

# Buffer Overflow 101

A Tutorial and Workshop by Miles Crabill

# Flip the Script

- Are you comfortable with C?
- Have you seen hex before?
- Do you know what a stack is?
- Have you heard about “x86” or “assembly”?

# Super Basic C

- loops
- functions
- scoping
- argc & argv
- ascii
- pointers
- strings
- strcpy
- doesn't check memory boundaries

# Hexadecimal Primer

- Base 16 pretend you have sixteen fingers
- Let's count up to 30 in hex!
- 0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11 12 13 14 15  
16 17 18 19 1A 1B 1C ...
- When you see someone say 0x1B they are displaying a number in hex
  - Similarly 0b denotes binary

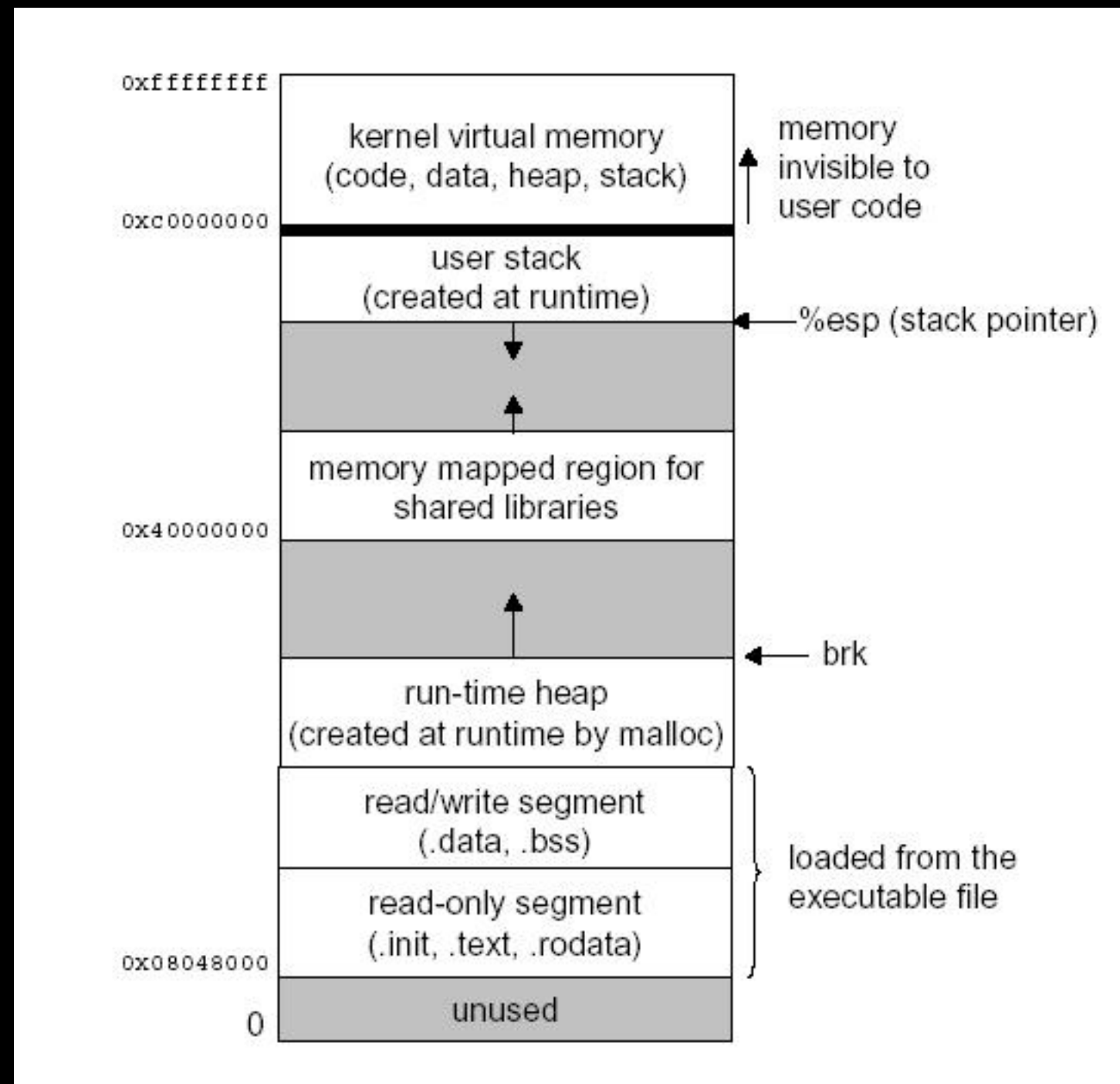
# What is a stack?

