Kapitel 4:

Security-Aspekte der Dienst- und Kommunikationsebene

Sicherheit von Web-Diensten

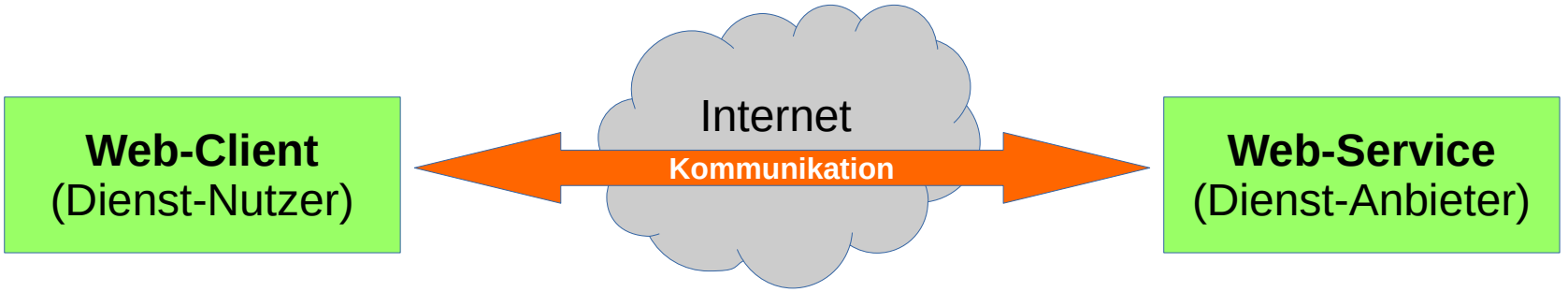
- **Web-Services sind ein interessantes Angriffsziel**
  - Zunehmend viele Dienste werden als Web-Dienste angeboten
  - Für einen Angreifer sind solche sensiblen Dienste lohnend, z.B.
    - **Geld** stehlen / Betrug
      - z.B. Online-Banking, Börsen-Depot, Handels-Plattformen, …
    - **Waren** stehlen / Betrug
      - z.B. Handels-Plattformen, Logistik-Dienste, …
    - Diebstahl von **Geheimnissen** (Wirtschaft, Privatleben)
      - z.B. Webmail-Dienste, Cloud-Services, Online-Storage, …
    - **Identitäts**diebstahl, Betrug
      - z.B. Webmail-Dienste, Sozial-Plattformen, ...
  - **Kompromittierung** des Zugangs eines Nutzers

    → Angreifer hat ggf. vollen Zugriff mit den Rechten des einen Nutzers
  - **Kompromittierung** des Web-Services selbst

    → Angreifer hat ggf. vollen Zugriff auf alle verbundenen Dienste aller Nutzer

Sicherheit von Web-Diensten

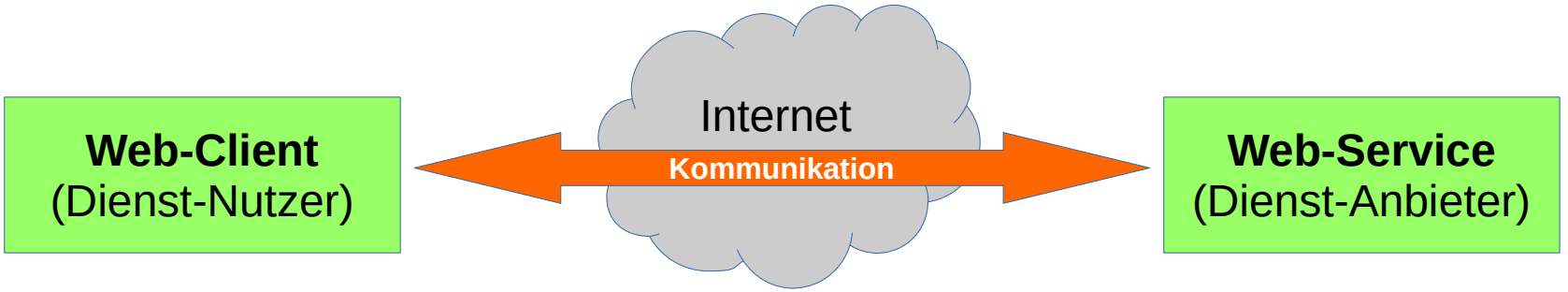
- **Es gibt drei wesentliche Angriffspunkte**

