25 Million Flows Later - Large-scale Detection of DOM-based XSS

Sebastian Lekies
SAP AG
sebastian.lekies@sap.com

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

Martin Johns
SAP AG
martin.johns@sap.com

Abstract

In recent years, the Web witnessed a move towards sophisticated client-side functionality. This shift caused a significant increase in complexity of deployed JavaScript code and thus, a proportional growth in potential client-side vulnerabilities, with DOM-based Cross-site Scripting being a high impact representative of such security issues. In this paper, we present a fully automated system to detect and validate DOM-based XSS vulnerabilities, consisting of a taint-aware JavaScript engine and corresponding DOM implementation as well as a context-sensitive exploit generation approach. Using these components, we conducted a large-scale analysis of the Alexa top 5000. In this study, we identified 6167 unique vulnerabilities distributed over 480 domains, showing that 9,6% of the examined sites carry at least one DOM-based XSS problem.

Categories and Subject Descriptors

H.4.3 [**Communications Applications**]: Information browsers; H.6.5 [**Security and Protection**]: Unauthorized access

Keywords

DOM-based XSS, Taint Tracking, Vulnerability Detection, Exploit Generation

1. INTRODUCTION

The times in which JavaScript was mainly used for eye candy and small site enhancements are long gone. Since the advent of the so-called Web 2.0, the Web browser is the host of sophisticated, complex applications, such as Gmail or Google Docs, written entirely in JavaScript, that rival their desktop equivalents in scope and features. More and more functionality, which in traditional Web applications would have been implemented on the server, moves to the client. Consequently, the amount of required JavaScript code is increasing proportionally to this shift. Furthermore, the capabilities of client-side JavaScript are continuously increasing, due to the steady stream of new "HTML5" APIs being added to the Web browsers.

In parallel to this ever growing complexity of the Web's client side, one can observe an increasing number of security problems that manifest themselves only on the client [26, 11, 17]. One of these purely client-side security problems is *DOM-based XSS* [16], a vulnerability class subsuming all Cross-site Scripting problems that are caused by insecure handling of untrusted data through JavaScript. DOM-based XSS is caused by unsafe data flows from attacker-controlled sources, such as the `document.location` property, into security sensitive APIs, e.g., `document.write`.

While the existence of DOM-based XSS is known since 2005 [16], this vulnerability class is frequently still perceived as a minor, fringe issue, especially when being compared to *reflected* and *persistent* XSS. In this paper, we re-evaluate this assumption and examine how prevalent DOM-based XSS is in the wild.

Unfortunately, testing of client-side security properties in general, and DOM-based XSS in particular, is difficult. In comparison to the conditions on the server side, the Web's client side has several challenges that affect both static and dynamic security testing approaches: For one, all server-side code is completely under the control of the application's operator and available for processing, monitoring and analysis. This is not the case at the Web's client-side, where the code execution occurs on the user's machine. Furthermore, compared to server-side languages such as Java or C#, a large portion of JavaScript code frequently relies on runtime interpretation of string data as executeable code via APIs such as `eval()`. The resulting code is interpreted and executed on the client, making it invisible to the server. Finally, it is common practice for modern Web applications to include third-party JavaScript code using `script`-tags that point to cross-domain hosts. In 2002, Nikiforakis et al. [22] measured that 88.45% of the Alexa top 10,000 web sites included at least one remote JavaScript resource from a cross-domain host. This JavaScript is transported directly from the third-party provider to the user's Web browser and gets executed immediately. Thus, this code is neither directly controlled by the application nor is it visible at the server.

In this paper, we propose a fully automated system to identify DOM-based XSS issues, that overcomes the outlined obstacles through integrating the vulnerability detection directly into the browser's execution environment. Our system consists of a taint-aware JavaScript engine and DOM imple-

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516703.

mentation as well as a context-sensitive exploit generation technique.

The main contributions of this paper are the following:

- We present the design and implementation of a dynamic, byte-level taint-tracking approach in JavaScript engines. Through directly altering the JavaScript engine's implementation of the low-level string type, we achieve complete coverage of all JavaScript language features and the full DOM API.

- We propose a novel, fully automatic vulnerability validation mechanism, that leverages the fine-grained context information provided by our taint-aware JavaScript engine. Due to our exact knowledge of data source and syntactical context of the final data sink, our system can create attack payloads that match the syntactic surroundings of the injection point. This in turn allows unambiguous vulnerability validation through verification that our injected JavaScript was indeed executed. Thus, our system reports no false positives.

- We report on a large-scale empirical study on insecure data flows in client-side JavaScript and the resulting DOM-based XSS vulnerabilities. In total, we examined 504,275 URLs resulting from a shallow crawl of the Alexa top 5000 sites. In this study we observed a total of 24,474,306 flows out of which 69,987 caused validated DOM-based XSS exploits, resulting in 6,167 unique vulnerabilities affecting 9,6% of the examined sites.

The remainder of this paper is organized as follows: First we briefly revisit the technical background of DOM-based XSS (Sec. 2) and give a high-level overview of our approach (Sec. 3). Then, we describe our techniques for vulnerability detection (Sec. 4) and validation (Sec. 5). In Section 6 we present the methodology and results of our empirical study. We end the paper with a discussion of related work (Sec. 7) and a conclusion (Sec. 8).

2. DOM-BASED XSS

Cross-Site Scripting is an attack in which an attacker is able to inject his own JavaScript code into a Web application, in such a way that the code is executed within a victim's browser in the context of the application. Since 2000, when one of the first XSS vulnerabilities was reported [3], novel attack variants were discovered. In 2005, Amit Klein published a paper in which he first mentioned the term *DOM-based XSS* and described the basic characteristics of this vulnerability [16]. In contrast to traditional (reflected and persistent) Cross-Site Scripting, DOM-based XSS is caused by incorrect client-side code rather than by server-side code. As described earlier, the dynamic nature of this client-side code makes it hard to detect or verify this kind of vulnerability.

