UNIVERSITÀ DEGLI STUDI DI TRENTO

Report submitted in Partial Fulfilment of the Course

Security Testing

Università degli Studi di Trento
EIT Digital Master of Science in Security and Privacy

https://sites.google.com/site/sectestunitn/home

Taint Analysis using Pixy
Case study:schoolmate

Table of Contents

Introduction

Taint Analysis is a method that tries to identify variables that have been “tainted’ with user controllable input and traces them to possible vulnerable functions known as a sink. If the tainted variable gets passed to a sink without first being sanitised, it’s flagged as a vulnerability. It is a popular method which consists to check which variables can be modified by the user input. All user input can be dangerous if it is not properly checked. Taint analysis can be seen as a form of flow analysis. If the source of the value of the object X in untrustworthy we say X is tainted. To taint user data is to insert some kind of tag/label for each object of the user data. The tag allows us to track the influence of the tainted object along the execution of the program.

Tools

- Schoolmate

A vulnerable web application that is susceptible to cross site scripting as validation of user input is missing

- Pixy

Automate the static source code analysis of the vulnerable web application schoolmate representing the values in the form of graphs with tainted values and sensitive sinks.

- JWebUnit

An open source tool that replicates the behaviour of the webpage to automate the dynamic audit with the using test cases to inject a malicious link into the reported

