

thecodecampus</>  
w1k

# Sicherheit in SPAs

mit

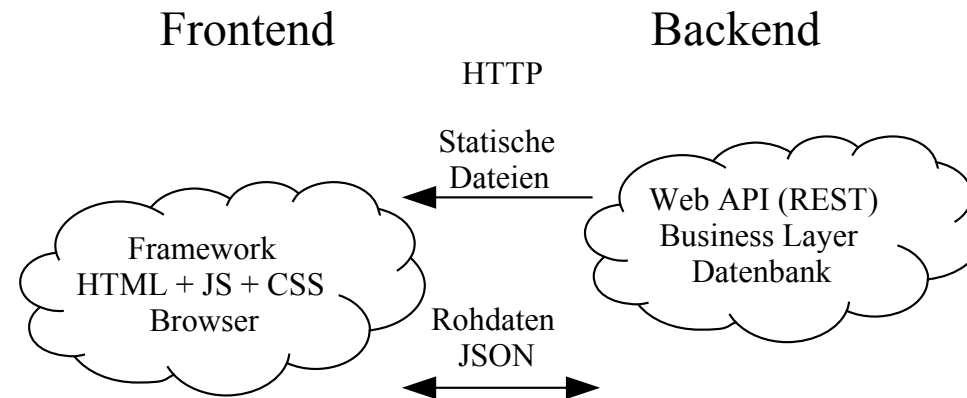


- Ausgangssituation
- Sicherheitskonzept
- Gängige Probleme
  - Ursache
  - Auswirkung
  - Test
  - Gegenmaßnahme

# ÜBER MICH

- Philipp Burgmer
  - Software-Entwickler, Trainer
  - Fokus: Frontend, Web-Technologien
  - [burgmer@w11k.de](mailto:burgmer@w11k.de)
- w11k GmbH
  - Software Design, Entwicklung & Wartung
  - Consulting, Schulungen & Projekt Kickoff
  - Web-Apps, Mobil-Apps, Rich Clients
  - AngularJS, TypeScript, Eclipse RCP

# ARCHITEKTUR VON SPAs



- Rich Client im Browser
- Server liefert statische Dateien für den Client
- Server bietet API für Daten (REST, WebSocket) (JSON, XML)
- Backend weis nichts über verwendete Technologien im Client
- Client weis nichts über verwendete Technologien im Backend
- Stateful Client, Stateless Backend

# TECHNOLOGIES

- Datenbanken (SQL | NoSQL) & Backend-Sprache
- HTTP
- JavaScript & HTML
- Historisch betrachten
- Vieles gewachsen
- Nicht für heute Verwendung gedacht

# SICHERHEITSKONZEPT

## NAIV

- Öffentlicher und privater Bereich
- Login -> Session
- Benutzer-Rollen
- Grundgedanke: Jeder sichert sich selbst ab
  - Client schützt UI
  - Server schützt Datenzugriffe
  - Jeder schützt seine verwendeten Technologien
  - Alle schützen die Übertragung

# LOGIN

- Login vor Aufruf der Anwendung
- Login innerhalb der Anwendung
- Berechtigungen innerhalb der Anwendung

# LOGIN

## VOR DER ANWENDUNG

- Server stellt sicher
  - Anwendung nur mit gültigem Login aufrufbar
  - Ohne gültigen Login: HTTP-Redirect auf Login-Seite
  - Nach erfolgreichem Login: HTTP-Redirect auf Anwendung
- In Anwendung
  - HTTP 401: Navigation zu Login-Seite
- Weniger Angriffsfläche: Nicht jeder sieht die Anwendung
- Schnelles Laden der ersten Seite
- Immer ganze Anwendung geschützt



# LOGIN

## IN DER ANWENDUNG

- Rein Client-seitiges Handling (für UI)
- Login-Formular als Route / State in Anwendung
- Ajax-Request für Login
- Prüfung auf gültigen Login
  - State-Change + Event-Handler `$stateChangeError`
  - API-Requests + HTTP Interceptor
- Weniger Request notwendig
- Öffentliche und geschützte Bereiche möglich

# BERECHTIGUNGEN VERWALTEN

## IN ANGULARJS

- Berechtigungen über Rollen verwalten
- Bereiche mit Rollen versehen
- Im UI per Direktive

```
1 <ul class="menu">
2   <li user-role-required="'ADMIN'">
3     <a href="#!/admin">Admin</a>
4   </li>
5 </ul>
```

# BERECHTIGUNGEN VERWALTEN

## IN ANGULARJS

- An Route / State per `resolve`

```
1 angular.module('app').config(function() {
2   $stateProvider.state('admin', {
3     url: '/admin',
4     templateUrl: 'route/admin/admin.html',
5     resolve: {
6       authorized: /* @ngInject */ function (UserService) {
7         return UserService.hasRoles('ADMIN');
8       }
9     }
10  });
11 }
```