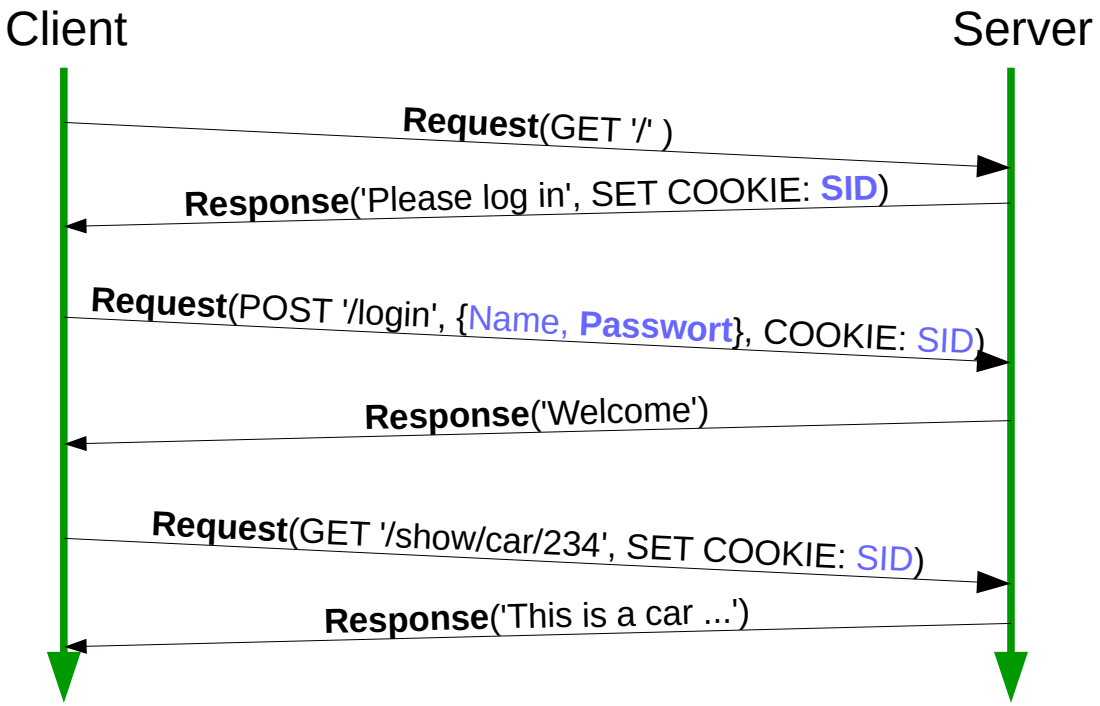


  - Client-System
    - Akut: CSRF, Umleitung auf andere Server, ...
    - Dauerhaft: Trojaner-Installation (auch als Addon), falsche SSL-Zertifikate, ...
  - Kommunikationskanal
    - Abhören von Daten,
    - Manipulieren von Daten, MITM (Man in the Middle), DNS-Manipulationen
  - Server-System
    - Akut: direkte JS-Injection (XSS), Cookie Stealing, Session Fixation, ...
    - Dauerhaft: indirekte JS-Injection (XSS), Diebstahl des privaten SSL-Keys, ...
    - Kompromittierend: SQL-Injection, Privilegieneskalation, ...

Sicherheit von Web-Diensten

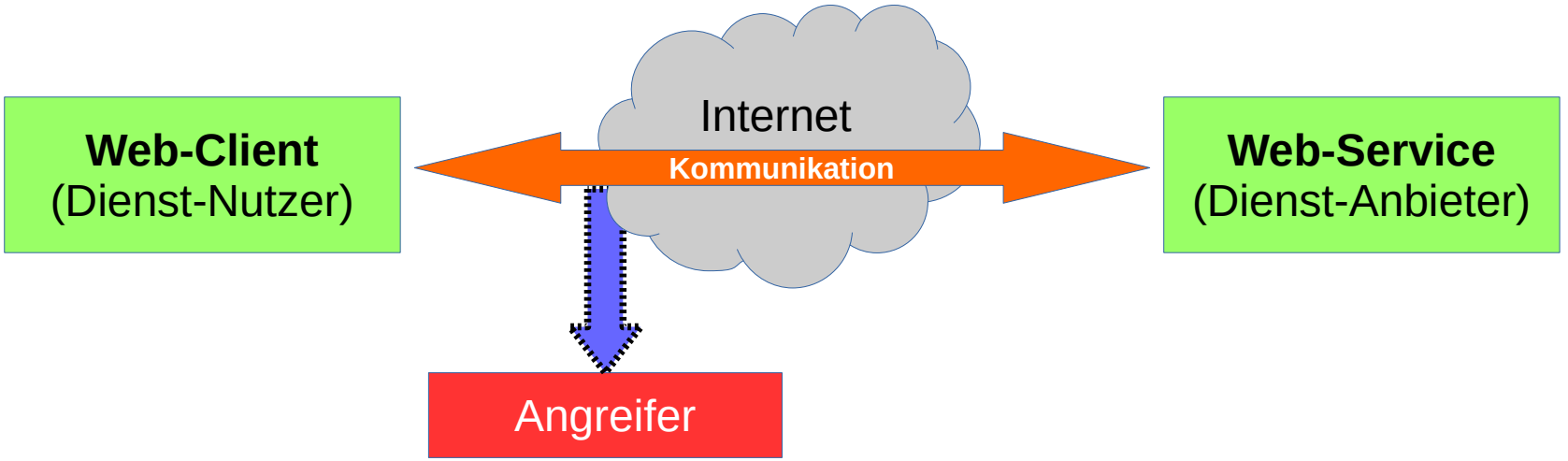
- **Typische Anfragesequenz**





Sicherheit von Web-Diensten: Abhören

- **Bedrohung: Abhören der Internet-Kommunikation**



  - häufig durch Manipulation des Netzes auf Client-Seite
    - Anbieten eines (offenen) WLAN-Access-Points (typisch: Café-WLAN)
    - Manipulation des heimischen DSL-Routers
  - Aber auf der ganzen Kommunikationsstrecke möglich
    - Geheimdienste / Industriespionage: Jede Komponente manipulierbar

Sicherheit von Web-Diensten: Abhören

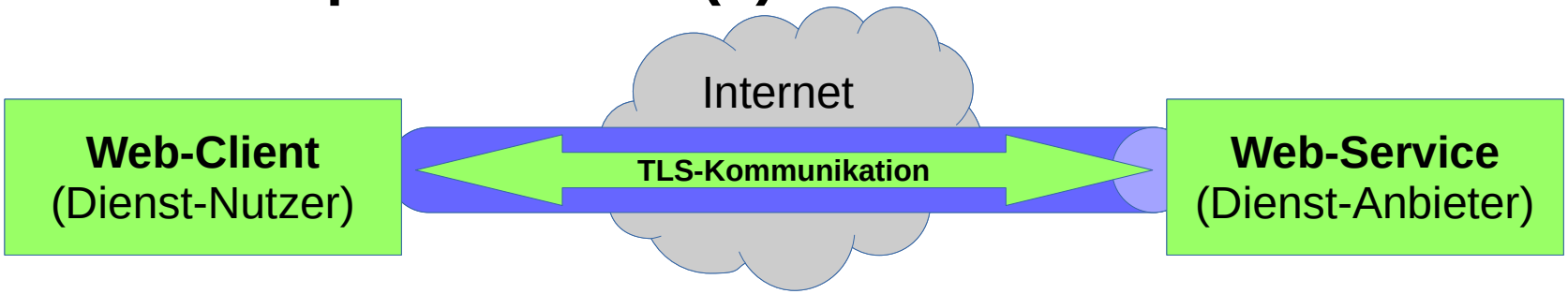
- **Unverschlüsselte Verbindungen können abgehört werden**
- Dabei können sensible Daten in falsche Hände geraten
  - Passwörter, TANs (Authentifizierungs-Daten)
    - ermöglichen es, neue Sitzungen zu eröffnen (aktiver Zugriff)
  - Authentifizierungs-Tokens (oben: SID = Session-ID Cookie)
    - ermöglichen es, eine Sitzung zu übernehmen (aktiver Zugriff)
  - Alle Nutzdaten aus dem folgenden Client-Server-Dialog
    → ermöglichen es, Daten mitzulesen (passiver Zugriff)
    - abgerufene Seiten, Formulardaten (z.B. sensible Email-Inhalte, ...)
    - vom Server gelieferte Daten (z.B. sensible Daten, Email-Inhalte, …)
- **Kann man erfolgte Zugriffe erkennen?**
  - passive Zugriffe (Mitlesen) sind nicht zu erkennen
  - aktive Zugriffe evtl. zu erkennen z.B. über **Server-Meldungen**
    - Meldung „*Zuletzt eingeloggt am: ...*“
    - Email „*Sie haben eine Einstellung geändert*“, „*Sie haben bestellt: ...*“

