Dynamische Webanwendungen gegen Cookie Missbrauch und ungewollte Code Ausführung absichern

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Nihad Abou-Zid
Matrikelnummer 01426835

ausgeführt am
Institut für Information Systems Engineering
Forschungsgruppe Industrial Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Clemens Hlauschek

Wien, 24. Juni 2020

Kurzfassung

Eine vorherrschende Tatsache in der Entwicklung von Webanwendungen ist, dass viele der bereits seit über 10 Jahren bekannten Sicherheitslücken in neu entwickelter Software oft dennoch vorhanden sind. Das ist eine nicht zufriedenstellende Tatsache, mit der große Risiken einhergehen. Ein großes Risiko entsteht vor allem durch die weite Verbreitung von Webanwendungen in unserem Alltag für verschiedenste Zwecke wie Social Media, Online-Shops, Online-Banking oder Webseiten zum Ansehen von Filmen. Je nach Anwendung können die Folgen des Ausnutzens solcher Sicherheitslücken für ein Unternehmen und deren Benutzer gravierend sein. In solchen Fällen können beispielsweise Unternehmensdaten oder Benutzerdaten gestohlen werden sowie die Verfügbarkeit von Unternehmensleistungen gestört werden. Diese Arbeit hilft, durch eine verständliche Aufarbeitung der relevantesten Sicherheitslücken sowie Sicherheitsmaßnahmen, die sichere Entwicklung von dynamischen Webanwendungen zu vereinfachen und zu beschleunigen. Abschließend behandelt die Arbeit, als zentrales Thema, eine aktuelle Schadsoftware namens *Poisontap* und stellt eine Browsererweiterung Browsererweiterung namens *PoisonProtect* vor welche den Einsatz von Poisontap erkennt und unterbindet.

Schlüsselwörter

Sicherheitslücken, dynamische Webseiten, Poisontap, Browsererweiterung

Abstract

A prevalent fact in software engineering of web applications is that many of the vulnerabilities that are known for over 10 years are present in newly developed software. That is a quite unsatisfying fact which leads to high risks for users and companies. The main reason behind the high risk lies in the high usage of web applications for daily tasks such as social media, online-shops, online-banking or streaming platforms. Depending on the application the consequences can be dramatic as attacks often result in the theft of information belonging to a company as well as user data. Also the disruption of services offered by a company is a common goal of attackers. This work helps on leading to a faster development of more secure dynamic web applications by providing a comprehensive explanation of the most relevant vulnerabilities and security measures. Afterwards a current exploit called *Poisontap* is discussed as a central part of this thesis. In order to protect from this software a browser extension named *PoisonProtect* is presented. The extension is able to detect the usage of Poisontap and to block any further execution.

Keywords

Security vulnerabilities, dynamic websites, Poisontap, browser extension

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

.NET NET ist eine Entwicklerplatform bestehend aus Tools, Programmiersprachen und Bibliotheken. Eine bekannte Implementierung ist das .NET Framework, welches unter anderem die Entwicklung von Webseiten, Services und Apps unterstützt.[1]

Bookmarklet Bookmarklets sind kleine Programme, welche dem Benutzer zusätzliche Funktionalitäten im Browser zur Verfügung stellen. Vor der Verwendung auf einer Webseite werden sie durch den Benutzer mit einem Klick aktiviert. Sie sind ähnlich wie Browsererweiterungen, allerdings browserunabhängiger.[2]

Buffer Overflow Buffer Overflow ist eine Schwachstelle, bei der auf Grund einer fehlenden Längenüberprüfung von Benutzereingaben, Speicher in einem Programm überschrieben wird. (vgl. Spafford, [64])

Clickjacking bezeichnet eine Angriffsart, bei der ein Angreifer transparente HTML-Elemente über Elemente platziert, die für den Benutzer sichtbar sind. Wenn der Benutzer auf das sichtbare Element klickt erhält das darüberliegende Element das Klick-Event.[3]

Cookie Ein Cookie ist ein Objekt welches Zustandsinformationen zwischen einem Server und einem Client austauscht. Der Cookie wird vom Server gesetzt und im Browser des Clients gespeichert. Cookies werden vorwiegend für das Session-Management verwendet. (vgl. Kristol und Montulli, [34])

Django ist ein Python-Framework für Webentwicklung.[4]

Domäne Domäne ist ein Adressbereich des Internets aus dem sich für einen bestimmten Knoten ein Domäneame. Der Domänename entspricht der Verbindung aller Kantenbeschriftungen vom Knoten zum Wurzelknoten des DNS-Baums, getrennt durch jeweils einen Punkt. Ein Beispiel dafür ist www.google.com. (vgl. Mockapetris und Dunlap, [46])

Fuzz testing Fuzz testing ist eine Art von Softwaretests bei denen Eingaben automatisch generiert werden. Oft wird ein bestimmtes Schema der Eingaben festgelegt. (vgl. Miller, Fredriksen und So, [44])

Hostname Hostname ist die eindeutige Bezeichnung eines Rechners in einem Netzwerk. (vgl. Braden, [6])

LocalStorage LocalStorage ist ein, sich im Browser befindlicher Speicher, dessen Daten kein Verfallsdatum haben.[5]

[1] https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework
[2] https://support.mozilla.org/de/kb/bookmarklets-verwenden
[3] https://owasp.org/www-community/Clickjacking
[4] https://www.djangoproject.com/start/overview/
[5] https://developer.mozilla.org/de/docs/Web/API/Window/localStorage

Man-in-the-Middle Bei einem Man-in-the-Middle-Angriff wird die Verbindung zwischen 2 Kommunikationspartnern, auf der Netzwerkebene, durch einen Angreifer unterbrochen. Der Angreifer gibt sich dabei gegenüber den beiden eigentlichen Kommunikationspartnern als der jeweilige andere Partner aus und kann so Nachrichten gegebenenfalls mitlesen und verändern.[6]

Middlebox Middlebox bezeichnet eine Software, die sich auf IP-Routern befindet und mehr Funktionalität übernimmt als die Standardaufgaben eines IP-Routers. Diese Standardaufgaben sind den Weg eines Datenpakets zu bestimmen, sowie dieses weiterzuleiten. Middleboxen können unter anderem die Adressen und Nutzlast der Pakete ändern, sowie die Reihenfolge in der sie gesendet werden oder das Senden der Pakete ganz verhindern (vgl. Carpenter und Brim, [8]).

