Precise client-side protection against DOM-based Cross-Site Scripting

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

Sebastian Lekies
SAP AG
sebastian.lekies@sap.com

Tobias Mueller
SAP AG
tobias.mueller07@sap.com

Patrick Spiegel
SAP AG
patrick.spiegel@sap.com

Martin Johns
SAP AG
martin.johns@sap.com

Abstract

The current generation of client-side Cross-Site Scripting filters rely on string comparison to detect request values that are reflected in the corresponding response's HTML. This coarse approximation of occurring data flows is incapable of reliably stopping attacks which leverage non-trivial injection contexts. To demonstrate this, we conduct a thorough analysis of the current state-of-the-art in browser-based XSS filtering and uncover a set of conceptual shortcomings, that allow efficient creation of filter evasions, especially in the case of DOM-based XSS. To validate our findings, we report on practical experiments using a set of 1,602 real-world vulnerabilities, achieving a rate of 73% successful filter bypasses. Motivated by our findings, we propose an alternative filter design for DOM-based XSS, that utilizes runtime taint tracking and taint-aware parsers to stop the parsing of attacker-controlled syntactic content. To examine the efficiency and feasibility of our approach, we present a practical implementation based on the open source browser Chromium. Our proposed approach has a low false positive rate and robustly protects against DOM-based XSS exploits.

1 Introduction

Ever since its initial discovery in the year 2000 [6], Cross-Site Scripting (XSS) is an ever-present security concern in Web applications. Even today, more than ten years after the first advisory, XSS vulnerabilities occur in high numbers [39] with no signs that the problem will be fundamentally resolved in the near future. Furthermore, in recent years, DOM-based XSS, a subtype of the vulnerability class that occurs due to insecure client-side JavaScript, has gained traction, probably due to the shift towards rich, JavaScript heavy Web applications. In a recent study, we have shown that approximately 10% of the Alexa Top 5000 carry at least one DOM-based XSS vulnerability [18].

The design of protection measures against XSS has received considerable attention. In its core, XSS is a client-side security problem: The malicious code is executed in the client-side context of the victim, affecting his client-side execution environment. Hence, a well-suited place to protect end users against XSS vulnerabilities is the Web browser. Following this concept, several client-side XSS filters have been developed over the years.

These contemporary client-side XSS filtering mechanisms rely on string-based comparison of outgoing HTTP requests and incoming HTTP responses to detect reflected XSS attack payloads. In essence, this string comparison is an approximation of server-side data flows that might result in direct inclusion of request data in the HTTP response. While this approximative approach is valid for server-based XSS vulnerabilities – the browser has no insight on the server-side logic – it is unnecessarily imprecise for client-side XSS issues.

In contrast to server-side problems, the complete data flow within the browser from attacker-controlled sources to the security-sensitive sinks into the browser's JavaScript engine occurs within one system and thus can be tracked seamlessly and precisely. Based on this observation, we propose a different protection approach: A client-side XSS filtering mechanism relying on precise dynamic taint tracking and taint-aware parsers within the browser.

To demonstrate the current limitations of the established approaches – which focus mainly on stopping reflected XSS attacks – we first conduct an in-depth analysis of the current state-of-the-art in client-side XSS filtering, with focus on the capabilities of thwarting DOM-based XSS attacks (see Section 3). In course of this analysis, we uncover a set of conceptual weaknesses which, taken together, render the existing techniques incapable of protecting against the majority of client-side XSS attacks (see Section 4). To practically validate our analysis, we report on a fully automatic system to create XSS attacks which evade the current protection mechanism:

Using a data set of 1,602 real-life DOM-based XSS vulnerabilities, we successfully created XSS vectors that bypassed client-side filtering in 73% of all cases, affecting 81% of all vulnerable domains we found.

Motivated by the results of our experiments, we propose an alternative approach for client-side prevention of DOM-based XSS: Using character-level taint tracking in the browser we can precisely detect cases in which attacker-controlled values end up in a syntactic parsing context which might lead to the browser executing the injected data as JavaScript. By enhancing the browser's HTML and JavaScript parsers to identify tokens that carry taint markers, we can efficiently and robustly stop injection attempts. To summarize, we make the following contributions:

- *Systematic analysis of conceptual shortcomings of current client-side XSS filters:* We report on a systematic investigation on the current state-of-the-art client-side XSS filter, the XSS Auditor, and identify a set of conceptual flaws that render the filter incapable of effectively protecting against DOM-based XSS attacks.
- *Automatic filter bypass generation:* To validate our findings and to demonstrate the severity of the identified filter limitations, we built a fully automated system to generate XSS payloads which bypass the string comparison based XSS filter. Our system leverages the precise flow information of our taint-enhanced JavaScript engine and our detailed knowledge on the Auditor's functionality to create exploit payloads that are tailored to a vulnerability's specific injection context and the applicable filter weakness. By practically applying our system to a set of 1,602 real-world DOM-based XSS vulnerabilities, we achieved a success rate of 73% successful filter bypasses.
- *Robust protection approach, utilizing client-side taint propagation:* Based on the identified weaknesses in the established XSS filtering approaches, we propose an alternative protection measure which is designed to address the specific characteristics of DOM-based XSS. Through combining a taint-enhanced browsing engine with taint-aware JavaScript and HTML parsers, we are able to precisely track the flow of attacker-controlled data into the parsers. This in turn enables our system to reliably detect and stop injected code on parse time.

2 Technical Background

In the following, we briefly discuss DOM-based Cross-Site Scripting and shed light on the technical basis used for this work, namely a taint-aware browsing engine.

2.1 DOM-based Cross-Site Scripting

Cross-Site Scripting (XSS) is a term describing attacks where the adversary is able to inject his own script code into a vulnerable application, which is subsequently executed in the browser of the victim in the context of this application. In contrast to the server-side variants of XSS, namely reflected and persistent, the term DOM-based Cross-Site Scripting (or DOM-based XSS) subsumes all classes of vulnerabilities which are caused by insecure client-side code. The term itself was coined by Klein in 2005 [16]. These issues come to light when untrusted data is used in a security-critical context, such as a call to `eval`. In the context of DOM-based XSS, this data might originate from different sources such as the URL, postMessages [38] or the Web Storage API.

2.2 Browser-level Taint Tracking

One of the underlying technical cornerstones of this paper is the taint-enhanced browsing browsing engine we developed for CCS 2013 [18]. This engine allows precise tracking of data flows from attacker-controlled sources, such as `document.location`, to sinks, such as `eval`.

