# CS202 (003): Stack Smashing

Instructor: Jocelyn Chen

Most of the slides are taken from CSE127 course material by Deian Stefan
(https://cseweb.ucsd.edu/~dstefan/cse127-winter20/slides/2-bufferoverflows.pdf)

# Last Time

# Non-traditional Unix Semantics

## Server failure

**Previously:** open(“some_file”, RD_ONLY) failed if “some_file” does not exist
**Now:** app might hang while trying to access the file

## Deletion or permission change of open files

***What if Client A deletes a file that Client B has “open”?***

**Previously:** Client B reads still work (file exists until all clients close() it)
**Now:** Client B reads fail

***What if Client A make the file inaccessible to others while Client B has the file open()?***

**Previously:** Nothing happens
**Now:** Client B reads fail

## ……

# Security

NFS's only security measure is IP address verification (which is quite weak)

**Previously:** Unix enforces read/write protections — cannot read my files w/o passwords
**Now:** Server believes whatever UID appears in NFS request (and anyone can put whatever in the request)

Not extremely vulnerable because of how FH works

```
Example structure (simplified):
struct file_handle {
    uint32_t filesystem_id;     // Random unique identifier
    uint32_t inode_number;      // File system location
    uint32_t generation_number; // Changes when inode is reused
    uint8_t  extra_data[20];    // Additional metadata
}
```

It does not solve all types of attack though!

Vulnerabilities are technically fixable (strong auth, secure protocols, …),
but hard to reconcile with the stateless design

# 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?

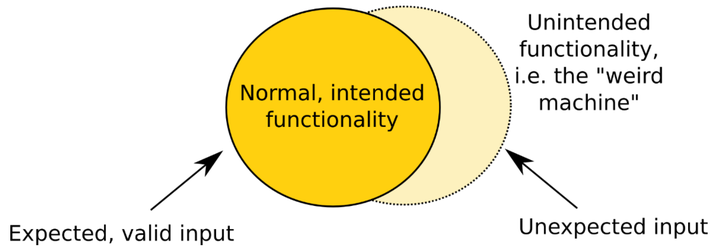
- Try 2: When it doesn't do bad things
- Easier to specify a list of "bad" things:
  - Delete or corrupt important files
  - Crash my system
  - Send my password over the Internet
  - Send threatening email to the professor

# 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




https://en.wikipedia.org/wiki/Weird_machine#/media/File:Weird_machine.png

# 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?

- 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

# 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.

# 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!

```c
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);
    //...
    return(0);
}
```

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?
  - 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 }
```

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

# Example 1: spot the vuln!

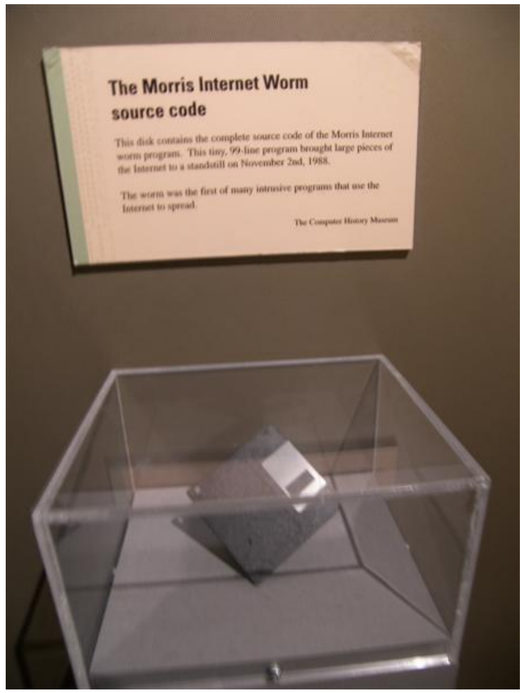
- What does `gets()` do?
  - How many characters does it read in?
  - Who decides how much input to provide?
- How large is `line[]`?
  - Implicit assumption about input length
- What happens if, say 536, characters are provided as input?

```
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




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

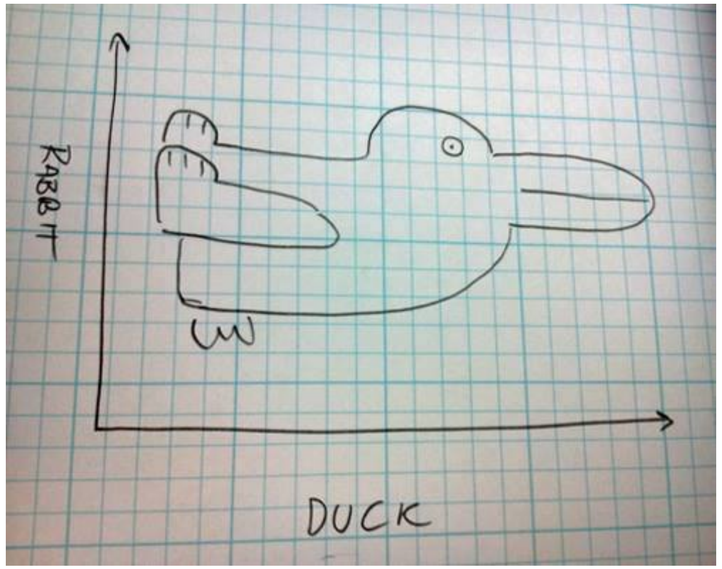
# OK but...

- 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

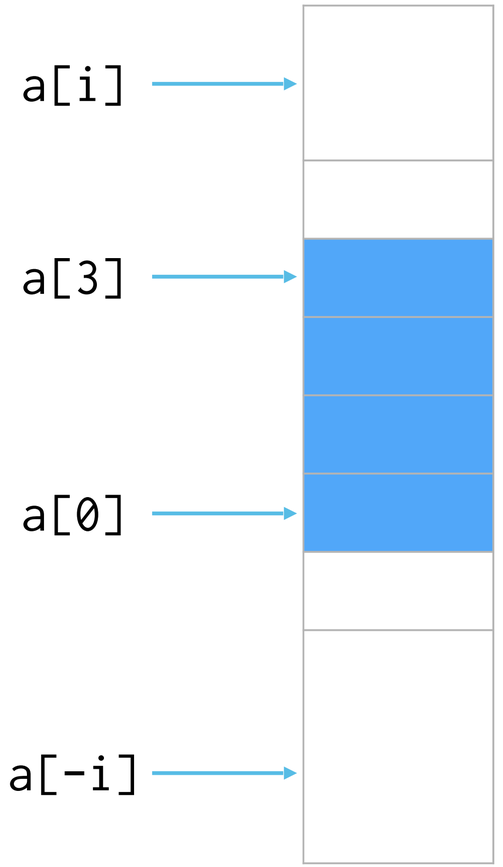


# What we need to know

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

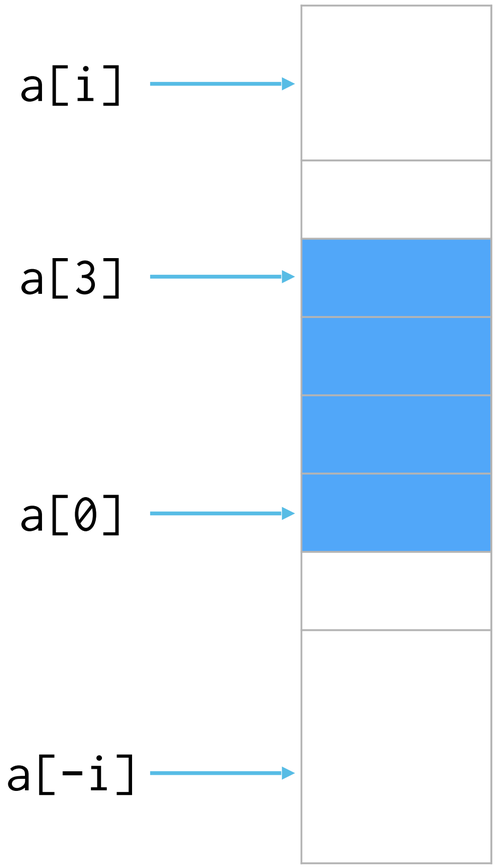
# How does an array work?

