WORCESTER POLYTECHNIC INSTITUTE

MAJOR QUALIFYING PROJECT

Automated Web Application Testing Using Selenium

Author:
Benjamin CHANEY

Supervisor:
Prof. Micha HOFRI

A thesis submitted in fulfillment of the requirements for the degree of Bachelors of Science

in the

WPI Department of
Computer Science

March 7, 2017

Contents

Glossary

API	application programing interface
CSS	Cascading Style Sheets
GUI	graphical user interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
OS	operating system

Abstract

Source Defense, a web security startup, needs a powerful and efficient testing framework to ensure the correctness of their platform. In this project, we begin by discussing existing work in web application testing. We then design and implement a solution built on Selenium, an existing software testing framework. Our solution is tailored to Source Defense's unique needs. This includes the ability to constantly monitor the proper operation of their product, and alert them in the event of erroneous behavior or an outage. Finally, we utilize our solution to demonstrate its effectiveness at ensuring the proper functionality of their product.

Acknowledgements

I would like to thank the sponsors at Source Defense for hosting my project. Everyone at Source Defense was extremely welcoming and supportive throughout the project. I have learned a lot about web application security, and I am very grateful to have received this opportunity. I would also like to thank Professor Micha Hofri, who advised the project, and Professor Isa Bar-On, who organized the MQP project center in Israel.

Introduction

Testing effectively is an significant hurdle for any engineering effort. The traditional methods of testing do not extend easily into the world of web applications. This makes it difficult to ensure the correct operation of programs running in the browser. A number of testing frameworks have been developed that allow for testing web behavior.

Source Defense[1] is currently in the process of deploying their web based product. Because it is a security focused product, it is important that it be rigorously tested. As a small start up, they do not have significant resources to dedicate towards testing. Therefore, they need a powerful testing framework that is easy to use.

Existing testing frameworks targeted at web applications falls into two main categories. The first category is frameworks that require complicated scripting for each test that is written. These frameworks are very powerful, but they are time consuming to use, and learning to operate them effectively is difficult. An example of this type of test framework is phantom.js[2]. The other category is graphical user interface (GUI) based frameworks. These frameworks are often very easy to use, however they do not offer fine grained control of test functionality. An example of this type of test framework is Screenster[3].

In order to properly fulfill Source Defense's needs, this project attempts to get the best aspects of both types of frameworks. In order to do this, we need to build on top of a powerful test framework, but offer simple functionality. Striking this balance requires targeting our solution to fit Source Defense's specific needs.

The remainder of this report is organized as follows. The Background Section discusses the project sponsor, Source Defense, explains the problem in more detail, and examines the existing state of the art. The Methodology Section discusses the process used to make high level decisions about how the test system will be designed. The Results Section contains a detailed discussion of the functionality provided by our solution, and shows some example tests that demonstrate its performance.

[1]sourcedefense.com
[2]phantomjs.org
[3]screenster.io

Background

The purpose of this project is to develop an automated web testing system for Source Defense. Source Defense's platform operates in the browser to protect customer information. They require a testing system to ensure that their platform functions correctly. This project will examine existing testing frameworks, and compare them to Source Defense's needs. Then we will develop a solution that is designed to be an effective and comprehensive tool for their intended use case.

2.1 About Source Defense

Source Defense is an Israeli startup headquartered in Be'er Sheva. They are developing a product that will allow website operators to set permission on the execution of third part scripts. Their platform is intended to prevent violations of this security policy without otherwise changing the functionality of the website. This requires executing the website's source code in a virtual environment to prevent any unauthorized interaction with the web page. Ensuring that this visualization is performed correctly is of the utmost importance to Source Defense.

2.2 Automated Web Testing

In order to ensure that a large scale software system is functioning correctly, significant testing is required. Most types of software are able to be tested by supplying simulated data, and checking the results against expected values. While this general principle still holds for testing web applications, there are added complications. A complex site running in a browser is often implemented using multiple technologies such as HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. The runtime environments capable of interpreting and executing HTML, CSS, and JavaScript are browsers which are intended to be used by humans actively consuming content. They aren't intended to be used in an automated manner. This necessitates the use of a special purpose test environment.

