Untangling the Web of Client-Side Cross-Site Scripting

Untersuchungen zu Client-seitigem Cross-Site Scripting

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg
zur Erlangung des Grades

D O K T O R - I N G E N I E U R

vorgelegt von

Benjamin Stock
aus Mainz

Als Dissertation genehmigt von
der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Tag der mündlichen Prüfung:	31.08.2015
Vorsitzende des Promotionsorgans:	Prof. Dr.-Ing. habil. Marion Merklein
Gutachter:	Prof. Dr.-Ing. Felix Freiling
	Prof. Dr. Michael Backes

Abstract

The Web's functionality has shifted from purely server-side code to rich client-side applications, which allow the user to interact with a site without the need for a full page load. While server-side attacks, such as SQL or command injection, still pose a threat, this change in the Web's model also increases the impact of vulnerabilities aiming at exploiting the client. The most prominent representative of such client-side attacks is Cross-Site Scripting, where the attacker's goal is to inject code of his choosing into the application, which is subsequently executed by the victimized browsers in the context of the vulnerable site.

This thesis provides insights into different aspects of Cross-Site Scripting. First, we show that the concept of password managers, which aim to allow users to choose more secure passwords and, thus, increase the overall security of user accounts, is susceptible to attacks which abuse Cross-Site Scripting flaws. In our analysis, we found that almost all built-in password managers can be leveraged by a Cross-Site Scripting attacker to steal stored credentials. Based on our observations, we present a secure concept for password managers, which does not insert password data into the document such that it is accessible from the injected JavaScript code. We evaluate our approach from a functional and security standpoint and find that our proposal provides additional security benefits while not causing any incompatibilities.

Our work then focusses on a sub-class of Cross-Site Scripting, namely Client-Side Cross-Site Scripting. We design, implement and execute a study into the prevalence of this class of flaws at scale. To do so, we implement a taint-aware browsing engine and an exploit generator capable of precisely producing exploits based on our gathered data on suspicious, tainted flows. Our subsequent study of the Alexa top 5,000 domains shows that almost one out of ten of these domains carry at least one Client-Side Cross-Site Scripting vulnerability.

We follow up on these flaws by analyzing the gathered flow data in depth in search of the root causes of this class of vulnerability. To do so, we first discuss the complexities inherent to JavaScript and define a set of metrics to measure said complexity. We then classify the vulnerable snippets of code we discovered according to these metrics and present the key insights gained from our analysis. In doing so, we find that the reasons for such flaws are manifold, ranging from simple unawareness of developers to incompatibilities between, otherwise safe, first- and third-party code.

In addition, we investigate the capability of the state of the art of Cross-Site Scripting filters in the client, the XSS Auditor, finding that several conceptual issues exist which an attacker to subvert of its protection capabilities. In total, we show that the Auditor can be bypassed on over 80% of the vulnerable domains in our data set, highlighting that it is ill-equipped to stop Client-Side Cross-Site Scripting. Motivated by our findings, we present a concept for a filter targeting Client-Side Cross-Site Scripting, combining taint tracking in the browser in conjunction with taint-aware HTML and JavaScript parsers, allowing us to robustly protect users from such attacks.

Zusammenfassung

Das Web hat sich gewandelt: von rein server-seitig implementierter Funktionalität zu mächtigen client-seitigen Applikation, die einem Benutzer die Interaktion mit einer Seite ohne Neuladen ermöglichen. Obwohl server-seitige Angriffe, wie etwa SQL- oder Command-Injections, weiterhin ein Risiko darstellen, erhöht sich durch die Änderung am Konzept des Webs auch die Auswirkung von client-seitigen Angriffen. Der bekannteste Vertreter solcher Verwundbarkeiten ist Cross-Site Scripting (XSS), eine Code-Injection-Attacke, die darauf abzielt, dass der vom Angreifer eingeschleuste Code im Browser seines Opfers mit der verwundbaren Applikation interagieren kann.

Diese Arbeit beschäftigt sich mit verschiedenen Aspekten von Cross-Site Scripting. Zuerst legt sie die Funktionsweise von Passwort-Managern dar, die Benutzer bei der Wahl sichererer Passwörter unterstützen sollen. Eine Analyse des Konzepts an sich und aktueller Browser zeigt jedoch auf, dass Passwort-Manager anfällig für Cross-Site Scripting-Angriffe sind, die das Stehlen von gespeicherten Zugangsdaten ermöglichen. Basierend auf den Erkenntnissen stellt diese Arbeit daraufhin das Konzept eines Passwort-Managers vor, der Zugangsdaten für eingeschleusten Code unerreichbar macht. Die Evaluation in Hinblick auf Funktionalität und Sicherheit zeigt, dass sich durch den vorgestellten Ansatz Vorzüge für die Sicherheit bieten und der präsentierte Ansatz keinen negativen Einfluss auf bestehende Applikationen hat.

Anschließend liegt der Fokus dieser Arbeit auf einer Unterklasse von XSS, dem Client-Seitigen Cross-Site Scripting. Sie präsentiert das Design, die Implementierung und die Durchführung einer Studie, deren Ziel es ist, die Verbreitung von Client-Seitigem XSS im Web zu erforschen. Dazu kombiniert die Arbeit einen Browser, der Taint Tracking unterstützt, und einen Exploit-Generator, der basierend auf den gesammelten Datenflüssen Exploits erzeugen kann. Die daraufhin durchgeführte Studie zeigt auf, dass fast jede zehnte Webseite in den Alexa Top 5000 Domains anfällig für Client-Seitiges Cross-Site Scripting ist. Anschließend analysiert diese Arbeit die gefundenen Schwachstellen ausführlich, um die zugrunde liegenden Ursachen von Client-Seitigem XSS zu erkunden. Dazu erläutert sie die Komplexität von JavaScript, leitet daraus Metriken zur Messung der Komplexität von verwundbarem Code ab, klassifiziert die erkannten Verwundbarkeiten und präsentiert die Schlüsselerkenntnisse. Dabei zeigt sich, dass Client-Seitiges Cross-Site Scripting sowohl durch Unwissen von Entwicklern, aber auch durch die Kombination von inkompatiblem eigenen und fremden Code verursacht wird.

Im Anschluss analysiert diese Arbeit aktuelle XSS-Filter und zeigt auf, dass moderne Filter, wie der XSS Auditor, konzeptionelle Probleme aufweisen, die es einem Angreifer erlauben, den Filter auf 80% der verwundbaren Domains zu umgehen. Diese Arbeit präsentiert daraufhin ein neues Konzept, welches auf Taint Tracking basierend von einem Angreifer eingeschleusten Code im HTML- und JavaScript-Parser erkennt und stoppt. Dies ermöglicht es, Client-Seitiges XSS zu unterbinden, wobei ein akzeptabler Performanz-Verlust und geringe False Positives entstehen.

Acknowledgments

This thesis was made possible by the invaluable support given to me by others. First and foremost, I want to thank my PhD advisor Felix Freiling for providing me with the opportunity to conduct my research and, more importantly, for initiating my interest in the field of computer security during my days as a student. In addition, I would like to thank Michael Backes for agreeing to be the secondary reviewer for my dissertation. Also, the research conducted throughout this thesis would not have been possible without Martin Johns and Sebastian Lekies, whom I want to thank for the fruitful collaboration during the pursuit of my PhD.

I also want to thank my colleagues (in alphabetical order) for supporting this thesis by proofreading it: Johannes Götzfried, Michael Gruhn, Tilo Müller, Lena Reinfelder, and Johannes Stüttgen. Next to these, I want to thank all my colleagues from the Security Research Group of the University Erlangen-Nuremberg for a friendly and entertaining work environment. Last but not least, I want to thank my parents Karin and Gerd as well as my wife Corina for their love and support over the years and especially during my time in Erlangen.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

With the advent of the so-called *Web 2.0*, the Web has witnessed a shift from purely server-side applications to the implementation of mature applications, such as Google Mail, on the client side. This change in the client/server paradigm is naturally accompanied by an increase in complexity of client-side code and, thus, a higher potential for vulnerabilities in said code. As Michael Zalewski put it in his book *The Tangled Web* (the de facto standard in browser security handbooks): "The design flaws and implementation shortcomings of the World Wide Web are those of a technology that never aspired to its current status and never had a chance to pause and look back at previous mistakes." (Zalewski, 2012).

The most prominent representative of the such flaws is *Cross-Site Scripting* (XSS), a type of code injection vulnerability which allows an attacker to execute arbitrary JavaScript code in the victim's browser, enabling him to interact with a vulnerable application in the name of his victim. Given the right circumstances, these attacks can have an even more severe impact, as is highlighted by the Ubuntuforums.org hack (Leyden, 2013). The attacker leveraged multiple Cross-Site Scripting flaws to steal authentication tokens for an administrative interface, allowing him full access to the SQL database containing, among other data, usernames and hashed passwords for all users of the site. In addition to this high-profile case, Cross-Site Scripting is an issue affecting even the most visited and audited sites, such as Google (Juenemann, 2012), Facebook and Twitter (Kafle, 2014).

In the most famous scenarios, an attacker tries to abuse a Cross-Site Scripting flaw to extract authentication tokens, as witnessed in the Ubuntuforums.org hack. The attacker may, however, also use a Cross-Site Scripting flaw towards other ends. As the attacker's code is executed in the browser of the victim, he can modify the page's content and script markup to his liking. Examples of such attacks range from defacements or placing of fake content (Heise online, 2006) to phishing attacks (Heise Security, 2006). This thesis investigates a different scenario, in which XSS can be abused to steal stored credentials from password managers.

Cross-Site Scripting has been a known issue for almost 15 years at the time of this writing (Ross, 2009). Around the year 2000, security engineers at Microsoft coined the term for what is known today as *Reflected Cross-Site Scripting*. While initially, Cross-Site Scripting was believed to be a server-side issue, Klein (2005) later discussed another variant of XSS, namely *DOM-based Cross-Site Scripting*, also referred to as *Client-Side Cross-Site Scripting*. Research in the previous years has mostly focussed on detection and mitigation of the two server-side variants, i.e.,

reflected and persistent XSS. These approaches range from static (Wassermann and Su, 2008; Tripp et al., 2014) and dynamic analysis (Bisht and Venkatakrishnan, 2008) of code to the detection of malicious, injected code on the server (Johns et al., 2008; Louw and Venkatakrishnan, 2009), in transport (Kirda et al., 2006) or in the client (Ismail et al., 2004; Maone, 2007; Nadji et al., 2009; Bates et al., 2010; Pelizzi and Sekar, 2012).

Only few researchers, however, have focussed on the detection and examination of DOM-based Cross-Site Scripting (Saxena et al., 2010a,b; Criscione, 2013). Therefore, this thesis, in addition to the highlighted attacks on password managers, focusses on the exploration of Client-Side XSS. Our motivation is to gain insights into the prevalence and nature of such flaws and, after having identified unique features of this class of vulnerability, design a filter capable of robustly stopping Client-Side Cross-Site Scripting attacks.

1.1 Contributions

This work consists of four major parts: we first present an analysis of browser-based password managers with respect to their susceptibility against Cross-Site Scripting attacks and propose a new design for more secure password managers. We follow this discussion with the design, implementation and execution of a study aimed at gaining insights into the prevalence of Client-Side Cross-Site Scripting in the wild and provide an in-depth analysis of real-world flaws. Based on this knowledge, we analyze the state of the art in Cross-Site Scripting filtering and after showing that it is inadequate to protect users against this class of attacks, we discuss the concept and evaluation of a new XSS filter. To summarize, this thesis makes the following contributions.

Exemplifying the Impact of Cross-Site Scripting Flaws After presenting the inner workings of the current generation of browser-based password managers, we highlight the potential for Cross-Site Scripting attackers to leverage the password managers' features to their advantage. We outline that the fill-in behavior of password managers can be categorized in four dimensions and analyze several browsers according to these categories. After this analysis, we present a study on observable characteristics of real-world login forms. Doing so, we find that password managers built into modern browsers are susceptible to an XSS attacker trying to extract stored credentials from them. In addition, our analysis shows that Web application developers do not deploy measures which would provide additional security against such attacks. Based on these observations, we present the concept and implementation of a new type of password manager, which ensures that an injected Cross-Site Scripting payload is unable to extract the stored login credentials. We evaluate our approach with respect to the added security and also conduct a functional evalu-

ation, showing that our approach is in no way inferior to the currently deployed solutions, while effectively stopping the described attacks.

Investigating the Prevalence of Client-Side Cross-Site Scripting in the Wild While research has focussed on the server-side variants of Cross-Site Scripting, not much attention has been given to a sub-class of XSS, which was first discussed by Amit Klein in 2005 and dubbed *DOM-based* or *Client-Side Cross-Site Scripting* (Klein, 2005). In order to gain an understanding of how prevalent this class of vulnerabilities is on the Web, we present the design of a study aimed at determining the number of vulnerable applications. To this end, we employ taint tracking inside the Chromium browsing engine to allow for a precise tracking of data flows from attacker-controllable sources to security-sensitive sinks. By combining the gathered flow information with a precise exploit generator, we are able to find such flaws at scale. Our empirical study on the Alexa top 5,000 domains discovered that almost one out of ten domains carries at least one Client-Side Cross-Site Scripting vulnerability, showing that this type of flaw is a serious risk to Web applications and their users.

Analyzing the Causes for Client-Side Cross-Site Scripting Based on the results from our empirical study, which allowed us to find a large body of real-world vulnerabilities, we aim at finding out what the underlying issues causing this class of Cross-Site Scripting are. To do so, we implement an infrastructure which allows for persistent storage of vulnerabilities for later analysis as well as normalization of the vulnerable code. After identifying metrics which allow us to measure the complexity of the discovered flaws in different dimensions, we use our infrastructure to classify each vulnerability in our data set. Doing so, we find that while a significant number of flaws have a low complexity score, several real-world flaws exist which are hard to spot and understand even by seasoned analysts. Based on key insights we gathered throughout the execution of our study, we find that the underlying causes vary from security-unaware developers and exploitable third-party libraries to complex combinations of incompatible first- and third-party code.

Protecting Users from Client-Side Cross-Site Scripting Given the knowledge that Client-Side Cross-Site Scripting is a severe and wide-spread issue, the final pillar of this thesis analyzes protection schemes which are deployed in modern browsers. We discover several issues related to the concept and placement of these measures, which allow us to bypass their protection capabilities on 776 out of 958 vulnerable domains in the Alexa top 10,000. After discussing the identified problems, we propose a new means of thwarting Client-Side Cross-Site Scripting in the browser. To do so, we extend the taint tracking scheme used to discover the vulnerabilities to ensure that taint is propagated into the JavaScript and rendering engines' parsers. In the JavaScript parser, we deploy a policy which allows user-provided data to only

produce data tokens, i.e., string, numeric or boolean literals. Similarly, we ensure that remote content can not be included from a remote host for which the origin is derived from user-provided input. By doing so, we ensure that attacker-controllable data may not be interpreted as code, thus stopping any XSS attack. We evaluate our proposed filter with respect to false negatives, false positives and performance overhead and find that it provides robust protection capabilities while only causing low false positives and a runtime overhead between 7 and 17%.

1.2 Publications

This thesis is based on different papers from which at the time of this writing all but one have been accepted to peer-reviewed conferences. This research was conducted in collaboration with several co-authors. Therefore, in the following, we will discuss the papers underlying each of the chapters and will state which parts of the research were conducted by others than the author of this thesis.

Chapter 3 presents joint work with Martin Johns, which was accepted at AsiaCCS 2014 in Kyoto, Japan (Stock and Johns, 2014). In this publication, a conceptual overview over server-side defenses, which is not part of this thesis, was conducted by Martin Johns. The basis for Chapter 4 is the publication *25 Million Flows Later: Large-Scale Detection of DOM-based XSS*, which was accepted and presented at CCS 2013 (Lekies et al., 2013). The implementation of the exploit generation scheme, which we briefly discuss in this thesis to allow for a complete understanding of the work, was conducted by Sebastian Lekies. The same work was later presented at GI Sicherheit 2014, providing additional insights into the vulnerabilities (Stock et al., 2014a).

Chapter 5 is based on the publication *From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting* to appear in CCS 2015 (Stock et al., 2015). The implementation of the taint-aware Firefox engine was carried out by Stephan Pfistner and, therefore, the details of this engine are kept to a necessary minimum in this thesis. Lastly, the concept of a taint-aware Cross-Site Scripting filter was published at USENIX Security 2014 (Stock et al., 2014b), whereas the investigation of Auditor flaws was conducted by Sebastian Lekies and Tobias Müller worked on the implementation of the filter itself. Therefore, the explanation of bypassable flaws is abbreviated and no implementational details are presented.

In addition to the papers presented in this thesis, we have co-authored three more papers. The work *Eradicating DNS Rebinding with the Extended Same-Origin Policy* has been accepted and presented at USENIX Security 2013 (Johns et al., 2013), discussing an extended concept of the Same-Origin Policy which robustly stops DNS rebinding attacks.

1.3 Related work

Considerable research has been conducted in almost all areas covered in this thesis. Rather than presenting several single results, this thesis presents a consecutive discussion of threats caused by Cross-Site Scripting, studies into the prevalence and nature of Client-Side Cross Site Scripting and, finally, a means of robustly stopping such attacks. Thus, we opt to not interrupt this discussion with a presentation of related work in each chapter.

Therefore, in the following, we present relevant related work on password manager security, the detection of Client-Side Cross-Site Scripting vulnerabilities and analysis of vulnerable JavaScript code as such. This is followed by work aiming at filtering Cross-Site Scripting as well as the general idea of utilizing taint tracking to prevent injection attacks.

1.3.1 Password Manager Security

Most research in the area of password managers focussed mainly on three different aspects: generating pseudo-random and unique passwords for each single Web application based one some master secret (Halderman et al., 2005; Ross et al., 2005; Chiasson et al., 2006), storing passwords in a secure manner (Zhao and Yue, 2013; Bojinov et al., 2010; Karole et al., 2010; Gasti and Rasmussen, 2012) and protecting users from phishing attacks (Wu et al., 2006; Ye et al., 2005).

The problem of weak password manager implementations with respect to their vulnerability towards Cross-Site Scripting attacks has been discussed by browser vendors since 2006 (O'Shannessy, 2006). However, researchers did not re-evaluate possibilities in terms of adopting new concepts to protect users from these kinds of attacks. In a recent blog post, Ben Toews again brought up the issue of password managers that were prone to XSS attacks (Toews, 2012). However, the question on how to improve the security of password managers remained unanswered.

Furthermore, Gonzalez et al. (2013) present an attack similar to the scenario we present in Chapter 3. They describe a network-based attacker that can inject code of his own choosing into any unencrypted HTTP connection. To leverage this, they injected multiple invisible frames into pages loaded by the victim and iterated through the login pages of different domains. They automated their attack using a self-developed tool called Lupin and were able to extract 1,000 passwords in 35 seconds from a victim's machine. In their follow up work, as a countermeasure, they block any JavaScript access to a password field once this has been filled by a password manager, therefore ensuring that the credentials can not be extracted in this manner (Silver et al., 2014).

1.3.2 Finding DOM-based Cross-Site Scripting Vulnerabilities

After its initial discovery by Amit Klein in 2005 (Klein, 2005), no large-scale studies have been conducted to ascertain the prevalence of DOM-based Cross-Site Scripting. The concept of FLAX, presented by Saxena et al. (2010b), is closest to the work we present in Chapter 4. The authors utilize byte-level taint tracking to discover insecure data flows in JavaScript. To achieve this goal, they translate the JavaScript code under investigation to an intermediary language denoted *JASIL* which expresses the operational semantics of a subset of JavaScript operations. This is contrary to our approach, which relies on taint tracking embedded into a real browsing engine, allowing us to achieve full language and API coverage. Apart from this difference, they do not aim to generate exploits in a precise manner. Rather, based on the observed data flows, they apply fuzzing to discover vulnerabilities.

Saxena et al. (2010a) also presented a concept of using symbolic execution to detect DOM-based XSS vulnerabilities. Given a URL, they automatically generate test cases to "systematically explore its execution space". Rather than using existing solvers, they propose a solver dubbed Kaluza which specifically aims at relevant JavaScript operations such as string modification or regular expressions. They apply their proposed tool Kudzu to 18 live Web applications, discovering two previously unknown vulnerabilities as well as nine additional flaws previously contained in a manually constructed test suite. Contrary to our approach, however, they do not conduct an empirical study on a large set of real-world Web sites.

The concept of using taint tracking in the browser to detect Client-Side Cross-Site Scripting was first pursued by DOMinator, developed by Di Paola (2012). Similar to our approach, Di Paolo enhanced the underlying JavaScript engine — SpiderMonkey inside Firefox — to allow for tracking of tainted data throughout the execution of a JavaScript program. His work, however, does not rely on a byte-level approach. Instead, DOMinator employs a function tracking history to store the operations which were called on the original, tainted input, transforming it into the final, still tainted string flowing into a security-critical sink. Therefore, contrary to our presented work, this scheme does not allow for the automated generation of exploit candidates and, thus, cannot be used to detect Client-Side Cross-Site Scripting at scale.

Another approach which tries to detect DOM-based Cross-Site Scripting at scale was presented by Criscione (2013). Criscione employs a fuzzing scheme inside a large number of Chrome browsing instances to discover such flaws. Rather than generating a precise payload, the goal here is to cause a parsing error which is caught by the JavaScript debugger. If such an error occurs, the author assumes that an exploitable flaw exists. His work also only aims at detecting such flaws in Google products and, thus, no information is provided on how well this approach worked on real-world applications.

A special class of flaw related to DOM-based XSS was investigated by Son and Shmatikov (2013). In their work, they focus on the newly added postMessage API of HTML5, which allows Web pages to communicate across domain boundaries when loaded in the same browsing instance. The API provides the developers with a secure way of determining the origin of such a message. The authors, however, analyzed postMessage receivers from the Alexa top 10,000 domains and found that "many perform origin checks incorrectly or not at all", allowing them to exploit vulnerabilities on 84 popular sites. Doing so, they were able to conduct Cross-Site Scripting attacks and inject arbitrary content into local storage. As our analysis in Section 4.3.5 shows, these types of message are rarely encoded or sanitized and, thus, might cause even more issues on the Web.

1.3.3 Analysis of Vulnerable JavaSript Code

In our work, we present an analysis of real-world vulnerable JavaScript code responsible for a number of Client-Side Cross-Site Scripting flaws. While no work has focussed specifically on such flaws, previous research has been conducted in the area of analysis of JavaScript errors.

Yue and Wang (2009) have conducted an empirical study aimed at characterizing insecure JavaScript practices on the Web, focussing on JavaScript inclusion and dynamic generation of JavaScript and HTML code. To achieve this goal, they instrument a Web browser to execute all discovered JavaScript code and record execution trace information, which can be analyzed offline. In their study, they found that 96.9% of all Web pages use JavaScript and that 44.4% of the analyzed Web pages utilize `eval` to generate code at runtime. In addition, they discovered that Web developers tend to use `document.write` and `innerHTML` rather then their secure equivalent `createElement`. Furthermore, their work revealed that 66.4% of the investigated Web applications include third-party script content, thereby increasing the attack surface.

The inclusion of third-party content is also the focus of *You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions* (Nikiforakis et al., 2012). The authors studied the Alexa top 10,000 domains for relations between the sites and third-party content, also monitoring the evolution of included script content. In doing so, they discovered that the amount of domains from which third-party content is included has risen by more than 400% in the time between 2001 and 2010 and that the Alexa top 10,000 domains include content from 2,249 unique domains. In several cases, the domain names from which the content was to be included contained typos, allowing an attacker to register these mistyped domain names and host malicious script code on them, thereby providing him with the equivalent of a stored Cross-Site Scripting flaw. In addition, they discovered that more than every fourth included script uses `document.write` or `eval` and, thus, may introduce Client-Side Cross-Site Scripting vulnerabilities.

Other studies have focussed more on the use of `eval` in Web applications. Richards et al. (2011) present different strategies to explore the JavaScript code contained in applications and find that "between 50% and 82% of the most popular websites used `eval`". They find that the most common usage for `eval` is the parsing of JSON, although the `JSON.parse` method provides an equivalent, yet secure functionality. In addition, they identify a number of patterns for the use of `eval` and provide functionally equivalent, but secure alternatives to use in Web applications. Similar to this work, Jr. et al. (2011) studied the reliability of JavaScript Web 2.0 applications in order to identify common errors. They found that regardless of the matureness of an application, the same categories of errors were discovered on all analyzed pages, but that there was no direct correlation between the use of `eval` and the encountered errors. They did, however, find evidence to suggest that more complex pages (such as news sites) are prone to contain more errors, i.e., determining that code complexity is correlated to the number of discovered flaws.

In 2011, Guarnieri et al. (2011) discussed their tool Actarus which uses static analysis techniques to identify flawed JavaScript code. In their study, which analyzed 9,726 Web pages, they discovered 526 vulnerabilities contained in eleven domains, which allow for JavaScript execution if exploited. To rid an application of insecure uses of `eval`, Meawad et al. (2012) present their work dubbed Evalorizer. Based on the patterns detected by Richards et al. (2011), they rewrite JavaScript code in a semi-automated manner, replacing `eval` with functionally equivalent constructs. To do so, they first instrument `eval` to log all calls to it in a proxy. Secondly, based on the gathered information on the strings passed to `eval`, they patch the discovered code. In total, they were able to successfully replace 97% of the encountered `eval` calls, incurring virtually no overhead when running the patched code.

1.3.4 Filtering and Mitigating Cross-Site Scripting

Researchers have spent considerable time on ways of detecting, stopping or at least mitigating Cross-Site Scripting exploits. This work has been conducted both on the server (Ismail et al., 2004; Mui and Frankl, 2011; Bisht and Venkatakrishnan, 2008; Wassermann and Su, 2008; Louw and Venkatakrishnan, 2009; Johns et al., 2008; Jovanovic et al., 2006) as well as the client side. As our work is contained to client-side solutions, we limit the presentation of related work to client-side or hybrid filtering approaches.

One of the first client-side mitigation schemes was presented in 2006 by Kirda et al. (2006) in their work called Noxes. Rather than following a browser-based approach, Noxes act as a Web proxy which can be configured system-wide and, thus, protects the user while "requiring minimal user interaction and customization effort". In order to ensure protection, they use a combination of automatically generated and user-provided rules which are enforced in their proxy system. Due to the fact that it is located on the network layer, they are not able to provide protection against Client-Side Cross-Site Scripting, in which the payload does not traverse the server.

In their work *Regular expressions considered harmful in client-side XSS filters*, Bates et al. (2010) analyze the concept employed by Internet Explorer's Cross-Site Scripting filter. Doing so, they find that the used approach comes with several pitfalls such as a high performance overhead, circumventable protection when using UTF-7 characters and even vulnerabilities which are *caused* by the filter. To tackle these issues, they propose a new design for a Cross-Site Scripting filter, which is inlined during HTML parsing and allows for "high performance and high precision". In contrast to previous approaches, which used regular expressions to detect and block Cross-Site Scripting payloads, they hook into the parser to conduct checks only for dangerous elements, e.g., a `script` tag or an event handler. When such an element is discovered, they use string matching to determine if the payload was contained in the request and, if so, replace the payload with benign JavaScript code. We provide more details on the inner workings of the Auditor in Section 6.1. They evaluated the implemented filter — called the *XSS Auditor* — and found that in contrast to IE's filter, false positives are hard to induce and more importantly, no vulnerabilities can be caused by the filtering approach. The XSS Auditor is currently deployed in all WebKit- and Blink-based browsers such as Chrome or Safari, showing the maturity of the approach.

On the basis of the XSS Auditor and several bypasses they identified, Pelizzi and Sekar (2012) propose an extension of the aforementioned concept. Instead of searching for the payload in the request, they split the request up into its parameters and subsequently try to detect the parameter values inside the rendered document, i.e., determine if a *request parameter* is a substring of the *payload* rather than to check if the *payload* is a substring of the *request*. They evaluated their approach in comparison to both NoScript (Maone, 2007) and the XSS Auditor, finding that it does incur a higher number of false positives but decreases the number of false negatives. Their evaluation, however, has a major drawback: as they only analyze Web sites in the absence of an attacker, they do not account for induced false positives, which are easy to achieve by appending parameters that contain snippets of code contained in the legitimate response to a request.

A hybrid approach to a Cross-Site Scripting filter has been proposed by Nadji et al. (2009). Based on their attacker model, in which an attacker tries to inject elements or at least attributes into the DOM, they introduce the notion of *Document Structure Integrity* (DSI). To achieve its protection capabilities, the server-side code of DSI separates trusted and user-provided data and serializes the document's structure information for transfer to the client. In the client, they propose to patch the JavaScript and HTML parsers such that *quarantined* data, i.e., data which was provided by the user, does not cause changes to the document's structure. If any such structural change is detected, they abort the parsing process. Their approach, however, lacks a means of protecting against Cross-Site Scripting attacks which occur by insecure uses of `eval`, as these do not require a change in the document's structure.

1.3.5 Utilizing Taint Tracking to Combat Injection Attacks

The concept of using taint tracking has been applied by several research groups to thwart different types of injection attacks, such as SQL injection, buffer overflow attacks and Cross-Site Scripting. In the following, we will discuss the use this technique in previous research.

The Perl programming language has a built-in option that allows a programmer to attach taint to user-controllable input (Allen, 1989). Also, taint mode is enabled by default if a program is run under an effective user id different from the real user id. Whenever a security-sensitive function such as `system` is called, Perl checks whether the string originated from a user-controllable source, and, if so, refuses to conduct the insecure function call.