- What's the abstraction?



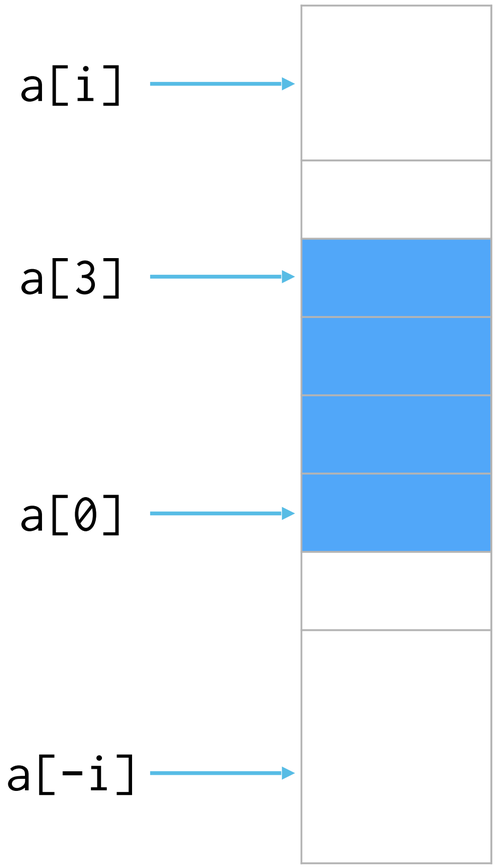
# How does an array work?

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



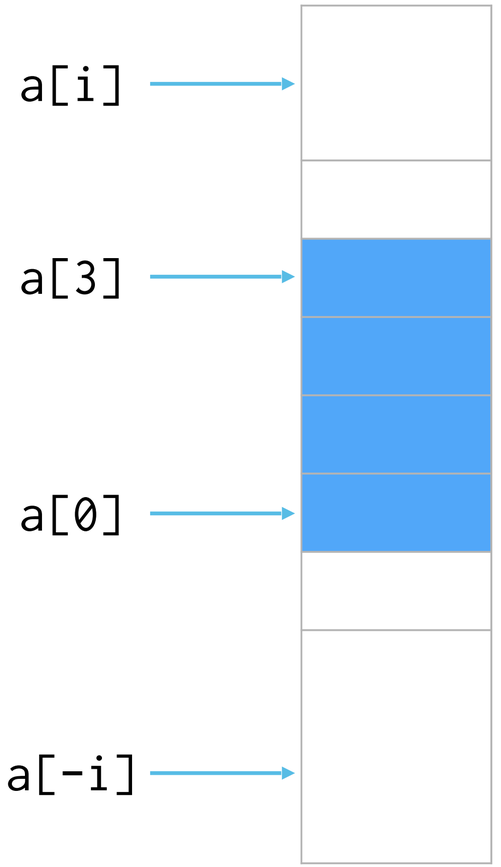
# How does an array work?

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



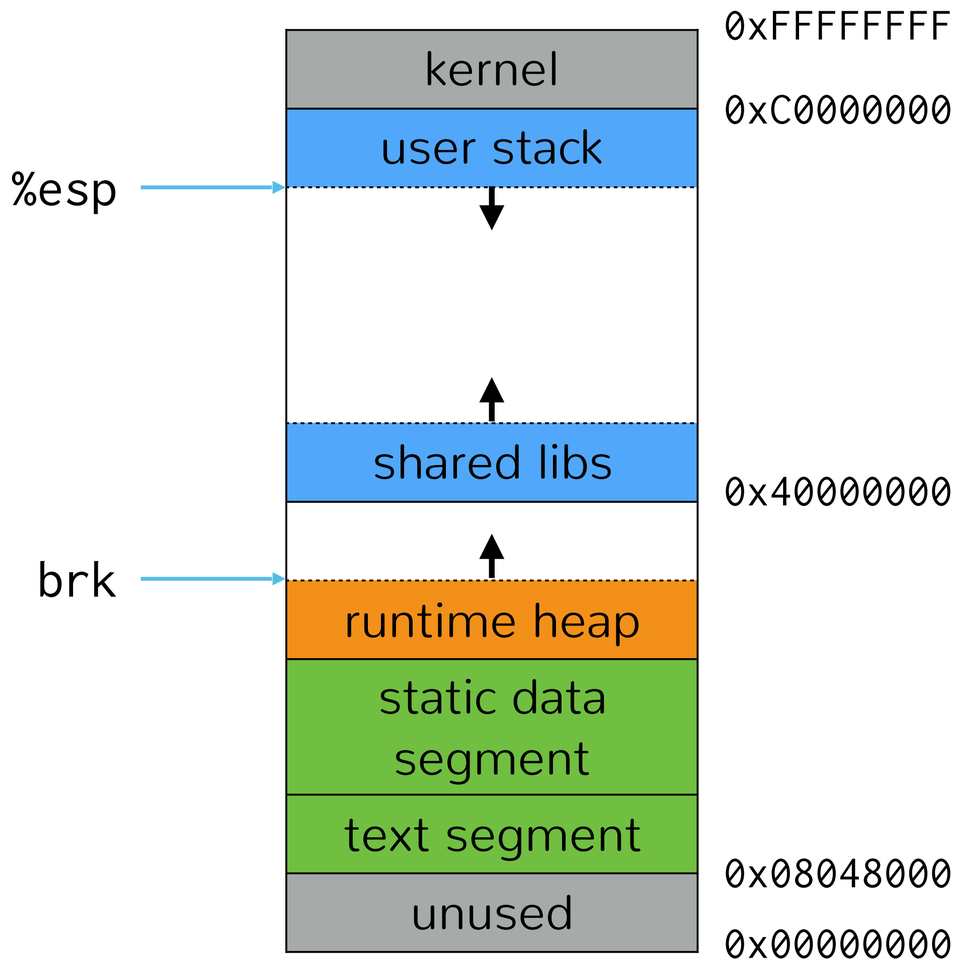
# How does an array work?