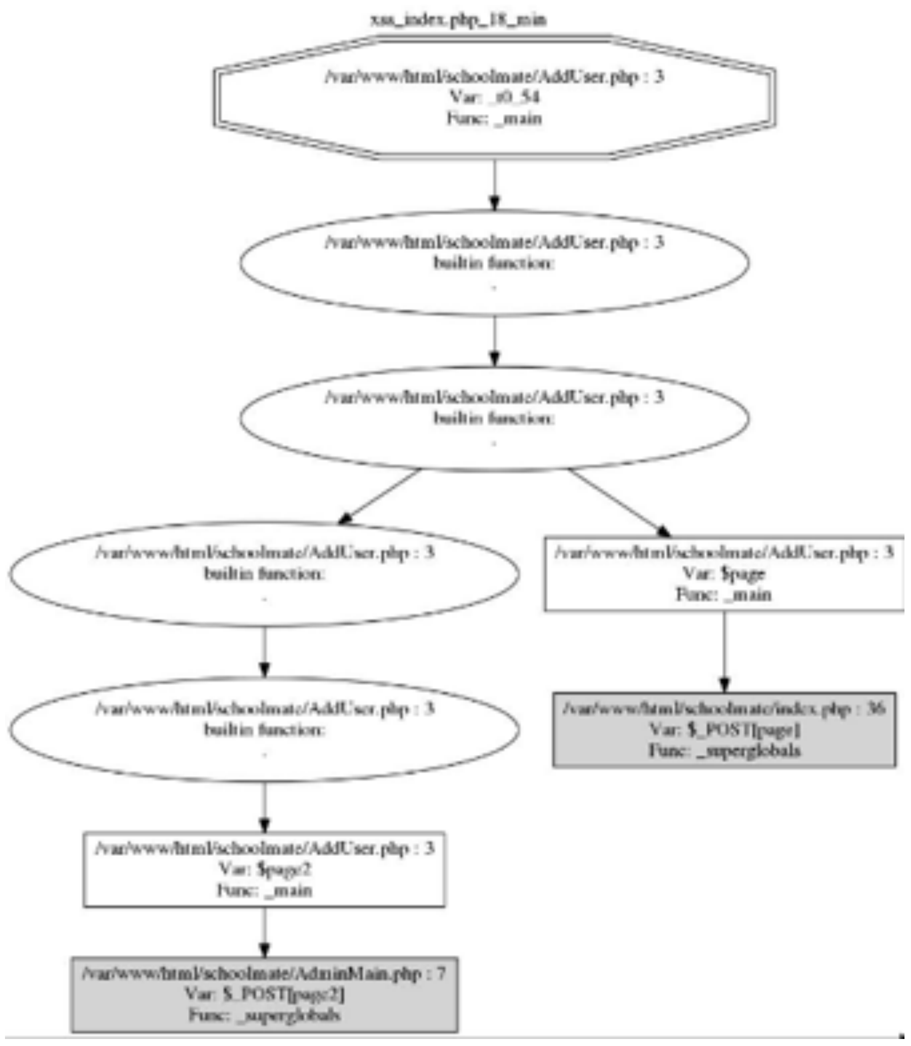
- WebScarab

Acting as a proxy for actual manipulation of the web-traffic simulating the actions of an attacker injecting a malicious link. This was used alongside checking the source code of the browser.

Vulnerability Report Results

Pixy Graphs False Positives/True Positives

To identify false positives vs true positives. Pixy prints the sinks or vulnerable statements at the top. Following a concatenation of strings to the bottom where the data for the tainted variable is actually input into the page. This could from client side, a $_POST value from the client html request or on the server side or as variables supplied from the database. Since we are using php, variables are stored as $variable x . Please see example below of the data flow graph. - see Graph 1.0


xss_index.php_18_min
/var/www/html/schoolmate/AddUser.php : 3
Var: _t0_54
Func: _main
/var/www/html/schoolmate/AddUser.php : 3
builtin function:
.
/var/www/html/schoolmate/AddUser.php : 3
builtin function:
.
/var/www/html/schoolmate/AddUser.php : 3
builtin function:
.
/var/www/html/schoolmate/AddUser.php : 3
Var: $page
Func: _main
/var/www/html/schoolmate/AddUser.php : 3
builtin function:
.
/var/www/html/schoolmate/index.php : 36
Var: $_POST[page]
Func: _superglobals
/var/www/html/schoolmate/AddUser.php : 3
Var: $page2
Func: _main
/var/www/html/schoolmate/AdminMain.php : 7
Var: $_POST[page2]
Func: _superglobals

Graph 1.0

Note: All 71 graphs for all vulnerabilities reported have been included in the folder security testing under file pixy

<u>Conservative approximation of taint analysis</u>

Vulnerabilities can be considered as false positives because of the following reasons

- Tainted Values are propagated on infeasible paths
- Constraints on values could make injection impossible for example numeric values
- Database values are considered tainted by pixy but it is not feasible to inject tainted data in the database

The following are some of the false positives identified from the schoolmate case study;-

1.Vulnerability Identified by Pixy
xss_index.php_2_min.jpg, xss_index.php_3_min.jpg, xss_index.php_4_min.jpg, xss_index.php_6_min.jpg, xss_index.php_10_min.jpg, xss_index.php_53_min.jpg, xss_index.php_92_min.jpg

$Reported Tainted Variable
$schoolname

False Positive-Yes
Reason: $schoolname is sanitised with a validation statement htmlspecialchars with the update sql function/query in the database.

2. Vulnerability Identified by Pixy
xss_index.php_92_min.jpg

$Reported Tainted Variable
$numperiods

False Positive- Yes
Reason: $numperiods is an integer and is supplied by the database so it is not feasible to inject tainted data in the database.

3.Vulnerability Identified by Pixy
xss_index.php_92_min.jpg

$Reported Tainted Variable
$numsemesters

False Positive- Yes
$numsemesters is an integer and is supplied by the database so it is not feasible to inject tainted data in the database.

