Security Testing
XSS vulnerability assessment

Nicolò Fornari

July 6, 2016

Abstract

Flow analysis is a general static analysis framework that can be instantiated in several specific code analyses, among which taint analysis. Taint analysis aims at keeping track of tainted variables (i.e. variables containing unsanitized user input) along the execution paths of a software. In this report I describe how I conducted a vulnerability assessment, limited to XSS vulnerabilities, using *pixy*, a static taint analysis tool, on the open-source web application *Schoolmate*.

Report layout

The vulnerability assessment is divided in the following steps:

1. Verify Vulnerabilities
2. Proof of Concept injections
3. Fix the vulnerabilities
4. Test the fixed vulnerabilities

In Section [1 I list the software used for the vulnerability assessment.
In Section [2] I introduce Schoolmate and I explain how it works.
Section [3] is about the application of taint analysis with *pixy*.
Section [4] regards test cases: I describe briefly JWebUnit by commenting a single test case.
In section [6 I show how I fixed the vulnerabilities in Schoolmate.
Finally in Section [7] I illustrate how I ensured the vulnerabilites were truly fixed.

1 Technical setup

For this vulnerability assessment I used the following software:

- Eclipse 3.8.1
- External JARs:
  - JUnit-4.12
  - JWebUnit-core 3.2 [5]
  - Hamcrest-core-1.3
  - Jwebunit-htmlunit-plugin-3.2 [5]
  - Mysql-connector-java-3.1.14 [6]
- Schoolmate 1.5.4 [1]
- Pixy 3.03 [2]
- Graphviz [3]
- Webscarab [7]

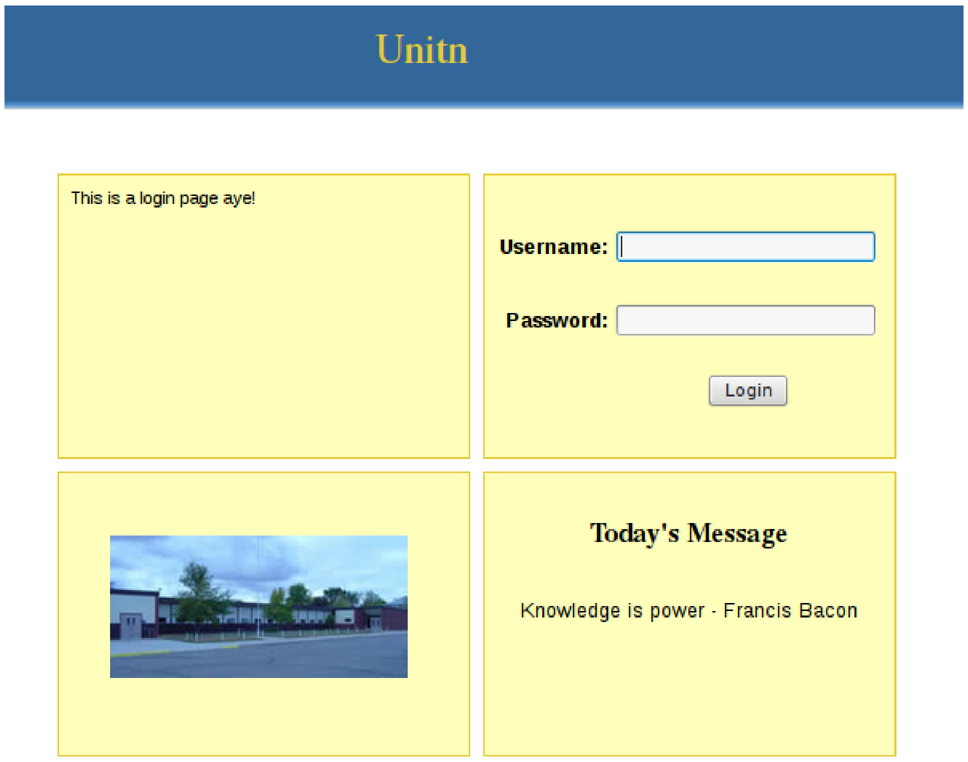
2 Schoolmate

Schoolmate is an open source web-platform for school management: it is aimed at teachers,students and parents. This software has a natural and logical division in four kind of users:

- **Admin:** the Administrator has clearly more rights than every other user: he can add (or edit or remove) users, classes, semesters, terms, announcements.
- **Teacher:** the scope of the teacher is limited to his own classes: he can create assignment and grade students. Moreover he can visualize some information about the school and the announcements of the administrator.
- **Parent:** while the teacher is limited to his classes, a parent is limited to his children, of whom he can see the school career.
- **Student:** students, similar to parents, can look at their own school career selecting a class at time.

