Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2015, 2(7): 33-37

Research Article ISSN: 2394 - 658X

Stack Overflow and Program Control

Biswajit Sarma and Srishti Dasgupta

Department of Computer Science and Engineering, Jorhat Engineering College, Jorhat, Asaam, India
eduneristbiswa@gmail.com

ABSTRACT

A buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. A buffer overflow or buffer overrun occurs when more data is put into a fixed-length buffer than that the buffer can handle. Adjacent memory space becomes overwritten and corrupted. In computer security and programming, a buffer overflow, or buffer overrun, is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory. This is a special case of violation of memory safety. A buffer overflow is a common software coding mistake. This paper discusses certain issues about buffer overflow from the very basic and its details in the system.

Key words: Buffer, computer security, memory, stack overflow

INTRODUCTION

A process is program in execution. There are four parts or sections in a process. These are

- Code or text section: Which holds the executable code of the program
- Data section: Which holds all the global variables
- Stack section: The stack is the main important control part in the process. It grows when a function is called and shrinks when the function finishes its execution. When it grows it holds the return address and local variables.
- Heap section: This part of the memory is used to allocate dynamic memory for the process.

The heap and the stack sections grow in opposite direction. The stack grows from high memory to the low memory. We will concentrate only on the stack here. Before we discuss the details we want to mention some important points:

a) A general-purpose register to take note of is the extended stack pointer register (ESP) or simply the stack pointer. ESP points to the memory address where the next stack operation will take place. The ESP register will point to the top of the stack. Here is an example for ESP pointer:
Suppose we have two instructions in assembly language:
PUSH 1
PUSH varA
Here 1000h is memory address and var1 is a variable.
1. after execution of the instruction PUSH 1
2. after execution of the instruction PUSH ADDR var1

b) Another relevant register to the stack is EBP. The EBP register is usually used to calculate an address relative to another address sometimes called a frame pointer. Although it can be used as a general-purpose register, EBP has historically been used for working with the stack.
The stack: Let us take an example program and see how the stack grows. Here is a C program:

```
void test()
{
int A;
A=10;
}
int main()
{
```

```
int x=100;
test();
x=x+100;
}
```

This is a simple C program where a function is called (test()) from the main function. In the test function an integer variable is declared and one initialization is done and an executable statement is there. This C program stack is shown below step by step:

I) The starting point of main before the test function which is called the stack looks like:

II) When the function test is called the stack looks like:

The stack works according to a LIFO model (Last in First Out). Since the spaces within the stack are allocated for the lifetime of a function, only data that is active during this lifetime can reside there.