In order to trigger a DOM-based XSS exploit an attacker is able to utilize a multitude of different attack vectors to inject his malicious payload (such as `location.href`, `document.referrer`, `window.name`, and many, many more). Depending on the Web application's program logic, it processes attacker-controllable inputs and at some point in time conducts a string-to-code conversion. As shown in our empirical study, this is a very common scenario. If input is not sanitized correctly, the attacker may be able to inject his own code into the application. Thereby, different subtypes of DOM-based XSS exist depending on the method used for converting the string to code:

HTML context.

Web applications commonly insert generated HTML code into the DOM via functions such as `document.write`, `innerHTML` or `insertAdjacentHTML`. When these functions are called, the browser parses the string parameter and interprets the contents as HTML code, which is then inserted into a certain position within the DOM. If user input flows into these sinks, sanitization or encoding functions have to be used in order to avoid code injection vulnerabilities. If the input is not sanitized correctly an attacker is able to inject own HTML tags including `<script>`, which enables JavaScript execution. For the specific differences between `innerHTML` and `document.write`, we refer the reader to Sec. 5.2.1.

JavaScript context.

Another commonly used method, which is sometimes vulnerable to DOM-based XSS, is the `eval` function. `eval` takes a string parameter, interprets it as JavaScript code and executes it. Besides `eval` and its aliases `setTimeout` and `setInterval`, there are also other contexts in which strings are converted into JavaScript code such as `script.innerText`, `script.text`, `script.textContent` and the assignment of strings to event handler attributes.

URL context.

If an attacker-controlled input flows into a URL attribute of any DOM node (such as `img.src`, `iframe.src`, `object.data` or `a.href`), an immediate conversion from a string to code does not occur. However, there are still several security problems related to this kind of flows. For example, if the attacker is able to control the complete URL, he could make use of the `javascript:` or `data:` schemes to execute script code. If only parts of the URL are controlled, the attacker could still conduct redirects or phishing and in some cases even achieve JavaScript code execution as shown in Section 6.5.1.

Other contexts.

Besides those contexts that allow code execution, there are further sinks/contexts that are security sensitive such as `document.cookie`, the Web Storage API, postMessage or `setAttribute`. In Section 6.5.3, for example, we present a persistent DOM-based XSS vulnerability via `document.cookie`, which was discovered by our framework.

3. APPROACH OVERVIEW

In this paper, we propose a system to automatically detect and validate DOM-based XSS vulnerabilities. To address the outlined challenges in the assessment of client-side security problems (see Sec. 1), we decided to address the problem as follows: Instead of building analytical processes that complement [29] or emulate [25] the client-side behavior, we chose to integrate our techniques directly into a full browser.

More precisely, our system consists of two separate components: For vulnerability detection, we utilize a modified browsing engine that supports dynamic, byte-level taint-tracking of suspicious flows. Through directly altering the engine's string type implementation, we achieve complete coverage of all JavaScript language features and the full DOM API. We discuss the design and implementation in Section 4.

The second component is a fully automated vulnerability validation mechanism, that leverages the fine-grained context information provided by our taint-aware browsing engine. Due to the exact knowledge of data source and syntactical context of the final data sink, our system is able to create attack payloads that match the syntactic surroundings of the injection point. This in turn allows unambiguous vulnerability validation through verification that our injected JavaScript was indeed executed. This component is presented in Section 5.

4. VULNERABILITY DETECTION: MODIFIED CHROME

To automatically detect the flow of potentially attacker-controllable input (called 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 into 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 get information on the origin of each given character in said string.

4.1 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 assigned a numerical identifier to each of the 14 identified sources (e.g. `location.href`, `location.hash` or `document.referrer`). Hence, we were able to encode this information into the lower half of the byte. To also be able to determine whether a given character was encoded using the built-in functions *encodeURI*, *encodeURIComponent* and *escape*, we used the lower three of the four remaining bits to store whether one or more of these functions were applied to the string. To represent a benign character, the lower four bits are set to 0.

4.2 Patching the V8 JavaScript engine

Google's JavaScript engine V8 is highly optimized in regards to both memory allocation 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 objects 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 desired information directly into the header. Every object in V8 stores a pointer to its *map*. The map describes the class of an object. In V8, there are maps for each type of object. We found an used part of a bitmap in the maps and used it to create new map objects for tainted strings. Obviously, for strings of dynamic length, additional memory must be allocated to store the actual data. Based on whether a string is pure ASCII or also contains two-byte characters, this memory is allocated on creation of the object. The address of this newly created space is then written to one of the aforementioned 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 done without using the more complex 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 sub-string access.

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

4.3 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.

4.4 Detection of sink access

Until now we discussed the tracking of tainted data inside the V8 JavaScript engine and WebKit. The next step in our implementation was to detect a tainted flow and to notify

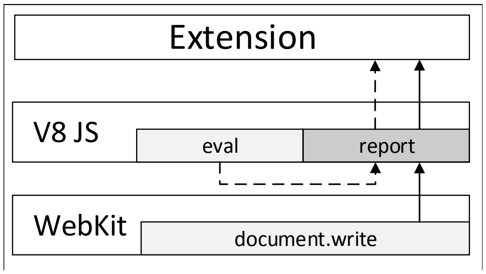

Figure 1: Report functionality

the user. Therefore, we modified all DOM-based Cross-Site Scripting sinks – like `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 6.1.

In WebKit's API used to provide access to the DOM tree for V8, the passed arguments are of V8's string class 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 1 depicts this layout. Both the 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.

5. VULNERABILITY VERIFICATION: AUTOMATIC EXPLOIT GENERATION

Although the taint-tracking engine delivers first indications for potential Cross-Site Scripting vulnerabilities, detecting a flow alone is not sufficient to ensure that a vulnerability was discovered. There are various reasons why a successful exploitation is not possible for an existing flow. For example, the Web site could use built-in or custom encoding or filter functions that are capable of defusing a malicious payload. Furthermore, other, random circumstance can occur that prevent an exploit from executing. For example, if the tainted value originates from a GET parameter, tampering with this parameter could trigger the Web server to load a different page or to display an error message in which the vulnerable flow is not present anymore. Therefore, a verification step is needed to tell vulnerable data flows apart from non-exploitable flows. In order to do so our system uses the data received from the taint- tracking engine to reliable generate valid Cross-Site Scripting Exploits. In this Section we describe the inner workings of the generation process.

