CSE 127: Computer Security

Stack buffer overflows

Deian Stefan

Some slides adopted from Kirill Levchenko and Stefan Savage

When is a program secure?

- Formal approach: When it does exactly what it should
  - Not more
  - Not less
- But how do we know what it is supposed to do?

When is a program secure?

- Formal approach: When it does exactly what it should
  - Not more
  - Not less
- But how do we know what it 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?

- Pragmatic approach: When it doesn't do bad things
- Often 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

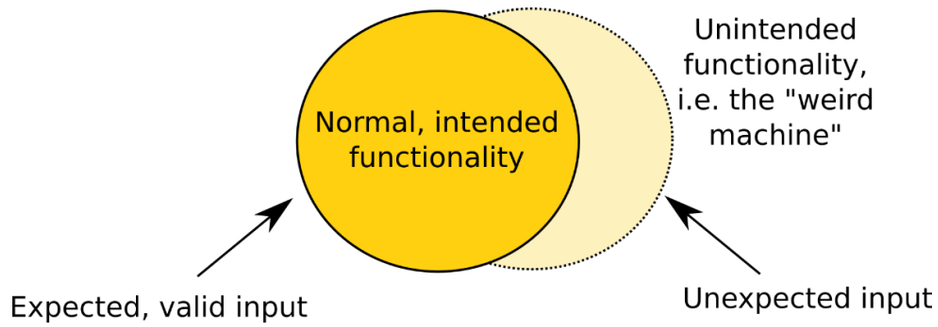
When is a program secure?

But … what if the program doesn't do bad things, but could?

Is it secure? A: yes B: no

Weird machines

- Complex systems contain unintended functionality



- Attackers can trigger this unintended functionality
  - I.e., they are exploiting vulnerabilities

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 vulnerabilities
  - Today: bugs 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 vulnerabilities
  - Today: bugs 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

Exploiting vulnerabilities (the start)

- Dive into low level details of how exploits work
  - How can a remote attacker get victim program to execute their code?

- Threat model: Victim code is handling input that comes from across a security boundary
  - What are some examples of this?

- Security policy: 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 vulns can be exploited
- Identify buffer overflows and assess their impact
- Avoid introducing buffer overflow vulnerabilities
- 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?

- Core concept → 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?

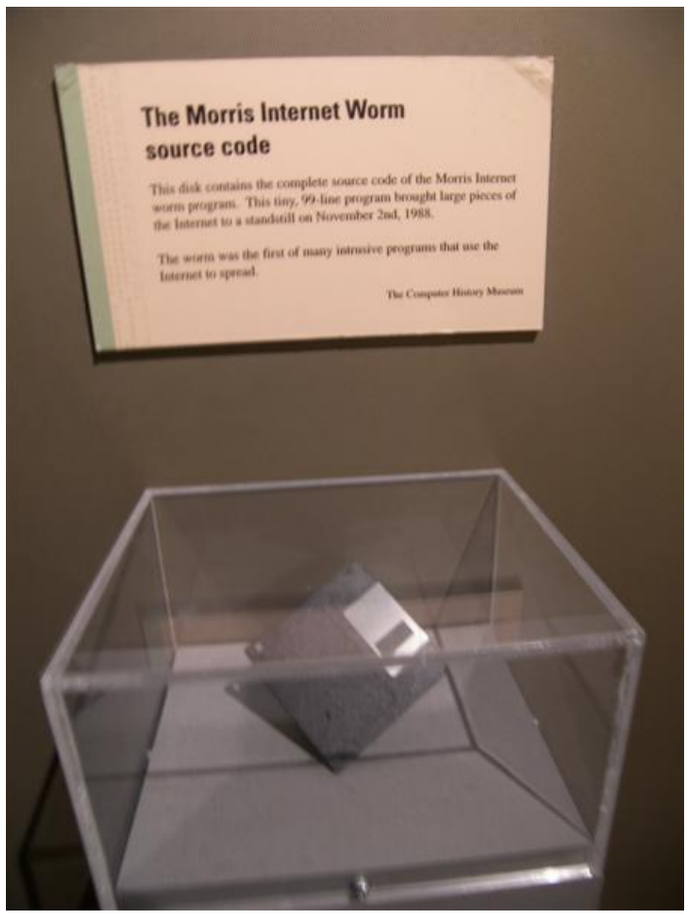
- No automatic bounds checking in C/C++
- The problem is made more acute by the fact many C stdlib functions make it easy to go past bounds
  - String manipulation functions like `gets()`, `strcpy()`, and `strcat()` all write to the destination buffer until they encounter a terminating '`\0`' byte in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

Let's look at the finger daemon in BSD 4.3

```
/*
 * Finger server.
 */
#include <sys/types.h>
#include <netinet/in.h>

#include <stdio.h>
#include <ctype.h>

main(argc, argv)
        char *argv[];
{
        register char *sp;
        char line[512];
        struct sockaddr_in sin;
        int i, p[2], pid, status;
        FILE *fp;
        char *av[4];

        i = sizeof (sin);
        if (getpeername(0, &sin, &i) < 0)
                fatal(argv[0], "getpeername");
line[0] = '\0';
gets(line);
sp = line;
av[0] = "finger";
i = 1;
while (1) {
        while (isspace(*sp))
                sp++;
        if (!*sp)
                break;
        if (*sp == '/' && (sp[1] == 'W' || sp[1] == 'w')) {
                sp += 2;
                av[i++] = "-l";
        }
        if (*sp && !isspace(*sp)) {
                av[i++] = sp;
                while (*sp && !isspace(*sp))
                        sp++;
                *sp = '\0';
        }
}
```

Morris worm

- This fingerd vuln 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
  - Took over hundreds of computers and shut down large chunks of the Internet
- Aside: First use of the US CFAA



https://en.wikipedia.org/wiki/Morris_worm

That was over 30 years ago!

Surely buffer overflows are no longer a problem...

Project Zero

News and updates from the Project Zero team at Google

Thursday, July 16, 2020

MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface

Posted by Mateusz Jurczyk, Project Zero

This post is the first of a multi-part series capturing my journey from discovering a vulnerable little-known Samsung image codec, to completing a remote zero-click MMS attack that worked on the latest Samsung flagship devices. New posts will be published as they are completed and will be linked here when complete.

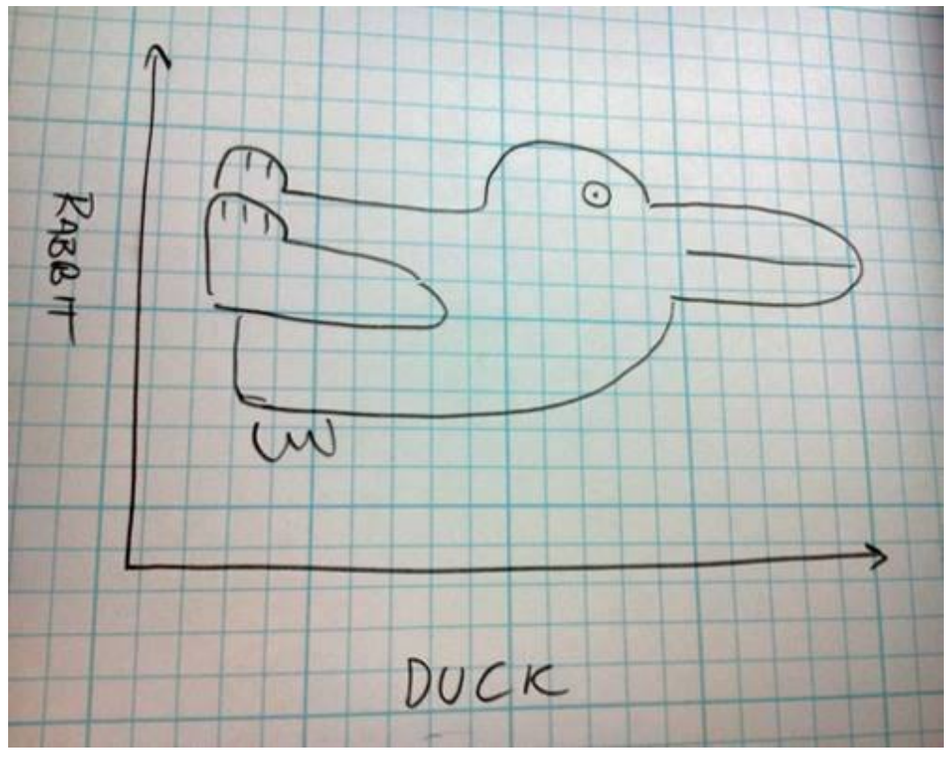
- [this post]
- MMS Exploit Part 2: Effective Fuzzing of the Qmage Codec
- MMS Exploit Part 3: Constructing the Memory Corruption Primitives
- MMS Exploit Part 4: MMS Primer, Completing the ASLR Oracle
- MMS Exploit Part 5: Defeating Android ASLR, Getting RCE

Introduction

In January 2020, I reported a large volume of crashes in a custom Samsung codec called "Qmage", present in all Samsung phones since late 2014 (Android version 4.4.4+). This codec is written in C/C++ code, and is baked deeply into the Skia graphics library, which is in turn the underlying engine used for nearly all graphics operations in the Android OS. In other words, in addition to the well-known formats such as JPEG and PNG, modern Samsung phones also natively support a proprietary Qmage format, typically denoted by the .qmg file extension. It is automatically enabled for all apps which display images, making it a prime target for remote attacks, as sending pictures is the core functionality of some of the most popular mobile apps.

How does a buffer overflow let you take over a machine?

- Your program manipulates data
- Data manipulates your program



What we need to know

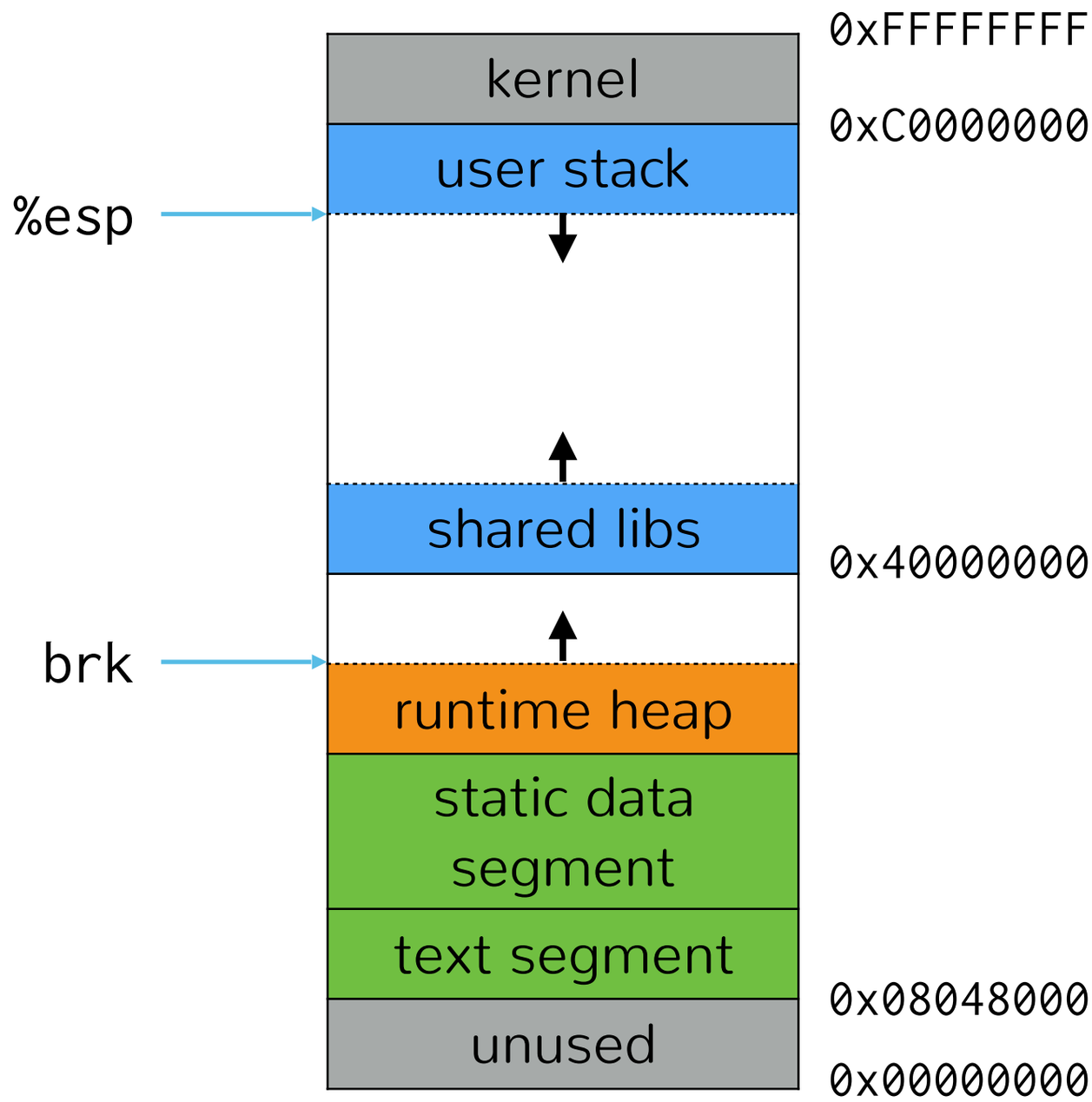
- How C arrays work
- How memory is laid out
- How the stack and function calls work
- How to turn an array overflow into an exploit

How do C arrays work

- What does `a[idx]` get compiled to?
  - *((a)+(idx))
- What does the the spec say?
  - 6.5.2.1 Array subscripting in ISO/IEC 9899:2017

Linux process memory layout

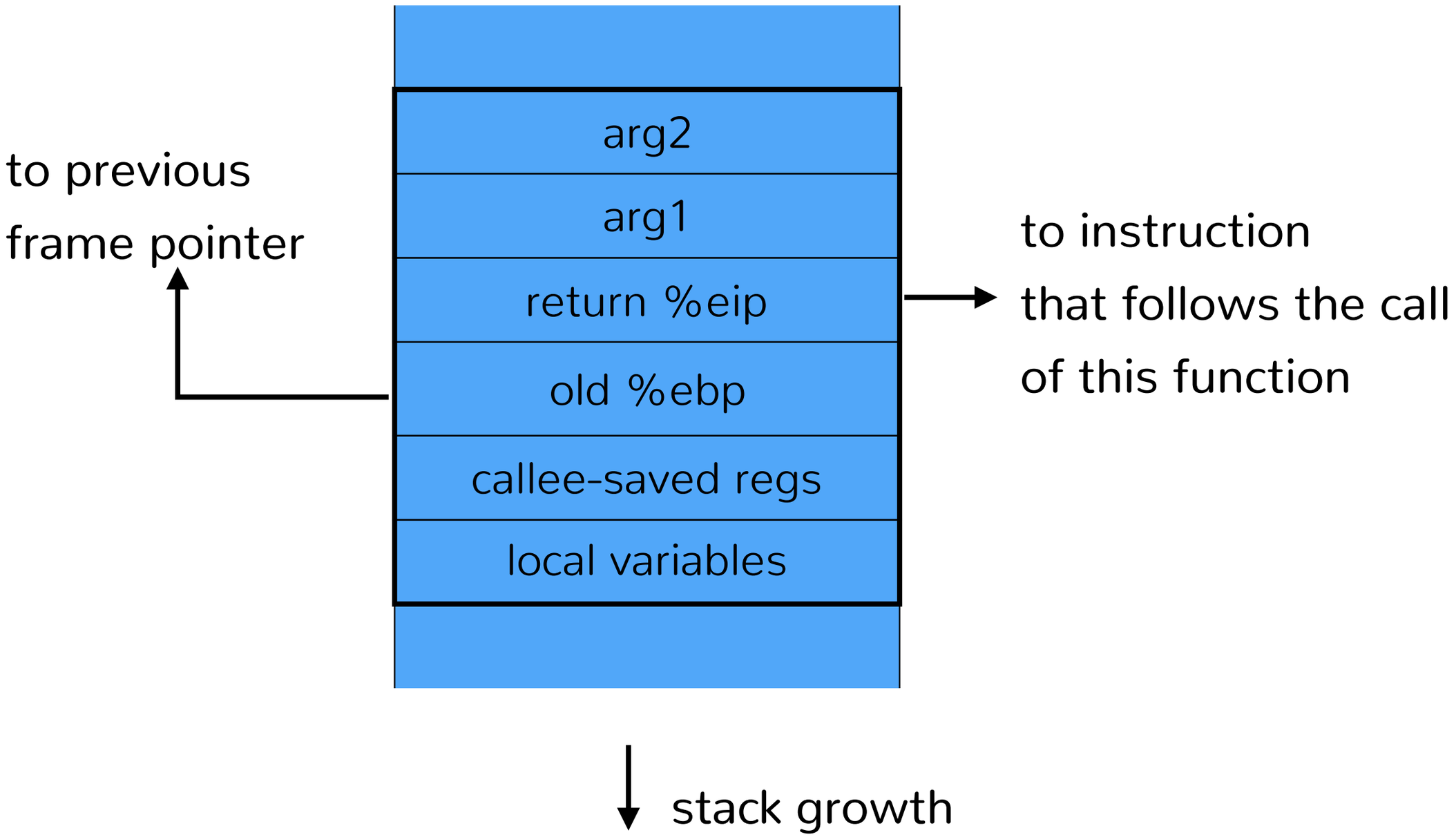
- Stack
- Heap
- Data segment
- Text sement
  - binary instructions



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


arg2
arg1
return %eip
old %ebp
callee-saved regs
local variables
to previous
frame pointer
to instruction
that follows the call
of this function
stack growth

Brief review of x86 assembly

- Two syntaxes
  - Intel syntax: `op dst, src`
  - ATT/gasm syntax: `op src, dst`
- Examples:

```
movl %eax, %edx    ->

movl $0x123, %edx  ->

movl (%ebx), %edx  ->

movl 4(%ebx), %edx ->
```

Slide adopted from David Mazières

Brief review of x86 assembly

- Two syntaxes
  - Intel syntax: `op dst, src`
  - ATT/gasm syntax: `op src, dst`
- Examples:

```
movl %eax, %edx     -> edx = eax

movl $0x123, %edx   -> edx = 0x123

movl (%ebx), %edx   -> edx= *((int32_t*) ebx)

movl 4(%ebx), %edx  -> edx= *((int32_t*) (ebx+4))
```

Slide adopted from David Mazières

Brief review of stack instructions

```
pushl %eax              ->

popl %eax               ->

call $0x12345           ->

ret                     ->

leave                   ->
```

