
BUILDING A TESTING FRAMEWORK WITH OWASP JAVA HTML SANITIZER

Steven Aldinger

Marta Pancaldi

Seth Stoudenmier

Michael Stenhouse

CSCI 362-03

Dr. Jim Bowring

Fall 2015

Table of Contents

[Introduction](#)

- Background
- Choosing a Candidate
- The Testing Framework
- Installation

[Chapter One: Git Clone](#)

-

[Chapter Two: Test Plan](#)

-

[Chapter Three: Testing Framework](#)

- TestCases.txt
- Driver.java
- Script.sh

[Chapter Four: Test Cases](#)

- Testing Other Methods
- Displaying Our Results

[Chapter Five: Fault Injection](#)

- Faults
- Results

[Chapter Six: Final Thoughts](#)

- Overall Experiences

Introduction

Background

This report is the culmination of our group work in Dr. Jim Bowring's Fall 2015 Software Engineering course. We were assigned to choose an HFOSS (Humanitarian Free Open Source Software) project, clone it into our own GitHub repository, and begin building an automated testing framework for the project. This assignment was divided into five different deliverables which are collected and revised as chapters in this report. The majority of us were new to Git and Linux at the beginning of this project.

All of the components we created were built and compiled using Linux distributions, specifically RHEL and Ubuntu.

Choosing a Candidate

Our first task was to choose an HFOSS project. The only constraint was that the code base must be designed to compile and run on Linux. We began looking at a wiki of potential [HFOSS projects](#). We needed a project that was well documented and easy to test.

DHIS 2

A modular web-based tool for managing and visualizing aggregate statistical data. The software is primarily used for integrated health information management activities. DHIS' home page cites specific use cases such as "health program monitoring and evaluation, facility registries, service availability mapping, logistics management, and mobile tracking of pregnant mothers in rural communities." Built with open source Java frameworks. Active development. ([Link to project repository.](#))

Mifos X

A Java-based management information system for financial services centered on microfinance and financial inclusion. The Mifos Initiative states that their platform and open-source service model will "increase access to technology for all microfinance institutions, ultimately enabling them to extend their reach to the world's poor." Active development. ([Link to project repository.](#))

Color Reader

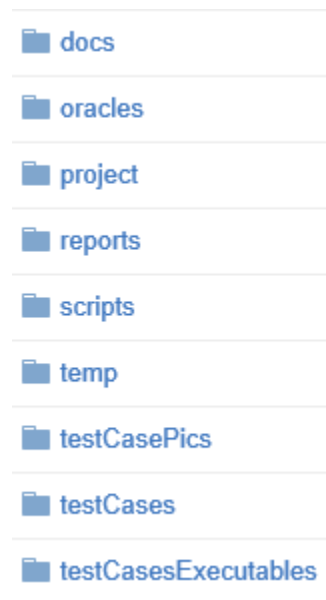
A Java application that "reads" color for those with color blindness. Within the interface, the user can click on a part of an image and the application will read the RGB values of nearby pixels. The RGB values are then converted into HSB (Hue Saturation Brightness) values. The name of the color is then output on the GUI via text and text-to-speech. A database is available with HSB values and their corresponding colors. Inactive development. ([Link to project repository.](#))

Between these three candidates, we chose Mifos X as it was frequently updated and our group is familiar with Java. However, we ran into trouble compiling the code. A web server was needed to run the service, so we changed our project to a standalone application. After researching and testing additional projects, we chose the OWASP Java HTML Sanitizer.

The OWASP Java HTML Sanitizer is described as "an efficient HtmlSanitizer configurable via a flexible HtmlPolicyBuilder". A method in the program named takes an HTML stream, and dispatches events to a policy object which will decide which elements and attributes to allow. In other words, the Sanitizer can take an input of HTML and output new HTML code without any undesirable tags that have been blacklisted by a Policy object. A user would be able to define their own policy for their own customizable white-list. The Sanitizer is open source and written in Java. ([Link to project repository.](#))

The Testing Framework

The directory structure for our framework is below.



Several test cases are in the /testCases folder. These .txt files are accessed by test case specification files, or test drivers. All test cases contain a [standard template](#) that anyone could follow to create their own test cases, given that a driver has been written for the method that is being tested.