2.3 Design Goals

Before the project began, Source Defense developed a list of requirements that are intended to ensure that the resulting product fills their needs. This included functionality that they want the end result to have. It also included architectural design considerations that would allow them to continue to make modifications to the software after the end of our involvement.

source
DEFENSE
Enhance Your Site, Securely
Are you sure your website is secured ?
"85% of the exploits detected at recent years were of third-party plug-ins"
"71% of compromise victims did not detect breaches themselves"
"Median number of days from initial intrusion to detection 87 days"
RoboTiCan

FIGURE 1: Source Defense V.P. of product Avital Grushcovski at NexTech Be'er Sheva (Retrieved from: https://twitter.com/JVPVC/status/803890586038235136/photo/1)

2.3.1 Core Features

Source Defense outlined the core structure of the test system. They wanted a collection of workers processes that perform tests, and a master server that controls the workers. The master is required to create the workers, schedule tests to them, and track the results. A test consists of visiting a web page, evaluating its contents, iterating over a list of web elements, and performing a specified action on each element. The actions may either measure a property of the page or modify the contents of the page, so they must be executed sequentially.

2.3.2 Extra Features

In addition to core testing functionality, there are a number of features that Source Defense requested. They include:

- Testing through an HyperText Transfer Protocol (HTTP) proxy server
- Tests that reoccur at a fixed interval
- Automatic alerts
- Configurable timeouts
- Ability to test with multiple browsers

2.3.3 Architectural Goals

In order to allow the project to continue to be useful after we leave, it is important that Source Defense is able to modify and upgrade the test system. Specifically, they want to ability to swap out the software frameworks and libraries that we selected and replace them with alternative libraries. This requires that the software be implemented in a modular fashion so that old technologies can be replaced with minimal extra effort.

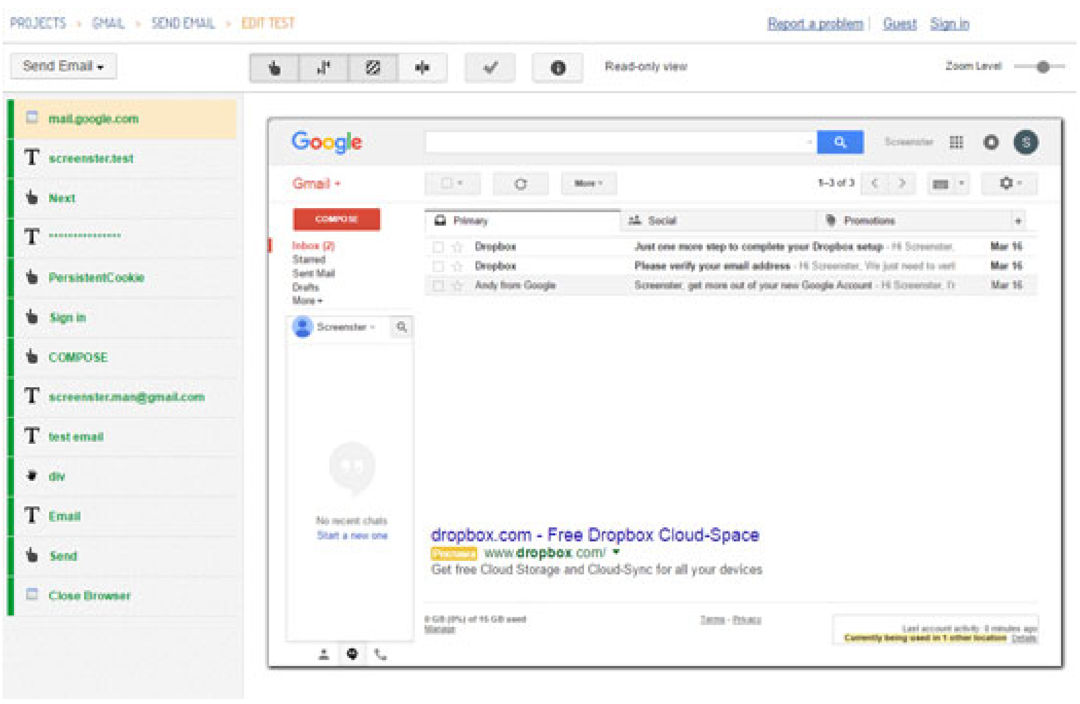

FIGURE 2: Screenster visual regression test tool in action (Retrieved from: screenster.io)

2.4 State of the Art

There are a number of test frameworks designed to allow for automated testing of user facing functionality. They allow for a tester to script a series of actions to be performed in a simulated browser environment. They fulfill some of the needs described in Section 2.3, however they have a number of limitations. They will be discussed in the following sections.

