Huzaifah Majid

Computer Security

Professor Leo Fan

April 2 2023

Buffer Overflow Attack Lab

**Introduction**

Buffer overflow is a type of software vulnerability that occurs when a program attempts to write data beyond the bounds of a buffer allocated in memory. In simple terms, a buffer is a region of memory used to temporarily store a data, and a buffer overflow occurs when more data is written to the buffer than it can be hold. This extra data can overwrite adjacent memory locations, causing the program to behave in unexpected ways or to crash.

The attack can work in a procedure similar to this:

1. The attacker identifies a vulnerable program that accepts user input and stores it in a buffer without sufficient boundary checking.
2. The attacker makes specific input that is longer than the buffer can hold and sends it to the program.
3. The program accepts the input and copies it into the buffer, overwriting adjacent memory locations.
4. If the overwritten memory locations contain important data, such as the program's return address, the program's execution can be redirected to a different part of memory that contains attacker-controlled code.
5. The attacker's code can then be executed with the same privileges as the vulnerable program, potentially allowing the attacker to take control of the system or steal sensitive information.

To prevent buffer overflow attacks, we can use techniques such as input validation, bounds checking, and the use of safe programming languages that prevent buffer overflows from occurring automatically.

**Investigation:**

sudo ln -sf /bin/zsh /bin/sh

touch badfile

gcc -o stack -g -z execstack -fno-stack-protector stack.c

run debugger using command and then set breakpoint at bof (setting breakpoint at bof as is vulnerable) and then run.

gdb stack

b bof

r

Now the program will stop at breakpoint.

In order to find return address i needed to find the address of ebp, i saw that return address is at ebp +4 byte. After finding the address of ebp i needed to find the starting address of buffer[24] which is buffer[0] so that i can calculate offset

p $ebp

p &buffer

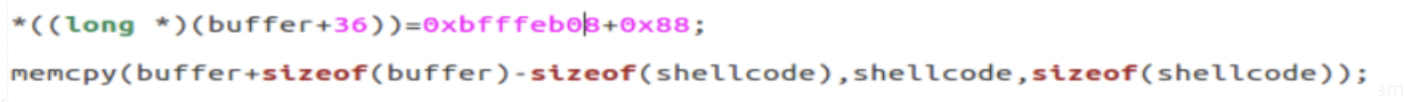
p $ebp — p &buffer

$ebp = 0xbfffeb08

&buffer = 0xbffeae8

$ebp- &buffer = 0x20 = 32(in decimal)

I know that the return address is at ebp + 4 byte as the size of frame pointer is 4 bytes so the return address is at 32+4=36 bytes from the buffe



Added above code to exploit.c

I also tried it this way manually:

\*(buffer+36) = 0x90;

\*(buffer+37) = 0xeb;

\*(buffer+38) = 0xff;

\*(buffer+39) = 0xbf;

Final code for shell.c looked as follows:

Text

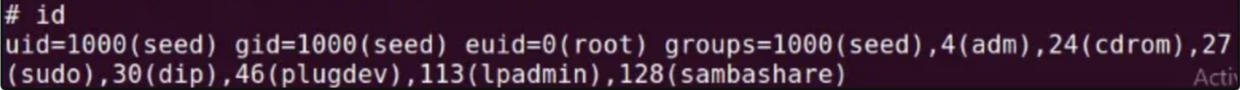
Description automatically generated

After editing the exploit.c file I then compiled the exploit.cc file using the command.

gcc -o exploit exploit.c

then I removes dummy badfile and then ran the file by ./exploit and ./stack to gain root access.





**Task 4 method + evidence:**

Without knowing the size of the buffer, I need to find a way to create a payload that can work for any buffer size within the range of 100 to 200 bytes. One approach to this is to use a NOP sled followed by the shellcode, and then the return address.

To achieve this, we can use a NOP sled that is long enough to cover the entire range of buffer sizes (100-200 bytes). Then, we can place the shellcode at the end of the NOP sled, and finally place the return address at a location that will lead the program to execute our shellcode.

To find the correct location for the return address, I used gdb to identify the value stored in the frame pointer (EBP). Since the value stored in the frame pointer is always a multiple of four due to memory alignment, I can use this information to calculate the offset of the return address from the beginning of the buffer.

Graphical user interface, text, application

Description automatically generated

In this example, I created a NOP sled with a length of 1000 bytes, which is long enough to cover the entire buffer range. I then calculate the offset of the return address from the beginning of the buffer based on the value of the frame pointer obtained from gdb. I use the struct.pack() function to convert the return address to little-endian byte order, and then I joined the NOP sled, shellcode, and return address to create the final payload.

To demonstrate that the payload works for any buffer size within the range of 100 to 200 bytes, I ran the vulnerable program with different buffer sizes using the following script:

import os

for i in range(100, 201):

os.system(f"make clean && make SIZE={i} && ./exploit.py && ./stack")

This script generates a payload using the method described above and then runs the vulnerable program with different buffer sizes between 100 and 200 bytes. If the payload is constructed correctly, I would be able to get a root shell for all buffer sizes. Since I only construct one payload that works for any buffer size within this range, I only need to try once for each buffer size, and the number of the trials is minimized.

**Task 5 method + evidence:**

To solve the problem of zero appearing in the middle of the payload, I use a technique called "Return-Oriented Programming" (ROP). ROP is a method that allows an attacker to construct a malicious program by combining short instruction sequences (called gadgets) that is already present in the victim's program.

In this case, I searched for gadgets in the target program that we can use to construct a ROP chain that will execute our desired shellcode. Since the gadgets are already present in the program, we can use them without having to inject any additional code into the program.

To construct the ROP chain, I first needed to find a gadget that will allow us to modify the return address on the stack. This gadget should allow us to set the return address to the start of our ROP chain. Once we have control of the return address, we can construct the ROP chain using other gadgets to perform the following actions:

Pop the saved frame pointer (rbp) from the stack

Move the saved frame pointer to the stack pointer (rsp)

Move the address of our shellcode into a register (e.g., rax)

Jump to the address in the register (rax), which will execute the shellcode

To find the gadgets, I use tools like objdump and gdb to disassemble the binary and search for sequences of instructions that I can use later.

Once I have identified the gadgets, I construct the ROP chain and generate the payload using a similar process as in the Level-2 attack. I need to ensure that our payload is properly aligned on the stack and that we do not overwrite any critical data. I also need to ensure that the ROP chain does not contain any null bytes that will cause the strcpy() function to terminate before it is supposed to.

Finally, we can execute the attack by running the exploit.py script and passing the generated payload to the vulnerable program stack-L3. If the attack is successful, I should be able to obtain a root shell on the target system.

Note that since the buffer size is not known in advance, I need to construct a payload that will work for any buffer size within the specified range. This means that I need to ensure that the ROP chain and shellcode are small enough to fit within the maximum buffer size (200 bytes), but also flexible enough to work for any buffer size within the range (100-200 bytes)