DOM based Cross Site Scripting - Client-Side Attacks on Browsers

Andrea Hauser
Offense Department, scip AG
anha@scip.ch
https://www.scip.ch

Marc Ruef (Editor)
Research Department, scip AG
maru@scip.ch
https://www.scip.ch

Abstract: DOM stands for Document Object Model. XSS stands for cross-site scripting. The main difference between DOM based XSS and other XSS vulnerabilities is that the payload is embedded on the client side rather than the server side. DOM based XSS vulnerabilities therefore have to be prevented on the client side.

Keywords: Browser, Burp, Cross Site Scripting, Detect, Exploit, HTML, Javascript, OWASP, Payload, Request

1. Preface

This paper was written in 2017 as part of a research project at scip AG, Switzerland. It was initially published online at *https://www.scip.ch/en/?labs.20171214* and is available in English and German. Providing our clients with innovative research for the information technology of the future is an essential part of our company culture.

2. Introduction

Cross-site scripting (XSS) vulnerabilities first became known through the *CERT Advisory CA-2000-02* [1] (Malicious HTML Tags Embedded in Client Web Requests), although these vulnerabilities had been exploited before. So XSS has already been around for a while. And as you'd expect, there have already been a number of Labs articles on the subject, including *Cross Site Scripting – The Underestimated Danger* [2] and *Cross-Site-Scripting and Preventing Script Injection – A Brief Guide* [3]. However, this article focuses largely on *DOM based cross-site scripting*, a term first coined in 2005 by *Amit Klein* [4]. He publicized the definition of DOM based cross-site scripting (XSS) and delineated it from other cross-site scripting vulnerabilities. To understand DOM based cross-site scripting, you first need a deeper understanding of DOM.

3. What is DOM?

The *Document Object Model* [5] is a *program interface* that defines the *structure of documents*. What is relevant for our purposes is that DOM provides for the *object-oriented representation of HTML documents*, specifically:

- How an HTML element is represented as an object
- What the attributes of an HTML element are
- Which methods can be used to access an HTML element
- What the events of an HTML element are

This means that DOM specifies how HTML elements can be found, added, changed or deleted in the browser. That means that DOM offers an opportunity for manipulating HTML pages. These days, JavaScript is usually used for this kind of manipulation.

It should be noted that each browser implements the DOM standard differently. This can affect whether a specific DOM XSS attack works in a particular browser.

4. What is DOM based XSS?

A DOM based XSS vulnerability arises when the DOM is used to generate dynamic content containing user input that can be processed without checking. This kind of attack is carried out with JavaScript in the user's browser. Here the locations that (malicious) user input bring into the DOM are designated as *source*. The locations in which (malicious) user input can be executed in the DOM are designated as *sink*.

An example of a simple cross-site scripting vulnerability:

```
<!DOCTYPE html>
<html>
    <body>
        <script>
            var source = "Hello " +
decodeURIComponent(location.hash.split("#")
[1]);  //Source
            var divElement =
document.createElement("div");
            divElement.innerHTML = source;
//Sink

document.body.appendChild(divElement);
        </script>
    </body>
</html>
```

The vulnerability can, for instance, be triggered with the following GET request:

```
GET www.vulnerable-website.example#<img
src="test" onerror="alert('XSS')">
```

We can see that this example required using the `decodeURIComponent` method. This is because modern browsers encode special characters in the URL and the attack would not function without these special characters

being decoded. The content of the variable `source` is added to the div element using `innerHTML`. This is the problematic element in the code, because its assignment to `innerHTML` causes the value included with it to be interpreted as HTML. If the value contains JavaScript, this is executed as well.

It should be pointed out that this does not alter the server response. The malicious modification only arises when the JavaScript code is executed on the client side, which the DOM subsequently alters.

The best-known sources and sinks are listed below. It should be noted that this is by no means an exhaustive list.

4.1. Sources

- document.URL
- document.referrer
- location
- location.href
- location.search
- location.hash
- location.pathname

4.2. Sinks

- eval
- setTimeout
- setInterval
- document.write
- element.innerHTML

5. Preventing DOM based XSS

Because there are cases where the payload never actually makes it to the server, preventing this vulnerability is not a task for the server side. The example above is one such a case. The reason is that content which appears after a # character is not sent by the browser to the server.

The first and most important recommendation is to avoid using user-controlled data for the dynamic generation of content whenever possible. If this is completely unavoidable, the best way to prevent DOM based XSS is to use secure output methods (sink) that prevent the execution of any maliciously embedded code. In the code snippet above, the only way to make the code safe is to replace `.innerHTML` with `.textContent`. This is because – unlike `innerHtml` – HTML elements in the `textContent` method are not executed.

The *DOM based XSS Prevention Cheat Sheet* [6] from OWASP has plenty of useful tips for developing secure dynamic websites with JavaScript. There you will also find descriptions of context-dependent encoding, which is required whenever it is impossible to switch to a secure output method.

6. Differences from other XSS vulnerabilities

The most important difference is where the attack is embedded in the code. In contrast to other cross-site scripting vulnerabilities, the code is not embedded on the server side, but rather on the client side. This means that the payload cannot be found in the response code. DOM based XSS vulnerabilities can therefore only be detected at runtime or by checking the website's DOM.

In most cases, finding DOM based XSS vulnerabilities is no easy matter. One helpful approach that uses the *static code analysis* [7] approach is described in the article *Burp* [8] based-xss.

7. External Links

[1] https://www.cert.org/historical/advisories/CA-2000-02.cfm
[2] https://www.scip.ch/en/?labs.20060519
[3] https://www.scip.ch/en/?labs.20120105
[4] http://www.webappsec.org/projects/articles/071105.html
[5] https://www.w3.org/DOM/" title="DOM
[6] https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
[7] https://www.scip.ch/en/?labs.20140424is
[8] https://support.portswigger.net/customer/portal/articles/2325926-using-burp-scanner-to-test-for-dom