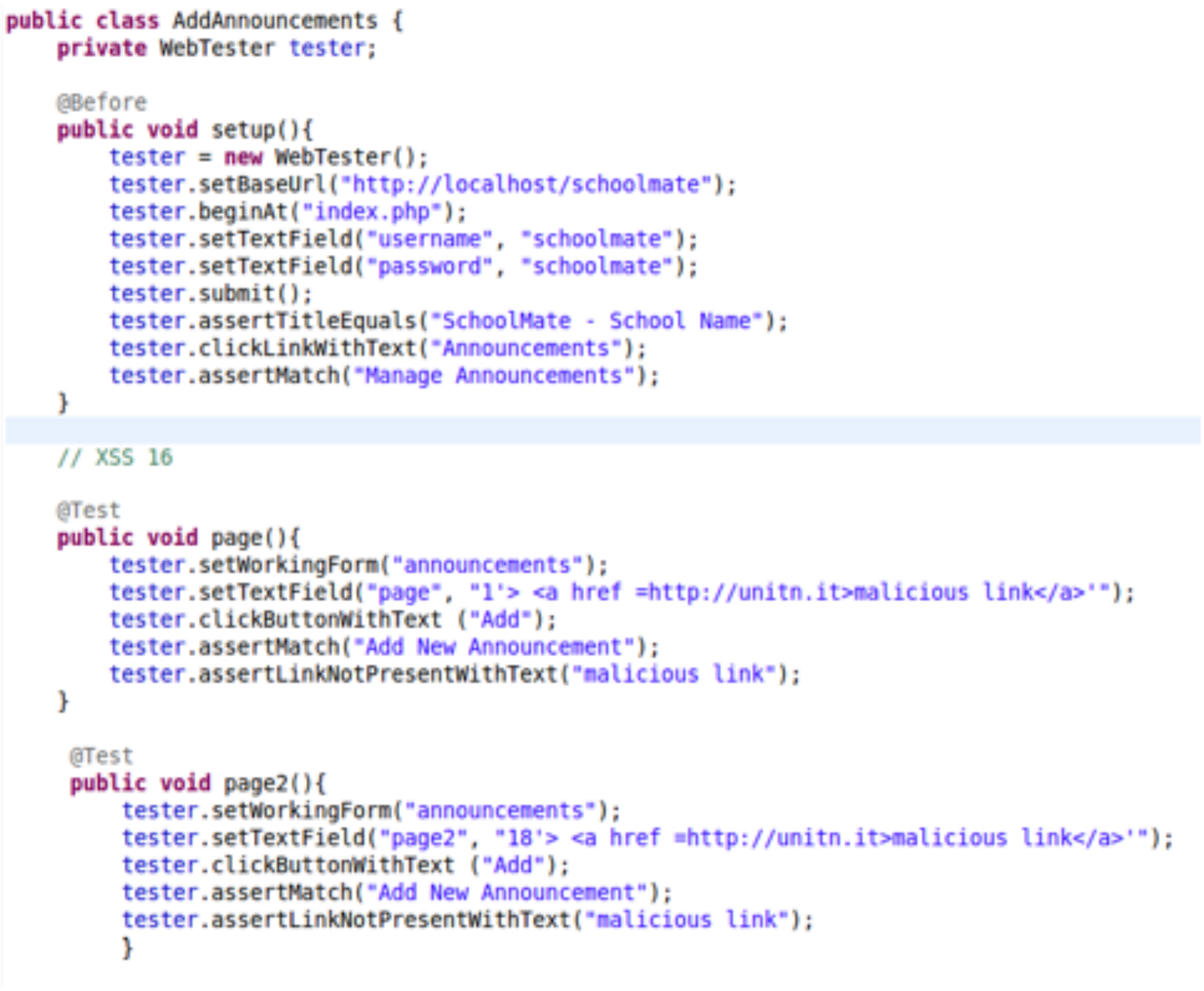
Note: The summary of true and false positives has been included in the folder security project testing under file vulnerabilities.xls

Proof Of Concept Injection

Attacks implemented with JWebUnit

JWebUnit is used to simulate the behaviour of the browser. I crafted a malicious link that I injected into a true positive vulnerability identified by Pixy. For reflected XSS attacks, I crafted a malicious link which is reflected on that particular page after injection. Please see an example below of a proof of concept attack for a reflected XSS attack.Some of the pages are vulnerable to persistent cross-site scripting. For these pages, we create a separate “After” case in JWebUnit to restore the page to it’s original state. We use the appropriate syntax to log into the page as if we were a user and mimic the behaviour of the browser using it’s source code. WebScarab can be quite handy here as well.

