HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Architekturbeschreibung

Dennis Grabowski, Julius Zint, Philip Matesanz, Torben Voltmer

Masterprojekt „Entwicklung und Analyse einer sicheren Web-Anwendung“
Wintersemester 18/19

11. November 2018

Inhaltsverzeichnis

1 Architekturaufbau

1.1 Klassendiagramm

Das in Abbildung 1.1 dargestellte Architekturdiagramm verzichtet zugunsten der Übersichtlichkeit auf die Abhängigkeiten zwischen den Klassen. Es dient lediglich zur Veranschaulichung der Top-Level-Architektur und zur Illustration der wichtigsten Klassen. Der Zusammenhang der farblich abgetrennten Gruppierungen wird zusätzlich textuell erklärt.

Zu jedem Controller gibt es einen Manager in der Domänenlogik. Dieser enthält jegliche Logik zur Abarbeitung der Anfrage und liefert auftretende Fehler über eine Exception zurück. Ein Datenbankzugriff erfolgt somit niemals von einem Controller, sondern immer aus dem zugehörigen Manager. Beim Sessionmanager handelt es sich um einen Spezialfall, da das Session-Konzept ein integraler Bestandteil der Anwendung ist, da die Prüfung auf vorhandene Authentifizierung Grundlage aller Aktionen ist[1].

Das Gleiche gilt für das `PasswordSecurityModule`, welches beim Login, Erstellen eines Nutzers sowie dem „Passwort zurücksetzen“ benötigt wird, da in ihr alle Passwort-Operationen gekapselt sind. Daher wurden diese beiden Klassen unter „Cross-Cutting-Concerns“ zusammengefasst. Zugriffe auf diese Klassen können sowohl von Controllern als auch von Managern in der Domänenlogik erfolgen.

Im Folgenden geben Sequenzdiagramme weitere Einsicht in das Zusammenspiel dieser Klassen.

[1] Siehe Anforderungsbeschreibung 4.2: Sämtliche Funktionen von HsH-Helfer sind nur angemeldeten Benutzern zugänglich.

