Regular Expressions Considered Harmful in Client-Side XSS Filters

Daniel Bates
UC Berkeley
dbates@berkeley.edu

Adam Barth
UC Berkeley
abarth@eecs.berkeley.edu

Collin Jackson
Carnegie Mellon University
collin.jackson@sv.cmu.edu

ABSTRACT

Cross-site scripting flaws have now surpassed buffer overflows as the world's most common publicly-reported security vulnerability. In recent years, browser vendors and researchers have tried to develop client-side filters to mitigate these attacks. We analyze the best existing filters and find them to be either unacceptably slow or easily circumvented. Worse, some of these filters could introduce vulnerabilities into sites that were previously bug-free. We propose a new filter design that achieves both high performance and high precision by blocking scripts after HTML parsing but before execution. Compared to previous approaches, our approach is faster, protects against more vulnerabilities, and is harder for attackers to abuse. We have contributed an implementation of our filter design to the WebKit open source rendering engine, and the filter is now enabled by default in the Google Chrome browser.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized Access*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security*

General Terms

Design, Security

Keywords

cross-site scripting, XSS, filter, web, browser

1. INTRODUCTION

Cross-site scripting (XSS) is recognized as the biggest security problem facing web application developers [22]. In fact, XSS now tops buffer overflows as the most-reported type of security vulnerability [2]. Although each individual XSS vulnerability is easy to fix, much like each individual buffer overflow is easy to fix, fixing every XSS vulnerability in a large web site is a more challenging task, a task that many web sites never fully accomplish. Worse, there are large public repositories of unpatched XSS vulnerabilities (e.g., `xssed.com`) that invite attackers to exploit a wide variety of sites.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

Instead of waiting for every web site to repair its XSS vulnerabilities, browsers can mitigate some classes of XSS vulnerabilities, providing protection for sites that have not yet, or might not ever, patch their vulnerabilities. In principle, such client-side XSS filters are easy to build. In a reflected XSS attack, the same attack code is present in both the HTTP request to the server and the HTTP response from the server. The browser need only recognize the reflected script and block the attack. However, there are a number of challenges to building a filter with zero false negatives, even for a restricted set of vulnerabilities.

In this paper, we analyze the best known client-side XSS filters: the IE8 filter, the noXSS filter, and the NoScript filter. In each case, we find that the filter either is unacceptably slow (e.g., 14% overhead in page load time for noXSS) or is easily circumvented. For example, an attacker can circumvent the IE8 filter by encoding the injected content in the UTF-7 character set, which is not decoded by the filter's regular expressions. Worse, these filters can actually *introduce* vulnerabilities into otherwise vulnerability-free sites.

We argue that the attacks we discover are not simply implementation errors: the attacks are indicative of a design error. Each of the filters we examine analyzes the HTTP response *before* the response is processed by the browser. This design decision lowers the filter's precision because the filter examines the syntax of the response—not its semantics. To increase precision, some filter use a higher fidelity simulation of the browser's HTML parser, reducing performance by, effectively, parsing the response twice.

Instead of examining the pre-parsed response, we propose that client-side XSS filters mediate between the HTML parser and the JavaScript engine, achieving both high performance and high precision. By examining the response *after* parsing, the filter can examine the semantics of the response, as interpreted by the browser, without performing a time-consuming, error-prone simulation. Examining the semantics of the response reduces both false positives and false negatives by preventing the filter's interpretation of the response from getting "out of sync" with the browser's interpretation of the same response. Moreover, such a filter can block XSS attacks safely instead of resorting to "mangling" the injected script by altering the pre-parsed stream.

We demonstrate our approach by implementing the design in WebKit, the open-source rendering engine used by Safari and Google Chrome (see Figure 1). We find that our design is high-performance, incurring no measurable overhead to JavaScript execution or page load time. We estimate the percent of "naturally occurring" vulnerabilities our fil-

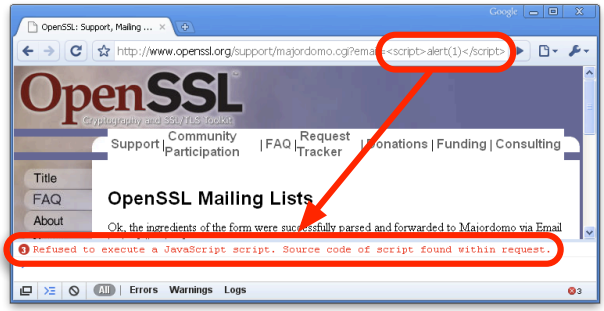

Figure 1: Our filter blocks a reflected XSS attack on openssl.org. Because the site does specify a character set, IE8's XSS filter does not have sufficient fidelity to repair this vulnerability.

ter mitigates by analyzing 145 reflected XSS vulnerabilities from `xssed.com`. We find that 96.5% of the vulnerabilities are "in-scope," meaning our filter is designed to block 100% of the script injection vectors for these vulnerabilities. In practice, we find that our filter has a low false positive rate. Although false negatives from implementation errors are inevitable, our design lets us repair these vulnerabilities without building an ever-more-complex simulator.