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



# Linux process memory layout

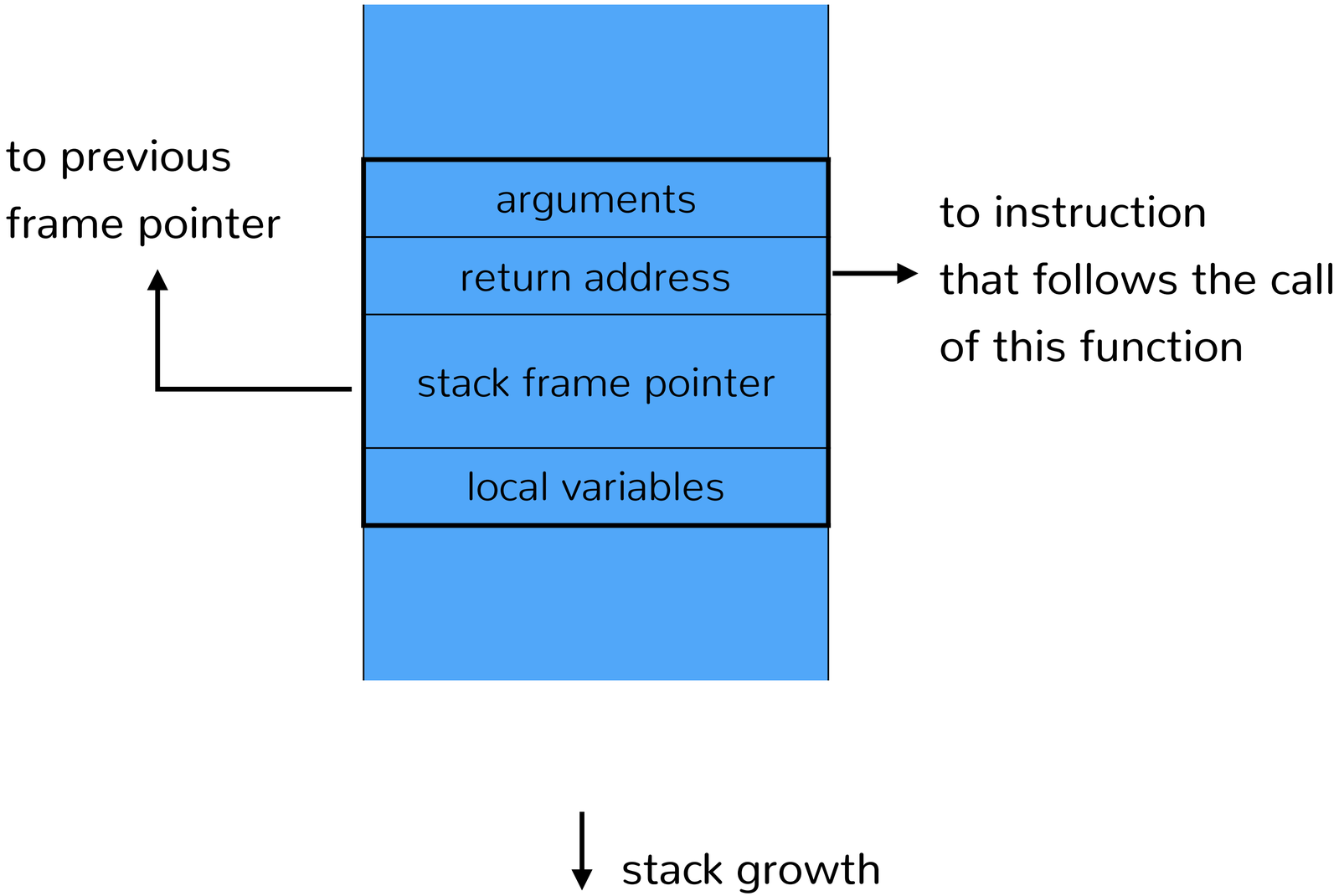
- Stack
- Heap
- Data segment
  - .data, .bss
- Text sement
  - Executable code



# 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


to previous
frame pointer
arguments
return address
stack frame pointer
local variables
to instruction
that follows the call
of this function
stack growth

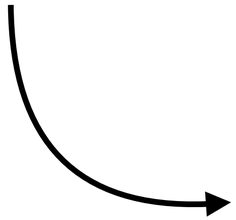
# Example 0

```
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

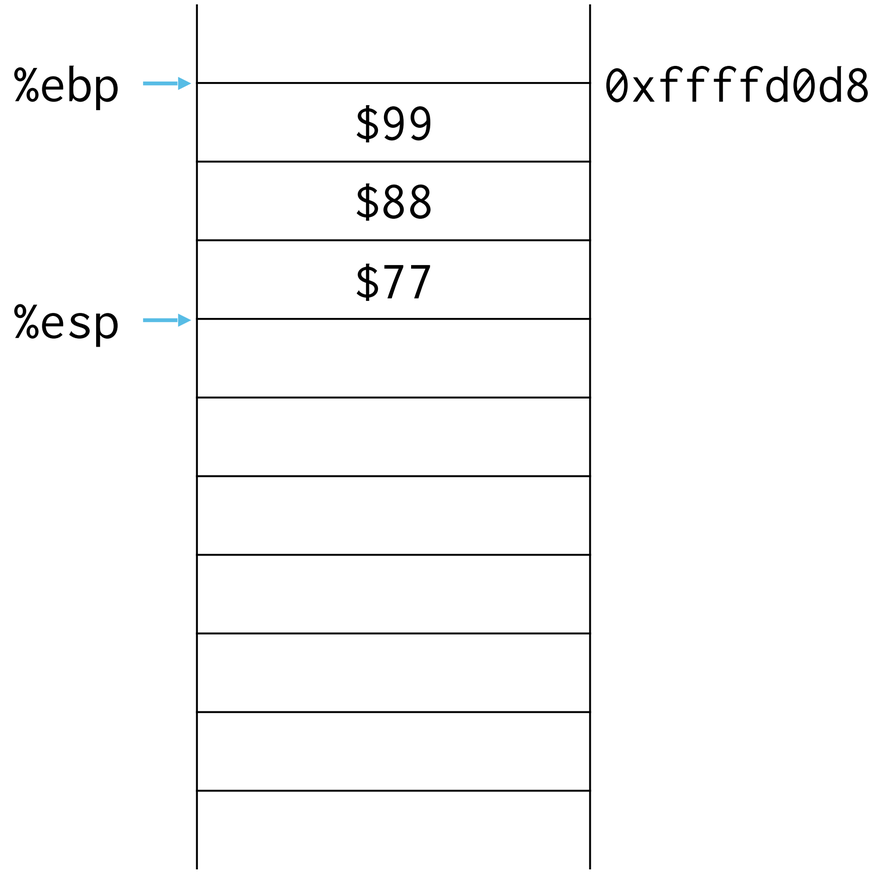
# Compiled to x86

```
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 }
```

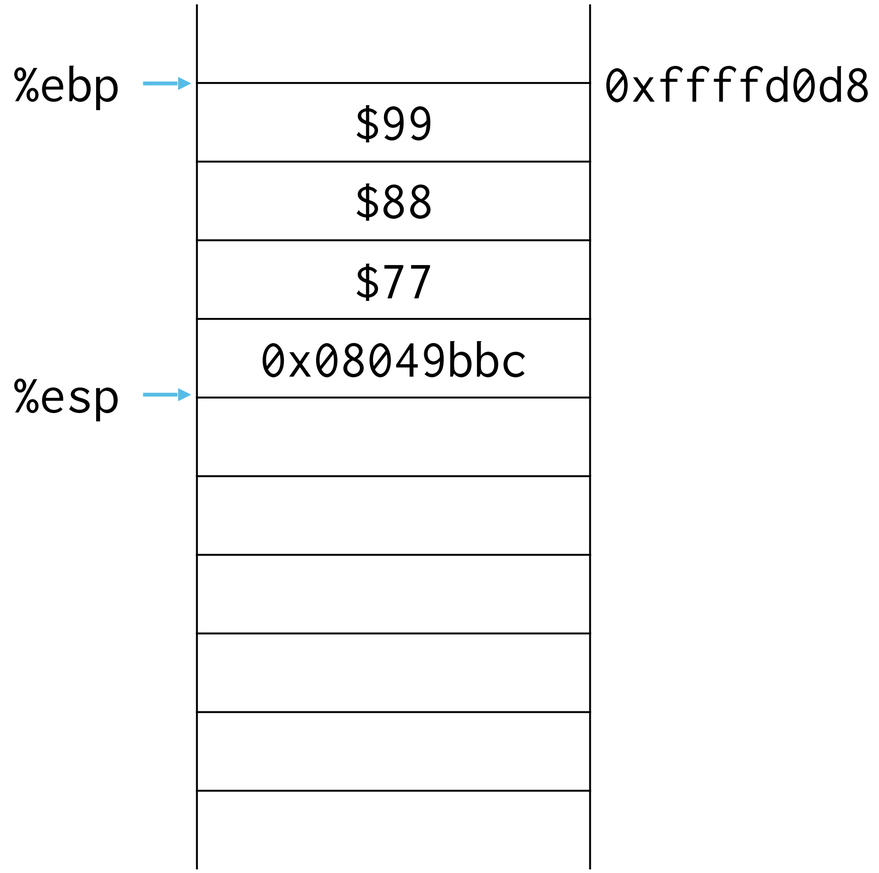


