]

Return Oriented Programming

Joseph Hallett

September 4, 2024

The plan

- Quick recap as to how C functions work
- Quick recap of the *smashing the stack for fun and profit* attack
- How we fixed it
- How we broke it again
- How we're going to try and fix it

Stack smashing

Lets suppose we have this function:

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

What does the stack look like as we call it?

Function call first

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

No arguments to this function to make things easy!

- (Though if we did then we'd look up the calling convention: stack for 32bit cdecl, registers then stack for AMD64; somnething else on Windows/ARM)

First we `call` it by saving where we were and sticking it on the stack (`SIP`), and loading the address of `greet()` into `rip`.

```
top of stack                          bottom of stack
                                              esp
                                              v
-------------------------------------------------
                                              |SIP |
-------------------------------------------------
```

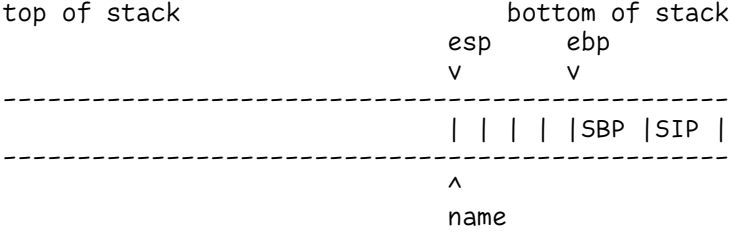
Next we need to create a new stack frame:

- How do we do that and how much space do we need?
- How much space do we need?

Stack frame next

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

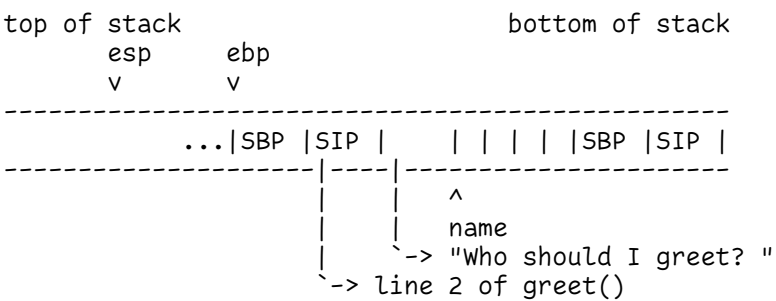
We need to save the previous stack frame so we need enough space for the current base pointer, plus space for the 4 chars of the name.



Then we need to make those three function calls to printf(), gets() and printf()…

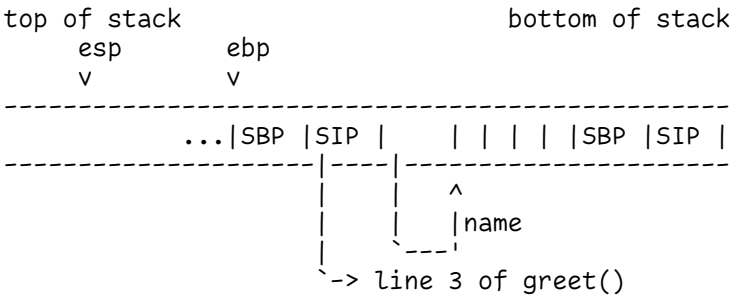
First printf()!

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```



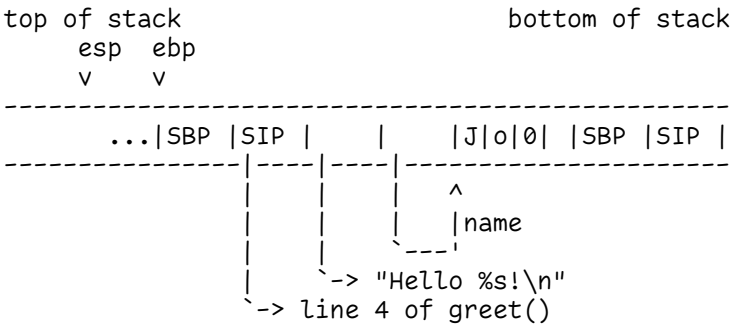
Next gets()!

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```



Finally printf() again!

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```



At which point we print *"Hello Jo!"*...

We're all done now...