Sicherheit von Web-Diensten: Transportverschl.

- **Lösung: Kryptographisch gesicherte Kommunikation**
  - Die Kommunikation erfolgt durch einen verschlüsselnden „Tunnel“
    - Nachrichten können nicht mitgelesen werden
    - Nachrichten können nicht unbemerkt verändert werden
- **Konkret in Webdiensten: HTTPS = HTTP über TLS**
  - **TLS**: „Transport Layer Security“
    - Früherer Name: „Secure Sockets Layer“ (**SSL**)
    - SSL-Version 3.1 ist identisch mit TLS-Version 1.0
  - Die Nutzung von HTTP über TLS ist an der URL erkennbar
    - `https://mein.server.xy/test`
  - Die Verschlüsselung und des Schlüsselmanagement erfolgen transparent für die Applikation und Nutzer
  - siehe https://de.wikipedia.org/wiki/Transport_Layer_Security

Sicherheit von Web-Diensten: Transportverschl.

- **Grundkonzepte von TLS (1)**



  - TLS benutzt zur Verschlüsselung von Massendaten ein **symmetrisches** Verschlüsselungsverfahren
    - z.B. DES, IDEA, **AES**
    - symmetrische Verfahren benutzen **den selben Schlüssel** zum ver- und entschlüsseln
      - entsprechend müssen Client und Server beide den Schlüssel kennen
  - Beim Verbindungs-Aufbau wird für jede Verbindung ein neuer symmetrischer Schlüssel ausgehandelt
    - Die beiden Endpukte der Verbindung kennen den Schlüssel (nur diese beiden!)
      - Nur diese können die Pakete ver- und entschlüsseln
      - Ein Mithören ist nicht möglich

Sicherheit von Web-Diensten: Transportverschl.

2 Offene Fragen:

Wie kommen beide Seiten an den gemeinsamen Schlüssel?

- Im Webclient-Webserver-Szenario kennen sich beide Seiten im Allgemeinen zunächst gar nicht.

  → Der Schlüssel muss dynamisch erzeugt und untereinander ausgetauscht werden (Zufallswert)

- Wie kann ein neu erzeugter gemeinsamer Schlüssel dann sicher übertragen werden? (Abhörgefahr!)

  → Dazu bräuchte man schon einen sicheren Kanal …

  ↔ Den wollen wir doch gerade erst aufbauen → Henne-Ei-Problem

Warum kann der Kommunikationspartner nicht auch der Angreifer sein?

- Wir können zwar geheim kommunizieren,
  aber **wir wissen nicht, mit wem wir gerade kommunizieren**

Sicherheit von Web-Diensten: Transportverschl.

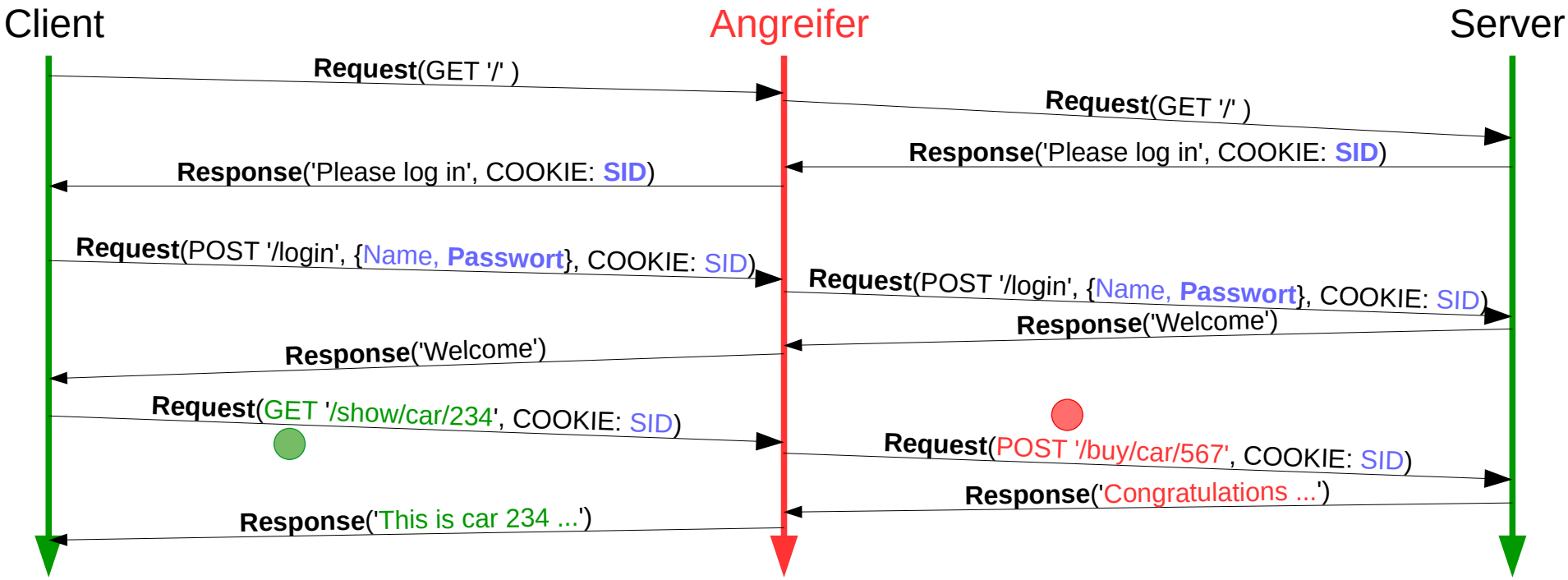
- **Angriffszenario: MITM-Angriffe**
  - MITM = „Man In The Middle“
  - Die TLS-Verbindung führt dann zum Angreifer

