Buffer Overflows

•••

Pooja Chivukula

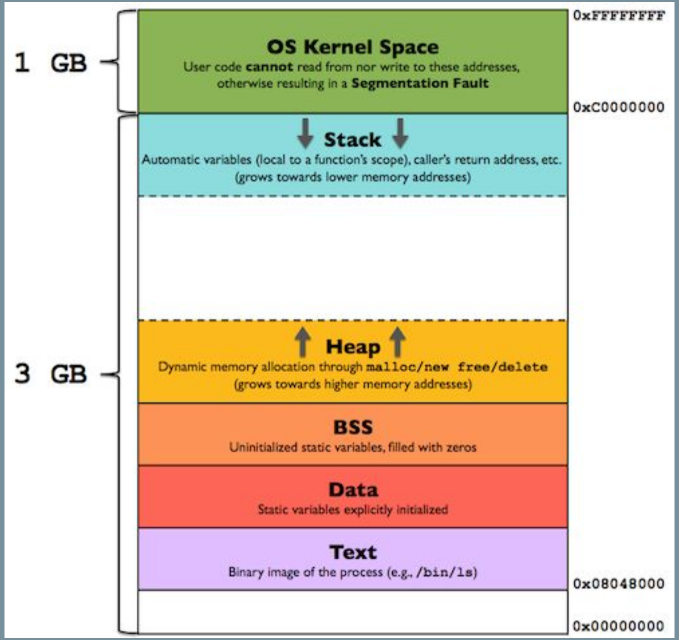
Disclaimers & Credit

- This talk is based off of CS 361s — with Dr. Nielson’s examples/slides
- You will need a Linux System to participate in the exercise

Memory Layout

- The Stack
  - Contains local variables
  - Grows Down (address are decreasing)
- The Heap
  - Contains global/dynamically allocated variables
    - Ex. malloc()
  - Grows upward (addresses are increasing

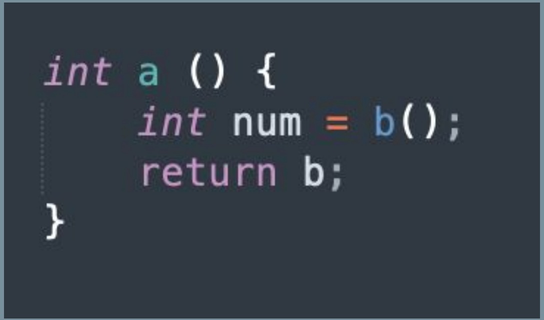
The RAM


1 GB
3 GB
0xFFFFFFFF
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
0xC0000000
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)
0x08048000
0x00000000

***Can do a buffer overflow in stack or the heap, but we will focus on stack based buffer overflows (it is simpler)*

Terminology

- When a function call is made there are 2 functions, the caller and the callee
- **Caller:** The function in which the function call is made
- **Callee:** The function call being made
- In x86, the Caller cleans up the stack after the Callee returns



```
int a () {
    int num = b();
    return b;
}
```

%ebp and %esp

- ➔ ‘r’ stands for 64 bit registers
- ➔ ‘e ‘stands for 32 bit registers

%rbp / %ebp (Stack Base Pointer): Register that maintains the memory address of where the stack was when the function started