2.1 Page navigation

The most important page in Schoolmate is *index.php* as it manages all the requests. If index.php does not receive any input it displays a login page as shown in [1].


Unitn

This is a login page aye!

Today's Message

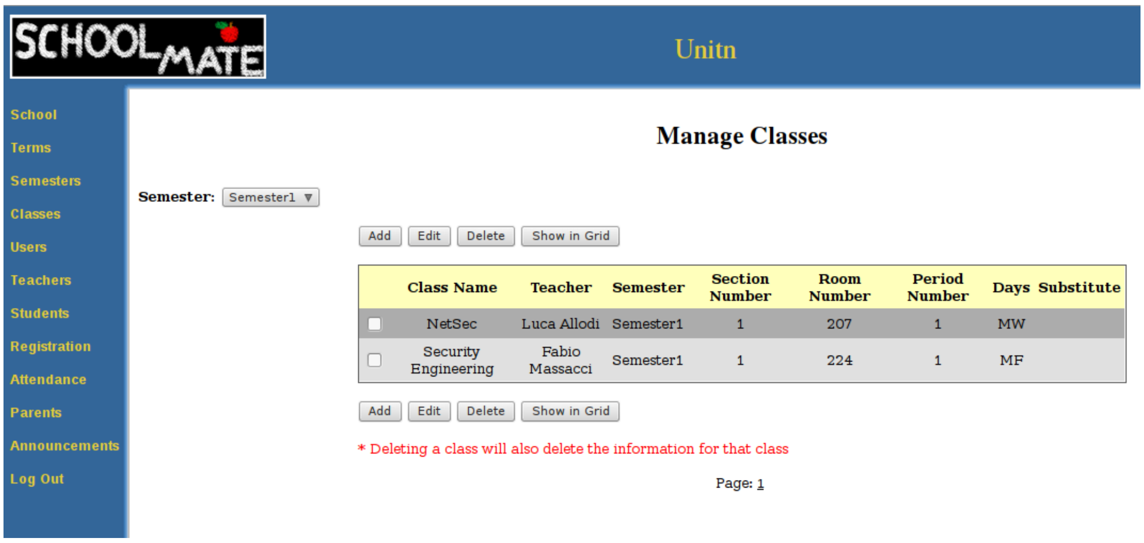
Knowledge is power - Francis Bacon

Figure 1: Default view for index.php

There are two fundamental variables for page navigation: *$page* and *$page2*. The first one is used to determine which "section" the user belongs to: depending on the value of $page the appropriate php file is required. Here are the values used in the switch case by index.php

$page	*require_once*
0	Login.php
1	AdminMain
2	TeacherMain
3	SubstituteMain
4	StudentMain
5	ParentMain

The second variable $page2 is used to navigate thorugh the pages within a section. Consider for example a user logged-in as admin, the front-end looks like in [2]



Unitn

Manage Classes

Semester: Semester1

	Class Name	Teacher	Semester	Section Number	Room Number	Period Number	Days	Substitute
☐	NetSec	Luca Allodi	Semester1	1	207	1	MW	
☐	Security Engineering	Fabio Massacci	Semester1	1	224	1	MF	

* Deleting a class will also delete the information for that class

Page: 1

Figure 2: User logged in as admin

On the left hand side there is always a navigation form: when the user clicks on a link, a javascript function assigns the proper value to $page2 before the form is submitted. Note that $page and $page2 are accessible as hidden fields in the page.

$page2	Link	require_once
1	School	ManageSchoolInfo.php
6	Terms	ManageTerms.php
5	Semesters	ManageSemesters.php
0	Classes	ManageClasses.php
⋮	⋮	⋮

2.2 Make it running

The first step is to download Schoolmate at [1]. Since this web application is written in PHP and relies on MySQL for database queries, I installed Apache, MySQL and PHP under an Ubuntu machine.

