FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Automated XSS Vulnerability Detection Through Context Aware Fuzzing and Dynamic Analysis

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von

**Tobias Fink, BSc**
Matrikelnummer 1026737

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Mitwirkung: Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Wien, 21. Juni 2018

| Tobias Fink | Edgar Weippl |
|---|---|

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Automated XSS Vulnerability Detection Through Context Aware Fuzzing and Dynamic Analysis

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering & Internet Computing**

by

**Tobias Fink, BSc**
Registration Number 1026737

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar Weippl
Assistance: Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Vienna, 21st June, 2018

Tobias Fink

Edgar Weippl

# Erklärung zur Verfassung der Arbeit

Tobias Fink, BSc
Windmühlgasse 22/20, 1060 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Juni 2018

Tobias Fink

# Kurzfassung

Cross Site Scripting (XSS) Angriffe sind bereits seit langem bekannt und obwohl eine breite Palette an Gegenmaßnahmen vorgestellt wurde, ging das Auftreten von neuen XSS Schwachstellen nur marginal zurück. Auf der anderen Seite erhöht sich die Zahl an Angriffen sowie deren Raffinesse, da immer mehr Anwendungen Code vom Server auf den Client auslagern. Existierende Programme zur Schwachstellenanalyse in diesem Sektor weisen oft nur eine ungenügende Genauigkeit beim Aufspüren solcher Fehler auf.

Deshalb beschäftigt sich diese Arbeit mit der Lösung der folgenden Probleme: (i) Unzureichende Testumgebungen zur Auswertung bestehender Tools, (ii) Fehlende Vergleiche bereits existierender Black-Box Analysetools, (iii) Entwicklung und Implementierung von Methoden zur effektiven Erkennung von XSS Schwachstellen.

Die automatisierte Auswertung von Analyseprogrammen bedarf einer Testumgebung, welche eine möglichst hohe Abdeckung an diversen XSS Schwachstellen bietet. Da existierende Frameworks hier große Mängel aufweisen, wurde eine eigene Anwendung entwickelt um möglichst viele Kombinationen unterschiedlicher Dateneingangs und -ausgangspunkte, sowie Filtermechanismen in Webanwendungen abzubilden, welche zuvor in keinem Testset enthalten waren.

Basierend auf dieser Testumgebung wurde eine großangelegte Studie zu frei verfügbaren XSS Scannern durchgeführt. Die Auswertung zeigt, dass existierende Tools nur unzureichende Ergebnisse liefern. Speziell bei den Varianten der DOM-basierten und persistenten Schwachstellen konnten die meisten überhaupt keine Testfälle erkennen.

Um den Stand von offenen und frei verfügbaren Werkzeugen zu verbessern stellen wir daher eine eigene Black-Box Methodik zum effizienten Erkennen von XSS Schwachstellen in Webanwendungen vor, welche aus zwei wesentlichen Teilen besteht. Zuerst wird eine Analyse der Datenflüsse innerhalb der Anwendung durchgeführt um Informationen zu deren Eingangs- und Ausgangskontext sowie eventuell vorhandene Filter- und Sicherheitsmaßnahmen zu erhalten. Anschließend werden diese zuvor identifizierten Datenkanäle durch Fuzzing mit speziell adaptierten Angriffen getestet und dynamisch in einem integrierten Browser ausgewertet. Diese dynamische Verifizierung erlaubt es falsch positive Ergebnisse zu vermeiden.
Die Evaluierung des entwickelten Prototypen zeigt auf, dass die neu entwickelte Methodik im Vergleich zu existierenden frei verfügbaren Tools in der Lage ist in annähernd der selben Zeit eine signifikant höhere Anzahl an ausnutzbaren Schwachstellen zu identifizieren.

# Abstract

Cross Site Scripting (XSS) attacks have been around for a long time and while a multitude of countermeasures and mitigation techniques have been researched, XSS vulnerabilities did not decline much. The number of attacks and their sophistication increases as more and more code is shifted into the client side of web applications. Therefore this thesis deals with the following challenges: (i) Creating and implementing a method for efficient XSS detection, (ii) the lack of coverage of current testbeds regarding types and possibilities of XSS, (iii) the missing comparisons and analysis of existing black-box scanners.

A testing environment was created covering a large number of diverse XSS vulnerabilities in 1808 distinct test cases so that black-box scanning tools can be evaluated and compared with regard to their performance. The focus was to maximize combinations of different input and output contexts together with filtering mechanisms and also to minimize the complexity of the web application and test case structure in order to make them easily accessible for automated scanners. The test cases of already existing testbeds, which often only feature a handful of simple XSS test cases, were integrated and many more new and advanced ones implemented.

Based on the previous findings an approach to efficiently detect XSS vulnerabilities is presented and implemented in a fully automated scanner prototype. The approach is based on data flow detection together with input and output context analysis. This allows the construction of specialized and environment sensitive attack payloads. The scanner additionally collects information about potential input filters and sanitization mechanisms and evaluates these against several evasion methods. The gathered data flows are then fuzz tested with specifically tailored payloads. Finally, detected vulnerabilities and attack vectors are verified dynamically to ensure a zero false positive rate.

Several automated vulnerability scanners that try to detect XSS flaws exist, but no large scale comparison of their performance regarding detection rate is available. Therefore existing free and open source analysis tools were evaluated against the created testbed, uncovering that most of them lack proper detection capabilities especially in the sectors of DOM-based and stored cross site scripting. Many could not even detect a single vulnerability in the test cases of these two categories. The evaluation of the prototype implementation of the new approach shows, that it is able to detect significantly more vulnerabilities than other open source scanners. This is achieved while the time required for the scanning process stays in the range of the best performing open source tools.

# Contents

CHAPTER 1

# Introduction

Cross-Site Scripting (XSS) vulnerabilities have been among the most common vulnerabilities in web applications for a very long time, nevertheless the number of attacks is still on the rise [1–5]. Even big, well tested websites that are used by millions of users are continually facing newly discovered exploits in this category [6–12]. When those software bugs are abused, it allows malicious payloads to be executed, which opens the door for adversaries to perform a wide range of attacks against the web applications, their users and owners. This includes among other types of attacks the stealing of user credentials or sensitive information, taking over accounts, performing malicious actions like clickjacking, impersonation or the spreading of malware [13–18].

One of the major challenges of preventing such vulnerabilities is the ever increasing complexity and dynamicity of modern web applications. The ongoing changes in the JavaScript and HTML standards continually introduce new features which enable a more interactive user experience [19–22]. This leads to a shift from mostly static websites that are loaded in a simple HTTP request from a web server, to adaptively changing sites depending on server events and web applications that include rich client-side functionality which also work offline. WebSockets [23], WebRTC [24] and other new JavaScript elements [25–28] allow continuous background communication channels between the web application and multiple servers. While it was sufficient to perform a simple static string search for malicious HTML tags that could be injected to identify data flows that lead to possible vulnerabilities formerly, this is not adequate any more as many likely vulnerable entry points may be missed. Additionally the dynamic nature of JavaScript allows dynamic code generation and evaluation at runtime which makes obfuscation easy [29, 30]. Furthermore new HTML elements and the addition and introduction of new properties create novel ways to bypass existing XSS filters [31, 32].

White-box testing approaches could provide very detailed insights about potential vulnerabilities, because of the actual source code that is analyzed. However the dependency on the code makes the testing approach inflexible when it needs to be applied on various

kinds of programming languages or web applications that combine a diverse technology stack. In order to be able to detect XSS vulnerabilities efficiently and independently of the programming languages and technologies black box testing can be used. This method avoids the necessity to have access to and the need to analyze the actual source code. Automation and integration into regular testing processes during development is easy. Furthermore a generic analysis approach does not depend on specific manual refinements and adaptations for different types of websites. Even closed source applications can be analyzed this way [33, 34].

Several approaches of black-box testing for cross site scripting exploitability were developed over the years. Many of those scanning tools are outdated and have low detection ratios or high numbers of false positives [34–40]. Often a lot of manual work is required to identify and examine all web application entry points. Standard XSS payloads are tested most of the time, but special payloads that are only applicable in specific contexts are often missed by automated tools [4, 34]. A more advanced method of automated vulnerability scanning is required to improve the detection quantity and quality of XSS vulnerabilities in modern web applications.

## 1.1 Approach

In this thesis we propose FOXSS, a autonomous XSS scanning tool featuring fully automatic discovery of data flows that is capable of intelligent fuzzing with regard to their context. This allows the detection of possible vulnerabilities with high accuracy and a large coverage rate. It is achieved by instrumented web application execution and a combination of static and dynamic analysis. The validity of each vulnerability that gets reported by FOXSS is dynamically verified in a browser engine by automatically generated proof of concept attack payloads. Thus guaranteeing zero false positives and a high detection ratio of XSS vulnerabilities. The processing steps that FOXSS performs are as follows: (i) The data flow detection module first identifies all possible input channels in the different parts of the web application. This is accomplished by checking the static parts like URL, HTML and HTTP headers as well as the dynamic parts which include JavaScript event handlers, background communication like WebSockets or JavaScript initiated HTTP requests etc. Then those input channels are tested with specific identifiers and monitored for possible output channels. To achieve this the Document Object Model (DOM) is searched for the identifiers and a set of relevant built-in JavaScript functions and objects is instrumented to detect when any identifier is passed through. Several contextual aspects are gathered and aggregated when a full data flow from input to output is recognized. (ii) Afterwards the payload generation module decides based on the data flow, the input and output contexts which exploits should be generated, transformed, assembled and then tested. (iii) An evaluation component of the XSS analysis module then checks each of the previously generated XSS tests for successful injection and execution of payloads. This is done by loading the derived request in a browser window and automatically perform all actions required to trigger the vulnerability like simulating

user interactions. A XSS execution is verified by exposing a custom function in the global JavaScript context of the browser engine and triggering it through the XSS payload.

Additionally a testing environment for evaluating black-box XSS scanners was created. It is separated into two parts: The test cases of the publicly available testbed "Firing Range" [41] and the newly developed testbed "XSS Playground". A combined number of 1808 distinct test cases covers the majority of the XSS vulnerability landscape. They range from basic and trivial ones which can be found in existing testbeds to different rare vulnerabilities and heavily dynamic JavaScript based test cases. The web application behind the XSS Playground has a flat tree like structure where every test case can be reached through a custom URL. Its design allows automated scanners to easily access and examine the test cases. The focus lies on the detection of the actual XSS vulnerability rather than testing a scanners ability to navigate through hidden or protected areas. So no authentication or session dependent parts of the web application exist.

The finalized testing environment allows us to evaluate and compare our own solution FOXSS with other black-box scanners. Since many of them were created and also abandoned over time first the available existing XSS scanners and approaches were gathered and reviewed in detail. After an initial assessment where outdated and otherwise inappropriate or unobtainable scanning tools and projects were discarded, the remaining were analyzed in the testing environment.

## 1.2 Contributions

In the course of working on this thesis for XSS detection the following scientifically relevant contributions are made:

- Numerous existing XSS detection solutions are compared and their functionality discussed and analyzed. This is based on their documentation, source code (if available) and the actual results they generated when executed on the specially crafted test web application. Their performance is recorded and rated taking into account the runtime, coverage of detected vulnerabilities, false positive and false negative rates as well as special side effects that might occur during their execution. This leads to recent up-to-date comparative results and an overview of the features of existing black-box scanners. Even lesser-known open source programs were tested, which might be hard to find and know about otherwise. Also scanners that were created many years ago are taken into account. All that were obtainable were first analyzed whether they are worth a deeper analysis and ready for the testing environment. Since several ones failed some checks, they are documented separately and the problems encountered are discussed. To the best of my knowledge no such wide-ranging comparison was previously created.
- A testbed dedicated to XSS vulnerabilities is created. It includes tests covering all classes and many different types of XSS bugs. It provides combinations of all entry

sources with different exit contexts and input sanitizing functions. A special focus is put on vulnerability corner cases, very unusual ones, multi stage vulnerabilities and those which are only possible through new HTML5 and JavaScript language additions. Also existing test environments that are dedicated to Cross-Site Scripting test cases were analyzed and influenced the test cases that are featured in the testbed.

- The analysis of different attack payloads and the different execution contexts where they can appear. Also the possibilities of how they can be transformed or manipulated so that filtering and sanitization techniques can be successfully evaded is investigated. Data sources and sinks that can be found in modern web applications are listed, analyzed and presented. Those are important for identifying data flows and subsequently for vulnerability testing.

- The main contributions are the newly developed XSS scanning approach and the implementation of a fully automated black-box vulnerability scanner, capable of identifying the data flows of modern, highly dynamic web applications. Those can be thoroughly tested for potential cross site scripting vulnerabilities in an intelligent way: The context in which information flows happen are considered which allows adapting attack payloads and leads to a reduction of the test space. The client-side code of web applications under test is executed dynamically in a prepared environment that monitors JavaScript execution and DOM access. It detects and is able to verify injected context refined payloads. The scanner is able to find vulnerabilities which are undetectable by existing tools and features a false positive rate of zero.

## 1.3 Structure of the Thesis

The remaining parts of the thesis are organized as follows: In the next chapter (2) existing approaches to detect and defend against XSS vulnerabilities are analyzed. Chapter 3 provides the technical background about XSS and presents the different classes. Chapter 4 explains the theoretical concepts and the design of the scanner and its mechanisms. Chapter 5 describes the actual implementation of the testbed and the details about the test cases. After that the evaluation of existing XSS detection applications and the newly created advanced scanner is presented in chapter 6. The final chapter 7 provides a conclusion and ideas for future work.

CHAPTER 2

# Related Work

A lot of research has been conducted over the years in the area of Cross-Site Scripting. The scientific work and research by interested individuals that was published on their personal websites, blogs or other channels will be discussed in this chapter. Several different methods to detect and prevent cross site scripting were examined. The fact that this topic is still very actively investigated, the amount of XSS vulnerabilities found, abused and publicly disclosed and the multitude of varying approaches shows that there is still a lot of potential for improvement and new ideas [3]. The research in the area of XSS can be split into a few high-level categories:

- Client side protection, mostly deployed in the browser as an add-on or an addition to the browsers source code.
- Server side protection focusing on filtering mechanisms, input sanitization, test generation and additional security layers.
- Automated vulnerability scanners targeting web applications, performing black-box testing.
- Automated code analysis finding security flaws or predicting vulnerabilities (white-box testing).
- Analysis of new XSS attack vectors, XSS types and other work of more theoretical nature (like improving test coverage or detection algorithms).

The approach which will be presented in this thesis belongs to the third and the last category.

## 2.1 XSS Vulnerability Detection

The approaches listed in this section primarily try to identify XSS vulnerabilities in web applications through different methods. The algorithms subsection contains mostly theoretical concepts.

White-box approaches analyze the source code of the applications. They depend on the availability of the code for analysis and have the disadvantage of being specific to one or more languages. They need to understand the different programming styles and models which they follow together with all the API differences. In return they can examine all entry points and the actual handling of inputs and outputs.

Black-box approaches work like an external attacker which does not have any information about the inner functionality of the web application. The program entry points have to be identified and tested (often with numerous) payloads in order to eventually find a software bug which can be exploited. The advantage is that actual proof of concept exploits can be found and the application is exercised dynamically such that vulnerabilities that might not exist in the development environment because of a different configuration can be found. Furthermore those approaches can be executed on any web application independently of the programming language it was written in. The weakness lies in the discovery of data channels. Unlinked ones or dynamically generated ones might not be found and not tested at all. Such tests are also more time consuming since many requests are needed to exercise different XSS payloads and entry points.

### 2.1.1 Algorithms

An algorithm for XSS detection that analyzes the position of certain characters is presented in [42]. 32 characters that regularly occur in malicious scripts represent the features. Each feature has a certain value, based on their importance. The highest valued characters are `" > / < (space) = '`. For each input sequence a value can be computed and compared with a threshold to determine if the input is classified as containing an XSS payload or not. The detection threshold is defined beforehand by computing values of many learning samples and choosing it with the help of a special function. This algorithm is even able to detect obfuscated XSS payloads.

Detecting XSS bugs through model checking was proposed in [43]. They build a behavioral model of the website by inspecting the HTML structure and deriving legitimate actions a user can perform. This model can be built automatically by the algorithm discussed in the paper. The model is expressed in CTL (Computation Tree Logic). After the model is created, it can be checked, which allows finding bugs that can lead to XSS. The authors recommend that this should be done before deploying a web application at the end of development.

### 2.1.2 White-box Approaches

Hydara et al. propose a method for detecting cross-site scripting vulnerabilities already

during development, before a web application is deployed. With the help of a genetic algorithm, which operates on the control flow graph constructed of the source code, software flaws are found. The approach was validated in a prototype implementation which is able to automatically scan Java based web apps and report possible findings [44]. In [45] also a genetic algorithm in combination with static analysis of the source code of web applications, that are written in PHP, is presented. The focus of this paper lies on optimizing existing genetic algorithms and to optimize the generation of test data. All three types of XSS vulnerabilities could be detected with this method.

The tool saferXSS operates on Java source code and performs static analysis. Through pattern matching it finds unvalidated input channels and tries to autonomously apply a proper sanitization function. To achieve this, control flow graphs are created and the contexts in which user data is used are evaluated. Then code rewriting secures the application considering the findings [46]. In [47] a scanning program that employs a combination of static and dynamic analysis was created. Instead of targeting the analysis of websites it was designed to author browser extensions, focusing on XSS vulnerabilities that are caused through DOM-based sources. In a first step they create candidate vulnerabilities, then proof of concept exploits are generated by dynamic symbolic execution. The program is designed to test user provided scripts for the Greasemonkey browser extension.

A machine learning based method that operates on the source code of PHP applications is presented in [48]. First a data flow graph is built and functions that are executed on those data nodes are classified into different categories. With the use of a training data set and statistical classifiers the effectiveness of those sanitization functions which are applied on the data is calculated. This allows predicting whether they are safe or vulnerable. Another approach by Gupta et al. is also based on machine learning. The analysis process of PHP source code identifies IO-channels, filtering and validating functions and combines them with information about the context in which the output of user generated information happens. Through different machine learning algorithms multiple prediction models are built and used to analyze the extracted features [49].

An approach that uses symbolic execution and constraint solving to discover XSS is presented in [50]. It operates on the bytecode of Java web applications. The detection of multiple classes of injection vulnerabilities are supported. Regarding XSS they do not consider DOM-based vulnerability sinks which is a major drawback. The same authors also released another paper that focuses primarily on constraint solving for XSS detection in [51]. In [52] a concolic testing approach is explored. Java web applications are processed and IO channels are gathered to detect any dependencies which might be susceptible for XSS. Then the source code is transformed to another language which is understood by concolic testing applications. At this point they can be executed in the testing engine and the handling of XSS injection attacks is observed by monitoring components which are injected into the code. The final values of input data can then be checked before they are used in critical parts of the application, like the database or in the HTML output. In their approach simple pattern checking is performed in this final

stage for XSS detection.

In the area of taint tracking and analysis the following approaches have been developed: Andromeda is a source code analyzer for Java, JavaScript and .NET applications. Similar to other taint tracking solutions it builds a control flow graph out of input, output and sanitization function triples. Vulnerabilities are found when there is no appropriate data transformation between an input and an output [53]. Gupta et al. created a tool for XSS detection employing a "context-sensitive approach based on static taint analysis and pattern matching techniques" [54]. It operates on the source code of web applications programmed in the PHP language, because it is the most used server-side language. In its three steps the tool identifies output statements that might be vulnerable and tracks the information flow back to its input source, while also considering the input context. The final decision whether a vulnerability exists is made based on the existence of proper sanitization mechanisms on the IO path. XSSDM was evaluated on a large set of PHP source code files, where 2856 files contained at least one XSS vulnerability. Its major limitations are the restriction to a single programming language and the inability to handle object oriented code [54]. Similarly XProber creates a control flow graph and performs static taint analysis in PHP scripts. Although the final detection of XSS is quite different from other approaches. They are using a unique technique for analyzing strings [55]. An approach for analyzing XSS vulnerabilities in mobile HTML5 based applications is presented in [56]. It tries to statically identify all functions that interact with input data sources and constructs a call graph from those functions and any dependents. Then it performs taint and data flow analysis on these functions. The approach was evaluated on a corpus of over 15000 of such apps and showed a low false positive rate.

To find missing or ineffective input sanitizer functions in the source code, "POSTER" [57] was created. It automatically extracts all methods that are applied on external inputs and generates test cases in which those functions are evaluated on XSS attack vectors to determine their reliability. A very similar approach was made in [58], where the efficiency of input filtering or sanitizing functions that are found in the source code is tested by automatically generated test cases. This approach focuses on Java applications and considers the context of the output. In [59] a method to automatically generate unit tests and create a testing framework to examine the efficiency of encoding and sanitization functions, in order to find XSS vulnerabilities. Their approach was also developed for Java based web applications. In [60] test cases that exercise possible XSS vulnerabilities and their corresponding paths are automatically generated based on the PHP code. Static taint analysis is used to build a control flow graph. A genetic algorithm followed by concrete symbolic execution create, refine and select appropriate test cases. Similar concepts are presented in previous work by the authors in [61] and [16].

Another research tries to detect XSS flaws with a fuzzy logic system. It analyzes the code metrics with respect to a complex rule set to find injection vulnerabilities and determine their severity. The reference implementation is designed to analyze PHP code [62].

### 2.1.3 Black-box Approaches

Many tools capable of detecting XSS in this area were created by companies that want to sell their vulnerability scanners. So obviously they do not want to reveal publicly the exact details of the inner workings, mechanisms, technologies and strategies their tools follow to detect XSS vulnerabilities, in order to preserve their possible competitive advantage. Many times free versions of those tools (with limited capabilities) exist. Furthermore many open source tools exist which are also lacking documentation about the already mentioned abstract methods and functionalities for their particular scanning approach. More information about those scanning applications is provided in Section 6.1 on page 53. However also several XSS scanning applications are either listed and discussed on various blogs and websites related to vulnerability scanning or discussed in papers [36, 63–70]. Also some benchmarks about their capabilities and performance exist [36–40, 63]. But most of those benchmarks only compare a handful of scanners or test the web vulnerability detection capabilities in general. XSS detection performance is most of the time not a priority and is not analyzed in depth by those benchmarks.

XSS payloads which were introduced through new HTML5 features were investigated by Dong et al.. They created a collection of new attack vectors and used them in a tool for testing web-mail providers. The tool is able to create mails which include malicious XSS content inside various attributes and transform the XSS payloads with different encodings to bypass filters. The results of those test-mails have to be verified manually. According to the authors they found seven vulnerabilities in real world web-mail systems [31].

In [71] Pan et al. proposed a new approach for control flow analysis of web applications to be able to determine the existence of XSS vulnerabilities. It is based on taint analysis, a method in which user inputs are mapped to server responses and the influences of the input data is tracked. Their technique is programming language and source code agnostic. They also try to correctly process websites that employ URL rewriting and HTML input sanitization where, according to the authors, other taint tracking techniques fail.

Duchene et al. developed a XSS scanner which performs black-box testing on web sites. It can detect reflected and stored XSS through fuzzing. A scan starts by building an abstract control flow model of the web application and detects possible taints. After optimizing the model through reduction algorithms, XSS payloads are generated with a genetic algorithm constrained by a special attack grammar to mimic a human like attacker. Previously evaluated payloads and the context in which one was tested is considered in follow-up tests, which enables bypassing of filters. The prototype of their scanner was evaluated on 7 different web applications and compared with a few other open-source black-box scanners. The results show that KameleonFuzz can detect more vulnerabilities than those it is compared with. However it is not possible to handle web applications that heavily use JavaScript and provide a different functionality than the static version does. Also the attack grammar has to be formulated manually and many parameters have to be adjusted by hand, which can change the results drastically [35].

DexterJS is specialized in detecting DOM-based XSS. It retrieves web-pages, instruments the JavaScript code and executes them in a browser engine. Then possible unsafe data propagation is detected and XSS payloads created, which are verified by sending them to the original server hosting the website. The implementation works like a proxy server. According to the paper the tools should be available on `https://dexterjs.io/`, but this seems not to be the case any more because several attempts at accessing it at various times over multiple months failed [72].

ETSSDetector [73] is another black-box scanner. The tool emulates a browser in a Java framework. It focuses on finding all components and data channels of a web application. So it starts by gathering all links and forms. In contrast to other tools it tries to fill forms with valid data to uncover possible hidden data entry points. The testing of XSS payloads is context sensitive, first a dummy is injected to determine the location of the payload then an appropriate exploit is sent. Results are analyzed not only in direct responses to a request but in the whole application, which allows detecting stored vulnerabilities more efficiently. The evaluation was executed on three varying test scenarios and the results compared with 5 other XSS scanners. ETSSDetector is limited to the web-page structure, the authors describe that cookies are not covered and it seems like JavaScript channels are not tested either.

Vishnu and Jevitha propose a XSS scanning technique which heavily relies on machine learning to predict an attack. Their implementation extracts and identifies features of web applications as training data, separated in known vulnerable or exploited websites and secure ones. Seven different features are analyzed in URLs and five in JavaScript code of websites. In their evaluation they test and compare the performance of various classifiers they implemented, considering "correctly and incorrectly classified instances, True Positive Rate (TPR), False Positive Rate (FPR), Precision and the time taken to build the model". Although they do not discuss the evaluated data set in detail and no comparison with similar approaches is given [74].

In [4] the authors address the issue of XSS attacks which contain intermediate steps and are harder to detect than simple reflected XSS vectors. Those so called multi-step vulnerabilities are inserted in an entry point of the web application, stored and later executed on a different point. To be able to identify such flaws the authors created a "Pattern-driven and Model-based Vulnerability Testing (PMVT) approach", in which XSS test patterns combined with a behavioral model of the website generate abstract test cases. From those the actual tests are derived later and executed in the HTMLUnit Java framework. All XSS payloads are taken from the OWASP filter evasion cheat sheet. A significant amount of manual work is also required, the abstract operations of the model has to be manually implemented in the Java framework. The approach was evaluated on two vulnerable test web apps and compared with other black-box scanners. Although a high level of detection was reached, such test runs require a lot of time (multiple hours).

A tool especially for DOM-based XSS detection was created by Lekies et al.. They modified a browser engine to track data flows and identify those which are dangerous and might cause XSS vulnerabilities. Since they can exactly correlate data sources with

their corresponding sinks, context aware payloads can be created and evaluated. The execution in a browser engine avoids false positives. The scanner was evaluated on over 500000 URLs of the Alexa Top 5000 websites and uncovered multiple vulnerabilities [75].

#### Improving Black-box Testing

The question of how to efficiently perform black-box fuzzing of websites regarding cross site scripting and which important aspects have to be considered is discussed in [33]. Especially the questions "How to obtain a notion of coverage on unstructured inputs? How to capture human testers intuitions and use it for the fuzzing? How to drive the search in various directions?" are addressed.