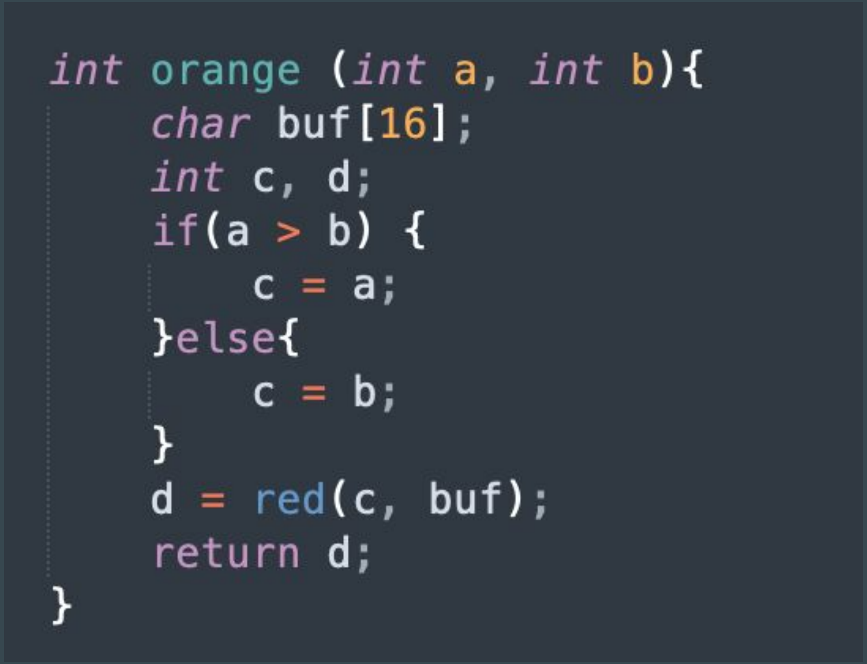
%rsp / %esp (Stack Pointer): Register maintaining the memory address of the current spot in the stack

How is a Function Call Executed?

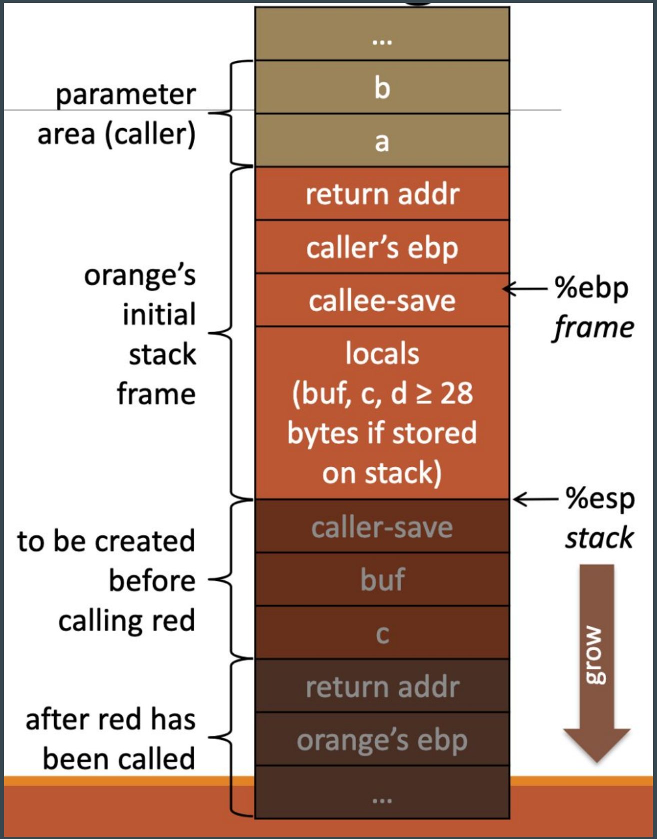
1. Load the parameters for the Callee
2. Jump to memory address of callee
3. Save the registers and variables from the Caller’s function on the stack
4. Load the address of the Caller to return to onto the stack (the return address)

The stack layout for a specific function is known as the **Call Stack**

Function Call Example



```
int orange (int a, int b){
    char buf[16];
    int c, d;
    if(a > b) {
        c = a;
    }else{
        c = b;
    }
    d = red(c, buf);
    return d;
}
```



...
b
a
parameter area (caller)
return addr
caller's ebp
callee-save
locals (buf, c, d ≥ 28 bytes if stored on stack)
orange's initial stack frame
%ebp
frame
%esp
stack
caller-save
buf
c
to be created before calling red
return addr
orange's ebp
...
after red has been called
grow