Client-side XSS filters are an important second line of defense against XSS attacks. We caution web developers not to rely on client-side XSS filters as the primary defense for vulnerabilities in their applications, but we do recommend that every browser include an XSS filter to help protect its users from unpatched XSS vulnerabilities. Instead of using regular expressions to simulate the HTML parser, client-side XSS filters should integrate with the rendering pipeline and examine the response after it has been parsed. Our implementation of this design has been adopted by WebKit and has been deployed in Google Chrome.

Organization. Section 2 presents a threat model for reasoning about client-side XSS filters. Section 3 demonstrates attacks against previous filters. Section 4 describes the design and implementation of our filter. Section 5 evaluates our design, both in terms of correctness and performance. Finally, Section 6 concludes.

2. THREAT MODEL

Attacker Abilities. Client-side XSS filters are designed to mitigate XSS vulnerabilities in web sites without requiring the web site operator to modify the web site. We assume the attacker has the following abilities:

- The attacker owns and operates a web site.
- The user visits the attacker's web site.
- The target web site lets the attacker inject an arbitrary sequence of bytes into the entity-body of one of its HTTP responses.

Vulnerability Coverage. Ideally, a client-side XSS filter would prevent all attacks against all vulnerabilities. However, implementing such as filter is infeasible. Instead, we focus our attention on a narrower threat model that covers a certain class of vulnerabilities. For example, we consider only *reflected* XSS vulnerabilities, where the byte sequence chosen by the attacker appears in the HTTP request that retrieved the resource.

Instead of attempting to account for every possible transformation the server might apply to the attacker's content before reflecting it in the response, we restrict our attention to mitigating vulnerabilities in which the server performs only one of a limited number of popular transformations. Also, we consider mitigating injections at a *single* location only and do not seek to provide protection for so-called "double injection" vulnerabilities in which the attacker can inject content at multiple locations simultaneously.

Covering *vulnerabilities* is useful because the filter will protect a web site that contains only covered vulnerabilities. However, covering *attacks* is of less utility. If an attacker can evade the filter by constructing a convoluted attack string (e.g., by injecting script via CSS expressions [16] or via obscure parser quirks [8]), then the filter does not actually prevent a sophisticated attacker from attacking the site. Each filter, then, defines a set of vulnerabilities that are *in-scope*, meaning the filter aims to prevent the attacker from exploiting these vulnerabilities to achieve his or her goals.

Attacker Goals. We assume the attacker's goal is to run arbitrary script in the user's browser with the privileges of the target web site. Typically, an attacker will run script as a stepping stone to disrupting the confidentiality or integrity of the user's session with the target web site. In the limit, the attacker can always inject script into a web site if the attacker can induce the user into taking arbitrary actions. In this paper, we consider attackers who seek to achieve their goals with *zero interaction* or a *single-click interaction* with the user.

3. ATTACKS

In this section, we present attacks on existing client-side XSS filters. We first explain an architecture flaw in filters that block exfiltration of confidential information. We then exhibit inaccuracies in the simulations of the HTML parser used by filters that mediate before the response is parsed, showing how an attacker can bypass these filters. Finally, we demonstrate how client-side XSS filters can introduce vulnerabilities into otherwise vulnerability-free web sites.

3.1 Exfiltration Prevention

A number of client-side XSS filters attempt to mitigate XSS vulnerabilities by preventing the attacker's script from leaking sensitive data to the attacker's servers [5, 11, 23]. Typically, these filters monitor the flow of information within the web site's JavaScript environment and aim to block the attacker from exfiltrating that information to his or her servers.

One technical difficulty with preventing exfiltration is that web sites frequently export data to third-party web sites. For example, every web site that contains a hyperlink to another site leaks some amount of data to that site. Worse, modern web sites often have rich interactions with other web sites, e.g., via `postMessage`, OAuth, or advertising. To distinguish between "benign" and "malicious" information leaks, these client-side XSS filters often employ sophisticated analysis techniques, including taint tracking and static analysis, with the attendant false negatives and false positives.

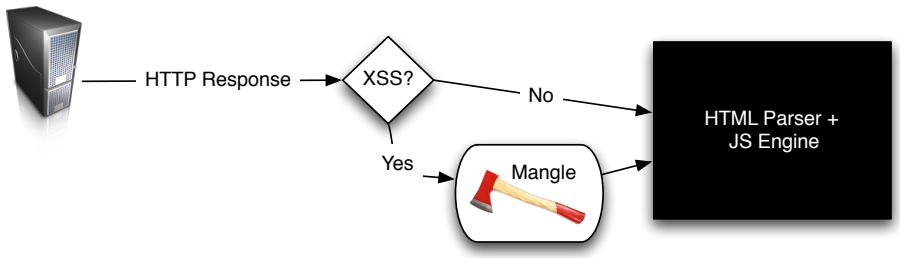

Figure 2: IE8 XSS Filter Architecture

Architectural Limitations. However, even if these filters could track sensitive information with zero false negatives and zero false positives, the exfiltration approach does not actually prevent attackers from disrupting the confidentiality or integrity of the user's session with the target site. For example, if the attacker can inject script into the user's online banking web site, the attacker can transfer money to the attacker's account by generating fake user input events programmatically. Worse, an attacker can almost always steal confidential information via *self-exfiltration*: exfiltrating the sensitive information via the honest web server. For example, many web sites provide a user-to-user messaging facility (e.g., YouTube, Flickr, and Facebook all provide in-site messaging). If the attacker sends the confidential information to his or her own user account in a user-to-user message, the attacker can log into the site later and retrieve the information, circumventing the exfiltration filter.