In their work DIGLOSSIA, Son et al. (2013) propose a byte-level taint tracking approach built into PHP to track the flow of user-provided data through a Web application to the SQL parsing engine. In the engine, they ensure that the user-provided data can not be interpreted as code, thereby enforcing that no SQL injection attack can occur. Rather than using additional storage to persist the taint information, they translate each character of user-provided data to a specific character set which is otherwise not used at runtime. This way, operations such as concatenation and substring access can be conducted without the need to patch the underlying PHP functions. Similar approaches to protect applications from SQL injections have been followed by Pietraszek and Berghe (2005), Halfond et al. (2006), Xu et al. (2006), Chin and Wagner (2009) and Papagiannis et al. (2011).

Rather than trying to prevent Cross-Site Scripting flaws from being exploited, Vogt et al. (2007) attach taint information to critical data, such as form input or cookies. They track the flow of such sensitive data throughout the application and patch all means which an attacker might use to leak this data back to him such that no tainted data may leave the client. While this allows them to thwart the extraction of secret information, it does not mitigate the risks of an attacker controlling the victim's browser, e.g., by posting on a social network or interacting with an application in any other way.

1.4 Outline

This thesis is structured into seven chapters. Chapter 1 presents the contributions and publications of this thesis, discusses related work, and outlines the remainder of the thesis. Following that, we give an overview over the technical background required for a complete understanding of this work in Chapter 2.

Subsequently, Chapter 3 highlights an attack scenario in which Cross-Site Scripting can be used to extract stored credentials from password managers. After an introduction to the concept of these attacks and browser-based password managers

themselves, we report on an analysis of both password managers and fields on the Web, finding that built-in password managers are prone to the outlined attacks and that Web sites often do not properly protect their login fields. Based on our findings, we then provide a secure and functionally equivalent concept to shield users from such attacks. In the following Chapter 4, we present the implementation of an infrastructure capable of detecting Client-Side Cross-Site Scripting at scale as well as the execution of an empirical study on the Alexa top 5,000 domains. In doing so, we find that almost every tenth domains carries at least one DOM-based XSS vulnerability. Based on this study, Chapter 5 presents an in-depth analysis of the discovered vulnerabilities by discussing and applying metrics aimed at quantifying the complexities inherent to these vulnerabilities. Chapter 6 then highlights drawbacks of currently deployed filtering approach and discusses the concept of a new, taint-aware Cross-Site Scripting filter, capable of robustly protecting users against DOM-based XSS. Finally, Chapter 7 discusses limitations and interesting paths for future work, summarizes the thesis and concludes.

Chapter 2

Technical Background

In this chapter, we lay the technical foundation for the remainder of this thesis. First, we introduce common Web technologies such as HTML, JavaScript and the interconnecting Document Object Model. We then follow with the discussion of the Web's principal security policy, namely the Same-Origin Policy. Based on these, we then discuss the concept of Cross-Site Scripting attacks, which aim at bypassing the Same-Origin Policy. In doing so, we discuss server- and client-side variants and highlight their differences. We conclude the chapter with an introduction into the concept of taint tracking.

2.1 Web Technologies

This section presents an overview over relevant Web technologies, namely HTML, JavaScript and the DOM API, which acts as an interconnecting component between HTML and JavaScript realm.

2.1.1 HTML and Browsers

In 1989, Tim Berners-Lee – at that time working for CERN – wrote a document called *Information Management: A Proposal*, outlining his idea on how to keep up with managing the large amounts of information acquired throughout CERN without losing any of said information (Berners-Lee, 1989). Three years later, his proposal was extended and enhanced, resulting in the first specification of the Hypertext Markup Language, or short *HTML* (Connolly, 1992). In this initial specification, HTML only consisted of a few tags, allowing the author of an HTML document to set titles for documents, use anchors to link to other documents and use lists. Over the years, HTML evolved to allow for inclusion of images, tables, forms and numerous other entities. In recent years, HTML5 has been proposed and partially implemented, adding powerful features such as video and audio elements and semantic elements like `header`, `footer` or `section` (Hickson et al., 2014).

HTML itself is a text-based markup language, i.e., it contains information which is parsed by a document viewer (more specifically, a *browser*) and subsequently rendered according to the markup language's rule set. Throughout the course of this thesis, we will refer to the rendering window as a *viewport*. In their implementations, modern browsers allow for different types of these viewports, namely complete

pages, popup windows and frames. A *frame* is an HTML element containing another document, whereas frames can be stacked on top of each other.

In the form specified by the W3C and WHATWG (Web Hypertext Application Technology Working Group, 2015), HTML as such is a *static* markup language. This means that a server sends a static document to the browser, which subsequently renders it according to the specification. As such, this does not allow for interaction of the user with the document itself other than submitting a form to the server or clicking a link. To allow for dynamic interaction with the document, JavaScript was proposed and implemented.

2.1.2 JavaScript

The most commonly used scripting language in today's Web applications is *JavaScript*. Studies have shown that almost 90% of the top ranked Web sites use JavaScript to enhance the user experience (W3Techs, 2015a; Richards et al., 2011). It was initially designed for and built into a beta version of Netscape Navigator 2.0 in 1995 and called *LiveScript* then. Its target use was to allow interaction with otherwise static HTML document, such as client-side verification of form inputs. Before Netscape Navigator 2.0 left its beta stages, the scripting language was renamed and hence forward called *JavaScript*. The term itself is nowadays trademarked by Sun Microsystems, whereas the name of the specification of the language is *ECMAScript*. Nevertheless, the commonly used term still is JavaScript and, thus, we will refer to it as such through this thesis.

JavaScript is a scripting language which is shipped with every modern browser. Even more so, it is nowadays also used to power server-side applications such as node.js (Joyvent, Inc., 2015). In the browser, it provides a programmer with ample means to execute complex programs. Although single-thread, it allows for a smooth execution of a Web page's code by employing an event-based concurrency model, making it well-suited for the deployment inside a Web browser. We will discuss more on the details of this model as well as other complexities inherent to both the programming language and the browser as an environment in Chapter 5.

2.1.3 Document Object Model

On its own, JavaScript is not a very powerful language to be used in the browser. In order to achieve its designated task, i.e., interact with the document, the need for an API capable of accessing the rendered document occurs. Therefore, browsers implement the so-called *Document Object Model* (*DOM*), which serves as just this API between JavaScript and the document. When an HTML document is parsed, it is stored as a tree-like structure in the browser. The DOM provides functionality that allows JavaScript to access and modify this structure as well as additional properties such as cookie belonging to the current domain.

URL	access	reason
`http://example.org:8080/page1.html`	✗	port mismatch
`https://example.org/page1.html`	✗	protocol mismatch
`http://sub.example.org/page1.html`	✗	domain mismatch
`http://example.org/page2.html`	✓	path not considered

Table 2.1: Same-Origin Policy in relation to `http://example.org/page1.html`

By combining the three components, namely the HTML rendering, the DOM API and the JavaScript runtime engine, a programmer can easily create dynamic Web applications, such as Facebook or Twitter. These powerful features, however, need to be contained such that no program may interact with other pages, rendered in the same browser. Therefore, the need for isolation emerges, which today is ensured by the Same-Origin Policy, which we will outline in the following.

2.2 The Same-Origin Policy

With upcoming dynamic client-side technologies such as Java and JavaScript, the need for protection between different applications arose. Although not using the exact term, Netscape built a security policy into their Navigator 2.0, allowing only resources of the same origin to interact with each other (Zalewski, 2012). The exact source of the term *Same-Origin Policy* is unknown and browser vendors implemented different concepts of origins for different parts of their engines. The W3C summarizes this in their wiki, stating "There is no single same-origin policy". The concept of an origin was only formalized by Adam Barth in 2011 under the RFC 6454, denoting that "two URIs are part of the same origin [..] if they have the same scheme, host, and port" (Barth, 2011). This policy is enforced in both major browsing components, namely the HTML rendering and the JavaScript engine.

The basic concept of the *Same-Origin Policy* for JavaScript is straight-forward: only resources with the same origin, defined by the protocol, the domain and the port, may interact with each other. Table 2.1 shows the different scenarios which require checking by the Same-Origin Policy. If no port is explicitly provided, HTTP uses port 80 to establish a connection. Since the first listed resources is located on port 8080, the ports mismatch and, thus, access is restricted. Similarly, the second URL points to an HTTPS-hosted document, i.e., the protocol (as well as the port) mismatch. In the next case, the domains do not match, although the shown URL is a sub domain of `example.org`. Finally, for the last shown resource, access is granted although the path differs, since this is not part of the Same-Origin Policy's decision process.

The *Same-Origin Policy* therefore implements protection to resources of applications that do not share the same boundaries (as denoted by their origin) and should

therefore by definition not trust each other. There are, however, options to relax this strict policy, allowing interaction even if the involved resources do not match. Among these are *domain relaxation* (Zalewski, 2012), a way for domains located under the same second-level domain to relax their origins to just this second-level domain. To opt-in to this relaxed Same-Origin Policy, both involved documents must explicitly set their `document.domain` property. In these cases, access is granted across the sub-domain boundary. In addition, HTML5 introduced the concept of postMessages (Hickson, 2015), which allow documents rendered in the same browser to exchange message across domain boundaries. This allows, if used correctly, for a secure means of exchanging data between these documents while not providing full access to the other document, respectively.

In addition to the Same-Origin Policy for JavaScript interaction and DOM access between documents, other types of Same-Origin Policies exist for XmlHttpRequests (van Kesteren et al., 2014) or Flash (Zalewski, 2009b). For XmlHttpRequests, the policy can be relaxed using *Cross-Origin Resource Sharing* (van Kesteren, 2014) to allow cross-domain communication. Similarly, Flash provides a mechanism which governs HTTP connections across domain boundaries, for which a Web site may host a cross-domain policy file (Lekies et al., 2011), specifying which remote host may connect to the target server.

Summarizing, the Same-Origin Policy for DOM access provides a basic security measure that isolates mutually distrusting Web applications from each other, i.e., it ensures that code hosted on an attacker's site can not access sensitive data from a different site, even though both sites are rendered in the victim's browser at the same time. Similarly, other such policies exist for XmlHttpRequests as well as third-party plugins, such as Silverlight or Flash.

2.3 Cross-Site Scripting

The term *Cross-Site Scripting* was first used in a CERT report in 2000 (Ross, 2009), to describe an attack in which an adversary is capable of injecting either HTML or script code of his choosing into a Web site. Essentially, this type of attack constitutes a bypass of the Same-Origin Policy since the code injected by the attacker is used to gain access to resources which would normally not be accessible to it. More precisely, the attacker's code is injected into the context and, thus, origin of the vulnerable Web application and can fully interact with the flawed domain (see above). The term Cross-Site Scripting is often abbreviated as *XSS* rather than the apparently more logical CSS. This, however, is done to ensure that no confusion can occur between Cross-Site Scripting and Cascading Style Sheets (Etemad, 2011).

Over the years, three main forms of this attack have presented themselves. In the following, we will briefly outline types of Cross-Site Scripting which are caused by server-side vulnerabilities and will follow this with a presentation of vulnerabilities caused by client-side vulnerabilities. Although literature (Cook, 2003; Kirda

et al., 2006) usually refers to all client-side vulnerabilities as *DOM-based Cross-Site Scripting*, we believe that a distinction must be made between non-persistent and persistent vulnerabilities on the client side as well, since, e.g., the Web Storage API provides a means of persistently storing data on the client, which may eventually cause a vulnerability. Important to note in this is the fact that – regardless of whether the vulnerability is caused by server-side or client-side code – Cross-Site Scripting ultimately leads to execution of attacker-chosen JavaScript code in the victim's *browser* and under the origin of the vulnerable Web site.

2.3.1 Server-Side Cross-Site Scripting

On the server side, we distinguish between two types of Cross-Site Scripting, namely non-persistent and persistent XSS, which we will discuss in the following.

Non-persistent Server-Side Cross-Site Scripting As the name of this type of Cross-Site Scripting, which is also often referred to as *Reflected Cross-Site Scripting*, correctly suggest, this variant does not persistently store the attacker's XSS payload on the server. In contrast, vulnerabilities of this kind are caused by Web pages which consume user-provided input (such as a GET parameter) and subsequently *reflect* or echo it back to the user. Once an attacker has discovered such a vulnerability in a Web service and built the appropriate payload, he has different options of luring victims to the crafted URL. The first means to do so is by sending the resulting URL to his victims, e.g., by e-mail spam. This, however, requires the victim to click on a link containing the payload, which might be too obvious. The second way of attacking his target is to lure them to a seemingly benign Web site under the control of the attacker, and including a hidden `iframe` pointing to the crafted URL.

The stages of this attack are outlined in Figure 2.1. Initially, the attacker ensures that the victim visits the crafted URL as discussed before. In the second step, the victim's browser loads the vulnerable Web page, including the attacker-chosen payload as part of the URL. The server then reflects the user-provided (and attacker-chosen) input back to the victim, whose browser ultimately executes the malicious code. Depending on what the attacker wants this code to accomplish, it might either extract sensitive information from the victim's browsing session on the target site or interact with the site in the name of the victim.

This form of Cross-Site Scripting requires some form of targeting by the attacker. He can abuse such vulnerabilities only if the victim either clicks on an attacker-provided link or visits an attacker-controlled Web site.

Persistent Server-Side Cross-Site Scripting In contrast to the previously mentioned, non-persistent Cross-Site Scripting flaws, persistent XSS (or *Stored Cross-Site Scripting*) vulnerabilities occur when attacker-provided data is stored in the

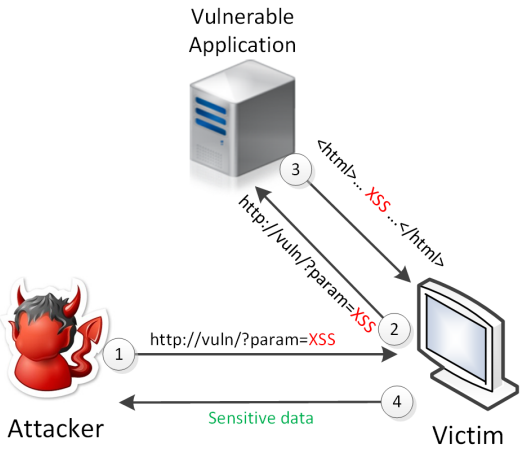

Figure 2.1: Stages of a Reflected XSS attack

target application. Common examples for such vulnerabilities include forum posts, news comments or even attacker-chosen usernames for arbitrary services.

Figure 2.2 shows the way in which such an attack occurs. In the first step, after having identified a vulnerable target application, the attacker sends his malicious payload to be stored in that service. In an optional second step, the attacker then tries to make his victims visit that page. Note, that this step can be omitted, as the stored payload will be sent to any user of the vulnerable Web service. Subsequently, any victim visits the Web service which now—rather than just serving the content as intented by the Web page's administrator—also contains the attacker-injected payload. Therefore, this code is executed in the victim's browser, potentially leaking back sensitive information to the attacker.

The most notable differences between reflected and persistent Cross-Site Scripting attacks are the fact that persistent XSS attacks can easily reach a wider range of victims and, more importantly, the victim has no way of discerning the attacker-provided code from the code that actually is part of the application. In cases of reflected Cross-Site Scripting attacks, this is part of the *request* which is sent out by the victim's browser. Since the injected payload is not part of the request in persistent XSS attacks, it is almost impossible for purely client-side based XSS filters to detect and stop such attacks.

2.3.2 Client-Side Cross-Site Scripting

Next to server-side Cross-Site Scripting vulnerabilities, a second class of XSS flaws exist: *Client-Side Cross-Site Scripting*. Initially formalized by Klein (2005), these types of Cross-Site Scripting have also become known under the then-coined term

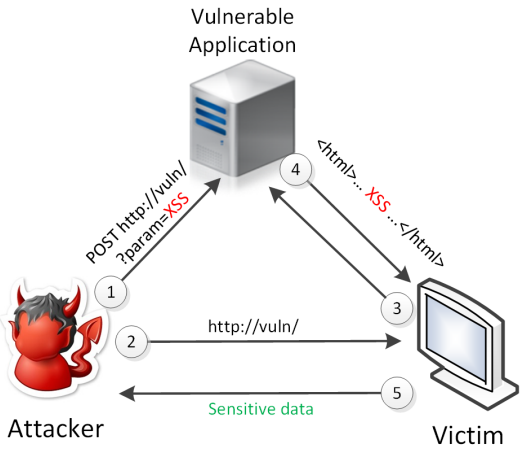

Figure 2.2: Stages of a Stored XSS attack

DOM-based Cross-Site Scripting, and subsume all types of vulnerabilities that are caused by client-side rather than server-side code. Similarly to server-side functionality which allows for dynamic generation of parts of the response based on user-provided inputs, client-side JavaScript allows a programmer to dynamically change a page's content using either `document.write`, `innerHTML` (or its derivatives) as well as execute dynamically generated JavaScript code using `eval`.

While this functionality allows for a great deal of flexibility on the part of the Web programmer, improper use of user-provided data in conjunction with these *sinks* can lead to Cross-Site Scripting vulnerabilities. Listing 2.1 shows an example of such a vulnerability. The purpose of this snippet is to simply inform the user of the current URL by writing it to the document. Although the example is fabricated, we encountered real-world examples similar to this, especially in conjunction with documents indicating that a URL was not found (see also Section 5.3.5).

```
document.write("The current URL is " + location.href);
```

Listing 2.1: Example of a Client-Side Cross-Site Scripting vulnerability

An attacker can abuse this flaw in a straight-forward manner: the used property `href` of the `location` object contains the URL of the page which is being visited, including the hash fragment. In Internet Explorer and Chrome, this is not automatically encoded when retrieved from the DOM (Zalewski, 2009a). Therefore, the attacker can lure his victim to the URL `http://vuln.com/#<script>alert(1)</script>` to trigger execution of the `alert` function (or any other arbitrary JavaScript func-

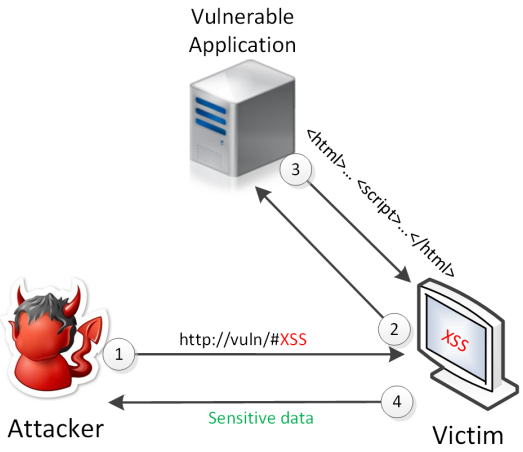

Figure 2.3: Stages of a Client-Side XSS attack

tion of his choosing), as highlighting in Figure 2.3 (1). In addition to the benefit that the fragment is not encoded, it is also not sent to the server when the client conducts the request (2). This way, the injected payload is not visible in the server logs, severely limiting the chances of attack detection by the server administrator. In the stage (3), the vulnerable code is retrieved from the server. We assume that the code shown in Listing 2.1 is now executed in the browser of the victim. Rather than simply opening an alert box, the attacker may now use this to leak data, such as session cookies, back to his server (4).

Apart from sinks like `eval` or `document.write`, which provide an attacker with a means of directly injecting JavaScript code or HTML markup, the assignment of attacker-controllable values to security-sensitive DOM properties, such as a `script` tag's `src` property also fall into the category of Client-Side or DOM-based Cross-Site Scripting. In the specific case of `script.src`, an attacker can retrieve script code from a server of his choosing without having to inject any HTML markup.

Although academia typically only distinguishes between three kinds of Cross-Site Scripting, namely persistent, reflected and client-side XSS, we believe that an additional distinction must be made between reflected and persistent Client-Side Cross-Site Scripting.

In the sense of Client-Side Cross-Site Scripting, additional security-sensitive sinks exist, which — given the right circumstances — can be used by an attacker to inject his malicious payload. Among these are assignments to cookies or the Web Storage API (World Wide Web Consortium, 2013). The latter provides a means of storing large amounts of data on the client. As Lekies and Johns (2012) observed, this feature can be used to store code on the client, which is later passed to `eval`. Thus, in cases where user-controllable input is provided to the Web Storage and

subsequently passed to `eval`, an attacker can leverage this to conduct a persistent client-side XSS attack. Similarly, cookies provide Web developers with a means of storing (albeit smaller amounts of) data on the client. Thus, we consider these types of exploitable flaws persistent Client-Side Cross-Site Scripting vulnerabilities.

Sources and Sinks As already mentioned, several sinks for a vulnerable flow of attacker-controlled data exist. The DOM XSS Test Case wiki (Di Paola, 2010) lists several vectors for data which might be controllable by an attacker, such as the URL, the name of the current window, cookies or the referrer. Albeit not all of these are easily controllable, they all provide a means for an attacker to potentially introduce data of his choosing to a Web application. Therefore, we initially consider all these *sources* in the sense of Client-Side Cross-Site Scripting.

Next to the *sinks* such as `innerHTML`, `eval` or the Web Storage API, other APIs exist in the JavaScript and rendering engine which, if provided with attacker-controllable data, may lead to undesirable results up to code execution. Although these sinks do not directly lead to code execution, they may be abused to enforce changes in the state of the application, which may eventually lead to a Cross-Site Scripting vulnerability. Therefore, in the following, we will refer to all such functions as sinks.

2.4 Taint Tracking

The term *taint tracking* refers to a technique which is used to track the flow of data throughout the execution of a program (Schwartz et al., 2010). The process can conceptually be split up into three stages: taint introduction, taint propagation and taint checking.

In order to track the flow of a piece of data, such as a string, throughout the program, the object has to be *tainted*. Depending on the implementation and context of taint tracking, this can either be done automatically, e.g., by tainting any data which originated from a `read` system call (Bosman et al., 2011), or by manually tainting it. Taint can be stored in different granularities, e.g., using only a single bit to mark a string as tainted or employing a per-character tainting approach.

The next important aspect of taint tracking is the propagation of taint. Again, depending on the context, different strategies can be applied to allow for tracking of explicit and implicit flows (King et al., 2008). For explicit flows, especially in regards to strings, only functions which modify strings must be patched. In contrast, implicit flows, such as the assignment of a variable depending on whether a certain character is upper or lower case, require a different strategy. For an XSS attacker trying to inject his malicious code into an application, only explicit flows have to be considered. Any protection or detection approaches are highly dependent on a sound implementation of taint propagation, since loosing the taint would undermine these undertakings.

Finally, the last pillar of taint tracking is *taint checking*. In the scenario of a Cross-Site Scripting attack, which we covered in the previous sections, taint checking must be performed whenever a string is passed to a security-sensitive sink. Similarly, in the realm of binaries, taint checking is often conducted when assinging the instruction pointer (Bosman et al., 2011). The actions, which are taken when the taint check confirms the presence of tainted data, are manifold, as we will discuss in Chapters 4 and 5.

2.5 Summary

In this chapter, we covered the technical background relevant to this thesis. First, we introduced the Web technologies HTML and JavaScript as well as the bridge between them, namely the Document Object Model. We then discussed the principal security policy governing all client-side interaction, the Same-Origin Policy. Subsequently, we introduced the term Cross-Site Scripting and outlined its server- and client-side variants, specifically discussing the sources and sinks relevant for Client-Side Cross-Site Scripting. Finally, we presented the concept of taint tracking, which we will employ in Chapters 4, 5 and 6 for the detection, in-depth examination and blocking of Client-Side Cross-Site Scripting exploits, respectively.

Chapter 3

XSS in Action: Abusing Password Managers

The most commonly known attack scenarios for Cross-Site Scripting vulnerabilities are the extraction of session credentials (typically in the form of cookies) as well as the interaction with the application in the name of user, e.g., posting content in a social network. However, XSS allows for injection of arbitrary JavaScript and HTML, thus allowing an attacker to change the vulnerable Web site in a manner of his choosing. This can potentially be abused to conduct phishing attacks, without typical indicators such as an odd domain (Dhamija et al., 2006). While this type of attack still requires the user to insert his credentials before they can be retrieved by the attacker, password managers can be used to steal credentials without the user's knowledge.

In the following, we outline how password managers' features can be leveraged to retrieve stored passwords. After a discussion of the general attack pattern, we investigate the proneness of the current generation of browser-based password managers to such attacks. Finally, we present an extension of the concept currently employed by password managers which is not susceptible to such attacks.

3.1 Attacking Browser-based Password Managers

In this section, we first introduce the implementation of the current generation of browser-based password managers. After that, we discuss the general attack pattern usable for stealing passwords via Cross-Site Scripting vulnerabilities and follow up with means of leveraging password managers to automate these kinds of attacks. Afterwards, we give an overview of specific attack scenarios aiming to extract credentials from password managers.

3.1.1 Functionality of a Password Manager

As studies by Ives et al. (2004) and Mazurek et al. (2013) have shown, users tend to choose bad passwords and/or reuse passwords over multiple sites, therefore undermining the security of their login credentials. To support users in employing a more secure password strategy, browser as well as third-party vendors have implemented password managers capable of storing these secret credentials for the users. This allows users to choose more complex and possibly random passwords by lifting the burden of remembering numerous complicated passwords. Hence, password

managers can be beneficial for supporting better security practices in password handling.

Current implementations of password managers in browsers all work in a similar manner. Just before a form is submitted, the form is checked for password fields. If any such field exists, the username and password fields are determined and their values are subsequently extracted. These extracted credentials are then – along with the domain they were entered into – passed to the password manager. The password manager's database is subsequently checked for a matching entry, whereas no action is taken if the extracted credentials already match the stored ones. If, however, no matching entry is found, the user is prompted to approve storing of the password data. Analogously to that, if an entry for the same username but different password is found, the user is prompted to consent to updating the stored data. This process only works with forms that are submitted, either by the user clicking a submit button or by JavaScript invocation of the `submit()` method of that form. According to Mozilla (Dolske, 2013), storing passwords which are sent using JavaScript XMLHttpRequests is not supported since no actual submission of the form takes place.

In turn, if the user opens a page containing a username and password field, the password manager is queried for entries matching the URL (or domain, depending on the implementation). If an entry is found, the fields on that page are automatically filled with the previously persisted credentials. Hence, the user then only has to submit the form to log into the application.

3.1.2 Stealing Passwords with XSS

Cross-Site Scripting gives an attacker ample opportunity to steal secret data from his victim. Typically, login forms for Web applications are realized using two input fields, which the user fills with his username and password, respectively. By design, JavaScript may interact with the document and thus is also capable of accessing the username and password field. This feature is often used by applications to verify that certain criteria are met, e.g., ensuring that the username is a valid e-mail addresses. However, this functionality also allows an attacker to retrieve the credentials utilizing Cross-Site Scripting. If the attacker can successfully inject his own JavaScript code into the login page, that code can extract the credentials entered by the user and subsequently leak them back to the attacker. This kind of vulnerability obviously only works if the user is not yet logged in when clicking on a crafted link. However, this is where password managers come to the aid of the attacker, as we discuss in the following.

3.1.3 Leveraging Password Managers to Automate Attacks

Password managers provide a convenient way for users to automate parts of logins into Web applications. To make the login as simple and comfortable as possible,

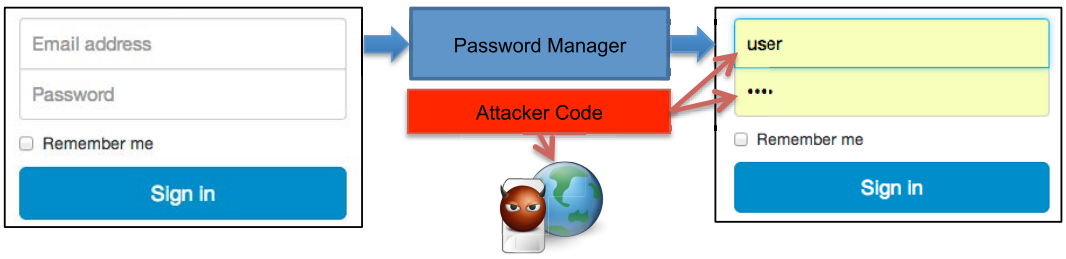

Figure 3.1: Process of leveraging the password manager to steal stored credentials

they automatically pre-fill forms for which the user stored the credentials beforehand. This feature can be exploited by a Cross-Site Scripting attacker since, if a site is susceptible to XSS attacks, the adversary can inject his own code into the application's login form in a similar manner as described earlier.

However, the injected code no longer needs to wait for the form filled by the user, since the password manager auto-fills the required fields. The attacker's code can then automatically retrieve the information and leak it back to the attacker. The biggest advantage in comparison to the aforementioned attack is the fact that the user does not need to be involved at all. This process can – depending on the browser – be fully automated in a hidden frame while the user is looking at a seemingly innocent page.

Figure 3.1 shows this process. First, when the login page is initially loaded, both the username and password fields are empty. When the password manager discovers these fields, it automatically fills in the username and password. Although the password is masked from the user using asterisks, the attacker's code can use the DOM API to retrieve the inserted credentials in clear text. The stolen credentials are then automatically leaked back to the attacker as depicted in the lower part of the figure.

3.1.4 Specific Attack Patterns

In terms of Cross-Site Scripting, an attack targeting password managers specifically aims at extracting the stored user credentials in an automated way. The attacker therefore tries to embed a form into a vulnerable application which is then filled in by the password manager. This form can afterwards be read by the attacker's JavaScript code to retrieve the data that was inserted by the victim's browser and eventually leak the information back to the attacker.

In our study of current password manager implementation, which we discuss in further detail in Section 3.2.1, we found that their behavior could be categorized in four dimensions. In the following, we discuss these distinct factors and the attack patterns associated with them.

Matching Requirements for the URL and Form The first factor we examined was the way in which password managers react to changes both to the URL and the form itself. Password managers often fill passwords regardless of the context, as long as the domain matches, and, potentially, other easily fabricated indicators such as field names and types or form action are present.

