# Sig•Mal

Meeting 5

# How is everyone doing?

:)

# Binary Hacking / Binary Exploitation

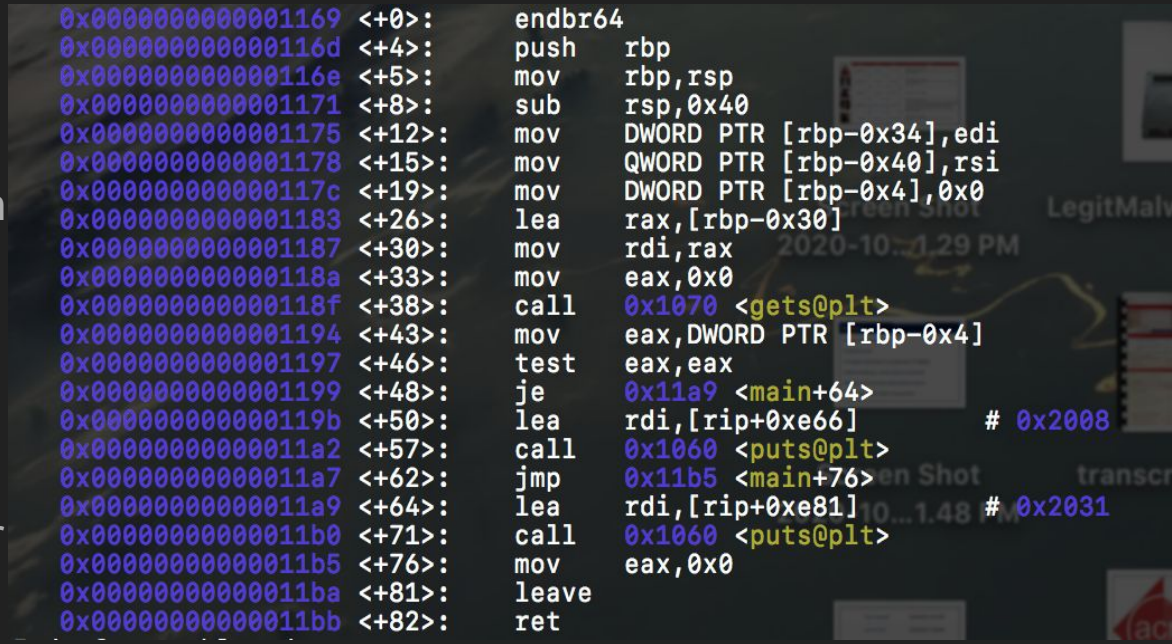
- Binary exploitation is the process of subverting a compiled application such that it violates some trust boundary in a way that is advantageous to the attacker.

Example(s):

  + Buffer Overflows → Memory Corruption
  + Format Strings
  + Heap overflows

# gdb

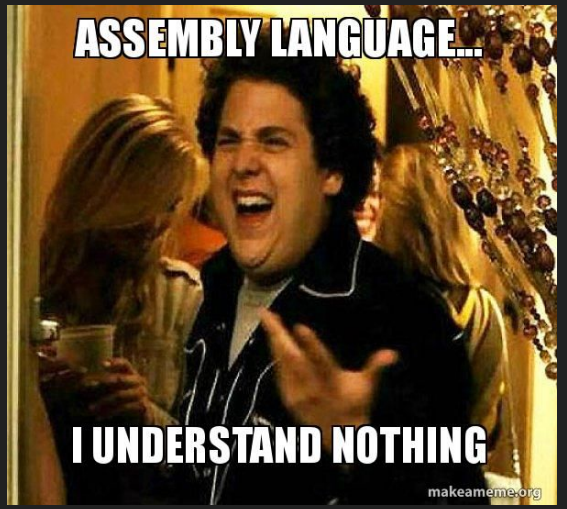
- GNU debugger is portable debugger and works with a ton of programming languages.
- Disassemble executables (pretty technical, less user friendly than Ghidra)



