



Technische Hochschule  
Ingolstadt  
Fakultät Informatik

# Softwaresicherheit & Security Testing

## Kapitel 3: Security by Design

Prof. Dr.-Ing. Hans-Joachim Hof

## *Ziele der heutigen Veranstaltung*



- **Studierende können das Prinzip „Security by Design“ in implementierungsnahen Fragestellungen erfolgreich anwenden um eigene Anwendungen zu schützen.**

- **Definition:** Security by Design bezeichnet das Prinzip, Security in allen Phasen der Softwareentwicklung zu berücksichtigen, insbesondere von Anfang der Entwicklung an.
- **Heute oft “Security as an Afterthought”**
  - Kurze Time-to-Market wichtigstes Ziel
  - Security-Betrachtungen, wenn Zeit ist oder reaktiv, wenn Vorfälle auftreten
  - Erfahrung zeigt, dass das nicht zu sicherer Software führt
    - Frage: Warum?
- **Agile Methoden**
  - Klar abgegrenzte Phasen für Softwareentwicklung existieren nicht mehr
  - Schnelle Reaktionen auf ändernde Randbedingungen und Kundenwünsche
  - Frage: was bedeutet das für Security?

- **Security by Design betrifft die ganze Software-Entwicklung**
  - Gesamter Security-Prozess ausführlich betrachtet in der Vorlesung „Security Architekturen & Security Engineering“ im nächsten Semester
  - Security by Design als Basis für Security-Architekturen ebenfalls in dieser Vorlesung
- **In dieser Vorlesungseinheit Fokus auf den implementierungsnahen Design-Aspekten**

# *Implementierungsnahe Design-Aspekte*

## *Überblick*



- **Minimierung der Angriffsflächen**
- **Security by Default**
- **Minimale Rechte**
- **Sichere Fehlerbehandlung**
- **Vertrauensdesign**
- **Trennung von Verantwortlichkeiten**
- **Keine Security by Obscurity**
- **Komplexitätsreduktion**
- **Verwendung von kryptographischen Methoden in Software**
- **Software-Updates**
- **Sichere Datenspeicherung auf unsicheren Medien**



## *Minimierung der Angriffsfläche*

### *Übersicht*

- **Angriffsfläche aus Sicht einer Software-Komponente (nach Domäne):**
  - APIs und Interfaces zu anderen Software-Komponenten
  - User Interfaces (einzelne Komponenten)
  - Konfigurationsdaten
  - Datenbankzugriff

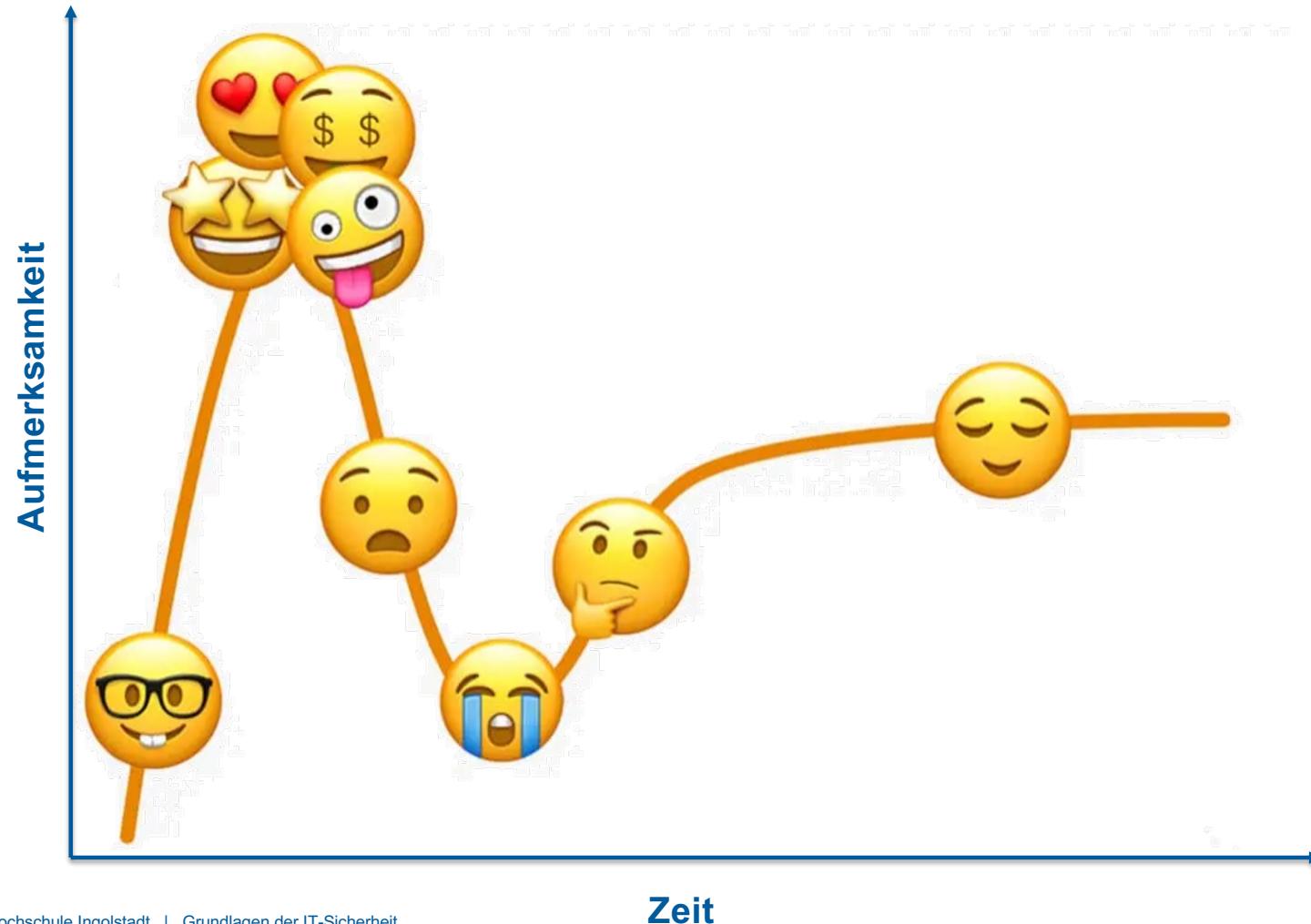


- **Definition:** Eine Application Programming Interface (API) ist eine Schnittstelle, die Funktionalität zur Verfügung stellt, über die verschiedene Anwendungen miteinander kommunizieren können. Dazu definiert eine API üblicherweise einen Satz von Befehlen und Funktionen sowie die zugehörigen Parameter, die genutzt werden können.
- Im Web wird Funktionalität häufig in Form von APIs zur Verfügung gestellt
- **Verschiedene Arten von APIs:**
  - Internal API/Private API: sollen nur von internen Systemen verwendet werden
  - Customer API/Partner API: eingeschränkter Kreis von Nutzern jenseits der Unternehmensgrenze
  - Public API: öffentlich nutzbare API



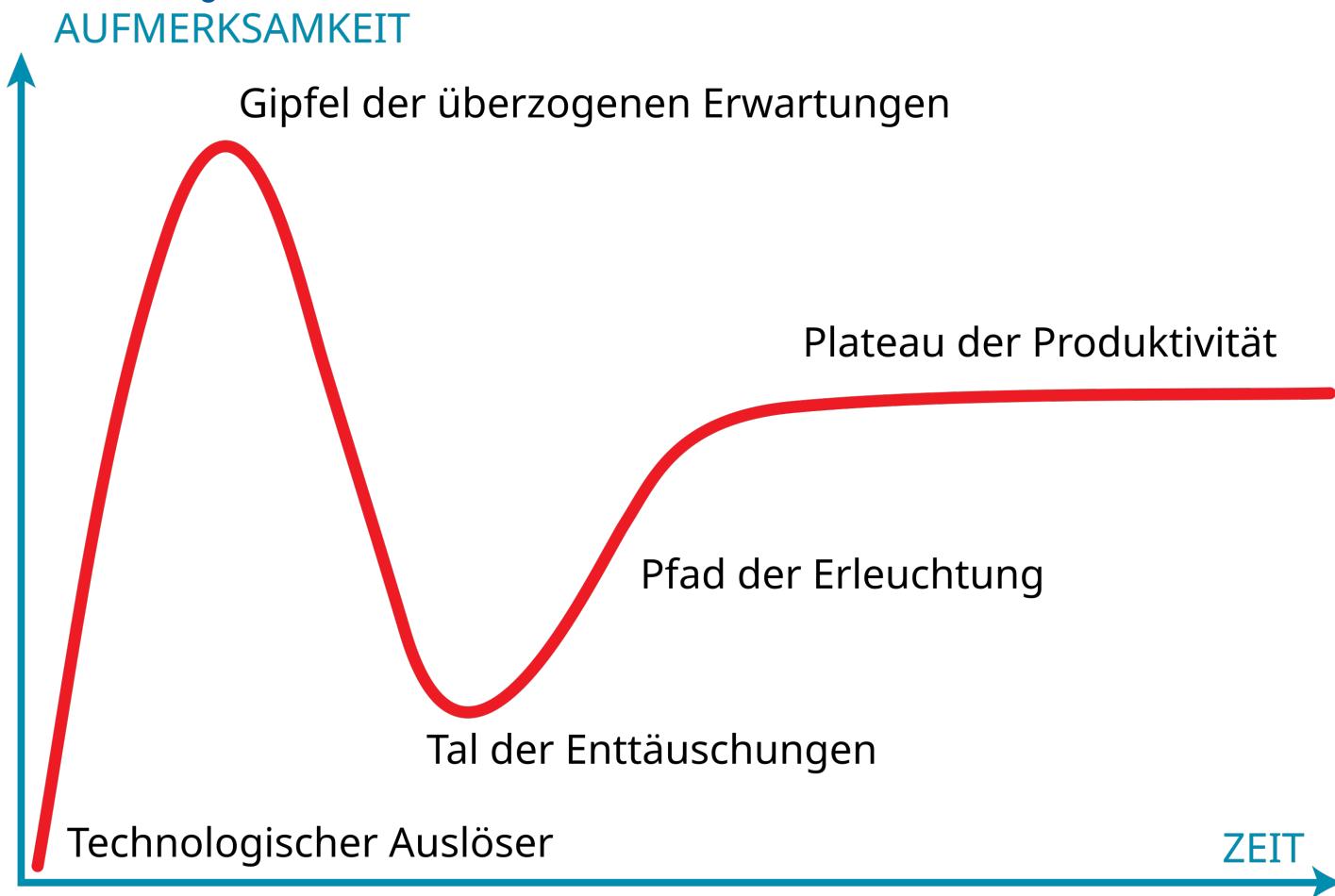
## Gartner Hype Cycle für einen Tochnologiebereich (z.B. API)