For xss_index.php_16_min.jpg



```
public class AddAnnouncements {
    private WebTester tester;

    @Before
    public void setup(){
        tester = new WebTester();
        tester.setBaseUrl("http://localhost/schoolmate");
        tester.beginAt("index.php");
        tester.setTextField("username", "schoolmate");
        tester.setTextField("password", "schoolmate");
        tester.submit();
        tester.assertTitleEquals("SchoolMate - School Name");
        tester.clickLinkWithText("Announcements");
        tester.assertMatch("Manage Announcements");
    }

    // XSS 16

    @Test
    public void page(){
        tester.setWorkingForm("announcements");
        tester.setTextField("page", "1'> <a href =http://unitn.it>malicious link</a>'");
        tester.clickButtonWithText ("Add");
        tester.assertMatch("Add New Announcement");
        tester.assertLinkNotPresentWithText("malicious link");
    }

     @Test
     public void page2(){
         tester.setWorkingForm("announcements");
         tester.setTextField("page2", "18'> <a href =http://unitn.it>malicious link</a>'");
         tester.clickButtonWithText ("Add");
         tester.assertMatch("Add New Announcement");
         tester.assertLinkNotPresentWithText("malicious link");
         }
```

Note: The summary of all proof of concept cases has been included in the folder security testing under file test cases.

<u>Fixing The Vulnerabilities</u>

The tainted variables reported by pixy were similar across several pages i.e $_POST(page), $_POST(page2), $_POST(onpage) and $_POST[selectclass]

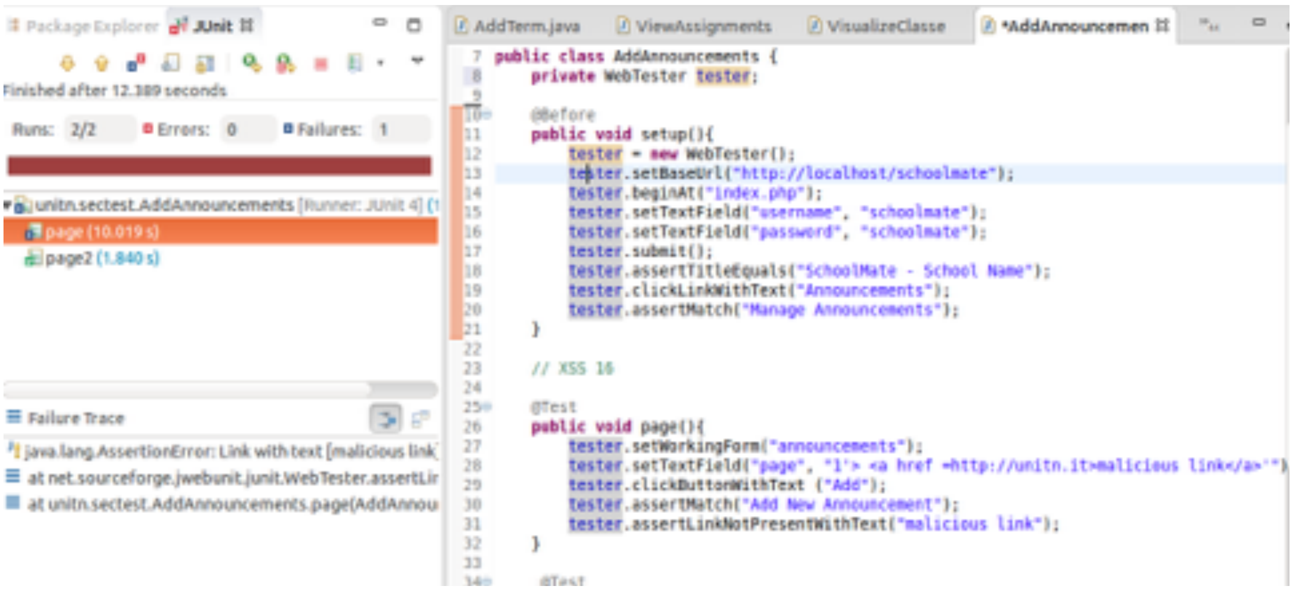
To fix these vulnerabilities, we employ either Client-Side protection or Server Side protection. In general terms, depending on the logic of the application , you can sanitise the the input file i.e php file with tainted variable that takes input or the output file the prints output i.e sensitive sink. In the case of schoolmate, the index.php doesn't handle any data but includes all files & dispatches all features. I sanitised the input value on the index.php file by including a statement htmlspecialchars infant of every $_POST variable and re-run the JWebUnit test cases