Slide adopted from David Mazières

Brief review of stack instructions

```
pushl %eax                  -> subl $4, %esp
                               movl %eax, (%esp)

popl %eax                   -> movl (%esp), %eax
                               addl $3, %esp

call $0x12345               -> pushl %eip
                               movl $0x12345, %eip

ret                         -> popl %eip

leave                       -> movl %ebp, %esp
                               pop %ebp
```

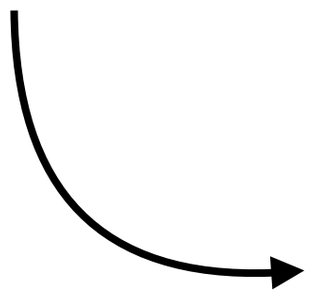
Slide adopted from David Mazières

Example 0

```c
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);
}
```

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 }
```



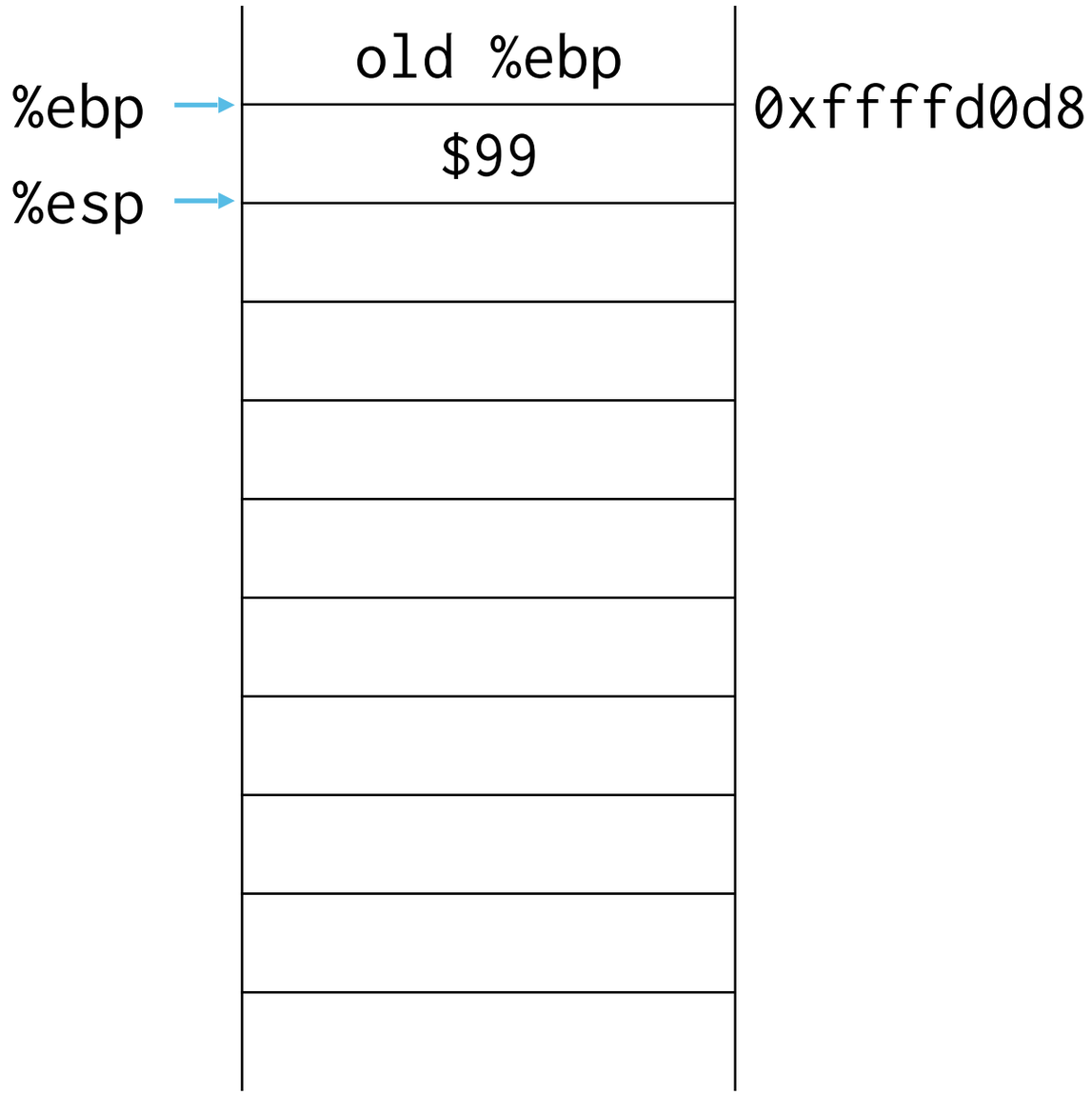
```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)
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 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
