From Facepalm to Brain Bender: Exploring Client-Side Cross-Site Scripting

Ben Stock
FAU Erlangen-Nuremberg
ben.stock@fau.de

Stephan Pfistner
SAP SE
stephan.pfistner@sap.com

Bernd Kaiser
FAU Erlangen-Nuremberg
bk@dfjk.eu

Sebastian Lekies
Ruhr-University Bochum
paper@sebastian-lekies.de

Martin Johns
SAP SE
martin.johns@sap.com

ABSTRACT

Although studies have shown that at least one in ten Web pages contains a client-side XSS vulnerability, the prevalent causes for this class of Cross-Site Scripting have not been studied in depth. Therefore, in this paper, we present a large-scale study to gain insight into these causes. To this end, we analyze a set of 1,273 real-world vulnerabilities contained on the Alexa Top 10k domains using a specifically designed architecture, consisting of an infrastructure which allows us to persist and replay vulnerabilities to ensure a sound analysis. In combination with a taint-aware browsing engine, we can therefore collect important execution trace information for all flaws.

Based on the observable characteristics of the vulnerable JavaScript, we derive a set of metrics to measure the complexity of each flaw. We subsequently classify all vulnerabilities in our data set accordingly to enable a more systematic analysis. In doing so, we find that although a large portion of all vulnerabilities have a low complexity rating, several incur a significant level of complexity and are repeatedly caused by vulnerable third-party scripts. In addition, we gain insights into other factors related to the existence of client-side XSS flaws, such as missing knowledge of browser-provided APIs, and find that the root causes for Client-Side Cross-Site Scripting range from unaware developers to incompatible first- and third-party code.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**]: Information browsers; C.2.0 [**General**]: Security and protection

Keywords

Client-Side XSS; Analysis; Complexity Metrics

1. INTRODUCTION

In recent years, Web applications showed a notable move of functionality from the server side towards the client side, where the functionality is realized by the Web browser's JavaScript engine. This trend was caused by various factors, such as the addition of new capabilities in the form of powerful HTML5 APIs, the significantly increased performance of modern JavaScript engines, and the availability of convenient mature JavaScript programming libraries like jQuery. With the growing amount of JavaScript code executed in the Web browser, an increase in purely client-side vulnerabilities can be observed, with the class of Client-Side Cross-site Scripting (also known as DOM-based XSS [11]) being among the most sever security issues. Our previous work demonstrated that about 10% of the Alexa Top 10k carry at least one exploitable client-side XSS vulnerability, with current filtering approaches being unable to stop such exploits [27]. Like most security vulnerabilities, client-side XSS is caused by insecure coding. The underlying reasons for that insecure code, however, are mostly unexplored.

Are developers simply overwhelmed by the complexity of the client-side code? The increase in functionality on the client side naturally increases the amount of code that is executed in a Web page. This also holds true for code which causes the client-side XSS flaws and in turn leads to an increase in the complexity of a developer's task to spot vulnerable data flows.

Alternatively, is the missing security awareness of Web programmers the dominant causing factor? The DOM and the current JavaScript engines offer many different methods to turn arbitrary strings into executable code. Therefore, the use of insecure APIs seems natural to an average developer for solving common problems like the interaction with the page (e.g. using `innerHTML`) or parsing JSON (e.g. using `eval`). Although secure APIs exist in modern browsers, these insecure ways of dealing with such problems are often much faster and more convenient, albeit potentially making a site susceptible to a client-side XSS flaw.

Or is the rise in client-side vulnerabilities rooted in the Web's unique programming model? Compared to classic application platforms and programming languages, the Web model offers numerous open and hidden complexities: The combination of the browser's Document Object Model API (or *DOM API*), the highly dynamic nature of JavaScript, and the process in which Web content is assembled on the fly within the browser, frequently leads to non-obvious control and data flows that can potentially cause security problems.

In this paper, we systematically examine this problem space. To do so, we investigate a large set of real-world zero-day vulnerabilities to gain insights into the complexity of client-side XSS vulnerabilities. Our goal is to determine if such vulnerabilities are easy to spot by security-conscious developers or whether the complex nature of the Web platform hinders detection even for security professionals.

This paper makes the following contributions:

- We examine client-side XSS vulnerabilities and identify characteristic features of complex JavaScript code. Based on these, we define a set of metrics to measure that complexity (Section 3).
- We implement and present an infrastructure that allows us to persist vulnerable code, and enables us to gather all data needed to apply our metrics (Section 4).
- We report on a study of 1,273 real-world vulnerabilities, derive a set of boundaries for our metrics and use these to assign a complexity score to all of the vulnerabilities (Section 5).
- We discuss additional insights gathered from our data, showing that causes for client-side XSS are manifold (Section 6).

2. CLIENT-SIDE XSS

Cross-Site Scripting (also called *XSS*) is a class of code-injection vulnerability in the browser. Web applications run in a protected environment, such that only code from the same origin as the application can interact with it. Therefore, the goal of an XSS attacker is to execute arbitrary JavaScript code in the browser of his victim, in the context (or origin) of the vulnerable Web page. If successful, this allows him to conduct any action in the name of the victimized user, e.g., using the application as the user or retrieving secret information such as cookies.

While server-side XSS attacks have been known for a number of years, the youngest representative, i.e., client-side XSS, was first discussed in 2005 by Amit Klein [11]. Rather than being caused by vulnerable server-side code, this class of vulnerability occurs if user-provided input is insecurely processed on the client side, e.g., by using this data for a call to `document.write`, as shown in Listing 1 using the `location`, which returns the current URL.

From a conceptual standpoint, XSS is caused when an unfiltered *data flow* occurs from an attacker-controlled *source* to a security-sensitive *sink*. In the concrete case of client-side XSS, such a source can be, e.g., the URL, whereas an example for a sink is `eval` or `document.write`. Both these APIs accept strings as parameters, which are subsequently parsed and executed as code (JavaScript and HTML, respectively). Therefore, passing attacker-controllable input to such functions eventually leads to execution of the attacker-provided code. There are additional sinks, which do not allow for direct code execution (such as cookies or Web Storage). These sinks, however, are not in the focus of our work.

