Protection, Usability and Improvements in Reflected XSS Filters[*]

Riccardo Pelizzi
Stony Brook University
r.pelizzi@gmail.com

R. Sekar
Stony Brook University
sekar@cs.sunysb.edu

ABSTRACT

Due to the high popularity of Cross-Site Scripting (XSS) attacks, most major browsers now include or support filters to protect against reflected XSS attacks. Internet Explorer and Google Chrome provide built-in filters, while Firefox supports extensions that provide this functionality. In this paper, we analyze the two most popular open-source XSS filters, XSSAuditor for Google Chrome and NoScript for Firefox. We point out their weaknesses, and present a new browser-resident defense called XSSFilt. In contrast with previous browser defenses that were focused on the detection of whole new scripts, XSSFilt can also detect *partial script injections,* i.e., alterations of existing scripts by injecting malicious parameter values. Our evaluation shows that a significant fraction of sites vulnerable to reflected XSS can be exploited using partial injections. A second strength of XSSFilt is its use of approximate rather than exact string matching to detect reflected content, which makes it more robust for web sites that employ custom input sanitizations. We provide a detailed experimental evaluation to compare the three filters with respect to their usability and protection.

Categories and Subject Descriptors

D.4.6 [**Security And Protection**]: Information Flow Controls

1. INTRODUCTION

Cross-site scripting (XSS) has emerged as one of the most serious threats on the Web. CWE/SANS Top-25 [25] lists XSS fourth in its list of "Top 25 Most Dangerous Software Errors," while the web-application focused OWASP [26] lists XSS second in its list of top-10 security risks. In terms of raw numbers, it is one of the most commonly reported vulnerabilities in 2011, accounting for 14.7% of all reported CVE vulnerabilities.

The increase in prevalence and severity of XSS attacks has spawned several research efforts into XSS defenses. Many of these efforts [12, 13, 15, 27, 11, 28, 3, 23] have focused on the server-side, and attempt to detect (or prevent) unauthorized scripts from being included in the server output. Modern web-browsers incorporate very complex logic to "fix" HTML syntax errors and hence provide an acceptable rendering of syntactically incorrect pages. Several researchers [11, 15, 27, 13] have eloquently argued that no server-side logic can accurately account for all such "browser quirks." As a result, hybrid approaches that combine client-side support with a primarily server-side XSS defense have been developed [27, 15, 23].

Since XSS is a server-side vulnerability, it seems natural to employ a server-side defense. Unfortunately, the party that is most directly affected by an XSS attack is a browser-user that accesses a vulnerable server. Consequently, there may not be enough of an incentive for some web sites to implement XSS defenses — this is one reason why XSS vulnerabilities are so easy to find. Client-side protections are thus desirable, despite their limitation to so-called *reflected XSS:* in these attacks, the output of a vulnerable server includes some script content provided in a malicious request.

Reflected XSS attacks are launched via invalid (and malicious) values for parameters in a HTTP request. Consequently, NoScript[1], a Firefox plug-in, defends against these attacks by filtering out parameter values that look suspicious. Specifically, it uses regular expressions to identify the presence of JavaScript code in HTTP request parameters, and sanitizes them before submitting the request.

An important difficulty with NoScript's approach (of client-side parameter sanitization) is that it should work without any knowledge about how a server may use a parameter. In particular, it needs to prevent XSS attacks regardless of how the server-side may use a parameter value. This may lead to overly strict filtering that can prevent some web pages from being used, or interfere with their functionality. In contrast, an approach that can examine both the request and the response from the server can be more discriminating about

[*]This work was supported in part by an NSF grant CNS-0831298, an AFOSR grant FA9550-09-1-0539, and an ONR grant N000140710928.

[1]NoScript is mainly thought of as a tool that blocks the execution of scripts from most sites. However, blocking script execution is not a realistic option for web sites trusted by a user, which will be the ones targeted in a reflected XSS. For this reason, NoScript also contains a client-side XSS filter that sanitizes requests submitted by the browser. From here on, we will refer to this XSS filter when we use the term "NoScript filter."

```
<script>
document.write(
  '<a href="../plugin.php?passed_id=' +
  '<?=$_GET["id"];?>"></a>');
</script>
```

```
id:
'); do_xss(); document.write('
```

Figure 1: An example server-side script (abstracted from the popular SquirrelMail web-based email program) with partial injection vulnerability (left) and a malicious parameter value to exploit it (right).

which requests are unsafe. Microsoft's IE8 uses such an approach for client-side XSS defense. It first marks requests that look suspicious. Responses to such requests are then scanned for script content that may be derived from suspicious parameters, and this content is then "sanitized" to prevent its interpretation as a script.

Unfortunately, identification of unsafe content is very hard because of a browser's HTML parsing quirks. Researchers have shown [16] that there are several ways to bypass detection by IE8 filter. Worse, the sanitization technique used in IE8 could be exploited [17] to perpetrate XSS attacks on some sites that weren't previously vulnerable! In particular, the sanitization technique caused some other part of the page that was previously interpreted as passive content to be interpreted as a script. Although the specific vulnerability reported in [17] has been fixed, the nature of their architecture makes it difficult to rule out similar vulnerabilities in the future.

Google Chrome's XSSAuditor [3] is based on the same approach, but employs a different architecture that avoids "browser quirks" problem by directly interposing at the JavaScript engine interface. Consequently, XSSAuditor does not rely on guess work, but gets to examine content that is *actually interpreted as a script.* If this script "resembles" a request submitted by the browser, XSSAuditor skips its execution. This simple approach avoids IE8's sanitization pitfall, since preventing the execution of a script does not change the interpretation of the rest of the page. Moreover, this new architecture can address DOM-based XSS vulnerabilities that can arise in pages which are created dynamically as the result of client-side script execution. (With the advent of Web 2.0, such dynamic pages are increasingly becoming the norm.)

1.1 Limitations of existing filters

Although XSSAuditor overcomes the main drawbacks of the IE8 filter, it does not address the following problems:

- *Whole Vs partial script injection:* XSSAuditor is geared towards detecting the most common form of XSS, where an entire script is injected into a victim page. However, damage can also be effected by altering the structure of an existing script. Figure 1 shows an example abstracted from the web application SquirrelMail, where a GET parameter named `id` is inserted into a `document.write` call in an existing script (left frame). Even though the intent of the developers is to dynamically write an anchor tag, the logic can be subverted to inject arbitrary JavaScript code. Note the similarity of the malicious input (right frame) with those used in SQL injections: first, the string argument previously opened by benign code is closed; then the payload is inserted; finally, tokens are inserted to synchronize the syntax with the rest of the benign script and thus avoid syntax errors. In nearly all cases (including the example presented), the vulnerability allows for arbitrary code injection into the existing script, thus being as severe as a whole script injection.