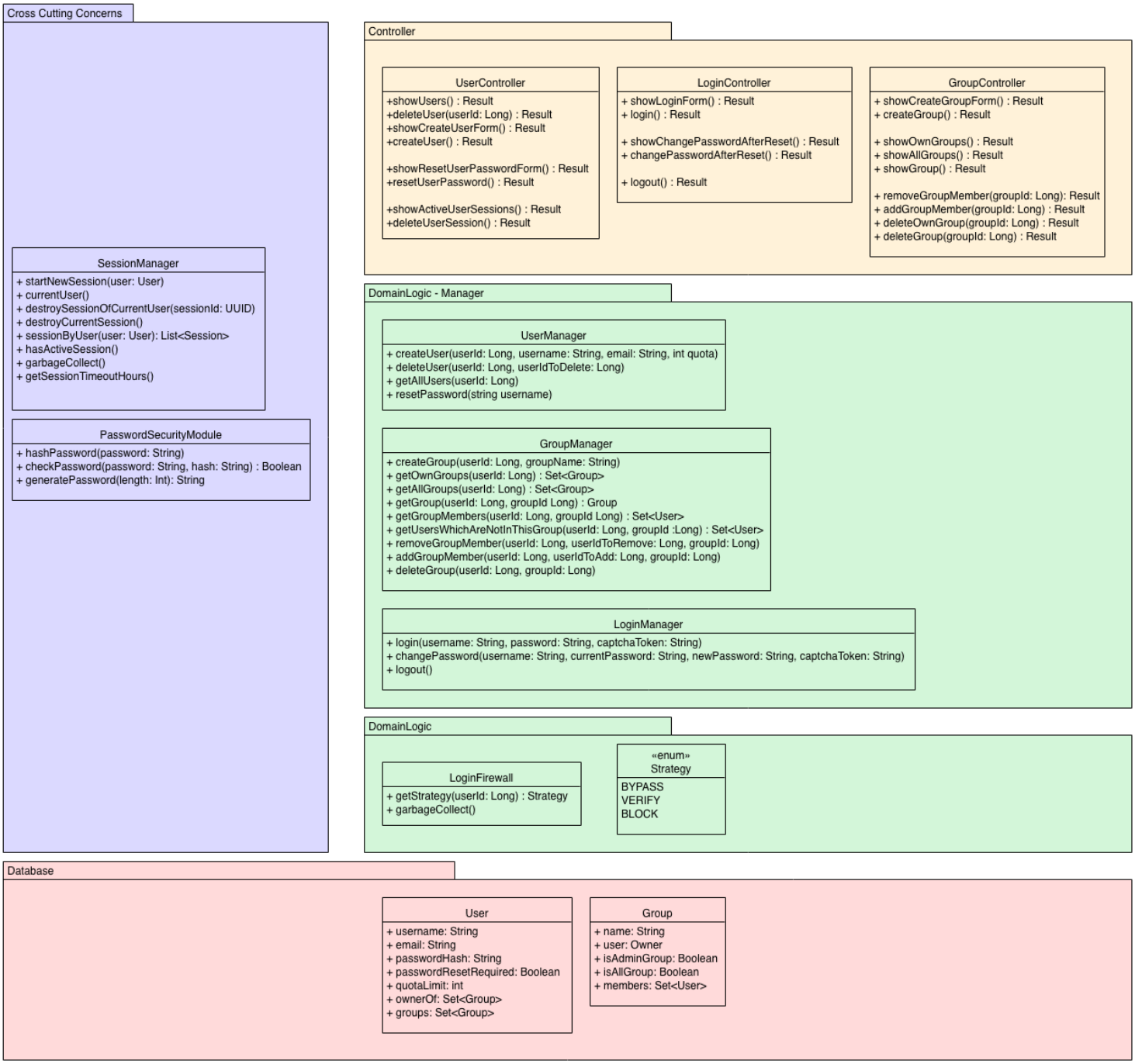

Abbildung 1.1: „High-Level“-Klassendiagramm unseres Systems

1.2 Sequenzdiagram: Benutzer erstellen

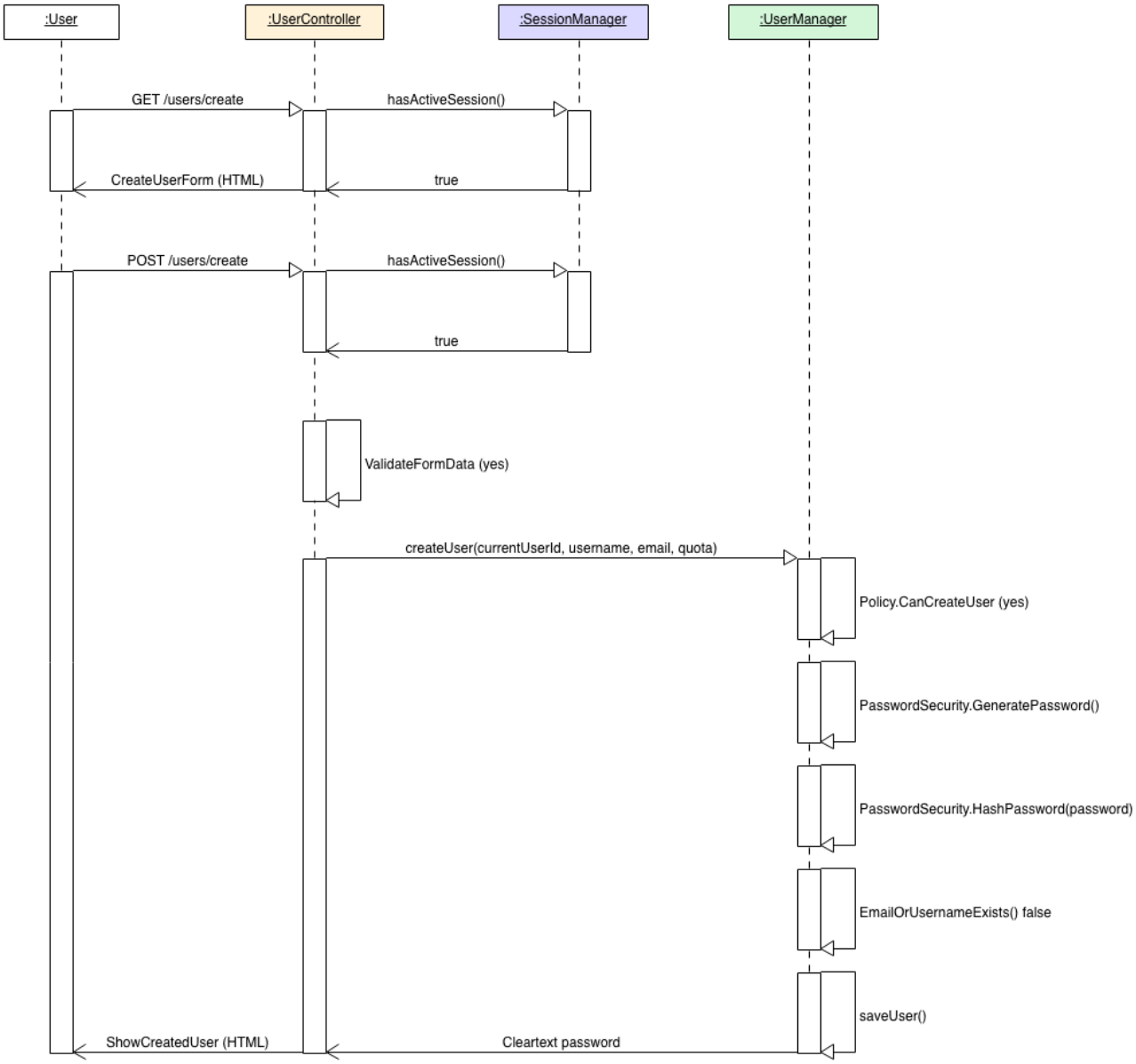

Abbildung 1.2: Sequenzdiagramm für „Benutzer erstellen“-Prozess

Beim Anlegen eines neuen Benutzers wird zuerst durch ein HTTP GET die Form angefragt. Dies ist notwendig um ein gültiges CSRF-Token zu erhalten, welches beim anschließenden HTTP POST mitgeliefert werden muss. Auch hier wird bereits, wie aus dem Sequenzdiagramm ersichtlich, überprüft ob der anfragende User authentisiert ist.

Ist das der Fall, wird eine HTML Form zurückgeliefert, die ausgefüllt und durch eine weitere Anfrage wieder an den Server geschickt wird. Hier wird zuerst über die in Play integrierte Möglichkeit der Annotations eine erste Validierung der Eingaben vorgenommen. Somit werden Eingaben mit nicht unterstützten Zeichen oder ungültiger Länge früh herausgefiltert und dem Benutzer durch Anzeigen von Fehlermeldungen direkt am Eingabefeld mitgeteilt. Sind alle Eingaben korrekt, wird der Aufruf an den Manager weitergeleitet. Dieser befragt zuerst die Policy, ob der authentisierte Benutzer autorisiert ist, einen neuen Benutzer anzulegen. Nach erfolgreicher Prüfung generiert er ein neues Passwort, hashed es und legt in einer Transaktion einen neuen Benutzer an. Das Klartext-Passwort wird zurückgegeben und dem Administrator angezeigt, um dies dem neuen Benutzer zukommen zu lassen.