When a password manager does not explicitly store the complete form the credentials were stored for, but rather only the origin, an attacker can easily extract the credentials. To achieve this, he can abuse a Cross-Site Scripting vulnerability on an arbitrary part of the application to inject his form and corresponding JavaScript code. This form is then filled by the password manager and the stolen data can be sent to the attacker. In cases where a password manager also does not store the names of the fields the data was stored from, the attack is even easier since an attacker does not need to craft a form specifically mimicking the login page of the target application, but may use a generic form. This allows him to automate the attack for multiple vulnerable pages in a very simple manner.

Viewports If a password manager explicitly checks the URL rather than the origin, the attacker has to force the victim's browser to load the original login page to make the password manager fill out all the relevant fields. Hence, the second criterion we found is the difference in handling viewports. In our notion, a viewport can either be a top frame, a sub frame or a popup window. With respect to that, the interesting question is whether a password manager still fills out forms if they are not located in the top frame of a page.

If login field data is inserted regardless of the viewport, an adversary can place a hidden frame, pointing to the login page, inside the vulnerable document. As enforced by the Same-Origin Policy, any page may only access another document's content and resources if the protocol, the domain and the port of both involved documents match. As we assume the attacker has control over some page inside the vulnerable Web application, he can therefore access the aforementioned frame's content, thus enabling extraction of the credentials that were filled in by the password manager. If a password manager does not automatically fill in the values of interest to the attacker, or the application itself forces not to be framed using the `X-Frame-Options` header (Microsoft, 2009) or the Content Security Policy `frame-ancestors` directive (World Wide Web Consortium, 2015), the login page can be opened in a popup window. Still operating under the assumption that vulnerable and login page are of the same origin, the attacker's code can retrieve the data from the opened popup.

User Interaction As a third distinguishing feature of the examined password managers, we identified user interaction, i.e., whether the user has to somehow interact with the password manager before it fills out the forms, e.g., by clicking or typing into the password field. If a given password manager requires such interaction, fully automated XSS password stealing attacks are not feasible.

However, in such cases, an attacker can attempt to conduct a ClickJacking (Rydstedt et al., 2010) attack. ClickJacking attacks work by tricking the user to interact with a security-sensitive Web UI without his knowledge. In the general attack case, the adversary loads the document which contains the security-sensitive UI into an iframe and hides the frame from the user's eyes via CSS properties, such as `opacity`. Subsequently, he overlays the targeted (and now invisible) UI elements with unsuspicious elements and motivates the user to click them, for instance in the context of a game or a competition. If the user falls for the adversary's bait, he involuntary interacts with the hidden UI.

Using this attack, the adversary can trick the victim to interact with the password field in the required fashion, thus, causing the password manager to fill the field with the stored value.

Adherence to the Autocomplete Attribute The fourth and last dimension we found was the adherence to the `autocomplete` attribute for fields. According to the W3C standard (Hickson, 2005), a browser must not store data that is inserted into input fields which have `autocomplete` set to `off`.

From the attacker's point of view, this feature is very interesting. If a password manager does not respect the `autocomplete` value when *storing* the credentials but only when later *filling out* the input fields, it is still susceptible to attacks. In order to extract password data from clients, the adversary can simply add a second form with the same names and input types to the document, this time without the `autocomplete` attribute, which is then filled with the persisted credentials.

After having outlined the four dimensions in which a password manager's behavior can be categorized, the following section discusses in detail how the browser we examined behaved with respect to these dimensions.

3.2 Exploring the Password (Manager) Landscape

As explained above, several potential XSS attack patterns on password managers exist. To examine the degree to which these theoretic attacks are applicable with the currently deployed password managers and Web sites, we conducted two comprehensive studies. For the first, we systematically examined the built-in password managers of the current browser generation. Furthermore, we conducted a large scale study on how password fields are used by existing, real-world Web applications.

3.2.1 Password Managers

In this section we present the results of our experiments on the behavior of different modern browsers. Our tests were aimed in the four different dimensions previously discussed in Section 3.1.4.

To ensure a broad coverage of internet users, we opted to examine Google Chrome (version 31), Mozilla Firefox (version 25), Opera (version 18), Safari (version 7), Internet Explorer (version 11) and the Maxthon Cloud Browser (version 3). Although the latter one might not be as well-known as the other candidates, it is one of the options that is shown to users installing the latest Windows versions. Hence, we looked at the behavior of this browser along with the previously named.

Before investigating the behavioral changes when tampering with the form or the URLs the form was located in, we first analyzed the general fill-in behavior of our test subjects according to the specific attacks discussed in Section 3.1.4.

Filling only in the Top Frame To assess whether password managers would fill out forms only in top frames, we created a page that framed the original, unchanged login page we had initially stored our credentials for. Apart from Internet Explorer, which refused to insert any data, all browsers filled in the username and password field.

Explicit User Interaction Next, we investigated whether a browser would actually require any interaction from the user to fill in passwords. Again, Internet Explorer was the positive outlier, being the only browsing engine that required any form of interaction. In Internet Explorer, the user has to manually put the focus to the username field and is then presented with a dropdown menu allowing him to select which credentials he wants to insert. The user then has to explicitly click on an entry to trigger the browser's fill-in action. Also, this is done properly outside of the DOM, thus the ClickJacking attacker discussed in Section 3.1.4 can also not force the filling of password fields.

URL Matching We assume that the attacker wants to steal the credentials from his victim in a stealthy manner. We consider the following example: an application hosts its login at `/login`. The attacker has found a XSS vulnerability at `/otherpage` which he wants to abuse to steal the stored credentials. Hence, if a password manager only supplies the password to the exact URL it stored the passwords for, the attacker would have to open a popup window or embed a frame to the login page to steal the secret data. However, opening a popup window is very suspicious and therefore not desirable. Also, framing the login page in an invisible frame might not work due to `X-Frame-Options` headers. In our study, which we discuss in Section 3.3.3, we found that only 8.9% of login pages make use of this header to ensure that they are not framed. Thus, in our work, we wanted to determine how easy it was to make password managers fill in the stored credentials into forms if the URL did not match the one the password was originally stored for. To examine the browsers' behaviours, we created a simple Web application with a login form. We visited this login and let the password manager under investigation save the credentials that we entered. We then

created multiple other pages running under different protocol (HTTP vs. HTTPS), different ports, different (sub-)domains as well as changing paths to determine what the implemented matching criteria were for all our test subjects. In the following, we discuss the results of the analysis of the aforementioned browsers.

- *Google Chrome:* Our tests showed that changing the protocol, sub domain or port lead to the password to not be filled in anymore. In contrast, when visiting a form running under a different path, Chrome still inserted the stored credentials. This leads us to reason that Chrome stores the password alongside their origin in the sense of the Same-Origin Policy, namely the tuple of protocol, domain and port, which is also supported by the source code (Chromium Developers, 2009).
- Our second candidate was *Firefox*. Similar to the behaviour Chrome exhibited, Firefox also refused to fill out login fields if either protocol, (sub-)domain or port were changed. It also behaved in a similar manner to Chrome with respect to a change in the path – still automatically setting the username and password fields to the stored values.
- Both, *Opera and Safari* behaved in a similar manner. With changed origins, they refused to fill out forms, whereas the path was not taken into consideration in the decision whether to insert the stored credentials or not.
- *Internet Explorer:* In contrast to all the aforementioned, Microsoft's Internet Explorer apparently stores the complete URL of the form it saved the password data for. In our tests, it showed to be the only browser that did not insert stored credentials even if only the path changed.
- *Maxthon Cloud Browser:* Most interestingly, in this browser the passwords were apparently only stored coupled with the second-level domain they stemmed from. In our tests, the browser would still fill in password fields even if the protocol, sub domain, port or path changed.

Summarizing, our tests showed that out of the most commonly used browsers on the Web, all but Internet Explorer gladly fill forms on any part of the same Web application, whereas the application borders are determined by the Same-Origin Policy. The Maxthon Cloud Browser even fills in credentials if only the same second-level domain is visited – ignoring both the protocol and the port of resource – making it even easier for an attacker to extract the passwords from its storage.

Form Matching After having examined how browsers treat changes in the URL with respect to their password managers, we analyzed what kind of information browsers would store on the actual form. To gain insight into this, we built another set of test pages – this time with different modifications to the login form itself. Our test pages were different from the original form in several aspects, which we discuss briefly in the following.

For the first test case, we removed the action and the method of the form. Our second modification was the removal of the names of all fields in the form, whereas the third change was to only change the names of all fields rather than removing them. For the next part of our analysis, we removed the types from all fields, essentially resetting them all to `type=text`. We then derived a minimal form as shown in Listing 3.2, only consisting of two input fields with random names, no action or method as well as no additional submit buttons. After these changes to the form fields, we build a final testing page, setting the `autocomplete` attribute for the *password* field to off. According to the W3C specification (Hickson, 2005), this value indicates the browser should neither store the data inserted into that field nor automatically fill it in later.

```
<form>
<input name="random1">
<input name="random2" type="password">
</form>
```

Listing 3.2: Minimal HTML form used in our tests

Utilizing the different created forms, we now discuss the matching criteria with respect to the structure of the form presented to the password manager.

- *Google Chrome:* We observed that neither action nor method of the form were criteria in the decision, whereas the same held true for changes to the names of the fields we provided. However, if we presented Chrome with fields without any name, it would not provide the credentials to the form. Chrome did not strictly adhere to the autocomplete setting of the password field, prompting the user to save the password nonetheless. It did however adhere to the setting when inserting the password into the form – nevertheless, we could extract secret data by adding a second form, as described in Section 3.1.4. Since the matching is done on a structural basis, the minimal form shown in Listing 3.2 was sufficient for this attack.
- *Firefox* also only performed matching against a form's structure, not the content itself. In contrast to what we had seen with Chrome, Firefox did, however, also insert credentials into forms that only contained input fields without names. Also unlike Chrome, Firefox adhered to the `autocomplete` attribute – if either field had this set to off, Firefox would not store any data. Due to these factors, injecting the minimal form would still trigger the auto-fill functionality of Firefox's password manager.
- *Opera and Safari* again behaved alike, filling in passwords into the minimal form but not into forms containing only input fields without names. On our test

machine, a Macbook running OS X Mavericks, we discovered that both Opera and Safari also use the OS X keychain to store their passwords. Thus, after having stored a password in Opera, Safari automatically filled out our fields although we had not previously stored a password in its database. While Opera – similar to Chrome – also offered to store passwords if at least one of the posted fields did not have `autocomplete` set to off, Safari behaved like Firefox and did not save any data in that case. Again, the test subjects only performed structural rather than content matching, leading to both of them also auto-filling the minimal form. Contrary to Firefox, both browsers would not fill input fields without names.

- *Internet Explorer:* As explained in Section 3.2.1, Internet Explorer was the only browser that required any form of user interaction to fill in the passwords. To nevertheless check the functionality, we manually interacted with the browser to ensure that it would fill in the username and password. In that, we discovered that Internet Explorer applies matching criteria in the same manner as Firefox, namely inserting passwords even into forms containing only input fields without any name. In terms of adhering to the autocomplete attribute, Internet Explorer did respect the value by not saving any information if either field had the autocomplete value set to off.
- *Maxthon Cloud Browser:* Not unlike the insecure behaviour it showed regarding matching the URL, the Maxthon Cloud Browser was not at all strict in matching the form, even filling in input fields that had no name and – most notably – that had autocomplete set to off.

To sum up: Most browsers are very relaxed in terms of matching criteria. All but Internet Explorer would still fill in passwords if only the origins matched, whereas the Maxthon Cloud Browser even only took the second-level domain into consideration for its decision. Similar to that, matching against a form was mostly performed on a structural level, i.e. meaning that any two fields were filled out if the latter was a password. According to Mozilla (Dolske, 2013), this is done by design as a convenience feature. Looking at the results, the tests with different forms showed that the attacker only has to create a minimal form as shown in Listing 3.2 to trick the browser's password managers into providing the stored passwords from any site that uses two input fields for its login dialogue.

All the previously discussed results are depicted in Table 3.1, in which ✓denotes that the criterion must match. For the minimal form, ✓denotes that the minimal form was sufficient, whereas ✓for autocomplete means that the browsers would not save passwords if the autocomplete attribute was set to off.

3.2.2 Password Fields

To obtain a realistic picture on how password fields are currently used in practice and to which degree real-world password dialogs are susceptible to the attacks discussed

	port	path	sub domain	any name required	name match	input type match	min. form	auto-complete
Chrome 31	✓	✗	✓	✓	✗	✓	✓	✗
IE 11	✓	✓	✓	✗	✗	✓	✓	✓
Firefox 25	✓	✗	✓	✗	✗	✓	✓	✓
Opera 18	✓	✗	✓	✓	✗	✓	✓	✗
Safari 7	✓	✗	✓	✓	✗	✓	✓	✓
Maxthon 3	✗	✗	✗	✗	✗	✓	✓	✗

Table 3.1: Overview of tested browsers and their matching criteria

in Section 3.1, we conducted a survey on the password fields of the top ranked Web sites according to the Alexa index (Alexa Internet, Inc., 2015).

Methodology To locate and analyze password fields in real-world Web sites, we conducted a lightweight crawl of the top 4,000 Alexa sites. As many modern sites rely on client-side markup creation and DOM manipulation via JavaScript, we chose a full-fledged browser engine as the technical foundation of our crawling infrastructure: We implemented an extension for the Chrome browser, that pulls starting URLs from a backend component, which are subsequently visited by the browser. This way, we can not only examine the same final DOM structure that is also presented to the browser, but this also gives us the opportunity to observe client-side actions after a password has been entered (more on this below). Our Chrome extension consists of the following JavaScript components:

- A single *background script*, which is able to monitor network traffic and distribute the crawling process over multiple browser tabs.
- Multiple *content script* instances, one for each Web document that is rendered by the browser. A content script is instantiated as soon as a new document is created by the browser engine. This script has direct access to this document's DOM tree. However, the script's execution context is strictly isolated from the scripts running in the document.
- Thus, the content script injects a *user script* directly into the page's DOM. Unlike the content script, which is cleanly separated from the document's script environment, the user script runs directly in the same global context as the document's own script content. This in turn grants us the ability to wrap and intercept native JavaScript functions, such as `XMLHttpRequest` or getter/setter properties of HTML objects (Magazinius et al., 2010).

Using this infrastructure, our extension conducted the following steps: The home-page URL of the next examination candidate is pulled from the backend and loaded

criterion	number of sites	% rel.	% abs.
Password found	2143	100,0 %	53,6 %
PW on HTTPS page	821	38,3 %	20,5 %
Secure action[1]	1197	55,9 %	29,9 %
`autocomplete` off	293	13,6 %	7,3 %
X-Frame-Options	189	8,9 %	4,7 %
JavaScript access	325	15,1 %	8,1 %

Table 3.2: Recorded characteristics of the Alexa top 4,000 password fields

into one of the browsers tabs. After the rendering process has terminated, the DOM tree is traversed to find password fields. However, most sites do not immediately contain the login dialog (if they have one at all) on their homepages. Instead, it is usually contained in a dedicated subpage, linked from the homepage. Hence, in case no password field could be found on the homepage, all hyperlinks on this page are examined, if they contain indicators that the linked subpage leads to the site's login functionality. This is done via a list of indicative keywords, consisting of, e.g., "sign in", "login", or "logon". If such a link was found, the browser tab is directed to the corresponding URL and this document is examined for password fields. While this methodology is apparently incomplete, e.g., due to the keyword list only containing terms derived from the English language, turned out to be sufficient to find a representative number of password fields, as we discuss in Section 3.2.2.

If at least one password field was found, important characteristics of the document were recorded, including the hosting document's URL, the corresponding HTML form's `action` attribute, as well as the presence of `autocomplete` attributes and `X-Frame-Option` headers.

Furthermore, to observe potential client-side processing after a password has been entered, we instrumented the `get`-property of discovered password fields using JavaScript's `Object.defineProperty` API (Mozilla Developer Network, 2013), after the page's rendering process has terminated, but before the page's own scripts are executed. This allows us to intercept, record and then forward all requests to access the value of the field.

Subsequently, after the page's scripts have been run, the user script simulates user interaction with the password field to potentially activate JavaScript snippets that access the password value legitimately. More precisely, our script triggers JavaScript events, that would occur if a user clicks into the field, changes its values, and leaves the password field, i.e., moves the focus to a different field. Finally, the script submits the form, in order to activate any JavaScript that is tied to the `onsubmit` event.

[1] Password form submitted to an HTTPS URL

Results During our crawl, we could successfully detect a login form on 2,143 of the 4,000 domains. In the following, we outline the analysis of the data we gathered from these fields with respect to different, security-relevant questions. An overview of the results is depicted in Table 3.2.

As discussed in Section 3.1.4, the use of the `autocomplete` attribute on input fields allows an application to ensure that no data is persisted into the password manager's storage. We therefore investigated how often this was explicitly set to `off`, essentially instructing the browser to neither store login data nor to automatically fill these forms. Out of the 2,143 domains we examined, a total of 293 domains prohibited password managers from storing the credentials this way.

With respect to the ClickJacking attack on a password manager that requires user interaction, an applicable remedy is the usage of the `X-Frame-Options` HTTP response header (Rydstedt et al., 2010). By using this header an application can explicitly tell the browser that a page may not be rendered inside a frame. However, this only helps against the discussed attacks if the header is set to `DENY`, since we must assume that the XSS attacker is capable of positioning an iframe containing the login form on a page located in the same application, thus running under the same origin. In our investigation, we found that only 189 domains set the header to `DENY`, while another 173 had set it to the `SAMEORIGIN`, which is useless in the context of the discussed attacks.

Furthermore, to gain insight on the extent of legitimate client-side functionality that uses JavaScript to read password fields, we instrumented the password field, such that we were able to record read access (see above). For a total of 325 password fields, we were able to witness read operations via JavaScript.

Finally, we examined to which degree the sites were potentially susceptible to network attackers. To do so, we checked how many forms containing password fields are delivered via plain HTTP rather than HTTPS. While this distinction is not relevant to a Cross-Site Scripting attack on the password manager, a similar attack scenarios was described by Gonzalez et al. (2013). Instead of exploiting a vulnerable application, they propose to conduct a man-in-the-middle attack to inject arbitrary content into targeted applications. This way, similarly to the attack we outlined, they can gain access to the stored credentials. Therefore, enabling HTTPS effectively blocks their outlined attacker model. In our study, we found that only 821 domains utilize HTTPS when transmitting the password field itself. The remaining 1,289 domains are hence susceptible to the network-based attacker who directly inserts JavaScript into the server's response to retrieve the password data from the victim.

Additionally, a network-based attacker may also retrieve passwords from users once they log in to an application if the credentials are sent to the server using HTTP and not HTTPS. Investigating how many applications send out secret login data in an unencrypted manner, we found that in total, 1,197 sites used HTTPS to send the password data to the server, leaving 946 sites with unencrypted communication.

3.2.3 Assessment

As shown in Section 3.2.1, most browsers only store the origin of a password and not the complete URL of the form it was initially stored from. Thus, placing a form on an arbitrary page with the same origin as the login form is sufficient to extract credentials from the victim.

The Cross-Site Scripting attacker, which we discussed in the previous sections, is capable of injecting his malicious payload into applications that are delivered via HTTP as well as over HTTPS. Thus, the only line of defense in this case is the `autocomplete` feature. As discussed earlier, this is only used in 293 login pages, thus resulting in a total number of 1,850 out of 2,143 domains which are potentially vulnerable to password stealing by an XSS attacker. This amounts to a total of 86.3% of analyzed pages which are susceptible to the attack scenario we outlined. Apart from Microsoft's Internet Explorer, the built-in password managers of all browsers we examined automatically filled out forms presented to them and would also behave in the same manner if the login page was put into a frame. In order to successfully conduct an attack on Internet Explorer, the attacker would have to have found a vulnerability on the exact login page and would also have to rely on the victim actively selecting the credentials to insert.

The network-based attacker, who is only capable of injecting his malicious payload into login pages which are not served using HTTPS, can only successfully attack 1,029 different domains, summing up to 48% of all applications we analyzed. These observations lead us to the conclusion that the current implementation of browsers' password managers is highly vulnerable with respect to password stealing – both by a network and an XSS attacker. Also, we find that server-side measures such as the `autocomplete` attribute are not employed in prevailing web applications in a satisfactory manner. Therefore, in the following section, we discuss a new approach to the concept and implementation of a password manager capable of tackling these issues.

3.3 Client-Side Protection

Our analysis has shown that popular browsers implement password managers in a way that is susceptible to Cross-Site Scripting attacks. We have shown that most of the browsers neither save information on the URL the password was initially stored for nor do they require user interaction to fill out forms. This allows the attacker to retrieve passwords in the presence of an XSS vulnerability. In the following, we propose a simple yet effective solution to counter these types of attacks on password managers.

3.3.1 Concept

The common enabling factor of the attack types we documented in Section 3.1 is the fact that the secret data is directly inserted into the forms when the page is loaded, and can subsequently be retrieved by JavaScript.

The underlying problem is that concept and implementation of password managers are not aligned. Abstracting what a password manager's task is, we see that it should aid users in the login process to Web applications. This process can be seen as the credentials being sent to the server. The implementation of that paradigm, however, aims at filling out forms before the actual, clear-text login data is required. An XSS attacker aims specifically at this conceptual flaw, extracting the credentials from the auto-filled form. In our notion, a password manager should ensure that only once the login data is sent to the server, the plain-text password is contained in the request. Hence, in the following, we propose an enhanced password manager which tackles this conceptual flaw.

Our proposal is that a password manager should only insert place-holding nonces into a form. Once the user then submits the form towards the application, the password manager replaces the nonce with the original, clear-text password. Thus, if an attacker can extract the content of this form utilizing a XSS vulnerability, he is nevertheless unable to retrieve the real password of the targeted user.

Furthermore, our mechanism requires strict matching of the password field `name` attribute and the corresponding POST value. For better understanding of the rationale behind this, consider the following scenario: The attacker is able to inject a new field called `query` into the form. Once the password manager has filled in the placeholder into the `password` field, the attacker's code copies the value of that field into the newly added `query` field. He then changes the action of the form to the application's search functionality. If the password manager now replaced all occurrences of the placeholder in the request, the `query` parameter would also contain the clear-text password. Under the assumption that a search page will in some manner reflect the search term back to the user, the attacker could then extract the password from this response. Therefore, making sure that the password manager only exchanges the correct field is essential.

3.3.2 Implementation

To investigate the soundness of our proposal, we implemented a proof-of-concept password manager. Since completely changing the implementation of one of the built-in password managers in the modern browsers would have been to complex, we opted to instead build an extension for Firefox. The extension is built upon the original, built-in password manager which is used to only store the placeholder values. The clear-text passwords in turn are stored in a separate storage located inside the extension. For our prototype, we did not implement any form of encryption

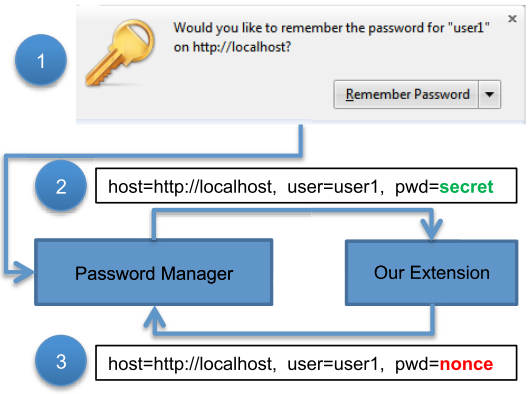

Figure 3.2: Initial login and credential storing

for these values, since securely storing the passwords inside the browser is out of scope for this work.

Figure 3.2 shows how our approach works when a new username and password combination is saved. First, the user is prompted to have the login manager remember the recently sent password. In the second step, once the user has agreed to do so, the login manager stores the username and password combination. Firefox provides all plugins with a means of being notified when credentials are stored in the password manager (Mozilla, 2015). The notification message contains – along with the recently stored username and password – the origin of the site the password was posted to as well as the names for both the username and password field in the submitted form. Our extension saves all this information in its own storage and replaces the password in the built-in storage with a random placeholder value (nonce). This placeholder value is subsequently also stored inside the extension's database alongside the previously persisted data to ensure that the matching credentials can be extracted later on.

Figure 3.3 outlines how the placeholder is later restored in a normal login. When opening the login page, the built-in password manager inserts the username and the placeholder into the form. Similar to the internal password manager, the extension is notified of a password form being submitted (Dolske, 2013). Subsequently, the next outgoing POST request is scanned by our extension for the easily discernible placeholder value. If the nonce is found, the extension searches its own database for the corresponding entry. Next, the entry's origin is checked against the origin of the page the data is being sent to. If the origins match, the placeholder is replaced with the actual passwords adhering to the aforementioned constraint that only the password field (whose name is stored in the extension's data storage) should be changed. This concept is shown on the lower half of the figure. Although the nonce is contained in the HTTP request, it is not replaced with the actual secret data since the name of the POST parameter does not match. Thus, the attacker cannot

Figure 3.3: Replacement of login credentials by our enhanced password manager

utilize the search functionality to extract the secret password data from our password manager.

We also evaluated the option of exchanging GET parameters in a request. In our empirical study we found that none of the sites use a form in combination with a GET request. An attacker could however easily exchange the method of a form from POST to GET. If our proposed password manager would then exchange the nonce with the secret password, the adversary could easily read the complete URL of the newly loaded page and thus retrieve the password. Therefore, we explicitly disable the replacement of our nonces in GET parameters and only exchange them for the real credentials in POST requests.

3.3.3 Evaluation

In this section, we discuss both the security and the functional evaluation of our approach, showing that it adds security while not causing incompatibility with existing applications.

Security Evaluation After the password value has initially been stored by the password manager, it is never again inserted into Web documents. Hence, it is kept out of reach of potentially malicious JavaScript.

Furthermore, our implementation enforces strict matching constraints, before the replacement process executes: Only password nonces for which the combination of *target origin* and *password parameter name* matches the recorded values are substituted with the actual password value in the outgoing request. This requirement effectively thwarts attack attempts in which the adversary tries to leak the password via tampering with the password field's form element in the timespan between the autofill process and the form submission. Thus, our proposed implementation of a secure password manager effectively hinders an attacker who utilizes XSS attacks against his victim.

However, the attacker model discussed by Gonzalez et al. (2013) – positioned at the network layer – could still be successful if password data is transmitted to the server in clear text. In our study of the Alexa top 4,000 sites, we found that 44,1% of the

examined sites utilize HTTP instead of HTTPS to transmit the credentials to the server. In these cases, the network-based attacker can simply wait for the form to be submitted and subsequently retrieve the secret login data from the traffic capture. Nevertheless, this kind of attack does not specifically target password managers and can therefore not be fully prevented by a secure password manager in any case.

Functional Evaluation From the user's point of view, nothing changes compared to the behavior of the current generation of deployed password managers: After page load, the password field is automatically filled with characters, which are presented to the user with masquerading asterisks. After form submit, the browser exchanges the password nonce with the actual values, before it is sent to the server.

Our approach aims at only putting the real password of a user in the outgoing request to the server and not into the password field. This however leads to potential problems with Web applications that perform some transformation on the original field's value before submitting it. For instance, an application might derive the hash sum of the user-provided password on the client-side before submitting it.

In the evaluation of the top 4,000 Alexa sites, we detected 325 JavaScript accesses to password data (cp. Table 3.2). We then manually analyzed the snippets responsible for these accesses and detected that a total 96 domains used client-side functionality such as XmlHttpRequests to transmit the password data to the server. Out of these 96 cases, 24 pages transformed the provided password before forwarding it to the server, whereas 23 employed hashing functions like MD5 and SHA1 and the remaining case encoded the password as Base64. Of the remaining 72 pages that did not post the form directly to server, only 6 pages employed HTTP GET requests to transmit the credentials, whereas the rest used HTTP POST in their XmlHttpRequests. Our proposed approach would obviously not work in these 30 cases, since our extension neither exchanges passwords directly in the input field nor does it modify HTTP GET requests. However, the current implementations of the password managers do not support storing passwords that are not sent via submitting HTML forms and, thus, our approach is in no way inferior to the currently deployed concepts (Dolske, 2013). Also, if the built-in password manager stored these credentials, there is no way of detecting whether access to a given password field is conducted by the legitimate page or is a Cross-Site Scripting attack. Hence, we deliberately fail in the aforementioned scenario by not replacing the nonce in the input field with the real password. Therefore, our approach is secure by default and can also not be undermined by an unknowing user.

The purpose of the remaining 229 scripts was to verify that certain criteria had been met in filling the user and password field, e.g., the username being an e-mail address or the password consisting of at least a certain amount of characters.

3.4 Summary

This chapter highlighted the attack potential inherent to Cross-Site Scripting vulnerabilities. We discussed the general attack pattern of leveraging password managers to leak sensitive information in the presence of an XSS vulnerability. We then presented an empirical study on password fields on the Alexa top 4,000 Web pages, finding that at least 2,143 contain password fields. Based on the specific attack patterns and the knowledge of how password fields are used in the Web, we outlined an extension of existing browser-based password managers that is not susceptible to XSS-enabled attacks. We evaluated the concept with respect to the password fields we discovered in our study and found that our proposed approach only causes issues with 30 of the investigated sites. More to the point, these sites utilize a non-standard means of submitting password and, thus, password managers lack support for these sites to begin with. Our approach is therefore in no way inferior to existing implementations, but provides a password manager that is secure by default with respect to the outlined attacks.

While our presented solution provides robust protection against Cross-Site Scripting attacks targeting password managers, a large number of additional scenarios exist in which the presence of an XSS vulnerability can be abused. Therefore, in the following chapter, we present a study aimed at determining the prevalence of a subclass of Cross-Site Scripting, namely Client-Side or DOM-based Cross-Site Scripting, on real-world Web applications.