Mixed content Webseiten die HTTPS für eine sichere Übertragung verwenden, aber ebenso Ressourcen beinhalten, die über HTTP geladen werden, bezeichnet man als Mixed Content Webseiten.[7]

OSI-Modell ist ein Modell welches den Informationsaustausch zwischen verschiedenen Anwendungen und vor allem auch in verschiedenen Netzwerken, in Schichten aufteilt.1, [1]

Polymorphismus Ist ein Programmierkonzept welches ermöglicht verschiedene Methodenimplementierungen mit der gleichen Signatur zu haben. Polymorphismus tritt gemeinsam mit Vererbung auf. Welche Implementierung aufgerufen wird ist davon abhängig welche Klasse das Objekt hat auf dem der Aufruf ausgeführt wird. (vgl. Milner, [45])

Proxyserver ist eine Schnittstelle zwischen einem Client und einem Service. Die ganze Kommunikation läuft über den Proxy. Es gibt verschiedenste Gründe für den Einsatz einer solchen Schnittstelle. Einige davon sind Abstraktion (Verbergen einer darunterliegenden Logik), Lokalität (Caching am Proxy), Zurverfügungstellung von Funktionalitäten wie Autorisierung oder Parametervalidierung) (vgl. Shapiro, [62])

Race Condition Race Conditions treten auf wenn Daten von verschiedenen Prozessen, ohne eine explizite Synchronisierung, verwenden. Das Verhalten des Programms hängt dabei davon ab in welcher Reihenfolge die Prozesse auf die gemeinsamen Daten zugreifen. (vgl. Netzer und Miller, [49])

React ist eine JavaScript Bibliothek zur Erstellung von Benutzerschnittstellen (engl. *user Interfaces*) für Webanwendungen.[8]

Regular Expression ist eine Zeichenkette die der Mustererkennung in Texten dient. Mit Texten ist hier die Bedeutung aus Programmiersprachen gemeint. Somit können Texte auch Zahlen oder nicht sichtbare Zeichen sein beziehungsweise enthalten.[9]

Session-Hijacking Übernimmt die Sitzung (engl. *session*) zwischen zwei Kommunikationspartnern. Der Angreifer kann somit eine fremde Sitzung für sich selbst nutzen, um Anfragen in einem eingeloggten Zustand zu erstellen.

[6] https://owasp.org/www-community/attacks/Man-in-the-middle_attack
[7] https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content
[8] https://reactjs.org
[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

Side-channel attack ist eine Angriffsart bei der Unterschiede bei der Performance von Berechnungen, dem Stromverbrauch, der Geräuschentwicklung oder ähnlichen Merkmalen ausgenutzt werden um unauthorisiert an Informationen zu gelangen (vgl. Schwarz, Lipp und Gruss)

Spoofing Das Vortäuschen einer Identität – Beim E-Mail-Spoofing beispielsweise wird eine E-Mail abgeschickt und eine falsche Absendeadresse vorgetäuscht.

Tautologie ist eine Wahrheitsbedingung die, unabhängig von den Variablenbelegungen, immer wahr ergibt.

Webpack ist ein *module bundler* dessen Aufgabe es ist verschiedene, in einer Anwendung verwendeten Ressourcen zusammenzufassen und als ein Bündel an den Browser zu liefern (vgl. Koppers und contributors, [33]).

WebSocket Ist ein Protokoll für den Datenaustausch zwischen Client und Server. Im Gegensatz zu HTTP ist es auf bidirektionale Kommunikation ausgerichtet. Die Verbindung bleibt solange erhalten, bis sie von einem der beiden Kommunikationspartner geschlossen wird. Das ist beispielsweise der Fall, wenn der zugehörige Browsertab geschlossen wird (vgl. [16]).

X-Frame-Options Ist ein HTTP-Header welcher vom Server in einer Antwortnachricht angegeben werden kann um festzulegen, ob die Antwort im aufrufenden Browser in einem *frame*-Objekt verwendet werden darf (vgl. Ross und Gondrom, [58]).

Abkürzungen

ACAO Access-Control-Allow-Origin

API Application Programming Interface

BCP Browser Cache Poisoning

CA Certification Authority

CDN Content Delivery Network

CORS Cross-Origin Resource Sharing

CSP Content Security Policy

CSRF Cross-Site Request Forgery

CSS Cascading Stylesheet

DDoS Distributed Denial of Service

DFA Deterministic Finite Automaton

DLL Dynamically Linked Library

DNS Domain Name System

DNSSEC Domain Name System Security Extensions

DOM Document Object Model

DOS Denial of Service

ECMA European Computer Manufacturers Association

FIFO First in – First out

GPS Global Positioning System

HSTS HTTP Strict Transport Security

HTML Hyptertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IETF Internet Engineering Task Force

IP Internet Protocol

JSON JavaScript Object Notation

KB Kilobyte

MB Megabyte

NFA Nondeterministic Finite Automaton

RATS Rough Auditing Tool for Security

ReDoS Regular Expression Denial of Service

SOP Same-Origin Policy

SQL Structured Query Language

SQLIA Structured Query Language Injection Attacke

SRI Subresource Integrity

SSL Secure Sockets Layer

THP Transparent Huge Page

TLS Transport Layer Security

URI Uniform Resource Identifier

URL Uniform Resource Locator

USB Universal Serial Bus

WLAN Wireless Local Area Network

XSS Cross-Site-Scripting

1 Einleitung

1.1 Problemstellung

Der Großteil aller aktuellen Webanwendungen ist dynamisch und verwendet im Normalfall unter anderem JavaScript. Um diese Anwendungen nach dem aktuellen Stand der Technik sicher zu gestalten müssen viele verschiedene Maßnahmen getroffen werden. Die schnelle Weiterentwicklung der eingesetzten Technologien führt stetig dazu, dass Sicherheitsmaßnahmen angepasst werden müssen. Teilweise entsteht auch ein Mehrbedarf an Sicherheitsvorkehrungen. Auffallend ist, dass es den Großteil der aktuellen relevanten Sicherheitslücken schon seit vielen Jahren gibt (siehe OWASP-Foundation, [55]), doch trotzdem oft die gleichen Fehler gemacht werden. Wie kann eine dynamische Webanwendung, entsprechend der Schutzbedürfnisse, gegen bösartige Angriffe, am effektivsten abgesichert werden?

1.2 Motivation und Zielsetzung

Das Ziel dieser Arbeit ist es die Überprüfung, ob die sicherheitsrelevanten Empfehlungen eingehalten wurden, einfacher zu machen. Behandelt werden verschiedene Arten von Sicherheitslücken. Ebenso wird näher auf die Schadsoftware "Poisontap"OWASP-Foundation, [29] eingegangen. In diesem Rahmen wird die Erstellung einer Browsererweiterung, die den Einsatz dieses Programms erkennen soll, erläutert. Außerdem beinhaltet diese Arbeit eine Checkliste in der die empfohlenen Maßnahmen in kurzer Form aufgelistet sind.

1.3 Methodik

Die Arbeit basiert auf Literaturrecherche über aktuelle Sicherheitsproblem von Webanwendungen sowie möglicher Gegenmaßnahmen, sowie diverse Grundlagen dieser Thematik. Anschließend werden durch das erlangte Wissen weit verbreitete und gefährliche Sicherheitslücken ausgewählt und ausführlicher behandelt.
Der Konzeption und Entwicklung der Browsererweiterung ging eine Recherche über die Funktionsweise der Schadsoftware Poisontap voraus. Anschließend wurden Schwachstellen der Schadsoftware, bei denen Abwehrmaßnahmen ansetzen können, identifiziert und ein grobes Konzept des Mechanismus, der den Einsatz von Poisontap erkennen soll, erstellt. Anschließend wurde die Implementierung geschrieben und am Ende mit Hilfe der Schadsoftware manuell getestet.

1.4 Aufbau der Arbeit

Diese Arbeit ist in 3 Bereiche eingeteilt. Kapitel 2 behandelt die Grundlagen welche notwendig sind um die folgenden Themen der Arbeit zu verstehen. Die Kapitel 3 und 4 behandeln verschiedene Sicherheitsprobleme sowie dafür geeignete Gegenmaßnahmen. In den Kapiteln 5 und 6 werden zuerst die Schadsoftware Poisontap sowie anschließend die erstellte Browsererweiterung behandelt. In Kapitel 7 befindet sich ein Überblick relevanter, aktueller Arbeiten die mit dem Themengebiet dieser Arbeit in Zusammenhang stehen. Eine Zusammenfassung der gewonnenen Erkenntnisse befindet sich in Kapitel 8. In Kapitel 9 befinden sich die Checkliste zur Überprüfung der umgesetzten Sicherheitsmaßnahmen sowie der Code der Browsererweiterung.

