

# Cross Site Scripting (XSS)

Ilgaz Senyuz, Okan Yildiz

December 13, 2022

## **Abstract**

With the increased number of web applications, web security is becoming more and more significant. Cross-Site Scripting vulnerability, abbreviated as XSS, is a common web vulnerability. Exploiting XSS vulnerabilities can cause hijacked user sessions, malicious code injections into web applications, and critical information stealing. This article gives brief information about XSS, discusses its types, and designs a demo website to demonstrate attack processes of common XSS exploitation scenarios. The article also shows how to prevent XSS attacks with code illustrations.

# Contents

<b>1</b>	<b>Introduction to Cross Site Scripting (XSS)</b>	<b>3</b>
<b>2</b>	<b>XSS Attack Types</b>	<b>4</b>
2.1	Reflected XSS . . . . .	4
2.2	Stored XSS . . . . .	5
2.3	DOM-based XSS . . . . .	6
<b>3</b>	<b>Risks Caused by XSS</b>	<b>7</b>
3.1	Session Hijacking . . . . .	7
3.2	Capturing Keystrokes . . . . .	7
3.3	Compromising the Security of a Website . . . . .	8
<b>4</b>	<b>How to Exploit XSS Vulnerabilities</b>	<b>8</b>
4.1	Reflected XSS . . . . .	8
4.2	Stored XSS . . . . .	13
4.3	DOM-based XSS . . . . .	18
<b>5</b>	<b>How to Prevent XSS Attacks</b>	<b>20</b>
<b>6</b>	<b>Attempt to Exploit More Secure Code</b>	<b>29</b>

# 1 Introduction to Cross Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection in which malicious scripts are injected into usually harmless and trusted websites. These scripts are executed by unsuspecting users who visit the compromised website or interact with the web application. The end user's browser has no way to know that the script should not be trusted and executes the malicious script. Because it thinks the script is from a trusted web application, the malicious script could gain access to cookies, session tokens, or other sensitive information held by the browser and use them however the cyber-attacker intends to.

Exposure to these attacks may occur wherever a web application uses input from a user without properly validating or sanitizing it. XSS attacks are one of the most common web-based attacks and yet, almost 65% of the web applications are guessed to be XSS vulnerable since web developers may not be aware of the risks of XSS and may not take the necessary steps to protect their applications from this type of attack.

XSS attacks can be used to steal sensitive information, such as passwords and credit card numbers, or to perform unauthorized actions on a website, such as modifying or deleting data. In addition to the risks to users, XSS attacks can also damage a website's reputation and credibility, which can be difficult to recover from.

## 2 XSS Attack Types

There are three main types of XSS attacks: reflected, stored, and DOM-based. Each type of attack has its own unique characteristics and can be exploited in different ways.

### 2.1 Reflected XSS

Reflected XSS is the most common type of XSS attack. It is typically encountered when user input is displayed on the screen without undergoing any validation. In other words, it occurs when user input is directly echoed back to the user without being properly sanitized.

In a reflected XSS attack, an attacker injects a malicious script into a vulnerable web page. This script is then executed when a user accesses the page, allowing the attacker to steal sensitive information or perform other harmful actions.

To carry out a reflected XSS attack, the attacker must first find a vulnerable web page that allows user input without proper validation and sanitization. They can then craft a URL that includes their malicious script and send it to potential victims through email, social media, or other means. If a victim clicks on the malicious link, the injected script will be executed in their browser, allowing the attacker to access sensitive information or perform other harmful actions.

