**1. IMPLEMENTING SIMPLE BUFFER OVERFLOWS**

**2. IMPLEMENTING SIMPLE FORMAT STRING ATTACKS**

**SUBJECT NAME**: CRYPTOGRAPHY AND NETWORK SECURITY

**SUBJECT CODE:** CS6008

**MODULE:** 2

|  |  |
| --- | --- |
| **NAME** | BHUVANESHWAR S |
| **REG.NO** | 2019103513 |
| **DATE** | 05/04/2022 |

1. **Implementing a simple buffer overflow**

**AIM :**

To implement a buffer overflow attack using binary executable binary files.

**TOOLS INVOLVED:**

* GCC
* GDB
* KALI LINUX TERMINAL (WSL)

**PROBLEM DESCRIPTIONS:**

Buffers are memory storage regions that temporarily hold data while it is being transferred from one location to another. A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to buffer overwrites adjacent memory locations.

**INPUT:**

Getting an input from user in executable binary files.

**OUTPUT:**

Debug the binary code to find how stack values are modified.

**SCREENSHOT:**

**Filename:** overflow.c

#include <stdio.h>

void ReadInput()

{

  char buffer[8];

  gets(buffer);

  puts(buffer);

}

int main()

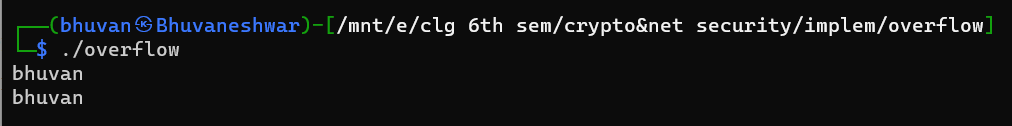
{

  ReadInput();

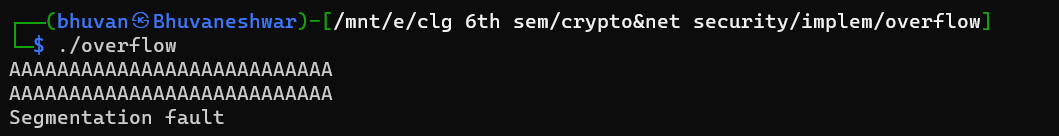
  return 0;

}

Following output will be



If user enter more than 8 characters.



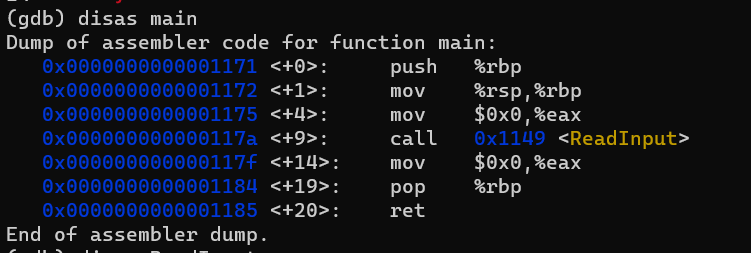
Because the size of buffer was defined and it filled with more than 8 characters, so the buffer was overflowed.

Let see in gdb debugging.

