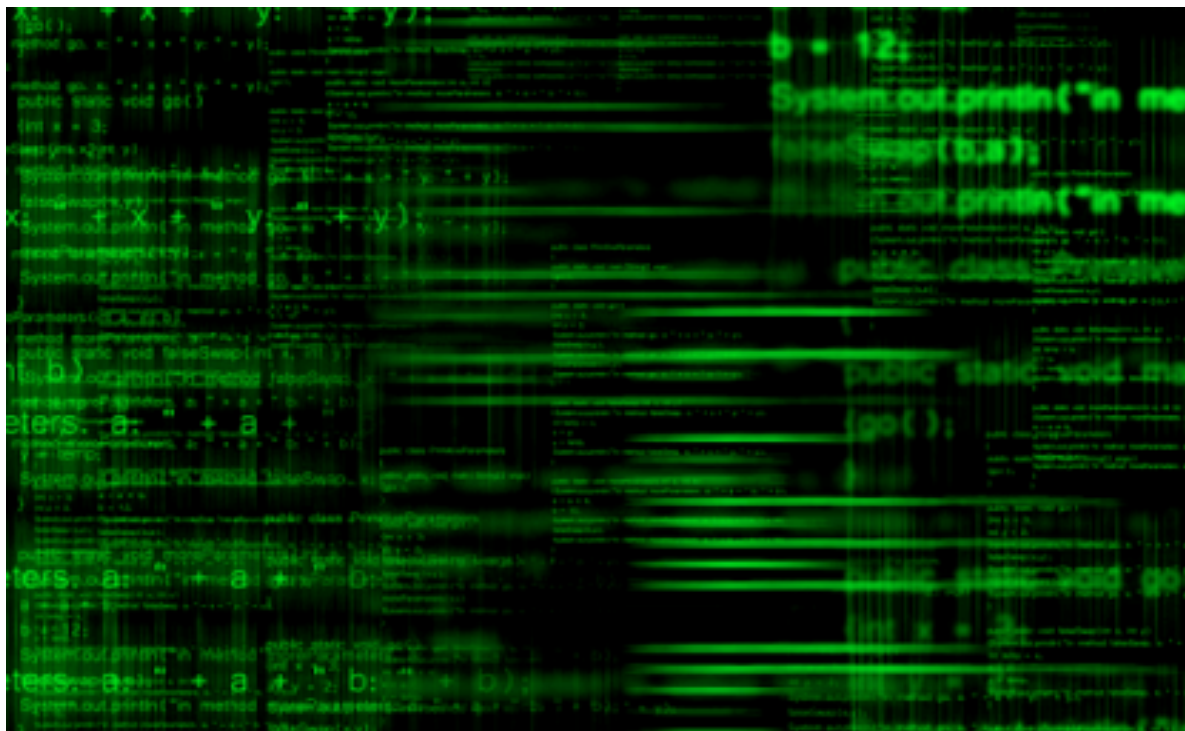


Malicious users employ various types of attacks to exploit vulnerabilities in systems or programs used by websites and applications. One prominent example is the **Buffer Overflow attack**, where users take advantage of systems, forms, or applications that fail to perform proper boundary checks on variables. Additionally, poorly implemented functions that do not proactively handle such scenarios exacerbate the issue. During this attack, a malicious user inputs more characters than a variable can accommodate, potentially leading to the destruction of the program or the corruption of variables linked to other processes. Another common type of attack is **SQL Injection**, where attackers exploit design flaws in the system. By injecting specific commands in SQL, they can manipulate, destroy, or gain unauthorized access to the system, highlighting the critical need for robust security measures.