3. CHARACTERIZING CLIENT-SIDE XSS

The goal of our work is to gather insights into the root causes of client-side XSS flaws on the Web, first and foremost to answer the question whether the complexity of client-side code is the primary causing factor. In order to spot a vulnerability, an analyst has to follow the data flow from source to sink and fully understand all operations that are performed

on the data, with several properties increasing the difficulty of this task. In the following, we discuss several measurable properties of the perceived analysis complexity and present metrics to measure them accordingly. In addition to these, vulnerabilities have characteristics which do not necessarily have an impact on the complexity of a flaw, but provide interesting additional insights. Therefore, after presenting the measurable properties and matching metrics, we discuss these additional characteristics.

3.1 Measurable Properties of JS Complexity

In this section, we outline properties which we deem useful in determining how hard it is for an analyst to spot a vulnerable flow of data in JavaScript code. For each of the properties, we then define a metric that measures the perceived complexity.

3.1.1 Number of Operations on the Tainted Data

In our notion, data that originates from a user-controllable source is considered *tainted*. A client-side XSS vulnerability constitutes a flow of such tainted data from a source to a security-critical sink. An analyst therefore has to decide whether the user-provided data is filtered or encoded properly and, thus, must understand all operations conducted on that data. Thus, one necessary consideration to make is that each operation naturally increases the perceived complexity as more code has to analyzed and understood. Such operations can either be read accesses, e.g., regular expression checking, or write accesses to the data, such as concat.

Therefore, we define our first metric, M_1, to measure the number of string-accessing operations which are conducted in a read or write manner throughout the flow of attacker-controllable data from source to sink, including source and sink access. In order to not over-approximate this number, we join the same type of operations on a per-line level, e.g., several concatenating operations on the same line are treated as a single operation. An example of such a case is shown in Listing 1, where two concatenation operations occur in one line. Therefore, our metric M_1 measures this snippet to have three operations, i.e., source access, concatenation and sink access. In real-world vulnerabilities, the string which eventually ends in a sink may consist of substrings that originated from different sources. Therefore, we use M_1 to measure the longest sequence of operations between any source access and the final sink access.

```
// actual code
document.write("<a href=' " + location + "'>this page</a>")
// as interpreted by our metric
var x = location; // source access
var y = "<a href='" + x + "'>this page</a>"; // joined concats
document.write(y); // sink access
```

Listing 1: Vulnerability Example

3.1.2 Number of Involved Functions

JavaScript, not unlike any other programming language, employs the concept of functions, which can be used to split up functionality into smaller units. While this is best practice in software engineering, it increases the difficulty a security auditor has to overcome as he has to understand specifically what each of these units does. Therefore, in addition to the number of operations which are conducted on a tainted

string, the number of functions that are traversed is another indicator for the complexity of a flow or a vulnerability.

We therefore define our second metric, M_2, to count the number of functions which are passed between source and sink access. Although all code may also reside in the top execution level, i.e., not specifically within defined functions but rather in a *virtual main* function, the minimum number of traversed functions must always be one. In this context, we define the *virtual main* to be the main JavaScript execution thread which executes all inline script blocks and external JavaScript files.

3.1.3 Number of Involved Contexts

The Web's model allows for JavaScript files to be included from other sites, while inheriting the including page's origin and, thus, running in that origin. Therefore, a single JavaScript block or file is not executed independently, but within a Web page possible containing tens of other JavaScript resources, potentially stemming from other domains and developers. The main interaction point of all these scripts is the global object, in the case of the browser the `window` object. Since all script elements within a Web page may gain access to that object, different snippets may register global variable or functions to allow for interaction with other parts of the executed code.

In our notion, we call each of these script elements, which the analyst has to fully understand to decide whether a flow might be vulnerable, *contexts*. Thus, we define our third metric, M_3, to count the number of contexts which are traversed in the execution of a vulnerable JavaScript program between source and sink.

3.1.4 Code Locality of Source and Sink

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. Naturally, this is easier if not only the number of operations that are conducted is low, but also both source- and sink-accessing operations are within a smaller number of lines of code. In contrast, even if the number of operations is low, a vulnerability is harder to detect if source and sink access are further apart in the code, as the analyst has to specifically search for the tainted variables to be used again.

Thus, as a fourth metric, M_4, we measure the amount of code between source and sink access. This metric, however, can only be applied to vulnerabilities flow which the source and sink access is conducted within the same file, i.e., either within two inline script blocks or the same external JavaScript file.

3.1.5 Callstack Relation between Source and Sink

Another property that increases the perceived complexity when dealing with potentially vulnerable JavaScript code is the relation between source and sink access in the call stack. As discussed before, the code responsible for an exploitable flaw might be spread across multiple functions or contexts, i.e., source and sink access do not have to be contained within the same function or context.

In the easiest case, access to the source and sink is conducted within the same function, i.e., on the same level in the call stack. Figure 1 shows the different relations between these two operations with respect to the sink access being conducted in the red script element `SE #3`. For our first relation, R_1, the source access also occurs in this element.

The second scenario occurs when the source access is conducted in the blue script element `SE #1`. In this case, the tainted data is passed as a parameter to the function the sink access resides in. From an analyst's perspective, this means that he can follow the flow of data from the source to the function. He can subsequently analyze the function with the knowledge that the passed parameter contains tainted and unfiltered data, allowing him to decide whether the following sink access can be deemed safe or not. We refer to this callstack relation as R_2.

In contrast to the previous case, the source access may also occur in an element which is lower in the callstack than the sink access. This is depicted in Figure 1 when the source access occurs in the yellow `SE #4`. In such a case, the analyst has to follow the called function to determine that user-provided data is accessed before having to go up to `SE #3` again to see how the tainted data is handled. This complicates the analysis, since it requires a switch back and forth between functions, and potentially contexts/files. We deem this to be R_3.