A driver has been written for each method that has been tested. These [drivers](#) are written in Java and are found in the /testCasesExecutables/org/trotting folder. We have written drivers for four methods: cssContent(String), decodeHtml(String), isHex(int), and sanitize(String, Policy).

These drivers are invoked by a [single script](#), runAllTests.sh. The script is written in Bash and found in the /scripts folder. Once the script has run all of our test cases through their respective drivers, the results are posted in the /reports folder as individual .txt files. An additional script, toCompiledHtml.sh, echoes these results and formats them into an easy to read HTML table that will automatically open in the default browser.

A README.txt is placed in the /docs folder. The original Sanitizer source code is placed within the /project folder, as well as instructions for installing the software with Maven to add dependencies. Images displaying examples of our results are posted in /testCasePics.

Installation

Here we will detail how to install our testing framework on a fresh Ubuntu install so that anyone can add test cases to the test drivers that we have written.

Chapter One: Git Clone

Task: Checkout or clone project from its repository and build it... Run existing tests and collect results. Evaluate experience and project so far.

This image shows the output of a successful "mvn clean install" to test and build the html sanitizer.

Chapter Two: Test Plan

Task: Produce a detailed test plan for your project. As part of this plan, you will specify at least 5 of the eventual 25 test cases that you will develop for this software.

The Sanitizer is advertised as a program that can detect malicious HTML tags and remove them from input. We searched through the source code of the software and found a detailed testing framework that heavily influenced our test plan.

The first test plan:

1. Create a test around a new HTML document with several HTML tags.
2. Create a test around a new HTML document with a heavy amount of CSS styling.
3. Create a test around sanitizing the HTML source code of a popular website, such as Yahoo.
4. Create a test which reports all changes of HTML tags into a .txt file.
5. Create a test around creating a new policy for detecting certain HTML tags.

It turned out we misunderstood the assignment and created overly complex tests that would take far too much effort to implement. The idea of a testing framework is that anyone can add a simple test case to a folder and see whether it passes or fails.

A new test plan:

1. Test if the `cssContent` method can interpret '\\\\' correctly as \\
2. Test if the `decodeHtml`

Chapter Three: Testing Framework

Task: Design and build an automated testing framework that you will use to implement your test plan.

Our framework directory was modeled after a [predetermined structure](#).

TestCases.txt

Each individual test case exists in its own separate .txt file. All test cases are found in the /testCases folder.

These .txt files contain a template for which we will follow when writing all future test cases.

1. ## test number or ID
2. ## requirement being tested
3. ## component being tested
4. ## method being tested
5. ## command-line arguments
6. ## expected outcomes

As stated in the previous chapter, the author of the Sanitizer included several test cases in the source code that we used for our own testing framework. We decided to test the decodeHtml method for our first five test cases. We tested if the method can handle HTML entities to produce a string containing only valid Unicode scalar values. One of our test cases is listed below.

1. 01##test number or ID
2. decodeHtml handles HTML entities to produce a string
3. orgowashtmlEncoding##component being tested
4. decodeHtml##method being tested
5. "
"##command-line arguments
6. "\\n"##expected outcomes, only \n will actually be te

Driver.java

Our test drivers are found in the /testCaseExecutables folder. Our first driver was built specifically to test the requirements of decodeHtml. Our driver will determine whether or not the expected outcome (theOracle) matches the actual outcome (result). Additionally,

```
public static void main(String[] args){
    try{
        String theOracle = args[1];
        String theTest = args[0];

        String result = org.owasp.html.Encoding.decodeHtml(theTest);
        if(theOracle.contains("\\n")){
            //System.out.println("Oracle contains: \\n");
            theOracle = theOracle.replace("\\n","\n");
        }
    }
```


Script.sh

We had to write a single script to invoke our testing framework. The script accesses a folder of test drivers that will

Chapter Four: Test Cases

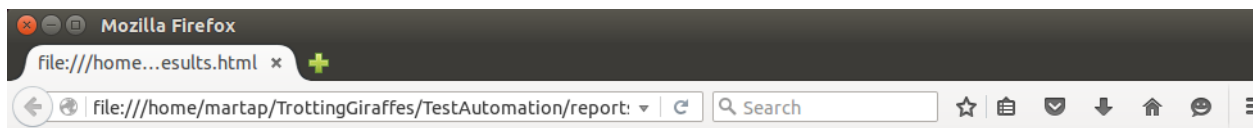
Task: Complete the design and implementation of your testing framework... You will create 25 test cases that your framework will automatically use to test your H/FOSS project.

