Application Security (apsi)

Lecture at FHNW

Lecture 7, 2020

Arno Wagner, Michael Schläpfer, Rolf Wagner

<arno@wagner.name>, <{michael.schlaepfer, rolf.wagner}@fort-it.ch>

Agenda

- 12:15 – 13:00: Introduction OWASP
- 13:15 – 14:00: SOP / CORS & CSP
- 14:15 – 15:00: Exercise Session

OWASP – Open Web Application Security Project

Source: OWASP

- Online community producing freely-available articles, methodologies, and tools for web application security
- Founded in 2001 (non-profit organization)
- 32’000+ Volunteers worldwide
- Numerous local chapters (e.g., Switzerland)
- Talks and conferences

Most prominent products:

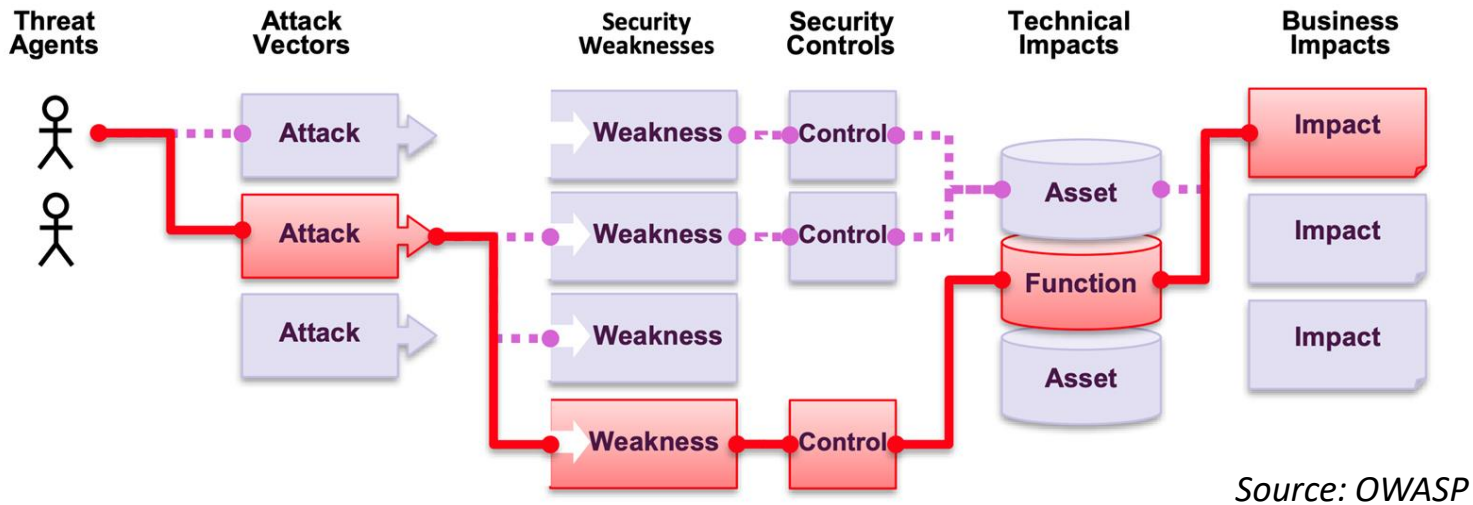
- OWASP Top Ten: Most critical security risks to web applications
- OWASP Zed Attack Proxy (ZAP): Web App Scanner
- OWASP WebGoat: Insecure application (Learn to Hack)

OWASP Top Ten

- First edition In 2003, updated every couple of years (current version is 2021)
- Powerful und widely adopted **awareness information** for web application security e.g. WAFs implement protection mechanisms based on this
- Catalogue of description and countermeasures for the **most critical Applicatoin Security Risks**
- Mix of root causes (Broken Access Control) and symptom (e.g. Sensitive Date Exposure) 2021 tries to focus more on root causes
- Usage
  - ✓ Awareness
  - ✓ Categorization of found vulnerabilities
  - ✗ Not necessarily easily testable issues (e.g. Insecure Design)

OWASP Top Ten

- So, what are the concrete **Application Security Risks** in your company?
- It depends ;-), not least on business impact!
- **Each of these paths** represents a (lower or higher) risk

