Buffer Overflow Explanation

Written by Dr-Freak

June 16, 2012

For this Article readers should have a simple understanding of C programming language the way stack is organize and little assembly knowledge is going to helpful for the reader to take maximum benefit from this article. In this article I am going to talk about stack based over flow, there is difference between stack based and heap based over flow.

Before we go further in this article let me explain some simple words terminology which I am going to use in this article.

ASM: Abbreviation for assembly Language, which is a second generation programming language.

Register: This is used by your processer to hold information and control execution.

EIP: This is the instruction pointer which is a register (32 bit), it points to your next command which is going to be execute after executing the previous cycle of commands. Simply this register tells the CPU which instruction is going to be executing after each execution of command. It holds the address of next instruction.

EBP: EBP is the base pointer, it points to the top of the stack, and when a function is called it is pushed, and popped on return.

OllyDbg: It is a debugger which helps you to study the flow of execution of your program. There are many debugger you can use any you want (Immunity debugger, IDA etc). In this article I am going to use OllyDbg.

Bloodshed Dev-C++: A C/C++ Compiler.

little endian: It is how memory addresses are stored on most systems, little bytes first.

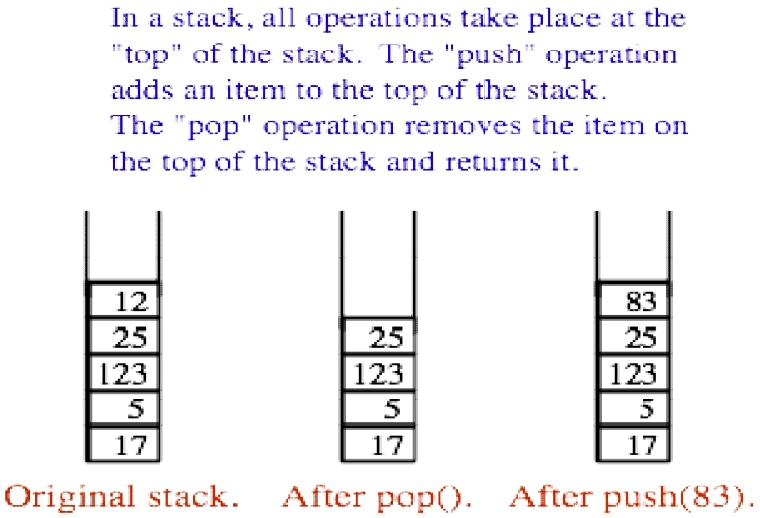
SHORT ABOUT BUFFER OVERFLOWS

Buffer overflows are a common vulnerability on all platforms, but are by far the most commonly exploited bug on the Linux/Unix Operating systems. Buffer over flow occurs when you try to insert data into consecutive memory addresses more than its capacity of storage.

Commonly buffer overflows are exploited to change the flow in a programs execution, so that it points to a different memory address or overwrites crucial memory segments. If you

know how memory is organized, you would know that on all x86 Linux platforms, memory is organized in 4byte (32 bit) segments, consisting of a hex memory address, and will need to be converted to little endian byte ordering. Stack is consist of consecutive memory addresses which follows last in first out terminology (LIFO) ,which means the data comes first in stack will go out from stack in last. The stack and EIP is the most important part of the buffer over flow vulnerabilities which you have to take care off in exploiting.

Here is a simple diagram showing how stack looks like



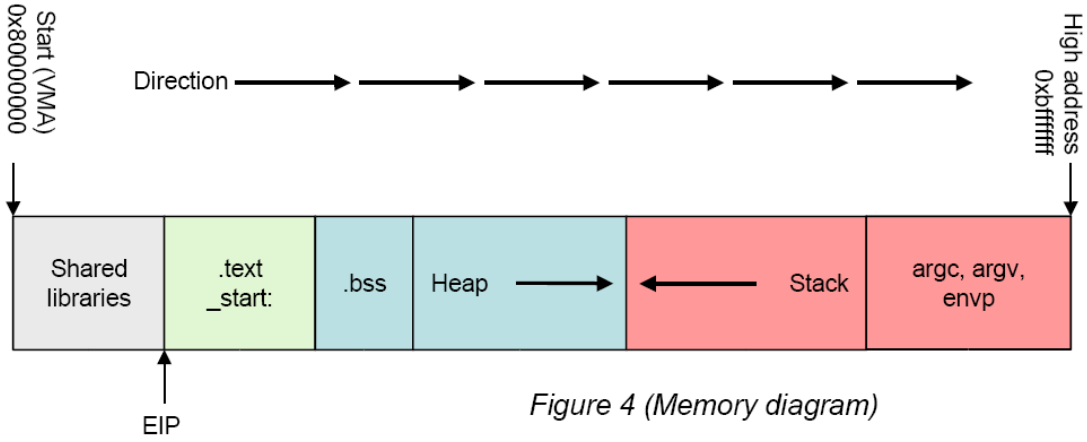


Figure 4 (Memory diagram)