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

All done now...

```
top of stack                          bottom of stack
                                  esp      ebp
                                  v        v
-------------------------------------------------
                                  |J|o|0| |SBP |SIP |
-------------------------------------------------
                                  ^
                                  name
```

Move the stack pointer back down

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

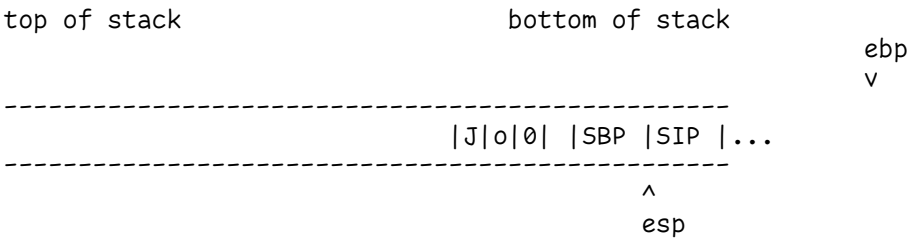
Leave any leftover data in the stack *wilderness* by moving the stack pointer back up.

```
top of stack                          bottom of stack
                                          ebp
                                          v
-------------------------------------------------
                                |J|o|0| |SBP |SIP |
-------------------------------------------------
                                          ^
                                          esp
```

Restore the previous stack frame…

```
void greet() {
  char name[4];
  printf("Who should I greet? ");
  gets(name);
  printf("Hello %s!\n", name);
}
```

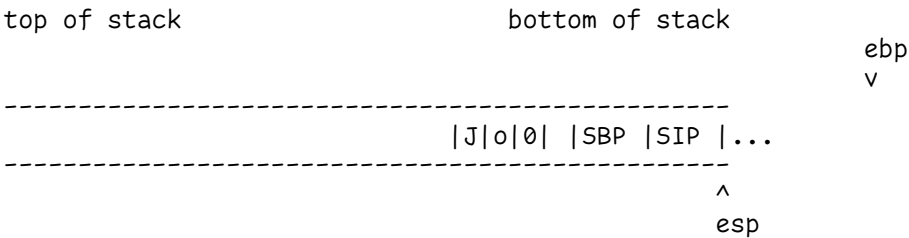
Pop the stack into the base pointer…



Restore the instruction pointer

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

Pop the stack into the instruction pointer



What happens when you actually run this?

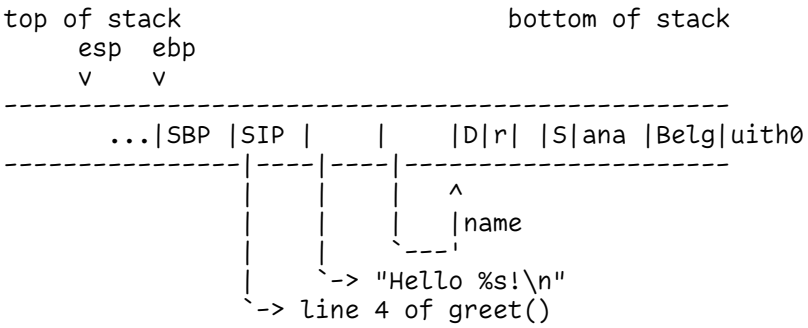
```
$ make test
cc test.c -o test

$ ./test
warning: this program uses gets(), which is unsafe.
Who should I greet? Jo
Hello Jo!
```

No warning at compile time but one at *runtime*

Smashing the stack

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

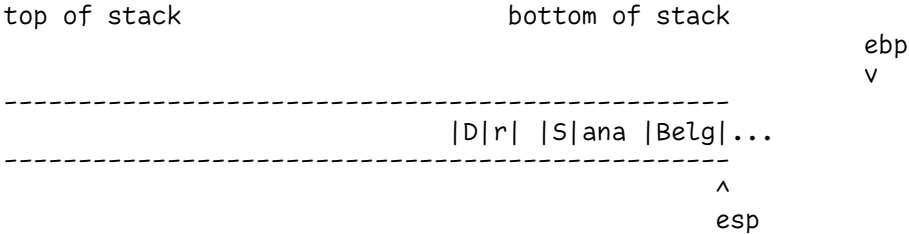
If we overflow the buffer…



Then we crash

```
void greet() {
  char name[4];
  printf("Who␣should␣I␣greet?␣");
  gets(name);
  printf("Hello␣%s!\n", name);
}
```