As a fourth scenario, we identify snippets of code in which source and sink only share a common ancestor in the callstack, but neither are parents of the other in the callstack. Figure 1 shows this for a source access in the orange element `SE #2`, which shares the common parent `SE #1` with the sink-accessing `SE #3`. The increased complexity in this scenario stems from the fact that the analyst must first investigate the function which accesses the source, continue his analysis of the common parent, and then decent into the sink-accessing function. We denote this relation to be R_4.

Finally, the most complex type of flows occurs if source and sink access share no common ancestors apart from the virtual main function. This is enabled by the fact that all JavaScript operates on the same global object, i.e., may set global variables accessible by any other snippet of JS running on the same Web page. This is shown in Figure 1 when accessing the source in the purple `SE #0`. Hence, there is no path of code an analyst can follow between source and sink access. Instead, he has to understand that the first script element accesses the source and stores the retrieved value in a global variable, and that in turn the second element accesses this stored value before passing it to the sink. In our notion, this is callstack relation R_5.
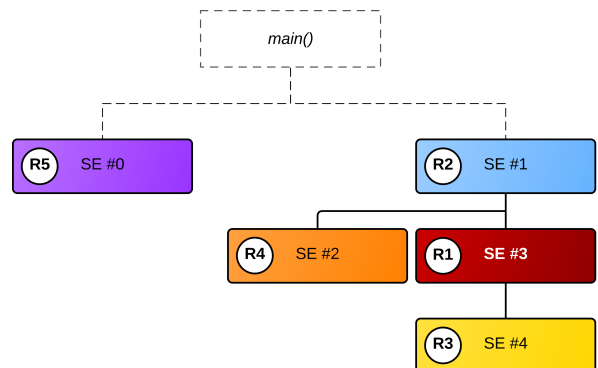

Figure 1: Relations between source and sink access

3.2 Additional Characteristics

In addition to the previously outlined properties which can be measured to either result in a numerical value or a specific relation between source and sink access, we find that more characteristics can be observed for vulnerable code. In this section, we shed light on these and discuss their implications.

Non-Linear Flows: An additional important observable characteristic of Client-Side Cross-Site Scripting is a classification of the vulnerability's data and control flows in respect to their *linearity*:

In the context of this paper, we consider a data flow to be *linear* (LDF), if on the way from the source to the sink, the tainted value is always passed to all involved functions directly, i.e., in the form of a function parameter. In consequence, a non-linear data flow (NLDF) 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 NLDFs 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 (NLCF) are instances of interrupted JavaScript execution: A first JavaScript execution thread accesses the tainted data source and stores 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 NLCFs 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 NLDF, the inspection of such flows is significantly more difficult.

Code Origin: The previously outlined model of JavaScript allows for externally hosted code to be executed in the context of the including application. This implies that a vulnerability in included third-party code results in a vulnerability in the including application and, in addition, an analyst may need to follow the data flow through code from different authors and origins. This switch between contexts is already covered by M_3 and, thus, we do not consider the code origin to be an additional indicator for the complexity of the analyst's task. It is, however, interesting to determine what code caused the actual flaw.

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 Alexa to match 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.

Multiflows: A single sink access may contain more than one piece of user-provided data. This leaves an attacker with a means of splitting up his malicious payload to avoid detection. As we [27] have shown, given the right circumstances, such flows can be used to bypass existing filter solutions such as Chrome's XSSAuditor [1].

Sinks: Another characteristic which is not directly related to the complexity of a given flow is the sink involved in the flaw. While a larger number of sinks exist, our data set (which we present in Section 5.1) consists only of flaws that target `innerHTML`, `document.write` and `eval`. These also contain conceptually similar sinks, such as `outerHTML`, `doc-` `ument.writeln` and `setTimeout`, respectively. These sinks differ in terms of what kind of payload must be injected by an attacker to successfully exploit a vulnerability.

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. However, the fact that a script uses `eval` to generate code at runtime cannot be measured in a discrete manner, thus we opt not to use it for complexity metric.

4. INFRASTRUCTURE

The basis of our study is a data set of real-world vulnerabilities. While this enables us to investigate the complexities of such flaws at a large scale, it brings its own set of challenges we had to overcome. In this section, we discuss these challenges and present the infrastructure we developed for our experimentations.

4.1 Initial Data Set and Challenges

The basis of the results we present in this paper is a set of exploits detected with the methodology we developed in 2013 [12]. 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 request 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 to gather detailed information on data flows, 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, 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 jQuery functions as a direct sink and remove them from the trace information we collect.

4.2 Persisting and Preparing Vulnerabilities

To allow for a reproducible vulnerability set, we needed to implement a proxy capable of persisting the response to all requests made by the browser when visiting a vulnerable site. To achieve this, we built a proxy on top of `mitm-proxy` [4], 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 [13].

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 [6]. 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 [28]. 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 each URL by simply replacing each number in the URL with a fixed value. As we had a ground truth of vulnerable sites, we were able to verify that our proxy would correctly replay all vulnerabilities. This might, however, not be true for random sites and would then warrant further testing.

4.3 Taint-Aware Firefox Engine

Our analysis methodology relies on a dynamic analysis of real-world vulnerable JavaScript. Naturally, this requires the execution of the vulnerable code and the collection of corresponding trace information. Although taint-aware engines, such as DOMinator [7], exist, we opted to implement our own engine specifically tailored to fit the needs of our study. To ensure a fine-grained analysis of vulnerable flows from sources to sinks, we patched the open-source Web browser Firefox. The browser was enhanced to track data originating from sources, across all processing steps in the SpiderMonkey JavaScript engine as well as the Gecko rendering engine, and into sinks.