Even if the site does not provide an explicit user-to-user messaging mechanism, the attacker can almost always exfiltrate the confidential information anyway. For example, consider an attacker who is able to inject script into the Bank of America web site and wishes to exfiltrate some piece of confidential information, such as the user's soft second factor authentication token. The attacker's script can perform the following steps:

1. Simulate a click on the logout link.
2. Log into the attacker's account (in the user's browser) by filling out the login form (answering the attacker's secret questions as needed.)
3. Under account settings, select mailing address.
4. Save a mailing address that contains the information the attacker wishes to exfiltrate.
5. Log out of the attacker's account.

The attacker can then log into his or her own account at Bank of America (this time in his or her own browser), view the stored mailing address, and learn the confidential information. To determine how many bytes the attacker can leak using this technique, we examined the Bank of America web site for user-local persistent storage. Our cursory examination revealed that the attacker can exfiltrate at least 400 bytes per attack.

Alternatives. Some filters (e.g., [20, 14, 19]) avoid the above difficulties by blocking XSS attacks earlier. Instead of letting the attacker's script co-mingle with the target web site's script, these filters prevent the attacker from injecting malicious script in the first place. Typically, these filters block injection by searching for content that is contained in both the HTTP response and the HTTP request that generated the response. Although not necessarily indicative of a reflection, such repeated content suggests that the server simply reflected part of the request in the response.

One disadvantage of this technique is that filters based on matching content in the request and the response cannot mitigate *stored* XSS vulnerabilities because the attacker's script need not be present in the request. In a stored XSS attack, the attacker stores malicious content in the target web site's server. Later, when the user visit the server, the server sends the attacker's content to the user's browser. Unfortunately, exfiltration prevention techniques cannot block stored XSS attacks either. By definition, the presence of a stored XSS vulnerability implies that the attacker can store content in the server. Using this storage facility, the attacker can self-exfiltrate confidential information.

3.2 Pre-Parse Mediation

Client-side XSS filters that block injection typically match content in an HTTP response with content in the HTTP request that generated the response. Because responses often contain benign information from the request, these XSS filters narrow their focus to detecting script that is present in both the request and the response. However, detecting whether particular bytes in an HTTP response will be treated as script by a browser is not as simple a task as it appears.

Fidelity/Performance Trade-Off. Existing filters mediate between the network layer and the browser's HTML parser (see Figure 2). To determine whether a sequence of bytes in an HTTP response will be treated as script by the browser, these filters simulate the browser's HTML parser. Unfortunately, the browser's HTML parser is quite complex. The bytes in the response are decoded into characters, segmented into tokens, and then assembled into a document object model (DOM) tree. Simulating this pipeline is a trade-off between performance and fidelity.

- *Low performance.* The filter could re-implement exactly the same processing pipeline as the browser, but such a filter would double the amount of time spent parsing the HTTP response. For example, noXSS [19] contains an entire JavaScript parser for increased fidelity. Unfortunately, to achieve perfect fidelity, the filter would need to fetch and execute external scripts because external scripts can call the `document.write` API to inject characters into the processing pipeline, altering the parsing of subsequent bytes.

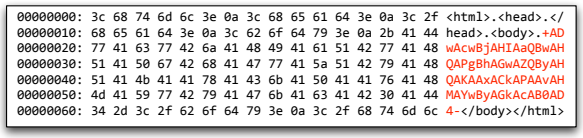

Figure 3: Identifying scripts in raw responses requires understanding browser parsing behavior.

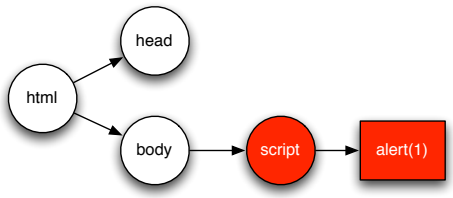

Figure 4: After the HTTP response is parsed, the script is easy to find.

- *Low fidelity.* Instead of implementing a high-fidelity simulation, the Internet Explorer 8 (IE8) [20] and NoScript [14] filters approximate the browser's processing pipeline with a set of regular expressions. These regular expressions are much faster than a complete HTML parser, but they over-approximate which bytes in the response will be treated as script. Low-fidelity simulations are forced to incur a large number of false positives because the penalty for incurring a false negative is high: an attacker can construct an attack that bypasses the filter. For example, consider this content:

  ```
  <textarea><script> ... </script></textarea>
  ```

  The IE8 filter flags this content as script even though the `<textarea>` element prevents the content from being interpreted as script, leading to a false positive.

To work around the false positives caused by its low-fidelity simulation, Internet Explorer 8 disables its XSS filter for same-origin requests. However, this reduction in false positives also comes with false negatives: instead of injecting script directly, an attacker can inject a hyperlink that fills the entire page and exploits exactly the same XSS vulnerability. When the user clicks this hyperlink, the filter will ignore the exploit (because the request appears to be originating from the same origin), letting the attacker run arbitrary script as the target web site.