5.1 Anatomy of a Cross-Site Scripting Exploit

To develop a system that is capable of generating valid XSS payloads, we first analyzed the nature of a Cross-Site Scripting exploit. In general, an exploit is context dependent. This means, that a payload, which an attacker seeks to execute, depends on how the Web application processes the attacker's input. So, if the input flows into the `eval` sink it has to utilize a different syntax than an exploit targeting flows into `document.write` (More details on context-dependent exploit generation can be found in the Section 5.2). However, the structure of an exploit can be generalized to a non-context-dependent form.

Listing 1 shows two typical exploits. The first exploit targets a JavaScript context (e.g. `eval`), while the second one contains an exploit for an HTML sink (e.g. `document.write`). In many cases a tainted value was concatenated from several different strings, which are hard coded (benign/non-attacker-controllable) or coming from either one or more sources (attacker-controllable). Therefore, an attacker is only able to control parts of the string that flows into the sink. Immediate execution of JavaScript is often not possible at the location where the tainted/controllable parts are inserted into the string/code (e.g. within quoted strings). Therefore, the exploit first has to break out of the current context to be able to execute the malicious script code. The first part of each exploit serves as a "break out sequence" to escape to a context where JavaScript execution is possible. In the cases presented in Listing 1 these sequences are "**');**" and "**>**", respectively. Following the break out sequence, an arbitrary JavaScript payload or `<script>` tag can be executed. Afterwards, the exploit has to take care of trailing string fragments in such a way that these fragments do not interfere with the execution of the payload. For example, if a string that is passed to *eval* contains a syntax error, no code will be executed at all, even if the syntax error occurs at the very end of the string. To prevent this from happening an exploit has to include an escape sequence that renders trailing characters harmless. In the JavaScript case we simply comment out everything that follows our payload and in the HTML case we close the script block and include a `<textarea>` to interpret the rest of the string as simple text instead of HTML. To summarize our analysis, we conclude that a Cross-Site Scripting exploit takes the following generalized form:

$$exploit := breakOutSequence\ payload\ escapeSequence \tag{1}$$

In this, only the *breakOutSequence* and the *escapeSequence* are context-specific. While the *escapeSequence* is very trivial to choose, the *breakOutSequence* needs careful crafting to result in a successful exploit.

Listing 1 Example Cross-Site Scripting exploits

```
');alert('XSS');//
"></a><script>alert('XSS')</script><textarea>
```

5.2 Context-Dependent Generation of Breakout Sequences

After discovering a data flow from one or more sources to a sink, the taint-tracking engine delivers three pieces of information to the exploit generation framework:

1. Information on the the data flow (sources, sink, applied built-in filters)
2. Tainted value: the complete string that flowed into the sink (including benign and tainted parts from one or more sources)
3. Byte-wise taint information for each byte contained in the tainted string.

Based on the given sink the framework first determines the target context. Depending on this context, the tainted value and the taint information are passed to a context-sensitive *break out sequence* generation function. In the next step, the generator adds the desired payload and a context-specific fixed escape sequence. After constructing the exploit, the system builds a test case that can be executed in a completely automated fashion and reports back to the framework in case of successful exploitation.

5.2.1 HTML context-specific generation

An HTML context is present whenever a tainted string is directly converted into HTML code. This is the case for many DOM manipulation functions such as `document.write` or `innerHTML`.

As mentioned before, often only parts of a string may be tainted. Therefore, our system first determines the exact location of the tainted parts by analyzing the taint information. In order to create a valid exploit, the system needs to determine into which DOM node the tainted parts will be transformed when the string-to-HTML conversion takes place. In order to do so, the generator parses the complete string and identifies the corresponding nodes. Based on the node types the generator is able to plan the next step within the generation process. In this first step we distinguish between three different node types (See Listing 2 for examples):

1. **Tainted TagNode:** The tainted value is located inside an HTML tag. Either it is part of the tag name, an attribute name, an attribute value or a combination of those three possibilities.
2. **Tainted CommentNode:** The tainted value is contained within an HTML comment.
3. **Tainted TextNode:** The tainted value is placed outside of an HTML tag or in between a closing and an opening tag.

Listing 2 Example Vulnerabilities

```
document.write('<script src="//example.org/'
          + taintedValue + '"></script>')

document.write('<div>' +taintedValue+ '</div>')

document.write('<!--' +taintedValue+ '-->')
```

Depending on the the node type, break out sequences have to be generated differently. In the following, we explain the three different approaches:

TagNode generation.

If the tainted value is included within an HTML tag we first need to break out of this tag. Otherwise, opening a `<script>` tag would have no effect. If the tainted value is directly located within the tag, we can simple do so by adding a ">" sign to the break out sequence. If the tainted value resides within an attribute of the tag, the system first needs to determine the delimiter of the attribute. Most of the time such attributes are either enclosed by single or double quotes, however, sometimes, also no delimiter is present. So in order to break out of the tag in this case we need to add the delimiter of the attribute node before the angle brackets.

Now our payload is able to break out of the current (opening) tag and would be able to open a `script` tag, to execute the payload. However, some tags have special semantics for the text between the opening and the closing tag. So for example, HTML markup between an opening and closing `iframe` tag is only rendered in case iframes are not supported by the browser. Therefore, our generator optionally adds one or more additional closing tags at the end of the break out sequence for all present tags with special semantics. To summarize this, a TagNode break out sequences looks as follows:

$$TagNodeBS := [delimiter] > [closingTags] \quad (2)$$

CommentNode generation.

The generation of CommentNode break out sequences is very trivial in most of the cases. As comments in HTML do not have any special semantics for their content, we can simply break out of a comment by adding "- ->" to our break out sequence. However, such a comment could in rare cases be placed in between opening and closing tags of scripts, iframes, etc. So, again our system analyzes the string and adds closing tags for these elements if necessary. Summing up, a CommentNode break out sequence takes the following form:

$$CommentNodeBS := -- > [closingTags] \quad (3)$$

TextNode generation.