# gdb

- GNU debugger is portable debugger and works with a ton of programming languages.
- Disassemble executables (pretty technical, less user friendly than Ghidra)

```
0x0000000000001169 <+0>:     endbr64
0x000000000000116d <+4>:     push   rbp
0x000000000000116e <+5>:     mov    rbp,rsp
0x0000000000001171 <+8>:     sub    rsp,0x40
0x0000000000001175 <+12>:    mov    DWORD PTR [rbp-0x34],edi
0x0000000000001178 <+15>:    mov    QWORD PTR [rbp-0x40],rsi
0x000000000000117c <+19>:    mov    DWORD PTR [rbp-0x4],0x0
0x0000000000001183 <+26>:    lea    rax,[rbp-0x30]
0x0000000000001187 <+30>:    mov    rdi,rax
0x000000000000118a <+33>:    mov    eax,0x0
0x000000000000118f <+38>:    call   0x1070 <gets@plt>
0x0000000000001194 <+43>:    mov    eax,DWORD PTR [rbp-0x4]
0x0000000000001197 <+46>:    test   eax,eax
0x0000000000001199 <+48>:    je     0x11a9 <main+64>
0x000000000000119b <+50>:    lea    rdi,[rip+0xe66]        # 0x2008
0x00000000000011a2 <+57>:    call   0x1060 <puts@plt>
0x00000000000011a7 <+62>:    jmp    0x11b5 <main+76>
0x00000000000011a9 <+64>:    lea    rdi,[rip+0xe81]        # 0x2031
0x00000000000011b0 <+71>:    call   0x1060 <puts@plt>
0x00000000000011b5 <+76>:    mov    eax,0x0
0x00000000000011ba <+81>:    leave
0x00000000000011bb <+82>:    ret
```



# Assembly

- Assembly is “low-level” language
- 2 types of different syntax, Intel and AT&T
  - INTEL gang
- Not portable (must be written based on processor, OS, assembler you wish to use, bit on processor)
- “Human Readable” to the the Machine’s opcode/hex

# Assembly

On Linux/MacOS:

hexdump [location of executable]

# Assembly

On Linux/MacOS:

hexdump [location of executable]

On Windows:

Use Linux :)

{ or use Linux Subsystem for Windows }

# Assembly Example (The Chad “Hello World”)

Windows (32-bit processor)

```
    global  _main
    extern  _printf
    section .text
_main:
    push    message
    call    _printf
    add     esp, 4
    ret
message:
    db  'Hello, World!', 10, 0
```

Linux

64-bit

Both:

Intel

Nasm

```
        extern printf
        section .data
msg:    db "Hello, world!", 0
fmt:    db "%s", 10, 0
        section .text
        global main
main:
        push    rbp
        mov     rdi,fmt
        mov     rsi,msg
        mov     rax,0
        call    printf
        pop rbp
        mov rax,0
        ret
```

# YEAH, I know Assembly

Assembly

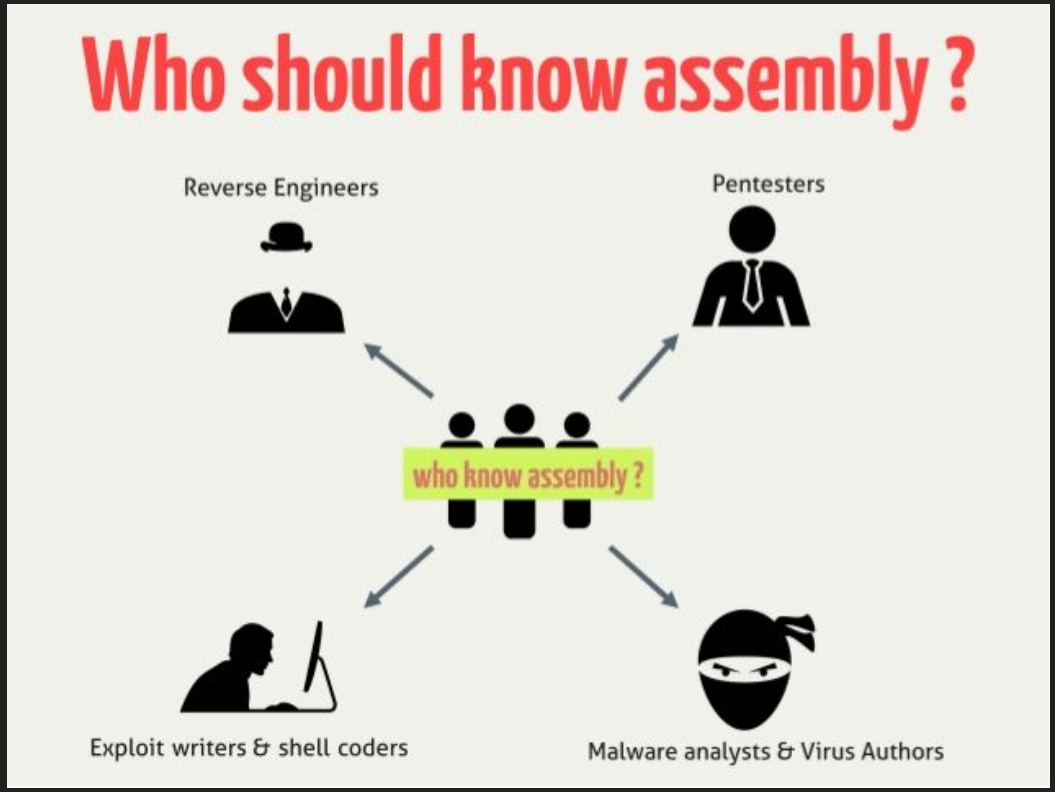
Somebody please help me
I am slowly deteriorating
mentally as I step through
my code in GDB for 80 hours
a week and all I can think
about is the sweet release
of death why the fuck
can't I just use an actual
language with libraries

Teachers: If you learn one programming language, you'll be able to learn any other in one or two weeks.