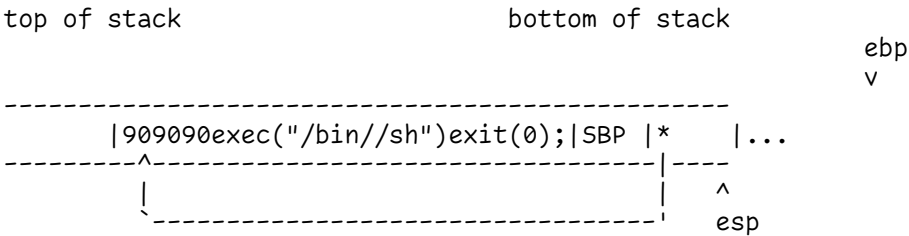
Then when we return we try and jump to `Belg` (`0x6c674265`)... and we'll promptly crash.



Shellcoding

But stack addresses are (somewhat) easily to guess…

- If we arrange for there to be program code where we're returning to…
- We can trick the program into running it for us.



(Get on with lab 3!)

So how do we respond to this?

Two *main* issues:

ASLR, or why is it so easy to guess where the stack is?

If we make it random, then it should be harder to guess *exactly* where things are

- But randomness is *crazily* expensive
- Randomising things costs time at program startup
- 32-bit x86 has limited numbers of bits for doing randomness

W⊕X, or why are we running code off of the stack?

Program code shouldn't exist on the stack: why are we running from there?

- Can implement the protection in hardware (fast!)

W⊕X

For each page of program memory, add an extra bit. Memory can be marked as either *writeable* or *executable*.

Writable you're allowed to change the value of memory

Executable you're allowed to run code from this memory

Program loading will be slower (2 extra systemcalls) but *overall* good improvement to security

- Mark code region as writable with `mprotect`
- Load code into memory
- Mark code as executable with `mprotect`

There's one class of programs that's going to hate this!

JITting compilers

JavaScript and Java both compile programs on the fly a few instructions at a time

- Makes for really fast VMs…
- But two extra syscalls per JIT'd block means *significant* overhead

(So still mechanisms to turn it off…)

(Also makes polymorphic malware harder to write for you virus lovers…)

But why do we need to load code anyway?

```
$ man 3 system
SYSTEM(3)                  Library Functions Manual                  SYSTEM(3

NAME
     system - pass a command to the shell

SYNOPSIS
     #include <stdlib.h>

     int
     system(const char *string);

DESCRIPTION
     The system() function hands the argument string to the command
     interpreter sh(1).  The calling process waits for the shell to finish
     executing the command, ignoring SIGINT and SIGQUIT, and blocking SIGCHLD
```

Return to libc

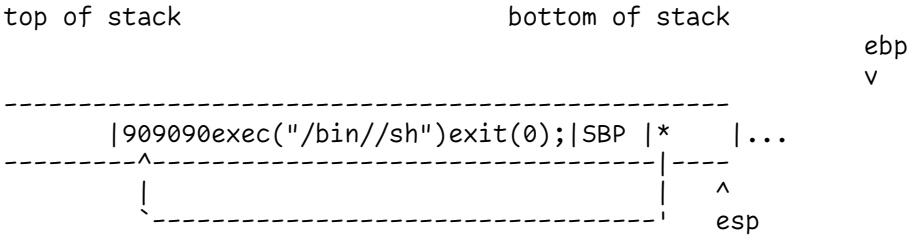
Solar Designer strikes again!

The `system()` function does everything an `execve` shellcode does

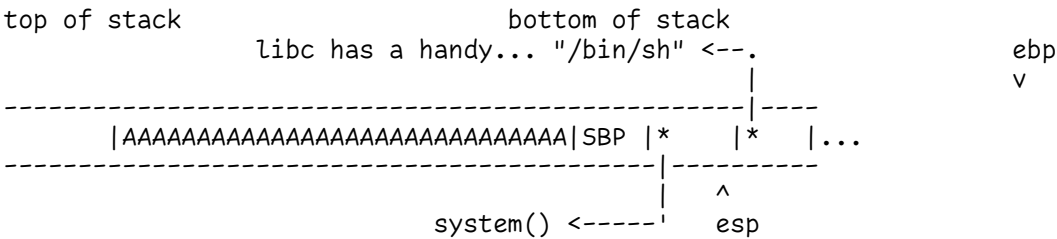
- And its already loaded into memory and marked executable
  In 32bit cdecl calling conventions arguments to functions go via the stack
  - Which we control with an overflow

So lets do this!

