PWNING 101 - p.1

spritzers - CTF team

spritz.math.unipd.it/spritzers.html

Disclaimer

All information presented here has the only purpose to teach how vulnerabilities work.

Use them to win CTFs and to build secure systems.

Do not hack your neighbor’s fancy IoT fridge.

Pwning in CTFs

In pwn challenges you have to exploit a remote service.

You typically want to get a shell and `cat flag`.

Most of the time, it’s a memory corruption vulnerability.

What’s memory corruption?

Modifying a process’ memory in a way the programmer (or compiler) didn’t intend.

If we control the memory, we control the process.

Memory corruption in the wild

- Malware
  - Morris (1988!), CodeRed, Blaster, Sasser, Conficker, ...
  - More recently, StuxNet and WannaCry
- Remote services and user applications
  - Exposed to untrusted data
- Unlocking devices
  - Android roots, iOS jailbreaks, gaming consoles (I started here!)

Memory corruption vulnerabilities

- Buffer overflows
- Format strings
- Use of uninitialized memory
- Dangling pointers (e.g., use-after-free)
- Type confusion
- Heap metadata corruption

… and many more

Memory corruption attacks

Two main subclasses:

- *Non-Control-Data Attacks* manipulate the application's state and data
- *Control-Flow Attacks* manipulate the execution flow

Exploitation

Finding a vulnerability is just the first step.

Uncontrolled memory corruption typically results in a crash (e.g., SIGSEGV).

We need to channel the vulnerability into whatever we want to do.

Exploitation

First, we set things up for the upcoming corruption.

Then, we trigger it and watch the dominoes fall down.

The tool that performs this is an *exploit*.

What’s memory?

Memory is a *flat sequence of bytes*. That’s it.

Each byte is identified by an *address*.

Via *memory protection*, areas of memory can be marked as readable, writable, executable.

	+0	+1	+2	+3
+0	01	cd	4b	3f
+4	96	a1	39	bb
+8	22	33	cd	e0

Interpretations of memory

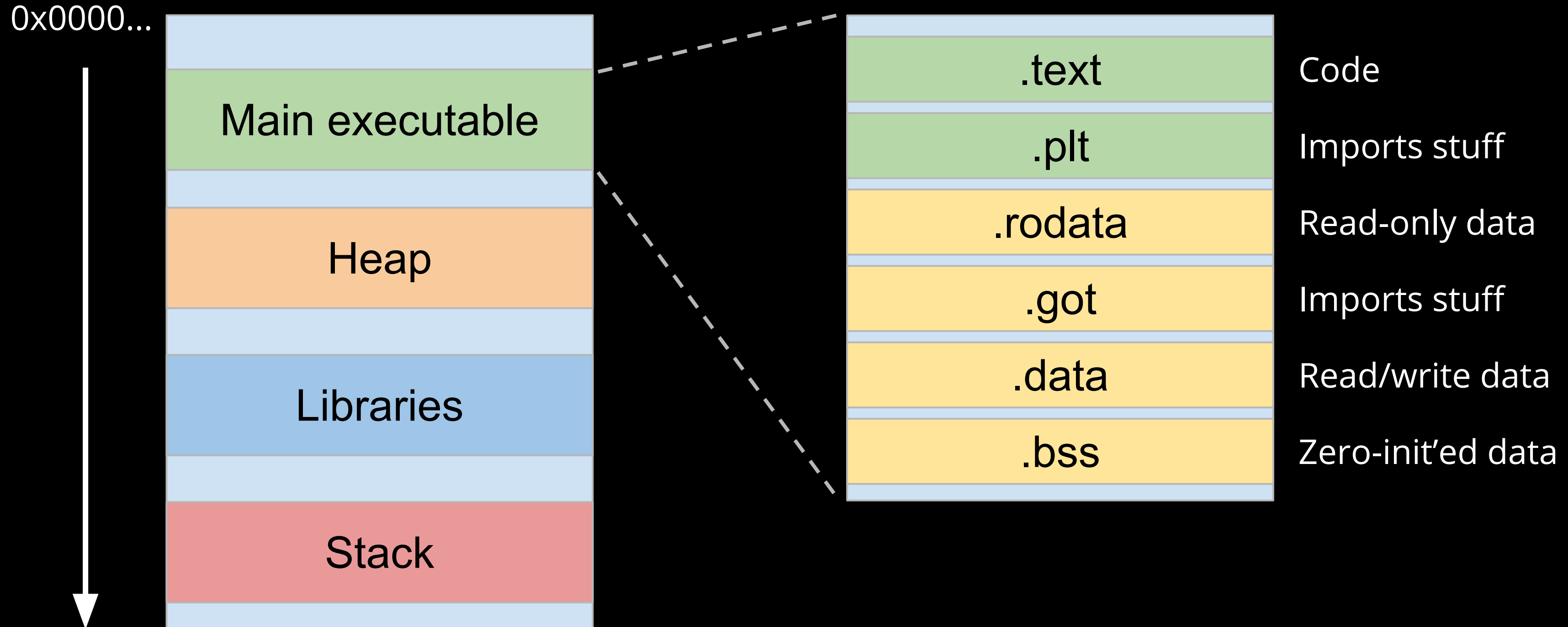
Types do not exist in memory. They are just abstractions that define how a certain range of bytes is interpreted.

Example: integers (and pointers) are little-endian on x86.

```
78 56 34 12 <-> 0x12345678
```

Example: C arrays are a contiguous sequence of elements.