    - dieser hat eine zweite Verbindung zum originalen Service und reicht Anfragen und Antworten durch

- **Alternative: Der Angreifer simuliert den Service**

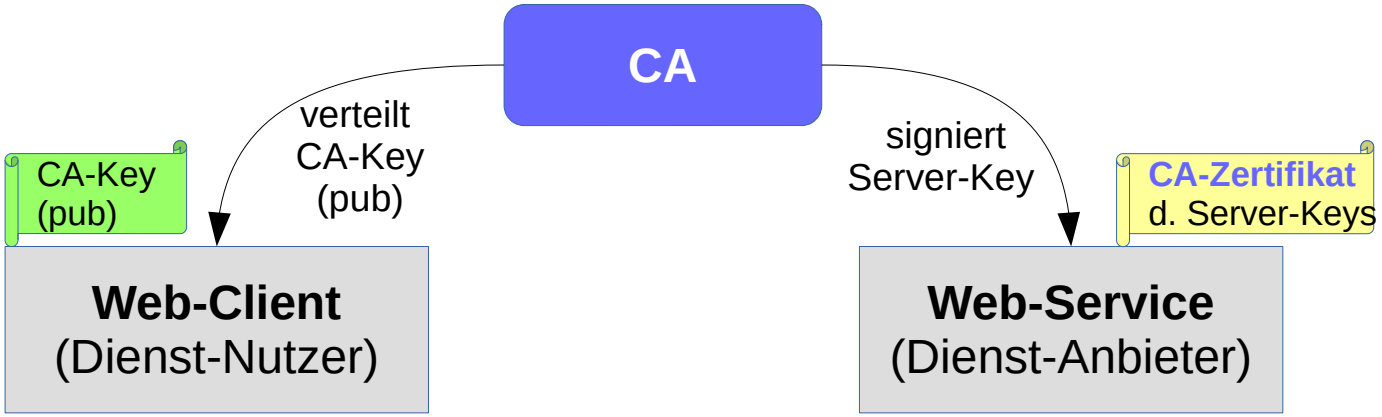
Sicherheit von Web-Diensten: Transportverschl.

- **MITM sieht Requests und Responses unverschlüsselt**
  - Der MITM kann mitlesen, aber auch Änderungen vornehmen



Sicherheit von Web-Diensten: Zertifikate

- **Lösung: Zertifikate zur Bestätigung der Identität (PKI)**



  - Es gibt eine Zertifizierungsstelle, bei der der Dienst-Anbieter ein Zertifikat erhalten kann. Dieses Bestätigt seine **Identität**.
    - Diese Zertifizierungsstellen heißen **CA** („Certificate Authority“)
  - Es gibt einen Schlüssel zur Prüfung der Zertifikate (CA-pub-Key)
    - Mit CA-pub-Key kann man die Echtheit der CA-Zertifikate prüfen
  - Jeder Web-Client besitzt den CA-pub-Key (bereits eingebaut)
    - Der Client vertraut dem CA-pub-Key, d.h. kann er damit die Echtheit eines Zertifikats bestätigen, so akzeptiert er dieses als echt.

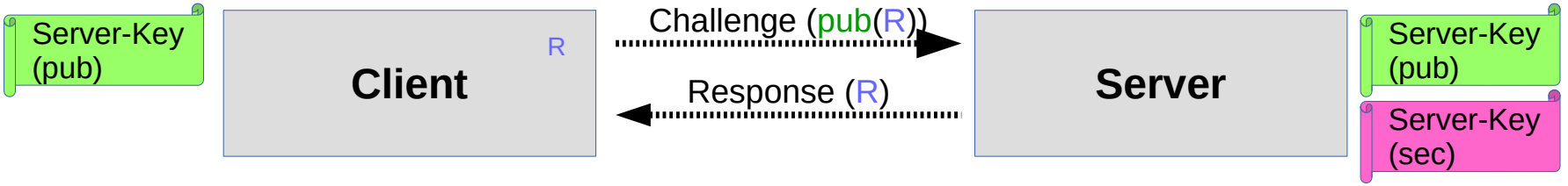
Sicherheit von Web-Diensten: Zertifikate

- **Wie funktioniert die Zertifikatprüfung** (Grundkonzept)**?**
  - Das Verfahren beruht auf **asymmetrischer** Verschlüsselung
    - Jeder Teilnehmer (CA, Server) generiert eine **Schlüsselpaar** (pub und sec)
      - Der pub-Teil wird frei verteilt (**public**)
      - Der sec-Teil wird geheim gehalten und ist nur dem Teilnehmer bekannt (**secret**)
    - Was mit pub verschlüsselt wird, kann nur mit sec entschlüsselt werden