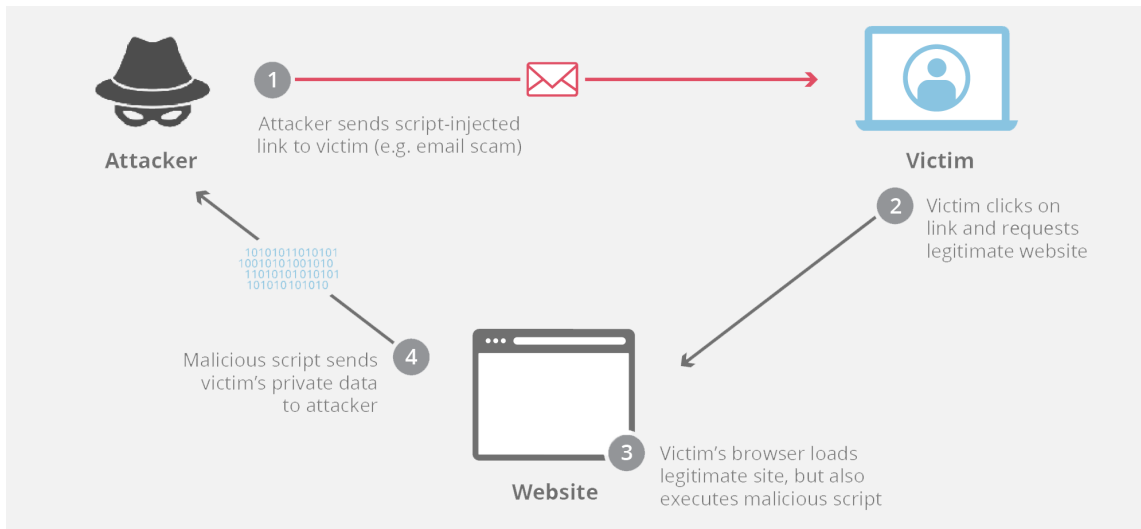


Figure 1: Reflected XSS Scenario

## 2.2 Stored XSS

Stored XSS attacks occur when an attacker injects malicious code into a website's database or other persistent storage. This code is then executed whenever an innocent user accesses the relevant section of the website, allowing the attacker to steal sensitive information or perform other malicious actions.

Stored XSS attacks are particularly dangerous because the code is stored on the website and can be executed by multiple users over a long period of time. Unlike reflected XSS, it may reach every user without phishing attacks. This can allow the attacker to steal a large amount of sensitive information or to perform a large number of unauthorized actions. These attacks can be difficult to detect, as the malicious code is stored on the website's server.

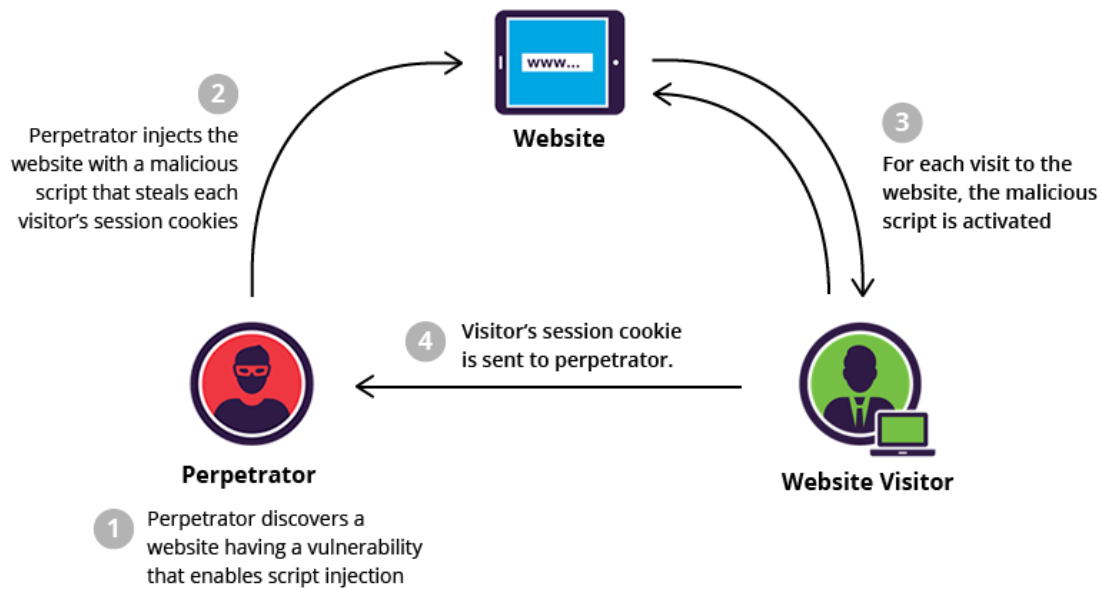


Figure 2: Stored XSS Scenario

## 2.3 DOM-based XSS

DOM is an internal data structure that stores all objects and properties of a web page. For example, each tag used in HTML code represents a DOM object. Additionally, it contains information about properties such as a web page's DOM, page URL, and meta information. Using JavaScript, developers can reference these objects and properties and change them dynamically.

The Document Object Model is what enables dynamic, single-page applications. However, this is what makes DOM-based cross-site script-

ing possible.