Our previous work relied on numerical identifiers in shadow data to mark a part of a string as originating from a specific user-provided source [12]. This, however, only allowed for tracking of basic source and encoding information. Our study aims at gathering additional insight into the inner workings of data flows, therefore a more elaborate approach must be taken to ensure that all necessary information can be stored. More precisely, for each vulnerable data flow we want to capture exact, step-by-step operations which impacted the tainted data or provide additional knowledge relevant to interpret the data flow. This includes access to source, calls to both built-in and user-defined functions which operate on a tainted string, as well as stack traces for each operation to allow for execution context analysis.
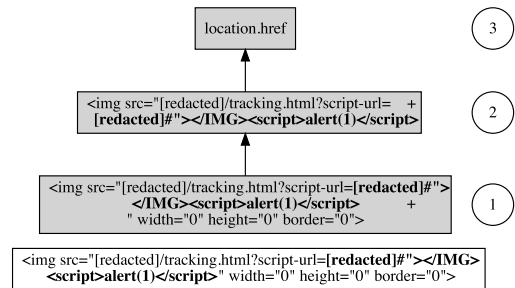

Figure 2: In-Memory representation of taint information

To match these requirements and keep the performance impact as low as possible, we designed a scheme for efficiently tracking data flows while allowing to record all the relevant flow information and also enable propagation between strings. During execution, a tree-like memory structure is built consisting of nodes representing the operations, which are annotated using the context information present when executing the operation. Edges between the nodes represent execution order and dependency: child operations are executed after their parents and consume the output of them. For each tainted substring, string objects contain references to these nodes, thereby implicitly expressing the complete history of the tainted slices. Following the ancestors of a taint node allows to recreate the chain of operations that led to the current tainted string - beginning with the source access. Derived strings copy the references and add their taint information to the tree, keeping the existing information for the old strings intact.

Figure 2 shows a graphical representation of this memory model. The sink access is depicted at the lower part of the figure; the shown string was written to the sink `document.write`. To reconstruct all operations which eventually led to the tainted string being written to the document, we traverse upwards, detecting that the final string was the result of a concatenation of an untainted part (depicted in normal font weight) with the the output of another concatenation operation (1). This operation put together two additional strings, consisting of an untainted part and a tainted string (shown in bold, (2)). In order to extract the source of said tainted data, we traverse up the tree once more to find that it originated from `location.href` (3). Similarly, the data structure can be traversed for sink accesses with multiple pieces of tainted data.

As our study is based on data derived from previous work which used a patched variant of Chromium to detect and exploit flows, we expect these vulnerabilities to be at least partially dependent on Chromium's encoding behavior with respect to the URL, and more important, the URL fragment, which is — in contrast to Firefox — not automatically encoded. Therefore, we opted to align our browsing engine's encoding behavior with that of Chromium, i.e., by not automatically encoding the fragment.

4.4 Post-Processing

Before the raw data gathered by our engine can be analyzed, it needs to processed to ensure a correct analysis. In the following, we illustrate these post-processing steps.

Identifying Vulnerable Flows: 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`. Next, we determine if our payload was completely contained in the string which was passed to the sink, i.e., we ensure that the data only contains actual exploitable flows.

jQuery Detection and Removal: One of the most commonly used libraries in many Web applications is jQuery [2, 29]. 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, increasing the perceived complexity. This, however, is not true since the vulnerability was introduced by *using* jQuery in an insecure manner, not jQuery itself. Therefore, we select to filter jQuery functions and use them as sinks, i.e., the `html` and `append` functions of jQuery are treated like an assignment to an element's `innerHTML` property.

We therefore implemented a detection mechanism to pinpoint which functions were provided by jQuery. The flowchart in Figure 3 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* [18] 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 is found, we mark the generated report for later manual analysis. This happens when Web site owners use custom packers, rendering full-text search infeasible. If any of these 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 artefacts from sink-like operations such as `html` and jQuery-enabled uses of events.

Stack Gap Detection and Removal: After the jQuery detection finishes, we proceed to the next phase of post-processing. Our taint-aware engine is able to record all calls to functions a tainted string is passed to. In some cases, however, 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, although an analyst has to follow this call in his analysis. To allow for a correct value for metric M_2, we close this *stack gap* by inserting virtual function calls into the trace.

Operation Fusing: As discussed in Section 3.1, M_1 measures the number of operations which were conducted on tainted data. In terms of understanding a vulnerability, several concatenations on a single line do not complicate the
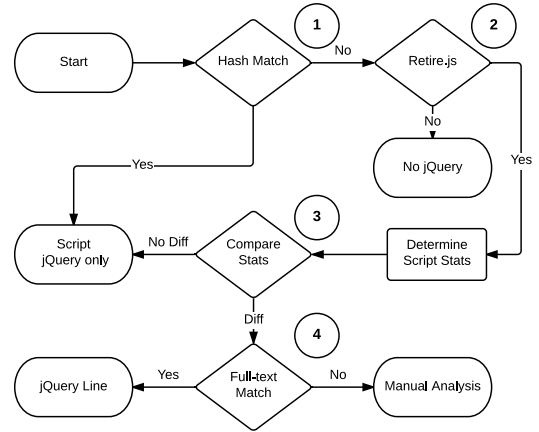

Figure 3: Flowchart for jQuery detection process

analysis; therefore, all consecutive concat operations on the same source code line are fused to form a single operation.

4.5 Overall Infrastructure

An overview of our complete infrastructure is depicted in Figure 4. 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 the 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. EMPIRICAL STUDY

In this section, we outline the execution of our study as well as the results. We then present classification boundaries for the metrics discussed in Section 3.1 and describe the results of that classification.

5.1 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 by our methodology presented in 2013 [12]. In total, this set consisted of 1,146 URLs in the Alexa Top 10k which contained at least one verified vulnerability, i.e., a crafted URL which would trigger our JavaScript to execute in the vulnerable document. 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 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 3.1, a single sink access may use data from several sources. In total, we found that the 1,273
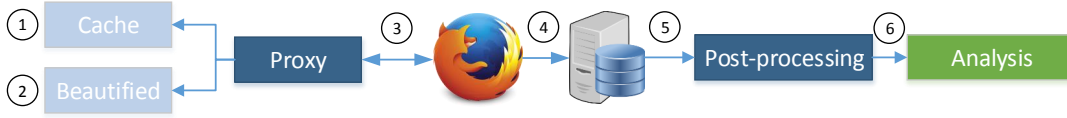
Figure 4: Overview of our analysis infrastructure

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.

5.2 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 5.4.

M_1 **Number of string-accessing operations:** Figure 5 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.

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 6.

