Hochschule
Albstadt-Sigmaringen

Albstadt-Sigmaringen University

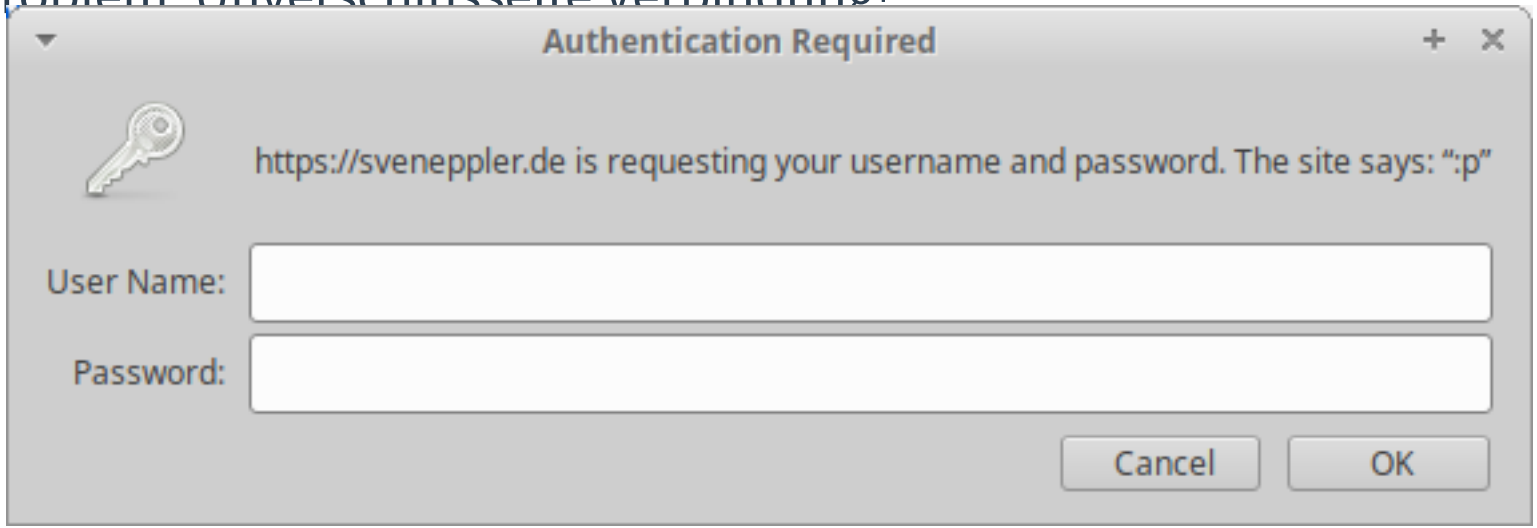
Authentication

Dipl. Ing. Sven Eppler (FH)
sodge IT GmbH

Wer benutzt da meine App?

HTTP Basic Auth

- Benutzername und Passwort im HTTP Header
- Muss bei jedem Request enthalten sein
- Wird als base64 encoded header geschickt
  - Problem: Unverschlüsselte Verbindung!



HTTP Basic Auth

- **Algorithmus:**
  - Verbinde $Username und $Passwort mit einem „:“
    `$Username:$Password`
  - Encode diesen String mit Base64:
    `VXNlcm5hbWU6UGFzc3dvcmQ=`
  - Setze diesen String als Authorization Header
    `Authorization: Basic VXNlcm5hbWU6UGFzc3dvcmQ=`
- **Der Server wendet den Algorithmus in umgekehrter Reihenfolge an, um an $Username und $Passwort zu kommen**

Erinnerung: HTTP ist stateless!

- Basic Auth muss daher IMMER Benutzer und Passwort neu übertragen
- Das erhöht die Angriffsfläche. Ein einzelner unverschlüsselter Request exposed den kompletten Login
- Browser cached Login-Daten nur bis zum neustart, danach ist ein erneuter Login notwendig
- Wie bleiben User länger und sicher eingeloggt?

