­­

Assignment II – Preventing Web Application Vulnerabilities

CPS 499-02/592-02

Software/Language Based Security

Fall 2020

Dr. Phu Phung

Evan Krimpenfort

**URL:** <https://github.com/Krimpenfort23/autumn-2020/tree/master/assignments/assignment2>

# Task 0: Web Administration – Preparation

## Database Setup

### Imported the database and created a new user

Text, letter

Description automatically generated

Text, letter

Description automatically generated

**Figure 1: The database**

### Demonstration

Text

Description automatically generated

**Figure 2: classes/db.php**

## Deployment

### Using IP Address

Graphical user interface, text, application

Description automatically generated

**Figure 3: http://192.168.56.101/www.myblog.com/**

### With local domain name

#### SEEDVM

Graphical user interface, application

Description automatically generated

**Figure 4:** [**http://krimpenforte1.myblog.com/**](http://krimpenforte1.myblog.com/) **on the VM**

#### Personal Computer

Graphical user interface, text, application

Description automatically generated

**Figure 5:** [**http://krimpenforte1.myblog.com/**](http://krimpenforte1.myblog.com/) **on the Mac**

## Misconfiguration Security

### Deleted the database file (blog.sql)

Graphical user interface, text, application, website

Description automatically generated

**Figure 6:** [**http://krimpenforte1.myblog.com/blog.sql**](http://krimpenforte1.myblog.com/blog.sql) **not found**

### Changed default username and password

A picture containing diagram

Description automatically generated

**Figure 7: Users available**

**Graphical user interface, text, application

Description automatically generated**

**Figure 8: can log in with** [**krimpenforte1@udayton.edu**](mailto:krimpenforte1@udayton.edu)

## HTTPS Setup

### Certificate was made

Graphical user interface

Description automatically generated

**Figure 9: Certificate Details**

### Deployment was done

Graphical user interface, text, application

Description automatically generated

**Figure 10: Site has https and the footer has been changed**

## Repository

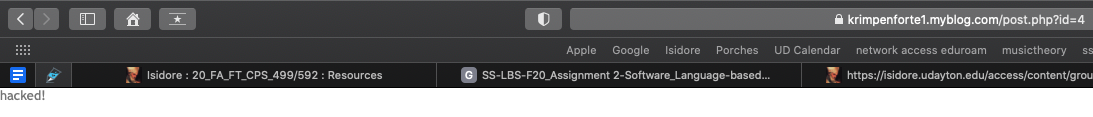
Graphical user interface, application, Teams

Description automatically generated

**Figure 11: Footer was changed**

# Task I: Cross-Site Scripting (XSS) Prevention

## XSS vulnerability in **post.php** code



**Figure 12: hacked in *post.php?id=4***

By placing in the code *<script.document.body.innerHTML=hacked!”;</script>*, whoever is to click on the post of *Task I – Part I*, they will be taken to a page that modifies the HTML to this text. We can fix the problem by going into our code of *classes/post.php* and encapsulate the *$comment->text* block of code (line 81). By doing this, we can see that in Figure 13 the first comment has been prevented of manipulation. Another fix that we can do is better help the function *htmlentities(..)* by encapsulating the inputs before they go into the database. We can do this by encapsulating those calls *$\_POST[<insert name>]* with *htmlspecialchars(..)*. By doing this protection, we can see that in Figure 13 the second comment has been changed to show the specific html characters used.

Graphical user interface, text, application

Description automatically generated

**Figure 13: fixed code with *htmlspecialchars* and *htmlentities***

By doing these fixes, we can safely say that the *classes/post.php* code has been protected from XSS scripting attacks. In Figure 14 the code is pushed and the changes are shown.

Graphical user interface, text, application

Description automatically generated

**Figure 14: Fixed code**

However, with these protections put in place, are there other areas where the code can be protected? Part b will show this.

## Further Revisions

Another location that allows someone to post malicious code could be inside of the admin page. But, *index.php* is being protected by *classes/phpfix.php* with the function *h(..)*. The only further locations with revisions that could be made were further in *classes/post.php*. They were inside the *update(..)* and *create()* functions. In Figure 15, a new post shows the fix.

Graphical user interface, text, application

Description automatically generated

**Figure 15: Further Fix made for the actual post itself**

The encapsulation used in each function was the *htmlspecialchars(..)* function for further sanitation. However, that was all I could find. The update to the repo can be found in Figure 16.

