Roll Number:

Thapar Institute of Engineering & Technology, Patiala
Department of Computer Science and Engineering

BE (3rd year) EST 28May 2022 UCS638: Secure Coding
Elective Focus: Cyber and Information Security
Time: 02 Hours; MM: 50 Note: 5 questions in total over a spread of 2 pages.

Q1.

i) Assume that a database only stores the sha256 value for the password and eid columns, The following SQL statements are sent to the database, Does this program have a SQL injection problem.

a) The values of the $passwd and $eid variables are provided by users.

```
$sql = "SELECT * FROM employee
    WHERE eid='SHA2($eid,256)'
and
password='SHA2($passwd,256)'";
```

b) Hash values is calculated in the PHP code using PHP's hash() function.

```
$hashed_eid = hash('sha256',
$eid);

$hashed_passwd = hash('sha256',
$passwd);
$sql = "SELECT * FROM employee
    WHERE eid='hashed_eid' and
password='hashed_passwd'";
```

(2, 2)

ii) The following SQL statement is sent to the database to add a new user to the database, where the content of the `$name` and `$passwd` variables are provided by the user, but the EID and Salary field are set by the system. How can a malicious employee set his/her salary to a value higher than 80000?

```
$sql = "INSERT INTO employee (Name, EID,
Password, Salary)
        VALUES
('$name','EID6000','$passwd',80000)";
```

(2)

iii) The following SQL statement is sent to the database to modify a user's name and password where the content of the `$name`, `$oldpwd` and `$newpwd` variables are provided by the user. You want to set your boss Bob's salary to `$1` (using the Salary field), while setting his password to something that you know, so you can later log into his account.

```
$hashed_newpwd = hash('sha256', $newpwd);
$hashed_oldpwd = hash('sha256', $oldpwd);

$sql = "UPDATE employee
 SET name='$name', password='$hashed_newpwd'
WHERE eid = '$eid' and
password='$hashed_oldpwd'
```

(4)

Q2.

i) What if the SQL statement is constructed in the following way (with a line break in the WHERE clause), can you still launch an effective SQL injection attack?

```
SELECT * FROM employee
WHERE eid= '$eid' AND
password='$password'
```

(2)

ii) The following SQL statement is sent to the database, where $eid and $passwd contain data provided by the user. An attacker wants to try to get the database to run an arbitrary SQL statement. What should the attacker put inside $eid or $passwd to achieve that goal. Assume that the database does allow multiple statements to be executed.

```
$sql = "SELECT * FROM employee
WHERE eid='$eid' and
password='$passwd'"
```

(2)

iii) To defeat SQL injection attacks, a web application has implemented a filtering scheme at the client side: basically, on the page where users type their data, a filter is implemented using JavaScript. It removes any special character found in the data, such as apostrophe, characters for comments, and keywords reserved for SQL statements. Assume that the filtering logic does it job, and can remove all the code from the data; is this solution able to defeat SQL injection attacks? Please, elaborate with help of an example. **(3)**

iv) Please modify the following program using the prepared statement.

```
$sql = "UPDATE employee SET
password='$newpwd'
WHERE eid ='$eid' and
password='$oldpwd'";
```

(3)

Q3.

i) To defeat XSS attacks, a developer decides to implement filtering on the browser side. Basically, the developer plans to add JavaScript code on each page, so before data are sent to the server, it filters out any JavaScript code contained inside the data. Let's assume that the filtering logic can be made perfect. Can this approach prevent XSS attacks?

ii) These days, most of the websites use HTTPS, instead of HTTP. Do we still need to worry about CSRF attacks?

iii) The fundamental cause of XSS vulnerabilities is that HTML allows JavaScript code to be mixed with data. From the security perspective, mixing code with data is very dangerous. XSS gives us an example. Please provide two other examples that can be used to demonstrate that mixing code with data is bad for security.

(3,2,5)

Q4

i) Does the following Set-UID program have a race condition vulnerability?

```
if (!access("/etc/passwd", W_OK)) {
f = open("/tmp/X", O_WRITE);
write_to_file(f);
}
else{
fprintf(stderr, "Permission denied\n");}
```

ii) Assume we develop a new system call, called faccess(int fd, int mode), which is identical to access(), except that the file to be checked is specified by the file descriptor fd. Does the following program have a race condition problem?

```
int f = open("/tmp/x", O_WRITE);
if (!faccess(f, W_OK)) {
write_to_file(f)
} else {
close(f);}
```

iii) In the open() system call, it first checks whether the user has the required permission to access the target file, then it actually opens the file. There seems to be a check-and-then use pattern. Is there a race condition problem caused by this pattern?

iv) The least-privilege principle can be used to effectively defend against the race condition attacks. Can we use the same principle to defeat buffer-overflow attacks? Why or why not? Namely, before executing the vulnerable function, we disable the root privilege; after the vulnerable function returns, we enable the privilege back.

(2, 2, 3, 3)

Q5.

i) The fork() system call creates a new process from a parent process. The new process, i.e., the child process, will have a copy of the parent process's memory. Typically, the memory copy is not performed when the child process is created. Instead, it is delayed.

(1) Please explain when the memory copy will occur.

(2) In the Dirty COW attack, can we run two processes, instead of two threads?

ii) When a process maps a file into memory using the MAP PRIVATE mode, the memory mapping is depicted in Figure Q5.2

(1) Please describe what is going to happen when this process writes data to address 0x5100.

(2) The Dirty COW race condition occurs inside the write() system call. Please explain exactly where the problem is.

(3) How can this race condition vulnerability be exploited?

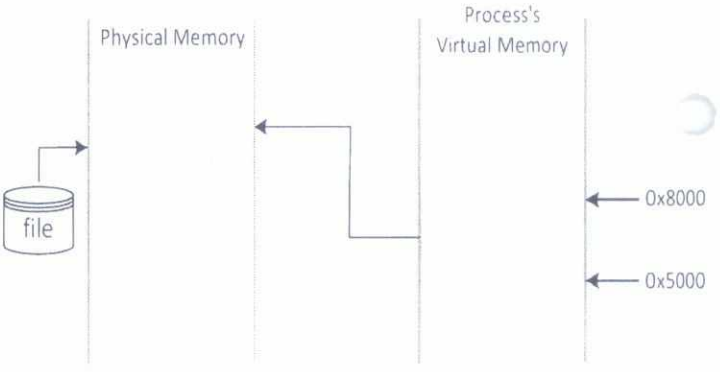

Figure Q5.2

(4, 6)