# TOP 10 SICHERHEITSPROBLEME

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

Quelle: OWASP Top10 2013

# OWASP

- The Open Web Application Security Project
- Non-Profit Organisation
- Finanziert über Mitgliedsbeiträge und Spenden
- Existiert seit 2001
- Stellt Informationen zu Sicherheitsthemen bereit
  - detaillierte Beschreibungen und Erklärungen
  - gängige Lösungsansätze

# GENERELLE GEGENMASSNAHMEN

- Benutzereingaben nie trauen
- Im Backend nie davon ausgehen, dass Request vom Client kommen
- Verwendete Komponenten auf Security-Updates prüfen
- Security testen
  - [punkspider.org](http://punkspider.org): Suchmaschine für Sicherheitslücken
  - [BeEF - The Browser Exploitation Framework](#): Tool für Penetrationstests
  - [OWASP - Vulnerability Scanning Tools](#)

# UNZUREICHENDE GEGENMASSNAHMEN

- Code-Minimierung / -Obfuscating
- Verwendung von HTTPS
- Berechtigungen im Client prüfen
- Eingaben im Client validieren

# CODE INJECTION



# BEISPIEL: SQL

*Java Code um SQL Abfrage zusammen zu bauen*

```
1 statement = "SELECT * FROM users WHERE id = " + request.getParameter("id") + ";"
```

*URL-Aufruf des Angreifers*

```
1 http://example.com/user?id=42;UPDATE+USER+SET+TYPE="admin"+WHERE+ID=23;--
```

*Ausgeführtes SQL*

```
1 SELECT * FROM users WHERE id = 42; UPDATE USER SET TYPE="admin" WHERE ID=23;--;
```

# CODE INJECTION

- Daten aus Sprache A werden zu Code in Sprache B
- Code wird dynamisch an einen Interpreter übergeben
- Code enthält Benutzereingaben (Formular-Daten, URL-Parameter, ...)
- Benutzereingaben werden nicht oder unzureichend überprüft
- An vielen Stellen möglich
  - SQL
  - HTML (z.B. bei Cross-Site-Scripting)
  - Script-Sprachen mit eval-Funktion (JS, PHP)
  - Dynamisches Laden von Code aus Dateien
  - Shell / Command Execution

# SCHWACHSTELLEN FINDEN

- Manuell am Code
  - Verwendung von Interpretern ausfindig machen
  - Eingaben von Interpretern auf dynamische Teile untersuchen
  - Datenfluss zurückverfolgen (Wo kommen dynamische Teile her?)
- Automatisiert
  - Code Analyse Tools um Interpreter zu finden
  - Penetration-Test-Tools finden häufig gemachte Fehler

# GEGENMASSNAHMEN

- Möglichst wenig Interpreter verwenden, besser APIs
  - Prepared-Statements
  - Stored-Procedures
- Benutzereingaben nicht vertrauen
  - Kontextuelles Escapen (HTML, JS, SQL)
  - White-Listing

# BEISPIEL: SQL

*Sicherer Java Code um SQL Abfrage zusammen zu bauen*

```
1 PreparedStatement pstmt = connection.prepareStatement("SELECT * FROM users WHERE id = ?");  
2 pstmt.setInt(1, request.getParameter("id"));  
3 ResultSet rset = pstmt.executeQuery();
```

# BROKEN AUTHENTICATION AND SESSION MANAGEMENT

# SESSION MANAGEMENT

- Zugangsdaten oder Session können entwendet werden
- Session kann geklaut werden
  - z.B. Session-ID in der URL, oft bei URL Rewriting
- Kein Session-Timeout (öffentlicher PC)
- Vorhersagbare Session IDs
- Übertragung per unverschlüsselter Kommunikation
- Cross-Site-Scripting um Cookie zu entwenden

# BEISPIELE

- Passwörter stehen im Klartext in der Datenbank
  - Datenbank wird entwendet
  - Angreifer kann sich als jeder User einloggen
- Session-ID steht in URL

```
1 http://example.com/shoppingcart?sessionid=268544541
```



# GEGENMASSNAHMEN

- Login, Logout und Session Management nicht selbst implementieren
- Bewährte, gut getestete Bibliotheken verwenden (OAuth?)
- Verschlüsselte Kommunikation
- Keine Passwörter speichern, Hash mit Salt
- Cross-Site-Scripting verhindern

## HERAUSFORDERUNG

# STATELESS BACKEND

- Weniger Zustand im Server -> Bessere Skalierbarkeit
- Gut: Session = Mapping Session ID -> User ID
- Besser: keine Session im Backend, Session ID enthält allen Zustand
- Im Backend benötigter Zustand wird bei jedem Request übertragen