Assembler:

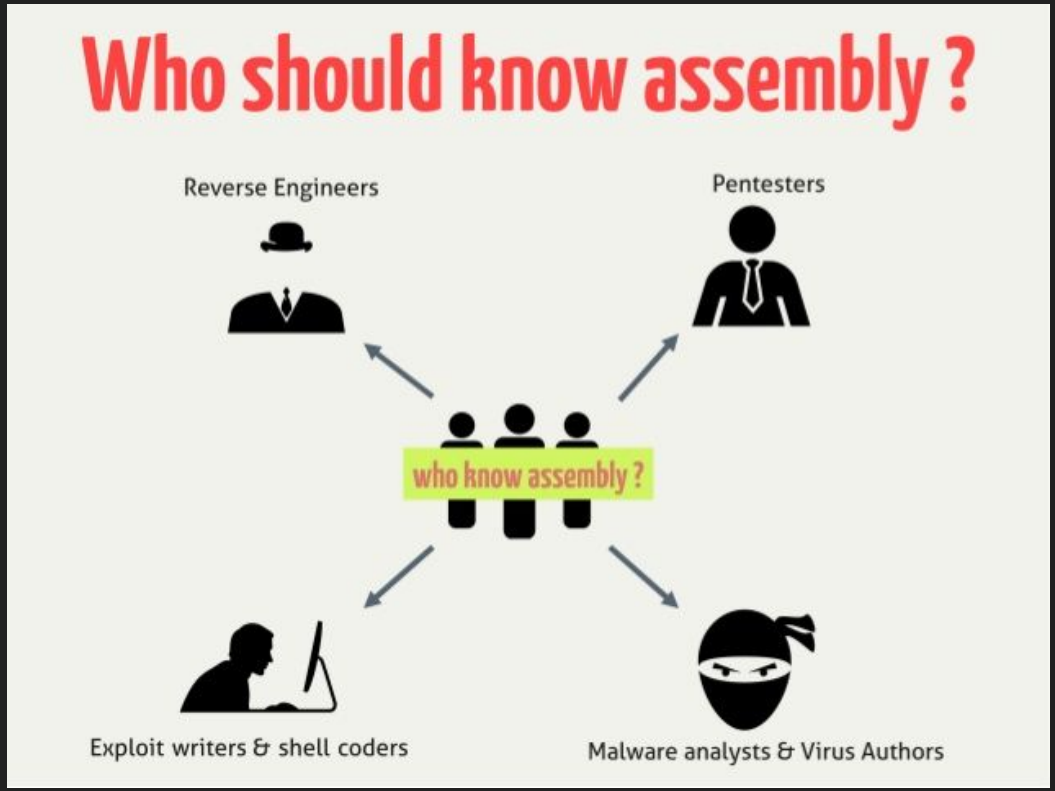
# Case for Assembly

- → → →



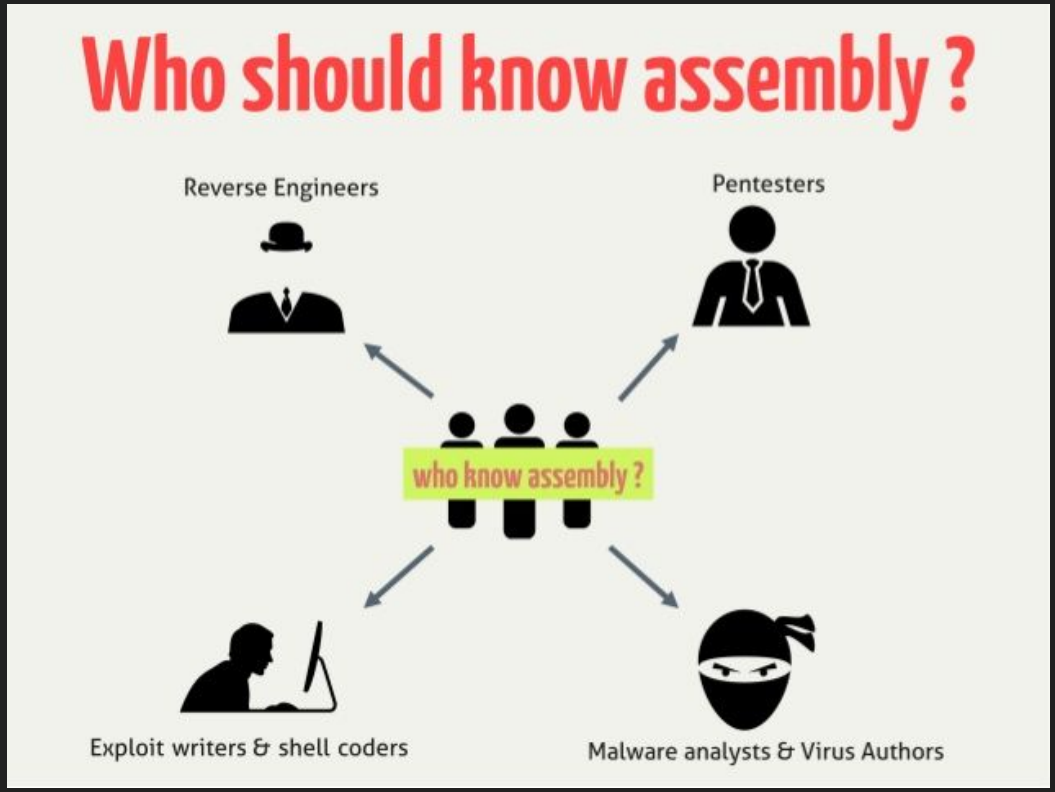
# Case for Assembly

- → → →
- More control over programs