1.3 Sequenzdiagram: Login

Das Login-Sequenzdiagramm zeigt den Fall, dass der Benutzer bereits mehrfach das Passwort falsch eingegeben hat und ein Captcha lösen muss. In der Antwort, die auf die erste GET Anfrage folgt, sind sowohl das CSRF-Token als auch das HTML zum Lösen des Captchas enthalten. In der darauffolgenden POST Anfrage werden die Captcha-Information, sowie der Benutzername und das Passwort als Form input mitgeliefert. Bei beiden Anfragen wird noch sichergestellt, dass der Benutzer nicht bereits eine gültige Session besitzt, um mehrfache Logins zu vermeiden. Die Eingaben werden, wie auch schon bei Nutzer erstellen, zuerst durch Annotations validiert und anschließend an den `LoginManager` weitergereicht. Diese berechnet unabhängig von der Existenz des Nutzerkontos den Hash des Klartext-Passworts und vergleicht diesen mit dem Hash, der zum Benutzerkonto abgespeichert wurde, falls vorhanden. Anschließend wird die Login-Firewall befragt, ob es Unstimmigkeiten für diesen Benutzer oder die anfragende IP-Adresse gibt. In dem hier vorgestellten Fall liefert diese `Verify` zurück, was bedeutet, dass dieser Benutzer ein Captcha lösen muss. Anschließend wird überprüft, ob es erfolgreich gelöst wurde und dann wird beim `SessionManager` durch den Aufruf von `startNewSession` eine neue Session erzeugt. Damit wurde der Login erfolgreich abgeschlossen.