A take on the theoretical and practical concepts of how black-box XSS scanners work, how they generate their payloads and which problems they encounter while performing a web page analysis is investigated in [34]. Existing testbeds are enhanced with new test cases and existing scanning tools are compared. A tool called XSSPeeker was developed to analyze their functionality. It operates on the network layer and analyzes the requests that are made. An extensible testbed framework that was used in the tests was also published publicly [41, 76]. The authors address the research questions of how to improve XSS payloads and how to increase detection capabilities of black-box scanners.

Another interesting approach is shown in [77], which concentrates on optimizing and generating XSS payloads to reduce the overall test space thus improving the speed of a scan. In multiple stages cross site scripting attack vectors are generated, mutated and refined continuously through machine learning with historical test data of previously executed tests. A third party web crawler harvests input channels of interest. Those cover HTTP headers, form data, URL parameters and the end of the URL. JavaScript and cookies are not included. The authors Guo et al. created a prototype implementation which was tested on several real world websites.

In [78] a model for automatically attacking web applications with XSS payloads is presented. Manual work is required to refine the payload generation.

## 2.2 XSS Execution Prevention

This section covers attempts that protect the end user or the web applications from Cross-Site Scripting. The client-side solutions reside in the browser either as part of the browsers default protection mechanisms like Content Security Policy (CSP), as an add-on or even as a modification of the browsers source code resulting in a modified browser binary. Potential XSS exploits are filtered and blocked. They lack the detection of stored XSS.

Server-side approaches introduce additional layers of security like Web Application Firewall (WAF) or special proxies that inspect and filter or sanitize the traffic. Also several approaches that discuss the effectiveness of various sanitization and filtering

functions and libraries or the integration of security evaluation tools into the development life cycle exist.

### 2.2.1 Client-side Approaches

#### Content Filtering

Today many browsers include default filtering mechanisms, which target reflected XSS. In browsers using the WebKit rendering engine like Chrome, Chromium and Safari the filter determines if a piece of JavaScript code that can be found in a HTTP response was also present in the associated request. When this is the case it will take further actions to block its execution [75, 79, 80]. In Internet Explorer already since version 8 a similar technique is used. It is enabled by default but can be disabled by sending a simple HTTP header `X-XSS-Protection: 0` in the response. Those filtering components are just a simple defensive layer and should not be considered as sophisticated mechanisms that make browsing secure against all kinds of reflected XSS. They offer some advantages for the broad masses of inexperienced users, because no extra configuration or user interaction is required which might be confusing [81, 82]. However many ways to bypass the default browser filters exist [83–85].

Another security mechanism which every modern browser supports is the Same-Origin-Policy. It limits the possibilities in which resources that are retrieved from varying locations (different hosts, ports, protocols, etc.) can access and communicate with each other. Thus scripts loaded from untrusted sources can not offhandedly manipulate trusted content [86, 87].

Browser add-ons like NoScript, ScriptSafe or uMatrix (and others) allow fine-grained control over the content that is loaded in a web page. They are far more aggressively blocking content than the previously discussed browser filters and most likely break websites that rely on JavaScript or other dynamic elements (embedded content, multimedia, etc.) when their default configuration is used. The detection algorithms heavily use regular expressions. It is up to the user to allow specific elements and remote resources [88–90]. XSS-immune is an extension for the browser Google Chrome. It tries to defend against Cross-Site Scripting attacks by comparing scripts existing in the request with those in the response and JavaScript injections in parameters. Another feature is the context aware handling of XSS worms and auto-sanitization of them. The Java based extension was tested against several open source attack payloads [91]. In [92] three different filtering components are applied on each request and response the browser performs. Those filters restrict loading or sending data to/from cross domain resources, remove dynamic script creation and block inline JavaScript, pop-up windows, cookie access in combination with cross-origin links and frame creation at runtime. The authors claim that only malicious elements are filtered and benign code is unaffected. An evaluation of those claims is not shown nor discussed. XSS-ME [93] is an add-on for the Firefox browser which tries to test all forms of a website, sending them to the server and analyzing the response. So it follows an offensive approach in its first phase, by trying to

attack the web application. Then in a second phase, when it detects XSS vulnerabilities that are reflected back it is able to block those scripts and warn the user.

A taint tracking browser engine is proposed in [82] to battle XSS. It tracks data flows and provides the information to HTML and JavaScript parsers which allows assorting XSS fragments already in the parsing stage. Code originating from tainted data sources is not allowed to execute in the JavaScript engine which is also enhanced with taint information. The approach was implemented as a modification of the Chromium browser. It is especially targeting DOM-based XSS.

#### Content Security Policy

One of the most widespread defensive mechanisms, which has become a W3C standard, continually evolves and is increasingly adapted by browser vendors and web application developers is the so called Content Security Policy (CSP) [94–96]. When present on a web page, it determines which resources can be accessed, where data can be sent to and retrieved from and which scripts can be executed in the context of the web-page. This allows a fine grained control of different sources and eliminates many possibilities for malicious content injections such as XSS. Inline JavaScript is blocked by default for example. However there is need for improvement as recent research has shown that numerous side-channel attacks exist which can bypass security features of CSP. This is caused partly because of ambiguous or imprecise specifications and also because of edge cases not considered. For example CSP bypassing methods through DNS and resource prefetching and caching [97]. Often the CSP configuration is ineffective. Weichselbaum et al. have analyzed the CSP of about 100 billion web sites and identified "that 94.68% of policies that attempt to limit script execution are ineffective, and that 99.34% of hosts with CSP use policies that offer no benefit against XSS" [98]. Another study about CSP adoption found that its use was not very widespread and that most of those web applications that used it, had a weak configuration which did not prove efficient at countering injection attacks [99].

To address the issue of faulty CSP configurations, AutoCSP [100] was created. It is a tool which analyzes web applications and derives proper CSPs. It is even capable of automatically implementing the required server side changes. The presented prototype was created for the PHP language.

In [101] multiple weaknesses and how they could bypass a CSP are discussed. To oppose them a policy based approach called PMHJ to increase the security of CSP and add further preventive mechanisms is shown. It even further expands the granularity down to single HTML elements, introduces integrity protection for them to counter structural manipulation like node-split attacks [102] and unsafe JavaScript functionality is disabled. Since the blocking of JavaScript that is considered unsafe is accomplished by blacklisting, it suffers from possible missing entries. Furthermore the policy needs to be supported on the server and on the client side. A lot of work has to be done to implement the policy, because the HTML elements need "nonce" attributes for integrity checks and JavaScript

code has to be adapted. This has to be done on every website. Also the browser code needs to be changed to understand and enforce the policy.

Recent work on CSP discovered a new methodology to bypass CSP protections through so called "script-gadgets" which are present in most websites that make use of popular JavaScript libraries. In such attacks HTML code can be injected into a website where it is later accessed by already existing JavaScript code that reacts to the presence of the HTML code in the DOM. The JavaScript code transforms the piece of HTML, that in itself is not malicious, into some executable JavaScript. In 13 out of 16 analyzed JavaScript libraries and frameworks such script-gadgets were discovered that make this kind of code reuse attack possible [5].

### 2.2.2 Server-side Approaches

#### Additional Security Layers

Prandl et al. studied the efficiency of different free Web Application Firewall (WAF) [103] solutions. A WAF is a proactive defensive solution and works similar to common firewalls which block malicious traffic. Instead of inspecting low level network traffic they examine application level protocols in order to detect and prevent attacks targeting web applications. The advantage of such firewalls is that no source code changes have to be applied and multiple services can be protected at once. However this comfortable way of protecting the application by a separate maybe even third party source is no guarantee for perfect security. The actual vulnerabilities of the web application itself will not be fixed and might be exploitable through different channels bypassing the WAF [104]. In the study by Prandl et al. they also discovered that the security is highly dependent on the quality of the WAF. Too restrictive configurations might block many benign requests, on the other hand attacks might be missed if they are too liberal. Advanced, new and sophisticated payloads could also sneak through. In [105] a high level analysis of techniques for detecting and preventing XSS through log file analysis, the usage of WAFs and filtering are discussed. In [106] algorithms for a web application firewall that can detect XSS attacks among others are presented.

The approach of [107] deals with creating an abstract model of a website to identify and distinguish between benign and malicious users. First the model is created by collecting all static resources and analyzing dynamic actions a user should be able to do. When the web app is deployed and used in the production phase, requests are validated against the model and flagged if discrepancies are found.

Noncespaces by Gundy and Chen tries to protect the HTML content from being altered in a malicious way by an attacker. It introduces randomization of tags and attributes which allows identifying third party code and protects the integrity of the document structure. As long as the randomization is not guessable by an attacker he/she will not be able to manipulate the DOM tree in a way that allows XSS execution. Additionally it is possible to apply policies to the untrusted content provided by users. Their implementation is based on a PHP template framework. Sanitization becomes unnecessary with this

approach. However for its full functionality to work there has to be a proxy application running on the client side [108]. A similar approach that tries to protect the HTML is presented in the paper about XSS-SAFE, a web application framework for Java/JSP projects. It injects tokens that are based on the content into the HMTL and JavaScript code at the server side. This works like a hash function over parts of the web page. It enables the detection of any alterations or injections which can in turn be sanitized by the framework [109].

In [110] a framework for PHP applications is proposed that can detect injection attacks based on computing the divergence of the actual input with the expected input using an algorithm based on the Kullback-Leibler distance. In a setup phase the code is analyzed and the expected input is derived. The web applications source code is instrumented with analysis components. At runtime the inputs can be checked and attack payloads identified when the HTTP response would contain unexpected data (e.g. JavaScript code).

A complete architecture for XSS safe Java web applications is proposed in [111]. A reverse proxy in front of the actual web server, that inspects the HTTP requests and responses for scripts, forwards the request to a sanitization component. The input is normalized (like charset encoding, HTML structure and tags), XSS filtering (encoding of special characters into HTML escape sequences) and comparing against whitelisting patterns is performed. Also a special model for accessing databases is proposed. Similarly the authors of [112] show how to utilize Snort, a popular intrusion detection system, to detect XSS attacks outside of the actual web application. This approach relies on static pattern matching and tries to recognize the payloads that would be injected. In [113] a detection framework in the shape of a proxy server is proposed. A significant part of the workload is offloaded on the client, where a request is preprocessed and might already be dropped if certain conditions are satisfied before sending it to the proxy. In the proxy features are extracted and analyzed which determine if an attack is in progress.

#### Secure Programming and Sanitization

The paper [114] encourages defensive programming techniques to harden a web application already at development time against XSS attacks. The evaluation of their method was conducted on two versions of a test web app, one implemented while using defensive programming, the other one without it. This security model "blocks all the html tags, scripts, programming language, constructs, event handlers, character codes, insecure keywords, if present in input, but allows only those tags that are known to be safe for performing XSS" [114]. In contrast to casual sanitization filtering Maurya applies an additional sanitization step on the input data before passing it to the actual filtering component of the web application in order to get a completely clean input before any attempt at storing the information persistently is made. Two whitelists are also employed, one for HTML elements and one for their attributes. The major problem of this solution is the need for manual fine tuning and configuration. Unknown or unexpected attacks might not be recognized, especially because only a list of well known XSS vectors is used

in their evaluation. Teto et al. focus on defensive programming principles and propose them as the best countermeasure against XSS vulnerabilities [115].

Templating frameworks are commonly used in web applications today, as they reduce the need to repeat code and make it possible to reuse components. To reduce the risk of missing output sanitization or applying the wrong transformation functions Samuel et al. introduce a framework which automatically handles the filtering issues. With respect to the context and the data types, the values which get dynamically inserted in the web templates get secured against XSS attacks. The largest part of the approach can be executed statically thus only a small performance overhead exists. Only Java is currently supported on the server side [116].

Multiple sanitization and filtering libraries as well as API functions for the languages Java, PHP and ASP.NET are tested and compared in [117]. The tests were achieved by creating a simple web application where the relevant methods were applied, one at a time on the input. The effectiveness regarding security and performance regarding processing time was recorded and presented. In [118] various defensive approaches against injection attacks in web applications are analyzed regarding their effectiveness, runtime impact and ease of use. Besides discussing an extensive number of tools, the authors also take a look at the practicability of using those defensive applications an libraries.

JoanAudit [119] helps developers detect possible injection attacks by analyzing security related code fragments and reporting flaws. It can even automatically fix vulnerable parts of the code. The bytecode of Java applications is processed and analyzed. In [120] a method for automatically transforming web applications written in ASP.NET in order to separate data and code (HTML and JavaScript) is presented. This is especially useful for legacy applications because the approach works on the application binaries. This should prevent XSS injections and make it easier to introduce CSPs. There are several limitations imposed on the web application and the programming APIs it uses by this approach. Furthermore only ASP.NET is supported.

CHAPTER 3

# Background

Before we advance to the description of FOXSS and the details of its internal mechanisms the concepts of XSS are presented. It is necessary to understand how this type of attack on web applications works and which variants exist. Further a short explanation of black-box vulnerability testing is provided, since FOXSS falls into this category of vulnerability scanners.

## 3.1 Cross-site Scripting

Cross-Site Scripting is basically a special type of injection attack in which untrusted input gets embedded in the HTML document of an application and then interpreted as if it was an actual part of this document. It allows an attacker to execute JavaScript code in the context of the browser of a victim. The injection can occur as part of an HTTP request or locally when user input is handled by JavaScript code, processed and included in the web page. XSS vulnerabilities allow the inclusion of markup ranging from simple HTML tags manipulating the appearance of the page layout to JavaScript code that can furthermore execute a multitude of malicious actions on behalf of an attacker. The main cause why this type of attack is possible, is the absence of proper input filtering or sanitization in web applications when data from an untrusted source is accepted, processed inside the app and included in the output. Since the attack surface of websites is many times huge, it is often hard to cover all possible input channels. XSS payloads can be infiltrated through URLs, forms, cookies, HTTP headers and other data sources that are visible to the web-server, the programming language runtime or the web application on the client side.

A typical example would be a website that contains an input box (e.g. to search for some items). The data of this input box is passed as parameter in the URL to a server side script that generates the response dynamically. In the resulting response page the data of the input box (i.e. the search term) gets included. An exploitation of this functionality

is illustrated with the following URL that accepts the parameter `p` which carries the data of the input box.

```
http://example.com/?p=<script>alert("XSS")</script>
```

When a value like this is provided and simply mirrored into the resulting response by a PHP script like the following:

```
<html>
<head><title>test</title></head>
<body>
    The parameter p was: <?php echo $_GET['p']; ?>
</body>
</html>
```

Then the script tag and its contents will be embedded and interpreted by the browser, resulting in an "alert" message pop-up. So an attacker could craft a link that leads to this URL and provide arbitrary JavaScript code. A user that gets tricked into following this link will then unknowingly execute the code in the context of his or her browser with all the privileges and access to data the website has there. While this is a trivial example that can be easily avoided, many more sophisticated techniques exist [13, 15, 18, 77, 121–123]. In the year 2000 one of the first XSS attacks was publicly documented [75]. Since then they have evolved a lot.

Now that we know how XSS works and that we can execute JavaScript one might ask why do we need to worry about this? JavaScript runs in a sandboxed environment in the browser. The interaction with everything else outside the browser is very limited. There is no direct access to read or write files in the file system. You can not execute or install arbitrary programs or change any settings of the browser or operating system, without the user explicitly allowing it or taking actions by him/herself. But JavaScript can do a lot of other things that might not seem dangerous at the first glance. It can record keystrokes, clicks and several other user interactions. It can make HTTP requests through many different techniques. While it cannot read from arbitrary network locations it can send data to arbitrary URLs and IP addresses. It has access to all the data the vulnerable website can write data to, like cookies and web storage. It can manipulate the appearance of the web application and hide malicious actions behind benign ones.

All this leads to a multitude of serious threats to which users of a vulnerable web application are exposed. A malicious actor can perform phishing by changing the target of forms or duplicating the data when the submit button is clicked and also sending them to their own server effectively stealing the data. Another possibility would be to create new forms that appear as legitimate part of the website and grabbing additional data that a user would never be required to share on the website this way. A keylogger could be easily injected into the website that records everything a user writes. This

can be achieved through listening for DOM events that are emitted when a user presses keys of the keyboard. Cookies of an authenticated user can be stolen. This allows the attacker to overtake the account of a victim and perform actions on their behalf like transferring money. Cryptographic keys that are used by a website and stored in the browsers local storage can be read and abused to decrypt private data. The address where a user sends a payment or sensitive information can be changed behind the scene while the user still sees the one he/she typed. The data would thus be transferred to some different destination without the user noticing. An attacker could create a botnet out of a websites users making them involuntarily participants in Distributed Denial of Service (DDoS) attacks that can take down web applications by producing excessive loads of HTTP traffic.

## 3.2 Cross-site Scripting Types

There exist three distinct categories in which XSS attacks can be classified. Throughout the literature it is generally agreed on those three. In some cases DOM-based (type-0) XSS is considered as part of reflected XSS [18, 121, 124–126].

### 3.2.1 Reflected (type-1)

In a reflected XSS attack, the payload that should be executed is present in the HTTP request and will be transfered back to the user in the response. The web application that is running on the server is responsible for embedding the payload in the HTML response. Usually this is done by an adversary through tricking a victim into clicking a prepared link (like a HTML link that hides the actual URL text) that contains the exploit code.

A visualization of an example can be seen in Figure 3.1. The web application in this example can be seen as some kind of social network where users can view pictures. The parameter `p` of the URL where a picture can be viewed is vulnerable. The steps required to perform a reflected XSS attack using a vulnerability in this example web application are as follows:

1. The attacker identifies the XSS vulnerability in the web application that can be abused for malicious purposes.
2. The attacker sends a poisoned link to the victim (in this example as part of an email) and tricks him/her into clicking it.
3. The victim accesses the link containing the XSS payload and loads it in the browser.
4. The payload that was sent with the request is included in the response by the website.
5. In the browser of the victim the exploit gets executed and the session cookie gets stolen allowing the attacker to log into the victims account and impersonate it.

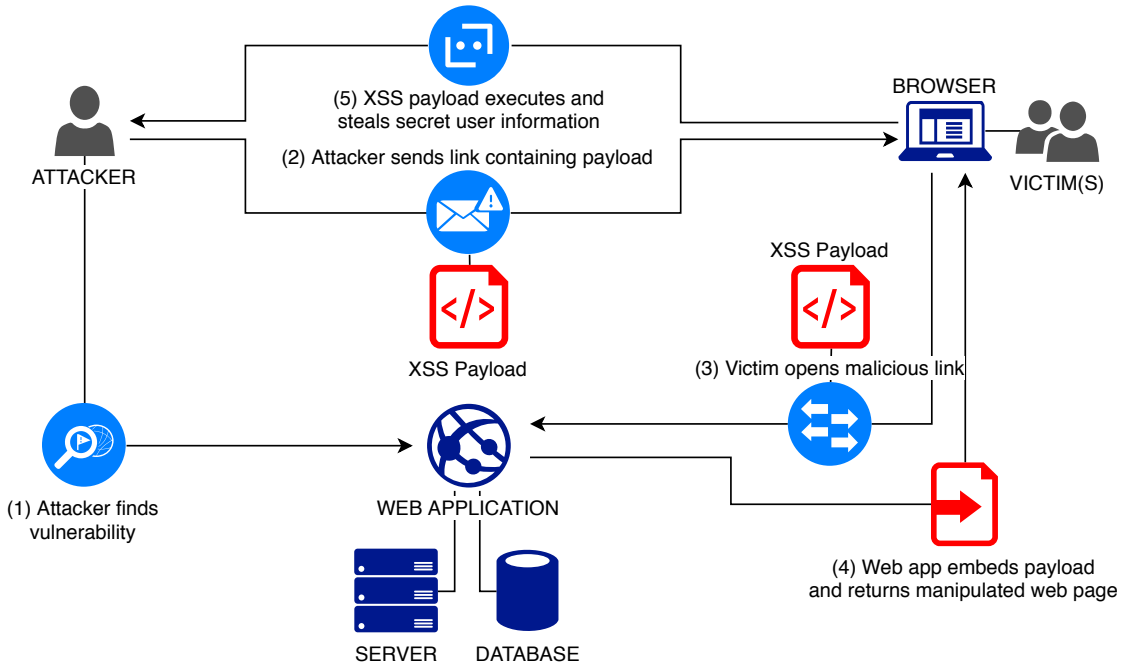


Figure 3.1: Visualization of a *reflected* XSS attack.

### 3.2.2 Persistent (type-2)

Persistent XSS, also known as stored XSS, gets permanently saved in the web application and is always served to every user that visits a vulnerable page that includes the payload. So no direct interaction with a victim is required. It is harder to detect than reflected XSS, since it is not necessary that the malicious content can be found in the request. Also the payload might appear on a completely different and unrelated page than where the actual injection happened because it is loaded from some kind of persistent storage.

A visualization of an example can be seen in Figure 3.2 on page 21. In this example of a persistent XSS attack, the website represents a simple message board where users can post comments. On the page there is a HTML form to submit new messages which will be stored in a database on the server by the web application. Furthermore the last message is always displayed on the page. Data submitted through the form will be shown there without any sanitization applied. To exploit this behavior the following steps can be taken:

1. The attacker identifies the XSS vulnerability in the web application that can be abused for malicious purposes.

2. Then the attacker injects JavaScript code via the form when posting a new message. The website saves it in its database.

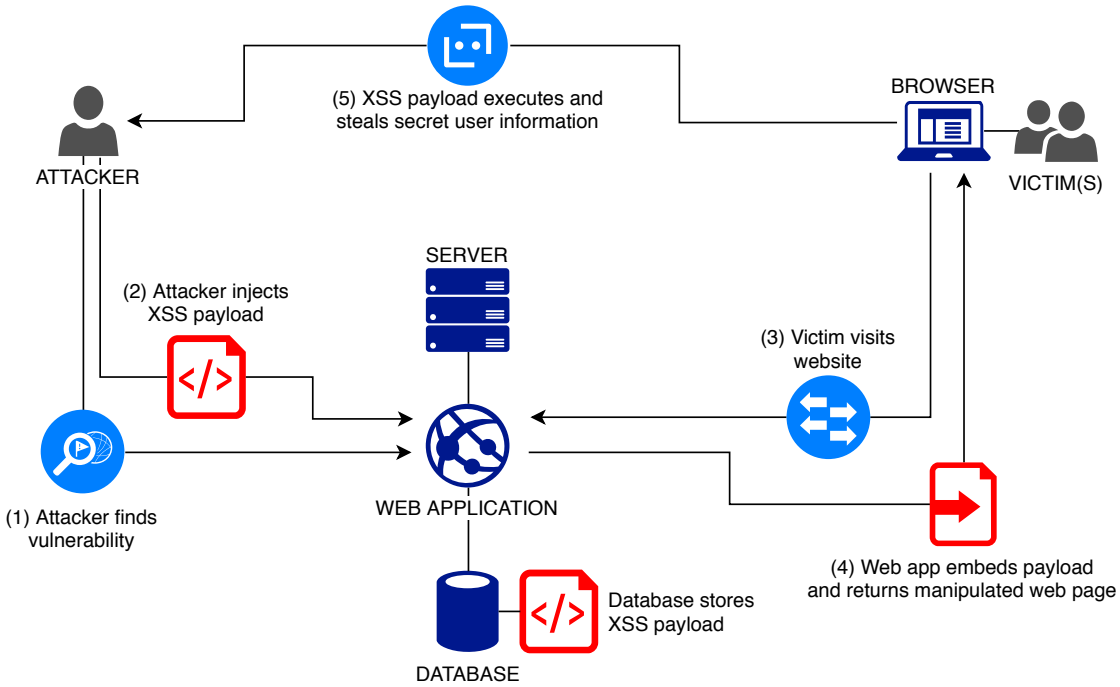


Figure 3.2: Visualization of a *persistent* XSS attack.

3. A victim later accesses the page where the last comment can be viewed. The attacker does not have to interact with the victim at all.

4. The web-server delivers the page in the response. The malicious JavaScript payload gets loaded from the database and inserted, afterwards it is interpreted by the browser of the victim.

5. The payload gets executed and like in the previous example sends the victims session cookie to an attacker controlled server.

Note that any user that visits the page and sees the message the attacker posted is susceptible for the attack. The exploit will send the cookies of users to the attacker as long as it is accessible on the website, which in this specific case can be an unlimited amount of time as long as the attack payload does not get removed from the database or a newer comment gets posted such that another message is shown instead of the attacker's.

### 3.2.3 DOM-based (type-0)

The third type of XSS called DOM-based XSS, named after the DOM [127], can have some similarities with reflected and persistent XSS. Like in reflected XSS the URL is also one of the most interesting attack points here, but instead of involving a server,

type-0 attacks can appear in purely static HTML sites. Also it can be stored persistently somewhere in the browsers data storages and later retrieved and executed from there. The main difference between this type and the other two is that a server is not involved in the actual vulnerability, but the client-side JavaScript code. This is the reason why it is sometimes also called local XSS. It was also documented much later than the other two types, first in the year 2005 [128]. In DOM-based attacks JavaScript code that already exists in the page is utilized and serves as an entry point to either manipulate the structure of the page through methods provided by the DOM API or perform a conversion of the string payload to JavaScript code and run it. A script that accesses the URL or parts thereof and mirrors this information somewhere in the page, can be abused for XSS injection as an example. The URL is not the only input channel, several others exist and are explained later in more detail.