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 7 shows this, highlighting that more than 90% of the flaws were spread across one or two contexts.

M_4 **Locality of source and sink access:** For all vulnerabilities which were contained either in one single external file or inside the same HTML document, 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
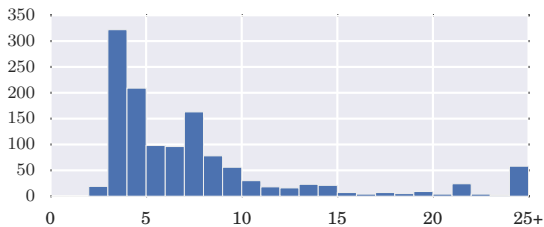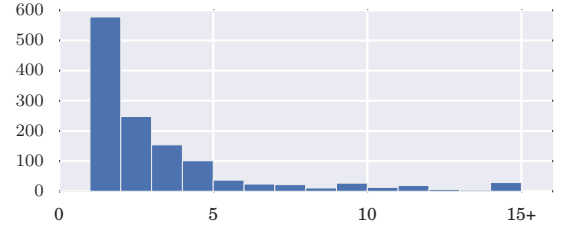

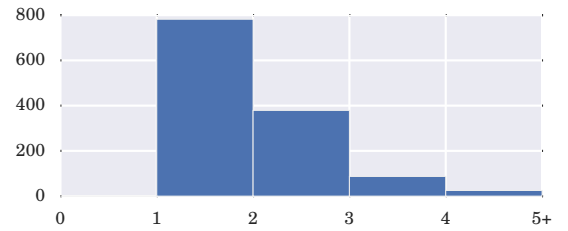Figure 6: Histogram for M_2 (number of functions)

Figure 7: Histogram for M_3 (number of contexts)

access. In the most complex case, the access to the source occurred 6,831 lines before the sink access. The distribution of this metric is shown in Figure 8.

M_5 **Relation between source and sink:** In our analysis, we found that the most common scenario was R_1, which applied to a total of 914 flows. Second to this, in 180 cases, the source was an ancestor of the sink (R_2), whereas this relation was reversed 71 times (R_3). Source and sink shared a common ancestor in 49 flows (R_4) and finally, there was no relation between source and sink in 59 traces (R_5).

5.3 Additional Characteristics

In addition to the proposed metrics, several additional characteristics can be observed for our data set of vulnerabilities, which we outline in the following.
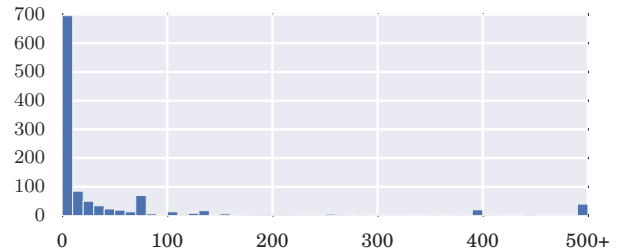

Figure 5: Histogram for M_1 (string-accessing ops)

Figure 8: Histogram for M_4 (Code locality)

As we discussed beforehand, both code and data flows may occur in a non-linear manner. Table 1 shows the distribution of this feature in 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. We observe that 59 cases, which are also matched by R_5, both data and control flow are non-linear. In addition, we found that in 98 of the vulnerable flows, a non-linear data flow occured, i.e., the data was not passed as a parameter to all functions it traversed.

In terms of code origin, our analysis revealed interesting results. While the biggest fraction, namely 835 vulnerabilities, was caused purely by self-hosted code, we found that 273 flaws were contained exclusively in third-party scripts, leaving the Web page exposed to a Cross-Site Scripting flaw to no fault of its developer. The remaining 165 flaws occurred in a combination of self-hosted and third-party code.

An attacker may leverage a single sink access which contains more than one attacker-controllable piece of data to circumvent popular Cross-Site Scripting filters [27]. Therefore, an additional characteristic is whether a flaw constitutes a multiflow, which we discovered in 344 of the exploited Web pages.

Although the sink in which the vulnerable flow of data ended is not directly related to the complexity of the vulnerability itself, it is relevant for remedies, as different filtering steps must be taken depending on the type of sink. In our study, we found that 732 exploitable flows ended in `document.write`, 495 in `innerHTML` and remaining 46 in `eval` and its derivatives.

In addition to `eval` being a sink, we also observed flows in which it was used to generate code, which was ultimately responsible for a flow, at runtime. In total, only eleven of such cases were contained in our data, while the most common scenario was deobfuscation of code at runtime, e.g., by base64-decoding it.

5.4 Analysis

In Section 3.1, we defined a set of metrics to measure the complexity of a vulnerable flow, which we then applied to a set of real-world vulnerabilities in our study. To better quantify the complexity of a flaw, we need to translate the numeric values derived by our metrics into a classifying scheme. 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.

Classification Scheme: Based on the gathered data of all metrics, we derive the 80th and 95th percentiles, i.e.,
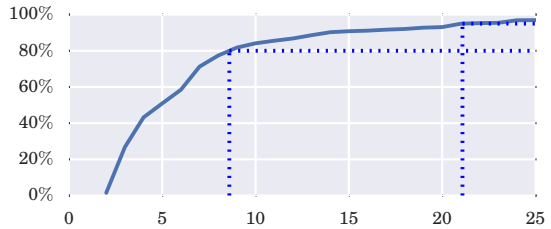

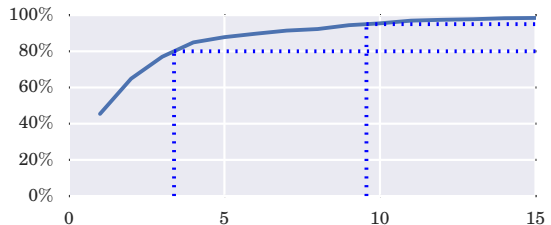Figure 9: Cumulative Sum for M_1 (string-accessing ops)

Figure 10: Cumulative Sum for M_2 (number of functions)

derive the numbers for which at least 80% and 95% of all vulnerable flows have a *lower* metric value, respectively. For M_1 and M_2, the cumulative sums are depicted in Figures 9 and 10, highlighting also both the percentiles. Although metric M_5, which denotes the relation of source and sink accessing operations, does not return a numerical value, the perceived complexity rises with the identifier, i.e., R_2 is more complex than R_1 and so on and so forth. For this metric, more than 80% of the flows were contained in the relation classes R_1 and R_2 and less than 5% of the flows were made up out of flows which had no relation between source and sink (R_5).