The diagram above illustrates a typical program's memory layout when it has been loaded

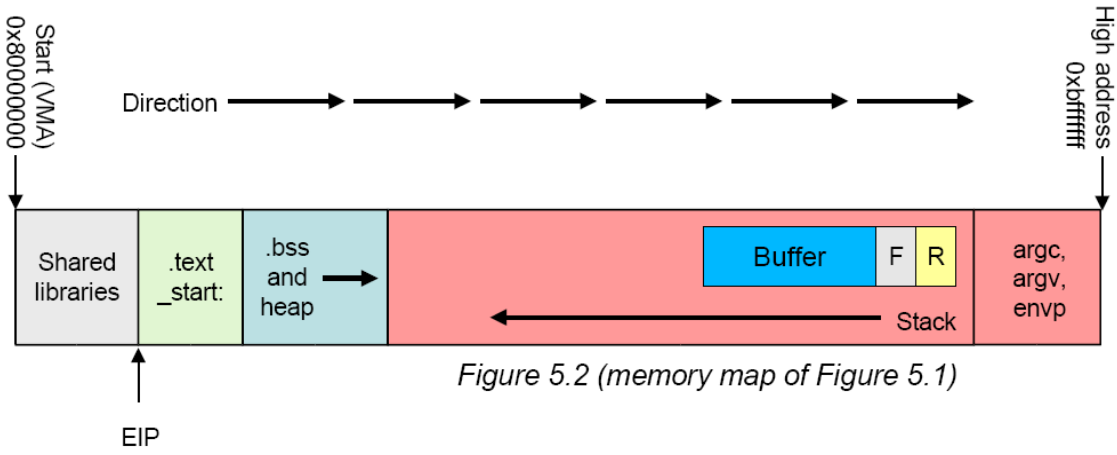


Figure 5.2 (memory map of Figure 5.1)

As you see in stack there is a buffer after which it has a Frame pointer and after that it has the return address this is called EIP and in buffer over flow we are considering to change this pointer value so that we can change the flow of execution of the program.

Now after you have an idea about buffer, stack and EIP here is our vulnerable C program.

Vulnerable C Code

Vuln.exe

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

```
int vulnFunction(char *str){
        char buffer[10]; //our buffer
        strcpy(buffer,str); //the vulnerable command
        return 0;
        }

int main(int argc, char *argv[])
{
    char code[]="AAAAA";
    printf("You are in main fucntion now\n");

    vulnFunction(code); //call the vulnerable function
    checking(); //this should never happen
    printf("Quitting vuln.exe\n");
    getch();
    return 0;
}
int checking(){
        printf("*******  You have done it!  *******\n");

        printf("*******  This is checking() executing  *******\n");
}
```

This is a simple C program which has one array that contains some string and later in the program this string is pass to vulnerable function which is vulnFunction(char *str).

The vulnFunction has an buffer of 10 bytes.1 char is equal to1 byte and in program we have char buffer[10] ,so size is equal to 10 bytes.Moreover the vuln function copies the content of variable code into buffer without checking the size of variable code,and this is the vulnerability.