2 Grundlagen

In den folgenden Kapiteln werden, für den Kontext der Arbeit, fundamentale Begriffe und Konzepte, oberflächlich beschrieben. Damit soll ein grundlegendes Verständnis und somit die Basis für die darauf folgenden Kapitel geschaffen werden.

2.1 Hypertext Transfer Protocol

Das Hypertext Transfer Protocol (HTTP) findet in der Datenübertragung in Netzwerken aller Größe Verwendung. Es wird genutzt um zwischen Netzwerkkomponenten Daten verschiedenster Art auszutauschen (vgl. Gettys u. a., [18]).

2.1.1 Übertragungsmethoden

Es gibt 8 verschiedene Übertragungsmethoden und trotzdem genügen die Methoden POST und GET und daher sollten auch nur diese verwendet werden. Alle anderen sollten am Server deaktiviert sein. Die GET-Methode sollte nur verwendet werden, um Informationen vom Server zu bekommen. In GET-Anfragen werden oft Parameter übergeben beispielsweise bei einer Suchanfrage. Da diese Parameter in der Uniform Resource Locator (URL) übertragen werden, sollten mit GET keine sensiblen übergeben werden. Im Grunde könnte man immer die POST-Methode verwenden aber normalerweise wird es dort verwendet, wo GET nicht verwendet werden sollte. Somit kommt eine POST-Anfrage üblicherweise mit einer Nutzlast (engl. *payload*) deren Größe im *content-length*-Header definiert ist. (vgl. Zalewski, [75, S. 52])

2.1.2 Sessionmanagement

Sessions werden verwendet um die Performance zu verbessern, da mit ihnen verhindert wird, dass bei jeder neuen Clientanfrage ein neuerlicher TCP Handshake gemacht wird, da das zu zeitaufwändig wäre. Diese Form von Sessions werden Keepalive Sessions genannt. Sessions werden auch verwendet, damit eingeloggte Benutzer auch eingeloggt bleiben bis sie sich ausloggen. Das ist wichtig, da bestimmte Ressourcen auf Servern oft nur von bestimmten Benutzern oder Benutzergruppen abgefragt werden können sollten (vgl. Zalewski, [75, S. 56]).

2.1.3 Caching

Der Zweck des Cachings ist ebenso die Performanceverbesserung. So einfach es ist statische Inhalte zwischenzuspeichern, so schwierig und komplex ist es bei dynamischen Inhalten. Per Definition wird die Antwort des Servers auf eine GET-Anfrage bei bestimmten Statuscodes (unter anderem 200 und 301) ohne zeitliches Ablaufdatum zwischengespeichert, wobei dieses Verhalten am Server anders definiert werden kann. Diese Antwort kann dann für zukünftige Anfragen wiederverwendet werden, selbst wenn andere Parameter wie zum Beispiel der Cookie-Header abweichen. Um Probleme durch Caching zu vermeiden, sollte bereits vom Server festgelegt werden welche Ressourcen wie zwischengespeichert werden können (vgl. Zalewski, [75, S. 58f.]). Seit der HTTP-Version 1.1 (Gettys u. a., [18]) gibt es den *Cache-Control"*-header mit dem das Verhalten definiert werden kann. Mit dem Wert *public* beispielsweise wird festgelegt, dass diese Ressource für verschiedene Clients verwendet werden kann. Mit *private* wird somit definiert, dass die Ressource nicht in einem geteilten (engl. *shared*) Cache verwendet werden soll. Außerdem kann mit

max-age=value festgelegt werden, wie lange eine Ressource maximal wiederverwendet werden kann (vgl. Fielding, Nottingham und Reschke, [17]). Ebenso kann festgelegt werden, dass die Ressource überhaupt nicht wiederverwendet werden soll. Das geschieht mit dem Wert *no-cache* oder *no-store*. Ein grundlegendes Problem ist, dass Proxyservers und Browser die Header nicht immer gleich interpretieren (vgl. Fielding, Nottingham und Reschke, [17]). Ein weiteres Problem ist, dass beim Verwenden von öffentlichen Drahtlosnetzwerken die Antworten des Servers so verändert werden können, dass sie am Client beispielsweise ohne zeitliches Limit wiederverwendet werden können. Auch die eigentlichen Inhalte können abgeändert werden (vgl. Zalewski, [75, S. 60]). Es gibt verschiedene Arten von Browser-Caches zum Beispiel den Web Cache sowie den HTML5 Anwendungscache. Der Web Cache ist in Browsern standardmäßig für alle Webressourcen aktiv, während der zweite explizit verwendet werden muss. Das heißt die Ressourcen die zwischengespeichert werden sollen, müssen als solche angegeben werden. Im Gegensatz zum Anwendungscache ist der Web Cache global, das heißt die enthaltenen Ressourcen werden von allen Webseiten gemeinsam genutzt (vgl. Yaoqi u. a., [74]).

2.1.4 Cookies

Mit Cookies kann jede Art von Information in Form von Schlüssel-Werte Paaren gespeichert werden. Diese Dateien werden im Browser gespeichert. Sie stellen die häufigste Art der Authentifizierung dar. Auch bei Cookies unterscheidet sich das Verhalten der Browser stark. Ursprünglich sollten Cookies vom Server gelesen werden. Der Server bekommt in der Client-Anfrage den Cookie über den Cookie-Parameter nachdem der Server zuvor eine Antwort an den Client mit dem *Set-Cookie*-Header geschickt hat. Trotz der ursprünglichen Intentionen ist es möglich Cookies mit JavaScript zu lesen, erstellen und verändern. Daher ist es recht naheliegend, dass das zu Sicherheitsproblemen führen kann, vor allem wenn sensible Daten in Cookies gespeichert werden. Mit dem *Expires*-Parameter wird festgelegt, wann der Cookie gelöscht werden soll. Wenn dieser Parameter nicht gesetzt ist, wird er mit dem Schließen des Browsers gelöscht. Oft kann diese Zeitspanne, je nach den Gewohnheiten des Nutzers, mehrere Wochen umfassen. Ein weiterer wichtiger Parameter ist *Domain*. Mit ihm wird der Cookie einer glsdomain zugeordnet anstatt dem Hostnamen der den *Set-Cookie*-Header gesetzt hat. Mit dem *Secure*-Attribut wird festgelegt, dass der Cookie nur über verschlüsselte Verbindungen übertragen werden soll. Ein weiterer wichtiger Parameter ist textitHttpOnly, der festlegt, dass der Cookie nicht von JavaScript gelesen werden kann (vgl. Zalewski, [75, S. 60f]).