2.4.1 Phantom.js

Phantom.js is a scriptable headless browser. It is designed to allow automated web browsing for testing purposes. It implements a JavaScript application programing interface (API) that allows user actions to be simulated by an automated test system. Its API can be used to navigate a page automatically, without manual interaction by a user. There are a number of competing scriptable headless browsers with similar functionality such as zombie.js[1]. Because it they use their own web engines, they may sometimes differ achieve a different result than a standard browser. This could cause them to miss issues that impact users.

2.4.2 Selenium

Selenium[2] is a testing framework that designed to facilitate in the creation of automated tests of websites. It allows the scripting of actions that mimic user behavior as well as measuring the resulting document by directly examining the properties of web elements present on the page. These actions can be performed by many different web browsers including phantom.js. Selenium has a uniform API across the

[1]zombie.js.org
[2]seleniumhq.org

browsers it supports, which allows tests to be run on multiple browsers with minimal modification. Because it must remain compatible with many browsers, it offers fewer features.

Selenium has a built in tool called Selenium Grid which allows for the distribution of tests across multiple machines. It can connect to different virtual machines located on the same server in addition to be able to manage multiple physical machines. It contains a hub which dispatches tests to different nodes. This is useful for speeding up test suites containing many tests as well as running tests against lots of browser and operating system (OS) combinations.

Selenium also has an GUI called Selenium IDE. Selenium IDE allows for tests to be recording by monitoring a testers browser. It can then generate code that will repeat the test using the standard Selenium API. Despite the fact that it purports to be a purely graphical method of creating tests, it is often necessary to modify the generated code, which negates some of its usefulness.

2.4.3 Screenster

Screenster (shown in Figure 2) is a web based regression test tool. It is designed to detect visual changes in the contents of a web page. This allows a tester to easily detect if a change to a website has unintended side effects by quickly locating all differences in the new version of the page. It is very easy to use, however it is limited by only being capable of detecting issues that have an impact on the visible state of the web page.

Methodology

This purpose of this project is to design and develop a testing tool to assist Source Defense in ensuring the proper operation of their platform. In order to do this, we first discussed the capabilities and features that they need in a testing environment. Then we created a high level design architecture. Next we implemented the solution according to our architecture. Finally we deployed the solution so that it could be used to test Source Defense's infrastructure. Throughout this process we received feedback from Source Defense to ensure that we are meeting their needs effectively.

3.1 Choice of Testing Framework

One of the first decisions made in the creation of this system was the choice of testing framework. It was decided that the framework must be scriptable in order to allow for fully autonomous testing. This reduces us to two main options. The first is a scriptable headless browsers such as phantom.js, zombie.js and headless.js. The other is Selenium, an automated web testing tool.

3.1.1 Scriptable Headless Browsers

Phantom.js, zombie.js, and headless.js are all scriptable headless browsers. They each contain a virtual browser instance which can be used to programmatically access web pages and perform tests. They can evaluate HTML, CSS, and JavaScript, without interacting with a standard browser.

3.1.2 Selenium

Selenium is a web browser automation tool. It is similar to the headless browsers in the sense that it allows scripting of interaction with a web page for testing purposes. Unlike the headless browsers, it can be used to view web pages as they would appear in a number of real browsers such as Google Chrome, Firefox, and Internet Explorer. It also has the ability to connect to the phantom.js engine as if it were a browser.

3.1.3 Decision

After consulting with the sponsor, Source Defense, it was decided that it is important to be able to test with multiple browsers. This is because Source Defense's product is reliant on the precise details of how the browser renders certain web pages. This means that their service may have slightly different functionality when they run on different browsers. For this reason we chose Selenium as the framework to build our solution around.