Instead of this:



Lets do this:



It's an art

Solar Designer described the exploit in a mailing list post to the *Bugtraq* list in 1997. It ends with…

> *That's all for now.*
> *I hope I managed to prove that exploiting buffer overflows should be an art.*
> *Signed,*
> *Solar Designer*

So how are we going to deal with this?

Return to libc works because:

- Library functions are in predictible locations
- Arguments go via the stack which is corruptable

How are we going to fix this?

ASLR no really, we do need to randomise stuff… but 32bit computers really just don't have enough bits…

Arguments Window's fastcall convention passes things via registers first and then the stack. So do syscalls. Maybe time to retire it?

Both these fixes mean fundamentally changing how the OS and CPU work which we normally try and avoid…

Welcome to 64bit

New 64bit architecture! Everyone upgrade!

- This was what I had in my iBook when I first came to uni
  - (back in 2006 :-()

No more passing on the stack (by default)!
Loads of bits for randomisation!

We're good now right…?

Randomness is still expensive though

```
nm -D /usr/lib/libc.so.96.2 | wc -l
```

1679

That's a lot of symbols to randomise! And a lot of entropy to spend at program link time…

(Sana will cover ASLR implementation in a few weeks…)

Quick ASLR

So instead of loading *each function* into a random location lets load *each library* at a random offset!

- One random number per library instead of 1679.
- You can just `mmap()` in the whole library which is *fast*
- You still don't know where the functions are *precisely*

But it does mean that if a single pointer is leaked from that library, then *all* the pointers are leaked

- You might not know where `fprintf` and `sscanf` are in memory…
- But you know there are precisely `b0` bytes between them

```
$ nm -nD /usr/lib/libc.so.96.2
...
0006a900 T fprintf
0006a9b0 T sscanf
...
```

Return to libc 2.0

Now to use *return to libc* our attack chain is a bit more complex

1. Find a buffer overflow
2. Break ASLR by leaking a pointer to a library
3. Return to `main` to restart the program without re-randomising
4. Re-exploit buffer overflow to jump to the library function you now know the address of

Its hardly arbitrary code execution though… can we do better?

And you thought assembly was bad…

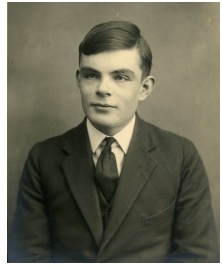
```
++++++++++[>+++++++>++++++++++>+++>+<<<<-]
>++.>+.+++++++..+++.>++.<<+++++++++++++++.
>.+++.------.--------.>+.>.
```

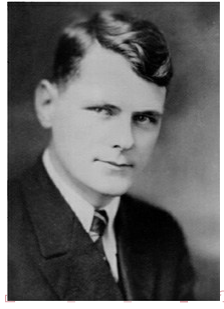
This is *brainfuck*. It assumes memory is a big tape of *cells*.

- `+` increments a cells value
- `-` decrements a cells value
- `>` moves to the next cell
- `<` moves to the previous cell
- `[` and `]` define a loop until the cell at the end is zero
- `.` outputs the current cell value
- `,` reads an input to the current cell

It is *provably* Turing complete (given an infinite tape):

- So *any* program that can be written…
- *Could* be written in brainfuck





Good films start at the end

Why do we need to start a function at the beginning?

- Functions do a bunch of interesting stuff then `ret`

Return oriented programming

A *gadget* is the stuff immediately before the `ret`

- Typically 1 or 2 instructions
- If we could construct a brainfuck compiler out of just these gadgets…
  - Then we could encode *any* program as a sequence of returns through the gadgets…
  - …by dumping *multiple* return addresses on the stack with our overflow

```
...
xor rax, rax
ret
```

```
...
inc rax
ret
```

```
...
inc rbx
ret
```

```
...
pop rbx
ret
```

```
...
syscall
ret
```

Oh no...

Lets pretend we want to call `exit(0)` to crash a program early (but cleanly).

- `exit` is syscall number 6
- Calling convention is syscall in `rax`, return code in `rbx`.