2.2 Domain Name System

Das Domain Name System (DNS) ist ein Verteiltes System im Internet und zuständig für die Namensauflösung - es findet zu einem Hostnamen die zugehörige Adresse. Es bildet das Rückgrat des Internets, da Menschen normalerweise Hostnamen und nicht ihre Adressen verwenden, zum Datenaustausch aber die Adresse (in erster Linie die IP-Adresse) benötigt wird. Die Spezifikation des DNS[1] wird von der Internet Engineering Task Force (IETF)[2] erstellt und veröffentlicht. Primärer Bestandteil des DNS sind die Namenserver, welche Anfragen zur Namensauflösung annehmen und beantworten. Einen weiteren Bestandteil bilden die sogenannten *Resolver*, welche sich unter anderem auf den Endgeräten des Benutzers befinden können. Sie sind im Betriebssystem integriert und können von Programmen angesprochen werden, welche eine Internet Protocol (IP)-Adresse zu einem Namen benötigen. Sie sind oft auch auf Routern und Nameservern vorhanden. Um die Performance der Namensauflösung zu verbessern spielt das *Caching* eine wesentliche Rolle. Ähnlich

[1] https://tools.ietf.org/html/rfc1035
[2] https://www.ietf.org

wie bei beim Caching im Zusammenhang mit Webseiten (siehe Kapitel 2.1.3) werden hier Ergebnisse von Anfragen zwischengespeichert. Im Fall des DNS speichern *Resolver* die IP-Adresse zu einem Namen lokal für einen gewissen Zeitraum (meistens 1 bis 2 Tage). Solange der Name in der Liste des *Revolvers* vorhanden ist, muss dieser die Anfrage nicht weiterleiten, sondern kann die Anfrage sofort selbst beantworten. Umgekehrt bedeutet das allerdings auch, dass Änderungen einer IP-Adresse zu einem Namen nicht ohne weiteres sofort propagiert werden (Mockapetris und Dunlap, [46]).

Ein Problem bei der genannten Basis-Spezifikation ist, dass ein Empfänger keine Möglichkeit hat festzustellen, ob die Antwort echt ist oder gefälscht wurde. Wenn eine Antwort auf eine Anfrage zur Namensauflösung gefälscht wird, wird üblicherweise die IP-Adresse verändert (siehe Kapitel 3.11). Aus diesem Grund wurde das Protokoll 1999 um zusätzliche Sicherheitsfunktionen erweitert[3]. Domain Name System Security Extensions (DNSSEC) ermöglicht *Resolvern* die Überprüfung der erhaltenen Nachrichten auf ihre Integrität und die Authentifizierung des Absenders. Das wird mit einem asymmetrischen Verschlüsselungsverfahren (*Public-Private Key - Verschlüsselung*) erreicht.

2.3 Transport Layer Security

Transport Layer Security (TLS) ist ein Protokoll, welches in Verbindung mit HTTP verwendet wird und das Secure Sockets Layer (SSL)-Protokoll ablöst. TLS bietet für die Antwort auf eine HTTP-Anfrage folgende sicherheitsrelevante Möglichkeiten: die Authentifizierung des Absenders, das Sicherstellen der Integrität der Nachricht sowie die Gewissheit, dass keine 3. Partei am Weg vom Sender zum Empfänger die Nachricht entschlüsseln kann. Wenn HTTP gemeinsam mit TLS verwendet, wird spricht man von Hypertext Transfer Protocol Secure (HTTPS). Um sich als Server zu authentifizieren enthält das Zertifikat des Servers dessen Domänenname. Das Zertifikat wird von Zertifitierungsstellen (engl. *Certificate Authority*) signiert. Deshalb enthalten Browser die Zertifikate vieler Certification Authority (CA)s (vgl. Modadugu und Rescorla, [47]).

2.4 JavaScript

JavaScript wurde 1995 von Netscape und Sun vorgestellt. JavaScript wird vorwiegend verwendet um Code im Browser des Benutzers auszuführen. Seit dem Jahr 1999 legt European Computer Manufacturers Association (ECMA)[4] die Spezifikationen für diese Sprache fest um die Unterschiede zwischen der Ausführung von JavaScript-Code in verschiedenen Browsern möglichst gering zu halten (vgl. Zalewski, [75, S. 96]). Es gibt auch serverseitiges JavaScript (Node.js[5]), das in den letzten Jahren an Bedeutung gewonnen hat, worauf in dieser Arbeit jedoch nicht näher eingegangen wird. Clientseitiges JavaScript (im Folgenden ist mit JavaScript clientseitiges JavaScript gemeint) wird verwendet, um Webseiten dynamisch zu gestalten. Es ist unter anderem möglich, die Darstellung von Hyptertext Markup Language (HTML)-Elementen zu ändern (beispielsweise den Text, den ein Element beinhaltet) beziehungsweise neu zu erstellen oder zu löschen. Außerdem können Serveranfragen geschickt und die Antworten verarbeitet werden.

[3] https://tools.ietf.org/html/rfc2535
[4] http://www.ecma-international.org
[5] https://nodejs.org/en/

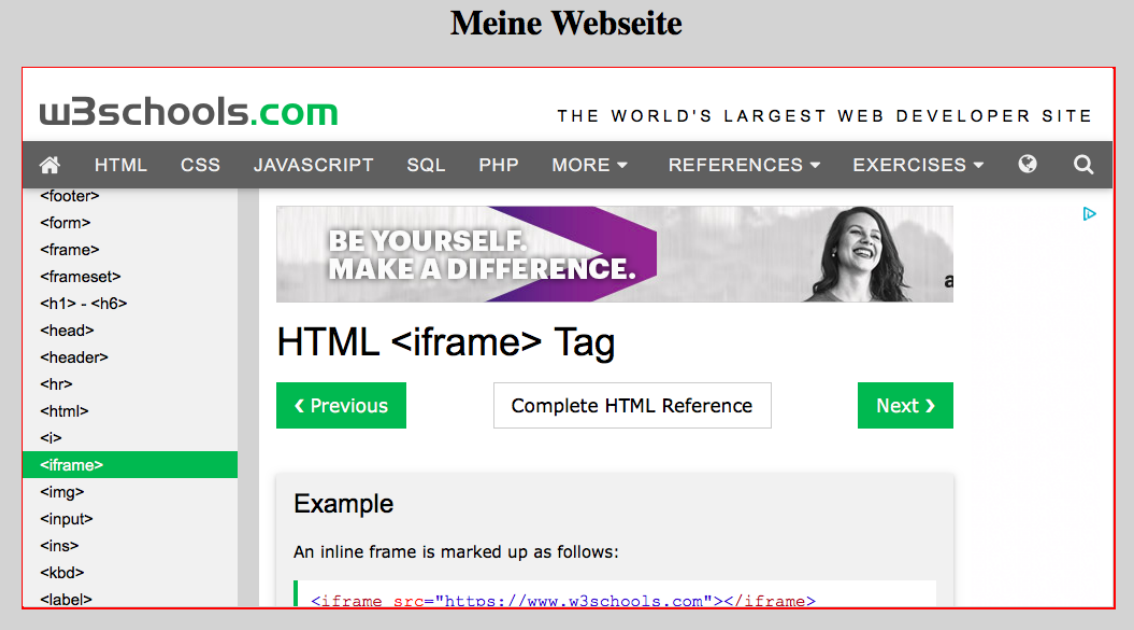

Abbildung 2.1: Einbindung einer Webseite mit einem iframe-Element.

Im Zusammenhang mit JavaScript fällt oft der Begriff jQuery[6]. Das ist eine JavaScript-Bibliothek, durch die viele häufig gebrauchten Funktionalitäten gekapselt angeboten werden. Beispielsweise fällt das Suchen bestimmter Elemente auf einer Webseite oder das Senden und Verarbeiten von Serveranfragen, so oft leichter (vgl. Liang, [37, S. 7]).