Note: The fixed schoolmate code has been included in the folder security testing under file schoolmate.

Conclusion

Report on passed/failed tests

On automation of injection of the malicious link when i run the test, i get red for an error and it fails with this response. Link with text "malicious link".

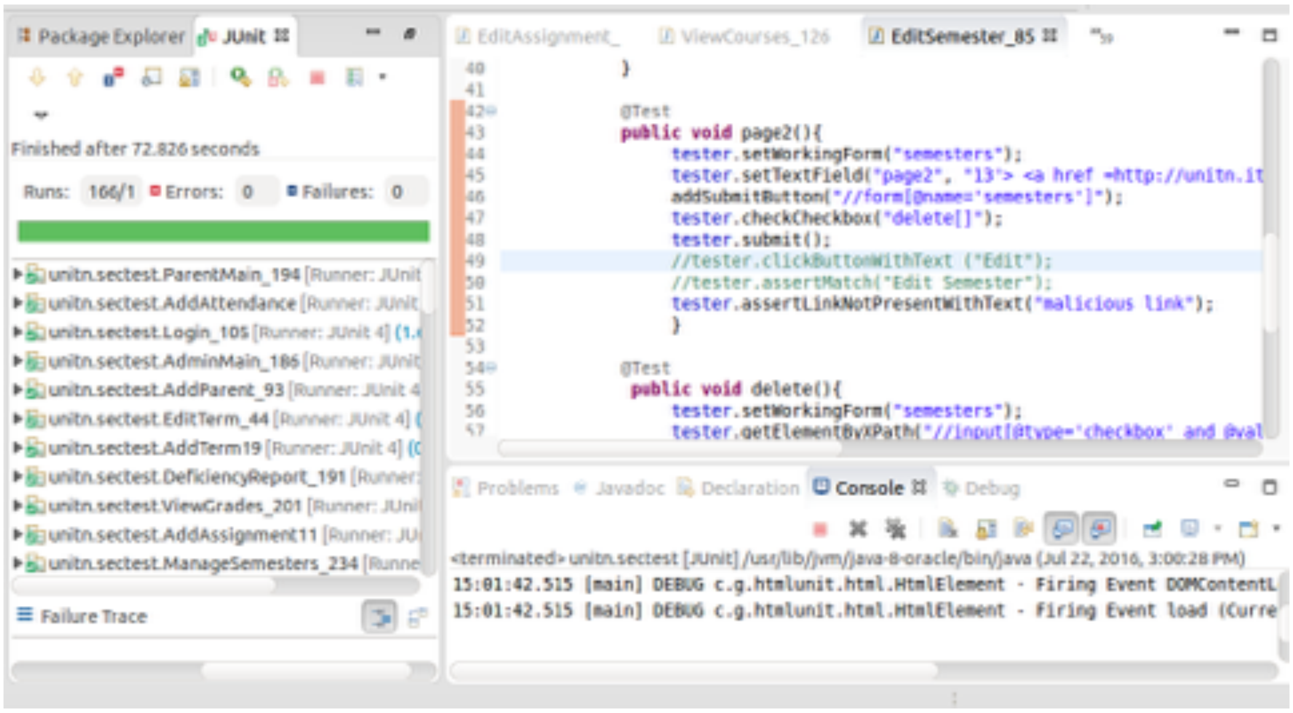


```
Finished after 12.389 seconds
Runs: 2/2    Errors: 0    Failures: 1

unitn.sectest.AddAnnouncements [Runner: JUnit 4] (1
    page (10.019 s)
    page2 (1.840 s)

java.lang.AssertionError: Link with text [malicious link]
at net.sourceforge.jwebunit.junit.WebTester.assertLir
at unitn.sectest.AddAnnouncements.page(AddAnnou
```

```
public class AddAnnouncements {
    private WebTester tester;

    @Before
    public void setup(){
        tester = new WebTester();
        tester.setBaseUrl("http://localhost/schoolmate");
        tester.beginAt("index.php");
        tester.setTextField("username", "schoolmate");
        tester.setTextField("password", "schoolmate");
        tester.submit();
        tester.assertTitleEquals("SchoolMate - School Name");
        tester.clickLinkWithText("Announcements");
        tester.assertMatch("Manage Announcements");
    }

    // XSS 16

    @Test
    public void page(){
        tester.setWorkingForm("announcements");
        tester.setTextField("page", "1'> <a href =http://unitn.it>malicious link</a>'")
        tester.clickButtonWithText ("Add");
        tester.assertMatch("Add New Announcement");
        tester.assertLinkNotPresentWithText("malicious link");
    }

    @Test
```