```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)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

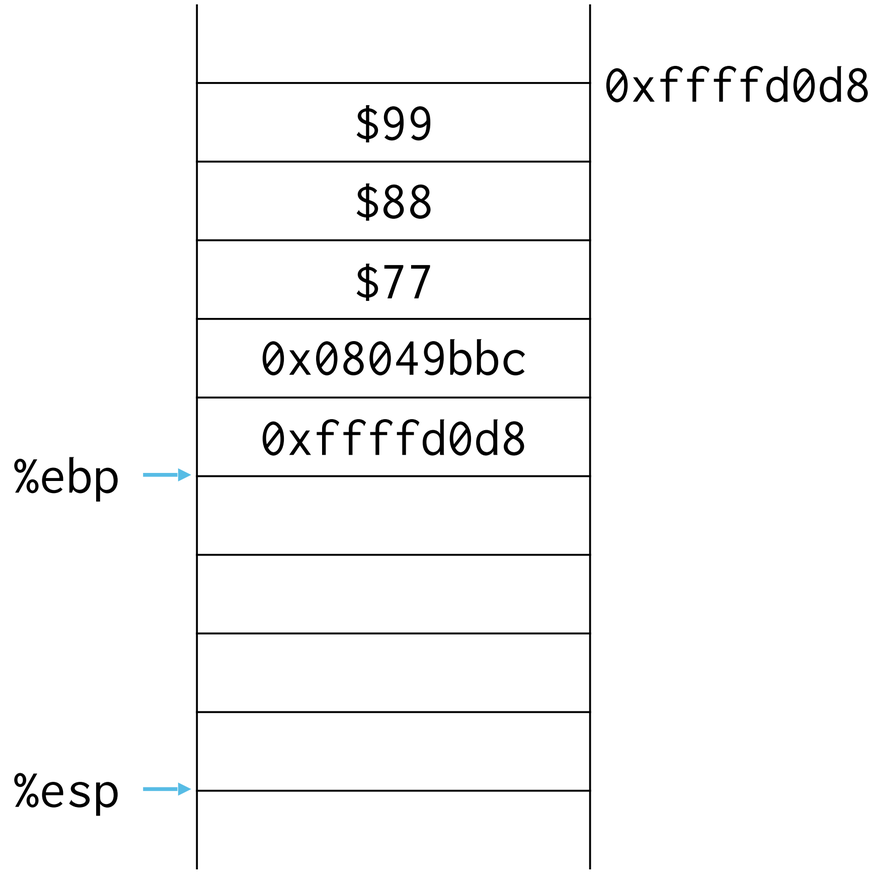


```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax,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

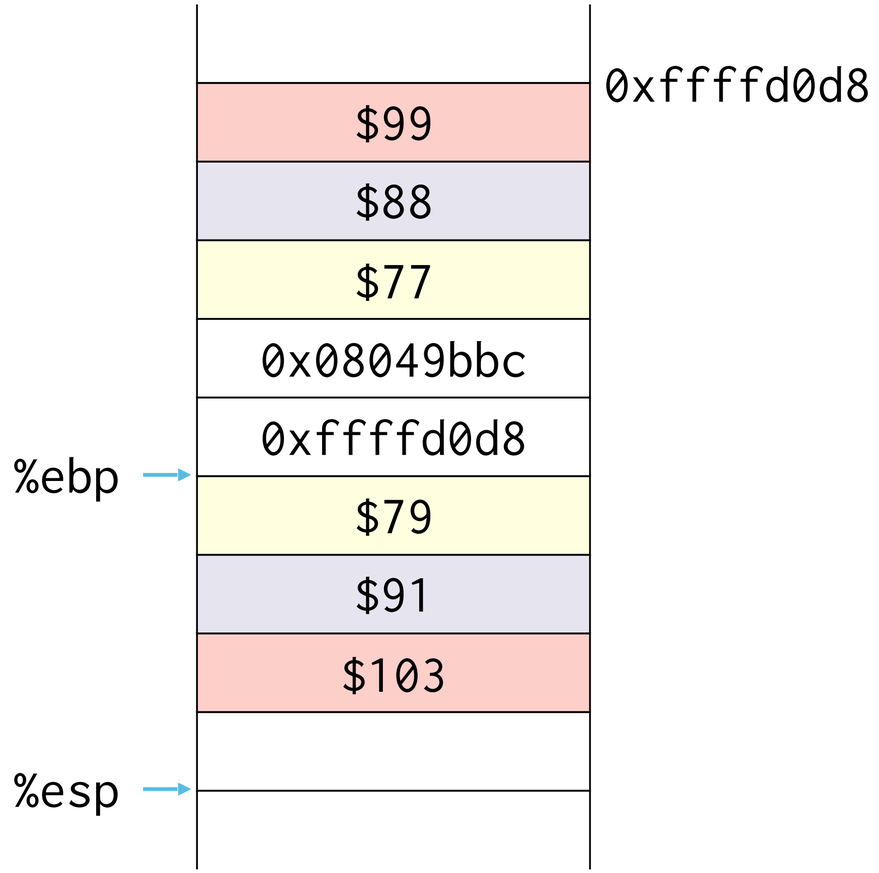



%eip = 0x08049ba7

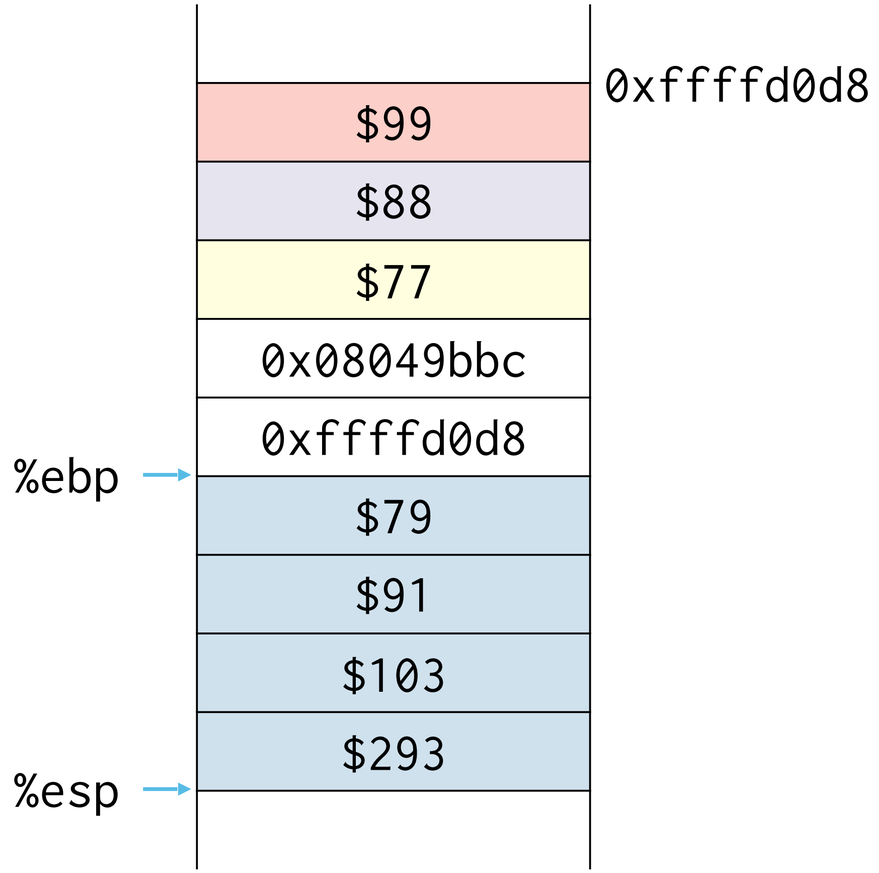
```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)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