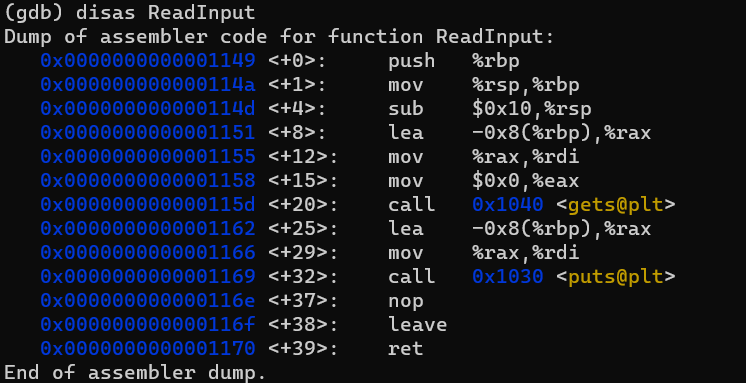


Disassemble

* Main

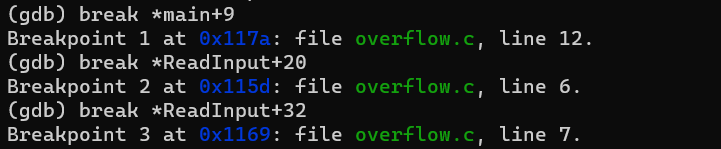


* ReadInput



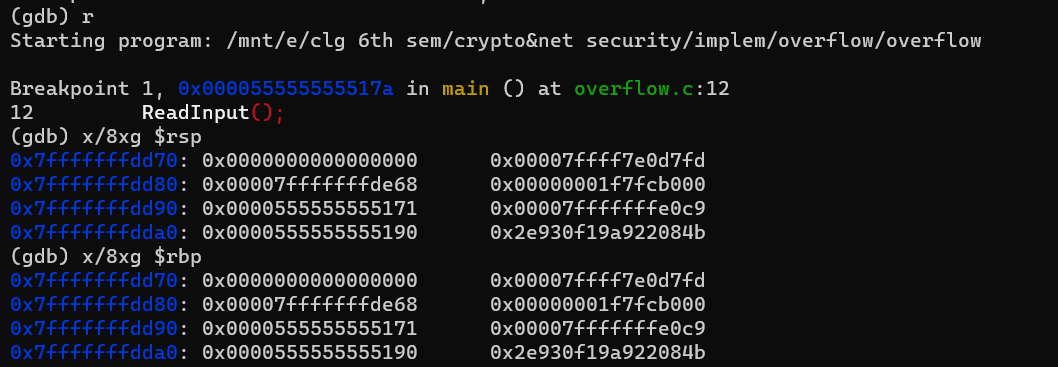
3 break point are set at the

* GetInput function
* gets
* puts

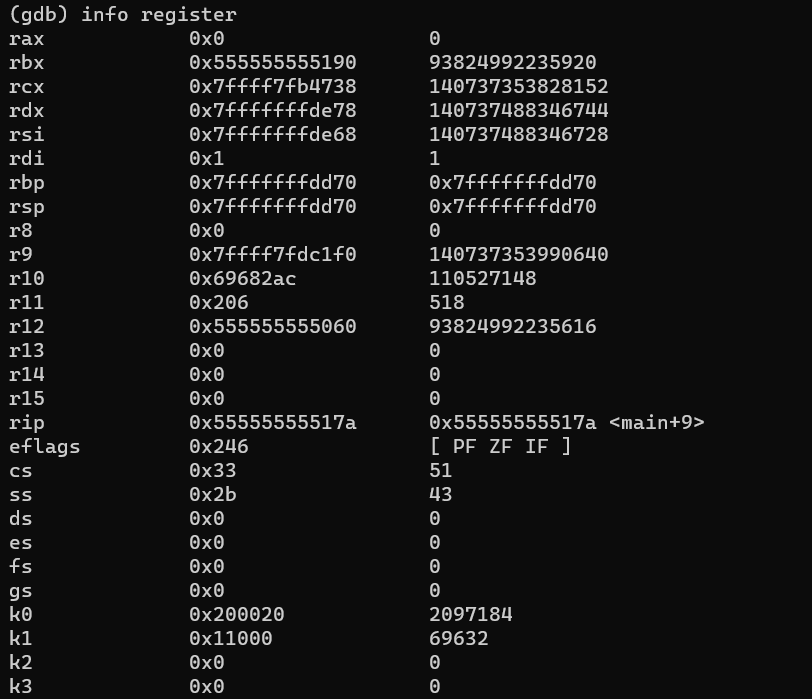


Now run the program

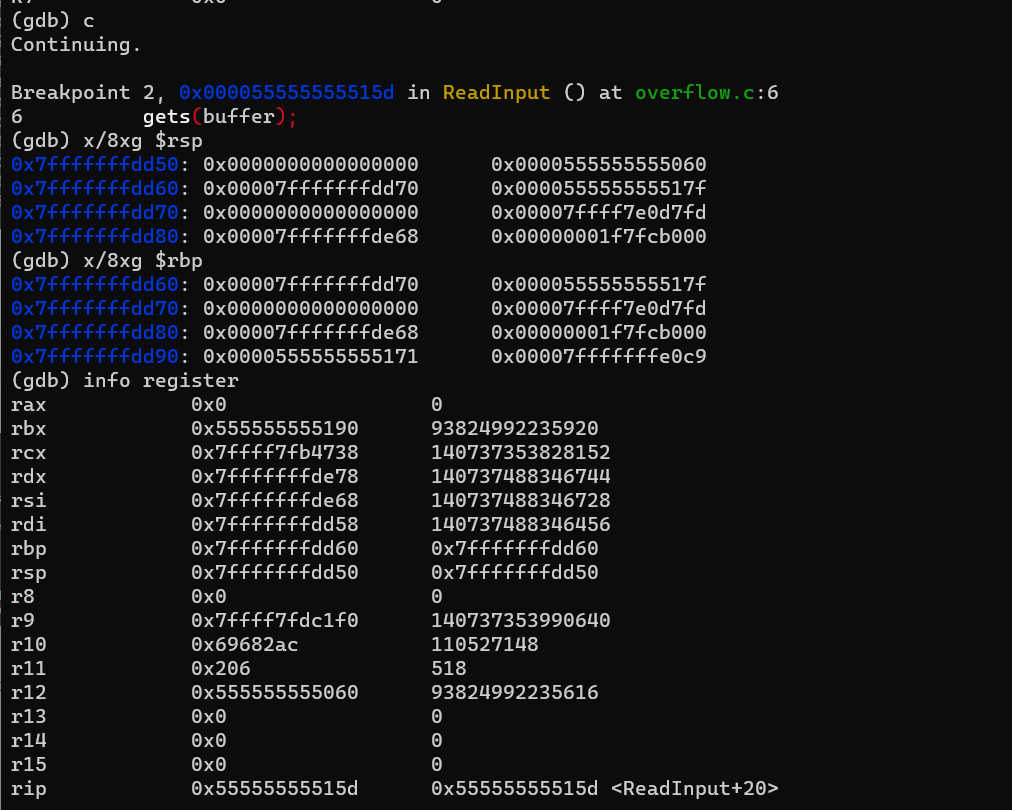
* It will run upto breakpoint 1 (\*main+9)