Our implementation, based on the open source browser Chromium, provides support for tracking information flow on the granularity of single characters by attaching a numerical value to identify the origin of the character's taint. This taint marker is propagated whenever string operations are conducted and is also persisted between the two realms of the rendering component, i.e., Blink, and the V8 JavaScript engine.

As this part of our system is not one of the paper's major contributions, we omit further details for brevity reasons and refer the reader to the aforementioned paper.

3 Current Approaches for Client-side XSS Filtering

In this section we investigate the current in-browser techniques used to detect and prevent XSS attacks. More specifically, we describe the concepts of the Firefox plugin NoScript [20], Internet Explorer's XSS Filter [29] and Chrome's XSS Auditor [2].

3.1 Regular-expression-based Approaches: NoScript and Internet Explorer

One of the first mechanisms on the client side to protect against XSS attacks was introduced by the NoScript Firefox Plugin [22] in 2007. NoScript utilizes regular expressions to filter outgoing HTTP *requests* for potentially malicious payloads. If one of the regular expressions matches, the corresponding parts are removed from the HTTP request. The malicious payload will thus never reach the vulnerable application and hence an attack is thwarted. Nevertheless, as described in NoScript's feature list, this potentially leads to false positives [21] due to its aggressive filtering approach. NoScript works around this issue by prompting the user whether to repeat the request, this time disabling the protection mechanism. While this seems to be a valid approach for NoScript's security-aware users, it is not acceptable as a general Web browser feature, as many studies have shown that an average user is not able to properly react to such security warnings [9, 13, 34].

In order to tackle this problem Microsoft slightly extended NoScript's approach and integrated it into Internet Explorer [29]. Similar to NoScript, IE's XSS filter utilizes regular expressions to identify malicious payloads within outgoing HTTP requests. Instead of removing the potentially malicious parts from a request, the filter generates a signature of the match and waits for the HTTP response to arrive at the browser. If the signature matches anything inside the response, i.e., if the payload is also contained within the response, the filter removes the parts it considers to be suspicious. Thus, attacks are only blocked if the payload is indeed contained in the response and, hence, depending on the situation, false positives are less frequent. In fact, avoiding false positives is one of the filter's many design goals [30], even if this results in a higher false negative rate, as Microsoft's David Ross states: "Thus, the XSS Filter defends against the most common XSS attacks but it is not, and will never be, an XSS panacea." [30].

In 2010, Bates et al. [2] demonstrated that regular-expression-based filtering systems have severe issues and proposed a superior approach in the form of the XSS Auditor, which has been adopted by the WebKit browser family (Chrome, Safari, Yandex).

3.2 State-of-the-Art: The XSS Auditor

Based on the identified weaknesses of regular-expression-based XSS defenses, Bates et al. proposed the XSS Auditor – a new system that is "faster, protects against more vulnerabilities, and is harder for attackers to abuse" [2]. Up to now, the XSS Auditor constitutes the state-of-the art in client-side XSS mitigation, albeit focusing mainly on reflected XSS.

As we will demonstrate in this paper, the XSS Auditor also has shortcomings, especially related to DOM-based XSS attacks. Before we explore the limitations of the system in the next section, we provide an overview of the Auditor's protection mechanism.

One of the key differences between Chrome's XSS Auditor and previous filter designs is the filter's placement within the browser architecture. Instead of applying regular expressions on the string representations of the HTTP requests or responses, the Auditor is placed between the HTML parser and the JavaScript engine [2]. The idea behind this placement is, that an attacker's payload has to be parsed by the HTML parser to be transferred to the JavaScript engine where the injected payload is being executed.

In order to block XSS attacks, the Auditor receives each token generated by the HTML parser and checks whether the token itself or some of its attributes are contained in either the request URL or the request body. If so, the filter considers the token to be injected and replaces JavaScript or potentially harmful HTML attributes with a benign value. Such a benign value is a payload that has no effect, such as `about:blank`, `javascript:void(0)` or an empty string. The injected fragments will thus not be passed to the JavaScript engine and hence attacks are prevented.

The main design goals of the filter are to avoid false positives and to minimize performance impact. Before demonstrating that these goals severely impact the filter's detection capabilities, we will first provide details on the detection algorithm (simplified to satisfy space and readability constraints):

1. **Initialization (For document fragments)**
   (a) Deactivate the filter
2. **Initialization (For each full document)**
   (a) Fully decode the request URL
   (b) Fully decode the request body
   (c) Check if request could contain an injection
      i. If not, deactivate the filter
      ii. Otherwise continue
3. **For each start token in the document do...**
   (a) Check and delete dangerous attributes
      i. Delete injected event handlers
      ii. Delete injected JavaScript URLs
   (b) Conduct tag specific checks
4. **For each script token in the document do...**
   (a) Check and delete injected inline code

As soon as the so-called *HTMLDocumentParser* is spawned by Chrome, an initialization routine of the XSS Auditor is called. The parser can either be invoked for parsing document fragments or complete documents. While the XSS filter is deactivated for document fragments, it guesses whether an injection attack is likely to be present for full documents. If either the URL or the request body contains one of the characters shown in Listing 1, the filter is activated. If none of these characters is found, the filter assumes the browser not being under attack and skips the complete filtering process.

If, on the other hand, one of the characters mentioned in Listing 1 is present in the request the Auditor investigates every token within the document for injected values that might cause script execution. This process

Listing 1 Required characters to activate the filter

```
static bool isRequiredForInjection(UChar c)
{
    return (c == '\'' || c == '"' ||
            c == '<'  || c == '>');
}
```

is threefold: First the Auditor looks for dangerous attributes, second it conducts tag specific checks for certain attributes and third it filters injected inline scripts.

Dangerous Attributes are, in the view of the Auditor, attributes that either contain a JavaScript URL or have the name of an inline event handler (`onclick`, `onload`, etc.) as these attributes can enable XSS attacks. If such an attribute is found, the Auditor searches for it within the corresponding request. If a match is found, the filter assumes the attribute to be injected and either deletes the complete attribute value in case of event handlers or replaces the JavaScript URL with a benign URL.

Tag-specific filtering Besides event handlers and attributes containing JavaScript URLs, other tag specific attributes that need to be filtered exist. An attacker could, for example, inject a script tag and use the `src` attribute to load an external script file. Hence, for any script token, the Auditor additionally checks the legitimacy of the `src` attribute. In total, the Auditor conducts such checks for 18 additional attributes contained in 11 tokens (script, object, param, embed, applet, iframe, meta, base, form, input and button).

Filter inline scripts Whenever the Auditor encounters a script tag, it also validates whether the content between opening and closing tag has been injected. If the content can be found in the request, it is replaced with an empty string.