Nachteile HTTP Basic Auth

- **Keine Verschlüsselung der Login-Daten**
- **Keine langfristige Pufferung beim Client**
  - Auch nicht gewünscht, weil dann Username/Passwort gespeichert werden müssen
- **Der Passwort-Prompt ist Browser spezifisch und integriert sich nicht in eine App**
- **Andere Konzepte wie OneTimePassword, 2-Faktor-Authentifizierung nur schwer umsetzbar**

Gib mir eine Session!

Sessions: Keeping state in a stateless world

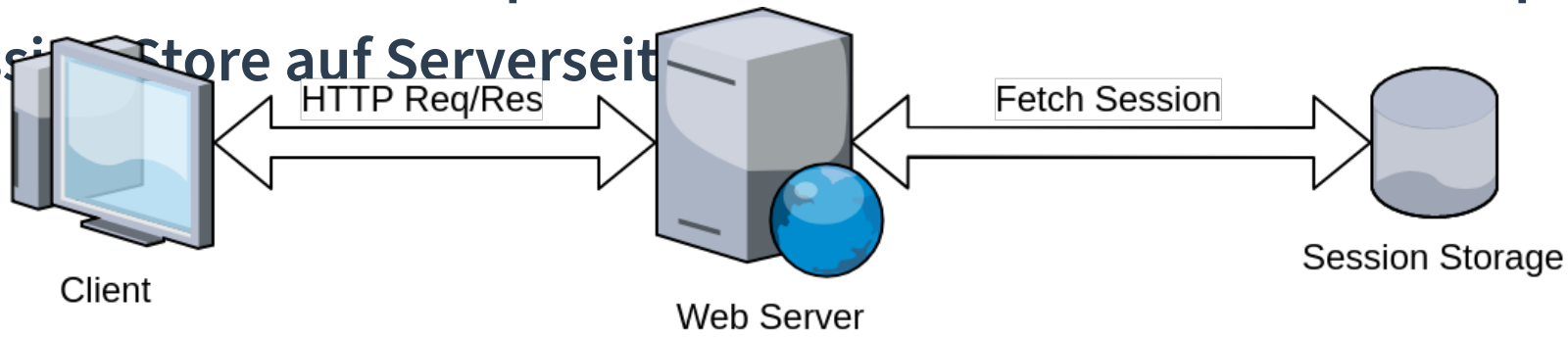
- Eine Session beschreibt einen zustandsbehafteten Vorgang zwischen Client und Server
- Es gibt zwei Arten von Sessions:
  - Server Side Sessions
  - Client Side Sessions
- Um eine Session zu starten wird einmalig Authentifiziert
- Danach reicht das Vorzeigen der Session um als authentifiziert zu gelten

Sessions: Keeping state in a stateless world

- **Verwendung von Sessions:**
  - Logins / Eingeloggt bleiben
  - Warenkorb
  - Bestellvorgang
  - User Tracking
- **Mithilfe von Sessions wird also das Stateless-Prinzip umgangen und nachfolgende Request mit einem vorangegangen Verknüpft**
- **Session Lifetime definiert wie lange die Session „lebt“**
  - Stunden, Tage, Jahre, bis zum Tab schließen