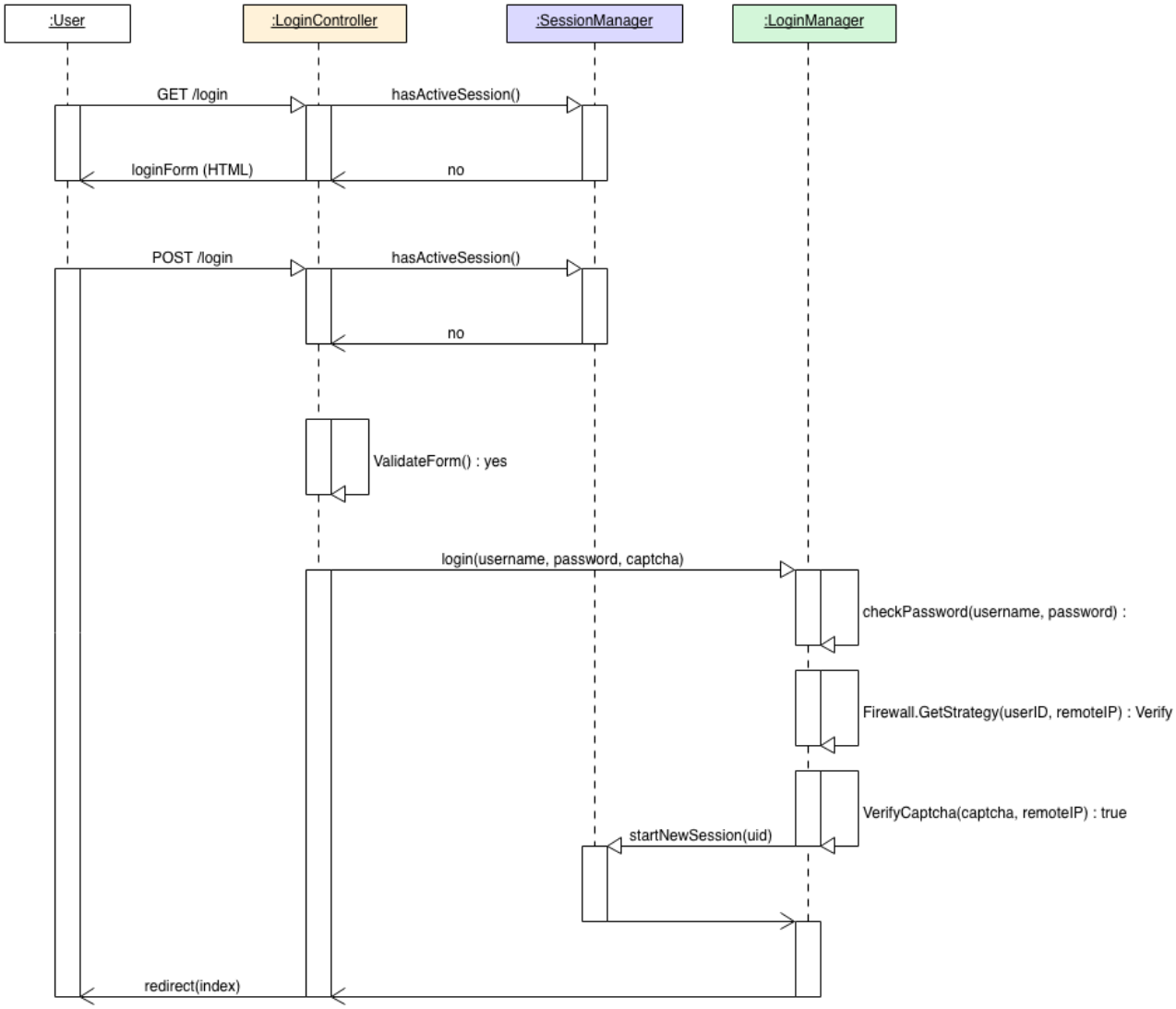

Abbildung 1.3: Sequenzdiagramm für „Login“-Prozess

2 Kryptografisch relevante Informationen

Folgender Abschnitt enthält gesammelte Informationen über die kryptographischen Algorithmen, die im HsH-Helfer Anwendung finden. Die für die Algorithmen benötigten Geheimnisse und deren Verwaltung wird ebenfalls hier dokumentiert, da die Sicherheit von der Geheimhaltung dieser abhängt.

2.1 Passwörter

Die Passwörter, die Benutzer verwenden um sich beim HsH-Helfer zu authentisieren, werden grundsätzlich nur als Hash persistiert. Konkret wird die Hashfunktion bcrypt zum Hashen der Passwörter verwendet.