Remark 1. *The code of Schoolmate is affected by multiple vulnerabilities so I configured Apache to listen only to localhost with:*

```
"Listen 127.0.0.1:80" in /etc/apache2/ports.conf
```

With Schoolmate up and running the second step is to populate the database by creating some users (at least one for type), classes,semesters,etc. After doing so there are around 70 .php files which need to be analyzed.

3 Taint analysis with pixy

Pixy is a tool for static taint analysis. Unfortunately not all vulnerabilities reported by pixy are true positives due to the conservative approach of the tool. There can be different reasons for this behaviour:

· Tainted variables may suffer from constraint such as limited input which make injection attacks infeasible

· Tainted variables could be traced along unfeasible path

· Variables are labelled as tainted because they come from an *untrusted source* (eg. a database)

It is fundamental to check all the reported vulnerabilites and classify them in true and false positives. First of all I downloaded pixy at [2] and I run it with the command:

```
$ ./pixy/run-all.pl -a -A -g -y xss schoolmate/index.php -o project/graphs/
```

which produces multiple graphs with extension *.dot*. In order to handle these files easier I converted them into *.jpg* using *dot*, a command from the *graphviz* package [3].

```
$ dot -O -T jpg *.dot
```

Then I renamed the generated files, which I found too verbose, with the following:

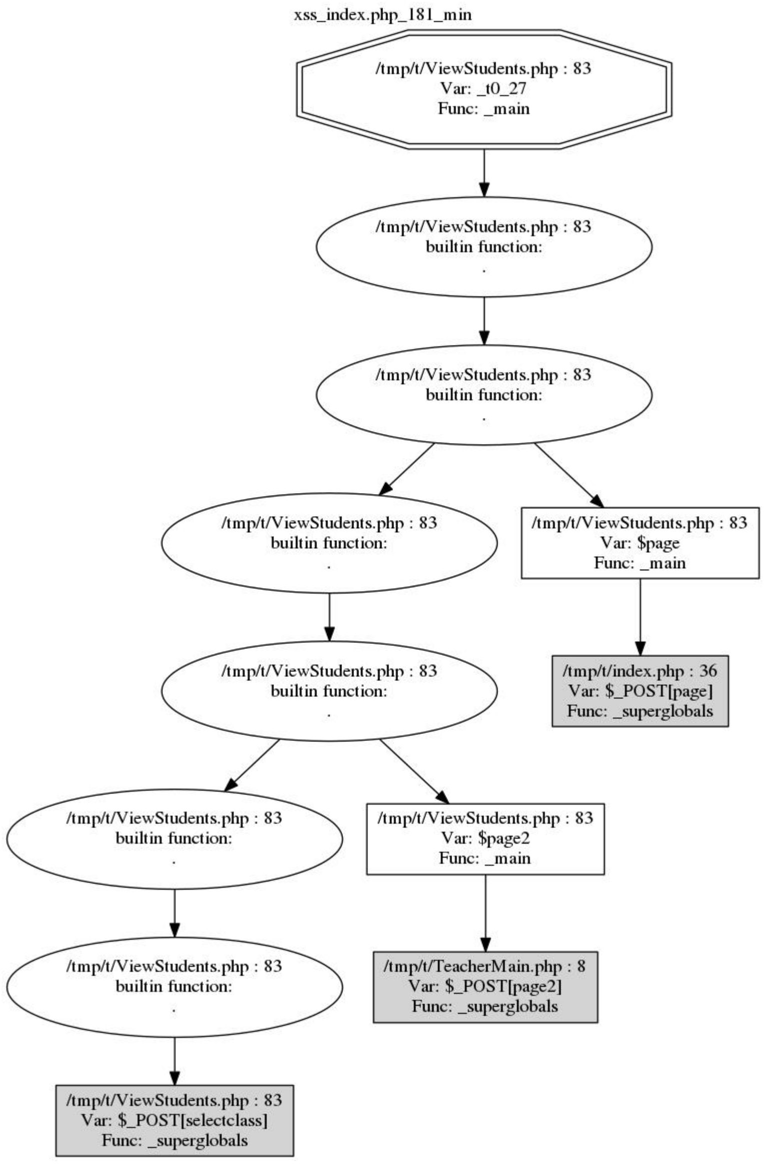
```
$ for f in *.jpg; do mv "$f" "‘echo $f | sed s/xss_index.php//‘"; done
```

Finally I obtain a set of numbered graphs in jpeg format.