```

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

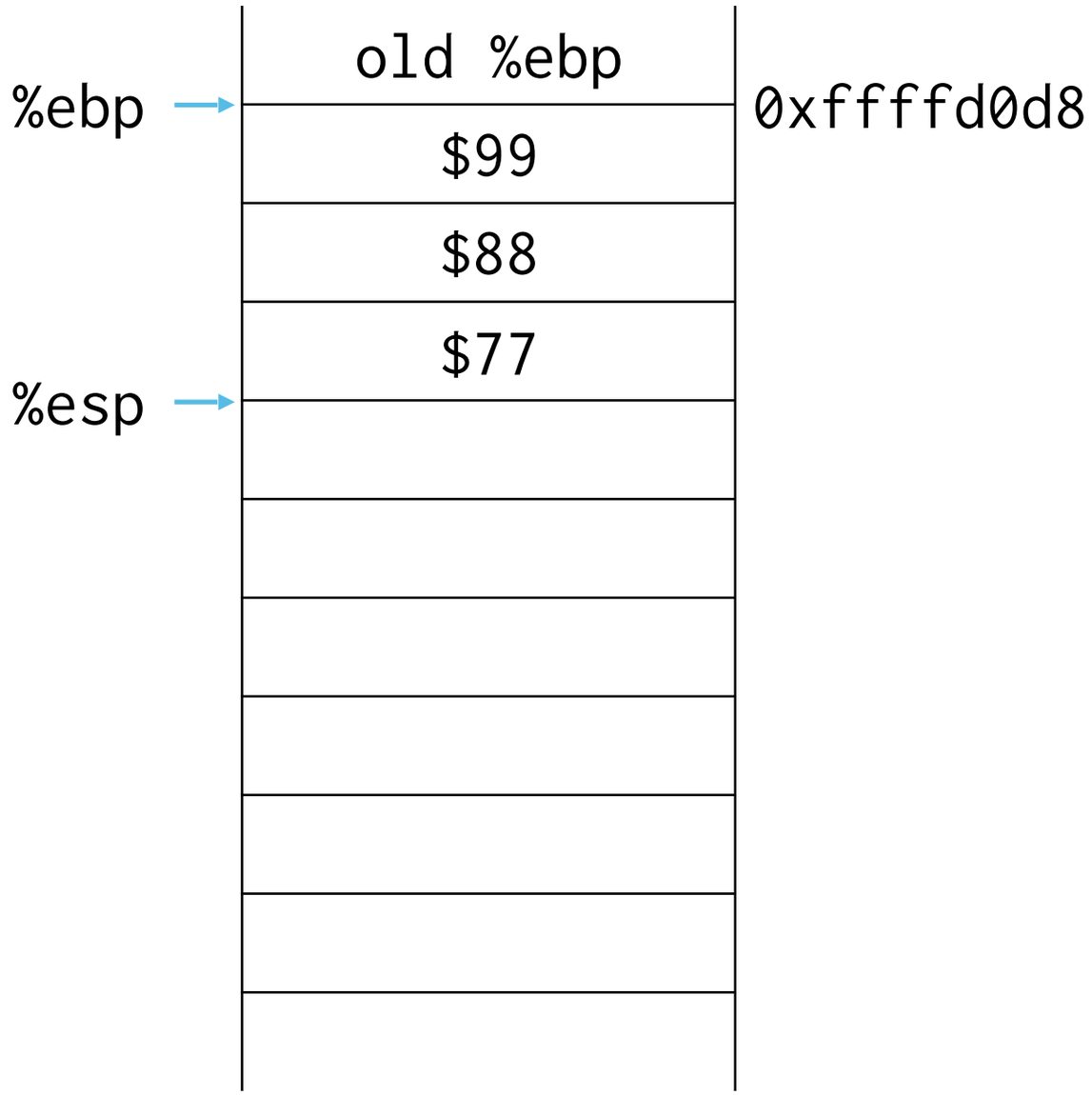
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



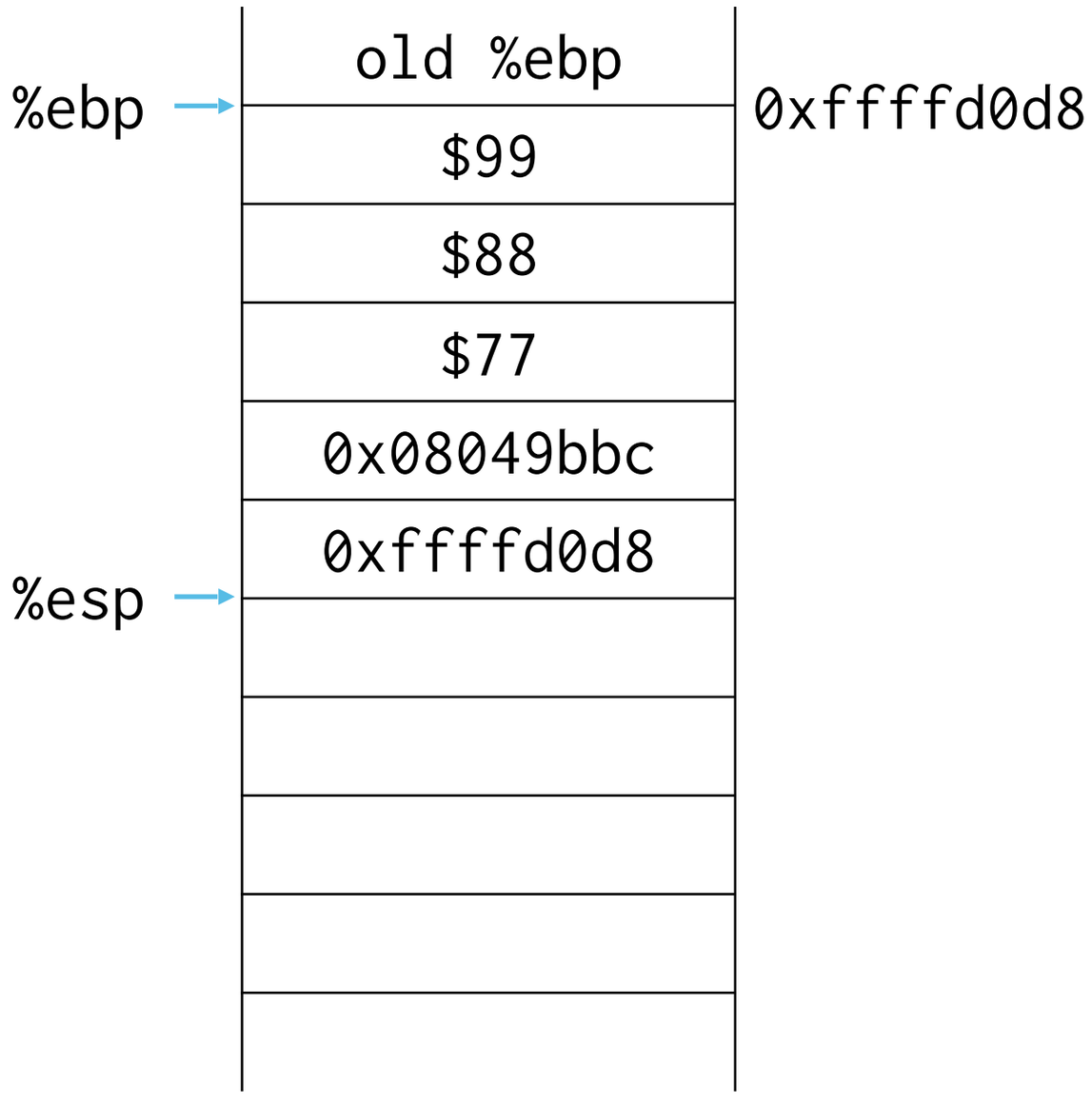
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



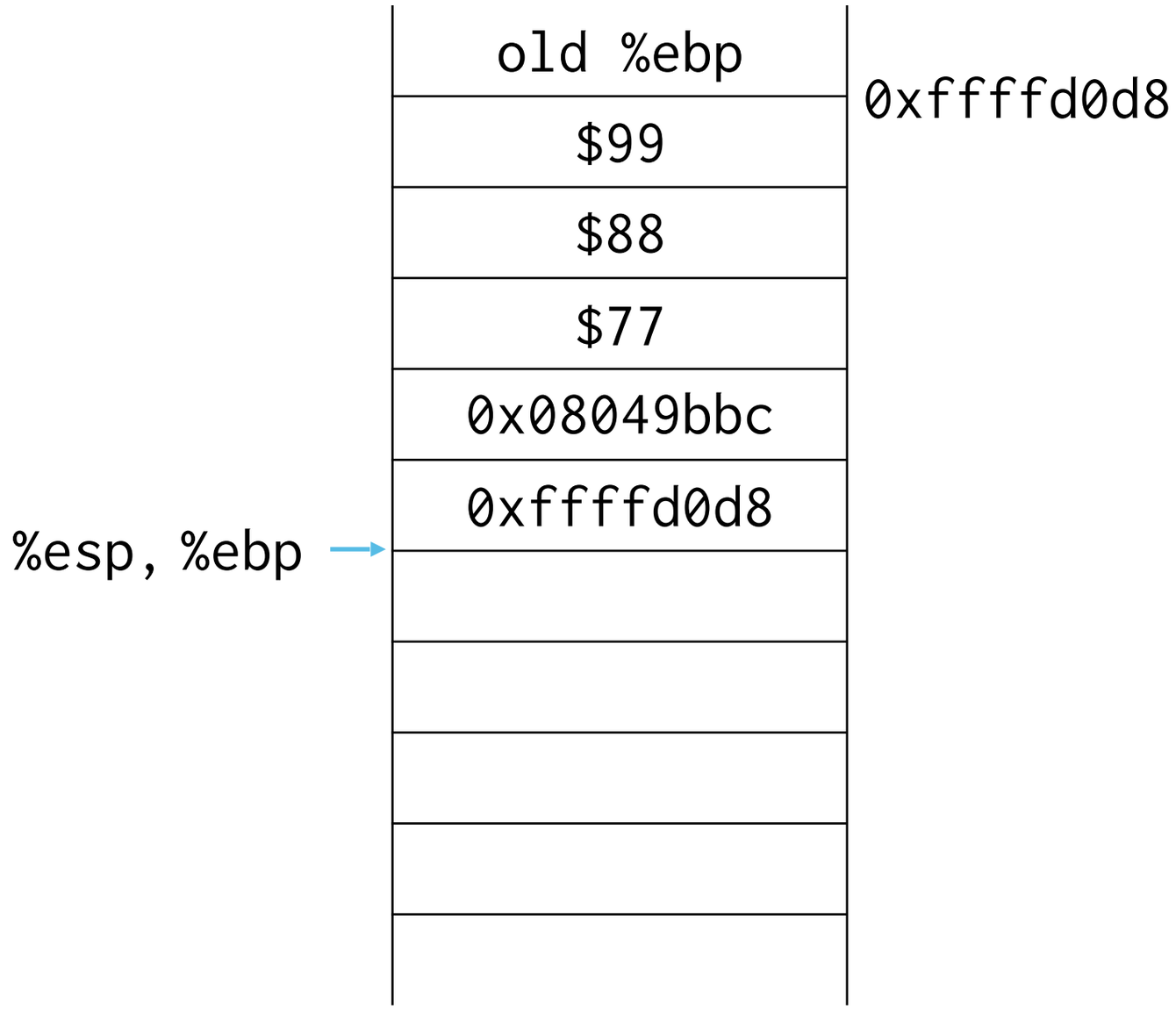
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```

%eip = 0x08049ba7

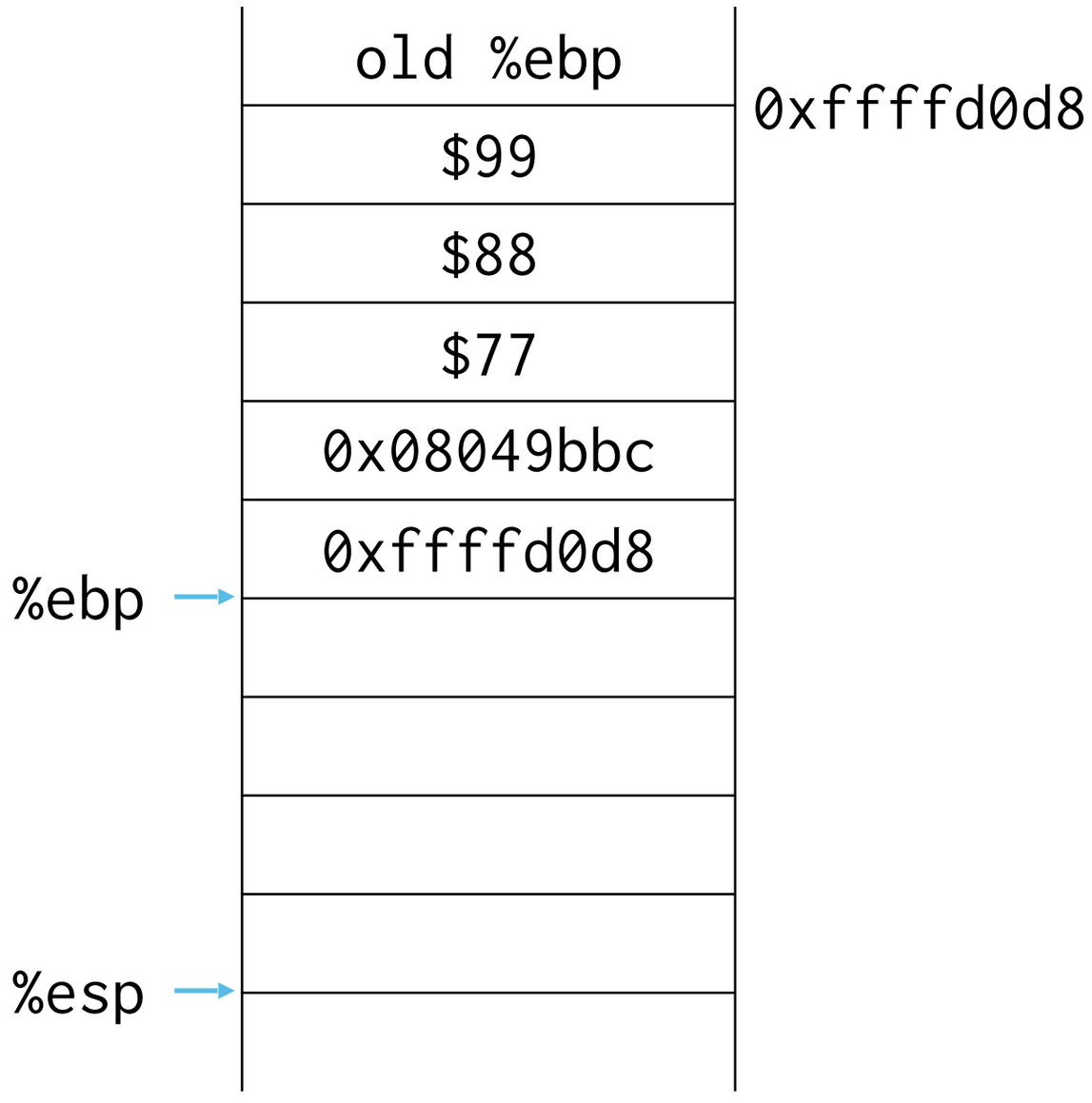
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



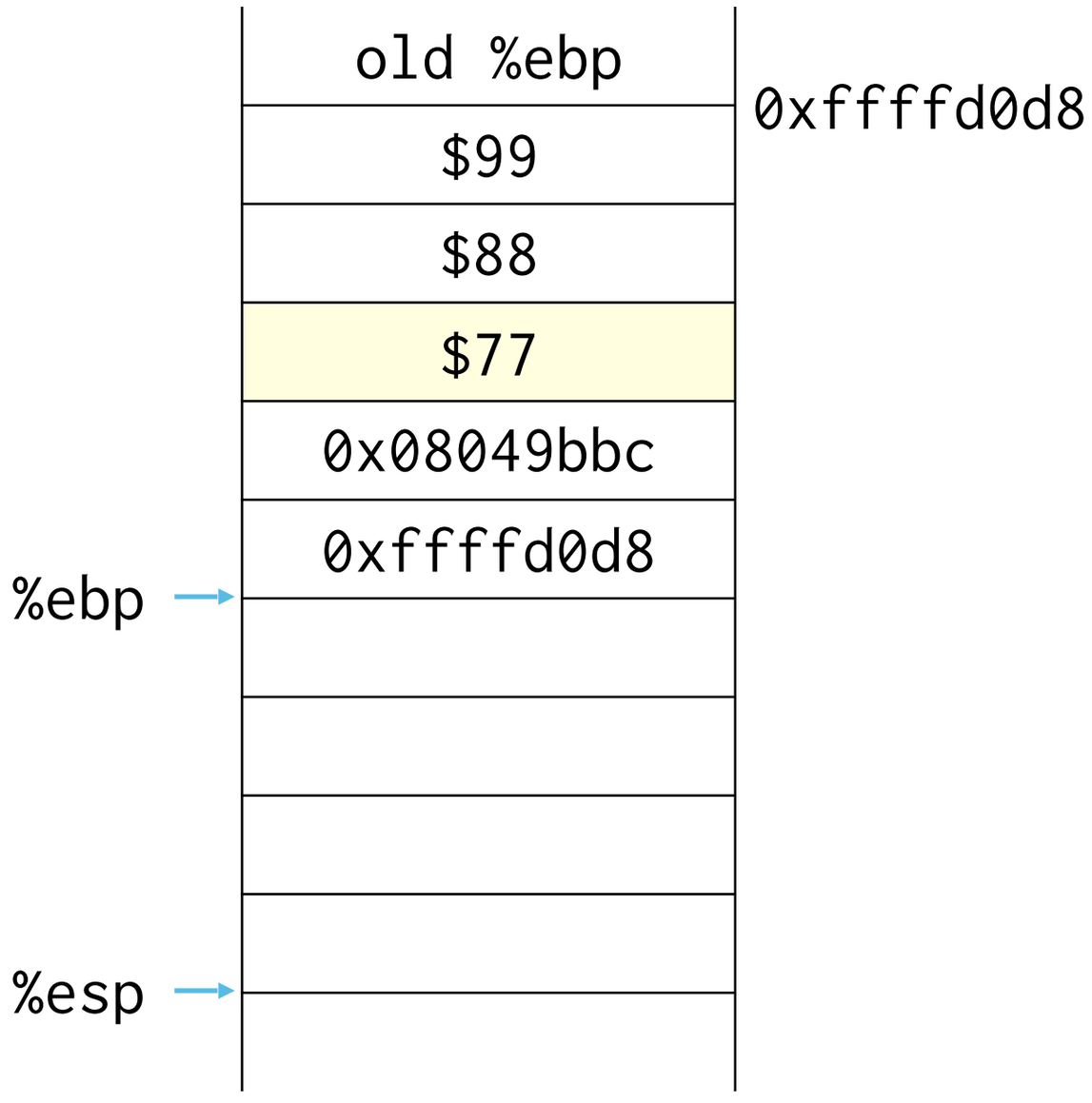
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



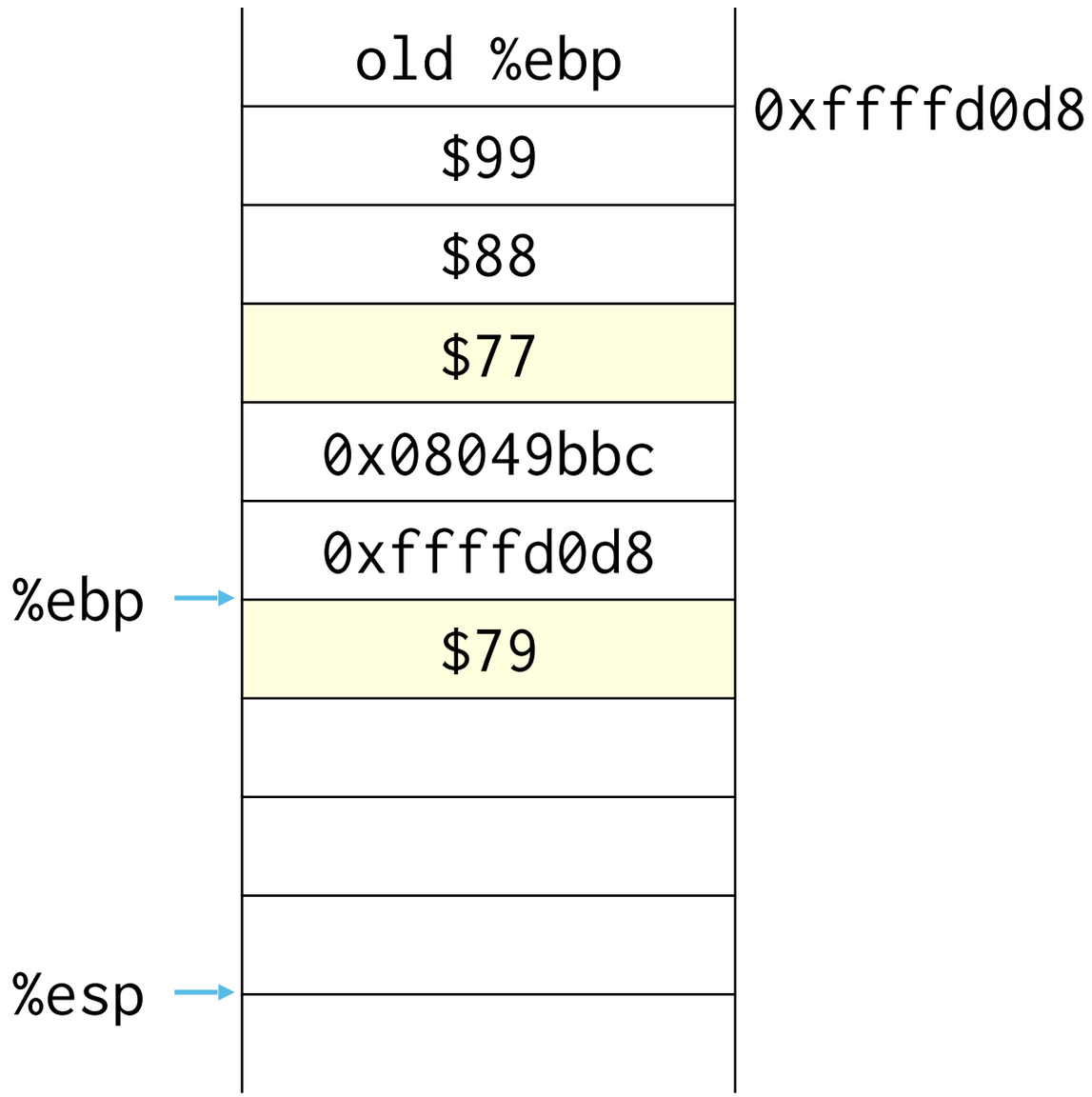
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



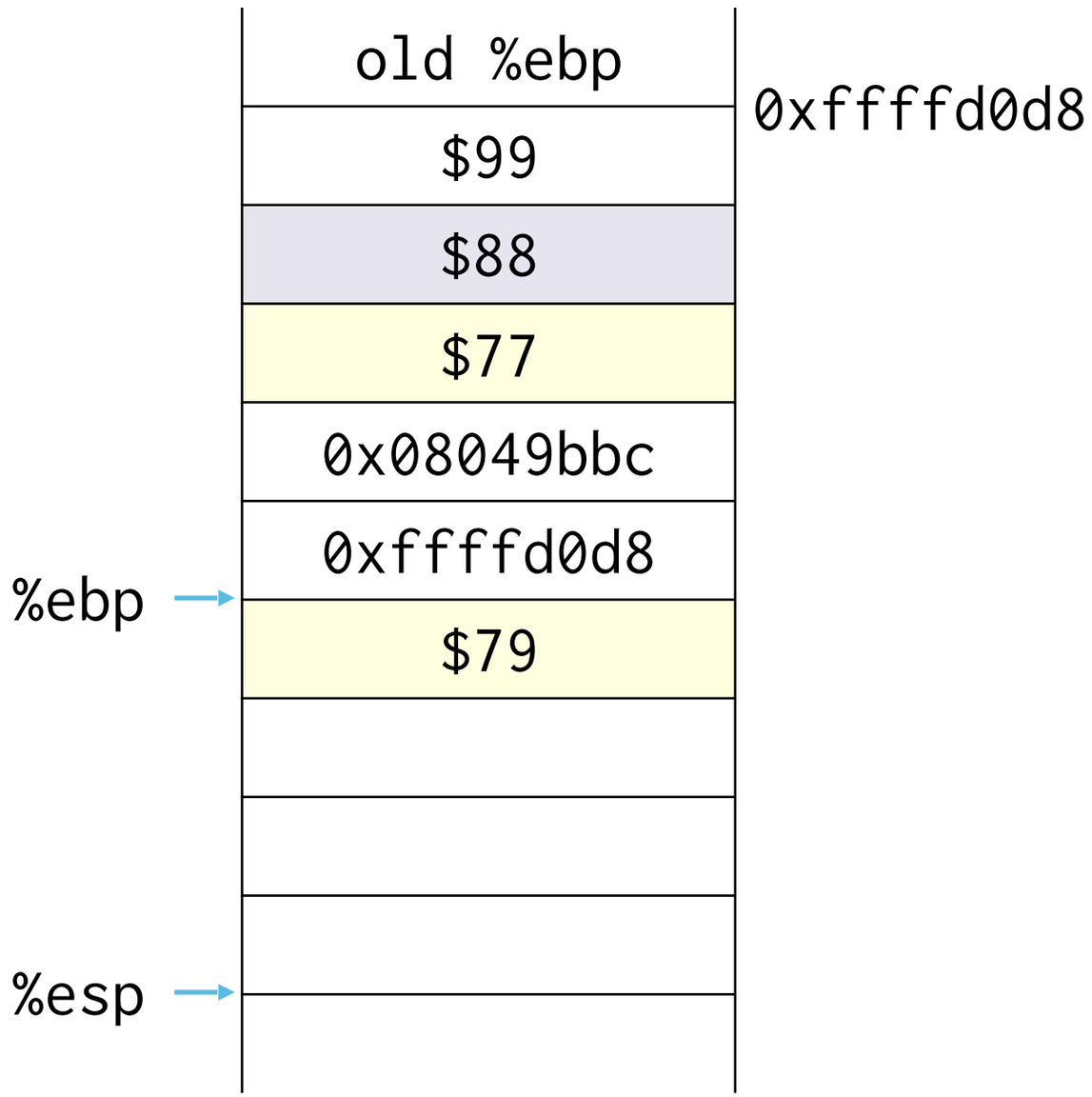
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



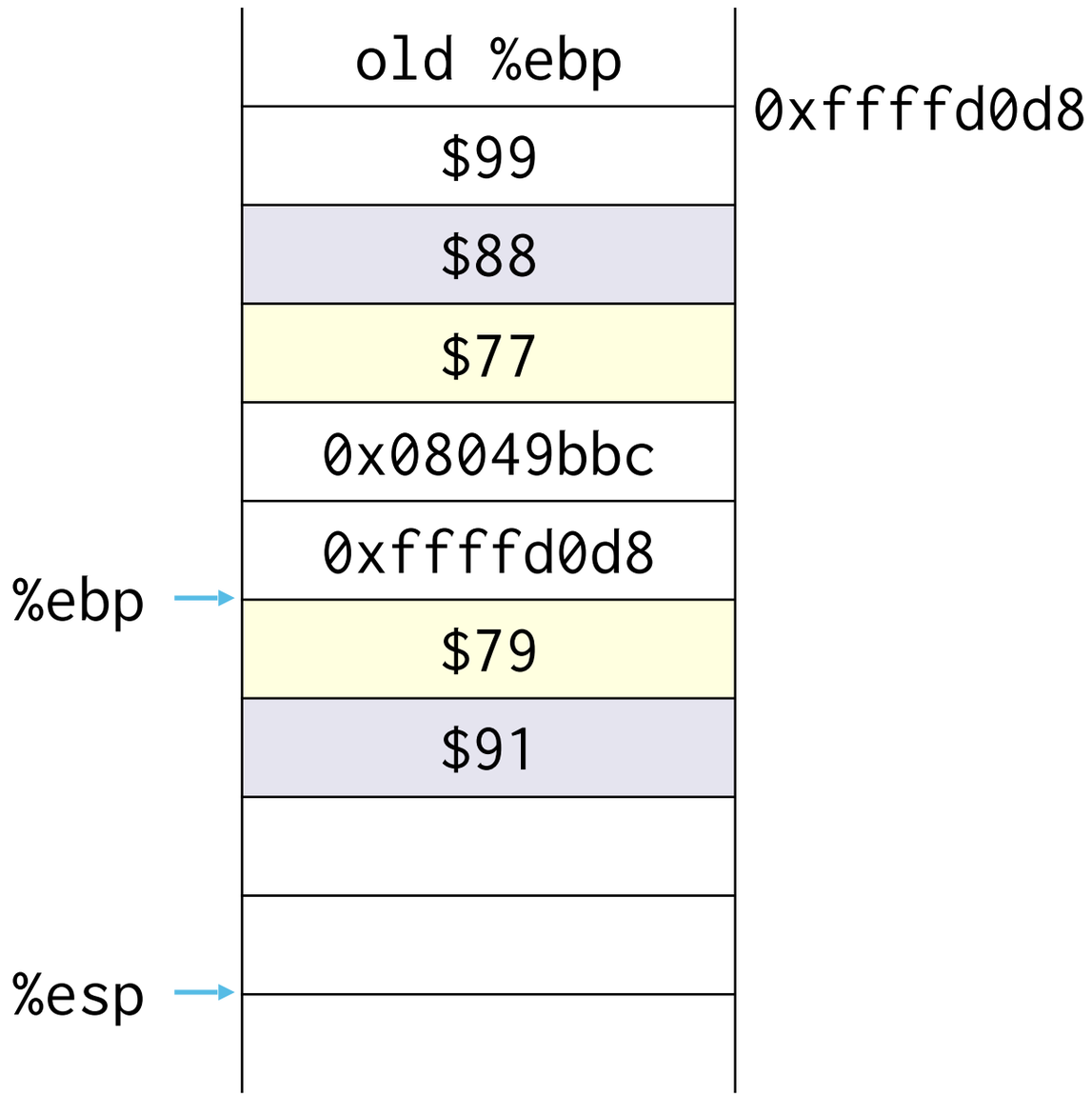