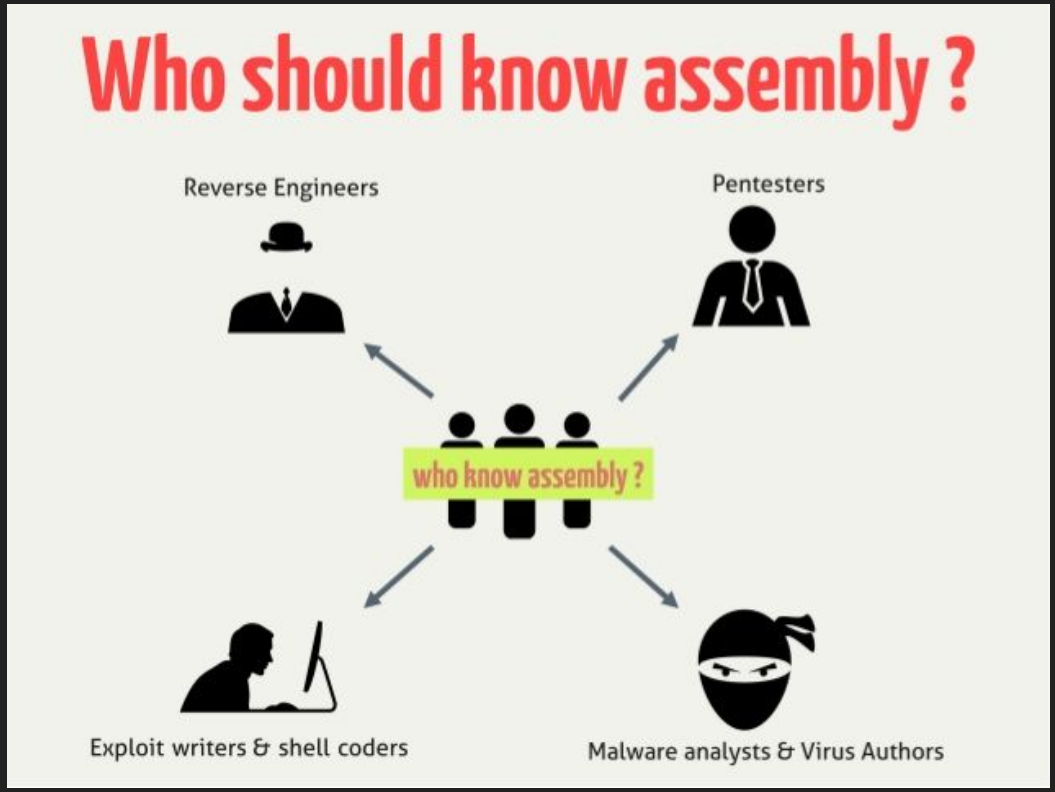
# Case for Assembly

- → → →
- More control over programs
- Clout



# Case for Assembly

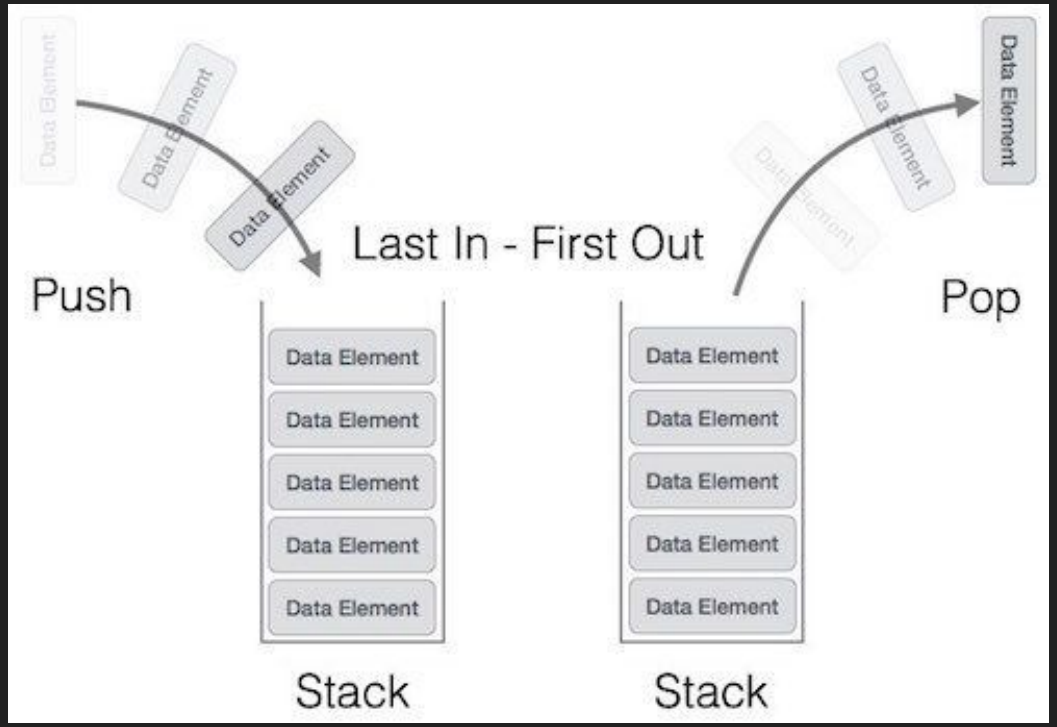
- → → →
- More control over programs