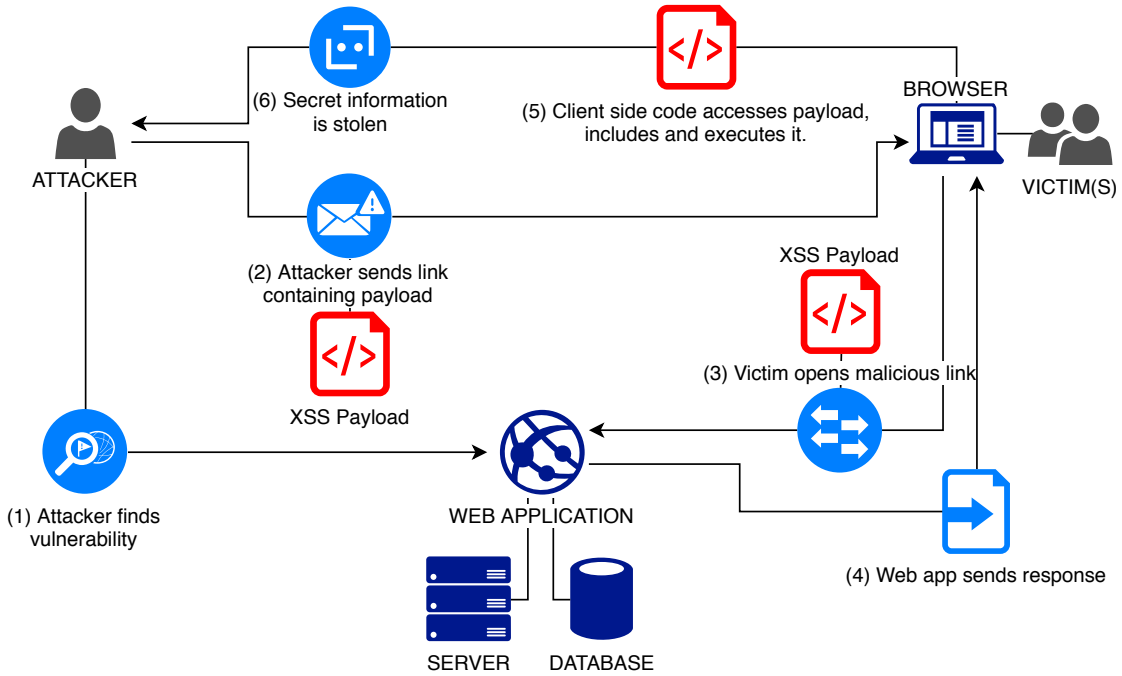


Figure 3.3: Visualization of a *DOM-based* (client-side) XSS attack.

An example is visualized in Figure 3.3. This time the website consists of a single purely static HTML document. The server does not manipulate the response in any way, it just serves the HTML file. The functionality of this page is just a simple search box, but instead of processing the search on the server, the client side JavaScript is responsible for that. The DOM-based attack that can be seen works as follows:

1. The attacker identifies the XSS vulnerability in the web application that can be abused for malicious purposes.

2. Similar to the reflected attack, the attacker sends a URL to the victim. The payload is included as the value of an URL parameter.
3. The victim clicks the link and retrieves the web page.
4. The website returns the response, which is still clean at this point, it is just the same static HTML file that would be served if no parameter was provided.
5. As soon as the browser interprets the response it executes the JavaScript. The payload will get accessed by a JavaScript function, included and executed.
6. Finally the attacker can steal the sensitive information.

Since code like this, where a user can perform simple tasks on a website like a search box, is increasingly offloaded on the client side in order to save processing power for other tasks, this type of XSS is more prevalent than ever.

## 3.3 Black-box Vulnerability Testing

In black-box testing the functionality of an application is tested without regard to the actual inner workings. Knowledge about the implementation, the programming languages or the technology stack behind the application is not necessary. This property makes black-box testing approaches portable which is important since web applications are written in many different programming languages. For almost all languages a web application framework exists, which removes the hassle of dealing with low level HTTP requests and other common issues and makes developing for the web in a certain language easy. So a generic black-box vulnerability testing approach once developed can be used for practically any web application without further changes. However there are also several challenges specific to this type of testing methodology.

The absence of the actual source code makes it necessary to explore all possible entry points of the web application where external data is accepted and processed. Usually the different pages and subsections can be found by following links from a starting page at the root and recursively iterating over all newly discovered links. Problems might arise when pages are not linked with simple HTML anchor tags but dynamically generated buttons and links. Then the scanner needs to interpret the HTML and JavaScript in order to get to the correct links. Also handling authentication protected areas and avoiding state changes in the web application need to be considered. When the scanner triggers an irreversible state change and locks down the web application it might have a disastrous impact on the vulnerability analysis phase. For example if the program deletes several data records that open special areas in the web application where a vulnerability resides it might not be accessible later on and makes it impossible to discover the vulnerability.

Another challenge is the generation of test data or better the actual attack payloads. The more different payloads are generated the more time is consumed to test them against the

test subject and determine if they reveal a vulnerability. The detection rate also depends on sophisticated payloads. It is important to minimize the amount of test data and different test cases while having enough payloads to exercise all functionality provided by a web application and maximize the discovery of exploitable flaws. The questions of how to mutate and adapt payloads when filtering mechanisms are applied, how to deal with sanitizer functions and if they can be evaded need to be thought of.

Furthermore a black-box vulnerability scanner needs an oracle to determine whether a payload was actually successful and found a vulnerability or if the test case failed and the web application handled the input data correctly or at least in a way that does not open the door for an attacker. This can be as simple as performing a string search in the HTTP response, but it can also be much harder, especially in the case of XSS testing. For example when trying to identify DOM-based XSS there is no response to inspect. A JavaScript interpreter is necessary to track down through which functions the data gets propagated and where it ends up. For a static scanner it makes no difference if the data gets escaped and securely stored or passed to a function that parses it as if it was JavaScript code. It will be blind and can impossibly detect the vulnerability other than taking a lucky guess. When searching for vulnerabilities that get persisted in the web application the author(s) of a vulnerability scanner also needs to consider how to map the payload that triggered the vulnerability to the actual payload that was sent to the entry point. This might be on a completely different location and might be transformed on the way to the exit point where it appears on the surface again.

CHAPTER 4

# FOXSS

In this thesis we present FOXSS (short for "Finding Only XSS"), a prototype implementation to automatically detect XSS vulnerabilities in Web applications. It is written in JavaScript, uses the electron[1] framework as its runtime and consists of more than 7000 lines of program code. The system can detect all types of XSS vulnerabilities in web applications automatically and requires only minimal configuration. This is achieved without the need of a human analyst to manually provide information about which parameters or entry points to use. All potential vulnerabilities are verified by the system and, if found valid, at least one proof of concept exploit will be provided which is guaranteed to execute and being able to trigger arbitrary JavaScript code. This approach ensures zero false positives and easy confirmation of the results for a human analyst. Furthermore, the payload template generated by FOXSS can be used to build custom XSS payloads to further verify and expose the impact of the found vulnerability.

On the technical side, the majority of requests during analysis of a web application are made through a browser engine. This enhances the process to reliably detect the necessary contextual information. It must be ensured that the payload data, which consists of HTML and JavaScript is interpreted in exactly the same way as it would be in a regular browser. Environments that emulate a browser are not enough for this purpose. They often lack features and do not provide the same level of support for the HTML and JavaScript standards as an up to date regular browser. This might void certain DOM elements or JavaScript features that can be abused for XSS attacks. Analysis results might become imprecise and miss crucial aspects that are required for the exploit generation algorithm. Also the verification might not be the same, thus leading to either false positives or and increase of false negative results. However, in some cases the raw textual HTTP communication is important. Especially when filters and encodings in the response data need to be detected. JavaScript APIs for accessing the DOM only

[1] `https://electronjs.org/`

have access to the data that was already interpreted by the browser engine. But the raw data might have been changed after it was processed by the browser. Furthermore, the delimiters of attributes of HTML elements can only be detected by statically analyzing the raw textual response.

The electron framework provides the perfect basis for our needs. It combines `Node.js`[2] with the Chrome browser and builds a comfortable API over those two components. So we can use a modern browser engine which allows us to hook and manipulate JavaScript at a lower level than the one that is available to the tested web application while not needing to manipulate it at the source code level of the browser engine. This keeps our changes portable between updates of the browser engine and also allows us to make changes adaptable at a fine grained level. Some JavaScript modifications are just relevant for certain requests, the processing of a group of pages or only in a specific stage of the analysis. We can also use regular HTTP requests provided by `Node.js`. The ability to process a web application like it was run in a real browser also opens the possibility to simulate user interactions and better analyze applications that heavily use dynamic content. For data extraction the DOM APIs can be used, which provide simple methods for the search and access of specific attributes as well as the gathering of contextual information. Custom parsing code that might be error prone can be avoided in many cases.

To accomplish the previously made claims there are basically three steps involved in the analysis process of a web application:

i First it is necessary to identify all or as many data flows as possible. Without knowing where user controlled data can enter the application or failing to find the corresponding exit sink makes it impossible to further test these potentially vulnerable contexts. Also data validation and filtering mechanisms that might be in-between an entry and an exit point are important. Finding and correctly analyzing their behavior provides vital information which can be used later in refining the payloads to test. The more fine-grained information that can be gathered in this first step, the more precise attacks can be created which avoids unnecessary tests and as a result spares time and processing power later on.

ii After the data flows are located the next step is to create a set of payloads that could be executed according to the contextual constraints like the surroundings of the exit point and filtering techniques. Since most of the time there is a large degree of unknown parameters, multiple different payloads and variations thereof are generated.

iii In the third step the payloads are encoded according to the information that was gathered about the entry point and tested until the execution of one is triggered. At this point testing of the data flow can be stopped, since a valid exploit was

[2] `https://nodejs.org/en/`

found. Otherwise the number of untested payloads will exhaust after some time which will categorize the data flow as safe.

## 4.1 Data Flow Analysis

The identification of data flows is the first and most important part in the whole process from requesting a URL to finding an exploitable vulnerability. A *data flow* in the system consists of a single entry point and one or more exit points. An *entry point* contains all information to make a certain HTTP-request as well as instructions to perform optional user actions on the web-page like clicking a button or similar. All fields of the entry point that can be used as data entry sources are stored individually and easily accessible. An *exit point* contains a reference to a specific field of the associated entry point that is relevant for reproducing the injection of data at a certain location by using the entry point information and replacing this field with a payload. Furthermore all relevant context sensitive information about the final data sink is stored. If the sink is in a HTML context the hierarchy of parent elements, other attributes of a HTML element and attribute delimiters are saved. In a JavaScript context the surrounding code and function calls are saved. Possible filtered characters are also analyzed, added and saved in a separate scan stage. Also included is the actual URL since the exit point might be located on a different page than the entry point was, because of redirects or persistent storage of the input data that was passed to the entry point.

### 4.1.1 Entry Point Detection

Detecting entry points combines the obvious approach of extracting the targets of link elements with more advanced ones. Naturally a user navigates the web application through links that can be most commonly found in HTML anchor tags. But there are several other possibilities to discover areas of the web application. Basically all `href` attributes and others which might reveal structural information like `src` and `data` attributes of elements are extracted. Also the targets of forms are resolved with respect to the host of the web application, normalized into the internal URL format and added. For all requests non-standard HTTP headers that might appear are recorded. This data could also be extracted in a static environment, just dissecting the raw HTTP requests. However many more entry sources are gathered dynamically by instrumenting the JavaScript that is exposed by the browser engine. So URLs and other data relevant for entry points which might be assembled at runtime can be gathered. Functions that can perform HTTP requests or retrieve data somehow need to be monitored. Those include the XMLHttpRequest, Websocket and fetch objects. Access to unsafe data sources like `document.cookie` and `document.referrer` that can be accessed by JavaScript but are dependent on HTTP headers are observed. Mechanisms to share data between different origins like setting `window.name` and the `postMessage` API need to be observed. Since those dynamic and potentially also the static features might be hidden behind some user interaction and generated dynamically, after the page load

click-, submit- and drop events are artificially crafted and dispatched in order to reveal them.

The full list of entry points and data sources that are interesting for XSS and get tested in FOXSS are listed in Table 4.1.

### 4.1.2 Exit Point Detection

All URLs that were gathered in the previous phase are split into their parts creating a large number of possible entry sources. Every path element, query parameter (name as well as value), hash, header, cookie and found DOM-entry point is separately tested. For each of those entry fields a unique identifier is generated, their original value is replaced and a modified request is sent. The exit point detection component will then examine the response for those identifiers.

Like the entry point analysis, the exit point analysis consists of static and dynamic analysis components. Static analysis uses XPath selectors to detect exit point identifiers located in element name, element content, HTML comment, attribute name and attribute value contexts. The dynamic analysis component hooks interesting JavaScript functions and objects. When one of them is called at runtime they are automatically detected and the calls inspected whether they contain our identifiers or not. These functions include the *script generating* contexts `eval`, `setTimeout`, `setInterval` and various aliases of the `Function`-constructor call, all of which can create JavaScript code at runtime. *DOM-generating* function calls like those to `document.write`, `document.writeln`, `Element.prototype.innerHTML`, `Element.prototype.outerHTML`, `Element.prototype.insertAdjacentHTML` and `createContextualFragment` of the `Range.prototype` object. Also various versions of `document.createElement` in combination with setting a specific attribute are intercepted. *Location changing* contexts resulting through the manipulation of the `location`-object together with its functions and properties and the `window.open` function are dangerous sinks and are monitored. HTTP request methods in JavaScript provided by `XMLHttpRequest`, `Websocket` and `fetch` get observed for *request* contexts. Additionally all DOM-insertion methods are proxied and checked if dynamically generated elements which have some properties that contain the exit point marker are passed to them. Otherwise they would be missed since they are not detected by the static analysis.

In Table 4.2 all relevant exit contexts and their specific sinks are listed. A description of why they are interesting and what they can do can be found next to them.

### 4.1.3 Filter Detection

Filter analysis requires the raw HTML response in textual form. The HTML response interpreted by the browser might be cleaned of erroneous elements and invalid characters, but to detect filter transformations in particular of special characters that resemble control sequences, the output of the web applications needs to be original. That is why

| Data source | Description |
|---|---|
| URL sources | |
| path fragment | URL paths are split up on slashes. For each of those parts/fragments a test is generated. Example: /a/b/c would lead to 3 tests. |
| query parameter name | The name part of a query parameter is tested separately. Example: For ?q=a a test for q will be created. |
| query parameter value | Example: For ?q=a a test for a will be created. |
| hash | The hash part of an URL. It is often used in SPAs and is an important source for DOM-based XSS. Sometimes it is used for navigation and a path like structure is appended there. |
| auth* | URLs might have an auth part consisting of a user:password. This is currently not tested. |
| HTTP sources | |
| Header | There are many different HTTP headers that could carry exploit data. In this approach some commonly used standard as well as non-standard headers are tested: Cookie, Date, Forwarded, From, Referrer, User-Agent, Via, X-Requested-With, X-Forwarded-For, X-Forwarded-Host, X-Request-ID, X-Csrf-Token. |
| Body | HTTP requests that can carry data in the body, like POST, can also carry attack vectors there. Depending on the encoding type name-value pairs are tested or just a single test for the body is created. |
| DOM sources | |
| document.cookie | Cookies are exposed in the client side. Name and value will be tested. |
| document.referrer | Also the HTTP referrer if available is exposed. |
| window.name | This property can be used (although not intended for this use) to exchange data across different frames. As such it can be manipulated from the outside without restrictions and has to be considered always as dangerous when data is read from this property. |
| postMessage API | This API was designed to communicate between different frames and windows. The programmer needs to ensure only data from valid origins are accepted, failing to do so makes this a dangerous source because anyone can send data. |
| location** | Access to the full URL. |
| location.href** | Access to the full URL. |
| location.hash** | Access to the URL hash. |
| location.search** | Access to the URL search part containing the query. |
| location.pathname** | Access to the URL path. |
| document.documentURI** | Access to the full URL. |
| document.URL** | Access to the full URL. |
| document.baseURI** | Access to the full URL. |
| localStorage*** | Data storage that can only serve as an indirect source because something needs to be stored before retrieving it. |
| sessionStorage*** | Data storage that can only serve as an indirect source because something needs to be stored before retrieving it. |
| IndexedDB*** | Data storage that can only serve as an indirect source because something needs to be stored before retrieving it. |
| history.pushState*** | Indirectly manipulates the location object. |
| history.replaceState*** | Indirectly manipulates the location object. |

* Currently not used in the approach.
** Not separately tested and/or observed, since they are already covered by the tests for URL sources.
*** Not a direct source, needs another source to be useful. Not currently used.

Table 4.1: Possible entry points and data sources.

only HTTP requests are used instead of performing the requests of this analysis stage in the browser engine.

Since the actual data flows are already known at this stage, only those need to be tested instead of all possible entry points. Payloads consisting of identifiers and special characters get injected into the input channels and the resulting string that arrives at the exit point is analyzed. If the original string can be found it is likely that no filters are active. On the other hand when some characters are missing or replaced with others this information about those characters is saved.

The most important characters are angle brackets `< >` for HTML tags, quotes `" '` for attributes and parenthesis and the backtick character `( ) ` ` for JavaScript function calls. Several more interesting characters or strings can be tested like `data:` and `base64;` for data-URLs. Depending on the actual exit context of the respective data flow the importance of these might vary. In a HTML attribute context angle brackets and quotes are important, because the availability of them determines if a breakout of the context is possible which increases the number of potential XSS payload candidates. In a JavaScript context the availability of parenthesis and backtick character are important because either the former or the latter are necessary to perform a function call. Also the other characters are interesting in this context since a breakout might be the only option for execution, depending on the surroundings.

A few problems need to be considered. Testing all special characters at once would be the fastest way since the number of requests is at the minimum. Depending on the filter, testing multiple characters at once might lead to some interference. For example a filter might strip the whole input if one of the characters is blacklisted. Testing one character or string at a time would avoid this, but also increase the scan time the more special strings are tested. It might be even faster not restricting the payloads at all by the filter information. There might also be filtering mechanisms that only remove certain characters in combination with other characters, like a filter that removes angle brackets when a HTML-element is detected but not otherwise. In general the filter functions cannot be reconstructed at all times. This is why the first approach, which has the least performance impact is used.

## 4.2 Payload Generation

All data that was gathered up to now will influence the payload creation and selection. Each exit context has its own set of attack vectors which are then reduced according to the available filtering information. If possible alternative representations that avoid restricted characters are generated. The availability of a breakout from the current context to a different context is also considered.

The attack vectors were composed from various sources [32, 129–132]. All of them meet the requirement that they will *automatically and directly execute* a specific piece of JavaScript code when they are injected successfully into the web-page. No additional action is

required. Furthermore they should run in all modern browsers (without guarantee) and at least in Chrome/Chromium (hard requirement). No browser specific vectors or vectors for outdated browsers are used.

All the vectors that can be seen in Table 4.3 represent base attack vectors. They can be obfuscated or changed to a certain degree through the following techniques:

**JavaScript:** For JavaScript code multiple different encoding techniques can be used. Discussing them all would be out of scope so only a few aspects will be covered here, details can be found in [29, 30, 133, 134]. Since it is possible to generate code from strings at runtime through various functions, it is easy to change the actual characters that are used in the payloads. Strings can be encoded as numbers or as sequences of special characters like in [135]. Required functions can be accessed through alternate indexes or keys in the object and prototype chains without specifying their actual name.

**HTML:** HTML parsers ignore some characters in specific locations of HTML elements and have a certain degree of fault tolerance. This allows the introduction of additional (useless) characters or leave some out, to break up the structure. In the formal specification the *space characters* are defined as (in ascii hex representation) `\x09 \x0A \x0C \x0D \x20` [3]. From first to last these are the tab character (ascii hex: `0x09`), the line feed (`0x0A`), the form feed (`0x0C`), the carriage return (`0x0D`) and the space (`0x20`). Considering the following HTML tag structure and its different sections, several possibilities for obfuscation exist:

`<(a)tagname(b)(c)attributename(d)=(e)(f)attributevalue(f)(g)>`

- **(a)** Between the tag opener meta character (<) and the tagname: Only more < characters are allowed.
- **(b)** Between the tagname and the *first* attribute: The space characters `\x09 \x0A \x0C \x0D \x20` and the slash (`/`). All of them are valid separators of the tagname and the first attribute. A valid example would be `<tag////attr=x>`.
- **(c)** Between attributes: The space characters `\x09 \x0A \x0C \x0D \x20`. A slash before the attribute name is also allowed, but only in combination with at least one of the space characters, not as a standalone separator.
- **(d)** Before the attribute name-value separator (=): The space characters `\x09 \x0A \x0C \x0D \x20`.
- **(e)** After the attribute name-value separator (=): The space characters `\x09 \x0A \x0C \x0D \x20`.
- **(f)** Before and after the attribute value delimiter: The space characters `\x09 \x0A \x0C \x0D \x20`.

[3] `https://www.w3.org/TR/html51/infrastructure.html#space-characters`

**(g)** Before the tag closing meta character (>): The space characters `\x09` `\x0A` `\x0C` `\x0D` `\x20` and the slash (`/`).

There are additionally differences between browsers. Those can be found here[4].

**URL protocol:** Also the protocol schema can be obfuscated. Assuming the following URL and its sections:

```
(a)java(b)script(c):alert(1)
```

The following characters are allowed in the respective section:

**(a)** At the beginning or before the protocol: `\x09` `\x0A` `\x0D` `\x20` The tab, newline, carriage return and space character.

**(b)** Between any characters of the protocol: `\x09` `\x0A` `\x0D` The tab, newline and carriage return character.

**(c)** Before the colon: `\x09` `\x0A` `\x0D` The tab, newline and carriage return character.

Again there are differences between browsers and even between different versions of the same browser [5].

**data URLs:** Data URLs help avoiding character restrictions because they allow base64 encoding which can be used to convert textual data containing special characters into mostly alphanumeric data. The format of data URLs is like this:

```
data:[<MIME-Type>][;charset=<Charset>][;base64],<Data>
```

For most of our use cases specifying the mime-type is not necessary. Using data URLs in contexts that import HTML will automatically treat the data as HTML even without the correct `text/html` mime-type. Also the charset is negligible. So most of the time the data URL can be reduced to a format that only contains the following special characters `:` `;` `,` `+` `/` `=` when encoded in base64.

## 4.3 Exploit Verification

After creating the payloads for every data flow they are tested in a sandbox. The actual requests are derived from the data flow and the payload. In the sandbox the requests are executed and the responding content interpreted by the browser engine. The JavaScript functionality of the browser engine in the sandbox is extended by a special non-standard function which is exposed in the global JavaScript context of the sandbox. This top level function can be called standalone like other global functions. It accepts a single

[4] `https://html5sec.org/#100`
[5] `https://html5sec.org/#101`

parameter, the id of the payload that was responsible to trigger it. Internally this id is transferred back to FOXSS where it can be processed. So if an XSS payload is successful it will trigger exactly this function. In this way it can be guaranteed that a potential vulnerability can in fact be exploited. The custom function can be replaced by pretty much any JavaScript code for real world exploits. When the payload can not be verified through a function invocation it will not be counted as XSS vulnerability, so no false positive results will be generated.

So why was exactly this mechanism for verifying the vulnerabilities and payloads chosen? This approach has the advantage that a vulnerability verified through it has the following guaranteed properties:

i The payload can execute a JavaScript function. This automatically means we can almost always choose any arbitrary code as a replacement because of the runtime code generation abilities available in JavaScript. To show that the vulnerability exists a human analyst can for example then just create a pop-up window or make a HTTP request in order to present the successful XSS attack.

ii Another property is that it will not conflict with any existing code or overwrite existing variables. The name can be assigned arbitrarily. If we would have chosen to hook or replace an already existing function we would always have some interference when the code of the web application also uses the same function. So additional filtering would be required in order to know if the function call originates from the payload or the web application.

iii The impact on the execution of the web application in the browser is minimal and the characters required in the payload can be kept small. If we would have decided to verify the exploit by issuing a HTTP request we would need to expose additional server side functionality to handle these requests. The time a HTTP request takes is many times higher than an internal function call. Also the code for the payload would be much bigger and would require more special characters that can be avoided otherwise. HTTP requests might also not be possible in every case or might be blocked.

## 4.4 Scanner Structure

Figure 4.1 shows a diagram of the generalized interacting components. In the user interface one can provide a URL as a starting point, together with one of three crawl options. The first one limits crawling only to the provided URL itself. The second one does the same but to a single depth level so that only the provided URL and all of the URLs found on it are analyzed. The last one provides a full recursive crawl. The domain of the specified URL will be set as the base host. All further URLs that are found will be checked if they belong to the same base host. If this is not the case they will not get analyzed. Furthermore the current processing steps of a scan can be monitored in the UI.

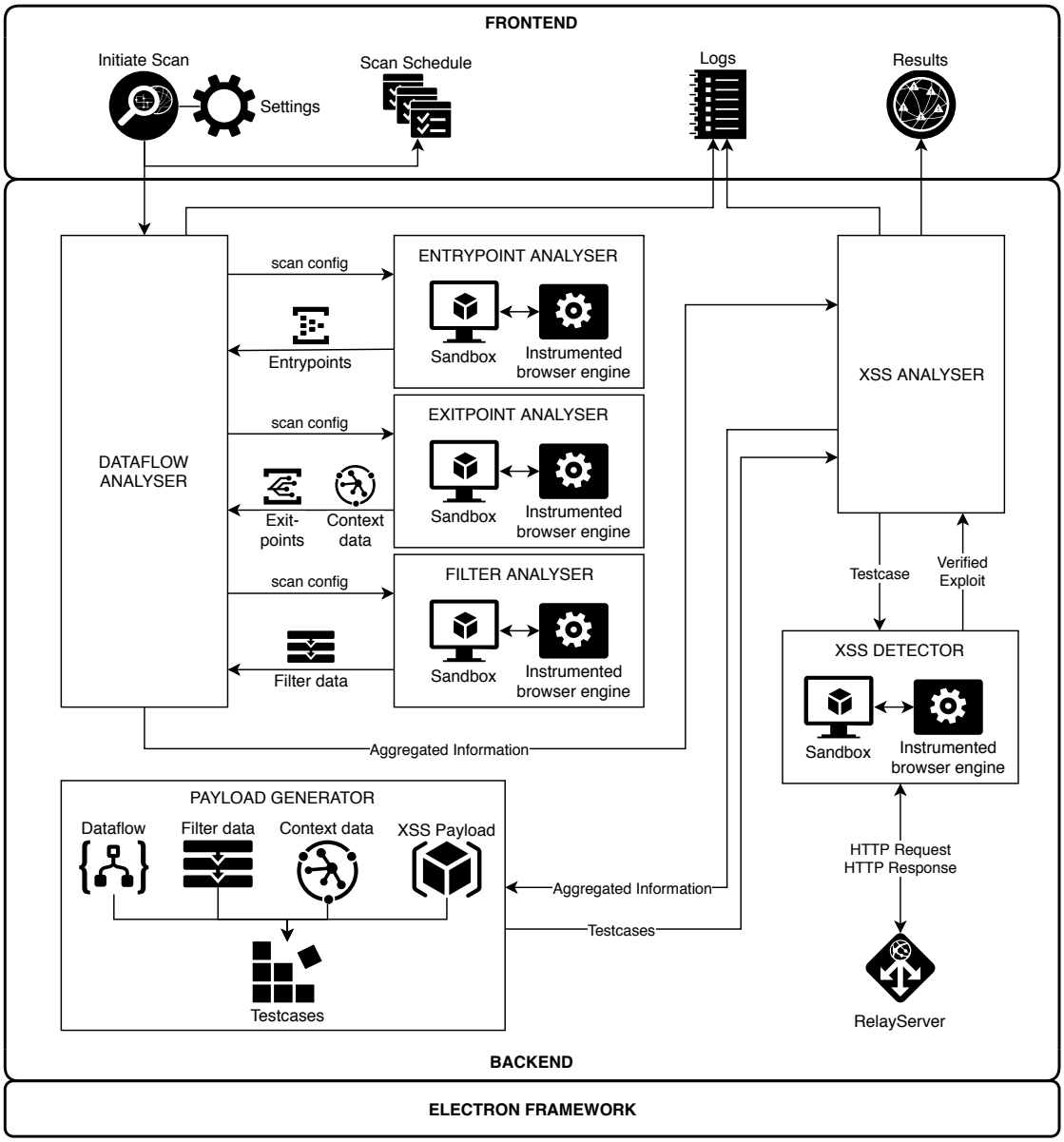


Figure 4.1: High level view of the scanner architecture.