# STATEFUL SESSION-ID

- Session-ID ist kein Random oder Hash
- Session-ID enthält Zustand
  - User-ID
  - Login-Timestamp
  - XSRF-Token?
  - Base64 encoded
- Session-ID wird gegen Manipulation und Nachahmung geschützt
  - Verschlüsselung
  - Signierung
  - Message Authentication Code (z.B. HMAC)
  - Nur auf dem Server bekannt

XSS

CROSS-SITE-SCRIPTING

# BEISPIEL

```
1 var source = $('#insecure-input');  
2 var text = source.val();  
3 var target = $('#insecure-output');  
4 target.append(text);
```

Ausprobieren ...

# CROSS-SITE-SCRIPTING

- Spezielle Art der HTML Injection
- HTML-Injection wird ausgenutzt um anderen Benutzer Code unterzuschieben
- Benutzereingabe wird ohne Prüfung in HTML ausgegeben
- Ermöglicht Ausführen von Code
- Angriffe
  - Daten auslesen und an Angreifer übermitteln (z.B. Session-Cookie)
  - Code ruft URL auf um Aktion mit Rechten des Benutzers auszuführen (ähnlich wie XSRF)

# GEGENMASSNAHMEN

- Benutzereingaben immer escapen
- Daten vom Server escapen
- Sanitizer Bibliothek verwenden
- Kontext beachten in dem Wert verwendet wird

# ANGULARJS

- Angular escappt alle Data-Bindings automatisch
- \$sanitize Service um sicheres HTML-Subset ausgeben zu können
- \$sce Service um beliebiges HTML aus vertrauenswürdiger Quelle ausgeben zu können
- Ausführliches Beispiel



# ANGULARJS

## BESPIEL

```
1 <input type="text" ng-model="text"/>
2 <div ng-bind="text"></div>
3 <div ng-bind-html="text"></div>
```

# ANGULARJS

## NG-BIND-HTML

- `ng-bind` und `{{}}` escaped alle HTML Sonderzeichen
- `ng-bind-html` lässt ein sicheres Subset durch
- `ngSanitize`: zusätzliches Modul mit erweitertem Sanitizer für sicheres Subset
- Muss eingebunden werden für `ng-bind-html`, ansonsten Fehler auf Konsole

# ANGULARJS

## STRICT CONTEXTUAL ESCAPING

- `$sce` Service stellt Methoden zum wrappen bereit
- JS, URL, HTML
- `$sce.trustAsHtml` wrapt Text in Objekt
- Objekt markiert Text als sicheren Code
- `ng-bind-html` übernimmt ursprünglichen Text als Code in DOM

XSRF

CROSS-SITE-REQUEST-FORGERY

# BEISPIEL

- Ausgangssituation: Benutzer in App eingeloggt (hat gültiges Session-Cookie)

*Aufruf von Business Logik ohne zusätzlichen Schutz*

```
1 http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243
```

*XSRF Attacke per Social Engineering*

```
1 <a href="http://bit.ly/xyz">Link zu einer "vertrauenswürdigen" Seite</a>
```

*XSRF Attacke per XSS*

```
1 
```

# XSRF

- Angreifer bringt Benutzer dazu URL aufzurufen
- Request wird mit Rechten des Benutzers ausgeführt
- Verschiedene Angriffsformen
  - Cross-Site-Scripting
  - Social-Engeneering / Unterschieben einer URL
- Cookies allein sind nicht sicher
  - Für Session-Cookie immer `httpOnly` und `secure` verwenden
  - Cookie kann nicht abgegriffen werden (per JS)
  - Cookie wird aber immer gesendet (XSRF immer noch möglich)
- Zusätzlicher Schutz notwendig

# GEGENMASSNAHMEN

- Server
  - Schickt bei Login Session-ID als Cookie mit `httpOnly` und `secure`
  - Schickt bei Login zusätzliches Token als Cookie `XSRF-Token` ohne `httpOnly`
- Client
  - Token wird zwischengespeichert (JS Variable) und Cookie gelöscht
  - Token wird bei jedem Request als Header mitgesendet
- Server validiert bei jedem Request mitgesendetes Token

# ANGULARJS

- HTTP-Interceptor Konzept
- Interceptor schon mit dabei
  - Liest Cookie `XSRF-TOKEN`
  - Sendet Header `X-XSRF-TOKEN`
  - Namen konfigurierbar
- Problem: *Öffne Link in neuem Tab*
- Lösung: Server sendet Token noch mal bei *GET api/login*



Philipp Burgmer  
burgmer@w11k.de

[www.w11k.de](http://www.w11k.de)  
[www.thecodecampus.de](http://www.thecodecampus.de)