HyperionDev

Cross-Site Scripting (XSS) Vulnerability

Task

Visit our website

Introduction

This task introduces you to the first type of web attack you're going to learn about. While it's important to know how to use the tools for making applications secure, it's just as important to recognise which tools and practices can make an application insecure.

In cyber security, if you know how attackers think and work, you'll be able to better understand how to defend your system.

Understanding cross-site scripting

Cross-site scripting (XSS) is a common web security vulnerability that allows attackers to inject malicious scripts into websites that are then executed in the browser of unsuspecting users. These scripts can steal sensitive information, such as login credentials, or perform actions on behalf of the user without their knowledge.

One common form of attack involves phishing, where the attacker tricks the victim into interacting with a fraudulent website. For example, you might receive an email that appears to be from a trusted source, like **facebooksecurity@gmail.com**, claiming that your account has been hacked. The email instructs you to click a link to recover your account. In a moment of panic, you click the link, which takes you to a website that looks nearly identical to Facebook.

However, this website is a fake. The attacker has recreated the HTML of Facebook to make the page look authentic. The key difference is that the fake site contains malicious JavaScript code. This code could be designed to steal your login credentials by capturing what you type or to download malware onto your computer without your knowledge.

The web is a particularly vulnerable space, because websites often run scripts, such as JavaScript, by default. These scripts are what enable interactive features on websites, but they also provide a vector for attackers to execute harmful code. This is why it's crucial to be cautious about the links you click on and the sites you visit, even if they appear legitimate.

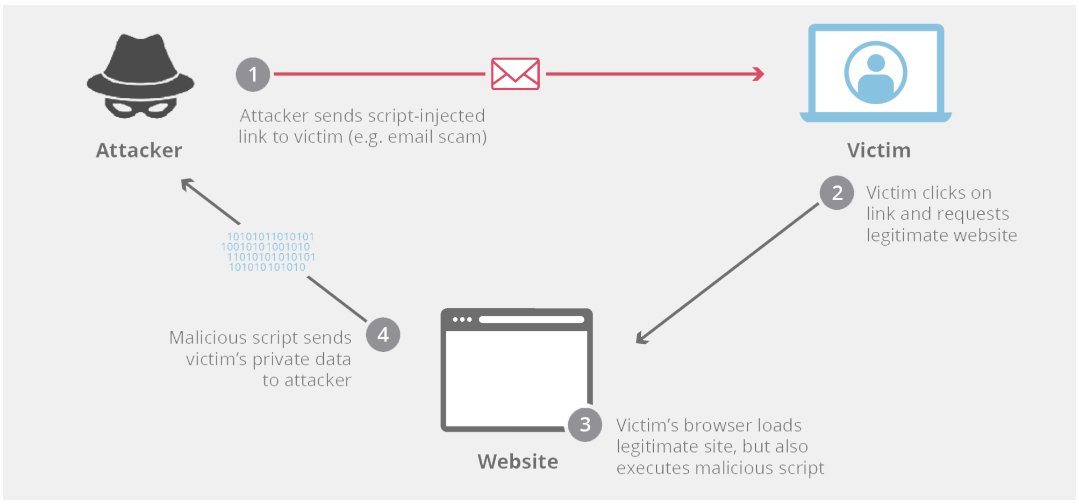
Attacker
1
Attacker sends script-injected link to victim (e.g. email scam)
Victim
2
Victim clicks on link and requests legitimate website
3
Victim's browser loads legitimate site, but also executes malicious script
Website
4
Malicious script sends victim's private data to attacker

Illustration (Cloudflare; n.d.)

Because your web browser has access to sensitive information, such as cookies (which store data like login sessions or user preferences), JavaScript running on a compromised site can be exploited by an attacker to gain access to this information. Once an attacker has access to your browser's data, they can perform a variety of malicious actions, including:

- Redirecting your browser to more dangerous websites, potentially exposing you to further attacks or phishing attempts.
- Executing malicious activities on your computer. If an attacker can run commands on your PC through the browser, they can manipulate your system to perform harmful tasks. For instance, they might use your computer to launch attacks on other machines, effectively making your PC part of a **zombie botnet**. A botnet is a network of compromised computers that can be controlled remotely by the attacker, allowing them to carry out large-scale distributed attacks, such as DDoS (Distributed Denial of Service) attacks.

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 type of XSS occurs when an attacker tricks a victim into clicking on a malicious link. 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. The link usually contains a script embedded in the Uniform Resource Locator (URL) that gets reflected off the server and executed in the victim's browser.

For example, if a user were to click on a link like `https://example.com/search?query=<script>stealCookies()</script>`, then the server might include the injected script in its response. The browser then runs this script, which could steal sensitive data like cookies and send them to the attacker. This type of XSS is called **reflected**, because the malicious code is reflected off a web server to the user's browser.

- **Stored XSS:** In this type, the attacker permanently stores the malicious code on the server, typically in a database, and it is later served to users who visit the affected page. For example, an attacker might inject a script into a comment section on a website. When other users view the page, the malicious script is executed in their browsers. Unlike reflected XSS, stored XSS does not require the victim to click on a link; they are simply affected by visiting the compromised page. For instance, if an attacker injects `<script>alert('Hacked!');</script>` into a comment section on a website and the site fails to sanitise inputs, the script will run each time a user views the comment, potentially leading to data theft or other malicious actions.