- **Buffer overflow attack**
- **SQL injection attack**

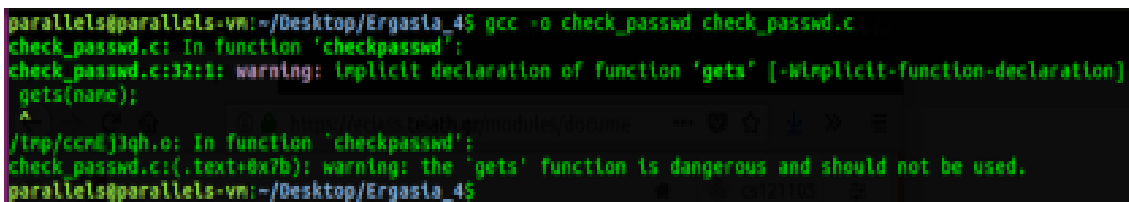


Question 1)

a) In the provided code, which is based on the program `check_passwd.c`, we can quickly identify a significant vulnerability. This issue arises from the way the code handles user input using the `gets(name)` and `gets(passwd)` functions. The problem is that these functions do not check or limit the number of characters a user can input, meaning that they do not enforce any boundaries on the amount of data being stored in the respective variables. As a result, an attacker could input more characters than the variable can hold, potentially causing a **buffer overflow**. This flaw allows attackers to overwrite adjacent memory locations, which could lead to unexpected behavior, data corruption, or even the execution of malicious code.

```
while ((attempts!=3)&&(correct==0))
{
    printf("Enter login:");
    gets(name);
    printf("Enter password:");
    gets(passwd);
```

If the user enters more characters than the variable can accommodate, the excess characters will likely overwrite adjacent memory locations. This can lead to unintended consequences, such as the corruption or destruction of other critical variables within the program. When the program is compiled, the system issues a warning indicating that the use of the `gets()` function is **unsafe**. This is because `gets()` does not check the length of the input, making it highly vulnerable to **buffer overflow** attacks. These warnings are designed to alert developers about the security risks associated with using this function, as it can compromise the integrity of the program and lead to potential exploits.



```
parallels@parallels-vm:~/Desktop/Ergasia_4$ gcc -o check_passwd check_passwd.c
check_passwd.c: In function 'checkpasswd':
check_passwd.c:32:1: warning: implicit declaration of function 'gets' [-Wimplicit-function-declaration]
    gets(name);
    ^
/trp/ccm4j1qh.o: In function 'checkpasswd':
check_passwd.c:(.text+0x7b): warning: the 'gets' function is dangerous and should not be used.
parallels@parallels-vm:~/Desktop/Ergasia_4$
```

Figure 1

By entering the user's **username** and **password**, we are able to log in to the "system" as expected. At this point, the program functions normally without any issues or errors. The login process appears to work smoothly, with no visible problems or interruptions in the system's behavior. This indicates that the program successfully validates and processes the user's credentials, allowing access to the system without any apparent vulnerabilities at this stage.

```
parallels@parallels-vm:~/Desktop/Ergasia_4$ ./check_passwd
Enter login:admin
Enter password:secret
Password correct - Login approved
parallels@parallels-vm:~/Desktop/Ergasia_4$
```

Figure 2

However, if we enter more characters, we will see that these extra characters cause a Buffer Overflow. The image below shows the attempts made with more characters as well as the message that the system displays in the case of a Buffer Overflow.

```
parallels@parallels-vn:~/Desktop/Ergasia_4$ ./check_passwd
Enter login:administraroradmin
Enter password:secretsecretsecret
Enter login:hooooooooooooooooooooo
Enter password:hooooooooooooooooooooooooooooooooooooo
Enter login:hoooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
Enter password:hoooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
*** stack smashing detected ***: ./check_passwd terminated
Aborted (core dumped)
parallels@parallels-vn:~/Desktop/Ergasia_4$
```

Figure 3

At this stage, the system displays the message: **"stack smashing detected,"** indicating that a **Buffer Overflow** has occurred. This message signifies that the program has detected an attempt to overwrite memory beyond the allocated stack boundaries. Such a breach can potentially lead to the destruction or corruption of critical program variables. This warning serves as a safeguard, alerting developers to the overflow issue and highlighting the need to address the vulnerability to prevent malicious exploits or unintended program behavior.