  These sorts of vulnerabilities arise naturally in template-based web application development frameworks and in dynamic web applications in general. Our experimental results (see Section 7) demonstrate that partial injection vulnerabilities are common, accounting for 8% and 18% respectively in two collections of vulnerabilities. Moreover, as the first generation client-side defenses (against whole-script injection) get deployed, attackers are bound to try evading them through partial script injections.

- *Accurate algorithms for detecting injections:* XSSAuditor uses an exact string matching algorithm to detect components of a web application's output that have been derived from the input request. Character encoding and sanitizations incorporated into typical web applications are performed before string matching. However, this approach does not handle application-specific sanitizations that may take place. A more systematic approach would rely on approximate string matching in order to (a) better cope with application-specific sanitizations, and (b) to more precisely identify the beginning and end of injected strings in the presence of sanitizations.

NoScript's approach suffers from a different set of problems:

- *False positives:* since NoScript sanitizes outgoing requests rather than incoming responses, it cannot confirm whether the offending content actually appears in the response, let alone whether it leads to the execution of JavaScript code. These filters, in effect, have to be stringent enough to handle the worst-case server behavior. Hence NoScript can have a higher rate of false positives, a result confirmed by our experiments.
- *Complex Policies:* NoScript's detection is largely based on regular expressions. Unfortunately, covering all corner cases while avoiding false positives requires very complex detection logic. Manual analysis of NoScript's code revealed more than 40 non-trivial regular expressions involved in detection and sanitization of XSS attacks. Clearly, these can be very cumbersome to maintain.
- *Usability Impact:* NoScript's false positives normally cause more disruption to users compared to XSSAuditor. Since NoScript sanitizes the outgoing request URL, the browser may end up making a request with very different set of parameter values. These new parameter values can cause the request to fail on the server side, corrupt data that is stored on the server-side, or return an incorrect page. In contrast, XSSAuditor only prevents the offending script from executing, and this can impact dynamically constructed elements such as advertisements, comment frames, etc, but will usually not prevent the main content of the page from being displayed.

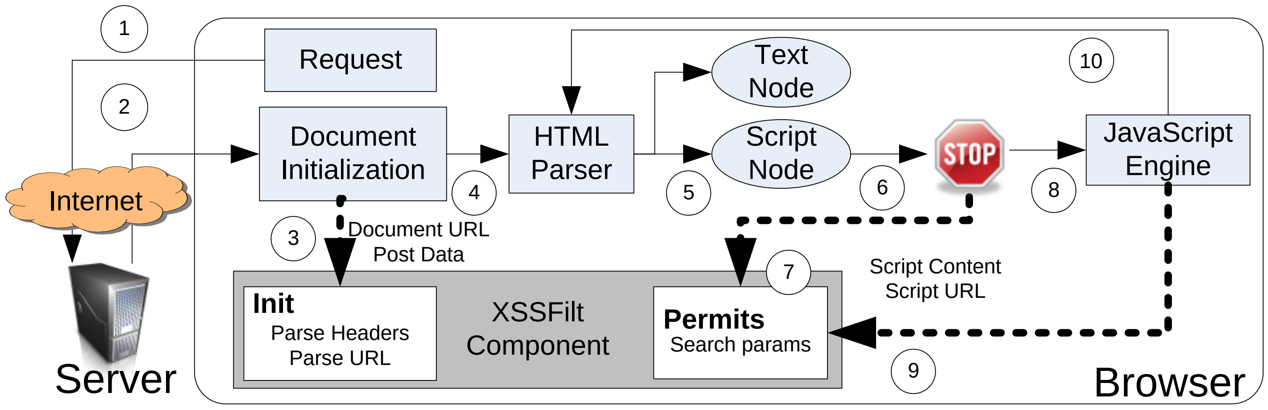

Figure 2: XSSFilt architecture

1.2 Overview of XSSFilt and Contributions

We present XSSFilt, a new client-side XSS defense that addresses the above-mentioned drawbacks of previous filters. In particular, this paper makes the following contributions:

- In Section 3, we present the architecture of XSSFilt, a browser-resident XSS defense. Unlike previous browser-resident defenses that all relied on exact string matching, XSSFilt uses approximate string matching. (See Section 4.) This enables our defense to cope with web applications that perform application-specfic sanitizations.
- In Section 5, we present a set of policies to detect XSS attacks. These policies detect attacks involving injection of whole scripts, or those occurring due to injection of parameters within scripts (partial injections).
- We present a discussion of the full range of attacks possible on XSSFilt, and ensure that our design can successfully defend against these attacks (see Section 6).
- In Section 7, we present a comparative study of the protection and usability of XSSFilt (implemented for the latest development version of Firefox), XSSAuditor (readapted for Firefox) and NoScript (XSS filter only). In particular, our evaluation shows:
  - the importance of addressing partial script injections, which accounted for 8% of the 400 vulnerabilities we studied from `xssed.com`, and 18% of the 10K vulnerabilities we discovered on the web using an XSS vulnerability discovery tool that we built.
  - that scripts generated dynamically from input parameters, the necessary condition for partial script injections, are commonplace. Our results suggest that more than 9% of web pages contain dynamic scripts.
  - the benefits of using an approximate string matching algorithm over the exact matching algorithm employed by XSSAuditor: our false negatives were decreased five-fold due to approximate matching.
  - that false positives generated by XSSAuditor and XSSFilt are more likely to be symptoms of underlying injection vulnerabilities rather than mere annoyances: 85% of the false positives reported by XSSFilt were in fact caused by an underlying XSS vulnerability. On the other hand, all NoScript false positives were benign strings that matched its regular expressions.

2. BACKGROUND ON XSS ATTACKS

This section provides a short overview of XSS attacks, and may be skipped by readers that are very familiar with them. An XSS attack involves three entities: a *web-site* that has an XSS vulnerability, a legitimate *user* of this web-site, and the *attacker*. The attacker's goal is to be able to perform sensitive operations on the web-site using the credentials of the legitimate user.