- Clout
- Flexing on others



# Stack

- Last In, First Out (LIFO)

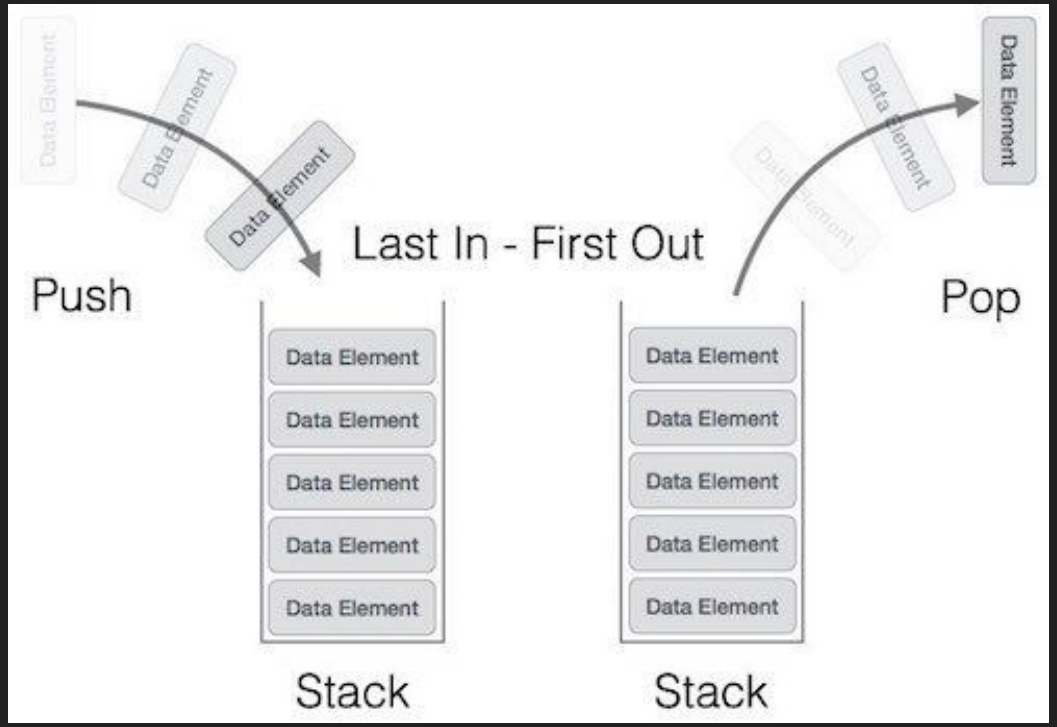
- Local variables operate on a stack, called a stack frame during program execution.