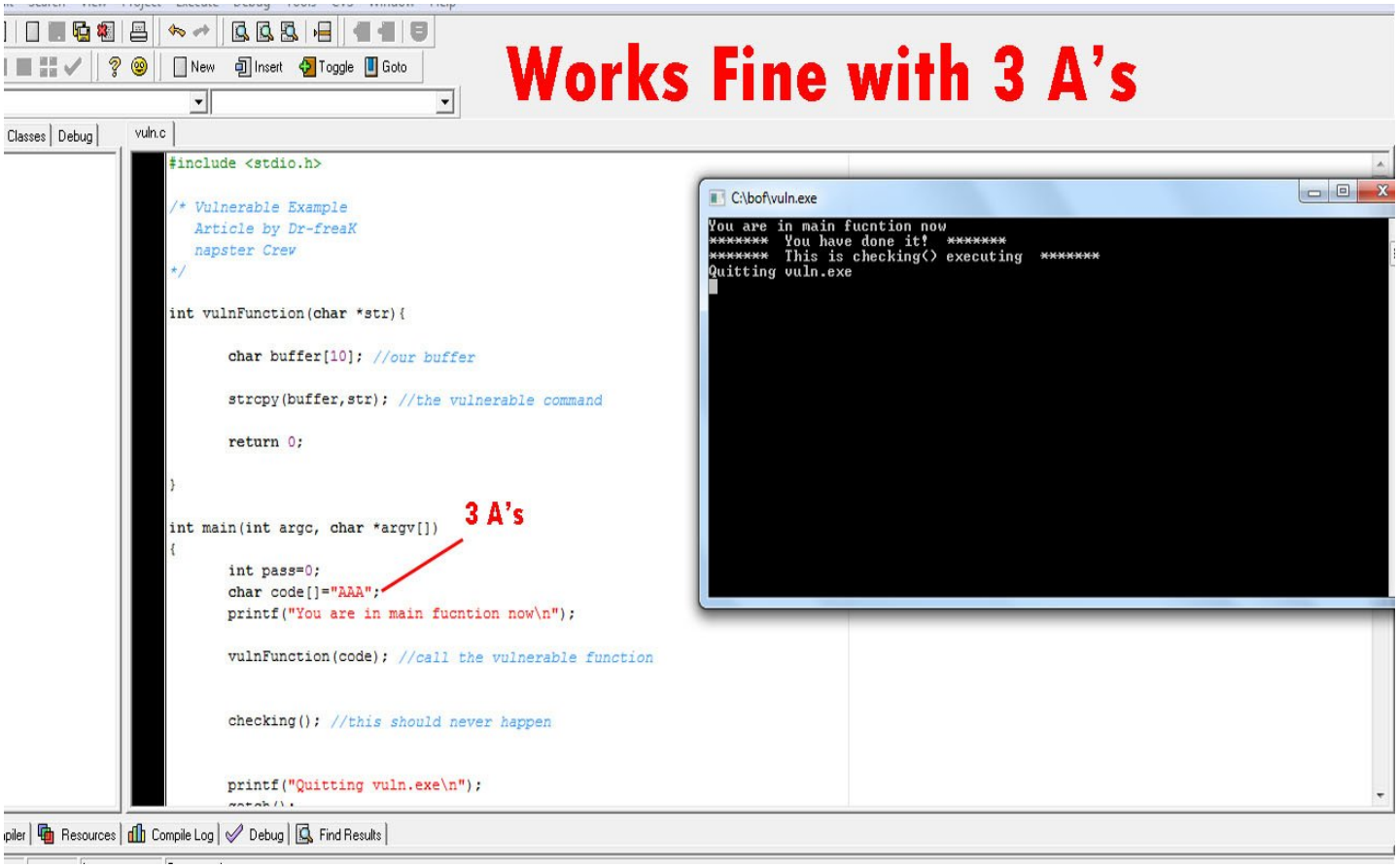
```
strcpy(buffer,str); //the vulnerable command
```

In this article we will going keep changing the value of array code (cha code[]=”AAAa”;) and testing the program.

So first let us check the program by simply taking the value of array code to 3 A’s

char code[]=”AAA”;

Change the value of array code to 3 A’s and Compile the program using DevC++ by simply going to Execute tab and clicking Compile and run. After doing this you will see it works alright.



Now we will confirm the vulnerability of this program by changing the value of char code to about 30 A’s.

Char code[]=”AAAAAAAAAAAAAAAAAAAAAAAAAAAA”;