This issue can be corrected by replacing the `gets()` function with the safer `fgets()` function. The `fgets()` function takes three arguments: the first is the variable to store the input (in our case, either **name** or **password**), the second is the maximum number of characters that can be entered (such as 16), and the third is the input stream, which is typically `stdin` for standard input. By using `fgets()`, the program performs a **boundary check** on the variable, ensuring that the user cannot enter more than 16 characters. If the user attempts to enter more characters, the system will display an error message, preventing any buffer overflow from occurring. This approach enhances the security and stability of the program by limiting input size and preventing unintended memory overwrites.

```
printf("Enter login:");
fgets(name,16,stdin);
printf("Enter password:");
fgets(passwd,16,stdin);
```

b) The function `strcpy(s1, s2)` does not perform any checks to verify whether the destination string `s1` can actually hold the content of the source string `s2`. This can lead to potential buffer overflows if `s1` is not large enough. To improve security, the function `strncpy(s1, s2, 50)` is used instead. This function adds an extra parameter, which specifies the maximum number of characters to be copied from `s2` to `s1`. In this case, it ensures that no more than 50 characters are copied, preventing the possibility of overwriting memory beyond the bounds of `s1`. By using `strncpy()`, the program becomes more robust, as it enforces length constraints to safeguard against buffer overflow vulnerabilities.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      char source[] = "username12"; // username12 to source[]
7      char destination[7]; // Destination is 8 bytes
8      strncpy(destination, source); // Copy source to destination
9
10     return 0;
11 }
```

Figure 4

c)

```
while ((c=getchar()) !='\n');  
    str[i++]=c;
```

In this program, inside the recursive structure, the function `getchar()` reads a character from the input and stores it in the variable `c` without checking the size of the input. This can potentially lead to buffer overflows if the input exceeds the size allocated for `c`. To address this, we can introduce a second argument that specifies the maximum number of characters that can be accepted by the variable `c`. With this modification, the function will only store the characters in `c` as long as the input does not exceed the predefined size limit. This ensures that the variable `c` can safely store the characters read by `getchar()`, preventing any overflow issues.

