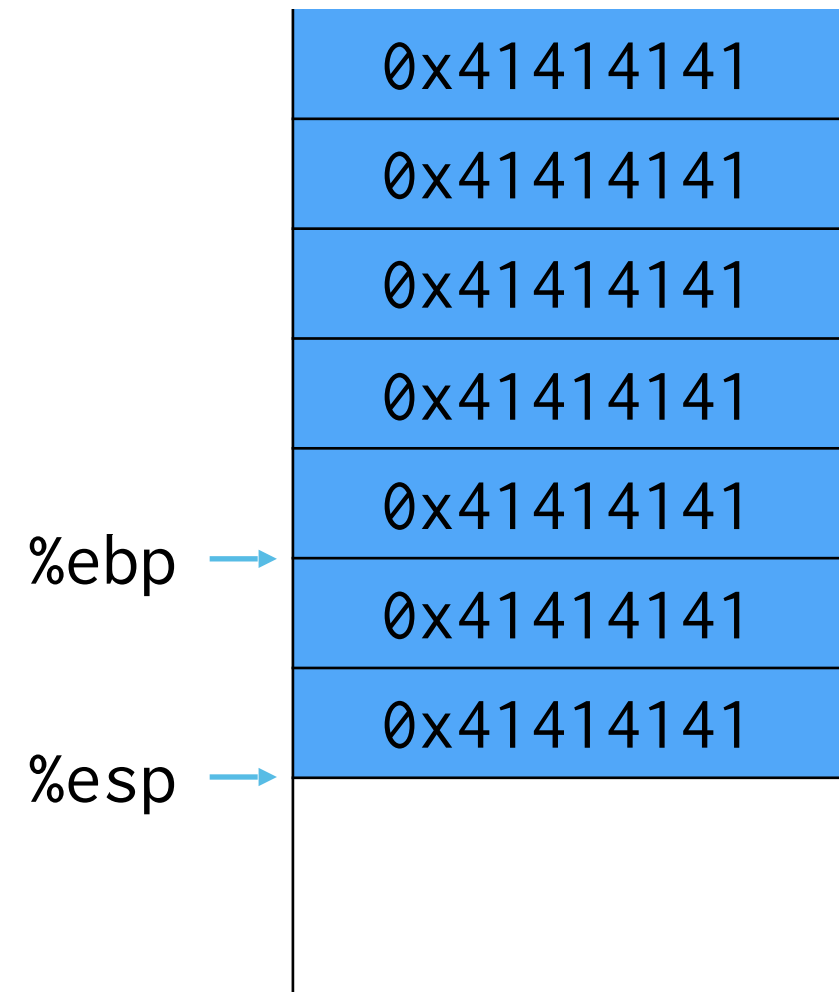
Example 2

```
#include <stdio.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;
}
```



Example 2

```
#include <stdio.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;
}
```

%esp, %ebp →

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

0x41414141

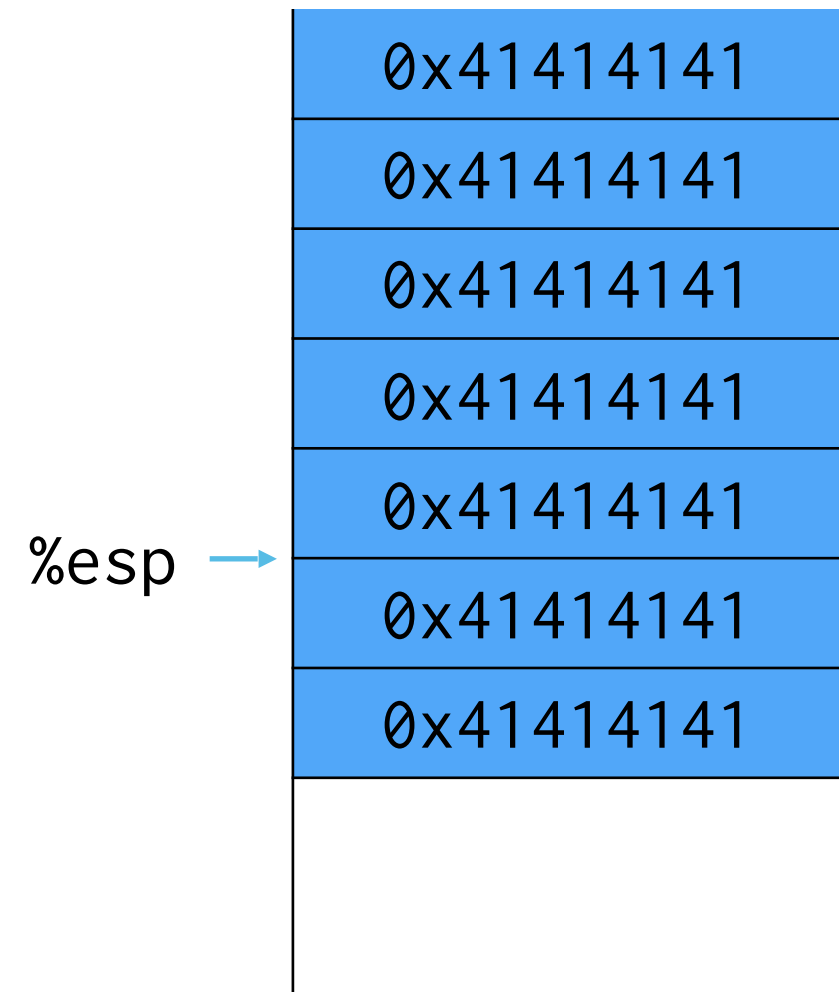
Example 2

```
#include <stdio.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;
}
```