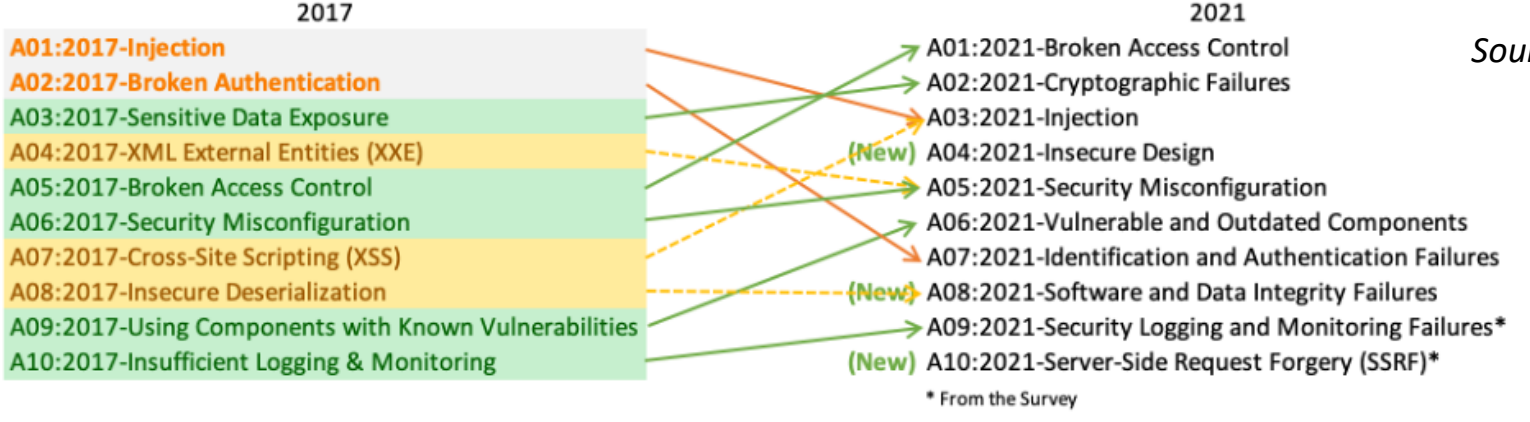


Source: OWASP

OWASP Top Ten

- For years, same topics but changing rankings
- Large revision in 2017 & 2021
- *Homework: Read and understand "Description" and "How to Prevent" section of A1-A10 2021 (https://owasp.org/Top10/)*

OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	☒	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	☒	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]



Source: OWASP

XSS – Cross Site Scripting

- Listed as A7:2017 and under A3:2021 “Injection attack”.
- Cross site scripting is the method where the attacker injects malicious script into trusted website.
- Types
  - Stored XSS: User inputs (malicious scripts) stored in websites (databases) and displayed to other users
    => e.g. Comment section
  - Reflected XSS: URLs with malicious scripts which are directly displayed in web pages to the user (which clicked the URL)
    => e.g. Query URL sent by email
  - DOM based XSS: URLs with malicious scripts which are directly displayed in web pages to the user (which clicked the URL), with difference that DOM based XSS doesn't even go to the server. The reason for this is that browsers do not send content that occurs in the URL after a # character to the server (interpretation within the browser).
    => e.g. Query URL sent by email with # in the get parameter

Cross Site Request Forgery (CSRF)

A malicious web site or email tricks a user's web browser to call a trusted site when the user is authenticated

- Victim must be logged into the target site at attack time, as browser requests automatically include all (session) cookies
- Attack commands can come from a second site the victim visits at the same time

Example:

- A site offers help configuring a specific router model. It also contains the following 1x1 "image":

```
<img src=”http://192.168.1.1/admin/config/outsideInterface?nexthop=123.45.67.89”
alt=”pwned” height=”1” width=”1”/>
```

- If the router accepts this command (e.g. because of cookie-based session authentication), the router now sends all traffic through 123.43.67.89 and the attacker (that owns the IP) can listen to it.