A process' memory



SHIT'S ABOUT TO GET REAL

UP IN THIS CLASS

Buffer overflows

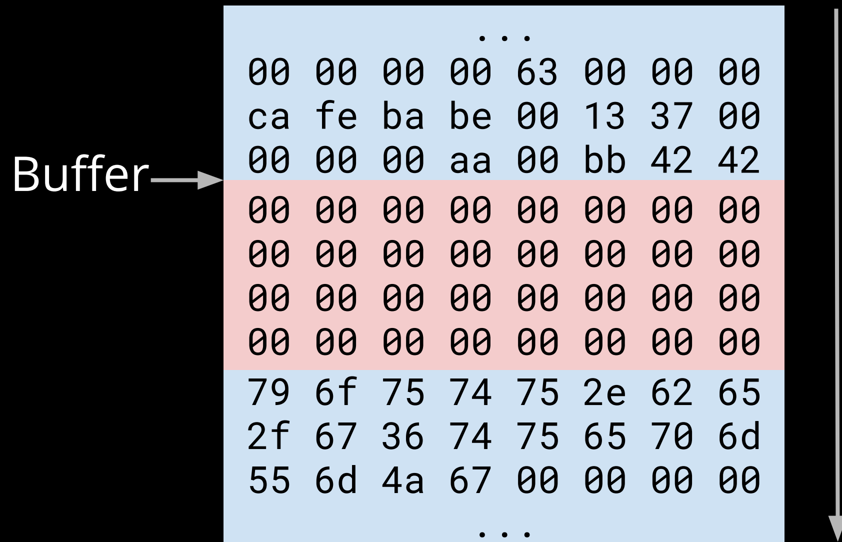
Some languages (such as C/C++) do not check array bounds.

If the programmer doesn’t perform those checks, he might write data beyond the buffer’s boundaries.

This is bad. Like, really bad, man.

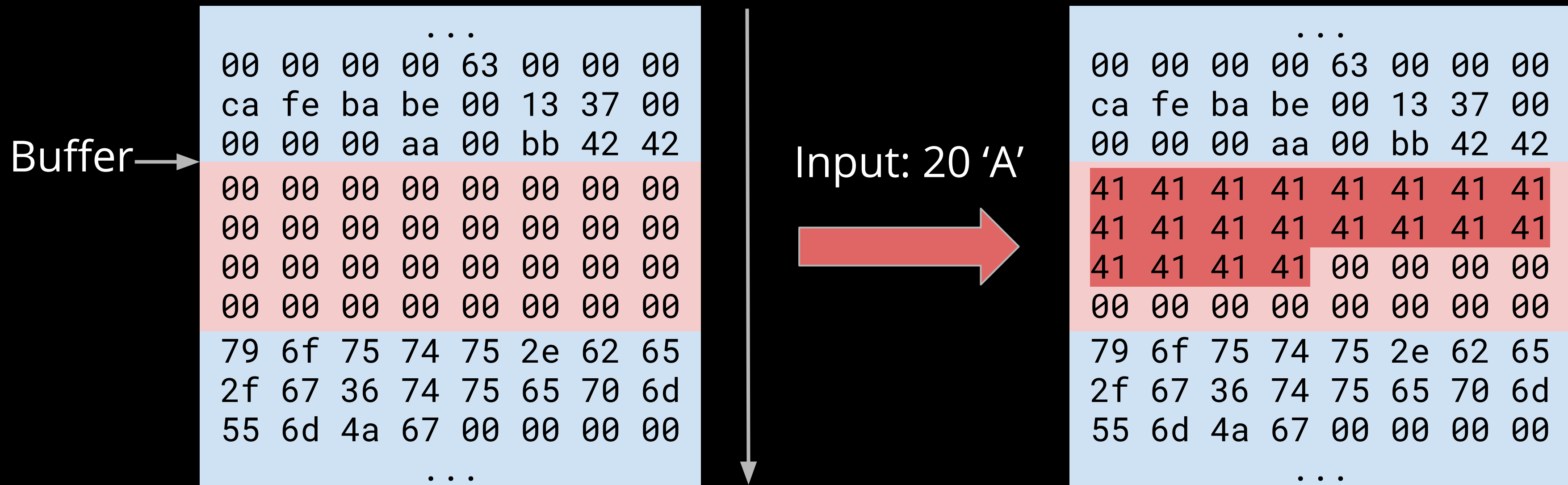
Buffer overflows

This program copies the user's input to a fixed size 32-byte buffer.



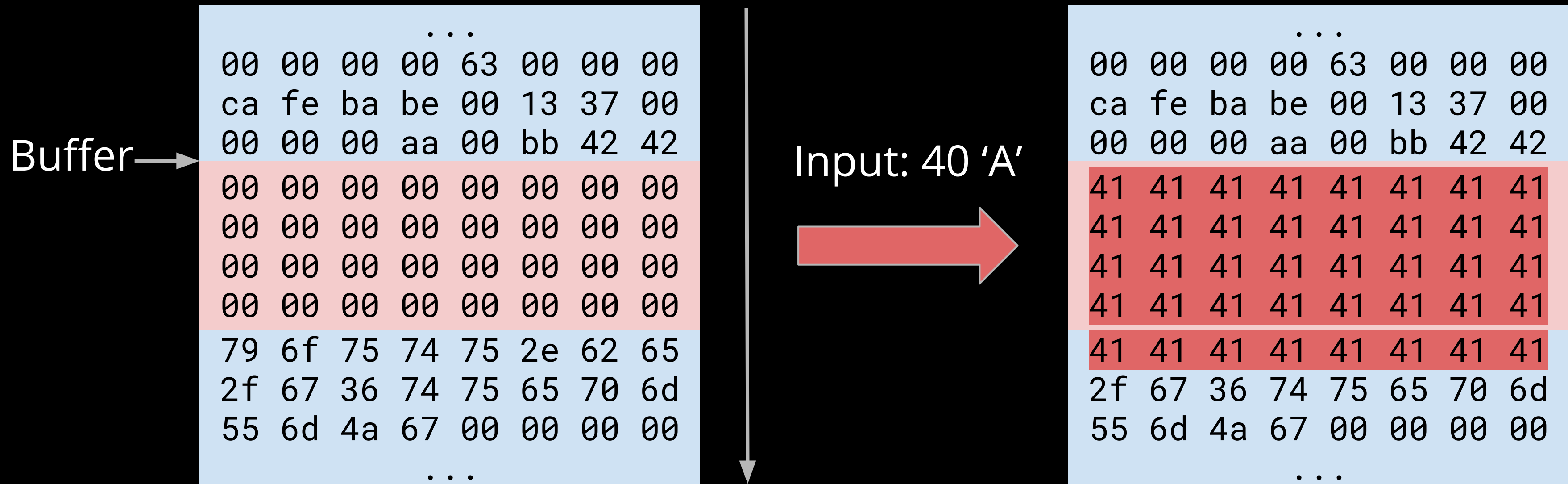
Buffer overflows

This program copies the user's input to a fixed size 32-byte buffer.