%ebp = 0x41414141

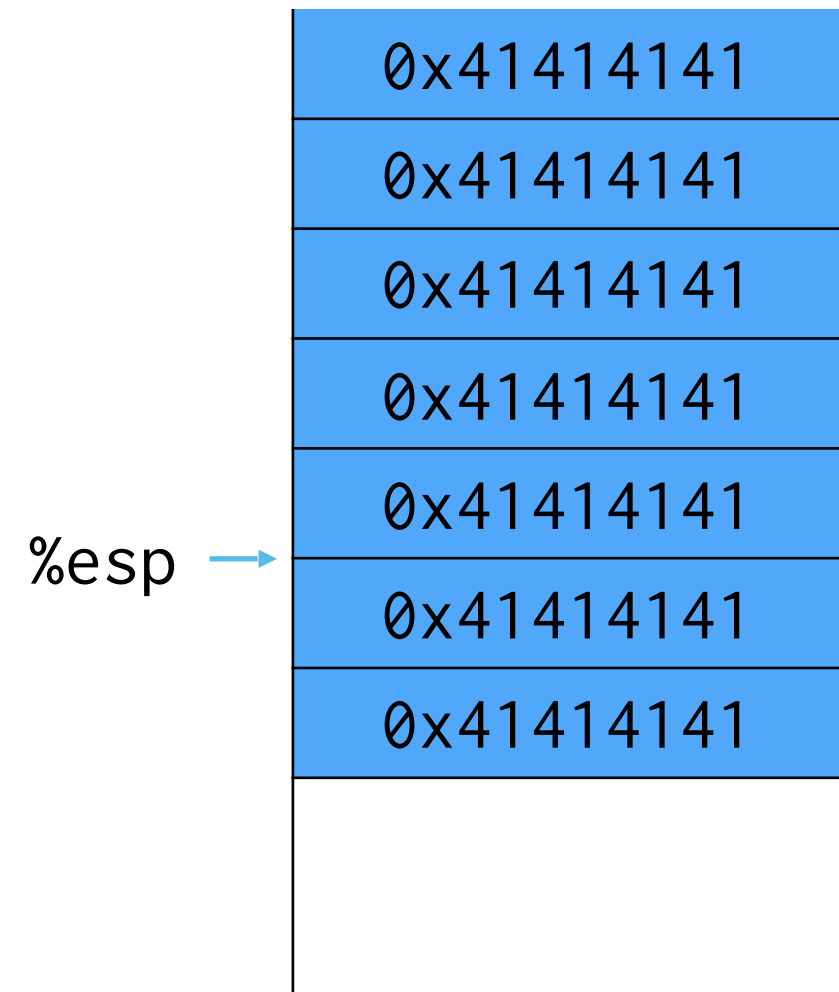
Example 2

```
#include <stdio.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;
}
```



%ebp = 0x41414141

%eip = 0x41414141

Example 2



`%eip = 0x41414141`

Stack Buffer Overflow

- If source string of strcpy controlled by attacker (and destination is on the stack)
 - Attacker gets to control where the function returns by overwriting the return address
 - Attacker gets to transfer control to anywhere!
- Where do you jump?

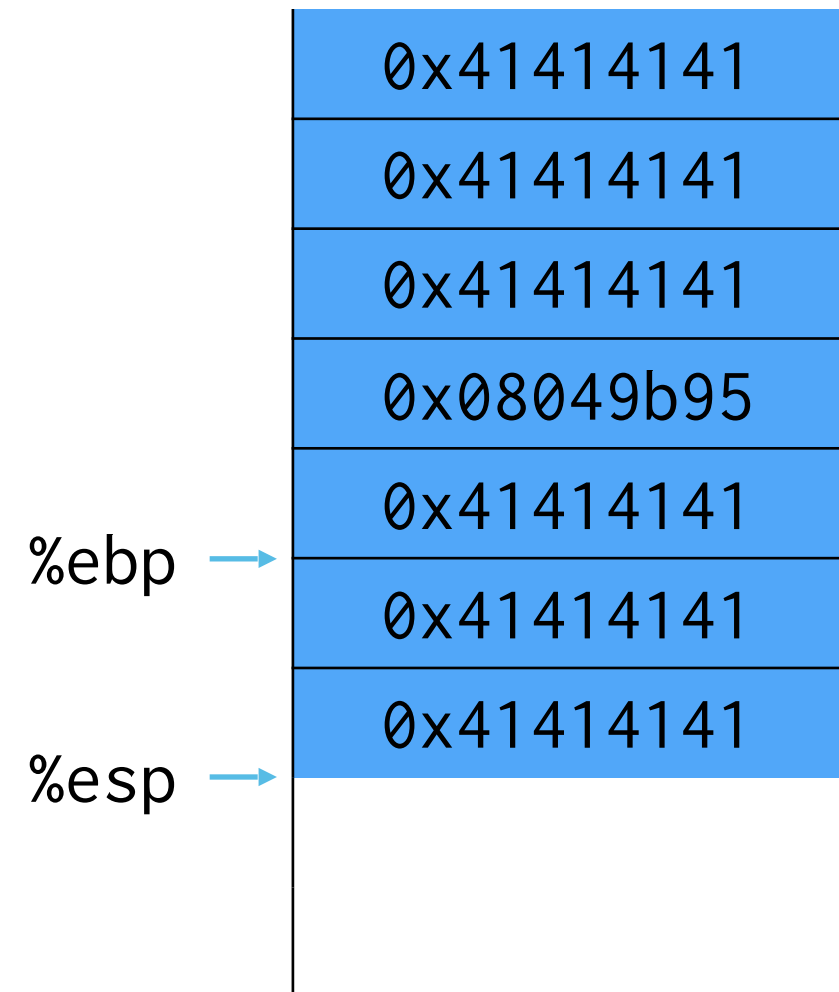
Existing functions

```
#include <stdio.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;
}
```



Existing functions

```
#include <stdio.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;
}
```

%esp, %ebp →

0x41414141

0x41414141

0x41414141

0x08049b95

0x41414141

0x41414141

0x41414141

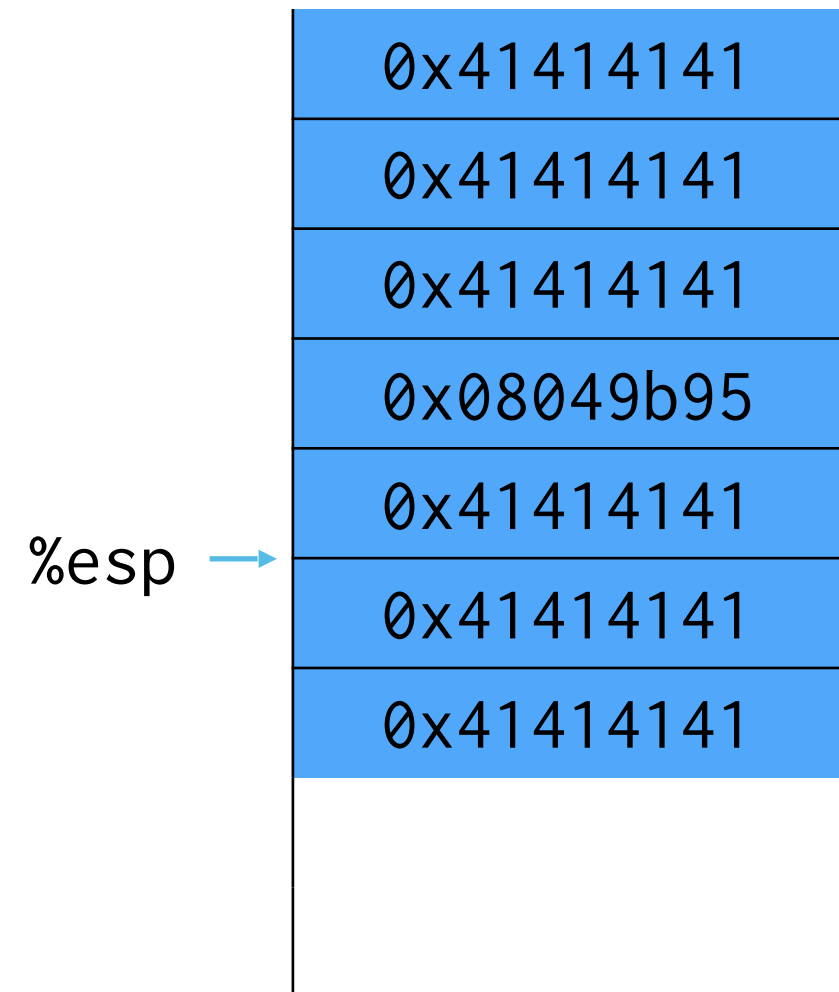
Existing functions

```
#include <stdio.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;
}
```



