

**Fix Security Vulnerabilities for
the Hackazon Application**

Hayden Eubanks

School of Business, Liberty University

CSIS 486-D01

Prof. Backherms

December 3, 2023

Fix Security Vulnerabilities for the Hackazon Application

Introduction:

With the static scan of the source code for the Hackazon application complete, an analysis of mitigating the vulnerabilities discovered within the application can then be performed. One of the most prevalent vulnerabilities discovered throughout the application was vulnerability to cross-site scripting where a malicious actor could inject code into the page document allowing it to be executed by the browser (Simpson & Anthill, 2017). This highlights an extremely dangerous vulnerability as it can be exploited by a malicious actor to launch attacks against a database or the users of the site. Further, the analysis of the Hackazon application revealed vulnerabilities of two types of cross-site scripting (XSS) attacks being reflected and stored XSS attacks. In a reflected XSS attack, the code is sent by the attacker to the server where that code is then executed, and the results are reflected back to the attacker (OWASP, 2021). This vulnerability can be commonly observed in fields that echo the user's input back to the screen after it is submitted (IBM, n.d.) such as the search field within the Hackazon application. Stored XSS attacks then represent persistent XSS attacks where the malicious code is saved on the database, such as in a comment for a comment field, and then executed whenever that portion of the page is later accessed by a user (OWASP, 2023). This vulnerability can be observed within the review section of the Hackazon application and highlights another access point that must be addressed in resolving XSS vulnerabilities.

Fortunately, while XSS attacks mark a severe risk to the application, they are relatively easy to correct and can be mitigated through proper care in encoding both the input and output of variables to not include the special characters used in scripting attacks (PortSwigger, 2023). It is vital for a developer to consider encoding on both the input and output of data as both of these

points indicate a part of the process where the code could be executed. Additionally, the context within which the code exists must be thoroughly examined to ensure that proper encoding is used to mitigate the vulnerability (OWASP, 2023). For example, if the vulnerable code lies within a JavaScript portion of the code, then the JavaScript code can be executed in the browser before any encoding on the backend can take place (Microsoft, 2023). Without this understanding, the dangerous situation where a vulnerability that is believed to be covered still exists can occur leaving an application open to attack. Additionally, as PHP code addresses environment variables these variables must be encoded upon retrieval to ensure that a malicious actor cannot alter the values of these variables to contain malicious code (PortSwigger, 2023). Within the Hackazon application, vulnerabilities can be observed within the context of PHP, HTML, and JavaScript portions of code, and as such an understanding of the way that each of these areas process encoding must be performed. As PHP and JavaScript both enable dynamic webpage content, it is a best practice to, when possible, first place input into an HTML element that has its contents retrieved at runtime (Microsoft, 2023). This can make encoding easier and give the developer more control over the sanitizing of untrusted input. Context is an essential aspect of mitigating XSS attacks and as such a further analysis will be given to this area when evaluating solutions for XSS vulnerabilities within Hackazon.

Before and After for Code Segments:

Screenshot 1: Vulnerable Start File When retrieving Host Environment Variable

```

79 // To use express install, set to playerProductInstall.swf, otherwise the empty string.
80 var xiSwfUrlStr = "/swf/playerProductInstall.swf";
81 var flashvars = {
82   host: "<?php echo $_SERVER['HTTP_HOST'] ? 'http://'.$_SERVER['HTTP_HOST'] : $this->pixie->config->get('parameters.host'); ?>";
83 };
84 var params = {};