We use the resulting percentiles as cut-off points for our complexity metric classification. Therefore, we set boundaries for all of our metrics accordingly (as depicted in Table 2), such that any value maps to either a low (LC), medium (MC) or high (HC) complexity. We calculate the overall complexity of a flaw from the single highest rating by any metric; this appeals naturally to the fact that a vulnerable flow is already hard to understand if it is complex in just a single dimension.

Classification Results: Based on our classification scheme, we categorize each of the flaws in our data set to a complexity class. The results of this classification scheme are depicted in Table 3, where C_{MX} denotes the results for M_X. Although the boundaries were derived from the 80th and 95th percentile, the results (especially for M_2) are not evenly distributed. This stems from the fact that the boundaries are a fixed value which denotes that *at least 80% or 95% of*

	LCF	NLCF	Sum
LDF	1,116	—	**1,116**
NLDF	98	59	**157**
Sum	**1,214**	**59**	**1,273**

Table 1: Data and code flow matrix

	LC	MC	HC
M_1	≤ 9	≤ 22	>22
M_2	≤ 4	≤ 10	>10
M_3	≤ 2	3	>3
M_4	≤ 75	≤ 394	>394
M_5	R_1, R_2	R_3, R_4	R_5

Table 2: Classification boundaries

	LC	MC	HC
C_{M1}	1,079	134	60
C_{M2}	1,161	85	27
C_{M3}	1,035	178	60
C_{M4}	*920*	*179*	*51*
C_{M5}	1,094	120	59
Combined	**813**	**261**	**199**
	63.9%	20.5%	15.6%

Table 3: Classification by applied metrics

	80th	95th	100th
M_1	≤ 20	≤ 44	>44
M_2	≤ 9	≤ 19	>19
M_3	≤ 2	3	>3
M_4	≤ 189	$\leq 1,208$	$>1,208$
M_5	R_1, R_2, R_3	R_4	R_5

Table 4: Percentiles for randomly-sampled flows

the flows had a lower ranking, not necessarily exactly that number. Note also that M_4 can only be applied to subset of 1.150 of the flows in our data set, as only in these cases source and sink access were contained within the same file.

By design, each of our metrics assigns at least 80% of the flaws to lowest complexity class. The results of the combination of all metrics is also shown in Table 3: we observe that in combining the result, less than two thirds of the flows are categorized as having an overall low complexity, i.e., that for each metric their value was below the 80th percentile. 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, putting emphasis on the different proposed metrics.

5.5 Summary of Our Findings

In summary, by grouping the results from each of the metrics, separated by the 80th and 95th percentile, and combining the resulting classifications into either low, medium or high complexity, about two thirds of our data set is still labeled as having a low complexity, whereas 20% and 15% of the flows are labeled as having a medium or high complexity, respectively. This shows that taking into account only a single metric is not sufficient to ascertain the complexity of a vulnerability, but that rather all dimensions must be analyzed. Given the large fraction of simple flows, which consist of at most nine operations on the tainted data (including source and sink access) and span no more than two contexts, we ascertain that Client-Side Cross-Site Scripting is often caused by developers who are unaware of the risks and pitfalls of using attacker-controllable data in an unfiltered manner.

Although the fraction of straight-forward vulnerabilities is very high, the unawareness of developers is only one root cause of Client-Side Cross-Site Scripting. As this section has shown, developers may also be overwhelmed by the sheer amount code they have to understand, potentially passing first- and third-party code on the way. In addition, we found 273 cases in which the vulnerability was contained strictly inside third-party code and, thus, the developer of the then vulnerable application was not at fault.

5.6 Comparison to Randomly-Sampled Flows

From our data set, we gathered a large number of flows which appeared to be vulnerable, but did not trigger our payload. We can, however, not state with certainty that these flows were indeed secure. For instance, the applied filtering might be incomplete, a condition that is not covered by our exploit generator. Therefore, to put the results of our study into perspective, we randomly sampled 1.273 flows

from an attacker-controllable sink to a direct execution sink. Table 4 shows the results for the percentiles of these flows. Interestingly, the percentile values are higher for each of the metrics compared to vulnerable flows. This shows that the complexity of such flows alone can not be the causing factor for the widespread occurrence of client-side XSS.

6. ADDITIONAL INSIGHTS

Our analysis so far uncovered that the biggest fraction of vulnerabilities are caused by programming errors which, according to the results of our metrics, should be easy to spot and correct. We conducted a more detailed analysis into several low complexity flaws as well as those flaws which were ranked as having a high complexity and found a number of interesting cases, which shed additional light on the issues that cause client-side XSS. Therefore, in the following, we highlight four different insights we gained in our analysis.

6.1 Involving Third Parties

In our analysis, we found that vulnerabilities were also caused when involving code from third parties, either because first- and third-party code were incompatible or because a vulnerable library introduced a flaw.

Incompatible First- and Third-Party Code: A more complex vulnerability, which was rated as being of medium complexity for M_3 and high complexity by M_5, utilized meta tags as a temporary sink/source. Listing 2 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 2: 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 3. 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 generate keywords from user 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 3: Third-party code extracting previously set meta tags

Vulnerable Libraries: An example for a vulnerability which was rated as being of low complexity is related to a vulnerable version of jQuery. 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 [10], if attacker-controllable content was passed to the function—even if a `#` tag was hardcoded in at the beginning of that string. Listing 4 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.

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

Listing 4: Vulnerable code if used with jQuery before 1.9.0b1

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 paper. In total, we discovered that 472 of the exploited Web pages contained outdated and vulnerable versions of jQuery, albeit only a fraction contained calls to the vulnerable functions. jQuery was accompanied by a number of vulnerable plugins, such as `jquery-ui-autocomplete` (92 times). Second to jQuery came the YUI library, of which vulnerable versions were included in 39 exploited documents. This highlights that programmers should regularly check third-party libraries for security updates or only include the latest version of the library into their pages.