Simulation Errors. Worse, even high-fidelity simulations are likely to deviate from the browser's actual response processing pipeline in subtle ways. If the attacker can desynchronize the simulated parser from the actual parser, the attacker can usually bypass the filter. In each of the filters we examined, we discovered attacks of this form:

- *noXSS.* The HTML parsing simulation used by noXSS does not correctly account for HTML entity encoded JavaScript URLs. An attacker can bypass the filter by injecting a full-page hyperlink to an HTML entity encoded JavaScript URL. If the user click anywhere on the page, the attacker can run arbitrary script as the target web site.
- *NoScript.* The HTML parsing simulation used by NoScript does not correctly account for the fact that the `/` character can be used to delimit HTML attributes. For example, the attacker can bypass the filter using an attack string that uses some complex parsing tricks such as `<a<img/src/onerror=alert(1)//<`.
- *IE8.* The Internet Explorer 8 filter does not correctly approximate the byte-to-character decoding process. If the browser decodes an HTTP response using the UTF-7 code page, the attacker can freely inject script (see Figure 3). This issue is particularly severe because, in Internet Explorer, the attacker can force a web page that does not declare its character set explicitly to be decoded using the UTF-7 code page [10], making the IE8 XSS filter ineffective at protecting web sites that do not explicitly declare their character set.

3.3 Induced False Positives

Once the filter has decided that a sequence of reflected bytes constitutes an XSS attack, the filter must prevent the browser from running the attacker's script. If the filter blocks the entire page, each false positive seriously degrades the user experience because users would not be able to view web pages that trigger false positives. Instead, pre-parse filters typically "mangle" injected script by altering the HTTP response in the hopes of preventing the injected script from executing. For example, IE8 replaces the `r` in `<script>` with a `#`, tricking the parser into skipping the script block.

Although a nuisance, unintentional false positives rarely open new security vulnerabilities in web sites. By contrast, false positives *induced* by an attacker can mangle or block security-critical code. An attacker can induce a false positive by including the security-critical code in a request to the victim site, confusing the filter into believing the server reflected the content and is the victim of an XSS attack. For example, the following URL will prevent `victim.com` from executing the `secure.js` JavaScript library:

```
http://victim.com/?<script src="secure.js"></script>
```

Because the string `<script src="secure.js">` is contained in both the request and the response, the filter believes that the attacker has injected the script into the victim web site and mangles the script. Induced false positives lead to a number of security issues, described below.

Container escape. Recently, mashups such as Facebook, iGoogle, Windows Live, and Google Wave have begun displaying third-party "gadgets" that seamlessly combine content from more than one source into an integrated experience. Because the gadget author is not trusted with arbitrary access to the user's account, these sites use frames or a JavaScript sandboxing technology such as FBJS [4], ADsafe [3], or Caja [6] to prevent the gadget from escalating its privileges.

Gadgets are typically rendered in a small rectangle and are not allowed to draw outside this area. Facebook uses cascading style sheets to confine gadgets to a limited region of the page. Because Internet Explorer lets style sheets

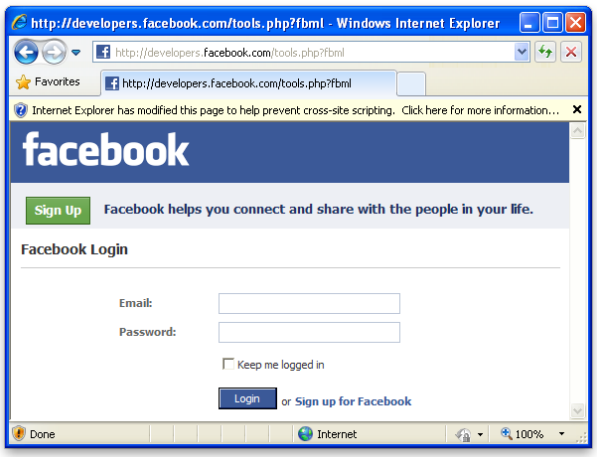

Figure 5: Container escape phishing attack using IE8's XSS filter to bypass Facebook's style restrictions.

contain script [16], IE8's XSS filter blocks attackers from injecting style sheets. An attacker, therefore, can trick IE8's XSS filter into mangling Facebook's protective style sheet by inducing a false positive, letting a malicious gadget escape its container. The attacker can then display a convincing fake login page hosted on `facebook.com` (see Figure 5) even though Facebook does not contain an XSS vulnerability. If WebKit allowed scripts in style sheets, we could block the injected script instead of mangling the style sheet.

Parse Tree Divergence. Mangling an HTTP response before parsing makes it difficult to predict how the remainder of the response will be interpreted by the browser. When an attacker induces a false positive and intentionally mangles a page, the browser will construct a different parse tree than the one intended by the author: code might be interpreted as static data and data might be interpreted as code. Parse tree divergence vulnerabilities have been discovered in the IE8 XSS filter in the past, allowing attackers to conduct XSS attacks against web sites that have no "inherent" XSS vulnerabilities [17].

Rather than sanitizing untrusted content in a way that is robust to arbitrary mangling of the page, some security-conscious web sites prefer to rely on their own server-side defenses to prevent code injection. For this reason, a number of popular web sites, including Google, YouTube, and Blogger, disable the IE8 XSS filter using the `X-XSS-Protection` header.