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



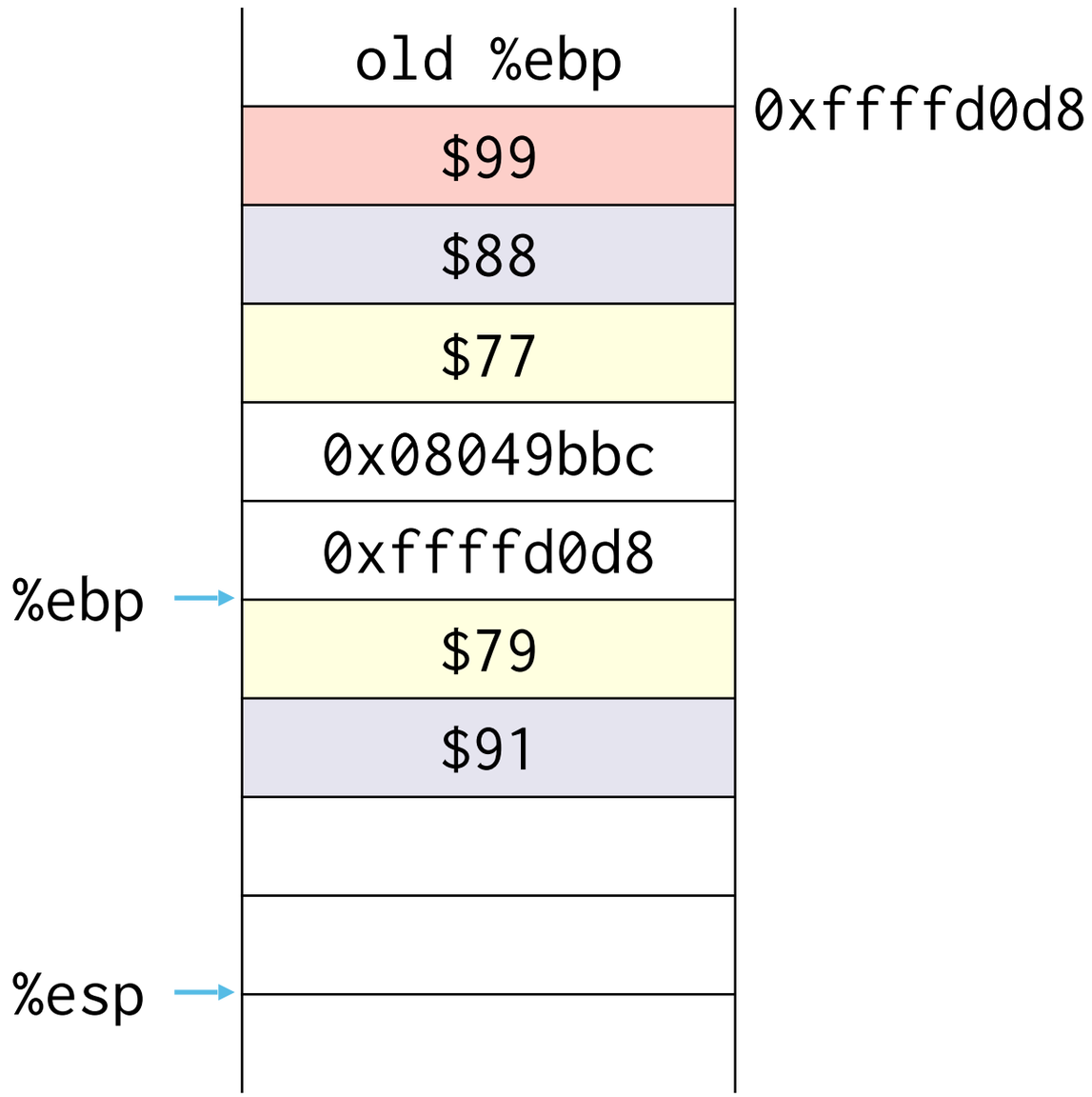
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



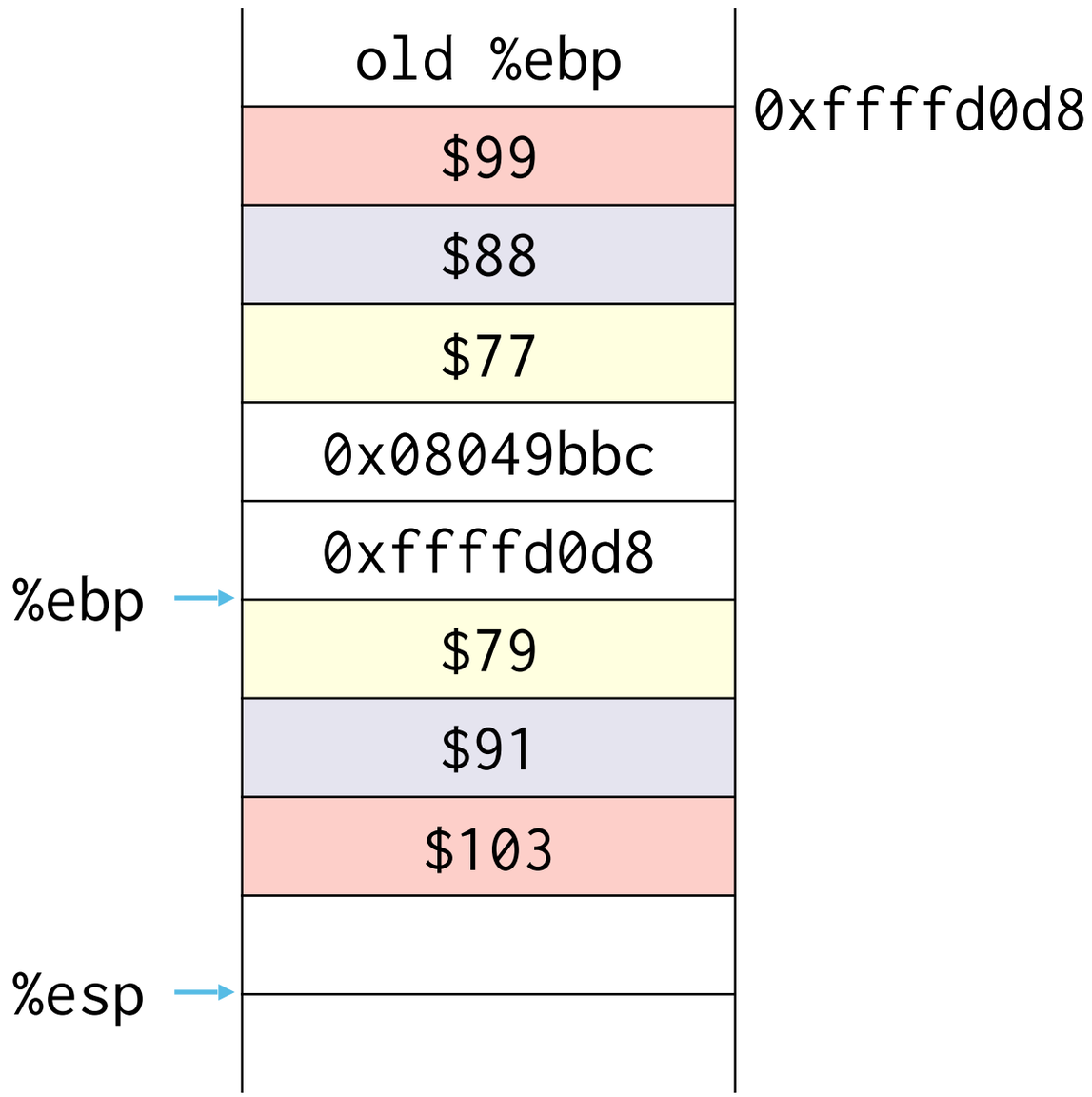
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



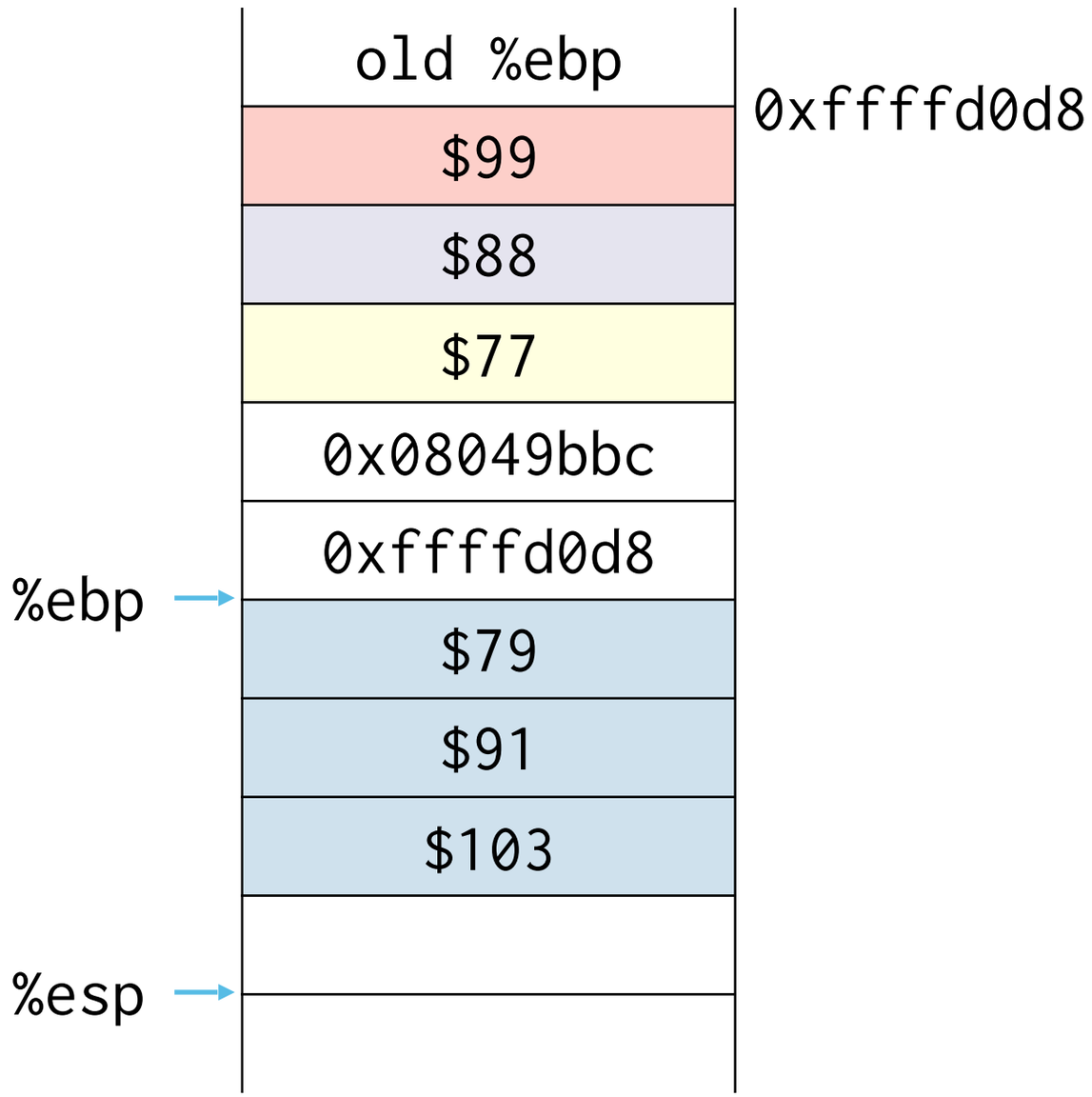
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



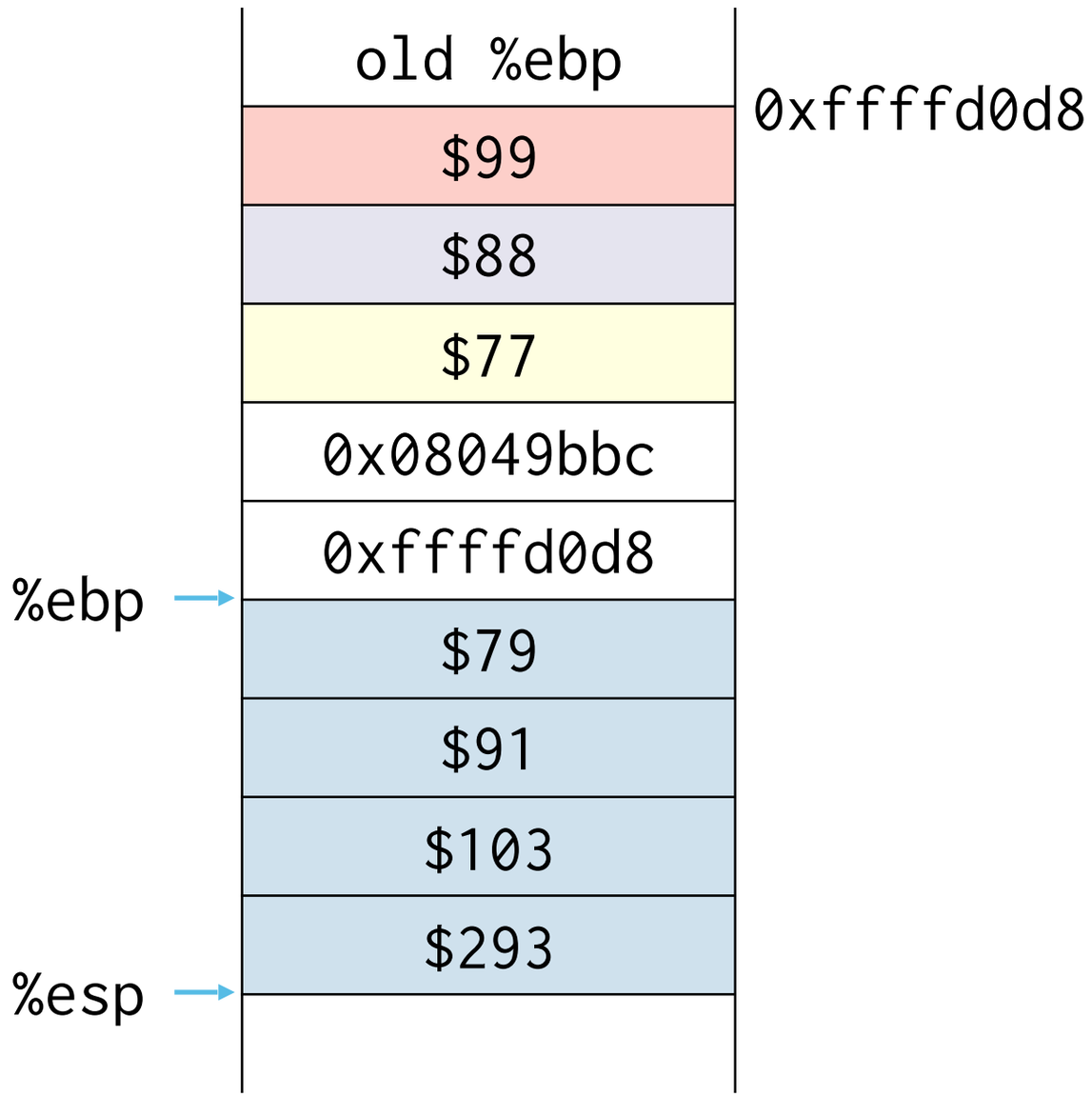
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



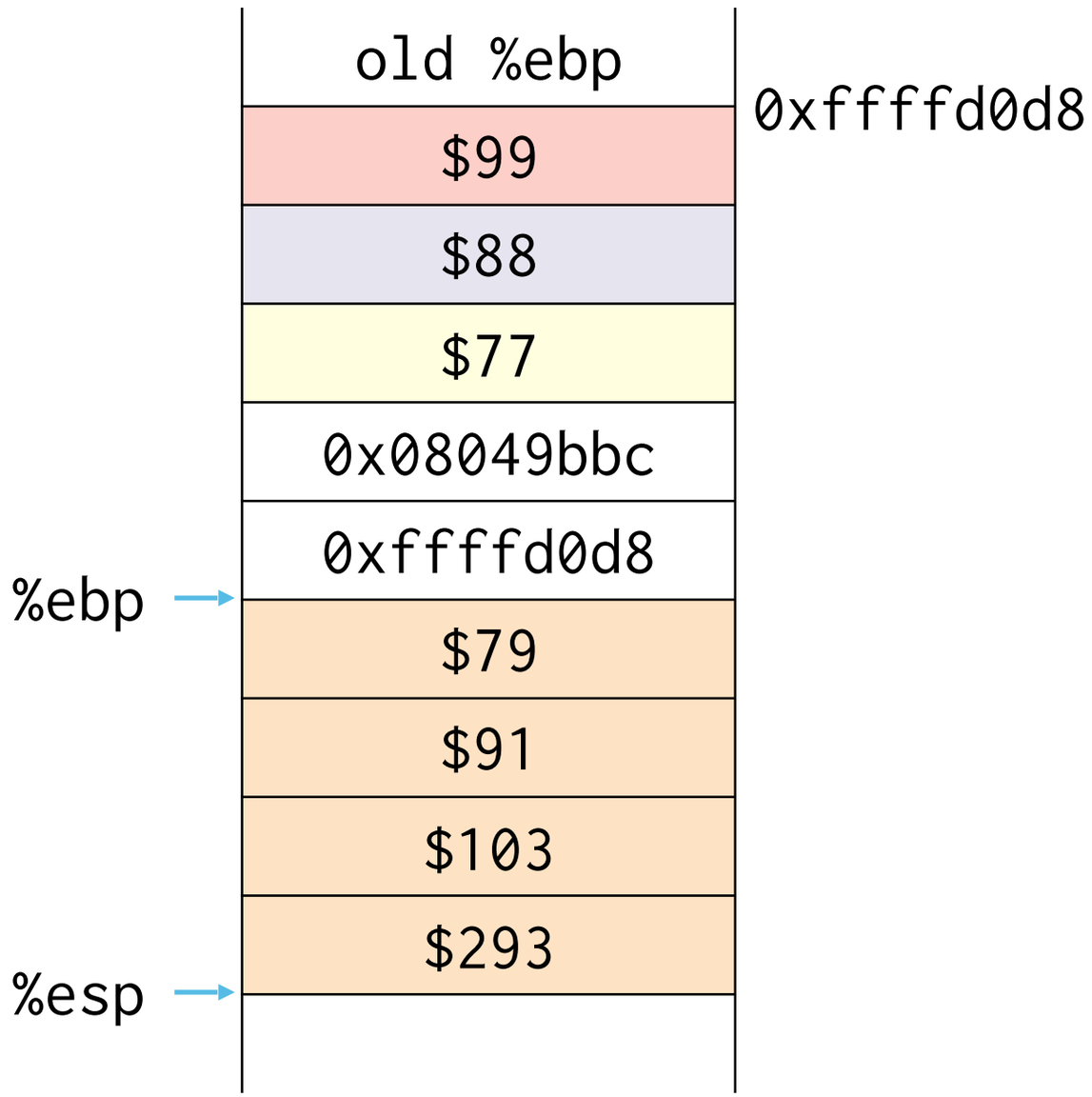
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



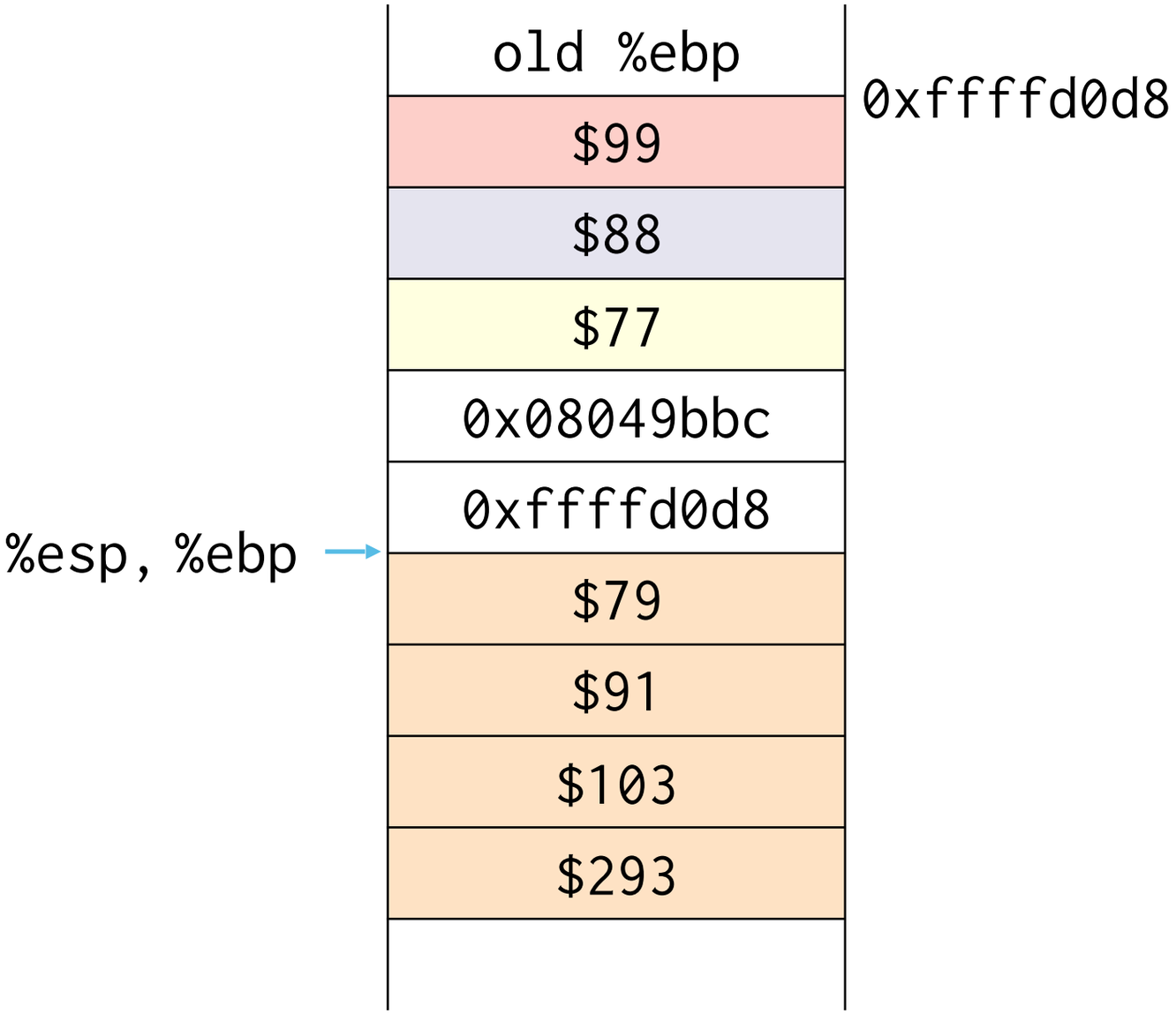
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



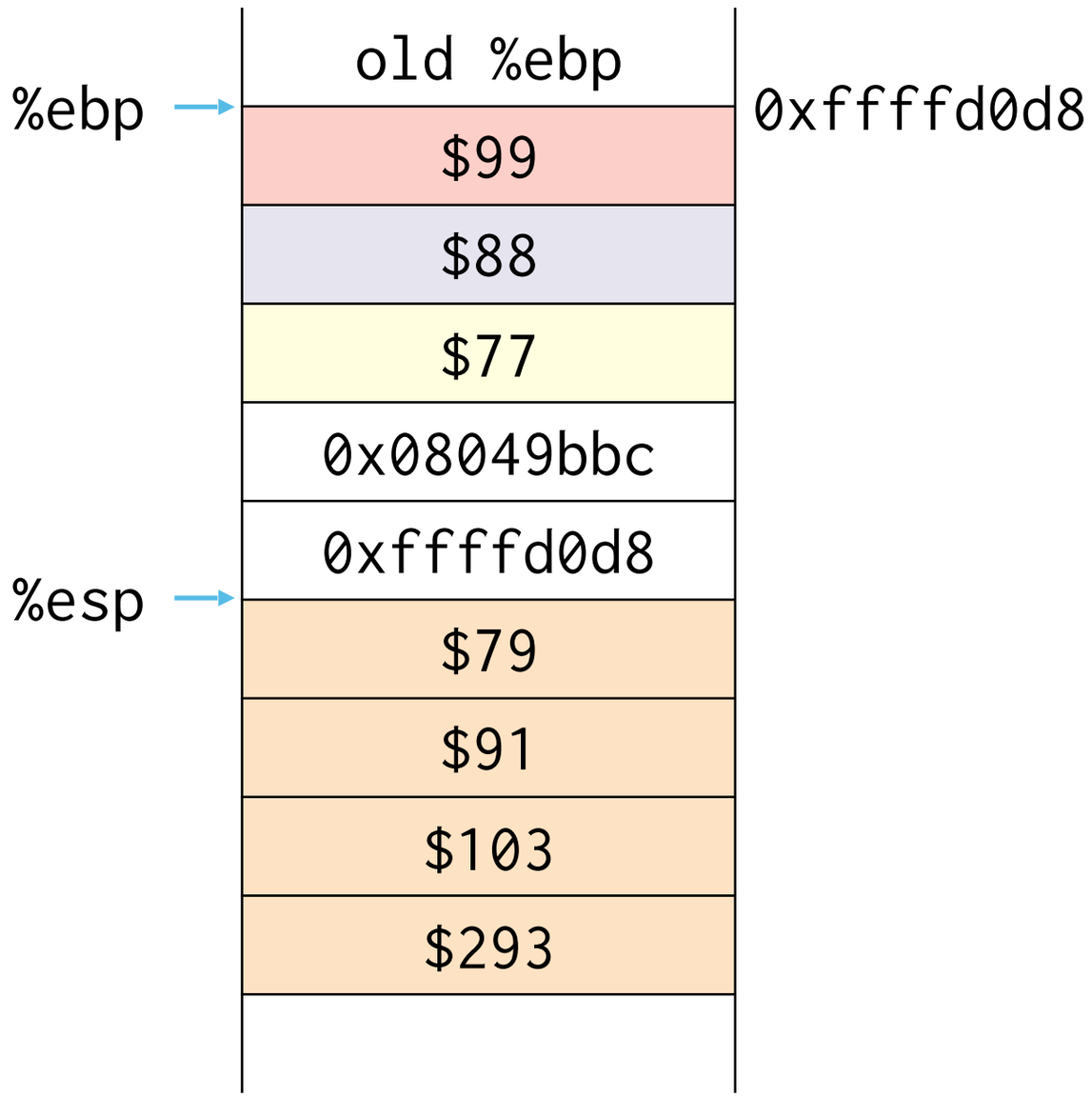
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



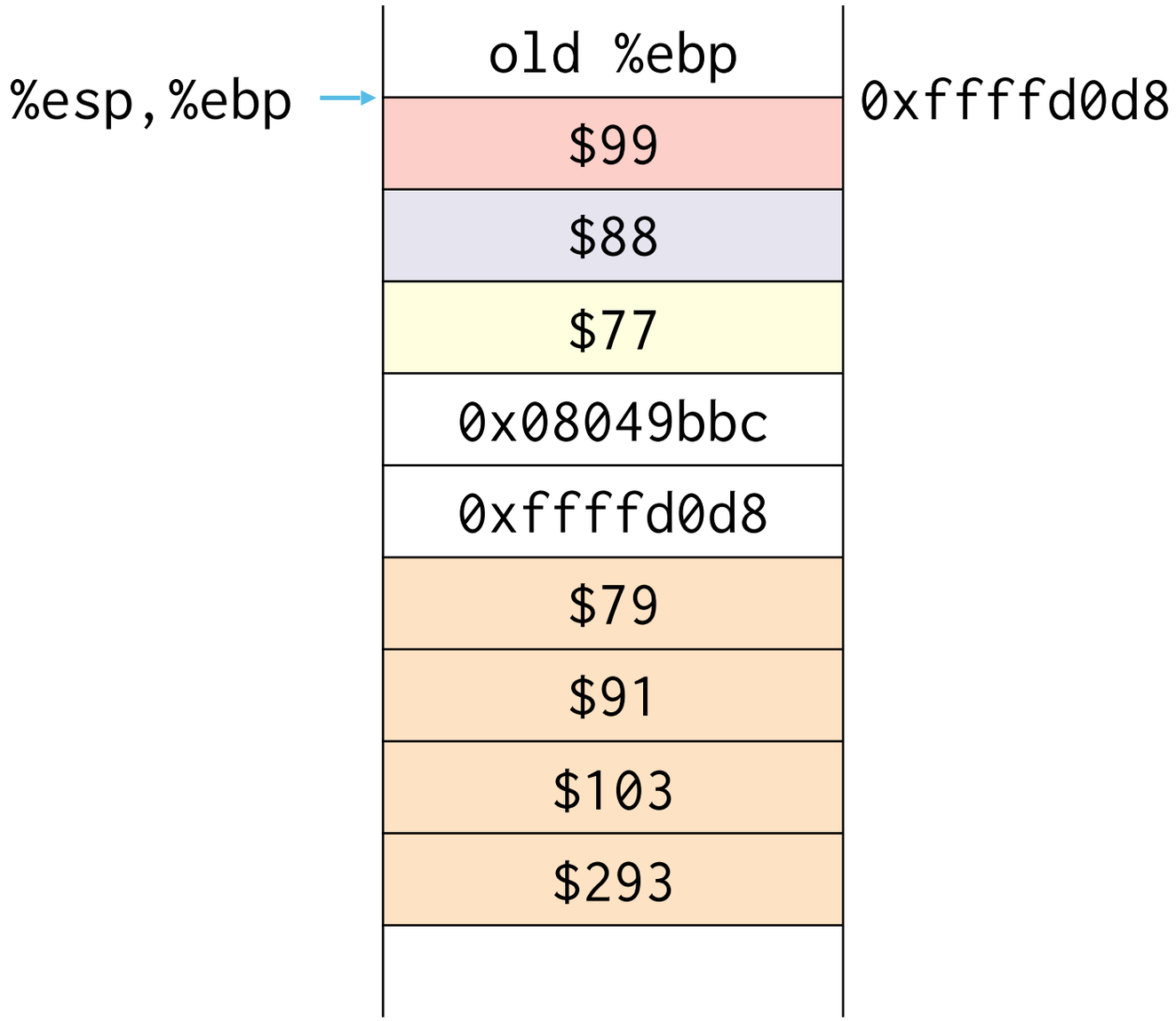
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax, -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
```