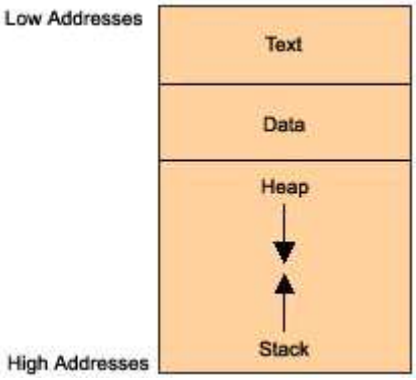

Fig. 1

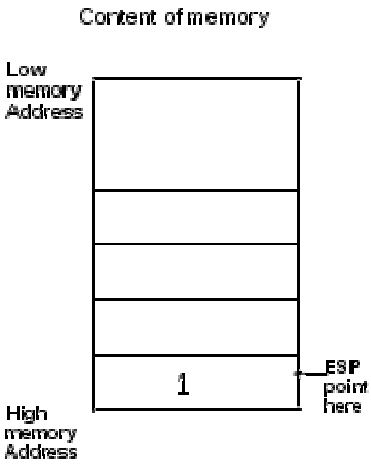

Fig. 2

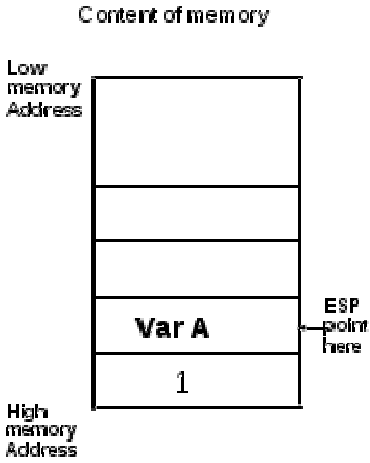

Fig. 3

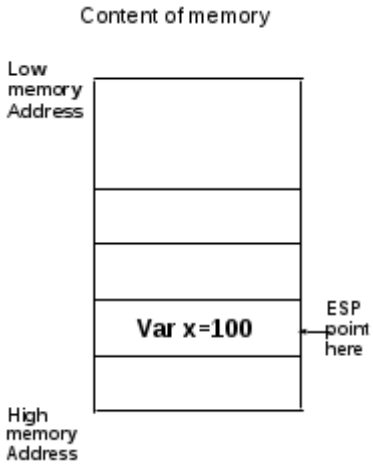

Fig. 4

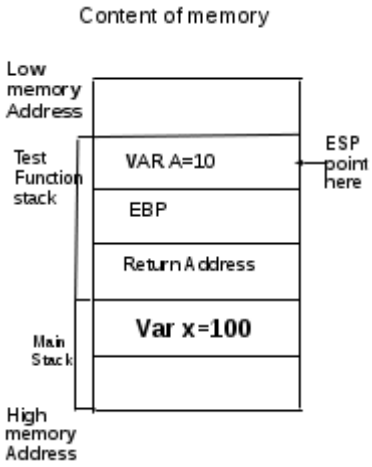

Fig. 5

VULNERABLE POINT IN THE STACK OF A PROCESS

Now we have a solid understanding of what happens when a function is called and how it interacts with the stack. In this section, we are going to see what happens when we stuff too much data into a buffer. We modify the above program and use a buffer in the test function. The reason why we use a buffer is that array bound checking is not done by the system. Here is a simple program which uses the buffer (a [1]).

```
void test()
{
        har A[1];
        A[0]='a';
}
int main()
{
        int x=100;
        test();
        x=x+100;
```

```
}
```

If we compile and run the program we do not have any problem. But here is another program which does something else.

```
void never_return()
{
while(1)
{
}
}
void call()
{
char A[1];
ar[5]= never_return;
}
int main()
{
int x=100;
call();
}
```

If we run the above program it does something abnormal, which never returns back to the terminal.

```
[biswa@localhost buffer]$ ./a.out
```

Now let us try to find out the reason of why this abnormality happens. We have to go to the stack of the system and try to find the reason. When the instruction ar[5]=never_return executes, something abnormal happens. In our code we include a function never_return which basically includes a forever loop doing nothing. But most important thing is that this function is not called properly in the whole C program. The name of the function is basically used in that particular instruction. Here is the diagram of stack for the above program.

Before the execution of the instruction a[5]=never_return; the stack looks like above.

After the execution of the instruction a[5]=never_return the stack looks like above. The most important thing here is that when the function call is made from main the return address is pushed into the stack which is later used by the system to return back to main function. Since the array bound is not checked by the system we write a statement in the function call trying to over write the return address by some other address. Basically that address is our function address of function never_return. So when the system will try to return back from the function call to main, it will use the return address already stored in the stack. But here it will find the address of the function never_return and the control automatically goes to the function never_return and starts execution there instead of going to the main function.

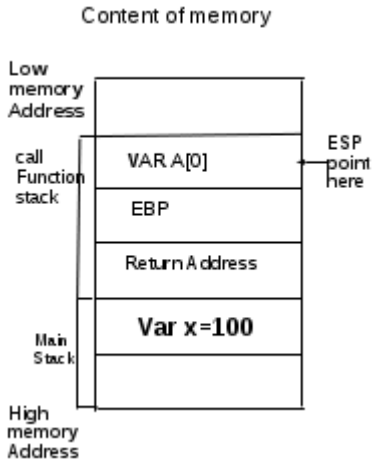

Fig. 6

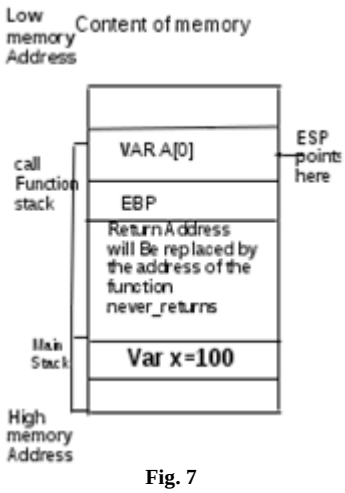

Fig. 7

Finding the Address of a Process and Using Vulnerability of the Process Stack and Changing the Transfer Control

So now we have a little idea of system stack and the return address. Now we will try to use this vulnerability to change the execution sequence. But before we proceed we have to find out the address of the function in the system. For this we need to know little bit about gdb debugger. We use the same C program and compile using -g option.