Meme Version



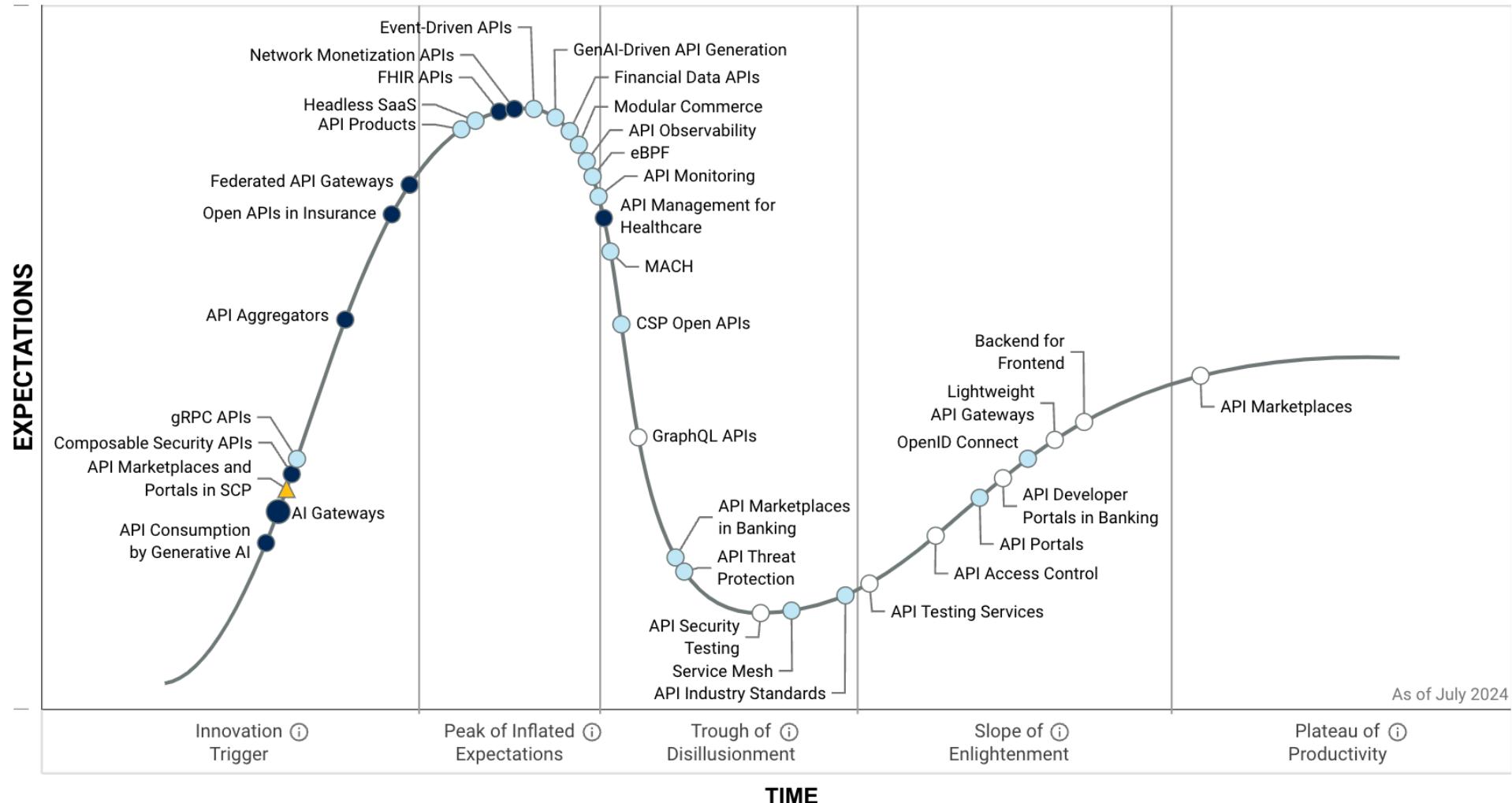
## Gartner Hype Cycle für einen Technologiebereich (z.B. API)

Wissenschaftliche Darstellung





## Gartner Hype Cycle for API, 2024





## Beispiel API

### Google Maps API (Public API)

In Dokumenten nach den Hauptfunktionen der Maps JavaScript API suchen

<b>Kartentypen</b> Straßen-, Satelliten-, Hybrid-, Gelände- und benutzerdefinierte Karten anzeigen	<b>Lokalisierung</b> Text auf der Karte automatisch in über 40 Sprachen lokalisieren	<b>Markierungen</b> Sie können die Standardmarkierungen von Google Maps oder benutzerdefinierte Markierungen verwenden.	<b>Infofenster</b> Markierungen mit interaktiven Infofenstern mehr Kontext und zusätzliche Informationen hinzufügen	<b>Formen</b> Mit den integrierten Funktionen können Sie eine Vielzahl von Formen auf der Karte zeichnen, darunter <a href="#">Polylinien</a> und <a href="#">Polygone</a> .	<b>Benutzerdefinierte Overlays</b> Benutzerdefinierte Overlays erstellen, um Daten, Bilder und mehr auf der Karte anzuzeigen
<b>UI-Steuerelemente</b> Die UI-Steuerelemente anpassen, die auf der Karte angezeigt werden	<b>Ereignisse</b> Code schreiben, der auf Nutzerinteraktionen und Lebenszyklus-Ereignisse reagiert	<b>WebGL-Overlay</b> Mithilfe von WebGL aussagekräftige 3D- und 2D-Grafiken auf der Vektorbasiskarte erstellen	<b>Boden-Overlays</b> Eigene benutzerdefinierte Bilder als Overlays verwenden, die beim Schwenken und Zoomen der Karte automatisch synchronisiert werden	<b>Data-Ebene</b> GeoJSON und andere Datentypen in einer Vielzahl von Formaten auf der Karte anzeigen	<b>Benutzerdefinierte Stile</b> Fast alle visuellen Aspekte der Karte anpassen
<b>Infofenster</b> Markierungen mit interaktiven Infofenstern mehr Kontext und zusätzliche Informationen hinzufügen	<b>Formen</b> Mit den integrierten Funktionen können Sie eine Vielzahl von Formen auf der Karte zeichnen, darunter <a href="#">Polylinien</a> und <a href="#">Polygone</a> .	<b>Benutzerdefinierte Overlays</b> Benutzerdefinierte Overlays erstellen, um Daten, Bilder und mehr auf der Karte anzuzeigen	<b>Neigung und Drehung</b> Vektorbasiskarte programmatisch in drei Dimensionen neigen und drehen	<b>Markierungscluster</b> Markierungen gruppieren, um die Oberfläche übersichtlicher zu gestalten	<b>Heatmaps</b> Die Datendichte an geografischen Punkten visualisieren



# Beispiel API

## Google Maps API (Public API)

HTML ▾

```
1  <!doctype html>
2  <!--
3  @license
4  Copyright 2019 Google LLC. All Rights Reserved.
5  SPDX-License-Identifier: Apache-2.0
6  -->
7  <html>
8  <head>
9  <title>Simple Map</title>
10 <script src="https://polyfill.io/v3/polyfill.min.js?features=default"></script>
11 <!-- jsFiddle will insert css and js -->
12 </head>
13 <body>
14 <div id="map"></div>
15
16 <!-- prettier-ignore -->
17 <script>(g=>{var h,a,k,p="The Google Maps JavaScript API",c="google",l="importLibrary",q="__ib__",
m=document,b=window;b[b[c]]||b[c]={}};var d=b.maps||b.maps={},r=new Set,e=new URLSearchParams,u=()=>
h||ch=new Promise((e,f)=>f(resolve,(o,m)=>createElement("script").src=o).set("libraries",l).set("callback",m).onload=resolve).then(f).catch(e))</script>
```

CSS ▾

```
1  /**
2  * @license
3  * Copyright 2019 Google LLC. All Rights Reserved.
4  * SPDX-License-Identifier: Apache-2.0
5  */
6  /*
7  * Always set the map height explicitly to define the size of the div element
8  * that contains the map.
9  */
10 <#map {
11   height: 100%;
12 }
13
14 /*
15 * Optional: Makes the sample page fill the window.
16 */
17 <html,
18 <body {
19   height: 100%;
```