```

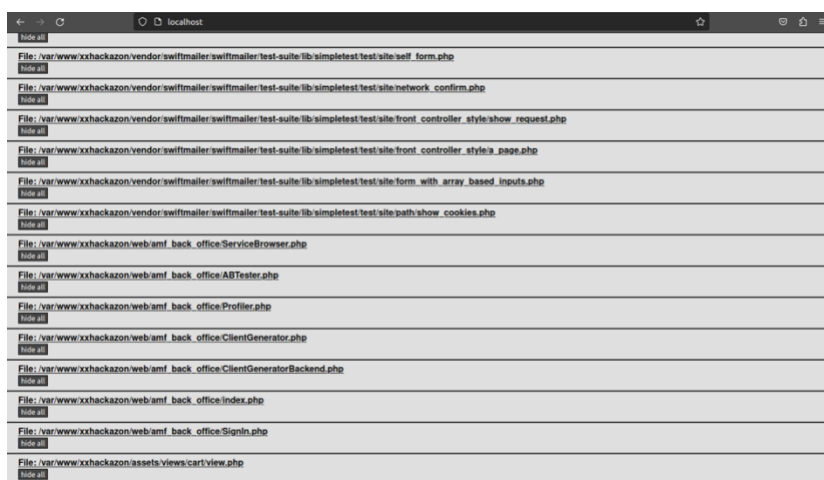
Screenshot 2: Encoding Characters From Host Environment Variable

```

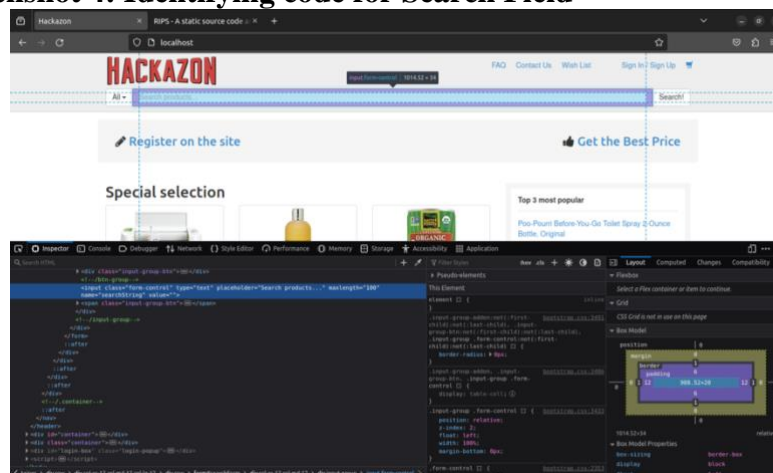
<script type="text/javascript">
// For version detection, set to min. required Flash Player version, or 0 (or 0.0.0), for no version detection.
var swfVersionStr = "11.1.0";
// To use express install, set to playerProductInstall.swf, otherwise the empty string.
var xiSwfUrlStr = "/swf/playerProductInstall.swf";
var flashvars = {
  host: "<?php echo htmlspecialchars($_SERVER['HTTP_HOST'] ? 'http://'.$_SERVER['HTTP_HOST'] : $this->pixie->config->get('parameters.host')); ?>";
};

```

Screenshot 3: Start file vulnerability no longer present in RIPS scan



Screenshot 4: Identifying code for Search Field



Screenshot 5: Vulnerable Input Field for Item Search

```

24     </ul>
25 </div>
26 <!-- /btn-group -->
27 <input type="text" class="form-control" placeholder="Search products..." maxlength="100" name="searchString" value="<?php $_($searchString); ?>">
28 <span class="input-group-btn">
29   <button class="btn btn-default" type="submit">Search!</button>
30 </span>
31 </div>

```


Screenshot 6: Encoding Input from Search Field

```

4 </div>
5 <!-- /btn-group -->
6 <input type="text" class="form-control" placeholder="Search products..." maxlength="100" name="searchString" value="<?php
7 $_(htmlentities($searchString, ENT_QUOTES, 'UTF-8')); ?>">
8 <span class="input-group-btn">
9   <button class="btn btn-default" type="submit">Search!</button>
10 </span>
11 </div>

```

Screenshot 7: Search Output Post Encoding


[FAQ](#)
[Contact Us](#)
[Wish List](#)
[Sign In / Sign Up](#)


All ▾

Search!