Um ein Passwort mit bcrypt zu Hashen muss zunächst ein Salt generiert werden. Neben dem Salt und dem gehashten Passwort wird bei bcrypt außerdem noch die bcyrpt Version und die Anzahl der Runden gespeichert. Über die Anzahl der Runden kann die Laufzeit, die bcrypt benötigt, um ein Passwort zu hashen, beeinflusst werden.

Die verwendete Bibliothek ist jBCrypt von mindrot.org (org.mindrot.jbcrypt) in der Version 0.4. Sie verwendet intern die Java Klasse SecureRandom als CPRNG.

Auch zur Generierung der initialen (und temporären) Benutzer-Passwörter beim Anlegen eines neuen Benutzerkontos wird SecureRandom als Basis verwendet. Dafür zuständig ist die Klasse `PasswordGenerator`. Die generierten Passwörter können die Zeichen `a-z`, `A-Z`, `0-9` und die Sonderzeichen `!%?#-_*+` enthalten.

2.2 CSRF-Tokens

Play verwendet zum Generieren von CSRF-Tokens die Java Klasse SecureRandom als CPRNG. Für jede Play Session und jeden Request wird aus dem Token und einer Nonce ein mit HMAC-SHA1 signiertes CSRF-Token generiert.

2.3 Play Cookies

Play verwendet ein „JSON Web Token (JWT)“ als Session Cookie mit dem Namen *PLAY_SESSION*. Es ist durch eine HMAC-SHA256 Signatur gegen Manipulationen geschützt. Das Cookie wird durch HsH-Helfer für die Implementierung des Session-Konzepts genutzt. Außerdem sind im JWT das aktuell gültige CSRF Token und zwei Zeitstempel enthalten.

Neben dem Session Cookie gibt es noch das Flash Cookie *PLAY_FLASH*, was allerdings in der aktuellen Iteration keinerlei Anwendung findet. Der Error-Handling-Code verwendet zwar den Flash-Scope um Fehlermeldungen über eine Klassengrenze hinweg verfügbar zu machen, diese Daten werden aber innerhalb einer Anfrage direkt wieder gelesen und landen somit niemals beim Client.

Die Dokumentation von Play vermerkt, dass das Flash Cookie nicht signiert wird:

> The Flash scope works exactly like the Session, but with two differences:
> - data are kept for only one request,
> - the Flash cookie is not signed, making it possible for the user to modify it.
> [1]

Dies hat sich allerdings bei einem Test als falsch herausgestellt. Das Flash Cookie wird in unserer Anwendung sowohl signiert als auch validiert, wenn es vom Client bei einer weiteren Anfrage wieder an den Server geschickt wird.

2.4 Schlüsselmanagement

Für das System sind mehrere Schlüssel und Zugangsdaten notwendig:

- Der private Schlüssel zum signieren der Play Cookies
- Der private Schlüssel für reCAPTCHA
- Der öffentliche Schlüssel für reCAPTCHA
- Zugangsdaten für den E-Mail Server
- Zugangsdaten für die Datenbank

Diese werden in der `secrets.conf` Datei auf jedem Entwicklungsrechner der Teammitglieder persistiert, welche nicht versioniert wird. Die `application.conf` bindet diese Datei mit dem `include`-Statement ein und somit werden die Schlüssel in der Anwendung verwendet. Dies hat den offensichtlichen Vorteil das Zugangsdaten nicht im Repository einsichtlich sind, aber Anwendungseinstellungen jedoch schon.

3 Sicherheitsmaßnahmen

3.1 Login-Firewall

Der Login einer Web-Anwendung ist potentiell anfällig für Brute-Force-Angriffe, die darauf abzielen, eine valide Kombination aus Benutzername und Passwort zu finden. Aus diesem Grund muss die Anwendung fehlerhafte Zugriffsversuche protokollieren und wenn eine Häufung selbiger auftritt, Maßnahmen ergreifen, um den Angriff zu verlangsamen oder gar zu stoppen. Die zu ergreifende Maßnahme sollte so gewählt werden, dass sie nicht selbst einen Angriffsvektor darstellt, wie z.B. das vollständige Sperren eines Accounts nach N falschen Logins - man könnte so legitime Nutzer vorsätzlich an der Benutzung der Anwendung hindern (Denial of Service).