Clickjacking. In a typical clickjacking attack, the attacker's web page embeds a frame to the target web site. Instead of displaying the frame to the user, the attacker obscures portions of the frame and tricks the user into clicking on some active portion of the frame, such as the "delete my account" button, by displaying user experience that implies that the button serves a different purpose and belongs to the attacker's site. Until recently, the recommended defense for clickjacking was for the victim site to use a "frame busting" script to break out of the attacker's frame. As a result of misleading advice on sites such as Wikipedia, the Web is littered with poorly written frame busting scripts that can be circumvented. For example, PayPal uses this script:

```
if (parent.frames.length > 0) {
  top.location.replace(document.location);
}
```

PayPal's frame busting can be easily circumvented in several different ways. For example, the attacker can create a variable called `location` in the parent frame, preventing the above script for successfully changing the location of the attacker's frame [24]. The attacker can also cancel the navigation using an `onbeforeunload` handler [21]. Client-side XSS filters add yet another way to circumvent frame busting: the attacker can induce a false positive that disables the frame busting script [18].

4. XSSAUDITOR

In this section, we describe the design and implementation of a client-side XSS filter that achieves high performance and high precision without using regular expressions.

4.1 Design

Instead of mediating between the network stack and the HTML parser, we advocate interposing a client-side XSS filter between the HTML parser and the JavaScript engine, as shown in Figure 6. Placing the filter *after* the HTML parser has a number of advantages:

- *Fidelity.* By examining the response after parsing, the filter can easily identify which parts of the response are being treated as script (see Figure 4). Instead of running regular expressions over the bytes that comprise the response, the filter examines the DOM tree created by the parser, making the semantics of those bytes clear. Placing the filter after parsing also lets the parser correctly account for external scripts that use `document.write`.
- *Performance.* When the filter processes the response after the parser, the filter does not need to incur the performance overhead of running a high-fidelity simulation of the browser's HTML parser.
- *Complete interposition.* By placing the filter in front of the JavaScript engine, the filter can interpose completely on all content that will be treated as script. In particular, because the JavaScript engine has a narrow interface, we can have reasonable assurance that the filter is examining every script before it is executed. When the filter wishes to block a script, the filter can simply refuse to deliver the script to the JavaScript engine instead of mangling the response.

4.2 Implementation

We implemented a client-side XSS filter, called XSSAuditor, in WebKit. Our implementation has been accepted into the main line and is enabled by default in Google Chrome 4. The filter mediates between the WebCore component, which contains the HTML parser, and the JavaScriptCore component, which contains the JavaScript engine.

Interception Points. The filter interposes on a handful of interfaces. For example, the filter intercepts any attempts to run inline scripts, inline event handlers, or JavaScript URLs. The filter also interposes on the loading of external scripts and plug-ins. In addition to these interception points, two other points require special consideration.

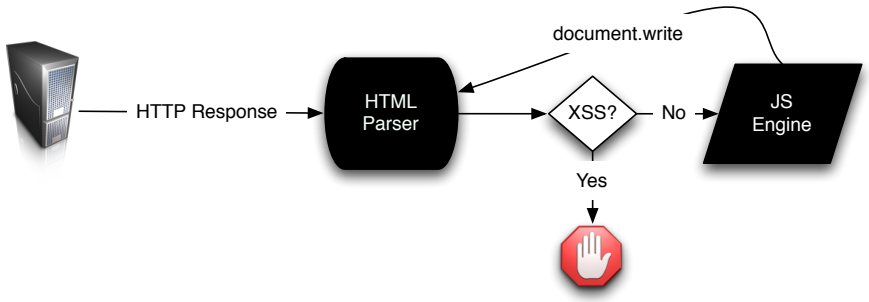

Figure 6: XSSAuditor Architecture

The HTML `<base>` element [1] is used to specify the base URL for all relative URLs in an HTML page. By injecting a `<base>` element (or altering the `href` attribute of an existing `<base>`), an attacker can cause the browser to external scripts from the attacker's server if the script are designated with relative URLs. For this reason, the filter causes the browser to ignore base URLs that appear in the request. To reduce false positives, the filter blocks base URLs only if the URLs point to a third-party host.

Data URLs [15] require special attention for Firefox XSS filters because data URLs inherit the privileges of the web page that contains the URL. However, data URLs are neither an XSS attack vector for Internet Explorer nor WebKit-based browsers because data URLs either do not work (in IE) or do not inherit the privileges of their referrer (in WebKit). Because our filter is implemented in WebKit, the filter does not need to block data URLs in hyperlinks or iframes. However, because data URLs contain attacker-supplied content, the filter prevents the attacker from injecting a data URL as the source of an external script or plug-in.

Matching Algorithm. Before searching for scripts in the HTTP request, the filter transforms the URL request (and any POST data) as follows:

1. URL decode (e.g., replace `%41` with `A`). This step mimics the URL decoding that the server does when receiving an HTTP request (e.g., before PHP returns the value of `$_GET["q"]`).

2. Character set decode (e.g., replace UTF-7 code points with Unicode characters). This step mimics a transformation done by the HTML tokenizer.

3. HTML entity decode (e.g., replace `&amp;` with `&`). The filter applies this transformation only to some of the interception points. For example, inline scripts are not entity decoded but inline event handlers are.

These steps assume that the server does not perform a complex transformation on the attacker's content. If the server does perform an elaborate transformation, the filter will not find an injected script in the request. In analyzing server vulnerabilities, we found that servers commonly apply two transformations: Magic Quotes and Unicode normalization.