Screenshot 8: Vulnerable Sign In Field

```

Cross-Site Scripting
Userinput reaches sensitive sink when function () is called.
  73: echo echo 'var amfphpEntryPointUrl = "" . $config->resolveAmfphpEntryPointUrl() . "\n";
requires:

```

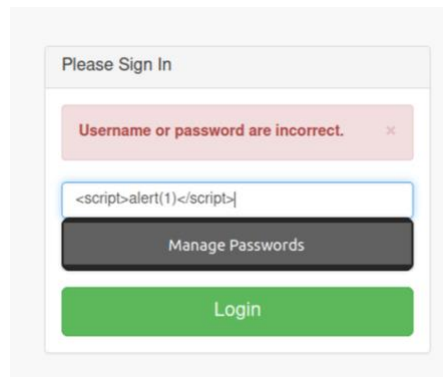
Screenshot 9: Fixed Sign-In Field

```

71 <?php
72 echo 'var amfphpVersion = "" . AMFPHP_VERSION . "\n";
73 echo htmlspecialchars('var amfphpEntryPointUrl = "" . $config->resolveAmfphpEntryPointUrl() . "\n");
74 if ($config->fetchAmfphpUpdates) {
75   echo "var shouldFetchUpdates = true;\n";
76 } else {

```

Screenshot 10: Proof of No Code Execution

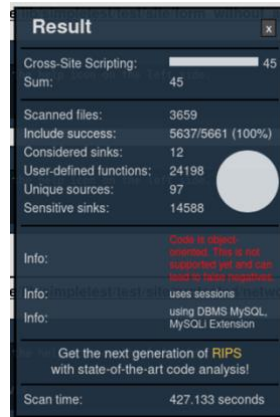


Screenshot 11: Fixed Self Form

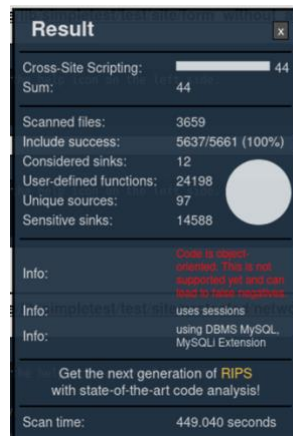
```
10 </form>
11 <p>[<?php print htmlspecialchars($_GET['visible']); ?>]</p>
12 <p>[<?php print htmlspecialchars($_GET['secret']); ?>]</p>
13 <p>[<?php print htmlspecialchars($_GET['again']); ?>]</p>
14 <form>
15 <input type="text" name="visible">
```

Updated RIPS Scan:

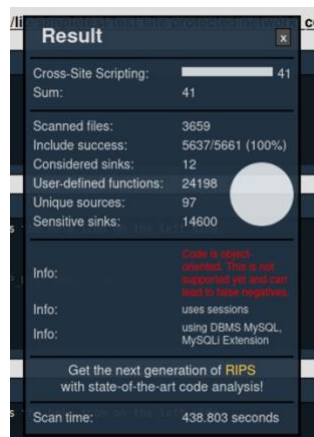
Screenshot 12: Initial RIPS Scan



Screenshot 13: RIPS Scan After First Vulnerability Fixed



Screenshot 14: Final RIPS scan showing 4 vulnerabilities that have been mitigated



Discussion of XSS Vulnerabilities:

As can be observed by the RIPS security scan ([Screenshot 12](#)), several XSS vulnerabilities exist within the Hackazon application and an analysis of these vulnerabilities as well as their solutions can assist a security professional in mitigating XSS within applications they oversee. Additionally, best practices exist for mitigating XSS attacks and following the principles outlined by these best practices can allow for XSS to be mitigated efficiently across an application's code (Microsoft, 2023). Because of this, the analysis of the vulnerable code within the Hackazon application will begin by addressing solutions to specific vulnerabilities before extrapolating principles and best practices that apply to mitigating XSS attacks as a whole. This analysis can then serve a security professional in evaluating vulnerable segments of code and further applying efficient mitigation strategies for their entire code set.