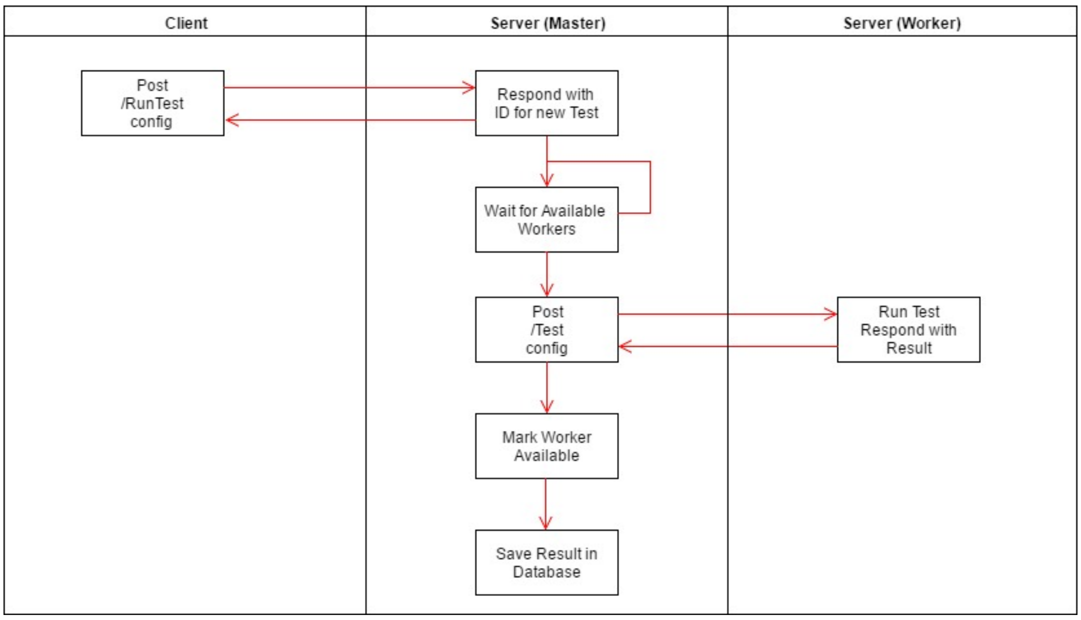

FIGURE 3: The lifetime of a test as it is initialized, scheduled, and run

3.2 Design

The design of the system is based on the needs communicated by Source Defense. As shown in Figure 3, there is a master server that initializes workers and tests. The master server then assigns tests to workers and records the results as the tests complete. If there are additional actions needed after the test completes such as sending an alert, or scheduling a repeat of the test, it is handled by the master server. Each worker receives tests from the master and executes them using Selenium. A worker can only be running a single test at any point in time.

3.2.1 Workers

Whenever a worker receives a test, it immediately parses the configuration data to create an object that contains all the data sent in the HTTP request. It creates an Selenium session using the requested browser, and executes each test operation sequentially until the test is complete. It then aggregates the result of each test operation and sends them as a response to the HTTP request that started the test.

3.2.2 Master

The master manages all requests sent by the user. It contains much more functionality than the workers. The master can create and destroy workers. It also schedules tests and aggregates their results so they can be queried at a later time. The master also sends alerts if an alert criteria was met, and it handles the storage of all persistent data in a Redis database.

3.2.3 Software Architecture

Source Defense requested that the software be written to allow for swapping out components with minimal difficulty. In order to achieve this, we specified internal APIs that were implemented on top of each lower level API component. If it they ever decide to replace one library with a competing solution, they would only need