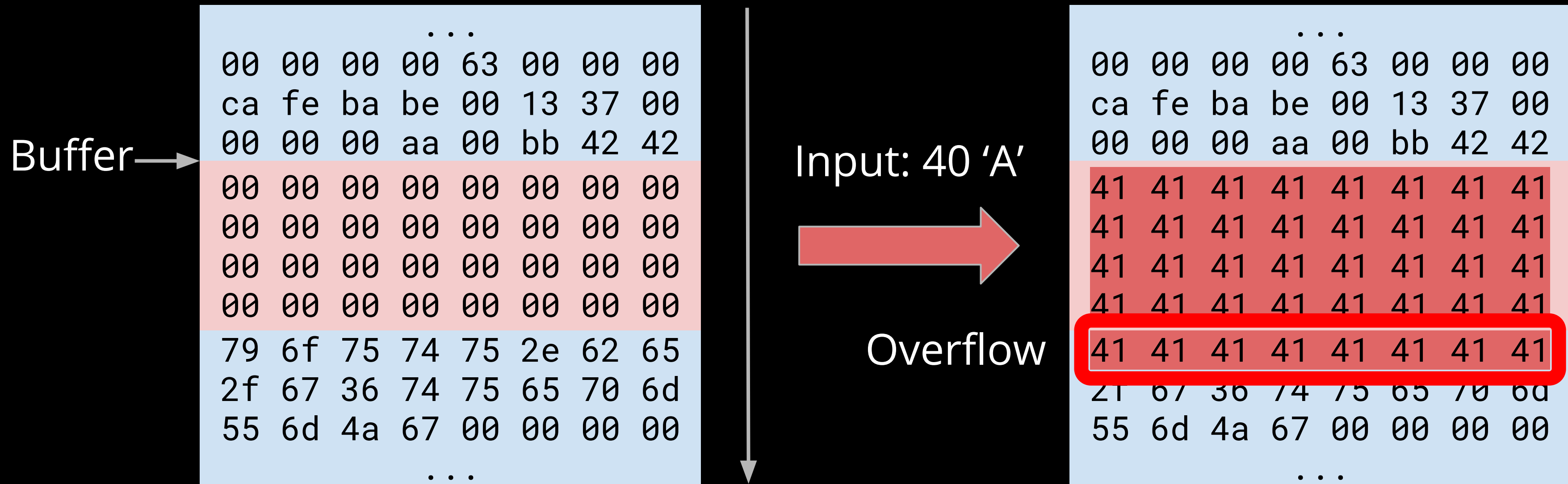
Buffer overflows

This program copies the user's input to a fixed size 32-byte buffer.



Buffer overflows

This program copies the user's input to a fixed size 32-byte buffer.



Exercise platform

http://spritzctf.pythonanywhere.com/

Exercise 1 - Auth Overflow

Inspired from Jon Erickson’s “*Hacking: The Art of Exploitation*”

```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf(“Enter password”);
   scanf(“%s”, password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

Exercise 1 - Auth Overflow

Inspired from Jon Erickson's "*Hacking: The Art of Exploitation*"

```
int check_authentication() {
   int auth_flag = 0;
   char password_buffer[16];
   printf("Enter password");
   scanf("%s", password_buffer);
   /* password_buffer ok? => auth_flag = 1 */
   return auth_flag;
}
```

Exercise 1 - The overflow

Buffer

Offset				
+0x00	??	??	??	??
	??	??	??	??
	??	??	??	??
	??	??	??	??
+0x10	??	??	??	??
	??	??	??	??
	??	??	??	??
+0x1c	00	00	00	00

Flag = 0

Exercise 1 - The overflow

Buffer

Offset				
+0x00	??	??	??	??
	??	??	??	??
	??	??	??	??
	??	??	??	??
+0x10	??	??	??	??
	??	??	??	??
	??	??	??	??
+0x1c	00	00	00	00

Flag = 0

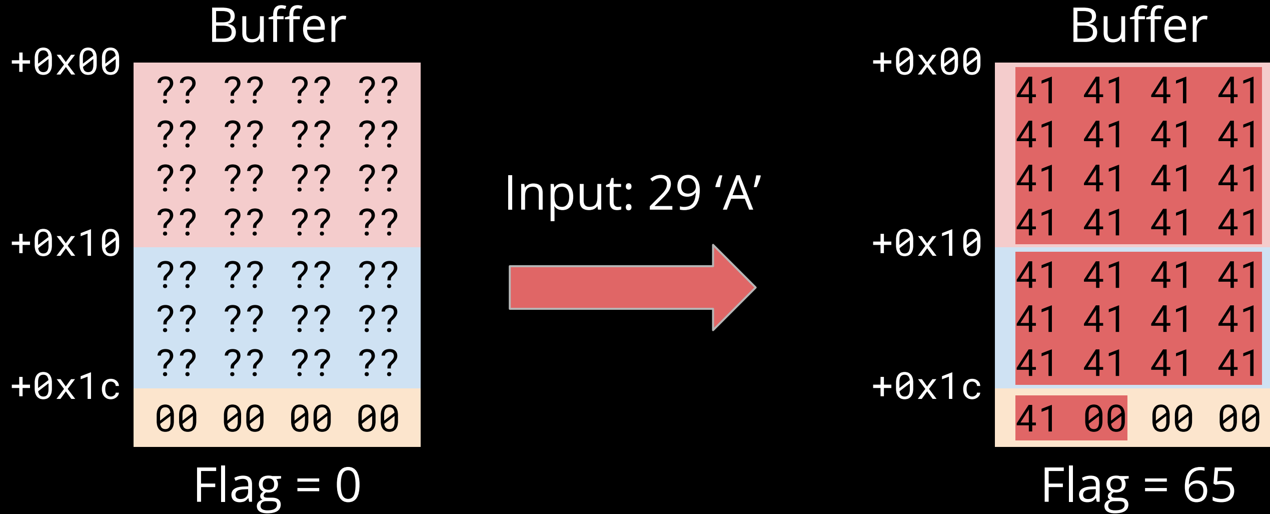
Input: 29 ‘A’

Buffer

Offset				
+0x00	41	41	41	41
	41	41	41	41
	41	41	41	41
	41	41	41	41
+0x10	41	41	41	41
	41	41	41	41
	41	41	41	41
+0x1c	41	00	00	00

Flag = 65

Exercise 1 - The overflow



check_authentication will now return 65.

Exercise 1 - The check

In C, anything `!= 0` is `true`.

The check will pass and grant us access. Profit!

Pwntools installation

Install python and pip:

- `[sudo] apt install python-pip`
- `[sudo] dnf install python-pip`
- `[sudo] pacman -S python2-pip`

Install pwntools:

```
pip2 install --user pwntools
```

Stack overflows

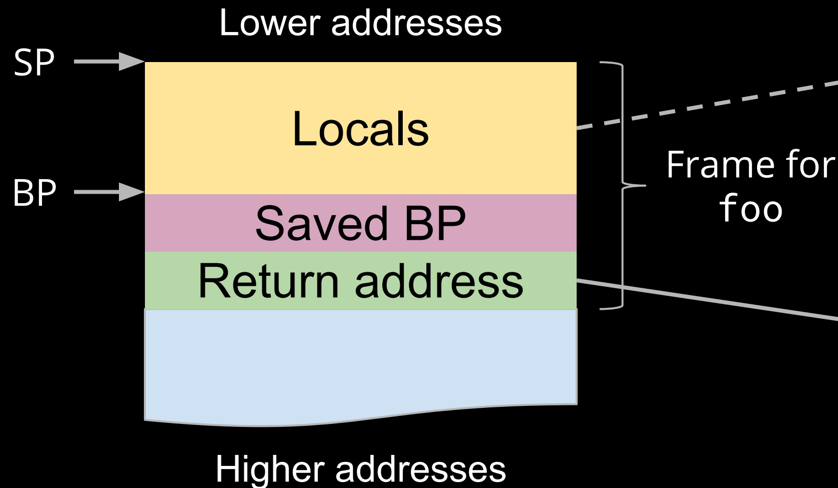
The stack contains information that keeps track of the program’s control flow.

Overflowing a buffer located on the stack could allows us to hijack the flow to wherever we want.

Must read: Aleph One, *Smashing the stack for fun and profit*, Phrack (1996)

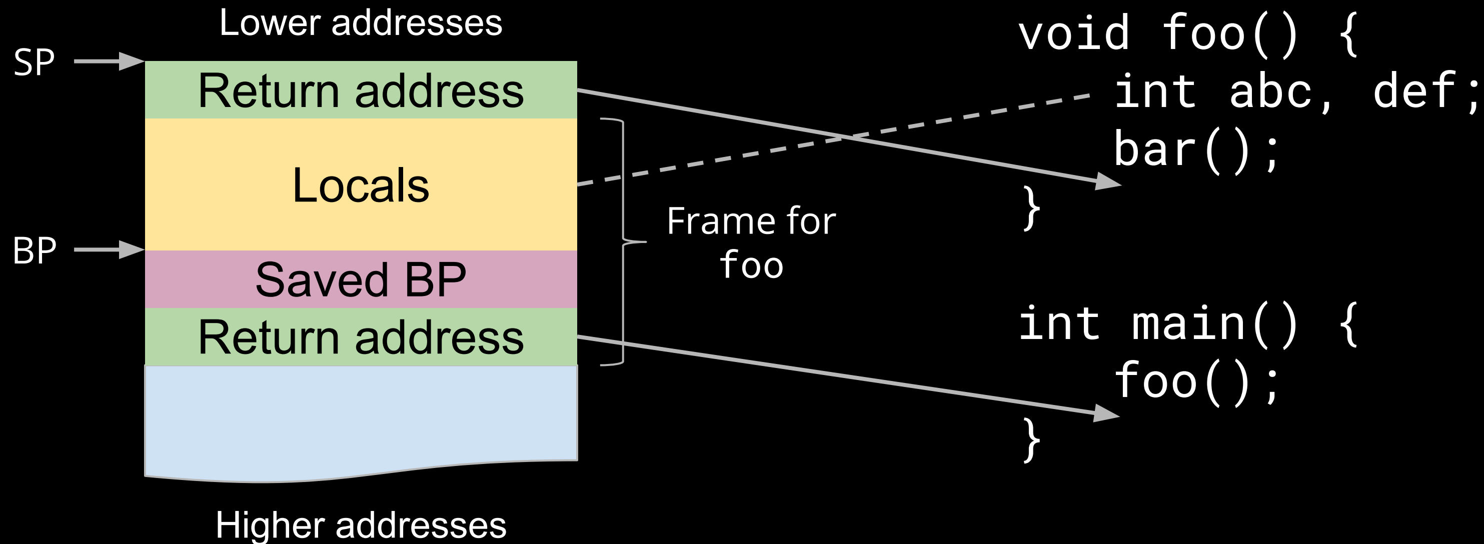
The x86 stack

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

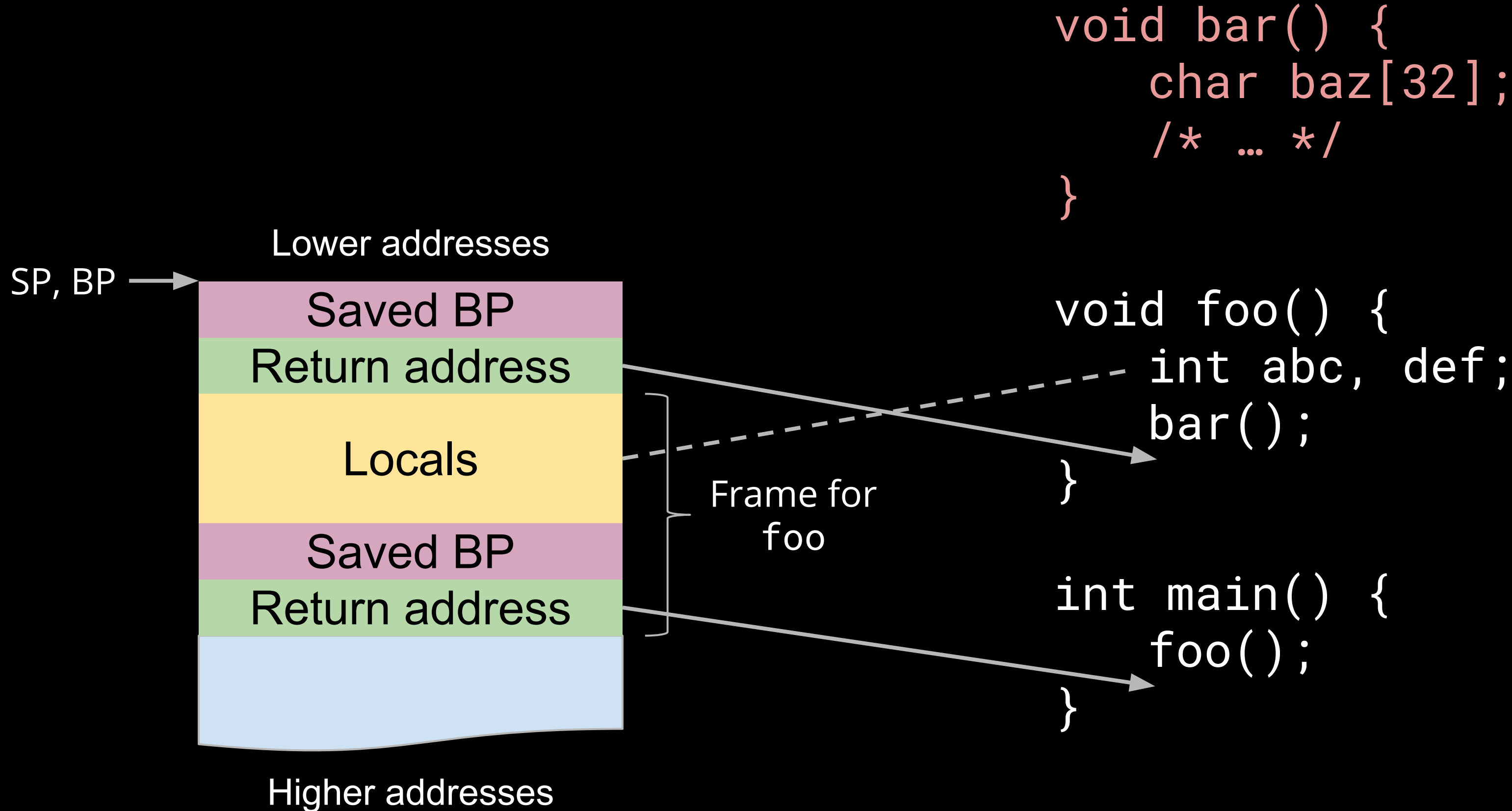


The x86 stack