4 Limitations of String-based XSS Filters

In this section we report on a detailed analysis we conducted to assess the XSS Auditor's protection capabilities with a focus on DOM-based XSS. Although the XSS Auditor was designed to stop reflected Cross-Site Scripting attacks, it is also the most advanced and deployed filter against DOM-based XSS attacks. In the following, we therefore analyze issues related to the concept of the Auditor, which impair its capabilities of stopping DOM-based XSS attacks. In doing so, we show the arising need for a filter capable of stopping DOM-based XSS attacks.

4.1 Scope-related Issues

In general, the Auditor is called whenever potentially dangerous elements are encountered during the initial parsing of the HTTP response. These are, however, not the only situations in which attacker-controlled data might end up being interpreted as code. In this section, we explore situations in which the filter is not active and hence does not protect against attacks.

eval As mentioned earlier, the Auditor is placed between the HTML parser and the JavaScript engine to intercept potential XSS payloads. Still, not every DOM-based XSS attack needs to go through the HTML parser. If a Web site invokes the JavaScript function `eval` with user-provided data, the execution will never pass the HTML parser. Therefore, the Auditor will never see a malicious payload that an attacker injected into a call to `eval`. As we will demonstrate later, `eval` is commonly used in Web applications.

innerHTML While script tags inserted via `innerHTML` are not executed, it is still possible to execute JavaScript via inline event handlers. Hence, `innerHTML` is also prone to XSS attacks. In earlier versions of the Auditor content parsed via `innerHTML` was also filtered. Google later experienced some performance drawbacks in innerHTML-heavy applications [17] and as a consequence, the Auditor is nowadays disabled for document fragment parsing which is invoked upon an assignment to `innerHTML`.

Direct assignment to dangerous DOM properties Besides `eval` and `innerHTML` it is also possible to trigger the execution of scripts without invoking a HTML parsing process as a few examples in Listing 2 show. As no HTML parsing takes place in these cases, the XSS Auditor is never invoked. Hence, if a Web application assigns a user-controlled value to such a DOM property, an attacker is able to evade the filter.

Listing 2 Examples for dangerous DOM properties

```
var s = document.createElement("script");
s.innerText = "myFunction(1)";        // 1.
s.src = "http://example.org/script.js"// 2.

var i = document.createElement("iframe");
i.src = "javascript:myFunction(1)"    // 3.

var a = document.createElement("a");
a.href = "javascript:myFunction(1)"   // 4.
```

Second order flows When investigating a token, the Auditor always validates whether a suspicious value was contained within the preceding HTTP request's URL or body. As demonstrated by Hanna et al. [10], second order flows are relevant for DOM-based XSS. So, for example, if a value is written into LocalStorage within one request/response cycle, it can be used to cause a DOM-based XSS attack in another request/response pair. As the Auditor only investigates the last request, it will not find the value sent with the second-last request. LocalStorage is only one of many ways to persist data across multiple HTTP requests as Cookies, WebStorage or the File API exist nowadays.

Alternative attack vectors It is not sufficient to only check the URL and the request body in order to prevent DOM-based XSS attacks. Multiple other sources of attacker-controllable data exist which could be abused to inject malicious content into an application. Examples are the PostMessage API, the `window.name` attribute, or the `document.referer` attribute. As the Auditor does not take these sources into account, they can be used to evade the filter.

Furthermore, Bojinov et al. demonstrated that data can be injected by an attacker via alternative communication channels [4]. Thus, so-called cross-channel scripting attacks also bypass the XSS Auditor.

Unquoted attribute injection During initialization, the Auditor checks whether filtering is necessary by verifying the presence of the characters shown in Listing 1. In doing so, it implicitly assumes that an attack is not possible without these characters. This assumption, however, is wrong. In Listing 3 we show a common vulnerability and the corresponding attack (note: the value of the id attribute is not surrounded by quotes). In this example, the payload does not make use of the required characters. Normally, the XSS Auditor would block the `src` attribute containing the JavaScript URL. In this case, however, it does not conduct any checks as it is deactivated.

Listing 3 Unquoted Attribute injection

```
var id = location.hash.slice(1);
var code = "<iframe id=" + id + " [...]>";
    code += "</iframe>";
document.write(code);

// attack payload within URL
"//example.org/#1 src=javascript:eval(name)"
```

4.2 String-matching-related Issues

In the following we explore the limits of the implemented string matching algorithms. Whenever the Auditor finds a potentially dangerous element or attribute, it verifies whether the corresponding string representation can be found in the request. If an attacker is able to mislead the string-matching algorithm, the filter can be bypassed. Hence, the precision of this process determines the filter's effectiveness and as a result its false positive and false negative rates.

4.2.1 Partial Injections

One of the assumptions the Auditor makes is that an attacker has to inject a complete tag or attribute to successfully launch an attack. As a consequence the filter always aims to find the complete tag or the complete attribute within the request. While this approach reduces false positives as it is very unlikely that an application contains an existing tag or attribute in its URL legitimately, it does not regard application-specific scenarios. This assumption leads to potential problems in three different cases:

Attribute Hijacking One of the first things the Auditor does is to check whether a dangerous attribute was injected into the application. Hence, whenever it discovers a dangerous attribute during the parsing process it regenerates the string representation of the attribute and matches it against the URL and the request body. Listing 4 shows the string generation process:

Listing 4 Attribute string matching

```
// current start token
<iframe [...] onload="alert('example')">
// Step 1: extract the dangerous attribute
onload="alert('example')"
// Step 2: Truncate after 100 characters
onload="alert('example')"
// Step 3: Truncate at a terminating char
onload="alert('
```

After detecting a potentially dangerous attribute the Auditor extracts its decoded string representation. Then, it truncates the attribute at 100 chars to avoid the comparison of very long strings. It finally truncates the string at one of seven so-called terminating characters (this is done to detect attacks, that we will cover later). The resulting string is then matched against the URL. Obviously, the resulting string always contains the name of the potentially dangerous attribute. Hence, the underlying assumption here is that the attacker always has to inject the attributes herself. In real-world applications, however, attributes can often be hijacked by an attacker as shown in Listing 5. Although the `onload` attribute is a dangerous event handler attribute, the Auditor will not discover it within the URL as the `onload` attribute's name is hardcoded within the application and not injected by the attacker.