6.2 Erroneous Patterns

Apart from the involvement of third-party code which caused vulnerabilities, we found two additional patterns that highlight issues related to client-side XSS, namely the improper usage of browser-provided APIs and the explicit decoding of user-provided data.

Improper API Usage: In our data set, we found a vulnerability in the snippet shown in Listing 5, which was assigned the lowest complexity score by any of our metrics. 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` [9], the attacker can pass a payload containing an `img` with an error handler [15]. 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. Next to this flaw, we found examples of the improper use of built-in functions, such as `parseInt` and `replace`.

```
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: Improper use of `innerHTML` for sanitization

Explicit Decoding of Otherwise Safe Data: As outlined in Section 4.3, the automatic encoding behavior of data retrieved from the `document.location` source varies between browsers: Firefox will automatically escape 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.

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. Nevertheless, 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 the persisted vulnerabilities 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.

6.3 Summary of Our Insights

In summary, we find that although a large fraction of all vulnerabilities are proverbial facepalms, developers are often confronted with additional obstacles even in cases, where the complexity is relatively low. In our work, we have found evidence for faults which are not necessarily to blame on the developer of the vulnerable Web application, but rather are either a combination of incompatible first- and third-party code or even caused completely by third-party libraries. This paradigm is enabled by the Web's programming model, which allows for third-party code to be included in a Web page, gaining full access to that page's DOM.

In addition, we have found patterns of mistakes, which are caused by developers due to their misunderstanding of browser-provided APIs or even the explicit decoding of user-provided data to allow for a convenient use of such data.

Apart from the example shown here, we have found additional misguided attempts at securing applications, allowing an attacker to easily bypass these measures. This leads us to believe that even developers, who are aware of the potential pitfalls of using attacker-controllable data in their application, sometimes lack the knowledge of how to properly secure their application.

7. RELATED WORK

Client-Side Cross-Site Scripting: In 2005, Amit Klein coined the term of *DOM-based XSS* [11]. One of the first tools specifically designed to detect DOM-based XSS was DOMinator, which used taint-tracking to detect vulnerable flows [7]. We extended the concept, presenting an automated means of detecting and verifying client-side XSS vulnerabilities, finding that about 10% of the Top 10k Web pages carry at least one such vulnerability [12]. On top of this taint-tracking engine, we built a taint-aware XSS filter aimed at thwarting client-side XSS [27]. Previous work by Saxena et al. has investigated so-called *Client-Side Vulnerabilities*, mainly aiming at the exploitation of client-side XSS flaws. Using "taint enhanced blackbox fuzzing", they were able to discover eleven previously unknown vulnerabilities on real-world Web pages [22]. In contrast to that, Criscione presented a brute-force blackbox testing approach [5]. To find bugs in complex JavaScript code, Saxena et al. developed a symbolic execution framework for JavaScript, which helped them to find two previously unknown vulnerabilities [21].

JavaScript Analysis: Furthermore, attention has been given to JavaScript error sources in general. Two research groups presented empirical studies of JavaScript source code included in popular sites. Richards et al. investigated the general runtime behavior of JavaScript and concluded that the language is a "harsh terrain for static analysis", confirming our decision to use a dynamic approach [20]. Ocariza et al. categorized the different kinds of errors they encountered with specific test cases they created for the top 100 Web sites [17]. Although these sites were in a mature productive state, the same well-defined categories of errors were discovered frequently.

Richards et al. evaluated the use of `eval` in depth and came to the conclusion that up to 82% of all Web sites utilize the construct, often in a dangerous way. They did, however, find that replacing `eval` altogether is still unfeasible [19]. In 2011, Guarnieri et al. [8] presented ACTARUS, which is capable of conducting taint-aware static analysis of JavaScript on real-world Web sites, finding over 500 vulnerabilities on eleven sites. Later, Meawad et al. developed the tool `Evalorizer`, aiming to assist programmers in the removal of unnecessary `eval` constructs [14]. As our work has shown, Web sites often incorporate cross-domain JavaScript code. In their work, Nikiforakis et al. investigated the trust relationships between web pages of the Alexa Top 10k sites [16], demonstrating that sites often open themselves to attacks by including third-party content.

Vulnerability Analysis: Other research has focussed on more general vulnerability analysis. In 2008, Shin et al. conducted an analysis on how well code complexity metrics can be used to predict vulnerabilities in the Mozilla JavaScript Engine [26]. They conclude that while complexity metrics can be useful in finding flaws with a low false positive rate, they carry a high false negative rate. Besides complexity,

Shin et al. looked at code churn and developer activity metrics to determine indicators of vulnerabilities and were able to use them to "to prioritize inspection and testing effort" [25]. Complexity, coupling, and cohesion metrics with similar results have also been used to discover vulnerabilities statically by Chowdhury et al. [3]. Another approach, aiming at finding improper input validation was presented by Scholte et al. [23], using automatic data type inferring for the validation functions. Doing so, they found that 65% of the (server-side) XSS flaws in their set were simple enough to be stopped by their approach, without causing any hassle for the developers, which coincides with the number of simple flows we discovered in our study.Static taint analysis has been used by Wassermann and Su to identify server-side XSS using a policy of allowed inputs based on the W3C recommendation [30]. Yamaguchi et al. have conducted additional work to identify new vulnerabilities using machine learning and a set of previously known vulnerabilities [31] as well as based on Abstract Syntax Trees [32].

To summarize, while previous research has focussed on the detection of client-side XSS, JavaScript security analysis and general vulnerability analysis, no prior work has investigated the underlying causes of client-side XSS flaws. Our work aims at closing this gap in previous research.

8. LIMITATIONS AND FUTURE WORK