```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

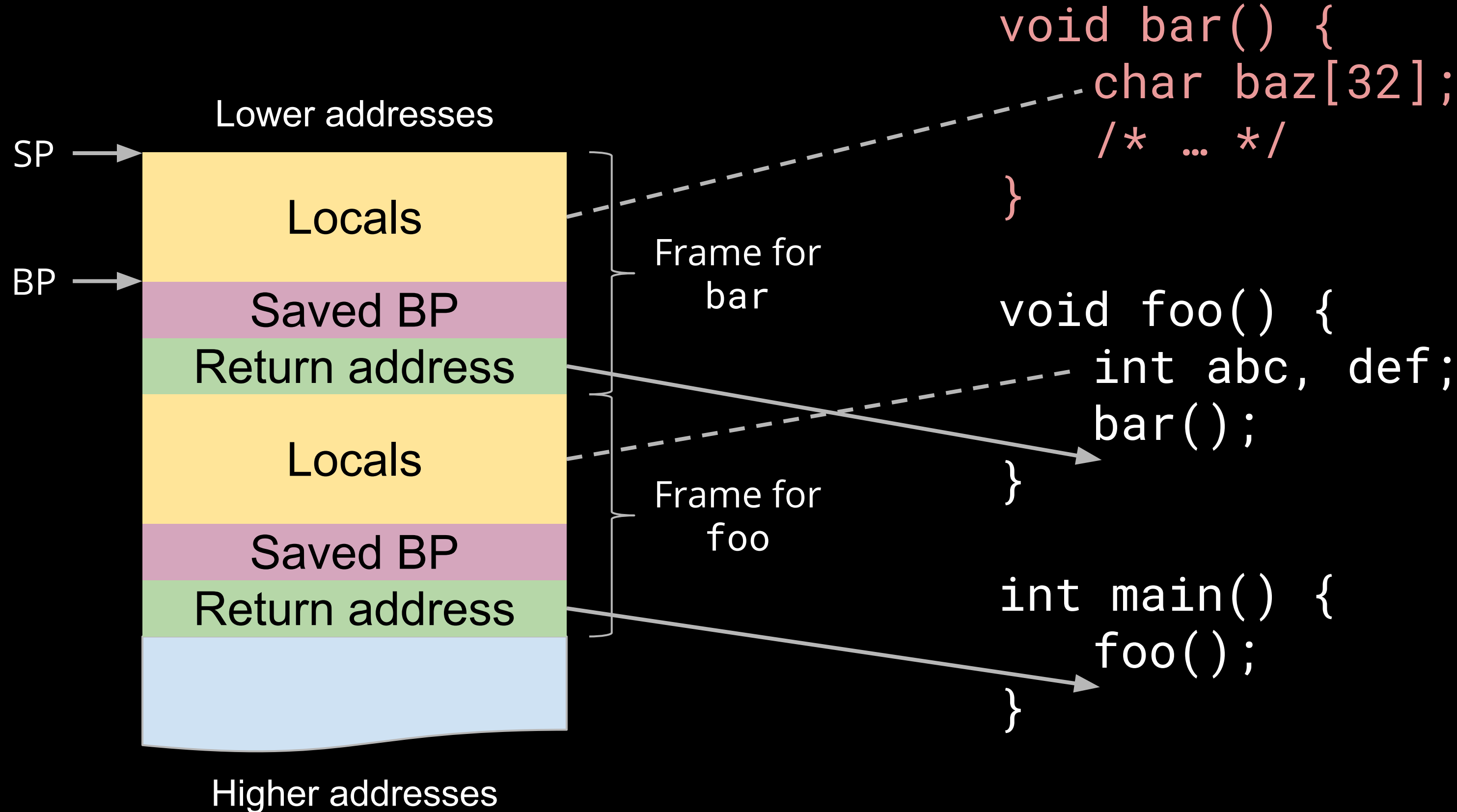


The x86 stack

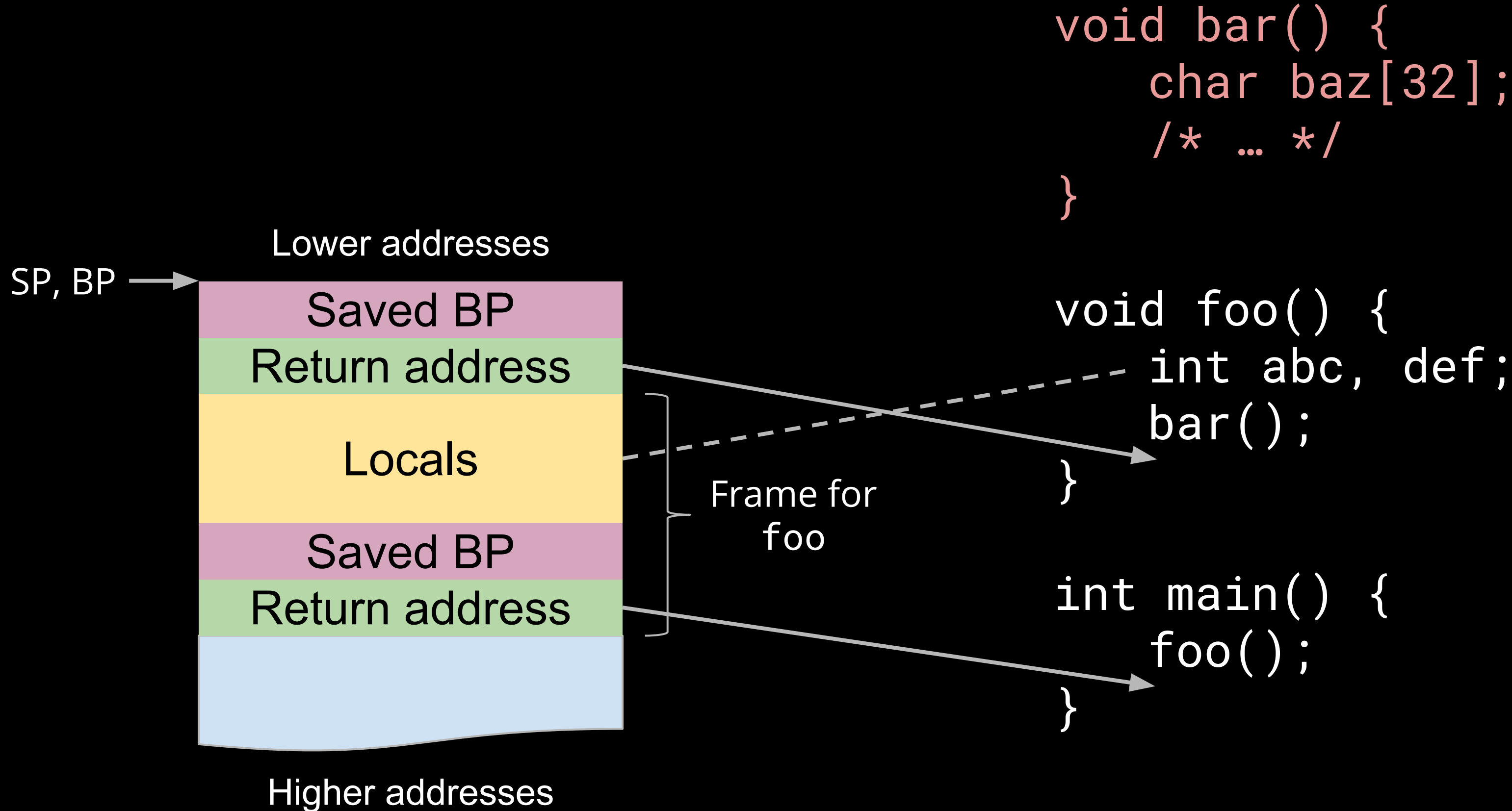



