TASK

XSS (Cross-Site Scripting) Vulnerability

Visit our website

Introduction

WELCOME TO THE CROSS-SITE SCRIPTING TASK!

This task introduces you to your first web attack. While it is important to know how to use the tools for making applications secure, it is just as important to recognise which tools make an application insecure. In the words of Sun Tzu: “If you know the enemy and know yourself, you need not fear the result of a hundred battles. If you know yourself but not the enemy, for every victory gained you will also suffer a great deal. If you know neither the enemy nor yourself, you will succumb in every battle.” If you know how attackers think and work, you’ll be able to better understand how to defend your system.

WHAT IS CROSS-SITE SCRIPTING?

In the world of hacking, security vulnerabilities only exist where custom code can be run. If the hacker can send you something that runs on your PC, and you unknowingly run it, only then is it possible to hack you. This is why the Web is such a dangerous place – you visit various websites, and each site runs code (JavaScript) by default.

In the case of cross-site scripting, the attack is a pretty blatant one. For example, you receive an email from **facebooksecurity@gmail.com** that states that you’ve been hacked! In order to recover your account, there is a link included in the email that you need to click. In a blind panic, you click the link, and it takes you to a website that looks like the normal Facebook website. However, that is not the case; hidden behind a somewhat legitimate-looking URL is a website specifically designed to steal your login details to hijack your account.

It turns out that that attacker had re-created the HTML for Facebook so that it is almost indistinguishable from the actual site. However, there is one difference: there is some JavaScript in the site that tells your computer to download a virus in the background without your knowledge.

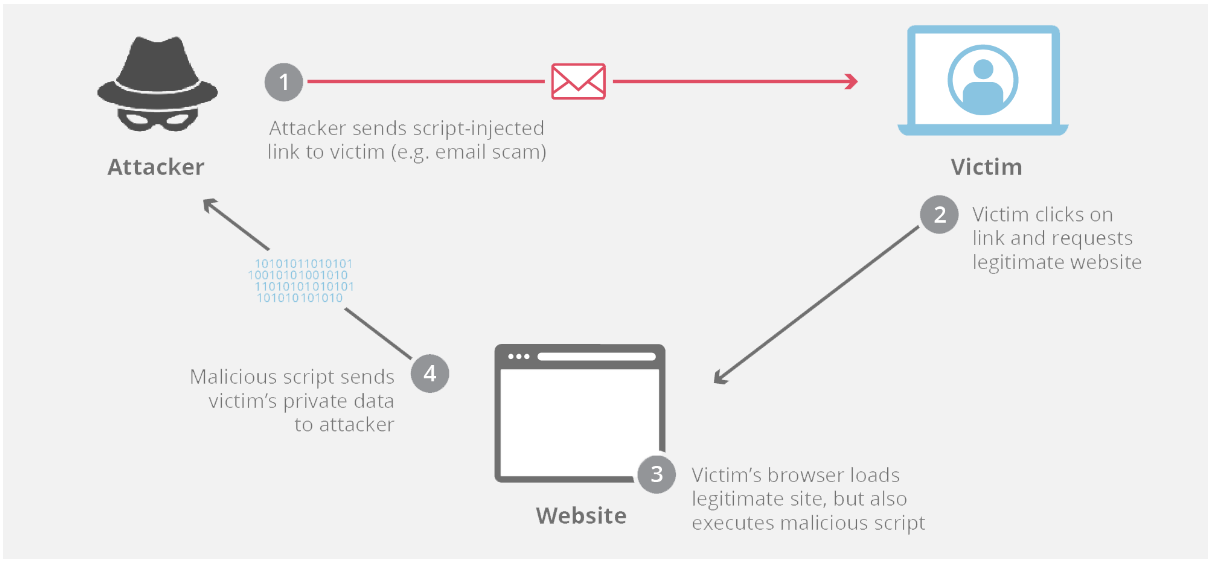

Attacker
1
Attacker sends script-injected link to victim (e.g. email scam)
Victim
2
Victim clicks on link and requests legitimate website
Website
3
Victim's browser loads legitimate site, but also executes malicious script
4
Malicious script sends victim's private data to attacker
10101011010101
10010101001010
11010101010101
101010101010

Image Source: **https://www.appsealing.com/types-of-cyber-attacks/**

Because your web browser trusts JavaScript with personal information, such as cookies (data stored by websites on your browser), the hacker has access to this information. Common actions performed by attackers on the victim's web browser include:

- Redirecting the browser to a more malicious site.
- Performing malicious activities on the attacker's behalf. If the attacker can gain access to run commands on your PC, then they are able to tell your computer to do things. For example, hacking other PCs from your computer. The attacker can then add the vulnerable PC to a zombie botnet, from where more major distributed attacks can be organised.