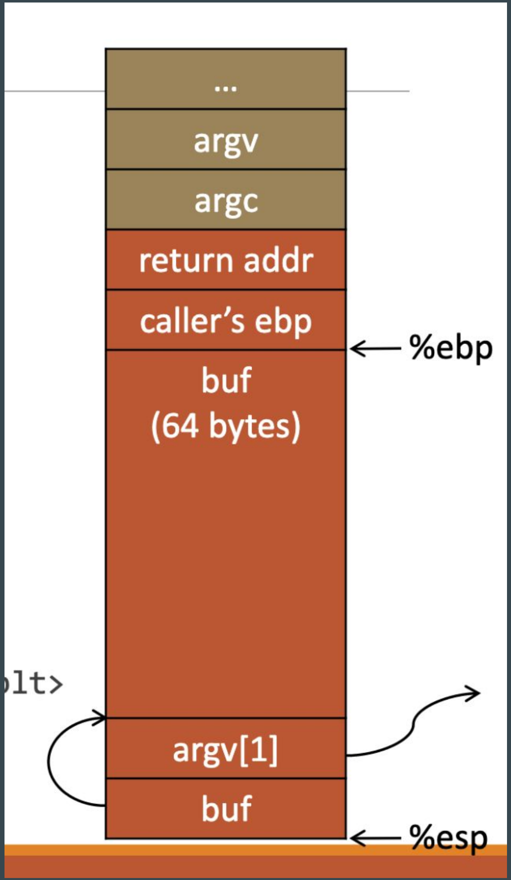
Buffer Overflow

An attack where you are able to write beyond the memory that is allocated to a buffer

Goal : To gain control of the execution, and execute our own code

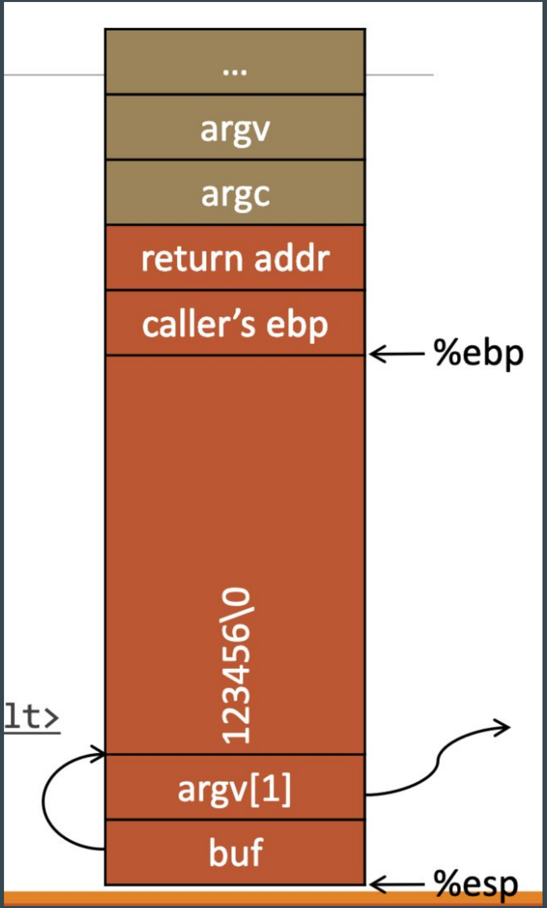
Example: Setup

```
int main(int argc, char **argv){
    char buf[64];
    strcpy(buf, argv[1]);
}
```