```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

The x86 stack

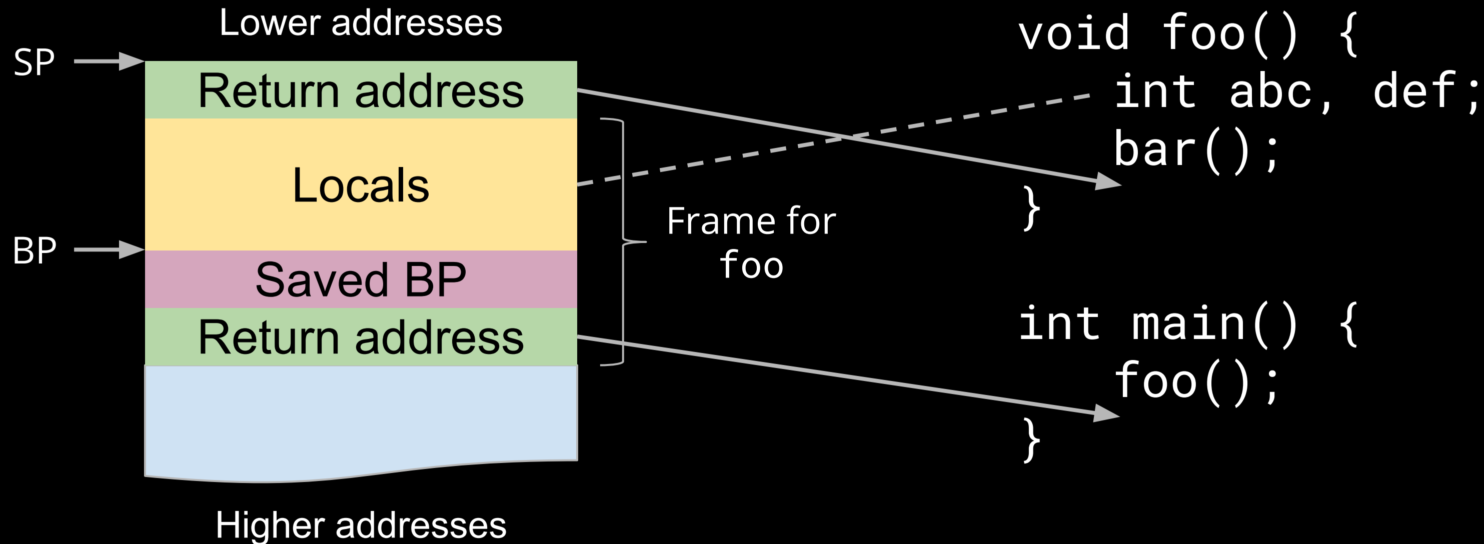


The x86 stack

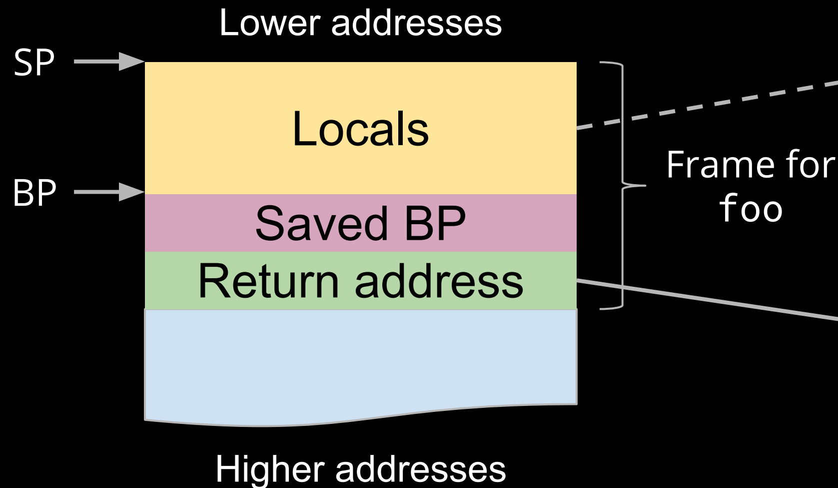


The x86 stack