Graphical user interface, text, application

Description automatically generated

**Figure 16: Further fixed code**

# Task II: Session Hijacking Prevention

## Secure the Cookies

Graphical user interface, application

Description automatically generated

**Figure 17: Secured the cookies**

## Defense in Depth

Graphical user interface, text

Description automatically generated

**Figure 18: Session Hijacking Prevented**

**A picture containing graphical user interface

Description automatically generated**

**Figure 19: Additions for Session Hijacking Prevention**

# Task III: CSRF Prevention

## CSRF vulnerability in the code of **admin/new.php**

This attack is able to happen because someone with administrational access logged in and they were able to use their access against them to post something. As we see in Figure 20, a post was made because the logged in user went to the <http://﻿192.168.56.101/csrflab.html> page.

Graphical user interface, text, application

Description automatically generated

**Graphical user interface, text, website

Description automatically generated**

**Figure 20: CSRF Attack**

This attack is prevented by implementing a Secret Validation Token. We could have implemented other methods previously discussed in class like the Referrer Validation methodology or the Custom HTTP Header methodology, but the Secret Validation Token seemed quickest to implement and enough for the point of defending against a CSRF Attack.

The Secret Validation Token involves working in *admin/new.php* and *admin/index.php*. While inside of *admin/new.php*, a random variable is made and stored in the session. While inside of *admin/index.php*, that random variable is retrieved during the attack and is checked to see if *$nocsrftoken* is even set or if it matches. If the variable is not set or doesn’t match, there is a CSRF attack taking place and that will trigger the site. The demonstration can be viewed in Figure 21.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, website

Description automatically generated

**Figure 21: CSRF Prevention**

The code that was developed during this prevention can be viewed in Figure 22.

Graphical user interface, text, application

Description automatically generated

**Figure 22: Fixed Code for CSRF Attack**

## Further Revisions

Further Revisions could be found in both the *admin/edit.php*, *admin/index* and *admin/del.php* files. These scripts could allow CSRF attacks to happen. Therefore, additions were in need of making. Edit was just like new in that it was a *$\_POST* so the same randomization and check in the index file could be done. However, with delete being a *$\_*GET call, the randomization had to be placed in the index file, sent via the URL, and finally checked inside of the del file. By making these changes, the CSRF attacks could be stopped. All of those adjustments can be seen in Figure 27.

A picture containing text

Description automatically generated

**Figure 23: Successful add of a post**

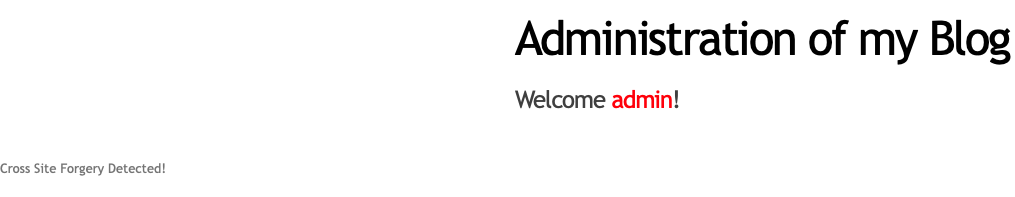
**A picture containing text

Description automatically generated**

**Figure 24: Successful edit of a post**

****

**Figure 25: Successful Deletion of a post**

****

**Figure 26: Still Successful Protection Against CSRF**

All changes made regard a successful admin account experience.

Graphical user interface, application

Description automatically generated

A picture containing graphical user interface

Description automatically generated

**Figure 27: Code Revisions Seen in del.php, index.php and edit.php**

# Task IV: SQL Injection Prevention

## SQLi in **admin/post.php**

Graphical user interface, text, website

Description automatically generated

**Figure 28: Execution of the SQLi Attack**

The reason this attack can happen in Figure 28 is because *id* is not protected inside of admin. The user asking doing this attack is not validated. Therefore, this needs to be protected by validating the user doing this attack.

Graphical user interface, text, website

Description automatically generated

**Figure 29: SQLi Prevention**

The prevention demonstrated in Figure 29 was done by disabling the ability to give user input in the first place. Union should not work now. By doing a *$prepared\_sql*, update can no longer allow union to work. The code for that is shown in Figure 30.

Graphical user interface, text, application

Description automatically generated

**Figure 30: Code Revisions in Update(..) for SQLi**

## Further Revisions

In Figure 30, a demonstration on editing a post that doesn’t exist doesn’t cause an error and union was not able to post 2 and 3, thus keeping the SQLi Prevention.

Graphical user interface, text, application, website

Description automatically generated

**Figure 31: Fixing edit through update by making sure it’s null (which it is)**

By going through all of *classes/post.php*, All methods besides the fix made all had protections. Whether that was integer assertion or escape protections. A snippet of this can be seen in Figure 32.

A picture containing text

Description automatically generated

**Figure 32: Further Analysis Snippet**