- LIFO last in, first out data structure
- Operations:
  - push
  - pop
  - isEmpty



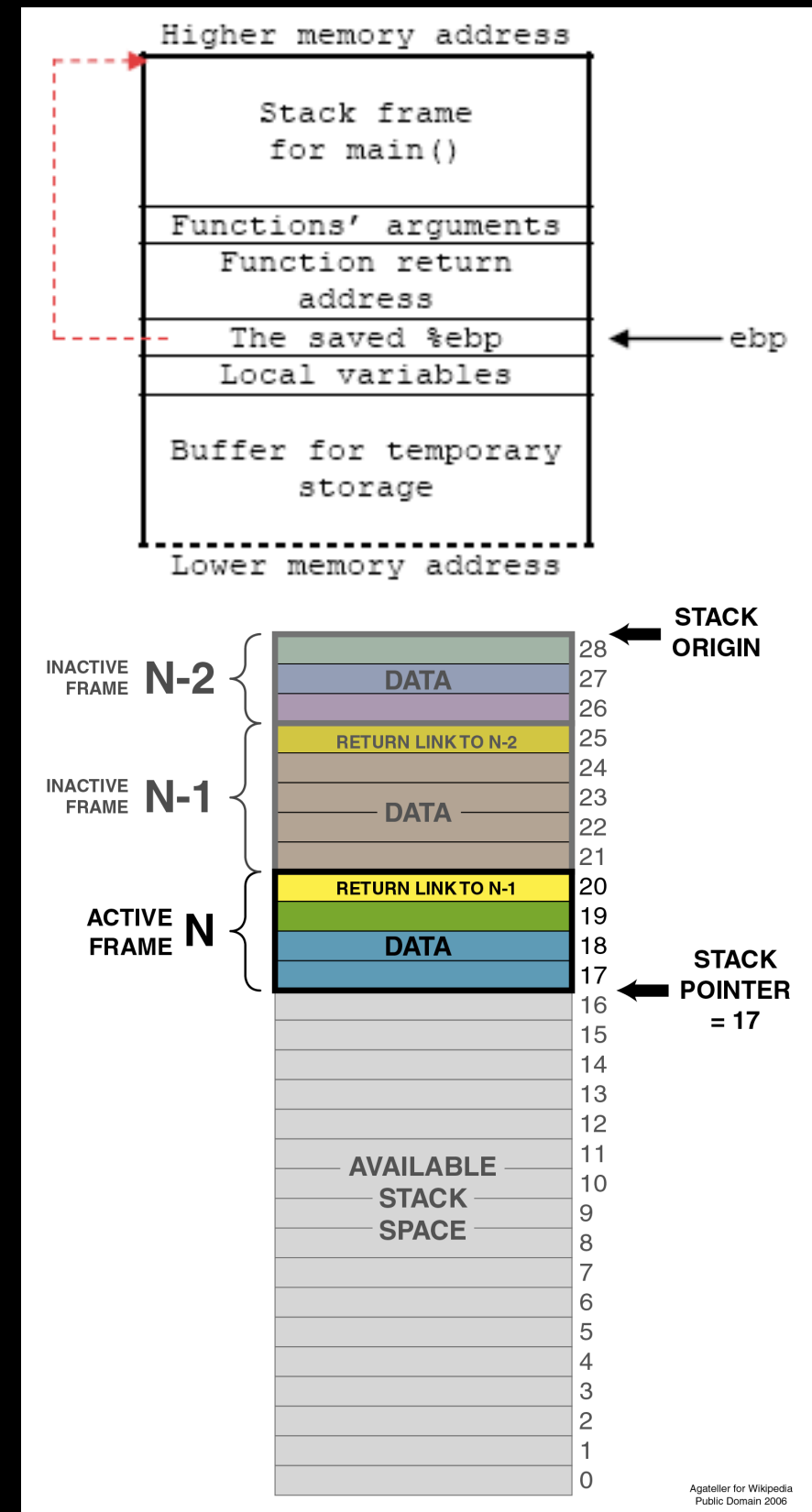
# x86 Stack Implementation

- 0xFFFFFFFF
  - highest memory address
- The stack grows downwards
  - high to low address
- stack pointer
  - register that points to the current top of the stack
- stack base pointer
  - register that points to a fixed location stores the original value of the stack pointer