Multiple scans can be scheduled and will be executed one after another. A log section records low level internal data. When data flows and later XSS vulnerabilities are found, they and their associated details can be viewed in separate categories.

In the backend the scan configuration will be passed to the data flow analyzer where internal data structures for further analysis components are created. Scan requests are then passed to sandboxes where they are processed and loaded as if it was a regular browser. Each sandbox loads its own analysis component and JavaScript context which might have been instrumented according to rules of the analysis component it contains. The data flow analyzer will begin executing the initial information provided by a user. The entry point analyzer will then possibly gather more URLs that are recursively analyzed for further entry points. Afterwards the exit point analyzer will probe all entry points succeeded by the filter analyzer that will test the detected data flows for filtering mechanisms. After all information from the entry point analyzer, exit point analyzer and filter analyzer has been aggregated and optimized the detected data flows are available for further processing. data flows will be also listed in the UI for examination. In the current prototype the XSS analyzer will access this data and will create XSS attack payloads for each data flow. The payload generator will utilize all the data flow details and provide as many payloads that might be successful as possible. In a separate sandbox the attacks will be executed and verified. A local HTTP server exposes an endpoint that serves payloads which can only be included remotely through URLs. So eventually the sandbox will request data from this server. Finally detected vulnerabilities together with their proof of concept exploits are reported in the UI.

| Data sink | Description |
|---|---|
| HTML sinks | |
| Element name | The name of a HTML tag: `<SINK attr="">` |
| Element content | The content of a HTML tag: `<tag>SINK</tag>` |
| Comment | Inside a HTML comment: `<!- SINK ->` |
| Attribute name | Name of an attribute of an element: `<tag SINK="">` |
| Attribute value | Value of an attribute of an element: `<tag attr="SINK">` |
| DOM script generation | |
| eval | Runtime code generation of the input. `eval(SINK)` |
| setTimeout | Runtime code generation of the input. `setTimeout(SINK, 123)` |
| setInterval | Runtime code generation of the input. `setInterval(SINK, 123)` |
| new Function | Runtime code generation of the input. `new Function(arg, SINK)` |
| DOM HTML generation | |
| document.write | Interprets the string as DOM elements and inserts them into the DOM. |
| document.writeln | Same as above. |
| Element.prototype.innerHTML = | Interpretation of the argument as DOM nodes and insertion as element content. |
| Element.prototype.outerHTML = | Interpretation of the argument as DOM nodes and insertion surrounding content of the element. |
| Element.prototype.insertAdjacentHTML | Interpretation of the argument as DOM nodes and insertion at a specific place relative to the element. |
| Range.prototype.createContextualFragment | Creation of DOM from a string. Needs to be inserted separately. |
| document.createElement('script').src = | Code generation and execution of the argument. |
| document.createElement('script').textContent = | Code generation and execution of the argument. |
| document.createElement('script').innerText = | Code generation and execution of the argument. |
| document.createElement('script').text = | Code generation and execution of the argument. |
| document.createElement('link').href = | Import HTML or execute script from URL argument. |
| document.createElement('iframe').src = | Import HTML or execute script from URL argument. |
| document.createElement('iframe').srcdoc = | Create HTML content. |
| document.createElement('object').data = | Import HTML or execute script from URL argument. |
| document.createElement('embed').src = | Import HTML or execute script from URL argument. |
| DOM location changers | |
| location = | Changes location to a specific URL. Can be used to execute attack vectors through `data` or `javascript` protocol or change to attacker controlled site. |
| location.href = | Same as above. |
| location.assign | Same as above. |
| location.replace | Same as above. |
| window.open | Same as above, but opens a new window instead of changing current. |
| JavaScript request | |
| XMLHttpRequest.prototype.open | Retrieve attacker controlled content or exfiltrate data. |
| XMLHttpRequest.prototype.send | Same as above. |
| window.fetch | Same as above. |
| DOM node insertion | |
| Node.prototype.appendChild | Indirect sink. Dangerous when the node/element itself or certain attributes or content of the inserted Node can be controlled. |
| Node.prototype.insertBefore | Same as above. |
| Node.prototype.replaceChild | Same as above. |
| Element.prototype.insertAdjacentElement | Same as above. |
| Element.prototype.append | Same as above. |
| Element.prototype.prepend | Same as above. |
| Element.prototype.before | Same as above. |
| Element.prototype.after | Same as above. |
| Element.prototype.replaceWith | Same as above. |

Table 4.2: Possible exit contexts and data sinks.

| Vector | Description |
|---|---|
| PAYLOAD | A raw JavaScript payload. |
| javascript:PAYLOAD | A raw JavaScript payload with the javascript protocol. This can be used in contexts where a URL is required. |
| <script>PAYLOAD</script> | A HTML script element. |
| <img src onerror=PAYLOAD> | Using the onerror-handler on elements that try to retrieve content from a specific location. The location (src or href attribute) is not set or set to some invalid value (x) which will generate an error and call the code specified in the onerror attribute. |
| <video src onerror=PAYLOAD> | Variation of previous. |
| <video><source onerror=PAYLOAD> | Variation of previous. |
| <audio src onerror=PAYLOAD> | Variation of previous. |
| <audio><source onerror=PAYLOAD> | Variation of previous. |
| <script src onerror=PAYLOAD></script> | Variation of previous. |
| <input type=image src onerror=PAYLOAD> | Variation of previous. |
| <link rel=stylesheet href=x onerror=PAYLOAD> | Variation of previous. |
| <link rel=prefetch href=x onerror=PAYLOAD> | Variation of previous. |
| <link rel=preload href=x onerror=PAYLOAD> | Variation of previous. |
| <iframe onload=PAYLOAD> | Uses the onload event handler to execute the code of this attributes' value. The code is immediately executed as soon as the element is loaded. |
| <svg onload=PAYLOAD> | Variation of previous. |
| <style onload=PAYLOAD> | Variation of previous. |
| <input autofocus onfocus=PAYLOAD> | The onfocus handler in combination with the autofocus attribute will directly execute the payload. This is available on some form elements. |
| <select autofocus onfocus=PAYLOAD> | Variation of previous. |
| <textarea autofocus onfocus=PAYLOAD> | Variation of previous. |
| <button autofocus onfocus=PAYLOAD> | Variation of previous. |
| <video src onloadstart=PAYLOAD> | Specific for audio and video elements there exists an onloadstart attribute which will execute code. A source or src attribute is necessary for this handler otherwise it will not get the loadstart event. |
| <video onloadstart=PAYLOAD><source> | Variation of previous. |
| <audio src onloadstart=PAYLOAD> | Variation of previous. |
| <audio onloadstart=PAYLOAD><source> | Variation of previous. |
| <iframe src=PAYLOAD> | Loads and embeds the HTML or JavaScript content from a URL provided to src. Can use JavaScript URLs. |
| <link rel=import href=PAYLOAD> | Can use URLs to import HTML or JavaScript and execute it. |
| <object data=PAYLOAD></object> | Variation of previous. |
| <embed src=PAYLOAD></embed> | Variation of previous. |
| <script src=PAYLOAD></script> | Variation of previous. |
| <iframe srcdoc=PAYLOAD> | The payload here is an inline HTML fragment or in otherwords another attack vector of this list. Special characters can be encoded as HTML entities. |
| <svg><script>PAYLOAD</script></svg> | A script inside an svg element supports special characters encoded as HTML entities. Directly executes the JavaScript payload. |
| <details open ontoggle=PAYLOAD> | The combination of open and ontoggle on the details element automatically executes the payload. |
| <body onload=PAYLOAD> | Onload on the body tag. |
| <body onpageshow=PAYLOAD> | The onpageshow eventhandler of the body tag works similar to onload and automatically executes the payload. |
| <body onscroll=PAYLOAD><h1><br>[...] | If the content of the body is long enough to overflow the window size, the onscroll handler can be used to directly execute code. Here h1 (to increase the line height) and br elements are used. |
| HOST+"/s/PAYLOAD/" | Represents an external URL that loads a piece of JavaScript code. |
| HOST+"/h/PAYLOAD/" | Represents an external URL that loads a complete HTML page. |
| HOST+"/f/PAYLOAD/" | Represents an external URL that loads a HTML fragment. |

Table 4.3: XSS attack vectors used in FOXSS. The keyword *PAYLOAD* will be replaced by JavaScript code.

CHAPTER 5

# Testing Environment

Creating a testing environment with numerous test cases for automated XSS vulnerability scanners was another main part of this thesis. To properly compare and evaluate scanners, a large number of different test cases that cover all possible input channels, exit contexts and also provide filtering mechanisms on the input data are required. To ensure that as many different types and aspects of XSS vulnerabilities as possible are covered in the test cases, I started with examining already existing testbeds that contain tests with XSS flaws. Most of the publicly available testbeds for web application vulnerabilities focus on providing few test cases for a broad range of vulnerability classes rather than focusing on diverse and specialized test cases for each class. Since the test amount and quality is often very low an evaluation of scanners in these testbeds would not yield good comparative results. Another problem with existing testbeds is their layout. Those that are designed to model real world web applications contain unnecessary barriers. User accounts, login zones and restricted areas need a lot more additional logic just to correctly crawl through the site structure. Overcoming the structural complexity of a web application is not a primary research focus of this thesis. It would require additional configuration and logic in the scanner to handle such applications in order to not terminate a test session prematurely for example because a “logout”-function is triggered accidentally. Furthermore some scanners do not even have such configuration options, including FOXSS. A suitable testbed should also have a separation of vulnerabilities by type such that it is possible to specifically test a certain category or a small amount of vulnerabilities at a time, which makes identifying problems in the scanner implementation easier. To sum it up an optimal testing environment should have the following properties:

- A maximum number of different test cases for all three categories of XSS organized in a tree like structure of the web application.
- Requires no complicated scanner setup like associating session cookies or login routines and areas that depend on session information.

- Each test case can be identified by its own URL. Additional pages like indirect placements of the XSS payloads should be only reachable through the subtree of a test case.
- A single URL should not contain more than one variant of a bug that can be exploited, in order to make the result assessment easier.
- No additional unnecessary styling, image or multimedia content that is not required for the execution of a test case. This would just increase the amount of data transfered each request and slow down the scanning process.

Three publicly accessible testbeds were chosen for a more detailed examination, since they stated in their description that they featured XSS test cases. Those were WebGoat [136], DVWA [137] and Google Firing Range [41, 76]. All were packaged as a docker image for portability. The four testbeds are discussed below.

**WebGoat** WebGoat version 7.1 is not dedicated to XSS vulnerabilities only but also showcases many other flaws that can be present in web apps. It is designed to act more as a learning tool and teach the concepts of the included vulnerabilities rather than to be evaluated by an automated scanner. Hints and solutions for each vulnerability as well as detailed descriptions are provided. In order to access the test cases of WebGoat a login has to be performed. The test cases contain: Three simple stored XSS attacks, two reflected and a few DOM-based tests. There are no interesting escaping or filtering mechanisms. There are simply not enough test cases that would justify running WebGoat as a separate testbed, so this testbed was dropped.

**DVWA** DVWA version 1.9 also features various web application vulnerabilities. Regarding XSS, all 3 types are supported with four different filtering mechanism examples each. Two of those are no-filter and secure-filter (security levels low and impossible respectively). Furthermore two filters exist that depending on the XSS category either remove ‘<script‘ tags, use case sensitive filters that can be bypassed by mixed case HTML tags, or are simply not applied to all input fields. The filter level that is applied is controlled by a cookie. So in an automated test one would need to change the cookie value whenever a different filter should be applied and to be able to test all those 12 test cases. Since this functionality is tedious for automated testing, it is much easier to recreate those few test cases.

**Firing Range** Version 0.47 which is the latest version published on Github was used. Firing Range was designed for automated analysis and contains a huge amount of different XSS vulnerabilities spreading over various data-sources and -sinks. The test cases are structured into different categories and each test case has a unique URL. Not all categories are relevant, for example “Flash Injection” is out of scope for our scanner comparison since it would require generating flash files and testing them which none of the scanners supports. Furthermore some individual tests that cannot be exploited in a meaningful

way, or only work in a specific browser because they exploit some non-standard features are available. Test cases of the reflected and the DOM-based XSS categories are included, however none of the stored type can be found.

After manually investigating the testbeds and their relevant vulnerabilities, I came to the conclusion that the quality as well as the quantity of XSS based test cases in all but Firing Range was very poor. In fact Firing Range was the only testbed that focused mainly on XSS and some related vulnerabilities and was actually designed for automated testing as described in [34].

This lead to the creation of a custom testbed which I called *XSS Playground*. This testbed was specifically engineered for the evaluation of automated tools and XSS vulnerability testing. It contains a large number of test cases ranging from basic vulnerabilities, over various filtering techniques to exotic test cases that can only be exploited with special context aware payloads. It is basically separated into three categories representing the three types of XSS. Each test case has a unique URL, one entry point and one exit point where the payload, depending on the filtering mechanism is placed either unmodified or transformed. Some test cases cannot be exploited due to secure filters and serve as a measurement for false positives that might be generated by scanners. The number of test cases that serve this purpose is rather low in contrast to the exploitable ones.

## 5.1 Structure

A high level view of the implementation of *XSS Playground* can be seen in Figure 5.1 on page 42. It is a web application written in JavaScript for `Node.js`. In its 3000+ lines of code it provides all kinds of combinations of input filters, entry and exit contexts. The main part (WebApp) loads the configuration (some of which can be configured at runtime) at startup and registers routers for the XSS categories. Each router is responsible to handle the HTTP requests targeting one of its test cases. They handle retrieving the payload from the accepted source, getting the HTML template, applying the respective filter on the input and embedding it in the template before the HTTP response is sent to the client. Contexts for DOM-based XSS are handled slightly different in the DOMRouter that is why it is not using the ContextHandler in the diagram. The StoredRouter saves the inputs in memory. When the application is restarted all stored data records will be cleared.

The final complete test environment consists of *selected test cases of Firing Range and XSS Playground*, since not all are relevant or fall in the scope. This is described in detail in Section 5.2 on page 43. The technology stack of the complete test environment can be seen in Figure 5.2 on page 43. As a base Docker containers are used to provide an isolated environment. The operating system running in each container is Alpine Linux[1], a very small and minimalistic Linux distribution. The base runtime and programming language for Firing Range is Java. Specifically version 8 is used in the container. XSS

[1] `https://alpinelinux.org/`

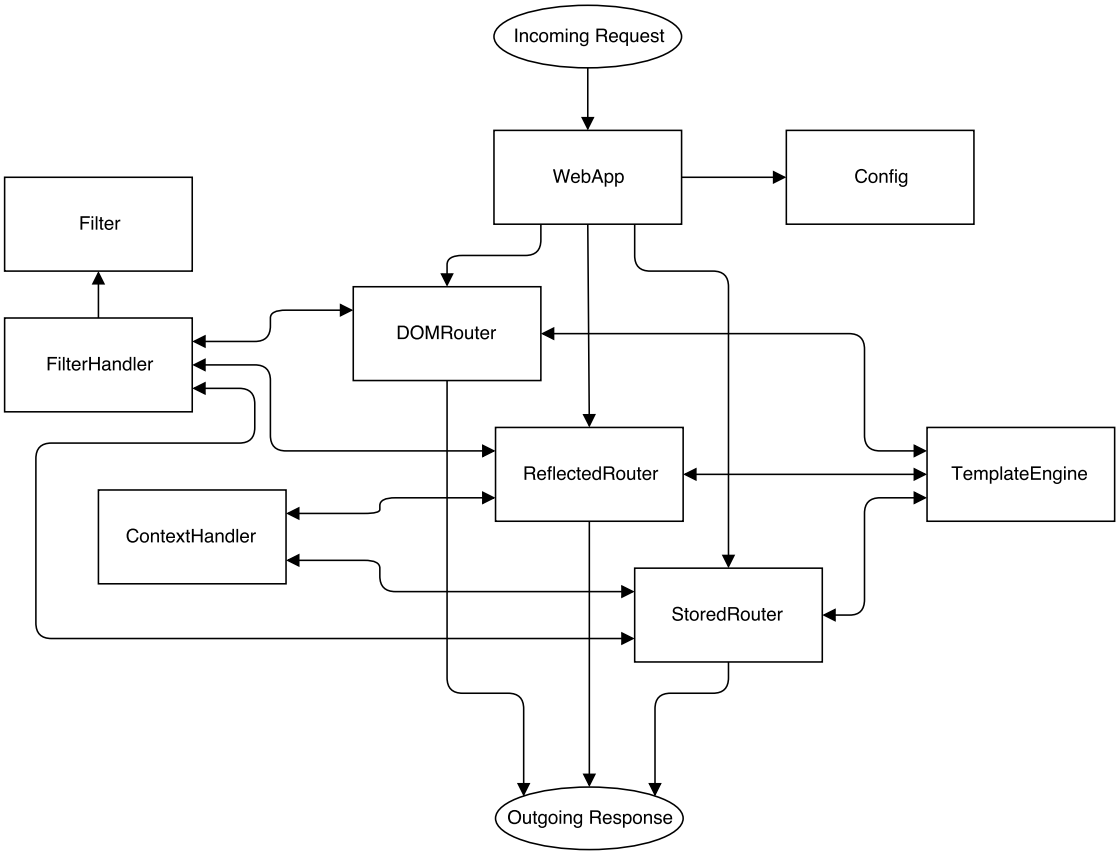


Figure 5.1: High level view of the XSS Playground implementation.

Playground uses NodeJS version 8.2.1. Firing Range uses the Google Appengine[2] as a framework and provider of a web server. Internally an embedded Jetty web-server comes into operation. In the XSS Playground the Express[3] framework is used under the hood to provide better HTTP request handling.

The overall test case and website structure for Firing Range and XSS Playground is visualized in Figure 5.3 and Figure 5.4 respectively.

Firing Range has a very flat structure. From the main index page each category of test cases can be reached, from which the actual individual test cases are linked. The unused categories are grayed out and dashed.

In XSS Playground everything is split up into the 3 main XSS categories in a first level. From there on various categories that indicate the origin of the entry source can be found. Regarding DOM-based XSS the links to the individual test cases can be accessed after the group was selected. Reflected and Stored XSS categories have an additional

[2] https://cloud.google.com/appengine/docs/
[3] http://expressjs.com/

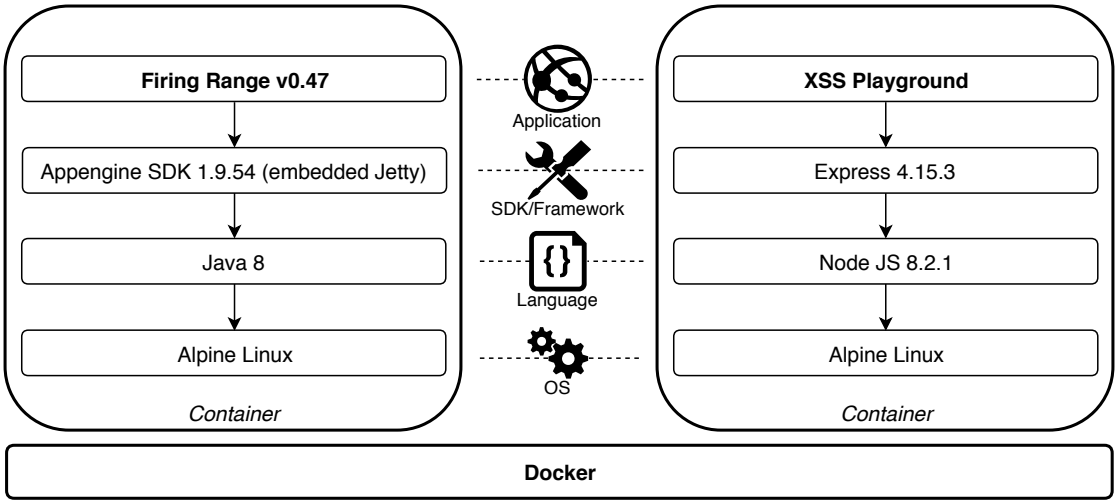


Figure 5.2: Composition of technologies used in the test environment.

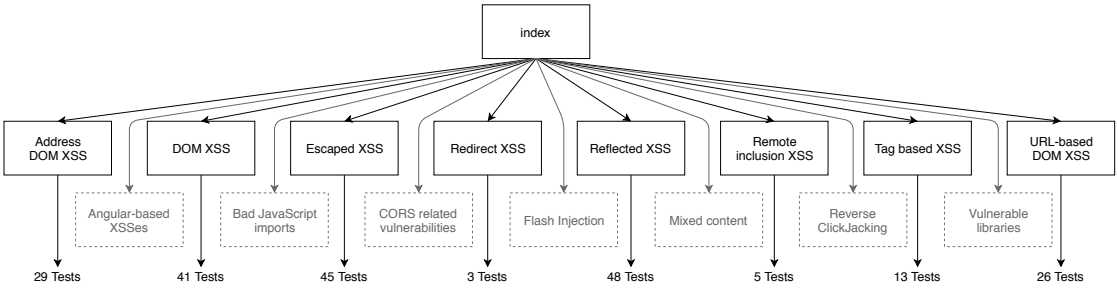


Figure 5.3: Firing range test case structure.

subcategory (view). The “all view” leads to a list of all test cases for a particular entry point. The “filter view” restricts the test cases to a specific filter. The “context view” provides all test cases for a certain exit context.

## 5.2 Test Cases

Since the exploitability of a XSS vulnerability always depends on the underlying browser that interprets the content of a web application, it is necessary to define how a test case is determined to be exploitable or not. Furthermore some XSS vulnerability might only be present in certain versions of a browser and might already be secured in current versions. As an example XSS attacks that were made available through features of a Cascading Style Sheet (CSS) are not possible any more in modern browsers [138].

Looking at the market share of browsers one can see that Google Chrome absolutely dominates the space. Two different sources show that nearly sixty percent of all website visits in November 2017 combining desktop and mobile browsers are made with Chrome:

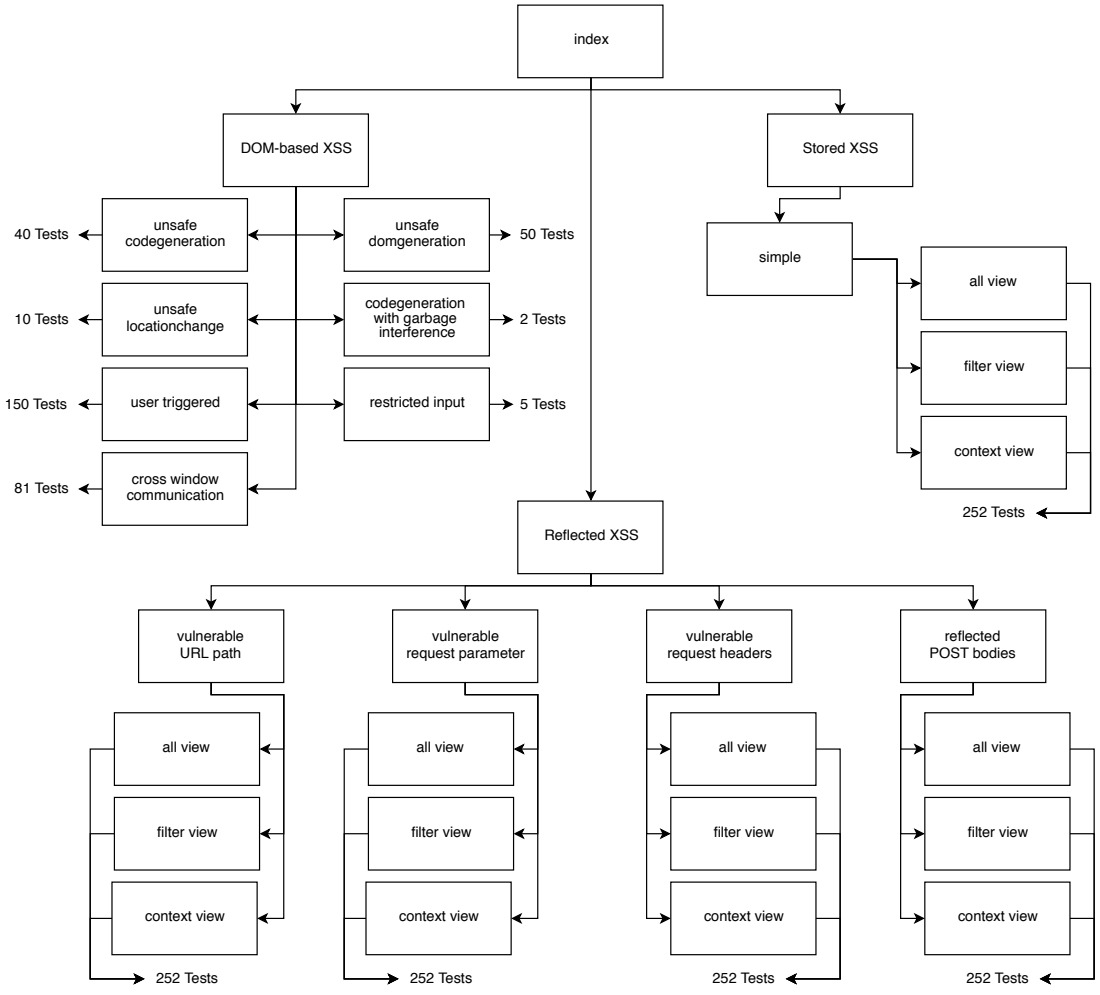


Figure 5.4: XSS Playground test case structure.

59.49% [139] and 59.2%[140] to be precise. All tests were manually verified with Chrome version 63, whether they could be exploited or not. If a test case was not exploitable, it is marked as such in the evaluation table and the reason why it cannot be is discussed in the following subsections for both testbeds. Furthermore some test cases are out of scope for the evaluation. Those consist of vulnerabilities that are specific to certain JavaScript libraries or when guessing has to be made by a scanner.

All eligible tools were tested against the test cases which are described in the following sections. The number of test cases for each testbed can be found in Figure 5.5.