Every character sequence that is placed outside a tag or a comment or that is located in between an opening and a closing tag is regarded as a TextNode by the HTML parser. In many cases executing a payload within a TextNode is straight forward. As we do not need to break out of the node itself, we can simply open a `script` tag and execute a payload. However, if the TextNode is placed between an opening and a closing tag of a `script` or `iframe` we again have to add closing tags if necessary.

$$TextNodeBS := [closingTags] \quad (4)$$

innerHTML vs document.write.

After we have generated the break out sequence for HTML context exploits, the system needs to choose a payload to execute. When doing so, some subtle differences in the handling of string-to-HTML conversion comes into play. When using `innerHTML`, `outerHTML` or `adjacentHTML` browsers react differently than `document.write` in terms of script execution. While `document.write` inserts `script` elements into the DOM and executes them immediately, `innerHTML` only performs the first step, but does not execute the script. So adding the following payload for an `innerHTML` flow would not result in a successful exploit:

```
<script>__reportingFunction__()</script>
```

However, it is still possible to execute scripts via an injection through `innerHTML`. In order to do so, the framework makes use of event handlers:

```
<img src="none_existent_resource"
onerror="__reportingFunction__()">
```

When `innerHTML` inserts the `img` tag, the browser creates an HTTP request to the non-existing resource. Obviously, this request will fail and trigger the `onerror` event handler that executes the given payload. Depending on the sink we simply choose one of these two payloads.

5.2.2 *JavaScript context-specific generation*

JavaScript context-specific generation is necessary whenever a data flow ends within a sink that interprets a string as JavaScript code. This is the case for functions such as `eval` & `Function`, Event handlers (such as `onload` and `onerror`) and DOM properties such as `script.textContent`, `script.text` and `script.innerText`. While browsers are very forgiving when parsing and executing syntactically incorrect HTML, they are quite strict when it comes to JavaScript code execution. If the JavaScript parser encounters a syntax error, it cancels the script execution for the complete block/function. Therefore, the big challenge for the exploit generator is to generate a syntactically correct exploit, that will not cause the parser to cancel the execution. In order to do so, the system again has to determine the exact location of the tainted bytes.

Listing 3 shows a very simple vulnerable piece of JavaScript code. In the first step, the code constructs a string of benign/hard coded and tainted (`location.href`) parts. In a second step, it executes the code using `eval`. Thereby, this code can be exploited in slightly different ways. Either the attacker could break out of the variable `x` and inject his code into the function named test, or he could break out of the variable `x` and the function `test` and inject his code into the top level JavaScript space. While the first method requires an additional invocation of the test function, the second exploit executes as soon as eval is called with a syntactically correct code. However, for the last case, the complexity of the break out sequence grows with the complexity of constructed code. Nevertheless, we do not want do rely on any behavior of other non-controllable code or wait for a user interaction to trigger an invocation of the manipulated code.

Therefore, we always seek to break out to the top level of the JavaScript execution. In order to do so, our system first parses the JavaScript string and creates a syntax tree of the code. Based on this tree and the taint information we extract the branches that contain tainted values. Listing 4 shows the resulting syntax tree for our example code and

Listing 3 JavaScript context example

```
var code =  'function test(){' +
            'var x = "' + location.href + '";'
            //inside function test
            + 'doSomething(x);'
        + '}'; //top level
eval(code)
```

Listing 4 JavaScript Syntax Tree

```
FunctionDeclaration
  Identifier : test
  FunctionConstructor
    Identifier : test
    Block
      Declaration
        Identifier : x
        StringLiteral : "http://example.org"
    ExpressionStmt
      SpecialOperation : FUNCTION_CALL
        Reference
          Identifier : doSomething
```

the extracted branch (in gray). For each of the extracted branches the generator creates one break out sequence by traversing the branch from top to bottom and adding a fixed sequence of closing/break out characters for each node. So in our example the following steps are taken:

1. FunctionDeclaration: ';'
2. FunctionConstructor: ''
3. Block: '}'
4. Declaration: ';'
5. StringLiteral: '"'
6. Resulting Breakout Sequence: '";};'

To trigger the exploit we can simply construct the test case as follows: Based on the source (`location.href`), the system simple adds the break out sequence, an arbitrary payload and the escape sequence to the URL of the page:

```
http://example.org/#";};__reportingFunction__();//
```

When executed within a browser, the string construction process from Listing 3 is conducted and the following string flows into the `eval` call (Note: Line breaks are only inserted for readability reasons):

```
function test(){
    var x = "http://example.org/#";
};
__reportingFunction__();
//doSomething(x);}
```

6. EMPIRICAL STUDY

As mentioned earlier, an important motivation for our work was to gain insight into the prevalence and nature of potentially insecure data flows in current JavaScript applications leading to DOM-based XSS. For this reason, we created a Web crawling infrastructure capable of automatically applying our vulnerability detection and validation techniques to a large set of real-world Web sites.

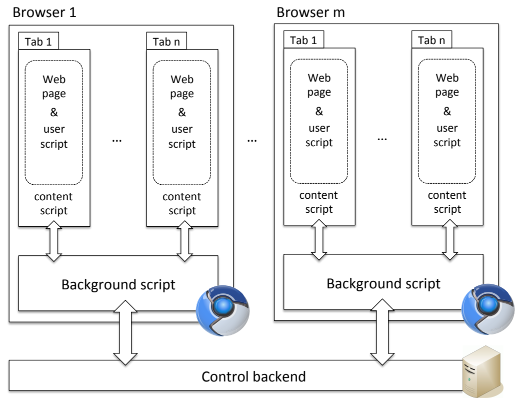

Figure 2: Crawling infrastructure

6.1 Methodology & Architecture Overview

To obtain a realistic picture on the commonness of insecure data flows that might lead to DOM-based XSS, it is essential to sample a sufficiently large set of real-world Web sites.

