Report of Security Testing of Schoolmate

Date of delivery: 28/08/2017
Security Reviewer: Lorenzo Angeli - 183847 - lorenzo.angeli@unitn.it
Beneficiary: Mariano Ceccato

1. Taint Analysis

The first taint analysis run, carried out through Pixy, identifies more than a hundred XSS vulnerabilities, grouped in 71 reports. The **`vulnerabilities.pdf`** annex details all vulnerabilities and their classification between false and true positives.

Ultimately, the only false positives appear to be reports 2-10, 53, 321, and some vulnerabilities in report 92. All other vulnerabilities are true positives, mostly reflected XSS attacks, and 7 stored XSS vulnerabilities. Most reflected XSS vulnerabilities affect the same variables: `page`, `page2`, `selectclass`, `delete`, and `onpage`.

2. Security Test Cases

Test cases have been grouped by application user: Admin, Teacher, Parent, and Student. These are defined as subclasses of the more generic User class.

Each test case, named as the corresponding PHP file name plus the report number (e.g. AddAnnouncements16.java), is defined as a subclass of one of the four Users, the one that can perform/is vulnerable to the attack.

In this way, instancing a test case runs the constructors of the User class and of their specific user. The constructors set the base URL, entry point and username/password combination, as well as some shared helper methods such as a method to add a fictitious submit button to forms.

Additionally, the User class defines some generic tests for the `page`, `page2`, `selectclass`, `delete`, and `onpage` variables, which account for the majority of the vulnerabilities in the application. These generic test cases have been always tried as the first option on vulnerabilities affecting those variables. In this way, the test cases are more modular and less prone to mistakes due to the high degree of similarity of the vulnerabilities.

All generic test cases except for the one on the `delete` variable also execute an assertion to check that the test case landed on a properly-formatted page without returning, for example, SQL errors. While this is mostly not necessary for the purpose of this exercise or for practical reasons, the injection appears to be more elegant when conducted this way. When the page resulted to be broken anyway, an ad-hoc test case has been used which skips this extra assertion.

All JWebUnit test cases are attached in the **`tests/`** folder annex, and manual step by step proof-of-concept attacks can be found in annex **`vulnerabilities.pdf`**.

3. Fixes

The process of fixing the codebase results, in a nutshell, in the application of sanitization functions (`intval` and `htmlspecialchars`) to all variables that are `POST`ed by the user to the webpage.

Due to how the application is structured, with a single entry point in `index.php` which then *`include`*s/*`require`*s the various subpages, a quick fix to ensure that no malicious value is echoed would be to sanitize with htmlspecialchars all variables in the `$_POST` array at the beginning of the `index.php` file.

The proposed solution, however, approaches the problem more systematically, sanitizing all outputs, DB inputs, and query results (even though this last step might be superfluous) in all files. Depending on the specific deployment scenarios of the application, sanitizing the input can be sufficient to block stored XSS vulnerabilities, but if the DB is not under control of the same administrator that has access to the application's codebase, it could already contain malicious values. Sanitizing outputs, however, also ensures that no reflected XSS attacks can be conducted.

The fixes for the reflected XSS vulnerabilities have been conducted, largely, in batch through find/replace commands.

Before proceeding for the fixes, the vulnerability reports have been sorted by vulnerable variables. This helped greatly in identifying which vulnerabilities were identical, and also helped with applying and verifying the batch fixes.

Since all vulnerabilities pertaining the page, page2, `selectclass`, onpage and `delete` variables are of the same type (the value is used as-`POST`ed without sanitization, but is supposed to contain an integer), wrapping them in the `intval` function fixes the vulnerabilities without going through each occurrence. Vulnerabilities pertaining to other variables have instead been fixed on a case-by-case basis, either by applying `intval` or by applying `htmlspecialchars`.

Annex **`vulnerabilities.pdf`** details the specific fixes that have been deployed to fix the vulnerabilities.

4. Testing

All test cases fail on the assertion that checks for the vulnerability on the original version Schoolmate, and pass on the fixed version.

After the fixes, running Pixy again only reports the false positive vulnerabilities.

No further vulnerabilities have been added as side effects of the fixes

5. Steps to reproduce results

- Install apache, php and MySQL
- Create DB `schoolmate` with username `schoolmate` and password `schoolmate`
- Import the `tests/` folder in a Java IDE, including JWebUnit in the classpath
- Run the DB script `initSchoolmate.sql`
- Run all tests (they will be ran on the fixed version). All tests should pass
- Run the DB script again
- On file `User.java` comment line 18 and uncomment line 19
- Run all tests again (this time on the original version). All tests should fail

6. Annexes

- Script to populate the application's DB with baseline data `initSchoolmate.sh`
- Test cases in JWebUnit `tests/`
- Fixed application `schoolmate/`
- List of all vulnerabilities with classification, description, proof-of-concept attacks and fixes `vulnerabilities.pdf`
- (extra) Script to convert the output of Pixy to jpg images `dot2jpg.py`
- (extra) Script to re-run Pixy and update jpg graphs (mostly to check that batch fixes worked and didn't have side effects) `updateGraphs.sh`
- (extra) Link to the GitHub repository for the project that also highlights the folder structure that was used in the process: https://github.com/RaelZero/sectesting