JavaScript + No-Library (pure JS) ▾

```
1  /**
2  * @license
3  * Copyright 2019 Google LLC. All Rights Reserved.
4  * SPDX-License-Identifier: Apache-2.0
5  */
6  let map;
7
8  async function initMap() {
9    const { Map } = await google.maps.importLibrary("maps");
10
11  map = new Map(document.getElementById("map"), {
12    center: { lat: -34.397, lng: 150.644 },
13    zoom: 8,
14  });
15
16 initMap();
```



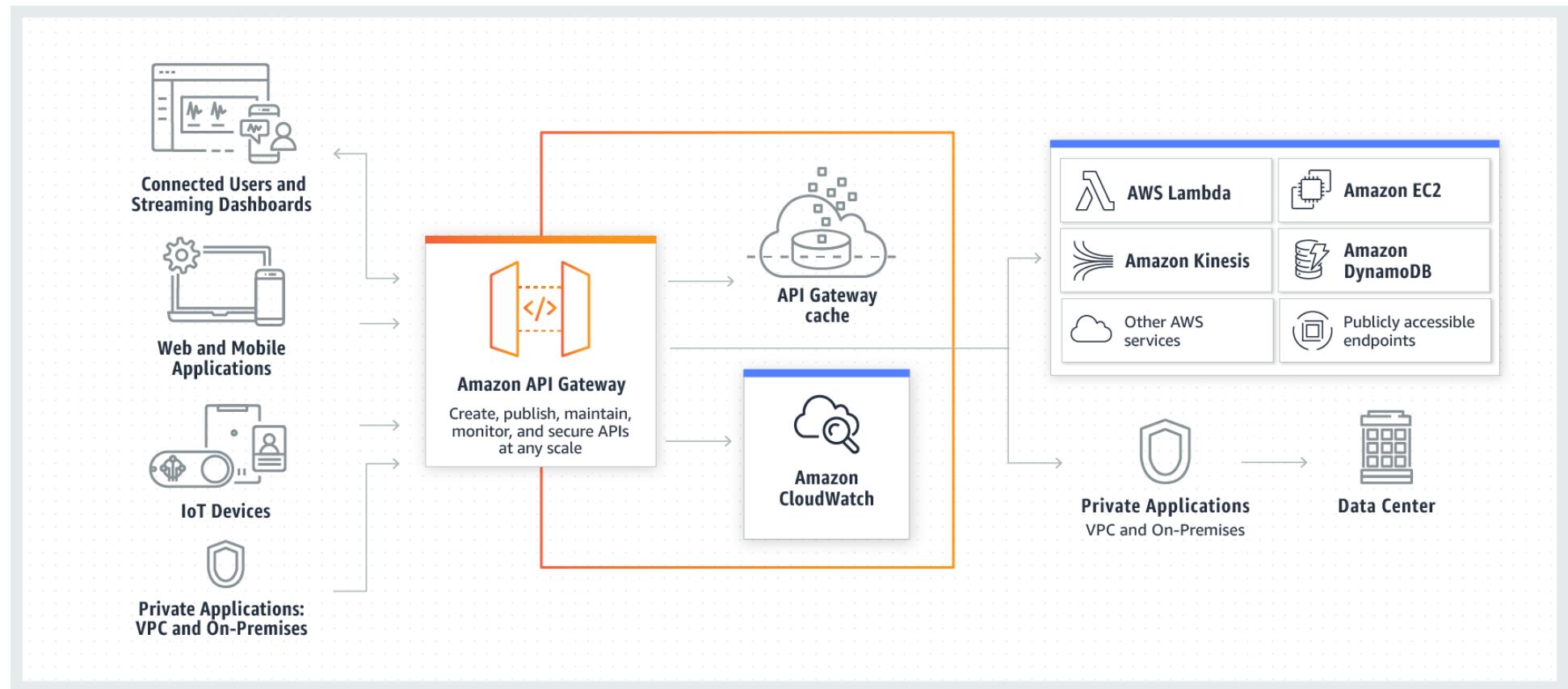
## Reduzierung Angriffsfläche APIs

### Methoden

- **Anzahl der angebotenen APIs reduzieren**
  - Heißt auch: ältere Versionen API nur begrenzte Zeit parallel pflegen
- **Zugriffsrechte möglichst restriktiv regeln**
  - Braucht es eine Public API oder auch Customer API/Partner API oder Internal API möglich?
- **API-Gateways einsetzen**
  - Definition: ein API-Gateway ist ein Dienst, der als zentraler Vermittler für API-Zugriffe dient und diese an Backend-Dienste weiterleitet. Typische Aufgaben eines API-Gateways beinhalten Benutzerverwaltung, Überwachung und Analyse des Datenverkehrs, Durchsetzung von Richtlinien (engl. Policy) und Authentifizierung/Autorisierung
- **Typische Risiken in APIs vermeiden (siehe übernächste Folie)**

## Exkurs: API-Gateway (...Security bei Design im Großen...)

Beispiel: AWS API-Gateway



- Quelle: <https://aws.amazon.com/de/api-gateway/>

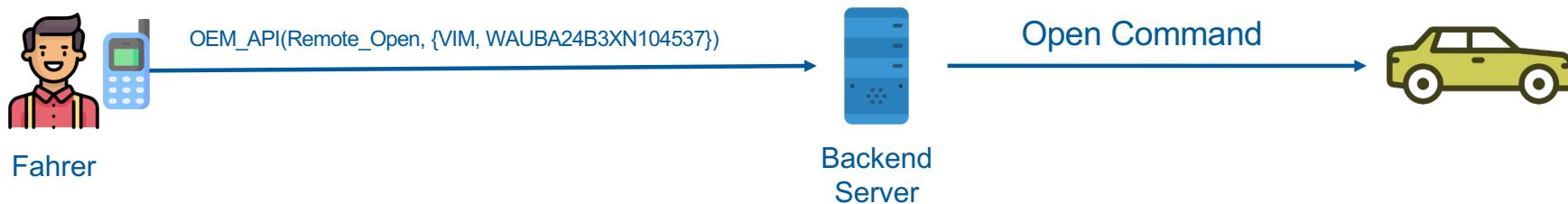


- Liste mit den 10 häufigsten Security-Risiken für APIs
  1. Broken Object Level Authorization
  2. Broken Authentication
  3. Broken Object Property Level Authorization
  4. Unrestricted Resource Consumption
  5. Broken Function Level Authorization
  6. Unrestricted Access to Sensitive Business Flows
  7. Server Side Request Forgery
  8. Security Misconfiguration
  9. Improper Inventory Management
  10. Unsafe Consumption of APIs



- **Stellt Problem auf allen Ebenen dar:**
  - Broken Object Level Authorization
  - Broken Object Property Level Authorization
  - Broken Function Level Authorization
- **Definition: Mit Autorisierung wird die Zuweisung von Rechten an eine Identität sowie die Durchsetzung dieser Rechte bezeichnet.**
- **Definition: Mit Authentifizierung bezeichnet die Überprüfung der Echtheit einer vorgegebenen Identität. Nach erfolgreicher Authentifizierung liegt eine sichere Identität vor.**
- **Eine sichere Authentifizierung ist also die Basis für eine sichere Autorisierung**
  - Broken Authentication

- **Beispiel Angriffsszenario: Ein Automobilhersteller ermöglicht eine Fernsteuerung von Fahrzeugfunktionen über eine App. Die Funktionalität wird der App über eine API im Backendsystem des Herstellers bereitgestellt. Parameter der API ist die Vehicle Identification Number (VIN). Der Security Mangel ist, dass die App nie überprüft, ob die übergebene VIN zum Benutzer der App gehört.**

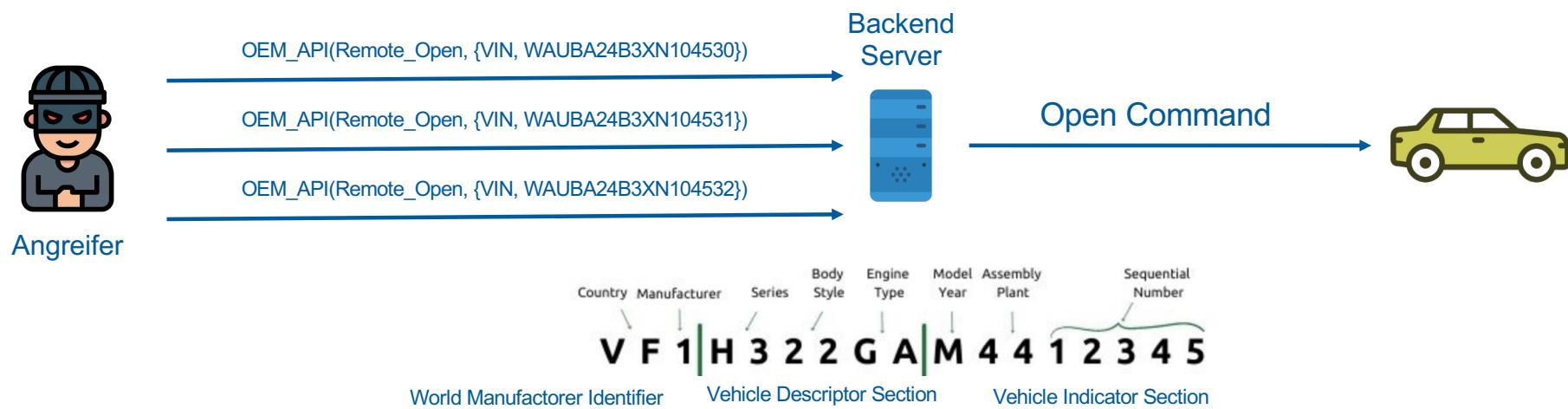




## OWASP API Security Top 10 (2023)

### Broken Object Level Authorization

- Beispiel Angriffsszenario: Ein Automobilhersteller ermöglicht eine Fernsteuerung von Fahrzeugfunktionen über eine App. Die Funktionalität wird der App über eine API im Backendsystem des Herstellers bereitgestellt. Parameter der API ist die Vehicle Identification Number (VIN). Der Security Mangel ist, dass die App nie überprüft, ob die übergebene VIN zum Benutzer der App gehört.