DOM-based XSS attacks can be either DOM-based Reflected XSS or DOM-based Stored XSS attacks.

## **3 Risks Caused by XSS**

XSS attacks can have serious consequences for both website owners and users. Some of the risks associated with XSS include:

### **3.1 Session Hijacking**

One of the most common and dangerous risks associated with XSS attacks is session hijacking. This occurs when an attacker is able to gain access to a user's web browser session, allowing them to access sensitive information or perform unauthorized actions on the user's behalf. This can be particularly damaging for websites that handle sensitive information, such as financial or medical data, as the attacker can steal this information or use it to perform fraudulent transactions.

### **3.2 Capturing Keystrokes**

XSS attacks can also be used to capture a user's keystrokes, allowing the attacker to steal sensitive information, such as passwords and credit card numbers. This can be done by using a malicious script to record the user's keystrokes and then transmitting that information to

the attacker. This can be particularly dangerous, as users may not be aware that their keystrokes are being recorded, and they may continue to use the website or web application without realizing that their sensitive information has been stolen.

### 3.3 Compromising the Security of a Website

XSS attacks can also be used to compromise the security of a website, allowing the attacker to gain access to sensitive information or make other changes to the web application. This may disrupt the normal functioning of a website, which can lead to a poor user experience and damage the reputation of the affected web page.

## 4 How to Exploit XSS Vulnerabilities

### 4.1 Reflected XSS

```
def xss(request):  
    if request.method == 'GET':  
        search_query = request.GET.get('search', None)  
        return render(  
            request,  
            'xss.html', {'search': search_query })  
    elif request.method == 'POST':
```

Figure 3: Back-end code

Without further explanation, let's look at the back-end code of a search page. This code block takes the value of the search parameter



of the GET method and passes it to the template file as a parameter so that it can be used when rendering the `xss.html` template file. The template is a special concept in Django that allows us to use the Django Template language in an HTML file, but I do not want to distract you with Django syntax. Currently, this code only prints the search on the website, it does not provide any search function, we will come to that later.

```
{% if search%}
  {% block messages %}
    <div class="col-md-6">
      <p class="mb-0" id="searchQuery"> You searched for {{ search }}</p>
    </div>
  {% endblock %}

{% else %}
  <div class="col-md-6">
    <p class="mb-0" id="searchQuery"></p>
  </div>
{% endif %}
```

Figure 4: xss.html code

This is a code snippet from the xss.html file where the searched word is printed to the user. If the search parameter exists, the search value is displayed on the screen. Otherwise, an empty div element is rendered.

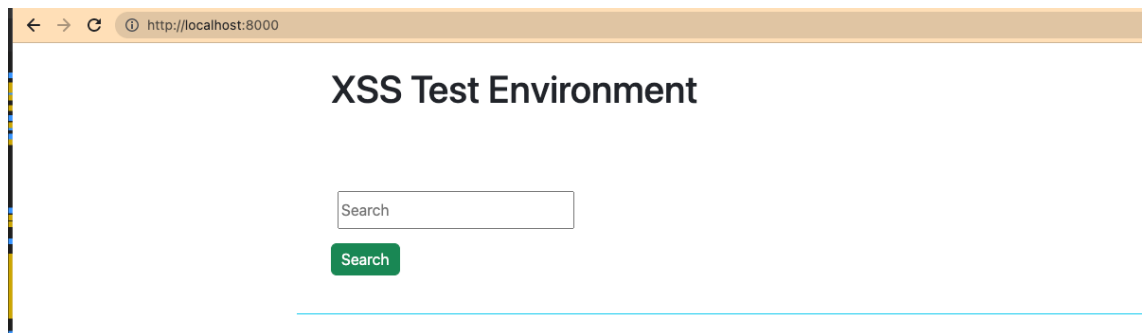


Figure 5: Search Page

Our web page looks like this. We know that the application prints the searched word directly on the page, but let's test it. The `< b >` tag is an HTML tag used to make words bold. Let's see if the tag will maintain its function. Yes, the sentence inside the '`< b >`' tag was



Figure 6: Observing HTML tags are directly echoed

printed in bold. Now let's write an alert script into the input field. The alert function is very easy to test because it displays a warning on the screen when it is executed. Therefore, it is often used to test for XSS vulnerabilities.