The first vulnerability within the Hackazon application to be examined can be observed within the start file that retrieves the HTTP host environment variable ([Screenshot 1](#)). This code segment reflects a portion of PHP code where the host variable is echoed out indicating the ability for an attack to occur. If a malicious actor were able to modify the host environment variable to contain malicious code, that code could then be executed upon the running of this PHP statement performing an XSS attack. This vulnerability can be seen to exist as non-sanitized environmental values are directly passed to the echo method meaning that no character encoding will occur (PortSwigger, 2023). This highlights the first principle of mitigating XSS attacks which is that all dynamic input from variables should first be sanitized by removing any problematic special characters that could be used in an attack (Microsoft, 2023). Within this specific instance, the variable was sanitized using the htmlspecialchars method ([Screenshot 2](#)) which is a PHP method used in encoding characters to a format safe for processing by HTML

(PHP, n.d.). For example, the “<” and “>” characters used in denoting HTML elements will be encoded to “<” and “>” respectively stopping the injection of elements and scripts into the page (IBM, n.d.). After sanitizing the variable with this method it can then be observed that the start page has been removed from the list of vulnerable pages denoted in the RIPS scan indicating that the variable has been successfully encoded ([Screenshot 3](#), [Screenshot 13](#)).

The next vulnerability to be examined involves the search field that accepts user input when searching for items on the site ([Screenshot 4](#)). Examining the code for this element, it can then be seen that this field submits user input through an HTML form, and as before this input is initially non-sanitized ([Screenshot 5](#)). The HTML input can then be sanitized by using the `htmlentities` method ([Screenshot 6](#)) which again removes any problematic characters (OWASP, 2023). However, an interesting behavior will be observed upon attempting to perform an XSS attack on this newly sanitized field. After submitting the form, the output reflected in the search bar appears to be filtered ([Screenshot 7](#)) but the injected code still executes proving that the vulnerability still exists. This then highlights the next principle to remember when approaching XSS vulnerabilities which is that the context within which the vulnerability exists is of extreme importance (PortSwigger, 2023). In this example, while the initial input was submitted through an HTML form, the search itself was carried out in the browser by a JavaScript method. JavaScript uses a different encoding scheme than HTML, and additionally, the JavaScript code is executed in the browser before the HTML encoding occurs (OWASP, 2023). In general, it is bad practice to directly process user input within JavaScript and it is better practice to first place the data in an HTML element whose contents are accessed at runtime (Microsoft, 2023). However, JavaScript variables can also be encoded using the Unicode format and many libraries such as `DOMPurify` contain functions that perform this encoding (PortSwigger, 2023). This example

identifies the importance of understanding the context of a vulnerable portion of code and security professionals should evaluate context wherever input is processed.

The remaining two vulnerabilities addressed in the screenshots ([Screenshot 8](#), [Screenshot 11](#)) address the vulnerable input field on the sign-in page as well as the referencing of PHP environment variables in the “self” file. These vulnerabilities follow the format of the two previous examples and likewise were mitigated by examining the context and applying the appropriate encoding methods to the PHP and JavaScript code ([Screenshot 9](#), [Screenshot 10](#)). After applying these solutions, rerunning the RIPS scan then indicates that all four of the addressed vulnerabilities have been mitigated in the application ([Screenshot 14](#)). Further, by observing the remaining vulnerabilities it can be seen that the remaining portions of code vulnerable to XSS could be mitigated through applying the appropriate encoding methods discussed above to all areas where variables are processed.

