Robert Rash

Professor Estes

CSE 5359

10 May 2018

Lab 4: Static Code Analysis

Summary

This report is the result of a static code analysis for the `Ghost Box` blogging web application included in the OWASP Broken Web Apps (OWASPBWA) virtual machine. This web application is purposely very vulnerable, but that does not make it any less viable of a candidate for this lab.

Methodology

The workflow for this analysis mimics the way a web crawler indexes a website: start at the home page, and then recursively search through each of the links.

For simplicity, the source code was accessed at its location on Github.

Findings & Recommendations

1. `blogView.php` -- SQL Injection Vulnerability [HIGH]

`/var/www/ghost/blogView.php:14` :

```
1 <?php
2 $user = $_POST['user'];
3 $blogPost = $_POST['vuln'];
4 ...
5 $sql = "UPDATE q SET blog='".$blogPost."' WHERE user='".$user."'";
6 ...
7 $valid = mysql_query($sql, $connect);
```

The SQL query is generated by concatenating raw user inputs into a query string, allowing arbitrary input into the query.

Impact

A malicious user could perform unauthorized and potentially dangerous operations on the database.

Solution

Do not concatenate raw user input to create the SQL query. Instead, use parameterized queries and / or escape the user-provided input.

2. `blogSub.php` -- Stored XSS Vulnerability [HIGH]

`/var/www/ghost/blogSub.php:18` :

```
1 $sql = "SELECT * FROM q";
2 $valid = mysql_query($sql, $connect);
3     while($data = mysql_fetch_array($valid))
4         {
5                 echo "<div><p>".$data['user']." wrote: ".stripslashes($data['blog'
6
7 }
```

The `echo` string evaluates user input that has been stored in the database. This input has not been sanitized previously. In the case of malicious input, the offending code gets reproduced whenever the data gets reproduced.

Impact

XSS attacks can allow anything from session hijacking, account compromise, or installing a Trojan horse.

This particular instance is a stored XSS, which means it is reproduced everywhere that the content is shown. This significantly broadens the range of affected users.

Solution

Properly escape the user-provided data before displaying it. Do not directly evaluate user-provided data that has not been sanitized.

3. `blogView.php` -- XSS Vulnerability [HIGH]

`/var/www/blogView.php:5` :

```
<?php
...
$blogPost = $_POST['vuln'];
...
echo $blogPost;
```

The user-provided input in the `vuln` field gets evaluated as code.

Impact

XSS attacks can allow anything from session hijacking, account compromise, or installing a Trojan horse.

Solution

Properly escape the user-provided data before displaying it. Do not directly evaluate user-provided data that has not been sanitized.

4. `submit.php` -- XSS Vulnerability [HIGH]

`/var/www/submit.php:28` :

```
echo "<div>".stripslashes($user)."</div><br />";
```

The user-provided input in the `$user` variable gets evaluated as code.

Impact

XSS attacks can allow anything from session hijacking, account compromise, or installing a Trojan horse.

Solution

Properly escape the user-provided data before displaying it. Do not directly evaluate user-provided data that has not been sanitized.

5. various files -- Publicized Login Credentials [MEDIUM]

`/var/www/index.php:13`, `/var/www/walk.php:10` :

```
<!--Attn Developers: Username: test Password: 1234  please remove when development
```

Valid, authorized login credentials `test:1234` are included in public-facing code.

Impact

Login credentials left in as HTML comment. Visible to anybody who inspects the HTML source. Could be used to gain unauthorized access to the system.

Solution

Remove the offending comment(s) from the files in which they appear.

Caution the offending developer(s) who included the comment(s) on why this is poor practice, and take measures to ensure that this does not happen again.

6. `submit.php` -- XSS Vulnerability [MEDIUM]

`/var/www/submit.php:21` :

```
if($data['user'] == $user && $data['pass'] == $pass || $user == "\'or 1=1--")
            {
                echo "<input type='hidden' value='".$user."' />";
                echo "<META HTTP-EQUIV=REFRESH CONTENT='0; URL=iframe.php?page=for
            }
```

The user-provided input in the `$user` variable gets evaluated as code.

Impact

XSS attacks can allow anything from session hijacking, account compromise, or installing a Trojan horse.

In this instance, however, the `$user` variable must also be a value from the database. For this attack to work, a user must be registered with the XSS attack as a name.

Solution

Properly escape the user-provided data before displaying it. Do not directly evaluate user-provided data that has not been sanitized.

7. various files -- Hardcoded Database Credentials [MEDIUM-LOW]

`/var/www/submit.php:5`, `/var/www/blogSub.php:3:`, `/var/www/blogView.php:6` :

```
1 | $connect = mysql_connect("localhost", "ghost", "ghost");
```

Credentials for the database have been included in the code that has been committed to source control, presumably public-facing source control.

Impact

A malicious user could perform unauthorized and potentially dangerous operations on the database with the credentials. However, to do so would require previously gaining access to the database server through other means, either legitimately or illegitimately.

Solution

Separate the credentials from the source code. One common solution is to pull the credentials from environment variables, preferably defined in a file that has a shareable format but that does not get checked into source control repositories.

Caution the offending developer(s) who included the credentials(s) on why this is poor practice, and take measures to ensure that this does not happen again.