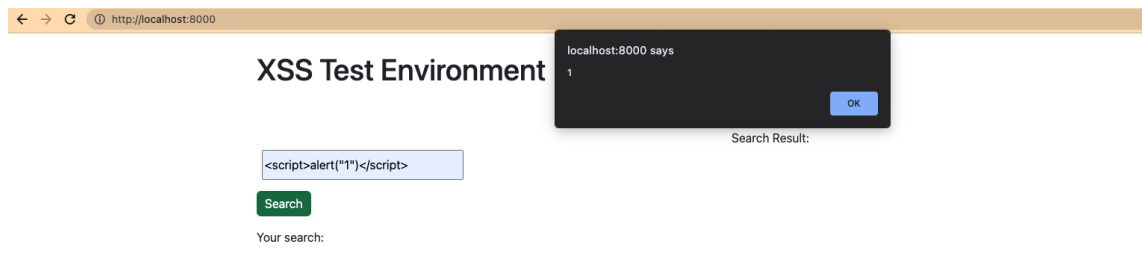


Figure 7: Executing alert function

As you can see, the alert function also worked successfully. Our web page looks quite insecure. Now, for the attacker to complete the attack, they need to craft a URL that will execute the script they want when the user clicks on the link. Since we have already seen the back-end code, we know that the parameter name is 'search', but let's look at how someone who knows nothing about the source code can prepare it. We go to the network tab of the browser's 'developer tools' settings. The network tab only starts recording after it is opened so it is important to open the network tab first. Then we search for any word using the search bar and we see that a query goes to the backend with a

```
{http:///localhost:8000/?search=my_search_input}
```

request. As you can notice, after a question mark, 'search' and equal sign, the request is followed by the word we searched for. This means that if we prepare such a URL by hand, the script written in it will be executed when the URL is visited by an end-user.

We changed the URL a little bit and it is ready:

```
http://localhost:8000/?search=<script>alert("1");</script>
```

With the URL we prepared, let's test it right away.

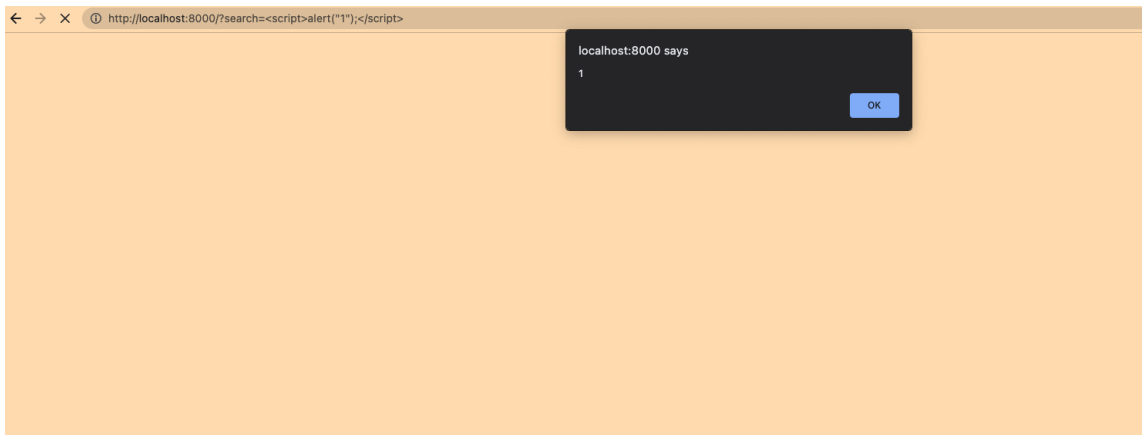


Figure 8: Executing alert function from URL

Bingo! The attacker has everything they need to initiate a reflected XSS attack. Unlike our harmless script, their script won't execute the alert script for sure! Since the end-users browser will think that this script is started by the website it trusts, it will have access to everything, such as the user's cookies. All that remains is to distribute this URL to people in different ways, but it isn't our topic today.

## 4.2 Stored XSS

We are here again with the same web page we used in the reflected XSS demonstration. This time there is a section for the user to register. The search button also searches users by username and now prints the results.