Example: Step 1

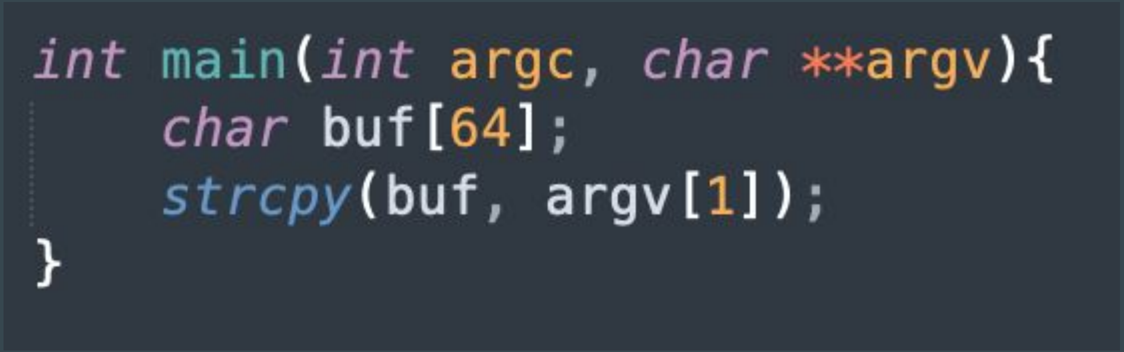
```c
int main(int argc, char **argv){
    char buf[64];
    strcpy(buf, argv[1]);
}
```



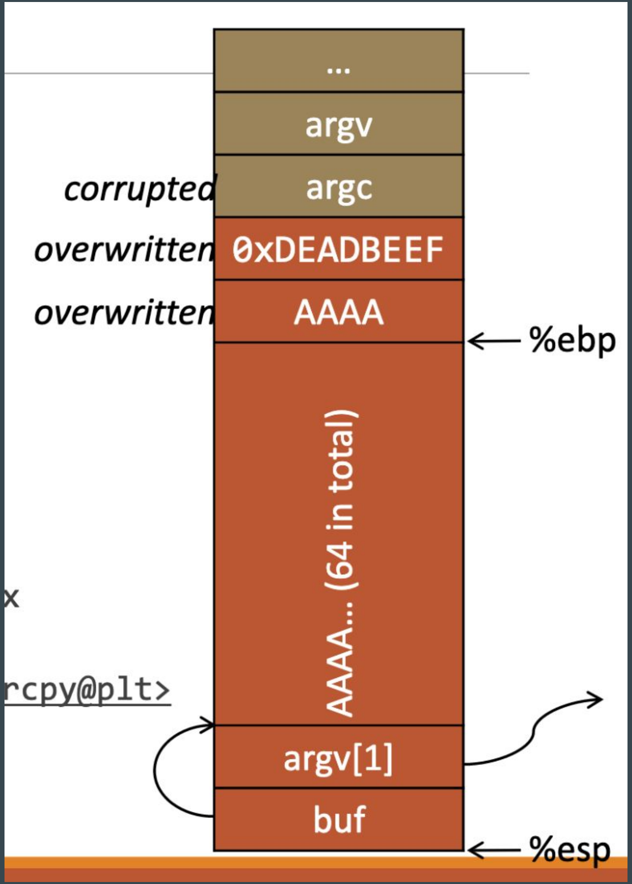
argv[1] = “123456”

This is called the payload

Example: Step 2



```
int main(int argc, char **argv){
    char buf[64];
    strcpy(buf, argv[1]);
}
```

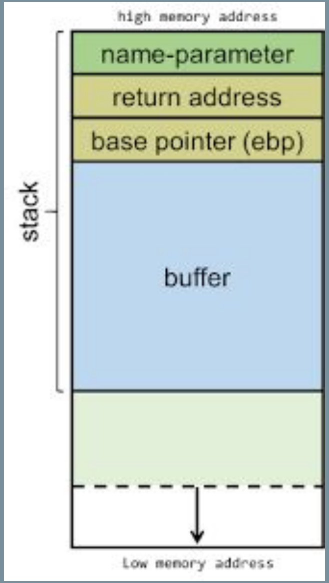


argv[1] = “A” * 68 + “\xEF\xBE\xAD\xDE”