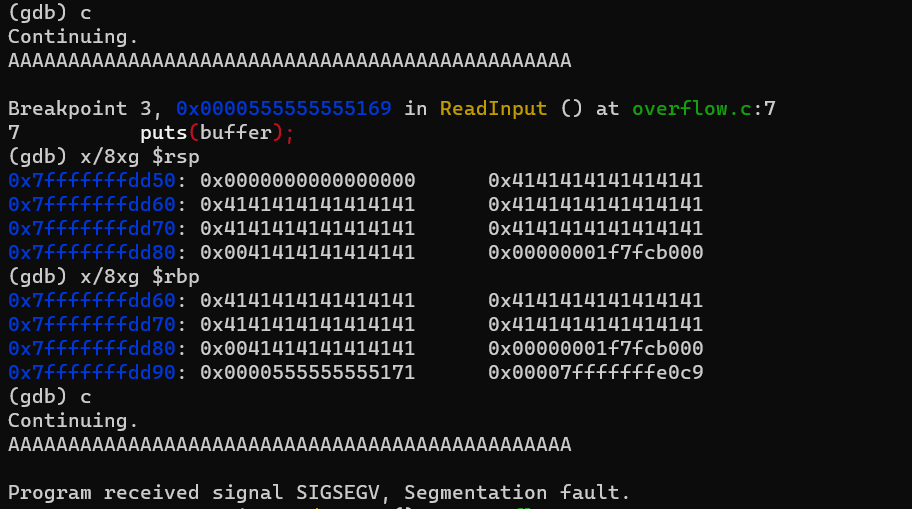
Info register



Continue the execution. It will stop at breakpoint 2 gets() (\*ReadInput+20).



User input is given with more than 8 bytes. So it overflows the buffer and it overwrites the epb values.



SIGSEV due to modified ebp value. Therefore program is terminated.

1. **Implementing a format string vulnerabilities**

**AIM :**

To implement a format string attack in executable binary files.

**TOOLS INVOLVED:**

* GCC
* KALI LINUX TERMINAL (WSL)

**PROBLEM DESCRIPTIONS:**

The Format String exploit occurs when the submitted data of an input string is evaluated as a command by the application. In this way, the attacker could execute code, read the stack, or cause a segmentation fault in the running application, causing new behaviors that could compromise the security or the stability of the system.

**INPUT:**

Getting an input password from user in executable binary files.

**OUTPUT:**

To find how it leaked the memory from the stack based on user inputs.

**SCREENSHOT:**

**FILENAME:** frmt\_vuln.c

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main(int argc,char \*argv[]){

    char text[1024];

    static int test\_val = -72;

    if(argc < 2){

        printf("Usage: %s <text to print>\n",argv[0]);

        exit(0);

    }

    strcpy(text,argv[1]);

    printf("The right way to print user-controlled imput : ");

    printf(" %s\n",text);

    printf("\nThe wrong way to print user-controlled imput : ");

    printf(text);