← → ↻ http://localhost:8000

## XSS Test Environment

Search Result:

test

Search

Your search: test

---

Enter a username and password to register

Username Password

Save

---

Figure 9: Web Page for Stored XSS Demonstration

```

elif request.method == 'POST':
    username = request.POST["username"]
    password = request.POST["password"]

    try:
        user = Person(username = username, password = password)
        user.save()
        return HttpResponse(status = 200)

    except ConnectionError:
        return HttpResponse(
            """
            Registration failed.
            """,
            status=500,
        )

```

Figure 10: User Register Implementation

As you can see, username and password attributes from the post request are retrieved and directly inserted into the user table in the database. Let's verify that registration is XSS prone by testing it with an illegal username and then checking the database.

---

Enter a username and password to register

---

Figure 11: Stored XSS Injection

	id [PK] bigint	username character varying (100)	password bytea
1	13	<script> alert("1");</script>	[binary data]

Figure 12: Verifying Injected Username

What a shame! The input taken from the user has been directly transferred to the database as it is. Before checking if retrieving this user via the search button executes the injected script or not, let's examine the corresponding code snippets.

```
def search(request):
    if request.method == 'GET':
        query = request.GET["search"]
        users_list = manager.get_queryset().filter(username__contains = query).values('username')
        res = []
        for user in users_list:
            username = user['username']
            res.append([username + " "])
        return HttpResponse(res, status = 200)
```

Figure 13: Implementation of User Retrieving

Usernames are fetched from the database and displayed in the browser without going through any validation.



```

$.ajax({
  type: 'GET',
  url: '/xss/',
  data: {
    search:$('#search').val(),
  },
  success: function(data){
    $('#searchResult').html("Search Result: " + data);
  }
})

```

Figure 14: Implementation of User Displaying

Now let's verify the injected XSS script is being executed while retrieving usernames using the search button.

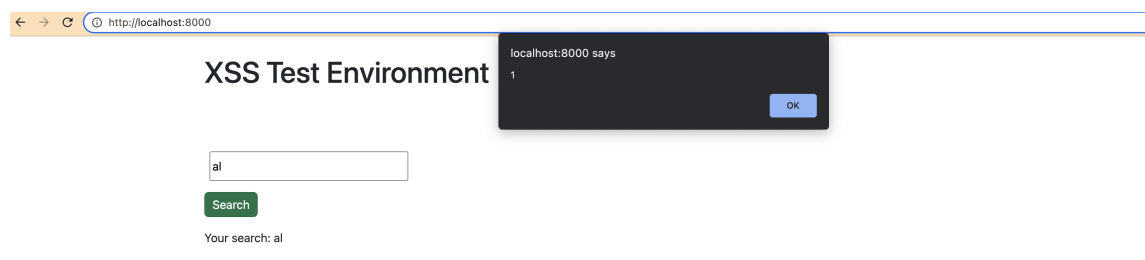


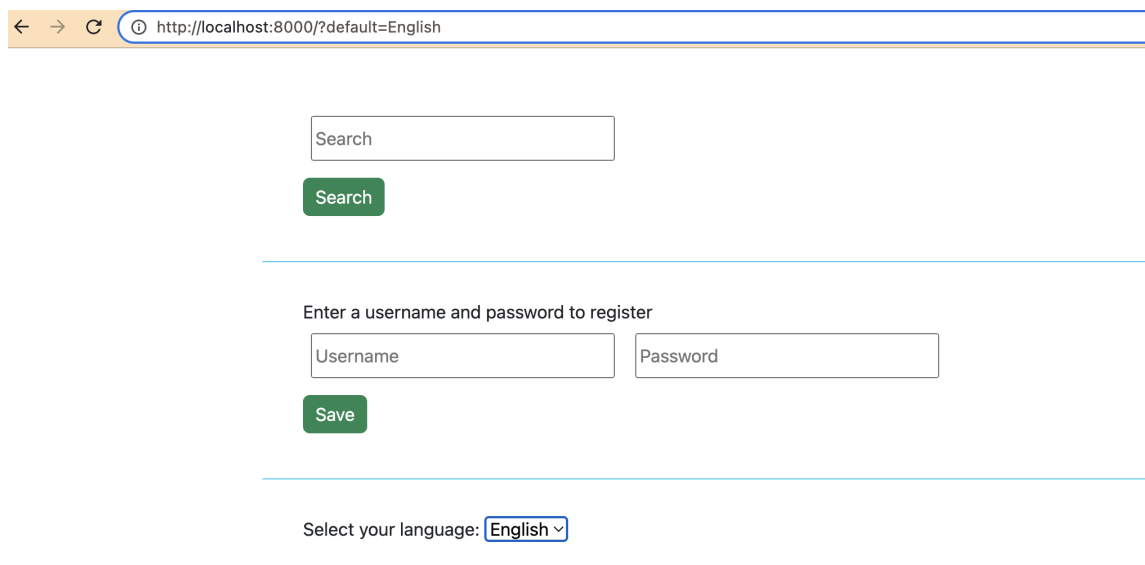
Figure 15: Verifying Injected Username

Bingo! We searched for the keyword "al". Then the search function got the corrupted username because it contained the string "al" and the browser tried to print it. However, no precaution was taken against XSS attacks and the potentially malicious script was executed.