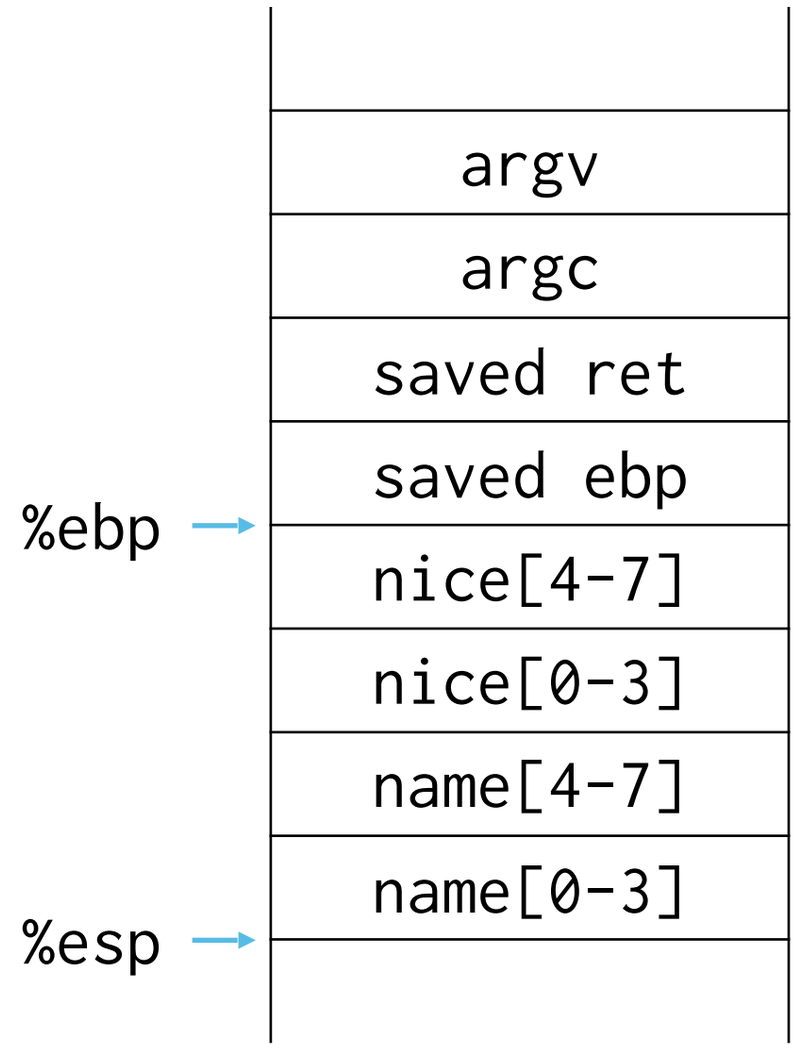
%eip = 0x08049bbc

Example 1

```c
#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;
}
```

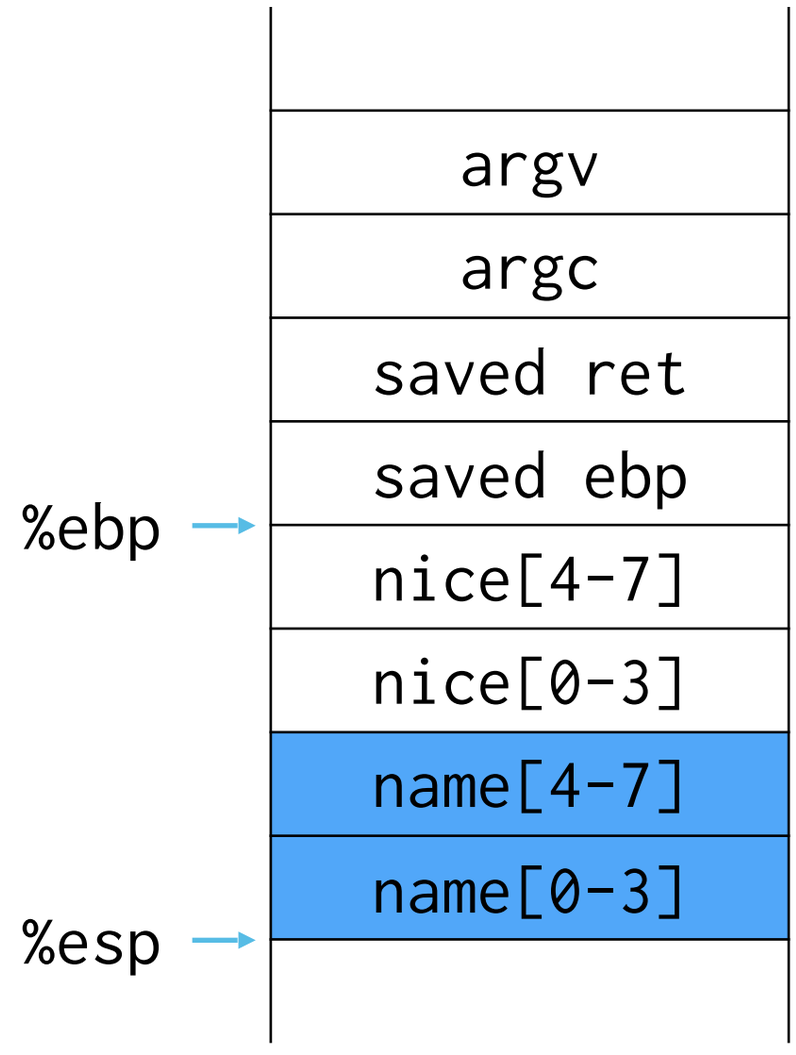
Example 1

```
#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;
}
```



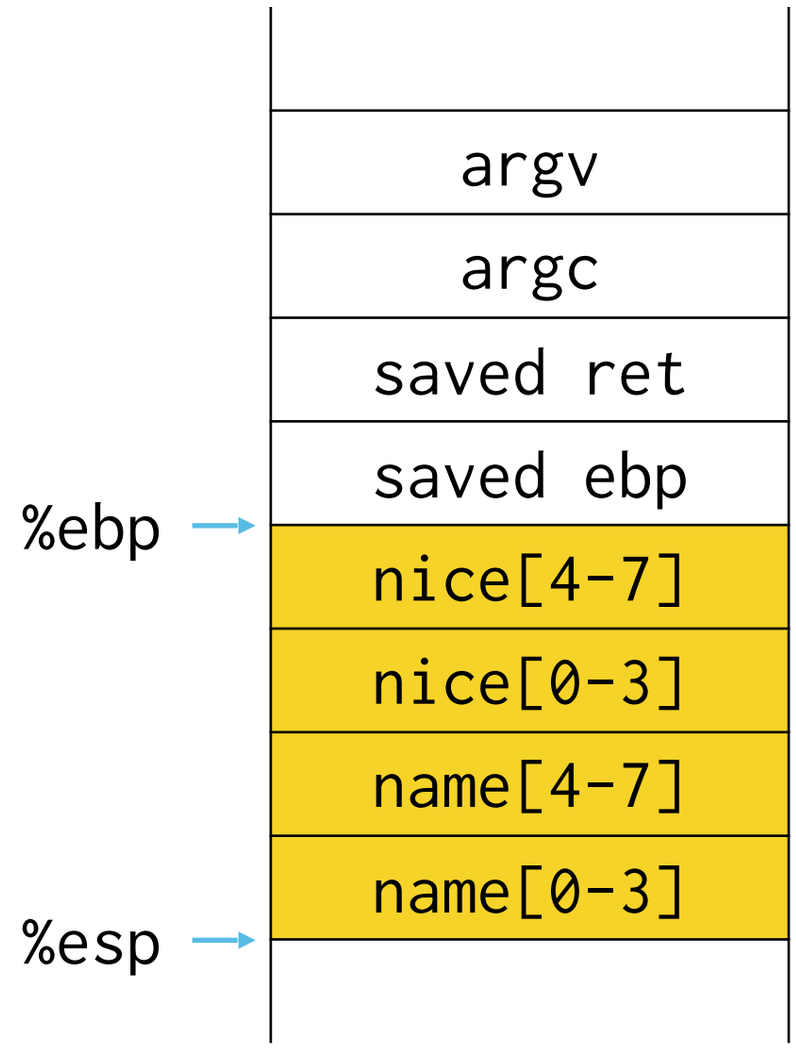
Example 1

```c
#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;
}
```