    - Was mit sec verschlüsselt wird, kann nur mit pub entschlüsselt werden

Sicherheit von Web-Diensten: Zertifikate

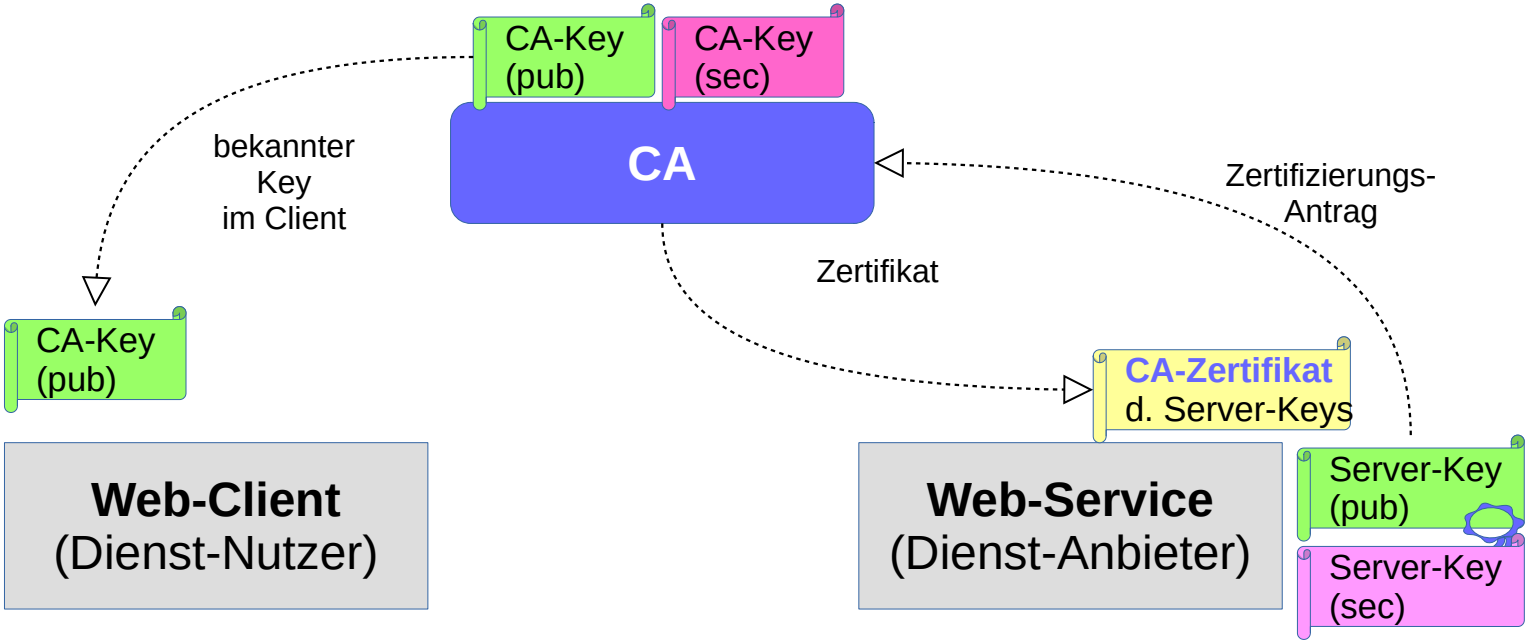
- **Wie funktioniert die Zertifikatprüfung?**
  - Szenario:
    - Der Server hat eine Schlüsselpaar sec und pub erzeugt.
    - Nur der Server kennt sec
    - Der Client kennt pub



  - **Schlüssel-Prüfung**: Challenge-Response-Verfahren
    - Der **Client** erzeugt einen Zufallswert R, verschlüsselt ihn mit pub und schickt pub(R) an den Server („Challenge“)
    - Der **Server** entschlüsselt mit sec den Wert R = sec(pub(R)) und schickt R an den Client („Response“)
    - Der **Client** erkennt die richtige Antwort (R) und weiß nun, dass er mit dem echten Server (zu pub) kommuniziert
      - Da die Entschlüsselung nur mit sec möglich ist und nur der Server sec besitzt.

Sicherheit von Web-Diensten: Zertifikate

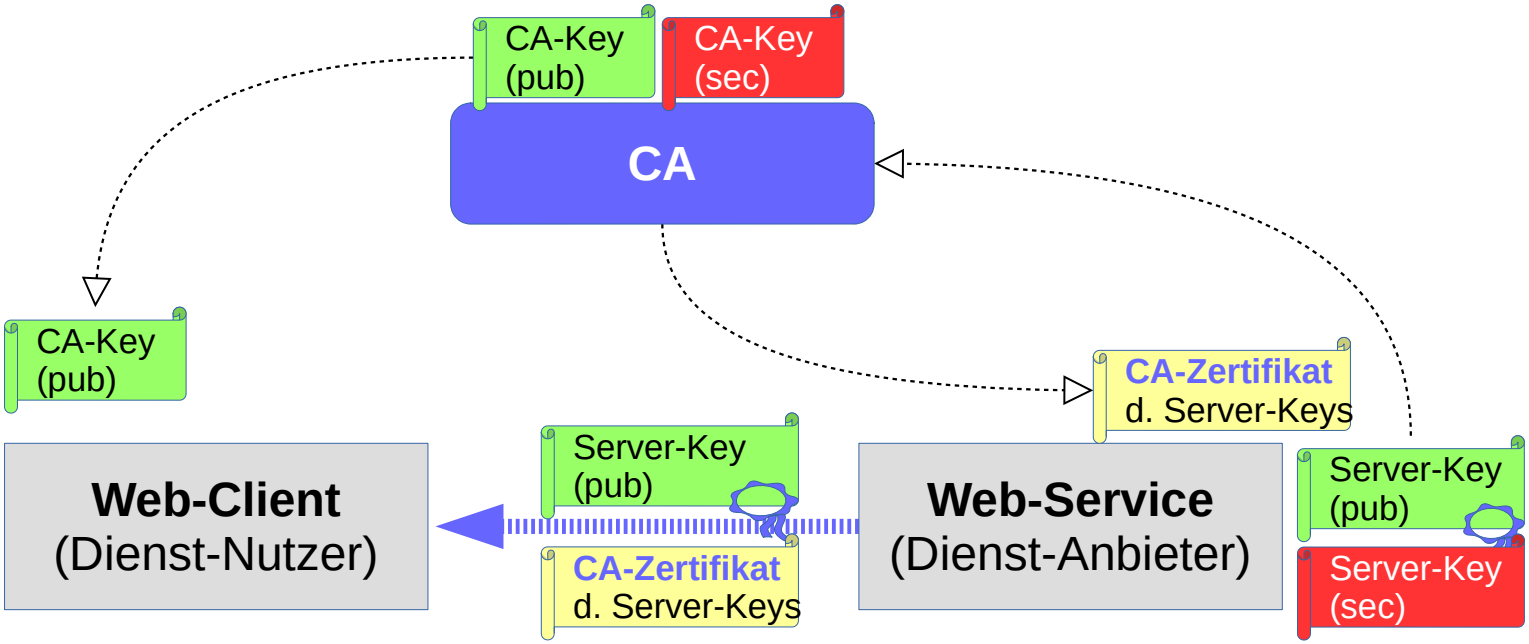
- **Die Echtheit der Zertifikate bestätigt die CA**
  - CA = certificate authority (Zertifizierungsstelle)