We designed our experiment set-up to meet these requirements, utilizing the following components: Our flow-tracking rendering engine to identify and record potentially unsafe JavaScripts (as discussed in Sec. 4), our exploit generation and validation framework (as presented in Sec. 5), and a crawling infrastructure that automatically causes the browsing engine to visit and examine a large set of URLs.

Our crawling infrastructure consisted of several browser instances and a central backend, which steered the crawling process. Each browser was outfitted with an extension that provided the browser with the required external interface for communication with the backend (see Fig. 2). In the following paragraphs, we briefly document both the backend's and the extension's functionality.

6.1.1 Analysis engine: Central server backend

The main duty of the central analysis backend is to distribute the URLs of the examination targets to the browser instances and the processing of the returned information. The backend maintains a central URL queue, which was initially populated with the Alexa Top 5000 domains and subsequently filled with the URLs that were found by the browsers during the crawling process.

The browser instances transmit their analysis report and their findings to the backend. For each analyzed URL, analysis reports for several URLs are returned, as the browser instances not only check the main page but also all contained `iframes`. In our study, we received results for an average of 8.64 (sub-)frames for each URL that was given to a browser instance. After pre-processing and initial filtering, the backend passes the suspicious flows to the exploit generation unit (see Sec. 6.3).

6.1.2 Data collection: Browser Extension

As discussed in Section 4, we kept direct changes to the browser's core engine as small as possible, to avoid unwanted side effects and provide maintainability of our modifications. Our patches to the browser's internal implementation consisted mainly in adding the taint-tracking capabilities to the Javascript engine and DOM implementation. All further browser features that were needed for the crawling and analyzing processes were realized in the form of a browser extension.

Following the general architecture of Chrome's extension model [8], the extension consists of a background and a content script (see Fig. 2). The *background script*'s purpose is to request target URLs from the backend, assign these URLs to the browser's tabs (for each browser instance, the extension opened a predefined number of separate browser tabs to parallelize the crawling process), and report the findings to the backend. The *content script* conducts all actions that directly apply to individual Web documents, such as collecting the hyperlinks contained in the page for the further crawling process and processing the data flow reports from the taint-tracking engine (see Sec. 4.4). Furthermore, the content script injects a small *userscript* into each Web document, that prevents the examined page from displaying modal dialogues, such as `alert()` or `confirm()` message boxes, which could interrupt the unobserved crawling process.

After the background script assigns a URL to a tab, the content script instructs the tab to load the URL and render the corresponding Web page. This implicitly causes all further external (script) resources to be retrieved and all scripts, that are contained in the page, to be executed. After the page loading process has finished, a timeout is set to allow asynchronous loading processes and script execution to terminate. After the timeout has passed, the content script packs all suspicious data flows, which were reported during execution of the analyzed page, and communicates them to the background script for further processing.

In addition to data flow and hyperlink data, the extension also collects statistical information in respect to size and nature of the JavaScripts that are used by the examined sites.

6.2 Observed Data Flows

As mentioned above, our initial set of URLs consisted of the Alexa top 5000. For each of these URLs we conducted a shallow crawl, i.e., all same-domain links found in the respective homepages were followed, resulting in 504,275 accessed Web pages. On average each of those Web document consisted out of 8.64 frames resulting in a final number of 4,358,031 (not necessary unique) URLs.

In total our infrastructure captured 24,474,306 data flows from potentially tainted sources to security sensitive sinks. Please refer to Table 1 for details on the distribution of flows, depicted by their sources and sinks.

6.3 Selective Exploit Generation

As shown in the previous Section, the total number of potentially vulnerable data flows from insecure sources to security sensitive sinks is surprisingly high. In our study, the sheer number of found flows exceeds the number of analyzed pages by a factor of about 48.5.

Both our exploit generation and validation processes are efficient. Generating and testing an XSS exploit for a selected data flow requires roughly as much time as the initial analyzing process of the corresponding Web page. However, due to the large amount of observed flows, testing all data flows would have required significantly more time than the actual crawling process. Hence, to balance our coverage and broadness goals, we selected a subset out of all recorded, po-

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

Table 1: Data flow overview, mapping sources (top) to sinks (left)

tentially vulnerable data flows, based on the following criteria:

(C1) The data flow ended in a sink that allows, if no further sanitization steps were taken, direct JavaScript execution. Hence, all flow into cookies, Web Storage, or DOM attribute values were excluded.

(C2) The data flow originates from a source that can immediately be controlled by the adversary, without programmatic preconditions or assumptions in respect to the processing code. This criteria effectively excluded all flows that come from second order sources, such as cookies or Web Storage, as well as flows from the `postMessage` API.

(C3) Only data flows without any built-in escaping methods and data flows with non-matching escaping methods were considered. Data flows, for which the observed built-in escaping methods indeed provide appropriate protection for the flow's final sink were excluded.

(C4) For each of the remaining data flows we generated exploits. However, many flows led to the generation of exactly the same exploit payloads for exactly the same URL - e.g. when a web page inserts three scripts via `document.write` and always includes `location.hash` at a similar location. In order to decrease the overhead for testing the exploits, our system only validates one of these exploits.

Starting from initial 24,474,306 flows, we successively applied the outlined criteria to establish the set of relevant flows:

$$24,474,306 \xrightarrow{C1} 4,948,264 \xrightarrow{C2} 1,825,598 \\ \xrightarrow{C3} 313.794 \xrightarrow{C4} 181,238 \quad (5)$$

Thus, in total we generated 181,238 test payloads, out of which a total of 69,987 successfully caused the injected JavaScript to execute. We discuss the specifics of these results in the next Section.

6.4 Found vulnerabilities

In total, we generated a dataset of 181,238 test payloads utilizing several combinations of sources and sinks. As discussed in Section 6.3 (C3), all flows which are encoded are filtered early on. For Google Chromium, which we used in our testing infrastructure, adhering to this rule we also must filter all those exploits that use either `location.search` or `document.referrer` to carry the payloads. This is due to the fact that both these values are automatically encoded by Chromium. Hence, we chose to test these vulnerabilities in Internet Explorer 10 whereas the rest of the URLs were verified using our aforementioned crawling infrastructure. Since the number of exploits utilizing `search` vulnerabilities amounts to 38,329 and the sum for `referrer` reached 5083, the total number of exploits tested in Chromium was reduced to 137,826, whereas the remaining 43,412 exploits were tested using Internet Explorer.