## OWASP API Security Top 10 (2023)

### Broken Object Level Authorization



Drivers Side Door Jam



Angreifer

OEM\_API(Remote\_Open, {VIM, WAUBA24B3XN104537})



Backend  
Server

Open Command





- Das Problem im Beispiel entsteht durch die fehlende Begrenzung der Zugriffsrechte des Benutzers auf Objekte (CWE-285, CWE-639)
  - Benutzer ist dazu autorisiert, Autos zu öffnen, basierend auf VIN
  - Es fehlt die Begrenzung des Rechts „Auto öffnen“ auf die dem Benutzer zugeordneten Autos (Objekte)
- Weiteres Problem: Enumeration
  - Ist der Aufbau der VIN bekannt, können VINS durchprobiert werden
  - Aufbau VIN siehe vorne

- **Least Privilege: Grundsätzlich so wenig Rechte wie möglich vergeben**
  - Falls möglich für Benutzer mitschreiben, auf welche Objekte er Zugriff hat
- **Verwendete IDs zur Identifizierung von Objekten sollten lange (128 Bit+), zufällige und unvorhersagbare GUIDs sein**
  - GUID = Globally Unique ID
  - UUID = Universally Unique Identifier
  - Zur Erzeugung von UUIDs nach RFC 4122 steht in Python die Bibliothek `uuid` zur Verfügung, verschiedene Methoden der Herstellung
- **Sicherstellen, dass Zugriffsberechtigung bei jedem Zugriff überprüft wird, keine Zugriffsrechte cachen.**
- **Ausführlich beschreiben im OWASP Authorization Cheat Sheet [1]**



## Broken Object Level Authorization

### Erzeugung UUID

```
>>> import uuid
>>> # make a UUID based on the host ID and current time
>>> uuid.uuid1()
UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

>>> # make a UUID using an MD5 hash of a namespace UUID and a name
>>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

>>> # make a random UUID
>>> uuid.uuid4()
UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

>>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
>>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')

>>> # make a UUID from a string of hex digits (braces and hyphens ignored)
>>> x = uuid.UUID('{00010203-0405-0607-0809-0a0b0c0d0e0f}')

>>> # convert a UUID to a string of hex digits in standard form
>>> str(x)
'00010203-0405-0607-0809-0a0b0c0d0e0f'

>>> # get the raw 16 bytes of the UUID
>>> x.bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f'

>>> # make a UUID from a 16-byte string
>>> uuid.UUID(bytes=x.bytes)
UUID('00010203-0405-0607-0809-0a0b0c0d0e0f')
```



## *Broken Object Level Authorization*

### *Aufgabe*

- Szenario: Eine E-Commerce-Plattform für Online-Shops (Shops) stellt eine Auflistungsseite mit den Umsatzdiagrammen für die von ihnen gehosteten Shops bereit. Durch die Untersuchung der Browseranfragen kann ein Angreifer die API-Endpunkte, die als Datenquelle für diese Diagramme verwendet werden, und deren Muster identifizieren: **/shops/{shopName}/revenue\_data.json**.
- Frage: wo könnte hier ein Angreifer ansetzen?



- Benutzer können auf Eigenschaften von Objekten zugreifen, obwohl diese nur innerhalb der Anwendung zugreifbar sein sollten. (CWE-213, CWE-915)
- Beispiel: Auf einer Videowebseite werden Videos automatisch geblockt. Auch bei geblockten Videos können Benutzer die Beschreibung ihres Videos mit folgendem API Request ändern:

```
PUT /api/video/update_video

{
    "description": "a funny video about cats"
}
```

- Das Video hat als Eigenschaft eine boolsche Variable „blocked“, wenn diese true ist, dann ist das Video blockiert. Da der Zugriff auf diese Eigenschaft nicht autorisiert wird, funktioniert folgender API Request:

```
{
    "description": "a funny video about cats",
    "blocked": false
}
```



- **Funktionen vermeiden, die automatisch Input des Nutzers an Variablen im Code, interne Objekte oder interne Objekteigenschaften zuweisen („Mass Assignment“)**
- **Möglichst wenig Domain Objekte serialisiert weitergeben (`to_json()`, `to_string()`) und Nutzung von Data Transfer Objects**
  - mit nur den Feldern, die der Benutzer ändern kann, definieren
  - Daran das Mass Assignment durchführen
  - Erfordert dann übersetzung von/nach Domain Objekt, welches auch die internen Felder besitzt
- **Änderungen von Objekt-Eigenschaften überwachen und nur zulassen, wenn gewünscht**



Suppose there is a form for editing a user's account information:

## Beispiel

```
<form>
    <input name="userid" type="text">
    <input name="password" type="text">
    <input name="email" type="text">
    <input type="submit">
</form>
```

Here is the object that the form is binding to:

```
public class User {
    private String userid;
    private String password;
    private String email;
    private boolean isAdmin;

    //Getters & Setters
}
```

Here is the controller handling the request:

```
@RequestMapping(value = "/addUser", method = RequestMethod.POST)
public String submit(User user) {
    userService.add(user);
    return "successPage";
}
```

## Beispiel



Here is the typical request:

```
POST /addUser
...
userid=bobbytables&password=hashedpass&email=bobby@tables.com
```

And here is the exploit in which we set the value of the attribute `isAdmin` of the instance of the class `User`:

```
POST /addUser
...
userid=bobbytables&password=hashedpass&email=bobby@tables.com&isAdmin=true
```



## Beispiel

### Data Transfer Object

```
public class UserRegistrationFormDTO {  
    private String userid;  
    private String password;  
    private String email;  
  
    //NOTE: isAdmin field is not present  
  
    //Getters & Setters  
}
```



■ **Privilegierte Funktionen sind auch unprivilegierten oder anonymen Nutzern zugänglich (CWE-285)**

■ **Beispiel:** Während des Registrierungsprozesses für eine Anwendung, die nur eingeladenen Benutzern die Teilnahme ermöglicht, löst die mobile Anwendung einen API-Aufruf aus GET `/api/invites/{invite_guid}`.

Die Antwort enthält einen JSON-Code mit Details zur Einladung, einschließlich der Rolle des Benutzers und der E-Mail-Adresse des Benutzers.

Ein Angreifer dupliziert die Anfrage und manipuliert die HTTP-Methode und den Endpunkt zu POST `/api/invites/new`. Dieser Endpunkt sollte nur von Administratoren über die Admin-Konsole aufgerufen werden. Der Endpunkt implementiert keine Berechtigungsprüfungen auf Funktionsebene.

Der Angreifer nutzt das Problem aus und sendet eine neue Einladung mit Administratorrechten:

```
POST /api/invites/new

{
  "email": "attacker@somehost.com",
  "role": "admin"
}
```



- **Die Anwendung sollte ein Autorisierungs-Modul nutzen, welches von allen Funktionen aufgerufen wird mit folgenden Eigenschaften:**
  - Per Default Zugriff ablehnen, Zugriffsrechte müssen explizit vergeben werden
- **Beispielsweise realisieren, indem alle Administrations-Controller abgeleitet werden von einem Abstract Administrations-Controller, der das Autorisierungs-Modul nutzt**



- **Sichere Authentifizierung notwendig, um Rechte effektiv umzusetzen**
  - Sonst: Impersonifizierung möglich
- **Umsetzung Authentifizierung in APIs**
  - Signaturbasiert
  - HMAC-basiert
  - Passwortbasiert
  - Access Tokens Oauth 2.0
  - ...



### ■ Registrierung für die Nutzung der API

#### ■ Generierung Schlüsselpaar (Private Key + Public Key) (alle Beispiele in Typescript)

```
const crypto = require('crypto');

interface KeyPair {
  publicKey: string;
  privateKey: string;
}

function generateCredentials(size = 2048): Promise<KeyPair> {
  return new Promise<KeyPair>((resolve, reject) => {
    crypto.generateKeyPair('rsa', { modulusLength: size },
      (err, pub, priv) => {
        if (err) { return reject(err); }           ← If any errors arise, simply
                                                 reject the promise.
        return resolve({
          publicKey: pub.export({ type: 'pkcs1', format: 'pem' }),
          privateKey: priv.export({ type: 'pkcs1', format: 'pem' })
        });
      });
  });
}
```

Use the string serialized  
data for the public and  
private key values.



### ■ Registrierung für die Nutzung der API

#### ■ Generierung Schlüsselpaar (Private Key + Public Key) (alle Beispiele in Typescript)

```
const crypto = require('crypto');

interface KeyPair {
  publicKey: string;
  privateKey: string;
}

function generateCredentials(size = 2048): Promise<KeyPair> {
  return new Promise<KeyPair>((resolve, reject) => {
    crypto.generateKeyPair('rsa', { modulusLength: size },
      (err, pub, priv) => {
        if (err) { return reject(err); }           ← If any errors arise, simply
                                                 reject the promise.
        return resolve({
          publicKey: pub.export({ type: 'pkcs1', format: 'pem' }),
          privateKey: priv.export({ type: 'pkcs1', format: 'pem' })
        });
      });
  });
}
```

FÄLLT IHNEN ETWAS AUF?

