thecodecampus</>

w11k

Sicherheit in SPAs

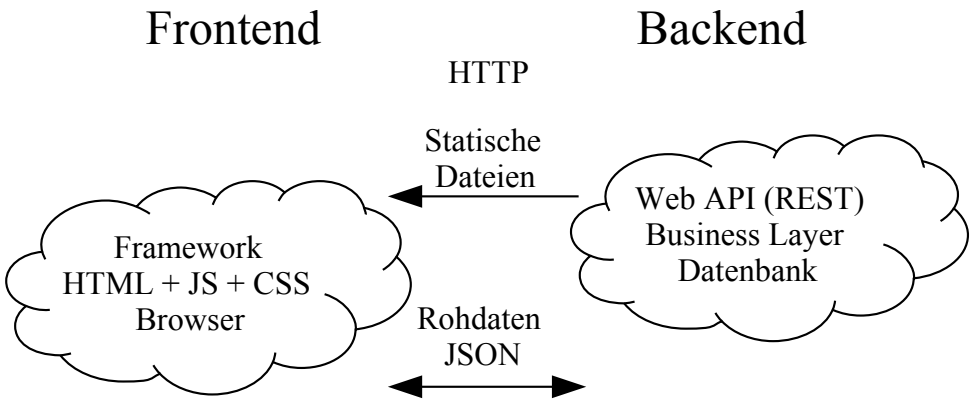
mit

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

ÜBER MICH

- Philipp Burgmer
  - Software-Entwickler, Trainer
  - Fokus: Frontend, Web-Technologien
  - burgmer@w11k.de
- 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

TECHNOLOGIEN

- Datenbanken (SQL | NoSQL) & Backend-Sprache
- HTTP
- HTML, JavaScript & CSS

- 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 schütz UI
  - Server schütz Datenzugriffe
  - Jeder schützt seine verwendeten Technologien
  - Alle schützen die Übertragung

GENERELLE GEGENMASSNAHMEN

- Benutzereingaben nie trauen
- Im Backend nie davon ausgehen, dass Request vom Client kommen
- Security testen
  - Grundlegend: Von Entwicklern selbst
  - Tiefgründig: Von Spezialisten

GENERELLE GEGENMASSNAHMEN

HTTP HEADER

- List of Useful HTTP Headers
  - `Strict-Transport-Security: max-age=86400; includeSubDomains`
  - `X-Frame-Options: deny`
  - `Content-Security-Policy: default-src 'self'`
  - `X-XSS-Protection: 1; mode=block`
  - `X-Content-Type-Options: nosniff`

UNZUREICHENDE GEGENMASSNAHMEN

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

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
- Stellt Informationen zu Sicherheitsthemen bereit
  - detaillierte Beschreibungen und Erklärungen
  - gängige Lösungsansätze
- Non-Profit Organisation
- Finanziert über Mitgliedsbeiträge und Spenden
- Existiert seit 2001

A1

INJECTION

BEISPIEL

SQL

Java Code um SQL Abfrage zusammen zu bauen

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

URL-Aufruf des Angreifers

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

Ausgeführtes SQL

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

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 (Content-Spoofing und 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
  - Peneration-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();
```

ANGULARJS

- Angular escapt 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 eigebunden 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

ANGULARJS

`$interpolate`

> *For security purposes, it is strongly encouraged that web servers escape user-supplied data, replacing angle brackets (<, >) with < and > respectively, and replacing all interpolation start/end markers with their escaped counterparts.*
>
> **$interpolate API Documentation at angularjs.org**

```
<div>
  <input type="text" ng-model="text_1"/><span ng-bind="text_1"></span>
</div>
<div>
  <input type="text" ng-model="text_2"/><span ng-bind="text_2"></span>
</div>
```

```
<div ng-init="text_3={{ text_2 = 1}}">
  <input type="text" ng-model="text_3"/>
  <span ng-bind="text_3"></span>
</div>
```

ANGULARJS

`$interpolate`