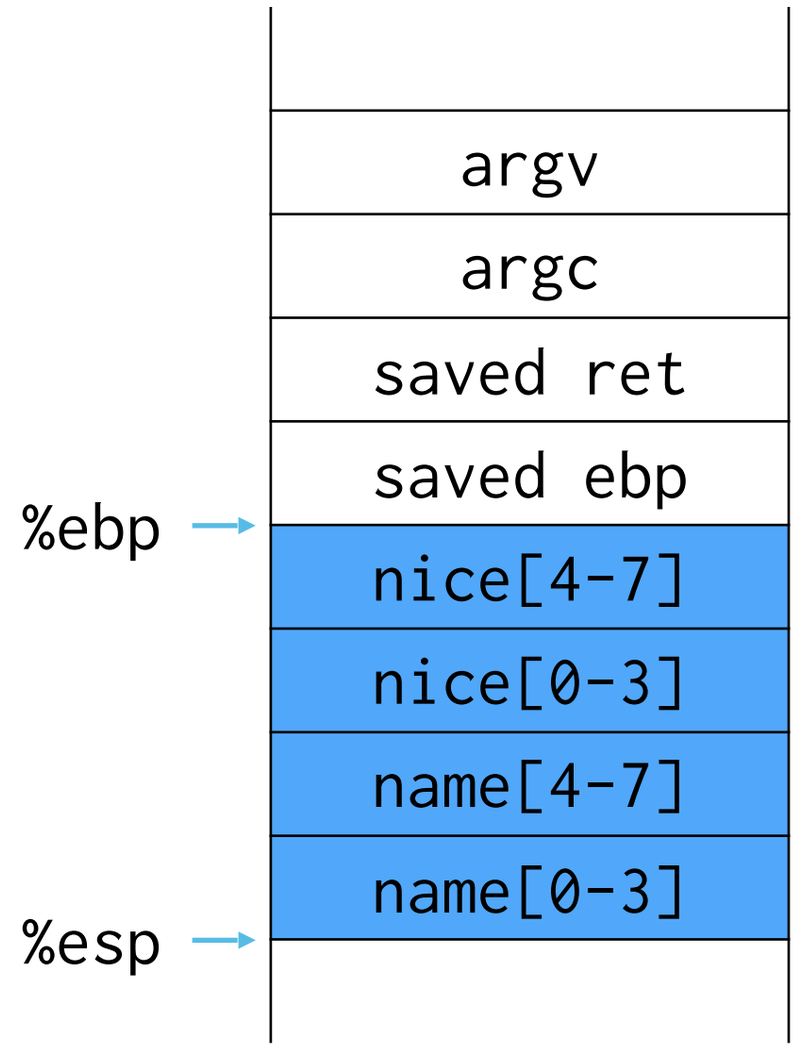
Example 1

```
#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

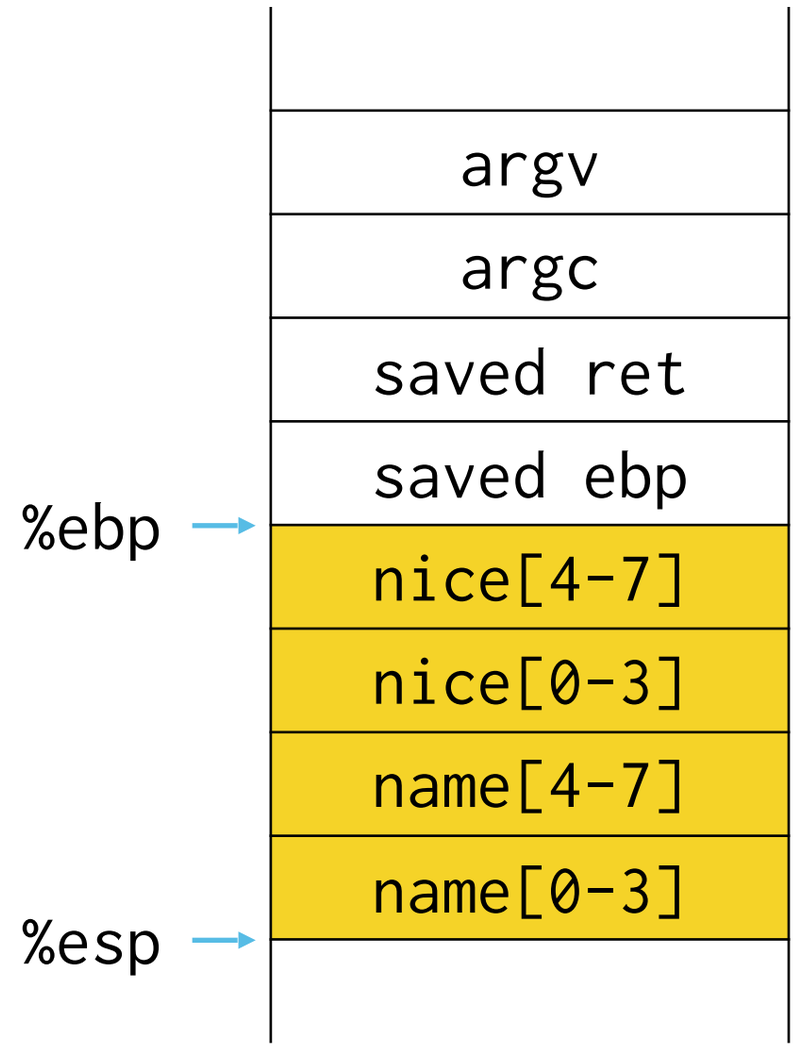
```
#include <stdio.h>
#include <string.h>

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

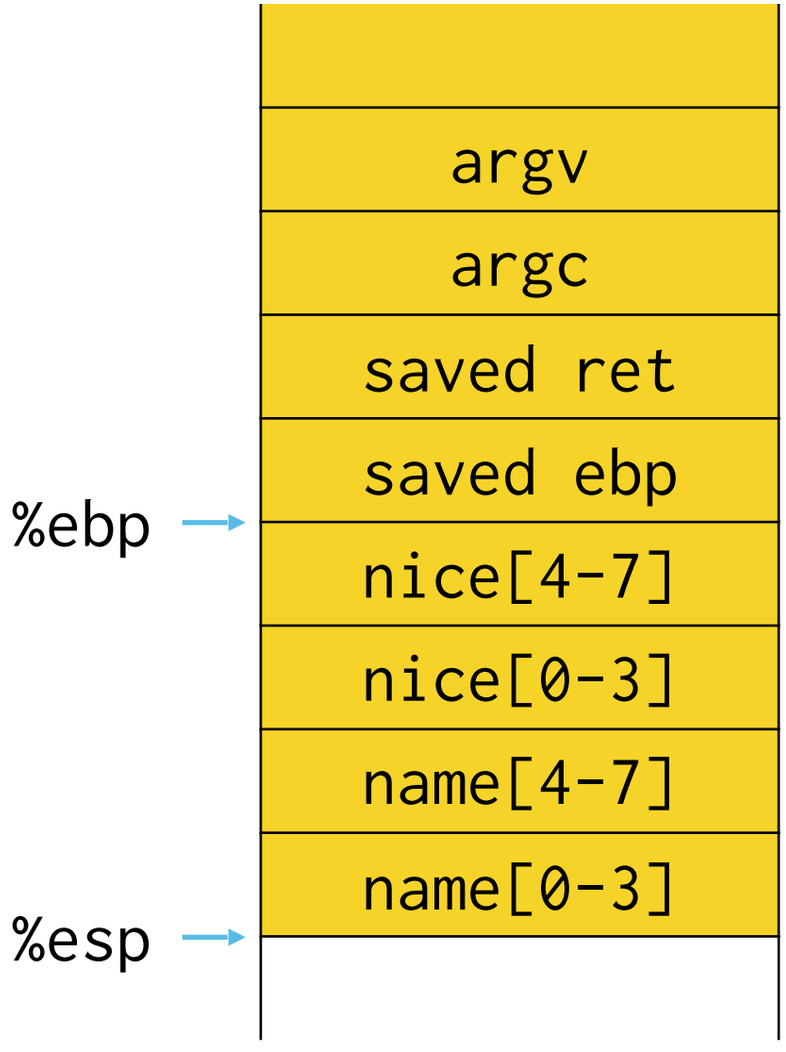
```
#include <stdio.h>
#include <string.h>

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

Example 2

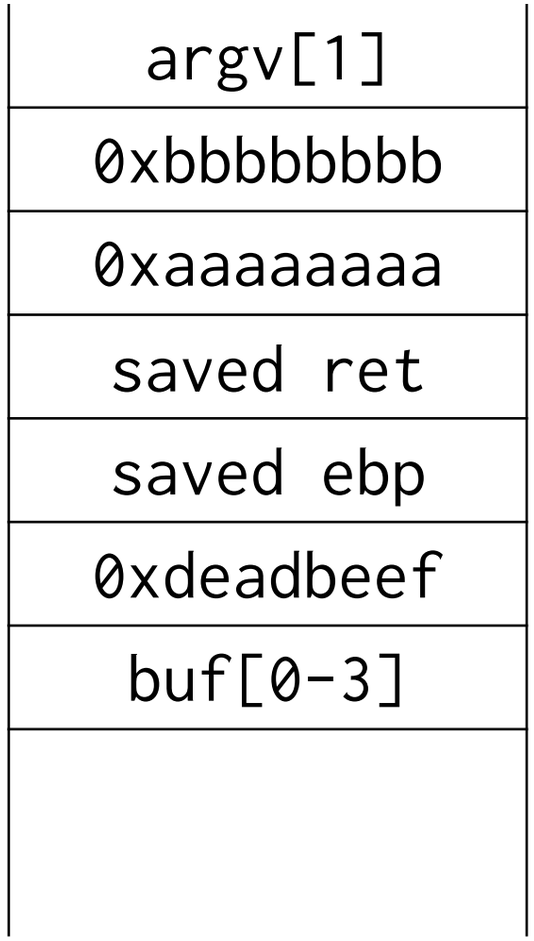
```c
#include <stdio.h>
#include <string.h>

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;
}
```

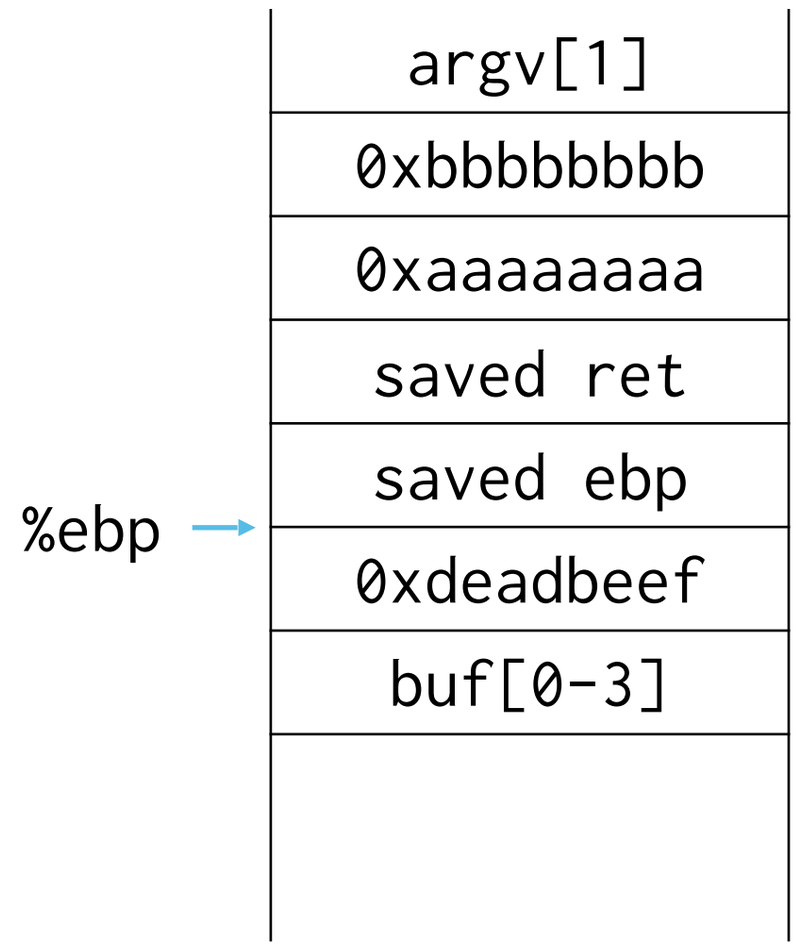
Example 2

```c
#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;
}
```



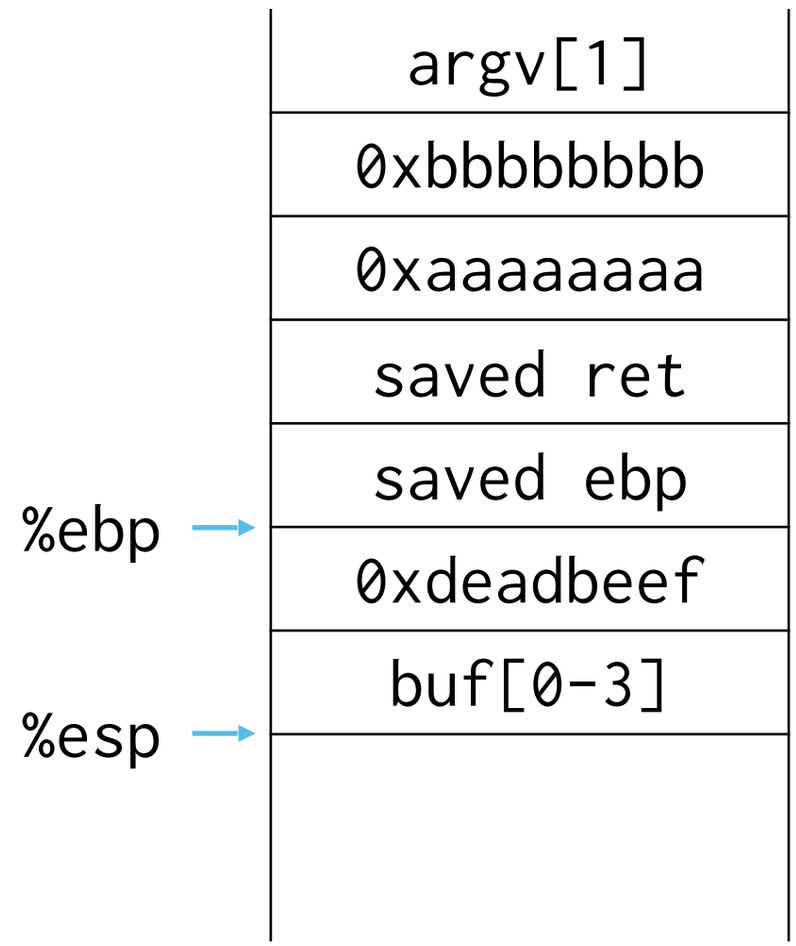
Example 2

```c
#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;
}
```



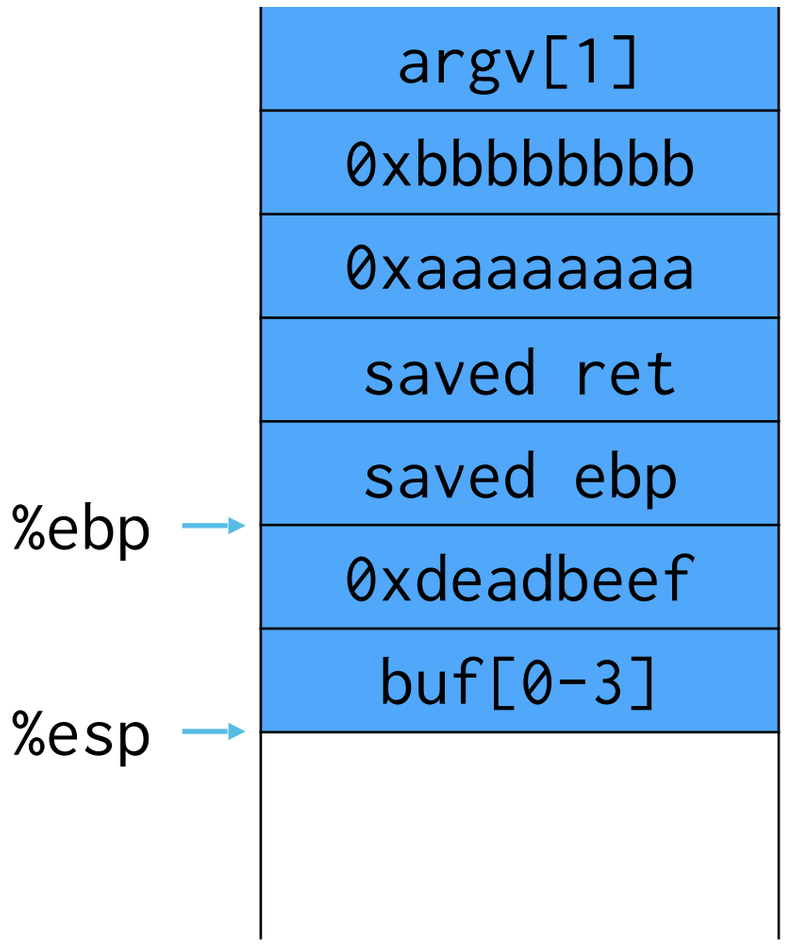
Example 2

```c
#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;
}
```



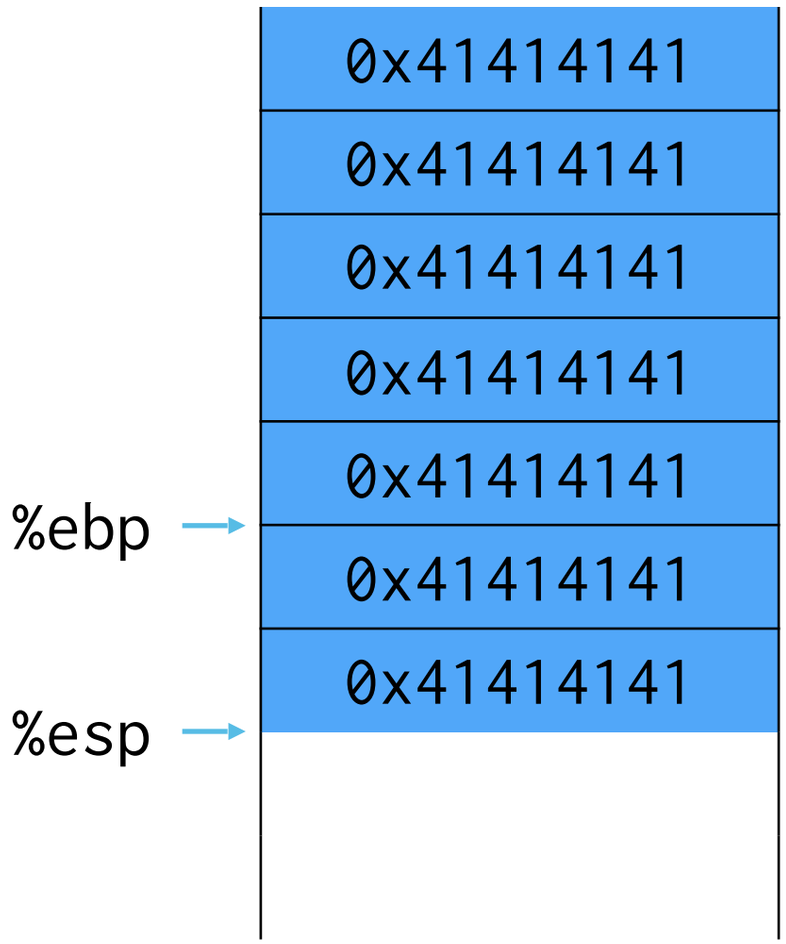
Example 2

