Inventory Management System
Cross-Site Scripting Vulnerabilities Analysis
Technical Procedure and Results Report

Francesco Tescari

February 2020

Table of contents

Summary

The analysis of the `PHP` based Inventory Management System (IMS) web application is performed with the goal of discovering and fixing Cross-Site Scripting (XSS) vulnerabilities. The approach used consists in a static taint analysis of the source code to spot unsanitized, tainted and potentially harmful variables sent in the responses of the web application, followed by a manual review of the identified vulnerabilities and the classification of those in true positive and false positive ones. After the assessment of the presence of a true XSS vulnerability, which is achieved with the development of a proof of concept (PoC) exploit, we create a test case to replicate the exploit using `Java` and the `Selenium` web driver. Finally, we fix the found vulnerabilities maintaining the functional properties of the original web application.

The analysis detected a total of 72 critical and exploitable XSS vulnerabilities, 69 are stored XSS and the other 3 are reflected XSS.

The static taint analysis is performed using Pixy, an open source tool that can be used to discover SQL and XSS vulnerabilities on `PHP` based applications. The tool was not updated in the last 7 years and thus we manually update it to support the latest functions of `PHP 7.0` in order to perform a valid and sensible analysis (section 1).

We track the taint status flow of the stored XSS vulnerabilities by saving the dependencies between the reported vulnerabilities, the input sanitization, the data types and the output sanitization, in separate CSV files, which are then joined using an ad-hoc python script to get different variants of the same reported vulnerability, with their dependencies on the database. Thanks to this script, we can filter out the majority of the false positive vulnerabilities with an efficient and accurate semi-automatic process. This process identifies a total of 108 stored XSS vulnerabilities, 39 of which are flagged as false positives, while the other 69 require an additional manual analysis and seeking of a PoC exploit (section 2).

The PoC exploits are developed by intercepting and modifying the HTTP request using a proxy, then injecting a specific XSS payload, depending on the sink statement context. We create a helper class of functions to improve the efficiency of the realization of PoC exploits and the related test cases for stored XSS vulnerabilities. The assessment of the presence of a vulnerability is achieved with a challenge-response scheme in order to reduce false positives and making sure that the injected payload is also interpreted by the browser. A test case replicating the PoC exploit for each vulnerability is developed in `Java` using the `Selenium` web driver (section 3).

All the vulnerabilities are fixed using the `htmlentities` function at the sink statement. For the API files vulnerabilities, we decided to enforce the encoding of the output values even if they are already JSON encoded, in order to mitigate possible vulnerabilities generated by the improper usage of those values in the client-side Javascript source. We finally execute the test cases and a new Pixy analysis to assess the success of the fixing process of all of the 72 identified vulnerabilities (section 4).

1 Analysis approach: Pixy

There are several approaches to the analysis of a digital service in order to find vulnerabilities: for the analysis of the Cross-site scripting (XSS) vulnerabilities of the Inventory Management System (IMS) web application, we decide to use Pixy. Pixy is an open source software, originally developed by Jenad Jovanonic, used to perform a static taint analysis on `PHP` source code files, to find XSS or SQL Injection vulnerabilities. Pixy follows the dependencies between the taint status of the variables in order to find where tainted, and potentially harmful variables are printed to the HTTP response by the server. Pixy finally generates a report that is saved in form of dependencies flow graphs from the source of the tainted status to the sink, one for each potential vulnerability found.

1.1 Updating and configuring Pixy

The last update to the source code of the Pixy tool was made in 2013 and thus Pixy is an outdated tool, therefore we need to manually update it in order to reach a usable level. The main problems are the:

- Unresolved functions: Pixy doesn't know all the built-in functions of the latest `PHP 7.0` and will mark any unknown function as possibly tainted, creating lots of false positive vulnerability reports;
- Unmodeled functions: Pixy doesn't know the behavior (the transfer function) of some of the built-in functions and thus it will mark them as possibly tainted, creating even more false positives;
- Special dependencies: the taint status of the output of `json_encode` depends on all of the sub-elements of the input array. This special kind of dependency was not foreseen by the developer, hence the original Pixy doesn't mark the output of `json_encode` as tainted even if if one of the sub-elements is tainted, resulting in a false negative (risky!).

In the case of the source code provided we have the following problems:

- Unresolved: `json_encode`, `mysqli_fetch_all`;
- Unmodeled: `json_encode`, `mysqli_fetch_all`, `mysqli_fetch_assoc`, `mysqli_fetch_array`, `mysqli_fetch_row`, `mysqli_query`, `mysqli_num_rows`, `mysqli_error`, `mysqli_insert_id`.

To update Pixy we add the unresolved functions to the built-in functions list in the Java source and the unmodeled functions to the model configuration file (`/config/model_xss.txt`). Finally, we change the internal logic of the tool to add the special dependencies needed by the `json_encode` function by modifying the Java source in the dependency graph section.