Listing 5 Attribute & Tag hijacking vulnerability

```
var h = location.hash.slice(1);
var code = "<iframe onload='" + h + "'"
    code += "[...]></iframe>";
document.write(code);

//attack for attribute hijacking
"//example.org/#alert('example')"
//attack for tag hijacking
"//example.org/#' srcdoc='...'"
```

Tag Hijacking After checking for dangerous attributes the Auditor conducts tag specific attribute checks. Matching all attributes of all tokens within an HTML document against the URL and request body, however, can be a very time consuming and error-prone task. Therefore, the auditor only matches an attribute against the URL if it can find the tag's name in the URL. For example, if the filter investigates an iframe token it validates whether the sequence `<iframe` is contained in the request before matching the `src` or `srcdoc` attribute [1]. Hence, if the injection point of a vulnerability lies within such a tag, the attacker can hijack the tag and inject additional attributes to it. As the tag itself is hardcoded the Auditor will skip any of its checks for specific attributes. An example of this attack is provided in Listing 5.

In-Script Injections Another vulnerability that is not detectable by the XSS Auditor is an injection inside of an existing inline script. As described in Section 3.2, whenever the filter encounters a script tag, it matches the *complete* inline content of the script against the request. Real-world Web applications however often make use of dynamically generated inline scripts made up from user-controllable input mixed with hardcoded values. Hence, instead of injecting a script tag via the URL an attacker is able to simply inject code into an existing dynamic inline script. As a consequence searching for the complete script content within sources of user input will not be successful.

4.2.2 Trailing Content

A very similar problem to partial injections is trailing content. When real-world Web applications write input to the document, they do not simply write one single value coming from the user but rather use a string that was constructed from hardcoded values as well as potentially attacker-controlled values. Listing 6 shows a real-world example.

Listing 6 An example of String construction

```
var code = "<iframe src='//example.org/";
    code += getParamFromURL("page_name");
    code += ".html'></iframe>";;
document.write(code);

// attack payload:
"' onload='alert(1);foo"
// resulting code
"<iframe src='//example.org/'
        onload='alert(1);foo.html'>"
```

Note, that the injection point is inside the src attribute of the iframe tag. Within this src attribute, the attacker-controllable input starts in the middle of the attribute (after `//example.org/`) and some more content is following the injection point (`.html`). When crafting an attack, the attacker is able to use the trailing content within the payload to confuse the string matching process. Despite the fact that the Auditor is aware of this issue (source code comments indicate this) and defends against it, the current defenses are not able to reliably detect which parts are actually injected by the attacker and which parts are hardcoded within the Web application. We found four bypasses which allow an attacker to exploit this problem in different and partly unexpected ways. Due to the high complexity and the limited space, we omit a detailed explanation here.

4.2.3 Double Injections

Another conceptional flaw of string-matching-based approaches is the inability to discover concatenated values coming from more than one source of user input. As we have shown in previous work [18], a call to a security sensitive function contains on average three potentially attacker provided substrings. Listing 7 shows an example for such a double injection.

Listing 7 An example of double injection

```
var id = getParamFromURL("id");
var name = getParamFromURL("name");
var code = "<iframe id='" + id + "'";
    code += " name='" + name +"'";
    code += "[...]></iframe>";
document.write(code);

// attack
id="'/><script>void('"
name="');alert(1)</script>"
// resulting code
<iframe id=''/>
<script>void(' name=');alert(1)</script>
[...]></iframe>
```

As the call to `document.write` contains two injection points (id, name) an attacker is able to split the payload. A specially crafted set of inputs, as shown in the Listing, therefore leads to the creation of a valid script tag that is a combination of both attacker inputs. In this case, the Auditor's string matching algorithm would search for `void('name=');alert(1)` within the request. Finding this value in the URL, however, is not possible as the `' name ='` part is hardcoded and not originating from the URL. Furthermore, the attacker is able to arbitrarily change the order in which the values appear within the URL. Hence, double injections are a severe conceptional problem for string-matching-based approaches. In total, we identified three different classes of double injection. The first class has been explained in the example above. A call to `document.write` contains two injection points

[1]For iframe.srcdoc the tag hijacking attack is not possible anymore, as concurrent research discovered this issue and reported it to Google. Upon the report Google changed the behavior for srcdoc. Nevertheless, for any other of the 18 special attributes, tag hijacking still is an issue

and the injected values are independent from each other. Very similar to this approach, the double injection pattern also applies to situations in which a single value is used twice within a single call to a security sensitive function. Finally, double injection attacks can be conducted if subsequent calls to `document.write` are made containing attacker-controllable values.

4.2.4 Application-specific Input Mutation

Another assumption of the XSS Auditor is that input of the user always reaches the parser without any modifications. If even one character of the input changed, the string matching algorithm will fail to find the payload and hence is not able to block the resulting attack. Application-specific encoding functions or data formats, therefore, lead to situations in which the filter can be bypassed.

4.3 Practical Experiments

As previously demonstrated we found numerous conditions under which the protection mechanisms of the XSS Auditor can be evaded, especially with respect to DOM-based Cross-Site Scripting. In order to assess the severity of the identified issues for real-world applications, we conducted a practical experiment. We used the methodology applied for our previous paper [18] to collect a set of 1,602 unique real-world DOM-based XSS vulnerabilities on 958 domains. We then built a bypass generation engine to verify whether a certain vulnerability allows employing one of the bypassing techniques described above.

Using our taint-aware infrastructure we are able to determine the exact injection context of a vulnerability. As soon as our infrastructure detects a call to a security sensitive sink such as `document.write`, `eval`, or `innerHTML`, it stores the string value and the exact taint information. Using a set of patched HTML and JavaScript parsers, we can exactly determine the location of the injection point. Using this data, we cannot only give an indication for a filter evasion possibility, but also generate an exact bypass that takes the injection point's context as well as the specific flaws of the Auditor into account. Applying this technique we compiled a set of bypasses that we evaluated against the vulnerabilities.

In doing so, we were able to bypass the filter for 73% of the 1,602 vulnerabilities, successfully exploiting 81% of the 958 domains in our initial data set.

4.4 Analysis & Discussion

As demonstrated by our practical experiments, the XSS Auditor – which aims at stopping reflected Cross-Site Scripting – can not stop DOM-based XSS attacks in the aforementioned cases. We therefore believe that additional defenses are necessary to combat this type of Cross-Site Scripting. Furthermore, the results of our analysis lead us to believe that the design of the XSS Auditor is prone to being bypassed in certain reflected XSS attack scenarios which are related to string-based matching issues. Since the focus of our work is on DOM-based XSS, we leave the investigation of this assumption to future work.

In our analysis, we identified two conceptual issues that limit the Auditor's approach in detecting and stopping DOM-based XSS attacks.