%ebp = 0x41414141

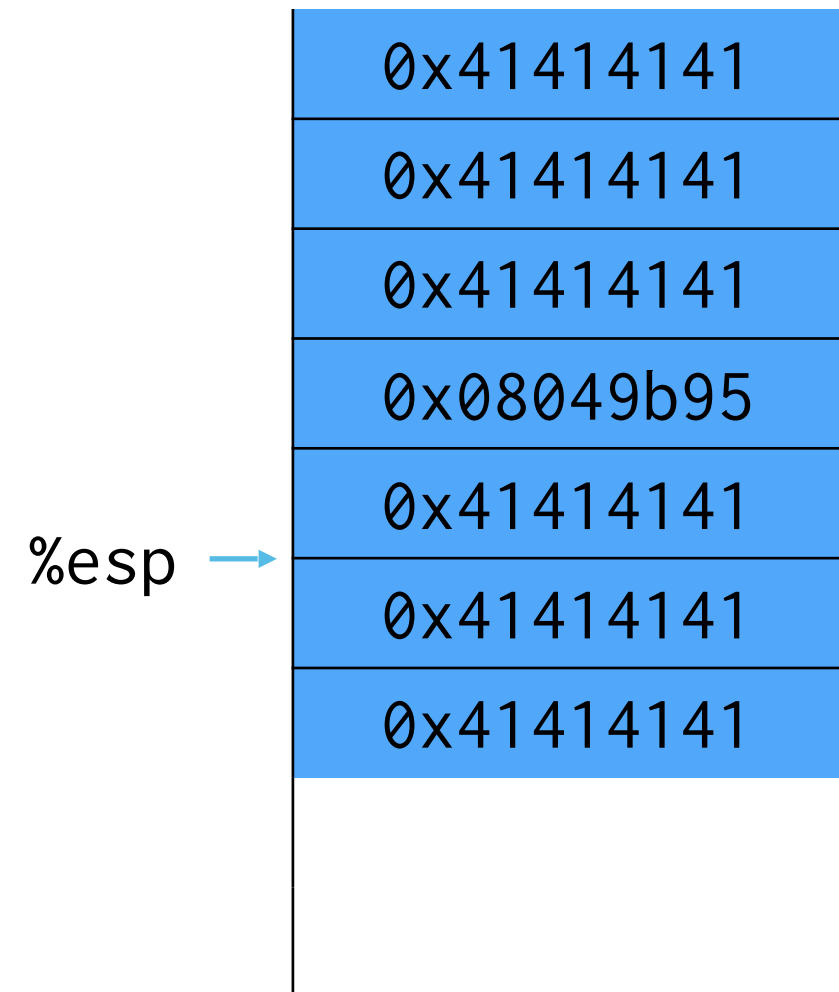
Existing functions

```
#include <stdio.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);
}

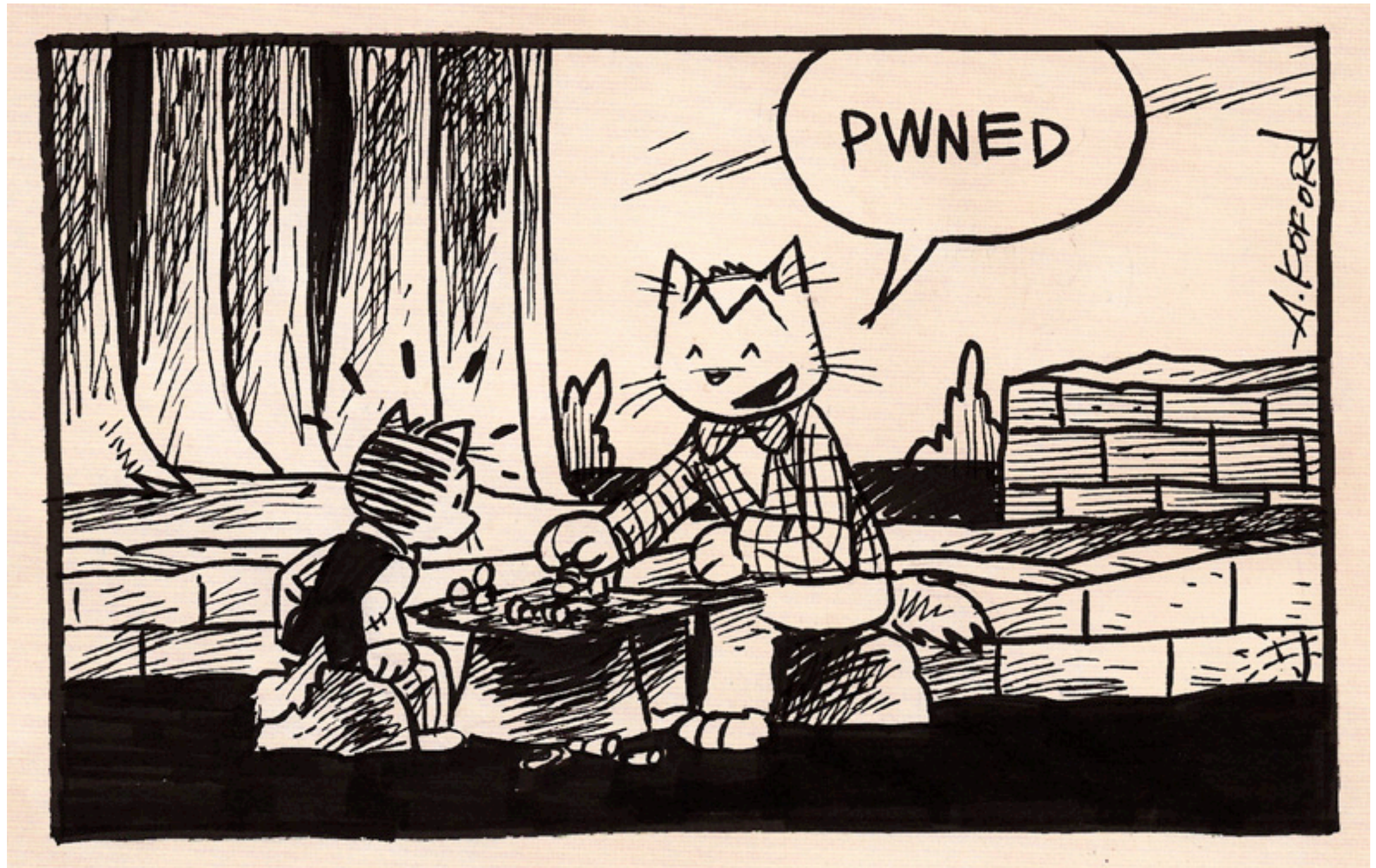
int main(int argc, char**argv) {
    func(0xaaaaaaaa, 0xbbbbbbbb, argv[1]);
    return 0;
}
```