Stored XSS attacks are much more dangerous than reflected XSS attacks. Imagine the most popular social media platform and assume that one of its usernames contains a malicious script. Each user's

browser will unwittingly run the dangerous script in the stored username. The consequences would be a real disaster.

## 4.3 DOM-based XSS



The screenshot shows a web browser window with the address bar displaying `http://localhost:8000/?default=English`. The page content is divided into three sections by horizontal lines. The first section contains a search bar with the placeholder text "Search" and a green "Search" button below it. The second section contains the text "Enter a username and password to register" followed by two input fields labeled "Username" and "Password", and a green "Save" button below them. The third section contains the text "Select your language:" followed by a dropdown menu currently showing "English".

Figure 16: Web Page with Select Language Component

As I mentioned earlier, there exists both reflected and stored DOM-based XSS attacks. Today we will examine the reflected DOM-based XSS attacks. Our default web page is back again, with a slight change. Now there is a new language selection section. Let's inspect the source code of this language component.

```

Select your language:

<select><script>
    var document: Document
    document.write("<OPTION value=1>" + decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=") + 8)) + "</OPTION>");
    document.write("<OPTION value=2>English</OPTION>");
    document.write("<OPTION value=2>French</OPTION>");
    document.write("<OPTION value=2>Deutsch</OPTION>");
</script></select>

```

Figure 17: Source Code of Language Component

Briefly, this code gives the user choice to select from 3 languages. The decoded value of the default query parameter in the current URL is selected by default. This value is obtained by using the substring and indexOf methods to extract the substring of the URL that comes after "default=", and then using the decodeURIComponent function to decode the URL-encoded value.

The first line of code is potentially unsafe because it uses the document.write method to insert content into the current web page. This can be a security issue since it allows an attacker to potentially inject malicious code into the page that could be executed by other users who visit the site. Additionally, the decodeURIComponent function is used to decode the URL of the current page, which could allow to manipulation of the URL in such a way as to inject malicious code into the page. So let's manipulate it!

Since the value of the default parameter in the URL is created as a DOM element in the HTML page, we are going to replace the value English with a script of our choice.

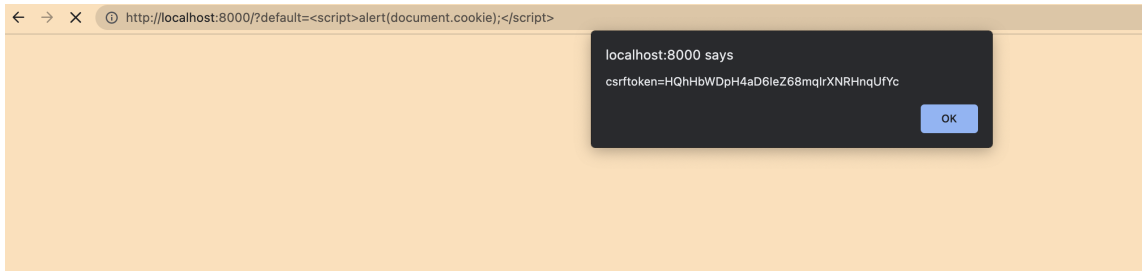


Figure 18: DOM-based XSS

It seems worked! Since it is a DOM-based reflected XSS attack, the attacker must need to distribute this URL after this step. Since DOM-based is a very general name, basically anything caused by DOM, there can be limitless examples of this type of attack. Any modified or rendered DOM element from user input without validating and sanitizing may result in such vulnerability.

## 5 How to Prevent XSS Attacks

In this section, we will discuss how to prevent XSS attacks, mention general prevention techniques, and finally fix the codes we exploited.