Aus diesem Grund verfügt unsere Anwendung über eine Komponente[1], die falsche Logins protokolliert und eine der folgenden Maßnahmen ergreift:

- Spezifischen Benutzer-Account in Captcha-Mode versetzen
- Spezifische IP-Adresse in Captcha-Mode versetzen
- Spezifische IP-Adresse von Logins ausschließen

Beim Captcha-Mode handelt es sich um eine Maßnahme, bei der ein Login lediglich möglich ist, wenn zugleich ein Google reCAPTCHA gelöst wird. Eine vollständige Sperrung eines Accounts findet durch unsere Firewall so nicht statt. Es wurde die Annahme getroffen, dass das automatisierte Lösen der reCAPTCHAs zu aufwändig & teuer für einen Angreifer ist. Zugleich ist das „unnötige“ Lösen eines Captchas für das Opfer ein vertretbares Hindernis.

Lediglich IP-Addressen werden als schärfste Maßnahme vollständig von Logins ausgeschlossen. Sofern ein Angreifer nicht die gleiche IP-Adresse wie sein Opfer verwendet, kann er sich höchstens selbst vom Dienst ausschließen. Sollte eine Situation vorliegen, bei der sich eine Vielzahl an Nutzern eine IP-Adresse teilt, besteht die Möglichkeit, diese von der vollständigen Sperre auszunehmen.

Die interne Datenstruktur der Login Firewall ist darauf optimiert, möglichst wenig Speicher zu verwenden und nutzt Buckets, um aggregierte Informationen über Login-Versuche zu speichern. Die Buckets bilden ein fixes Zeitinterval ab. Führt ein Angreifer

[1] policy.ext.loginFirewall

eine Million fehlerhafte Logins in zehn Minuten durch, werden lediglich zwei Datenbank-Einträge erstellt: Einen mit Bezug auf den betroffenen Nutzer-Account und einen mit Bezug auf die IP des Angreifers. Beide Einträge verfügen über eine Zählvariable, die jeweils die Anzahl der fehlerhaften Logins beinhaltet.

Der Verweis auf den Nutzer-Account findet über die numerische und eindeutige ID statt, über die jeder Account verfügt. Jener Fallgestaltung, bei der ein Login bei einem nicht-existenten Account erfolgt, wurde ebenfalls Rechnung getragen. Hierbei ist es erforderlich, dass die Firewall exakt so funktioniert, wie bei existierenden Accounts, um ein Informationsleck zu verhindern: Würde die erste Maßnahme nie oder anders greifen als bei tatsächlich existenten Accounts, wäre es möglich zu prüfen, ob ein Account beziehungsweise der dazu korrospondierende Benutzername tatsächlich existiert[2].

Zu diesem Zweck wird bei nicht-existenten Accounts der Benutzername mittels Hashing auf eine „virtuelle“ User-ID gemappt. Die User-ID wird über ein Long repräsentiert, es wird jedoch ausschließlich der positive Zahlenraum verwendet. Vom Username wird der MD5-Hash gebildet, die ersten 8 Byte werden als Long verwendet und gegebenfalls invertiert. Der so resultierende negative Long-Wert wird in diesem Fall als Grundlage für die Protokollierung fehlerhafter Logins verwendet. Dass MD5 kryptografisch bereits längst als geknackt gilt, ist irrelevant. Wir machen uns lediglich die gute Streuung und hohe Geschwindigkeit von MD5 zu eigen.

3.2 Session-Konzept

Integraler Bestandteil unserer Anwendung ist eine Session-Funktionalität. Theoretisch bietet Play bereits eine Implementierung unter Verwendung von JWT mit der es möglich ist, Daten in Form von signierten Cookies im Client-Browser zu speichern (vgl. 2.3). Die Sicherstellung der Authentifizierung ist jedoch für nahezu jede Funktionalität unserer Anwendung von Bedeutung. Aus diesem Grund wurde das Session-Konzept soweit abstrahiert, dass eine Java-Annotation über einer Controller-Klasse oder einzelnen Controller-Methoden ausreicht, um zu garantieren, dass ausschließlich authentifizierte Nutzer auf selbige zugreifen können.

Die Annotation bewirkt, dass der korrospondierende Request vor Erreichen der dazugehörigen Controller-Methode abgefangen wird und einer Prüfung unterzogen wird, ob der Absender authentifiziert ist. Ist dies nicht der Fall, wird die weitere Ausführung abgebrochen und der Nutzer auf die Login-Seite weitergeleitet. Ansonsten wird der Request an den entsprechenden Controller weitergereicht.