Can we use these gadgets to make this syscall?

```
...
xor_a:      xor rax, rax
            ret

            ...
inc_a:      inc rax
            ret

inc_b:      ...
            inc rbx
            ret

            ...
pop_b:      pop rbx
            ret

            ...
sys:        syscall
            ret
```

Oh no, oh no…

Lets pretend we want to call `exit(0)` to crash a program early (but cleanly).

- `exit` is syscall number 6
- Calling convention is syscall in `rax`, return code in `rbx`.

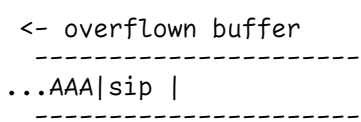
Can we use these gadgets to make this syscall?

Yes! We'd just have to return back through 10 instructions!

```
xor_a       ; rax = 0, rbx = ?
pop_b       ; rax = 0, rbx = 0xffffffff
0xffffffff
inc_b       ; rax = 0; rbx = 0
inc_a       ; rax = 1, rbx = 0
inc_a       ; rax = 2, rbx = 0
inc_a       ; rax = 3, rbx = 0
inc_a       ; rax = 4, rbx = 0
inc_a       ; rax = 5, rbx = 0
inc_a       ; rax = 6, rbx = 0
sys         ; exit(0)
```

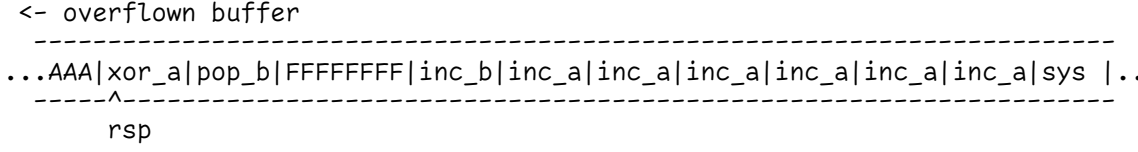
Are we really doing this? (yes)

Lets start with an overflow onto a return address, like we've been doing all week.



Are we really doing this? (setup)

Lets stick our ROP chain on the stack



And lets start returning!

Are we really doing this? (1)

```
rip -> xor rax, rax                rax: ?
       ret                         rbx: ?

 <- overflown buffer
  ------------------------------------------------------------------------
...AAA|xor_a|pop_b|FFFFFFFF|inc_b|inc_a|inc_a|inc_a|inc_a|inc_a|inc_a|sys |..
  -----------^------------------------------------------------------------
             rsp
```

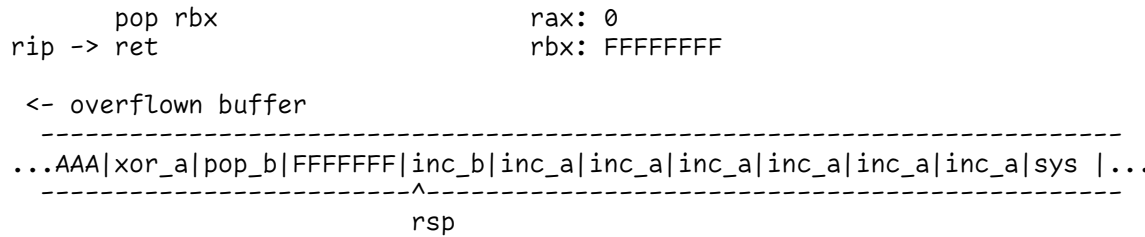
Are we really doing this? (2)

```
       xor rax, rax                rax: 0
rip -> ret                         rbx: ?

 <- overflown buffer
  ------------------------------------------------------------------------
...AAA|xor_a|pop_b|FFFFFFFF|inc_b|inc_a|inc_a|inc_a|inc_a|inc_a|inc_a|sys |..
  -----------^------------------------------------------------------------
             rsp
```

Are we really doing this? (3)

```
rip -> pop rbx                          rax: 0
       ret                              rbx: ?

 <- overflown buffer
  ------------------------------------------------------------------------
...AAA|xor_a|pop_b|FFFFFFF|inc_b|inc_a|inc_a|inc_a|inc_a|inc_a|inc_a|sys |...
  -----------------^------------------------------------------------------
                   rsp
```