Placement One of the Auditor's strengths compared to Internet Explorer's and NoScript's approach is its placement between the HTML parser and the JavaScript engine. This way the Auditor does not need to approximate the browser's behavior during the filtering process. As we have shown in Section 4.1 the current placement is prone to different attack scenarios which are not taken into account by the filter. Currently the Auditor is not able to catch JavaScript-based injection attacks and situations in which HTML parsing is not conducted prior to a script execution.

String matching Even if it would be possible to extend the Auditor's reach to the JavaScript engine and the so-called *DOM bindings*, the string matching algorithm is another conceptual problem that will be very difficult if not impossible to solve. In order to cope with the situation the XSS Auditor introduced many additional checks and optimizations to thwart attacks. Nevertheless and despite the fact that a lot of bug hunters regularly investigate the filter's inner workings, we were able to find 13 bypasses targeting the string matching algorithm. All the mentioned problems will not disappear as employing string matching is inherently imprecise.

5 Preventing Client-side Injection Attacks during Parse-time

As the previous section has shown, current concepts of Cross-Site Scripting filters are not designed to thwart DOM-based XSS and, thus, are not sufficient to protect users against these kinds of attacks. In this section, we discuss the methodology behind our newly proposed filter as well as the corresponding policy considerations. We then go into detail on the issue of handling postMessages in our filter and finally outline the technical challenges we had to overcome to implement the concept into a real-world browser.

5.1 Methodology Overview

As we have demonstrated in Section 4.2, client-side XSS filters relying on string comparison lack the required precision for robust attack mitigation. String comparison as an approximation of occurring data flows is a necessary evil for flows that traverse the server. For DOM-based XSS, this is not the case: The full data flow occurs within the browser's engines and can thus be observed precisely. For this reason, we propose an alternative protection mechanism that relies on runtime tracking of data-flows and taint-aware parsers and makes use of two interconnected components:

- A taint-enhanced JavaScript engine that tracks the flow of attacker-controlled data.
- Taint-aware JavaScript and HTML parsers capable of detecting generation of code from tainted values.

This way our protection approach reliably spots attacker-controlled data during the parsing process and is able to stop cases in which tainted data alters the execution flow of a piece of JavaScript. In the following, we discuss the general concept and security policy, whereas we go into more detail on the implementation in Section 5.4 and investigate the implications of our proposed filtering approach in Section 6.1.

5.2 Precise Code Injection Prevention

As we outlined in the previous section our protection approach relies on precise byte-level taint tracking.

In the following we give a detailed overview on the necessary changes we performed in order to implement our filtering approach. More specifically, we made changes to the browser's rendering engine, the JavaScript engine and the DOM bindings, which connect the two engines.

JavaScript Engine When encountering a piece of JavaScript code, the JavaScript engine first tokenizes it to later execute it according to the ECMAScript language specification.

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

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

DOM bindings Very similar to the previous case the execution of remote active content can also be triggered via a direct assignment to a script or object tag's src attribute via the DOM API. This assignment does not take place within the HTML parser but rather inside the DOM API. We therefore patched the DOM bindings to implement the same policy as mentioned above.

Intentional Untainting As our taint-aware browser rejects the generation of code originating from a user-controllable source, we might break cases in which such a generation is desired. A Web application could, for example, thoroughly sanitize the input for later execution. In order to enable such cases we offer an API to taint and untaint strings. If a Web application explicitly wants to opt-out of our protection mechanism, the API can be used to completely remove taint from a string.

5.3 Handling Tainted JSON

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

5.4 Implementation

To practically validate the feasibility of our protection approach we conducted a prototypical implementation based on the open source browser, Chromium, version 30.0.1561.0. This section will provide details on a selection of issues we encountered when implementing the desired protection capabilities.

Equality problem for tainted strings: We had to decide when a tainted string should be considered equal to an untainted version as this requirement is dependent on the situation at hand. Under certain circumstances we do want to consider them as being equal but there are also conditions under which equality should not be given. For example, when creating DOM elements from tainted strings, we do want a tainted string to be equal to an untainted version because the tainted string should match the untainted version for the correct element to be created. If the strings would not match, the correct element could not be looked up and hence an unknown (or custom) element would be created. On the other hand, when looking up a string in a cache, we do not want the tainted version to be equal to an untainted one. If that were the case, we might loose taint as we retrieve the untainted version. For performance reasons WebKit uses addresses of certain strings it considers to be unique to perform an equality check. We thus needed to implement a fallback method for the equality check on tainted strings if we desire a tainted string to be equal to its untainted version.

Attaching taint to JavaScript Tokens: To prevent code strings from untrusted sources to generate code, we needed to forward taint information from strings to these generated tokens. We thus needed to broaden the interface not only leading to the JavaScript lexer but also to the parser. V8 not only has a parser for the JavaScript language but also for JSON to efficiently read serialized data. While it was conceptually easy to attach another bit to the generated tokens, a sophisticated buffering logic inside V8 needed to be made taint aware. A variety of `CharacterStream` classes buffer characters of an input stream to be consumed by the scanner and also enables it to push back characters if it did not accept them. To enable taint propagation all classes of an inheritance hierarchy at least three levels deep needed to be changed.

6 Practical Evaluation

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

6.1 Compatibility

While a secure solution seems desirable, it will not be accepted by users if it negatively affects existing applications. Therefore, in the following, we discuss the compatibility of our proposed defense to real-world applications. We differentiate between the two realms in which our approach is deployed – namely the JavaScript parser and the HTML/DOM components – and answer the questions:

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

6.1.1 Analysis methodology

To answer these questions for a large body of domains, we conducted a shallow crawl of the Alexa Top 10,000 domains (going down one level from the start page) with our implemented filter enabled. Rather than just blocking execution we also sent a report back to our backend each time the execution of code was blocked. Among the information sent to the backend were the URL of the page that triggered the exception, the exact string that was being parsed as well as the corresponding taint information. Since we assume that we are not subject to a DOM-based XSS attack when following the links on said start pages, we initially count all blocked executions of JavaScript as false positives. In total, our crawler visited 981,453 different URLs, consisting of a total of

9,304,036 frames. The percentages in the following are relative to the number of frames we analyzed.

6.1.2 Compatibility of JavaScript Parsing Rules

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

To get a better insight into whether these false positive were in fact caused by vulnerable JavaScript snippets, we manually tried to exploit the flows which had triggered a parsing violation. Out of the 23 issues related to data from postMessages, we found that one script did not employ proper origin checking, allowing us to exploit the insecure flow. Of the 27 other scripts which were not using data from postMessages, we were able to exploit 21 scripts and hence 21 additional domains. This constitutes a total number of 50 domain on which one functionality caused a false positive, while 22 domains contained an actual vulnerability in just the functionality our filter blocked.