Zur Umsetzung dieser Funktionalität setzen wir auf dem Session-Konzept von Play auf. Wird eine neue Session initialisiert, wird ein dazugehöriger Eintrag in der Datenbank erstellt. Dieser beinhaltet das Initialisierungsdatum, die dazugehörige IP-Adresse und

[2] Diese Angriffsform nennt man auch „Side-Channel Attack“

User-ID sowie eine kryptografisch sichere „Universally Unique Identifier (UUID)“ als Identifikator. Die UUID wird in der Play-Session persistiert. Ein Angreifer kann aufgrund der JWT-Signatur nicht beliebige UUIDs an unsere Anwendung schicken.

Die UUID selbst dient als 2nd Layer of Defense: Sollte die JWT-Signatur geknackt werden, muss der Angreifer noch eine in der Datenbank enthaltene UUID erraten. Selbst wenn er diese Hürde meistert, muss er noch die dazugehörige IP-Addresse kennen, sowie über diese auf unsere Anwendung zugreifen können und dies in einem engen Zeitfenster schaffen, da die Lifetime einer Session begrenzt ist.

Die Anwendung verfügt über eine Ansicht, die dem Nutzer sämtliche „aktive“ Sessions anzeigt und ihm die Möglichkeit bietet, diese nach Bedarf einzeln zu entfernen. Ein Logout-Vorgang führt ebenfalls zur Entfernung der korrospondierenden Session. Um eine Vermüllung der Datenbank zu vermeiden, läuft im Hintergrund ein Prozess, der in festen Zeitintervallen Sessions aus der Datenbank entfernt, die „veraltet“ sind, denn wenn der Benutzer sich nicht ausloggt, sondern einfach den Browser schließt, kann unsere Anwendung keine Kenntnis davon erlangen.

3.3 Content Security Policy

Play verwendet standardmäßig eine sehr restriktive Content Security Policy, die XSS-Angriffe maßgeblich erschwert: Inline-Javascript wird vom Client-Browser grundsätzlich blockiert, es ist lediglich möglich, ein externes Script in eine HTML-Seite einzubinden, das jedoch von der gleichen Top-Level-Domain stammen muss. Zugleich muss dieses Script explizit Javascript als Content-Type im HTTP-Header benennen.

Da unsere Anwendung zum jetzigen Zeitpunkt nur mittelbaren Gebrauch von JavaScript macht, haben wir uns dazu entschlossen, die Content Security Policy zu verschärfen: JavaScript wird grundsätzlich nicht mehr ausgeführt, es sei denn, es handelt sich um die JavaScript-Libraries von reCAPTCHA. Zu diesem Zweck haben wir die entsprechenden Quell-URLs auf die Whitelist gesetzt und somit gleichzeitig die Annahme getroffen, dass wir Google als Host-Provider der Bibliotheken vertrauen[3].

Ebenso haben wir uns entschlossen, Browser, die Content Security Policy nicht unterstützen, von unserer Anwendung auszuschließen. Hiervon sind nur hinreichend alte Browser betroffen[4].

[3] Erforderliche Urls für die CSP Whitelist: https://developers.google.com/recaptcha/docs/faq
[4] Siehe „Browser compatibility“ https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

3.4 Eingabevalidierung

Für einige Parameter gibt es festgelegte Constraints, die in allen entsprechenden Dtos verwendet werden. Das sind:

- username
  - Maximale Länge: 20
  - Regex: [a-z][a-z0-9.]+
- password
  - Maximale Länge: 72 (Beschränkung von bcrypt)
- groupname
  - Minimale Länge: 3
  - Maximale Länge: 20
  - Regex: [a-z][a-z0-9.]+
- email
  - Regex: Siehe Play Implementierung von `EmailValidator`

4 Implementierte Zusatzfunktionalität

4.1 Nutzer kann mit seinen aktiven Sessions interagieren

Im Sinne von Ka-Ping Yee, der innerhalb seines Papers „User interaction design for secure systems“ [4] zehn Designprinzipien beschrieben hat, die Benutzer bei Bedienung eines sicheren Systems benutzerfreundlich unterstützen sollen, haben wir mithilfe dieser neuen Funktionalität das Designprinzip „Revocability“ abgebildet.

Ein Nutzer ist in der Lage, seine eigenen, aktiven Sessions in tabellarischer Form zu betrachten. Während dieser Ansicht kann der Nutzer einzelne Sessions invalidieren, sofern er es für nötig hält.

