KARLSRUHER ENTWICKLERTAG

thecodecampus</> wtk

Sicherheit in SPAs

mit

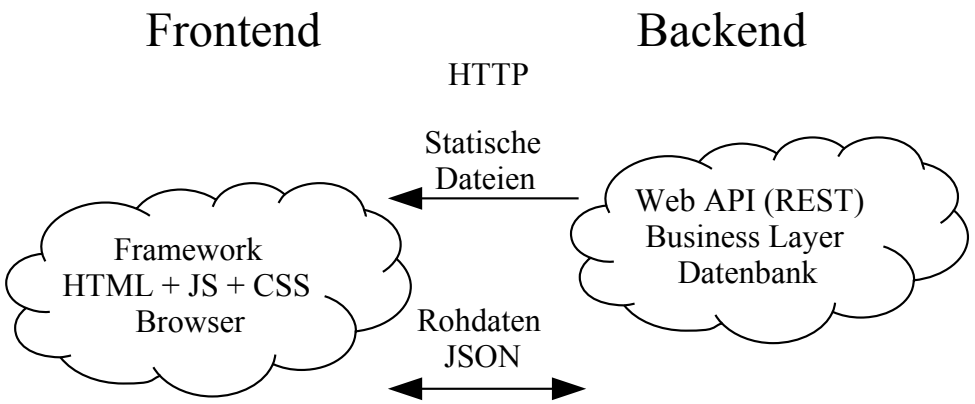
- Ausgangssituation
- Sicherheitskonzept
- Gängige Probleme
  - Ursache
  - Auswirkung
  - Test
  - Gegenmaßnahme

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 Anwendungen mit AngularJS
- Native Rich Clients mit 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 schütz UI
  - Server schütz Datenzugriffe
  - Jeder schützt seine verwendeten Technologien
  - Alle schützen die Übertragung

LOGIN

- Vorgelagert als extra Seite
  - Anwendung nur mit gültigem Login aufrufbar
  - Nicht eingeloggt: HTTP-Redirect auf Login
  - Eingeloggt: HTTP-Redirect auf Anwendung
  - Weniger Angriffsfläche: Nicht jeder sieht die Anwendung
  - Schnelles Laden der ersten Seite
- Login als Route in Anwendung
  - Einfacheres Handling
  - Kein Zusätzlicher Request für Benutzer-Daten notwendig

BERECHTIGUNGEN VERWALTEN

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

```
<li match-route="/admin.*" user-role-required="'ADMIN'">
  <a href="#!/admin/overview">Admin</a>
</li>
```

- An Route / State per `resolve`

```
module.config(function($stateProvider, ResolveFunctions) {
  $stateProvider.state('admin', {
    url: '/admin', templateUrl: 'route/admin/admin.html', controller: 'AdminCtrl',
    resolve: { authorized: ResolveFunctions.userRolesRequired('ADMIN') }
  });
});
```

BERECHTIGUNGEN VERWALTEN

```
1 angular.module('app').constant('ResolveFunctions', {
2   userRolesRequired: function (roles) {
3     return /* @ngInject */ function (UserService) {
4       return UserService.hasRoles(roles);
5     };
6   }
7 });
```

BERECHTIGUNGEN VERWALTEN

```
angular.module('app').service('UserService', function ($http, $q) {
  return {
    getUser: function () {
      // load user information from server and return promise
    },
    hasRoles: function (roles) {
      // get user then check if user has roles
      // return promise
    }
  };
})
```

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: 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

```
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;--;
```

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
  - 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();
```

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

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

GEGENMASSNAHMEN

- Login, Logout und Session Managemnt nicht selbst implementieren
- Bewährte, gut getestete Biblotheken verwenden (OAuth?)
- Verschlüsselte Kommunikation
- Keine Passwörte 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

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
- Verschiedene Type: Persistent / Non-Persistent, DOM-Based
- Benutzereingabe wird in HTML ausgegeben
- Ermöglicht Ausführen von Code

- Code-Ausführung übermittelt Daten an Angreifen (z.B. Cookies)
- Code ruft URL auf um Aktion mit Rechten des Benutzers auszuführen

GEGENMASSNAHMEN

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

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
- Detaillierte Erklärung

ANGULARJS BESPIEL

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

CROSS-SITE REQUEST FORGERY

BEISPIEL

Aufruf von Business Logik ohne zusätzlichen Schutz

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

XSRF Attacke per XSS

```
<img src="http://example.com/app/transferFunds?amount=1500&destinationAccount=attackersAcct#" width="0"
height="0" />
```

CROSS-SITE REQUEST FORGERY

- Angreifer ruft URL mit den Rechten des Benutzers auf
- Verschiedene Angriffsformen
  - Cross-Site-Scripting
  - Social-Engeneering / Unterschieben einer URL

- Cookies sind nicht sicher!
- Auch nicht mit `httpOnly` und `secure`
- Cookies können abgegriffen werden bzw. werden automatisch gesendet

GEGENMASSNAHMEN

- Login schickt Session-ID als Cookie mit `httpOnly` und `secure`
- Login antwortet mit zusätzliches Token im Header (`X-XSRF-Token`)
- Token kann nur vom eigenen JS Code aus abgegriffen werden, der den Request gestartet hat
- Token muss danach vom eigenen Code explizit als Header mitgesendet werden
- Server vergleicht mitgesendetes Token mit erwartetem

ANGULARJS

- HTTP-Interceptor Konzept
- Interceptor für X-XSRF-Token
- Speichert Token Arbeitsspeicher
- Sendet Token bei jedem Request zur gleichen Domain mit

- Problem: *Öffne Link in neuem Tab*
- Server vergibt neues Token wenn gleiche IP

Philipp Burgmer
burgmer@w11k.de

www.w11k.de
www.thecodecampus.de