The configuration of Pixy is not finished yet: we have to modify again the configuration file in order to set the database policy: by default, the policy is set to "clean database", assuming that the database contains untainted values and thus the results coming from the database are safe to be printed. This is not the scenario of this security analysis, in fact we cannot make that assumption. We will hence create a "tainted database" configuration, by changing the values of the functions `mysqli_fetch_*` from 0, always untainted, to 4, always tainted.

1.2 Pixy versions comparison

We can compare the output of the original Pixy analysis versus the updated one: the first one produces 76 possible vulnerabilities and the output graphs contain unresolved and unmodeled errors. The second one produces 52 vulnerabilities with none of the previous errors, providing more accurate graphs of the tainted code flow and reducing the number of false positive vulnerabilities that need to be checked manually.
Therefore, the rest of the analysis will be based on the output of the updated and improved version of Pixy.

2 Pixy vulnerabilities classification

In this section we will present the results of the initial analysis of the possible vulnerabilities detected by Pixy and their dependencies. Before continuing with the report we have to notice the following fact: each Pixy vulnerability is reporting a unique sink statement where the data is printed, but there might be different sources of the tainted data for a single sink. Because of each source of the vulnerability can be a true positive or false positive and has to be sanitized independently, we would rather consider those as different vulnerabilities: we will call the those "variants" of the single Pixy vulnerability and we will label them with the Pixy name plus an incremental char suffix (A, B, C, ...)

The second step of the security analysis is the manual verification and classification of each of the vulnerabilities graph generated by Pixy. We need to check if each vulnerability variant is a true positive or false positive. To do that we take two different approaches depending the source of the tainted data:

- Reflected XSS: the source of tainted data is in the request: (e.g. `$_GET` or `$_POST` variables);
- Stored XSS: the source of taitned data is in the database (e.g. `mysqli_fetch_*` functions).

2.1 Reflected XSS

To analyze the possible reflected XSS vulnerabilities, we simply follow the flow of the tainted values in the source code and verify whatever there is some kind of sanitization applied. If no or weak sanitization is applied, we try to attack the vulnerability by crafting a possible XSS paylaod, depending on the context and position of the sink statement. The payload will be injected by crafting a malicious request, with the payload inserted in one of the `$_GET` or `$_POST` parameter, in the request path or an `HTTP` header.

2.2 Stored XSS

To analyze the possible stored XSS vulnerabilities, we need to track the entire flow of the possible tainted data: check whatever it is sanitized before the insertion in the database, verify that the database data-type is compatible with a XSS payload and check whatever the data is sanitized after the query from the database. To perform this whole verification, we used two CSV files to track different parts of the data-flow:

- xss_vulnerability_dependency.csv: Each vulnerability reported by Pixy is associated to the columns of the database where the tainted data of the vulnerability is taken from (the source of the tainted data), classified either in the sanitezed or unsanitized column of the CSV file, depending whenever the data is sanitized before reaching the sink.
- xss_database_dependency.csv: Each column of the database which is referred by the first CSV file is associated with its data type and, if the datatype can contain an XSS payload, the source file where unsanitized, user-controlled data is stored in such column.

Once we gathered all of this information, we use a simple, ad-hoc python script to join the two csv structures and get the results of this second analysis: each variant of the Pixy vulnerabilities associated to the tainted columns of the database that are used as source of data and also associated the source files where the unsanitized input is inserted.

The result of this further filtering of the stored XSS vulnerabilities is the following: from the 49 stored-XSS vulnerabilities reported by Pixy, we extracted a total of 108 vulnerability variants. 39 of those are false positive, meaning that the data is either sanitized before insertion, saved as a non-string format or sanitized after the retrieval of the data from the database. The remaining 69 might be true positive.

The reflected XSS vulnerabilities reported by pixy are only 3, hence we end up having 72 total possible true vulnerabilities to manually test in the next section.

2.2.1 Example of stored XSS filtering process

Table 2.1: Example of xss_vulnerability_dependency.csv content

Vulnerability	Sanitized entries	Unsanitized entries
XssFetchCategoriesPhp1Min		categories.categories_id, categories.categories_name
XssProductPhp3Min		brands.brand_id, brands.brand_name
XssProductPhp4Min		categories.categories_id, categories.categories_name

Table 2.2: Example of xss_database_dependency.csv content

Database entry	Data type	Unsanitized inserts
categories.categories_id	INT	
categories.categories_name	VARCHAR	createCategories.php, editCategories.php
brands.brand_id	INT	
brands.brand_name	VARCHAR	createBrand.php, editBrand.php

Table 2.3: CSV data of the previous examples joined using the ad-hoc python script

Vulnerability variant	Database vector	XSS injection points	Positive
XssFetchCategoriesPhp1MinA	categories.categories_id		FALSE
XssFetchCategoriesPhp1MinB	categories.categories_name	createCategories.php, editCategories.php	TRUE
XssProductPhp3MinA	brands.brand_id		FALSE
XssProductPhp3MinB	brands.brand_name	createBrand.php, editBrand.php	TRUE
XssProductPhp4MinA	categories.categories_id		FALSE
XssProductPhp4MinB	categories.categories_name	createCategories.php, editCategories.php	TRUE

3 Proof of concept and test cases

The ultimate way to prove the presence of a vulnerability is to develop a proof of concept (PoC) exploit, therefore we do that for each one of the 72 vulnerabilities that we have to test.