Testing Other Methods

In Chapter Three, we used the `decodeHTML` method to determine if the Sanitizer was capable of properly decoding Unicode character codes, such as decoding `
` into an HTML carriage return, `\n`. While the task for this deliverable didn't require us to test a new method for our additional 20 test cases, we wanted to include additional test drivers for a more useful test framework.

After scanning through the Java classes in the project source code, we decided that the `sanitize` method should be tested. We began to write a test driver that would determine if input with HTML tags, would be removed completely. The `sanitize` method takes in two arguments: a `String` and a `Policy`, a global object that can be. Due to an overwhelming amount of dependencies and an hour of trial and error, we gave up and found a new method. The `cssContent` method

Displaying Our Results



17	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	FORMATTING.sanitize(String)	Hello,World!	Hello,World!	passed
18	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	FORMATTING.sanitize(String)	>!	>!	passed
19	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	FORMATTING.sanitize(String)	Hello,World!	Hello,World!	passed
20	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	BLOCKS.sanitize(String)	Hello,World!	Hello,World!	passed
21	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	BLOCKS.sanitize(String)	Hello,World!	Hello,World!	passed
22	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	BLOCKS.sanitize(String)	Hello,World!	Hello,World!	passed
23	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	BLOCKS.sanitize(String)	Hello,World!	Hello,World!	passed
24	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	BLOCKS.sanitize(String)			passed
25	org.owasp.html.Sanitizers	org.owasp.html.Sanitizers	BLOCKS.sanitize(String)	hello	hello	passed

Chapter Five: Fault Injection

Task: Design and inject faults into the code you are testing that will cause at least 5 tests to fail, but hopefully not all tests to fail. Exercise your framework and analyze the results.

Faults

For our initial batch of test cases, we tested three methods: `decodeHTML(String)`, `cssContent(String)`, and `sanitize(String, Policy)`. We went into `cssContent` to vandalize some of the method code. All faults are commented on in the code with the original code being commented out below the fault.

We noticed a conditional towards the beginning of the method. The conditional takes the very first character of a `String` and checks if it is a quotation mark or a backslash. In CSS, these characters will result in a character escape. ([CSS specs.](#)) As such, this code will decode any escape sequence and strip any quote from the input. With this first conditional forced to always return false (char `ch0` can never be both a quote and a backslash), any of our test cases containing these characters would fail.

```
public static String cssContent(String token) {
    int n = token.length();
    int pos = 0;
    StringBuilder sb = null;
    if (n >= 2) {
        char ch0 = token.charAt(0);
        if (ch0 == '"' && ch0 == '\\') { // FAULT: Changed an || to an &&
            //if (ch0 == '"' || ch0 == '\\') {
                if (ch0 == token.charAt(n - 1)) {
                    pos = 1;
                    --n;
                    sb = new StringBuilder(n);
                }
            }
        }
    }
}
```

In addition, while searching for additional code to mess around with, we discovered an `isHex(int)` method directly below `cssContent(String)`. The method would interpret a character as an `int` to determine whether or not it could be a valid hex value.

```
public static boolean isHex(int codepoint) { // FAULT: 0x09Aa should be false now
    return ('1' <= codepoint && codepoint <= '8')
        //return ('0' <= codepoint && codepoint <= '9')
        || ('B' <= codepoint && codepoint <= 'Z')
        //return ('A' <= codepoint && codepoint <= 'F')
        || ('b' <= codepoint && codepoint <= 'e');
    //return ('a' <= codepoint && codepoint <= 'f')
}
```

After changing the method rules of what values could return “true”, we quickly wrote a new test driver and four additional test cases to see whether or not the method would return “false” with a valid hex value and vice versa.

Results

Chapter Six: Final Thoughts

Steven:

Marta:

Seth:

Michael: At the beginning of this course, I had zero experience with Linux and Git. Up to this point, I had only worked on simple programs written in various languages.