Chapter 4

Detecting Client-Side Cross-Site Scripting on the Web

In the previous chapter, we discussed an attack scenario in which a Cross-Site Scripting attacker can leverage a discovered vulnerability to retrieve credentials from browser-based password managers. Apart from this specific attack, a plethora of additional techniques exist which an attacker can use to cause harm using an XSS flaw. Therefore, one goal of our research is to determine how frequent these kinds of vulnerabilities occur on real-world applications. While previous research has focussed on server-side code causing exploitable issues, the focus of this work is on the sub-class of Cross-Site Scripting resulting from insecure client-side code: *Client-Side* or *DOM-based Cross-Site Scripting*.

In this chapter, we outline an empirical study aimed at determining how prevalent this class of Cross-Site Scripting is on the Web. To do so, we first discuss the implementation of a taint-aware browsing engine, which allows for precise tracking of data originating from an attacker-controllable source to a security-sensitive sink. We then present details on an exploit generator, which takes into account the taint information from our browsing engine to precisely generate potential exploits. On the basis of these two core components, we present the results of our study of the Alexa top 5,000 domains and show that almost one out of ten domains in our data set carries at least one client-side XSS vulnerability.

4.1 Vulnerability Detection

To automatically detect the flow of potentially attacker-controllable input from a source into a sink in the sense of DOM-based XSS, we decided to implement a dynamic taint tracking approach. To ensure that edge cases, which might not be implemented properly in pure testing engines like HTMLUnit, were to be properly executed, we chose to implement taint tracking into a real browser. For this, we modified the open-source browser Chromium in such a manner that its JavaScript engine V8 as well as the DOM implementation in WebKit were enhanced with taint tracking capabilities. For both components of the browser, we selected to use a byte-wise taint tracking approach built directly into the respective string representations. In this fashion, we enabled our tool to not only distinguish between a completely untainted string and a string containing any potentially harmful content, but also to specifically retain information on the origin of each character in a given string. In the following, we discuss the implementation details of our patched browsing

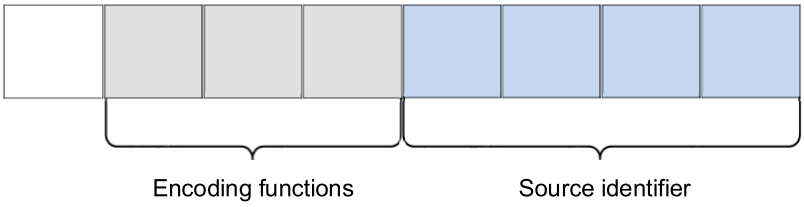

Figure 4.1: Representation of sources and encoding functions in Chromium

engine. After this, we present the concept of an exploit generator, capable of precisely producing candidate exploits for the discovered flows.

4.1.1 Labeling Sources and Encoding Functions

To keep the memory overhead as small as possible, we chose to implement our approach in such a way, that information on a given character's source is encoded in just one byte. We therefore assign a numerical identifier to each of the 14 relevant sources (such as `location.href`, `location.hash` or `document.referer`). Hence, we were able to encode this information into the lower half of the byte. In addition to the information on the character's origin, our engine should also be able to determine whether a given character was encoded using the built-in functions *encodeURI*, *encodeURIComponent* or *escape*. To this end, we used the lower three of the four remaining bits to store whether one or more of these functions were applied to the character. To represent a benign, i.e., hardcoded, character, the lower four bits are set to 0. Figure 4.1 outlines the described layout.

4.1.2 Patching the V8 JavaScript Engine

Google's JavaScript engine V8 is highly optimized in regards to both memory consumption and execution speed. Although the code is written in C++, V8 for the most parts does not make use of a class concept using member variables when representing JavaScript objects like strings or arrays. Instead, a small header is used and an object's components are addressed by only using given offsets relative to the object's address.

After careful examination of the given code, we chose to only encode the minimal necessary information directly into the header. The V8 JavaScript engine recognizes an object by its *map*, i.e., there is a different map allocated for each type of object. We found an unused part of a bitmap in the maps and used it to create new maps for tainted strings. Obviously, for strings of dynamic length, additional memory must be allocated to store the actual data. Depending on the type of string, i.e.,

pure ASCII or two-byte strings, the required size differs and on creation of a string object, the necessary memory is allocated. The address of this newly created space is then written to one of the offsets in the header. Along with the information that a string is tainted, we also need to store the taint bytes described above. To do this, we changed the string implementation such that additional `length` bytes are allocated. Since we wanted to keep the changes to existing code as small as possible, we chose to store the taint bytes into the last part of the allocated memory. This way, the functionality for normal access to a string's characters did not have to be changed and only functionality for taint information access had to be added.

As mentioned before, the V8 engine is optimized for performance. It therefore employs so-called *generated code* which is assembler code directly created from macros. This way simple operations, such as string allocation, can be executed without using the more complex and, thus, slower runtime code written in C++. However, for our approach to easily integrate into the existing code, we chose to disable the optimizations for all string operations such as creation or substring access.

After patching the string implementation itself, we also instrumented the string propagation functions such as `substring`, `concat` or `charAt` to ensure to ensure that the byte-wise taint tracking information is also propagated during string conversions.

4.1.3 Patching the WebKit DOM Implementation

In contrast to the V8 engine, WebKit makes frequent use of the concept of member variables for its classes. Therefore, to allow for the detection of a tainted string, we were able to add such a member denoting whether a string is tainted or not. The string implementation of WebKit uses an array to store the character data. Hence, we added a second array to hold our taint bytes. Since strings coming from V8 are converted before being written into the DOM, we patched the corresponding functions to allow the propagation of the taint information. This is necessary because tainted data might be temporarily stored in the DOM before flowing to a sink, e.g. by setting the `href` attribute of an anchor and later using this in a `document.write`. To allow for correct propagation of the taint information, we not only needed to change the string implementation but also modify the HTML tokenizer. When HTML content is set via JavaScript (e.g. using `innerHTML`), it is not just stored as a string but rather parsed and split up into its tree structure. Since we want to ensure that taint information is carried into the tag names and attributes in the generated tree, these changes were also necessary.

4.1.4 Detection of Sink Access

Until now we discussed the tracking of tainted data inside the V8 JavaScript and WebKit rendering engine. The next step in our implementation was to detect a tainted flow and to generate a corresponding report. Therefore, we modified all

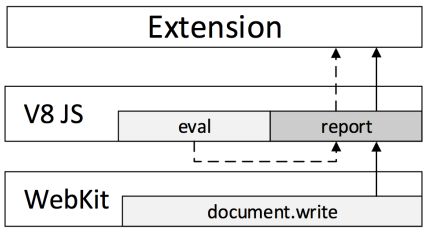

Figure 4.2: Report functionality

DOM-based Cross-Site Scripting sinks – such as `document.write`, `innerHTML` or `eval`. We changed them in such a way that a reporting function is called each time a tainted string is passed to such a sink. In order to pass on the report to the user interface, we implemented a Chrome extension, that injects the JavaScript reporting function into the DOM. As such a function is callable from inside the runtime engine, we are able to report the flow to the extension. The details on the layout and implementation of this extension are presented in Section 4.3.1.

In WebKit's API used to provide access to the DOM tree for V8, the passed arguments are of V8's string type and are then converted to WebKit's string type. Hence, we chose to implement our reporting function into V8's string class, therefore allowing us to invoke it from the DOM API as well as directly from V8 using the provided string reference. When called, this function gathers information on the code location of the currently executed instruction and reports these alongside the taint information and details on the type of sink to the extension.

Figure 4.2 depicts this layout. Both indicated functions `eval` and `document.write` use the reference to the passed string to invoke the reporting function which in turn passes on the information to the Chrome extension shown at the top. Analogous to the depicted functions, all relevant sinks were patched to ensure proper taint reporting.

4.2 Automated Exploit Generation

While the fact that a flow of data occurs from an attacker-controllable source to a security-sensitive sink might constitute a vulnerability, not every flow is actually exploitable. Listing 4.3 shows an example of such a flow. When visiting this site without trying to actively exploit it, our browsing engine would gather information indicating that a string, which is completely controllable by an attacker, was passed to `document.write`. Note that, however, in this case, the access to `document.write` only occurs if the string matches the regular expression `^a-z0-9$`, i.e., only lowercase alphanumerical values are allowed. Thus, this flow does not constitute a vulnerability.

```
if (/^[a-z0-9]+$/.test(location.hash.slice(1))) {
  document.write(location.hash.slice(1));
}
```

Listing 4.3: Non-vulnerable flow example

As the previous example has shown, we must verify that a flow is an actual vulnerability. Given the fact that we employ dynamic analysis techniques, a natural next step is to verify the vulnerability by exploiting it. To do so at scale, we chose to implement an exploit generator capable of precisely producing URLs crafted such that they allow for exploitation of a vulnerability. To generate the appropriate input needed to trigger a vulnerability, the exploit generator takes into account the exact taint information provided by the browsing engine as well as the parsing rules for HTML and JavaScript, respectively. In the following, we explain the process of generating a potential exploit for a discovered flow of data.

Listing 4.4 shows a vulnerable snippet of JavaScript code. When executed inside our taint-aware browser, the access to `eval` emits a report showing that a partially tainted string was passed to it. We assume the user visited `http://example.org` and, thus, the passed string is `function x() { var y="http://example.org";}`, whereas the highlighted part denotes the attacker-controllable substring.

```
eval('function x() { var y = "' + location.href + '";}');
```

Listing 4.4: Vulnerable JavaScript snippet

Figure 4.3 shows the resulting JavaScript tokens generated when parsing the passed string, highlighting the user-controlled part in red. In order to allow for the execution of a payload of our choosing, we need to break out of the existing context, i.e., close the string literal and complete the declaration and assignment of `y`. In this case, however, we can not ensure that our injected code will be executed. Rather, this code shows the declaration of function `x` and we cannot be sure that this function is going to be called by the normal execution path taken by the vulnerable application. Thus, we opt to break out to the top execution level, i.e., out of the function declaration. Hence, we need to craft our URL such that we can:

- Break out of the string literal: `"`
- Break out of the declaration of y: `;`
- Close the block: `}`
- Append our payload: `alert(1);`

```
FunctionDeclaration
  Identifier : x
  FunctionConstructor
    Identifier : x
    Block
      Declaration
        Identifier : y
        StringLiteral : "http://example.org"
```

Figure 4.3: Parsed JavaScript tokens

The payload `alert(1);` is to be considered arbitrary JavaScript code of the attacker's choosing. Combining these components and appending them to the URL, however, does not suffice to exploit the vulnerability as the remaining, hard-coded characters `";}` would cause a parsing error. In such a case, rather than executing the code until a parsing error occurs, the JavaScript engine throws an exception and no code is executed (Ecma International, 2011). Therefore, we need to ensure that these trailing characters do not cause any parsing errors. A straight-forward solution to this problem is to comment out the remaining string by simply appending `//` to the payload.

Thus, our crafted input now consists of a sequence of characters to break out of the existing context, our arbitrary payload, and a comment. To ensure that the payload is not encoded automatically by Chrome, we prepend it with a hash mark (Zalewski, 2009a). To exploit the vulnerability, the attacker has to lure his victim to `http://example.org/#";}alert(1);//`. Listing 4.5 shows the code which is executed in this case, indented for better readability. As we can observe, our injected call to `alert` is contained in the top level of execution, and, thus, will be executed regardless of whether function `x` will ever be called.

```
function x() {
  var y="http://example.org/#";
}
alert(1);
//";}
```

Listing 4.5: Exploited `eval` vulnerability from Listing 4.4

Similarly to the process outlined here to exploit a vulnerable flow of attacker-controllable data to `eval`, the exploit generator is capable of generating exploits for HTML contexts, i.e., flows to `document.write` and `innerHTML`. As the exploit

generation is not a major contribution of this thesis, we refer the reader to Lekies et al. (2013) for a more detailed explanation of this process.

4.3 Empirical Study

An important motivation for our work was to gain insight into the prevalence of potentially insecure data flows in JavaScript applications and moreover, the number of exploitable flows. Therefore, based on the components we presented in the previous section, we decided to create a Web crawling infrastructure capable of gathering flow information and, after the generation of exploit candidates, verifying the vulnerabilities on a large set of real-world Web sites. In the following, we give an overview of our study's methodology and used architecture, followed by the results and insights we gathered.

4.3.1 Architecture Overview

One key aspect of getting a grasp on the prevalence of client-side XSS on the Web is to sample a sufficiently large set of real-world Web sites. We therefore designed our experimental set-up to meet this requirement, utilizing the developed and previously presented components.

First, we use our taint-aware browsing engine to gather flow information. In order to do this in an automated and unsupervised manor, we set up a control backend to provide URLs to crawl and store reported flows. We then deploy the Chrome extension, which we discuss in the following, to the browser, allowing it to drive the engine to visit a page, collect all links contained in it and recursively crawl the discovered pages, while also reporting back all observed data flows. After this initial data gathering phase, we pass the acquired flows to our exploit generator and use the browsing engine to subsequently crawl the produced, potentially exploitable URLs.

In the following, we discuss details of the central backend and the Chrome extension, depicted in Figure 4.4. After that, we present results on the number of observed data flows and, subsequently, on the amount of discovered, exploitable vulnerabilities.

Central Server Backend The central backend's main task is two-fold: it distributes the URLs which are to be analyzed to the browsing instances and collects the reports of flows which occurred in the engines when crawling these pages. An initial set of URLs is therefore stored in the backend's database and subsequently crawlers retrieve a single URL at a time. When a single browsing instance has finished analysis of a given URL, it reports back the discovered flows. Due to the fact that a single URL may contain more than one frame (e.g., if it includes advertisements in frames), the report is divided on a per-frame basis. This way, if a

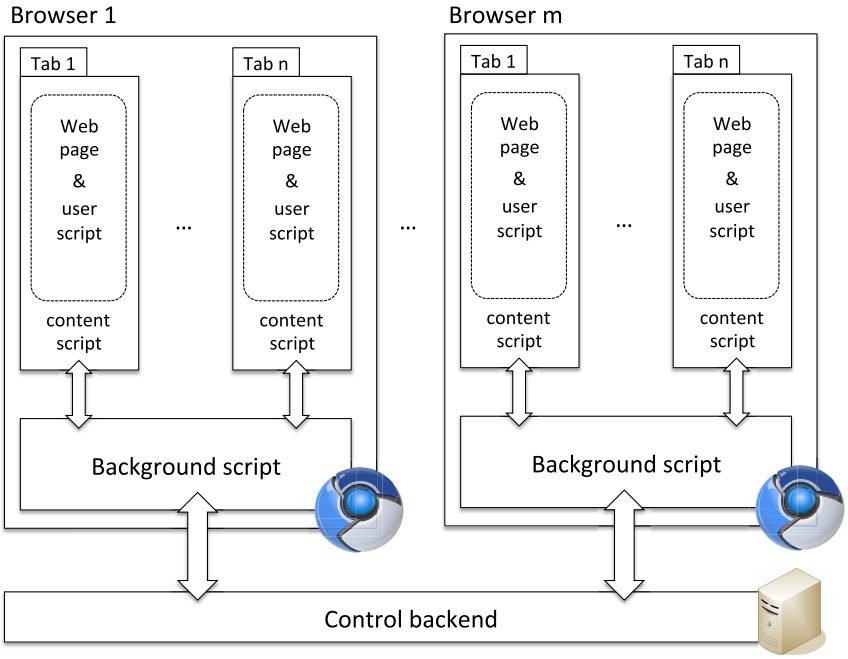

Figure 4.4: Crawling infrastructure

flow occurred in a subframe of the analyzed page, the flow of data can be properly attributed to said subframe rather than the top frame.

When reports have been gathered for all domains under investigation, the data stored in the backend's database is passed to our exploit generation engine. As previously discussed, the exploit generator then produces URLs for all the prototype exploits. These, in turn, are then stored back in the central server backend to be crawled by the browsing engines. To simplify the automation of the complete process, the Chrome extension will ask for a set of configuration parameters (such as crawl depth and whether to report back any discovered flows). Therefore, in the exploit verification phase, the overhead can be massively reduced by simply disabling the sending of reports from the clients to the server. Finally, the server also provides an interface which allows the browsing engines to report the successful exploitation of a URL.

Browser Extension To allow for an easy maintainability of our changes to the core codebase, we kept direct changes to it as small as possible. As the Chrome Extension API provides powerful features to developers, such as allowing full control over a tab, we opted to implement our crawling functionality as a Chrome extension rather than inside the core code. In contrast to changes to the browsing engine's core, Chrome extensions can be developed without the need for compilation, as they are implemented in JavaScript. This allows for easier development and debugging and, thus, a lower chance of implementation bugs and issues.

As intended by the general architecture of Chrome's extension model (Google Developers, 2012), our extension is divided into two components: the background and the content script (cp. Figure 4.4). The purpose of the background script is to communicate with the central backend (retrieving URLs and reporting flows) and controlling the crawl conducted by the browser. To do so, it opens a predefined number of browsing tabs, assigns a URL to analyze to each tab and, after the analysis of said page has finished, collects the newly discovered links and stores them in its local data storage. Note that the background script cannot execute any code in the context of a rendered page, but has to rely on the content script to gather the links contained in a page and report them back to the background script. Whenever the content script sends a raw taint report, it is initially post-processed by the background script before being sent to the central backend server.

Due to the message-driven concept employed in Chrome extensions, a page cannot directly communicate with the background script, but rather has to use an intermediary content script. Rather than running once in the extension, this script is injected into each page which is rendered. This enables the content script to run arbitrary JavaScript code in the context of the analyzed page, i.e., it can gather the taint reports from the browsing engine, collect anchors and the URLs they reference, and invoked events on the target page. To ensure that crawling can be conducted without interruption, the content script also overwrites blocking modal functions such as `alert` or `prompt`.

Whenever the background script assigns a URL to a tab, effectively invoking the analysis of that page, the page's HTML and JavaScript code is retrieved and executed. All data flows which occur during this execution are buffered in the content script. To allow for execution of script code not contained in the initial HTML document (i.e., scripts which reference external content), the page is executed for a fixed amount of time. When the timeout is triggered, the content script packs all information on the occurred flows and sends a message containing said data to the background script for further processing (see above).

4.3.2 Observed Data Flows

The initial data set for our study were all domains contained in the Alexa top 5,000. For each of these domains, we conducted a crawl with a depth of one, i.e., analyzing the main page and all same-domain links contained on this initial page. In total, this resulted in access to 504,275 URLs, whereas, on average, each contained 8.64 frames. Thus, our analysis engines gathered flow information on 4,358,031 (not necessarily unique) documents.

In total, these documents caused 24,474,306 data flows from tainted sources to security-sensitive sinks, which were observed by our instrumented browsers and subsequently stored in our central database. Table 4.1 shows the distribution of sources and sinks, respectively. The majority of all flows originates from cookies and also

	URL	cookie	document referer	window name	post-Message	Web Storage	sum
HTML	1,356,796	1,535,299	240,341	35,466	35,103	16,387	3,219,392
JavaScript	22,962	359,962	511	617,743	448,311	279,383	1,728,872
URL	3,798,228	2,556,709	313,617	83,218	18,919	28,052	6,798,743
Cookie	220,300	10,227,050	25,062	1,328,634	2,554	5,618	11,809,218
Web Storage	41,739	65,772	1,586	434	194	105,440	215,165
postMessage	451,170	77,202	696	45,220	11,053	117,575	702,916
sum	5,891,195	14,821,994	581,813	2,110,715	516,134	552,455	24,474,306

Table 4.1: Data flow overview, mapping sources (top) to sinks (left)

ends in the cookie sink, and thus, is not directly exploitable. Similarly, several combinations of sources and sinks exist which do not allow for an automated generation and verification of exploits (e.g., flows starting from the Web Storage API). Therefore, in the following, we shed light on the process of selecting flows for exploit generation.

4.3.3 Selective Exploit Generation

As we have shown in the previous section, the total number of potentially vulnerable data flows from attacker-controllable sources to security-sensitive sinks is surprisingly high, i.e., averaging 50 flows per analyzed URL. While our vulnerability verification method using a generated exploit allows us to confirm the existence of a flaw in roughly the same time it takes the engine to gather the flow information in the first place, doing so for almost 25 million URLs is unfeasible. Thus, in order to balance cost and benefit of such a verification, we selected to only generate exploits for a subset of all recorded data flows, based on the following criteria:

C1: The data flow originates from a source that can be directly controlled by an adversary, i.e., this excludes all such flows that originate from second-order sources, such as cookies or Web Storage as well as flows from the postMessage API.

C2: The data flow ends in a sink that allows for direct execution of the injected payload, i.e., all flows *into* sinks like cookie, Web Storage, postMessage or DOM APIs are excluded.

C3: The data which is passed to the sink is not encoded by any of the built-in encoding functions provided by JavaScript. While the application of the incorrect encoding function can still result in a vulnerability in very rare cases (such as the use of `encodeURI` in conjunction with a flow to `eval`), the benefit of finding a small number of additional vulnerabilities is far outweighed by the cost of crawling a sufficiently larger set of URLs. Thus, we exclude all flows which passed through any encoding function.

C4: For each of the remaining data flows, we generate an exploit prototype. As Web applications may incorporate the same piece of vulnerable JavaScript code more than once in a single page, multiple flows on such a page would lead to the generation of the same exploit prototype URL more than once, as all these flows require the same break out sequence and payload. To decrease the overall overhead of testing the exploits, we opt to only generate one exploit in the outlined cases.

We successively applied these criteria to the 24,474,306 flows we initially gathered, establishing the relevant set of flows and, thus, candidate exploits:

$$24,474,306 \xrightarrow{C1} 4,948,264 \xrightarrow{C2} 1,825,598 \xrightarrow{C3} 313.794 \xrightarrow{C4} 181,238$$

Thus, in total we generated 181,238 test payloads, which we passed on to our verification step. In the following section, we discuss the results of this exploit verification phase.

4.3.4 Discovered Vulnerabilities

In the next step, we needed to determine how many of the 181,238 candidate exploits would trigger our injected payload. For 43,412 of the exploits, the source of data was either `location.search` or `document.referer`. While our methodology uses Chromium to detect vulnerable flows, it cannot be used to verify these exploits due to the fact that these values are automatically encoded by Chromium. In contrast to Chromium, Internet Explorer does not apply any encoding when retrieving the data from these sources (Zalewski, 2009a). Thus, we opted to implement a minimalistic crawler for Internet Explorer to verify those vulnerabilities. For the remaining 137,826 URLs, we relied on our existing crawling infrastructure using Chromium and the outlined control backend.

Using this two-pronged approach, a total of 58,066 URLs tested in Chromium triggered our verification payload. Additionally, we could exploit 11,921 URLs visited in Internet Explorer, resulting in a total of 69,987 successfully exploited flaws. This corresponds to a success rate of 38.6% in total, and a success rate of 42.1% when only considering vulnerabilities exploitable in Chromium.

As outline before, our crawler followed all links discovered on the entry page. We assume that a high number of Web sites utilize content management systems and thus include the same client-side code in each of their sub pages. Hence, to zero in on the number of actual vulnerabilities we decided to reduce the data set by applying a uniqueness criterion.

For any finding that triggered an exploit, we therefore retrieved the URL, the used break out sequence, the type of code (inline, eval or external) and the exact location of the executed instruction. Next, we normalized the URL to its corresponding second-level domain. To be consistent in regards to our selection of domains, we used the search feature on alexa.com to determine the corresponding second-level domain for each URL. We then determined for each of the results the tuple:

$$\{\text{domain}, \text{break out sequence}, \text{code type}, \text{code location}\}$$

With respect to the code location, we chose to implement the uniqueness to be the exact line and column offset in case of external scripts and evals, and the column offset in inline scripts. Applying the uniqueness filter to the complete dataset including those pages only exploitable on Internet Explorer, we found a total of 8,163 unique exploits on 701 different domains, whereas a domain corresponds to the aforementioned normalized domain. Due to the nature of our approach, among these were also domains not contained in the top 5,000 domains. Thus, we applied another filter, removing all exploits from these domains outside the top 5,000.

This reduced the number of unique exploits to 6,167, stemming from 480 different domains. With respect to the number of domains we originally crawled, this means that our infrastructure found working exploits on *9.6%* of the 5,000 most frequented Web sites and their sub-domains.

When considering only exploits that work in Chromium, we found 8,065 working exploits on 617 different domains, including those outside the top 5000. Again filtering out domains not contained in the 5000 most visited sites, we found 6,093 working exploits on 432 of the top 5,000 domains or their sub-domains.

Among the domains we exploited were several online banking sites, a poplar social networking site as well as governmental domains and a large internet-service provider running a bug bounty program. Furthermore, we found vulnerabilities on Web sites of two well-known AntiVirus products.

4.3.5 Insights of our Study

In the following, we want to discuss select insights gained in our study. Doing so, we analyze the usage of encoding functions with respect to the sources of data, the origin of vulnerable code as well as multi flows, which combine several sources in a single sink access.

Encoding with respect to Data Sources When analyzing our data set we found significant differences in the way the user-provided input was handled. In almost 65% of flows originating from the URL, an encoding function was applied before the data eventually ended in a sink. Similarly, almost 84% of all flows which began in

`document.referer`, i.e., the URL, which contained a link to the currently analyzed page, that our crawler followed, were encoded. In contrast to this, only about 1.5% of all flows originating from postMessages were encoded. The postMessage API allows browser windows to communicate across domain boundaries by exchanging messages, which may contain arbitrary JavaScript objects. While the API offers a means for the receiving window to ensure that a message was sent by a page on a specific origin, this check is often omitted (Son and Shmatikov, 2013). Although our work did not aim at exploiting these kinds of flows, the work conducted by Son and Shmatikov (2013) highlights the fact that this feature is not well understood, allowing them to exploit 84 popular sites.

Origin of Vulnerable Code Web applications may include third-party code such as advertisement or analytics code. When this code is included from a remote origin, it inherits the origin of the including application, i.e., flaws in third-party code can cause vulnerabilities in the including Web app. Therefore, another dimension of our analysis focussed on the source of vulnerable code. We found that in 13.09% of the cases, third-party code was responsible for the sink access with attacker-controllable data. As we discuss in Section 5.3.5, this does not necessarily constitute that the vulnerability is caused only by third-party code, but may also be the result of incompatible first- and third-party code. The biggest portion of vulnerable code, namely 79.64%, was contained in external script files which were hosted by the Web application itself. In contrast, a comparably small number of vulnerabilities were caused by inline script code (3.81%). The remaining 3.46% of vulnerabilities were caused by code which was created at runtime using `eval` — for this kind of code, our analysis engine was unable to provide the exact location, i.e., whether the call to `eval` occurred inline or in external JavaScript files.

Multiple Sources in Single Sink Access In our study we found that for each call to a sink, on average three substrings consisted of tainted data. Contained in these were cases in which the same value was concatenated more than once (such as appending the URL multiple times) as well as flows, that utilized several sources. An example for such a flow is the extraction of different parameters from the URL and combining them into a new URL, possible appending strings from other sources such as cookies. We deem such vulnerabilities *multi flows*, as they allow an attacker to construct his attack payload by combining multiple sources of data. An example of such a snippet is shown in Listing 4.6. In the depicted code, the URL of the `img`'s source is derived by combining both the current URL and the URL of the referring page. An attacker might therefore use the URL to inject parts of his payload and append the remainder of it to the `referer`. As we discuss in Chapter 6, this can be used to fool and, thus, circumvent the XSS Auditor deployed in Google Chrome.

```
var ref = document.referer;
var url = document.location.href;

document.write("<img src='//advert.ise/?url=" + url + "&referer=" + ref +
↪  "'>");
```

Listing 4.6: Example of a multi flow vulnerability

4.4 Summary

In this chapter, we presented the design, implementation and results of an empirical study to detect Client-Side Cross-Site Scripting vulnerabilities on the Web. This was motivated by the fact that although this form of XSS attacks has been known since 2005, no previous research had tried to determine to number of vulnerable sites. Therefore, after implementing a taint-aware browsing engine, we developed an exploit generator capable of precisely crafting prototypical exploit payloads based on the taint information provided by our engine. Using this methodology, we conducted an empirical study on the Alexa top 5,000 domains, analyzing over 500,000 URLs, containing more than 4.3 million documents and causing almost 25 million potentially harmful data flows. Focussing on directly exploitable data flows, we evaluated a set of 181,238 exploit candidates, out of which 69,987 successfully triggered our injected payload. Filtering out any exploits on domains outside the Alexa top 5,000, we found that 9.6% of these domains carry at least one Client-Side Cross-Site Scripting vulnerability.

While this study allowed us to gain a glimpse into the prevalence of DOM-based Cross-Site Scripting vulnerabilities on the Web, it did not provide us with additional insight into the nature of such exploits. Therefore, in the following chapter, we explore the complexities behind modern day Web applications and investigate whether these are a root cause of Client-Side Cross-Site Scripting, or if these flaws are caused by developers unaware of the risks that accompany the use of user-provided data in such applications.

Chapter 5

On the Complexity of Client-Side Cross-Site Scripting

In the previous chapter, we outlined the results of an empirical study aimed at finding Client-Side Cross-Site Scripting vulnerabilities on the Web. In total, we discovered that 9.6% of the Alexa top 5,000 Web pages carry at least one client-side XSS flaw. While this study allowed us to approximate a lower bound for the number of flaws currently in existence on the Web, it did not provide us with any information on the root causes of such flaws. Therefore, in the following, we outline an empirical study aimed at determining just these causes. To do so, we first discuss the inherent complexities of JavaScript and present metrics that allow us to measure this complexity. Subsequently, we present the infrastructure necessary to conduct an analysis of the complexities of real-world vulnerabilities, followed by the discussion of the empirical study executed using that infrastructure. We conclude this chapter by discussing the key insights gathered from our study, showing that while client-side XSS is often easy to spot and remove, a notable number of flaws shows a significant complexity.