Note Linux uses Little Endian Encoding

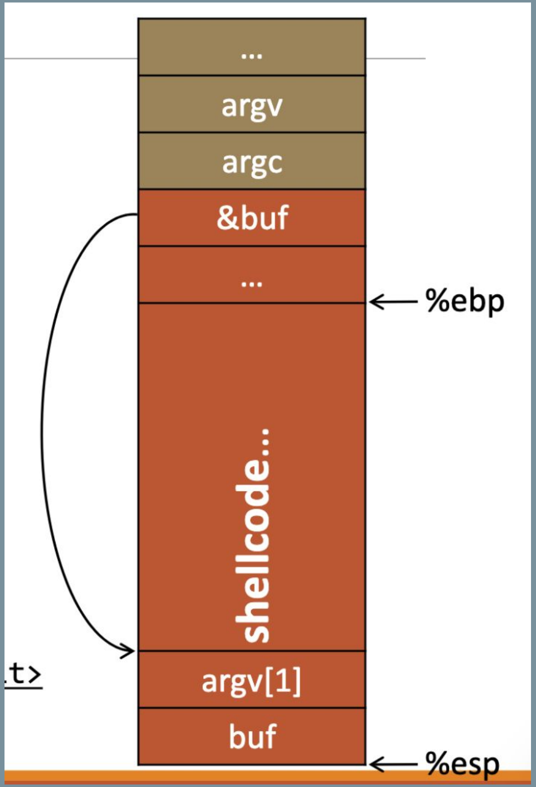
Key Information to execute a Buffer Overflow

1. Where you are in memory (otherwise there is no way to know how far you are from where %ebp is saved)
2. Where %ebp is saved
3. Memory Address of function/code you want to jump to



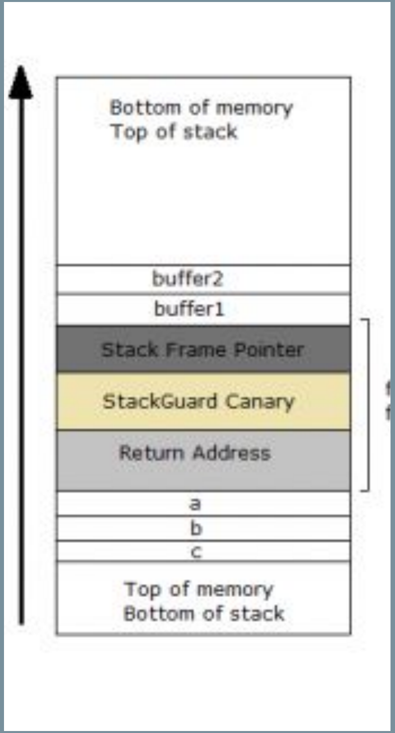
Traditional Buffer Overflow

- Assembly code for exec("/bin/sh") is common
  - Creates a new system shell
- We will learn why we cannot do this on the next slide



Mitigating Buffer Overflows

- Make memory pages writable or executable but not both
- **Stack Canary:** Insert a random value before where %ebp is saved, so before you jump to the return address, you check if the canary value is still the same or if it's been overwritten
  - Need to add a canary for every function that is called