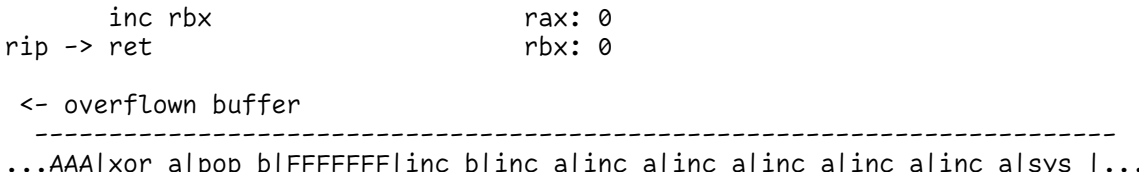
Are we really doing this? (4)



Are we really doing this? (5)

```
rip -> inc rbx                          rax: 0
       ret                              rbx: FFFFFFFF

 <- overflown buffer
  ------------------------------------------------------------------------
...AAA|xor_a|pop_b|FFFFFFF|inc_b|inc_a|inc_a|inc_a|inc_a|inc_a|inc_a|sys |...
  ------------------------------^-----------------------------------------
                                rsp
```

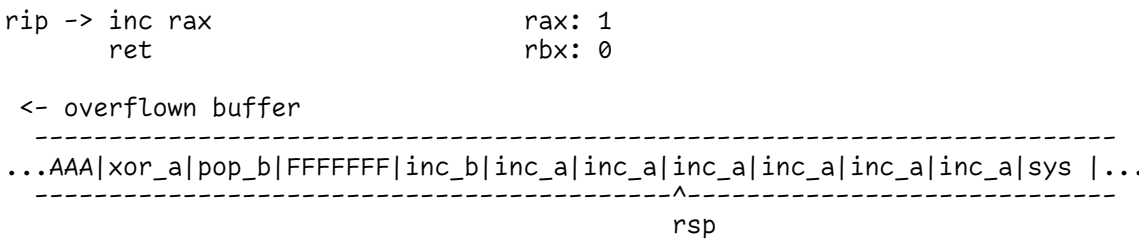
Are we really doing this? (6)



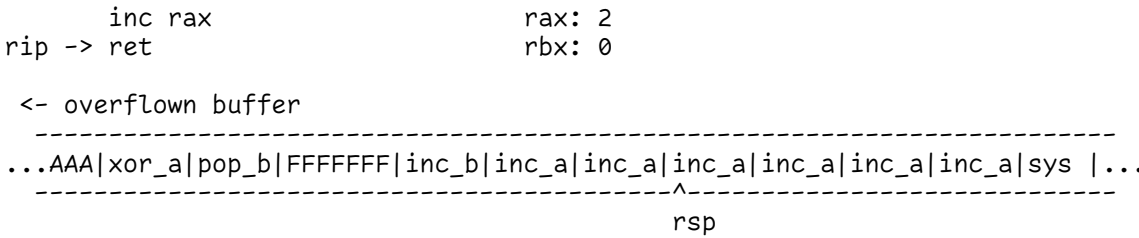
Are we really doing this? (7)

Are we really doing this? (8)

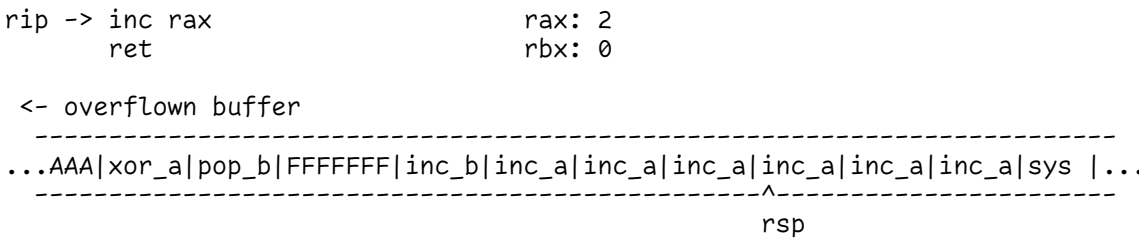
Are we really doing this? (9)



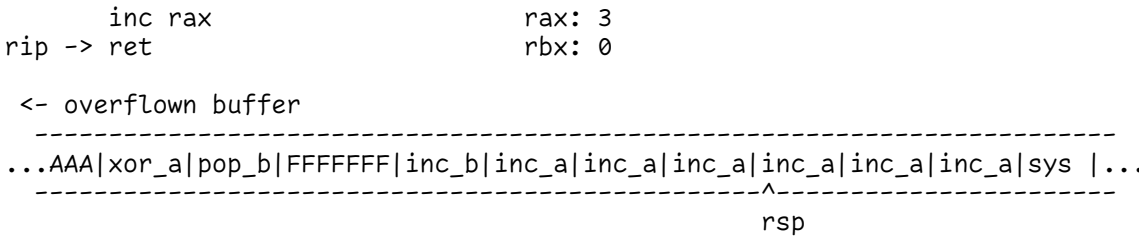
Are we really doing this? (10)