The results of our analysis are naturally influenced by the methodology used to find the vulnerabilities. Most importantly, our method only finds vulnerabilities in code that is executed during a normal page visit. Hence, flows that are dependent on a certain condition (such as a specific URL parameter) cannot be detected. Since no ground truth in terms of *all* real-world client-side XSS vulnerabilities is known, we have no means of ascertaining whether such flaws are even more complex than the ones underlying our study. Extending the detection methodology with static analysis to discover more vulnerabilities is therefore a promising extension which could subsequently be transferred to our current work.

An interesting extension of our work is the application of code coverage metrics by instrumenting the cached JavaScript code [24]. This approach, however, has limitations of its own, as rewriting cannot be conducted on code which is dynamically generated at runtime using `eval`.

In addition, with the gathered data, we are able to investigate the usage of filtering functions on the Web, e.g., the regular expression used. Therefore, we believe the analysis of these deployed filters might shed light on improper use of such functions, especially as we have anecdotal evidence of improperly used regular expression filtering.

9. CONCLUSION

In this paper we investigated the root cause of client-side XSS, i.e., the underlying issues related to this class of XSS vulnerabilities. To do so, we throughly analyzed a set of 1,273 real-world vulnerabilities and classified them according to their complexity.

Our work shows that a large number of flaws are comparatively simple and, thus, most likely rooted in insufficient security awareness of the responsible developers. Based on the classification approach introduced in this paper, about two thirds of all examined vulnerabilities fall into this category. In contrast, about 15% of the discovered flaws have a

high combined complexity rating, showing that developers may also be overwhelmed by the complexity of the vulnerable code; in 59 cases we even discovered interrupted code flows, significantly impeding the flaw discovery. Our study, however, also found that for randomly sampled flows, the complexity metrics generally yield higher values, i.e., non-exploitable code is often even more complex than vulnerable code.

In addition to these findings, our gained insights highlight that the aforementioned reasons are not the only causing factors for client-side XSS. The presented findings show that developers are not always to blame, as flaws might be caused by third-party code. In 273 of our vulnerabilities, this third-party code was solely responsible for the flaw, whereas an additional 165 flaws was caused by a combination of third- and first-party code, in parts related to the careless use of outdated and vulnerable libraries. Our work also uncovered patterns which highlight that developers lack the knowledge of the inner workings of browser-provided APIs, in at least one case actually introducing a vulnerability to begin with.

In summary, we find that there is no single reason for the existence of client-side XSS. Instead, the issues are caused by a number of factors, ranging from developers who are unaware of the security implications when using attacker-controllable data and their failed attempts at securing application to highly complex constructs of code and issues introduced by third parties.

Acknowledgements

10. REFERENCES

[1] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *WWW*, 2010.

[2] BuiltWith. jQuery Usage Statistics. `http://goo.gl/czK9XU` (accessed 16/05/15), 2015.

[3] I. Chowdhury and M. Zulkernine. Can Complexity, Coupling, and Cohesion Metrics Be Used As Early Indicators of Vulnerabilities? In *SAC*, 2010.

[4] A. Cortesi and M. Hils. mitmproxy. `https://goo.gl/VA9xw4` (accessed 16/05/15), 2014.

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

[6] M. E. Daggett. Enforcing Style. In *Expert JavaScript.* 2013.

[7] S. Di Paola. DominatorPro: Securing Next Generation of Web Applications. `https://goo.gl/L6tJth` (accessed 16/05/15), 2012.

[8] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the World Wide Web from Vulnerable JavaScript. In *International Symposium on Software Testing and Analysis*, 2011.

[9] I. Hickson and D. Hyatt. HTML 5 - A vocabulary and associated APIs for HTML and XHTML. W3c working draft, W3C, 2008.

[10] jQuery Bug Tracker. SELECTOR INTERPRETED AS HTML. `http://goo.gl/JNggpp` (accessed 16/05/15), 2012.

[11] A. Klein. DOM based cross site scripting or XSS of the third kind. *Web Application Security Consortium*, 2005.

[12] S. Lekies, B. Stock, and M. Johns. 25 Million Flows Later: Large-scale Detection of DOM-based XSS. In *CCS*, 2013.

[13] M. McDaniel and M. H. Heydari. Content based file type detection algorithms. In *HICSS*, 2003.

[14] F. Meawad, G. Richards, F. Morandat, and J. Vitek. Eval begone!: semi-automated removal of eval from javascript programs. *ACM SIGPLAN Notices*, 47, 2012.

[15] Mozilla Developer Network. Element.innerHTML - Web API Interfaces | MDN. `https://goo.gl/udFqtb` (accessed 16/05/15), 2015.

[16] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *CCS*, 2012.

[17] F. Ocariza, K. Pattabiraman, and B. Zorn. JavaScript errors in the wild: An empirical study. In *Software Reliability Engineering*, 2011.

[18] E. Oftedal. Retire.js - identify JavaScript libraries with known vulnerabilities in your application. `http://goo.gl/r4BQoG` (accessed 16/05/15), 2013.

[19] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. In *ECOOP*. 2011.

[20] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *PLDI*, 2010.

[21] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE S&P*, 2010.

[22] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[23] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda. Preventing input validation vulnerabilities in web applications through automated type analysis. In *Computer Software and Applications Conference*. IEEE, 2012.

[24] A. Seville. Blanket.js - seamless javascript code coverage. `http://goo.gl/hzJFTn` (accessed 16/05/15), 2014.

[25] Y. Shin, A. Meneely, L. Williams, and J. Osborne. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *Transactions on Software Engineering*, 2011.

[26] Y. Shin and L. Williams. An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics. In *International Symposium on Empirical Software Engineering and Measurement*, 2008.

[27] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against DOM-based cross-site scripting. In *USENIX Security*, 2014.

[28] The jQuery Foundation. Working with JSONP. `https://goo.gl/Wdqgo3` (accessed 16/05/15), 2015.

[29] W3Techs. Usage Statistics and Market Share of JQuery for Websites, February 2015. `http://goo.gl/jyQEZR` (accessed 16/05/15), 2015.

[30] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *International Conference on Software Engineering*, 2008.

[31] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *USENIX WOOT*, 2011.

[32] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *ACSAC*, 2012.