  - Die CA bestätigt die Echtheit des Server-Keys mit einem Zertifikat
    - Der Client hat aber normalerweise aber nicht die pub-Keys aller Server
    - Wie kann der Client einen Server-pub-Key bekommen?

Sicherheit von Web-Diensten: Zertifikate

- **Wie funktioniert die Zertifikatprüfung?**
  - **Lösung:** Der Client erhält der Server-pub-Key beim TLS-Aufbau direkt **vom Server**, zusammen mit dem passenden CA-Zertifikat
  - Der Client kann nun mit dem CA-pub-Key das CA-Zertifikat und damit dann die Echtheit des Server-pub-Keys prüfen



Sicherheit von Web-Diensten: Zertifikate

- **Ablauf des TLS-Verbindungsaufbaus**
  - Der Client baut zum Ziel-Server eine symmetrisch verschlüsselte Verbindung auf
    - Nun können wir nicht mehr abgehört werden. MITM-Angriff noch möglich
  - Der Server schickt dem Client seinen Server-pub-Key und das CA-Zertifikat
    - Der Client prüft mit seinem CA-pub-Key das CA-Zertifikat und dann mit dem CA-Zertifikat den Server-pub-Key
  - Da der Client der CA vertraut, weiß er nun, dass die Daten in dem CA-Zertifikat von der CA stammen, also zutreffend sind
    - z.B. der Host-Name, für den das Zertifikat gilt (X.509: Feld „CN“=„*Common Name*“)
  - Wenn der Host-Name mit dem übereinstimmt, der in der URL vorkommt, ist die Zertifikatsprüfung **erfolgreich** abgeschlossen.
    - Es gibt weitere Anforderungen, z.B. gibt es einen zeitlichen Gültigkeitsbereich

Sicherheit von Web-Diensten: Zertifikate

- **In der Realität (X.509-PKI) ist das Verfahren flexibler**

  1) Im Web-Client sind viele (einige Dutzend) **Root-CAs** eingetragen
     - Problem: Sind diese alle vertrauenswürdig? (Ein manipulierbarer genügt!)

  2) Nicht alle Zertifikate müssen direkt von einer Root-CA signiert sein
     - CAs sind hierarchisch organisiert: X.509 **PKI** („Public-Key-Infrastructure“)
     - Es gibt entsprechend **Zertifikats-Ketten**
       - z.B. „`vlu.cs.rptu.de`“
         ← „*Sectigo RSA Organization Validation Secure Server CA*“
         ← „*USERTrust RSA Certification Authority*“ (**Root-CA**)
       - Der Client (Browser) muss zunächst nur den CA-pub-Key der Root-CAs kennen

  3) Zertifikate können zurückgezogen werden
     - **CRL** (Certificate Revocation List)
       - Statische Liste mit zurückgezogenen Zertifikaten (Größe? Aktualisierung?)
     - **OCSP** (Online Certificate Status Protocol)
       - Online-Abfrage beim CA, ob das Zertifikat noch gilt (Verfügbarkeit? DOS-Gefahr!)

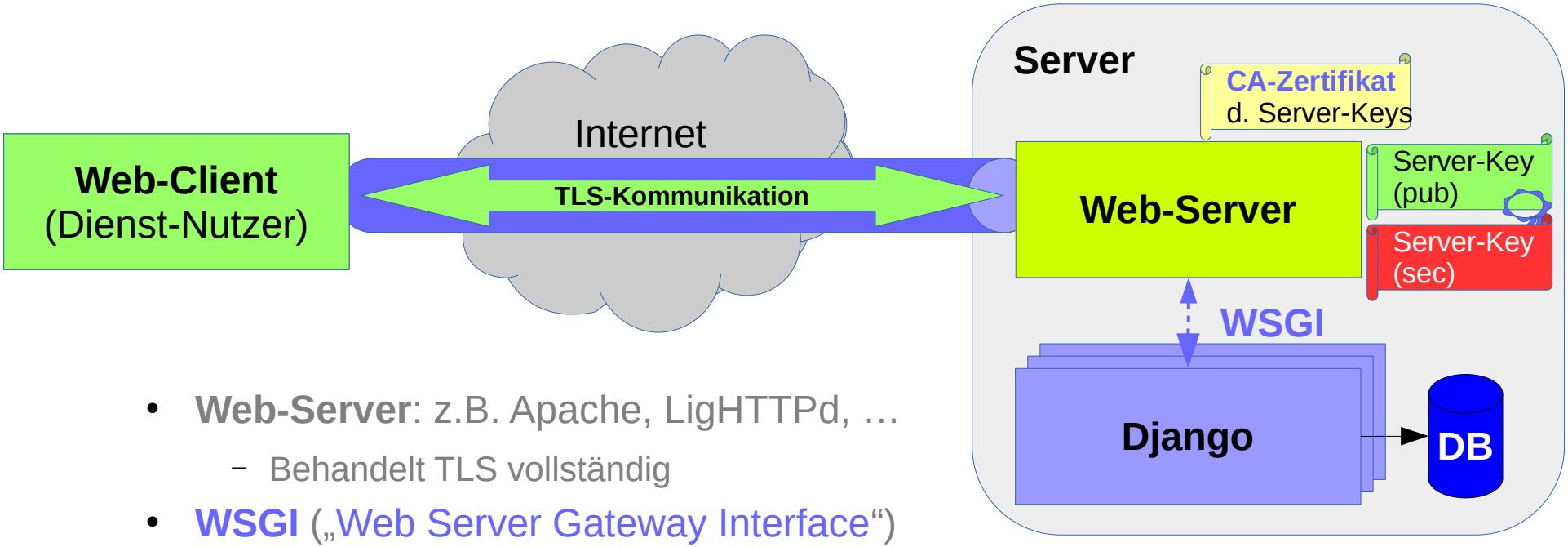
Sicherheit von Web-Diensten: Zertifikate