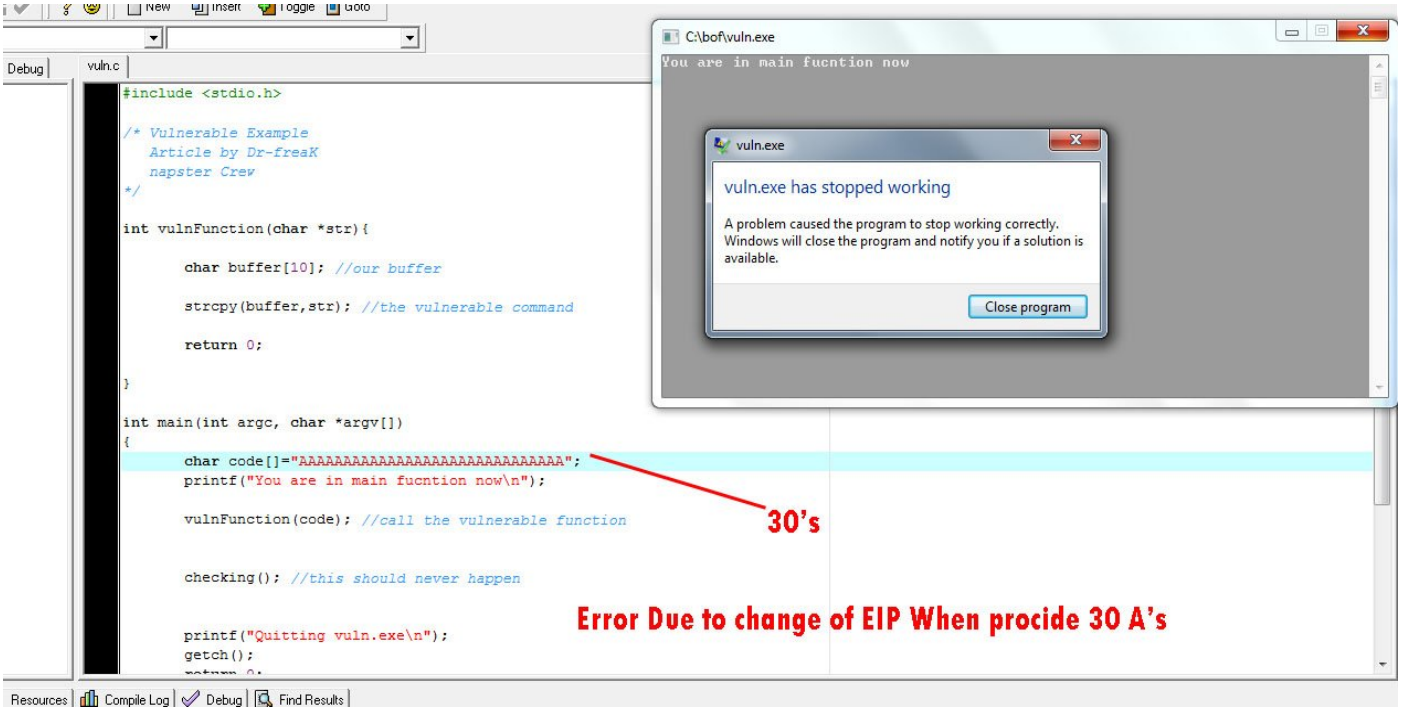
As you know our buffer is capable of to fill maximum of 10 characters and if we fill it with 30 A’s then it will cause our program to crash as when 30 A’s will pass to

vulnFunction ,this function will try to fill the buffer with it and our program will crash because our buffer has the capacity to fill with 10 characters and due to this value of EIP will be change. Let us check now.

Change the value of char code to 30 A's in the original program.

Char code[]="AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";

What you get an error segmentation fault.



This happens because when the buffer is filling with more size than it is capable of, now remember that in stack after it has frame pointer and after that it has EIP pointer. So 30 A's first fill the stack then they fill the frame pointer and then finally the EIP value,which has the return address so when return address is overwrite by us the program crash.

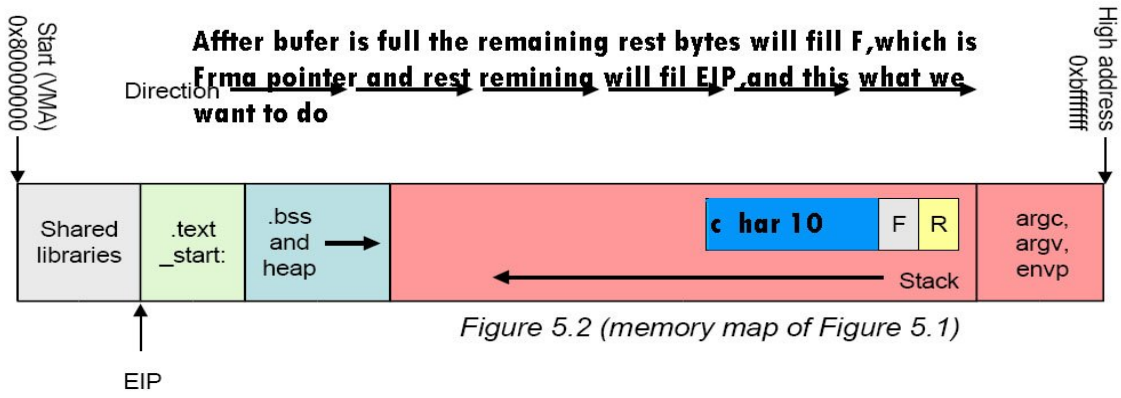


Figure 5.2 (memory map of Figure 5.1)

Now it's time to open OllyDbg and debug the program.

Open OllyDbg and open the vuln.exe in it.