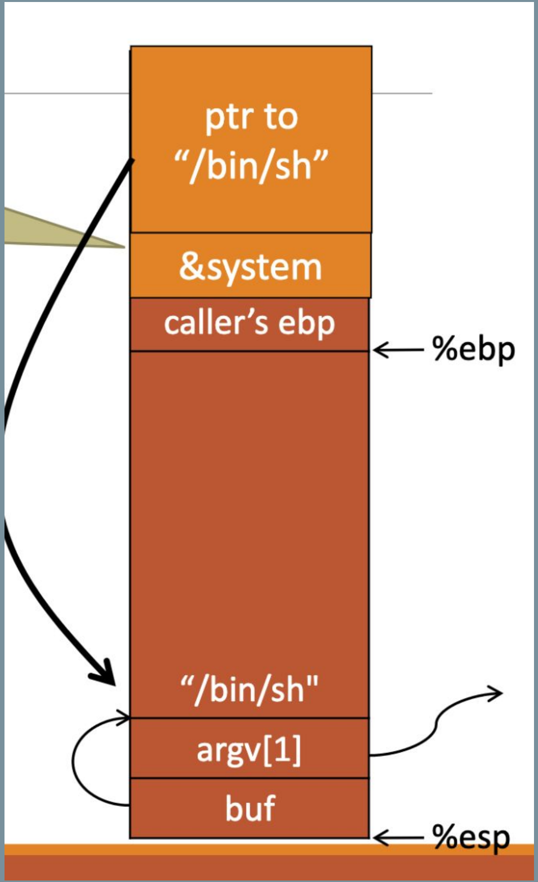
Return Oriented Programming (ROP)

- ROP is a type of buffer overflow, and built on the concepts we just covered
- ROP allows you to bypass write or executable stack page constraint that causes our simple buffer overflow example to fail
- Instead of writing our shell code in the buffer we will piece together our commands from existing standard libraries

Main Idea: Forming shellcode (attack code) from existing code

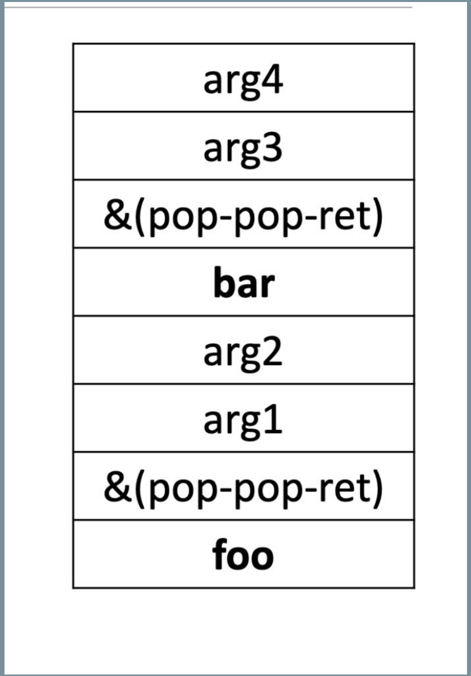
ROP Attack

- Having a system call instruction address in the return address because system code is executable
- Overflows above the return call with the parameters needed for the system call



Key Information to execute a ROP Buffer Overflow

1. A Vulnerability like before
2. Gadgets: common assembly commands (ex. From libraries) that typically end in return and carry out semantic operations (such as add,pop,push,etc.)
    a. Used to set up the parameters needed for function calls, and set up the registers to make function calls
3. Memory Addresses of gadgets should be consistent
    a. NOT RANDOMIZED



Mitigating Buffer Overflows — ASLR / PIE

- **ASLR: Address Space Layout Randomization**
  - Works with virtual memory to randomize different parts of the memory (stack, heap and libraries) every time the program is run
    - Though it is generally boot time based randomization
  - Since it randomizes memory addresses of libraries it works to prevent ROP attacks
- **PIE: Position Independent Executable**
  - Memory Addresses on the stack will change on each run so can no longer craft payloads by trial and error