3.1 Exploiting stored and reflected XSS

A PoC exploit can be manually found by intercepting and modifying the requests to the server using a proxy, for example Charles or ZAP. If we successfully exploit one vulnerability, we then formalize the procedure by creating an automated test case that replicates the PoC attack. The PoC exploits for reflected and stored XSS vulnerabilities are different:

- Reflected XSS: the vulnerability is identified in the response of the malicious request, hence we can just intercept and modify one request to find the PoC.
- Stored XSS: the payload must be first injected into the database with one request. Only then we can identify the vulnerability in a separate, non-correlated request. Therefore we need to craft at least two requests each time to create a PoC attack.

3.2 XSS payload types and presence assessment

There are different XSS payloads depending on the context of the sink statement: tainted values might be printed inside HTML tags but also inside HTML attributes, requiring a different payload. The main types of payloads used in this analysis, with the relative contexts, are showed in the table 3.1.

To assess the presence of a payload, we decided to follow a challenge-response scheme: we generate a payload that have the "id" attribute set to a random value, then we can be certain that a specific PoC attack was successful by searching an element with that specific id in the DOM tree of the browser.
This approach has two benefits:

- PoC attacks and the relative tests becomes independent one from the other: multiple payloads might be inserted in the same response, the challenge-response scheme identifies each of them and makes sure that you are assessing the presence of the intended one.
- Because we are searching an element by id using the browser functions, we are sure that the element was not only inserted in the response, but also interpreted as HTML by the browser.

We choose to use the tag `<script>` because it should never be allowed.
An example of the assessment process is the following:

- Generate a random challenge identifier: for example `xss_a7e53c89b781`
- Craft the malicious request(s) and inject the following example payload:
  `<script id="xss_a7e53c89b781"></script>`
- Check the presence of an HTML element with id:
  `document.getElementById("xss_a7e53c89b781");` using the browser or
  `webDriver.findElement(By.id("xss_a7e53c89b781"));` using selenium

After the analysis of the 72 possible vulnerabilities we find that all of them are exploitable with the previous assessment process, meaning that the website is vulnerable to critical XSS exploits in 72 variants. The PoCs are replicated in `Java` test cases, using the `Selenium` web driver library.

Table 3.1: Types of payloads used in the test cases, depending on the context

Type	Context	Payload
Plain payload	$	<script></script>
Double quote attribute		" ><script></script> <p id="
Single quote attribute		' ><script></script> <p id='
Option/Select escape	<select><option>$</option></select>	</option></select><script></script><select><option>

4 Fixing the vulnerabilities

To fix the vulnerabilities we need to sanitize the tainted variables before they reach the sink statement. The sanitization is achieved by using the function `htmlentities` provided by `PHP`. It is important to notice that in the cases where the tainted value is printed within a single quotes attribute (see table 3.1), we also need to specify the function flag `ENT_QUOTES` in order to prevent the escape from the attribute.

4.1 JSON API fixing policy

The vulnerabilities in the JSON API files can be fixed alternatively by specifying the right content-type header, which is `application/json`, without the need of applying the sanitization. There are both pros and cons if we decide to only modify the header or rather decide to sanitize the JSON content anyway:

- Using only the content-type header: the vulnerability is fixed on the API endpoints, the response is not interpreted as HTML and therefore scripting is disabled. However the values encoded in the JSON might be inserted in the HTML without being sanitized by some improper JavaScript code.
- Applying sanitization anyway: the vulnerability is fixed both on the endpoint and on the client-side. However there might be cases of double sanitization made in Javascript or some client-side scripting features (e.g matching a string) problems. Also, if in the future someone wants to use the same APIs without inserting the result as HTML, for example in a mobile application, it might become difficult to handle HTML encoded data.

Because the application of the first solution requires a second analysis on all of the client-side Javascript source code, and because the negative effects of the second solutions should not be visible with the standard usage of the web application, we decide to apply the latter, sanitizing the tainted values even in the APIs. We fixed the content-type header too, because it's good practice and provides a second layer of security on the APIs endpoints. This policy choice can be changed easily by not applying the fix patches to the API files though.

4.2 Fixed application results

After fixing the 72 XSS vulnerabilities (69 stored, 3 reflected), we tested the patched website using the developed test cases. All of them failed, suggesting that the vulnerabilities have been fixed successfully. We decided to run a final Pixy analysis on the patched code and we got output graphs that correspond only to the false positives of the first analysis, which remain false positives.

Even though the target of this analysis is XSS vulnerabilities, we can state that other kinds of security deficiencies have been found throughout the analysis process, such as SQL injection or inadvisable structure and data types in the database.