- *Magic Quotes.* Prior to version 5.3.0, PHP automatically performs the `addslashes` transformation on request parameters. This transformation attempts to mitigate SQL injection by adding \ characters before ' and \ characters and by transforming null characters into `\0`. To account for this transformation, the filter ignores any \, `0`, or null characters when searching for the script in the request.

- *Unicode normalization.* A number of servers "normalize" Unicode characters by representing each Unicode character with its canonical code point. For example, the character ü can be represented either by the code point `U+0252` or the code point sequence `U+0075`, `U+0308` (the "u" character combined with a diacritical mark). Mimicking Unicode normalization is difficult and error prone because different servers might use different normalization algorithms. For this reason, the filter ignores all non-ASCII characters when searching for the script in the request.

Although the matching algorithm does simulate some of the transformations the server and the HTML parser apply to the attackers content, the filter does not need to simulate the complex parts of the parser, such as tokenization or element re-parenting.

Overflow. In some cases, an attacker can craft an exploit for an XSS vulnerability that is partially composed of characters supplied by the attacker and partially composed of characters that already exist in the page. The filter will be unable to find the entirety of such a script in the request because only a portion of the script originated from the request. For example, consider the following XSS vulnerability:

```
<?php echo $_GET["q"]; ?>
<script>
/* This is a comment. */
</script>
```

If the attacker uses the following exploit, the injected script will extend until the end of the existing comment:

```
<script>alert(/XSS/); /*
```

Instead of attempting to find the entire script in the request, the filter searches for the first 7 characters[1] of the script. Our hypothesis is that an attacker cannot construct an attack in less than 7 characters. For example, the attacker cannot even specify a URL on another server in less than 7 characters because the scheme-relative URL `//aa.cc` is 7 characters long.

[1]The version of the filter that we deployed in Google Chrome 4 does not implement the 7 character limit.

5. EVALUATION

In this section, we evaluate the correctness and the performance of our client-side XSS filter. By way of correctness, we evaluate what percentage of "naturally occurring" XSS vulnerabilities are mitigated by the filter, the filter's false positive rate, and our assurance regarding the filter's false negative rate. By way of performance, we measure the performance overhead of running the filter on a number of JavaScript and page-loading benchmarks.

5.1 Correctness

Client-side XSS filters do not require perfect correctness to be useful. However, the usefulness of a filter depends what percent of vulnerabilities the filter covers and the rate of false positives and false negatives.

Vulnerability Coverage. To estimate the percent of reflected XSS vulnerabilities covered by the filter, we analyzed 330 randomly selected, publicly disclosed XSS vulnerabilities from `xssed.com`. Of the selected vulnerabilities, 76 were "dead links" (meaning the site did not respond within 10 seconds or responded with an HTTP response code other than 200), 87 were fixed, and 22 were not XSS vulnerabilities. We were able to verify that the remaining 145 vulnerabilities were live, reflected XSS vulnerabilities. (There were no stored XSS vulnerabilities in this data set.)

Instead of testing whether the filter blocks the example exploit in the database, we classified the underlying vulnerability to assess whether the filter is designed to block all exploits for the vulnerability (see Figure 7). We found that 96.5% of the vulnerabilities were "in scope" for the filter, meaning that the filter is designed to prevent the attacker from exploiting these vulnerabilities to inject script. The remaining 3.5% of the vulnerabilities were out-of-scope because they let the attacker inject content directly inside a `<script>` element.

There are a number of limitations of this evaluation. First, the `xssed.com` data set is biased towards easy-to-discover vulnerabilities because the researchers who contribute the example exploits often discover the vulnerabilities using automated vulnerability scanners. Second, the evaluation is biased towards unfixed vulnerabilities because we excluded 87 vulnerabilities that were repaired before we conducted our study. However, even with these biases, these observations suggest that a significant fraction of naturally occurring reflected XSS vulnerabilities are in-scope for our filter.

False Positives. To estimate false positives, we deployed the filter to all users of the WebKit nightly builds and the Google Chrome Developer channel and waited for users of these browsers to file bug reports. Initial iterations of the filter had a number of interesting bugs, described below. After examining the false positives, we were able to adjust the filter to remove the false positives in all but one case, also described below.

An early iteration of the filter had a large number of false positives on web sites that contained `<base>` elements. A number of web sites use a base URL of a form analogous to `http://example.com/` on pages with URLs analogous to `http://example.com/foo/bar`. The filter blocked these `<base>` elements because the base URL occurred in the page's URL. We removed these false positives by whitelisting base URLs from the same origin as the page.

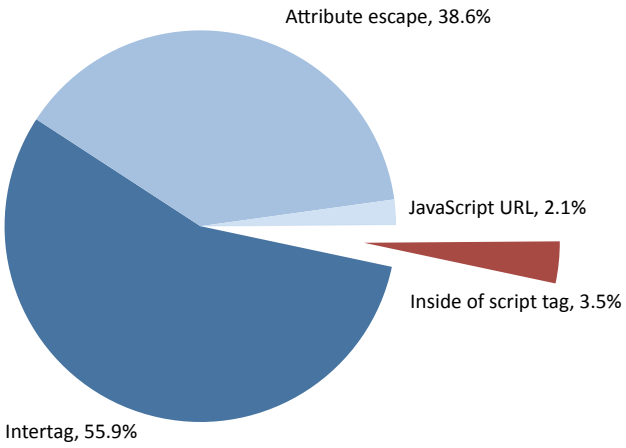

Figure 7: Underlying vulnerability for 145 verified reflected XSS vulnerabilities from `xssed.com`. 96.5% were "in-scope" for XSSAuditor.