CSRF Example 2

- The e-banking site www.bank.com has a CRSF vulnerability
- The victim is currently logged into bank.com
- An email arrives in the mail-client, that unfortunately also is a web-browser and automatically loads images
- The email contains
  `<img src="https://www.bank.com/transfer?amount=1000&destination=attacker">`
- What happens?
- The transfer request is sent to www.bank.com with all cookies for www.bank.com and from the IP address of the logged-in user!
- www.bank.com cannot distinguish this from a legitimate request

CSRF Countermeasures

- Verify Referer / Origin headers, as they are not writable by client-side JavaScript
  - Check, if Referer / Origin = target URL / Domain
- Use SameSite Cookie Attribute for session cookies
  - Defines whether to send cookies along with cross-site requests (none, lax, strict)
  - Lax: will be sent in GET request in top window navigations (a href, window.open())
- Use Synchronizer (CSRF) Tokens for each session or request
  - CSRF should be generated on service side
  - For every client request, the server checks the validity of the token (stateful)
  - CSRF tokens should not be transmitted using cookies or http get
    (but with headers, hidden fields)
- Require user interaction (e.g. individual sign-off on transactions)

Note: More possibilities exist and it make sense to combine them (e.g. SameSite & CSRF Token)

XSS vs. CSRF

- Cross-site scripting (or XSS) allows an attacker to **execute arbitrary JavaScript** within the browser of a victim user.
- Cross-site request forgery (or CSRF) allows an attacker to **induce a victim user to perform actions** that they do not intend to.
- The consequences of XSS are generally more serious than for CSRF:
  - **CSRF** can be described as a "**one-way**" vulnerability because an attacker can cause the victim to make an HTTP request but cannot retrieve the response to that request.
  - In contrast, **XSS** is a "**two-way**" vulnerability because the script injected by the attacker can perform arbitrary actions

Agenda

- 12:15 – 13:00: Introduction OWASP
- 13:15 – 14:00: SOP / CORS & CSP
- 14:15 – 15:00: Exercise Session

Same-Origin Policy (SOP)

- The same-origin policy is a web browser security mechanism that aims to prevent websites from attacking each other.
- Without SOP, if you visit a malicious website, it would be able to read the contents of your bank account, read your emails from GMail, private messages from Facebook, etc.
- Under the Same-Origin Policy, a web browser **permits scripts** contained in a first web page **to access data in a second web page**, but only if both web pages have the same origin
- Questions
  - Does SOP prevent CSRF?
  - Does SOP prevent XSS?
  - Does SOP prevent sending Cookies?

Same-Origin Policy (SOP)

Two URLs have the same origin if the **protocol, port, and host/domain** (not subdomains) are the same for both URLs.

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol, host and port
http://www.example.com/dir2/other.html	Success	Same protocol, host and port
http://username:password@**www.example.com**/dir2/other.html	Success	Same protocol, host and port
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://**en.example.com**/dir/other.html	Failure	Different host
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)
http://www.example.com:**80**/dir/other.html	Depends	Port explicit. Depends on implementation in browser.