### 5.2.1 Firing Range

In general the tests provided by Firing Range often only combine one entry point with a particular exit point, not exercising different combinations of the entry point and other

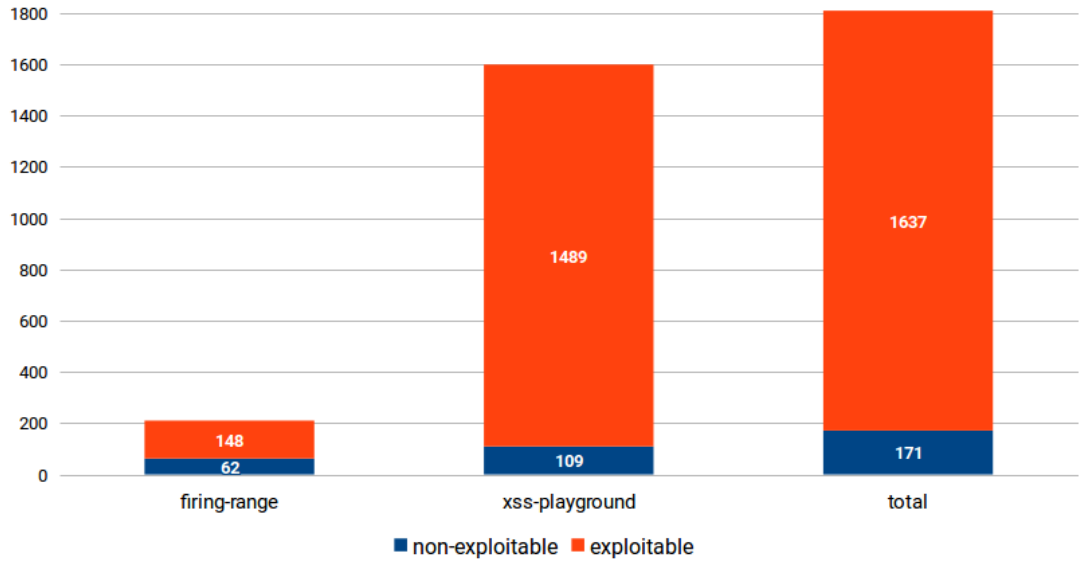


Figure 5.5: Number of test cases and category per testbed.

exit contexts and vice versa. It provides a total of 210 test cases for the evaluation.

The test cases (in total 41) of the following categories are not part of the evaluation because of being out of scope:

**Angular-based XSSes** 20 tests. Vulnerabilities are library specific.

**Bad JavaScript imports** 3 tests. Requires the detection of resources that are loaded from unsafe origins, that might be attacker controlled.

**CORS related vulnerabilities** 3 tests. No actual XSS vulnerability test cases.

**Flash Injection** 2 tests. JSONP in combination with Flash.

**Mixed content** 1 test. Not an XSS specific vulnerability or test case.

**Reverse ClickJacking** 11 tests. Mostly JSONP callback specific.

**Vulnerable libraries** 1 test. Vulnerabilities are library specific.

The categories and test cases that are included for the evaluation and possible restrictions thereof are:

**Address DOM XSS** Number of test cases: 29.

The test cases of this category belong to the DOM-based type and extract a possible payload from the URL. This is done through JavaScript which accesses the `location` object or one of its alternatives like `document.URL` or `document.documentURI`. The

payload data is then passed to JavaScript functions that either try to execute it, insert it into the DOM or set it as a new URL.

The following are marked as not exploitable:

- `/address/locationpathname/documentwrite` It uses the path as payload but does not allow user controlled parts of the path. Attempting so will result in a server error. This seems like a bug in the testbed.
- `/address/URLUnencoded/documentwrite` The test relies on a non-standard property `document.URLUnencoded` that can only be found in older versions of Internet Explorer.

**DOM XSS** Number of test cases: 41.

As the name suggests this category contains several different DOM-based vulnerabilities. The sources contain `document.cookie`, `document.referrer`, `window.name`, the postMessage API and event related tests. Payloads are either executed directly as JavaScript code or written into the DOM. Several tests utilizing the LocalStorage and SessionStorage APIs exist. However all of them try to read a predefined value ("badValue") from the web storage and use it as the payload. Since there is no way of setting this particular value provided by the test cases, without guessing the predefined key and manipulating the localStorage from outside, those tests are considered safe and thus marked as not exploitable:

- `/dom/toxicdom/localStorage/array/eval`
- `/dom/toxicdom/localStorage/function/eval`
- `/dom/toxicdom/localStorage/function/innerHtml`
- `/dom/toxicdom/localStorage/function/documentWrite`
- `/dom/toxicdom/localStorage/property/documentWrite`
- `/dom/toxicdom/external/localStorage/array/eval`
- `/dom/toxicdom/external/localStorage/function/eval`
- `/dom/toxicdom/external/localStorage/function/innerHtml`
- `/dom/toxicdom/external/localStorage/function/documentWrite`
- `/dom/toxicdom/external/localStorage/property/documentWrite`
- `/dom/toxicdom/sessionStorage/array/eval`
- `/dom/toxicdom/sessionStorage/function/eval`
- `/dom/toxicdom/sessionStorage/function/innerHtml`
- `/dom/toxicdom/sessionStorage/function/documentWrite`
- `/dom/toxicdom/sessionStorage/property/documentWrite`
- `/dom/toxicdom/external/sessionStorage/array/eval`
- `/dom/toxicdom/external/sessionStorage/function/eval`
- `/dom/toxicdom/external/sessionStorage/function/innerHtml`
- `/dom/toxicdom/external/sessionStorage/function/documentWrite`
- `/dom/toxicdom/external/sessionStorage/property/documentWrite`

**Escaped XSS** Number of test cases: 45.

These tests are of reflected type. The value of a query parameter "`q`" is used as the source of the payload. On the server, one of two filtering mechanisms is applied which either encodes HTML entities or applies URL-encoding on the input. The data is then

reflected into various contexts: into the DOM, CSS sections, JavaScript or URL contexts. Many of the tests are secure, because of the escaping of meta characters. Especially the URL encoding makes the tests safe in most cases:

- /escape/serverside/escapeHtml/body?q=a
- /escape/serverside/encodeUrl/body?q=a
- /escape/serverside/escapeHtml/head?q=a
- /escape/serverside/encodeUrl/head?q=a
- /escape/serverside/escapeHtml/body_comment?q=a
- /escape/serverside/encodeUrl/body_comment?q=a
- /escape/serverside/escapeHtml/textarea?q=a
- /escape/serverside/encodeUrl/textarea?q=a
- /escape/serverside/encodeUrl/tagname?q=a
- /escape/serverside/escapeHtml/attribute_unquoted?q=a
- /escape/serverside/encodeUrl/attribute_unquoted?q=a
- /escape/serverside/escapeHtml/attribute_singlequoted?q=a
- /escape/serverside/encodeUrl/attribute_singlequoted?q=a
- /escape/serverside/escapeHtml/attribute_quoted?q=a
- /escape/serverside/encodeUrl/attribute_quoted?q=a
- /escape/serverside/escapeHtml/attribute_name?q=a
- /escape/serverside/encodeUrl/attribute_name?q=a
- /escape/serverside/escapeHtml/css_style?q=a
- /escape/serverside/encodeUrl/css_style?q=a
- /escape/serverside/escapeHtml/css_style_value?q=a&escape=HTML_ESCAPE
- /escape/serverside/encodeUrl/css_style_value?q=a
- /escape/serverside/escapeHtml/css_style_font_value?q=a
- /escape/serverside/encodeUrl/css_style_font_value?q=a
- /escape/serverside/encodeUrl/js_assignment?q=a
- /escape/serverside/encodeUrl/js_eval?q=a
- /escape/serverside/encodeUrl/js_quoted_string?q=a
- /escape/serverside/encodeUrl/js_singlequoted_string?q=a
- /escape/serverside/encodeUrl/js_slashquoted_string?q=a
- /escape/serverside/encodeUrl/js_comment?q=a
- /escape/serverside/escapeHtml/attribute_script?q=a
- /escape/serverside/encodeUrl/attribute_script?q=a
- /escape/serverside/escapeHtml/href?q=a testbed bugged
- /escape/serverside/encodeUrl/href?q=a testbed bugged
- /escape/serverside/escapeHtml/css_import?q=a
- /escape/serverside/encodeUrl/css_import?q=a
- /escape/js/escape?q=a

**Redirect XSS** Number of test cases: 3.

These reflected XSS test cases will perform a redirect to the URL that is provided in the query parameter value. The first two will perform a server side redirect through HTTP status codes, the third will do this through a meta tag. Test number two does not accept URLs using the `javascript:`-protocol.

**Reflected XSS** Number of test cases: 48.

This is the main category for reflected XSS. The payload is extracted from the query parameter. Many different HTML contexts are featured where the payload gets embedded. Also a few tests with filtering mechanisms exist that enforce case sensitivity or only allow a certain type of quotes. There are a few browser specific test cases and one where guessing plays a major role, which are excluded:

- /reflected/parameter/json?q=a
- /reflected/contentsniffing/json?q=a
- /reflected/contentsniffing/plaintext?q=a
- /reflected/jsoncallback

**Remote inclusion XSS** Number of test cases: 5.

These reflected tests embed the payload in the `src` attribute of `script` and `data` attribute of `object` elements. Only URL-like payloads will get through.

**Tag based XSS** Number of test cases: 13.

In this category of reflected vulnerabilities the payload is filtered such that only certain HTML tags as well as specific tag-attribute combinations are allowed.

**URL-based DOM XSS** Number of test cases: 26.

This category of DOM-based XSS tests focus on URL payloads. Those are extracted from `location.hash` or `location.search`. Then they are injected to attributes of DOM elements that retrieve the resource or need additional user interaction to trigger the payload.

### 5.2.2 XSS Playground

This testbed is structured and separated into three different high level categories following the three types of XSS. Those three are then again divided into subcategories that depict the variant of the vulnerability. It provides a total of 1598 test cases.

#### DOM-based XSS

This category contains all DOM-based XSS test cases. Common DOM-sources for the first four subcategories are: `location`, `location.href`, `location.hash`, `location.search`, `location.pathname`, `document.documentURI`, `document.URL`, `document.baseURI`, `document.cookie`, `document.referrer`.

**unsafe codegeneration** Number of test cases: 40.

This group is all about creating and executing the payload as if it was valid JavaScript code. The four functions for code generation `eval`, `setTimeout`, `setInterval` and the function constructor (`new Function`) are combined with all sources. Strings that are passed to these functions will be interpreted by the JavaScript engine and executed.

**unsafe domgeneration** Number of test cases: 150.

The focus of this group is inserting the payload into the DOM. The payload is interpreted as HTML and converted to the corresponding DOM-nodes. 15 variants of how this can be achieved are permuted with the common DOM-sources.

**unsafe locationchange** Number of test cases: 50.

Contains different methods that change the current URL of the browser window as well as opening a new window with the payload data as new target.

**codegeneration with garbage interference** Number of test cases: 5.

This subcategory is similar to unsafe codegeneration since those tests try to interpret the payload as JavaScript code. The difference is that not all sources are used and that additional code is added to the raw payload. This might lead to syntax errors when it is evaluated, so the payload needs to adapt to the added "garbage" and try to either include it or somehow avoid its interference.

**user triggered** Number of test cases: 10.

These tests will not automatically execute the payload, but require additional user interaction like clicking on a link or submitting a form.

**restricted input** Number of test cases: 81.

Here test cases for multiple client side filters together with a mixture of DOM-generating, code generating and location changing sinks are provided. Four of them are not exploitable because of too restrictive filters:

- `/xss/dom/restrictedInput/source/location.hash/method/document.write/filter/stripUnsafeHTML/#insertPayload`
- `/xss/dom/restrictedInput/source/location.hash/method/createElementIframe/filter/stripUnsafeHTML/#insertPayload`
- `/xss/dom/restrictedInput/source/location.hash/method/document.write/filter/stripLtGt/#insertPayload`
- `/xss/dom/restrictedInput/source/location.hash/method/createElementIframe/filter/stripLtGt/#insertPayload`

**cross window communication** Number of test cases: 2.

These tests handle mechanisms that allow cross-window and cross-frame communication and data exchange, even with different origins, which can be potentially dangerous. More precisely the postMessage API and setting `window.name` are exercised in the test cases.

### Reflected XSS

There are four different subcategories for the reflected type. Each one stands for where the payload will be taken from or in other words where the entry point is. If the payload should be provided somewhere in the URL it is indicated by the placeholder token "insertPayload".

**vulnerable URL path** Number of test cases: 252.
The payload is taken from the last section of the path of the URL.

**vulnerable request parameter** Number of test cases: 252.
The payload should be provided as the value of the query parameter.

**vulnerable request headers** Number of test cases: 252.
A special HTTP header, indicated by the test case URL, is required which will then be embedded in the page on the server-side in the response.

**reflected POST bodies** Number of test cases: 252.
In this case a HTML form with a single input field can be found in the HTML of the page which needs to be submitted. The payload of the input field will then be reflected after the POST request was made.

For each of these categories 28 contextually different exit points are combined with each of 9 filters. These contexts are as follows:

**eventhandler** The payload will be inserted into the `onload` attribute of a `body` tag. The value is surrounded by double quotes.

**eventhandlerUserTriggered** Similar to the previous context, the payload will be inserted into a double quoted attribute of a body tag. Instead of `onload`, the `onclick` attribute is used. To execute the payload a user interaction is required or it needs to break out of the context.

**htmlComment** The payload will be located inside a HTML comment which means it is surrounded by `<!--` and `-->`. Escaping from this context is absolutely necessary.

**htmlElementContent** Standard HTML context in the HTML body without restrictions. Can be any element.

**htmlElementNeedsBreakout** The payload will be placed inside a `style` element in the body. This element needs to be closed in order to use an executable payload.

**script** The payload is embedded as the content of an inline `script` tag in the body.

**scriptAssignment** Similar to the previous one, but the payload data is additionally assigned to a JavaScript variable. So the form is of `var v = PAYLOAD;`.

**scriptAssignmentString** Variation of *scriptAssignment*. The payload is surrounded by double quotes and thus treated as a string (`var v = "PAYLOAD";`).

**scriptAssignmentTemplatestring** Another variant of the previous context. Instead of double quotes backticks are used to enclose the string.

**scriptCdata** The payload will be used inside a CDATA section of an inline `script` element.

**scriptComment** This variation of the *script* context puts the payload inside a JavaScript line comment. Format: `// PAYLOAD`.

**scriptCommentMultiline** This is the same as *scriptComment* but a multi-line JavaScript comment is used instead. Simply ending the line will not work here. Format: `/* PAYLOAD */`.

**scriptFunctionCall** The payload will be placed as argument of a JavaScript function call in an inline `script` element of the body. Format: `somefunction(PAYLOAD);`.

**scriptRegex** Here the payload is embedded in a regex that is assigned to a variable. Format: `var v = /PAYLOAD/g;`.

**specialFormAction** A button that is part of a `form` element and has a `formaction` attribute which will receive the payload data. This is interesting for URL payloads. It needs user interaction to trigger.

**specialHrefMath** Usage of the `math` environment. The payload will be set as the value of the `href` attribute of a `math` element.

**specialHrefMath2** Here the payload will be set as the value of the `xlink:href` attribute of `maction` element inside a `math` environment.

**specialLinkImport** The payload will be the value of the `href` attribute of a HTML import (`<link rel=import>`).

**specialMetaRedirect** The payload will be embedded as the URL part of a `meta`-tag redirect.

**specialObjectData** The `data` attribute of a `object` tag will be set to the payload data.

**specialSrcDocIframe** The payload will be the value of the `srcdoc` attribute of an `iframe`.

**specialSrcEmbed** The payload will be the value of the `src` attribute of an `embed` tag.

**specialSrcImg** The payload will be the value of the `src` attribute of an `img` tag.

**tagAttributeName** Here the name of an attribute of a `p` tag will be set to the payload data. The attribute name is followed by `="value"` and is the only attribute of this element.

**tagAttributeValueDoubleQuoted** The payload gets inserted as the value of a a `style` attribute of a `p` tag. The attribute value delimiters are double quotes.

**tagAttributeValueSingleQuoted** Same as the previous one, but single quotes are used instead of double quotes.

**tagAttributeValueUnQuoted** Again similar like the previous one, but no quotes are used around the attribute value.

**tagName** The payload is inserted as the name of a tag with text content. It is surrounded by angle brackets and occurs twice in the page, once as open and once as closing tag.

The available filters that can be paired with the test cases are:

**none** just returns the payload itself and does nothing.

**stripWhitespace** strips all kinds of whitespace characters, like applying the regex group `\s`.

**stripQuotes** removes double and single quotes. This prevents breaking out from attribute delimiters. The application of this filter makes some contexts secure against exploitation. Those are: *specialSrcImg, tagAttributeValueDoubleQuoted, tagAttributeValueSingleQuoted.*

**stripLtGt** removes angular brackets (<>). Injecting new HTML elements is not possible anymore. This makes the following contexts not exploitable: *htmlComment, htmlElementContent, htmlElementNeedsBreakout, specialSrcImg, tagAttributeName, tagAttributeValueDoubleQuoted, tagAttributeValueSingleQuoted, tagAttributeValueUnQuoted.*

**stripUnsafe** strips characters that might be dangerous in a HTML context (`&"'` <>). Makes most contexts secure against standard XSS injections however not against context aware ones. By applying this filter these contexts are secure and cannot

be exploited: *htmlComment, htmlElementContent, htmlElementNeedsBreakout, scriptAssignmentString, specialSrcDocIframe, specialSrcImg, tagAttributeName, tagAttributeValueDoubleQuoted, tagAttributeValueSingleQuoted, tagAttributeValue-UnQuoted*.

**escapeEventHandler** replaces all event handler attributes, i.e. all strings that start with on*. This will break many payloads.

**escapeCommonTag** escapes several (but not all) HTML tags that can be used for auto-executing payloads: `script img iframe svg link body audio video`.

**escapeUnsafeAttrib** escapes some attributes that are very likely to be dangerous when they can be injected into the DOM: `src data href srcdoc onload onerror`.

**escapeProtocolVectors** replaces the protocol of URLs starting with `data:` or `javascript:`.

#### Stored XSS

Number of test cases: 252. Only a simple standard group is included which just provides a HTML form with an input field and a submit button, that will perform a POST request once submitted. The payload will get stored and can be retrieved on a different URL that is linked from the page of the form. The same filter and exit context combinations that are available for reflected XSS can also be found here. The difference is that the payloads are stored until they are either overwritten by a different payload or until the testbed is restarted. Also the exit point of the payload contains one level of indirection. The triggering of the XSS payload is not located on the same page.

CHAPTER 6

# Evaluation

The performance of the prototype of FOXSS as well as the already existing individual XSS analysis tools were evaluated and compared in the specially tailored testing environment. It was hosted on a remote server located in Germany. Since several scanners incorporate a large number of HTTP requests in their analysis strategy a very low network latency significantly lowers the time required for analyzing the different test cases. If they supported multi-threaded execution they could profit of not having to share the CPU with the test environment. Furthermore in an initial assessment of the scanners it was discovered that a handful of tools could not perform scans of loopback addresses. Some of them stated that, others just behaved strange and did not produce proper results when a web application running on the localhost was analyzed. Using a remote server solved that issue. On the other hand there are tools that are limited to the local network only when the demo version is used like Milescan ParosPro. For those the test environment was hosted on the localhost.

The local machine contained an Intel Core i7-3630QM CPU @ 2.40GHz and 8GB of RAM. The remote machine an Intel Core i7-3770 CPU @ 3.40GHz and 16GB of RAM and was using Ubuntu 64bit 16.04 as operating system. The test environment was running inside Docker [141] containers, the Docker version used was `17.09.1-ce`. All scanners were executed on a Windows 10 64bit version 1709 operating system. The only exception was `w3af` which discouraged the usage of Windows on their website, so Ubuntu 64bit 16.04 was used for it.

## 6.1 XSS Vulnerability Scanners

Numerous XSS scanners that are publicly available at least to a certain degree were investigated. Publicly available means they can be found through search engines, security related websites and blogs (like [36, 63–68]), scientific papers [35, 69, 70, 73] or source

code hosting websites like Github[1] or Bitbucket[2]. A lot of time was spent to discover all relevant tools. A total of 67 third party scanners were examined this way.

While many tools that claim to be able to detect XSS can be "found", a deeper inspection in this evaluation revealed that many have very poor detection capabilities and show little robustness when evaluating a web application. A few acceptance criteria had to be met by each scanner in order to be considered for a deeper evaluation in the testing environment and to ensure the comparability of the results:

**Free:** The tool must be obtainable for free, at least for a limited time period, to perform the evaluation.

**No major limitations:** The program version that is used must not have any *major* limitations like not being able to perform scans of custom domains, IP addresses or network ports. Also there must not be any restriction on the number of scans that can be made. This is especially important for programs where a commercial and a free version exists. The free versions of those are often limited in its analysis capabilities.

**Original:** It must be obtainable through official sources provided or accepted by the original author. If there exist only mirrors or third party sources, that are not actively developing or improving the scanner it will not be considered.

**Not outdated:** Only applications that are still maintained. More specifically, it was last updated not further back than in 2013. In other words it was maintained in the last five years. This is necessary because of the rapid changes in the web application landscape and the ongoing additions to the HTML and JavaScript standards. A scanner that is not aware of modern web features will not be able to detect vulnerabilities that depend on those.

**Automated:** It must support automatic scanning of web applications, such that no additional user interaction is required. Possible input channels and parameters must be identified by the program itself. Ideally providing a URL for a XSS scan should be enough. No manual browsing on web pages (passive scanning) or explicitly specifying which parts of the URL, headers, etc. should be necessary.

For each scanner first the documented information was considered and checked against the acceptance criteria in order to sieve out the tools that lack important features. Then in an initial assessment the tool configuration was discovered. After that the settings were tested in an analysis run against the testbed. When problems appeared in the evaluation process or result generation, like program crashes, premature termination of the scan process or completely wrong results the configuration was refined and tested again until a correct analysis could be ensured. If it was not possible to get analysis results or to

[1] `https://github.com/`
[2] `https://bitbucket.org/`

run the tool at all it was discarded in this second stage. Finally the scanner was run against all test cases of the test environment twice and the results were aggregated and are presented in section 6.2.

### 6.1.1 Evaluated Scanners

| Open Source | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Supported OS | | | Detecting XSS Types | | | | | | |
| Name (Version) | W | L | M | DOM-based | Reflected | Stored | Automated Crawling | PoC Payload | GUI | CLI |
| IronWASP (0.9.8.6) | x | | | | x | | x | | x | |
| Vega (1.0) | x | x | x | | x | | x | | x | |
| Zed Attack Proxy (2.7.0 2017-11-28) | x | x | x | | x | x | x | x | x | |
| Arachni (v1.5.1-0.5.12) | x | x | x | x | x | x | x | | x | x |
| Wapiti (2.3.0) | x | x | x | | x | | x | x | | x |
| XSSer (1.7-1) | x | x | | | x | | x | x | | x |
| w3af (git e269868) | | x | x | | x | | x | | x | x |
| Nikto (2.1.6) | x | x | x | | x | | | x | | x |
| DSXS (git 7fd87d0) | x | x | x | (x) | x | | | | | x |
| **Free / Demo** | | | | | | | | | | |
| Syhunt Community Edition (6.0 RC1 (10.10.2017)) | x | | | | x | | x | x | x | |
| Tinfoil | | | | | x | | x | | x | |
| MileSCAN ParosPro (1.9.12) | x | | | | x | | x | | x | |

Table 6.1: XSS analysis tools that were successfully evaluated in the test environment. The “Supported OS” column describes the operating system that is supported. W stands for Windows, L for Linux and M for MacOS. Tinfoil has no OS because it is provided as SaaS. “Detecting XSS Types” is based on the actual results in the test environment, not on what the tools claim.

In Table 6.1 we can see an overview of all 12 scanners that satisfied the criteria and could be successfully tested. The table is divided into open source tools and free or demo-version tools that are proprietary but can be used without major limitations for at least a limited time period. The tool name and the specific version that was used can be seen in the first column, followed by the operating system the tool can run on and is not actively discouraged in its documentation. The next three columns show which types of XSS were detected in the test runs against the testbed. The following four columns show some meta information: Whether the tool can automatically crawl and discover sub-pages of a full web application when just a single URL is provided. If the tool creates and/or uses proof-of-concept payloads that can be reproduced by a user to verify a vulnerability. The last two indicate if the scanner provides a graphical user interface and/or a command line interface.

Some general remarks about the free versions of commercial scanners: The commercial

tools are all either Windows executables or provided as a SaaS solution. For those that are provided as download, all but MileSCAN ParosPro require you to provide some personal information and an email address. After those are provided and run through some validation either the download link is emailed or you are directly redirected on the website itself.

### 6.1.2 Excluded and Failed Scanners

The table 6.2 shows all 55 scanners that were examined and either failed the criteria or raised some problems during their execution in the evaluation. The first three columns show the name, the date they were checked the last time and a URL to their website. After that comes the reason why they were rejected or could not be evaluated. The problems that were encountered while trying to execute the analysis with the tool are described.