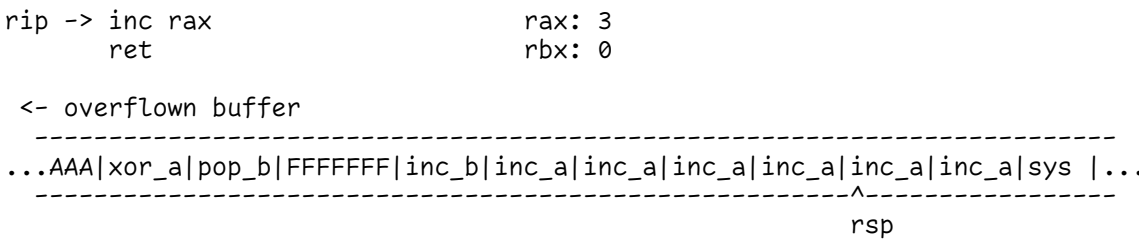
Are we really doing this? (11)



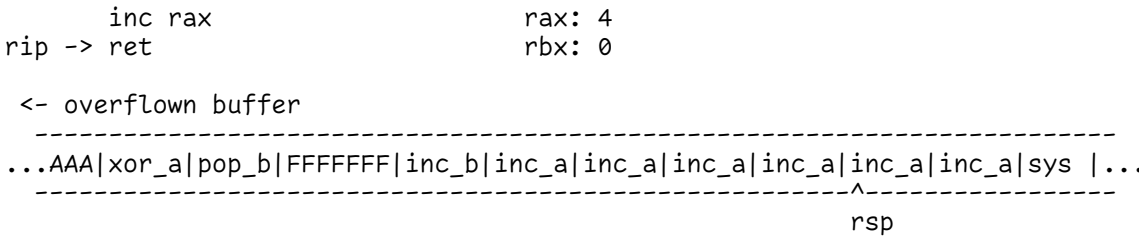
Are we really doing this? (12)



Are we really doing this? (13)



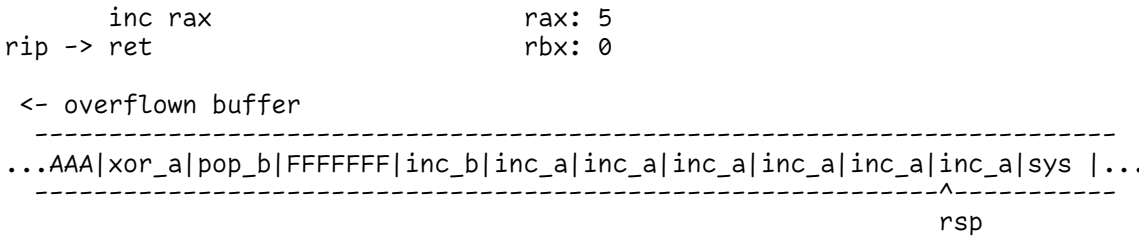
Are we really doing this? (14)



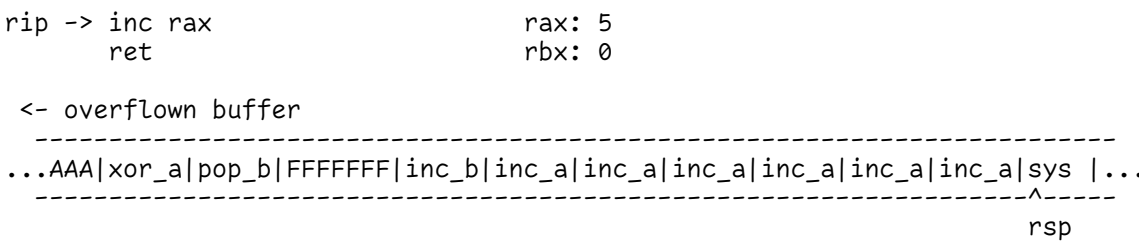
Are we really doing this? (15)

```
rip -> inc rax                          rax: 4
       ret                              rbx: 0

 <- overflown buffer
  ------------------------------------------------------------------------
...AAA|xor_a|pop_b|FFFFFFF|inc_b|inc_a|inc_a|inc_a|inc_a|inc_a|inc_a|sys |...
  ------------------------------------------------------------^-----------
                                                              rsp
```