Importance of JSON-handling policy As we outlined in Section 5.3, we do allow for postMessages to contain tainted JSON which is automatically selectively untainted by our prototype. To motivate the necessity for this decision, we initially also gathered taint exceptions caused by tainted JSON (stemming from postMessages) being parsed by `eval`. This analysis showed that next to the 5,979 taint exceptions we had initially encountered, 90,937 additional documents utilized tainted JSON from postMessages in a policy-violating manner. Albeit, with respect to our data set, this only caused issues with less than 1% of all documents we analyzed, it puts emphasis on the necessity for our proposed selective untainting, whereas on the other hand, it also shows that programmers utilize `eval` quite often in conjunction with JSON exchanged via postMessages.

6.1.3 Compatibility of HTML Injection Rules

As discussed in the Section 5.2, our modified browser blocks the execution of external scripts if any character in the domain name of the external script resources is tainted – only exempting those scripts that are located on the same domain as the including document. Analogous to what we had investigated with respect to the JavaScript parsing policy, we wanted to determine in how many applications we would potentially break functionality when employing the proposed HTML parsing policy. We therefore implemented a reporting feature for *any* tainted HTML and a blocking feature for policy-violating HTML. This feature would always send a report containing the URL of the page, the HTML to be parsed, as well as the exact taint information to the backend. We will go into further detail on injected HTML in Section 7 and will now focus on all those tainted HTML snippets which violate the policy we defined in Section 5.2.

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

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

6.1.4 Compatibility of DOM API Rules

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

After having counted the false positive, we yet again tried to exploit the flows that had been flagged as malicious by our policy enforcer. Out of the 60 domains our enforcer had triggered a block on, we verified that eight constitute exploitable vulnerabilities. In compari-

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

Table 1: False positives by blocking component

son to the amount of exploitable blocks we had encountered for the JavaScript and HTML injection polices this number seems quite low. This is due to the fact that both the aforementioned advertisement providers employed whitelisting to ensure that only script content hosted on their domains could be assigned. In total, this sums up to 0.60% false positives with respect to domains and just 0.002% of all analyzed documents.

6.1.5 Summary

In this section we evaluated the false positive rate of our filter. In total, the filtering rules inside the JavaScript parser, the HTML parser and the security-sensitive DOM APIs caused issues on 14,966 document across 183 domains. Considering the data set we analyzed this amounts to a false positive ratio of 0.16% for all analyzed documents and 1.83% for domains. Noteworthy in this instance is however the fact that out of the 183 domains on which our filter blocked a single functionality, 90 contained actual verified vulnerabilities in just that functionality. Table 1 shows the number of documents and domains on which our policy caused false positive, also denoting in which of the different policy-enforcing components the exception was generated as well as the amount of domains in which the blocked functionality caused an exploitable vulnerability.

6.2 Protection

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

To verify the effectiveness of our proposed protection scheme, we ran all generated exploits and bypasses against our newly designed filter. To minimize side-effects, we also disabled the XSS Auditor completely to ensure that blocking would only be conducted by our filtering mechanism. As we discussed in Section 4.2, alongside the scoping-related issues that were responsible for the successful bypassing of the Auditor by the first generation of exploits, other issues related to string matching arose. In the following, we briefly discuss our protection scheme with respect to stopping these kinds of exploits.

Scoping: eval and innerHTML In contrast to the XSS Auditor our filtering approach is fully capable of blocking injections into `eval` due to the fact that it is implemented directly in the JavaScript parser. In the XSS Auditor, innerHTML is not checked for performance reasons. To check whether a given token was generated from a tainted source, a simple boolean flag has to be checked, therefore we do not have these performance-inhibiting issues.

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

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

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

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

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

6.3 Performance

In order to evaluate the performance of our implementation we conducted experiments with the popular JavaScript benchmark suites Sunspider, Kraken, and Octane as well as the browser benchmark suite Dromaeo. Sunspider was developed by the WebKit authors to "focus on short-running tests [that have] direct relevance to the web" [28]. Google has developed Octane which includes "5 new benchmarks created from full, unaltered, well-known web applications and libraries" [5]. Mozilla has developed Kraken which "focuses on realistic workloads and forward-looking applications" [15]. Dromaeo, which is a combination of several JavaScript and HTML/DOM tests, finally serves as a measure of the overall impact our implementation has on the everyday browsing experience.

All tests ultimately lead to a single numerical value, either being a time needed for a run (the lower the better) or a score (the higher the better), reflecting the performance of the browser under investigation. For runtime (score) values the results were divided by the values obtained for the unmodified version of the Web browser (vice versa). These serve as the baseline for our further comparisons. With the obtained results we computed a slowdown factor reflecting how many times slower our modified version is. To set these numbers into context, we also evaluated other popular Web browsers, namely Internet Explorer 11 and Firefox 26.0. To eliminate side effects of, e.g., the operating system or network latency, we ran each of the benchmarking suites locally for ten times using an Intel Core i7 3770 with 16GB of RAM. All experiments, apart from Internet Explorer, were conducted in a virtual machine running Ubuntu 13.04 64-bit on that system whereas IE was benchmarked natively running Windows 7.

Table 2 shows the results of our experiments. To ascertain a baseline for our measurements we ran all benchmarks on a vanilla build of Chromium. The table shows the mean results (in points or milliseconds) as well as the standard error and the slowdown factor for each test and browser. Internet Explorer employs an optimization to detect and remove dead code, causing it to have significantly better performance under the Sunspider benchmark than the other Web browsers [40]. As the results generated by all browsers under the Kraken benchmark were varying rather strongly, we ran the browsers in our virtual machines 50 times against the Kraken benchmark. Regardless, we still see a rather high standard error of the mean for all the browsers.

We chose the aforementioned tests because they are widely used to evaluate the performance of both JavaScript and HTML rendering engines. Nevertheless, these tests are obviously not designed to perform operations on tainted strings. As we discussed in Section 5.4, our engine usually only switches to this runtime implementation if the operation is conducted on a tainted string. In the initial runs, which is denoted in Table 2 as *Tainted Best*, our engine incurred slowdown factors of 1.08, 1.01, 1.16 and 1.05, resulting in an average slowdown factor of 7%. Since the tests are not targeting the usage of tainted data, we conducted a second run. This time we modified our implementation to treat *all* strings as being tainted, forcing it to use as much of our new logic as possible. In this, the performance was naturally worse than in the first run. More precisely, by calculating the average over the observed slowdown factors for our modified (denoted as *Tainted Worst*) version, we see that our implementation incurs, in the worst case, an overhead of 17% compared to the vanilla version. While the performance hit is significant, we will elaborate on possible performance improvements in the next section.