Table 6.2: XSS analysis tools that did not meet the acceptance criteria or failed during execution of the vulnerability detection.

| Name | Last Checked | URL | Reason |
|---|---|---|---|
| Skipfish | 11.12.2017 | `https://github.com/spinkham/skipfish` | Not original, only a mirror exists, outdated 2012. |
| ProxyStrike | 11.12.2017 | `http://www.edge-security.com/proxystrike.php` | Not original / down, outdated 2009. |
| Grabber | 11.12.2017 | `https://github.com/neuroo/grabber` | Author discourages usage, recommends w3af. |
| XSSS | 11.12.2017 | `https://www.sven.de/xsss/` | Outdated 2005. |
| Andiparos | 11.12.2017 | `http://code.google.com/p/andiparos/` | Not original / down, outdated 2010. |
| WSTool | 11.12.2017 | `https://sourceforge.net/projects/wstool/` | Outdated 2007, not automated. |
| Oedipus | 11.12.2017 | `http://rubyforge.org/projects/oedipus` | Not original / down, outdated 2006. |
| Paros Proxy | 11.12.2017 | `https://sourceforge.net/projects/paros/` | Outdated 2006. |
| PowerFuzzer | 11.12.2017 | `http://www.powerfuzzer.com/` | Outdated 2009. |
| XSSploit | 11.12.2017 | `http://www.scrt.ch/en/attack/downloads/xssploit` | Outdated 2008. |
| Gamja | 11.12.2017 | `https://sourceforge.net/projects/gamja/` | Outaded 2006. |
| ScreamingCSS | 11.12.2017 | `http://www.devitry.com/screamingCSS.html` | Outdated 2002. |
| crawlfish | 11.12.2017 | `https://code.google.com/archive/p/crawlfish/` | Not original / down, Outdated 2007. |
| Grendel Scan | 11.12.2017 | `https://sourceforge.net/projects/grendel/` | No download available, only libraries. It was moved to github, but there is no documentation. |

| Name | last checked | URL | Reason |
|---|---|---|---|
| jsky free edition | 11.12.2017 | http://www.sectoolmarket.com/web-application-scanners/15.html | Not original / down, outdated 2008. |
| safe3wvs | 11.12.2017 | https://sourceforge.net/projects/safe3wvs/ | Outdated 2011, source code not available anymore. |
| WebScarab | 11.12.2017 | https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project | Outdated 2011, superseeded by Zed Attack Proxy. |
| Uber Web Security Scanner | 11.12.2017 | http://www.sectoolmarket.com/web-application-scanners/17.html | Not original / down, outdated 2009. |
| SecuBat | 11.12.2017 | http://secubat.codeplex.com/SourceControl/list/changesets | Outdated 2010, source code not available anymore. |
| iScan | 11.12.2017 | http://www.sectoolmarket.com/web-application-scanners/49.html | Not original / down, outdated 2009. |
| openAcunetix | 11.12.2017 | http://www.sectoolmarket.com/web-application-scanners/41.html | Not original / down, outdated 2009. |
| VulnDetector | 11.12.2017 | https://github.com/BCable/vulndetector | Outdated 2006. |
| Xcobra | 11.12.2017 | http://www.sectoolmarket.com/web-application-scanners/18.html | Not original / down, outdated 2010. |
| Watcher | 11.12.2017 | http://websecuritytool.codeplex.com/ | Only passive analysis or manual testing possible. |
| Zero Day Scan | 11.12.2017 | http://www.zerodayscan.com/ | Not original / down, outdated |
| Trustwave App Scanner | 27.11.2017 | https://www.trustwave.com/Products/Application-Security/App-Scanner-Family/ | Not free. |
| Retina Web Security Scanner | 27.11.2017 | https://www.beyondtrust.com/products/retina-web-security-scanning/ | Not free. Powered by acunetix which is evaluated separately. |
| HP WebInspect | 27.11.2017 | http://www8.hp.com/us/en/software-solutions/webinspect-dynamic-analysis-dast/ | Not free, only scanning of their test application in the trial version. |
| IBM AppScan | 27.11.2017 | http://www-03.ibm.com/software/products/en/appscan-standard | Not free. |
| GamaScan | 27.11.2017 | http://www.gamasec.com/Gamascan.aspx | Not free. |
| x5s | 11.12.2017 | http://xss.codeplex.com/ | Outdated 2010, not automated. |
| ratproxy | 11.12.2017 | https://github.com/wallin/ratproxy | Outdated 2008. |
| Burp Web Vulnerability Scanner | 27.11.2017 | https://portswigger.net/burp/scanner.html | Not free. |

| Name | last checked | URL | Reason |
|---|---|---|---|
| Wikto | 11.12.2017 | https://github.com/sensepost/wikto | Described as "Nikto clone for windows" on the website. Nikto is useless as seen in tests and this wikto does not have any additional features for XSS. |
| XSS-Scanner.com | 11.12.2017 | http://xss-scanner.com/ | Not automated: Manual specification of entry points is required. |
| AppSpider | 27.11.2017 | https://www.rapid7.com/products/appspider/ | Not free. |
| edgescan | 23.11.2017 | https://www.edgescan.com/index.php#solutions | Not free. |
| IKare | 23.11.2017 | http://www.ikare-monitoring.com/ | Not free. |
| Websecurify Webreaver | 23.11.2017 | http://www.webreaver.com/ | Not free. |
| WebCruiser Free Edition | 11.12.2017 | http://www.janusec.com/ | Not original / down, outdated. |
| UpGuard | 23.11.2017 | https://app.upguard.com/webscan | Not free. |
| Wfuzz | 11.12.2017 | https://github.com/xmendez/wfuzz | Not automated: Manual provision of payloads and entry points. |
| Shuriken | 11.12.2017 | https://github.com/shogunlab/shuriken | Not automated: Manual specification of entry points. |
| Xenotix XSS Exploit Framework | 11.12.2017 | https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework | Not automated. |
| Golismero | 11.12.2017 | https://github.com/golismero/golismero | It just aggregates third party tools. The only one which does XSS detection is XSSer that is already tested separately. |
| XssPy | 12.12.2017 | https://github.com/faizann24/XssPy | Tried all different kinds of invocations, even with the extensive mode it does not find anything. Documentation on Github is wrong. Usage documentation on website only provides blank window. |
| V3n0M-Scanner | 12.12.2017 | https://github.com/v3n0m-Scanner/V3n0M-Scanner | Just random Internet scanning by getting targets from a search engine, not able to target a specific host. |
| Acunetix | 13.12.2017 | https://www.acunetix.com/vulnerability-scanner/download/ | Trial version does not display the URL or path where a vulnerability was found (only the host), thus not able to analyze the results in detail. |
| Netsparker (4.9.5.17070) | 13.12.2017 | https://www.netsparker.com/web-vulnerability-scanner/download/ | Trial version does not display the URL or path where a vulnerability was found (only the host), thus not able to analyze the results in detail. |
| Nstalker Free Edition | 14.12.2017 | https://www.nstalker.com/products/editions/free/download/ | Not being able to download since no email which should contain the download link gets sent. |
| Watobo (0.9.23) | 16.12.2017 | http://watobo.sourceforge.net/index.html | Not being able to run it: Fails with 3 different ruby versions (2.1, 2.3, 2.4) and 2 different devkit versions that were tried. |

| Name | last checked | URL | Reason |
|---|---|---|---|
| XSStrike (1.2, git 286b53d) | 16.12.2017 | `https://github.com/UltimateHackers/XSStrike` | Thinks urls are POST urls and exits. It was retried with the testbeds hosted on a remote server which had the same bug. |
| BC Detect | 18.12.2017 | `https://www.blueclosure.com/product/bc-detect` | No email with demo version got sent. |
| Scan My Server | 18.12.2017 | `https://www.scanmyserver.com/` | Cannot connect to server (maybe because of non-default HTTP port that was used). Trying to register redirects to the start page where a url must be provided, resulting in the same issue ("Oops! We cannot reach this host. Please try again."). |
| Detectify | 18.12.2017 | `https://detectify.com/` | When starting a scan it always says that it could not connect to the server, which is not true since the webapp is running and no firewalls are active. However the initial verification of website ownership was successful. |

## 6.2 Analysis Results

This section shows the results of all vulnerability scanners that passed our acceptance criteria and could be examined and evaluated. Furthermore the analysis results of the prototype implementation of the scanner are included. For each tool the number of detected vulnerabilities, false positive and false negative results were collected (Figure 6.1, Figure 6.2 and Figure 6.3). Table views for the exact absolute numbers of detections and relative detection ratios in percent can be found in the Appendix.

A scanner was required to explicitly categorize a test case as vulnerable. This means it was necessary for the tool to indicate that a XSS vulnerability was detected in a specific test case associated with a specific URL. It did not matter if they got some level of severity or any other meta information associated with the result. However it must explicitly state that it identified a vulnerability. Only those results were counted as a detection. It was not necessary that the program provided a working exploit but it must provide the URL to the test case of the XSS incident.

| Scan times | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Sylhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 10 | 30 | 60 | 6 | 10 | 1 | 16 | 3 | 1 | 1 | 10 | 29 | 2 |
| xss playground | 96 | 68 | 310 | 168 | 82 | 26 | 254 | 28 | 7 | 7 | 375 | 7 | 15 |

Table 6.3: Scan times of different scanners for each testbed. Measured in minutes and rounded up to full minutes.

In Table 6.3 the times each scanner took for a complete scan of a particular testbed are

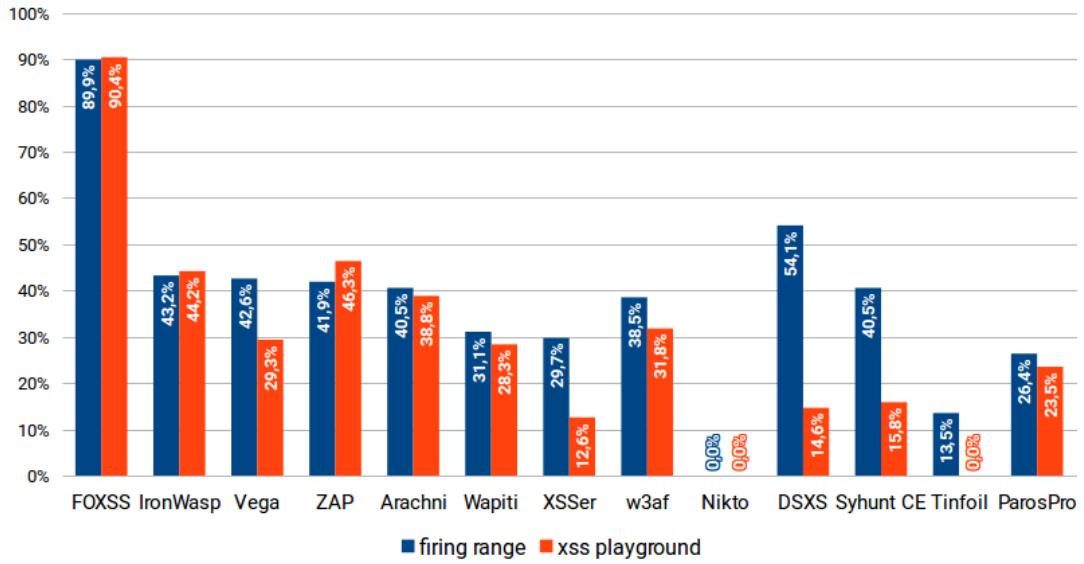


Figure 6.1: Detected XSS vulnerabilities (in %) of the test cases per testbed and scanner.

compared. The numbers are the average time of two independent test runs measured in minutes. Higher scan times do not correlate with higher detection ratios. Most of the scanners that took much more time also have bad results when compared to those who only took a fraction of their time.

### 6.2.1 FOXSS

The detection ratio of FOXSS significantly outperforms all other scanners. Most of the vulnerabilities that were undetected lie in the category of HTTP header entry points. Furthermore the two exit contexts that end in the HTML context of the math environment pose problems when combined with filters that do not allow to break out of it. This happens because the underlying Chrome browser engine currently does not support this environment. So exploits cannot be verified if they try to use this context in the payload, since the MathML related tags do not get interpreted. Also some filters can not be bypassed in the current prototype because some of the obfuscation methods are not implemented yet. Furthermore a few implementation bugs exist which prevent the automated verification of URL-style payloads that are injected into `object` or `embed` contexts. They do not get verified because of yet unknown reasons and thus do not appear as detections. However manual tests of the payloads suggested by FOXSS for these test cases were successful.

The verification of XSS payloads guarantees zero false positives which the analysis results have proven. Also the time required to analyze the test environment lies in the range of the top 3 tested third party scanners.

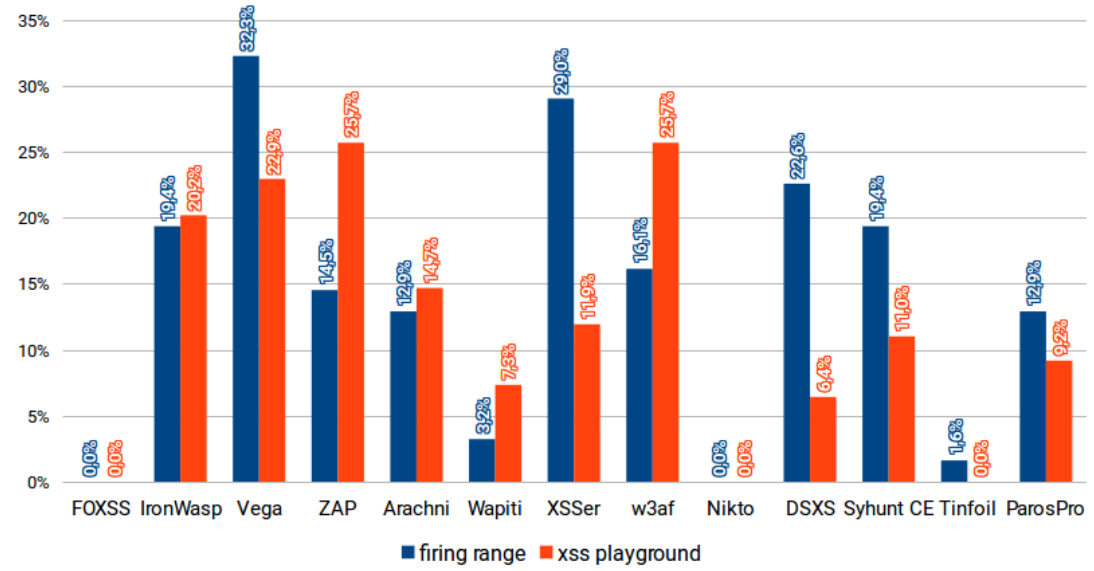


Figure 6.2: Percentage of false positive results each scanner detected per testbed.

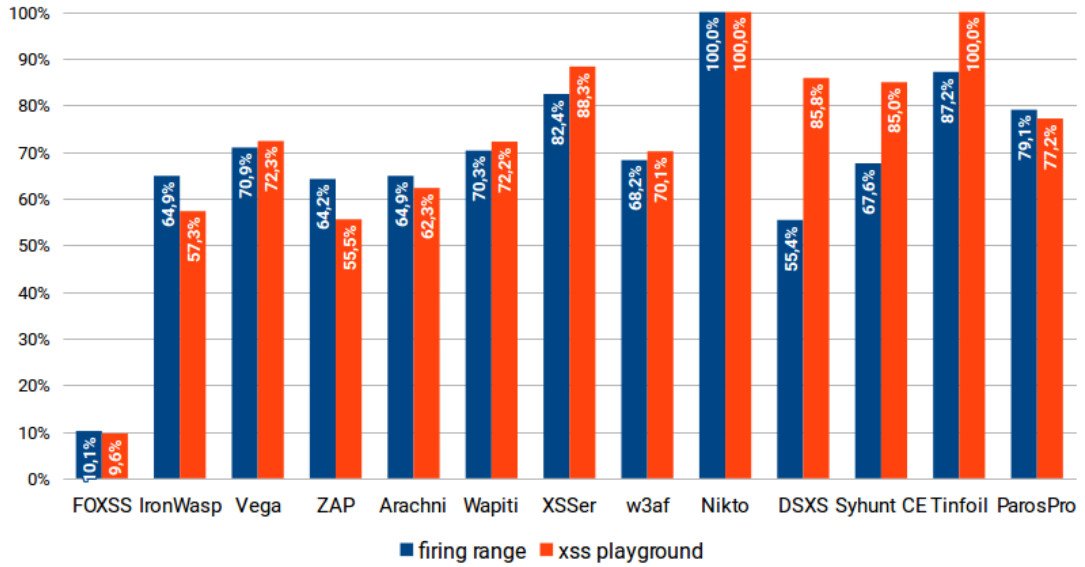


Figure 6.3: Percentage of missed vulnerabilities for each scanner and testbed.

### 6.2.2 IronWasp

IronWASP [142] is a multi-vulnerability scanner. It is open source and extensible with plug-ins written in several languages including Python and Ruby. Analysis of restricted areas is possible. Latest updates added JavaScript tracing and WebSocket fuzzing. Everything can be configured through a graphical user interface. The website provides several tutorial videos that show how to perform different kinds of vulnerability scans. Analysis reports can be exported in HTML and RTF format. The scanner was written in `C#` and designed for Windows. Third party solutions exist which allow this tool to run on Linux and MacOS.

On the first run this tool crashed because of some incompatibilities between the packaged "chromedriver" library and Windows 10. Downloading the most recent version[3] and replacing the existing one solved this problem. The browser based crawler that can be used optionally proved useless. Crawling Firing Range with it took already 30 minutes because IronWasp waits at least one second for every page and the browser based scanner sometimes hangs up and waits for 10 seconds until it gets restarted. So the default scan approach was used. While performing a test scan after 1312 "ScanJobs" IronWasp hung up. It is possible to save the current state and resume after a restart of the program, but not even a restart made it possible to continue. So the collected data was wiped and the scan was completely restarted which somehow solved the problem. Luckily this issue was not encountered again.

Another tool that is not integrated in the default scan is the DOM-XSS analyzer. Testing it on the Firing Range shows that it is done statically and just searches certain JavaScript functions in the code of a page. It produces results that can only assist a human analyst by showing which of those functions were found, but it does not find vulnerabilities on its own.

For the vulnerability scans the following configuration options were used: 4 threads, disable directory and file guessing since all test cases can be reached via URLs. The "scan settings" were adapted that all fields are scanned and only tested for XSS vulnerabilities. Under "scan filter" the skipping of the analysis of the `User-Agent` and `Referrer` headers was removed. In the last configuration step the "prompting for assistance" during the scan was disabled.

In the scanning process real payloads are only used sometimes like in css imports, otherwise just string identifiers are used to predict XSS heuristically.

Regarding the analysis results IronWASP could achieve good results in the reflected XSS category which was also the only category where vulnerabilities were found. It could find exploits for most contexts and most filters there.

[3] `https://chromedriver.storage.googleapis.com/index.html?path=2.33/`

### 6.2.3 Vega

Vega [143] detects XSS, shell injection, SQL injection and remote file inclusion among other bugs. Further the tool "probes for TLS / SSL security settings and identifies opportunities for improving the security of your TLS servers" [143]. After a scan the detected vulnerabilities can be seen in the GUI. No methods for exporting a report were found. Its analysis modules can be extended with JavaScript and supports automated and manual analysis. Passive security testing by running the browser through Vegas proxy is also possible. Vega was programmed in `Java` and uses the Eclipse framework. It can be run on all three major operating systems.

The crawler does not work when providing a url that ends in a file extension like ".html". Also a big disadvantage is that the results of a scan cannot be exported in any way, which makes an automated dissection of the results impossible.

The following scan settings were used: In the injection modules only "xss injection checks" were activated. In the response analysis modules the default settings were used.

Only XSS test cases of the reflected category were identified correctly. No vulnerabilities could be detected when the entry point was located in the URL path or in a HTTP header. The tool could also not handle several test cases with "special" contexts when in combination a filter was used. It had many false positive detections, the most in Firing Range and the third most in XSS Playground among the scanners.

### 6.2.4 ZAP (Zed Attack Proxy)

Zed Attack Proxy [144] is an OWASP project and as such open source, featuring fully automated and passive scanning, forced browsing and fuzzing. Several different types of web vulnerabilities can be detected. This application was also written in `Java`.

Among all tested tools this one has the most complicated user interface of all. Options and different tools are scattered everywhere and can be accessed through multiple menus. It takes a long time to find correct settings and the work flow to perform the intended scan.

When starting a new scan by providing localhost as the destination, it ends in the creation of tens of thousands of requests of non existent URLs that are tried to be scanned. None of those are meaningful and it seems like some brute force directory scanning based on a word list is performed. Trying to scan a subcategory results in response code 400 and that ZAP failed to attack it. Starting the "spider" to find entry points does not find anything useful, again it reports the 400 error that does not exist. Using the "new scan" option on the URLs found by the spider does not yield anything useful. The solution is hosting the testbeds on a remote server. This strange and faulty behavior only happened when trying to scan localhost.

Regarding the settings a new scan policy was created through the policy manager. All attacks other than XSS (located under injection) were disabled. The strength level was

set to maximum. The testbeds were scanned with the spider with the following settings: The base url was provided as a starting point. Under “Scope” the option to spider the subtree only was activated. Under “Advanced” the maximum crawl depth was set to 10. Then an “active scan” was performed where we activate the following: Under “scope” we selected the base url. Under “input vectors” enabled URL path, HTTP headers and cookies. All built-in input vector handlers but JSON were disabled because no tests where they are used exist. Under “policy” we selected the previously created XSS scan policy.

ZAP was one of the few scanners that could detect stored XSS. With 46,3% it achieved the highest detection ratio under third party scanners in the XSS Playground. It often had problems to find an exploit in the meta redirect context when combined with a filter. Also in the entry point category of HTTP headers no detections were made. Together with w3af this tool had the most false positive results in XSS Playground among the third party scanners.

### 6.2.5 Arachni

Arachni [145] is an open source web application penetration testing framework. A browser environment was integrated to improve JavaScript application analysis, especially dynamically generated data channels of websites. It seems like all variants of web vulnerabilities can be tested. The tool can be controlled through a web interface, where a pre-configured option for an XSS scan can be found. Alternatively the command line interface can be used to execute a scan. This scanner was programmed in `Ruby` and also uses PhantomJS for automated browser testing.

It is not documented where exactly the web interface (under which port) can be found. This can be found out when reading the console output after starting the tool. Also it cannot scan loopback interfaces like localhost. By default a profile for XSS tailored scans is provided. All scans were performed by selecting “New Scan” in the user interface and then inserting the base URL of the testbed.

Arachni was the only scanner other than FOXSS that was able to detect vulnerabilities in test cases of all three XSS categories. No other third party scanner was able to detect DOM-based XSS. However the detection rate in the stored and DOM-based categories is very low. Script contexts often posed a problem for vulnerability detection.

### 6.2.6 Wapiti

Wapiti [146] was created for black-box scanning and is open source. It extracts entry points of web applications and performs fuzzing of the detected input channels afterwards. According to the authors the following vulnerabilities can be discovered: “File disclosure (Local and remote include/require, fopen, readfile...), Database Injection (PHP/JSP/ASP SQL Injections and XPath Injections), XSS (Cross Site Scripting) injection (reflected and permanent), Command Execution detection (eval(), system(), passtru()...), CRLF Injection (HTTP Response Splitting, session fixation...), XXE (XML External Entity)

injection, Use of know potentially dangerous files (thanks to the Nikto database), Weak .htaccess configurations that can be bypassed, Presence of backup files giving sensitive information (source code disclosure), Shellshock (aka Bash bug)" [146]. Some authentication methods, basic examination of JavaScript and newer HTML5 objects are included as well. Attacks can be carried out through HTTP GET and POST. Analysis reports can be exported in multiple formats including HTML, XML, JSON and TXT. This scanner was written in `Python`.

Scan runs on localhost are much faster than on a remote host, this means a large part of its scan time is spent on HTTP-requests. Some intelligent payload finding was observed. For example when < is stripped by filters it is recognized and payloads that do not depend on this character are tested.

The configuration that was used can be seen in the following command line invocations of the tool for both testbeds. Basically unnecessary sub-categories were excluded from the scan, only XSS checks were performed and the results saved in a specific directory.

```
.\wapiti.exe "http://144.76.18.131:8080/" -x "http://144.76.18.131:8080/angular/index
    .html" -x "http://144.76.18.131:8080/badscriptimport/index.html" -x "http
    ://144.76.18.131:8080/cors/index.html" -x "http://144.76.18.131:8080/
    flashinjection/index.html" -x "http://144.76.18.131:8080/mixedcontent/index.html"
     -x "http://144.76.18.131:8080/reverseclickjacking/" -x "http
    ://144.76.18.131:8080/vulnerablelibraries/index.html" -m "-all,xss" -o ".\reports
    \firing-range"

.\wapiti.exe "http://144.76.18.131:8080/" -x "http://144.76.18.131:8080/entry points"
     -m "-all,xss" -o ".\reports\xss-playground"
```

Only XSS test cases belonging to the reflected category were identified correctly. It could not detect any test cases where the URL path or HTTP headers were used as entry point. The rate of test cases that were wrongly detected as positives is the lowest among the third party tools (excluding Nikto and Tinfoil since they barely detected any test case).

### 6.2.7 XSSer

XSSer [147] is a open source Python tool dedicated to detecting XSS vulnerabilities in websites. It can automatically bypass filters and certain web application firewalls and can create images or Flash files containing payloads. However those Flash based exploits can not be created automatically on the fly for automated testing. Proxy servers can be used and reports generated as XML or text.

This scanner was the most unstable of all. Multiple crashes and regular exceptions plagued the analysis. The tool often crashed during report generation with `-save -silent` parameters. So the two parameters were removed and the console output piped into a file. Still always an exception by the internally used `pycurl` gets thrown, and at the end after writing the results an error occurs that crashes the program:

```
Exception AttributeError: "Curl instance has no attribute '_closed'" in <bound method
     Curl.__del__ of <core.curlcontrol.Curl instance at 0x76C4B918>> ignored
```

Sometimes this:

```
Traceback (most recent call last):
  File "xsser", line 37, in <module>
    app.run()
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 1919, in
       run
    self.poll_workers()
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 1499, in
       poll_workers
    self.pool.poll()
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\threadpool.py", line
      348, in poll
    request.callback(request, result)
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 643, in
      _cb
    query_string, url, newhash)
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 774, in
      finish_attack_url_payload
    query_string, url, orig_hash)
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 1016, in
       _report_attack_success
    self.add_failure(dest_url, payload, hashing, query_string, attack_type)
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 1032, in
       add_failure
    self.hash_notfound.append((dest_url, payload['browser'], method, hashing))
MemoryError
```

Also often this happened:

```
Traceback (most recent call last):
  File "xsser", line 38, in <module>
    app.land(True)
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\main.py", line 1966, in
       land
    self.hub.shutdown()
  File "C:\Users\test\Downloads\xsser_1.7-1\xsser-public\core\tokenhub.py", line 66,
      in shutdown
    self.socket.shutdown(socket.SHUT_RDWR)
  File "C:\Python27\lib\socket.py", line 228, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 10057]
```

The program was configured to enable cookie and user-agent testing as well as using dom injections and automatic payload generation. It tests through brute-forcing/fuzzing with real payloads. The only plus was that in the report it explicitly states in which browser and version a vulnerability exists and is exploitable for each test case that was detected as vulnerable. The command line that was used for Firing Range and XSS Playground respectively:

```
python2 xsser -i "scanlist_firing-range.txt" --auto --Coo --Xsa --Xsr --Dom --no-head
     > firing-range.txt

python2 xsser -i "scanlist_xss-playground.txt" --auto --Coo --Xsa --Xsr --Dom --no-
    head > xss-playground.txt
```

Only XSS test cases of the reflected category were identified correctly. The detection of various contexts posed problems. The tool did not find HTTP header entry points and also reflected POST test cases were undetectable for it.

### 6.2.8 w3af

w3af [148] was also written in Python and was open sourced. Since 2006 it was continually improved and now features scanning and detection of several classes of web vulnerabilities. The following parts of a HTTP request are fuzzed: Query string, POST-data, Headers, Cookie values, multipart/form file content, URL filename and URL path. Reporting supports text, CSV, HTML and XML files. The dockerized CLI version was used for testing.