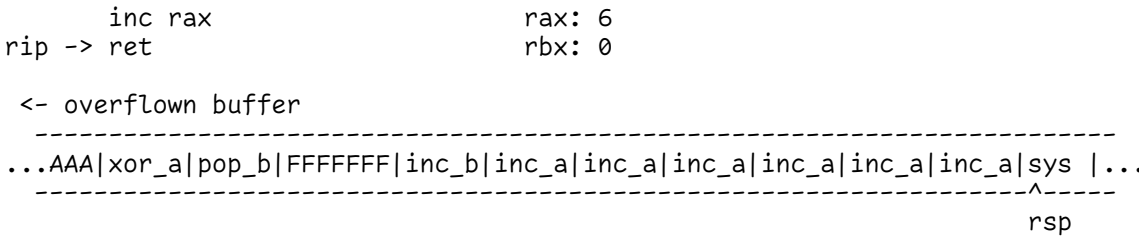
Are we really doing this? (16)



Are we really doing this? (17)



Are we really doing this? (18)



Are we really doing this? (19)

```
rip -> syscall                    rax: 6
                                  rbx: 0

 <- overflown buffer
  ------------------------------------------------------------------------
...AAA|xor_a|pop_b|FFFFFFF|inc_b|inc_a|inc_a|inc_a|inc_a|inc_a|inc_a|sys |...
  ------------------------------------------------------------------------
^
                                                                        rsp
```

Woohoo! `exit(0)` is called!

Return Oriented Programming

We just wrote an entire program by writing our shellcode as a path through pre-existing code in our program

- Arbitrary code execution purely through reuse!
- Most sufficiently large programs will contain enough usable gadgets that arbitrary code can be loaded

This is *(pretty much)* the state of the art for arbitrary code execution via a buffer overflow

- (*ish*… there's JOP too which is similar but for case statements)

We're going to be trying this *in practice* for next week's lab :-D

How do we stop this?

How do we stop this?

We can't trivially.
ROP falls out from fundamental decisions about how computers were architected that we made back in the 60s

- Von Neumann vs Harvard architectures

There are techniques that we can use to make it harder though

Shadow stacks keep a second stack for stack consistency checking and have the kernel kill the program if it ever gets out of sync

Full ASLR really randomise everything

Instruction Pointer Integrity Protections on return check that our instruction pointer goes only to whitelisted addresses (or is aligned)

How do we stop this?

…maybe we should just fix the buffer overflow?

- Rust/Zig!
- Not using unsafe languages

(But wheres the fun in that?)

Or at least that's what I used to say…

This year the OpenBSD project deployed a technique called *Branch Target Identification* to their entire OS.

- For x86-64 and ARM64
- Previewed in OpenBSD 7.3 (October 2023); on by default in OpenBSD 7.4 (February 2024)

Gist

- Compiler works out everywhere that you *could* jump to as the result of an *indirect jump*
  - A jump that depends on a variable (i.e. a function pointer loaded from a register)
- Add a `endbr64` instruction (on x64) to the target
- If you don't encounter an `endbr64` instruction… kill the program

Why does this break ROP?

How does this break ROP?

All your gadgets now have to start start with and `endbr64` instruction

- So typically are just the entry points to a whole function
- Much harder to find a useful set of gadgets
  - Can't divvy up functions as before

Okay, I'm technically simplifying

- It breaks a more practical variant of ROP called JOP which uses jump tables to do something similar
  - Think bytecode interpretters jumping based on next symbol seen
- This plus stack canaries makes ROP tricky though…
- This plus syscalls only through `libc` wrappers makes it **really** hard

Recap

We went from:

- buffer overflows in the 90s…
- …up to ROP in the 20s
- …and touched on state of the art mitigations

All that CS theory you learned in first year is actually useful for something.
There are worse ways of programming than assembly language!

Next time

In the lab ROP

Lectures Sana's taking over for a few weeks!

(I'll be back once more to talk about why computer hardware is fundamentally broken)