%ebp = 0x41414141

%eip = 0x08049b95

Existing functions

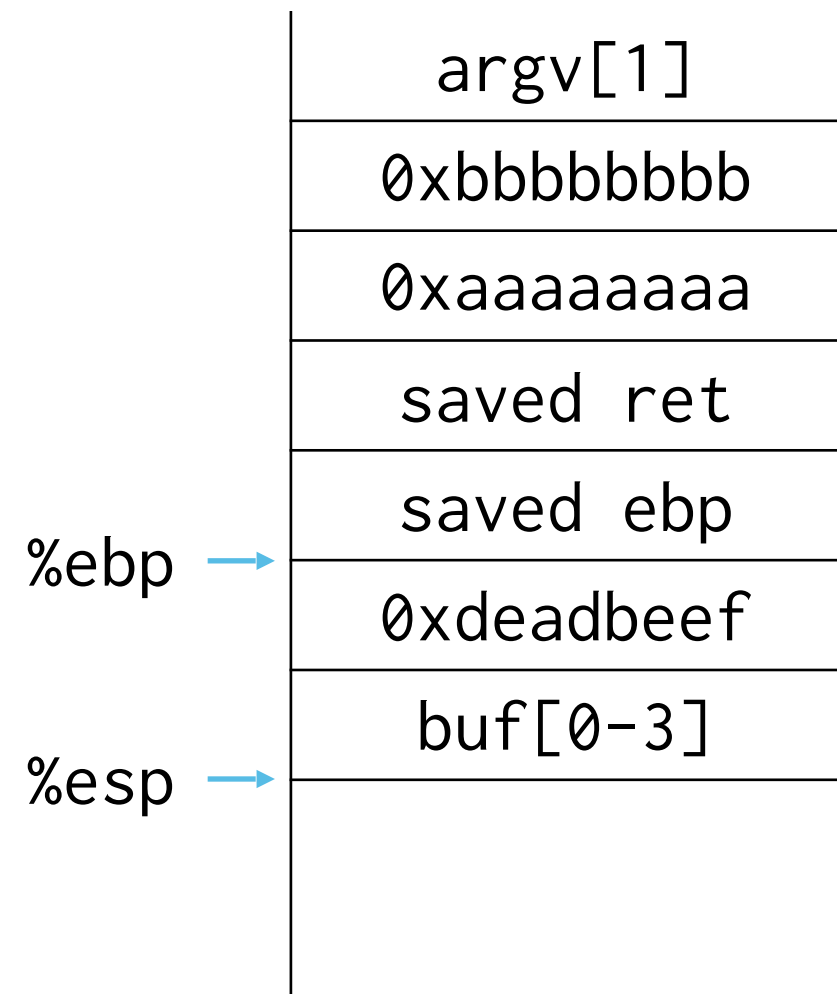


`%eip = 0x08049b95`

Let's look at this in GDB (w/ GEF)

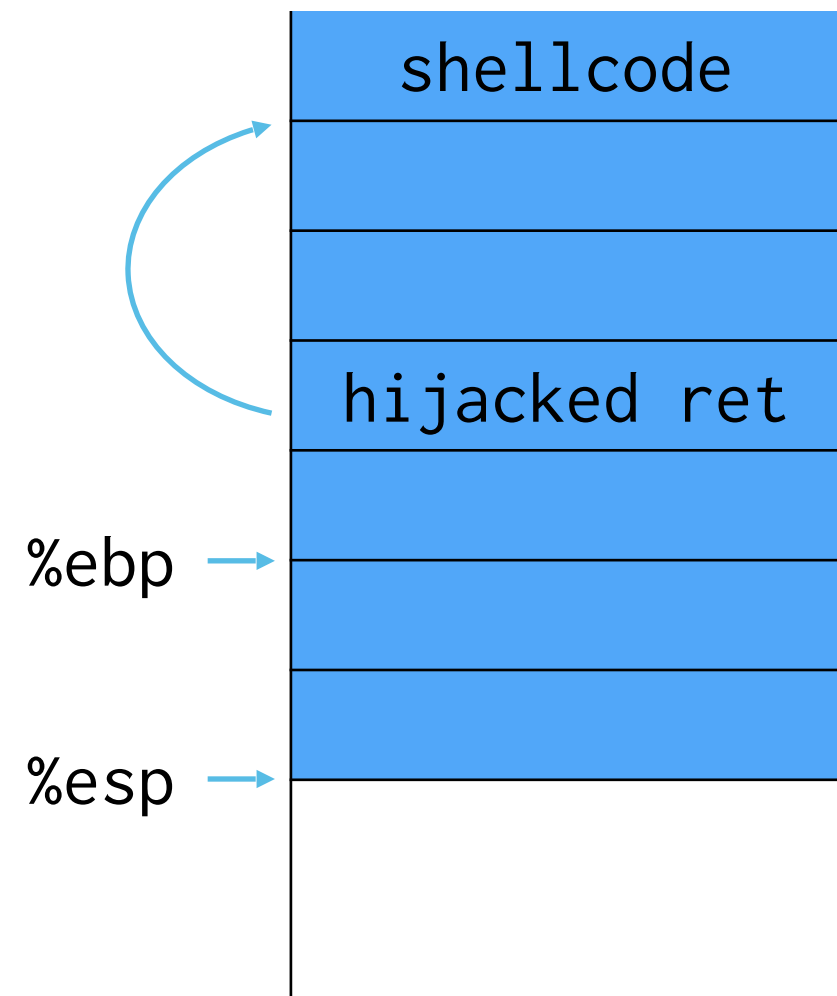
Better Hijacking Control

- Jump to attacker-supplied code
- Where? We have control of string!
 - Put code in string
 - Jump to start of string