The following configuration and run settings were used for the analysis of each testbed:

```
plugins
audit xss
output html_file, text_file
grep analyze_cookies, dom_xss
crawl web_spider
back
profiles
save_as test
back
target
set target http://144.76.18.131:8080/
back
start
```

Only XSS test cases of the reflected category were identified correctly. None of the test cases that had the URL path or the HTTP header as an entry point could be identified as vulnerable. Together with ZAP this tool had the most false positive results in XSS Playground among the third party scanners.

### 6.2.9 Nikto

Nikto [149] is open source and was made for scanning web servers. The approach is based on a database containing known vulnerabilities, misconfigurations and common web application flaws. It brute-force tests the servers. Nikto was coded in `Perl`. It is not suitable for XSS scanning, which can clearly be observed in a test run. It basically just tries several static URL/path extensions and based on the web-server response it decides if there is a vulnerability. So when scanning "/xss/reflected/urlpath/filter/none/-context/eventhandler/insertPayload" for example it appends all its test URLs at the end. Since this test allows arbitrary characters after "insertPayload" the scanner gets confused and reports all kind of vulnerabilities it has in its database. In fact those are all false positives and the actual vulnerability is not triggered. In general Nikto tries to test URLs that are known to be vulnerable when a specific software is running on the server like some content management system. It was not able to identify a single vulnerability with this strategy, because it does not crawl or try to discover entry points. Instead it just performs the static URL tests that are stored in its database.

The commandline for each testbed looked like this:

```
perl nikto.pl -h "http://144.76.18.131:8080/" -ask auto -Format txt -nointeractive -↩
    nossl -output "results" -Tuning 4
```

### 6.2.10 DSXS

DSXS [150] for "Damn Small XSS Scanner" is a minimalistic open source XSS scanner written in Python. HTTP GET and POST parameters can be tested.

Since it cannot crawl and detect further URLs, a small wrapper script that executes the program for all test cases was used. The script could be invoked with `node script.js <pathToUrlFile>`. It will then call DSXS for each test case URL and will automatically put the result in the correct format for the evaluation table.

```
const { exec } = require('child_process');
const fs = require("fs");
const path = require('path');

////////////////////////////////////////////////////////////////////////////
// Config:
const toolName = "DSXS";
const cmdline = `python2 dsxs.py -u "{URL}"`; // {URL} gets replaced with current url
const successStr = "scan results: possible vulnerabilities found";
////////////////////////////////////////////////////////////////////////////

if (process.argv.length !== 3) {
    console.log(`Usage: node ${process.argv[1]} <urlFile>`);
    return;
}

console.log("start: "+ (new Date()).toTimeString());

const scanFile = process.argv[2].slice(process.argv[2].lastIndexOf(path.sep) + 1);
const urlList = fs.readFileSync(process.argv[2], "utf8");
const urls = urlList.split("\n").filter(x => x.length > 1);

var i = 0;
var results = "";
var resultsRaw = "";

if (!urls || urls.length === 0) {
    console.log("No urls in file!");
    return;
}

function testUrl() {
    if (i === urls.length) {
        fs.writeFileSync(toolName + scanFile +".rawout.txt", resultsRaw);
        fs.writeFileSync(toolName + scanFile +".results.txt", results);
        console.log("end:   "+ (new Date()).toTimeString());
        return;
    }

    const currentURL = urls[i++];
    exec(cmdline.replace("{URL}", currentURL), (error, stdout, stderr) => {
        if (error) {
            console.error(`exec error: ${error}`);
            results += "0\n";
            return;
        }

        resultsRaw += stdout +"\n";
        if (stdout.indexOf(successStr) !== -1) {
```

```
50              results += "1\n";
51          }
52          else {
53              results += "0\n";
54          }
55
56          testUrl();
57      });
58  }
59
60  testUrl();
```

Only XSS test cases belonging to the reflected category were identified correctly, with the exception of a single test case of the DOM-based category that was detected. The entry points of URL path, HTTP header or reflected POST requests were not discoverable for DSXS. Surprisingly this tool performed best of all third party scanners in Firing Range, but very bad in XSS Playground.

### 6.2.11 Syhunt Community Edition

Syhunt Community Edition [151] is the free version of the commercial Syhunt scanner suite. The features are limited, but sufficient for the XSS testing purposes. Unfortunately the filter evasion components are not included. White-box code analysis and black-box dynamic testing can be executed.

After an initial scan it was clear that we need to enable a depth limit or the scanner will end up in an endless loop on some tests like those about iframe attributes in Firing Range. So in the preferences under the crawling tab a depth limit of 10 was set.

Further the following settings were adapted:

```
Dynamic Scan ->
Select XSS scan
check Edit site preferences
edit exclusions > URLs:
exclude urls (firing-range):
http://localhost:8080/vulnerablelibraries/index.html
http://localhost:8080/reverseclickjacking/
http://localhost:8080/mixedcontent/index.html
http://localhost:8080/flashinjection/index.html
http://localhost:8080/cors/index.html
http://localhost:8080/badscriptimport/index.html
http://localhost:8080/angular/index.html
exclude urls (xss-playground):
http://localhost:8080/entry points
```

Trying to test XSS Playground with the same settings revealed a very annoying bug: The depth limit could not be set again. The limit was always instantly overwritten. So in order to test XSS Playground, the sub-URLs of the test case categories were used as a starting point. Through this method the default depth limit of 2 (although it says 1) could be used to avoid the endless loops of the crawler.

The only test cases that this tool was able to detect were reflected vulnerabilities when the entry point was located in a query parameters.

### 6.2.12 Tinfoil

Tinfoil [152] is a SaaS security scanner. After registering one can add websites and needs to prove ownership, before a free vulnerability scan can be executed. There are not much configuration options. After the scan is started one just has to wait until the results arrive.

Since this scanner is provided as a web-service one needed to verify the tested site beforehand, so they can be sure that the server is owned by the one requesting the scan. This can be done by placing a prebuilt file (txt with hash) at the root path of the website. No settings about which types of vulnerabilities are scanned can be set.

Scanning XSS Playground was tried 3 different times but always with the same disappointing result, that it ended after a few minutes, not detecting any of the test cases. The port on which the testbed could be reached was varied but this did not lead to any change.

The results are very poor. Not only that one of the testbeds could not be scanned at all, even for the one that could be analyzed the number of detected vulnerabilities is extremely low.

### 6.2.13 MileSCAN ParosPro

MileSCAN ParosPro [153] is a web vulnerability scanner for XSS and SQL injection. Also Content Management System (CMS) fingerprinting can be executed and subsequently misconfigurations can be detected. A demo version can be obtained, which is limited to private networks but has no other restrictions. The full scanning capabilities can thus be used. However all testing had to be performed on the localhost.

Some of the settings had to be changed:

```
Modify Settings > Global Settings
Under Vulnerbility Scanner > Vulnerability Checks tick Cross-site Scripting.
Under Spider set max number of concurrent threads to 4 and increase max links to be ⤸
    crawled to 5000. Max depth to crawl to 9, max concurrent spider threads to 4 and ⤸
    check "crawl domain on specified port only".

Then a new project was created for Firing Range and for XSS Playground.
Under the project name > functions > URL Spider the base url (http://localhost:8080/)⤸
     was added and the scan started with "ok".
Under functions > vulnerability scanner make sure that the cross site scripting ⤸
    category is checked.
```

Like some other scanners only XSS test cases of the reflected category were identified correctly. None of the test cases where the entry point could be found in the URL path or the HTTP header could be identified as vulnerable.

## 6.3 Discussion of Results

This section examines the results of the tested scanning tools and compares which types of XSS vulnerabilities they could handle and which stayed undetected. We will also discuss possible reasons for the weaknesses in their detection capabilities and contrast them with our approach.

The detection ratios that all evaluated scanners achieved in each testbed are shown in Figure 6.1 . We can clearly see that the number of detected vulnerabilities of third party tools is not quite high. Only a single tool managed to detect more than half of all vulnerabilities in Firing Range. None of them was able to reach such a rate in XSS Playground. FOXSS however achieved about ninety percent correct detections in both testbeds underlining the success of our strategy.

When comparing and ranking the scanners purely by their numbers we can see the following: The overall best third party scanner Zed Attack Proxy (ZAP) did not even detect half of all vulnerable test cases. It is followed by IronWasp with a sightly lower detection ratio and then by Arachni. When looking at Firing Range alone, interestingly DSXS, the ultra small (by lines of code) XSS scanner achieved the highest result with 54,1%. The following tools all achieved very similar results: IronWasp (43,2%) takes the second place and Vega (42,6%) the third. Then ZAP (41,9%) precedes Arachni and Syhunt Community Edition (both 40,5%). Regarding XSS Playground Zed Attack Proxy (46,3%) ranked first among existing black-box scanners, followed by IronWasp (44,2%) and Arachni (38,8%). The rest has a much higher offset with w3af being the next (31,8%). With all those sub 50% detection ratios it can be said that existing XSS analysis tools lack a lot of detection capabilities.

After examining the individual results for each test case and its detection status among the scanners we can see the reason behind this. Only Zed Attack Proxy and Arachni were able to detect stored XSS vulnerabilities. However they also were not able to make more than a handful of detections in this category. The results for DOM-based XSS test cases are similar. Only Arachni was able to find at least some vulnerabilities of this type. Surprisingly also DSXS detected a single vulnerable test case in this category. Most certainly this was just a lucky guess, since several similar test cases like the one it identified (`/urldom/location/search/location.assign?//example.org`) which belongs to Firing Range are also available in XSS Playground. However these slight variations of the test case were not detected. Vulnerabilities of all three different XSS categories could only be detected by a single scanner (Arachni). There is a lot of room for improvement regarding DOM-based and persistent XSS detection, especially since so many scanners do not know how to handle these vulnerabilities at all. Five scanners also had problems with test cases of the reflected type when the entry point of the vulnerability appeared in the body of a POST request. It seems that they are not able to handle POST requests at all and might even be limited to GET requests only.

In addition all of the third party scanners have problems when it comes to certain contexts in combination with a certain filtering mechanism. These test cases would either require

a special HTML tag or property to trigger the malicious code, or a special format of the payload to be successful, depending on the context. The inability of the scanners to detect many of those test cases is caused by the lack of diverse attack payloads and/or the lack of being able to adapt and refine the payloads with regard to the context. Many just try a fixed set of hard coded attack payloads and variations of them, but do not craft them at runtime. In general it can be said there are a few free tools that are acceptable at detecting reflected XSS, but none exists that reaches a similar quality in DOM-based and stored XSS.

In contrast FOXSS nearly detected twice as many vulnerabilities as the best third party scanner Zed Attack Proxy. This shows that the approach of data flow detection, context information gathering and detection of filtering mechanisms combined with specially crafted attack payloads and dynamic verification is a good strategy. FOXSS efficiently detects XSS vulnerabilities of each category not just reflected ones. With a overall detection rate of about 90% the approach significantly outperforms all other third party scanners already at this early stage when there is still a lot of room for improvement of the scanner. Several factors that impacted its results negatively are due to some implementation bugs that exist in the current prototype. Some limitations imposed by the underlying browser engine like the support of certain HTML5 features would be necessary in order to automatically verify the proposed exploits. Currently they are only reported as potential vulnerability and were not counted as real detections in the evaluation.

When examining the diagram of the false positive results (Figure 6.2) we can see that none of the other tools that scored at least one detection were also false positive free. In relation to the detection rate Wapiti had the lowest number of false positive results among third party scanners. Only FOXSS could achieve zero wrongly classified test cases, because it verifies each exploit in a real browser engine dynamically by actually executing it.

CHAPTER 7

# Conclusion and Future Work

In this thesis I presented a new approach for efficiently detecting all types of XSS vulnerabilities in web applications together with its prototype implementation called FOXSS. While not all ideas are fully implemented yet, the evaluation shows that it already outperforms similar existing open source vulnerability scanners in terms of detection capabilities while not requiring significantly more time to analyze web applications than existing solutions. The method of identifying data flows and gathering contextual information about a data source and its corresponding sink allows the creation of payloads that are specially adapted to its specific context. This minimizes brute force testing of a large number of unsuccessful attack vectors and thus does not waste valuable time which can be spent in other parts of the analysis process. Further it reduces the workload for the system under test which might be important when we are bound to certain resource constraints. Verifying every detected vulnerability by executing the attack payload in a real browser engine allows to guarantee zero false positive results.

An extensive number of data sinks and sources in web applications and how they can be mapped and exploited was discussed. Characteristics of interesting XSS vectors were analyzed and possible obfuscation methods for creating variations thereof that might be able to bypass input sanitization mechanisms were presented.

Furthermore I engineered and presented a testing environment featuring 1808 different test cases in which automated vulnerability scanners can be evaluated. It is separated into two parts, one is the already existing testbed Firing Range that contributes 210 test cases, the other one is the newly created testbed XSS Playground that contributes 1598 test cases. XSS Playground covers a large variety of data source and sink combinations which can be combined with filtering mechanisms. test cases of all three XSS categories are included. Each test case can be identified by its own URL and contains a single variant of a XSS vulnerability. No complicated scanner setup is necessary and no additional unnecessary styling, image or multimedia content is included. This testbed provides an easy way to compare and examine new black-box scanning mechanisms.

Numerous open source and free XSS scanning tools were examined and tested in the testing environment. The results show that existing scanners have multiple deficiencies. Only a single tool was able to detect vulnerabilities in all three XSS categories. Two tools each were able to detect vulnerabilities of the stored XSS and the DOM-based categories. Although only very few vulnerabilities were detected by them. Furthermore also many third party scanners lack the detection of reflected XSS when the entry point of the vulnerability is located in the body of a HTTP POST request. The results also confirm that the new approach with FOXSS is significantly better. While the best other scanners did not even identify half of all vulnerabilities, FOXSS was able to find about 90 percent, while also staying in the time range that the better third party tools required for the whole analysis process.

Future work can expand on several aspects: First of all the prototype could be improved in various areas. Currently the processing of analysis requests in a sandbox for the various stages of data flow detection and exploit verification is completely sequential. The sub-steps of identifying entry points, exit points, filters etc. will only happen one after another. Every previous step is finished completely before the next step is started. The sandboxed analysis can be easily parallelized by creating multiple sandboxes and executing several scan requests from the same step (e.g. entry point analysis) at the same time. When entry points are found they could also be instantly processed in the exit point analyzing sandbox while the entry point analysis is not finished and continues processing. This could further greatly improve the scan times.

Since real world web applications very often feature authorization or session management and restricted areas or content that changes depending on a user login, performing tests of similar applications would be interesting. Handling this was not a focus of this thesis and for that reason is not implemented, as it would have caused a lot of additional programming effort. These mechanisms would pose a problem for the analysis tool in its current state. It would however be also interesting to test and evaluate the scanner and also the third party scanners in test applications that model real world web applications instead of focusing on test case diversity.

In the area of attack vectors some possibilities are not examined. These are CSP-bypassing and file-based vectors. CSP is a strong defensive option against XSS attacks, but as already discussed in the related work section, the many configuration options increase potential misconfigurations that can be exploited. Testing web applications that deal with files might reveal interesting vulnerabilities because files like images or videos have several properties that could carry XSS payloads.

The data flow detection approach provides a solid base for web application analysis in general. The adaption of other vulnerability detection mechanisms would be easy because the design of the prototype allows adding or replacing scanner modules in the same way the XSS analysis component is written. Only the detection, analysis and payload generation features that are specific to a certain type of web application vulnerability need to be implemented.

A comparison and evaluation of FOXSS and commercial scanners would also be very interesting. The demo versions of Acunetix and Netsparker that did not show which test case was detected as vulnerable, were able to find multiple DOM-based vulnerabilities in their test run, according to the reported numbers. Since there exists a commercially driven interest in having a high detection ratio, those scanners might perform much better than their open source competitors.

# List of Figures



# List of Tables

# Glossary

**clickjacking** Clickjacking, also known as user interface (UI) redressing, is a method to hide an attacker controlled action that will be triggered when a user performs a seemingly legitimate action provided by a website. An example would be when an attacker attaches an additional HTTP request to a button that sends a form, such that the data will not only be sent to the original website but also to an attacker controlled endpoint. . 1

**concolic testing** Concolic testing is a combination of symbolic execution and concrete execution with specific input data. . 7

**FOXSS** FOXSS (Finding Only XSS) is the name of the prototype implementation of the XSS analysis approach presented in this paper. . 2, 3, 17, 25, 28, 33, 37, 39, 53, 60, 64, 71–75, 77

**fuzzing** Is a testing technique in which many different kinds of invalid or unintended input is sent to the communication interfaces of an application. The processing and results of the testing input can be observed and possible flaws and vulnerabilities can be uncovered. . 2, 9, 11

**fuzzy logic** In fuzzy logic certain aspects are not only exactly true or false, but are rather measured in degrees of truth. Truth values can be any real numbers from 0 to 1. . 8

**obfuscation** Is the process of disguising the actual functionality of some program code, in order to make it hard for a human analyst to understand the actual behavior. . 1

**SaaS** SaaS, short for software as a service, is the concept of providing a software solution and its functionality over the network instead of distributing the software. The provider is responsible for the infrastructure, administration and maintenance, while the customer can concentrate on using the service. . 55, 70

**SPA** SPA stands for single page application. It consists of just one HTML document. Displaying of different sections of the content or interacting with servers is done dynamically with JavaScript. . 29

# Acronyms

**CMS** Content Management System. 70

**CSP** Content Security Policy. 11, 13, 14, 74

**CSS** Cascading Style Sheet. 43, 47

**DDoS** Distributed Denial of Service. 19

**DOM** Document Object Model. 2, 4, 7, 10, 13, 14, 19, 21, 22, 24–26, 28, 29, 36, 40–42, 45–49, 52, 55, 62, 64, 69, 71, 72, 74, 75, 77

**PoC** Proof of Concept. 55

**WAF** Web Application Firewall. 11, 14

**XSS** Cross-Site Scripting. 1–25, 28–30, 33, 35, 37, 39–43, 45, 47, 48, 51–56, 58–60, 62–74, 77

# Bibliography


[1] OWASP. Owasp top ten project, 2013. URL https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013. [Online] (visited on 2016-09-15).

[2] Bob Martin, Mason Brown, Alan Paller, Dennis Kirby, and Steve Christey. 2011 cwe/sans top 25 most dangerous software errors, 2011. URL http://cwe.mitre.org/top25/. [Online] (visited on 2016-10-05).

[3] Guy Podjarny. Xss attacks: The next wave, 2017. URL https://snyk.io/blog/xss-attacks-the-next-wave/. [Online] (visited on 2017-11-21).

[4] Alexandre Vernotte, Frédéric Dadeau, Franck Lebeau, Bruno Legeard, Fabien Peureux, and François Piat. *Efficient Detection of Multi-step Cross-Site Scripting Vulnerabilities*, pages 358–377. Springer International Publishing, Cham, 2014. ISBN 978-3-319-13841-1. doi: 10.1007/978-3-319-13841-1_20. URL http://dx.doi.org/10.1007/978-3-319-13841-1_20.

[5] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A. Vela Nava, and Martin Johns. Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1709–1723, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134091. URL http://doi.acm.org/10.1145/3133956.3134091.

[6] Lucian Constantin. Xss flaw in popular video-sharing site allowed ddos attack through browsers, 2014. URL http://www.computerworld.com/article/2489547/malware-vulnerabilities/xss-flaw-in-popular-video-sharing-site-allowed-ddos-attack-through-browsers.html. [Online] (visited on 2016-09-15).

[7] Dennis Fisher. Researchers uncover interesting browser-based botnet, 2014. URL https://threatpost.com/researchers-uncover-interesting-browser-based-botnet/105250/. [Online] (visited on 2016-09-15).

[8] Lisa Vaas. ebay xss bug left users vulnerable to (almost) undetectable phishing attacks, 2016. URL https://nakedsecurity.sophos.com/2016/01/13/

ebay-xss-bug-left-users-vulnerable-to-almost-undetectable-phishing-attacks/. [Online] (visited on 2016-09-15).

[9] Rodolfo Assis. baidu.com security vulnerability, 2015. URL https://www.openbugbounty.org/incidents/59483/. [Online] (visited on 2016-09-15).

[10] R3NW4. yahoo.com security vulnerability, 2016. URL https://www.openbugbounty.org/incidents/139816/. [Online] (visited on 2016-09-15).

[11] Bini Shala. amazon.com security vulnerability, 2016. URL https://www.openbugbounty.org/incidents/152371/. [Online] (visited on 2016-09-15).

[12] Bini Shala. Google stored xss-es, 2016. URL http://bini.tech/google-stored-xss-es/. [Online] (visited on 2016-09-15).

[13] OWASP. Cross-site scripting (xss), 2016. URL https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). [Online] (visited on 2016-09-15).

[14] dionach. The real impact of cross-site scripting, 2016. URL https://www.dionach.com/blog/the-real-impact-of-cross-site-scripting. [Online] (visited on 2016-09-15).

[15] Martin Johns. Code-injection vulnerabilities in web applications—exemplified at cross-site scripting. *It-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, 53(5):256–260, 2011.

[16] Andrea Avancini and Mariano Ceccato. Security testing of web applications: A search-based approach for cross-site scripting vulnerabilities. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*, pages 85–94, Sept 2011. doi: 10.1109/SCAM.2011.7.

[17] Isatou Hydara, Abu Bakar Md. Sultan, Hazura Zulzalil, and Novia Admodisastro. Current state of research on cross-site scripting (xss) – a systematic literature review. *Information and Software Technology*, 58:170 – 186, 2015. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2014.07.010. URL http://www.sciencedirect.com/science/article/pii/S0950584914001700.

[18] V Nithya, S Lakshmana Pandian, and C Malarvizhi. A survey on detection and prevention of cross-site scripting attack. *International Journal of Security and Its Applications*, 9(3):139–151, 2015.

[19] Sunil Arora. Javascript frameworks: The best 10 for modern web apps, 2016. URL http://noeticforce.com/best-Javascript-frameworks-for-single-page-modern-web-applications. [Online] (visited on 2016-09-15).

[20] Mikito Takada. Single page apps in depth, 2016. URL http://singlepageappbook.com/goal.html. [Online] (visited on 2016-09-15).

[21] Mozilla Developer Network and individual contributors. HTML5, 2016. URL https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5. [Online] (visited on 2016-09-15).

[22] Creative Bloq Staff. The top 10 realtime web apps, 2013. URL http://www.creativebloq.com/app-design/top-10-realtime-web-apps-5133752. [Online] (visited on 2016-09-15).

[23] Ian Fette and Alexey Melnikov. The websocket protocol. RFC 6455, RFC Editor, December 2011. URL http://www.rfc-editor.org/rfc/rfc6455.txt. http://www.rfc-editor.org/rfc/rfc6455.txt.

[24] Adam Bergkvist, Daniel Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. Webrtc 1.0: Real-time communication between browsers. Working draft, W3C, 7 2016. https://www.w3.org/TR/2016/WD-webrtc-20160913/.

[25] Mozilla Developer Network and individual contributors. Server-sent events, 2015. URL https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events. [Online] (visited on 2016-09-15).

[26] Mozilla Developer Network and individual contributors. Xmlhttprequest level 2, 2016. URL https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest. [Online] (visited on 2016-09-15).

[27] Mozilla Developer Network and individual contributors. Using web workers, 2016. URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. [Online] (visited on 2016-09-15).

[28] Mozilla Developer Network and individual contributors. Fetch api, 2016. URL https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. [Online] (visited on 2016-09-15).

[29] Wei Xu, Fangfang Zhang, and Sencun Zhu. The power of obfuscation techniques in malicious javascript code: A measurement study. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*, pages 9–16, 10 2012. doi: 10.1109/MALWARE.2012.6461002.

[30] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Jshadobf: A javascript obfuscator based on multi-objective optimization algorithms. In *International Conference on Network and System Security*, pages 336–349. Springer, 2013.

[31] Guowei Dong, Yan Zhang, Xin Wang, Peng Wang, and Liangkun Liu. Detecting cross site scripting vulnerabilities introduced by html5. In *Computer Science and*

*Software Engineering (JCSSE), 2014 11th International Joint Conference on*, pages 319–323, May 2014. doi: 10.1109/JCSSE.2014.6841888.

[32] cure53. Html5 security cheatsheet - a collection of html5 related xss attack vectors, 2016. URL https://html5sec.org/#html5. [Online] (visited on 2017-11-20).

[33] Fabien Duchene. How i evolved your fuzzer: Techniques for black-box evolutionary fuzzing. In *Sec-T*, 2014.

[34] Enrico Bazzoli, Claudio Criscione, Federico Maggi, and Stefano Zanero. *XSS PEEKER: Dissecting the XSS Exploitation Techniques and Fuzzing Mechanisms of Blackbox Web Application Scanners*, pages 243–258. Springer International Publishing, Cham, 2016. ISBN 978-3-319-33630-5. doi: 10.1007/978-3-319-33630-5_17. URL http://dx.doi.org/10.1007/978-3-319-33630-5_17.

[35] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. Kameleonfuzz: Evolutionary fuzzing for black-box xss detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 37–48, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2278-2. doi: 10.1145/2557547.2557550. URL http://doi.acm.org/10.1145/2557547.2557550.

[36] Shay Chen. The reflected xss detection accuracy of web application scanners, 2016. URL http://www.sectoolmarket.com/reflected-cross-site-scripting-detection-accuracy-unified-list.html. [Online] (visited on 2017-11-22).

[37] FAKHRELDEEN ABBAS SAEED and E ABED ELGABAR. Assessment of open source web application security scanners. *Journal of Theoretical and Applied Information Technology*, 61(2), 2014.

[38] Dengfeng Xia. Comparing web application scanners for xss attacks, 2013.

[39] Natasa Suteva, Dragi Zlatkovski, and Aleksandra Mileva. Evaluation and testing of several free/open source web vulnerability scanners. In *The 10th Conference for Informatics and Information Technology (CIIT 2013), 18-21 Apr 2013 , Bitola, Macedonia*, 2013.

[40] Kinnaird McQuade. Open source web vulnerability scanners: The cost effective choice. In *Proceedings of the Conference for Information Systems Applied Research ISSN*, volume 2167, page 1508, 2014.

[41] Claudio Criscione. Ready, aim, fire: an open-source tool to test web security scanners, 2014. URL https://security.googleblog.com/2014/11/ready-aim-fire-open-source-tool-to-test.html. [Online] (visited on 2016-10-07).

[42] Takeshi Matsuda, Daiki Koizumi, and Michio Sonoda. Cross site scripting attacks detection algorithm based on the appearance position of characters. In *Communications, Computers and Applications (MIC-CCA), 2012 Mosharaka International Conference on*, pages 65–70, Oct 2012.

[43] Yu Sun and Dake He. Model checking for the defense against cross-site scripting attacks. In *Computer Science Service System (CSSS), 2012 International Conference on*, pages 2161–2164, Aug 2012. doi: 10.1109/CSSS.2012.537.