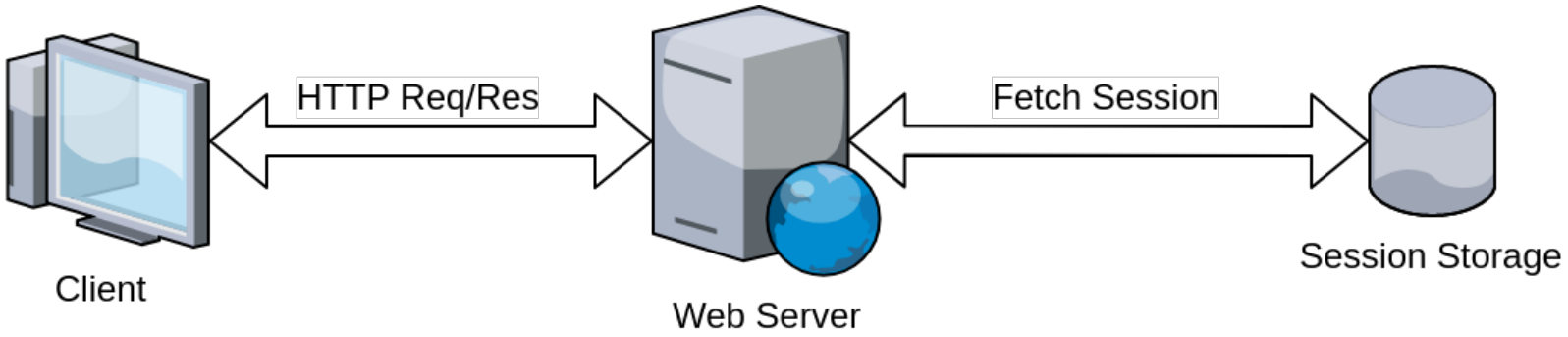
Server Side Sessions

- **Sessioninformationen werden Serverseitig im Session Store abgelegt**
  - Z.B. Datenbank, Session-File, Redis InMemory
- **Client erhält nur die SessionID**
  - z.B. als Cookie oder als URL-Parameter
- **Session-Relevante Request erfordern immer einen Roundtrip zum Sessi Store auf Serverseit**



Server Side Sessions

- **Server Side Session Flow:**
  - Client schickt Request mit Session ID zu Server
  - Server extrahiert Session ID von Request
  - Server sucht Session ID in Session Store
  - Session Store lifert Session Payload zurück
  - Server erzeugt gewünschte Antwort für Client in Abhängigkeit der Session



Server Side Sessions

- **Vorteile Server Side Sessions:**
  - Relativ einfach zu verstehen und straight forward zu implementieren
  - Grundsätzlich auch für sensible Session Daten geeignet
- **Nachteile Server Side Sessions:**
  - Performance Impact, da jeder Request mit Session ID ein Nachfragen am Session Store erzeugt
  - Skaliert schlecht: Werden statt 10 später 100 Server-Instanzen benötigt, müssen trotzdem alle mit einem synchronisiertem Session Store reden (single point of failure)

Client Side Session

- Die Sessioninformationen werden komplett dem Client übergeben
- Serverseitig wird keine Information gespeichert
- Bei jedem Request wird die Session komplett übertragen und muss vom Server validiert werden

Client Side Session

- **Die Sessioninformationen müssen gegen Veränderung gesichert werden**
  - Dem Client ist nicht zu trauen
  - Er könnte den Inhalt der Session beliebig ändern
- **Lösung: Hashed Message Authentication Code (HMAC)**

Hashed Message Authentication Code (HMAC)

- **Grundidee:**
  - An einen bekannte Payload wird ein Secret angehängt, danach wird von den Daten ein Hash erzeugt. Anschließend wird die Payload zusammen mit dem Hash an den Client übergeben.
  - Der Client übergibt beim nächsten Request den Payload und Hash zurück an den Server
  - Der Server kann jetzt prüfen:
    - Kommt mit dem selben Algorithmus für die zurück geschickte Message der selbe Hash heraus?
    - Wenn Ja: Wurde die Message nicht verändert und der Hash von jemanden erstellt, der über das selbe Secret verfügt.
    - Wenn Nein: Entweder wurde die Message verändert oder mit einem anderen Secret signiert. Der Request wird verworfen.

Hashed Message Authentication Code (HMAC)

- **Vereinfachter Algorithmus:**
  - Payload P
  - Secret K
  - Signatur S = HASH(P+K)
  - Ausgabe: P+S

Hashed Message Authentication Code (HMAC)

- **Beispiel:**
  - Session Payload:
    ```
    { "userID": "17", "basket": [] }
    ```
  - Secret Key: yada123
  - Payload + Key:
    ```
    { "userID": "17", "basket": [] }yada123
    ```
  - SHA256-Hash von Payload+Key:
    ```
    3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf
    ```
  - Session-Cookie das zum Client geschickt wird:
    ```
    { "userID": "17", "basket": [] }
    3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf
    ```

Hashed Message Authentication Code (HMAC)