Better Hijacking Control

- Jump to attacker-supplied code
- Where? We have control of string!
 - Put code in string
 - Jump to start of string



Shellcode

- **Shellcode:** small code fragment that receives initial control in an control flow hijack exploit
 - Control flow hijack: taking control of instruction ptr
- Earliest attacks used shellcode to exec a shell
 - Target a setuid root program, gives you root shell

Shellcode

```
void main() {  
    char *name[2];  
  
    name[0] = "/bin/sh";  
    name[1] = NULL;  
    execve(name[0], name, NULL);  
}
```

Shellcode

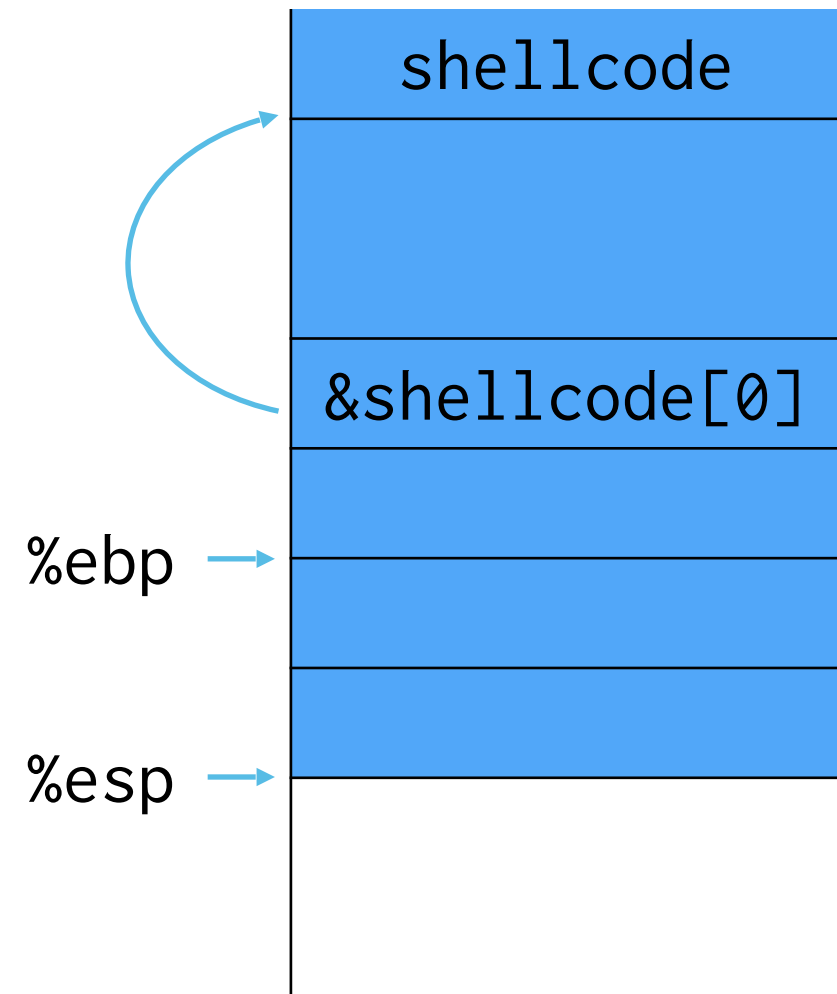
- Can we just take output from gcc/clang?

Shellcode

- There are some restrictions
 - 1. Shellcode cannot contain null characters '\0'
 - Why not?
 - Fix: use different instructions and NOPs to eliminate \0
 - 2. If payload is via gets() must also avoid line-breaks

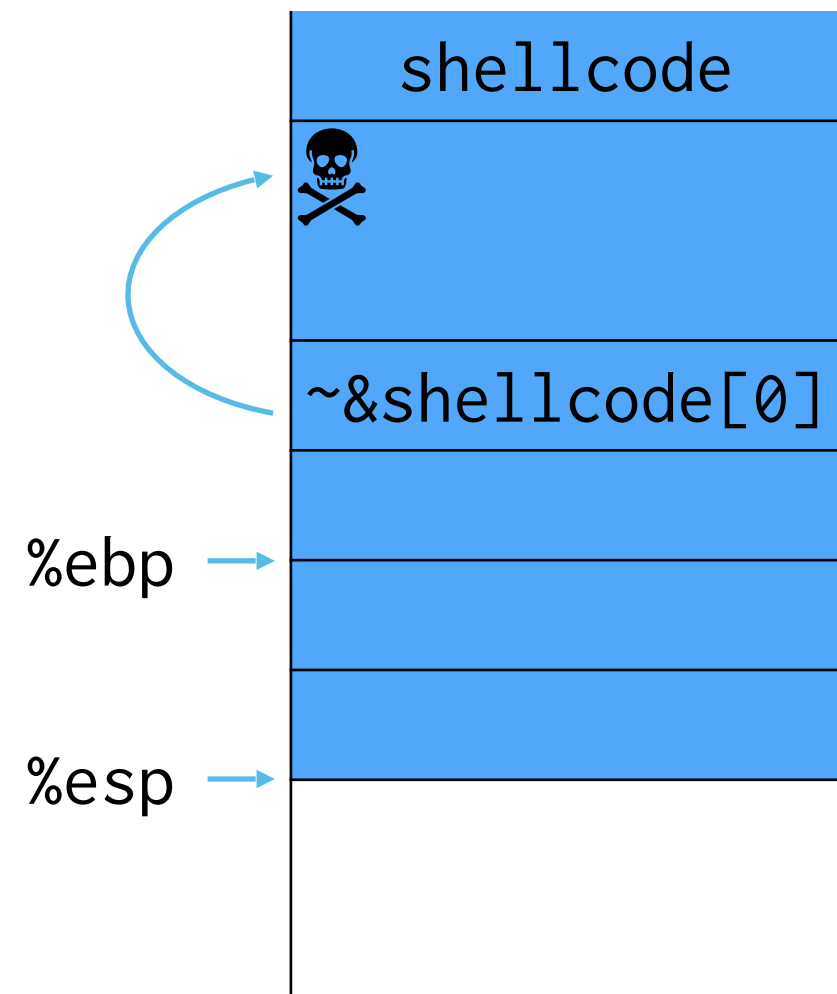
How do we make this robust?