Remark 2. *Notice that each .php page in the graph has the full path specified - see [3]. This feature makes the images really wide if Schoolmate is placed in a default directory like* /var/www/html/schoolmate*. Rather than modifying the source code of pixy I choose a far simpler workarond which is running pixy on the files saved in a shorter directory, namely* /tmp/t

Having generated and customized the necessary files the next step is to use them. Let's consider the following graph as example:

Figure 3: ViewStudents.jpg



Each graph has to be read bottom-up: the leaves of the treee are the entry point, the paths show the variable's flow along the pages. In this case there are three tainted variables: *selectclass*, *page2*, *page* with entry points respectively *ViewStudents.php* at line 83, *TeacherMain* at line 8 and *index.php* at line 36.

4 Test cases

After having performed taint analysis with pixy I have obtained a graph for each vulnerability. As already discussed not all vulnerabilities are true positives, it is then necessary to test all of them manually to check whether an attack is feasible or not. To achieve a systematic approach I wrote an automatic test case for each true positive using JWebUnit [5].

4.1 JWebUnit

JWebUnit provides a high-level Java API for navigating a web application combined with a set of assertions to verify the application's correctness. This includes navigation via links, form entry and submission, validation of table contents, and other typical business web application features.

```java
package it.unitn.nick0.security.testing.student;

import it.unitn.security.testing.utilities.Methods;
import org.junit.Test;

public class StudentMain extends it.unitn.security.testing.BaseStudent {
        @Test
        public void fieldPage() {
                this.selectClass();
                tester.setWorkingForm("student");
                tester.setTextField("page", "4"+this.pattern1);
                tester.clickLinkWithExactText("Settings");
                tester.assertMatch("Class Settings");
                tester.assertLinkNotPresentWithText(this.assertInjection);
        }

        @Test
        public void fieldPage2() {
                this.selectClass();
                tester.setWorkingForm("student");
                tester.setTextField("page2", "1"+this.pattern1);
                Methods.newSubmitButton("/html//form[@name='student']",tester);
                tester.submit();
                tester.assertLinkNotPresentWithText(this.assertInjection);
        }
}
```

Listing 1: Example of test case - StudentMain.java

Every test case performs three different kind of actions:

· page navigation: eg. *ClickButtonWithText, ClickLinkWithText*

· input insertion: eg. *SetTextField, SetHiddenField*

· assertions: eg. *AssertMatch, AssertLinkNotPresentWithText*

There are a lot of variants of these methods however they are not always sufficient: look again at [1]. At line 22 a new button is created calling a static method *newSubmitButton* which takes as arguments an *Xpath* string and *tester*. Why is this necessary? Page2 is an hidden field in a form, it is not sanitized however when the default button for submitting the form is clicked, a constant value is assigned to page2 through javascript. To circumvent this ”protection” mechanism it is sufficient to create a new button to submit the form without triggering the javascript check.

Annotations

Test methods must be annoted by the *@Test* annotation, see for example line 8 and 18. However it is possible to use the annotation *@Before* and *@After* to execute methods before or after all the tests methods. This turns out to be really useful: in my project I created a method *prepare* which creates a WebTester, sets the base url and then logs the user in.

```java
@Before
public void prepare() {
        tester = new WebTester();
        tester.setBaseUrl("http://localhost/sec-test/schoolmate");
        tester.beginAt("/index.php");
        setCredentials();
        tester.setTextField("username",username);
        tester.setTextField("password",password);
        tester.submit();

        tester.assertTitleEquals("SchoolMate - Unitn");
}
```

Listing 2: *prepare* method of TopClass.java

This method is present in *TopClass.java*, a super class which every test case extends. Actually a test case extends either *BaseAdmin*,*BaseTeacher*,*BaseParent* or *BaseStudent* which then extend TopClass. One advantage is that the *prepare* method is always the same because the credentials for the different users are changed by calling the *setCredentials* method which is overwritten by the subclasses of TopClass.

In my project I also used the *@After* annotation to restore the database to the state it was before the injection of a stored XSS. Except for one test case I directly queried the database as in this example:

```java
@After
public void Restore() {
        Integer status = Database.queryDB("DELETE from assignments WHERE title = ’Foo’");
        System.out.println("Status of restore assignments: "+status);
}
```

Listing 3: *restore* method of ManageAssignment.java

5 Additional vulnerabilities

While working on the project I noticed that pixy did not find all the vulnerabilites present in the code. Consequently I had to look for new vulnerabilities but I needed a systematic approach. I focused on stored XSS vulnerabilities and my idea was to look at the database structure and identify all the table fields of type *varchar* or *text*. In this way I knew in advance which pages of schoolmate I could attack.