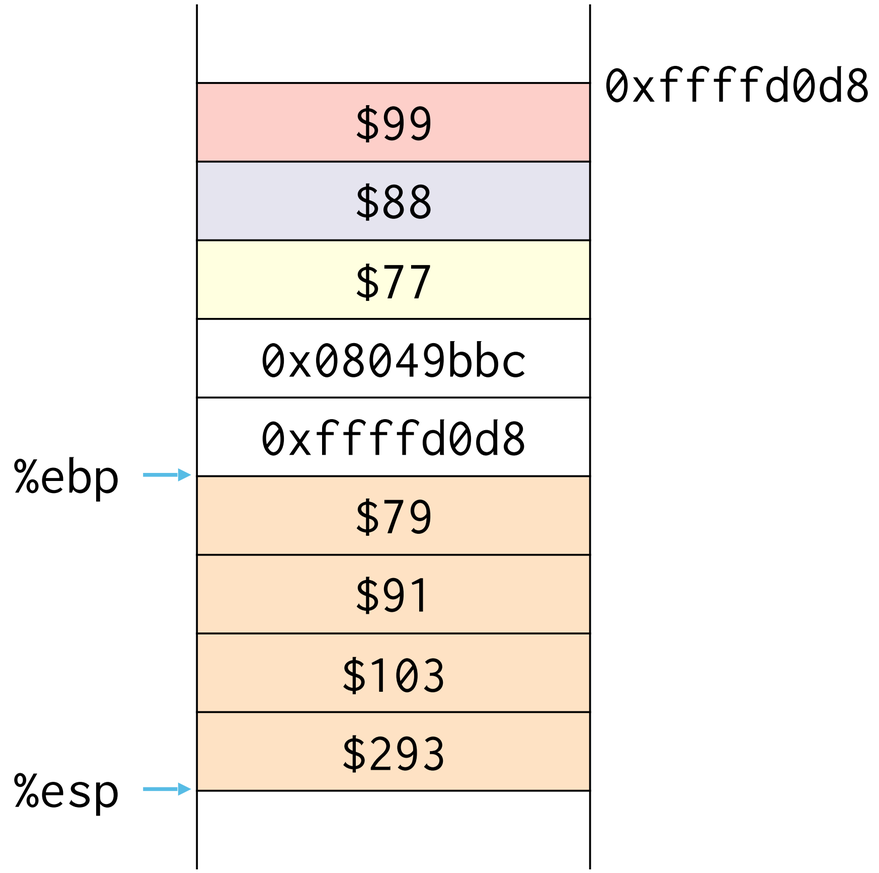
```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)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



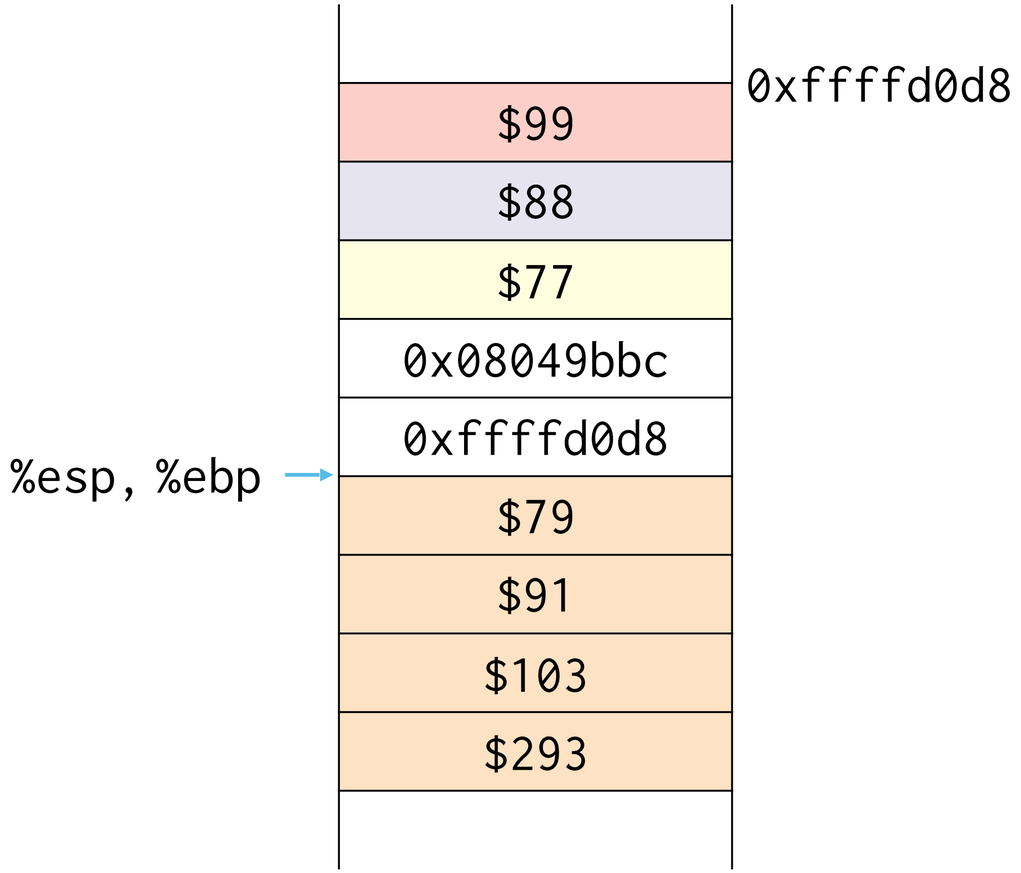
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax,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



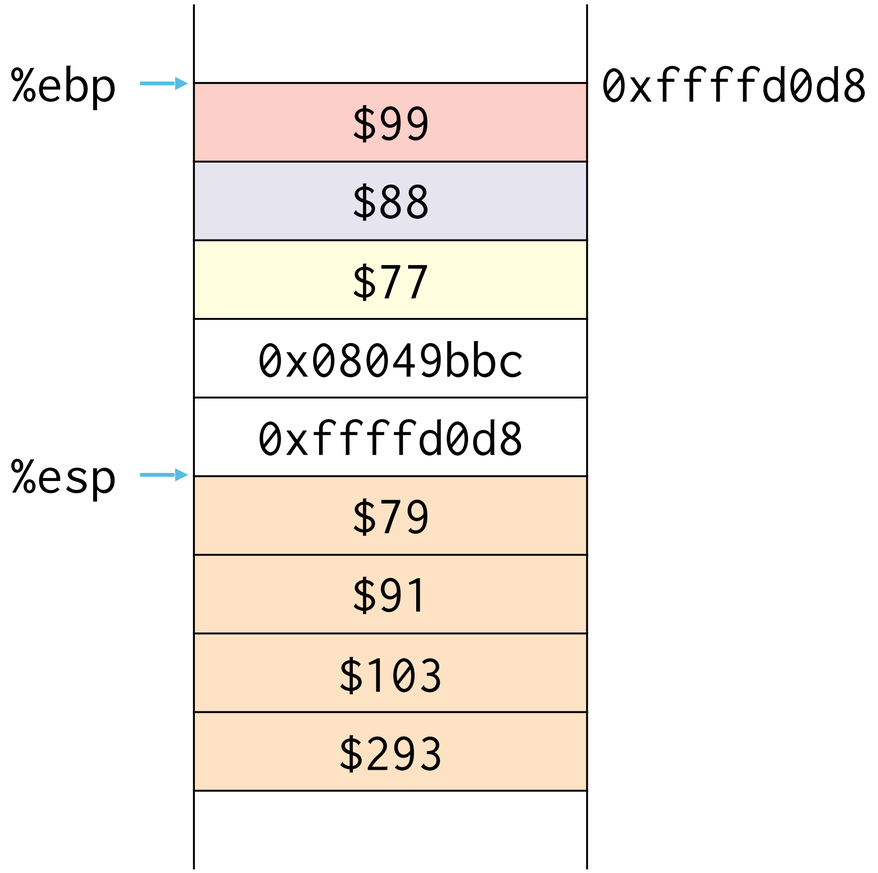
```
1  foobar(int, int, int):
2          pushl   %ebp
3          movl    %esp, %ebp
4          subl    $16, %esp
5          movl    8(%ebp), %eax
6          addl    $2, %eax
7          movl    %eax, -4(%ebp)
8          movl    12(%ebp), %eax
9          addl    $3, %eax
10         movl    %eax, -8(%ebp)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