5.1 The Inherent Complexities of JavaScript

JavaScript offers a plethora of powerful languages features that make it well-suited for usage in a dynamic environment such as a Web browser. While these features, such as the ability to use `eval` to generate and execute code at runtime, allow a programmer to perform manifold actions on the client side, they also increase source code complexity and, thus, the potential for security problems. In the following, we briefly outline some of these complexities.

JavaScript's Concurrency Model Unlike other programming languages, JavaScript specifically limits the amount of threads which may be run in parallel to one (Mozilla Developer Network, 2015a). In contrast, a Web browser often retrieves content from remote servers, i.e., conducts I/O-intensive operations. Given the fact that JavaScript is single-threaded, the need for a model arises that allows for a continuous execution of such parts of the script code which are not waiting for any I/O response. Therefore, JavaScript employs an event-driven concurrency model. In the concrete case of a request to an external resource, the JavaScript snippet can register a callback function, which is invoked when the response arrives. This way, it can yield to the remaining JavaScript code waiting to be executed. The execution of code is interrupted whenever a callback function is called (Mozilla Developer Network, 2015a).

The exact timing of that invocation is based on a number of factors, such as the latency of the network connection, the load of the remote server or a local cache hit. Hence, there is no guarantee when a callback function will be invoked. More importantly, there is no guarantee that the control flow follows the linear structure in which the code was defined in a file. Thus, when inspecting a piece of JavaScript code, an analyst has to be aware of this event-driven model and take into account all potential temporal execution sequences.

JavaScript's Global Object Although single script blocks or external files can be viewed as distinct JavaScript programs, they are executed in the context of the same Web page. Therefore, they all operate on the same global object, which in the case of the browser is the `window` object (Mozilla Developer Network, 2014a). These script resources may be included from third-party sites and, thus, origin from different developers. Yet, whenever such a resource is included in a page, it inherits the origin of the including page and operates on its global object.

To allow for different pieces of code to interact with each other, programmers may choose to do by using global variables, which are accessed by one another or by defining global functions, which act as an API. While this allows for easy interaction between different libraries, it may also cause unwanted side effects. One example of such a side effect is the assignment of a global variable called `name`. As all variables are allocated in the `window` namespace, this registers the variable `window.name`, which is actually a property that can be read and written across domain boundaries (Mozilla Developer Network, 2014b). Thus, relying on such a variable may allow attackers to inject their malicious code into it.

Generating Code at Runtime using eval According to Richards et al. (2011), up to 82% of the most popular Web sites utilize the `eval` function to construct and execute code at runtime. As discussed in the previous chapter, the use of `eval` can lead to security vulnerabilities, as the code is often assembled using runtime information that is controllable by the user, e.g., using parts of the URL or `window.name`. Therefore, the code which is evaluated at runtime does not exist in full at development time of application. Even if no attack is conducted when initially calling `eval`, using the function makes it harder for analysts to understand the code that is actually executed by the application. One way of doing this is to overwrite the `eval` function and watch its inputs at runtime. This task, however, is often hard to do from browser extensions or the JavaScript console due to the lexical scoping of JavaScript. Therefore, overwriting `eval` can only be done effectively within a proxy at the network level or by hooking the native implementation of `eval` within the browser's JavaScript engine.

5.1.1 Observing JavaScript Execution

When it comes to the automated analysis of security vulnerabilities, two basic approaches exist: dynamic analysis, which records the effects of JavaScript execution during runtime, or static analysis, that examines the source code. In case of finding security vulnerabilities, static analysis has the advantage of providing superior coverage but is prone to false positives. Since for this study, we already had a data set of existing real-world vulnerabilities, code coverage was of no concern. Thus, we chose a dynamic technique to precisely monitor the execution of code that causes the vulnerability without false positives or the collection of unrelated information.

When executing JavaScript code, various information can be gathered in the JavaScript engine. In the context of client-side XSS, portions of the code which operate on the attacker-provided string data on its way from the source to the security-sensitive sink are of special interest. In addition, all functions that the data passes through including their invocation contexts and code origin provide further intelligence on a flow. In the following section, we highlight which of these observable characteristics can be measured using a set of presented metrics.

5.1.2 Towards Measuring JavaScript Code Complexity

Based on the discussed observations regarding the complexity of JavaScript, we now derive a set of metrics that aid in measuring how complex a given vulnerable data flow is.

M_1 Number of String-accessing Operations When thinking about the complexity of a flow of data from a source to a sink, one necessary consideration is that each operation on this data naturally increases the complexity as more lines of code have to be understood by the analyst. We therefore define our first metric to be the number of string-accessing operations which occur throughout the flow of data from source to sink. In order to not over-approximate this number, we join the same type of operation on a per-line level, i.e., several concatenating operations on a single line are treated as one operation. Therefore, the example shown in Listing 5.7 constitutes three distinct operations, namely the source access, the joined concatenation of three strings, and the sink access. Listing 5.8 shows the same syntactical content as interpreted by our metric. The smallest possible number of observed operations is two, i.e., source and sink access. As outlined in Section 4.3.5, a sink access may occur with data from several sources, on which a different number of operations may have been conducted. Therefore, we define this metric to measure the longest path from any involved source to the sink.

```
document.write("<a href=' " + location.href + "'>this page</a>")
```

Listing 5.7: Single line example

```
var x = location.href; // source access
var y = "<a href='" + x + "'>this page</a>"; // joined concats
document.write(y); // sink access
```

Listing 5.8: Example as interpreted by our metric

M_2 Number of Functions Involved In addition to the number of operations that are passed when a flow of data occurs, the number of functions which are traversed is an indicator for the complexity of a flow or a vulnerability. Naturally, the analyst has to understand and audit each of those functions to ensure that a flow of user-controlled data is not exploitable. Thus, we define the second metric as the amount of functions that are traversed, whereas the minimum number must always be one.

M_3 Number of Contexts Involved We define an execution context as either the script block or the external file a given piece of codes resides in. The number of such contexts which are involved is therefore relevant to the complexity of a vulnerability, since the analyst has to view all blocks to fully understand the flow. Our third metric thus counts the number of contexts which are involved when executing the complete flow from source to sink.

M_4 Relation of Source to Sink Another factor which potentially makes the analysis of any code harder is the relation between the source of data and the sink it flows into. The term *relation* in this case refers to the positioning of the two operations in the call stack of the executed program. In the following, we describe the five different relations between source and sink which may occur, underlined by Figure 5.1. The top-level `main` is an abstraction of the top execution level of JavaScript and, thus, to be understood as a *virtual main*. All of the following relations are to be interpreted such that the sink access has occurred in the green Snippet #3 of the figure.

The first identified case occurs if both source and sink are accessed within the same function. In the cases, an analyst only has to analyze this one function to determine whether a flow might be dangerous. Thus, this relation, which we refer to as $\mathbf{R_1}$, is shown in the green snippet in Figure 5.1.

The second possible relation between source and sink can be observed if the source access is conducted in the orange box, i.e., Snippet #1. As shown in the figure,

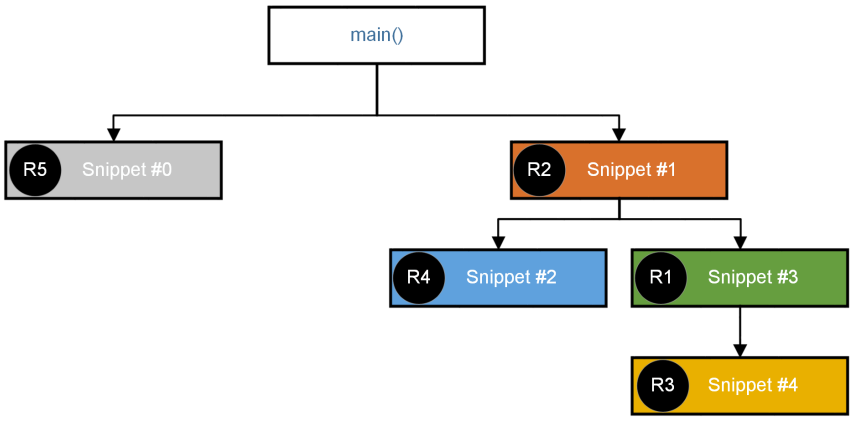

Figure 5.1: Relations between source and sink access

the sink access is conducted in a snippet which is lower in the call stack. In this case, the analyst can observe the potentially dangerous data being passed to the called function and, thus, can investigate said function with the knowledge that the parameters passed are unsanitized (*Source > Sink,* $\boldsymbol{R_2}$).

In the third variant, the source access is lower in the stack than the sink access. In Figure 5.1 this is shown in the yellow box of Snippet #4. Therefore, the analyst has to explore the function containing the sink access, check whether data might be provided by the user and subsequently continue his analysis of the previous function. This complicates the analysis, as the analyst has to switch back and forth between functions, and potentially contexts/files (*Sink > Source,* $\boldsymbol{R_3}$).

As a fourth case, we identify flows in which the source and sink share a common ancestor in the stack, but neither is in the call stack of the other. This scenario occurs if the source was accessed by Snippet #2 and the data ended in a security-sensitive sink in Snippet #3. In these situations, an analyst must follow the path from the common ancestor to the source and up again, subsequently having to analyze the path to the sink. (*Common Ancestor,* $\boldsymbol{R_4}$).

Finally, the most complex type of flow occurs when there is no relation between the sink and the source. An example of such a case is depicted in Listing 5.9. We assume that the external script `register_var.js` assigns any user-provided data (such as the URL) to the global variable `global_var`. As previously discussed, all JavaScript code which is executed within a document shares a global scope. Therefore, the inline script uses `global_var`, which was previously set by the external script, in the call to `document.write`, thus completing the flow. In order to understand that this constitutes a vulnerability, the analyst must examine both the external script files to determine in which of them `global_var` was assigned. In these cases, the operations which access sink and source, respectively, have no common ancestor in the call stack. (*No relation,* $\boldsymbol{R_5}$). This is depicted in Figure 5.1 by the grey Snippet

#0, as both source and sink accessing operations have the virtual main function as their only common ancestor.

```
<script src="/unrelated.js"></script>
<script src="/register_var.js"></script>
<script>
  document.write(global_var);
</script>
```

Listing 5.9: Example outlining the most complex relation between source and sink

M₅ *Code Locality* In order to understand that a certain flow constitutes a vulnerability, an analyst has to inspect all the code between the source and respective sink access. Therefore, if the corresponding instructions have a high code locality, i.e., are within a few lines of one another, this eases his task (and vice versa). Thus, as a fifth metric, we calculate the difference between source and sink access. Naturally, this is only sensible for cases in which both source and sink accessing operation are located in the same file, i.e., in the same external JavaScript or HTML file (spread across one or two script blocks).

5.2 Infrastructure

In the following, we outline the initial data set and associated challenges we had to overcome. Afterwards, we discuss the infrastructure we developed to conduct our study.

5.2.1 Initial Data Set and Challenges

The basis of the results we present in this chapter is a set of exploits detected with the methodology outlined in Chapter 4. Since this data set was slightly outdated, we first verified all of the exploits to identify vulnerabilities which were still in existence. Only verified vulnerabilities, i.e., cases in which a payload existed that reliably lead to JavaScript execution, were included in the data set.

In order to analyze this set of data in a sound and reproducible way, we had to overcome several challenges. First and foremost, interaction with live Web servers can induce variance in the data, as no two responses to the same requests are necessarily the same. Causes for such behavior might reside in load balancing, third-party script rotation or syntactically different versions of semantically identical code. Secondly, to gain insight into the actual vulnerabilities, we needed a means of gathering detailed

information on flows of data, such as all operations which were executed on said data.

Modern Web applications with complex client-side code often utilize minification to save bandwidth when delivering JavaScript code to the clients. In this process, space is conserved by removing white spaces as well as using identifier renaming. As an example, jQuery 2.1.3 can be delivered uncompressed or minified, whereas the uncompressed version is about three times as large as the minified variant. Our analysis, however, requires a detailed mapping of vulnerabilities to matching JavaScript code fragments and, thus, minified code presents another obstacle to overcome.

Finally, in our notion, if access to a sink occurs in jQuery, we assume that this is not actually a vulnerability of that library but rather insecure usage by the Web application's developer. Thus, to not create false positives when determining the cause of a vulnerability, we treat certain jQuery functions, such as `html` or `append`, as a direct sink and remove any following function calls to internal jQuery functions from the trace information we collect.

5.2.2 Persisting and Preparing Vulnerabilities

To allow for a reproducible vulnerability set, we first needed to implement a proxy capable of persisting the responses to all requests made by the browser when visiting a vulnerable site. To achieve this, we built a proxy on top of `mitmproxy` (Cortesi and Hils, 2014) which provides two modes of operation. We initially set the mode to *caching* and crawled all exploits which had previously triggered their payload and stored both request and response headers as well as the actual content. To ensure for proper execution of all JavaScript and, thus, potential additional requests to occur, we conducted the crawl in a real browser rather than a headless engine. Also, this allowed us to send an additional header from the browser to the proxy, indicating what kind of resource was being requested (e.g., HTML documents, JavaScript or images), as content type detection is inherently unreliable (McDaniel and Heydari, 2003).

As previously discussed, our analysis requires precise information on the statements that are executed. In order to ensure that a mapping between all operations which are involved in the flow of data and their corresponding source line can be achieved, we need all JavaScript to be beautified. Therefore, using the information provided by the browsing engine regarding the requested type of content, we first determine the cached files which were referenced as external JavaScript. We use the beautification engine *js-beautify* to ensure that the code is well-formatted and each line consists only of a single JavaScript statement (Daggett, 2013). Subsequently, we parse all HTML on disk, beautifying each script block contained in the files and finally, save the files back to disk.

We now switch the operation mode to *replay*, i.e., all responses are served from disk and not from the original server. To do so, the proxy simply queries its database for a matching URL and returns the content from disk, while attaching the response headers as originally retrieved from the remote server. Some requests that are conducted at runtime by JavaScript (such as jQuery `XmlHttpRequest` with the `JSONP` option) carry a nonce in the URL to avoid a cached response (The jQuery Foundation, 2015b). Therefore, if the proxy cannot find the requested URL in its database, it employs a fuzzy matching scheme which uses normalized URLs to determine the correct file to return. Since our initial tests showed that nonces in all cases consisted only of numbers, we normalize URLs by simply replacing each number in the URL with a fixed value.

5.2.3 Taint-aware Firefox Engine

The taint-aware browsing engine we presented in Chapter 4 allowed us to find client-side Cross-Site Scripting vulnerabilities at scale. Due to memory allocation strategies, we were not able to gather additional information on any given flow, such as the location of the source access or functions which were called with tainted data as parameters. In contrast, Firefox allowed us to allocate additional memory and more easily change the underlying string implementation, such that the engine could gather this additional data on all observed flows.

Apart from our previously presented concept, which allows to pinpoint the source of each character in a string, the taint-enhanced Firefox engine is also able to store a trace of all JavaScript statements which occurred throughout a data flow. To achieve this goal, for each operation which consumes at least one string and produces another string, such as string concatenation, our engine stores a reference to the strings, which were passed as input, in the resulting string resource. This way, at the time of the sink access, the analysis can trace back through all previous strings. Alongside these shadowed strings, the exact JavaScript operation, e.g., function call or concatenation, and source line is accessible, thereby allowing us to collect precise information on all stages of a data flow.

Our detection methodology uses Chromium to find and verify vulnerabilities. Firefox, however, behaves differently with respect to automatically encoding certain DOM values such as the location (Zalewski, 2009a). To be coherent with the previous analyses, we therefore decided to align the implementation of Firefox with Chromium's, i.e., it does not automatically encode the hash fragment of the URL.

5.2.4 Post-processing

Before the raw data gathered by our engine can be analyzed, it needs to be processed to ensure a subsequent, correct analysis. In the following, we illustrate the steps taken by our infrastructure.

Identifying Vulnerable Flows As we outlined in Section 4.3.4, only a fraction of all flows which occur during the execution of a Web application are vulnerable. Our browsing engine collects all flows which occur and sends them to our backend for storage. Thus, before an analysis can be conducted, we need to identify the actual vulnerable flows. We therefore discard all flows which do not end in `document.write`, `innerHTML` or `eval`, as our exploit generation specifically targets these, directly exploitable sinks. In the next step, we determine whether the combination of break out sequence, payload, and comment sequence was completely contained in the string which was passed to the sink, i.e., we ensure that the filtered data only contains actual exploitable flows.

jQuery Detection and Removal One of the most commonly used libraries in many Web applications is jQuery (BuiltWith, 2015; W3Techs, 2015b). It provides programmers with easy access to functionality in the DOM, such as a wrapper to `innerHTML` of any element. When analyzing the reports gathered from our taint-aware browsing engine, the calls to such wrapper functions increase the number of functions traversed by a flow and, thus, increase the perceived complexity. Therefore, we select to filter jQuery functions and use them as sinks, i.e., the `html` and `append` functions of jQuery (The jQuery Foundation, 2015a) are treated like an assignment to an element's `innerHTML` property.

To allow for this process to work properly, we needed to implement a detection mechanism to pinpoint which functions were provided by jQuery. The flow chart in Figure 5.2 shows this process. Initially, we iterate over all entries in the stack traces collected by our taint-aware browser and determine the file in which each line is contained. We then check the hash sum of that file against known jQuery variants (1). If no hash match is found, we utilize the methodology used by *Retire.js* (Oftedal, 2013) to detect whether jQuery is contained in that file at all (2). If this step indicates that the file *contains* jQuery, we proceed to gathering script statistics (such as the number of strings, identifiers and functions) and comparing them to known versions of jQuery, to assess whether the script solely consists of jQuery (3). If this does not produce a conclusive match, we resort to a full-text search of the line of code in a database of all lines of all known versions of jQuery (4). If no match can be found, we mark the generated report for later manual analysis. This effect occurs due to the use of custom packers by Web site owners, which make even a full-text search infeasible.

If any of the aforementioned checks indicate that the analyzed stack entry points to code located within jQuery, we remove said entry from our trace both at the bottom and the top of the stack. This allows to remove artifacts from sink-like operations such as `html` and jQuery-enabled uses of events.

Stack Gap Detection and Removal After this process has been finished, we proceed to the next phase of post-processing. Our taint-aware engine is able to

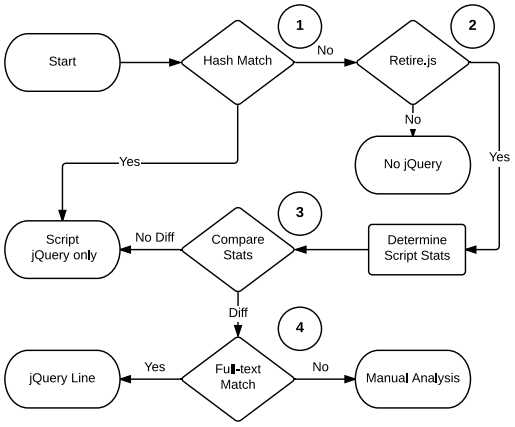

Figure 5.2: Flow chart for jQuery detection process

record all calls to functions to which a tainted string is passed. In some cases, an indirect data flow occurs, i.e., the string is not passed to a function but rather written to a global variable. If another function then accesses the tainted data, the accessing operation is logged; nevertheless, the actual call to said function is missing. An analyst, however, has to follow all function calls to manually track a flow of data. Therefore, we close this *stack gap* by inserting *restored function calls* into the trace, thus allowing a correct analysis, with respect to the amount of involved functions, in the next step.

Operation Fusing To ease further analysis, the final stage of the post-processing is the fusion of concatenating operations on a single line. As we discussed in Section 5.1.2, $\mathbf{M_1}$ aims at determining the number of operations that accessed the tainted data. In terms of understanding a vulnerability, a concatenation on one line does not complicate the analysis; therefore, all consecutive concat operations on the same source code line are fused to form a single operation.

5.2.5 Overall Infrastructure

An overview of our complete infrastructure is depicted in Figure 5.3. Initially, all responses to requests which were conducted when verifying the exploits are stored into a cache database (1). Afterwards, we beautify all HTML and JavaScript files and store them alongside the cache (2). In the next step, our taint-aware Firefox browsing engine visits the cached entries, while the proxy serves beautified version of HTML and JavaScript files as well as unmodified versions of all other files, such as images (3). The engine then collects trace information on all data flows that occur during execution of the page and sends it to a central backend for storage

(4). After the data for all URLs under investigation has been collected, the data is post-processed (5) and can then be analyzed (6).

5.3 Empirical Study

In this section, we outline the execution of our study as well as its results. We then present classification boundaries for the metrics discussed in Section 5.1.2 and present the results of that classification.

5.3.1 Data Set and Study Execution

After having an infrastructure that allowed for persisting and thus consistently reproducing vulnerabilities, we took a set of known vulnerabilities derived with the methodology outlined in Chapter 4. In total, this set consisted of 1,146 URLs in the Alexa top 10,000 domains which contained at least one vulnerability. After persisting and beautifying the vulnerable code, we crawled the URLs with our taint-enhanced Firefox, collecting a total of 3,080 distinct trace reports, whereas a trace report corresponds to *one* access to a sink, potentially consisting of *more than one* flow. Out of these reports, only 1,273 could be attributed to actual vulnerabilities; the rest of the sink accesses either occurred with sanitized data or involved sinks which are not directly exploitable (such as `document.cookie`).

As discussed in Section 5.1.2, a single sink access may use data from several sources. In total, we found that the 1,273 traces our engine gathered consisted of 2,128 pieces of data, whereas the maximum number of involved sources was 35. Note, that in this case data from a single source was used multiple times.

5.3.2 Result Overview

In this section, we present an overview of the results from our study. The presented data in then analyzed in further detail in Section 5.3.4.

Figure 5.3: Overview of our analysis infrastructure

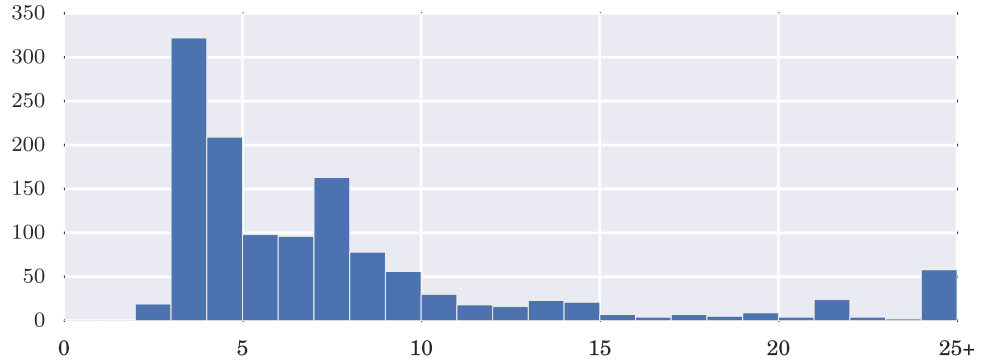

Figure 5.4: Histogram for $\mathbf{M_1}$ (string-accessing operations)

$\mathbf{M_1}$ Number of String-accessing Operations Figure 5.4 shows the distribution of the number of string-accessing operations in relation to the number of vulnerabilities we encountered. Out of the 1,273 vulnerable flows in our data set, we find that 1,040 only have less than 10 operations (including source and sink access). The longest sequence of operations had a total length of 291, consisting mostly of regular expression checks for specific values.

$\mathbf{M_2}$ Number of Involved Functions Apart from the number of contexts, we also studied the number of functions that were involved in a particular flow. We found that 579 flows only traversed a single function, i.e., no function was called between source and sink access. In total, 1,117 flows crossed five or less functions. In the most complex case, 31 functions were involved in operations that either accessed or modified the data. The distribution is shown in Figure 5.5.

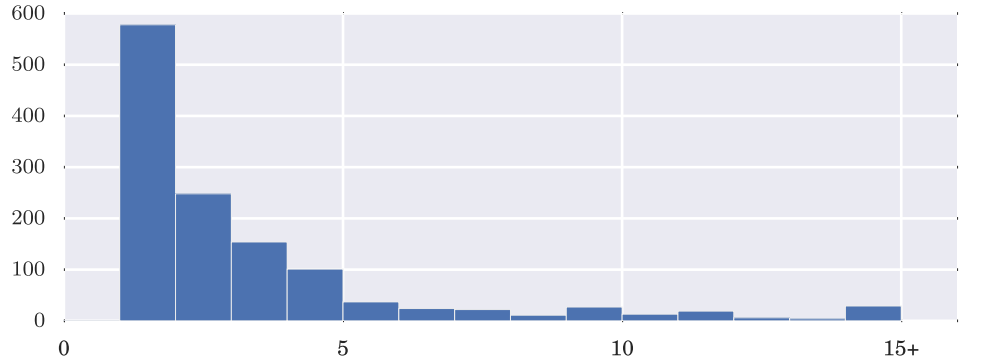

Figure 5.5: Histogram for $\mathbf{M_2}$ (number of functions)

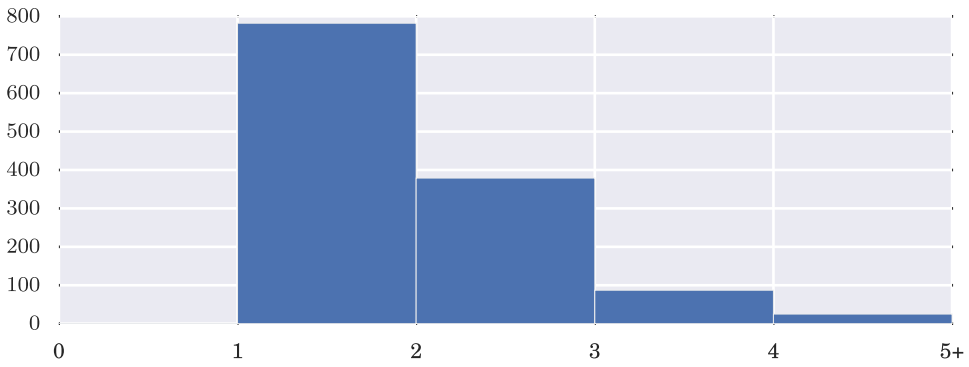

Figure 5.6: Histogram for $\mathbf{M_4}$ (number of contexts)

$\mathbf{M_3}$ Number of Involved Contexts Out of the 1,273 vulnerabilities we analyzed, the exploitable flow was contained within one context in 782 cases, and 25 flows traversed more than three contexts (with the maximum number of contexts being six). Figure 5.6 shows this, highlighting also that more than 90% of the vulnerabilities were not spread across more than two contexts.

$\mathbf{M_4}$ Relation Between Source and Sink In our analysis, we found that the most common scenario was $\mathbf{R_1}$, i.e, source and sink access were contained within one function, which applied to a total of 914 flows. Second to this, in 180 cases, the source was an ancestor of the sink ($\mathbf{R_2}$), whereas this relation was reversed 71 times ($\mathbf{R_3}$). Source and sink shared a common ancestor in 49 flows ($\mathbf{R_4}$) and finally, there was no relation between source and sink in 59 traces ($\mathbf{R_5}$).

$\mathbf{M_5}$ Locality of Source and Sink Access For all vulnerabilities which were contained either in one single external file or inside the same HTML document (in either one or two script blocks), we determined the lines of code between the two operations. Out of the 1,150 reports that matched this criterion, the contained vulnerability was located on a single line in 349 cases, and within ten lines of code 694 times. In contrast, 40 vulnerabilities had a distance of over 500 lines of code between source and sink access. In the most complex case, the access to the source occurred 6,831 lines before the sink access. The distribution of this metric are shown in Figure 5.7.

5.3.3 Additional Characteristics

In addition to the metrics we defined in Section 5.1.2, the gathered data can be analyzed in further dimensions, although these dimensions cannot be used to quan-

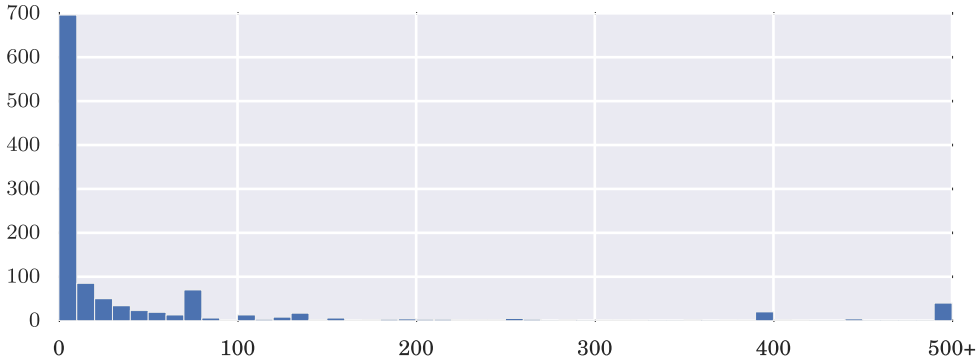

Figure 5.7: Histogram for $\mathbf{M_5}$ (code locality)

tify the complexity of a flow. In the following, we therefore discuss our results with respect to the number of multi flows, the origin of vulnerable code, the sinks which were involved and, subsequently, the amount of runtime-generated code we encountered.

Multi Flows A single source access may contain more than one piece of user-provided data. In our study, we found that 344 traces contained more than one piece of tainted data, i.e., constituted a multi flow. As we discuss in Section 6.1, given the right circumstances, such flows can be used to bypass existing Cross-Site Scripting solutions such as Chrome's XSSAuditor Bates et al. (2010).