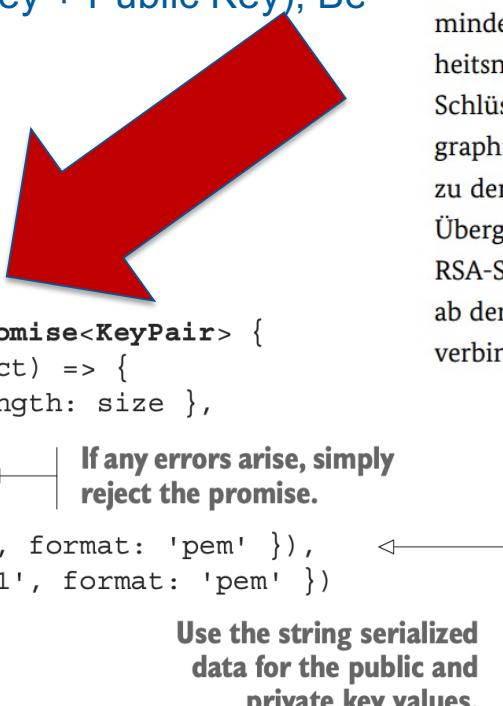
Use the string serialized  
data for the public and  
private key values.

- Registrierung für die Nutzung der API
  - Generierung Schlüsselpaar (Private Key + Public Key), Be

```
const crypto = require('crypto');

interface KeyPair {
  publicKey: string;
  privateKey: string;
}

function generateCredentials(size = 4096): Promise<KeyPair> {
  return new Promise<KeyPair>((resolve, reject) => {
    crypto.generateKeyPair('rsa', { modulusLength: size },
      (err, pub, priv) => {
        if (err) { return reject(err); }           ← If any errors arise, simply
                                                 reject the promise.
        return resolve({
          publicKey: pub.export({ type: 'pkcs1', format: 'pem' }),
          privateKey: priv.export({ type: 'pkcs1', format: 'pem' })
        });
      });
  });
}
```

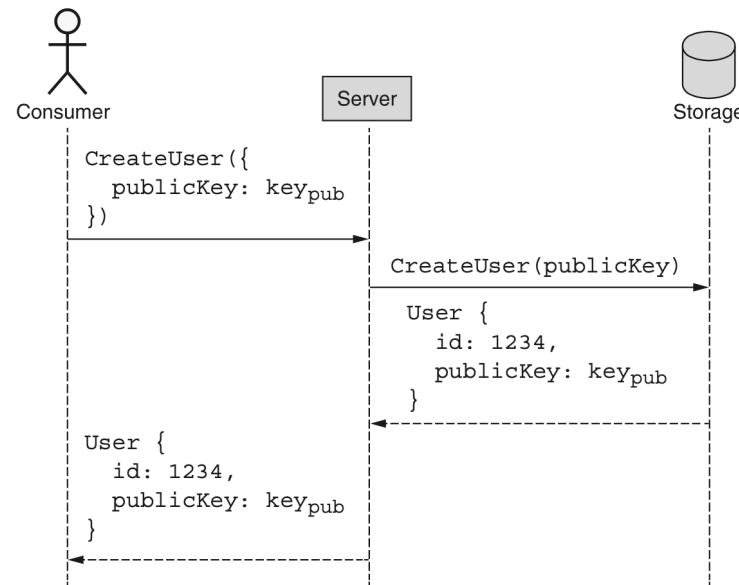


Blockchiffre	MAC	RSA	DH $\mathbb{F}_p$	ECDH	ECDSA
128	128	3000 <sup>a</sup>	3000 <sup>a</sup>	250	250

Tabelle 1.2: Empfohlene Schlüssellängen für verschiedene kryptographische Verfahren.

<sup>a</sup> Für einen Einsatzzeitraum ab dem Jahr 2023 wird durch die vorliegende Technische Richtlinie empfohlen, eine Schlüssellänge von mindestens 3000 Bits zu nutzen, um ein vergleichbares Sicherheitsniveau für alle asymmetrischen Verfahren zu erreichen. Eine Schlüssellänge von  $\geq 3000$  Bits wird ab dem Jahr 2023 für kryptographische DLIES- und DSA-Implementierungen verbindlich, die zu der vorliegenden Technischen Richtlinie konform sein sollen. Übergangsweise bleibt eine Schlüssellänge von  $\geq 2000$  Bits für RSA-Schlüssel bis Ende 2023 konform zur Technischen Richtlinie; ab dem Jahr 2024 wird eine RSA-Schlüssellänge von  $\geq 3000$  Bits verbindlich.

- Registrierung für die Nutzung der API
  - Generierung Schlüsselpaar (Private Key + Public Key)
  - Schlüsselaustausch während Registrierung



- **Registrierung für die Nutzung der API**
  - Generierung Schlüsselpaar (Private Key + Public Key)
  - Schlüsselaustausch während Registrierung
  
- **Signieren**

```
const crypto = require('crypto');

function generateSignature(payload: string, privateKey: string): string {
  const signer = crypto.createSign('rsa-sha256');
  signer.update(payload);
  signer.end();
  return signer.sign(privateKey);
}
```



## Authentifizierung

### Digitale Signatur

- **Registrierung für die Nutzung der API**
  - Generierung Schlüsselpaar (Private Key + Public Key)
  - Schlüsselaustausch während Registrierung
- **Signieren**
- **Signaturüberprüfung:**

```
const crypto = require('crypto');

function verifySignature(
    payload: string, signature: string, publicKey: string): Boolean {
    const verifier = crypto.createVerify('rsa-sha256');
    verifier.update(payload);
    verifier.end();
    return verifier.verify(publicKey, signature);
}
```



## *Authentifizierung*

### *Request Fingerprint*

- **Frage: Welcher Teil eines API Requests soll signiert werden?**



- **Frage: Welcher Teil eines API Requests soll signiert werden?**
- **Serialisierung und Codierung erzeugen eventuell unterschiedliche Ergebnisse, die jedoch semantisch gleich sind.**
  - Beispiel: JSON {"a":1, "b":2} ist semantisch gleich zu {"b":2, "a":1}
  - Problem für Signatur, da diese über eine Byte-Repräsentation erzeugt wird (unterschiedlich)
- **Request Body enthält interessante Daten => zu schützen**
  - Nutzdaten
- **Request Header enthält interessante Daten => zu schützen**
  - z.B. HTTP verb (DELETE, POST, GET, ...)
  - Zeit, wann Request abgesendet wird (Frage: warum wichtig?)



## Authentifizierung

### Request Fingerprint

- Erzeugung Request Fingerprint aus:

Field	Location	Description
Method	First line	Action of the request
Path	First line	Target resource of the request
Host	HTTP header	The destination host for the request
Content	HTTP body	Payload of the request
Date	HTTP header	When the request is created

- Dabei vom HTTP Body nur den Haswert nutzen (Body kann groß sein)
- Das dann signieren

## Authentifizierung

### Erzeugung des Hash-Werts

```
const crypto = require('crypto');

function generateDigestHeader(body: string): string {
    return 'SHA-256=' + crypto
        .createHash('sha256')
        .update(body)
        .digest()
        .toString('base64');
}
```

We prefix the hash with a specification of the hash algorithm used (in this case, SHA-256).

The hash should be Base64-encoded.



## Authentifizierung

### Erzeugung des Request Fingerprints

The “digest” field should generate a hash of the request body.

```
const HEADERS = ['(request-target)', 'host', 'date', 'digest'];

function generateRequestFingerprint(
    request: HttpRequest, headers = HEADERS): string {
    return headers.map((header) => {
        let value;
        if (header === '(request-target') {
            value = `${request.method.toLowerCase()} ${request.path}`;
        } else if (header === 'digest') {
            value = generateDigestHeader(request.body);
        } else {
            value = request.headers[header];
        }
        return `${header.toLowerCase()}: ${value}`;
    }).join('\n');
}
```

The “(request-target)” field should include the lowercase HTTP method and the path.

All other values should be just as specified in the header of the request.

Finally, return a set of key-value pairs, separated by a colon (just like headers are).

**Listing 30.6 Example fingerprint used for signing HTTP requests**

```
(request-target): patch /chatRooms/1
host: example.org
date: Tue, 27 Oct 2020 20:51:35 GMT
digest: SHA-256=HV9PltG0QPRNs11FB7ebQA8XPasvPyRg6hhU0QF214M=
```

The special “(request-target)” field is treated like any other header.

We also include the additional digest header.

Quelle: [2]

## Authentifizierung

### Metainformation für Signaturheader

Field	Description
headers	Ordered list of headers used to generate the request fingerprint
keyId	Unique ID used to choose the public key for verification
signature	Digital signature content to be verified
algorithm	Algorithm used to generate the signature



## Authentifizierung

### Erzeugung des Signature Headers

```
const HEADERS = ['(request-target)', 'host', 'date', 'digest'];

interface SignatureMetadata {
    keyId: string;
    algorithm: string;
    headers: string;
    signature: string;
}

function generateSignatureHeader(
    fingerprint: string, userId: string,
    privateKey: string, headers = HEADERS): string {
    const signatureParts: SignatureMetadata = {           ←
        keyId: userId,
        algorithm: 'rsa-sha256',
        headers: headers.map((h) => h.toLowerCase()).join(' '),
        signature: generateSignature(fingerprint, privateKey)
    };

    return Object.entries(signatureParts).map(([k, v]) => {           ←
        return `${k}=${v}`;
    }).join(',');
}
```

Specify a list of the different signature components.