```c
void bar() {
    char baz[32];
    /* … */
}
```



The x86 stack




```
void bar() {
    char baz[32];
    /* … */
}

void foo() {
    int abc, def;
    bar();
}

int main() {
    foo();
}
```

Stack overflows

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Stack overflows

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Input: 32 'A'

41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
c3	90	8b	00	ff	7f	00	00
d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Stack overflows

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Input: 40 'A'

41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Stack overflows

This program copies the user's input to a fixed size 32-byte buffer.

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Sv. BP	c3	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to `0x55b2307be0d5`

Input: 46 'A'

41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	41	41
41	41	41	41	41	41	00	00

Returns to `0x414141414141`

IP control achieved!

Exercise 2 - Remote Shell

Exploit plan:

Exercise 2 - Remote Shell

Exploit plan:

1. Find the offset between buffer and retaddr
2. Overwrite retaddr with `spawn_shell`
3. Make `main` return
4. ???
5. Profit!

Shellcode

Sometimes there’s no “magic” function we can return to.

So let’s inject our own code into the process.

This code is called *shellcode* because it usually opens a shell.

Shellcode

The program copies the user's input to a fixed size 32-byte stack buffer.

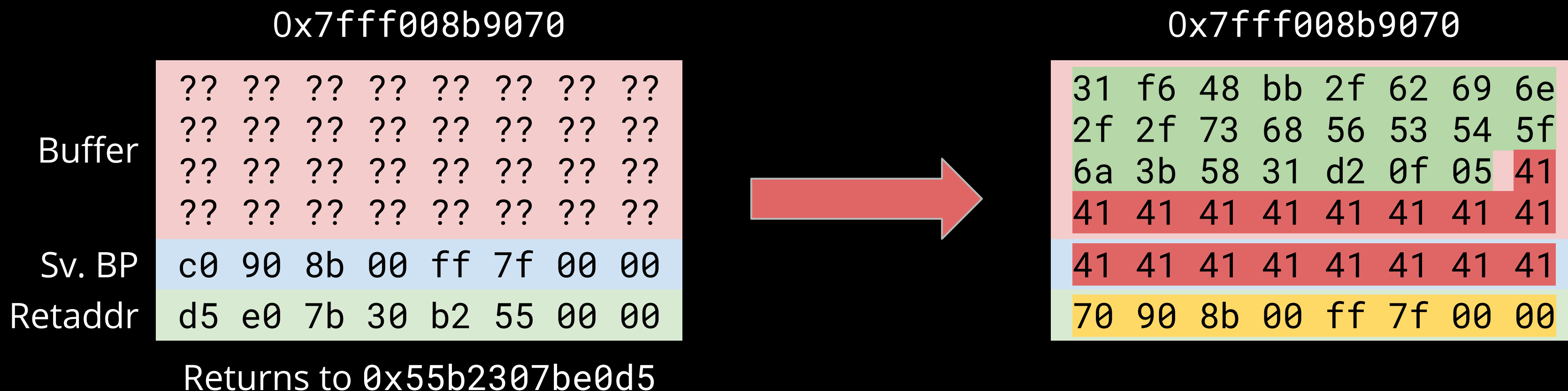
0x7fff008b9070

Buffer	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
	??	??	??	??	??	??	??	??
Sv. BP	c0	90	8b	00	ff	7f	00	00