Code Origin In terms of origin of the code involved in a flow, we differentiate between three cases: self-hosted by the Web page, code which is only hosted on third-party pages, and a mixed variant of the previous, where the flow traverses both self-hosted and third-party code. To distinguish the involved domains, we use the Alexa service (Alexa Internet, Inc., 2015), which resolves subdomains and Content Delivery Networks (*CDNs*) to their parent domain. Thus, code that is hosted on the CDN of a given site is treated as self-hosted. In total, we found that 835 vulnerabilities were caused purely by self-hosted code, 273 were contained exclusively in third-party scripts and 165 flaws were results of a combination of self-hosted and third-party code.

Sinks We distinguish between three types of sinks, namely `eval`, `innerHTML` and `document.write`. These also contain conceptually similar sinks, such as `setTimeout`, `outerHTML` and `document.writeln`, respectively. These sinks differ in terms of what payload must be injected by an attacker to successfully exploit a vulnerability. In our study, we found that 732 exploitable flows ended in the `document.write` sink, 495 in `innerHTML` and the remaining 46 in `eval` and its derivatives.

Runtime-generated Code Through the use of the `eval` function, JavaScript code can be dynamically created at runtime and executed in the same context. While this enables programmers to build flexible Web applications, it also complicates an analyst's task of understanding a given piece of code. Apart from the exploitable flaws which were the result of insecure use of user-provided data in `eval` as a sink, we also found that eleven traces contained calls to `eval`, i.e., parts of the code which contributed to the vulnerable flow were dynamically generated at runtime.

5.3.4 Analysis

In Section 5.1.2, we defined a set of metrics which aim at measuring the complexity of a vulnerable flow. In the following, we introduce the classification boundaries for these measures; based on these boundaries, we then classify each of the vulnerable flows in our data set to either have a low, medium or high complexity with respect to each metric. Finally, we combine the classification results and highlight the necessity for a multi-dimensional classification scheme.

Quantifying Complexities To better quantify the complexity of a flaw, we need to translate the numeric values derived by our metrics into a classifying scheme. We set boundaries for all of the metrics, such that any value maps to either a low (LC), medium (MC) or high (HC) complexity. The overall complexity is then deduced from the highest rating by any classifier.

The boundaries are derived from our experience with the analysis of client-side XSS vulnerabilities and are shown in Table 5.1 for $\mathbf{M_1}$ (Number of string-accessing operations), $\mathbf{M_2}$ (Number of functions), $\mathbf{M_3}$ (Number of contexts) and $\mathbf{M_4}$ (Relation between source and sink). Note, that since the locality metric ($\mathbf{M_5}$) can only be applied to a subset of the vulnerabilities in our data set, we opted not to add it to our complexity classification.

The results of our classification scheme are depicted in Table 5.2. We observe that for each metric, the classification results are similar for their corresponding *low complexity* cases, assigning this complexity score to around 1,100 vulnerabilities

	LC	**MC**	**HC**
$\mathbf{B_{M1}}$ string-accessing operations	<=10	<=25	25+
$\mathbf{B_{M2}}$ involved functions	<=5	<=10	10+
$\mathbf{B_{M3}}$ involved contexts	<=2	<=3	3+
$\mathbf{B_{M4}}$ source to sink relation	$\mathbf{R_1}$, $\mathbf{R_2}$	$\mathbf{R_3}$, $\mathbf{R_4}$	$\mathbf{R_5}$

Table 5.1: Classification boundaries

	LC	MC	HC
$\mathbf{C_{M1}}$ string-accessing operations	1,071	163	39
$\mathbf{C_{M2}}$ involved functions	1,118	98	58
$\mathbf{C_{M3}}$ involved contexts	1,161	87	25
$\mathbf{C_{M4}}$ source to sink relation	1,094	120	59
Combined	**893**	**236**	**144**

Table 5.2: Classification by applied metrics

each. When combining all classifiers to form the final complexity score, only 893 flows are marked having a low complexity. This highlights the fact that while a flow might be simple in terms of a single metric, flows are actually more complex if all metrics are evaluated.

Correlation between Metrics Our classification scheme aims at combining different classified measurements to produce a single rating of a vulnerability. The underlying metrics, however, are not necessarily orthogonal. Table 5.3 shows a comparison between different context lengths and the minimum, median, and maximum number of string-accessing operations that were conducted. Considering the median value, the number of string-accessing operations appears to be correlated to the number of involved contexts. While this seems to be natural, the maximum number of operations occurred in a flow which only traversed two contexts, highlighting the need for the second classification to mark such a flow as highly complex.

Table 5.4 shows the comparison of the results of metric $\mathbf{M_4}$ to functions and contexts, respectively. There is no correlation between the relation of source to sink and either function or context counts. This again highlights the fact that classification must be done in several different dimensions to get an accurate rating of the complexity of a vulnerable flow.

Contexts	Amount	Minimum	Median	Maximum
1	782	2	4.0	54
2	379	2	6.0	291
3	87	3	7.0	50
4	20	2	7.0	32
5+	5	4	20.0	36

Table 5.3: Contexts v. operations

		Functions			Contexts		
	#	**Minimum**	**Median**	**Maximum**	**Minimum**	**Median**	**Maximum**
$\mathbf{R_1}$	914	1	1.0	21	1	1.0	5
$\mathbf{R_2}$	180	2	4.0	19	1	2.0	5
$\mathbf{R_3}$	71	2	3.0	23	1	2.0	4
$\mathbf{R_4}$	49	3	7.0	31	1	3.0	5
$\mathbf{R_5}$	59	2	3.0	16	1	2.0	6

Table 5.4: $\mathbf{M_4}$ (source to sink relation) v. functions and contexts

Table 5.5 shows the comparison between the rating of complexity using only $\mathbf{M_1}$ (string-accessing operations) and the code locality. Note, that this table only contains all those flows for which source and sink access occurred in the same file. At first sight, the median value of the code locality appear to be related to the classification by $\mathbf{M_1}$, as the larger number of string-accessing operations also yields in more distance between source and sink (in terms of lines of code). In contrast, the case in which these lines were farthest away from each other falls into the LC category. In that specific case, just four operations were conducted, spread across 6,831 lines of code.

Non-linear Flows An additional important observable characteristic of client-side XSS is a classification of the vulnerability's data and control flows in respect to their *linearity*:

In the context of this work, we consider a data flow to be *linear* if on the way from the source to the sink the tainted value is passed to all involved functions directly, i.e., in the form of a function parameter. In consequence, a non-linear data flow includes at least one instance of transporting the tainted value implicitly, e.g., via a global variable or inside a container object. Manual identification of vulnerable data flows in case of non-linear data flows is significantly harder, as no obvious relationship between the tainted data and at least one of the flow's functions exist.

Furthermore, non-linear control flows are instances of interrupted JavaScript execution: A first JavaScript execution thread accesses the tainted data source and stores

$\mathbf{C_{M1}}$	#	**Minimum**	**Median**	**Maximum**
LC	993	0	3.0	6,831
MC	130	0	43.5	3,159
HC	29	7	100.0	1,974

Table 5.5: $\mathbf{C_{M1}}$ (string-accessing operations) v. code locality

	linear control flow	non-linear control flow	Sum
linear data flow	1,116	—[1]	**1,116**
non-linear data flow	98	59	**157**
Sum	**1,214**	**59**	**1,273**

Table 5.6: Data and code flow matrix

it in a semi-persistent location, such as a closure, event handler or global variable, and later on a second JavaScript thread uses the data in a sink access. Instances of non-linear control flows can occur, if the flow's code is distributed over several code contexts, e.g., an inline and an external script, or in case of asynchronous handling of events. Similar to non-linear data flows, the inspection of such flows is significantly more difficult.

Table 5.6 shows the results of that classification for our data set. Note, that a linear data flow cannot occur with a non-linear control flow, since this implies no relation between source and sink accessing operations, and, thus violates the previous stated criterion of a linear data flow, i.e., that the tainted string is always passed on as a parameter. Therefore, the value for non-linear data and control flows matches the value for source/sink relation $\mathbf{R_5}$.

Code Origin and Sink Distribution Table 5.7 depicts the distribution of the origin of the involved code with respect to their complexity classification. Naturally, vulnerabilities for which the flow traverses both self-hosted and third-party code have a higher complexity due to the fact that at the very least two contexts must be involved. This is underlined by the fact that over 20% of these flows are marked to have a high complexity. In contrast, more than 75% of the vulnerabilities caused only by self-hosted code have a low complexity score and only less than 9% are ranked as being highly complex.

[1] A linear data flow cannot occur with a non-linear control flow

	LC	MC	HC	Total
self-hosted	627 *75.1%*	135 *16.2%*	73 *8.7%*	**835**
third-party	173 *63.4%*	65 *23.8%*	35 *12.8%*	**273**
mixed	93 *56.4%*	36 *21.8%*	36 *21.8%*	**165**

Table 5.7: Code Origin v. Complexity Classification

	LC		MC		HC		Total
document.write	568	*77.6%*	113	*15.4%*	51	*7.0%*	**732**
innerHTML	293	*59.2%*	112	*22.6%*	90	*18.2%*	**495**
eval	32	*69.6%*	13	*28.2%*	1	*2.2%*	**46**

Table 5.8: Sink v. Complexity Classification

Additionally, Table 5.8 shows the distribution of sinks with respect to the complexity rating. For all sinks we analyzed, the largest number of flows falls into the low complexity category. Interestingly, only one vulnerability which was caused by `eval` has been ranked as having a high complexity, whereas more than 18% of flows to `innerHTML` are marked as being complex.

Summary of our Findings In summary, we find that when combining all of our classifiers, approximately 70% of all flows expose a low complexity (cp. Table 5.2). Taking into account only single classifiers, we find that this fraction is even larger, which shows that the combination of different metrics is necessary to correctly assess the complexity of a flow. This is, for instance, exhibited in Table 5.5: In one of the analyzed flows, the distance between source and sink spanned 6,831 lines of code, containing only limited number of string altering operations in between. If regarded exclusively under $\mathbf{M_1}$, the vulnerability would have ended in the LC bucket, which does not accurately mirror the vulnerability's character.

5.3.5 Key Insights

While the previous section outlined the classification derived from our metrics in combination with their corresponding classification boundary, it lacked the analysis of the underlying issues. Therefore, in this section, we outline our key insights motivated by a select number of vulnerabilities we encountered in our study, highlighting the causes of client-side XSS vulnerabilities.

Improper API Usage In our data set, we found a vulnerability in the snippet shown in Listing 5.10. In this case, the user-provided data is passed to the outlined function, which apparently aims at removing all `script` tags inside this data. The author of this snippet, however, made a grave error. Even though the newly created `div` element is not yet attached to the DOM, assigning `innerHTML` will invoke the HTML parser. While any `script` tag is not executed when passed to `innerHTML` (Hickson and Hyatt, 2008), the attacker can pass a payload containing an `img` with an error handler (Mozilla Developer Network, 2015b). The HTML parser will subsequently try to download the referenced image and in the case of a

failure, will execute the attacker-provided JavaScript code. While the effort by the programmer is commendable, this filtering function ended up being a vulnerability by itself.

```
function escapeHtml(s) {
  var div = document.createElement('div');
  div.innerHTML = s;
  var scripts = div.getElementsByTagName('script');
  for (var i = 0; i < scripts.length; ++i) {
    scripts[i].parentNode.removeChild(scripts[i]);
  }
  return div.innerHtml;
};
```

Listing 5.10: Improper use of `innerHTML` for sanitization

In addition to this flaw, which was caused by the misinterpretation of the functionality of `innerHTML`, we discovered another interesting code snippet, which is shown in Listing 5.11. The intent of the author of this script is quite clear. To ensure that no Cross-Site Scripting attack can occur, the code aims at replacing all opening HTML brackets (`<`) in the URL with their HTML entity equivalent, `<`. While at first, the presented code appears to achieve just this, the developer made a very subtle mistake. The first parameter to replace in the example is a regular expression, which does not include the global flag. Therefore, the replace function only replaces the *first* match rather than all of them (Mozilla Developer Network, 2015c). Thus, the attacker simply has to prepend an additional `<` to exploit the vulnerability despite the attempted filtering effort. This example, which we discovered on a top 1,000 domain, clearly shows missing knowledge on the part of the developer, underlined also by the fact that jQuery provides a safe means of writing user-provided input into the document, using the `text` function (The jQuery Foundation, 2015a).

```
jQuery("#warning404 .errorURL").html(location.href.replace(/</,"&lt;"))
```

Listing 5.11: Improper use of `replace`

Vulnerable Libraries The popular library jQuery provides a programmer with the `$` selector to ease the access to a number of functions inside jQuery, such as the selection by id (using the # tag) as well as the generation of a new element in

the DOM when passing HTML content to it. Up until version 1.9.0b1 of jQuery, this selector was vulnerable to client-side XSS attacks (jQuery Bug Tracker, 2012), if attacker-controllable content was passed to the function—even if a # tag was hard-coded in at the beginning of that string. Listing 5.12 shows an example of such a scenario, where the intended use case is to call the `fadeIn` function for a section whose name is provided via the hash. This flaw could be exploited by an attacker by simply putting his payload into the hash. In our study, we found that 25 vulnerabilities were caused by this bug, although the vulnerability had been fixed for over three years at time of writing this thesis. This highlights that programmers should regularly check third-party libraries for security updates or only embed the latest version of the library into their pages.

```
var section = location.href.slice(1);
$("#" + section + "_section").fadeIn();
```

Listing 5.12: Vulnerable code if used with jQuery before 1.9.0b1

Incompatible First- and Third-party Code One of the most complex vulnerabilities we encountered utilized meta tags as temporary sinks/sources. Listing 5.13 shows the code, which extracts the URL fragment and stores it into a newly created meta element called `keywords`. Since this code was found in an inline script, we believe that it was put there with intend by the page's programmer.

```
var parts = window.location.href.split("#");
if (parts.length > 1) {
  var kw = decodeURIComponent(parts.pop());
  var meta = document.createElement('meta');
  meta.setAttribute('name', 'keywords');
  meta.setAttribute('content', kw);
  document.head.appendChild(meta);
}
```

Listing 5.13: Creating meta tags using JavaScript

This page also included a third-party script, which for the most part consisted of the code shown in Listing 5.14. This code extracts data from the meta tag and uses it to construct a URL to advertisement. In this case, however, this data is attacker-controllable (originating from the URL fragment) and thus this constitutes

a client-side XSS vulnerability. This code is an example for a vulnerability which is caused by the combination of two independent snippets, highlighting the fact that the combined use of own and third-party code can significantly increase complexity and the potential for an exploitable data flow. In this concrete case, the Web application's programmer wanted to utilize the dynamic nature of the DOM to set the keywords on-the-fly (from user-provided input), while the third-party code provider reckoned that `meta` tags would only be controllable by the site owner.

```
function getKwds() {
  var th_metadata = document.getElementsByTagName("meta");
  ...
}
var kwds = getKwds();
document.write('<iframe src="...&loc=' + kwds + '"></iframe>');
```

Listing 5.14: Third-party code extracting previously set meta tags

Explicit Decoding of Otherwise Safe Data As outlined in Section 5.2.3, the automatic encoding behavior of data retrieved from the `document.location` source varies between browsers: Firefox automatically escapes all components of the URL, while Chrome does not encode the fragment, and IE does not encode any parts of the URL. In consequence, some insecure data flows may not be exploitable in all browsers, with Firefox being the least susceptible of the three, thanks to its automatic encoding.

As mentioned above, the data set underlying our study was validated to be exploitable if Chrome's escaping behavior is present, which leaves the fragment portion of the URL unaltered. Thus, although our taint-aware version of Firefox mirrored the behavior of Chrome, we wanted to investigate how many vulnerabilities would actually work in any browser, i.e., in how many cases data was intentionally decoded before use in a security-sensitive sink. Using an unmodified version of Firefox, we crawled all vulnerable URLs again and found that 109 URLs still triggered our payload. This highlights the fact that programmers are aware of such automatic encoding, but simply decode user-provided data for convenience without being aware of the security implications.

5.4 Summary

After having developed an infrastructure capable of detecting client-side XSS vulnerabilities at scale, this chapter presented the results of an in-depth analysis of

real-world vulnerabilities. To quantify the complexity of the discovered flaws, we derived a set of metrics that allowed us to measure several features of the code under investigation. We enhanced our existing infrastructure to allow for a more comprehensive analysis of the vulnerable code and conducted an empirical study on 1,293 distinct vulnerabilities. Applying the metrics we discussed and combining the complexity scoring assigned by each of the metric's values, we found that while the largest fraction of all vulnerabilities are straight-forward and easy to spot, a significant number of flaws show a high complexity, making them hard to detect even for seasoned security analysts. Finally, this chapter discussed the key insights we gained in our study, showing that non-trivial vulnerabilities are often caused by improper API usage, vulnerable libraries, the combination of incompatible first- and third-party and even explicit decoding of otherwise safe data.

Even though our analysis gives us an understanding of how these issues came into existence, the flaws need to be handled by the developers of the vulnerable applications. In order to also enable a user to have an effective remedy against this class of vulnerability, in the next chapter, we outline an analysis of currently deployed countermeasures against Cross-Site Scripting, showing that they are inadequate to protect users against these attacks. Subsequently, we present the design, implementation and evaluation of an XSS filter that is capable of robustly stopping Client-Side Cross-Site Scripting attacks.

Chapter 6

Precise Client-Side Cross-Site Scripting Protection

After we presented a methodology to find Client-Side Cross-Site Scripting flaws at scale, the previous chapter discussed the root causes of client-side XSS as discovered by an empirical study on a set of 1,273 unique, real-world vulnerabilities. Motivated by our insights on both the prevalence and complexity of such flaws, this chapter presents an analysis of the current generation of client-side XSS filters, aimed at determining how well existing approaches can protect users against this type of attack.

In this chapter, after discovering and discussing several drawbacks of current approaches, we propose the conceptual design of a new XSS filter, specifically aiming at blocking DOM-based Cross-Site Scripting. The discussion of the design is followed by an evaluation of its compatibility with current Web pages, the protection capabilities, and its performance. We conclude this chapter with a discussion of the feasibility of the proposed filtering approach.

6.1 Drawbacks of Current Approaches

In recent years, browser vendors and third-party programmers have developed client-side filters against Cross-Site Scripting attacks. While Firefox does not ship a built-in filter, the extension NoScript, which was first introduced in 2007, not only allows users to completely deactivate JavaScript, but also provides protection against XSS attacks (Maone, 2007). Internet Explorer (Maone, 2008) as well as all WebKit/Blink-based browsers (such as Chrome or Safari) on the other hand run a built-in filter. The latter implement the filter described by Bates et al. (2010) under the name *XSS Auditor*, which we consider to be the best and most widely deployed protection scheme. In the following, we highlight the functionality of the XSS Auditor und discuss several issues which allow an attacker to bypass the protection provided by it.

6.1.1 Inner Workings of the XSS Auditor

In contrast to the concept employed by NoScript and Internet Explorer, which try to detect the injected payload in the outgoing *request*, the XSS Auditor is only invoked when the resulting HTML *response* is parsed. To allow for fast and precise detection of Cross-Site Scripting payloads, it is located inside the HTML parser. A general

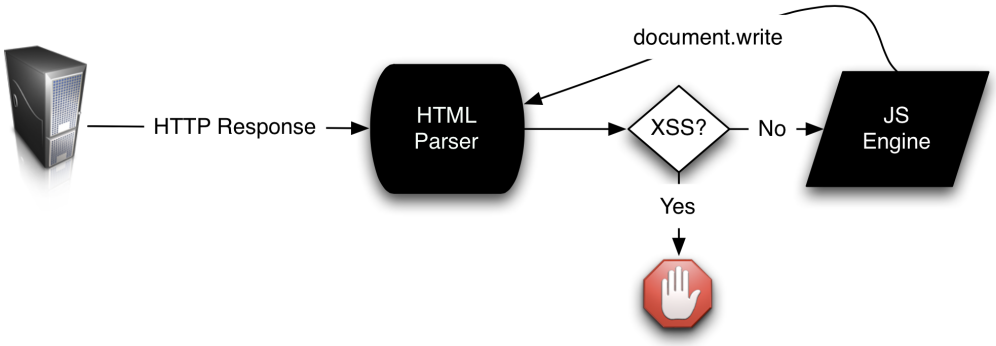

Figure 6.1: Conceptual overview of XSS Auditor (Bates et al., 2010)

overview of the XSS Auditor's concept is shown in Figure 6.1. Since one of the main design goals was performance, the Auditor has several checks in place to ensure that it is not invoked in cases where an attack is unlikely to happen. One of the first checks is whether the request contains characters that are deemed necessary for a successful attack by the Auditor, i.e., `<`, `>`, `‘`, and `"`. If neither of those characters is found, the Auditor is shut off. Also, in order to optimize performance, the Auditor is switched off when parsing so-called *document fragments*, i.e., HTML which is passed from JavaScript using `innerHTML` and its derivatives (Chromium Developers, 2015a). In these cases, the transition between HTML parser and XSS Auditor, as depicted in the center of Figure 6.1, does not occur.

If both these checks are passed, the Auditor is enabled for the current document. During the parsing process, the Auditor checks for potentially dangerous elements, i.e., such elements that allow for JavaScript execution. These can either be injected inline `script` tags, elements with event handlers or elements that can reference external content, e.g., `script` tags which have their `src` attribute set. Depending on the discovered element, the Auditor then applies a set of matching rules to locate the payload in the request (abstracted in the `XSS?` stage of Figure 6.1), and, if the payload is found, replaces the injected code with benign code, such as a call to the `void` function.

When the Auditor decides that no XSS attack has occurred, it passes the discovered script content to the JavaScript engine. If the executed code then uses `document.write` to add additional HTML to the DOM during runtime, the HTML parsing process is re-invoked and, thus, the Auditor's workflow starts again from the beginning.

6.1.2 Limitations of the Auditor

In the following, we highlight properties of the XSS Auditor's design which allow an attacker targeting client-side XSS vulnerabilities to bypass its protection capabilities. These range from conceptual issues due to placement of the Auditor to problems with the employed string matching scheme.

In our work, we discovered several conceptual problems with the Auditor, that allow an attacker to bypass the protection scheme. First and foremost, the Auditor is switched off for snippets of HTML which are passed via `innerHTML`, as these checks incur a significant performance overhead (Kling, 2013). As our results in Sections 4.3.4 and 5.3.4 have shown, a large portion of the discovered vulnerabilities are caused by flows of data to this sink. Thus, any vulnerability that is caused by an insecure flow to this sink allows for a bypass of the filter to begin with.

Similarly, all vulnerabilities caused by the insecure usage of `eval` do not even pass the HTML parser, and therefore are also out of scope for the Auditor. Next to these, Client-Side Cross-Site Scripting may also occur if attacker-controllable data is directly assigned to a DOM property such as the `src` attribute of a `script` tag. As this kind of flow also occurs purely inside the JavaScript engine, the Auditor again has no means of detecting and defending against such an attack.

In order to approximate the flow of data provided by the user back to the rendered document, the Auditor relies on string matching, i.e., tries to find the strings it encountered when parsing the HTML in the original request. Next to the issues caused by performance concerns and placement of the XSS Auditor, we discovered several problems related to this string matching approach. As an example, we consider the snippet shown in Listing 6.15. The `hash` property of the `location` object denotes attacker-controllable input. In a typical attack, the goal of an attacker is to break out of the existing HTML element and append his own `script` element.

```
var val = location.hash.slice(1);
var code = "<script>var x = '" + val + "';</script>";
document.write(code);
```

Listing 6.15: Inscript injection vulnerability

The Auditor assumes such an attacker model and, thus, when detecting any `script` element during HTML parsing, looks for both an opening `script` tag and the contained payload in the request. In this concrete example, the attacker does not need to break out of the existing script context. Rather, he can leverage the fact that his controllable input is inserted between opening and closing `script` tags, by simply breaking out of the assignment of the variable `x` and appending a payload of his

choosing. Let us assume the attacker wants to execute the function `evilcode`; he can thus modify the URL such that the hash fragment's value is `'; evilcode();y='`, i.e., he lures his victim to `http://vuln.com/#'; evilcode();y='`.

Listing 6.16 shows the resulting HTML markup which is passed to `document.write` in this attack scenario. When the HTML parser invokes the XSS Auditor to check whether the `script` tag was injected, the Auditor applies the built-in matching rules and hence, searches for an opening `script` tag in the request, i.e., the URL in this case. Since no such tag is contained in the URL, the Auditor determines that no XSS attack has occurred and passes the code to the JavaScript engine, where the attacker's injected payload is now executed.

```
<script>
  var x=''; evilcode();y='';
</script>
```

Listing 6.16: HTML passed to `document.write` when opening `http://vuln.com/#';evilcode();y='` and executing the code from Listing 6.15 (indented for readability)

Next to this bypass, we discovered a significant number of flaws related to string matching as well as conceptual flaws which assume that user-provided data must be contained in the request. In the following, we only briefly cover these topics, as this analysis is not a major contribution of this thesis and refer the interested reader to Stock et al. (2014b).

Trailing Content When trying to determine if a snippet of JavaScript code was injected by an attacker, the Auditor terminates the string to match at certain comment sequences, such as `//`. These comment characters, however, are also part of the string that is matched against the request. We consider the example shown in Listing 6.17. In this case, a viable option for an attacker is to break out of the existing `script` tag and inject his own, new `script` element. If this script is, however, completely contained in the request, the Auditor blocks it.

```
var path = location.hash.slice(1);
// includes script.js from path on current domain
var code = "<script src='/" + path + "/script.js'></script>";
document.write(code);
```

Listing 6.17: Trailing content vulnerability

In this concrete case, the character which follows the attacker-controllable string is a single forward slash. The attacker can use this *trailing* slash to inject his payload and bypass the Auditor, using the injection vector `'></script><script>alert(1)/`. While this snippet appears like an incomplete `script` tag, Listing 6.18 shows the actual HTML code which is written to the document. Given the trailing slash from the original code, the single, appended slash at the end of our injected vector now completes the comment sequence, essentially commenting out the remaining `script.js'>`.

```
<script src='/'></script><script>alert(1);//script.js'></script>
```

Listing 6.18: Resulting HTML code when exploit the flaw in Listing 6.17

The Auditor, however, searches for the content of the newly injected `script` up to and including both slashes. As discussed above, the request only contains a single slash and, thus, the matching fails, allowing us to bypass the Auditor's protection capabilities.

Double Injections Similar to the previous type of bypass, double injections also enable an attacker to fool the Auditor. In these scenarios, the attacker can control more than one non-consecutive substring in a sink access. Given the right circumstances, he can therefore split up the desired payload, e.g., in multiple GET parameters. As a result, the complete payload is not contained in the request as such and, thus, the matching fails.

Application-specific Input Mutation Another category of bypasses is the mutation of input by the application. Such mutations may occur if the Web application tries to perform filtering. In a concrete example we encountered in our analysis, we found a snippet that would remove all double quotes from a string. In these cases, we simply prepended such a quote to our payload, i.e., `"alert(1)`. While normally, this would cause a parsing error if executed by the JavaScript engine, the mutation allowed us to execute our injected payload. In contrast, the Auditor would in these cases look for an opening `script` tag and the truncated payload `alert(1)`. As this combination is not present in the request, we could successfully fool the matching algorithm.

6.1.3 Evaluating Bypasses of the Auditor

In order to evaluate how well the Auditor performs in detection and blocking of real-world Client-Side Cross-Site Scripting vulnerabilities, we took a set of 1,602 vulnerabilities spread across 958 domains discovered by the methodology outlined

in Chapter 4. While these vulnerabilities had all been discovered when the Auditor was manually disabled, we activated the Auditor to see how many would initially pass, e.g., due to the fact that they exploited `innerHTML` or `eval` flaws.

In addition, we analyzed all the exploits which had been initially caught by the Auditor. Based on the insights we gained from our analysis of the XSS Auditor's code we then specifically crafted our payload such that we could leverage one of the discovered issues. Combining the bypasses caused by the placement of the Auditor with those specifically crafted to fool the string matching algorithms, we were able to exploit 1.168 vulnerabilities on 776 domains, i.e., 73% of all vulnerabilities and 81% of all domains, respectively.

6.1.4 Discussion

As our evaluation shows, the Auditor is not capable of robustly stopping Client-Side Cross-Site Scripting attacks. Although it was never intended to provide protection against this class of attacks, it is the most advanced and widely deployed XSS filter on the Web. Our analysis has highlighted two important, conceptual issues which caused the bypasses: placement of the Auditor and the employed string matching.

Placement Compared to the approaches used by NoScript and Internet Explorer, the Auditor does not rely on regular expression matching on the outgoing request, but gains a significant performance bonus by being part of the HTML parsing engine. It is called before the JavaScript engine can be invoked by attacker-injected code. As we have shown, however, all such flaws that exploit vulnerable code in conjunction with `eval` are out of scope, i.e., the Auditor is not capable of stopping attacks in which HTML parsing is not conducted before execution of the attacker-provided code.

String Matching Even if the placement of the Auditor was to be changed to allow for filtering of JavaScript-based sinks, the string matching algorithm is another conceptual problem of the Auditor. In order to allow for covering of corner cases, several patches have been committed to the Auditor's code base to thwart specific, known attacks (Chromium Developers, 2015b). However, our work has shown that depending on the specifics of a piece of vulnerable code, the Auditor's matching algorithm can be fooled. In total, we discovered 13 such issues which allowed us to bypass the Auditor. Thus, regardless of additional attempts to fix these specific issues, the discussed issues are inherent to string matching and, thus, cannot be tackled by the Auditor.

6.2 Conceptual Design

As we shown before, the XSS Auditor is not able to provide adequate protection against Client-Side Cross-Site Scripting attacks. The problems which allowed us to bypass the Auditor are inherent to its design. Therefore, rather than enhancing the existing approach, we now propose a solution specifically targeting client-side XSS. In the following, we discuss the design and implementation of this new concept.