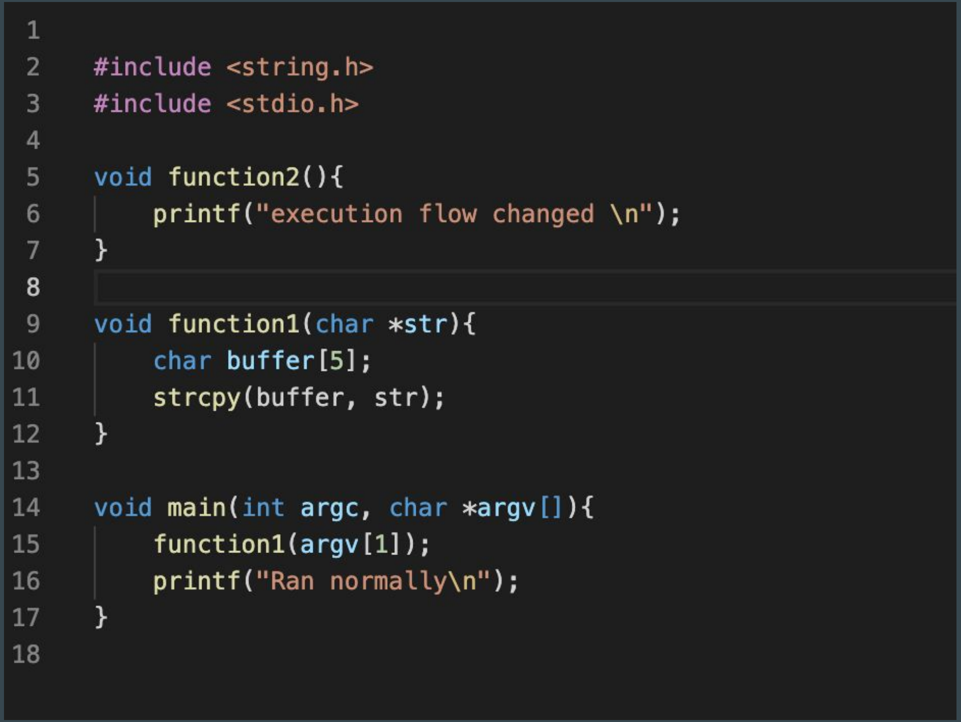
One of the reasons ASLR still gets bypassed though because not all third party libraries are compatible with ASLR and these libraries could be vulnerable

Exercise Time!

Objective: Exploit the buffer in order to run function2() instead of returning to main.

If done correctly this will result in printing “execution flow changed” followed by a segmentation fault

https://github.com/poojachiv/bufferoverflowexercise



```

#include <string.h>
#include <stdio.h>

void function2(){
    printf("execution flow changed \n");
}

void function1(char *str){
    char buffer[5];
    strcpy(buffer, str);
}

void main(int argc, char *argv[]){
    function1(argv[1]);
    printf("Ran normally\n");
}

```

Exercise Hints

- System is 64 bit based
- Linux Machines are Little Endian

Memory Units Review:

→ 4 bits in a hex character
→ 1 byte is 2 hex characters
→ 1 character is 1 byte (AKA 2 hex characters)

Useful GDB Commands:

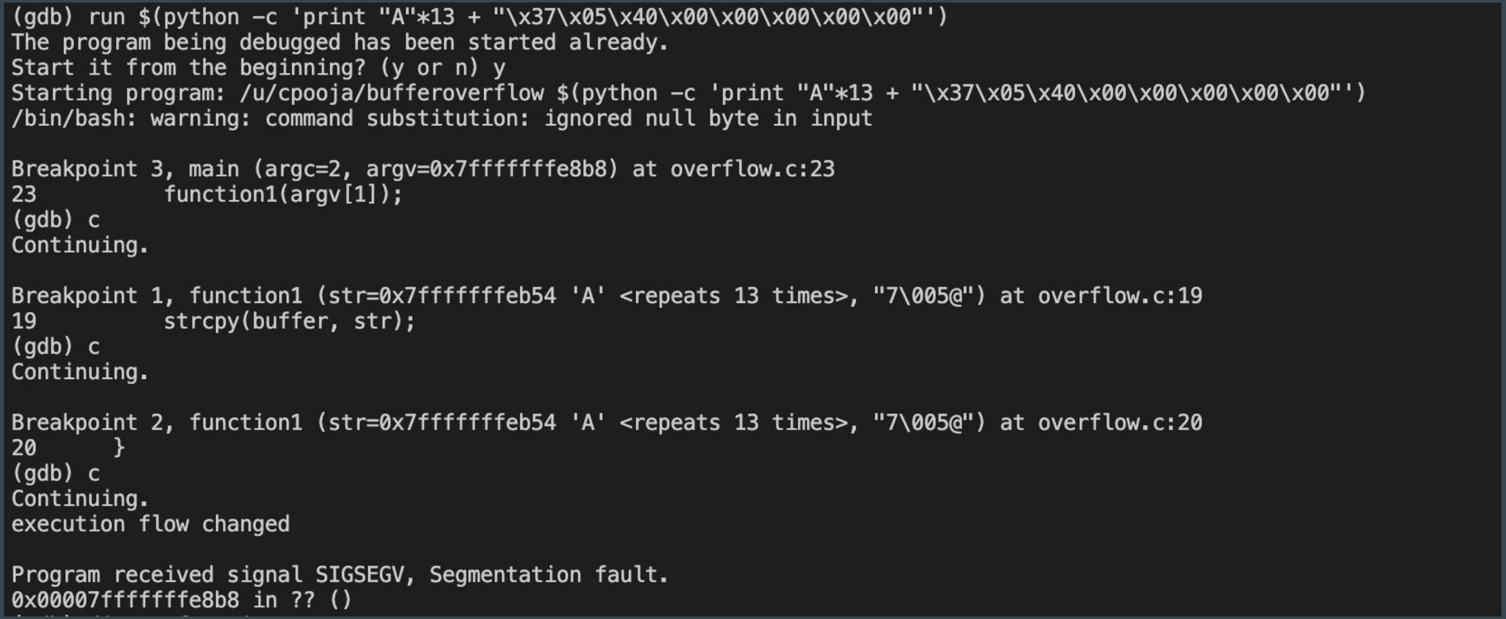
Info regs — shows contents of all registers
b <line number> — adds a breakpoint at that line number of the program
c — will step through the program until next breakpoint
disass <function name> - returns the disassembly of a function
x/<number of bytes>x $<register> — displays <number of bytes> worth of bytes after the memory address is <register> in hex format

Github link: https://github.com/poojachiv/bufferoverflowexercise

Solution

- Put breakpoints
- Check where %rbp is in relation to the buffer
- Add padding of 13 characters because (4 * 3) + 1
  - The one comes from alignment
- Need to figure out address of function 2
  - Objdump function2
    - Add this to the payload in little endian notation
- Final Payload for me was :
  - `run $(python -c 'print "A"*13 + "\x37\x05\x40\x00\x00\x00\x00\x00"')`
  - Your memory address will probably be different

This is what the successful buffer overflow will look like



```
(gdb) run $(python -c 'print "A"*13 + "\x37\x05\x40\x00\x00\x00\x00\x00"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /u/cpooja/bufferoverflow $(python -c 'print "A"*13 + "\x37\x05\x40\x00\x00\x00\x00\x00"')
/bin/bash: warning: command substitution: ignored null byte in input

Breakpoint 3, main (argc=2, argv=0x7fffffffe8b8) at overflow.c:23
23          function1(argv[1]);
(gdb) c
Continuing.

Breakpoint 1, function1 (str=0x7fffffffeb54 'A' <repeats 13 times>, "7\005@") at overflow.c:19
19          strcpy(buffer, str);
(gdb) c
Continuing.

Breakpoint 2, function1 (str=0x7fffffffeb54 'A' <repeats 13 times>, "7\005@") at overflow.c:20
20      }
(gdb) c
Continuing.
execution flow changed

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffe8b8 in ?? ()
```

Resources

- https://payatu.com/understanding-stack-based-buffer-overflow
- https://inst.eecs.berkeley.edu/~cs161/sp16/slides/buffer%20overrun.pdf