An early iteration of the filter broke the chat feature on Facebook because the chat feature loads an external script from a URL supplied as a query parameter. Left unchecked, this behavior would be an XSS vulnerability. However, the Facebook server validates that the supplied URL points to a server controlled by Facebook. We removed this false positive by reducing the set of vulnerabilities that we cover to exclude direct injections into the `src` attribute of script elements. Because these vulnerabilities accounted for zero verified vulnerabilities in our `xssed.com` survey, we believe declaring these vulnerabilities out-of-scope is an acceptable trade-off to reduce false positives. We implemented this change by preventing a script element from loading an external script only if *all* of the bytes of the `src` attribute (including its name) appear in the request.

One subtle issue involves a user who authors a wiki that lets authors supply JavaScript content. Typically, a wiki author edits a wiki page in a `<textarea>` element that is sent to the server via a POST request. After the user edits a page, the server responds to the POST request by reflecting back the newly edited page. If the author includes JavaScript in the wiki page, the filter blocks the JavaScript in this response because the script is contained in the POST request. Of course, the wiki page containing the JavaScript is stored correctly in the server's database, and the wiki page functions correctly for subsequent visitors.

One user reported this issue as a false positive in his personal wiki. Upon investigating the issue, we discovered that the version of `DokuWiki` the user was running is in fact vulnerable to XSS because the "edit wiki" form is vulnerable to cross-site request forgery (CSRF). Thus, the "false positive" correctly identified the web site as vulnerable to XSS. (Unfortunately, the filter is unable to mitigate this vulnerability because the vulnerability is a *stored* XSS vulnerability.) A more recent version of `DokuWiki` repaired this XSS vulnerability by adding a CSRF token to the "edit wiki" form. However, it is unclear how the filter could distinguish between the vulnerable and the non-vulnerable cases.

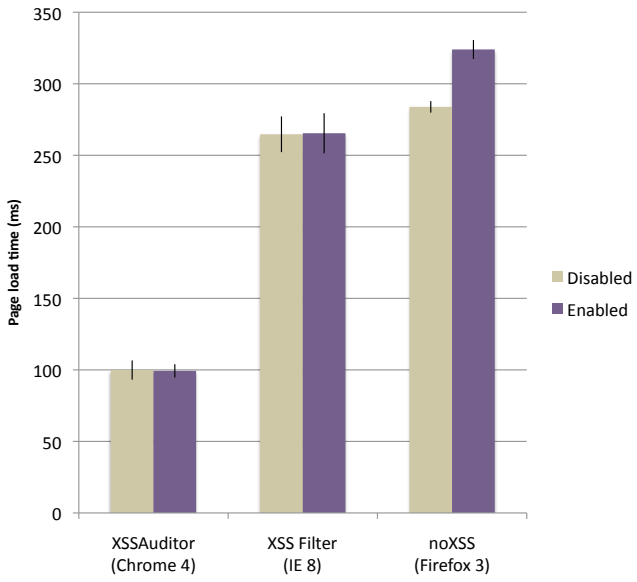

Figure 8: Score on the Mozilla page-load benchmark with 10 samples. Smaller is better. Error bars show 95% confidence.

False Negatives. Over the course of implementing the filter, we discovered a sequence of false negatives, but all of the false negatives were implementation errors that we repaired. After the implementation reached some level of maturity, we encouraged external security researchers to find additional false negatives. A number of researchers from `sla.ckers.org` participated [13] and found a false negative related to Unicode denormalization. In response, we changed the filter's matching algorithm to ignore all non-ASCII characters.

This experience suggests that we have low assurance that the filter lacks false negatives. We fully expect security researchers to discover more false negatives in the future, just as these researchers continue to discover arbitrary code execution vulnerabilities in mature code bases. However, the evidence is that these false negatives will be implementation errors that can be patched via auto-update.

Safety. Our filter resists two of the three induced false positive attacks described in Section 3.3:

- **Container Escape.** Because WebKit does not let web sites include script in style sheets, our filter does not prevent the attacker from injecting style sheets. Because our filter never disables style sheets, an attacker cannot induce a false positive to break out of a style container on Facebook.
- **Parse Tree Divergence.** Because we block JavaScript from executing directly rather than mangling the HTTP response before parsing, an attacker cannot create a parse tree divergence by inducing a false positive and sites do not need to worry about changing their server-side XSS filters to handle arbitrary mangling.

Our decision to block individual scripts rather than blocking the entire page when an XSS attack is detected means that, like the IE8 XSS filter, our filter can be used to disable poorly written frame busting scripts. However, because several techniques for disabling frame busting already exist, we recommend that sites replace their circumventable frame busting scripts with the `X-Frame-Options` HTTP response header [12], which was designed to help mitigate clickjacking. To protect users with legacy browsers that do not support this header, a web site operator should use a frame busting script that is robust to being disabled. For example, Twitter hides its pages by default and reveals them only if a script detects that the page is not in a frame.