```
#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

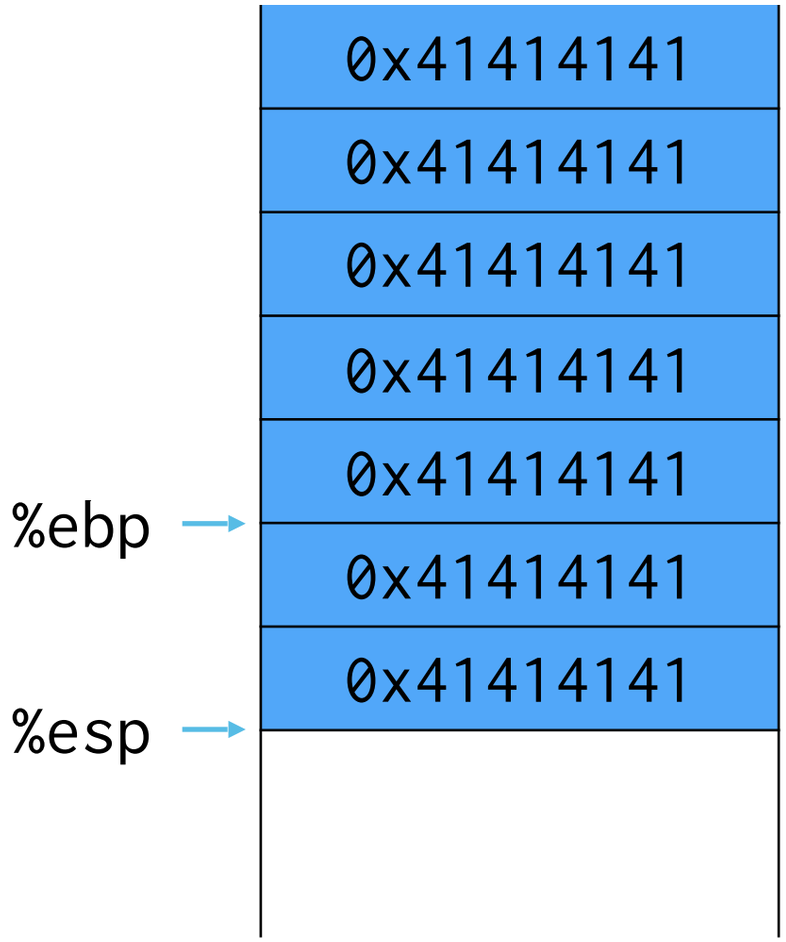
```
#include <stdio.h>
#include <string.h>

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 “AAAAAAAA...”

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;
}
```

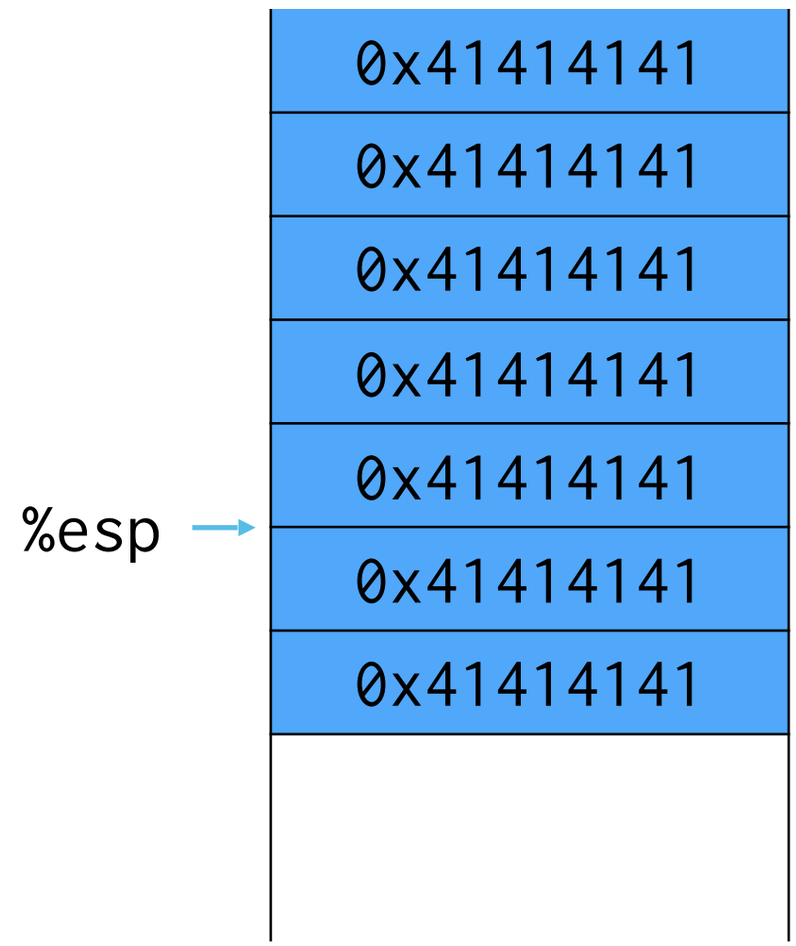


Example 2

```
#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

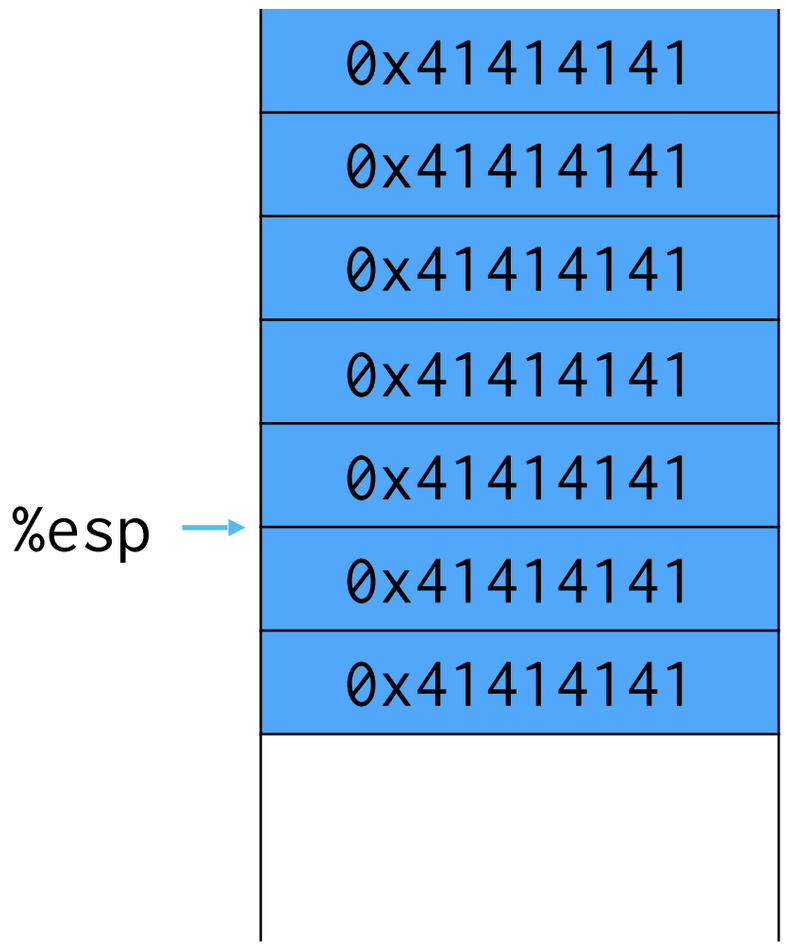
```
#include <stdio.h>
#include <string.h>

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;
}
```



%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

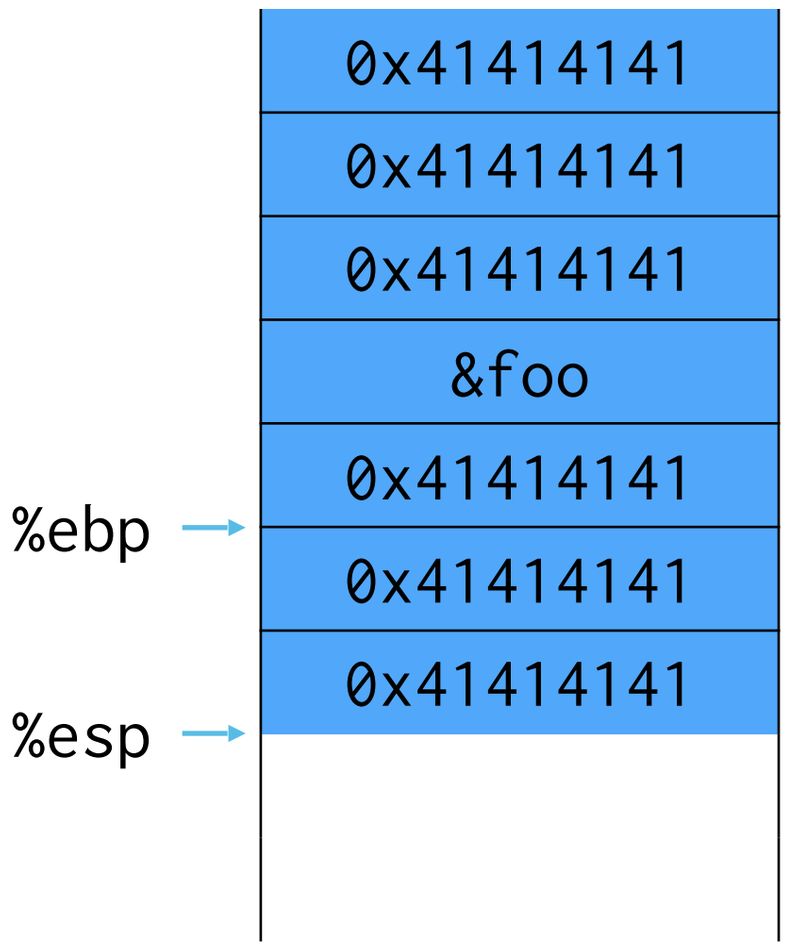
```
%eip = 0x41414141
```

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?

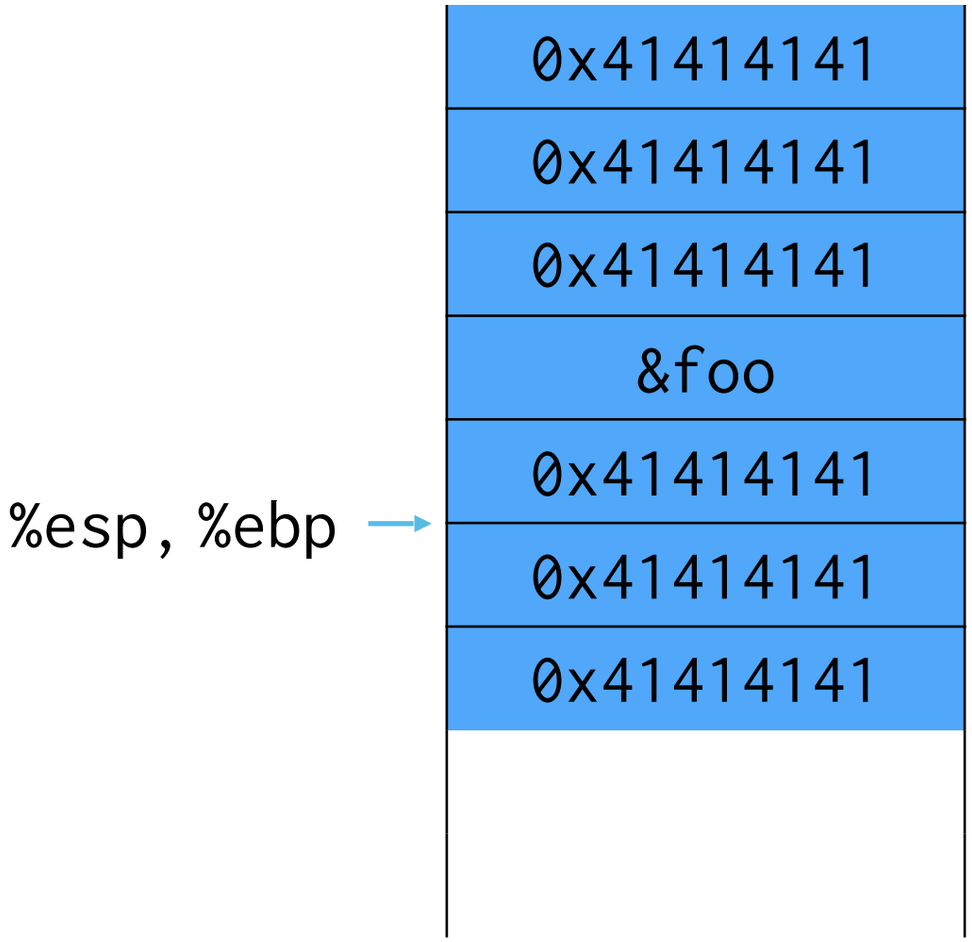
Existing functions

```
#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;
}
```



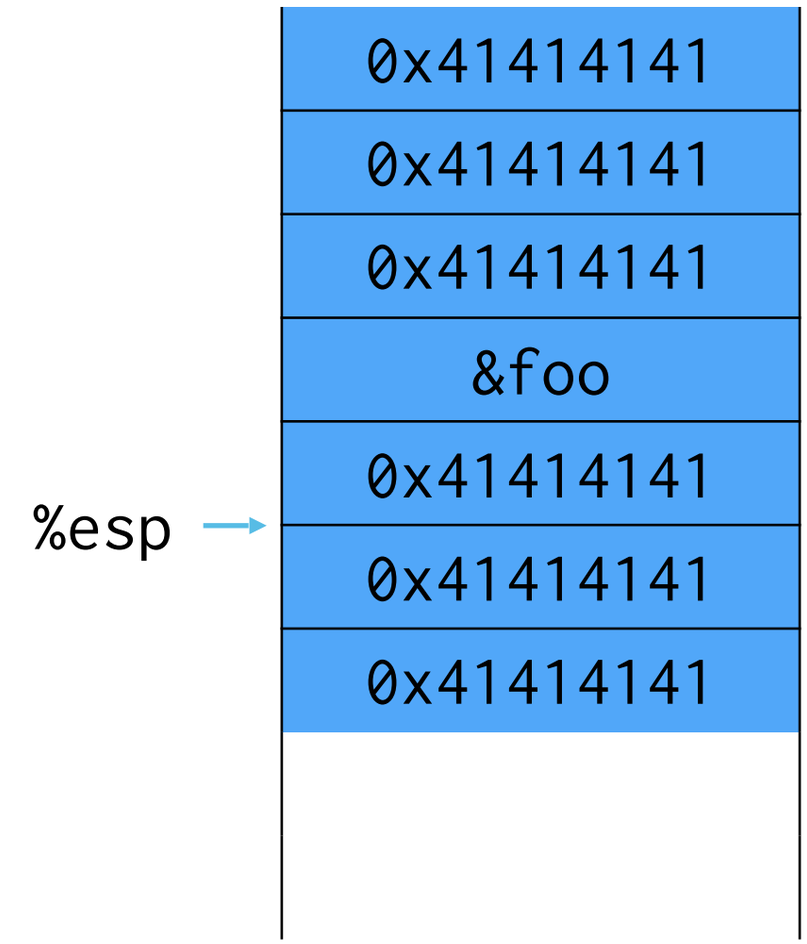
Existing functions

```
#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;
}
```

Existing functions

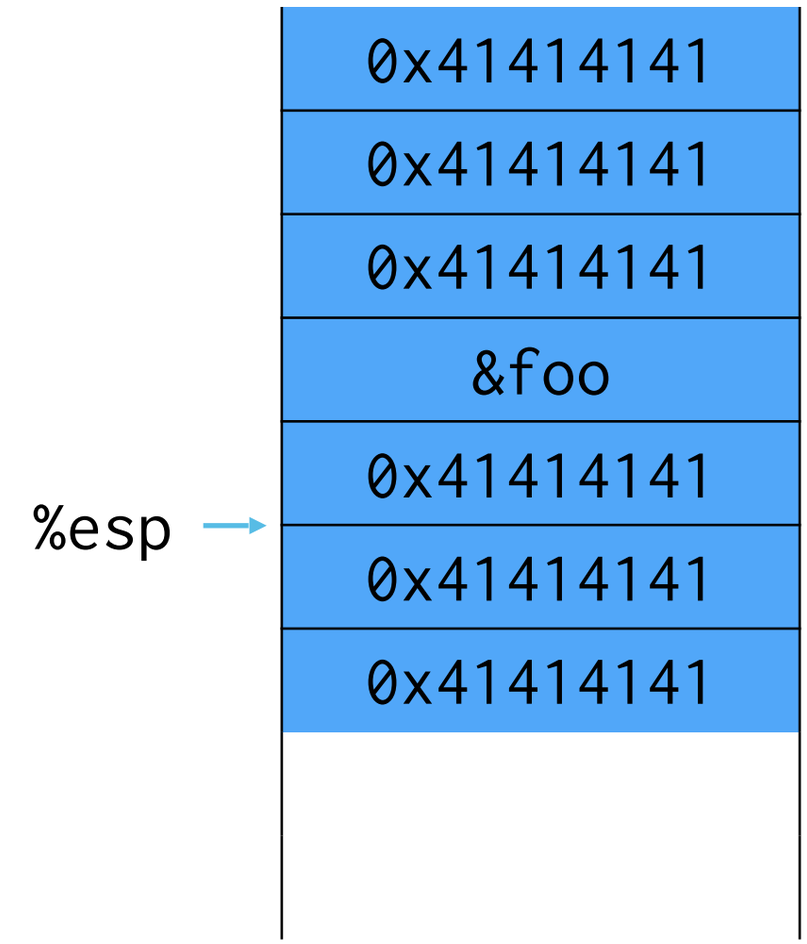
```
#include <stdio.h>
#include <string.h>