- **Zuverlässigkeit der PKI**
  - Zertifikate werden leider oft nach minimaler Prüfung ausgestellt
    - Man kann mit wenig krimineller Energie ein falsches Zertifikat bekommen
      - z.B. Heise-Meldung vom 17.3.2015 zu Microsoft-Zertifikat
  - Es gibt deshalb Extended-Validation-SSL-Zertifikate (**EV-SSL**)
    - Diese werden aufwendig geprüft (z.B. für Banken)
    - Die Browser zeigen sie besonders markiert an (mittlerweile oft nicht mehr)
  - Mit mehr „Durchsetzungsvermögen“ bekommt man jedes Zertifikat
    - Krimineller Vereinigungen, Geheimdienste, ...
- **Schwachpunkt Mensch / Routine**
  - Problem: Schauen Sie sich jede URL genau an, ob Sie nicht vielleicht gar nicht da sind, wohin sie wollten?
    - Buchstaben-Dreher, Schreibfehler, Nicht-TLS-Weiterleitungen, Phishing, ...
    - Unicode-Host-Namen mit zweideutigen Zeichen (optisch nicht zu erkennen)

Sicherheit von Web-Diensten: Django + TLS

- **Wie wird Django mit TLS betrieben?**
  - Django wird dazu „*hinter*“ einem Web-Server mit WSGI-Schnittstelle betrieben

Ab Django 3 alternativ zu WSGI auch **ASGI** (*asynchrone* Schnittstelle) verfügbar.



- **Web-Server**: z.B. Apache, LigHTTPd, …
  - Behandelt TLS vollständig
- **WSGI** („Web Server Gateway Interface“)
  - Nachfolger von FCGI, Protokoll relativ ähnlich zu HTTP
  - Spezialisiert auf die Schnittstelle Webserver zu Backend
  - Unterstützt Replikation von Backend-Prozessen und Verteilung im Netzwerk

Sicherheit von Web-Diensten

- **Sicherheit ist also offensichtlich erledigt** *(oder?)*
  - Wir kommunizieren abhör- und manipulationssicher (durch TLS)
    - Sind uns sicher mit dem richtigen Server zu reden (TLS+X.509-PKI)
    - Es gibt keine MITM-Probleme
  - Wir escapen zuverlässig (z.B. durch Django)
    - keine SQL-Injections
    - keine XSS-Attacken
  - Wir Vermeiden CSRFs (z.B. durch Django)
    - keine indirekte Nutzung der Nutzer-Authentifikation

- **Leider gibt es an einem besonders kritischen Punkt ein systematisches Problem in HTTP**
  - Cookie-Stealing, Session-Stealing, Session-Fixation

Sicherheit von Web-Diensten

- **Angriffs-Form Cookie- bzw. Session-Stealing**
  - Ziel ist es, aus einer laufenden Verbindung als Angreifer ein Cookie zu lesen (also zu stehlen)
    - Idealerweise die Session-ID SID (als Authentifizierungs-Token)
    - Mit der SID kann der Angreifer mit Authorisierung des Nutzers agieren
  - Das Cookie wird aber ja durch TLS verschlüsselt übertragen
    - Selbst wenn man das Netzwerk kontrolliert, bekommt man es so nicht
  - **Angriff 1**: **XSS** (Cross Site Scripting = JS-Injection)
    - Wir injizieren in eine Seite des Servers z.B. per **XSS** ein Stück Javascript, das das Session-Cookie an einen fremden Server weiterleitet
    - **Gegenmaßnahme 1:** Schutzmaßnahmen gegen XSS (korrektes escaping)
      - Dies scheitert z.B. bei beabsichtigtem fremden JS-Code (z.B. Werbung)
    - **Gegenmaßnahme 2:** JS-Zugriff auf Session-Cookie verbieten
      - Django: `HttpResponse.set_cookie(..., httponly=True)`
      - Django-Settings: `SESSION_COOKIE_HTTPONLY=True` (ist default)
      - PHP: analog

Sicherheit von Web-Diensten

- **Angriffs-Form Cookie-** bzw. **Session-Stealing**
  - **Angriff 2**: **Cookies unverschlüsselt abfangen**
    - Szenario: Client und Server Kommunizieren per TLS
      - Der Angreifer kann die Netzwerk-Kommunikation abhören, aber HTTPS nicht entschlüsseln
    - Der Angreifer provoziert eine unverschlüsselten HTTP-Zugriff auf den Server
      - z.B. durch ein IMG-Tag in einer parallel aufgerufenen Webseite des Angreifers
    - Wenn das (Session-) Cookie nicht als **Secure-Cookie** gesetzt wurde, wird es über die unverschlüsselte Verbindung übertragen
      - Der Angreifer kann es abgreifen und die Sitzung selbst übernehmen
    - **Gegenmaßnahme:** (Session-) Cookies nur mit Secure nutzen
      - Django: `HttpResponse.set_cookie(..., secure=True)`
      - Django-Settings: `SESSION_COOKIE_SECURE=True` (default ist False!)
      - PHP: analog
    - *Verständnisfrage*: *Warum ist `SESSION_COOKIE_SECURE=True` nicht schon default?*
      - Dann würde das Session-Management nur noch unter TLS funktionieren