6.4 Discussion

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

In our compatibility crawl we found that 183 of the 10,000 domains we analyzed had one functionality that was incompatible with our policies for the JavaScript parser, the HTML parser and the DOM APIs. Although this number appears to be quite high at first sight it also includes 90 domains on which we could successfully a vulnerability in just the functionality that was blocked by our filter. On the other hand, the total number of domains, which our approach protected from a DOM-based XSS attack amounts to 958. Although the XSS Auditor is not designed to combat DOM-based XSS attacks, it is the only currently employed defense for Google Chrome against such attacks. As we discussed in Section 4.3, the Auditor could be bypassed on 81% of these domains, protecting users on only 183 domains in our initial data

	Dromaeo			Octane			Kraken (ms)			Sunspider (ms)		
Baseline	1167.4	*1.89*	–	20177.9	*64.47*	–	1418.9	*94.29*	–	169.02	*0.37*	–
Tainted Best	1082.6	*2.40*	**1.08**	19851.0	*54.54*	**1.01**	1653.1	*59.84*	**1.16**	178.03	*0.70*	**1.05**
Tainted Worst	1015.6	*1.93*	**1.15**	18168.7	*70.24*	**1.11**	1814.4	*64.55*	**1.27**	192.66	*0.26*	**1.14**
Firefox 26.0	721.7	*2.94*	**1.62**	16958.5	*97.40*	**1.19**	1291.3	*1.14*	**0.91**	171.86	*0.65*	**1.02**
IE 11	607.0	*2.13*	**1.92**	17247.2	*47.15*	**1.17**	1858.5	*4.16*	**1.31**	78.05	*0.13*	**0.46**

Table 2: Benchmark results, showing mean, *standard error* and **slowdown factor** for all browsers and tests

set. This shows that with respect to its protection capabilities our approach is more reliable than currently deployed techniques, which do not offer protection against this type of attack.

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

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

7 Outlook: HTML Injection

As we discussed in Section 5.4, our engine allows for precise tracking of tainted data throughout the execution of a program and hence, also to the HTML parser. Therefore, our approach also enables the browser to precisely block all attacker-injected HTML even it is not related to Cross-Site Scripting. Although this was out of scope for this work, we believe that it is relevant future work. Therefore, we give a short glimpse into the current state of the Web with respect to partially tainted HTML passed to the parser.

As we discussed in Section 6.1, we conducted a compatibility crawl of the Alexa Top 10,000 in which we analyzed a total of 9,304,036 documents, out of which 632,109 generated 2,393,739 tainted HTML reports. Typically, each of the HTML snippets contained the definition of more than one tag. In total, we found that parsing the snippets yielded in 3,650,506 tainted HTML elements whereas we consider an element tainted if either the tag name, any attribute name or any attribute value is tainted. Considering the severity of attacker-controllable HTML snippets, we distinguish between four types:

1. Tag injection (**TI**): the adversary can inject a tag with a name of his choosing.
2. Attribute injection (**AI**): injection of the complete attribute, namely both name and value
3. Full attribute value injection (**FAVI**): full control over the value, but not the name
4. Partial attribute value injection (**PAVI**): attacker only controls part of the attribute

We analyzed the data we gathered in our crawl to determine whether blocking of tainted HTML data is feasible and if so, with what policy. Our analysis showed that out of the Top 10,000 Alexa domains, just one made use of full tag injection, injecting a `p` tag originating from a postMessage. This leads us to believe that full tag injection with tainted data is very rare and not common practice.

The analysis also unveiled that the most frequently tainted elements – namely `a`, `script`, `iframe` and `img` – made up for 3,503,655 and thus over 95% of all elements containing any tainted data. Hence, we focused our analysis on these and examined which attributes were tainted. Analogous to our definition of a tainted element, we consider an attribute to be tainted if either its name or value contains any tainted characters. Considering this notion, we – for each of the four elements – ascertained which attribute is most frequently injected using tainted data. For `a` elements, the most frequent attribute containing tainted data was `href` whereas `script`, `iframe` and `img` tags mostly had tainted `src` attributes. Although we found no case where the name of an attribute was tainted, we found a larger number of elements with full attribute value injection. The results of our study are depicted in Table 3, which shows the absolute numbers of occurrences. We also gathered reports from documents

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

Table 3: Amounts of full and partial value injection for domains in the Alexa Top 10,000 and beyond.

on domains not belonging to the Alexa Top 10,000 as content is often included from those. The first number in each column gives the amount for documents on the Alexa Top 10,000, whereas the second number shows the number for all documents we crawled.

Summarizing, we ascertain that taint tracking in the browser can also be used to stop HTML injection. Our study on tainted HTML content on the Alexa Top 10,000 domains has shown that blocking elements with tainted tag names is a viable way of providing additional security against attacks like information exfiltration [7] while causing just one incompatibly. We also discovered that the applications we crawled do not make use of tainted attribute names, hence we assume that blocking tainted attributes does also not cause incompatibilities with the current Web. In contrast, blocking HTML that either has fully or partially tainted attribute values does not appear to be feasible since our analysis showed that 8% of all domains make use of fully tainted attribute values whereas more than 44% used partially tainted values in their element's attributes. As there is an overlap between these two groups of domains, the total number of domains that would causes incompatibilities is 4,622, thus resulting in more than 46% incompatibilities. Thus, we established that although blocking HTML is technically possible with our implementation this would most likely break a large number of applications.

8 Related work

XSS Filter As already mentioned earlier, the conceptually closest work to this paper is Bates et al.'s [2] analysis of regular expression-based XSS filters and the subsequent proposal of the methodology that constitutes the basis for the XSS Auditor. Furthermore, Pelizzi and Sekar [26] proposed potential improvements for Bates et al.'s method in order to address the problem of partial injections. Similar to what Bates et al. discussed, they instrument the HTML parser and apply approximate string matching inside it. Due to the fact that DOM-based XSS allows an attacker to make use of insecure calls to `eval` as well as direct assignments to security-sensitive DOM APIs, it is still susceptible to some bypasses we discussed. Furthermore, the presented approach is not fully evaluated, especially with respect to the occurring false positive rate. Besides this, and the other two major browser-based XSS filters [21, 29], the majority of XSS protection approaches, such as [23, 14, 24, 35], reside on the server-side.