Figure 4: Database tables

```
mysql> show tables;
+----------------------+
| Tables_in_schoolmate |
+----------------------+
| adminstaff           |
| assignments          |
| courses              |
| grades               |
| parent_student_match |
| parents              |
| registrations        |
| schoolattendance     |
| schoolbulletins      |
| schoolinfo           |
| semesters            |
| students             |
| teachers             |
| terms                |
| users                |
+----------------------+
15 rows in set (0.00 sec)
```

For every table in [4] I considered all the fields of type *varchar* and *text* and listed them in [5]. I highlighted in bold the fields for which I wrote a test case, these fields are the only one of type text. As a matter of fact all the other fields are vulnerable but the number of injected charaters is very limited: either 15 or 20 (password is of length 32 but it does not account as the hash of the input is stored). I am not saying that an attack is not feasible however chances of success are little. Consider the following different attack scenarios:

- **Link injection:** a short link like <a href="goo.gl">A</a> has length 22.
- **Browser crashing:** the shortest javascript code to write a loop is $for(;;)alert()$ and it has 14 characters, however at least the <script> open tag is needed, for a total of 22 characters.

Figure 5: Summary of exploitable fields

Table	Field	Type
adminstaff	fname	varchar(20)
adminstaff	lname	varchar(15)
assignments	title	varchar(15)
assignments	**assignmentinformation**	text
courses	coursename	varchar(20)
courses	sectionnum	varchar(15)
courses	roomnum	varchar(5)
courses	dotw	varchar(5)
grades	**comment**	text
parents	fname	varchar(15)
parents	lname	varchar(15)
schoolbulletins	title	varchar(15)
schoolbulletins	**message**	text
semesters	termid	varchar(15)
semesters	title	varchar(15)
students	fname	varchar(15)
students	lname	varchar(15)
teachers	fname	varchar(15)
teachers	lname	varchar(15)
Terms	titles	varchar(15)
Users	username	varchar(15)
Users	password	varchar(32)

Being careful with asserts
For generality I assumed that test cases do not run in order. It could then happen the following situation:

1. A first test case executes a stored XSS injecting a *Malicious Link*
2. A second test case executes an XSS, the injection fails but the *assertMatch* does not as it finds *Malicious Link* from step 1

To overcome these false positives, when dealing with stored XSS, I chose to inject urls different from the default one: ”<a href=”http://unitn.it”>Malicious Link</a>”.

6 Fixing vulnerabilities

In Section [2] I briefly explained how Schoolmate works. Recall that *index.php* handles every single requests: it requires or includes the other php pages depending on the values of *$page*,*$page2* and some other variables. This architecture offers a great advantage as *index.php* receives all the POST variables. It is sufficient to sanitize the whole POST array with the following code:

```php
<?php

foreach ( $_POST as $key => $value ) {
    $_POST[$key] = htmlentities($value, ENT_QUOTES , "UTF-8");
}

...
?>
```

Listing 4: index.php - code to fix all the variables at once

Remark 3. *The option* ENT_QUOTES *is used to convert also the single quotes which are not modified by the default option* ENT_COMPAT. *See [8]*

Remark 4. *There are different options to sanitize the input. I chose* htmlentities *which is a stronger version of* htmlspecialchars.

7 Conclusions

After having sanitized the input the successive step is to verify whether the vulnerabilities are fixed or not. First of all I have re-run all the test cases and noticed with great pleasure that all the tests passed, meaning that the vulnerabilities were fixed. Furthermore I re-run pixy on the fixed version of Schoolmate but for reasons which are not clear to me it returned exactly the same results. However I can claim that the vulnerabilities found are fixed based on the results of my test cases.

8 References

1. https://sourceforge.net/projects/schoolmate/
2. https://github.com/oliverklee/pixy
3. http://www.graphviz.org/
4. https://sites.google.com/site/sectestunitn
5. https://sourceforge.net/projects/jwebunit/files/
6. http://dev.mysql.com/downloads/connector/j/3.1.html
7. https://sourceforge.net/projects/owasp/files/WebScarab/
8. http://php.net/manual/en/function.htmlentities.php