5 Verworfene Entwürfe

5.1 Trennung der Zuständigkeiten des Controllers

Innerhalb unserer ersten, prototypischen Implementation hatten wir einen Ansatz verfolgt, bei dem unsere `Controller`-Klassen für das Bearbeiten der Requests, das Validieren der Formulare, das Weiterleiten von Fehlern innerhalb eines Formulars, die Autorisierungsprüfung, die Ausführung der domänenspezifischen Operation sowie das Absenden des generierten HTMLs zuständig war. Auch wenn uns bewusst war, dass dies nur ein Ansatz für das Prototyping war, ist schnell offensichtlich geworden, dass dieser uns langfristig gesehen Probleme bereiten würde. Die Testbarkeit solcher Controller-Methoden litt schwer, da es kompliziert war, für einen einzelnen Unit-Test alle Klassen zu mocken, die benötigten Voraussetzungen zu erstellen, und zu prüfen, ob die richtigen Nachbedingungen gelten. Zusätzlich musste jeder dieser Tests entweder gegen eine `FakeApplication` oder gegen ein laufendes Testsystem laufen.

Um diese Testbarkeit wieder zu gewinnen, haben wir die Zuständigkeiten feiner aufgeteilt und die `Manager`-Klassen ins Leben gerufen, die von nun an die domänenspezifischen Operationen übernehmen. Die `Controller` beschäftigen sich seither nur noch mit dem Bearbeiten der Requests, dem Weiterleiten von Fehlern innerhalb eines Formulars zum Nutzer sowie dem Absenden des generierten HTMLs.

5.2 Session-Konzept / Datenschutz

Aufgrund von Datenschutzbedenken haben wir zunächst die Speicherung der mit einer Session-Verknüpften IP-Addresse im JWT-Cookie selbst forciert. Wir waren der Annahme, die IP Addresse sei ein personenbezogenes Datum und die Speicherung von selbigen in unserer Datenbank zumindest fragwürdig. Die Verlagerung der Speicherung auf den Cookie-Speicher des Client-Browsers wäre eine potentielle Umgehungsstrategie für diese rechtliche Problematik gewesen: Die IP-Adresse wird zwangsläufig vom Nutzer bei jedem Request „übertragen“. Herr Prof. Dr. Peine hat uns jedoch mitgeteilt, dass Datenschutzbedenken für dieses Projekt nicht von Bedeutung sind. Aus diesem Grund wurde diese Idee verworfen, wir speichern die IPs in unserer Datenbank.

5.3 Session-Konzept / Numeric IDs

Ursprünglich verfügte eine in unserer Datenbank abgebildete Session über eine numerische ID, die inkrementell aufsteigend war. Das JWT-Token beinhaltet den Schlüssel der korrospondierenden Session-Entität in der Datenbank. Würde ein Angreifer die JWT Signatur knacken, könnte er leicht passende IDs erraten: Er müsste sich anmelden, wüsste dann, welche ID „aktuell“ ist und könnte anhand dieser Information passende IDs ausprobieren. Wir haben diese Implementierung zugunsten von kryptografisch sicheren UUIDs verworfen, die ein „second layer of defense“ darstellen (vgl. 3.2).

6 Literatur

[1] *Cryptography API: Next Generation*. URL: https://www.playframework.com/documentation/2.6.x/JavaSessionFlash (besucht am 10. 11. 2018).

[2] M. Jones, J. Bradley und N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. http://www.rfc-editor.org/rfc/rfc7519.txt. RFC Editor, Mai 2015. URL: http://www.rfc-editor.org/rfc/rfc7519.txt.

[3] Paul J. Leach, Michael Mealling und Rich Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. http://www.rfc-editor.org/rfc/rfc4122.txt. RFC Editor, Juli 2005. URL: http://www.rfc-editor.org/rfc/rfc4122.txt.

[4] Ka-Ping Yee. „User interaction design for secure systems“. In: *International Conference on Information and Communications Security*. Springer. 2002, S. 278–290.

Abkürzungsverzeichnis

DFD Datenflussdiagramm *Glossareintrag:* Datenflussdiagramm

JWT JSON Web Token 9, *Glossareintrag:* JSON Web Token

UUID Universally Unique Identifier 12, *Glossareintrag:* Universally Unique Identifier

Glossar

JSON Web Token Ein auf JSON basiertes Access-Token, standardisiert in RFC7519 [2]. Ermöglicht den Austausch von verifizierbaren Daten. 9

Universally Unique Identifier Ein Standard für Identifikatoren, der sich besonders dazu eignet, Informationen auf verteilten Systemen ohne zentrale Koordination eindeutig kennzeichnen zu können. [3] 12

Abbildungsverzeichnis