# Stack

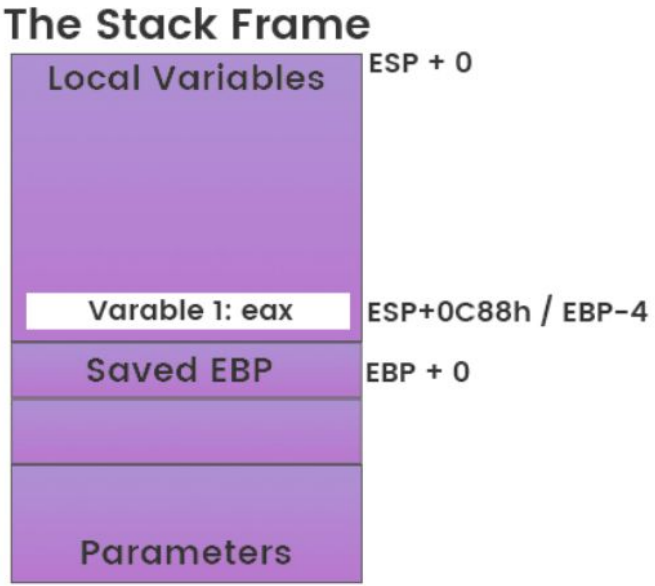
- Last In, First Out (LIFO)
- Local variables operate on a stack, called a stack frame during program execution.

  → “I wonder if we can exploit that someone how”