Combine parts into comma-separated quoted key-value pairs.

## Authentifizierung

### Erzeugung signierter Request

```
const HEADERS = ['(request-target)', 'host', 'date', 'digest'];

function signRequest(
    request: HttpRequest, userId: string,
    privateKey: string, headers = HEADERS): HttpRequest {
    request.headers['digest'] =
        generateDigestHeader(request.body);
    const fingerprint =
        generateRequestFingerprint(request, headers);
    request.headers['signature'] = generateSignatureHeader(
        fingerprint, userId, privateKey, headers);
    return request;
}
```

**Update the digest header.**

**Generate the request fingerprint needed for the signature.**

**Update the signature header.**

**Return the augmented request.**

#### Listing 30.10 Example of a complete signed HTTP request

PATCH /chatRooms/1  
Host: example.org  
Digest: SHA-256=HV9PltG0QPRNs11FB7ebQA8XPasvPyRg6hhU0QF214M=  
Signature: keyId="1234",algorithm="rsa-sha256",headers="  
 (request-target) host date digest",  
 signature="mgBAQEsEsoBCgIOBiNfum37y..."  
Date: Tue, 27 Oct 2020 20:51:35 GMT  
  
{"title": "New title"}

## Authentifizieren

### Überprüfung Signatur



First we need a function to parse a signature header into separate components.

```
function parseSignatureHeader(signatureHeader: string): SignatureMetadata {
  const data = Object.fromEntries(
    signatureHeader
      .split(',')
      .map((pair) => pair.split('='))
      .map(([k, v]) => [k, v.substring(1, v.length-1)]));
  data.headers = data.headers.split(' ');
  return metadata;
}

async function verifyRequestSignature(request: HttpRequest): Promise<Boolean> {
  if (generateDigestHeader(request.body) !==
      request.headers['digest']) {
    return false;
  }
  const metadata = parseSignatureHeader(
    request.headers['signature']);
  const fingerprint = generateRequestFingerprint(
    request, metadata.headers);
  const publicKey = (await database.getUser(
    metadata.keyId)).publicKey;
  return verifySignature(
    fingerprint, metadata.signature, publicKey);
}
```

Parse the signature header into a bunch of pieces.

Split the k="v",k="v" pairs.

Split the k="v" pair.

Remove quotes.

Split just the headers.

Verify that the request body matches the digest header.

Determine the payload that was signed based on the headers.

Figure out the public key for the provided ID (stored in the database somewhere).

Verify the signature against the fingerprint with the public key.

**Bilden Sie drei Gruppen. Jede Gruppe erarbeitet eines dieser Themen:**

- 1.) Recherchieren Sie, wie API Authentifizierung mit HMAC realisiert wird**
- 2.) Recherchieren Sie, wie Passwort-basierte API Authentifizierung für APIs realisiert wird**
- 3.) Recherchieren Sie, wie OAuth2.0-basierte API Authentifizierung realisiert wird**

**Bereiten Sie eine Präsentation (5-10 min) vor, in der Sie diese Techniken kurz vorstellen.**

## *Besprechung Aufgaben API Authentifizierung*





## Authentifizierung

### *Generelle Überlegungen Passwortverwendung bei APIs*

- Was unterscheidet einen interaktiven Login von einer Passwortverwendung zum Zugang zu einer API? Was resultiert daraus für das Design der API?



## Authentifizierung

*Credential Stuffing, Password Spraying und Brute Force*

- **Definition:** Mit Credential Stuffing wird ein Angriff bezeichnet, bei dem ein Angreifer ihm bekannte Benutzername:Password-Paare durchprobiert. Die Benutzername:Password-Paare stammen oft aus Einbrüchen in anderen Systemen.
- **Definition:** Mit Password Spraying bezeichnet man einem Angriff, bei dem unerichtet häufige Passwörter für beliebige Benutzernamen ausprobiert werden.
- **Brute Force** bereits bekannt aus der Vorlesung „Grundlagen der IT-Sicherheit“
- **Alles vorwiegend Angriffe auf APIs, welche zur Authentifizierung Passwörter verwenden.**



- **Ratenbegrenzung: nur eine begrenzte Anzahl von Anfragen zulassen**
  - Nutzerbasierte Limits (Anzahl Zugriffe pro Zeitraum)
  - Wenn Limit erreicht
    - Hartes Limit: Blockieren
    - Weiches Limit: Anfragen drosseln
    - Abrechenbare Limits: Mehr Geld abbuchen
- **Frage: woran kann man festmachen, welche Requests zu welchem Benutzer gehören?**
- **Nur sichere Passwörter bei der Registrierung zulassen**
- **Statt E-Mail als Benutzername ein Pseudonym verwenden – bei den meisten Leaks werden E-Mail-Adressen als Benutzernamen verwendet**

## Angriffsfläche reduzieren (System)

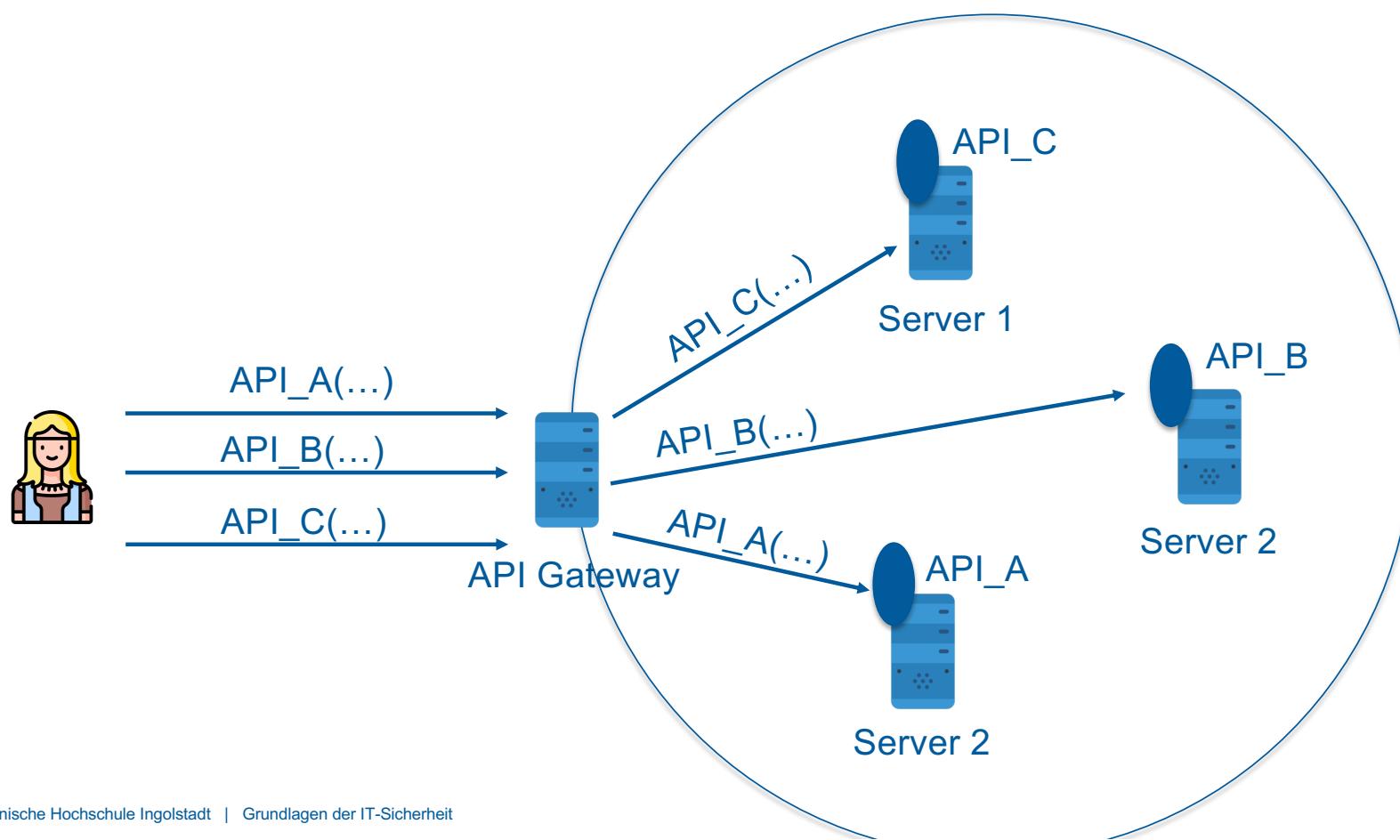
### API Gateway



- Werden von einer Organisation mehrere APIs angeboten, so kommen oft API Gateways zum Einsatz
- Ein API Gateway nimmt API Requests entgegen und leitet diese nach erfolgreicher Security-Überprüfung an das System weiter, welches die API bereitstellt
- Damit werden alle APIs von der Angriffsfläche genommen und durch die Proxys der APIs auf dem API Gateway ersetzt
- Damit können Security-Checks an einem zentralen Punkt vorgenommen und überwacht werden.

## Angriffsfläche reduzieren (System)

### API Gateway





## OWASP API Security Top 10 (2023)

### Unrestricted Resource Consumption

A social network implemented a “forgot password” flow using SMS verification, enabling the user to receive a one time token via SMS in order to reset their password.