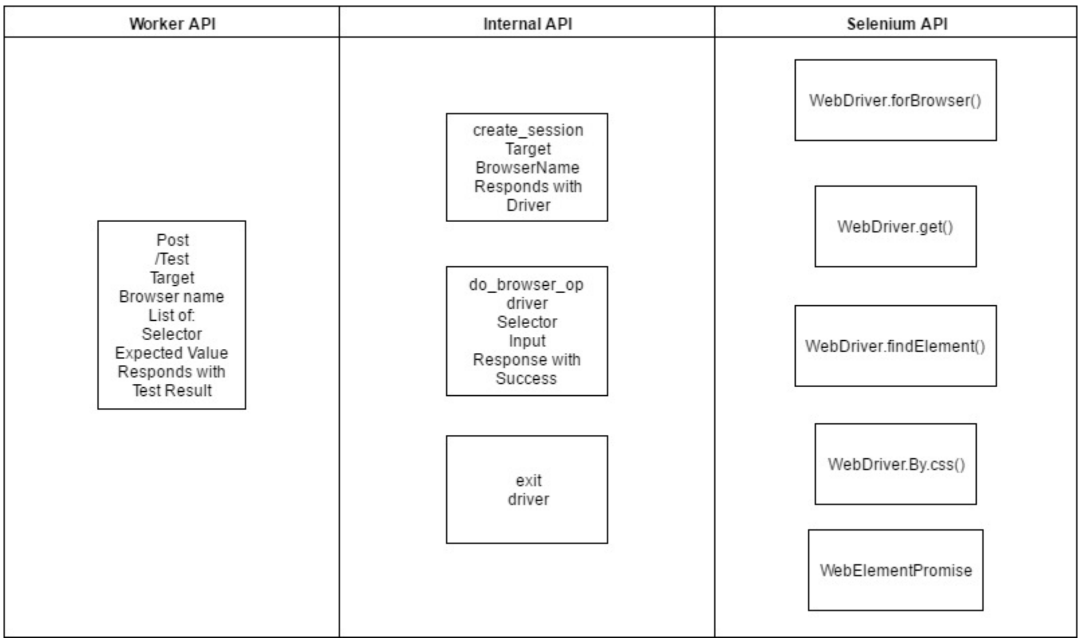

FIGURE 4: The APIs used to schedule tests within the workers at various levels of abstraction

to reimplement the internal API on top of the replacement library, and all of the application logic would continue to work without being affected. One example of this is the integration of Selenium. Figure 4 shows the API to control the worker, the internal API, and the Selenium API. If there was ever a need to replace Selenium with a different test framework, the internal API could be reimplemented on top of a different test framework. All other parts of the project would continue to work correctly without modification. This pattern of an API with multiple tiers is present in many places within our project.

Results

The main deliverable for this project was the testing framework that we developed to ensure the correct operation of Source Defense's platform. It was designed so that a tester could send a test configuration to a central server which would ensure that the test was performed and save the result. The operation of this testing framework is described in this section.

4.1 Worker

The worker is responsible for running the tests. The lifetime of a worker is depicted in Figure 5. It receives test configuration files from the master and runs them using Selenium. It first creates a Selenium session using the parameters in the test configuration. Then it iteratively performs each test operation, and compares the actual result to the expected result. It sends these results back to the Master.

4.1.1 Test Configuration Files

Test configuration files are JavaScript Object Notation (JSON) strings that are sent to the worker when a test is scheduled to it. They have the following elements:

browser A string that specifies what browser should be used to run the test. The possible values are "chrome", "firefox", and "phantomjs".

url A url to load to start the test. This is the web page being tested.

alert *Optional* A list of alert configurations used to send alerts. The structure of each alert configuration is described in Section 4.2.4. If this is omitted, no alerts are sent.

proxy *Optional* An address that specifies a HTTP proxy server that Selenium will use to perform the test. This takes the form of *host:port*. If this is omitted, no proxy is used.

interval *Optional* An integer that defines the frequency with which the test is run (measured in seconds). If this is omitted the test is only run once.

timeout *Optional* The default timeout to use for each operation (measured in milliseconds). If the operation is unable to complete within this amount of time, the operation fails with an error. If this is omitted, the default behavior is that every operation has an unlimited amount of time to complete. This can be overridden on a per operation basis as described in Section 4.1.2.

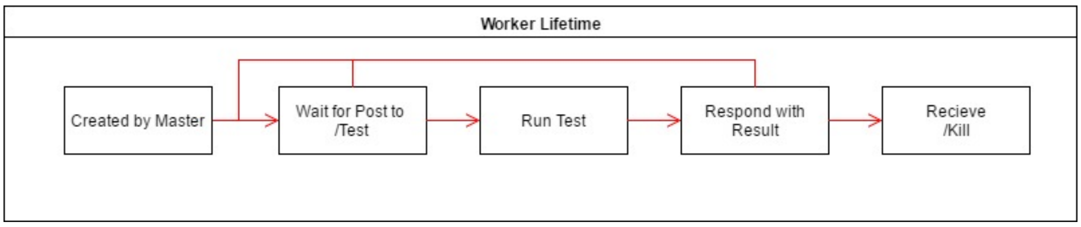

FIGURE 5: The lifetime of a worker

successScreenshot *Optional* A boolean that indicates whether or not to take a screenshot after each successful operation. This can be overridden on a per operation basis as described in Section 4.1.2. If this is omitted it is set to false by default.