2.4.1 Sicherheitsmodell

JavaScript erstellt für den Code jedes unabhängigen HTML-Dokuments eine eigene Laufzeitumgebung. Unter unabhängigen HTML-Dokumenten werden hier solche HTML-Dokumente verstanden, die sich in einem eigenen Browserfenster, Browsertab oder einem Element befinden, welches HTML-Dokuemente einbinden kann - das *iframe*-Element. In diesem Element wird die Webseite geladen als ob sie direkt über die Adresszeile im Browser aufgerufen würde. Abbildung 2.1 veranschaulicht die Verwendung des *iframe*-Elements, durch die Einbindung der W3C-Seite[7], wobei der rot umrandete Bereich das besagte *iframe*-Element ist. Diese verschiedenen Laufzeitumgebungen werden oft auch als Sandboxen, bezeichnet zwischen denen mit Hilfe eines vom Browser angebotenen Application Programming Interface (API) kommuniziert werden kann. Die Ausführung des Codes jeder dieser Umgebungen ist unabhängig und wird im Normalfall nicht von den anderen Laufzeitumgebungen beeinflusst (vgl. Zalewski, [75, S. 97-103]).

Eine interessante und zugleich bedenkliche Tatsache ist, dass es möglich ist vom Browser zur Verfügung gestellte Objekte zu überschreiben. In JavaScript sind Funktionen Objekte. In dem folgenden Codebeispiel wird die Funktion *log* des Objekts *console* mit einer anderen Funktion überschrieben. Statt einen Text oder ein Objekt in der JavaScript-Konsole auszugeben wird die überschriebene Funktion den übergebenen Parameter in einem Warnfenster (engl. *alert box*) anzeigen. Der dafür notwendige Code befindet sich zur Veranschaulichung in dem Codeauschnitt 2.1

[6] https://api.jquery.com
[7] https://www.w3schools.com/tags/tag_iframe.asp

```
1 console.log = function(object) {
2     alert(object);
3 }
```

Listing 2.1: *log*-Funktion des *console*-Objekts überschreiben

Funktionen und Objekte können auch mit null überschrieben werden. Das funktioniert zwar nicht mit allen Objekten (beispielsweise nicht mit dem *window*-Objekt, aber mit dem sich darin befindlichen *console*-Objekt. Ein Angreifer kann diese Tatsache ausnutzen, um das Verhalten von externen JavaScript Programmen zu ändern. Adida, Barth und Jackson, [2] beschreiben Angriffe auf kommerzielle, auf Bookmarklets basierenden Passwortmanager. In den beschriebenen Angriffen wird ausgenutzt, dass sich viele Objekte in JavaScript überschreiben lassen. Der Code eines Bookmarklets wird in der gleichen Laufzeitumgebung ausgeführt wie der Code der Webseite auf der das Bookmarklet gerade aktiviert ist. Der Code des Bookmarklets wird nach dem Code der Webseite geladen (vgl. Adida, Barth und Jackson, [2]).

2.4.2 Parallelismus

JavaScript wird standardmäßig synchron ausgeführt. Das bedeutet, dass der Code zeilenweise ausgeführt wird und zwar in der Reihenfolge in der der Code in der Skriptdatei steht. Das bedeutet auch, dass eine Codezeile erst ausgeführt wird, wenn die Zeile davor fertig ausgeführt wurde. Das führt oft zum Problem, dass Webseiten nicht reagieren zu scheinen. Das kann verschiedene Gründe haben. Einerseits kann es sein, dass eine Berechnung beispielsweise einfach viel Zeit benötigt. Andererseits kann es sein, dass eine Komponente nicht antwortet. Solange das der Fall ist, kann die Webseite standardmäßig nicht auf Aktionen des Benutzers reagieren.
Um dieses Problem zu beheben, gibt es mehrere Ansätze. Es gibt die Möglichkeit Funktionen asynchron auszuführen. Das grundlegende Konzept sind Callbacks, welches folgendermaßen funktioniert: Beim Aufruf einer Funktion wird angegeben was getan werden soll, sobald die Funktion ausgeführt wurde. Sobald die Funktion aufgerufen wurde, wird die nächste Codezeile nach dem Funktionsaufruf ausgeführt. Um Code parallel ausführen zu können, gibt es in JavaScript auch die Möglichkeit Threads, welche im Folgenden als Worker bezeichnet werden, zu verwenden. Einem Worker ist immer eine eigene JavaScript-Datei zugeordnet. Die Kommunikation zwischen Workern funktioniert mittels *onmessage*-Events. Das Codebeispiel 2.2 veranschaulicht die Funktionsweise.

```
1 var worker = new Worker('workerLogic.js');
2     worker.onmessage = function (response) {
3       console.log(response.data);
4     };
```

Listing 2.2: Erstellen eines *worker*-Objekts und Setzen eines Messagelisteners

(vgl. WHATWG, [72])

2.5 Sicherheitstools

Mit Sicherheitstools sind Programme gemeint, die helfen sollen Fehler beziehungsweise Schwachstellen in einer Software frühzeitig zu erkennen. Aktuelle Webanwendungen bestehen meistens aus vielen verschiedenen Komponenten und haben oft auch Abhängigkeiten zu externen Schnittstellen. Das führt zu einer hohen Komplexität die es erschwert, Code manuell auf Fehler zu überprüfen. Der Testprozess kann in folgende Phasen aufgeteilt werden: Target Discovery, Scanning, Result Analysis und Reporting. Der Schwerpunkt liegt in diesem Kapitel auf Scanning und Result

Analysis. Die Phase Scanning beinhaltet das Analysieren verschiedenster Bereiche einer Anwendung. Beispielsweise gehören dazu auch Netzwerkscans der involvierten Server. Näher eingegangen wird hier allerdings nur auf das Scannen der Software auf der Anwendungsebene. Um das Auseinandersetzen mit der Funktionsweise automatisierter Testsoftware zu rechtfertigen, gibt es einige Gründe. Der bedeutendste Grund ist der Faktor Zeit. Webanwendungen werden in verschiedenen Browsern aufgerufen und sollten in jedem der Browser das gleiche Verhalten zeigen. Hinzu kommen eventuelle Abhängigkeiten je nach dem Typ des Endgeräts (mobil vs. Desktop). Das bedeutet, dass die gleichen Testfälle mehrfach getestet werden müssen. Je nach Anforderungen des Softwareherstellers kann das bedeuten, dass nach jeder Änderung einer Funktionalität nur der betroffene Teil oder größere Teilbereiche der Anwendung getestet werden müssen. Ein Problem dabei ist unter anderem, dass Abhängigkeiten von Entwicklern beziehungsweise Testern übersehen werden. So passiert es häufig, dass durch das Ausbessern eines Fehlers neue Fehler in abhängigen Funktionalitäten entstehen. Ein weiteres Problem ist, dass Entwickler und Tester oft nicht ausreichend Wissen über die verschiedensten möglichen Schwachstellen besitzen und die Anwendung somit nur auf eine Teilauswahl bekannter Sicherheitslücken überprüft wird.
Im Folgenden werden die verschiedenen Kategorien von Testtools erläutert. Primär wird die statische von der dynamischen Analyse unterschieden, wobei bei der ersten die zu testende Anwendung nicht ausgeführt wird und bei letzterer schon. Bei der dynamischen Analyse wird oft ein Ansatz verwendet der sich *Fuzzing* oder *Fuzz Testing* nennt. Hier werden Eingaben automatisch oder halbautomatisch zufällig generiert. Diese Technik eignet sich gut für das Finden von folgenden Schwachstellen: Structured Query Language (SQL)-Injection, Cross-Site-Scripting (XSS), Buffer Overflow und Denial of Service (DOS). Auch neue Schwachstellen können mit diesem Ansatz gefunden werden. Beim Fuzzing ist es zu bevorzugen, dass die Testdaten mit einem bestimmten Schema erstellt werden. Das beinhaltet Eigenschaften wie die Verschlüsselung, das Format, die Kodierung oder Ähnliches. Das setzt zwar ein höheres Wissen über die Anwendung voraus und erhöht den Testaufwand, jedoch werden dadurch bessere Ergebnisse erzielt.
Zum Testzyklus gehört weiters das Penetration Testing, bei dem versucht wird aus der Rolle des Angreifers Lücken auszunutzen. Die Ziele dieser Phase sind unter anderem das Beweisen des Vorhandenseins der Lücke beziehungsweise dessen Ausnutzbarkeit und die damit einhergehenden Gefahren. Außerdem werden so oft Fehler gefunden, die die automatisierten Tools nicht gefunden haben (vgl. Shah und Mehtre, [61]).