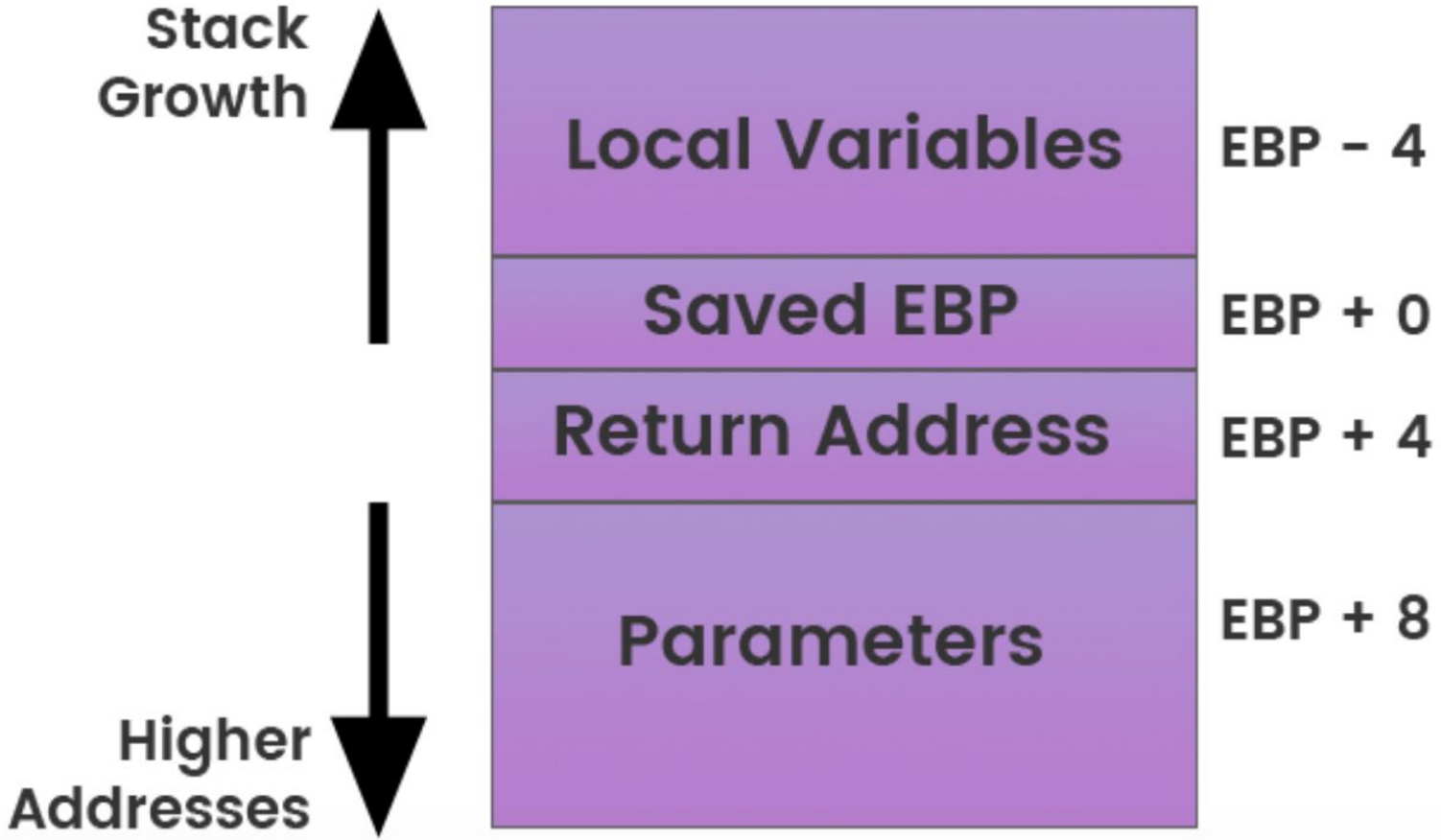


# Local Variables on the Stack

```
loc_4022B0:                                ; CODE XREF: start-82↓p
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF8h
                sub     esp, 0C8Ch
                mov     eax, ___security_cookie
                xor     eax, esp
                mov     [esp+0C88h], eax
                push    ebx
                push    esi
                mov     esi, [ebp+0Ch]
                push    edi
                push    0FFFFh
                push    offset unk_428DF0
                push    offset aAppdata ; "AppData"
                call    ds:GetEnvironmentVariableA
                neg     eax
                ...
                mov     esp, ebp
                pop     ebp
                retn
```



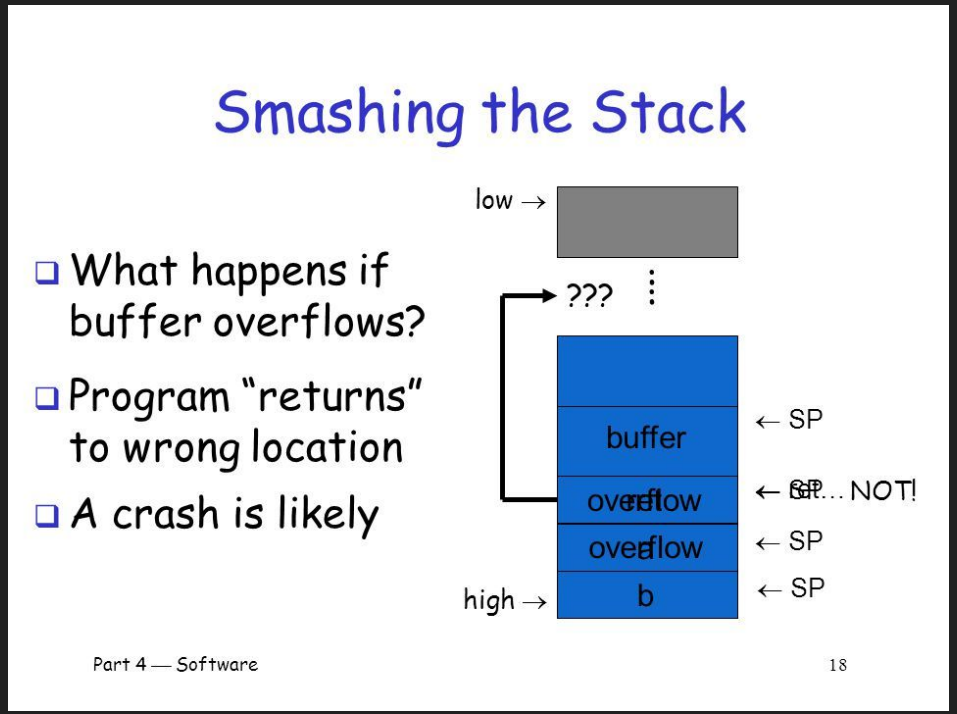
# The Stack Frame



# Buffer Overflow

- When a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations.
- Buffers are areas of memory set aside to hold data, often while moving it from one section of a program to another, or between programs.