failScreenshot *Optional* A boolean that indicates whether or not to take a screenshot after each failed operation. This can be overridden on a per operation basis as described in Section 4.1.2. If this is omitted it is set to false by default.

tests A list of test operations. The structure of a test operation is described in Section 4.1.2

4.1.2 Test Operations

Test operations define what action is taken at each step in a test. Each operation can either succeeded, fail, or terminate with an error. The test operations structure is as follows:

selector A CSS Selector that specifies which web page element (or elements) the operation will be performed on.

index *Optional* Integer that identifies which web element to operate on when the selector matches multiple elements. If this is omitted, the value 0 is used.

name The name of the operation. The possible operations are discussed in Section 4.1.3

value The value to use. This is either a input value or an expected value depending on the contents of name.

timeout *Optional* A timeout to use for this operation (measured in milliseconds). If the operation is unable to complete within this amount of time, the operation fails with an error. This overrides the test default. If this is omitted the test default is used.

successScreenshot *Optional* A boolean that indicates whether or not to take a screenshot if this operation succeeds. This overrides the test default. If this is omitted the test default is used.

failScreenshot *Optional* A boolean that indicates whether or not to take a screenshot if this operation fails. This overrides the test default. If this is omitted the test default is used.

4.1.3 Operation Types

Each possible operation type corresponds to a particular action that can be taken during a test. These possible operation types break down into two main categories. The first is operations that retrieve information. For these operations, **value** is an expected value which is compared to the retrieved data. If the expected value is equal to the retrieved value, then the operation succeeds. Otherwise it fails. The exception to this is test operations which expect a string. These operations also succeed if the expected value is a substring of the actual value. The other type of operation acts on the web page. For this type of operation, **value** is an input to the operation. Not all operations require an input value. For those that do not, **value** is ignored. The possible operations that retrieve information are:

Location Get the location of the web element {x, y}.

Size Get the size of the web element {height, width}.

TagName Get the tag name of the web element.

Text Get the text visible to the user that is within the web element.

Displayed Get a boolean that says if the web element is displayed.

Enabled Get a boolean that says if the web element is enabled.

Selected Get a boolean that says if the web element is selected.

The possible operations the act on the web pages are listed here:

SendText Input the specified text.

Submit Send the submit command. **Value** is ignored for this operation.

Click Click on the specified element. **Value** is ignored for this operation.

4.2 Master

The Master handles all functionality in the testing framework other than the running of the tests, which it delegates to worker threads. This includes scheduling tests, tracking tests that needs to be repeated, storing test results, sending alerts if they are deemed necessary, and answering user queries about test results.

4.2.1 HTTP Targets

The master supports a number of HTTP targets (shown in Figure 6). They allow users of the test framework to connect to create and destroy workers, schedule tests, and retrieve information. Each possible HTTP target is discussed in this section:

/RunTest

A HTTP POST to this target creates a new test and runs it if there is a worker available. The body of the HTTP POST is a JSON string containing all necessary configuration data used to run the test. When the request is first received, it is marked as ready to run and stored in an database. Then the scheduler executes, which runs tests that are ready if there are workers available. The scheduler is described in Section 4.2.2.