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

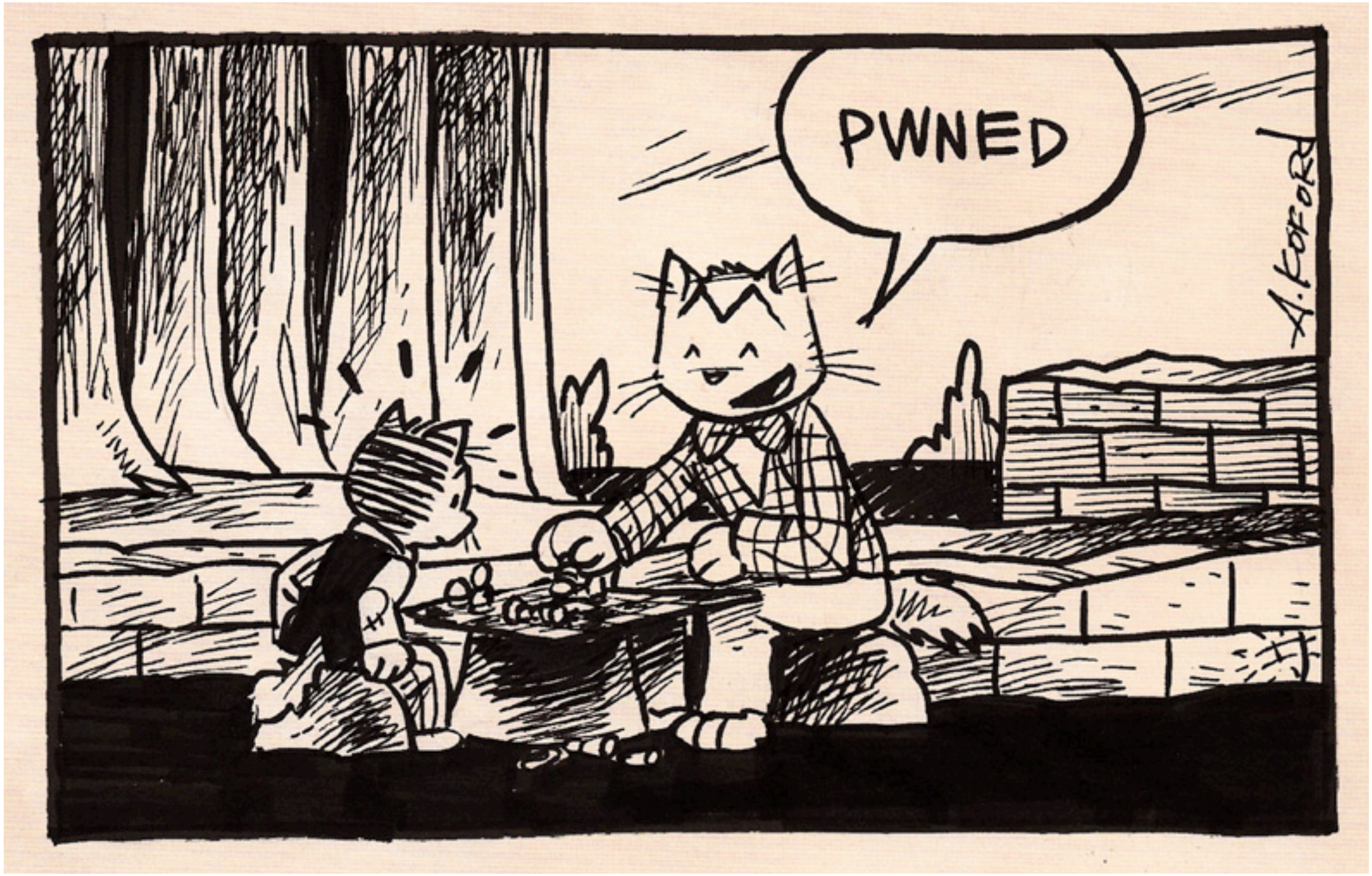
```
#include <stdio.h>
#include <string.h>

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 = &foo

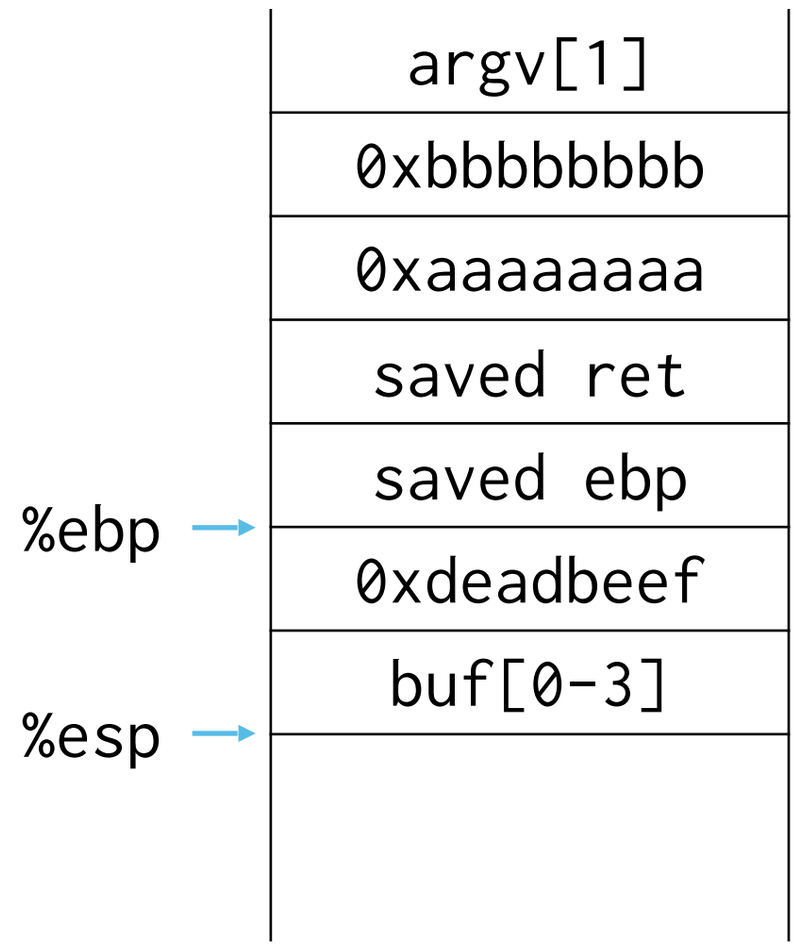
Existing functions




```
%eip = &foo
```

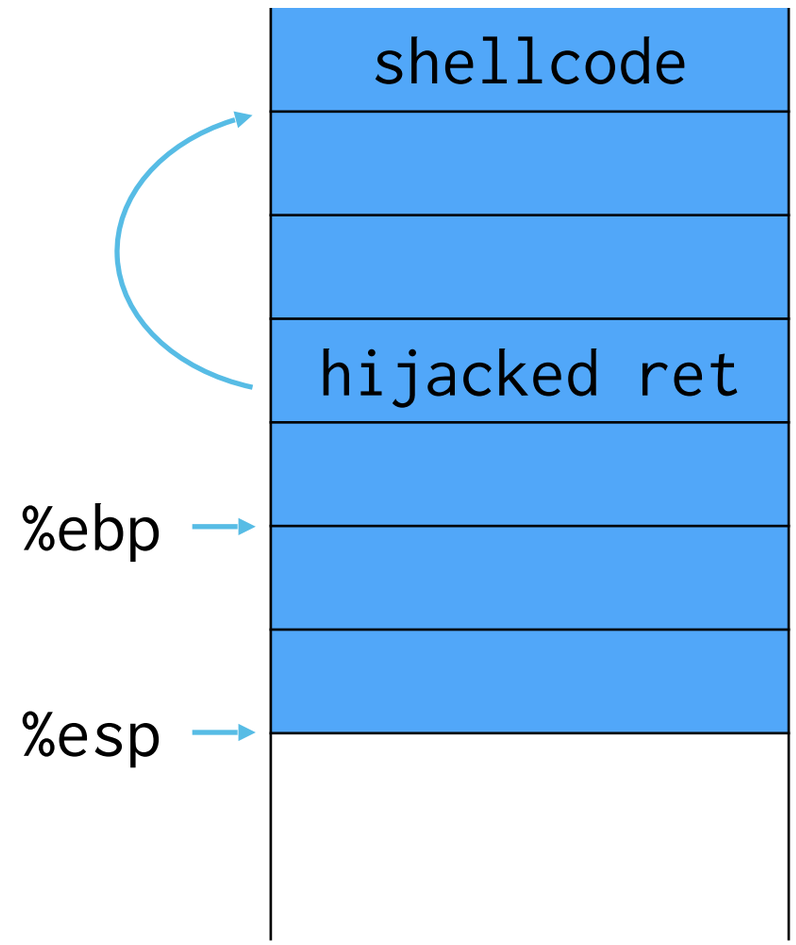
What if the function is not there?

- Jump to attacker-supplied code
- Where?
  - Put code in string
  - Jump to start of string



What if the function is not there?

- 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

```c
int main(void) {
  char* name[1];
  name[0] = “/bin/sh“;
  name[1] = NULL;
  execve(name[0], name, NULL);
  return 0;
}
```

Shellcode

```c
int main(void) {
  char* name[1];
  name[0] = “/bin/sh“;
  name[1] = NULL;
  execve(name[0], name, NULL);
  return 0;
}
```

Can we just take output from gcc/clang?

Shellcode

There are some restrictions

1. Shellcode cannot contain null characters '`\0`'
   - Why?
2. If payload is via `gets()` must also avoid line-breaks
   - Why?

Shellcode

There are some restrictions

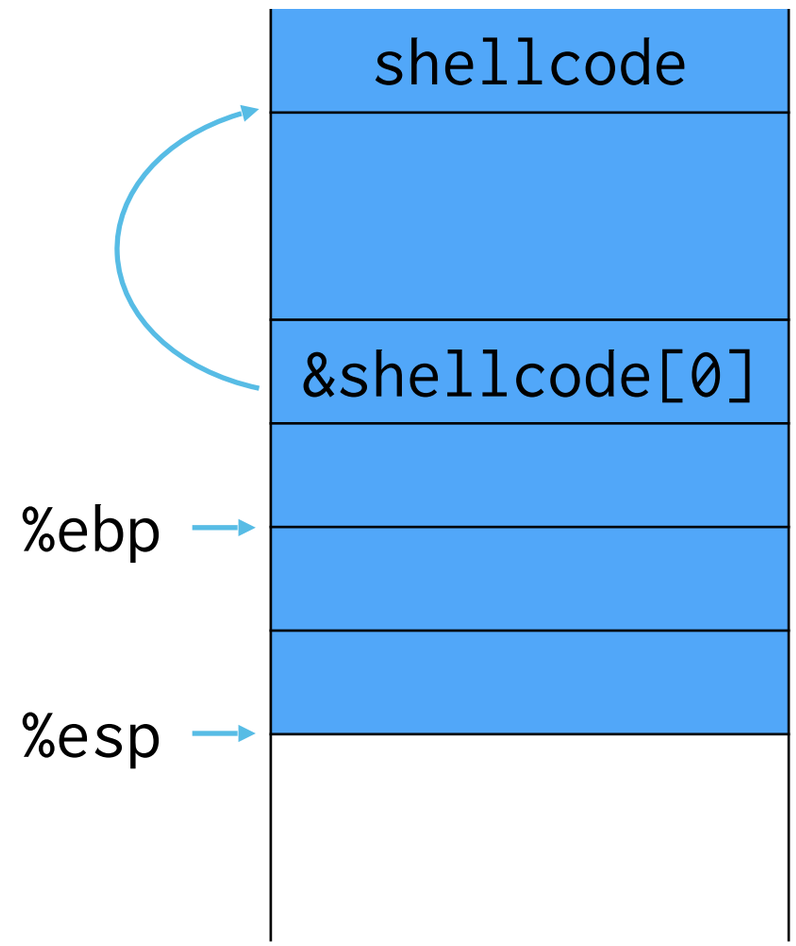
1. Shellcode cannot contain null characters '`\0`'
   - Why?

2. If payload is via `gets()` must also avoid line-breaks
   - Why?

- **Fix:** use different instructions and NOPs!

Payload not always robust

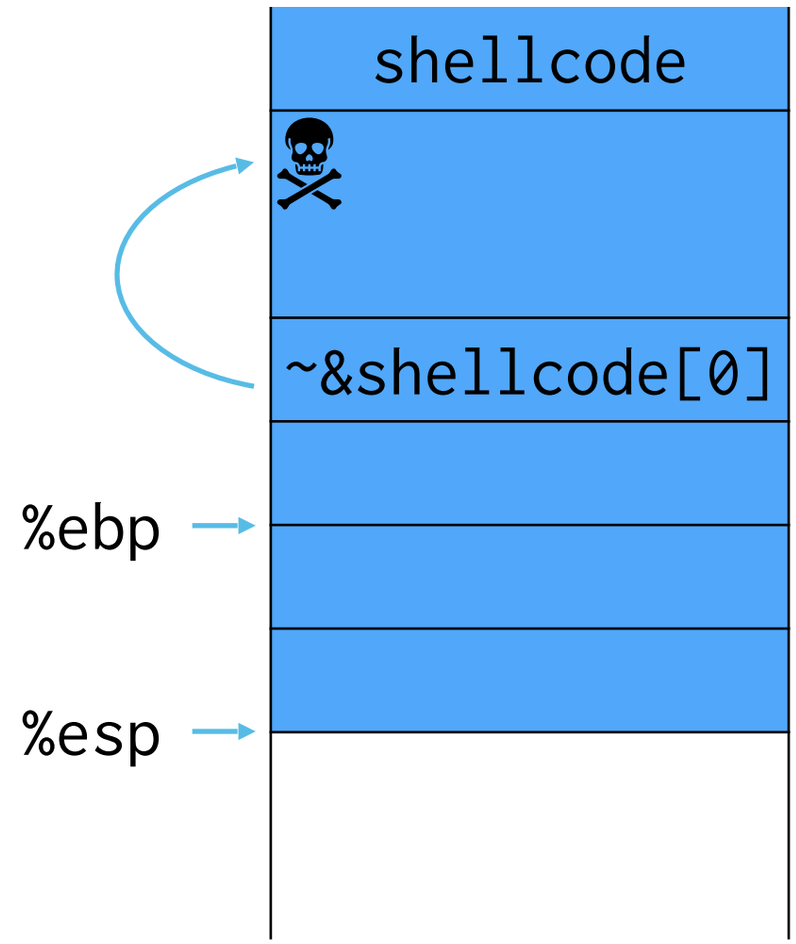
3. Exact address of shellcode start not always easy to guess
   - Miss?



Payload not always robust

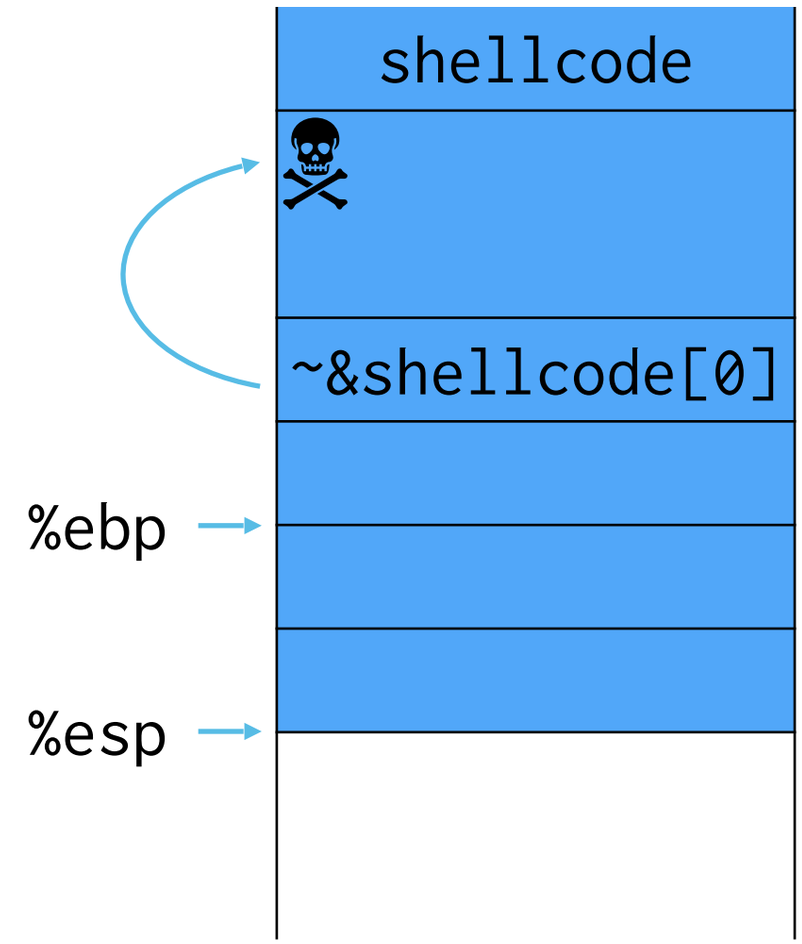
3. Exact address of shellcode start not always easy to guess

- Miss?



Payload not always robust

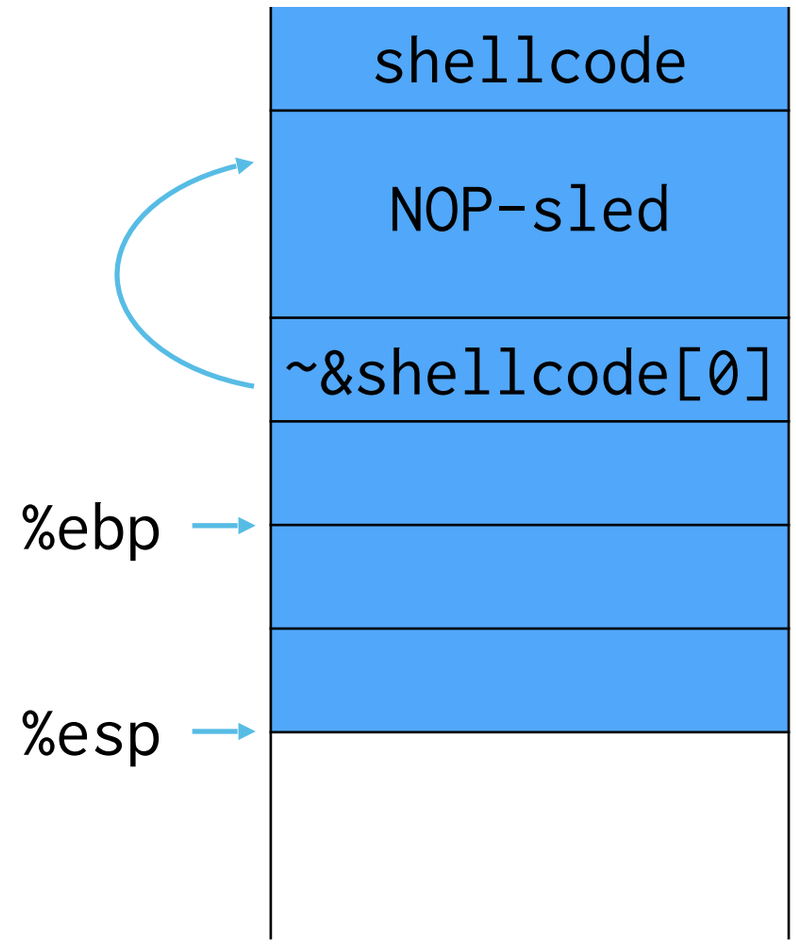
3. Exact address of shellcode start not always easy to guess
   - Miss? SEGFAULT!



Payload not always robust

3. Exact address of shellcode start not always easy to guess

- Miss? SEGFAULT!
- Fix? NOP sled!



shellcode compilers make this easy