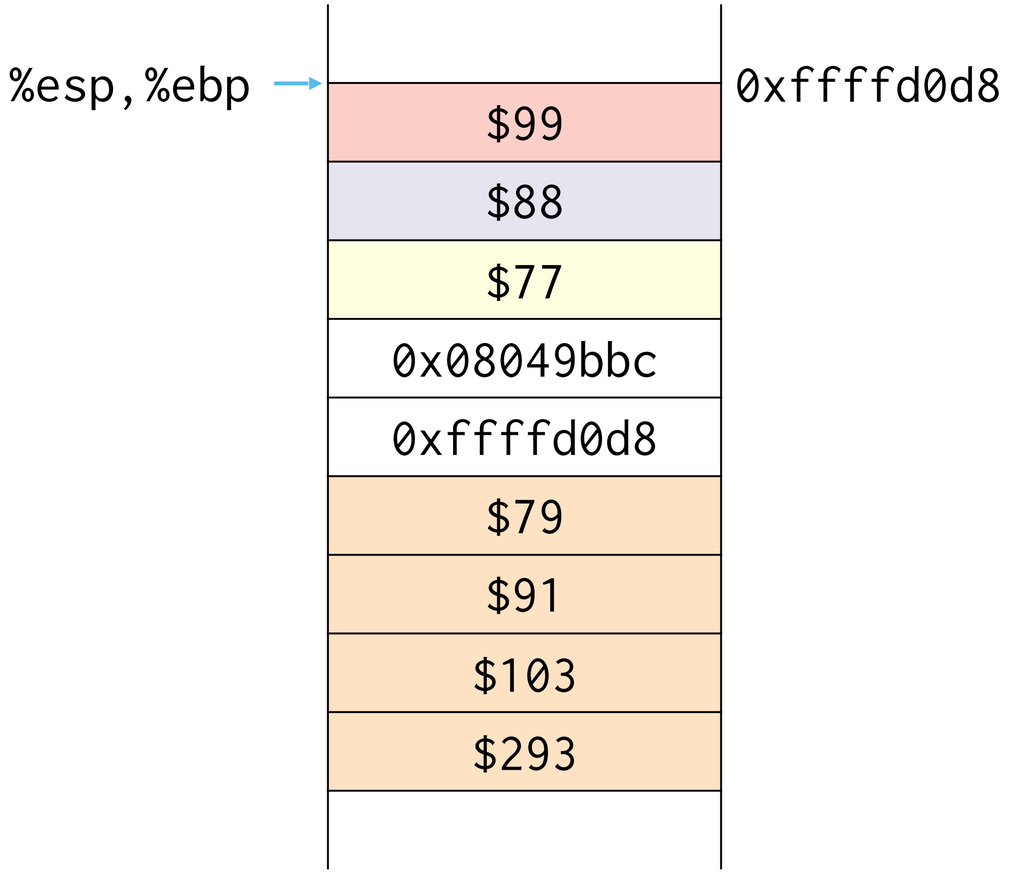
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax,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



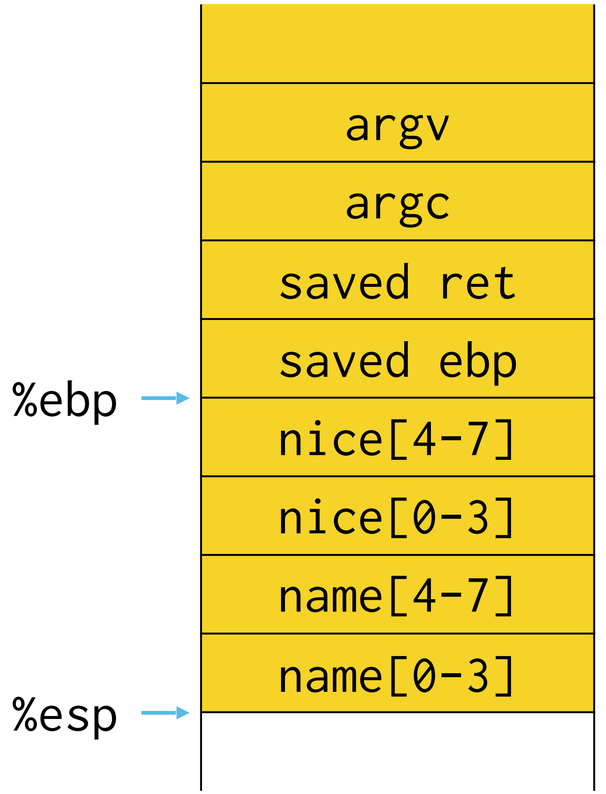
```
1   foobar(int, int, int):
2           pushl   %ebp
3           movl    %esp, %ebp
4           subl    $16, %esp
5           movl    8(%ebp), %eax
6           addl    $2, %eax
7           movl    %eax, -4(%ebp)
8           movl    12(%ebp), %eax
9           addl    $3, %eax
10          movl    %eax,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




%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;
}
```