Although an attacker is able to run her code on the user's browser, the same-origin policy (SOP) of the browser prevents her code from stealing the user's credentials, or observing any data exchanged between the user and the web-site. To overcome this restriction, the attacker needs to inject her code into a page returned by the web-site to the user. An XSS vulnerability in the web-site allows this to happen.

Exploiting an XSS vulnerability involves three steps. First, the attacker uses some means to deliver her malicious payload to the vulnerable web-site. Second, this payload is used by the web site during the course of generating a web page (henceforth called a *victim page*) sent to the user's browser. The left side of Figure 1 shows an example of a vulnerable page that uses a user-supplied parameter `id` to construct the `href` parameter via `document.write`, while the right side shows an example payload. In this case, the payload does not need to open a new script tag because it is already contained in one; rather, it closes the string where the parameter is supposed to be confined and writes additional JavaScript code.

If the web site is not XSS-vulnerable, it would either discard the malicious payload, or at least ensure that it does not contribute to code content in its output. However, if the site is vulnerable, then, in the third step, the user's browser would end up executing attacker-injected code in the page returned by the web site.

In a *reflected XSS attack,* an attacker lures a user to the attacker's web page, or to click on a link in an email. At this point, the user's browser launches a GET (or, in some cases, POST) request with attacker-chosen parameter values. When a vulnerable web site uses these parameters in the construction of its response (e.g., it echoes these parameters into the response page without adequate sanitization), the attacker's code is able to execute on this response page.

3. XSSFilt OVERVIEW

To illustrate XSSFilt's design, consider the sequence of operations that take place from the time the user's browser submits a request to a web server to the time the web page loads. The steps in this sequence are identified using numbers in Figure 2, and we describe them in more detail below.

In Step 1, the browser submits a request to a web site. This submission may be in response to a user clicking on a hyperlink in a web page or email, or may be the result of execution of scripts on a page that is currently being displayed. This submission may use a GET or POST request, and will include parameter data that is under the control of the web page or email containing the link.

In Step 2, the web site returns a response to the browser's request. This leads to Step 3, when a new document is created by the browser. In this step, the browser invokes the `Init` method of XSSFilt, providing information about the request submitted in Step 1. XSSFilt parses the URL and POST data for parameters and converts them into a list of (name, value) pairs, which enables more accurate techniques for inferring reflected content as compared to XSSAuditor. The filter then returns control to the browser so as to start rendering the page.

In Step 4, the web browser's internal HTML parser is used to parse the document received in Step 2. This causes the creation of various nodes in the document tree, including script and text nodes in Step 5. In Step 6, a script node would normally be sent to the JavaScript engine, but the browser intercepts the script and sends it to the `Permits` method of XSSFilt. At Step 7, XSSFilt uses an approximate substring matching algorithm to search for one or more of the GET/POST parameters inside the script. Any matching content is deemed *reflected* or *tainted*. Further details on this detection technique can be found in Section 4. If the tainted components of a script violate the policies described in Section 5 then the execution of the script is blocked. Otherwise, it is handed over to the JavaScript engine in Step 8.

Note that new script content may be created during script execution, e.g., due to the execution of the `eval` operation. Our architecture ensures that such newly created scripts are passed to the `permit` operation of XSSFilt in Step 9, thus ensuring that these dynamically created scripts are checked for XSS in the same manner as the scripts included statically.

It is also possible for script execution to result in the creation of new HTML content, e.g., as a result of `document.write` or setting `innerHTML` attribute of some DOM nodes. In all these cases, the HTML parser will be invoked in Step 10, and Steps 5 through 8 will be repeated. This ensures that DOM-based XSS attacks are handled the same way as XSS attacks contained within static content received in Step 2.

4. IDENTIFYING REFLECTED CONTENT

Detection of reflected content is a *taint analysis* problem, where the response page components that are directly derived from request data are to be considered tainted. XSS defenses implemented on the server side can rely on taint-tracking instrumentation on the server code for accurate detection of taint, but this is obviously not a choice for a browser-resident defense. Thus, the only option is to infer possible taint by comparing the input to the server and its output. A known limitation of such an approach is that if data goes through complex transformations, then there would be no match between the input and output and hence no taint can be inferred. Fortunately, the transformations used by most web applications seem to consist of character encodings and simple sanitizations, and these can be accommodated in a taint inference algorithm.

The core of our taint inference algorithm is the same as that of Reference [21]. However, since XSSFilt is resident on a browser, there are differences in terms of identifying taint sources, recognizing tainted content, and a few additional optimizations.

1. The URL is parsed into a list of (name, value) parameters. The parameter name is used for reporting purposes, but is of no other interest to XSSFilt. This decomposition into parameters is necessary to detect partial script injections. If the URL cannot be parsed properly, or if special characters are present in the URL path (or if they span more than one parameter), the entire path is also appended as a single parameter. This step ensures that the technique would not fail for applications that use non-standard parameter encoding, but instead will operate in a degraded mode where it can detect whole-script injection.

2. As an optimization, parameters whose content cannot possibly include JavaScript or HTML code are omitted from further consideration. Specifically, we discard parameters shorter than 8 characters, and parameters containing only alphanumeric characters, underscores, spaces, dots and dashes. These characters are commonly used in benign URLs. However, even if these parameter values are included in the returned page, the resulting content will not match the policies described in Section 5, and hence ignoring them will not cause attacks to be missed.

3. Before any inline script is executed, an approximate substring matching algorithm is used to establish a relationship between the parameters and the script. If the parameter is longer than the script, then the script is searched within the parameter, to detect *whole script injection*. On the other hand, if the script is longer than the parameter, then the parameter is searched within the script, to detect *partial script injection*.

   A similar check is performed before an external script is fetched for execution. If the script URL is longer than the parameter, then the parameter is searched within the URL to detect hijacking of existing external scripts, where the attacker is able to point them to a malicious domain. Otherwise the URL is searched within the parameter to account for *whole script injection* of an external script name.

Previous browser-resident techniques for XSS detection, including XSSAuditor [3] and noXSS [10] use *exact* substring matching rather than an *approximate substring matching* to identify reflected content. Another difference is that XSSAuditor does not parse parameters and hence it can only detect those cases where an entire script is injected, while XSSFilt can detect partial script injections as well. The main advantages of XSSAuditor's approach are:

- *Faster runtime performance:* exact substring matching has linear-time complexity, and hence can provide better performance over the quadratic-time worst-case complexity of approximate matching.