[44] Isatou Hydara, Abu Bakar Md Sultan, Hazura Zulzalil, and Novia Admodisastro. Cross-site scripting detection based on an enhanced genetic algorithm. *Indian Journal of Science and Technology*, 8(30), 2015. doi: 10.17485/ijst/2015/v8i30/86055. URL http://dx.doi.org/10.17485/ijst/2015/v8i30/86055.

[45] Abdalla Wasef Marashdih, Zarul Fitri Zaaba, and Herman Khalid Omer. Web security: Detection of cross site scripting in php web application using genetic algorithm. *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, 8(5):64–75, 2017.

[46] Lwin Khin Shar and Hee Beng Kuan Tan. Automated removal of cross site scripting vulnerabilities in web applications. *Information and Software Technology*, 54(5): 467 – 478, 2012. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2011.12.006. URL http://www.sciencedirect.com/science/article/pii/S0950584911002503.

[47] Jinkun Pan and Xiaoguang Mao. Detecting dom-sourced cross-site scripting in browser extensions. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pages 24–34. IEEE, 2017.

[48] Lwin Khin Shar and Hee Beng Kuan Tan. Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1293–1296, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337399.

[49] Mukesh Kumar Gupta, Mahesh Chandra Govil, and Girdhari Singh. Predicting cross-site scripting (xss) security vulnerabilities in web applications. In *Computer Science and Software Engineering (JCSSE), 2015 12th International Joint Conference on*, pages 162–167, July 2015. doi: 10.1109/JCSSE.2015.7219789.

[50] Julian Thome, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. Technical report, SnT Centre-University of Luxembourg, 2017.

[51] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Search-driven string constraint solving for vulnerability detection. In *Proceedings of the*

*39th International Conference on Software Engineering*, pages 198–208. IEEE Press, 2017.

[52] Michelle E. Ruse and Samik Basu. Detecting cross-site scripting vulnerability using concolic testing. In *Information Technology: New Generations (ITNG), 2013 Tenth International Conference on*, pages 633–638, April 2013. doi: 10.1109/ITNG.2013.97.

[53] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. *Andromeda: Accurate and Scalable Security Analysis of Web Applications*, pages 210–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-37057-1. doi: 10.1007/978-3-642-37057-1_15. URL http://dx.doi.org/10.1007/978-3-642-37057-1_15.

[54] Mukesh Kumar Gupta, Mahesh Chandra Govil, Girdhari Singh, and Priya Sharma. Xssdm: Towards detection and mitigation of cross-site scripting vulnerabilities in web applications. In *Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on*, pages 2010–2015, Aug 2015. doi: 10.1109/ICACCI.2015.7275912.

[55] R Suguna, T Kujani, N Suganya, and C Krishnaveni. Hunting pernicious attacks in web applications with xprober. *American Journal of Applied Sciences*, 11(7): 1164, 2014.

[56] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. Code injection attacks on html5-based mobile apps: Characterization, detection and mitigation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 66–77, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660275. URL http://doi.acm.org/10.1145/2660267.2660275.

[57] Mahmoud Mohammadi, Bill Chu, and Heather Ritcher Lipford. Poster: Using unit testing to detect sanitization flaws. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1659–1661, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2810130. URL http://doi.acm.org/10.1145/2810103.2810130.

[58] Mahmoud Mohammadi, Bill Chu, Heather Richter Lipford, and Emerson Murphy-Hill. Automatic web security unit testing: Xss vulnerability detection. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, pages 78–84, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4151-6. doi: 10.1145/2896921.2896929. URL http://doi.acm.org/10.1145/2896921.2896929.

[59] Mahmoud Mohammadi, Bill Chu, and Heather Richter Lipford. Detecting cross-site scripting vulnerabilities through automated unit testing. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on*, pages 364–373. IEEE, 2017.

[60] Andrea Avancini and Mariano Ceccato. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Information and Software Technology*, 55(12):2209 – 2222, 2013. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2013.08.001. URL http://www.sciencedirect.com/science/article/pii/S0950584913001602.

[61] Andrea Avancini and Mariano Ceccato. Towards security testing with taint analysis and genetic algorithms. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 65–71, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-965-7. doi: 10.1145/1809100.1809110. URL http://doi.acm.org/10.1145/1809100.1809110.

[62] Hossain Shahriar and Hisham Haddad. Risk assessment of code injection vulnerabilities using fuzzy logic-based system. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1164–1170, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2469-4. doi: 10.1145/2554850.2555071. URL http://doi.acm.org/10.1145/2554850.2555071.

[63] Shay Chen. Top 10: The web application vulnerability scanners benchmark, 2012 - commercial & open source scanners, 2012. URL http://sectooladdict.blogspot.co.at/2012/07/2012-web-application-scanner-benchmark.html. [Online] (visited on 2017-11-22).

[64] Chandan Kumar. 12 online free tools to scan website security vulnerabilities & malware, 2017. URL https://geekflare.com/online-scan-website-security-vulnerabilities/. [Online] (visited on 2017-11-22).

[65] Brian Shura. Web application security scanner list, 2010. URL http://projects.webappsec.org/w/page/13246988/Web%20Application%20Security%20Scanner%20List. [Online] (visited on 2017-11-22).

[66] OWASP. Vulnerability scanning tools, 2016. URL https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools. [Online] (visited on 2017-11-23).

[67] Gordon Lyon. Top 125 network security tools, 2015. URL http://sectools.org/tag/web-scanners/. [Online] (visited on 2017-11-22).

[68] Pavitra Shankdhar. 14 best open source web application vulnerability scanners, 2014. URL http://resources.infosecinstitute.com/14-popular-web-application-vulnerability-scanners/. [Online] (visited on 2017-11-22).

[69] Abdulrahman Alzahrani, Ali Alqazzaz, Ye Zhu, Huirong Fu, and Nabil Almashfi. Web application security tools analysis. In *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and*

*Security (IDS), 2017 IEEE 3rd International Conference on*, pages 237–242. IEEE, 2017.

[70] Yuan-Hsin Tung, Shian-Shyong Tseng, Jen-Feng Shih, and Hwai-Ling Shan. W-vst: A testbed for evaluating web vulnerability scanner. In *2014 14th International Conference on Quality Software*, pages 228–233, Oct 2014. doi: 10.1109/QSIC.2014.50.

[71] Jinkun Pan, Xiaoguang Mao, and Weishi Li. Taint inference for cross-site scripting in context of url rewriting and html sanitization. *ETRI Journal*, 38(2):376–386, 2016. doi: 10.4218/etrij.16.0115.0570. URL http://dx.doi.org/10.4218/etrij.16.0115.0570.

[72] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. Dexterjs: Robust testing platform for dom-based xss vulnerabilities. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 946–949, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2803191. URL http://doi.acm.org/10.1145/2786805.2803191.

[73] Thiago S. Rocha and Eduardo Souto. Etssdetector: A tool to automatically detect cross-site scripting vulnerabilities. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 306–309, Aug 2014. doi: 10.1109/NCA.2014.53.

[74] B. A. Vishnu and K. P. Jevitha. Prediction of cross-site scripting attack using machine learning algorithms. In *Proceedings of the 2014 International Conference on Interdisciplinary Advances in Applied Computing*, ICONIAAC '14, pages 55:1–55:5, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2908-8. doi: 10.1145/2660859.2660969. URL http://doi.acm.org/10.1145/2660859.2660969.

[75] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1193–1204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516703. URL http://doi.acm.org/10.1145/2508859.2516703.

[76] Claudio Criscione. Google firing range on github, 2014. URL https://github.com/google/firing-range. [Online] (visited on 2016-10-07).

[77] Xiaobing Guo, Shuyuan Jin, and Yaxing Zhang. Xss vulnerability detection using optimized attack vector repertory. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2015 International Conference on*, pages 29–36, Sept 2015. doi: 10.1109/CyberC.2015.50.

[78] Josip Bozic and Franz Wotawa. Xss pattern for attack modeling in testing. In *Proceedings of the 8th International Workshop on Automation of Software Test*, AST

'13, pages 71–74, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-6161-3. URL http://dl.acm.org/citation.cfm?id=2662413.2662429.

[79] Adam Barth. Chromium blog: Security in depth: New security features, 2010. URL https://blog.chromium.org/2010/01/security-in-depth-new-security-features.html. [Online] (visited on 2016-10-21).

[80] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 91–100, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772701. URL http://doi.acm.org/10.1145/1772690.1772701.

[81] David Ross. Ie8 security part iv: The xss filter, 2008. URL https://blogs.msdn.microsoft.com/ie/2008/07/02/ie8-security-part-iv-the-xss-filter/. [Online] (visited on 2016-10-21).

[82] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against dom-based cross-site scripting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 655–670, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/stock.

[83] Nick Nikiforakis. Bypassing chrome's anti-xss filter, 2011. URL http://blog.securitee.org/?p=37. [Online] (visited on 2016-10-21).

[84] Rodolfo Assis. Chrome xss bypass, 2016. URL http://brutelogic.com.br/blog/chrome-xss-bypass/. [Online] (visited on 2016-10-21).

[85] Mario Heiderich, Alex Inführ, Fabian Fäßler, Nikolai Krein, Masato Kinugawa, Tsang-Chi Hong, Dario Weißer, and Paula Pustułka. Cure53 browser security white paper, 9 2017. URL https://github.com/cure53/browser-sec-whitepaper.

[86] Mozilla Developer Network and individual contributors. Same-origin policy, 2016. URL https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. [Online] (visited on 2016-10-25).

[87] Adam Barth. The web origin concept. RFC 6454, RFC Editor, 12 2011. URL https://tools.ietf.org/html/rfc6454.

[88] Giorgio Maone. Noscript - javascript/java/flash blocker for a safer firefox experience!, 2016. URL https://noscript.net/features. [Online] (visited on 2016-10-21).

[89] Andrew Y. Scriptsafe, 2016. URL https://chrome.google.com/webstore/detail/scriptsafe/oiigbmnaadbkfbmpbfijlflahbdbdgdf. [Online] (visited on 2016-10-21).

[90] Raymond Hill. umatrix, 2016. URL https://chrome.google.com/webstore/detail/umatrix/ogfcmafjalglgifnmanfmnieipoejdcf?hl=en. [Online] (visited on 2016-10-21).

[91] Shashank Gupta and Brij Bhooshan Gupta. Xss-immune: a google chrome extension-based xss defensive framework for contemporary platforms of web applications. *Security and Communication Networks*, 2016. ISSN 1939-0122. doi: 10.1002/sec.1579. URL http://dx.doi.org/10.1002/sec.1579. SCN-16-0123.R1.

[92] Shashank Gupta and Brij Bhooshan Gupta. Bds: browser dependent xss sanitizer. *Book on Cloud-Based Databases with Biometric Applications, IGI-Global's Advances in Information Security, Privacy, and Ethics (AISPE) series*, pages 174–191, 2014.

[93] Bhawna Mewara, Sheetal Bairwa, Jyoti Gajrani, and Vinesh Jain. Enhanced browser defense for reflected cross-site scripting. In *Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 2014 3rd International Conference on*, pages 1–6, Oct 2014. doi: 10.1109/ICRITO.2014.7014761.

[94] Brandon Sterne and Adam Barth. Content security policy 1.0. Candidate recommendation, W3C, November 2012. http://www.w3.org/TR/2012/CR-CSP-20121115/.

[95] Mike West, Adam Barth, Dan Veditz, and Brandon Sterne. Content security policy level 2. Candidate recommendation, W3C, July 2015. http://www.w3.org/TR/CSP2/.

[96] Mike West. Content security policy level 3. Working draft, W3C, September 2016. http://www.w3.org/TR/CSP3/.

[97] Steven Van Acker, Daniel Hausknecht, and Andrei Sabelfeld. Data exfiltration in the face of csp. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 853–864, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4233-9. doi: 10.1145/2897845.2897899. URL http://doi.acm.org/10.1145/2897845.2897899.

[98] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[99] Michael Weissbacher, Tobias Lauinger, and William Robertson. *Why Is CSP Failing? Trends and Challenges in CSP Adoption*, pages 212–233. Springer International Publishing, Cham, 2014. ISBN 978-3-319-11379-1. doi: 10.1007/978-3-319-11379-1_11. URL http://dx.doi.org/10.1007/978-3-319-11379-1_11.

[100] Mattia Fazzini, Prateek Saxena, and Alessandro Orso. Autocsp: Automatically retrofitting csp to web applications. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 336–346, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5. URL http://dl.acm.org/citation.cfm?id=2818754.2818797.

[101] Shukai Liu, Xuexiong Yan, Qingxian Wang, Xu Zhao, Chuansen Chai, and Yajing Sun. A protection mechanism against malicious html and javascript code in vulnerable web applications. *Mathematical Problems in Engineering*, 2016, 2016. doi: 10.1155/2016/7107042. URL http://dx.doi.org/10.1155/2016/7107042.

[102] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 601–610, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242654. URL http://doi.acm.org/10.1145/1242572.1242654.

[103] Stefan Prandl, Mihai Lazarescu, and Duc-Son Pham. *A Study of Web Application Firewall Solutions*, pages 501–510. Springer International Publishing, Cham, 2015. ISBN 978-3-319-26961-0. doi: 10.1007/978-3-319-26961-0_29. URL http://dx.doi.org/10.1007/978-3-319-26961-0_29.

[104] Michael Becher. *Web Application Firewalls*. VDM Verlag, Saarbrücken, Germany, Germany, 2007. ISBN 383640446X, 9783836404464.

[105] G. Rama Koteswara Rao, R. Satya Prasad, and M. Ramesh. Neutralizing cross-site scripting attacks using open source technologies. In *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ICTCS '16, pages 24:1–24:6, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3962-9. doi: 10.1145/2905055.2905230. URL http://doi.acm.org/10.1145/2905055.2905230.

[106] Metin Sahin and Ibrahim Sogukpınar. An efficient firewall for web applications (efwa). In *Computer Science and Engineering (UBMK), 2017 International Conference on*, pages 1150–1155. IEEE, 2017.

[107] Piyush A. Sonewar and Nalini A. Mhetre. A novel approach for detection of sql injection and cross site scripting attacks. In *Pervasive Computing (ICPC), 2015 International Conference on*, pages 1–4, Jan 2015. doi: 10.1109/PERVASIVE.2015.7087131.

[108] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, 31(4):612 – 628, 2012. ISSN 0167-4048. doi: http://dx.doi.org/10.1016/j.cose.2011.12.004. URL http://www.sciencedirect.com/science/article/pii/S0167404811001477.

[109] Shashank Gupta and Brij Bhooshan Gupta. Xss-safe: A server-side approach to detect and mitigate cross-site scripting (xss) attacks in javascript code. *Arabian Journal for Science and Engineering*, 41(3):897–920, 2016. ISSN 2191-4281. doi: 10.1007/s13369-015-1891-7. URL http://dx.doi.org/10.1007/s13369-015-1891-7.

[110] Hossain Shahriar, Sarah M North, YoonJi Lee, and Roger Hu. Server-side code injection attack detection based on kullback-leibler distance. *International Journal of Internet Technology and Secured Transactions 8*, 5(3):240–261, 2014.

[111] A Duraisamy, M Sathiyamoorthy, and S Chandrasekar. A server side solution for protection of web applications from cross-site scripting attacks. *International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN*, pages 2278–3075, 2013.

[112] M Ridwan Zalbina, Tri Wanda Septian, Deris Stiawan, Moh Yazid Idris, Ahmad Heryanto, and Rahmat Budiarto. Payload recognition and detection of cross site scripting attack. In *Anti-Cyber Crimes (ICACC), 2017 2nd International Conference on*, pages 172–176. IEEE, 2017.

[113] Swaswati Goswami, Nazrul Hoque, Dhruba K Bhattacharyya, and Jugal Kalita. An unsupervised method for detection of xss attack. *IJ Network Security*, 19(5): 761–775, 2017.

[114] Swati Maurya. Positive security model based server-side solution for prevention of cross-site scripting attacks. In *2015 Annual IEEE India Conference (INDICON)*, pages 1–5, Dec 2015. doi: 10.1109/INDICON.2015.7443473.

[115] Joel Kamdem Teto, Ruth Bearden, and Dan Chia-Tien Lo. The impact of defensive programming on i/o cybersecurity attacks. In *Proceedings of the SouthEast Conference*, pages 102–111. ACM, 2017.

[116] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 587–600, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6. doi: 10.1145/2046707.2046775. URL http://doi.acm.org/10.1145/2046707.2046775.

[117] Jonas Ceponis, Lina Ceponiene, Algimantas Venckauskas, and Dainius Mockus. *Evaluation of Open Source Server-Side XSS Protection Solutions*, pages 345–356. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41947-8. doi: 10.1007/978-3-642-41947-8_29. URL http://dx.doi.org/10.1007/978-3-642-41947-8_29.

[118] Dimitris Mitropoulos, Panagiotis Louridas, Michalis Polychronakis, and Angelos D Keromytis. Defending against web application attacks: Approaches, challenges and implications. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[119] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Joanaudit: a tool for auditing common injection vulnerabilities. In *11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 2017.

[120] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1205–1216, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516708. URL `http://doi.acm.org/10.1145/2508859.2516708`.

[121] Vikas K. Malviya, Saket Saurav, and Atul Gupta. On security issues in web applications through cross site scripting (xss). In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 583–588, Dec 2013. doi: 10.1109/APSEC.2013.85.

[122] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 777–788, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516723. URL `http://doi.acm.org/10.1145/2508859.2516723`.

[123] Sebastian Lekies, Ben Stock, Martin Wentzel, and Martin Johns. The unexpected dangers of dynamic javascript. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 723–735, Washington, D.C., August 2015. USENIX Association. ISBN 978-1-931971-232. URL `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies`.

[124] Jeremiah Grossman. *XSS Attacks: Cross-site scripting exploits and defense*. Syngress, 2007.

[125] Haneet Kour and Lalit Sen Sharma. Tracing out cross site scripting vulnerabilities in modern scripts. *International Journal of Advanced Networking and Applications*, 7(5):2862, 2016.

[126] Ankita Singh and Amit Saxena. Cross site scripting: A survey paper. *prevention*, 1 (02), 2014.

[127] Philippe Le Hégaret, Jonathan Robie, Mike Champion, Lauren Wood, Steven B Byrne, Gavin Nicol, and Arnaud Le Hors. Document object model (DOM) level 3 core specification. W3C recommendation, W3C, April 2004. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407.

[128] Amit Klein. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium*, 2005. URL http://www.webappsec.org/projects/articles/071105.shtml.

[129] Gareth Heyes. Shazzer - shared fuzzer, 2017. URL http://shazzer.co.uk/vectors. [Online] (visited on 2017-11-22).

[130] R3NW4. New and 0day xss vectors collected from everywhere, 2015. URL https://www.openbugbounty.org/forum/viewtopic.php?t=7. [Online] (visited on 2017-11-22).

[131] Rodolfo Assis. Xss cheat sheet, 2017. URL http://brutelogic.com.br/blog/cheat-sheet/. [Online] (visited on 2017-11-22).

[132] Robert Hansen, Adam Lange, and Mishra Dhiraj. Xss filter evasion cheat sheet, 2017. URL https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet. [Online] (visited on 2017-11-22).

[133] Fraser Howard. Malware with your mocha: Obfuscation and anti emulation tricks in malicious javascript, 2010. URL http://www.sophos.com/medialibrary/PDFs/technical%20papers/malware_with_your_mocha.pdf.

[134] Kazumasa Itabashi. Portable document format malware. *Symantec Security Response*, 2011. URL http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/portable_document_format_malware.pdf.

[135] Martin Kleppe. Jsfuck - write any javascript with 6 characters, 2010. URL http://www.jsfuck.com/. Original idea of http://sla.ckers.org/forum/read.php?24,32930.

[136] OWASP. Owasp webgoat project, 2016. URL https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project. [Online] (visited on 2016-10-26).

[137] DVWA team. Damn vulnerable web application (dvwa), 2016. URL https://github.com/ethicalhack3r/DVWA. [Online] (visited on 2016-11-03).

[138] @filedescriptor. Css: Cascading style scripting, 2016. URL https://blog.innerht.ml/cascading-style-scripting/. [Online] (visited on 2018-01-09).

[139] netmarketshare.com. Browser market share, 2017. URL https://www.netmarketshare.com/browser-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22deviceType%22%3A%7B%22%24in%22%3A%5B%22Mobile%22%2C%22Desktop%2Flaptop%22%2C%22Tablet%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Custom%22%2C%22attributes%22%3A%22share%22%2C%22group%

22%3A%22browser%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22browsersMobile%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222017-11%22%2C%22dateEnd%22%3A%222017-11%22%2C%22segments%22%3A%22-1000%22%7D. [Online] (visited on 2017-12-22).

[140] w3counter.com. Browser & platform market share november 2017, 2017. URL https://www.w3counter.com/globalstats.php?year=2017&month=11. [Online] (visited on 2017-12-22).

[141] Docker Inc. Docker, 2017. URL https://www.docker.com/. [Online] (visited on 2017-12-20).

[142] Lavakumar Kuppan. Ironwasp - iron web application advanced security testing platform, 2014. URL http://ironwasp.org/. [Online] (visited on 2016-10-18).

[143] Subgraph. Vega vulnerability scanner, 2014. URL https://subgraph.com/vega/index.en.html. [Online] (visited on 2016-10-18).

[144] OWASP. Owasp zed attack proxy project, 2016. URL https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project. [Online] (visited on 2016-10-18).

[145] Sarosys OOD. Arachni - web application security scanner framework, 2016. URL http://www.arachni-scanner.com/. [Online] (visited on 2016-10-18).

[146] Nicolas Surribas. Wapiti : a free and open-source web-application vulnerability scanner in python for windows, linux, bsd, osx, 2014. URL http://wapiti.sourceforge.net/. [Online] (visited on 2016-10-18).

[147] "psy". Xsser: Cross site "scripter", 2016. URL https://xsser.03c8.net/. [Online] (visited on 2016-10-18).

[148] Andres Riancho. w3af - open source web application security scanner, 2013. URL http://w3af.org/. [Online] (visited on 2016-10-18).

[149] Chris Sullo and David Lodge. Nikto2, 2016. URL https://cirt.net/nikto2. [Online] (visited on 2016-10-18).

[150] Miroslav Stampar. Damn small xss scanner, 2016. URL https://github.com/stamparm/DSXS. [Online] (visited on 2016-10-18).

[151] Syhunt Security. Web application security scanner - syhunt, 2016. URL http://www.syhunt.com/en/. [Online] (visited on 2016-10-18).

[152] Tinfoil Security Inc. Website security | recurring, affordable, and usable, 2016. URL https://www.tinfoilsecurity.com/. [Online] (visited on 2016-10-18).

[153] MileSCAN Technologies. Milescan parospro, 2013. URL http://www.milescan.com/software.html. [Online] (visited on 2017-11-23).

# Appendices

## Appendix A: Detection Results

| Detected | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Syhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 133 | 64 | 63 | 62 | 60 | 46 | 44 | 57 | 0 | 80 | 60 | 20 | 39 |
| xss playground | 1346 | 658 | 437 | 690 | 578 | 422 | 187 | 473 | 0 | 218 | 236 | 0 | 350 |
| total | 1479 | 722 | 500 | 752 | 638 | 468 | 231 | 530 | 0 | 298 | 296 | 20 | 389 |

Table 1: Absolute number of detected test cases per scanner and testbed.

| Detected | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Syhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 89,86% | 43,24% | 42,57% | 41,89% | 40,54% | 31,08% | 29,73% | 38,51% | 0,00% | 54,05% | 40,54% | 13,51% | 26,35% |
| xss playground | 90,40% | 44,19% | 29,35% | 46,34% | 38,82% | 28,34% | 12,56% | 31,77% | 0,00% | 14,64% | 15,85% | 0,00% | 23,51% |
| total | 90,35% | 44,11% | 30,54% | 45,94% | 38,97% | 28,59% | 14,11% | 32,38% | 0,00% | 18,20% | 18,08% | 1,22% | 23,76% |

Table 2: Percentage of detected test cases for each scanner and testbed.

| False positive | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Syhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 0 | 12 | 20 | 9 | 8 | 2 | 18 | 10 | 0 | 14 | 12 | 1 | 8 |
| xss playground | 0 | 22 | 25 | 28 | 16 | 8 | 13 | 28 | 0 | 7 | 12 | 0 | 10 |
| total | 0 | 34 | 45 | 37 | 24 | 10 | 31 | 38 | 0 | 21 | 24 | 1 | 18 |

Table 3: Absolute number of false positive results of non-detectable test cases for each scanner and testbed.

| False positive | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Syhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 0,00% | 19,35% | 32,26% | 14,52% | 12,90% | 3,23% | 29,03% | 16,13% | 0,00% | 22,58% | 19,35% | 1,61% | 12,90% |
| xss playground | 0,00% | 20,18% | 22,94% | 25,69% | 14,68% | 7,34% | 11,93% | 25,69% | 0,00% | 6,42% | 11,01% | 0,00% | 9,17% |
| total | 0,00% | 19,88% | 26,32% | 21,64% | 14,04% | 5,85% | 18,13% | 22,22% | 0,00% | 12,28% | 14,04% | 0,58% | 10,53% |

Table 4: Percentage of false positive results of non-detectable test cases for each scanner and testbed.

| False negative | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Syhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 15 | 96 | 105 | 95 | 96 | 104 | 122 | 101 | 148 | 82 | 100 | 129 | 117 |
| xss playground | 143 | 853 | 1077 | 827 | 927 | 1075 | 1315 | 1044 | 1489 | 1278 | 1265 | 1489 | 1149 |
| total | 158 | 949 | 1182 | 922 | 1023 | 1179 | 1437 | 1145 | 1637 | 1360 | 1365 | 1618 | 1266 |

Table 5: Absolute number of missed vulnerabilities for each scanner and testbed.

| False negative | FOXSS | IronWasp | Vega | ZAP | Arachni | Wapiti | XSSer | w3af | Nikto | DSXS | Syhunt CE | Tinfoil | ParosPro |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| firing range | 10,14% | 64,86% | 70,95% | 64,19% | 64,86% | 70,27% | 82,43% | 68,24% | 100,00% | 55,41% | 67,57% | 87,16% | 79,05% |
| xss playground | 9,60% | 57,29% | 72,33% | 55,54% | 62,26% | 72,20% | 88,31% | 70,11% | 100,00% | 85,83% | 84,96% | 100,00% | 77,17% |
| total | 9,65% | 57,97% | 72,21% | 56,32% | 62,49% | 72,02% | 87,78% | 69,95% | 100,00% | 83,08% | 83,38% | 98,84% | 77,34% |

Table 6: Percentage of missed vulnerabilities for each scanner and testbed.