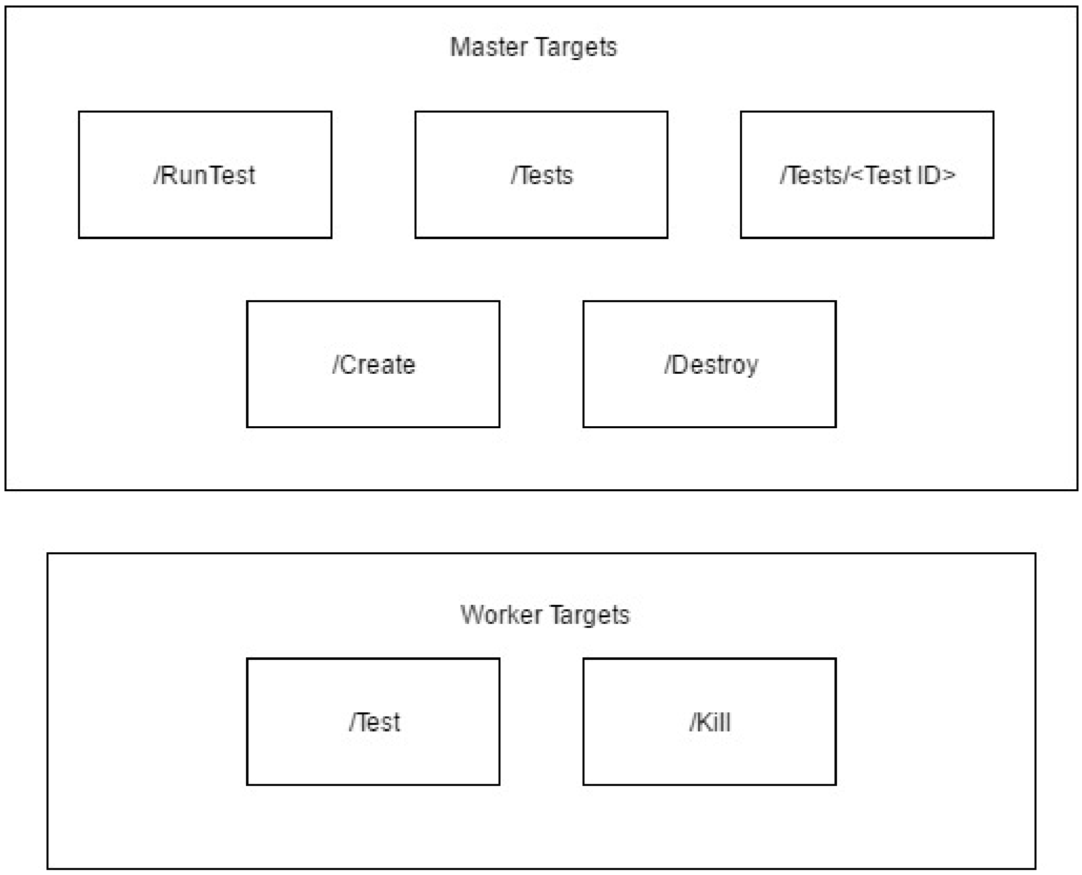

FIGURE 6: A visual representation of available HTTP targets

/Tests/<*testid*>

A HTTP GET to this target will prompt the master to respond with information about the test with an id equal to *testid*. This information takes the form as a JSON string with the following fields:

config The configuration file that was sent to the server when this test was created.

status An integer status code that describes where the test is in its lifetime. Possible values are 0 (ready), 1 (running), and 2 (complete).

result A list of results, one for each test operation. Each element of the list includes a copy of the test operation, the expected value, the actual value, and a success flag. Each result may also have a screenshot that was taken after the operation completed.

/Tests

A HTTP GET to this target will prompt the Master to respond with information about every test regardless of where it is in its lifetime. This information takes the form of a JSON formatted string with a list of tests. Each test in this list has the same form as described in the above section.

/Create

A HTTP POST to this target will prompt the Master to create a new worker. The worker resides in its own process. Once the worker has been created, it is marked as ready to accept a test, and added to the database. The database contains an ID field, a port used to connect to the worker, and a status field which keeps track of which test it is running. There is no response sent for this type of request.

/Delete

A HTTP POST to this target will prompt the master to destroy a worker. This only succeeds if there is a worker that is not currently running a test. If all workers are currently busy, or there are currently no workers, the response is an error *No Workers*. If there is a free worker, the action succeeds, the worker is instructed to terminate, the master removes it from the database that stores worker information, and the master responds to the request with a JSON string containing information about the worker that now no longer exists.

4.2.2 Scheduler

The scheduler is responsible for matching tests that are ready to run with available workers. It uses a very straightforward first come first serve algorithm so that tests are executed sequentially. Once it matches a test to a worker, it sends the worker the configuration file with the necessary test data via HTTP (described in Section 4.1.1). Then it modifies the test's database entry to signify that is running, and it updates the worker's database entry to show which test it is running. It also sets up the results handler that runs when the newly scheduled test is complete.

4.2.3 Results Handler

The results handler updates the database with the results of completed tests. It runs when a worker sends results back to the master. It stores the results in the database. It also marks the test as complete, and the worker as available to run a new test. If the test configuration specifies that it should be run repeatedly at a fixed interval, it schedules a timer that will eventually add a new copy of the test. Then it runs the scheduler (see Section 4.2.2) in case there are more tests that can be run on the newly freed worker. It also calls the alert subsystem to send out alerts if they are necessary.