Im Folgenden werden einige populäre (in den meisten Fällen Open-Source) Testtools vorgestellt.

- Rough Auditing Tool for Security (RATS)[8] ist ein Tool mit dem C/C++, Perl, PHP oder Python Programme analysiert werden können. Die beste Funktionalität bietet diese Software für C und C++ Programme, bei denen Buffer Overflows, Race Conditions und Sicherheitslücken im Design der Software gefunden werden können. Bei den anderen 3 Sprachen beschränkt sich die Funktionalität meistens auf das Hinweisen auf die Verwendung riskanter, in den Programmiersprachen oder durch Bibliotheken angebotene Funktionen (vgl. CERN-ComputerSecurityTeam, [9]).
- FindBugs[9] wird verwendet um Schwachstellen in Java Programmen zu finden und wendet dazu statische Code-Analyse an. FindBugs findet *bad practice* Code wie zum Beispiel das Vergleichen von Objekten mit dem Vergleichsoperator. Es wird aber auch Code gefunden der die Performance und die Sicherheit negativ beeinflusst. Beispielsweise werden Mög-

[8] https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml
[9] http://findbugs.sourceforge.net/findbugs2.html

lichkeiten für Attacken mittels SQL-Injection und XSS gefunden (vgl. SOURCEFORGE, [63]).

- OWASP ZAP[10] ist eines der am meisten genutzten Penetration Testing Tools und ist für alle gängigen Betriebssysteme verfügbar. Die Funktionalität des Proxys kann verwendet werden um Datentransfer zwischen der Webanwendung zu beobachten und zu ändern. Es macht sowohl das passive als auch das aktive Scanning möglich - beim passiven Scannen wird der Datentransfer nur beobachtet, doch auch das reicht oft schon aus um Fehler zu erkennen. Beim aktiven Scannen wird die Anwendung angegriffen. Außerdem gibt es die Möglichkeit mit einem Crawler Seiten zu finden, die übersehen wurden oder versteckt sind (vgl. OWASP-Foundation, [51]).

- Metasploit[11] ist das meistbenutzte Framework für Penetration Tests. Allerdings gibt es bei diesem Programm sowohl eine zahlungspflichtige als auch eine frei verfügbare Version. Auch wenn die zahlungspflichtige Version weitaus mehr Features bietet, ist die freie Version für die meisten Personen, die nicht direkt im Feld der IT-Sicherheit arbeiten, ausreichend. Die zahlunsfreie Version beinhaltet eine Brute-Force-Methode für Zugangsdaten, über 1500 Exploits sowie die Möglichkeit des manuellen Ausnutzens von Schwachstellen (vgl. Rapid7-LLC, [57]).

[10] https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
[11] https://www.metasploit.com

3 Sicherheitsprobleme dynamischer Webanwendungen

Der Grund warum es in vielen Webanwendungen, beziehungsweise Software im Allgemeinen, viele kritische Sicherheitslücken gibt, ist die große Komplexität von Anwendungen sowie meistens auch ein falscher Stellenwert von Sicherheit im Softwareentwicklungszyklus. Sicherheit sollte in alle Phasen der Entwicklung mit einfließen, angefangen mit der Konzepterstellung. Viele Sicherheitslücken gehen schon auf Fehler im Design zurück und sind im Implementierungsprozess nur noch schwer auszubessern oder werden gar nicht erst bemerkt. In diesem Kapitel wird auf eine Auswahl an aktuellen Sicherheitslücken von dynamischen Webanwendungen eingegangen. Die Auswahl orientiert sich an der OWASP Top 10 2017 OWASP-Foundation, [55] Liste der häufigsten, kritischen Risiken bei aktuellen Webanwendungen. Gleichzeitig wird aber darauf Wert gelegt besonders auf Sicherheitslücken einzugehen die ihre Ursachen in der Programmlogik haben und nicht auf einer niedrigeren Ebene.

3.1 Bedrohungsmodell

Der Großteil der im Folgenden vorgestellten Angriffe geht von 2 verschiedenen Typen von Angreifern aus: ein Angreifer welcher sich lediglich eine Schwachstelle einer verwendeten Webseite ausnutzt um Opfer anzugreifen. Eine Angriffsart als Beispiel für diesen Fall ist *Cross-Site Scripting*, beschrieben in Kapitel 4.2. Der andere Angreifertyp manipuliert den Netzwerkverkehr und erreicht so beispielsweise *Browser Cache Poisoning* welches im Kapitel 3.4 beschrieben ist. Angriffe die durch kompromittierte CAs (siehe Kapitel 2.3) oder Sicherheitslücken von Betriebssystemen, Compilern beziehungsweise Interpretern ermöglicht werden.
In den folgenden Abschnitten, die näher auf verschiedene Angriffsarten eingehen, ist mehr Information zu angenommenen Angreifern enthalten.

3.2 Cross-Site Scripting

Cross-Site Scripting ist ein Angriff bei dem bösartiger Code in eine vertraute Umgebung eingeschleust wird. Das ist möglich durch fehlende Validierung beziehungsweise Unschädlich machen (engl. *sanitization*) von Benutzereingaben. Abhängig von der Rolle des Benutzers im System, kann der Angreifer beispielsweise persönliche Daten abgreifen oder verändern, Benutzer anlegen oder löschen, Webinhalte verändern sowie dafür sorgen, dass das System nicht mehr verfügbar ist. Auf diese verschiedenen Angriffsarten wird in den folgenden Unterkapiteln näher eingegangen. In den meisten Fällen handelt es sich um Anwendungen bei denen Benutzer sich einloggen müssen um Änderungen vorzunehmen. Der Angreifer profitiert hier von den Rechten des Nutzers aber vor allem von der Tatsache, dass der Code von einem authentifizierten Nutzer ausgeführt wird (vgl. Hydara u. a., [28]).
Der eingeschleuste Code ist in einer Skriptsprache (meistens JavaScript) geschrieben. Diese Tatsache spiegelt sich auch im Namen "Cross-Site Scripting"wieder. Auch wenn die Zugriffsrechte von JavaScript im Browser ohnehin möglichst gering gehalten werden, hat JavaScript unter anderem folgende wichtige Befugnisse: Zugriff auf Cookies, Kommunikationsmöglichkeiten mit dem Netzwerk, Kontrolle über das Document Object Model (DOM). Gupta und Gupta, [19] nennt 3