- Input should be validated as strictly as possible on arrival, given the kind of content that it is expected to contain. However, input validation is not a foolproof method for preventing cross-site scripting (XSS) attacks, as it only checks user-provided input to ensure that it meets certain criteria and is in the expected format. It does not guarantee that an attacker will not be able to inject malicious code into your web application. Therefore, while input validation is a useful security measure, it should not be relied upon as the only method for preventing XSS attacks.

- Output Encoding should be used when you need to safely display data exactly as a user typed it in. Variables should not be interpreted as code instead of text.

Start with using your framework's default output encoding protection when you wish to display data as the user typed it in. Automatic encoding and escaping functions are built into most frameworks.

If you're not using a framework or need to cover gaps in the framework then you should use an output encoding library. Each variable used in the user interface should be passed through an output encoding function. A list of output encoding libraries is included in the appendix.

There are many different output encoding methods because browsers parse HTML, JS, URLs, and CSS differently. Using the wrong encoding method may introduce weaknesses or harm the functionality of your application.

- Output encoding is not sufficient either. It will not always prevent XSS and these locations are known as dangerous contexts. Dangerous contexts include:

```
<script>Directly in a script</script>
<!-- Inside an HTML comment -->
<style>Directly in CSS</style>
<div ToDefineAnAttribute=test />
<ToDefineATag href="/test" />
```

Other areas to be careful of include Callback functions where

URLs are handled in code such as:

```
CSS { background-url : \javascript:alert(xss)"; }
```

All JavaScript event handlers:

```
(onclick(), onerror(), onmouseover()).
```

Unsafe JS functions like `eval()`, `setInterval()`, `setTimeout()`

Don't place variables into dangerous contexts as even with output encoding since it will not prevent an XSS attack fully.

- The most effective way to avoid DOM-based cross-site scripting vulnerabilities is not to dynamically write data from any untrusted source into the HTML document. If the desired functionality of the application means that this behavior is unavoidable, then defenses must be implemented within the client-side code to prevent malicious data from introducing script code into the document. In many cases, the relevant data can be validated on a whitelist basis, to allow only content that is known to be safe. In other cases, it will be necessary to sanitize or encode the data. This can be a complex task, and depending on the context that the data to be inserted may need to involve a combination of JavaScript escaping, HTML encoding, and URL encoding, in the appropriate sequence.
- Use appropriate response headers. Response headers such as Content-Type and X-Content-Type-Options can help prevent cross-site scripting (XSS) attacks by instructing the browser how to handle certain types of data. The Content-Type header tells the

browser the media type, or MIME type, of the resource being sent, while the X-Content-Type-Options header prevents MIME sniffing, which can be used by attackers to inject malicious code. By setting these headers appropriately, you can ensure that the browser handles the data it receives in an intended way and does not execute any malicious code.

- The Content-Security-Policy (CSP) response header is used to specify which domains are allowed to serve resources, such as JavaScript, CSS, and other types of media, to the browser. This can help prevent attackers from injecting malicious code into a web page by serving it from a different domain.

For example, if a web page only uses JavaScript from the domain example.com, the Content-Security-Policy header could be set to `script-src example.com` to instruct the browser to only execute JavaScript from that domain. Any attempts to execute JavaScript from a different domain would be blocked by the browser.

In addition to preventing XSS attacks, the Content-Security-Policy header can also be used to prevent other types of attacks, such as clickjacking and cross-site request forgery (CSRF). By carefully configuring the Content-Security-Policy header, you can help protect your web site from a variety of attacks.

Additionally, website owners and developers should keep their software and applications up to date, as new vulnerabilities are often discovered and addressed in updates. This can help to prevent XSS attacks and other security vulnerabilities by ensuring that the website or web application is using the latest security patches and fixes.

Now let's try to make the codes we exploited more secure against XSS attacks:



```

Select your language:

<select><script>
    var document: Document
    document.write("<OPTION value=1>" + decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=") + 8)) + "</OPTION>");
    document.write("<OPTION value=2>English</OPTION>");
    document.write("<OPTION value=2>French</OPTION>");
    document.write("<OPTION value=2>Deutsch</OPTION>");
</script></select>
</div>

```

Figure 19: Language Selection

```

Select your language:

<select>
  <script>
    var defaultLanguage = decodeURIComponent(document.location.href.substring(document.location.href.indexOf("default=") + 8));
    var encodedLanguage = escape(defaultLanguage);
    document.write("<OPTION value=1>" + encodedLanguage + "</OPTION>");
    document.write("<OPTION value=2>English</OPTION>");
    document.write("<OPTION value=2>French</OPTION>");
    document.write("<OPTION value=2>Deutsch</OPTION>");
  </script>
</select>
</div>
<div class="col-md-6">