Keeping in mind the principles drawn from these examples, further best practices can be applied to efficiently mitigate XSS vulnerabilities throughout the scope of an application. The first of these best practices is to apply the principles of validating input and encoding output wherever variables are used (PortSwigger, 2023). Further, it is a best practice to apply the strategy of whitelisting approved characters over backlisting disapproved characters as this best ensures that only expected characters are used in the field (PortSwigger, 2023). Attacks can be introduced by several methods and this multilayered approach mitigates vulnerabilities by ensuring that the entry and exit points of processing are sanitized. Additionally, as observed in this application, a multilayered approach must be applied in accordance with the context by which the variable is used. This means ensuring the proper encoding takes place on each level of execution such as in an HTML form as well as the underlying JavaScript event (OWASP, 2023).

With this in mind, another aspect of context that must be addressed is the protocol in which the web application exists. Using secure protocols such as HTTPS can stop a malicious actor from embedding malicious code in a URL that is returned at different points on the page and therefore stop malicious code execution through URL manipulation (PortSwigger, 2023).

However, manually applying this filtering to every use of a variable is inefficient and may be more error-prone as a user could easily miss a vulnerable instance of code. To negate this risk and more efficiently protect a site from XSS, a dynamic template engine such as those found in Twig, Freemarker, Jinja, and React can be used to apply an encoding template across a site (PortSwigger, 2023). Using the appropriate syntax for each of these engines can vastly simplify the encoding process and better ensure that a high level of encoding is performed at each vulnerable portion of code. Finally, a content security policy (CSP) can be enforced to mitigate what an attacker could do should they surpass all of these mitigation techniques (PortSwigger, 2023). A CSP dictates rules regarding aspects such as the ability for inline scripts to be executed or for external scripts to be loaded and reducing these capabilities could mitigate the damage an attacker could perform should they bypass all other controls.

As can be seen in the Hackazon application and best practices regarding the mitigation of XSS, most of the mitigation strategies involve proper care and awareness when coding websites. Additionally, these vulnerabilities can be seen as relatively easy to mitigate and, in most instances, only require a function call to an encoding method. As a security professional, this then highlights the importance of educating developers about the dangers of XSS as well as the proper mindset when approaching mitigation. Cultivating a culture of validating input and encoding output can vastly increase a site's security against XSS attacks (PortSwigger, 2023) and remove these security defects earlier in development when they are less damaging and cheaper to

correct (Simpson & Anthill, 2017). XSS attacks are a serious threat facing web applications (OWASP, 2021) and by following best practices security professionals can enable their teams to mitigate XSS vulnerabilities and strengthen the security of their applications.

References

IBM. (n.d.). *Protect from cross-site scripting attacks*. IBM.

<https://www.ibm.com/garage/method/practices/code/protect-from-cross-site-scripting/>

Microsoft. (June 26, 2023). *Prevent cross-site scripting (XSS) in ASP.NET core*. Microsoft.

<https://learn.microsoft.com/en-us/aspnet/core/security/cross-site-scripting?view=aspnetcore-8.0>

OWASP. (2021). *OWASP top ten: Top 10 web application security risks*. OWASP.

<https://owasp.org/www-project-top-ten/>

OWASP. (2023). *Cross site scripting prevention cheat sheet*. OWASP.

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

PHP. (n.d.). PHP manual: *htmlspecialchars*. PHP.

<https://www.php.net/manual/en/function htmlspecialchars.php>

PortSwigger. (October, 2023). *How to prevent XSS*. PortSwigger. [https://portswigger.net/web-](https://portswigger.net/web-security/cross-site-scripting/preventing)

[security/cross-site-scripting/preventing](https://portswigger.net/web-security/cross-site-scripting/preventing)

Simpson, M. T., Anthill, N. (2017). *Hands-on ethical hacking (3rd ed.)*. Cengage Learning.

<https://ng.cengage.com/static/nb/ui/evo/index.html?deploymentId=5681612456081340358553076923&cISBN=9781337271721&id=1937025370&snapshotId=3720027&>