- **Serverseitige überprüfung des Session-Cookie:**
  - Erhaltenes Session-Cookie:
    `{ "userID": "17", "basket": [] }`
    `3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf`
  - Trennen von Hash und Payload:
    - `{ "userID": "17", "basket": [] }`
    - `3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf`
  - Erzeugen des Kontrollhashes: Payload+SecretKey
    `{ "userID": "17", "basket": [] }yada123`
    Ergibt:
    `3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf`
  - Erzeugten Hash und geschickten Hash vergleichen:
    `3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf`
    `3c2d5d17b28fcb46af35b0c257ae09bcf511c3c8bab787ab8f1c18efdc9687cf`
  - Ergebnis:
    - Session Payload unverändert!

Hashed Message Authentication Code (HMAC)

- **Im produktiven Umfeld wird der Payload typischerweise Base64 encoded**
- **Trennung von Payload und Hash ist einfach, da für eine definierte Hash-Funktion die Länge des Hashes bekannt ist.**
- **Wichtig: Der Payload ist NICHT verschlüsselt! Nur gegen Veränderung gesichert!**
  - Nur bei Verwendung von HTTPS/TLS kann die Session nicht einfach eingesehen werden
  - Client-Seitige Angriffe werden mit HttpOnly- und Secure-Cookies verhindert

Client Side Sessions

- **Vorteile:**
  - Stateless. Jede Server-Instanz kann ohne zusätzlichen Dienst die Session validiern und auslesen
  - Session Store eingespart (Bei vielen Session [z.B. Facebook] ist das signifikant)
- **Nachteile:**
  - Deutlich komplexer durch HMAC
  - In der größe Limitiert: Cookies dürfen max. 4kb groß sein
  - Große Sessions blähen jeden Request unnötig auf
  - Keine shared Session über Device-Grenzen

JSON Web Tokens

Stateless Session Tokens

JSON Web Tokens (JWT)

- JWT ist ein in RFC7519 standartisiertes Dateiformat zum Austausch von sog. „Claims“
- Im Context von JWT ist ein Claim der „Anspruch“ auf eine bestimmte Eigenschaft
- Der Aussteller eines JWT versichert durch digitale Signatur oder HMAC das der empfänger des JWTs die darin enthaltenen Claims tatsächlich besitzt
- Durch digitale Signatur/HMAC sind JWTs gegen Veränderung geschützt (vgl. Client Side Sessions)
  Optional sogar verschlüsselt
- JWTs sind komplett stateless, d.H. können i.d.R. ohne zuätzliche Rückfragen validiert und ausgewertet werden
  - Ausnahme: Shared Secret bzw. anfordern des Public Key zu einer Signatur

JSON Web Tokens

- **Aufbau eines JWT**
  - JSON Object Signing and Encryption (JOSE) Header
  - JSON Web Signature (JWS) Payload
  - JWS Signature
- **Die einzelnen Teile werden base64 encoded und mit einem „.“ (Punkt) als Trennzeichen zusammen gefügt:**
  - EyJ0eXAiOiJKV1QiLA0KICJhbGciOiJIUzI1NiJ9
    .
    eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFtcGxlLmNvbS9pc19yb290Ijp0cnVlfQ
    .
    dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk

JWT: Wichtige JOSE Header

- **„alg“: Welcher Algorithmus für Signatur und/oder Verschlüsselung (optional) wird verwendet**
  - HS256 = HMAC mit SHA256
  - ES256 = Public Key mit ECDSA und Kurve P-256
  - ECDH-ES+A128KW = ECDH mit EphemeralStatic keys, AES128 mit KeyWrap
- **„kid“: Welche Schlüssel-ID wurde verwendet**
- **„typ“: Um welchen Typ von Token handelt es sich**
- **„cty“: Content-Type, welcher Mediatype hat der Payload**

JWT: JWS Payload

- **Bei JWT ist der Payload selbst ein JSON Objekt mit den Claims**
  - „iss“: Issuer, wer hat den Token ausgestellt
  - „sub“: Subject, für wen wurde der Token ausgestellt
  - „nbf“, „exp“: Not before, Expiration, zeitliche Gültigkeit
  - „aud“: Audience, wem soll der Token präsentiert werden
  - „iat“: Issued at, wann wurde der Token erzeugt
  - „jti“: JWT ID, eindeutige ID dieses Tokens
  - Beliebige weitere „Public Claims“ und „Private Claims“