# Nitty Gritty x86 Stack

- The top image shows the contents of an individual stack frame
- Notice that the saved ebp base pointer points to the start of the stack frame
- Notice that local variables are at the bottom remember that the stack grows downwards!
- The bottom image shows how stack frames work with function calls



# Little Endian Trickiness

- In the x86 architecture, memory is stored in little endian format. In little endian, you store the least significant byte in the smallest address.
  - This means that the order of the bytes in memory is reversed.
- This is important because when you enter memory addresses into your code you will need to account for the change in endianness.
- Example:
  - `\xef\xbe\xad\xde` becomes `\xde\xad\xbe\xef`
    - `efbeadde` becomes `deadbeef`
  - Note: each byte is in hex, that's why there are the `\x`'s



# buf.c demo

## follow along:

```
ssh student@<censored_ip_address>  
the password is student
```

Make a folder for your team please!

# Stack Smashing

- It's all about overwriting the return address
  - We don't want to return to where the program wants to return to, we want to return to our shellcode!
- Components of the buffer we want to send
  - NOP sled
    - a series of “no operation” commands in x86, the opcode for this is 0x90 90 in hex
  - Shellcode
    - the payload of the exploit
  - Return address
    - to overwrite the return address of the vulnerable function with an address somewhere in our NOP sled

# Understanding Assembly

```
0x080484e2 <+0>:  push    ebp
0x080484e3 <+1>:  mov     ebp, esp
0x080484e5 <+3>:  and     esp, 0xfffffffff0
0x080484e8 <+6>:  sub     esp, 0x220
```

- Note: to get the output I have, you will need to run `set disassembly-flavor intel` in gdb
- The hex values on the left side are the addresses of each of the instructions
- The above is the assembly instructions from the main function of stack400 the program we will be exploiting.
- This is the prolog assembly function, saving the stack frame status
- The first line pushes the base pointer to the stack
- The second line sets the base pointer to the function's stack frame stack pointer
- The third line aligns the stack with the next lowest 16 byte boundary
- The fourth line makes room for local variables by subtracting from the stack pointer

# Understanding More Assembly

```
0x08048537 <+85>:    call    0x80484bd <bof>
0x0804853c <+90>:    mov     DWORD PTR esp,0x80485ea
```

- Here I have disassembled main in stack400
  - We see a call to bof - the address 0x80484bd is the location of the bof function
  - Again, the memory addresses on the left are the locations of these instructions within main
  - When we go into a function, its return address will typically be the instruction after it in the function that called it
  - So we can assume that the return address of bof will be 0x0804853c, the instruction after the function call

# 'sploiting

- We're going to connect to a machine I have set up to be exploited
- Run `ssh student@<censored_ip_address>` in a terminal on OSX, type 'Terminal' into Spotlight
  - The password is 'student'
  - Make a folder for your group, and make your `badfile` in there! This is essential to avoid overwriting other groups' files.
  - The programs can be run by typing 'buf' or 'stack' or 'stack400' into your shell
  - The source code is in the `/usr/local/bin/stack-files` directory
    - To navigate to this directory, type `cd /usr/local/bin/stack-files` into your shell

# stack400.c workalong

```
ssh student@<censored_ip_address>
```

the password is student

Make a folder for your team please!

# Runtime Buffer Overflow Prevention

- Canary value stored after local variables
  - Randomized value that is put after local variables on the stack and checked before the function return
- Boundary Checking
  - Ensure that buffers are only written to their limits
- Copy and store return addresses on separate stack
- Library wrappers
- Library patches
- Non-executable, randomized memory
  - Limits ability to write certain functions nesting, etc

# Resources & References

- - [Smashing The Stack For Fun And Profit](#)
- - [Purdue Buffer Overflow Lecture](#)
- - [Wikipedia Page on Stack Based Buffer Overflows](#)
- - [Buffer Overflow Attacks and their Countermeasures @ LinuxJournal](#)
- - [Wikibooks x86 dissassembly and the stack](#)
- - [Naked Security Heartbleed explanation](#)
- - [Chat Wars AOL intentionally put a buffer overflow in AIM to stop Microsoft from using their protocol!](#)
- - [Wired Conficker article](#)
- - [Wikipedia Stack Smashing](#)
- - [Shell Storm Shellcode Database](#)
- - [buf.c taken from Syracuse University](#)
- - [Here](#) is buf.c, the introductory file.
- - [Here](#) is the stack.c file.