```
while ((i<50)&& (c=getchar()) !='\n'))  
    str[i++]=c;
```

In this way, we define the maximum size of the characters that will be stored in the variable `c` in the program. By specifying the size limit, we ensure that the program only accepts a predefined number of characters, preventing any overflow beyond the allocated memory for `c`. This approach adds an extra layer of safety by ensuring that the variable can handle only the intended amount of input, effectively protecting the program from potential buffer overflows and memory corruption issues.

Question 2)

a) From the form provided in Question 4 to test for SQL injection vulnerabilities, we perform the following test:

In the **Username** field, we input the word **admin**, and in the **Password** field, we enter the string **pass'; DROP TABLE users; --**.

This input is specifically crafted to exploit a potential vulnerability in the system's SQL query. The **pass'** part is intended to close the current string in the SQL query, while the **DROP TABLE users; --** command attempts to delete the **users** table in the database. The **--** at the end is a SQL comment that ignores any subsequent code.

Based on the test results, we conclude that the form is vulnerable to **SQL injection**. This is because the application directly includes user input in the SQL query without properly sanitizing or escaping the input. This allows attackers to manipulate the query and potentially execute harmful commands, such as dropping tables or altering data in the database.

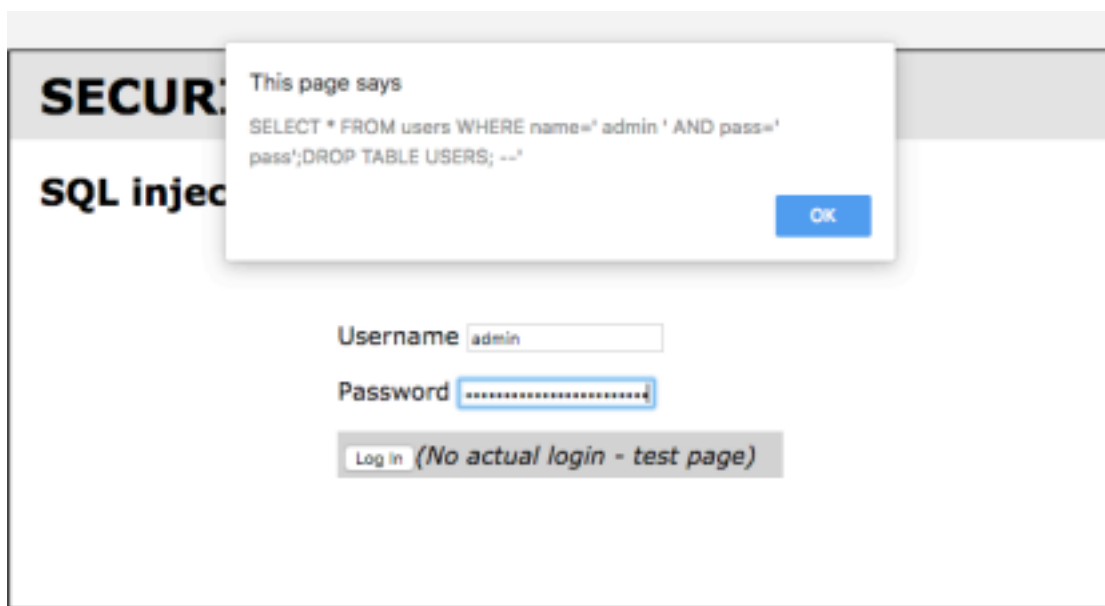


Figure 5

b) In the field: Username we put the word: admin and in the field:

Password 'pass';UPDATE users SET pass='123456' WHERE user_name='anna'

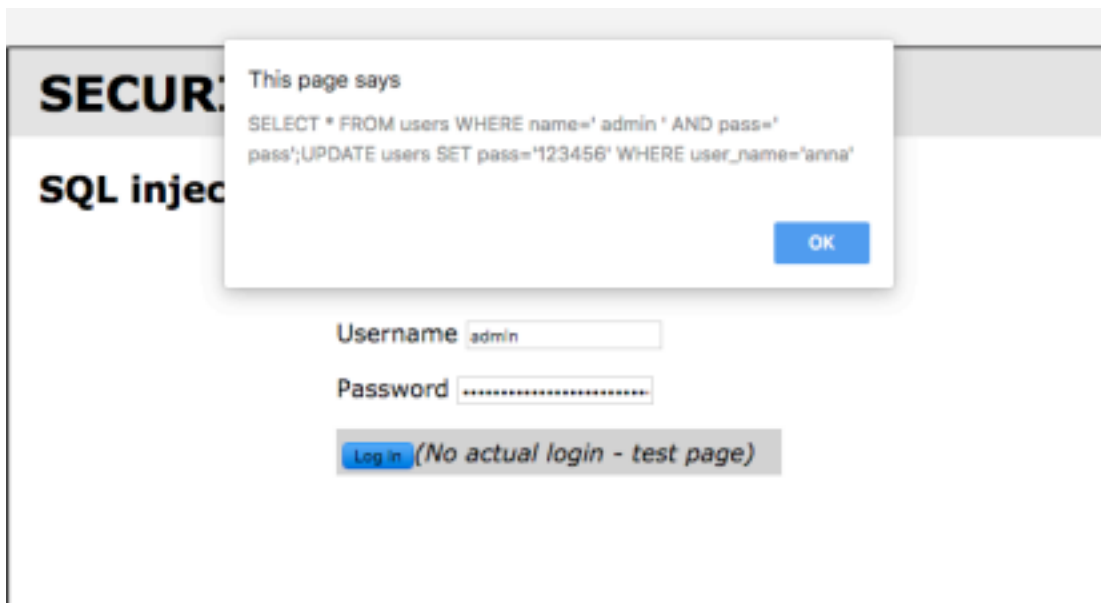


Figure 6

c) In this case, we perform another SQL injection test. In the **Username** field, we input the word **admin**, and in the **Password** field, we enter the string **pass'; INSERT INTO users(user_name, pass) VALUES('guest', 'hacked');** --.