Retaddr	d5	e0	7b	30	b2	55	00	00

Returns to 0x55b2307be0d5

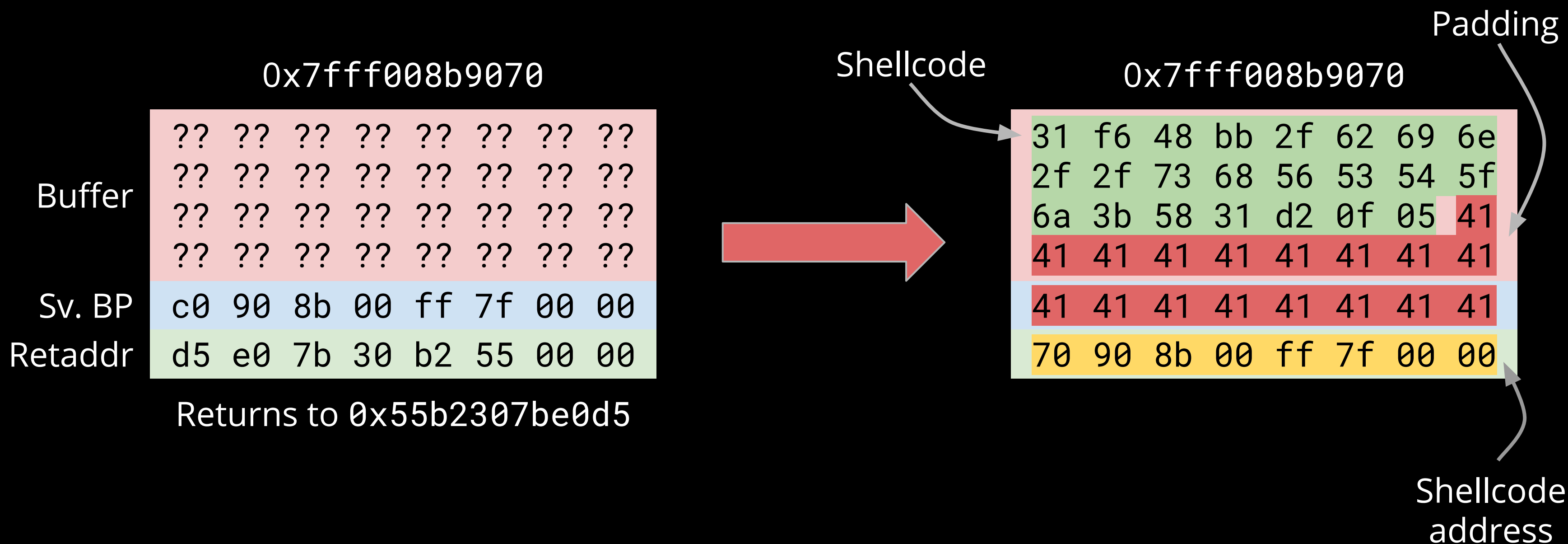
Shellcode

The program copies the user's input to a fixed size 32-byte stack buffer.



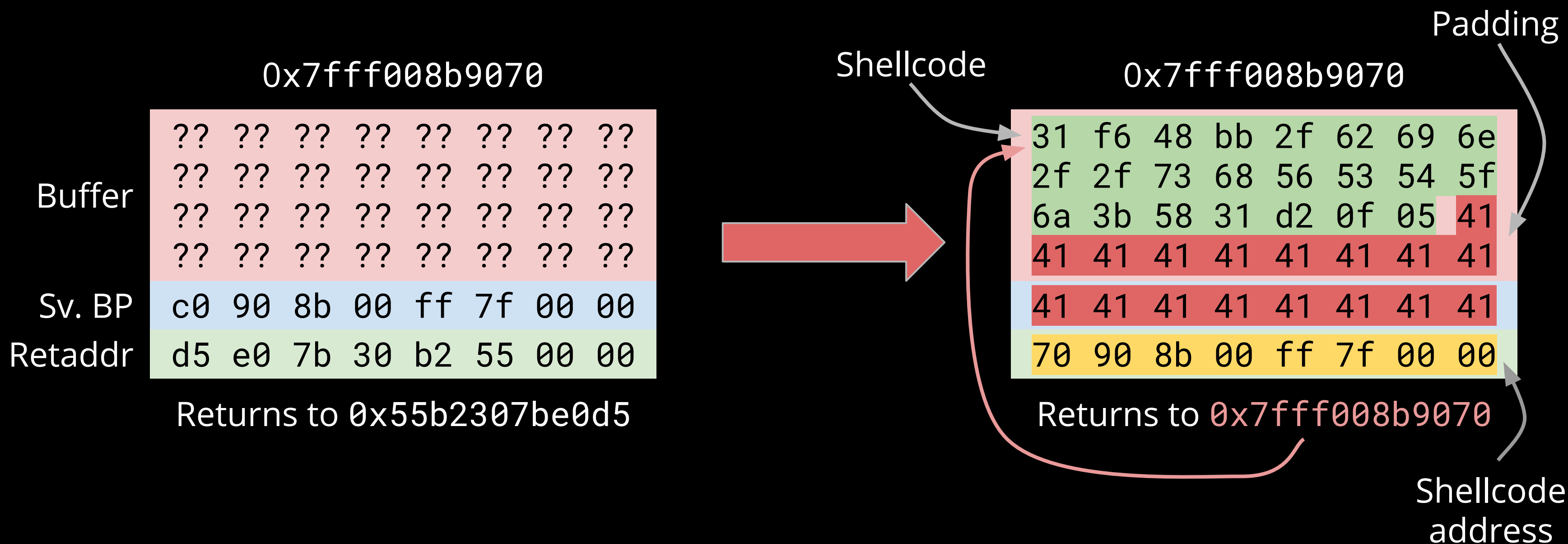
Shellcode

The program copies the user's input to a fixed size 32-byte stack buffer.



Shellcode

The program copies the user's input to a fixed size 32-byte stack buffer.



Exercise 3 - Useless

`buffer @ bp-0x70`

String operations: `‘A’*6 + ‘BC’ == ‘AAAAAABC’`

String length: `len(‘This is a string’)`

Numeric string to integer: `int(‘12345’)`

Import pwntools: `from pwn import *`

Configure pwntools: `context(os=’linux’, arch=’x86_64’)`

Process: `p = process(‘./useless’)`

Remote connection: `p = remote(‘207.154.238.179’, 8193)`

Line I/O: `p.recvline()` / `p.sendline(‘Hello!’)`

Interactive mode: `p.interactive()`

Assemble shellcode: `asm(shellcraft.sh())`

Mitigations

- Stack Canaries
  - Secret value overwritten by overflow
  - Bypass: infoleak, O(N) bruteforce (forkserver)
- Address Space Layout Randomization (ASLR)
  - Can’t jump if I don’t know where the code is
  - Bypass: infoleak, O(N) bruteforce (forkserver)
- Write XOR Execute (W⊕X, NX, DEP)
  - Prevent code injection
  - Bypass: code reuse (e.g., ROP)

What did we learn?

Always check your bounds!

As a general principle, if your application has a memory corruption vulnerability, most of the time a skilled and determined attacker will be able to exploit it.

Stuff to check out

- OverTheWire Vortex (http://overthewire.org/wargames/vortex/)
- sploitF-U-N (https://sploitfun.wordpress.com/2015/06/26/linux-x86-exploit-development-tutorial-series/)

?