Filter evasion is an active topic especially in the offensive community. Academic approaches in this area include, for instance, the work by Heiderich et al. on SVG-based evasions [11] and filter evasion by misusing browser-based parser quirks and mutations [12] as well as approaches that rely on parser confusion and polyglots, such as Barth et al. [1] and Magazinius et al. [19].

Dynamic taint tracking Taint propagation is a well established tool to address injection attacks. After its initial introduction within the Perl interpreter [37], various server-side approaches have been presented that rely on this technique [25, 27, 33, 24, 3]. In 2007, Vogt et al. [36] pioneered browser-based dynamic taint tracking, employing the technique to prevent the leakage of sensitive data to a remote attacker rather than trying to prevent the attack itself. The first work to utilize taint tracking for the detection of DOM-based XSS was DOMinator [8], which was later followed by FLAX [31] and Lekies et al. [18]. For NDSS 2009, Sekar [32] proposed and implemented a scheme for taint inference, speeding up taint tracking approaches which had been presented up to this point.

9 Conclusion

In this paper we presented the design, implementation and thorough evaluation of a client-side countermeasure which is capable to precisely and robustly stop DOM-based XSS attacks. Our mechanism relies on the combination of a taint-enhanced JavaScript engine and taint-aware parsers which block the parsing of attacker-controlled syntactic content. Existing measures, such as the XSS Auditor, are still valuable to combat XSS in cases that are out of scope of our approach, namely XSS which is caused by vulnerable data flows that traverse the server.

In case of client-side vulnerabilities, our approach reliably and precisely detects injected syntactic content and, thus, is superior in blocking DOM-based XSS. Although our current implementation induces a runtime overhead between 7 and 17%, we believe that an efficient native integration of our approach is feasible. If adopted, our technique would effectively lead to an extinction of DOM-based XSS and, thus, significantly improve the security properties of the Web browser overall.

Acknowledgment

This work was in parts supported by the EU Projects WebSand (FP7-256964) and STREWS (FP7-318097). The support is gratefully acknowledged. The authors would also like to thank Sven Kälber for providing the computational power to conduct our data collection as well as the anonymous reviewers and our shepherd Adrienne Porter Felt for their help comments and support.

References

[1] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 360–371. IEEE, 2009.

[2] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[3] Prithvi Bisht and VN Venkatakrishnan. Xss-guard: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2008.

[4] Hristo Bojinov, Elie Bursztein, and Dan Boneh. Xcs: cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 420–431. ACM, 2009.

[5] Stefano Cazzulani. Octane: the javascript benchmark suite for the modern web. Online, `http://blog.chromium.org/2012/08/octane-javascript-benchmark-suite-for.html`, August 2012.

[6] CERT. Advisory ca-2000-02 malicious html tags embedded in client web requests. [online], `http://www.cert.org/advisories/CA-2000-02.html`, February 2000.

[7] Eric Y Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Proceedings of the Workshop of Web*, volume 2. Citeseer, 2012.

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

[9] Serge Egelman, Lorrie Faith Cranor, and Jason Hong. You've been warned: an empirical study of the effectiveness of web browser phishing warnings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1065–1074. ACM, 2008.

[10] Steve Hanna, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The emperors new apis: On the (in) secure usage of new client-side primitives. In *Proceedings of the Web*, volume 2, 2010.

[11] Mario Heiderich, Tilman Frosch, Meiko Jensen, and Thorsten Holz. Crouching tiger-hidden payload: security risks of scalable vectors graphics. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 239–250. ACM, 2011.

[12] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z Yang. mxss attacks: attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 777–788. ACM, 2013.

[13] Cormac Herley. So long, and no thanks for the externalities: the rational rejection of security advice by users. In *Proceedings of the 2009 workshop on New security paradigms workshop*, pages 133–144. ACM, 2009.

[14] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Advanced Information Networking and Applications, 2004. AINA 2004. 18th International Conference on*, volume 1, pages 145–151. IEEE, 2004.

[15] Erica Jostedt. Release the kraken. Online, `https://blog.mozilla.org/blog/2010/09/14/release-the-kraken-2/`, Sept. 2010.

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

[17] Andreas Kling. Xssauditor performance regression due to threaded parser changes. [online], `https://gitorious.org/webkit/webkit/commit/aaad2bd7c86f78fe66a4c709192e3b591c557e7a`, April 2013.

[18] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204. ACM, 2013.

[19] Jonas Magazinius, Billy K Rios, and Andrei Sabelfeld. Polyglots: crossing origins by crossing formats. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 753–764. ACM, 2013.

[20] Georgio Maone. Noscript. [online], `http://www.noscript.net/whats`.

[21] Georgio Maone. Noscripts anti-xss protection. [online], `http://noscript.net/featuresxss`.

[22] Georgio Maone. Noscripts anti-xss filters partially ported to ie8. [online], `http://hackademix.net/2008/07/03/noscripts-anti-xss-filters-partially-ported-to-ie8/`, July 2008.

[23] Raymond Mui and Phyllis Frankl. Preventing web application injections with complementary character coding. In *Computer Security–ESORICS 2011*, pages 80–99. Springer, 2011.

[24] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[25] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. *Automatically hardening web applications using precise tainting*. Springer, 2005.

[26] Riccardo Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *AsiaCCS*, May 2012.

[27] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th international conference on Recent Advances in Intrusion Detection*, RAID'05, pages 124–145, Berlin, Heidelberg, 2006. Springer-Verlag.

[28] Filip Pizlo. Announcing sunspider 1.0. [online], `https://www.webkit.org/blog/2364/announcing-sunspider-1-0/`, April 2013.

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

[30] David Ross. IE8 Security Part IV: The XSS Filter. [online], `http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx`, July 2008.

[31] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

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

[33] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *ACM SIGPLAN Notices*, volume 41, pages 372–382. ACM, 2006.

[34] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. Crying wolf: An empirical study of ssl warning effectiveness. In *USENIX Security Symposium*, pages 399–416, 2009.

[35] Mike Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE, 2009.

[36] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007)*, 2007.

[37] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[38] Web Hypertext Application Technology Working Group. Cross-document messaging. Online, `http://www.whatwg.org/specs/web-apps/current-work/multipage/web-messaging.html`.

[39] WhiteHat Security. Website security statistics report. [online], `https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf`, May 2013.

[40] Windows Internet Explorer Engineering. Html5, and real world site performance: Seventh ie9 platform preview available for developers. Online, `http://blogs.msdn.com/b/ie/archive/2010/11/17/html5-and-real-world-site-performance-seventh-ie9-platform-preview-available-for-developers.aspx`.