```

Figure 20: More Secure Language Selection

We simply used Javascript's built-in escape function which replaces possible dangerous characters with escape sequences. Escape sequences are good examples of sanitizing.

```
def search(request):
    if request.method == 'GET':
        query = request.GET["search"]
        users_list = manager.get_queryset().filter(username__contains = query).values('username')
        res = []
        for user in users_list:
            username = user['username']
            res.append(username + " ")
        return HttpResponse(res, status = 200)
```

Figure 21: Get Users Function

```
def search(request):
    if request.method == 'GET':
        query = request.GET["search"]
        users_list = manager.get_queryset().filter(username__contains = query).values('username')
        res = []
        for user in users_list:
            username = user['username']
            escaped_username = escape(username)
            res.append(escaped_username + " ")
        return HttpResponse(res, status = 200)
```

Figure 22: More Secure Get Users Function

We used Django's built-in escape function to sanitize input.

```
def search(request):
    if request.method == 'GET':
        query = request.GET["search"]
        users_list = manager.get_queryset().filter(username__contains = query).values('username')
        res = []
        for user in users_list:
            username = user['username']
            res.append(username + " ")
        return HttpResponse(res, status = 200)
```

Figure 23: Search Function

```
def xss(request):
    if request.method == 'GET':
        search_query = request.GET.get('search', None)
        # Use the .escape() function to encode any potentially dangerous characters in the search query
        search_query = escape(search_query)

        return render(
            request,
            'xss.html',
            {'search': search_query }
        )
    elif request.method == 'POST':
```

Figure 24: More Secure Search Function

Again we used Django's escape function.

```
searchQuery = $('#search').val()
$('#searchQuery').html("Your search: " + searchQuery);
```

Figure 25: Displaying Searched Keyword

```
// Use a regular expression to only allow alphanumeric characters and spaces
var regex = /^[a-zA-Z0-9\s]+$/;

// Get the search query from the input field
var searchQuery = $('#search').val();

// Validate the search query using the regular expression
if (regex.test(searchQuery)) {
    // Set the value of the #searchQuery element using the validated search query
    $('#searchQuery').text("Your search: " + searchQuery);
} else {
    // If the search query is invalid, show an error message
    $('#searchQuery').text("Error: Invalid search query");
}
```

Figure 26: Displaying Searched Keyword Securely


We first validated the user input with regex, and then instead of directly rendering an HTML element with user input, we use the `text()` function.

Please mind that we took only very simple measures. To ensure full security against XSS, all the steps we described at the beginning of this section must be provided altogether.

## 6 Attempt to Exploit More Secure Code

Since we made our implementations more secure against XSS attacks, now it's time to check the scenarios we apply while exploiting XSS vulnerabilities in section 4.

Checking our search bar against reflected XSS first:



`<b> this text should not be bold. </b>`



Search

Error: Invalid search query

---

Figure 27: Displaying Searched Keyword Securely

After the HTML tag is detected in the search query, the search keyword is not echoed back.

---

al

Search

Your search: al

Search Result: <script> alert("1");</script>

---

Figure 28: Retrieving Users Securely

Remember that we had a user with a username containing a script. We searched for this user and their username is printed, but thanks to our sanitization it is unexecutable by the browser and alert functions didn't execute.

http://localhost:8000/?default=<script>alert(document.cookie);</script>

Search

Search

You searched for None

---

Enter a username and password to register

Username

Password

Save

---

Select your language:  
%3Cscript%3Ealert%28document.cookie%29%3B%3C/script%3E

---

Figure 29: DOM-based XSS Check

And our final attempt fails too. We provided a script as a value of the default language parameter. But as you see our escape imple-

mentation replaces the possibly dangerous HTML tags with escape sequences and once again the alert function didn't execute.

## References

- Omer Citak, Ethical Hacking- Offensive and Defensive
- <https://www.imperva.com/learn/application-security/cross-site-scripting/>
- <https://owasp.org/www-community/attacks/xss/>
- <https://portswigger.net/web-security/cross-site-scripting/>
- [https://cheatsheetseries.owasp.org/cheatsheets/Cross\\_Site\\_Scripting\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Cheat_Sheet.html)