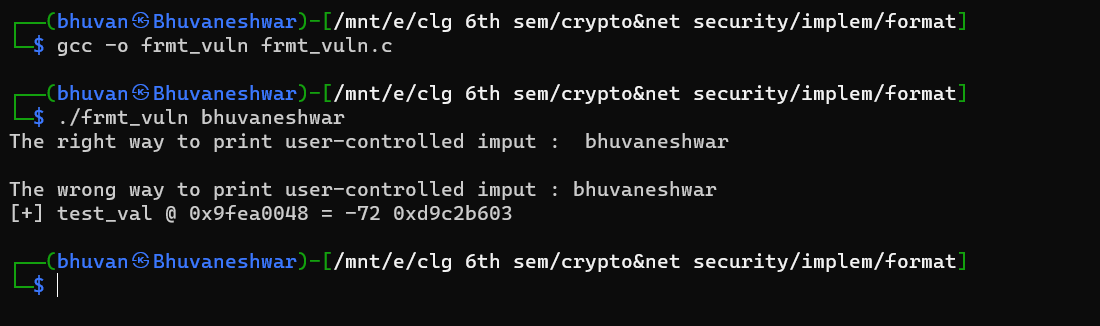
    printf("\n");

    printf("[+] test\_val @ 0x%08x = %d 0x%08x\n",&test\_val,test\_val);

    return 0;

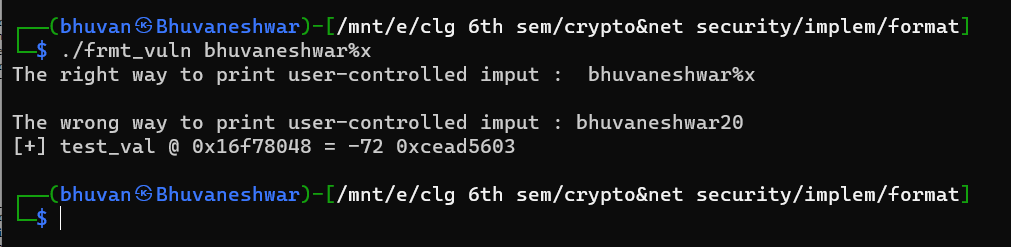
}

The following output shows the compilations and execution of frmt\_vul.c

****

Both method seem work with the string “bhuvaneshwar”.

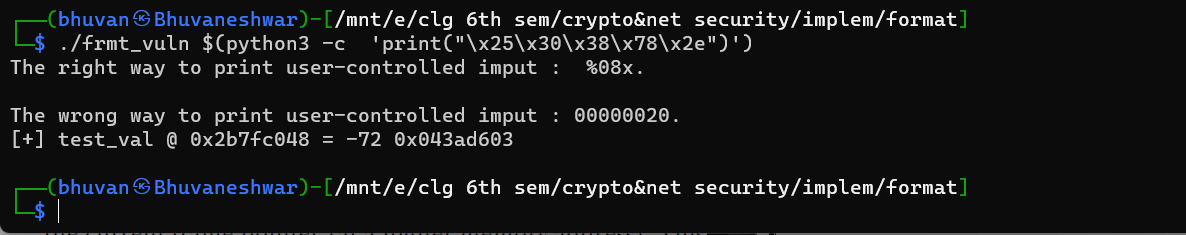
If user enter format parameter into the string to access the appropriate function argument by adding to the frame pointer.

****

When the %x format parameter was used, the hexadecimal representation of a four-byte word in the stack was printed. This process can be used repeatedly to examine stack memory.



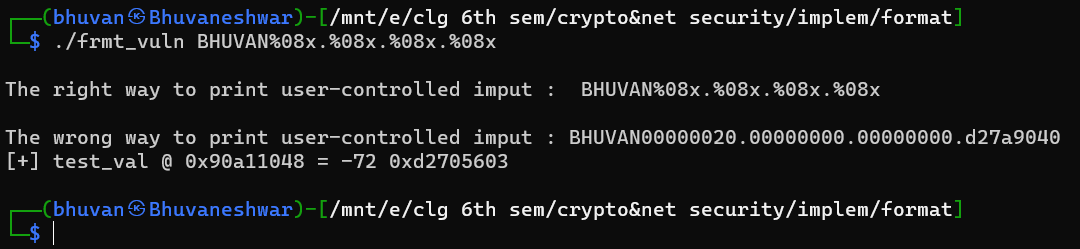
Here lower stack memory represented through the printf. Each four-byte word is backward due to an little end architecture.



