Cross Site Scripting (XSS)

White Hats: Team Members

Banks Cargill - cargillb@oregonstate.edu

Frederick Eley - eleyf@oregonstate.edu

Timothy Glew - glewt@oregonstate.edu

Description

Cross-site Scripting (XSS) is an attack where the attacker attempts to inject malicious code into an otherwise trustworthy website. The attacker's goal is to use the web application as a vehicle to send the malicious code to users where their browsers will execute it. In this way, XSS attacks are different from many of the other attacks discussed since the attack does not directly target the web application, but rather uses it to attack other users. This vulnerability can exist on sites where a user's input is used in creating the output of the page without appropriately validating it. Once the script has been injected, it can be run in another user's browsers since there is no way for the browser to determine that the script is malicious as it is coming from an otherwise trusted source. Additionally, this means the script will have access to any information stored by the browser and used on the site including cookies, session tokens, and other sensitive information. The three main types of XSS attacks are stored XSS attacks, reflected XSS attacks, and DOM-based XSS attacks, with each covered in detail.

Reflected XSS Attacks

A reflected XSS attack is an attack where the malicious code is inadvertently sent to the web server by the user where it is then reflected back to the user through any response by the web server that includes input from the user. Typically, this occurs in situations when the page takes a parameter from the user's request and renders the user's text back in its response. This can happen in instances such as error messages or search results where the information used to render the page comes partially from the URL. To get a user to send the malicious code to the server, the attacker may attempt to trick the victim into clicking a link that submits a form or request to the web server of a trusted site. Because the response that was reflected is coming from a trusted site, the browser will execute the code.

Stored XSS Attacks

A stored XSS attack can occur when a web application stores users input in a database and later serves that information to other users. For example, a blog site allows users to type out a blog post that is then saved to a database and served to other users who browse the site. If an attacker includes malicious code in their blog post it could be executed on another users' browser when it is served to them as they view the blog post. This is particularly dangerous because a user's browser may execute the malicious script simply by visiting the infected page compared to a reflected XSS attack which typically relies on using social engineering tactics to trick a user into clicking a link.

DOM-based XSS Attacks

Document Object Model (DOM) based XSS attacks aim to use DOM objects to create an XSS attack. The DOM is a convention used for HTML documents that allows programs to dynamically update the structure, style, and content of a page, but can be used to create XSS attacks. For example, if a web application has a search feature and sets the title of the page to be the search term that is also encoded in the URL, the attacker could include “<script>” tags in the URL so that when the site is building the search results page, it encodes the malicious script into the page. DOM-based XSS attacks use a mechanism that is different from the other two types of XSS attacks. With stored and reflected XSS attacks, the application takes user input and renders it to users in an unsafe way. This is different from DOM-based XSS attacks where the server’s response does not include the malicious code, instead it is when the browser is processing the response that the script is executed.

The consequences of each type of XSS attack are essentially the same, since each one can be used to execute an arbitrary script in the user’s browsers. The difference between the three is simply how the attacker gets the script to be run in the user’s browser.

XSS attacks rely on the ability for untrusted scripts entered as user input to be run in the browser. Thus, prevention methods to protect against all three types of XSS look to do the same thing, sanitize user input so that even if an attacker attempts to inject malicious code, it will not be executed. To do this, anywhere in your site where you are using user input when creating the web page served to the user, you must use the escape syntax for that part of the HTML document. The escape syntax is different for different parts of the page. For more detail on specific escape syntax, OWASP provides a XSS prevention cheat sheet that when implemented protects against XSS attacks:
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

How to Attack our Site

Reflected XSS Attacks

Since our site does not directly use user input to dynamically render any of pages (user input is stored in our database and then pulled later to render our pages) as would be the case with something like a search feature that shows “Search results for ‘[user input’]”, our site is not vulnerable to being used as a vehicle for a reflected XSS attack. With that said, in the “How to Defend our Site”, we discuss the measures taken to protect our site should a feature like the search example described above be implemented.

Stored XSS Attacks

Our site does not store any user input that is then shown to other users like would be the case on a website that has forum posts or commenting, thus, a stored XSS attack to compromise another user is not feasible. However, we can still demonstrate how a stored XSS attack would be conducted, and how it could be used if our site was structured differently.

To perform a stored XSS attack we can take advantage of the fact that a user can enter anything they like for names and descriptions of lists and tasks, and these inputs are pulled from the database when showing this information to users. Thus, when entering text into these form inputs we can include a script that will then be run when the page is rendered to show our entry. As an example, create a new list and set the list name to (the description can be anything):

```
<script>alert("XSS attack in progress");</script>XSS List
```

This will store our script along with the name of our list so that when the page is rendered again to show our new list, we will see our list with the name “XSS List”, but the script we added will be embedded in the page and run, so before seeing our page render we will see an alert with our message “XSS attack in progress”. You can change the script to run anything you would like. If our site had a functionality where other user’s lists could be viewed publicly (e.g. a “popular lists” section to share lists), the script that we wrote into the name field would be run in the browser of any user who loads it.

DOM-based XSS Attacks

Somewhat like reflected XSS attacks, our site is not vulnerable to DOM-based XSS attacks since we do not directly write to the DOM (user input makes its way into the DOM by being pulled from our database).

How to Defend our Site

To protect our site against XSS attacks of all types we need to make sure that we are properly sanitizing any user input that is used in rendering pages on the server side (to protect against stored and reflected XSS) as well as user input that will be processed on the client side (to protect against DOM-based XSS attacks.

Specific to our site, we sanitize user input by configuring Jinja2 (the templating engine we use with Flask) to automatically escape everything that is used when generating HTML from our template pages. This means that unless we specifically designate part of our template to *not* escape a variable, it is escaped. This protects us from XSS and is how we implemented the defense on our secure site so that attempting the XSS attack described above no longer works. Instead, the script we pass in as the list name will create a new list whose name is the script string.

Automatic escaping is the default configuration for Jinja2 when using Flask; however, if using Jinja2 without flask, the default configuration requires manual escaping, so you must either reconfigure to automatically escape everything yourself, or manually escape the variables that might contain malicious code. To manually HTML-escape with Jinja2, we pipe the input through the ‘e|’ filter. So, if we were rendering a list’s name and are concerned it could have malicious code, instead of rendering as:

```
{{ context.list_name }}
```

We would render as:

```
{{ context.list_name | e }}
```

Additionally, if we decide to configure Jinja2 to automatically escape everything, but there is a variable where we do not want it to be HTML-escaped, we can pipe it through the ‘|safe’ filter. So, if we decide the list_name is not a risk and do not want it to be HTML-escaped, to disable automatic escaping for the single variable we would render as:

```
{{ context.list_name | safe }}
```

References

https://owasp.org/www-community/attacks/xss/

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

https://www.veracode.com/security/xss

https://www.acunetix.com/websitesecurity/cross-site-scripting/

https://developer.mozilla.org/en-US/docs/Learn/Forms/Sending_forms_through_JavaScript

https://jinja.palletsprojects.com/en/2.10.x/templates/#html-escaping