Once a user clicks on “forgot password” an API call is sent from the user's browser to the back-end API:

#### ■ Beispielhafter Angriff:

```
POST /initiate_forgot_password

{
  "step": 1,
  "user_number": "6501113434"
}
```

Then, behind the scenes, an API call is sent from the back-end to a 3rd party API that takes care of the SMS delivering:

```
POST /sms/send_reset_pass_code

Host: willyo.net

{
  "phone_number": "6501113434"
}
```

The 3rd party provider, Willyo, charges \$0.05 per this type of call.

An attacker writes a script that sends the first API call tens of thousands of times. The back-end follows and requests Willyo to send tens of thousands of text messages, leading the company to lose thousands of dollars in a matter of minutes.



## OWASP API Security Top 10 (2023)

### *Unrestricted Resource Consumption*

**INPUT**

---

Matt Wille

June 29, 2021

CULTURE

**Hackers used LinkedIn's official API to leak tons of data... again**

User records leaked on the dark web.





## *Unrestricted Resource Consumption*

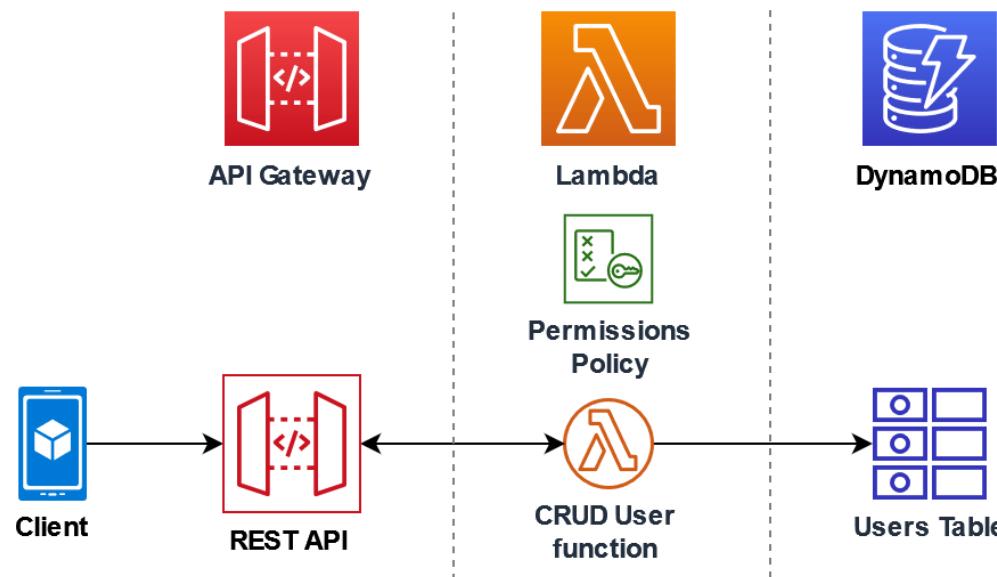
### *Gegenmaßnahmen*

- Schwachstelle entsteht dadurch, dass eine API die Anzahl der Interaktionen und den durch einen Zugriff entstehenden Ressourcenverbrauch nicht kontrolliert und einschränkt
- Gegenmaßnahmen
  - Limit für Interaktionen:
    - Ratenbegrenzung API Aufrufe (siehe vorne)
    - Limits für Ausführung (Ausführungszeit, Speicher, File Descriptors, Prozesse, maximale Größe Upload, maximale Anzahl Aufrufe in einem einzigen API Request (z.B. bei GraphQL batching), Aufrufe externe APIs, maximale Menge rückgegebene Daten)

## Unrestricted Resource Consumption

Gegenmaßnahmen

- Realisierung Kontrolle Ressourcen\_Verbrauch z.B. mit Containern/Serverless Code
  - AWS Lambda ist eine beispielhafte Lösung:





- Durch eine API werden Geschäftsprozesse öffentlich zugänglich und können automatisiert ausgelöst werden. Hierdurch können Angreifer Schaden anrichten.
- Beispielszenario:

An airline company offers online ticket purchasing with no cancellation fee. A user with malicious intentions books 90% of the seats of a desired flight.

A few days before the flight the malicious user canceled all the tickets at once, which forced the airline to discount the ticket prices in order to fill the flight.

At this point, the user buys herself a single ticket that is much cheaper than the original one.



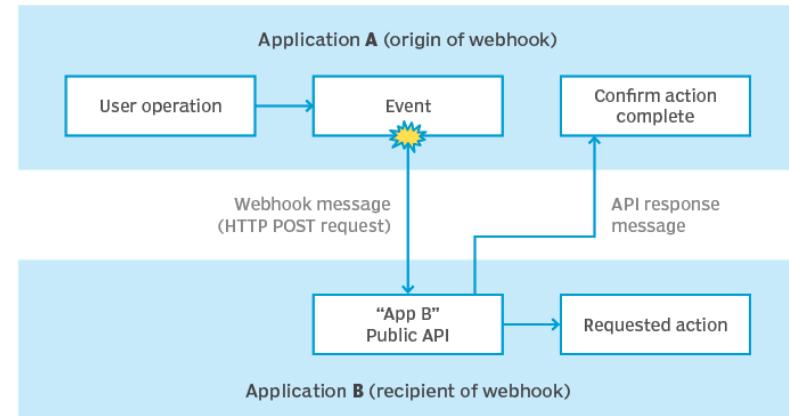
- **Überarbeitung der Geschäftsprozesse und Realisierung von Gegenmaßnahmen**
  - Fragestellung ist: Welche Geschäftsprozesse gefährden uns, wenn Sie von einem Angreifer automatisiert oft genutzt werden?
  - Frage: welche Gegenmaßnahmen im Beispiel vorne?
- **Technische Gegenmaßnahmen:**
  - Authentifizierung und Einschränkung des Service pro Person (erfordert Registrierung)
  - Device Fingerprinting (z.B. User Agent String, Screen Size, OS, ...)
    - Für ungewöhnliche Geräte den Service ablehnen (z.B. Headless Browser)
  - Erkennung von Menschen im Geschäftsprozess (z.B. Captcha)
  - Erkennung von nicht-menschlichem Verhalten (z.B. „add to cart“ und „complete purchase“ in weniger einer Sekunde ausgeführt)
  - Tor-Exit Nodes und bekannte Proxis blocken



- SSRF tritt auf, wenn eine API eine URL übergeben bekommt und diese ungeprüft aufruft
  - z.B. Bestandteil von Webhooks, File Fetching from URLs, custom SSO, URL Preview
- Definition: Ein Webhook (auch Reverse-API genannt) ist ein Dienst, der bei Eintritt eines Ereignisses eine URL aufruft.

## Example of a simple webhook design

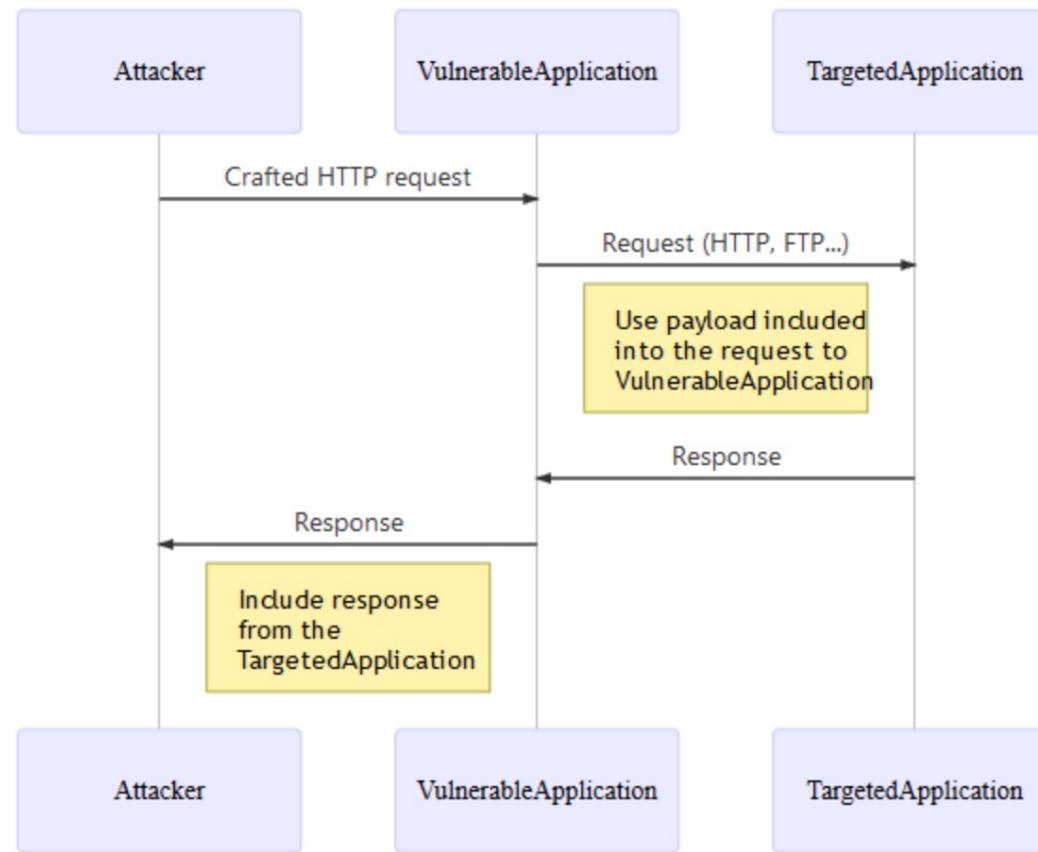
- 1 Application A registers a user operation as an event.
- 2 The event triggers and sends a HTTP POST request to Application A.
- 3 The Public API receives the POST request.
- 4 The Public API completes request and messages Application A to indicate the task is complete.



