Cross-Site Scripting or XSS

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.

Let us understand it with an example:

Suppose we visited a website named abc.com who sell some good cloths but prices were high and we close the website later we received a mail where we actually got an off of 90% and there was a link www.abc.com/shirts/fullsleves/productid=1asd212Sda/!@a1
so we ordered some clothes from that link and completed our transaction online. Suddenly we got an update and we came to know that some more transactions happened from our account which were not done by us. This means that our data has been leaked and as we did shopping online and after that only our data was lost so it means there is a high chance that our data has been lost from the website or we can say our data has been leaked from the website. We will understand in depth what might have gone wrong here which causes a data loss whereas the flaws that allow these attacks to succeed are quite more than we can think and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.
There are three types of XSS present:

1. Temporary XSS (AKA non-persistent or Type II)
2. Permanent XSS (AKA persistent or Type I)
3. DOM based XSS (AKA Type 0)

Cross-Site Scripting or XSS

1. Temporary XSS:

1.1. What is temporary xss

In layman language temporary xss are the vulnerabilities which allow hackers to insert malicious codes into the HTML code of the browser.

Temporary xss happens when a user clicks on a user made link as it is not stored on the website. In the above example we can see that the link looks fine www.abc.com/shirts/fullsleves/productid=1asd212Sda/!@a1 but if you observe clearly then /!@ extension to the original link is what represents that something has been added to the original link. Let us take some other example and see how good we are suppose I visit a website named abcklm.com there I purchased something and after the payment is completed I again got an popup saying Your order is placed successfully and a OK button as it seems fine we press OK button but when we wait for the product we didn't receive and then we went to website to check what has happened and there we saw that order has been cancelled by us but we never did it. Remember that POP up is the place where this magic happens.

In this scenario malicious code is stored anywhere in the application and only the user who clicks on a hacker made link is affected.

1.2. How can we find them?

Temporary XSS are hard to find as they can be present anywhere some common places where you can find them are:

1.2.1. Accept Cookies when we login to a website.
1.2.2. Pop Ups whenever we login to a user or signout from a user or make payment.
1.2.3. In emails

When a website uses customer details to show customs results are likely to be found vulnarable to temporary XSS. So whenever we pass something to a webpage wand we want to check whether it is vulnerable or not we can do that by looking and some code.

1.3. JavaScript usage in Temporary XSS.

Javascript helps us to use keyloggers which can steal user cookies or some token which can later be used by hackers to fetch important data about the user. If we are getting blocked we can use an event listener or check the source code. When we are working with a website and we use Javascript we can prepare POC(Proof Of Concept) for example:

```
http://www.xyz.com/post.php?user_id=ks1912<a
href=”abc.com”> Chek new messages </a> <script> alert(
“POC”)</script>
```

And in the output we got an alert box saying “POC” with a “OK” button then that box is Proof of Concept for us.

2. Permanent XSS:

2.1. What is Permanent XSS?

To understand permanent XSS lets see an example from real life. In 2005 like Facebook there was a social media company known as Myspace. It was brought down by a hacker named samy's. He did this by exploiting XSS vulnerability. He was able to reflect the special character of the source code of the application by entering them in a status. Samay also found that if he passes custom html status to his web application he can pass the same html status to anyone who views his status which can be furture executed by the browser. Samay then passes a malicious code which adds anyone to his friend list to whoever sees the status.

Permanent XSS are the vulnerabilities in which a hacker injects and executes malicious client side scripts through the browser which gets permanently stored in the database.

In this scenario malicious code is stored in the database and all the users of the website are affected.

2.2. Exploiting Permanent XSS.

As a defensive measure many websites use a protected filter to stop all these vulnerabilities but are they really secure? Let's find out.
To bypass these filters we can either see the source code and find out which part of the website is getting blocked by the server or we can use an event listener. We can also check whether data is being sent as a POST or GET method. If it is in the POST method then we can

play with the URL like injecting comments with a custom url which can be used to steal our data or using burp suite.

3. DOM based XSS.

3.1. What is DOM based XSS?

DOM based XSS arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.In this scenario the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. For example, the source (where malicious data is read) could be the URL of the page (e.g., document.location.href), or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data (e.g., document.write).”

4. How can we protect ourselves from Cross Site Scripting or XSS?

4.1. HTML Encode Before Inserting Untrusted Data into HTML Element Content
4.2. Attribute Encode Before Inserting Untrusted Data into HTML Common Attributes
4.3. JavaScript Encode Before Inserting Untrusted Data into JavaScript Data Values
4.4. CSS Encode And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values

4.5. URL Encode Before Inserting Untrusted Data into HTML URL Parameter Values

5. References:

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
https://portswigger.net/web-security/cross-site-scripting
https://owasp.org/www-community/attacks/xss/