4.2.4 Alert Subsystem

The alert subsystem is responsible for sending out alerts if the tests results match the supplied alert criteria. The alert criteria is a list of configuration objects that each define when an alert should be sent. An alert is sent if any of the configurations match the results. If multiple configurations match, then multiple alerts will be sent. Each configuration has the following elements:

on A list of strings that define events that will cause an alert to trigger. The possible values are "Success", "Failure", and "Error". Success triggers only if every test operation succeeds. Failure triggers if any test operation fails. Error can trigger if any test operation raises an error, or if the test as a whole cannot run.

type A string that defines the type of alert to send. The possible values are "SMS" which sends a text message, "POST" which sends an HTTP POST, and "email" which sends an email.

address A string that defines where to send the alert to. If type is "SMS",this is a phone number. If type is "email", this is an email address. If type is "POST", this is an HTTP target in the form *domain:port/path*

4.3 Tests

In addition to constructing a test framework, we have also written a number of tests that can be run on top of it. This serves several purposes. First, they verify that the test framework is working as intended. Second, they serve as examples to Source Defense's employees who will use the framework in the future. Finally, they ensure that some core features of Source Defense's product are working correctly.

4.3.1 Status Test

The first test we developed is designed to constantly monitor Source Defense's engine to detect if it ever goes down. It checks the status page using phantom.js once a minute, and ensures that the status is displaying a version number. If their engine goes down, this test will fail. The alert parameters specify that if the test fails, or if there is an error in executing it, Source Defense's head of research and development, Eido Gavish, will receive a text and an email. The configuration data for this test is shown below. His phone number and email have been removed.

```
{
    "browser" : "phantomjs",
    "url" : "http://vice-status-dev.azureedge.net/sditesting",
    "alert" : [{
        "on" : ["Failure", "Error"],
        "type" : "SMS",
        "address" : <removed>
    },
    {
        "on" : ["Failure", "Error"],
        "type" : "email",
        "address" : <removed>
    }],
    "interval" : 60,
    "tests" : [
        {
            "selector" : "*",
            "name" : "Text",
            "value" : "vice.version"
        }
    ]
}
```

4.3.2 Loader Test

Another test we wrote was designed to confirm that a script belonging to Source Defense is being delivered correctly. It attempts to download the script using phantom.js once an hour, and ensures that it receives a believable response. If the script is unable to load, this test will fail. The alert parameters specify that if the test fails, or if there is an error in executing it, Source Defense's head of research and development, Eido Gavish, will receive a text and an email. The configuration data for this test is shown below. His phone number and email have been removed.

```
{
    "browser" : "phantomjs",
    "url" : "http://vice-loader-dev.azureedge.net/sditesting",
    "alert" : [{
        "on" : ["Failure", "Error"],
        "type" : "SMS",
        "address" : <removed>
    },
    {
        "on" : ["Failure", "Error"],
        "type" : "email",
        "address" : <removed>
    }],
    "interval" : 3600,
    "tests" : [
        {
            "selector" : "*",
            "name" : "Text",
            "value" : "function"
        }
    ]
}
```

4.3.3 Release Test

This test also checks to confirm that a script belonging to Source Defense is being delivered correctly. The test is identical to the one described in Section 4.3.2. The only difference is that it checks a different script.

```
{
    "browser" : "phantomjs",
    "url" : "http://vice-release-dev.azureedge.net/sditesting",
    "alert" : [{
        "on" : ["Failure", "Error"],
        "type" : "SMS",
        "address" : "+972507098964"
    },
    {
        "on" : ["Failure", "Error"],
        "type" : "email",
        "address" : "ben@sourcedefense.com"
    }],
    "interval" : 3600,
    "tests" : [
        {
            "selector" : "*",
            "name" : "Text",
            "value" : "function"
        }
    ]
}
```

Conclusion

The goal of this project was to provide an effective and powerful test framework for Source Defense to test their web application security product. We began by examining what existing frameworks we could build upon. We selected Selenium because of its ability to interface with many browsers. Next we designed the high level architecture of our solution to ensure that it would be both powerful and extensible. Then we implemented a solution based on our architectural designs and the requirements communicated to us by Source Defense. Finally, we devised a number of tests that run on our test framework. These tests provide both practical utility to Source Defense, and demonstrate that we were successful in meeting our stated functionality goals.