- *Lower false positive rate:* This is because (a) exact matching is stricter than approximate matching, and (b) likelihood of coincidental matches for the entire script is smaller than that for any of its substrings.

Our approach, on the other hand, has complementary strengths:

- *Coping with application-specific sanitizations:* Approximate string matching is better able to cope with application-specific sanitizations that may take place, e.g., when a '*' character is replaced by a space. In contrast, an exact matching algorithm will fail to match even if a single such substitution takes place. Our results in Section 7, as well the results of References [1] and [21] show that such application-specific sanitizations do occur in practice.
- *Partial script injections:* As described in the introduction (Figure 1), template-based web application frameworks create natural opportunities where an existing script could be modified by injecting a parameter value into its middle. In this case, there would not be a match for the whole script, and hence XSSAuditor would miss such injections. As we show in the evaluation section, such partial script injection vulnerabilities are relatively common.

Although the results in Reference [21] seem to indicate that the above benefits could be obtained without undue performance overheads or false positives, a more careful examination indicates that those results are not necessarily applicable for client-side XSS defense:

- The false positive evaluation in Reference [21] was done in the context of SQL injection, specifically on simple web applications. In contrast, a browser-side XSS defense needs to avoid false positives on virtually all applications that have been deployed on the web.
- In terms of performance as well, the results in Reference [21] were obtained using a SQL injection data set. The volume of data subjected to approximate matching and policy checking are thus much smaller than that involved in HTTP requests and responses, and hence performance constraints are more stringent for XSS-defense within a browser.

Thus, it was unknown, prior to this work, whether a client-side XSS defense can benefit from the strengths of approximate matching without incurring its drawbacks. Our evaluation answers this question affirmatively. Section 7 shows that XSSFilt benefits from the increased power of approximate matching, while minimizing its drawbacks.

5. XSS POLICIES

XSSFilt is targeted at inexperienced users who are not expected to deal with false positives. We seek to provide a filter reliable enough to be enabled by default on a mainstream browser.

Previous research on injection attacks on web applications showed that a few generic policies can detect a wide range of attacks. In particular, Su et al [24] proposed the *syntactic confinement policy* that confines tainted data to be entirely within certain types of nodes of a parse tree for the target language (e.g., SQL or JavaScript). A lexical confinement policy has been used successfully by others [21]. However, these works primarily targeted SQL injection, which is relatively simple. In contrast, XSS is more challenging due to the diversity of injection vectors and the many evasion techniques available to attackers. Below, we describe policies that address these difficulties in a systematic manner.

5.1 Inline Policy

The inline policy is used for protecting against XSS attacks embedded in inline content. Specifically, the following types of content are addressed.

A. Inline code: This category includes code embedded directly in the web page using one of the following mechanisms:

  i. *Inline scripts:* Script content, enclosed between `<script>` and `</script>` tags

  ii. *Event listeners:* Code enclosed in an event handler specification, e.g., `<a onclick="alert(...)">`

  iii. *JavaScript URLs:* Code provided using JavaScript protocol, e.g., `<a href="javascript:alert(...)">`

  iv. *Data URL:* These provide a general mechanism to include inline data, which may be text, images, HTML documents, etc. The following data URL embeds the text "Hello" in base64 encoding: `data:text/plain,SGVsbG8=`

B. Dynamically created code: New code may come into being when a value stored in a variable is `eval`'d or used in an operation such as *setTimeout*.

The simplest (and most restrictive) inline policy is one that prohibits any part of a script from being tainted. Unfortunately, this policy produces many false positives because it is common for scripts to contain data from HTTP parameters. For this reason, we implemented a lexical confinement policy that restricts tainted data to be contained entirely within a limited set of tokens. In practice, we discovered that the data injected is normally inside strings. All of the attacks in our dataset consisted of such injections within strings. We therefore specialized the policy to ensure that tainted data appears only within string literals, and does not extend before or after the literal. This policy was sufficient to avoid false positives.

5.2 External Policy

This policy is enforced on external code that is specified by a name, i.e., injection vector (C) described below.

C. External code: Code that is referenced by its name using one of the following mechanisms:

  i. *External scripts:* Script name provided using a script tag, e.g., `<script src="xyz.js"></script>`

  ii. *Base tags:* These can be used to achieve an effect similar to external script injection by implicitly changing the URL from where scripts in the document are loaded.

  iii. *Objects:* Similar to external scripts, but embedded between `<object>` and `</object>` tags.

External policy addresses these injection vectors. It is applied to the name of external scripts or objects as follows.

1. If the host portion of the URL is untainted, then the script is allowed. Note that an attacker cannot typically upload a malicious script onto a server controlled or trusted by a web application. For this reason, the

attacker needs to control the host portion of the URL. Unlike the host component, our policy permits the path component of a URL to be tainted, since some web applications may derive script names from parameter values.

2. Even if the host portion of the URL is tainted, our policy permits the script if it is from the same origin. We use a relaxed same-origin check [5] which verifies if the registered domains of the URLs match. Thus, `www.google.com` is considered same-origin with `reader.google.com`.
3. Finally, if the tainted domain was previously involved in a check that was deemed safe, then it is allowed. Intuitively, XSSFilt assigns trust on a per-domain basis, and considers all requests from the same domain as trusted or untrusted.

6. SECURITY ANALYSIS

In this section we identify possible attack vectors and strategies that may be used for an XSS attack, and argue how our design addresses these threats.

We expect an attacker to deploy the full range of techniques available to evade detection by XSSFilt. There are two logical steps involved in the operation of XSSFilt: (I) recognizing script content in the victim page, (II) identifying if this code is derived from request data. The following techniques may be used to defeat Step I:

1. *Exploit all of the above-mentioned vectors to inject code,* hoping that one of more of the vectors may not be (correctly) handled by XSSFilt. However, the filter architecture makes it very simple to enforce complete mediation [20], since there is a very small number of code paths that call into the JavaScript engine. IE and NoScript rely on the completeness of their regular expressions, which need to strike a complicated balance between usability and protection.
2. *Exploit various browser parsing quirks* to prevent XSSFilt from recognizing one or more of the scripts in the victim page. Note, however, that browser quirks pose a problem for techniques that attempt to detect scripts by statically parsing HTML. In contrast, XSSFilt operates by intercepting scripts dispatched to the Javascript engine, and hence does not suffer from this problem.
3. *Exploit DOM-based attacks:* If the victim page uses a script to dynamically construct the page, e.g., by setting the `innerHTML` attribute, then try to defer script injection until this time. This technique defeats filters that scan for scripts at the point the response page is received. However, XSSFilt's architecture ensures that all scripts, regardless of the time of their creation, are checked before their execution. Hence, XSSFilt is not fooled by DOM-based attacks.
4. *Exploit sanitization to modify the parse tree* and force the parser to interpret another part of the page as script. This vulnerability existed in Internet Explorer [17], whose filter deactivated script nodes by modifying the `<script>` tag, and can potentially exist in NoScript. In contrast, XSSFilt's decision to block the execution of a script has no effect on how the rest of the page is parsed.