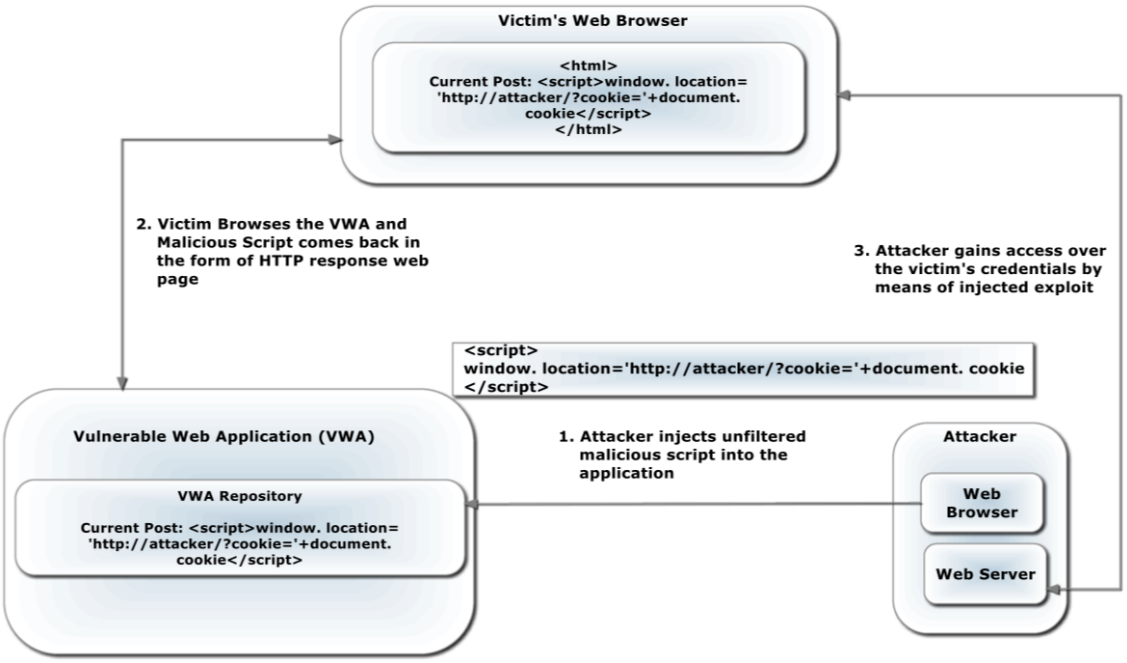

Abbildung 3.1: Der Ablauf einer Persistent XSS-Attacke (vgl. Gupta und Gupta, [19])

Formen als die schwerwiegendsten Exploits aktueller Webanwendungen. **Cookie-Stealing** wird meistens verwendet um die Sitzung des Nutzers zu übernehmen (*session hijacking*). Bei **Phishing Attacks** wird das DOM so manipuliert, dass das Opfer Daten in einem gefälschten Teil einer Webseite eingibt. In Folge werden diese Daten dann an den Angreifer gesendet. **Key Logging** bezeichnet den Vorgang bei dem die Tastatureingaben abgehört und gespeichert bzw. ausgewertet und auch an einen Server geschickt werden können.
Anschließend werden die folgenden 3 Unterkategorien von XSS genauer erläutert: nicht-persistent, persistent und DOM-basiert. In der Vergangenheit spielte auch eine weitere Form eine Rolle, deren Bedeutung aber stark abgenommen hat: Plugin-based XSS. Dabei machte sich der Angreifer Sicherheitslücken in Browser-Plugins für Flash oder Java zum Beispiel zu nutze. Da der Einsatz dieser Plugins unter anderem auf Grund vieler Sicherheitsprobleme stark zurück gegangen ist, wird dieser Unterart hier nicht mehr Bedeutung geschenkt.

3.2.1 Persistent Cross-Site Scripting

Persistent XSS wird oft auch als *Stored* XSS bezeichnet. Wie durch den Namen schon angedeutet, wird hier bösartiger Code eingeschleust und dauerhaft am Server gespeichert. Um das gut verständlich zu machen kann man sich folgendes Szenario vorstellen. Der Angreifer ist Mitglied eines Forums und postet einen neuen Eintrag, der seinen JavaScript-Code beinhaltet. Der Beitrag wird normal gespeichert und kann von anderen Benutzern angesehen werden. Wenn die Webseite nichts dagegen unternommen hat wird der Code vom Browser der Opfer ausgeführt. Opfer ist in diesem Fall jeder, der die Seite mit dem erstellten Beitrag öffnet. In diesem Szenario wird der Inhalt des zur Authentifizierung verwendeten Cookies an den Server des Angreifers übermittelt. Der Ablauf des Szenarios ist in Abbildung 3.1 veranschaulicht.

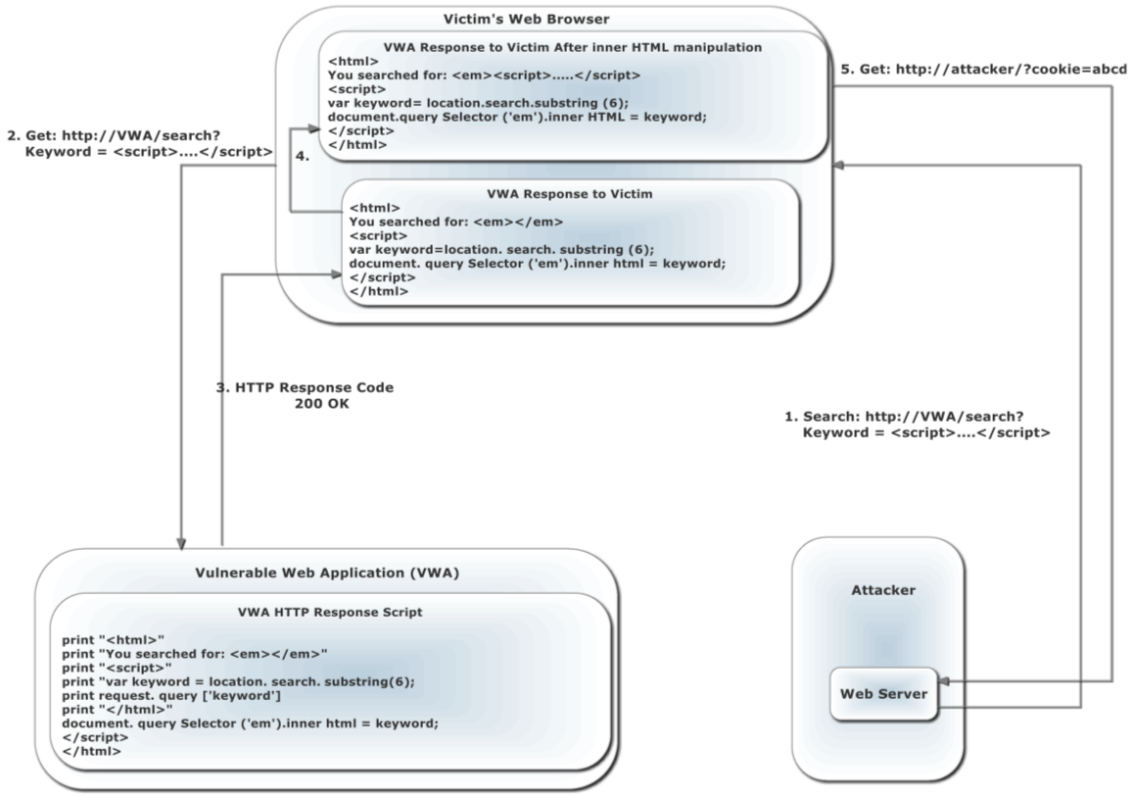

Abbildung 3.2: Der Ablauf einer DOM-basierten XSS-Attacke (vgl. Gupta und Gupta, [19])

3.2.2 Nicht-persistent Cross-Site Scripting

Nicht-persistent XSS wird auch als *Reflected*-XSS bezeichnet. Hier wird der schadhafte Code nicht am Server gespeichert. Um als Angreifer trotzdem das Selbe zu erreichen wie im obigen Szenario muss der Ablauf etwas anders aussehen. Der Angreifer erstellt einen Link, der den Code enthält und bringt das Opfer dazu den Link zu öffnen. Oft werden solche Links per E-Mail versendet und können sich oft auch weiterer bösartiger Techniken bedienen um das Vertrauen des Benutzers zu gewinnen. E-Mail-Spoofing und URL-Spoofing können verwendet werden um eine falsche Absenderadresse bzw. eine andere URL zu verschleiern. Nach dem Öffnen des Links durch den Benutzer generiert der Server eine HTTP-Antwort, die den Code des Angreifers beinhaltet. Beim Verarbeiten der Antwort durch den Browser wird der Code ausgeführt. Dieses Szenario ist nur ein Beispiel und kann natürlich auch anders aussehen. Der Code aus der Antwort wird nur ausgeführt, wenn er direkt als HTML-Text einem Element zugewiesen wird.

