Buffer Overflows 101

segF4ult

$ whoami

- One of the Founders of Console Cowboys
- Feature Injector/Vulnerability Researcher of 4 Years
- Nerd who breaks stuff

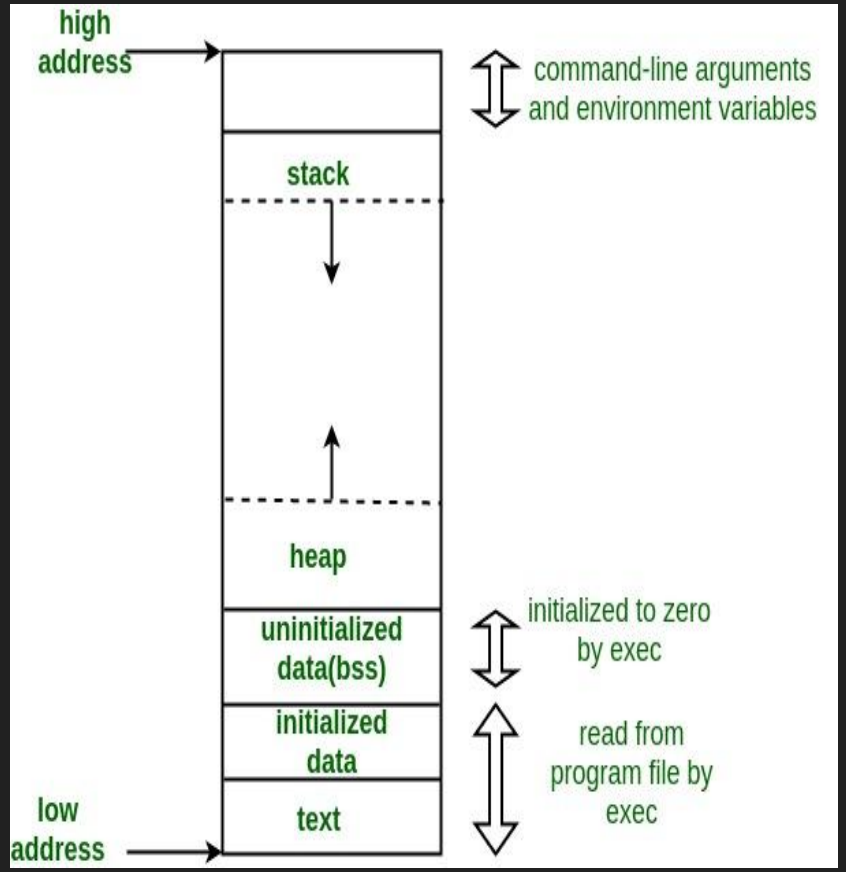
Buffer Overflows

- Bug class
- Main Idea
  - Read past Some buffer to corrupt data

Data Buffer

Variables in Memory

- Global Variables
  - Data/bss segments
- Local Variables
  - Static in size
  - Lives on the stack
- Heap Data
  - Dynamic in size
  - Allocated from malloc



Calling a Function

- Process of calling a function
- Setup Arguments to function
- Save where to go after
- Create space for local variables (Stack Frame)

Calling a Function

```c
int main(int argc, char **argv) {

    char *name = argv[1];


    printf("Hello World!, I am %s\n", name);


}
```

Pass Arguments and call printf

Not to bad right :)