```
<div ng-init="text_1 = {{ text_2 = 1}}">
  <input type="text" ng-model="text_1"/><span ng-bind="text_1"></span>
</div>
<div>
  <input type="text" ng-model="text_2"/><span ng-bind="text_2"></span>
</div>
```

1 1

- Daten vom Server, die durch `$interpolate` laufen, müssen escaped werden
- Soll der Server wirklich etwas über die Verwendung im Frontend wissen?
- Kann auch im Client per HTTP-Interceptor gemacht werden

A2

BROKEN AUTHENTICATION AND SESSION MANAGEMENT

SESSION MANAGEMENT

- Zugangsdaten oder Session können entwendet werden
- Session kann geklaut werden
  - Session-ID in der URL (URL Rewriting, Cookies deaktiviert)
  - Cross-Site-Scripting
- Kein Session-Timeout (öffentlicher PC)
- Vorhersagbare Session IDs
- Übertragung per unverschlüsselter Kommunikation

BEISPIELE

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

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

- Session-IDs kommen aus Pool

GEGENMASSNAHMEN

- Login, Logout und Session Managemnt nicht selbst implementieren
- Bewährte, gut getestete Biblotheken verwenden (z.B. OAuth)
- Verschlüsselte Kommunikation
- Keine Passwörter im Klartext speichern, sondern Hash mit Salt
- Cross-Site-Scripting verhindern

LOGIN

- Login vor Aufruf der Anwendung
- Login 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
  - Prüfen auf 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

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
- Session-ID wird gegen Manipulation und Nachahmung geschützt
  - Verschlüsselung oder Message Authentication Code (z.B. HMAC)
  - Nur auf dem Server bekannt
- Anfällig gegen Replay-Attacken
  - Server kennt keinen echten Logout
  - Session-ID entwendet -> weiter Requests möglich
  - Schwierig zu beheben (Request-Tokens)

A3

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 Angreifen 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 Biblothek verwenden
- Kontext beachten in dem Wert verwendet wird
- Content-Security-Policy anwenden

CONTENT SECURITY POLICY

- What is CSP and Why Haven't You Applied it Yet
- Per Header einschalten: `Content-Security-Policy: default-src 'self'`
- Verhindert Ausführen von *Text als Code* (JavaScript `eval`)
- Kein dynamisches Erzeugen von Script- und Style-Tags

CONTENT SECURITY POLICY

ANGULARJS

- Attribut bei ngApp

```
1 <html ng-app="app" ng-csp></html>
```

- AngularJS verwendet kann kein `new Function` mehr
- Langsamer, aber sicherer
- AngularJS kann kein CSS mehr in Seite injizieren
  -> `angular-csp.css` einbinden

A7

MISSING FUNCTION LEVEL ACCESS CONTROL

BEISPIEL

- Benutzer ist kein Admin
- Admin-Bereich wird im Menü nicht angezeigt
- Admin-Bereich ist aber über URL erreichbar

```
http://example.com/app/admin
```

- Angreifer kann API direkt aufrufen

```
http://example.com/api/admin/users
```

MISSING FUNCTION LEVEL ACCESS CONTROL

- Wer kann wann wo was aufrufen?
- Wird dabei seine Berechtigung geprüft?

GEGENMASSNAHMEN

- Berechtigungen über Rollen/Gruppen verwalten
- Berechtigungen an allen relevanten Stellen prüfen
- Im Client
  - Im UI nicht anzeigen was der User eh nicht Ausführen darf
  - Manuelle Ausführung verhindern
- Im Server
  - Requests vom Client nicht vertrauen
  - An jedem REST Endpoint
  - Eventuell pro Verb (jeder darf Lesen, nur Admin darf Schreiben)

BERECHTIGUNGEN

ANGULARJS

- Bereiche im Frontend mit Rollen versehen
- Im UI per Direktive

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

- Bereich zusätzlich vor manuellem Aufruf schützen

BERECHTIGUNGEN

ANGULARJS

- An Route / State per `resolve`

```
angular.module('app').config(function() {
  $stateProvider.state('admin', {
    url: '/admin',
    templateUrl: 'route/admin/admin.html',
    resolve: {
      authorized: /* @ngInject */ function (UserService) {
        return UserService.hasRoles('ADMIN');
      }
    }});});
```

- Event-Handler für `$stateChangeError`

A8

XSRF

CROSS-SITE-REQUEST-FORGERY

BEISPIEL

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

Aufruf von Business Logik ohne zusätzlichen Schutz

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

XSRF Attacke per Social Engeneering

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

XSRF Attacke per XSS

```
<img src="http://example.com/app/transferFunds?amount=1500&destination=attacker" />
```

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
  - Ließt 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
www.thecodecampus.de