This input is designed to inject a new SQL command into the existing query. The **pass';** part closes the original password string, while the **INSERT INTO users(user_name, pass) VALUES('guest', 'hacked')** statement attempts to insert a new user with the username **guest** and the password **hacked** into the **users** table. The **--** at the end is a SQL comment, which ignores any remaining code that might follow.

If the form is vulnerable to SQL injection, this input could execute the malicious **INSERT** query, adding an unauthorized user to the database with the given credentials. This further confirms that the form is susceptible to SQL injection, as it allows attackers to manipulate the database and insert arbitrary data without proper validation or sanitization of the user input.

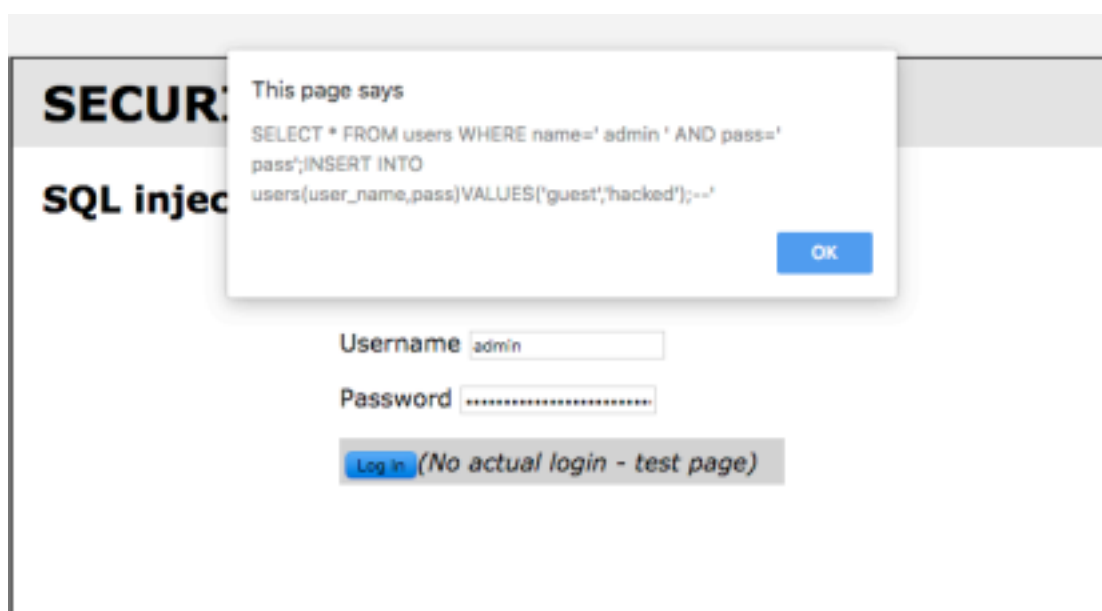


Figure 7

Conclusion:

The specific form provided in the fourth lab exercise is indeed vulnerable to SQL injection. Based on the actions performed, we observe that an attacker can exploit this vulnerability to perform several harmful operations. For example, they could delete an entire table, modify the **users** table by adding new users, or change the password of an existing user.

These actions demonstrate the potential severity of SQL injection vulnerabilities, as they allow attackers to manipulate the underlying database without proper authorization. This highlights the importance of securing user input by using parameterized queries, input sanitization, and other security measures to prevent SQL injection attacks and protect the integrity of the system.

Sources used in this article:

https://www.hackingtutorials.org/exploit-tutorials/buffer-overflow-explained_basics/

[https://owasp.org/www-community/attacks/SQL Injection](https://owasp.org/www-community/attacks/SQL_Injection)

Ambel Basha

Athens May 2018