Cross-Site Scripting as such can be abstracted to an issue related to improper separation of code and data. As we have already outlined in Section 2.3, all types of Cross-Site Scripting attacks have a common denominator: all of them are caused by attacker-provided *data* which ends up being interpreted as *code* by the victim's browser, and more specifically JavaScript engine. This general attack pattern also holds true for all types of injection attacks, such as SQL or command injection.

As we have demonstrated in Section 6.1.2, client-side XSS filters relying on string comparison lack the required precision for robust attack mitigation. String comparison as an approximation of occurring data flows is a necessary evil for flows that traverse the server.

For Client-Side Cross-Site Scripting vulnerabilities, this is not the case. Instead, the full data flow occurs within the browser's engines and can thus be observed precisely. For this reason, we propose an alternative protection mechanism that relies on runtime tracking of data flows and taint-aware parsers and makes use of two interconnected components:

- A taint-enhanced JavaScript engine capable of tracking attack-provided data through the execution of a given document, both in the JavaScript and WebKit realm and
- taint-aware JavaScript and HTML parsers which apply predefined policies on how tainted data may be interpreted.

This way our protection approach reliably spots attacker-controlled data during the parsing process and is able to stop cases in which tainted data alters the execution flow of a piece of JavaScript. In the following, we discuss the general concept and security policy as well as a means of opting out of our protection scheme.

6.2.1 Precise Code Injection Prevention

As we outlined in the previous section, our protection scheme relies on precise byte-level taint tracking. In the following, we give an overview of the necessary changes we performed in order to implement our filtering approach. More specifically, we made changes to the browser's rendering engine, the JavaScript engine and the DOM bindings, which connect the two engines.

JavaScript Engine When encountering a piece of JavaScript code, the JavaScript engine first tokenizes it to later execute it according to the ECMAScript language specification. While it is a valid use case to utilize user-provided data within data values such as String, Boolean or Integer literals, we argue that such a value should never be turned into tokens that can alter a program's control flow such as a function call or a variable assignment. We therefore propose that the tokenization of potentially attacker-provided data should never result in the generation of tokens other than literals. As our JavaScript engine is taint-aware, the parser is always able to determine the origin of a character or a token. Hence, whenever the parser encounters a token that violates our policy, execution of the current code block can be terminated immediately.

Rendering Engine Besides injecting malicious JavaScript code directly into an application, attackers are able to indirectly trigger the execution of client-side code. For example, the attacker could inject an HTML tag, such as the `script` or `object` tag, to make the browser fetch and execute an external script or plugin applet. Hence, only patching the JavaScript engine is not sufficient to prevent DOM-based XSS attacks. To address this issue, we additionally patched the HTML parser's logic on how to handle the inclusion of external content. When including active content we validate the origin of a script's or plugin applet's URL based on our taint information. One possible policy is to reject URLs containing tainted characters. However, as we assess later, real-world applications commonly use tainted data within URLs of dynamically created applets or scripts. Therefore, we allow tainted data within such a remote URL but do not allow the tainted data to be contained either in the protocol or the domain of the URL. The only exemption to this rule is the inclusion of external code from the same origin. In these cases, similar to what the Auditor does, we allow the inclusion even if the protocol or domain is tainted. This way, we make sure that active content can only be loaded from hosts trusted by the legitimate Web application.

DOM Bindings Similar to the previous case, the execution of remote active content can also be triggered via a direct assignment to a `script` or `object` tag's `src` attribute via the DOM API. This assignment does not take place within the HTML parser but rather inside the DOM API. Thus, to ensure that no malicious code can be executed following such an assignment, we patched the DOM bindings to implement the same policy as mentioned above.

Intentional Untainting As our taint-aware browser rejects the generation of code originating from a user-controllable source, we might break cases in which such a generation is desired. A Web application could, for example, thoroughly sanitize the input for later execution. In order to enable such cases, we offer an API to taint and

untaint strings. If a Web application explicitly wants to opt-out of our protection mechanism, the API can be used to completely remove taint from a string.

6.2.2 Handling Tainted JSON

While our policy effectively blocks the execution of attacker-injected JavaScript, only allowing literals causes issues with tainted JSON. Although JavaScript provides dedicated functionality to parse JSON, many programmers make use of `eval` to do so. This is mainly due to the fact that `eval` is more tolerant whereas `JSON.parse` accepts only well-formed JSON strings. Using our proposed policy we disallow tokens like braces or colons which are necessary for the parsing of JSON. In a preliminary crawl, we found that numerous applications make use of postMessages to exchange JSON objects across origin boundaries. Hence, simply passing on completely tainted JSON to the JavaScript parser would break all these applications whereas allowing the additional tokens to be generated from parsing tainted JSON might jeopardize our protection scheme. In order to combat these two issues, we implemented a separate policy for JSON contained within postMessages. Whenever our implementation encounters a string which heuristically matches the format of JSON, we parse it in a tolerant way and deserialize the resulting object. In doing so, we only taint the data values within the JSON string. This way incompatible Web applications are still able to parse JSON objects via `eval` without triggering a taint exception. Since we validated the JSON's structure, malicious payloads cannot be injected via the JSON syntax. If a deserialized object's attributes are used later to generate code, they are still tainted and attacks can be detected. If for some reason our parser fails, we forward the original, tainted value to the postMessage's recipient to allow for backwards compatibility.

6.3 Practical Evaluation

After the implementation of our modified engine as well as the augmented HTML and JavaScript parsers we evaluated our approach in three different dimensions. In this section, we shed light on the compatibility of our approach with the current Web, discuss its protection capabilities, and evaluate its performance in comparison to the vanilla implementation of Chromium as well as the most commonly used others browsers. Finally, we summarize the results of said evaluation and discuss their meaning.

6.3.1 Compatibility

While a secure solution is desirable, it is not going to be accepted by users if the protection mechanism negatively affects existing applications. Therefore, in the following, we discuss the compatibility of our proposed defense to real-world applications. We differentiate between the two realms in which our approach is deployed –

namely the JavaScript parser and the HTML/DOM components – and answer the questions:

1. In what fraction of the analyzed applications do we break at least one functionality?
2. How big is the fraction of all documents in which we break at least one functionality?
3. How many of these false positives are actually caused by vulnerable pieces of code which allow an attacker to execute a Cross-Site Scripting attack?

Analysis Methodology To answer these questions for a large body of domains, we conducted a crawl of the Alexa top 10,000 domains (going down one level from the start page) with our implemented filter enabled. Rather than just blocking execution we also sent a report back to our backend each time the execution of code was blocked. Among the information sent to the backend were the URL of the page that triggered the exception, the exact string that was being parsed as well as the corresponding taint information. Since we assume that we are not subject to a DOM-based XSS attack when following the links on said start pages, we initially count all blocked executions of JavaScript as false positives.

In total, our crawler visited 981,453 different URLs, consisting of a total of 9,304,036 frames. The percentages in the following are relative to the number of frames we analyzed.

Compatibility of JavaScript Parsing Rules In total, our crawler encountered and subsequently reported taint exceptions, i.e., violations of the aforementioned policy for tainted tokens, in 5,979 frames. In the next step, we determined the Alexa ranks for all frames which caused exceptions, resulting in a total of 50 domains. Manual analysis of the data showed that on each of these 50 domains, only one JavaScript code snippet was responsible for the violation of our parsing policy. Out of these 50 issues, 23 were caused by data stemming from a postMessage, whereas the remaining 27 could be attributed to data originating from the URL. With respect to the analyzed data set this means that the proposed policy for parsing tainted JavaScript causes issues on 0.50% of the domains we visited, whereas in total only 0.06% of the analyzed frames caused issues.

To get a better insight into whether these false positive were in fact caused by vulnerable JavaScript snippets, we manually tried to exploit the flows which had triggered a parsing violation. Out of the 23 issues related to data from postMessages, we found that one script did not employ proper origin checking, allowing us to exploit the insecure flow. We then manually analyzed the remaining 27 scripts which had caused a policy violation and found that out of these, we could successfully exploit an additional 21 domains.

Summarizing, our study has shown that the JavaScript parsing rules cause issues with 50 of the domains under investigation, while 22 contain an actual, exploitable vulnerability which triggered the exception.

As we outlined in Section 6.2.2, we allow for postMessages to contain tainted JSON which is automatically selectively untainted by our prototype. To motivate the necessity for this decision, we initially also gathered taint exceptions caused by tainted JSON (stemming from postMessages) being parsed by `eval`. This analysis showed that next to the 5,979 taint exceptions we had initially encountered, 90,937 violations of our policy were reported for JSON from postMessages. On the one hand, this puts emphasis on the necessity for our proposed selective untainting, whereas on the other hand, it also shows that programmers utilize `eval` quite often in conjunction with JSON exchanged via postMessages, even though secure alternatives like `JSON.parse` exist.

Compatibility of HTML Injection Rules As discussed in the previous section, our modified browser refuses to execute external scripts if any character in the domain or protocol is stemming from an untrusted source. Analogous to what we had investigated with respect to the JavaScript parsing policy, we wanted to determine in how many applications we would potentially break functionality when employing the proposed HTML parsing policy. We therefore implemented a reporting feature for *any* tainted HTML and a blocking feature for policy-violating HTML. This feature would always send a report containing the URL of the page, the HTML to be parsed, as well as the exact taint information to the backend. We go into further detail on injected HTML in Section 7.1.3 and now focus on all those tainted HTML snippets which violate the policy we defined in Section 6.2.1.

During our crawl, 8,805 out of the 9,304,036 documents we visited triggered our policy of tainted HTML whereas the 8,805 reports were spread across 73 domains. Out of these, 8,667 violations (on 67 domains) were caused by script elements with `src` attributes containing one or more tainted characters in the domain of the included external script. Out of the remaining six domains, we found that three utilized `base.href` such that the domain name contained tainted characters and thus, our prototype triggered a policy exception on these pages. Additionally, two domains used policy-violating `input.formaction` attributes and the final remaining domain had a tainted domain name in an `embed.src` attribute. In total, this sums up to a false positive rate of 0.09% with respect to documents as well as 0.73% for the analyzed domains.

Subsequently, we analyzed the 73 domains which utilized policy violation HTML injection to determine how many of them were susceptible to a DOM-based XSS attack. In doing so, we found that we could exploit the insecure use of user-provided data in the HTML parser in 60 out of 73 cases.

	blocked documents	blocked domains	exploitable domains
JavaScript	5,979	50	22
HTML	8,805	73	60
DOM API	182	60	8
sum	**14,966**	**183**	**90**

Table 6.1: False positives by blocking component

Compatibility of DOM API Rules As we discussed previously, we also examine assignments to security-critical DOM properties like `script.src` or `base.href` and block them according to our policy. In our compatibility crawl, our engine blocked such assignments on 60 different domains in 182 documents, whereas the largest amount of blocks could be attributed to `script.src`. Noteworthy in this instance is the fact that 45 out of these 60 blocks interfered with third-party advertisement by only two providers.

After having counted the false positive, we yet again tried to exploit the flows that had been flagged as malicious by our policy enforcer. Out of the 60 domains our enforcer had triggered a block on, we verified that eight constitute exploitable vulnerabilities. In comparison to the amount of exploitable blocks we had encountered for the JavaScript and HTML injection polices this number seems quite low. This is due to the fact that both the aforementioned advertisement providers employed whitelisting to ensure that only script content hosted on their domains could be assigned. In total, this sums up to 0.60% false positives with respect to domains and just 0.002% of all analyzed documents.

Summary In this section, we evaluated the false positive rate of our filter. In total, the filtering rules inside the JavaScript parser, the HTML parser and the security-sensitive DOM APIs caused issues on 14,966 document across 183 domains. Considering the data set we analyzed this amounts to a false positive ratio of 0.16% for all analyzed documents and 1.83% for domains. Noteworthy in this instance is, however, the fact that out of the 183 domains on which our filter *partially* impaired the functionality, 90 contained actual verified vulnerabilities in just that functionality. Table 6.1 shows the distribution of blocked documents and domains over the different policy-enforcing components as well as the amount of domains in which the blocked functionality caused an exploitable vulnerability.

6.3.2 Protection

To ensure that our protection scheme does not perform worse than the original Auditor, we re-ran all exploits that successfully triggered when the Auditor was disabled. All exploits were caught by the JavaScript parser, showing that our scheme is at least as capable of stopping DOM-based Cross-Site Scripting as the Auditor.

To verify the effectiveness of our proposed protection scheme, we ran all generated exploits and bypasses against our newly designed filter. To minimize side-effects, we also disabled the XSS Auditor completely to ensure that blocking would only be conducted by our filtering mechanism. As we discussed in Section 6.1.2, alongside the scoping-related issues that were responsible for the successful bypassing of the Auditor by the first generation of exploits, other issues related to string matching arose. In the following, we briefly discuss our protection scheme with respect to stopping these kinds of exploits.

Scoping: eval and innerHTML In contrast to the XSS Auditor, our filtering approach is fully capable of blocking injections into `eval` due to the fact that it is implemented directly in the JavaScript parser. Another issue that impairs the protection capabilities is the fact that `innerHTML` is not checked for performance reasons. In our implementation, conducting this check in the HTML parser is not necessary. To check whether a given token was generated from a tainted source, a simple boolean flag has to be checked. Therefore, our engine does not incur similar performance overhead issues and can robustly block exploits targeting `innerHTML`.

Injection Attacks Targeting DOM APIs In our experiments, we did not specifically target the direct assignment to security-critical DOM API properties. Inside the API, analogous to the HTML parser, assignment to one of these critical properties might cause direct JavaScript execution (such as a `javascript:` URL for an `iframe.src`) or trigger loading of remote content. For the first case, our taint tracking approach is capable of persisting the taint information to the execution of the JavaScript contained in the URL and hence, the DOM API does not have to intervene. For the loading of remote content, the rules of the HTML parser are applied, disallowing the assignment of the property if the domain name contains tainted characters.

Partial Injection One of the biggest issues, namely partial injection, was stopped at multiple points in our filter. Depending on the element and attribute which could be hijacked, the attack vector either consisted of injected JavaScript code or of an URL attribute used to retrieve foreign content (e.g. through `script.src`). For the direct injection of JavaScript code, the previously discussed JavaScript parser was able to stop all exploit prototypes whereas for exploits targeting URL attributes, the taint-aware HTML parser successfully detected and removed these elements, thus stopping the exploit.

Trailing Content and Double Injection The bypasses which we categorized as trailing content are targeting a weakness of the Auditor, specifically the fact

that it searches for completely injected tags whereas double injection bypasses take advantage of the same issue. Both trailing content and double injections can be abused to either inject JavaScript code or control attributes which download remote script content. Hence, analogous to partial injection, the filtering rules in the HTML and JavaScript parsers could, in conjunction with the precise origin information, stop all exploits.

Second-Order Flows and Alternative Attack Vectors Similar to injection attacks targeting the direct assignment of DOM properties through JavaScript, we did not generate any exploits for second order flows. Nevertheless, we persist the taint information through intermediary sources like the WebStorage API. Therefore, our prototype is fully capable of detecting the origin of data from these intermediary sources and can thus stop these kinds of exploits as well. As for postMessages, `window.name` and `document.referer`, our implementation taints all these sources of potentially attacker-controlled data and is hence able to stop all injection attacks pertaining to these sources.

Application-Specific Input Mutation Our engine propagates taint information through string modification operations. Therefore, it does not suffer the drawbacks of current implementations based on string matching. All exploits targeting vulnerabilities belonging to this class were caught within our HTML and JavaScript parsers.

6.3.3 Performance

In order to evaluate the performance of our implementation we conducted experiments with the popular JavaScript benchmark suites Sunspider, Kraken, and Octane as well as the browser benchmark suite Dromaeo. Sunspider was developed by the WebKit authors to "focus on short-running tests [that have] direct relevance to the web" (Pizlo, 2013). Google has developed Octane which includes "5 new benchmarks created from full, unaltered, well-known web applications and libraries" (Cazzulani, 2012). Mozilla has developed Kraken which "focuses on realistic workloads and forward-looking applications" (Jostedt, 2010). Dromaeo, which is a combination of several JavaScript and HTML/DOM tests, finally serves as a measure of the overall impact our implementation has on the everyday browsing experience.

All tests ultimately lead to a single numerical value, either being a time needed for a run (the lower the better) or a score (the higher the better), reflecting the performance of the browser under investigation. For runtime (score) values the results were divided by the values obtained for the unmodified version of the Web browser (vice versa). These serve as the baseline for our further comparisons. With the obtained results we computed a slowdown factor reflecting how many times slower our modified version is. To set these numbers into context, we also evaluated other popular Web browsers, namely Internet Explorer 11 and Firefox 26.0. To eliminate side effects of, e.g., the operating system or network latency, we ran each of the

	Kraken (ms)			Sunspider (ms)		
Baseline	1418.9	*94.29*	–	169.02	*0.37*	–
Tainted Best	1653.1	*59.84*	**1.16**	178.03	*0.70*	**1.05**
Tainted Worst	1814.4	*64.55*	**1.27**	192.66	*0.26*	**1.14**
Firefox 26.0	1291.3	*1.14*	**0.91**	171.86	*0.65*	**1.02**
IE 11	1858.5	*4.16*	**1.31**	78.05	*0.13*	**0.46**

Table 6.2: Benchmark results for Kraken and Sunspider, showing mean, *standard error* and **slowdown factor** for all browsers

benchmarking suites locally ten times using an Intel Core i7 3770 with 16GB of RAM. All experiments, apart from Internet Explorer, were conducted in a virtual machine running Ubuntu 13.04 64-bit whereas IE was benchmarked natively running Windows 7.

Tables 6.2 and 6.3 show the results of our experiments. To ascertain a baseline for our measurements we ran all benchmarks on a vanilla build of Chromium. The table shows the mean results (in points or milliseconds) as well as the standard error and the slowdown factor for each test and browser. Internet Explorer employs an optimization to detect and remove dead code, causing it to have significantly better performance under the Sunspider benchmark than the other Web browsers (Windows Internet Explorer Engineering, 2010). As the results generated by all browsers under the Kraken benchmark were varying rather strongly, we ran the browsers in our virtual machines 50 times against the Kraken benchmark. Regardless, we still see a rather high standard error of the mean for all the browsers.

We chose the aforementioned tests because they are widely used to evaluate the performance of both JavaScript and HTML rendering engines. These tests are, however, obviously not designed to perform operations on tainted strings. Our engine usually only switches from the highly optimized generated code to the slower run-

	Dromaeo			Octane		
Baseline	1167.4	*1.89*	–	20177.9	*64.47*	–
Tainted Best	1082.6	*2.40*	**1.08**	19851.0	*54.54*	**1.01**
Tainted Worst	1015.6	*1.93*	**1.15**	18168.7	*70.24*	**1.11**
Firefox 26.0	721.7	*2.94*	**1.62**	16958.5	*97.40*	**1.19**
IE 11	607.0	*2.13*	**1.92**	17247.2	*47.15*	**1.17**

Table 6.3: Benchmark results for Dromaeo and Octane, showing mean, *standard error* and **slowdown factor** for all browsers

time C++ implementation if an operation is conducted on a tainted string. In the initial runs, which is denoted in Table 6.2 and Table 6.3 as *Tainted Best*, our engine incurred slowdown factors of 1.08, 1.01, 1.16 and 1.05, resulting in an average slowdown factor of 7%. Since the tests are not targeting the usage of tainted data, we conducted a second run. This time we modified our implementation to treat *all* strings as being tainted, forcing it to use as much of our new logic as possible. In this, the performance was naturally worse than in the first run. More precisely, by calculating the average over the observed slowdown factors for our modified (denoted as *Tainted Worst*) version, we see that our implementation incurs, in the worst case, an overhead of 17% compared to the vanilla version. Although the performance hit is significant, other popular Web browsers have an even higher slowdown factor of at least 1.19.

6.3.4 Discussion

In this section we evaluated compatibility, protection capability as well as performance of our proposed filter against Client-Side Cross-Site Scripting. In the following, we briefly discuss the implications of these evaluations.

In our compatibility crawl we found that 183 of the 10,000 domains we analyzed had one functionality that was incompatible with our policies for the JavaScript parser, the HTML parser and the DOM APIs. Although this number appears to be quite high at first sight it also includes 90 domains on which we could successfully exploit a vulnerability in just the functionality that was blocked by our filter. On the other hand, the total number on domains which our approach protected from a DOM-based XSS attack amounts to 958. Although the XSSAuditor is not designed to combat DOM-based XSS attacks, it is the only currently employed defense for Google Chrome against such attacks. As we discussed in Section 6.1.3, the Auditor could be bypassed on 81% of these domains, protecting users on only 183 domains in our initial data set. This shows that with respect to its protection capabilities our approach is more reliable than currently deployed techniques.

Apart from reliable protection and a low false positive rate, one requirement for a browser-based XSS filter is its performance. Our performance measurements showed that our implementation incurs an overhead between 7 and 17%. Chrome's JavaScript engine V8 draws much of its superior performance from utilizing so-called *generated code*, i.e., ASM code generated directly from macros. To allow for a small margin for error, we opted to implement most of the logic — such as copying of taint information — in C++ runtime code. We therefore believe that an optimized implementation making more frequent use of said generated code would ensure better performance.

Our approach only aims at defeating DOM-based Cross-Site Scripting while the Auditor's focus is on reflected XSS. We therefore believe that deployment besides the Auditor is a sound way of implementing a more robust client-side XSS filter,

capable of handling both reflected and DOM-based XSS. Also, compared to other countermeasures such as the *Content-Security Policy* (World Wide Web Consortium, 2015), the burden of the proper implementation of a security mechanism is taken from the developers of many Web applications to a few highly skilled browser vendor programmers. Although approaches such as deDacota (Doupé et al., 2013) exist which try to automate the process of re-writing applications to be compliant with the Content-Security Policy, this task is very hard to accomplish for all vulnerable applications we discovered in our study.

6.4 Summary

After the previous chapters covered attack scenarios of Cross-Site Scripting and analyzed the prevalence and nature of Client-Side Cross-Site Scripting vulnerabilities on real-world Web sites, this chapter first discussed an analysis of the state of the art in Cross-Site Scripting filtering, namely the XSS Auditor. After having discovered numerous conceptional flaws which allow us to bypass the Auditor on more than 80% of the vulnerable domains in our data set, we presented the design, implementation and evaluation of a Cross-Site Scripting filter specifically targeting client-side XSS. In contrast to other, currently deployed, approaches, we rely on precise taint tracking to observe the flow of attacker-controllable data throughout an application into a security-critical sink, and, finally, to the JavaScript engine. This precision is enabled by the fact that all data flows with respect to Client-Side Cross-Site Scripting vulnerabilities occur within the browser and are thus trackable. In the JavaScript engine, we apply a set of predefined policies to only allow tainted data to produce boolean, numeric or string literals. Similarly, we deploy policies in the HTML parser and DOM APIs which ensure that script content retrieved from attacker-controllable URLs may not be executed. This allows for a precise and robust protection against Client-Side Cross-Site Scripting in the browser. Although the approach adds overhead in both memory consumption and execution runtime, it still outperforms any other browser without additional protection capabilities. Apart from that, the potential for performance optimization exists, as our approach was designed as a proof-of-concept rather than a production-ready browser.

This chapter concludes the research conducted throughout this thesis. In the following chapter, we discuss limitations of the approaches described in this thesis and, based on these, present potential future work. Following this, we finish our work with a summary of our contributions and conclude.

Chapter 7

Future Work and Conclusion

In this chapter, we discuss limitations of the presented work and potential future work based on these limitations. We do so by outlining means of enhancing our detection scheme (presented in Chapter 4), analysis metrics (Chapter 5) and discuss the feasibility of using the taint-aware Cross-Site Scripting filter, which we presented in Chapter 6, to not only block Cross-Site Scripting, but also arbitrary HTML injection attacks. We end the chapter and this thesis with a conclusion of the research explored throughout our work.

7.1 Limitations and Future Work

The approaches presented throughout this thesis have certain limitations. In the following, we present these limitations, which allow us to propose future work aiming at tackling these issues.

7.1.1 Enhancing Vulnerability Detection

Our analysis methodology to discover client-side XSS flaws on real-world Web sites does not employ any techniques to ensure code coverage. Instead, the browsing engine recursively follows the links it discovers in the analyzed pages, without attempting to conduct meaningful interaction (such as clicking elements and thus, triggering event handlers) with the page under investigation. Similarly, we have anecdotal evidence of vulnerabilities which are only triggered if a certain parameter exists in the URL of the analyzed document.

Thus, we believe that our approach could be extended by combining it with analysis techniques like static code analysis (Tripp et al., 2014). Another interesting approach was followed by Kolbitsch et al. (2012) for ROZZLE: they utilize pseudo-symbolic execution to cover both sides of branches dependent on the `navigator` object with the goal of detecting malware which targets specific browser and plugin versions. After all paths have been covered, they use a solver to determine versions which were targeted by the malware. This approach could be adopted to branches dependent on attacker-controllable data, allowing for a higher code coverage.

Web pages often use content management systems, that allow the administrator to set-up templates which can be populated with content later on. We consider a news site, which typically has a small number of templates, e.g., for news, live tickers or

image galleries. If such a site is analyzed and all links are followed, the probability of analyzing the same code multiple times is high, as the delivered JavaScript code is most likely dependent on the template rather than the content, e.g., of a news post. Thus, an interesting extension of our approach is to build an oracle capable of determining whether a given link might be of interest to an analysis. Initial, unpublished results have shown that the style properties and position of anchors on a Web page are good indicators for the type of content they link to. An in-depth investigation of these and other features is therefore desirable.

Another limitation of our detection approach is the fact that it allows us to only analyze documents which are accessible without an active login session to the hosting site. An example for an application in which a significantly smaller portion of JavaScript is executed when a user is not logged in is Google Mail: not a single external JavaScript file is included when a user is not logged in, whereas 23 files containing over 400kb of code are retrieved in case of an active login session. A promising approach to solving this problem is the utilization of so-called *social logins*. Web pages may opt to implement their user management based on a user's account on a social network, such as Facebook, Twitter or Google+. By implementing an automated process of discovering such logins and subsequently logging into such applications, the amount of covered code may be increased. Initial results have shown the approach bears its own issues: many sites only use the data provided by the social network to gather information (such as the name or email address) on the user, who subsequently has to register an account with the target site (Mueck, 2015). The process of completing this second phase of the registration is more complex, as it is specific for each application under investigation rather than just for the involved social login providers. Thus, we believe that the investigation of a means of achieving this second stage in a generalized manner is worthwhile.

7.1.2 Observing Execution Traces

In our study into the nature of Client-Side Cross-Site Scripting vulnerabilities, we analyzed the number of operations which were executed between source and sink access as well as the lines of code between these operations. While both these numbers allow us to approximate the number of lines of code which an analyst has to audit, a better indicator might be the amount of code which is actually executed between source and sink access. Therefore, instrumentation of the code (which is persistently stored by our proxy) appears to be a valid extension of our work.

In initial tests, however, we found that blindly instrumenting all JavaScript code is not feasible. We discovered that when instrumenting large libraries such as jQuery, a significant runtime and memory overhead occurs, even leading to a completely unresponsive browsing window. Thus, in order to apply instrumentation techniques to our code base, a strategy must be derived which allows for a sound, yet undisturbed analysis of the lines of executed code.

7.1.3 Blocking HTML Injection

The filtering approach we presented in Chapter 6 allows us to precisely block injected JavaScript. Apart from the false positives we discussed in Section 6.3.4, our implementation lacks the ability to defend against other classes of injection attacks, such as HTML injection. While this class of attack does not allow for direct execution of attacker-provided code, it might be abused to conduct phishing attacks, i.e., by injecting an apparent login form which posts the entered credentials to the attacker's server. Therefore, a viable path for future research is to use the proposed taint tracking approach to also block HTML injections. In the following, we discuss initial, promising results we gathered in our study presented in Chapter 6.

As previously discussed, our engine allows for precise tracking of tainted data during the execution of a program and, hence, also to the HTML parser. Therefore, our approach also enables the browser to precisely block all attacker-injected HTML even it is not related to Cross-Site Scripting. Although this was out of scope for this work, we believe that it is relevant future work. Therefore, we give a short glimpse into the current state of the Web in respect to partially tainted HTML passed to the parser.

As outlined in Section 6.3.1, we conducted a compatibility crawl of the Alexa Top 10,000 in which we analyzed a total of 9,304,036 documents, out of which 632,109 generated 2,393,739 tainted HTML reports. Typically, each of the HTML snippets contained the definition of more than one tag. In total we found that parsing the snippets yielded in 3,650,506 tainted HTML elements whereas we consider an element tainted if either the tag name, any attribute name or any attribute value is tainted. Considering the severity of attacker-controllable HTML snippets, we distinguish between four types:

1. Tag injection: the adversary can inject a tag with a name of his choosing.
2. Attribute injection: injection of the complete attribute, i.e., both name and value
3. Full attribute value injection: full control over the value, but not the name
4. Partial attribute value injection: attacker only controls part of the attribute

We analyzed the data we gathered in our crawl to determine whether blocking of tainted HTML data is feasible and if so, with what policy. Our analysis showed that out of the Top 10,000 Alexa domains, just one made use of full tag injection, injecting a `p` tag originating from a postMessage. This leads us to believe that full tag injection with tainted data is very rare and not common practice. The analysis also unveiled that the most frequently tainted elements – namely `a`, `script`, `iframe` and `img` – made up for 3,503,655 and thus over 95% of all elements containing any tainted data. Hence, we focused our analysis on these and examined which attributes were tainted. Analogous to our definition of a tainted element, we consider an attribute

	full attr. value		partial attr. value	
	Top 10k	*all*	*Top 10k*	*all*
`iframe.src`	349	2,222	384,946	438,415
`script.src`	4,215	8,667	1,078,015	1,292,046
`a.href`	124,812	133,838	1,162,093	1,191,598
`img.src`	5,128	6,791	275,579	312,033
Domains	799	1,014	4,446	6,772

Table 7.1: Amounts of full and partial value injection for domains in the Alexa Top 10,000 and beyond.

to be tainted if either its name or value contains any tainted characters. Considering this notion, we – for each of the four elements – ascertained which attribute is most frequently injected using tainted data. For `a` elements, the most frequent attribute containing tainted data was `href` whereas `script`, `iframe` and `img` tags mostly had tainted `src` attributes. Although we found no case where the name of an attribute was tainted, we found a larger number of elements with full attribute value injection. The results of our study are depicted in Table 7.1, which shows the absolute numbers of occurrences. We also gathered reports from documents on domains not belonging to the Alexa Top 10,000 as content is often included from those. The first number in each column gives the amount for documents on the Alexa Top 10,000, whereas the second number shows the number for all documents we crawled.