To defeat Step II, an attacker may use the following techniques:

5. *Employ partial rather than whole script injection:* We have already described how our taint inference implementation (Section 4), together with the policies described in Section 5, can detect partial injections.
6. *Employ character encodings* such as UTF-7 to throw off techniques for matching requests and responses. Note that by the time a browser interprets script content, it has already determined the character encoding to be used. Therefore XSSFilt applies the corresponding decoding operation before the taint inference step, and thus thwarts this evasion technique. NoScript might get confused because the encoding of the response is not known at request time.
7. *Exploit custom sanitizations performed by an application* to evade taint inference. As described before, XSSFilt uses approximate substring matching, which provides a degree of resilience against application-specific sanitizations employed by web applications. However, if a web application makes extensive use of non-standard character transformations, it may be possible to exploit them to evade XSSFilt. It seems unlikely that any purely client-side defense can address this evasion. Moreover, note that the attacker cannot induce such behavior on arbitrary applications — he can only exploit applications that already perform extensive, non-standard transformations.
8. *Employ second order attacks* that operate by injecting malicious parameters into links or forms contained in the victim web page. An XSS attack would be effected when these forms are subsequently submitted. Note, however, that XSSFilt will apply policies to these submissions as well, and hence detect second (or higher order) order attacks. IE has an exception for same-origin links and would be vulnerable to this attack.

7. EVALUATION AND COMPARISON

In this seciton, we evaluate and compare the protection and compatibility offered by XSSFilt, XSSAuditor and NoScript. Our results demonstrate that the techniques used in XSSFilt provide better compatibility as well as protection. In addition, we provide data that shows the prevalence of partial injection attacks.

7.1 Implementation

We implemented XSSFilt by modifying the Content Security Policies (CSPs) [23] implementation in Firefox. This allowed us to leverage the CSP interface and code, which include most of the required interposition callbacks.

CSPs implement the `nsIContentSecurityPolicy` interface, which is only used to check the URL of external resources being loaded, including scripts. We added a new method `permits` to check inlined resources for XSS injections.

We modified the existing CSP callbacks to pass the script content to `permits` where appropriate. We also added new callbacks for Base elements and Data URLs, which CSPs do not need to address for technical reasons.

To test NoScript, we downloaded the latest version (2.2.3) and disabled all features except for XSS protection, which we turned on for all requests.

To test XSSAuditor, we reimplemented it in Firefox to avoid instrumenting a second browser. XSSFilt is based on XSSAuditor's architecture, so reimplementing its policies is rather simple.

Dataset	XSSFilt	XSSAuditor	NoScript
xssed	399/400	379/400	400/400
cheatsheet	20/20	18/20	20/20

Figure 3: Results for xssed and cheatsheet dataset

Dataset	Partial Script Injection	String Transformation
xssed	16	5
cheatsheet	2	0

Figure 4: XSSAuditor failures

7.2 Protection Evaluation

We tested the three filters against two sources of XSS attack data that have been widely used in previous research.

xssed: `xssed.com` [6] contains reports of websites vulnerable to XSS, along with a URL for a sample attack. Since the dataset is very large, we randomly selected a subset of 400 recent, working attacks among these in order to estimate the effectiveness of our filter against real-world attacks.

XSS cheatsheet: The xssed dataset is biased towards very simple attack payloads, since most of them simply inject a script tag. To assess the filter's protection for more complex attacks, we created a web page with multiple XSS vulnerabilities and tried attack vectors from the XSS Cheat Sheet [8], a well-known and oft-cited source for XSS filter circumvention techniques.

To automatically test this large set of attacks, we modified Firefox to log XSS violations to a file and used its extension API to automatically navigate the browser to all URLs in the two datasets.

Figure 3 summarizes the results for both datasets. XSSFilt successfully stopped all but one of the attacks from the xssed dataset and all attacks from the cheatsheet dataset. Its lone failure is attributed to a limitation of taint-inference previously explained: when the web application applies extensive string transformations, the algorithm might fail to find a relationship between the parameter and the content. In this specific example, the filter failed because the parameter:

```
alert("HaCkEd By N2n-HaCkEr    -   3rd@live");
```

was transformed to

```
alert("HaCkEd N2n-HaCkEr 3rd@live");
```

by the web application. Some (but not all) spaces and dashes had been deleted, along with the word "By". In these situations, no client-side filter can realistically be expectd to detect the attack.

The 100% coverage on the cheatsheet dataset is not surprising: these attacks are designed to bypass server-side sanitization functions, which look for specific patterns in text and are vulnerable to browser quirks and unusual XSS vectors. Since this filter architecture is immune to browser quirks and covers all vectors uniformly, none of these attacks succeeded.

XSSAuditor missed several attacks in these datasets. Figure 4 identifies the underlying causes:

Partial Script Injection: XSSAuditor does not detect this type of attack because, unlike XSSFilt, it does not perform URL parsing and substring matching.

String Transformation: XSSAuditor relies on canonicalization to account for common string transformations in web applications. This approach can break when an uncommon transformation takes place. Taint-inference relies on approximate substring matching, which is more tolerant of exceptions.

The results report that the protection offered by XSSAuditor over the xssed dataset is 95% vs 99.75% for XSSFilt. However, the xssed dataset is biased towards simple vulnerabilities that inject a `script` tag, and a 4% prevalence of partial script injections does not necessarily imply that this is the case for XSS vulnerabilities in general.