Calling a Function

```
08049176 <main>:
 8049176:   8d 4c 24 04             lea    ecx,[esp+0x4]
 804917a:   83 e4 f0                and    esp,0xfffffff0
 804917d:   ff 71 fc                push   DWORD PTR [ecx-0x4]
 8049180:   55                      push   ebp
 8049181:   89 e5                   mov    ebp,esp
 8049183:   53                      push   ebx
 8049184:   51                      push   ecx
 8049185:   83 ec 10                sub    esp,0x10
 8049188:   e8 36 00 00 00          call   80491c3 <__x86.get_pc_thunk.ax>
 804918d:   05 73 2e 00 00          add    eax,0x2e73
 8049192:   89 ca                   mov    edx,ecx
 8049194:   8b 52 04                mov    edx,DWORD PTR [edx+0x4]
 8049197:   8b 52 04                mov    edx,DWORD PTR [edx+0x4]
 804919a:   89 55 f4                mov    DWORD PTR [ebp-0xc],edx
 804919d:   83 ec 08                sub    esp,0x8
 80491a0:   ff 75 f4                push   DWORD PTR [ebp-0xc]
 80491a3:   8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
 80491a9:   52                      push   edx
 80491aa:   89 c3                   mov    ebx,eax
 80491ac:   e8 9f fe ff ff          call   8049050 <printf@plt>
 80491b1:   83 c4 10                add    esp,0x10
 80491b4:   b8 00 00 00 00          mov    eax,0x0
 80491b9:   8d 65 f8                lea    esp,[ebp-0x8]
 80491bc:   59                      pop    ecx
 80491bd:   5b                      pop    ebx
 80491be:   5d                      pop    ebp
 80491bf:   8d 61 fc                lea    esp,[ecx-0x4]
 80491c2:   c3                      ret
```

Calling a Function

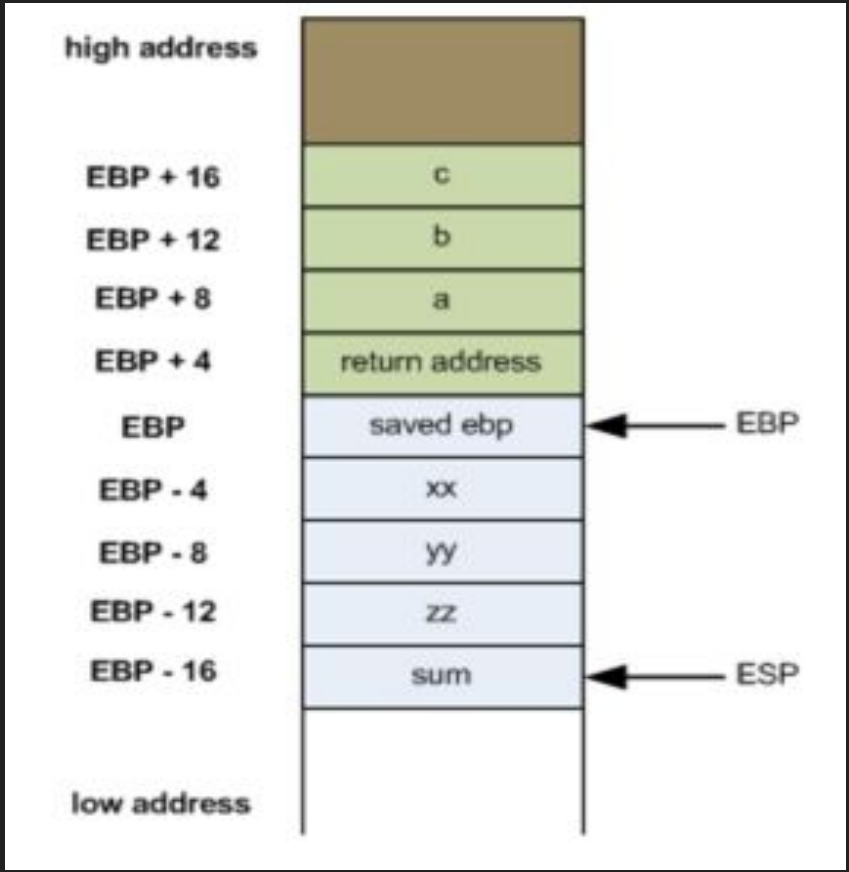
```
08049176 <main>:
 8049176:   8d 4c 24 04             lea    ecx,[esp+0x4]
 804917a:   83 e4 f0                and    esp,0xfffffff0
 804917d:   ff 71 fc                push   DWORD PTR [ecx-0x4]
 8049180:   55                      push   ebp
 8049181:   89 e5                   mov    ebp,esp
 8049183:   53                      push   ebx
 8049184:   51                      push   ecx
 8049185:   83 ec 10                sub    esp,0x10
 8049188:   e8 36 00 00 00          call   80491c3 <__x86.get_pc_thunk.ax>
 804918d:   05 73 2e 00 00          add    eax,0x2e73
 8049192:   89 ca                   mov    edx,ecx
 8049194:   8b 52 04                mov    edx,DWORD PTR [edx+0x4]
 8049197:   8b 52 04                mov    edx,DWORD PTR [edx+0x4]
 804919a:   89 55 f4                mov    DWORD PTR [ebp-0xc],edx
 804919d:   83 ec 08                sub    esp,0x8
 80491a0:   ff 75 f4                push   DWORD PTR [ebp-0xc]
 80491a3:   8d 90 08 e0 ff ff       lea    edx,[eax-0x1ff8]
 80491a9:   52                      push   edx
 80491aa:   89 c3                   mov    ebx,eax
 80491ac:   e8 9f fe ff ff          call   8049050 <printf@plt>
 80491b1:   83 c4 10                add    esp,0x10
 80491b4:   b8 00 00 00 00          mov    eax,0x0
 80491b9:   8d 65 f8                lea    esp,[ebp-0x8]
 80491bc:   59                      pop    ecx
 80491bd:   5b                      pop    ebx
 80491be:   5d                      pop    ebp
 80491bf:   8d 61 fc                lea    esp,[ecx-0x4]
 80491c2:   c3                      ret
```