- 3. Exact address of shellcode start not always easy to guess
 - Miss? SEGFAULT!
- Fix: NOP-sled



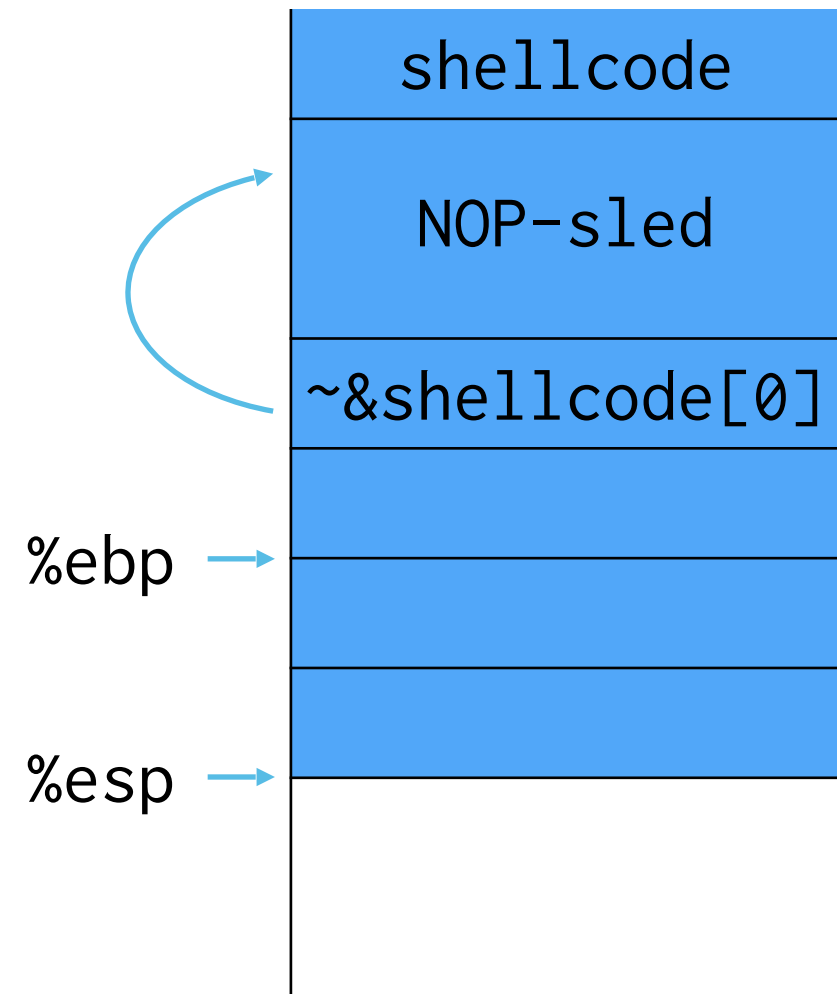
How do we make this robust?

- 3. Exact address of shellcode start not always easy to guess
 - Miss? SEGFAULT!
- Fix: NOP-sled



How do we make this robust?

- 3. Exact address of shellcode start not always easy to guess
 - Miss? SEGFAULT!
- Fix: NOP-sled



Metasploit helps!

```
msf payload(shell_bind_tcp) > generate -h
```

```
Usage: generate [options]
```

```
Generates a payload.
```

OPTIONS:

- E Force encoding.
- b The list of characters to avoid: '\x00\xff'
- e The name of the encoder module to use.
- f The output file name (otherwise stdout)
- h Help banner.
- i the number of encoding iterations.
- k Keep the template executable functional
- o A comma separated list of options in VAR=VAL format.
- p The Platform for output.
- s NOP sled length.
- t The output format: raw,ruby,rb,perl,pl,c,js_be,js_le,java,dll,exe,exe-small,elf,macho,vba,vml
- x The executable template to use

Buffer Overflow Defenses

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

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

Buffer Overflow Defenses

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

Stack Canary

- Special value placed before return address
 - Secret random value chosen at program start
 - String terminator '\0'
- Gets overwritten during buffer overflow
- Check canary before jumping to return address
- Automatically inserted by compiler
 - GCC: -fstack-protector or -fstack-protector-strong

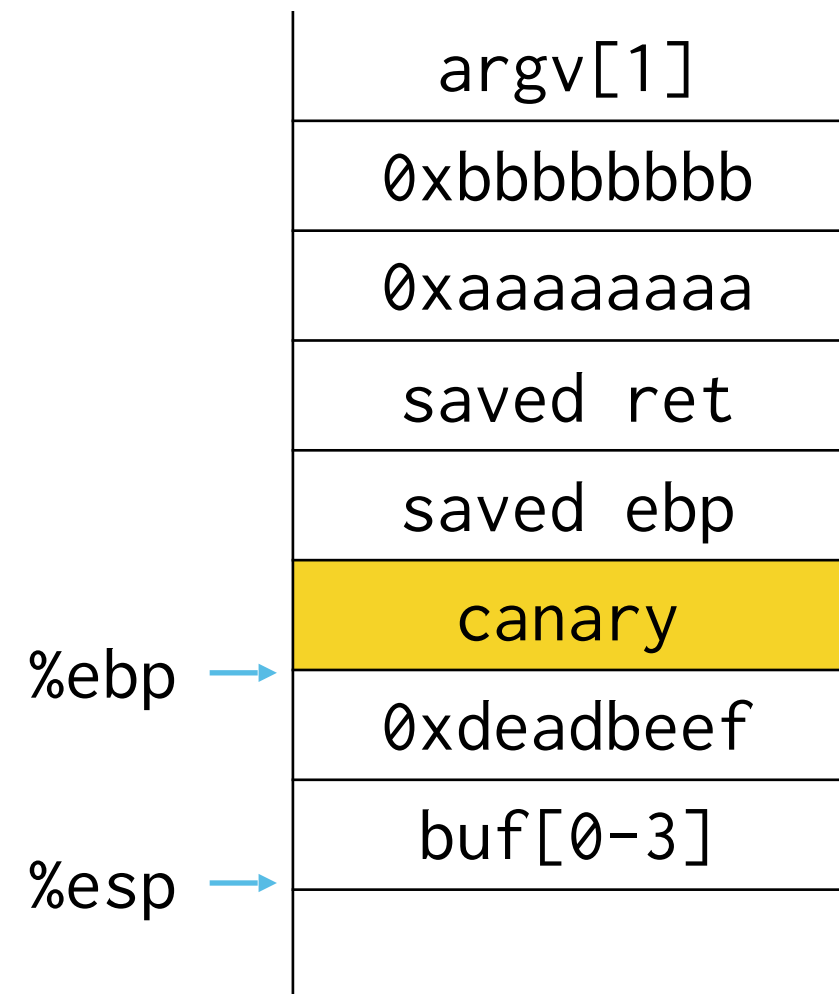
Back to Example 2

```
#include <stdio.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;
}
```