For a comprehensive overview of the different types of XSS attacks, please visit: **What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy**

Preventing XSS

Hacking often occurs when user input is improperly handled and executed as code. To prevent this, ensure that user input cannot be interpreted or stored as executable code. For example, a secure website should not allow the storage or execution of HTML tags as code. 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**: To prevent stored XSS, encode outputs so that HTML tags stored in the database are not interpreted as code by the browser. For instance, in **PHP**, the **htmlspecialchars()** function converts characters like `<`, `>`, and `&` into their HTML entity equivalents (`&lt;`, `&gt;`, and `&amp;`), so they are displayed as text rather than executed as HTML.
- **Response headers:** Specify the content type of the response to inform the browser about the type of data being delivered. For example, using the `content-type: text/html` header will tell your browser that the response only includes 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 instance, provide your web server with a list of all trusted sites to pull data from. That way, if someone tries to run a reflected XSS attack, it won't work if the XSS is trying to pull a script from a site that isn't whitelisted.

Injecting code

Code injection is a security vulnerability that occurs when user input is not properly validated and is treated as part of the program's code. This allows an attacker to insert and execute harmful code, potentially compromising the system. In essence, code injection lets someone run unauthorised commands within a program, which can lead to serious security issues. A common scenario where code injection can happen is with the **eval()** function. The `eval()` function in programming takes a string of code as input and executes it. While `eval()` can be useful, it becomes dangerous if it executes code that includes unvalidated user input.

For example:

```
print(eval("1 + 1"))
```

In this example, `eval("1 + 1")` runs the string `"1 + 1"` as code, which outputs `2`. However, if a program uses `eval()` to process user input, it could execute anything the user types, including harmful code.

Websites often process user input in ways that can be vulnerable to code injection. Consider the following instance, where a simple calculator is built using user inputs:

```
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 calculator allows users to input two numbers and an operator (like `+`, `-`, `*`, or `/`) to perform basic arithmetic. The inputs are combined into a string, which `eval()` then executes as code.

However, this method is vulnerable to code injection. For instance, if a user enters the following:

```
first_number = fetch_valuable_data()

second_number = ""

operator = ""
```

The resulting code executed by `eval()` would be:

```
# This runs: fetch_valuable_data()

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

In this context, `fetch_valuable_data()` represents a function that could be defined by an attacker to steal sensitive information, such as user passwords, credit card numbers, or other confidential data. By injecting this function into the program, the attacker can trick the program into running it, potentially exposing or compromising important data.

For example:

```
def fetch_valuable_data():

  return "Sensitive data has been accessed!"
```

If `fetch_valuable_data()` is defined in the program, it could be run by the `eval()` function, potentially leading to security breaches.

The **exec()** function in Python is even more powerful than `eval()`, because it can execute multiple lines of code, not just a single expression. Let’s consider a scenario where a program is designed to square a number provided by the user:

```
first_number = input("Enter a number to be squared: ")

exec(f"result = {first_number}**2")

print(result)
```

This program asks the user for a number, squares it, and then prints the result. For instance, if the user inputs `2`, the program will output `4`. However, if a malicious user inputs `fetch_valuable_data()`, the program will try to execute:

```
exec(f"result = fetch_valuable_data() ** 2")
```

This would raise an error, because `fetch_valuable_data() ** 2` is not valid. But an attacker could exploit how comments work in Python to bypass this restriction. For example, by entering:

```
Enter a number to be squared: fetch_valuable_data(); #
```

This input would result in:

```
result = fetch_valuable_data(); # ** 2

print(result)
```

The `fetch_valuable_data()` function would execute, and the comment (`#`) would effectively disable the rest of the code that squares the number.

This demonstrates how an attacker can use code injection to run unauthorised code. The use of `exec()` is dangerous when handling untrusted input, because it can execute arbitrary code, leading to serious security vulnerabilities.

Take note

Websites that display errors in response to user input can be valuable targets for hackers. When scanning for vulnerabilities, hackers often test web addresses containing comment characters or executable code. If this input causes the website to throw an error, it usually indicates that the input was interpreted as code. This encourages hackers to continue probing the site, searching for weaknesses until they successfully execute malicious code. To protect against this, it's important to design websites that do not expose errors based on user input, as this can help conceal potential vulnerabilities.

Take note

For this task, you will be injecting code using input. Open and read the contents of **example.txt**, which will form part of the practical 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. You will also need to use the scripts inside **Python Scripts**.

Practical task

- 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.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Share your thoughts

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

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

Share your thoughts anonymously using this **form**.

Reference list

Basatwar, G., and Global Business Head (2023, October 7). *Types of cyber attacks: An in-depth guide on the top 7 cyber attacks*. Appsealing.
https://www.appsealing.com/types-of-cyber-attacks/

Cloudflare. (n.d.). *What is a phishing attack?* Cloudflare.
https://www.cloudflare.com/en-gb/learning/access-management/phishing-attack/