Report on tainted analysis on fixed code

On sanitising the common vulnerable variables, i rerun the same tests of injections and this time i get green with no malicious link



```
Finished after 72.826 seconds
Runs: 166/1   Errors: 0   Failures: 0

unitn.sectest.ParentMain_194 [Runner: JUnit
unitn.sectest.AddAttendance [Runner: JUnit
unitn.sectest.Login_105 [Runner: JUnit 4] (1.
unitn.sectest.AdminMain_186 [Runner: JUnit
unitn.sectest.AddParent_93 [Runner: JUnit 4
unitn.sectest.EditTerm_44 [Runner: JUnit 4] (
unitn.sectest.AddTerm19 [Runner: JUnit 4] (0
unitn.sectest.DeficiencyReport_191 [Runner:
unitn.sectest.ViewGrades_201 [Runner: JUnit
unitn.sectest.AddAssignment11 [Runner: JU
unitn.sectest.ManageSemesters_234 [Runne
```

```java
        }

        @Test
        public void page2(){
            tester.setWorkingForm("semesters");
            tester.setTextField("page2", "13'> <a href =http://unitn.it
            addSubmitButton("//form[@name='semesters']");
            tester.checkCheckbox("delete[]");
            tester.submit();
            //tester.clickButtonWithText ("Edit");
            //tester.assertMatch("Edit Semester");
            tester.assertLinkNotPresentWithText("malicious link");
            }

        @Test
         public void delete(){
            tester.setWorkingForm("semesters");
            tester.getElementByXPath("//input[@type='checkbox' and @val
```

```
<terminated> unitn.sectest [JUnit] /usr/lib/jvm/java-8-oracle/bin/java (Jul 22, 2016, 3:00:28 PM)
15:01:42.515 [main] DEBUG c.g.htmlunit.html.HtmlElement - Firing Event DOMContentL
15:01:42.515 [main] DEBUG c.g.htmlunit.html.HtmlElement - Firing Event load (Curre
```

Basing on the dynamic tests with JWebUnit, i can conclude that all tainted variables have now been sanitised.

References

http://shell-storm.org/blog/Taint-analysis-and-pattern-matching-with-Pin/

https://www.youtube.com/watch?v=MNN7EiYV8p8

https://www.cs.ucsb.edu/~chris/research/doc/ndss07_xssprevent.pdf