Some web applications might wish that the XSS filter blocked the entire page when the filter detects an XSS attack, especially if an induced false positive might endanger the page's security. We let web developers enable full page blocking by sending the following HTTP header:

```
X-XSS-Protection: 1; mode=block
```

When the page includes this header, our filter will stop all script execution and display a blank page if the filter detects an XSS attack.

5.2 Performance

Performance is an essential factor in assessing the usefulness of a client-side XSS filter. Browser vendors are reluctant to deploy features that slow down key browser benchmarks, including JavaScript performance and page load time.

JavaScript. We evaluate the impact of the filter on core JavaScript performance using the industry-standard SunSpider [9] and V8 [7] benchmark suites. We were unable to measure any performance difference on these benchmarks as a result of the filter. This is unsurprising because the filter interposes on the interface to the JavaScript engine and does not interfere with the engine's internals.

Page-Load. We evaluated the impact of the filter on page-load performance using the `moz` page-load benchmark, which Mozilla and Google run in their continuous integration "buildbots" to detect performance regressions. Our filter does not incur a measurable performance overhead (see Figure 8). By contrast, the fidelity-focused noXSS filter incurs a 14% overhead on the benchmark, which is significant given the effort browser vendors spend improve their page load time score by even a few percentage points. (As expected, the IE8 filter did not incur a measurable overhead.)

6. CONCLUSION

We propose an improved design for a client-side XSS filter. Our design achieves high performance and high fidelity by interposing on the interface between the browser's HTML parser and JavaScript engine. Our implementation is enabled by default in Google Chrome.

Most existing client-side XSS filters simulate the browser's HTML parser with regular expressions that produce unnecessary false positives. These filters can be bypassed by exploiting differences between the simulation and the actual parser. Worse, when they detect an attack, the filters resort to mangling the HTTP response in a way that introduces vulnerabilities into otherwise vulnerability-free sites. Our post-parser design examines the semantics of an HTTP response, as interpreted by the browser, without performing a time-consuming, error-prone simulation. We block suspected attacks by preventing the injected script from being passed to the JavaScript engine rather than performing risky transformations on the HTML.

Cross-site scripting attacks are among the most common classes of web security vulnerabilities, and this trend shows no signs of reversing. Fixing every XSS vulnerability in a large web application can be a daunting task. Every browser should include a client-side XSS filter to help mitigate unpatched XSS vulnerabilities.

7. REFERENCES

[1] Tim Berners-Lee and Dan Connolly. Hypertext Markup Language - 2.0. IETF RFC 1866, November 1995.

[2] Steve Christey and Robert A. Martin. Vulnerability type distributions in cve, 2007. http://cwe.mitre.org/documents/vuln-trends/.

[3] Douglas Crockford. ADsafe.

[4] Facebook. Fbjs. http://wiki.developers.facebook.com/index.php/FBJS.

[5] David Flanagan. *JavaScript: The Definitive Guide*, chapter 20.4 The Data-Tainting Security Model. O'Reilly & Associates, Inc., second edition, January 1997.

[6] Google. Caja: A source-to-source translator for securing JavaScript-based web content. http://code.google.com/p/google-caja/.

[7] Google. V8 benchmark suite. http://v8.googlecode.com/svn/data/benchmarks/v5/run.html.

[8] Robert Hansen. XSS (cross site scripting) cheat sheet. http://ha.ckers.org/xss.html.

[9] Apple Inc. Sunspider. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[10] Inferno. Exploiting IE8 UTF-7 XSS vulnerability using local redirection, May 2009. http://securethoughts.com/2009/05/exploiting-ie8-utf-7-xss-vulnerability-using-local-redirection/.

[11] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC)*, 2006.

[12] Eric Lawrence. IE8 security part VII: Clickjacking defenses. http://blogs.msdn.com/ie/archive/2009/01/27/ie8-security-part-vii-clickjacking-defenses.aspx

[13] David Lindsay et al. Chrome gets XSS filters, September 2009. http://sla.ckers.org/forum/read.php?13,31377.

[14] Giorgio Maone. NoScript. http://www.noscript.net.

[15] Larry Masinter. The "data" URL scheme. IETF RFC 2397, August 1998.

[16] Microsoft. About dynamic properties. http://msdn.microsoft.com/en-us/library/ms537634(VS.85).aspx.

[17] Mitre. CVE-2009-4074.

[18] Eduardo Vela Nava and David Lindsay. Our favorite XSS filters/IDS and how to attack them, 2009. Black Hat USA presentation.

[19] Jeremias Reith. Internals of noXSS, October 2008. http://www.noxss.org/wiki/Internals.

[20] David Ross. IE 8 XSS filter architecture/implementation, August 2008. http://blogs.technet.com/srd/archive/2008/08/18/ie-8-xss-filter-architecture-implementation.aspx.

[21] Steve. Preventing frame busting and click jacking, Februrary 2009. http://coderrr.wordpress.com/2009/02/13/preventing-frame-busting-and-click-jacking-ui-redressing/.

[22] Andrew van der Stock, Jeff Williams, and Dave Wichers. OWASP top 10, 2007. http://www.owasp.org/index.php/Top_10_2007.

[23] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2007.

[24] Michal Zalewski. *Browser Security Handbook*, volume 2. http://code.google.com/p/browsersec/wiki/Part2#Arbitrary_page_mashups_(UI_redressing).