NoScript's XSS filter did well against both datasets. The reason is that, unlike XSSAuditor and XSSFilt, NoScript specifically looks for JavaScript syntax and common JavaScript functions such as `alert`. Since most attacks on xssed simply attempt to open a script tag and popup an `alert` box, it is clear that NoScript should have no trouble in detecting them. Even though considerable skills and efforts may be required to transform the payload until it bypasses the filter, it has been shown to be possible [22, 16]. In our own experiments, we were able to bypass the filter in some cases by substituting the `alert` call with another JavaScript identifier. Had the web application bound that identifier to a suitable function, we could have carried out a successful XSS attack.

7.3 Partial Injection Prevalence

Compared to XSSAuditor, XSSFilt is able to detect partial script injection vulnerabilities. Therefore, it is important to assess how prevalent these are. To estimate their prevalence, we used three different methods.

7.3.1 *Partial Injections in* `xssed.com` *Data Set*

Out of 400 real-world live XSS attacks, 4% were targeting partial injection vulnerabilities. We analyzed the rest of the vulnerable pages attacked through whole script injections to discover if they contained partial injection vulnerabilities as well, and we discovered that an additional 4% of pages are vulnerable, for a total of 8% of pages vulnerable to partial script injection.

Thus, even though the coverage against *attacks* on the xssed dataset for XSSAuditor was 95%, the actual coverage on *vulnerabilities* is lower at 91%. However, the size of this dataset is quite limited for the purpose of extrapolating statitics about the nature of XSS attacks in general. Moreover, the website does not review submissions and does not reward contributors for creative or complex attacks. For this reason, the dataset is biased towards simple vulnerabilities that can be discovered automatically.

7.3.2 *Partial Injection Vulnerabilities in the Wild*

We have developed a tool/scanner called gD0rk [18] to study the prevalence of XSS vulnerabilities in deployed sites. Although gD0rk was not developed for the purpose of this paper, we believe that it is very helpful for assessing the prevalence of partial injection vulnerabilities:

- gD0rk analyzed a much larger collection of web sites as compared to `xssed.com` data, and hence provides a broader basis for drawing inferences about vulnerabilities in deployed sites.
- gD0rk uses a mechanical procedure for finding vulnerabilities, with no built-in bias for partial injections.

We note that, due to the nature of reflected XSS attacks, we are targeting ourselves in these attacks, and hence believe that the web sites scanned by gD0rk were in no way subjected to any harm.

gD0rk uses Google's advanced search capabilities to short list candidate web sites that are likely to be vulnerable, probes them for reflected content by modifying the URL, and examines the context in which the content is injected in the web pages returned to build an attack. The exact details of the tool are not important for the purposes of this paper, but we do want to note that it is sophisticated enough to detect and circumvent many sanitizations performed by web applications where possible. For example, if a reflection is inside a JavaScript string in a `script` tag, the scanner attempts to write

```
"; payload(); //
```

through the filter to exploit the partial injection vulnerability. However, if the application sanitizes double quotes, the scanner attempts to close the script tag instead and open a new script node with

```
</script><script>payload();</script>
```

We describe an example from this dataset that contains a partial injection vulnerability. The server-side code for this page can be abstracted as:

```
<script><!--
select1 =
    "<?=sanitize(\$_GET["select1"])?>";
...
--></script>
```

The scanner detects that the sanitization function transforms %22 into double quotes, and derives the following string for an XSS attack:

```
\%22; alert(1); //
```

We ran gD0rk for one month and identified 272,051 vulnerable websites. For scalability and performance reasons, we did not validate the generated attacks for all these vulnerabilities. Instead, we used statistical sampling to estimate the fraction of these sites that were actually vulnerable. In particular, a random subset of 1000 vulnerabilities among these were selected, and then we were able to verify that 98% of the generated attacks worked on this subset. We then selected a random subset of 10000 vulnerable websites and used the scanner to identify the context of the vulnerable reflections. We found that 18% of these reflections were included within `script` tags or event handlers, and thus represent partial script injection vulnerabilities.

7.3.3 Dynamically Generated Scripts

Intuitively, the necessary requirement for a partial injection vulnerability is a script that is assembled dynamically from input parameters by the web application. We believe that it is reasonable to expect developers to fail to sanitize parameters which appear inside scripts just as often as they fail to sanitize them anywhere else in the page. Under this hyphothesis, the rate of pages that construct scripts dynamically is a good estimator for the ratio of partial injection vulnerabilities to whole script injection vulnerabilities. The benefit of this indirect approach over the previous one is that the dataset is not made out of vulnerable pages, which represents a skewed sample from mostly unpopular websites.

For this reason, we built a browser-resident crawler for Firefox and bootstrapped it with the 1000 most popular websites according to the Alexa rankings. When the crawler processes a page with non-trivial HTTP parameters and detects that a parameter appears in a script, it substitutes the parameter value with a placeholder, requests the page with the newly constructed URL and then attempts to find the original value and the new placeholder in the response. If the placeholder is found in the same script and the original parameter value is not found, then the relationship between script and parameter is confirmed and the page is marked as containing a dynamic script. When we stopped the crawler, it had crawled a total of 35145 pages, of which 9% contained dynamically generated scripts. Given the strictness of the requirements, we believe this is a conservative estimate.

7.4 Compatibility Evaluation

Browser-resident reflected XSS defenses restrict the capabilities of browsers with respect to content found in input parameters, such as GET parameters from the querystring. As a result, they have the potential to break some web pages, and thus lead to compatibility problems. To estimate the compatibility of these defenses, we instrumented Firefox to log information about XSS checks while performing the aforementioned crawl on major websites. The browser-resident crawler was developed using the new Addon-SDK [14] for Firefox. This allowed us not only to support discovery of dynamically constructed links and forms, but also to check all the resources loaded by the web page (including scripts and advertisements inserted through DOM manipulation) for XSS violations.

Overall, all filters reported very few false positives. This is due to the benign nature of the dataset, which contains very few special characters to begin with: only 26 URLs contained any of the characters in the following set: {",',<}. However, not all filters performed equally:

- NoScript's 15 false positives are complex URLs containing long identifiers that erroneously match NoScript regular expressions. For example: the following (simplified) URL triggers a violation:

  ```
  http://domain.com/dir/page.php?
   n=PHNjcmlwdP25plJmo9MCI%2BPC9zY3JpcHQ%2B&
   h=55e1652a183.
  ```

  An interesting property of NoScript's XSS filter is that if a parameter triggers a false positive in one web application, it will do so in *every other* web application, because the actual HTTP response does not matter. For this reason, when we devised the heuristics to fill and submit forms, we chose not to submit suspicious strings that would trigger XSS violations on every web application. Had we configured the crawler to fill forms with values such as `alert(1)`, NoScript would have triggered many more policy violations.