Out of these, a total number of 58,066 URLs tested in Chromium triggered our verification payload. Additionally, we could exploit 11,921 URLs visited in Internet Explorer. This corresponds to a success rate of 38.61% in total, and a success rate of 42.13% when only considering vulnerabilities exploitable in Chromium.

As we discussed earlier, we crawled down one level from 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. 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:

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

In regards 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 5000 domains. Thus, we applied another filter, removing all exploits from these domains outside the top 5000. This reduced the number of unique exploits to 6,167, stemming from 480 different domains. In respect to the number of domains we originally crawled, this means that our infrastructure found working exploits on *9.6%* of the 5000 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 5000 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.

6.5 Selected Case Studies

During the analysis of our findings, we encountered several vulnerabilities which exposed interesting characteristics. In the following subsections, we provide additional insight into these cases.

6.5.1 *JSONP + HTTP Parameter Pollution*

As stated in Section 2, flows into URL sinks are not easily exploitable. Only if the attacker controls the complete string, he can make use of data and javascript URLs to execute JavaScript code. However, in our dataset we found a particularly interesting coding pattern, that allows script execution despite the fact that the attacker only controls parts of the injected URLs. In order to abuse this pattern a Web page must assign a partly tainted string to a script.src attribute that includes a JSONP script with a callback parameter (See Listing 5).

Listing 5 JSONP script include

```
var script = document.createElement('script')
script.src = "http://example.org/data.json?u="
        + taintedValue + "&callback=cb_name";
```

In many cases the callback parameter is reflected back into the script in an unencoded/unfiltered fashion. Hence, the attacker could inject his own code into the script via this parameter. However, the callback parameter is hard coded and the attacker is not able to tamper with it at first sight. Nevertheless, it is possible to inject a second callback parameter into the script URL via the `taintedValue`. This results in the fact that two parameters with the same name and different values are sent to the server when requesting the script. Depending on the server-side logic the server will either choose the first or the second parameter (We found both situations, and depending on the position of the `taintedValue` we were able to exploit both situations). Hence, by conducting this so-called HTTP Parameter Pollution attack, the attacker is able to inject his value into the content of the script, which is afterwards embedded into the Web page.

One particularly interesting fact is that simply encoding the `taintedValue` will not protect against exploitation. Instead, the JSONP callback parameter needs to be sanitized. During our experiments we found one vulnerable callback parameter quite often on many different Web sites, which seemed to stem from jQuery (or at least, always called the same jQuery function).

6.5.2 *Broken URL parsing*

As browsers sometimes auto-encode certain parts of user controlled values, it is not possible to inject code into some of the analyzed sources. One example for this is `location.search` that is auto-encoded by all browser except Internet Explorer. Another source that is encoded by every modern browser is `location.pathname`. An injection via `location.pathname` is in general not possible until the application itself decodes the value. An additional encoding or sanitization step is therefore not necessary for these values. This fact, however, also leads to security vulnerabilities when Web developers trust in this auto-encoding feature while at the same time conducting incorrect URL parsing.

In our analysis, we found many examples where this fact leads to vulnerabilities. In the following we cover some examples where code fragments seemed to extract automatically encoded values (and hence no sanitization is needed), but due to non-standard parsing, extracted also unencoded parts in malicious cases.

1. Task: Extract host from URL
2. What it really does: Extract everything between `www.` and `.com` (e.g. whole URL)
3. e.g. `http://www.example.com/#notEncoded.com`

```
var regex = new RegExp("/www\..*\.com/g");
var result = regex.exec(location.href);
```

1. Task: Extract GET parameter `foo`
2. What it really does: Extracts something that starts with `foo=`
3. e.g. `http://www.example.com/#?foo=notEncoded`

```
var regex = new RegExp( "[\\?&]foo=([^&#]*)" );
var result = regex.exec(location.href);
```

1. Task: Extract all GET parameters
2. What it really does: Last GET parameter contains the unencoded Hash
3. e.g. `http://example.com/?foo=bar#notEncoded`

```
location.href.split('?')[1].split('&')[x]
                              .split('=')
```

6.5.3 *Persistent DOM-based XSS*

As seen in Table 1, our system captured also some flows into cookies and into the Web Storage API. However, we did not include it into our automatic exploit generation. Nevertheless, we were able to manually find several persistent DOM-based XSS. We detected flows that first came from user input and went into Cookie or Web Storage sinks effectively persisting the data within the user's browser. In the cases where we could trigger a successful exploit, this data was then used in a call to `eval`, hence exposing the Web site to persistent DOM-based XSS.

6.5.4 *window.name flows*

Within our dataset, we detected a surprisingly high number (>2 million) of flows originating from `window.name` that we couldn't explain at first sight. Although some of them were exploitable, we soon discovered the reason for this number. Most of these flows are not exploitable via DOM XSS as they are caused by a simple programming error. When declaring a local variable a developer has to use the `var` keyword. If someone declares a variable named `name` inside a function and misses the `var` keyword or if a local variable is created directly within a script block that is executed in the global scope, the variable is declared global (See Listings 6 and 7). Since inside a browser, the global object is `window`, the data is written to `window.name`. If the same variable is used within a call to a sink within the same script block, the corresponding flow is not exploitable as window.name was overwritten with benign data. However, this fact represents another serious issue: `window.name` is one of the very few

Listing 6 window.name bug 1: Missing var keyword

```
function test(){
    name = doSomething();
    document.write(name);
};
```

Listing 7 window.name bug 2: Declaration within the global scope

```
<script>
    var name = doSomething();
    document.write(name);
</script>
```

properties that can be accessed across domain boundaries. Hence, any data that is written to this property can be accessed by third parties. This programming error, therefore, represents a serious information leakage problem, if sensitive data is written to such a global `name` variable. Given the huge amount of flows, it is very likely that this pattern could be misused to steal sensitive information.