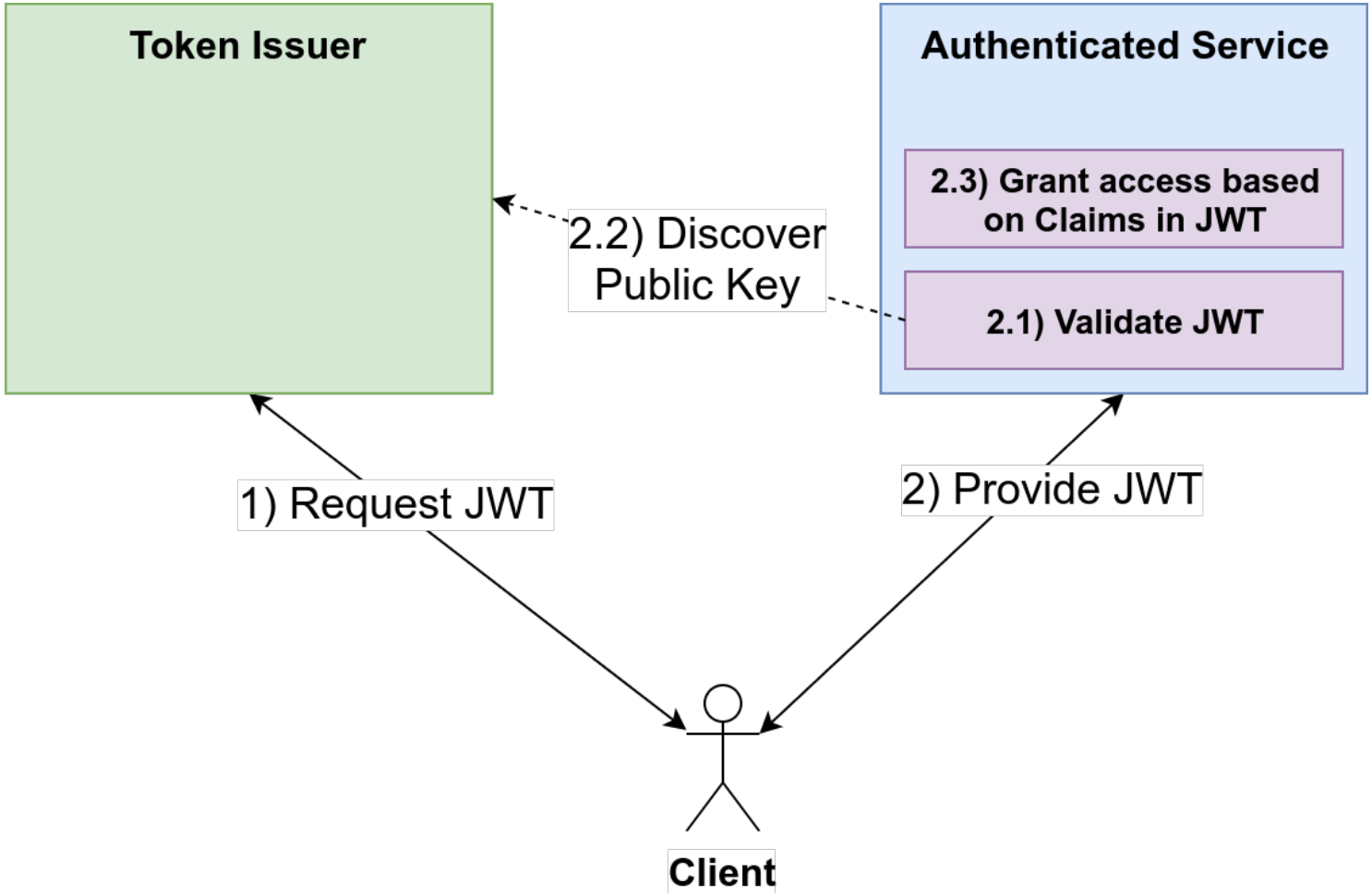
JWT: JWS Signatur

- Bei der Signatur werden Header und Payload aneinander gehängt und Base64 encoded
- Der Base64 String wird dann mithilfe des SharedSecrets oder des PrivateKey erweitert
- Daraus wird dann der SHA256 hash gebildet, dieser entspricht dann der Signatur

