Report of Security Testing of Schoolmate

Date of delivery: 19 May 2017

Security reviewer: Davide Pedranz

Beneficiary: Mariano Ceccato

1. Taint Analysis

The result of the Static Taint Analysis of the XSS vulnerabilities present in the SchoolMate application are reported in Table 1 in attachment. Please note that the analysis does not take into account SQL Injection vulnerabilities, which may allow an attacker to modify the database structure and possibly bypass some of the security measures implemented by the application against XSS attacks. To fix this issue, all SQL queries should be rewritten using prepared statements.

2. Security Test Cases

The vulnerabilities reported by Pixy are identified by a unique number. This number is used as the identifier for the table with the result of static Taint Analysis and the test cases. In case multiple vulnerabilities are reported in the same file (generated by Pixy), a letter is appended to distinguish the single vulnerabilities. Table 2 shows some examples of this naming convention.

Table 2: vulnerabilities naming convention

Pixy file	Taint Analysis	Test case
xss_index.php_2_min.jpg	2	Vulnerability2.java
xss_index.php_11_min.jpg	11.a	Vulnerability11a.java
	11.b	Vulnerability11b.java
	11.b	Vulnerability11b.java

All test cases are grouped in the "suite" package. The "utilities" package contains some utilities used to perform the tests. The test cases can be run using Gradle. Please checkout the *README.md* file in the security tests folder for detailed instruction on how to run the test cases.

In addition to the **_min** reports generated by Pixy, some of the **_dep** reports have been analysed (underlined in yellow in the vulnerabilities table in attachment). Pixy failed to convert some **_dep** to a **_min** report, missing some vulnerabilities. Due to the huge number of such files, they have been ignored for this assignment, as agreed in class.

3. Fixes

The vulnerabilities can be classified 4 groups, described in Table 3. All vulnerabilities in a given group can be fixed in the same way, as described in the table.

Table 3: classification of the XSS vulnerabilities

ID	Type	Description	Fix
1	Reflected XSS	The index.php uses the value $_POST[page] to select the right PHP script to load (eg. admin or teacher reserved area). In particular, the value is used as the argument of a switch statement and checked against integer values. However, the PHP typing system automatically casts the string to an integer just for the switch statement. The original string is then printed in each page in an hidden form field. This allows an attacker to send an arbitrary XSS vector as a POST parameter and have it displayed in the HTML on each page, without any sanitisation. The XSS vector should begin with a number that matches the pages the attacker wants to load, followed by any arbitrary string. An example of XSS vector is the following string: **2 '> <script>alert('xss')</script> <span id=‘xss**	Use a regular expression to extract only the number at the beginning of the string and discard the rest. It is possible to use the build in function **intval** to safely convert a string to an integer value.
2	Reflected XSS	This vulnerability is identical to #1, but the variable involved is $_POST[page2]. The variable $_POST[page] is used to select the main function to load, then the variable $_POST[page2] is used to select the right subpage. Each page generated by the application prints the variable as an hidden form field, without any sanitisation. The same type of attack described in #1 applies.	Same as #1.
3	Reflected XSS	A variable taken from the $_POST array is printed as HTML content without any sanitisation. This allows an attacker to inject arbitrary HTML content. Sometimes the value is used inside one or many SQL queries (without any sanitisation), so the attacker must craft a vector able to make SQL queries syntactically valid and perform the XSS attack. An example could be: **1 -- ' -- > <script>alert('xss')</script> <span id='xss** where the **-- ' --** is used to terminate the SQL queries, while the rest of the string is used to perform the XSS attack.	Use the PHP function **htmlspecialchars** to sanitise the user input before using them. Additional measured should be taken to prevent SQL injections, but this is outside the scope of this document.
4	Stored XSS	The application stores inputs form the user in the SQL database, without any validation both in the insert/update and the select statement. This allows an attacker to store arbitrary HTML content in the database, which is then displayed by the application in some other page without any validation. Most of the columns in the database are limited to 15 characters, which makes the attack difficult, but still possible. More advances attacks could exploit SQL injection vulnerabilities to alter the database scheme and allow longer strings to be stored, in order to mount much more effective XSS attacks.	Sanitise each string variable under control of the user with the PHP function **htmlspecialchars** before using it. The check should be done when storing the string in the database, but since the database could potentially already contain XSS vectors, we suggest to perform the check also before using any string extracted from the database.

4. Testing

All vulnerabilities reported by Pixy have been fixed. All security test cases run against the original code successfully managed to mount an XSS attack, while the same attack fails against the fixed code.

The Static Taint Analysis ran on the fixed code produces 8 vulnerabilities reports, which result to be all false positives, as shown in Table 4. The reported tainted variables were also revealed by the analysis on the original SchoolMate code: since the false positives tainted variables have not been changed in the fixed code, Pixy reports them again. The analysis shows that the security measured put in place are effective to prevent XSS attacks. Pixy did not reported any additional vulnerability: the fixed were correct and did non introduce vulnerable statements.

Table 4: Result of the Static Taint Analysis for XSS Vulnerabilities on the fixed application

Report	File	Class.	Motivation
2	maketop.php	FALSE	The statement prints a string extracted from the database. Since all writes to the given database field are sanitised properly against XSS attacks, the attacker has no way to store malicious content.
3	Identical to entry #2.		
4			
6			
10			
44	header.php	FALSE	The statement prints a string extracted from the database. Since all writes to the given database field are sanitised properly against XSS attacks, the attacker has no way to store malicious content.
88.a	ManageSchoolInfo.php	FALSE	The statement prints a string extracted from the database. Since all writes to the given database field are sanitised properly against XSS attacks, the attacker has no way to store malicious content.
88.b		FALSE	The statement prints only a variable extracted from integer field in the database, which can not be used as a XSS vector.
88.c		FALSE	The statement prints only a variable extracted from integer field in the database, which can not be used as a XSS vector.
321	ReportCards.php	FALSE	The statement generates a PDF with the data from the database. This may be vulnerable to attacks against the PDF programming language (eg. including JavaScript code in the document), but this is outside the aim of this analysis.

Annexes

1. Table 1 with the results of the static taint analysis
2. SQL scripts to populate the database for the tests
3. Security test cases implemented in JWebUnit, as a standalone Gradle project
4. Edited code of the subject application, with vulnerabilities fixed