INJECTING CODE

What is "injecting" code anyway? This is something that will be touched on in more detail further down the line. As a brief introduction, code *injection* is possible when user input is not validated and compiled along with the rest of a specific statement. A common example of this is the infamous **eval()** function. This is a method that reads in a string and runs code based on that string. An example of the eval() function that is used in a calculator:

```
print(eval("1 + 1")) # this should be 2
```

Quite often, websites won't be as blatant as this. For example, one can modify this code to interact with the user like this:

```
first_number = input("Enter first number")
```

```
second_number = input("Enter second number")
operator = input("Enter your operator (+-*/)")
print(eval(f"{first_number}{operator}{second_number}"))
```

This is a quick and lazy way of building up a calculator. The user can enter their own numbers and operator. While there is no obvious way to run your own code here, it is actually surprisingly easy to inject code. The user inputs would simply have to be:

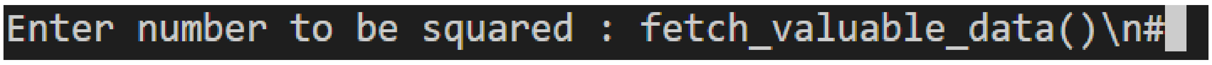
```
first_number = "fetch_valuable_data()"
second_number = ""
operator = ""
print(eval(f"{first_number}{operator}{second_number}"))
# "fetch_valuable_data()"
```

Where `fetch_valuable_data()` is some hypothetical function that fetches valuable data about users. This will then send all of that valuable data to the hacker, so that they can use it for their unlawful purposes.

The above example code made it a bit too easy to inject code. Let's try something a bit more difficult. Assume that we have an application that reads in a number from the user and prints the square of that number:

```
first_number = input("Enter number to be squared : ")
print(exec(f"{first_number}^2"))
```

Note the **`^2`** at the end of the string. If we enter `fetch_valuable_data()`, it will only throw an error, instead of fetching our valuable data. However, you can use the commenting feature of languages to get around this try entering: `fetch_valuable_data()\n#`. In the command-line, this would look something like:



```
Enter number to be squared : fetch_valuable_data()\n#
```

Note the use of the **exec()** function in Python. This is different from **eval()** as it is able to run multiple lines of code.

This will result in the server running the following code:

```
fetch_valuable_data()
#^2
```

This compiles and runs just fine. Instead of simply causing an error message to occur, our injection can now fetch our valuable data.

Just one note: websites that throw errors with user input are very valuable for hackers. Generally, when scanning for vulnerabilities, hackers will try to use URLs that contain comment characters or runnable code by default. If that causes a website to throw an error, that typically means that it interpreted the hacker's input as code. This usually prompts the hacker to poke and prod until that vulnerability is found. Eventually, they will get malicious code to run successfully. When designing websites, it is good to try and avoid errors being thrown by any user input whatsoever - this is a great way to hide vulnerabilities.

TYPES OF XSS

There are various classes of XSS. They all follow a general pattern – some code is injected somewhere, and that injected code runs malicious software on the victim's computer:

- **Reflected XSS**: this occurs from untrustworthy links. The victim follows a link sent by the attacker. Included within the link is a request to the server to pull the important information belonging to the user and send it to the attacker.
    - Typically, this is done through the URL - if the attacker can get the user to click on a dodgy URL, this can cause the server to pull a malicious script embedded in the URL and run it (normally causing important data to be sent to the attacker).
    - The URL is where the malicious script is injected. If it's something like https://facebook.com/account/<script>fetch_data()</script>, it is probably reflected XSS. This is easy to spot, as you may see HTML tags in the URL - URLs should never contain tags!
- **Stored XSS**: this doesn't even need a suspicious link to work. You can go to an entirely trustworthy website and still be hacked. In this type of XSS, the attacker will insert some code into the website itself, so that the website serves that code secretly to the users that access it. This code then runs any variation of malicious software.
    - For example, if a website allows users to write posts that contain HTML tags (such as the <b> tag), it is possible that it will also allow users to write posts containing <script> tags.

PREVENTING XSS

Hacking happens where users are able to write something that can be run as code. Therefore, the most surefire way to prevent this is to ensure that whatever is written by users cannot be stored as code. For example, storing and running HTML tags should not be possible in a secure website. Some general measures include:

- **Filtering input**: This is not a catch-all, but it can limit what the attacker can use. To prevent Reflected XSS, ensure that you are filtering URLs, and to prevent Stored XSS, ensure that you are filtering and sanitising whatever user inputs you store in the database. In addition, a server administrator can set their server up such that any URL data gets encoded as HTML special characters, effectively thwarting any attempts of reflected XSS attacks.

- **Encoding Outputs**: This is something that helps to prevent Stored XSS. Encoding outputs is a way to ensure that, even if users input HTML tags and store them on the database, the clients' browsers won't interpret any HTML tags as runnable code. Web servers running PHP can do this using the htmlspecialchars() function, which turns characters such as <, >, and & into `&lt;`, `&gt;`, and `&amp;` respectively, which tells your browser to display those characters directly instead of treating them as HTML code.

- **Response Headers**: By telling the clients' web browser the type of data included in the response, it will know not to run certain scripts. For example, using the **Content-Type: text/html** header will tell your browser that the response only include HTML, without any form of Javascript.

- **Content Security Policy**: This is a specific type of header you can include in your responses. It is a mechanism specifically built to mitigate XSS. With Content Security Policy (CSP), you can, for example, provide your web server with a list of all trusted sites to pull data from. That way, if a Reflected XSS tries to make itself known, it won't work if the XSS is trying to pull a script from a site that isn't whitelisted.

Instructions

For this task, you will be injecting code using input. Explore the example in the **Input example folder** about reading information from an external file, which will form a part of the compulsory task.

Next, open up **example.py**. This will show you how we will be injecting code. Code can be injected through the command line or through an input file.

Compulsory Task 1

- For **hack1.py**, **hack2.py** and **hack3.py**, your task is to create three input files called **hack1.txt**, **hack2.txt** and **hack3.txt**.
- In each input file, your goal is to call the **hack()** function.

Completed the task(s)?

Ask an expert code reviewer to review your work!

Review work

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.