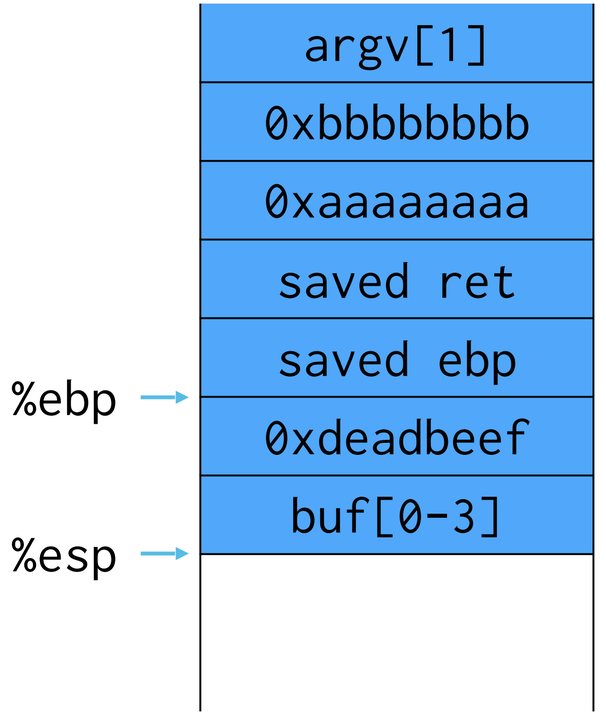

If not null terminated can read more of the stack

# 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;
}
```

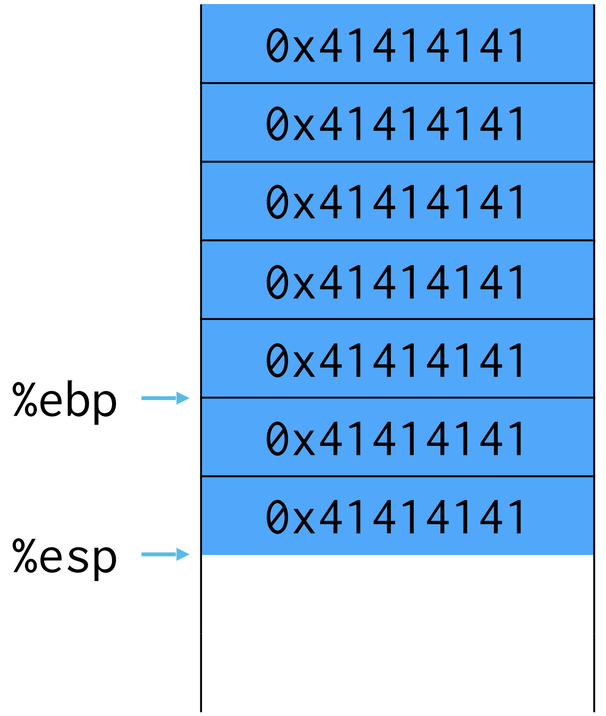
# Example 2

```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

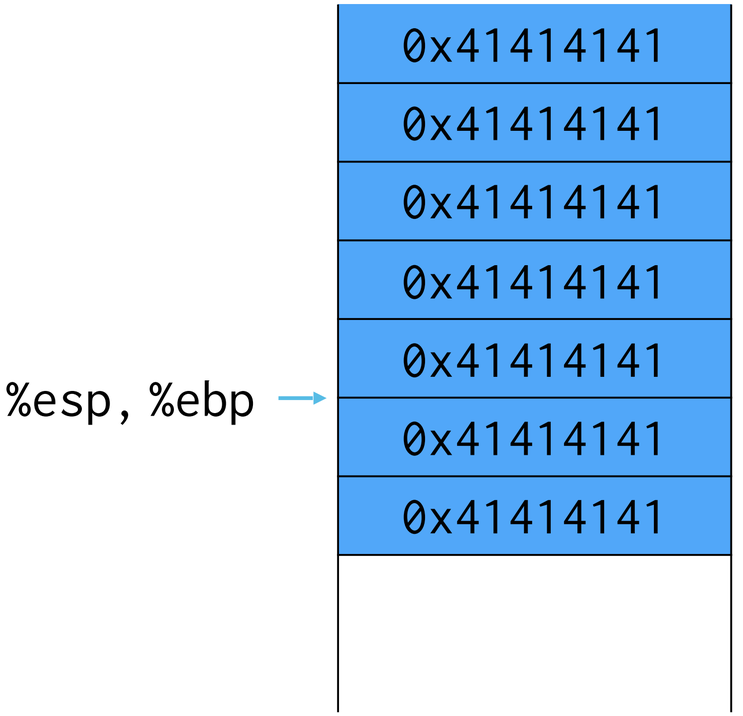
```c
#include <stdio.h>
#include <string.h>

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;
}
```




If first argument to program is “AAAAAAAA…”

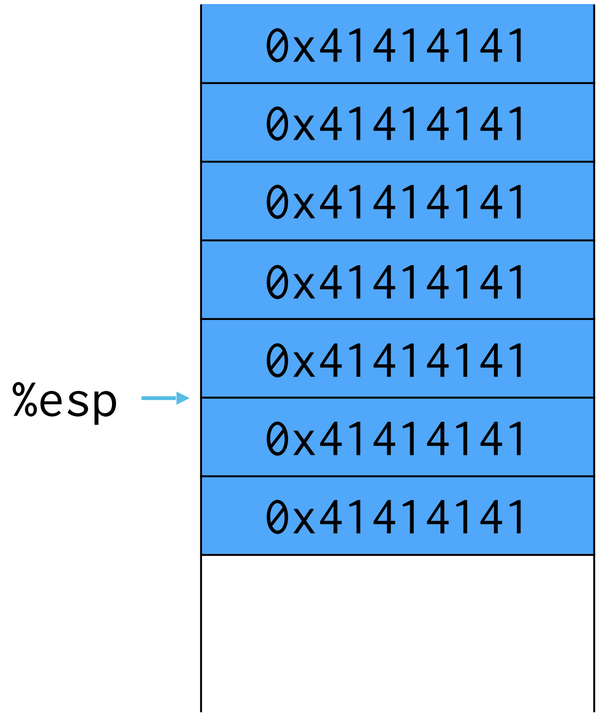
# Example 2

```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

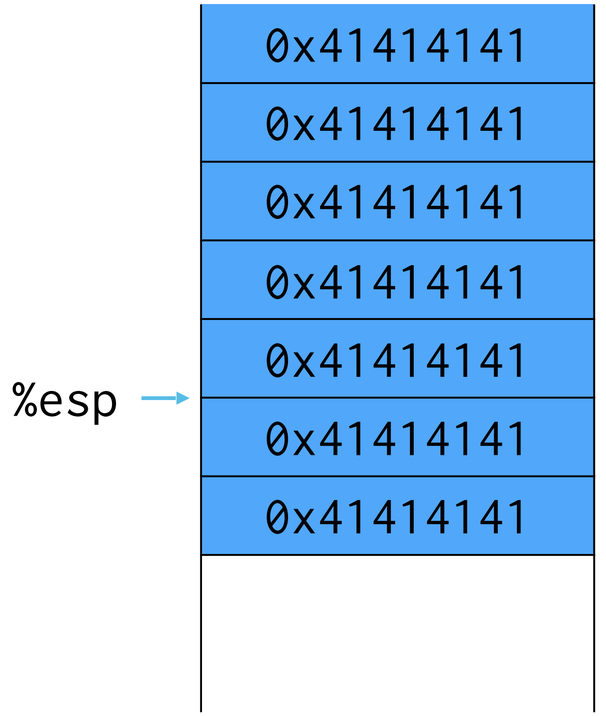
```c
#include <stdio.h>
#include <string.h>

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;
}
```




%ebp = 0x41414141

# 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;
}
```