Source: www.arroyolabs.com

Same Origin Policy (SOP) and its Limits

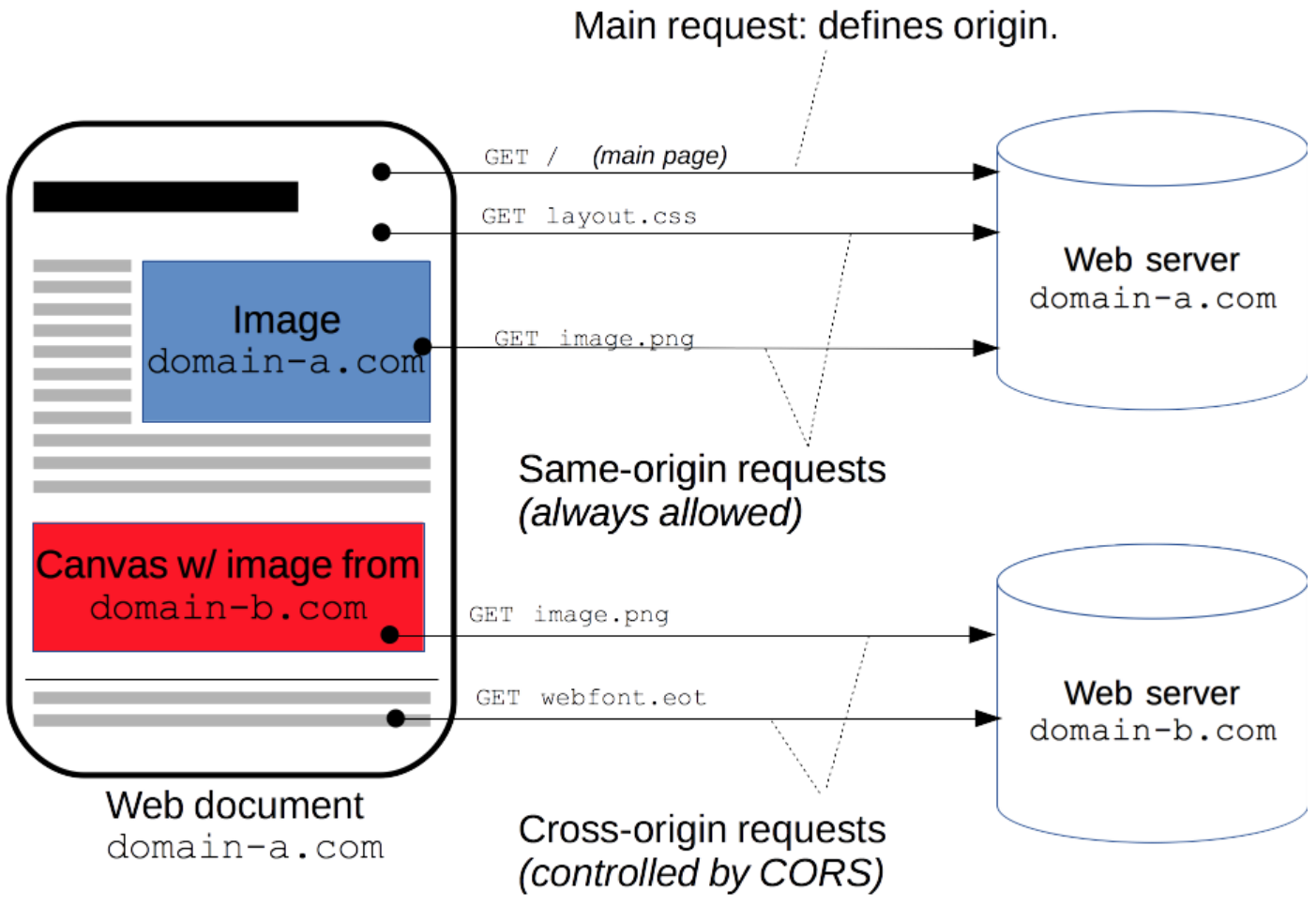
Sometimes, SOP does prevent legitimate requests.

Example:

- A web-page may want to embed content from a different server, and request this via a script on client-side. SOP prevents scripts from accessing a non same-origin page

Possible solution: **Cross-origin resource sharing (CORS)**

CORS Overview



Source: developer.mozilla.org

Cross-origin resource sharing (CORS)

1) Original page is from server A and the request from script goes to B
2) Browser asks B via OPTION request for access, stating A as Origin
3) B allows it via "Access-Control-Allow-Origin: A"
4) Request to B is made

OPTION: To find out which request methods a server supports, one can use the OPTIONS request. The response then contains an Allow header that holds the allowed methods (GET, POST, …).

Content Security Policy (CSP)

Content Security Policy (CSP) helps **detect and mitigate** certain types of attacks, including **cross-site scripting (XSS)** and data injection attacks.

- To enable CSP, you need to configure your web server to return the Content-Security-Policy HTTP header
- Content-Security-Policy: <policy-directive>; <policy-directive>
- Policy Directive categories (more and more directives to come)
  - Fetch directives control the locations from which certain resource types may be loaded (e.g. img-src).
  - Navigation directives govern to which locations a user can navigate or submit a form.
  - Reporting directives control the reporting process of CSP violations.
- Please browse through to get an impression of the concrete directives: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

Agenda

- 12:15 – 13:00: Introduction OWASP
- 13:15 – 14:00: SOP / CORS & CSP
- 14:15 – 15:00: Exercise Session

Aufgabe für nächste Vorlesung

Installation und Einrichtung der Vagrant-Umgebung (wer’s noch nicht gemacht hat)

- Anweisungen auf dem GitHub https://raw.githubusercontent.com/Fort-IT/FHNW-apsi/master/Demoenv/README.md
- Empfehlung: Vagrant Umgebung

Lesen und verstehen der A1-A10 Risiken Version 2021 (“Description” & “How to Prevent” Kapitel): https://owasp.org/Top10/

 Fragen können in der nächsten Vorlesung gestellt werden.

Exercises v06 6-1

Solution available on GitHub.