Bildquelle: <https://www.computerweekly.com/de/definition/Webhook>

## Server Side Request Forgery (SSRF)

### Allgemeiner Ablauf





## Server Side Request Forgery (SSRF)

### Angriffsbeispiel

A social network allows users to upload profile pictures. The user can choose either to upload the image file from their machine, or provide the URL of the image. Choosing the second, will trigger the following API call:

```
POST /api/profile/upload_picture  
  
{  
    "picture_url": "http://example.com/profile_pic.jpg"  
}
```

An attacker can send a malicious URL and initiate port scanning within the internal network using the API Endpoint.

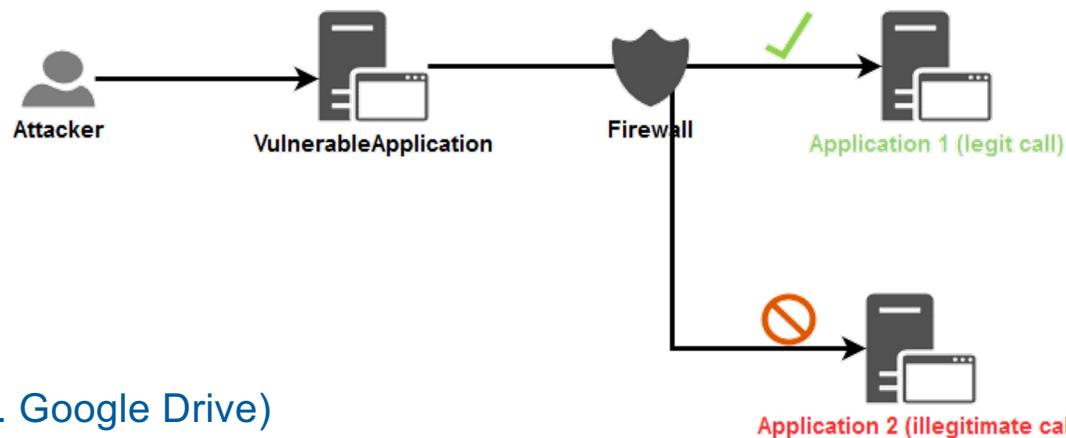
```
{  
    "picture_url": "localhost:8080"  
}
```

Based on the response time, the attacker can figure out whether the port is open or not.

## Server Side Request Forgery (SSRF)

### Gegenmaßnahmen

- Sicherstellen, dass der Mechanismus, der die URL aufruft, keine Aufrufe ins lokale Netzwerk zulässt.



- Whitelisting von URLs
  - Erwartete URLs (z.B. Google Drive)
  - URL Schema und Ports (z.B. nur https:// und http:// zulassen, nicht file://, ftp:// ...)
  - Erlaubte Medientypen (z.B. Bilder)
- HTTP Redirection deaktivieren



## Server Side Request Forgery (SSRF)

### Beispielcode Überprüfung Domain Whitelist

```
# Dependencies: pip install ipaddress dnspython
import ipaddress
import dns.resolver

# Configure the allow list to check
DOMAINS_ALLOWLIST = ["owasp.org", "labslinux"]

# Configure the DNS resolver to use for all DNS queries
DNS_RESOLVER = dns.resolver.Resolver()
DNS_RESOLVER.nameservers = ["1.1.1.1"]

def verify_dns_records(domain, records, type):
    """
    Verify if one of the DNS records resolve to a non public IP address.
    Return a boolean indicating if any error has been detected.
    """
    error_detected = False
    if records is not None:
        for record in records:
            value = record.to_text().strip()
            try:
                ip = ipaddress.ip_address(value)
                # See https://docs.python.org/3/library/ipaddress.html#ipaddress.IpAddress
                if not ip.is_global:
                    print("[!] DNS record type '%s' for domain name '%s' resolve to
a non public IP address '%s'!" % (type, domain, value))
                    error_detected = True
            except ValueError:
                error_detected = True
                print("[!] '%s' is not valid IP address!" % value)
    return error_detected
```

```
def check():
    """
    Perform the check of the allow list of domains.
    Return a boolean indicating if any error has been detected.
    """
    error_detected = False
    for domain in DOMAINS_ALLOWLIST:
        # Get the IPs of the current domain
        # See https://en.wikipedia.org/wiki/List_of_DNS_record_types
        try:
            # A = IPv4 address record
            ip_v4_records = DNS_RESOLVER.query(domain, "A")
        except Exception as e:
            ip_v4_records = None
            print("[i] Cannot get A record for domain '%s': %s\n" % (domain,e))
        try:
            # AAAA = IPv6 address record
            ip_v6_records = DNS_RESOLVER.query(domain, "AAAA")
        except Exception as e:
            ip_v6_records = None
            print("[i] Cannot get AAAA record for domain '%s': %s\n" % (domain,e))
        # Verify the IPs obtained
        if verify_dns_records(domain, ip_v4_records, "A") or verify_dns_records(domain, ip_v6_records, "AAAA"):
            error_detected = True
    return error_detected

if __name__ == "__main__":
    if check():
        exit(1)
    else:
        exit(0)
```



### ■ Beispieldaten:

A social network website offers a "Direct Message" feature that allows users to keep private conversations. To retrieve new messages for a specific conversation, the website issues the following API request (user interaction is not required):

```
GET /dm/user_updates.json?conversation_id=1234567&cursor=GRIFp7LCUAAAA
```

Because the API response does not include the `Cache-Control` HTTP response header, private conversations end-up cached by the web browser, allowing malicious actors to retrieve them from the browser cache files in the filesystem.

### ■ Gegenmaßnahmen:

- Härtung aller beteiligten Systeme (Out-of-Scope für diese Vorlesung)



### ■ Beispielszenario:

A social network implemented a rate-limiting mechanism that blocks attackers from using brute force to guess reset password tokens. This mechanism wasn't implemented as part of the API code itself but in a separate component between the client and the official API (`api.socialnetwork.owasp.org`). A researcher found a beta API host (`beta.api.socialnetwork.owasp.org`) that runs the same API, including the reset password mechanism, but the rate-limiting mechanism was not in place. The researcher was able to reset the password of any user by using simple brute force to guess the 6 digit token.

### ■ Gegenmaßnahmen (alle Out-of-Scope für diese Vorlesung):

- Inventar erstellen mit allen API Hosts und Dokumentation der Hosts mit Fokus auf API Environment, Zugriffssteuerung und API Versino
- Inventar aller Dienste und Dokumentation ihrer Rolle im System, Datenflows, ...
- Dokumentationen automatisch erstellen lassen im Rahmen einer CI/CD Pipeline
- Keine Verwendung von Production Data in Non-Production API Deployments



- Bei der Verwendung von APIs vertrauen Entwickler dem Output der APIs mehr als User Input
- Um sicherzustellen, dass keine Angriffe möglich:
  - Nur verschlüsselter Zugriff auf API
  - Input-Daten Validierung vor Weitergabe an andere Komponenten
- **Beispiel Angriff:** An API relies on a third-party service to enrich user provided business addresses. When an address is supplied to the API by the end user, it is sent to the third-party service and the returned data is then stored on a local SQL-enabled database.  
Bad actors use the third-party service to store an SQLi payload associated with a business created by them. Then they go after the vulnerable API providing specific input that makes it pull their "malicious business" from the third-party service. The SQLi payload ends up being executed by the database, exfiltrating data to an attacker's controlled server.

## Minimierung der Angrifsoberfläche

### Übersicht

#### ■ Angrifsoberfläche aus Sicht einer Software-Komponente (nach Domäne):

-  APIs und Interfaces zu anderen Software-Komponenten
  - User Interfaces (einzelne Komponenten)
  - Konfigurationsdaten
  - Datenbankzugriff



- **Kapselung des User Interfaces in einer einzigen Komponente**
  - Zum Beispiel mit dem Entwurfsmuster Model-View-Controller (siehe Vorlesung Web-Techniken)
  - Reduziert den Code, der mit Benutzereingaben arbeitet
- **Benutzeroberfläche für nicht-authentifizierte Benutzer reduzieren**
  - Beispielsweise vor der Authentifizierung nur Zugang zum Login
  - Angreifer benötigen dann für alle weiteren Teile der Benutzeroberfläche einen Zugang zum System
- **Benutzerinput als unvertrauenswürdige Daten behandeln**
  - Input Sanitization notwendig

### ■ SQL Injection Schwachstelle aus Kapitel 2

```
# Benutzername in der Variable name
sql_query = "Select * from users where user_name=\\"+name+"\\\";"
```

### ■ Verwendung eines Prepared Statements

```
# Benutzername in der Variable name
# MySQL Datenbank-Connection in db_cnx

cursor = db_cnx.cursor(prepared=True)
sql_query = """Select * from users where user_name=%s;"""      # %s Placeholder
cursor.execute(sql_query, (name,))                                # (name,) = Param.tupel
```



### **Minimierung der Angriffsflächen**

- **Security by Default**
- **Minimale Rechte**
- **Sichere Fehlerbehandlung**
- **Vertrauensdesign**
- **Trennung von Verantwortlichkeiten**
- **Keine Security by Obscurity**
- **Komplexitätsreduktion**
- **Verwendung von kryptographischen Methoden in Software**
- **Software-Updates**
- **Sichere Datenspeicherung auf unsicheren Medien**
-