6.6 Effectiveness of Chromium's XSS Filter

Modern browsers like Chromium and its commercial counterpart Google Chrome are equipped with client-side filter capabilities aiming at preventing XSS attacks [1]. In order to analyze the effectiveness of Chromium's XSS Filter, we utilized our successful exploits and tried to execute them with the activated filter. Out of the 701 domains we found, 300 domains were still susceptible to XSS even with Chromium's auditor enabled.

After further examination, we found three distinguishing characteristics for these working exploits. For one, none of the exploits abusing JavaScript sinks, such as `eval()`, were detected by XSS Auditor. This stems from the fact that the auditor is implemented inside the HTML parser and thus cannot detected direct JavaScript sinks. Furthermore, exploits that were caused by remote script includes were not detected. The third type of undetected exploits was caused by JSONP vulnerabilities as discussed in Section 6.5.1.

On a positive note, in our study, we found that none of the exploits that targeted inline vulnerabilities passed through the filter. However, please note, that this experiment carries no reliable indication of protection robustness in respect to the exploits, that were stopped. We did not make any attempts to obfuscate the exploit payload [12] or use other filter evasion tricks [13].

In 2011 Nikiforakis demonstrated that Chrome's filter is not able to cope with exploits that utilize more than one injection point at once [21]. If we take our figures from Section 6.2 into account, we see that a tainted string consists – on average – of about three tainted substrings. Thus, an attacker has on average three possible injection points in order to leverage the techniques presented by Nikiforakis. Therefore, we have good reasons to believe that the numbers presented in this Section must rather be seen as a lower bound.

7. RELATED WORK

To the best of our knowledge, DOMinator [7] was the first browser-based tool to test for DOM-based XSS via dynamic taint-tracking. For this purpose, DOMinator instruments Firefox's SpiderMonkey JavaScript engine. Unlike our technique, DOMinator does not track data flows on a byte level. Instead, it employs a function tracking history to store the operations which were called on the original, tainted input to result into the final, still tainted, string flowing into a sink. Also, it does not feature a fully automated vulnerability validation.

FLAX [25] is the conceptionally closest approach to our work. Similar to our system, FLAX also utilizes byte-level taint-tracking to identify insecure data flows in JavaScript. However, there are several key differences in which we improve over FLAX: For one, FLAX's taint analysis is not fully integrated in the JavaScript engine. Instead, the actual analysis is done on program slices which are translated into JASIL, a simplified version of JavaScript, which expresses the operational semantics of only a subset of JavaScript. In contrast, through extending JavaScript's low-level string type, we achieve full language and API coverage. Furthermore, FLAX employs fuzzing for vulnerability testing, while our approach leverages the precise source and sink context information to create validation payloads that deterministically match the respective data flows specifics. Finally, using a large scale study we successfully demonstrated that our system is compatible with the current code practices in today's Web. In contrast, FLAX was only practically evaluated on a set of 40 Web applications and widgets.

Criscione [5] presented an automatic tool to find XSS problems in a scalable black box fashion. Similar to our approach, they also use actual browser instances for test execution and vulnerability validation. However, they don't utilize taint propagation or precise payload generation. Instead, the tests are done in a fuzzing fashion.

Finally, a related approach was presented by Vogt et. al. [30], which utilizes a combination of static analysis and dynamic information flow tracking to mitigate XSS exploits. However, instead of following the flow of untrusted data, the focus is on security sensitive values, such as the user's cookie, and the potential exfiltration of those.

Server-side approaches and static taint analysis: On the server-side various approaches using dynamic taint-tracking to detect and mitigate XSS vulnerabilities have been proposed [20, 23, 4, 27, 19, 33, 2]. Furthermore, as an alternative to dynamic taint-tracking, static analysis of source code to identify insecure data flows is a well established tool [9, 28, 31, 14, 32, 10].

Attack generation: In order to decrease false positive rates several approaches have been studied to automatically generate a valid exploit payloads for validation purposes. In 2008, Martin et al. [18] presented a method to generate XSS and SQL injection exploits based on goal-directed model checking. Thereby, their system QED is capable of performing a goal-directed analysis of any Java Web application, which adheres to the standard servlet specification. Based on the constructed model, a model checker is able to generate a valid exploit that can be used to validate the finding. As opposed to our approach the system operates on the server-side code and thus focuses on server-side injection vulnerabilities. Similar to this approach, Kieyzun et al. [15] also focus on the automatic generation of attacks targeting server-side injection vulnerabilities. In order to do so, the authors use symbolic taint-tracking and input mutations to generate example exploits. Thereby, several test inputs are transmitted to the target service and depending on the registered data flows, inputs are mutated to generate malicious payloads. As opposed to our approach, their tool Ardilla also only works on server-side code and thus rather targets traditional XSS vulnerabilities. As it requires several HTTP requests for generating a valid exploit, scaling is far more difficult than with our approach. In [6], d'Amore et al. present the tool *snuck* that is capable of automatically evading server-side XSS filters. To function, however, the tool needs input from a human tester that identifies the application's intended workflows and possible injection vectors. The tool then automatically verifies whether the filter functions works in a correct manner. In order to do so the system identifies the exact injection context by using XPath queries.

Empirical studies on JavaScript security: Due to its ever growing importance in the Web application paradigm, several security-relevant aspects of client-side JavaScript execution have been studied empirically. For one, Yue and Wang [34] examined the commonness of JavaScript practices that could lead to unauthorized code execution, namely cross-domain inclusion of external JavaScript files and usage of APIs that could lead to XSS. Their study is purely statistically and no real vulnerability validation was conducted. Zooming in on `eval`, Richards et al. [24] study how this problematic API is used in the wild, identifying both usage patterns that could be solved with safe alternatives as well as instances, in which replacing `eval` would not be a straight forward task. Furthermore, selected "HTML5" JavaScript APIs have been studied in detail: Lekies & Johns [17] surveyed the Alexa top 500,000 for potentially insecure usage of JavaScript's `localStorage` for code caching purposes and Son & Shmatikov [26] examined the Alexa top 10,000 for vulnerabilities occurring from unsafe utilization of the `postMessage` API.