Setup our Arguments

Call printf

Stack Frame

- Manage function information
- Created with a Function Prologue at start of function
- Cleaned up at the end of a function called a Function Epilogue
- RBP is the base of the stack frame
  - Used to get offsets to our local variables and arguments



Function Prologue

- Save original rbp for later
- Make space for local variables
- Setup new base for stack frame

```
080491c1 <challenge>:
 80491c1:   55                      push   ebp
 80491c2:   89 e5                   mov    ebp,esp
 80491c4:   53                      push   ebx
 80491c5:   83 ec 34                sub    esp,0x34
 80491c8:   e8 03 ff ff ff          call   80490d0 <__x86.get_pc_thunk.bx>
```

Function Epilogue

- Restore saved registers
- Clean up stack
- Return to Callee

```
8049201:   90                    nop
8049202:   90                    nop
8049203:   8b 5d fc              mov    ebx,DWORD PTR [ebp-0x4]
8049206:   c9                    leave
8049207:   c3                    ret
```

What about the buffer overflows?

- All local variables next to each other in the stack frame
  - Including Arrays aka Buffers ;)
- Main Ideas of Buffer Overflows
  - Read past a buffer to corrupt data
- After our buffer is the other variables!

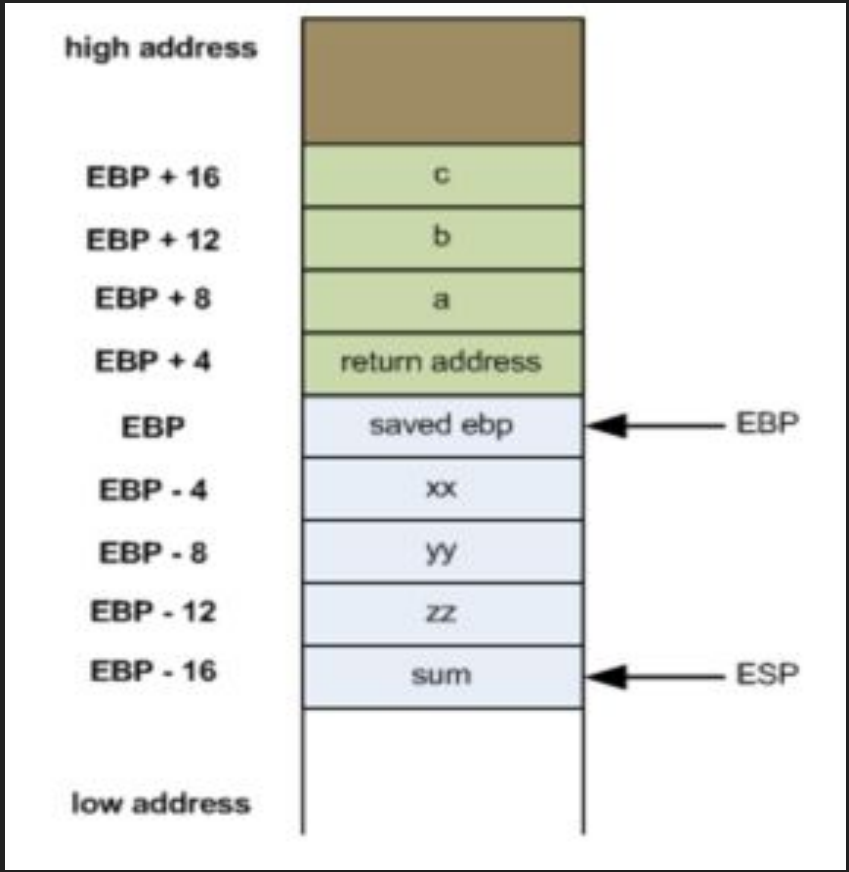
Demo Variable Overwrite

What else can we corrupt?

- How do we return to our Caller Function?
- Where does that information live?