Summarizing, we ascertain that taint tracking in the browser can also be used to stop HTML injection. Our study on tainted HTML content on the Alexa Top 10,000 domains has shown that blocking elements with tainted tag names is a viable way of providing additional security against attacks like information exfiltration Chen et al. (2012) while causing just one incompatibly. We also discovered that the applications we crawled do not make use of tainted attribute names, hence we assume that blocking tainted attributes does also not cause incompatibilities with the current Web. In contrast, blocking HTML that either has fully or partially tainted attribute values does not appear to be feasible since our analysis showed that 8% of all domains make use of fully tainted attribute values whereas more than 44% used partially tainted values in their element's attributes. As there is an overlap between these two groups of domains, the total number of domains that would causes incompatibilities is 4,622, thus resulting in more than 46% incompatibilities. Thus, we established that although blocking HTML is technically possible with our implementation this would most likely break a large number of applications.

7.1.4 Holistic Approach to Cross-Site Scripting Filters

In this work, we presented the design, implementation and evaluation of a Cross-Site Scripting filter which relies on precise taint tracking to block client-side XSS attacks. The approach is tailor-made for the attack scenario of a client-side data flow. However, the only restriction that this concept dictates is the existence of taint information on attacker-controllable data. We therefore believe that the concept can be extended to the server, such it acts as a "temporary sink", i.e., is able to persistent taint data. Server-side taint tracking has been proposed to solve a number of security issues, such as SQL injections (Son et al., 2013) and Cross-Site Scripting (Vogt et al., 2007; Xu et al., 2006). Therefore, we believe that an extension of our approach towards the server side is feasible. In this case, all user-provided input, i.e., the complete request including all headers, could be marked as tainted on the server side and tracked throughout execution of the server-side code. Whenever the HTTP response is sent out to the client, an additional header containing information on the tainted characters can be attached, allowing the client to recover the taint information. Subsequently, execution of JavaScript can commence inside our engine and protection can be achieved using the filter we proposed in Chapter 6.

7.2 Conclusion

This thesis conducted research in the area of Cross-Site Scripting, the most widespread class of vulnerabilities on the Web's client side. After discussing the technical background in Chapter 2, we presented a new attack scenario in which XSS is used to steal stored credentials from browser-based password managers in Chapter 3. Motivated by our findings, we proposed a new concept of password managers which are not prone to such attacks, by ensuring that clear text credentials are not accessible from JavaScript. This allowed us to stop all these attacks without inhibiting the functionality of real-world sites.

After we exemplified the impact a Cross-Site Scripting flaw might have, we focussed specifically on Client-Side Cross-Site Scripting, investigating the prevalence and nature of such vulnerabilities and proposing an XSS filter capable of robustly stopping attacks targeting such flaws. Using the combination of a taint-aware browsing engine and an exploit generator, we conducted a study aimed at finding Client-Side Cross-Site Scripting vulnerabilities on the Web at scale. In doing so, we discovered that at least 9.6% of the 5,000 highest ranked domains contain one or more such flaws, putting emphasis on the impact of such attacks.

Motivated by the high prevalence of these vulnerable code snippets, we then conducted an in-depth analysis of the flaws, aimed at discovering the underlying causes for this class of vulnerability. Therefore, we derived a set of metrics to measure the complexities inherent to JavaScript and Client-Side XSS in particular, classifying about 70% of all flaws to be of very simple nature. In doing so, we found that the

reasons are manifold, ranging from developers who are clearly unaware of the dangers of utilizing unfiltered, user-provided data and who lack knowledge of the exact functionality of JavaScript and browser APIs to incompatibilities between first- and third-party code.

Finally, we investigated whether currently deployed filtering mechanisms against Reflected Cross-Site Scripting provide adequate protection to also stop Client-Side Cross-Site Scripting exploits. Our analysis highlighted several conceptional issues related to the design of the state of the art in XSS filtering, the XSS Auditor, allowing us to bypass its protection capabilities on more than 80% of the vulnerable domains in our data set. Based on the identified conceptual flaws, we proposed a new design for a filtering approach capable of robustly stopping client-side XSS attacks, which relies on precise data flow tracking and taint-aware HTML and JavaScript parsers, ensuring that attacker-injected code is detected before it can be executed. We built a proof-of-concept of our proposal into Chromium, showing that it incurs a low performance overhead and few false positives, while effectively stopping all exploits in our data set. We therefore believe that it is feasible to adopt the concept to Chromium and deploy it side-by-side with the XSS Auditor, allowing the combined detection and stopping of both Reflected and Client-Side Cross-Site Scripting attacks.

Bibliography

Alexa Internet, Inc. (2015). Alexa Global Top Sites. online, `http://www.alexa.com/` (Last accessed 16/04/15), 2015.

Allen, Jon (1989). *Perl Programming Documentation*, 1989.

Barth, Adam (2011). The Web Origin Concept. RFC, online, `https://www.ietf.org/rfc/rfc6454.txt` (Last accessed 16/04/15), 2011.

Bates, Daniel; Barth, Adam; Jackson, Collin (2010). Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010* (pp. 91–100)., 2010.

Berners-Lee, Tim (1989). Information management: A proposal, 1989.

Bisht, Prithvi; Venkatakrishnan, V. N. (2008). XSS-GUARD: Precise Dynamic Prevention of Cross-Site Scripting Attacks. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA 2008, Paris, France, July 10-11, 2008* (pp. 23–43).: Springer, 2008.

Bojinov, Hristo; Bursztein, Elie; Boyen, Xavier; Boneh, Dan (2010). Kamouflage: Loss-Resistant Password Management. In *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS 2010), Athens, Greece, September 20-22, 2010* (pp. 286–302).: Springer, 2010.

Bosman, Erik; Slowinska, Asia; Bos, Herbert (2011). Minemu: The World's Fastest Taint Tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011* (pp. 1–20).: Springer, 2011.

BuiltWith (2015). jQuery Usage Statistics. online, `http://trends.builtwith.com/javascript/jQuery` (Last accessed 16/04/15), 2015.

Cazzulani, Stefano (2012). Octane: the JavaScript benchmark suite for the modern web. online, `http://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html` (Last accessed 16/04/15), 2012.

Chen, Eric Y; Gorbaty, Sergey; Singhal, Astha; Jackson, Collin (2012). Self-exfiltration: The dangers of browser-enforced information flow control. In *Web 2.0 Security and Privacy (W2SP 2012), San Francisco, California, USA, May 24, 2012*, 2012.

Chiasson, Sonia; van Oorschot, Paul C.; Biddle, Robert (2006). A Usability Study and Critique of Two Password Managers. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*: USENIX Association, 2006.

Chin, Erika; Wagner, David (2009). Efficient character-level taint tracking for Java. In *Proceedings of the 6th ACM Workshop On Secure Web Services, SWS 2009, Chicago, Illinois, USA, November 13, 2009* (pp. 3–12)., 2009.

Chromium Developers (2009). login_database.cc. Source code, online, `http://goo.gl/6bjgvB` (Last accessed 16/04/15), 2009.

Chromium Developers (2015a). HTMLDocumentParser.cpp source code. online, `https://chromium.googlesource.com/chromium/blink.git/+/master/Source/core/html/parser/HTMLDocumentParser.cpp#639` (Last accessed 16/04/15), 2015.

Chromium Developers (2015b). XSSAuditor.cpp Log. online, `https://chromium.googlesource.com/chromium/blink.git/+log/master/Source/core/html/parser/XSSAuditor.cpp` (Last accessed 16/04/15), 2015.

Connolly, Dan (1992). HyperText Markup Language. online, `http://www.w3.org/History/19921103-hypertext/hypertext/WWW/MarkUp/Connolly/MarkUp.html` (Last accessed 16/04/15), 1992.

Cook, Steven (2003). A Web developers guide to cross-site scripting. SANS Whitepaper, `http://www.sans.org/reading-room/whitepapers/securecode/web-developers-guide-cross-site-scripting-988`, 2003.

Cortesi, Aldo; Hils, Maximilian (2014). mitmproxy. online, `https://mitmproxy.org/index.html` (Last accessed 16/04/15), 2014.

Criscione, Claudio (2013). Drinking the Ocean - Finding XSS at Google Scale. Talk at the Google Test Automation Conference, (GTAC'13), `http://goo.gl/8qqHA`, 2013.

Daggett, Mark E (2013). *Enforcing Style*. Safari Books, 2013.

Dhamija, Rachna; Tygar, J. D.; Hearst, Marti A. (2006). Why phishing works. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems, CHI 2006, Montréal, Québec, Canada, April 22-27, 2006* (pp. 581–590).: ACM, 2006.

Di Paola, Stefano (2010). domxsswiki Sources. online, `https://code.google.com/p/domxsswiki/wiki/Sources` (Last accessed 16/04/15), 2010.

Di Paola, Stefano (2012). DominatorPro: Securing Next Generation of Web Applications. [software], `https://dominator.mindedsecurity.com/`, 2012.

Dolske, Justin (2013). On Firefox's Password Manager. online, `https://blog.mozilla.org/dolske/2013/08/20/on-firefoxs-password-manager/` (Last accessed 16/04/15), 2013.

Doupé, Adam; Cui, Weidong; Jakubowski, Mariusz H.; Peinado, Marcus; Kruegel, Christopher; Vigna, Giovanni (2013). deDacota: toward preventing server-side XSS via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (pp. 1205–1216).: ACM, 2013.

Ecma International (2011). ECMAScript® Language Specification. online, `http://www.ecma-international.org/ecma-262/5.1/#sec-16` (Last accessed 16/04/15), 2011.

Etemad, Elika J. (2011). Cascading Style Sheets. W3C Working Group Note, online, `http://www.w3.org/TR/CSS/` (Last accessed 16/04/15), 2011.

Gasti, Paolo; Rasmussen, Kasper Bonne (2012). On the Security of Password Manager Database Formats. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS 2012), Pisa, Italy, September 10-12, 2012* (pp. 770–787).: Springer, 2012.

Gonzalez, Raul; Chen, Eric Y; Jackson, Collin (2013). Automated Password Extraction Attack on Modern Password Managers. *arXiv preprint arXiv:1309.1416*, 2013.

Google Developers (2012). Chrome Extensions - Developer's Guide. online, `http://developer.chrome.com/extensions/devguide.html` (Last accessed 16/04/15), 2012.

Guarnieri, Salvatore; Pistoia, Marco; Tripp, Omer; Dolby, Julian; Teilhet, Stephen; Berg, Ryan (2011). Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011* (pp. 177–187).: ACM, 2011.

Halderman, J. Alex; Waters, Brent; Felten, Edward W. (2005). A convenient method for securely managing passwords. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005* (pp. 471–479).: ACM, 2005.

Halfond, William G. J.; Orso, Alessandro; Manolios, Panagiotis (2006). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006* (pp. 175–185).: ACM, 2006.

Heise online (2006). Lücke im Internetauftritt der Bundesregierung. online, `http://www.heise.de/newsticker/meldung/Luecke-im-Internetauftritt-der-Bundesregierung-2-Update-158525.html` (Last accessed 16/04/15), 2006.

Heise Security (2006). Paypal-Phishing via Cross-Site-Scripting. online, `http://www.heise.de/security/meldung/Paypal-Phishing-via-Cross-Site-Scripting-133382.html` (Last accessed 16/04/15), 2006.

Hickson, Ian (2005). Web Forms 2.0. W3C Member Submission, online, `http://www.w3.org/Submission/web-forms2/` (Last accessed 16/04/15), 2005.

Hickson, Ian (2015). HTML5 Web Messaging. W3C Proposed Recommendation, online, `http://www.w3.org/TR/2015/PR-webmessaging-20150407/` (Last accessed 16/04/15), 2015.

Hickson, Ian; Berjon, Robin; Faulkner, Steve; Leithead, Travis; Doyle Navara, Erika; O'Connor, Edward; Pfeiffer, Silvia (2014). HTML5. W3C Recommendation, online, `http://www.w3.org/TR/2014/REC-html5-20141028/` (Last accessed 16/04/15), 2014.

Hickson, Ian; Hyatt, David (2008). HTML 5 - A vocabulary and associated APIs for HTML and XHTML. online. `http://www.w3.org/TR/2008/WD-html5-20080610/dom.html#innerhtml0l` (Last accessed 16/04/15), 2008.

Ismail, Omar; Etoh, Masashi; Kadobayashi, Youki; Yamaguchi, Suguru (2004). A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications (AINA 2004), 29-31 March 2004, Fukuoka, Japan* (pp. 145–151)., 2004.

Ives, Blake; Walsh, Kenneth R.; Schneider, Helmut (2004). The domino effect of password reuse. *Communications of the ACM*, Volume 47(4), pp. 75–78, 2004.

Johns, Martin; Engelmann, Björn; Posegga, Joachim (2008). XSSDS: Server-Side Detection of Cross-Site Scripting Attacks. In *Proceedings of the 24th Annual Computer Security Applications Conference, ACSAC 2008, Anaheim, California, USA, 8-12 December 2008* (pp. 335–344).: IEEE Computer Society, 2008.

Johns, Martin; Lekies, Sebastian; Stock, Ben (2013). Eradicating DNS Rebinding with the Extended Same-origin Policy. In *Proceedings of the 22nd USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013* (pp. 621–636).: USENIX Association, 2013.

Jostedt, Erica (2010). Release the Kraken. online, `https://blog.mozilla.org/blog/2010/09/14/release-the-kraken-2/` (Last accessed 16/04/15), 2010.

Jovanovic, Nenad; Krügel, Christopher; Kirda, Engin (2006). Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings*

of the 27th IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA (pp. 258–263).: IEEE Computer Society, 2006.

Joyvent, Inc. (2015). Node.js. online, `https://nodejs.org/` (Last accessed 16/04/15), 2015.

jQuery Bug Tracker (2012). SELECTOR INTERPRETED AS HTML. online, `http://bugs.jquery.com/ticket/11290` (Last accessed 16/04/15), 2012.

Jr., Frolin S. Ocariza; Pattabiraman, Karthik; Zorn, Benjamin G. (2011). JavaScript Errors in the Wild: An Empirical Study. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering, ISSRE 2011, Hiroshima, Japan, November 29 - December 2, 2011* (pp. 100–109).: IEEE Computer Society, 2011.

Juenemann, Nils (2012). Cross-Site Scripting in Google Mail. online, `https://www.nilsjuenemann.de/2012/06/11/cross-site-scripting-in-google-mail-html/` (Last accessed 16/04/15), 2012.

Kafle, Abhibandu (2014). Stored XSS on Facebook and Twitter. online, `http://hak-it.blogspot.in/2014/12/stored-xss-on-facebook-and-twitter_18.html` (Last accessed 16/04/15), 2014.

Karole, Ambarish; Saxena, Nitesh; Christin, Nicolas (2010). A Comparative Usability Evaluation of Traditional Password Managers. In *Proceedings of the 13th International Conference on Information Security and Cryptology, ICISC 2010, Seoul, Korea, December 1-3, 2010* (pp. 233–251).: Springer, 2010.

King, Dave; Hicks, Boniface; Hicks, Michael; Jaeger, Trent (2008). Implicit Flows: Can't Live with 'Em, Can't Live without 'Em. In *Proceedings of the 4th International Conference on Information Systems Security, ICISS 2008, Hyderabad, India, December 16-20, 2008* (pp. 56–70).: Springer, 2008.

Kirda, Engin; Krügel, Christopher; Vigna, Giovanni; Jovanovic, Nenad (2006). Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006* (pp. 330–337).: ACM, 2006.

Klein, Amit (2005). DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium, Articles*, 4, 2005.

Kling, Andreas (2013). XSSAuditor performance regression due to threaded parser changes. online, `https://gitorious.org/webkit/webkit/commit/aaad2bd7c86f78fe66a4c709192e3b591c557e7a` (Last accessed 16/04/15), 2013.

Kolbitsch, Clemens; Livshits, Benjamin; Zorn, Benjamin G.; Seifert, Christian (2012). Rozzle: De-cloaking Internet Malware. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P 2012), 21-23 May 2012, San Francisco, California, USA* (pp. 443–457).: IEEE Computer Society, 2012.

Lekies, Sebastian; Johns, Martin (2012). Lightweight integrity protection for web storage-driven content caching. In *Web 2.0 Security and Privacy (W2SP 2012), San Francisco, California, USA, May 24, 2012*, 2012.

Lekies, Sebastian; Johns, Martin; Tighzert, Walter (2011). The state of the cross-domain nation. In *Web 2.0 Security and Privacy (W2SP 2011), Oakland, California, USA, May 26, 2011*, 2011.

Lekies, Sebastian; Stock, Ben; Johns, Martin (2013). 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (pp. 1193–1204).: ACM, 2013.

Leyden, John (2013). Ubuntu puts forums back online, reveals autopsy of a brag hacker. online, `http://www.theregister.co.uk/2013/08/02/ubuntu_forum_hack_postmortem/` (Last accessed 16/04/15), 2013.

Louw, Mike Ter; Venkatakrishnan, V. N. (2009). Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA* (pp. 331–346).: IEEE Computer Society, 2009.

Magazinius, Jonas; Phung, Phu H.; Sands, David (2010). Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Proceedings of the 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010* (pp. 239–255).: Springer, 2010.

Maone, Georgio (2007). NoScript's Anti-XSS Protection. online, `http://noscript.net/features`xss (Last accessed 16/04/15), 2007.

Maone, Georgio (2008). NoScript's Anti-XSS Filters Partially Ported to IE8. online, `http://hackademix.net/2008/07/03/noscripts-anti-xss-filters-partially-ported-to-ie8/` (Last accessed 16/04/15), 2008.

Mazurek, Michelle L.; Komanduri, Saranga; Vidas, Timothy; Bauer, Lujo; Christin, Nicolas; Cranor, Lorrie Faith; Kelley, Patrick Gage; Shay, Richard; Ur, Blase (2013). Measuring password guessability for an entire university. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (pp. 173–186).: ACM, 2013.

McDaniel, Mason; Heydari, Mohammad Hossain (2003). Content Based File Type Detection Algorithms. In *Procedings of the 36th Hawaii International Conference on System Sciences (HICSS-36 2003), January 6-9, 2003, Big Island, HI, USA* (pp. 332).: IEEE Computer Society, 2003.

Meawad, Fadi; Richards, Gregor; Morandat, Floréal; Vitek, Jan (2012). Eval begone!: semi-automated removal of eval from javascript programs. *ACM SIGPLAN Notices*, Volume 47(10), pp. 607–620, 2012.

Microsoft (2009). IE8 Security Part VII: ClickJacking Defenses. online, `http://blogs.msdn.com/b/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx` (Last accessed 16/04/15), 2009.

Mozilla (2015). Firefox Add-On SDK - Passwords. online, `https://addons.mozilla.org/en-US/developers/docs/sdk/latest/modules/sdk/passwords.html` (Last accessed 16/04/15), 2015.

Mozilla Developer Network (2013). Object.defineProperty(). online, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty` (Last accessed 16/04/15), 2013.

Mozilla Developer Network (2014a). this. online, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this` (Last accessed 16/04/15), 2014.

Mozilla Developer Network (2014b). Window.name. online, `https://developer.mozilla.org/en-US/docs/Web/API/Window/name` (Last accessed 16/04/15), 2014.

Mozilla Developer Network (2015a). Concurrency model and Event Loop. online, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop` (Last accessed 16/04/15), 2015.

Mozilla Developer Network (2015b). Element.innerHTML - Web API Interfaces | MDN. online, `https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML` (Last accessed 16/04/15), 2015.

Mozilla Developer Network (2015c). String.prototype.replace(). online, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace` (Last accessed 16/04/15), 2015.

Mueck, Ralph (2015). Two faces of the Web. Bachelor Thesis, 2015.

Mui, Raymond; Frankl, Phyllis (2011). Preventing Web Application Injections with Complementary Character Coding. In *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS 2011), Leuven, Belgium, September 12-14, 2011*, Lecture Notes in Computer Science (pp. 80–99).: Springer, 2011.

Nadji, Yacin; Saxena, Prateek; Song, Dawn (2009). Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*, 2009.

Nikiforakis, Nick; Invernizzi, Luca; Kapravelos, Alexandros; Van Acker, Steven; Joosen, Wouter; Kruegel, Christopher; Piessens, Frank; Vigna, Giovanni (2012). You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012* (pp. 736–747).: ACM, 2012.

Oftedal, Erlend (2013). Retire.js - identify JavaScript libraries with known vulnerabilities in your application. online, `http://open.bekk.no/retire-js-what-you-require-you-must-also-retire` (Last accessed 16/04/15), 2013.

O'Shannessy, Paul (2006). Bug 359675 - provide an option to manually fill forms and log in. Bug report, online, `https://bugzilla.mozilla.org/show_bug.cgi?id=359675` (Last accessed 16/04/15), 2006.

Papagiannis, Ioannis; Migliavacca, Matteo; Pietzuch, Peter (2011). PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *2nd USENIX Conference on Web Application Development, WebApps'11, Portland, Oregon, USA, June 15-16, 2011*, 2011.

Pelizzi, Riccardo; Sekar, R. (2012). Protection, usability and improvements in reflected XSS filters. In *Proceedings of the 7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012* (pp.5̃).: ACM, 2012.

Pietraszek, Tadeusz; Berghe, Chris Vanden (2005). Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, RAID 2005, Seattle, WA, USA, September 7-9, 2005* (pp. 124–145).: Springer, 2005.

Pizlo, Filip (2013). Announcing SunSpider 1.0. online, `https://www.webkit.org/blog/2364/announcing-sunspider-1-0/` (Last accessed 16/04/15), 2013.

Richards, Gregor; Hammer, Christian; Burg, Brian; Vitek, Jan (2011). The Eval That Men Do - A Large-Scale Study of the Use of Eval in JavaScript Applications. In *Proceedings of the 25th European Conference Object-Oriented Programming, ECOOP 201, Lancaster, UK, July 25-29, 2011* (pp. 52–78).: Springer, 2011.

Ross, Blake; Jackson, Collin; Miyake, Nick; Boneh, Dan; Mitchell, John C. (2005). Stronger Password Authentication Using Browser Extensions. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005* (pp. 17–32).: USENIX Association, 2005.

Ross, David (2009). Happy 10th birthday Cross-Site Scripting! online, `http://blogs.msdn.com/b/dross/archive/2009/12/15/happy-10th-birthday-cross-site-scripting.aspx` (Last accessed 16/04/15), 2009.

Rydstedt, Gustav; Bursztein, Elie; Boneh, Dan; Jackson, Collin (2010). Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites. In *Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

Saxena, Prateek; Akhawe, Devdatta; Hanna, Steve; Mao, Feng; McCamant, Stephen; Song, Dawn (2010a). A Symbolic Execution Framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA* (pp. 513–528).: IEEE Computer Society, 2010.

Saxena, Prateek; Hanna, Steve; Poosankam, Pongsin; Song, Dawn (2010b). FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*, 2010.

Schwartz, Edward J.; Avgerinos, Thanassis; Brumley, David (2010). All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proceedings of the 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA* (pp. 317–331).: IEEE Computer Society, 2010.

Silver, David; Jana, Suman; Boneh, Dan; Chen, Eric Yawei; Jackson, Collin (2014). Password Managers: Attacks and Defenses. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* (pp. 449–464).: USENIX Association, 2014.

Son, Sooel; McKinley, Kathryn S.; Shmatikov, Vitaly (2013). Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (pp. 1181–1192).: ACM, 2013.

Son, Sooel; Shmatikov, Vitaly (2013). The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

Stock, Ben; Johns, Martin (2014). Protecting users against XSS-based password manager abuse. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014* (pp. 183–194).: ACM, 2014.

Stock, Ben; Lekies, Sebastian; Johns, Martin (2014a). DOM-basiertes Cross-Site Scripting im Web: Reise in ein unerforschtes Land. In *Sicherheit 2014: Sicherheit, Schutz und Zuverlässigkeit, Beiträge der 7. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik e.V. (GI), 19.-21. März 2014, Wien, Österreich* (pp. 53–64).: GI, 2014.

Stock, Ben; Lekies, Sebastian; Mueller, Tobias; Spiegel, Patrick; Johns, Martin (2014b). Precise Client-side Protection against DOM-based Cross-Site Scripting. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (pp. 655–670).: USENIX Association, 2014.

Stock, Ben; Stephan, Pfistner; Kaiser, Bernd; Lekies, Sebastian; Johns, Martin (2015). From Facepalm to Brain Bender - Exploring Client-Side XSS. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security, CCS'15, Denver, Colorado, USA, October 12-14, 2015*: ACM, 2015.

The jQuery Foundation (2015a). Category: DOM Insertion, Inside. online, `http://api.jquery.com/category/manipulation/dom-insertion-inside/` (Last accessed 16/04/15), 2015.

The jQuery Foundation (2015b). Working with JSONP. online, `https://learn.jquery.com/effects/` (Last accessed 16/04/15), 2015.

Toews, Ben (2012). Abusing password managers with XSS. online, `http://labs.neohapsis.com/2012/04/25/abusing-password-managers-with-xss/` (Last accessed 16/04/15), 2012.

Tripp, Omer; Ferrara, Pietro; Pistoia, Marco (2014). Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014* (pp. 49–59).: ACM, 2014.

van Kesteren, Anne (2014). Cross-Origin Resource Sharing. W3C Recommendation, online, `http://www.w3.org/TR/cors/` (Last accessed 16/04/15), 2014.

van Kesteren, Anne; Auboung, Julian; Song, Jungkee; Steen, Hallvord R. M. (2014). XmlHttpRequest Level 1. W3C Working Draft, online, `http://www.w3.org/TR/XMLHttpRequest/` (Last accessed 16/04/15), 2014.

Vogt, Philipp; Nentwich, Florian; Jovanovic, Nenad; Kirda, Engin; Krügel, Christopher; Vigna, Giovanni (2007). Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.

W3Techs (2015a). Usage of JavaScript for websites. online, `http://w3techs.com/technologies/details/cp-javascript/all/all` (Last accessed 16/04/15), 2015.

W3Techs (2015b). Usage Statistics and Market Share of JQuery for Websites, February 2015. online. `http://w3techs.com/technologies/details/js-jquery/all/all` (Last accessed 16/04/15), 2015.

Wassermann, Gary; Su, Zhendong (2008). Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering. ICSE 2008, Leipzig, Germany, May 10-18, 2008* (pp. 171–180).: ACM, 2008.

Web Hypertext Application Technology Working Group (2015). HTML Living Standard. online, `https://html.spec.whatwg.org/multipage/` (Last accessed 16/04/15), 2015.

Windows Internet Explorer Engineering (2010). HTML5, and Real World Site Performance: Seventh IE9 Platform Preview Available for Developers. online, `http://blogs.msdn.com/b/ie/archive/2010/11/17/html5-and-real-world-site-performance-seventh-ie9-platform-preview-available-for-developers.aspx` (Last accessed 16/04/15), 2010.

World Wide Web Consortium (2013). Web Storage. online, `http://www.w3.org/TR/2013/REC-webstorage-20130730/` (Last accessed 16/04/15), 2013.

World Wide Web Consortium (2015). Content Security Policy 2.0. W3C Candidate Recommendation, online, `http://www.w3.org/TR/2015/CR-CSP2-20150219/` (Last accessed 16/04/15), 2015.

Wu, Min; Miller, Robert C.; Little, Greg (2006). Web wallet: preventing phishing attacks by revealing user intentions. In *Proceedings of the 2nd Symposium on Usable Privacy and Security, SOUPS 2006, Pittsburgh, Pennsylvania, USA, July 12-14, 2006* (pp. 102–113)., 2006.

Xu, Wei; Bhatkar, Sandeep; Sekar, R. (2006). Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*: USENIX Association, 2006.

Ye, Zishuang (Eileen); Smith, Sean W.; Anthony, Denise (2005). Trusted paths for browsers. *ACM Transactions on Information and System Security*, Volume 8(2), pp. 153–186, 2005.

Yue, Chuan; Wang, Haining (2009). Characterizing insecure javascript practices on the web. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009* (pp. 961–970).: ACM, 2009.

Zalewski, Michael (2009a). Browser Security Handbook, Part 1. online, `https://code.google.com/p/browsersec/wiki/Part1` (Last accessed 16/04/15), 2009.

Zalewski, Michael (2009b). Browser Security Handbook, Part 2. online, `https://code.google.com/p/browsersec/wiki/Part2` (Last accessed 16/04/15), 2009.

Zalewski, Michal (2012). *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

Zhao, Rui; Yue, Chuan (2013). All your browser-saved passwords could belong to us: a security analysis and a cloud-based new design. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013* (pp. 333–340).: ACM, 2013.