- XSSFilt initially reported a much higher number of false positives than NoScript. Most of them were either due to a URL being supplied as a parameter and then used by an existing script to construct a new script tag (for advertisements), or by a parameter being passed to a string-to-code function such as `eval`. These practices would be safe if the application code checked the value against a whitelist of pre-approved URLs for the former case or JavaScript snippets for the latter, and the violation could be indeed considered spurious. However, we found that out of 51 XSSFilt notifications, only 8 did

Filter	XSSFilt	XSSAuditor	NoScript
# of violations	8	6	15

Figure 5: Compatibility Comparison

such checks; the remaining violations were in fact due to vulnerable pages that could be subverted to load a script from an arbitrary host or execute arbitrary code. This set of pages include important websites such as `wsj.com`, `weather.com` and `tripadvisor.com`. For this reason, we do not consider these scenarios as false positives.

- XSSAuditor behaved similarly to XSSFilt, reporting the same vulnerable pages as XSS violations. We disconted them from XSSAuditor results as well. A couple of actual false positives produced by XSSFilt involving partial script injection were not notified by XSSAuditor.

Figure 5 reports the number of false positives for each filter. Note that while XSSAuditor and XSSFilt offer roughly the same compatibility, the latter protects against a wider range of threats. Also, as previously explained, NoScript's false positives tend to cause greater disruption for the user.

7.5 Performance Evaluation

The performance evaluation is focused on XSSFilt only. XSSAuditor's performance has already been evaluated in Reference [3], while NoScript overhead is trivially low, since it only examines one URL per request, unlike XSSFilt and XSSAuditor which have to perform checks for each script contained in the page.

Unfortunately, calculating the overhead imposed by the filter is a significant challenge. This is because XSSFilt contains many optimizations that bypass policy enforcement if parameters do not contain special characters, if they are too short or if an external script is fetched on the same origin of the page.

The Mozilla framework includes two performance tests: tp4 is an automated test that can be run on patches to the mozilla codebase, to estimate the overhead that these can impose on the browser. The test estimates the time required to load a set of predefined pages that are saved locally to produce consistent results over time. The overhead estimated by tp4 for XSSFilt is negligle. Unfortunately, since tp4 fetches homepages saved locally, it overestimates the effect of the filter's optimizations: requests don't have parameters to check, and all external scripts are from the same origin.

To produce more meaningful results, we used pageloader, the Firefox extension that is used internall by tp4 to measure load times. Instead of using the standard set of local pages used by tp4, we loaded 20 times a custom set of 120 URLs to be fetched remotely that also included many GET parameters[2]. We used an aggressive caching proxy to factor out network delay while transparently providing pageloader with remote resources. This way, we can avoid overestimating the speedup due to XSSFilt's optimizations. This test showed an overhead of 2.5%.

However, even though the dataset clearly triggered many XSSFilt checks, the figure is not necessarily representative of the overhead normally experienced by users, because this is a) heavily dependent on the amount of parameters in web applications and b) ultimately diluted when factoring in the delay involved with fetching a webpage off the network. For this reason, we used profiling data available from an ordinary user session consisting of 3000 unique pages. Since a web browser is a multithreaded execution environment, the overhead cannot be estimated by simply timing the calls to XSSFilt: the same call will take longer if the user is simultaneously watching a video on YouTube on a different tab. Therefore, the profiler logged the actual parameters of XSS checks, and given that the only expensive operation is approximate substring matching, we can perform the approximate string matching computations offline to estimate the time spent during XSS checks for each page load. This yields an average overhead of 0.5%, which shows that the overhead is almost negligible when factoring in network latency.

[2]The set of URLs can be found at `http://pastebin.com/kYqas9ae`.

8. RELATED WORK

Server-Side Approaches: Earlier works in XSS defense have been mostly in the form of server-side defenses. Given that the vulnerability exists solely on the server-side, this is natural. Moreover, techniques based on taint-tracking can be naturally applied on the server-side.

XSSDS [12] describes two server-side approaches: a reflected XSS filter based on string matching and a generic XSS filter that builds a whitelist of scripts during a training phase. Similar to IE8, the reflected filter compares input parameters and HTML output to look for untrusted input in scripts. However, it is a network-based filter, unable to detect DOM-based attacks. It leverages the Firefox parser, which is able to defeat browser quirks against Firefox clients; however, this parser comes with a higher overhead and cannot reliably handle quirks from other clients.

[29] uses string matching and taint-aware policies to stop generic injection attacks (SQL injection, XSS, Shell Injection). However, it uses heavyweight, precise taint-tracking. This limits its applicability to the server side, and it has the drawbacks associated with taint-tracking: high overhead and possible loss of reliability. Their policies are based on *syntactic confinement*: tainted tokens of sensitive operations should not span multiple syntactical constructs.

[21] is a server-side defense against injection attacks (SQL injection, XSS, Shell Injection). It offers server-side protection through library interposition, using taint-inference and taint-aware policies. An exclusively server-side protection can only be a static, network-based defense. Therefore, it cannot protect against DOM-based attacks. It also suffers from false positives and negatives as a result of parsing differences with a browser. Moreover, sanitization of the detected attacks is not discussed.

XSS-GUARD [4] uses the program transformation approach also found in Candid [2] to detect which scripts are intended by the web application, inferring a whitelist for each request before sending out the response to the client: the application is instrumented to build an alternative response along with the ordinary one; instead of using HTTP parameters to assemble the response, the alternative application logic uses dummy inputs. Once both responses have been built, XSS-GUARD checks that every script present in the real page is also present in the alternative page. Although XSS-GUARD is a server-side defense, the idea of sending a dummy request along with the original request for XSS protection has been used on the client-side as well by [9].

Blueprint [13] is a server side defense which converts the untrusted HTML embedded in a page into JavaScript code. The purpose of this transformation is to fix the browser's interpretation of the page at the server-side, adding JavaScript code to reliably reconstruct the parse tree once the page is rendered by the browser. This sidesteps one of the biggest issues with XSS, which is browsers' leniency towards errors, which opens many chances to overcome input sanitization. With this technique, the interpretation of untrusted input can be determined by the server, which also decides what type of content is allowed for each piece of untrusted information.