- Aufgabe 6-1: Secure Session Management

a) Weshalb ist Session Management bei Verwendung von HTTP so wichtig? Für welche Anwendungsfälle?

   ✓ HTTP ist ein zustandsloses Protokoll, d.h. aufeinanderfolgende Requests sind völlig zusammenhangslos. Sollen die Requests eines Benutzers also innerhalb eines Kon-textes betrachtet werden, muss dies durch ein Session Management entsprechend realisiert werden. Ein Beispiel ist ein Webshop, bei welchem der Benutzer Artikel in einem Warenkorb ablegen kann.

b) Welche Möglichkeiten kennen Sie für das Session Management?

   ✓ Grundsätzlich muss ein Session Identifier erstellt und mit den Requests an den Server gesendet werden. Dies erfolgt oft mittels Cookies oder einem anderen HTTP Request Header (z.B. Authorization HTTP Header). Alternativ kann dies auch als Parameter (GET oder POST erfolgen.

c) Was ist Session Hijacking? Wie könnte ein Angreifer hierfür vorgehen? Was können Sie dagegen machen?

   ✓ Unter Session Hijacking versteht man die Übernahme einer Benutzersession. In der RegeldurchVerwendungdesentsprechendenSessionIdentifiers.EineXSSSchwach-stelle könnte zur Ausführung eine Scriptes führen, welches das Session Cookie aus-liest und dessen Inhalt an den Angreifer übermittelt. Der Session Identifier muss geschützt werden. Bei Cookies sollte bspw. das HttpOnly Flag gesetzt werden, wel-ches verhindert, dass Scripts auf das Cookie zugreifen können.

Exercises v06 6-1

Solution available on GitHub.

- Aufgabe 6-1: Secure Session Management

d) Welche Eigenschaften muss ein Session Identifier erfüllen?

   ✓ Der Session Identifier darf nicht in den Besitz des Angreifers gelangen. Somit muss der Zugriff darauf verhindert, aber gleichzeitig auch die Entropie hoch gehalten werden. Der Angreifer soll also keine Möglichkeit erhalten, den Session Identifier zu erraten.

e) Ein Freund erzählt Ihnen, dass er als Session Identifier einen SHA-256 Hash des ak-tuellen Timestamps und des Benutzernamens verwendet. Wie beurteilen Sie dieses Vorgehen? Was raten Sie Ihrem Freund (ausser den Besuch der apsi Vorlesung)?

   ✓ Der Session Identifier darf unter keinen Umständen erraten werden. Ansonsten kann ihn der Angreifer verwenden. Timestamp und Benutzername sind ableitbar und haben demnach nur eine geringe Entropie. D.h. der Session Identifier kann erraten und die Session so übernommen werden.

Exercises v06 6-2

Solution available on GitHub.

- Aufgabe 6-1: HTTP Parameter

a) Welche beiden fundamentalen Möglichkeiten/Methoden stehen für die Übermitt-lung von Parametern zur Verfügung?

   ✓ HTTP stellt für die Übermittlung von Parametern die Methoden GET und POST zur Verfügung.

b) Was sind die Vor- und Nachteile der beiden Methoden

   ✓ GET ermöglicht es, Parameter direkt in der URL als Query String anzugeben. Dies stellteinesehreinfacheMöglichkeitdar.AllerdingsmussmansichdrüberimKlaren sein, dass Requests in vielerlei Logfiles gelogged werden. Dabei werden die GET Parameter natürlich mitgelogged. Ganz im Gegensatz zu POST parametern. Dort wird der Query String im HTTP Body übermittelt.

c) Der Freund aus Aufgabe 6-1 erzählt Ihnen weiter, dass er der Einfachheit halber die Login Credentials mittels der GET Methode überträgt. Er argumentiert, dass dies ja kein Problem sei, da er die Verbindung ja mittels TLS 1.3 abgesichert habe. Wie beurteilen Sie diese Aussage?

   ✓ Die Verbindung ist zwar geschützt und ein Angreifer kann die Login Credentials nicht direkt im Netzwerk lesen. Hingegen werden aber die Login Credentials in den Logfiles mitgelogged. Wer Zugriff auf die Logfiles erhält, kann demnach die Login Credentials auslesen. Sie sollten nicht als GET Parameter übertragen werden.

Exercises v06 6-3

Solution available on GitHub.

- Aufgabe 6-3: CGI

Siehe Musterlösung