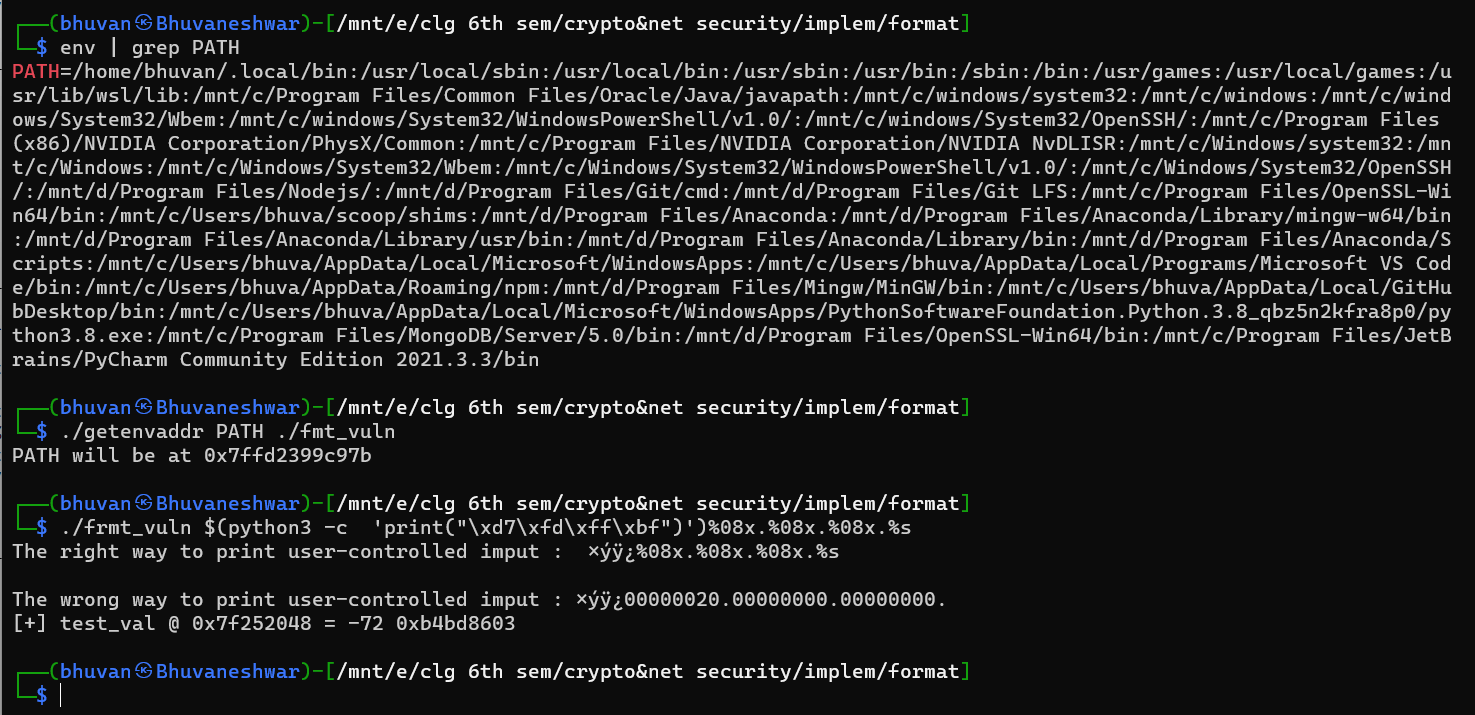
Here it represent the memory for the format string itself. Because the format function will always be on the highest stack frame as long as the format string stored anywhere one the stack. It will be located below the current frame pointer. It used to control arguments to the format function.

**READING FROM ARBITRARY MEMORY ADDRESS**

The %s format parameter can be used to read from arbitrary memory addresses. So it possible to read the data of the original format string.



The four bytes indicates that the fourth format parameter is reading from the beginning of the format string to get its data.



Here the getenvaddr program is used to get the address for the environment variable PATH. Since the program name fmt\_vuln is two bytes less than getenvaddr, four is added to the address, and the bytes are reversed due to the byte ordering. The fourth format parameter of %s reads from the beginning of the format string, thinking it’s the address that was passed as a function argument. Since this address is the address of the PATH environment variable, it is printed as if a pointer to the environment variable were passed to printf().

**WRITING TO ARBITRARY MEMORY ADDRESSES**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

void bad\_function(){

    printf("\nCant be executed\n");

}

int main(int argc,char \*argv[]){

    int val = 5;

    printf(argv[1],&val);

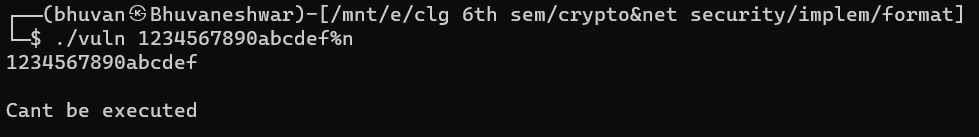
    if(val == 15)

        bad\_function();

    return 0;

}

The %n format parameter can be used to write to an arbitrary memory address.

****

The value of the variable val is modified to 15 to call the bad\_function().