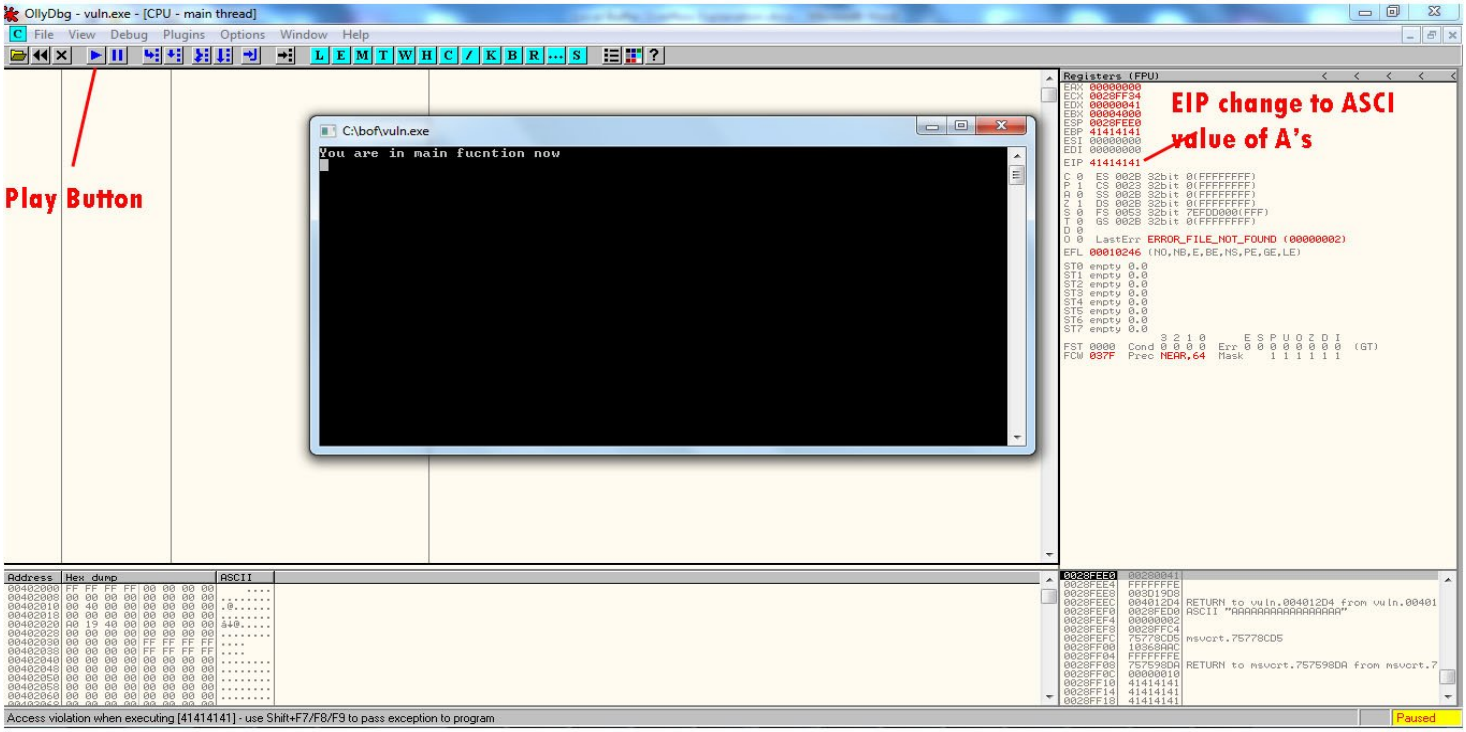
Open OllyDbg

File tabs →Open and select your vuln.exe

Debug tab→run

Then click play button

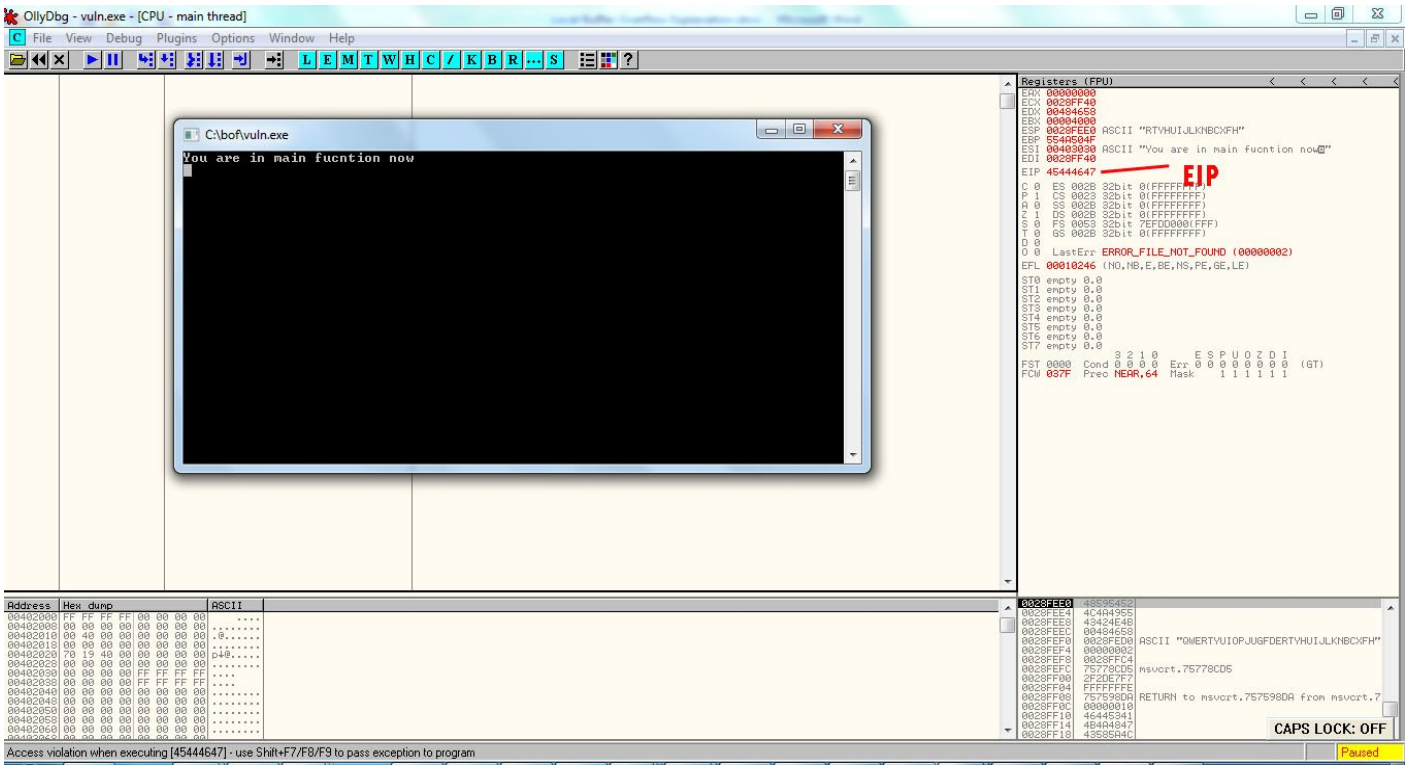
You will get this



THE EIP has value 41414141 which is ASCII value of A and not any physical address.

What is happening due to over flow of buffer our EIP is overwritten by what we provide via char code. The value of EIP is overwritten by ASCII value of A which is 42, and this can be seen in OllyDbg showing EIP content to 41414141 (ASCII A = 41).
Now when we use 30 A's the program will never execute after vulnFunction is call and check function will never be execute as program crash in vulnFucntion. As we successfully change the content of EIP with ASCII value of A, we can also do that change the EIP with some original address where our shell code is save. But here now we will change the EIP with the check function address so that our program run fine regardless of buffer is overflow.

But before we find the address of our check function, we must have to know after how many bytes our EIP is change, so that we will provide some junk data + original address via char code. To do that set the value char code to some random long characters doesn't write any character consecutive.
I will set value something this
ASDFGHJKLZXCVBNMQWERTYUIOPJUGFDERTYHUIJLKNBCXFH
Change in original code value of char code
Char
code[]="ASDFGHJKLZXCVBNMQWERTYUIOPJUGFDERTYHUIJLKNBCXFH";
Compile the program with DecC++ and open the vuln.exe in OllyDbg with same procedure define aboveWhat You will get something like this



Now our EIP is 45444647 now we have to convert this to character readable forum

So first convert this to Little Indian by simply 2 digit together and than putting last couple first and so on.

EIP 45444647 → 45 44 46 47

Little Indian → 47 46 44 45→ 47464445

Now convert it to hex to character

You will get GFDE

So EIP→45444647→ GFDE

Now search GFDE in that string which we provide via char code.

code[]=”ASDFGHJKLZXCVBNMQWERTYUIOPJUGFDERTYHUIJLKNBCXFH”;

Look closely ASDFGHJKLZXCVBNMQWERTYUIOPJU***GFDE***RTYHUIJLKNBCXF

All the data befor GFDE is of 28 byte so where we know now after how many bytes we can overwrite EIP. To conferm this you can change GFDE to CCCC in that string we provided.