%ebp = 0x41414141

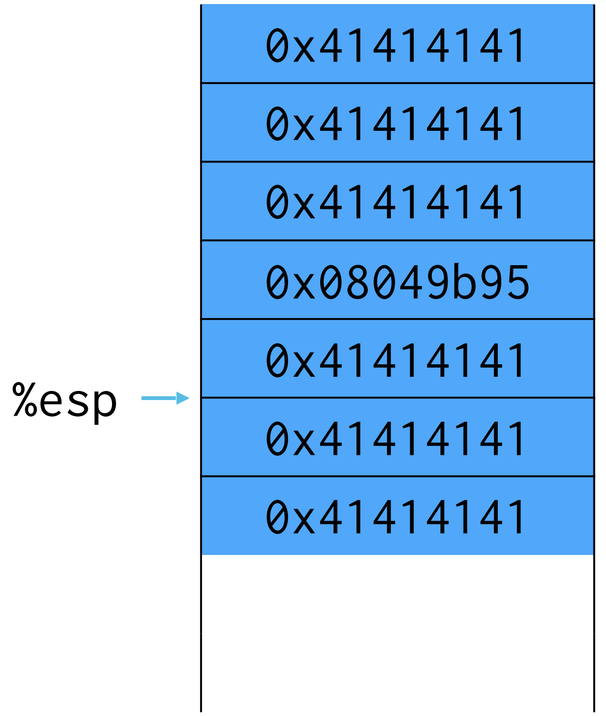
%eip = 0x41414141

# 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

```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;
}
```



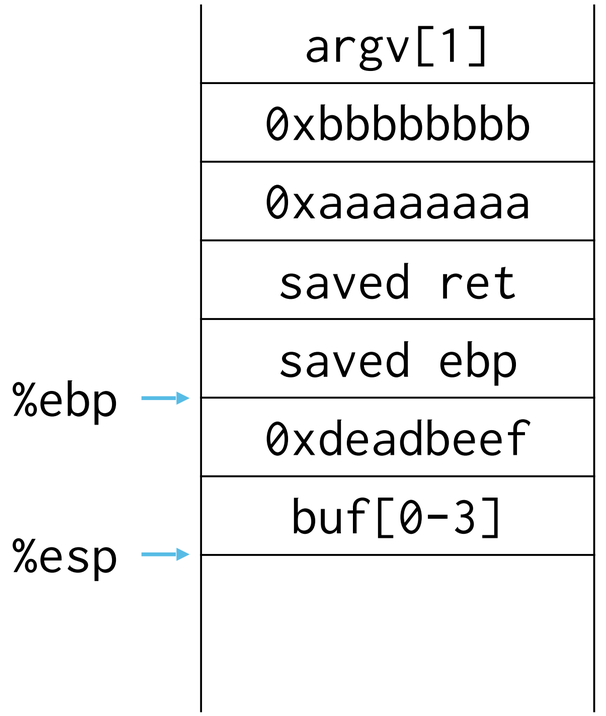

%ebp = 0x41414141

%eip = 0x08049b95

Let’s look at this in GDB (w/ GEF)

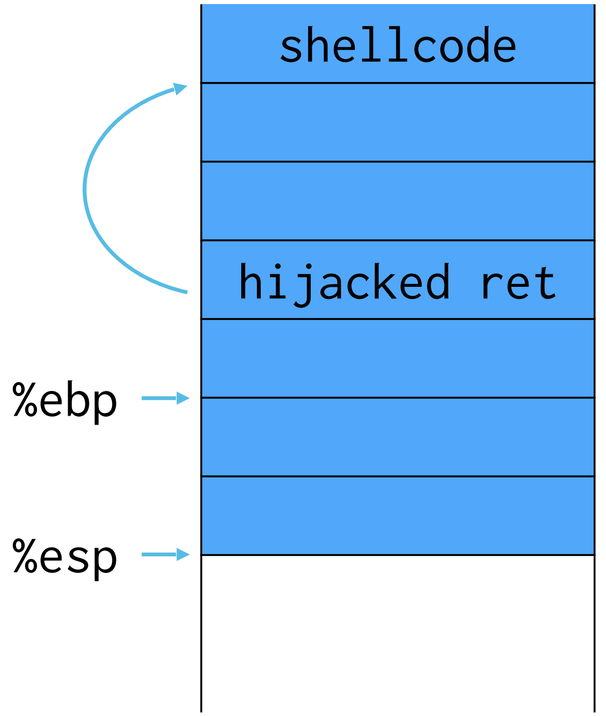
# Better Hijacking Control

- 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

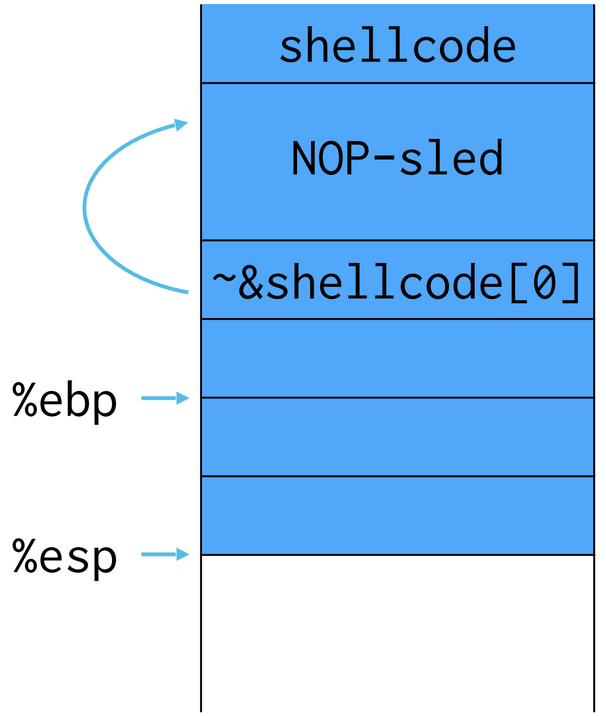
```
void main() {
   char *name[2];

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

# Shellcode

- 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



# 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

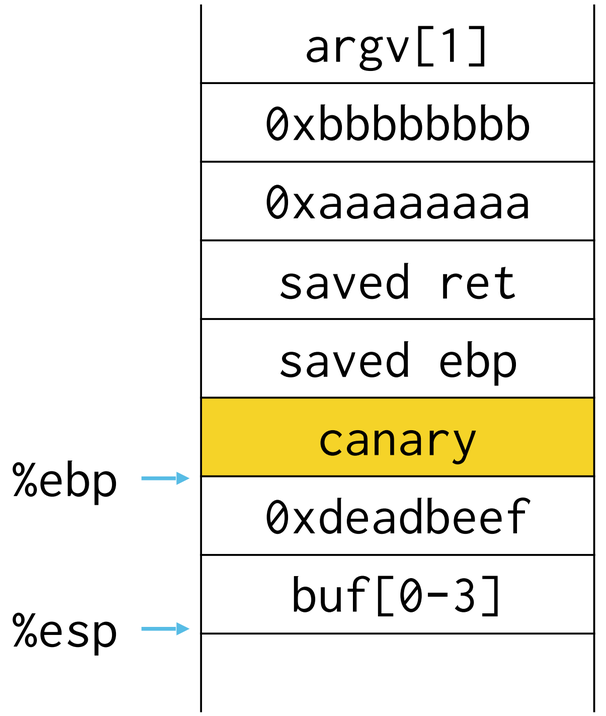
- strcpy, strcat, gets, etc.
- **Plus:** Good idea in general
- **Minus:** Requires manual code rewrite
- **Minus:** Non-library functions may be vulnerable
  - E.g. user creates their own strcpy
- **Minus:** No guarantee you found everything
- **Minus:** alternatives are also error-prone

# 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

```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;
}
```

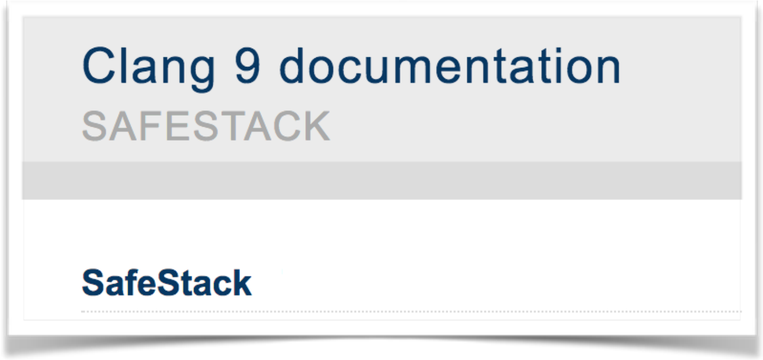



Check canary on ret

# 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

# 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

# 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

# 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