8. CONCLUSION

In this paper, we presented a fully automated approach to detect and validate DOM-based XSS vulnerabilities. By direct integration into the browser's JavaScript engine, we achieve reliable identification of potentially insecure data flows while maintaining full compatibility with productive JavaScript code. Furthermore, the precise, byte-level context informations of the resulting injection points enables us to create attack payloads which are tailored to the vulnerability's specific conditions, thus, allowing for robust exploit generation.

Using our system, we conducted a large scale empirical study, resulting in the identification of 6,167 unique vulnerabilities distributed over 480 domains, demonstrating that 9,6% of the Alexa top 5000 carry at least one DOM-based XSS problem.

Acknowledgments

This work was in parts supported by the EU Projects WebSand (FP7-256964) and STREWS (FP7-318097). The support is gratefully acknowledged.

9. REFERENCES

[1] Bates, D., Barth, A., and Jackson, C. Regular expressions considered harmful in client-side XSS filters. In *WWW '10: Proceedings of the 19th international conference on World wide web* (New York, NY, USA, 2010), ACM, pp. 91–100.

[2] Bisht, P., and Venkatakrishnan, V. N. XSS-GUARD: Precise dynamic detection of cross-site scripting attacks. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'08)* (2008).

[3] CERT. Advisory ca-2000-02 malicious html tags embedded in client web requests, February 2000.

[4] Conti, J. J., and Russo, A. A taint mode for python via a library. In *NordSec* (2010), T. Aura, K. Järvinen, and K. Nyberg, Eds., vol. 7127 of *Lecture Notes in Computer Science*, Springer, pp. 210–222.

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

[6] d'Amore, F., and Gentile, M. Automatic and context-aware cross-site scripting filter evasion. *Department of Computer, Control, and Management Engineering Antonio Ruberti Technical Reports 1*, 4 (2012).

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

[8] Google Developers. Chrome Extensions - Developer's Guide. [online], `http://developer.chrome.com/extensions/devguide.html`, last access 06/05/13, 2012.

[9] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable javascript. In *ISSTA* (2011), M. B. Dwyer and F. Tip, Eds., ACM, pp. 177–187.

[10] Guha, A., Krishnamurthi, S., and Jim, T. Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web (WWW'09)* (New York, NY, USA, 2009), ACM, pp. 561–570.

[11] Hanna, S., Chul, E., Shin, R., Akhawe, D., Boehm, A., Saxena, P., and Song, D. The emperor's new apis: On the (in) secure usage of new client-side primitives. In *Web 2.0 Security and Privacy (W2SP 2010)* (2010).

[12] Heiderich, M., Nava, E., Heyes, G., and Lindsay, D. *Web Application Obfuscation: -/WAFs..Evasion..Filters//alert (/Obfuscation/)-*. Elsevier/Syngress, 2010.

[13] Heyes, G. Bypassing XSS Auditor. [online], `http://www.thespanner.co.uk/2013/02/19/bypassing-xss-auditor/`, last accessed 08/05/13, February 2013.

[14] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy* (May 2006).

[15] Kieyzun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. Automatic creation of sql injection and cross-site scripting attacks. In *Proceedings of the 31st*

International Conference on Software Engineering (Washington, DC, USA, 2009), ICSE '09, IEEE Computer Society, pp. 199–209.

[16] Klein, A. Dom based cross site scripting or xss of the third kind. *Web Application Security Consortium, Articles 4* (2005).

[17] Lekies, S., and Johns, M. Lightweight Integrity Protection for Web Storage-driven Content Caching. In *6th Workshop on Web 2.0 Security and Privacy (W2SP 2012)* (May 2012).

[18] Martin, M., and Lam, M. S. Automatic Generation of XSS and SQL Injection Attacks with Goal-Directed Model Checking. In *Usenix Security* (2008).

[19] Nadji, Y., Saxena, P., and Song, D. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Network & Distributed System Security Symposium (NDSS 2009)* (2009).

[20] Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. Automatically hardening web applications using precise tainting. In *20th IFIP International Information Security Conference* (May 2005).

[21] Nikiforakis, N. Bypassing Chrome's Anti-XSS filter. [online], `http://blog.securitee.org/?p=37`, last access 08/05/13, September 2011.

[22] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Acker, S. V., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *19th ACM Conference on Computer and Communications Security (CCS 2012)* (2012).

[23] Pietraszek, T., and Berghe, C. V. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection (RAID2005)* (2005).

[24] Richards, G., Hammer, C., Burg, B., and Vitek, J. The eval that men do - a large-scale study of the use of eval in javascript applications. In *ECOOP* (2011), M. Mezini, Ed., vol. 6813 of *Lecture Notes in Computer Science*, Springer, pp. 52–78.

[25] Saxena, P., Hanna, S., Poosankam, P., and Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In *NDSS* (2010), The Internet Society.

[26] Son, S., and Shmatikov, V. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *Network and Distributed System Security Symposium (NDSS'13)* (2013).

[27] Su, Z., and Wassermann, G. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of POPL'06* (January 2006).

[28] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: Effective Taint Analysis for Java. In *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation (PLDI 2009)* (June 2009).

[29] Vikram, K., Prateek, A., and Livshits, B. Ripley: Automatically securing distributed Web applications through replicated execution. In *Conference on Computer and Communications Security* (Oct. 2009).

[30] Vogt, P., Nentwich, F., Jovanovic, N., Kruegel, C., Kirda, E., and Vigna, G. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)* (2007).

[31] Wassermann, G., and Su, Z. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of Programming Language Design and Implementation (PLDI'07)* (San Diego, CA, June 10-13 2007).

[32] Xie, Y., and Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *15th USENIX Security Symposium* (2006).

[33] Xu, W., Bhatkar, S., and Sekar, R. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *15th USENIX Security Symposium* (August 2006).

[34] Yue, C., and Wang, H. Characterizing insecure javascript practices on the web. In *WWW* (2009), J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, Eds., ACM, pp. 961–970.