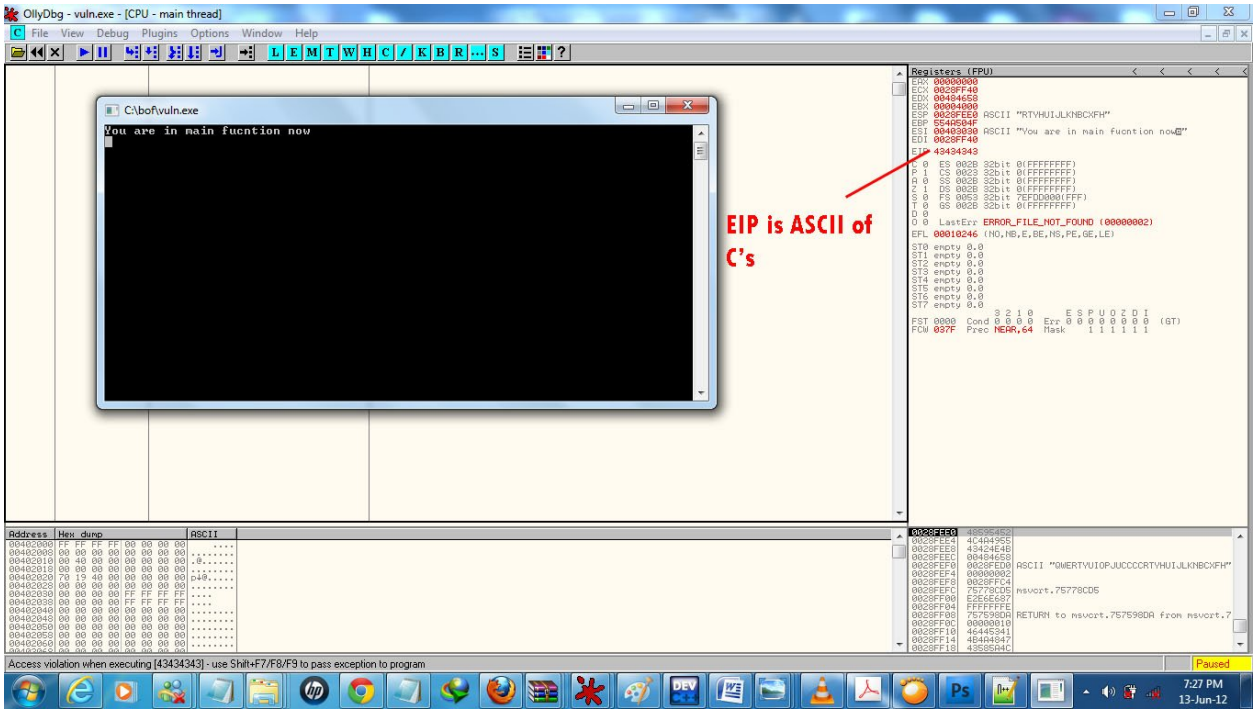
code[]=”ASDFGHJKLZXCVBNMQWERTYUIOPJU***CCCC***RTYHUIJLKNBCXFH”;

Now again change char

code[]=”ASDFGHJKLZXCVBNMQWERTYUIOPJU***CCCC***RTYHUIJLKNBCXFH”;

In original code and compile it via DevC++ and check the value of EIP via OllyDbg.

Here what we got



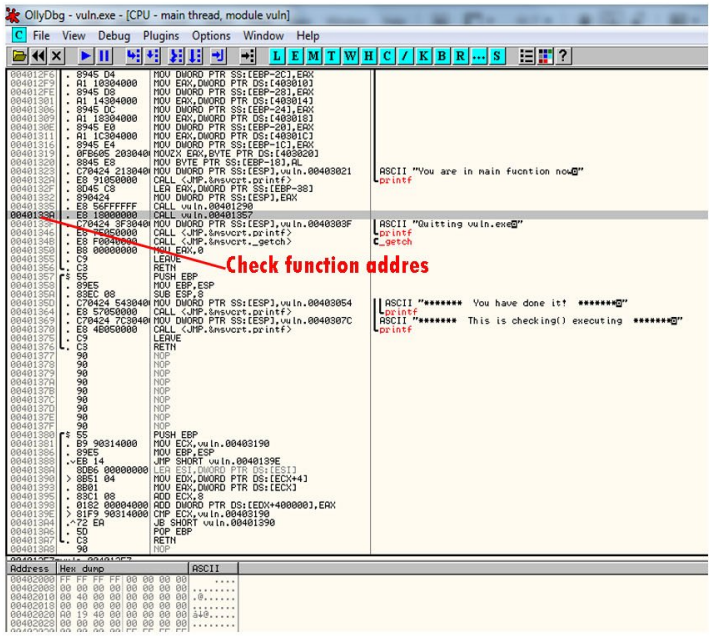
EIP →43434343

Which is equal to CCCC which we change in our string so this is confirmed now where we have to provide the address for EIP in the string.

Now we will find the address of check function In our program so that we will provide that address of check function to EIP and the our program will run correctly

You can easily find the address of check function via OllyDbg.