JWT: Beispiel Token

```
// Header
{
  "typ": "JWT",
  "alg": "HS256"
}
// Payload
{
  "iss": "joe",
  "exp": 1300819380,
  "http://example.com/publicClaim": true,
  "isAdmin": false,
  "dateOfBirth": "1990-01-01"
}
```

JWT: Infrastruktur



JWT: Enschränkungen

- **Ein ausgestelltes Token bleibt gültig bis es ab läuft**
  - Ausnahme: Reference-Tokens, diese sind aber nicht mehr stateless
  - Zurückziehen via JWT ID technisch möglich, erfordert aber einen Roundtrip zum Aussteller bei jedem Request
- **Änderung der Claims ist nur möglich, wenn der User ein neues Token beantragt**
- **Lösung: Kurzlebige Tokens (z.B: eine Stunde)**
- **Neues Problem: User muss sich nach dieser Zeit neu authentifizieren**
  - Lösung: Refresh-Tokens

JSON Web Tokens: Refresh Tokens

- **Refresh-Tokens sind NICHT Bestandteil des JWT Standards**
- **Idee: Beim ausgeben eines JWT wird auch ein Refresh-Token erzeugt. Mithilfe dieses Refresh-Tokens kann jederzeit ein neuer JWT beantragt werden**
- **Dadurch kann der User eingeloggt bleiben. Änderungen an seinen Claims werden dann bei der nächsten JWT-Austellung berücksichtigt**
- **Problem: Refresh-Token wird zum Ersatz für „Username/Password“, muss also sicher aufbewahrt werden**
  - Es gibt Single-Use Refresh-Tokens
  - Aber auch Multi-Use Refresh-Tokens

How to store a password?

Wie sichert man ein Passwort im Backend?

- **Big no no 1: Passwörter als plain text**
- **Big no no 2: „Verschlüsselt“ speicher**
  - Problem: Wenn das Programm die Daten entschlüsseln kann, muss der Schlüssel zugänglich sein. Dann kann auch ein Angreifer den Schlüssel erhalten.
- **Lösung: Passwort Hashes**

Passwort Hashes

- **Das Passwort wird durch eine Hash-Funktion geschickt**
  - Hash-Funktionen sind Einweg-Funktionen, d.H. vom Hash kommt man nicht trivial zu den Eingabedaten
  - Problem: Der selbe Input führt auch immer zum selben Hash. Dadurch ergibt sich eine Angriffsmöglichkeit für „Rainbow Tables“
  - Lösung: Salted-Hashes
    Das Passwort wird mit einer zusätzlichen Priese „Salz“ gehasht. Dadurch ergibt sich in öffentlichen Rainbow Tables kein Treffer mehr

Salted Hashes

- **Für einen „salted hash“ wird an den gegeben Input (hier ein Passwort) einfach eine weitere Zeichenkette angefügt (egal ob vorne oder hinten)**
  - User-Input: **Pa$$w0rd** Salt: **QLRM2z**
    Input für Hash-Funktion: **Pa$$w0rdQLRM2z**
    Erzeugter SHA-256 Hash:
    `8e873b70197c20905eaa38d81868adafa5de24ae85fdc5f1cc8cfd27f7c8e8d2`

    Vergleich: SHA-256 Hash von „Pa$$w0rd“:
    `97c94ebe5d767a353b77f3c0ce2d429741f2e8c99473c3c150e2faa3d14c9da6`
  - Der „salted hash“ kann jetzt – zusammen mit dem Salt – in der Datenbank abgelegt werden
  - Jedes Passwort sollte mit einem eigenen zufälligen Salt versehen werden um gezielte Rainbow Tables für den jeweiligen Dienst zu vermeiden