Check canary on ret

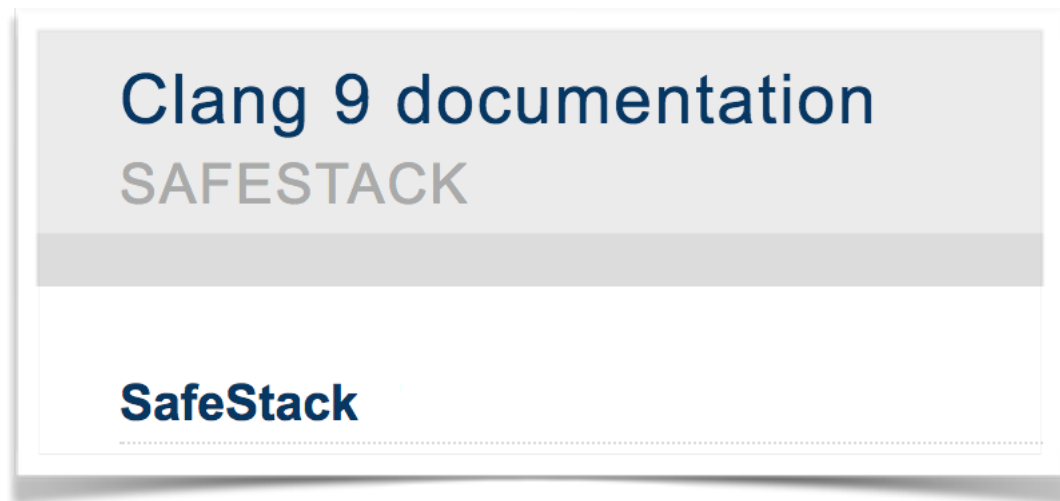
Stack Canary

- **Plus:** No code changes required, only recompile
- **Minus:** Performance penalty per return
- **Minus:** Only protects against stack smashing
- **Minus:** Fails if attacker can read memory

Buffer Overflow Defenses

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

Separate Stack



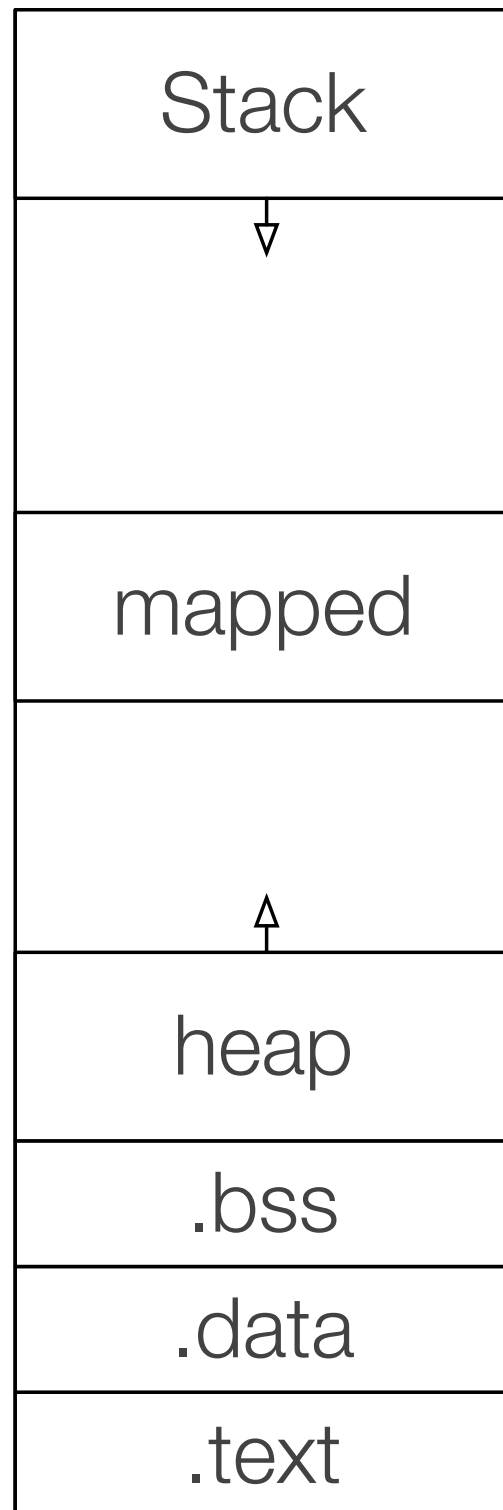
“SafeStack is an instrumentation pass that protects programs against attacks based on stack buffer overflows, without introducing any measurable performance overhead. It works by separating the program stack into two distinct regions: the safe stack and the unsafe stack. The safe stack stores **return 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.”

WebAssembly has separate stack (kind of)!