Client-Side Approaches: Client-side approaches protect users against XSS vulnerabilities without waiting for websites to fix them. Two major browsers now ship with an XSS filter.

Internet Explorer 8 [19] comes with built-in XSS protection. IE8's approach also uses the idea of matching inputs and outputs, but does so in a more simplistic way: from inputs, regular expressions of possibly malicious injections are created using heuristics, compiled and matched against the HTML output. IE8's goal is to provide a usable protection for ordinary users, thwarting basic attacks (which constitute the vast majority of attacks in the wild) without incurring false positives. Their regular expressions have been shown to be insufficient for protection: [16] shows that in case of an XSS vulnerability there are many ways to bypass the filter. Moreover, their choice of sanitization technique opens up further holes, which have been described in [17]. Finally, in spite of its implementation within the browser, it operates like a network-based filter, and hence does not detect DOM-based attacks.

XSSAuditor [3] is the name of the XSS filter integrated into Webkit (and consequently Google Chrome). This filter proposes a new architecture, which is only possible on browser defenses: instead of interposing on the network data, this solution interposes on the JavaScript engine interface. This approach has many advantages, the most important being interposing on *all* requests of script evaluation, defeating browser quirks and unusual attack vectors. XSSFilt too enjoys this advantage. However, unlike XSSFilt, XSSAuditor relies on exact string matching, and hence can miss attacks due to application-specific sanitizations. Moreover, XSSAuditor does not detect partial script injections.

NoScript [7] is a popular Firefox plugin which allows users to execute JavaScript only on trusted websites manually added to a whitelist. NoScript also includes an XSS filter: like IE8, it relies on regular expressions on parameters. However, regular expressions are used to extract and identify malicious data from the URL; NoScript does not actually check if malicious data is actually present in the response, but rather sanitizes the request before it is sent to the server. Thus, it can suffer from a higher rate of false positives.

Hybrid Approaches: Hybrid approaches are an interesting alternative: they can enforce the policy on the browser, defeating browser quirks, and rely on the server to provide a policy or taint information. However, while the two previous classes of defenses can be practically deployed, these solutions require a critical mass of browsers and websites willing to deploy the defense.

BEEP [11] is a hybrid defense framework which allows the server to supply a policy for the page through a JavaScript function. This function can interpose on script execution: it receives the script content and its DOM node as arguments, and can deny the script execution. Using the two arguments provided to the hook, BEEP also provides two sample policies. A whitelisting policies, where the web developers checks every script with a set of known script hashes, and a containment policy, where nodes can be prevented from having script content in descendants.

DSI [15] and Noncespaces [27] protect against injection attacks by providing an isolation primitive for HTTP. Using this primitive, the server can securely isolate untrusted content and trasmit it to the browser along with the HTML response. The browser can then refuse to execute untrusted content. This combines the advantages of server-side defenses with respect to identification of untrusted content (support for taint-tracking and developer annotation) and of client-side defenses with respect to enforcement (immunity to browser quirks, support for DOM-Based attacks).

Mozilla CSPs [23] is a feature added in Firefox 4.0 to support server-supplied content restrictions, to further limit the resources that can be embedded in a web page. For each content type, the web developer can specify a list of trusted hosts allowed to provide content for the web page. These policies can provide XSS protection by allowing scripts to be served solely by servers under the control of the web application. Unfortunately, inlined content (such as inline scripts) cannot be considered same-origin: it has to be considered untrusted and cannot be executed.

Reference [28] presents a modification to Firefox's JavaScript engine that prevents data leaks using fine grained taint-tracking, refusing to transfer sensitive information (e.g. cookies) to third parties.

9. CONCLUSIONS

This paper presented a thorough study of two popular XSS filters, NoScript and XSSAuditor, identifying their weaknesses and proposing a new filter, XSSFilt. Through extensive testing, we showed that XSSFilt covers more attack vectors and is more resilent in case of string transformations applied to reflected content.

10. REFERENCES

[1] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *IEEE S&P*, 2008.

[2] S. Bandhakavi, P. Bisht, P. Madhusudan, and VN Venkatakrishnan. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24. ACM, 2007.

[3] Daniel Bates, Adam Barth, and Collin Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *WWW*, 2010.

[4] Prithvi Bisht and V.N. Venkatakrishnan. XSS-Guard: Precise Dynamic Detection of Cross-Site Scripting Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, 2008.

[5] P. De Ryck, Lieven Desmet, Thomas Heyman, Frank

Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *ESSOS*, 2010.

[6] Kevin Fernandez and DP. XSSed. http://xssed.com/.

[7] Giorgio Maone. NoScript. http://noscript.net/.

[8] Robert Hansen. XSS Cheat Sheet. http://ha.ckers.org/xss.html.

[9] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA04)*, 2004.

[10] Jeremias Reith. NoXSS. https://addons.mozilla.org/en-US/firefox/addon/noxss/.

[11] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*. ACM, 2007.

[12] M. Johns, B. Engelmann, and J. Posegga. XSSDS: Server-side Detection of Cross-site Scripting Attacks. In *ACSAC*, 2008.

[13] M.T. Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE S&P*, 2009.

[14] Mozilla Corporation. Add-On SDK. https://addons.mozilla.org/en-US/developers/docs/sdk/1.2/, 2010.

[15] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[16] Eduardo Vela Nava and David Lindsay. Our favorite xss filters/ids and how to attack them. Black Hat USA 2009.

[17] Eduardo Vela Nava and David Lindsay. Universal xss via ie8's xss filters. Black Hat Europe 2010.

[18] Riccardo Pelizzi, Tung Tran, and Alireza Saberi. Large-Scale, Automated XSS Detection using Google Dorks. http://www.cs.sunysb.edu/~rpelizzi/gdorktr.pdf, 2011.

[19] David Ross. IE 8 XSS Filter Architecture/Implementation. http://blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[20] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *SOSP*. ACM, 1974.

[21] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[22] Sirdackat. Hacking NoScript. http://sirdarckcat.blogspot.com/2008/06/hacking-noscript.html, 2010.

[23] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the web with content security policy. In *WWW*, 2010.

[24] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL*, 2006.

[25] The MITRE Corporation. 2011 CWE/SANS Top 25 Most Dangerous Programming Errors. http://cwe.mitre.org/top25/.

[26] The Open Web Application Security Project (OWASP). OWASP Top Ten Project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[27] M. Van Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*, 2009.

[28] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[29] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, 2006.