Security testing report

Gianluca Bortoli
DISI - University of Trento
Student id: 179816
gianluca.bortoli@studenti.unitn.it

1. INTRODUCTION

This work aims at performing a security analysis study on *Schoolmate*. This web service is PHP/MySQL solution for elementary, middle and high schools where different type of users can manage all the needed information to fulfill their job.

This report is structured as follows. Section 2 describes the naming convention used for the test cases and how they are structured into packages. Section 3 describes the vulnerabilities that have been found on the application, their root causes and how to fix them. Section 4 reports the results of the proof-of-concept (PoC) attacks described in Section 2. Finally, Section 5 depicts the steps that have to be taken in order to be able to run all the tests from scratch, while Section **??** gives a general evaluation of the whole work.

2. SECURITY TEST CASES

Pixy is a scanner static code analysis tools that scans PHP applications for security vulnerabilities. This software is used in order to spot the possible vulnerabilities in *Schoolmate*, the subject web application.

The workflow followed developing this work is the following:

1. run Pixy on the application code to produce vulnerability reports regarding Cross Site Scripting attacks (XSS).

2. manually analyze all the reports to classify the vulnerabilities. Thus, each of them has to be categorized as **false positive**, **reflected XSS** or **stored XSS** (see *xss_classification.pdf* attachment).

3. write all the test cases for all the true positives (TP) found (ie. the vulnerabilities classified either as reflected or stored XSS) in the form of a PoC attack against the application under test.

4. check that every test case fails on the bugged application to demonstrate *Schoolmate* has such security vulnerabilities.

5. modify the *Schoolmate*'s PHP code to fix the vulnerabilities found in step 2.

6. run again the test suite and check all the tests pass, meaning that the PoC attack implemented in each test case is no longer possible.

7. run again Pixy on the fixed application source code to check the vulnerabilities are really fixed and making sure step 5 did not introduce any security breach.

The test suite developed to demonstrate that the application under revision has security concerns makes use of JWebUnit, a Java-based testing framework for web applications. Every Pixy report is identified by a number and may contain multiple vulnerabilities. The test suite's source code is structured as follows:

- the **src/main/java** folder contains a Java file for each Pixy report which has at least one TP. The test-case Java file naming follows the convention *Test<PixyReportNumber>.java* so that it is possible to easily match the test case with its Pixy report counterpart.

- the **src/main/java/common** folder contains three packages that implement utility functions that are available to all the test cases. These functions allows not to soil the actual test case implementation, separating all the sequences of operations that has to be done in many tests in order to navigate the web page depending on the situation.

A noteworthy aspect is how tests' assertions are written. When dealing with security PoC test cases, the security expert writes a test that is thought to fail if the vulnerability can be exploited and pass otherwise. Hence, in this scenario a failing test (and its assertion) means that the vulnerability is still present, while a passing one means that the vulnerability is not present any more. For the seek of generality all the test cases can run in any order. To achieve this, every test cases involving a stored XSS vulnerability has a *cleanup* function called after the test case execution which removes potentially dirty data from the database.

Furthermore, most of the tests perform three types of action on the web page. They can *navigate* through the pages of the web application clicking buttons and links, they can *input* some values in the available forms and text fields and finally they *make assertions* about the content of the page (eg. that a malicious link is not present in the page).

Last but not least, some of the test cases (ie. the ones involving the variable *page*2) required to modify the structure of the Document Object Model (DOM) of the webpage, since the submit buttons of the forms automatically set values for some of the form's hidden fields using some javascript code connected to the *onClick* action. To overcome this issue, a new ad-hoc submit button is created inside the form and

this one is used to perform the action instead of the original one. This functionality is implemented inside the *src/common/utils.java* file in the *addSubmitButton* function.

3. SOURCE CODE FIXES

All the vulnerabilities explained in Section 2 are exploited in the test cases by means of inserting a malicious link that appears on the page. This succeeds because the application does not perform any kind of validation and/or sanitization on the variables that are shown on the page. Unfortunately, many of these variables are (or contained) input under the control of the user. Thus, performing sanitization steps is not an option in this case.

The sanitization phase can be performed either before the user input is stored in the MySQL database or when the value is retrieved from the storage to be displayed on the web page. This is up to the developer and highly depends on the application structure. To fix the PHP source code of *Schoolmate*, this work makes use of the first approach. Recalling that the *index.php* file is the one handling every request coming from every other page in the application and is the one including all the other ones depending on the values of *page* and *page*2.

This architectural decision can be exploited in order to fix all the vulnerabilities affecting the entire application in a single page. The portion code shown below is the one responsible for the sanitization step of all the POST parameters reaching the application.

Listing 1: Fixing all the vulnerabilities in index.php

```
<?php
foreach ($_POST as $k => $val) {
 $_POST[$k] =
    htmlentities($val, ENT_QUOTES, "UTF-8");
}
...
?>
```

The decision to apply the **htmlentities** function to all the items in the POST array allows to eliminate the possibility to modify the page's DOM by means of malicious tags in the input text fields that are later printed on the page. This function converts all the characters that has a special meaning in HTML to its "safe" encoded version. The optional parameter *ENT_QUOTES* is used to convert both double and single quotes, while the *UTF-8* specities the encoding to be used when converting the characters. Finally, the addition of this code snippet as the first lines of the *index.php* file allows to fix both the stored and the reflected XSS vulnerabilities in *Schoolmate*, since only the XSS vulnerabilities are of interest for this work.

4. CONCLUSIONS

The test cases developed for this work and explained in Section 2 covers all the Pixy vulnerabilities classified as true positives. All the tests failed when the original code of *Schoolmate* is used to run the application, while they all successfully pass after the fixes explained in Section 3.

In order to check if the fixed code does not introduce any new vulnerability and does not contain any vulnerability at all, the Pixy static analyzer is run again on *Schoolmate*. After this final step, Pixy revealed the same vulnerabilities as if the code is still vulnerable. All the PoC tests successfully pass after patching the application, but it is possible to claim that all the reports created by Pixy are false positives. This is be due to the fact that static anaysis tools perform an overestimation of the paths that can be followed during an actual execution of the application[1]. Thus, based on the outcome of the test suite, it is possible to say that the vulnerabilities are fixed.

5. PRELIMINARY STEPS

Some preliminary steps has to be followed in order to run all the test cases:

1. install and run a webserver (eg. Apache), PHP and MySQL.
2. edit the $*dbuser* and $*dbpass* values in the *index.php* file in order to match the ones of the MySQL instance to be used.
3. copy the *Schoolmate* source files inside the webserver directory.
4. import the *dbDump.sql* script in the MySQL instance. It takes care of creating the a database with the right name and populating it with some initial values. All the queries on the database should succeed.

After all the abovementioned steps are completed successfully, *Schoolmate* is up and running and the test suite can be run.

[1]As explained in Section 2, this is the reason why an initial dinstinction between TP and FP has to be done in advance before implementing the PoC attacks.