Here is the address of check function



From OllyDbg the check function address is→ 0040133A

Now we will use this address and make flow of our program once again

Before we use this memory address we have to convert it into little Indian

So this becomes 3A134000→\x3A\x13\x40\00

Now pass this address to vulnFunction via value of char code after 28 bytes of junk data as we already figure out that we can overwrite EIP after 28 bytes

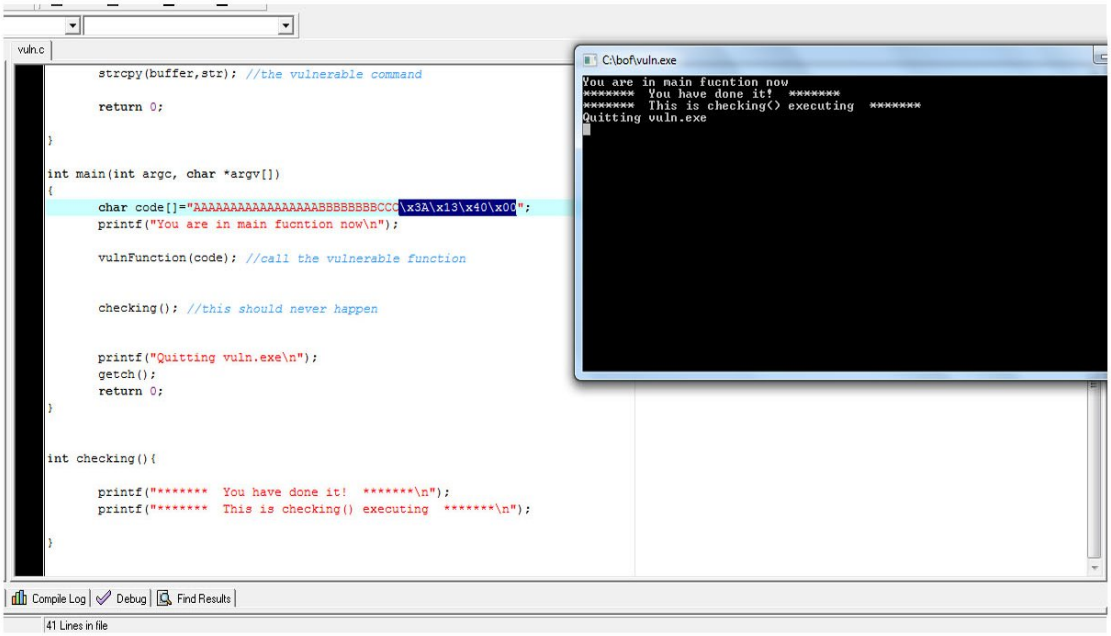
So

code[]=”ASDFGHJKLZXCVBNMQWERTYUIOPJU***CCCC***RTYHUIJLKNBCXFH”;

Change CCCC to Address we taken out ,this becomes

code[]=”ASDFGHJKLZXCVBNMQWERTYUIOPJU\x3A\x13\x40\00”;

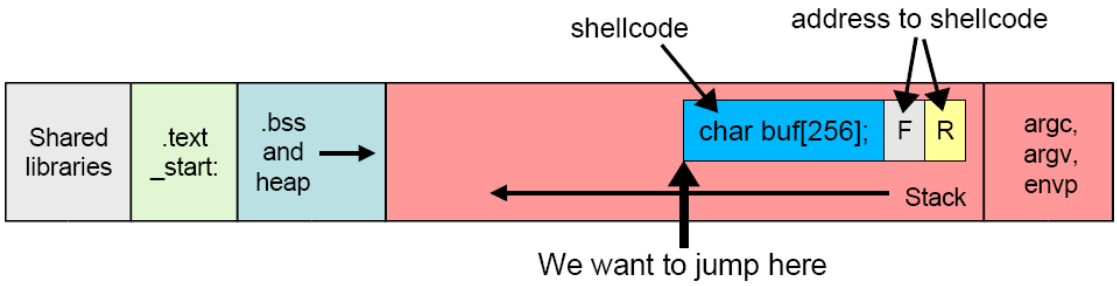
Now change this value of char code in the original code compile it and see the result your program will run fine.



So now you can see how we change the flow of program and execute what we want. You can try it more by making any function and don't call it in main program and as things stated above pass that function address to vulnFunction overwrite the EIP and execute the program as you want.

This is the basic idea of Buffer over flow, now days people make exploits on buffer over flow and run their own shell codes via buffer over flow. This can simply be done by storing your shell code somewhere middle in buffer and then as you know EBP pointer has the memory address of the top of the stack so using it you can get the stack top position address and your shell code is already store in buffer so you can change the value of EIP to the value of EBP. One more thing there is an instruction called NOP.NOP does nothing just jump to next instruction without doing anything so if you have 28 bytes buffer you can first fill it with 8 NOP(\x90) and then your shell code.

Here is picture for little more understanding



Video Explatation

http://www.youtube.com/watch?v=2-y_W7NTsJc

For more information

Dr-freak2011@hotmail.com

Greets MrCreepy . Virus Hima . Red Virus . SeeKer . MKhan . Napster Crew . Dr.Angel Hex786 . The Lions Heart . 0xf-Security . Hoax . Electroblaster . N477 . Darkx