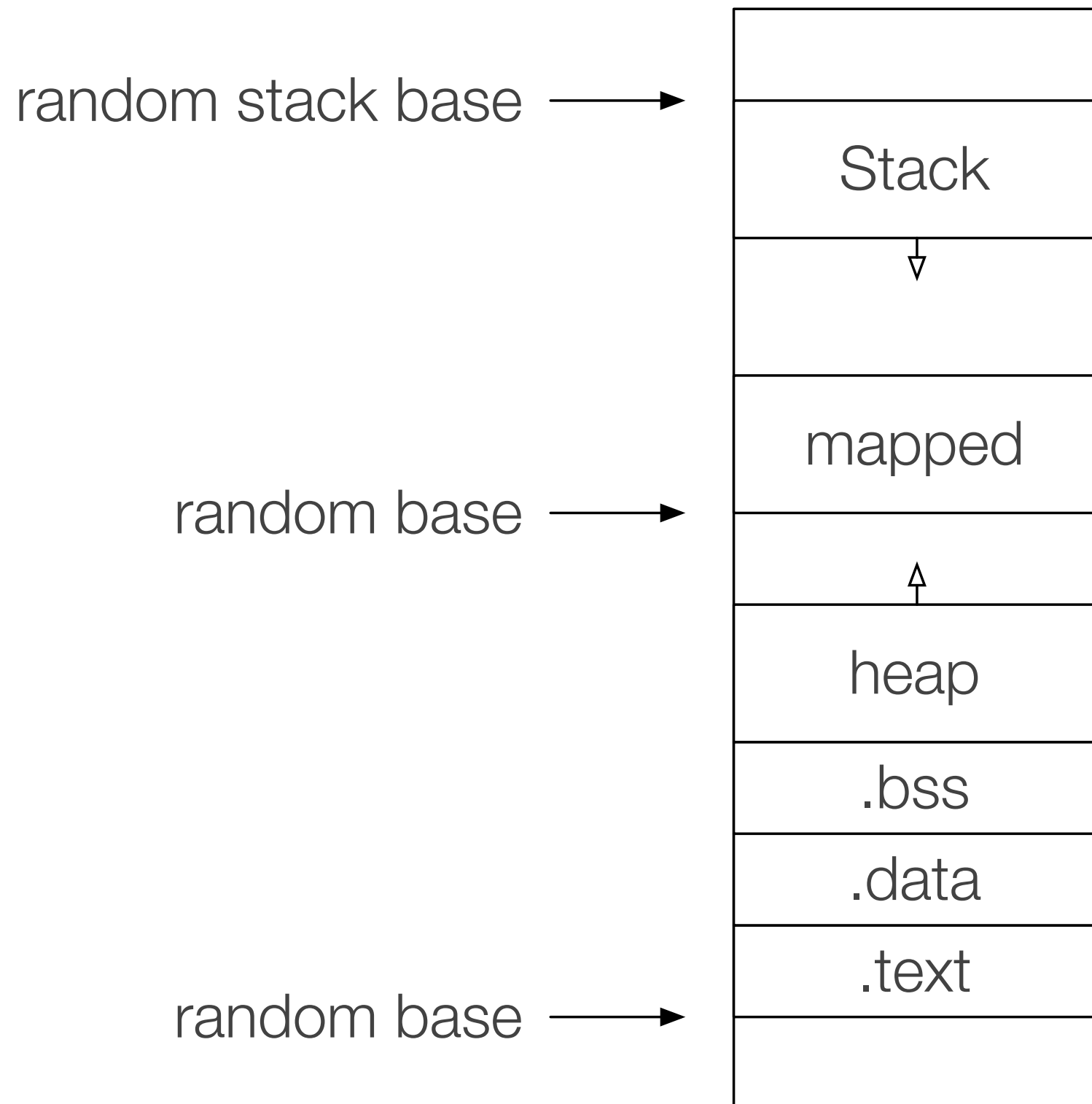
Address Space Layout Randomization

- Change location of stack, heap, code, static vars
- Works because attacker needs address of shellcode
- Layout must be unknown to attacker
 - Randomize on every launch (best)
 - Randomize at compile time
- Implemented on most modern OSes in some form

Traditional Memory Layout

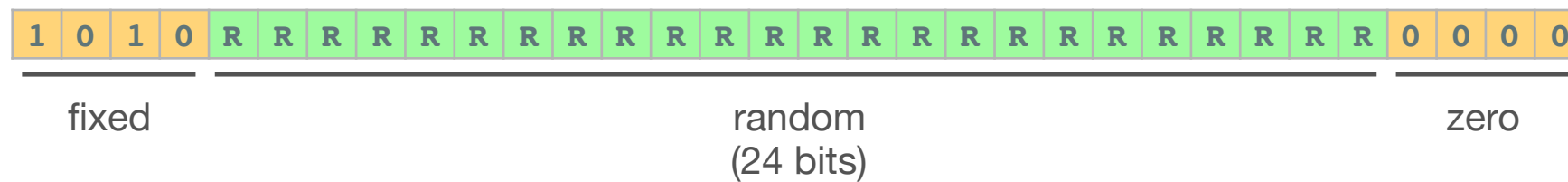


PaX Memory Layout

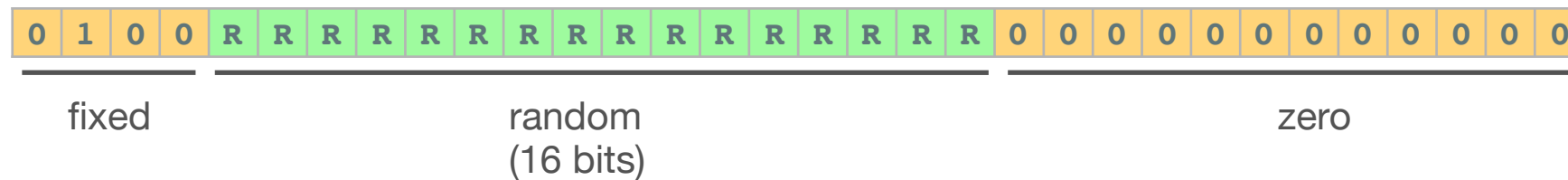


32-bit PaX ASLR (x86)

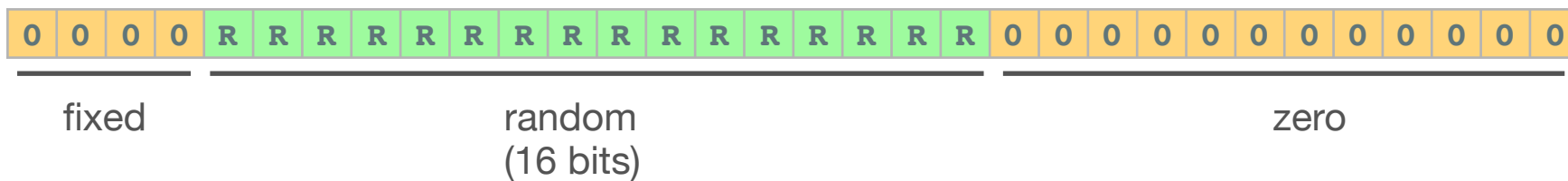
Stack:



Mapped area:



Executable code, static variables, and heap:



Address Space Layout Randomization

- **Plus:** No code changes or recompile required
- **Minus:** 32-bit arch get limited protection
- **Minus:** Fails if attacker can read memory
- **Minus:** Load-time overhead
- **Minus:** No exec img sharing between processes

W^X : write XOR execute

- Use MMU to ensure memory cannot be both writeable and executable at same time
- Code segment: executable, not writeable
- Stack, heap, static vars: writeable, not executable
- Supported by most modern processors
- Implemented by modern operating systems

W^X : write XOR execute

- **Plus:** No code changes or recompile required
- **Minus:** Requires hardware support
- **Minus:** Defeated by return-oriented programming
- **Minus:** Does not protect JITed code

Buffer Overflow Defenses

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

Control Flow Integrity

- Check destination of every indirect jump
 - Function returns
 - Function pointers
 - Virtual methods
- What are the valid destinations?
 - Caller of every function known at compile time
 - Class hierarchy limits possible virtual function instances

CFI

- **Plus:** No code changes or hardware support
- **Plus:** Protects against many vulnerabilities
- **Minus:** Performance overhead
- **Minus:** Requires smarter compiler
- **Minus:** Requires having all code available