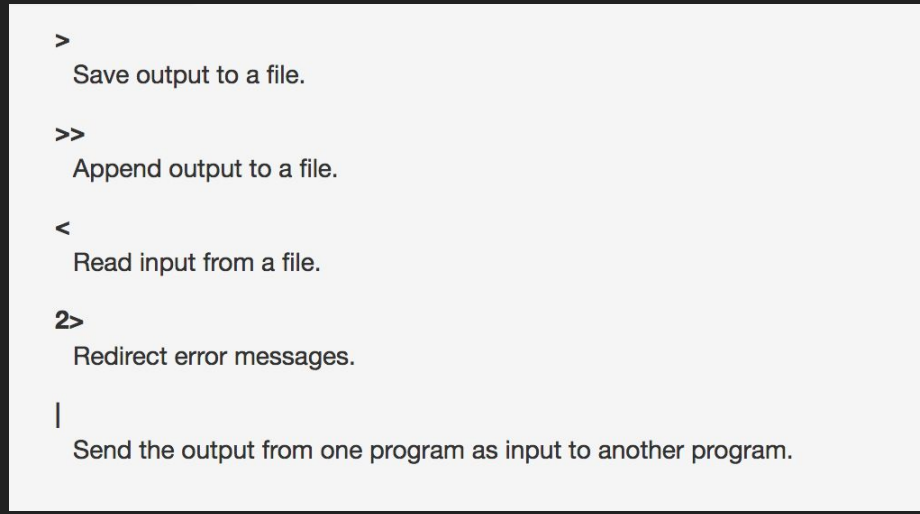
~ Data is written to space in memory ‘normally’ should be able to access.



# Piping and Redirection

Streams:

- Every program you may run on the command line has 3 streams, STDIN, STDOUT and STDERR.



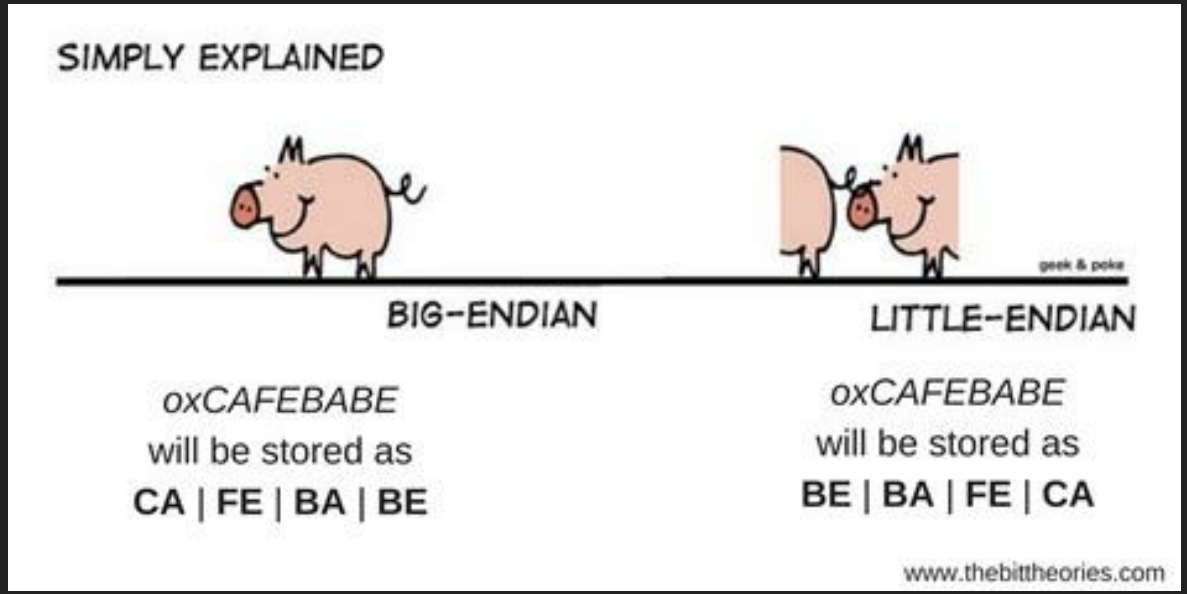

Bash stuff:

- Argument
  - `./a.out 'arg1'` || `a.exe 'arg1'`
  - `./a.out $('arg_fun')` || `a.exe $('arg_fun')` {I think}
- Input
  - `./a.out | 'input'` || `a.exe | 'input'` {I think}

# Computer Architecture (Endianness)

- Endian refers to how the order of bytes in a multi-byte value is perceived or acted upon.

The origin of the odd terms big endian and little endian can be traced to the 1726 book Gulliver's Travels, when a civil war broke out between ppl based on which side to crack an egg.



# More resources:

- https://microcorruption.com/
- Intel Manual (Good to look at for Assembly Instructions)
- https://malwareunicorn.org/workshops/re101.html#4 → For Assembly Basics
- https://www.hacksplaining.com/lessons
- https://old.liveoverflow.com/index.html
- http://security.cs.rpi.edu/courses/binexp-spring2015/

# LIVE DEMO TIME

# Buffer Overflow (Technical Under the Hood)

- Fun fact: the c code only works when compiled with -fno-stack-protector flag
  - gcc [file].c -fno-stack-protector
  - Otherwise, you might see an error about smashing the stack when trying to exploit it. (Stack Canaries)
  - Modern compilers and programs have stack canaries, built into them.
    - It helps mitigate simple buffer overflow attacks.