Sicherheit von Web-Diensten

- **Angriffs-Form Cookie-** bzw. **Session-Stealing**
  - **Angriff 3**: **Domain-Cookies**
    - Szenario: Der Server setzt ein Domain-Cookie, um die Session für viele Server nutzen zu können
      - z.B. Django-Settings: `SESSION_COOKIE_DOMAIN='.rptu.de'`
    - Der Angreifer kann zwar nicht den Server kontrollieren, aber …
      - vielleicht einen anderen Server in der Domäne übernehmen
      - auf einem Server in der Domäne einen beliebigen (auch unprivilegierten) Port öffnen (Login-Server?)
      - einen DNS-Eintrag fälschen, der der IP-Adresse eines seiner Server einen Host-Namen in der Domäne zuordnet.
        - Das kann durch Manipulation des echten DNS geschehen, oder
        - bei MITM-Attacken durch eine gefälschte DNS-Antwort.
    - **Gegenmaßnahme: Domain-Cookies meiden**
      - Gegen die DNS-Manipulation gibt es kaum Gegenmittel
        - evtl. die DNS-Erweiterung DNSSEC, die jedoch noch nicht verpflichtend ist
        - DNSSEC: https://de.wikipedia.org/wiki/Domain_Name_System_Security_Extensions
      - TLS hilft hier, aber nur zusammen mit `SESSION_COOKIE_SECURE=True`

Sicherheit von Web-Diensten

- **Wenn aber das Session-Cookie nicht zu stehlen ist ...**
  - … dann machen wir (als Angreifer) eben eines!
- **Angriffs-Form Session-Fixation**
  - **Idee**: Wir versuchen nicht, das fremde Session-Cookie zu bekommen, sondern wir schieben dem Client/Server eines unter.
  - Vorgehensweise:
    - Der Angreifer baut eine Session zum Server auf (noch nicht authentifiziert)
      - z.B. ruft er per GET die Login-Seite auf → der Server setzt ihm eine SID
    - Der Angreifer sorgt dafür, dass der Client diese Session übernimmt
      - indem wir dem Client ein Cookie für den Server setzen oder
      - indem wir einen Mechanismus des Servers nutzen, eine SID per GET zu setzen
    - Der Client loggt sich irgendwann mit diesem SID-Cookie ein
    - Der Angreifer hat somit eine vom Client authentifizierte Session
  - *Überraschend? Ja! Erschreckend? Ebenfalls!*

Sicherheit von Web-Diensten

- **Angriffs-Form Session-Fixation**
  - Wie setzt man das Session-Cookie des Clients für den Server?
    - z.B. per XSS auf einer Seite des Servers setzt der Angreifer per JS ein Cookie
      - `<script> document.cookie="sessionid=1234"; </script>`
    - z.B. per XSS auf einer beliebigen Seite in der Domain des Servers setzt der Angreifer per JS ein Domain-Cookie
      - ```
        <script>
            document.cookie="sessionid=1234;domain=.target.dom";
        </script>
        ```
      - Das funktioniert auch dann, wenn der Server gar keine Domain-Cookies benutzen will – für den Server sind diese nämlich nicht als solche zu erkennen
      - Wie wir oben bereits gesehen haben, kann das auch über einen beliebigen (auch unprivilegierten) Port erfolgen.
    - z.B. per HTML-Injection: Analog zum XSS ein Meta-Tag injizieren:
      - `<meta http-equiv=Set-Cookie content="sessionid=1234">`
    - **Session-Adoption**
      - Manche Web-Server (JRun) erlauben, als GET-Parameter eine SID anzugeben:
        - Den Client dirigieren auf `http://www.target.dom/?jsessionid=1234`
        ```

Sicherheit von Web-Diensten

- **Angriffs-Form Session-Fixation**
  - **Gegenmaßnahme 1**: Session-ID mit Client-Eigenschaft verbinden
    - Wenn man in den Session-Daten auch einige Eigenschaften des Clients beim Erzeugen der Session ablegt und diese bei jedem Zugriff prüft, wird die Session-Fixation erschwert.
      - IP-Adresse (Bindung behindert aber Roaming des ggf. mobilen Clients)
      - Zertifikat-Daten, Browser-Version, …
  - **Gegenmaßnahme 2**: Session-ID bei Login wechseln
    - Beim Login sollte die Session-ID ausgetauscht werden.
    - Dadurch wird die Session-Fixation gelöst
      - der Angreifer kennt die neue Session-ID ja nicht mehr
    - Djangos mitgelieferte Login-View macht das so.
    - Zusätzlich evtl. Session-Daten löschen bei Session-Timeout oder Logout
  - **Gegenmaßnahme 3**: Session-Adoption nicht erlauben
    - Session-ID in keiner Weise vom Client vorgeben lassen (meistens gegeben)

Sicherheit von Web-Diensten

- **Zusätzlicher Schutz gegen alle o.g. Session-Cookie-Sicherheitsprobleme**
  - Bei kritischen Operationen eine zusätzliche Authentifizierung neben dem Cookie (Authentifizierungs-Token) verlangen
    - Erneute Passwort-Eingabe im selben Formular
      - Typischerweise vorzufinden bei Passwort-Änderungs-Formularen
    - Eingabe eines zweiten Faktors (**2FA**, im selben Formular), z.B.
      - **TAN**: Liste von vorab vergebenen Codes (früher bei Bank-Transaktionen)
      - **mTAN**: Übertragung per SMS (nicht sehr sicher) oder einer speziellen App
      - **TOTP**: Time-based one-time password
        - Passwort wird von einer lokalen App erzeugt und gilt nur für kurze Zeit (1 Minute)
        - Dazu muss die App ein gemeinsames Geheimnis kennen (Einrichtung per QR-Code)
  - Dies schützt auch vor Missbrauch von versehentlich nicht ausgeloggten Browser-Sitzungen auf Client-Seite

Ende :)