- Can we corrupt it >:)

The Return of the Overflow

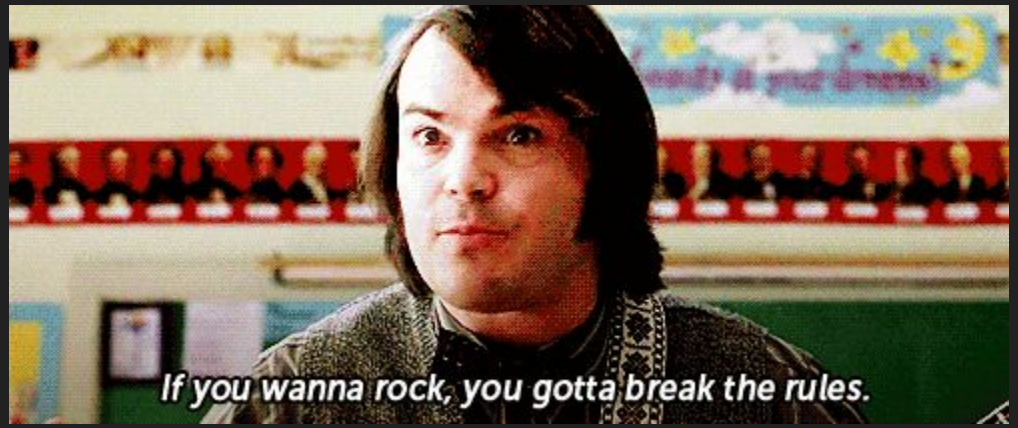
- Steps to the CALL instruction
  - Push the next address to the stack
    - The Return Address
  - Jump to the function we are calling
- Return Steps
  - Jump to whatever is on top of the stack
    - This should be the address we pushed
    - Unless someone corrupted it :)



Demo Return Overflow

Code Exec with Buffer Overflow

- We now control the flow of execution
- Where to go ?
- Why not the stack ?



Shellcode ?

- Shellcode
  - Assembly Code injected into a binary
- Some extra challenges:
  - Character limitations
  - Only working in text segment
  - Size limitations

What we working with

```
.text
_start:
    jmp string

main:
    pop %ebx
    xor %ecx, %ecx
    xor %edx, %edx
    mov %eax, 11
    int 0x80

string:
    call main
.string "/bin/sh\x00"
```

Get a string to data on the stack

What we working with

```
.text
_start:
    jmp string

main:
    pop %ebx
    xor %ecx, %ecx
    xor %edx, %edx
    mov %eax, 11
    int 0x80

string:
    call main
.string "/bin/sh\x00"
```

Setup Arguments to syscall

Set registers to 0

Syscall for execve

Jumping to the Stack

- Not gonna be 100% accurate
- But can get by this using a NOP sled
- NOP Sled:
  - A series of instructions that effectively do nothing
  - Can jump anywhere into the NOP sled
  - Slides you down to your real shellcode

Final Demo