[biswa@localhost buffer]$ gcc -g test.c
[biswa@localhost buffer]$ gdb a.out
(gdb) disassemble never_return
0x08048400 <+0>: push %ebp
0x08048401 <+1>: mov %esp,%ebp
0x08048403 <+3>: jmp 0x8048403 <never_return+3>
End of assembler dump.
Dump of assembler code for function never_return:
(gdb) quit

Here we try to use gdb and disassemble never_return which gives us the starting address along with the addresses for all instructions for the function never_return. Now we modify the above C program and use this information in that C program. Here is the code:

```
void never_return()
{
while(1)
{
}
}
void call()
{
char A[1];
A[5]=0x08048400; //HERE we modify
}
int main()
{
int x=100;
call();
}
```

If we notice the above code now we see that we are using the address 0x08048400 instead of the function name (never_return). If we try to execute we will find the same result the terminal never return back. So these things work interchangeably. The stack looks like:

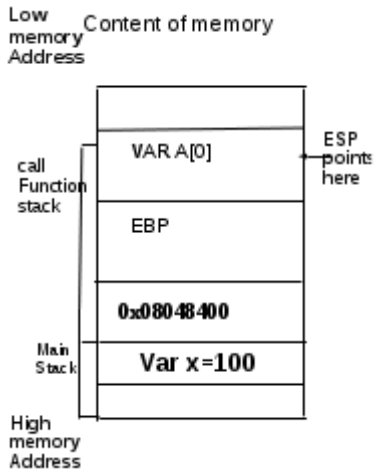

Fig. 8

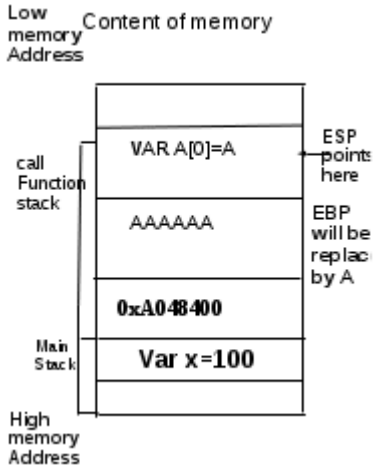

Fig. 9

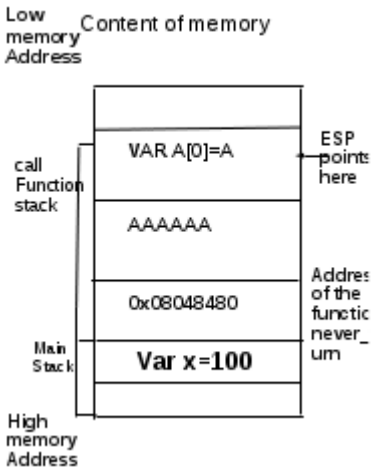

Fig. 10

Giving the input from user side and running it:

Now we have a clear idea about the stack overflow and change of execution sequence. But the most important thing is that whatever we did, basically we wrote the code itself in the program. Now we will try to give input from user side and to implement the same thing. Here is the code:

```
#include<stdio.h>
void never_return()
{
while(1)
{
}
```

```
}
void call()
{
char ar[1];
scanf("%s",ar);
printf("\n the array=%s",ar);
}
int main(){
call();
}
```

We use the scanf() function to fill up the string and put it in the character array A and then try to print the string. We compile the above program and use gdb debugger to find out the address of the function never_return (which is same as before 0x08048400). Then we try to run the program.

```
[biswa@localhost research]$ printf "A" | ./a.out
the array=A
[biswa@localhost research]$
```

It is running perfectly well. Then we try again

```
[biswa@localhost research]$ printf "AAAAAA" | ./a.out
the array=AAAAAA
[biswa@localhost research]$
```

That means the string can hold more than one character. We try again.

```
[biswa@localhost research]$ printf "AAAAAAAAAAAA" | ./a.out
Segmentation fault (core dumped)
[biswa@localhost research]$
```

We got a segmentation fault. But why? We again go back to the system stack and try to find out. Here is the system stack for the above input.

If we notice this carefully the return address 0x8048400 is over written by a A character there and the return address changes to 0xA048400. The system found that this is not a legal address and gave user a segmentation fault. We try again with the following input; at the last we put the address which we want to change the execution sequence. The stack looks like Fig. 9.

```
[biswa@localhost research]$ printf "AAAAAAAAAAAAA\x80\x84\x04\x08"| ./a.out
printf "AAAAAAAAAAAAA
hello
asdf
mnpo
```

It is taking the input forever. This means it is executing the function never_return(), which never returns back (Fig. 10).

CONCLUSION

We have studied the system stack in relation to the function call in a process, how the stack grows when the function is called and shrinks when the function finishes its work and returns back to the called function. The main intention of this paper is to find the vulnerable points in the stack and discuss the theory behind this. This information can be used in further research work.

REFERENCES

[1] Sobolewski Piotr, Over Flowing the Stack on Linux x86, *Hak in 9 Magazine*, **2004**, 4.
[2] Carbonneaux Quentin, Hoffmann Jan, Ramananandro Tahina, Shao Zhong, End-*to-End Verification of Stack-Space Bounds for C Programs*, Yale University, PLDI '14, Edinburgh, United Kingdom, **2014**.
[3] Anley Chris, Heasman John, Linder Felix, Richarte Gerardo, The Shellcoder's Handbook, (Discovering and Exploiting Security Holes), Second Edition, Wiley Publishing Inc, **2007**.