3.2.3 Document Object Model - basiertes Cross-Site Scripting

Diese Form unterscheidet sich grundlegend von den anderen beiden XSS-Kategorien darin, dass der JavaScript Code des Angreifers nicht in der Antwort des Servers mitgesendet wird. Somit kann man die DOM-basierten XSS-Angrffe als clientseitige und die oben genannten als serverseitige Schwachstellen klassifizieren. Auch hier wird eine URL verwendet um den bösartigen Code zu übergeben. Beim Aufruf des Links sendet der Server eine korrekte Antwort zurück. Anschließend wird der gutartige Code ausgeführt und greift aber auf den bösartigen Code zu, welcher somit ebenfalls ausgeführt wird.

Abbildung 3.2 zeigt den Ablauf anhand eines Beispiels. Der Angreifer erstellt einen Link, den das Opfer in seinem Browser aufruft. In diesem Link wird ein Suchparameter übergeben, dessen Wert

hier ein JavaScript-Code ist. Nach dem Aufruf des Links sendet der Server eine Antwort, welche den Schadcode nicht beinhaltet. Im Browser des Opfers wird der ursprüngliche Suchbegriff in die Antwort des Servers eingesetzt und somit der Code des Angreifers ausgeführt (vgl. Gupta und Gupta, [19]).

3.3 Hypertext Transfer Protocol Secure Interception

Die Verwendung von HTTPS statt HTTP ist unabdingbar im Bezug auf die Sicherheit. Dadurch wird die Vertraulichkeit und Integrität der Daten, sowie die Authentizität des Kommunikationspartners erreicht (vgl. Dierks und Allen, [13]). Auch wenn HTTPS immer öfter eingesetzt wird ist es keine Garantie, dass dieses Protokoll sicher eingesetzt wird. Netzwerk-Middleboxen und clientseitige Software wie zum Beispiel Anitvirensoftware können die TLS-Sitzung beenden, die zu sendenden Daten entschlüsseln und analysieren und danach eine neue Sitzung erstellen. Dieser Vorgang wird auch SSL-Inspection genannt. Die Software agiert in so einem Fall als Man-in-the-Middle, was im Design von HTTPS (bzw. TLS) versucht wurde zu verhindern. Um das Umgehen der Sicherheitsmechanismen trotzdem zu ermöglichen hinterlegt solch clientseitige Software ein selbstsigniertes CA-Zertifikat im Keystore des Browsers. Um das gleiche bei Middleboxen zu erreichen, wird das CA-Zertifikat der Middlebox auf den Client-Geräten im lokalen Netzwerk gespeichert. Bei dem Aufruf von Webseiten wird, nach dem Unterbrechen der sicheren Verbindung durch die erwähnte Software, ein Zertifikat für die Domain erstellt. Wenn der Benutzer die Zertifikatkette nicht manuell überprüft bleibt dieser Vorgang unbemerkt.
Die Auswirkungen von SSL-Inspection können sehr variieren. Dadurch, dass eine neue Sitzung aufgebaut wird, werden auch die Sitzungsparameter, also unter anderem die verwendeten kryptographischen Algorithmen, neu bestimmt. Das hat oft einen negativen Einfluss auf die Sicherheit, da meistens schwächere kryptografische Verfahren verwendet werden und so teilweise sogar große Sicherheitslücken entstehen (vgl. Durumeric u. a., [14]).

3.4 Browser Cache Poisoning

Bei dieser Attacke werden reguläre Webinhalte durch andere Ressourcen, die von dem Angreifer bereitgestellt werden, ersetzt. Inhalte des Angreifers sind typischerweise solche, die so lange wie möglich im Cache des Browsers gehalten werden. Man kann folgende 3 Arten von Browser Cache Poisoning (BCP)-Attacken unterscheiden, auf welche anschließend näher eingegangen wird. Alle gängigen Browser für Desktop und mobile Geräte sind anfällig für BCP-Angriffe. Diese Angriffe werden möglich durch Man-in-the-Middle-Attacken wobei der Angreifer ein eigenes Zertifikat benutzt um eine neue sichere Verbindung aufzubauen. Man-in-the-Middle-Attacken werden auf der Netzwerkebene ausgeführt.
Es wird hier davon ausgegangen, dass der Angreifer entweder ein selbst-signiertes Zertifikat oder eines mit nicht übereinstimmenden Domains verwendet. Daraufhin wird vor dem Öffnen der angefragten Ressourcen vom Browser eine Warnung über ein ungültiges Zertifikat ausgegeben. Das Problem dabei ist nun, dass der Großteil der Benutzer diese Warnung ignoriert und somit den Sicherheitsmechanismus des Browsers umgehen. Es gibt noch folgende Möglichkeit diesen Angriff durchzuführen ohne eine Warnung beim Browser auszulösen: der Angreifer erstellt ein gefälschtes Zertifikat für die Domain des Senders der Nachricht, welches durch eine kompromittierte CA signiert wird. In dieser Arbeit wird nicht näher auf dieses Szenario eingegangen da dagegen als Anwendungsbetreiber beziehungsweise Anwendungsentwickler keine sinnvollen Maßnahmen getroffen werden können. Abgesehen davon gibt es noch die Möglichkeit, dass HTTP zur Übertragung verwendet wird. In diesem Fall wird, wenn kein Zertifikat verwendet wird, keine Warnung ausgegeben.
Mit Hilfe des HTML 5 Anwendungscache kann eine komplette Seite für einen langen Zeitraum

(Monate oder auch mehr als ein Jahr) zwischengespeichert werden. Die ersetzten Ressourcen können sowohl solche von dem selben Webserver als auch externe Dateien sein. Oft werden externe Ressourcen von Content Delivery Network (CDN)s geladen. Im Gegensatz zu cross-origin werden bei same-origin in diesem Kontext nur die Ressourcen der Zielseite ausgetauscht. Mit Zielseite wird die Seite bezeichnet, auf die der Angriff abzielt. Sie wird im Folgenden auch A genannt. Beim cross-origin Ansatz wird anfänglich nicht die Zielseite A besucht. Die zuerst besuchte Seite B hat keine Abhängigkeit zur Seite A. Der Angreifer ändert die Ressourcen von Seite B so ab, dass eine Abhängigkeit zu Ressourcen von Seite A hergestellt wird. Sobald diese Ressourcen geladen werden, werden auch sie vom Angreifer ausgetauscht. Zur weiteren Veranschaulichung dient Abbildung 3.3.

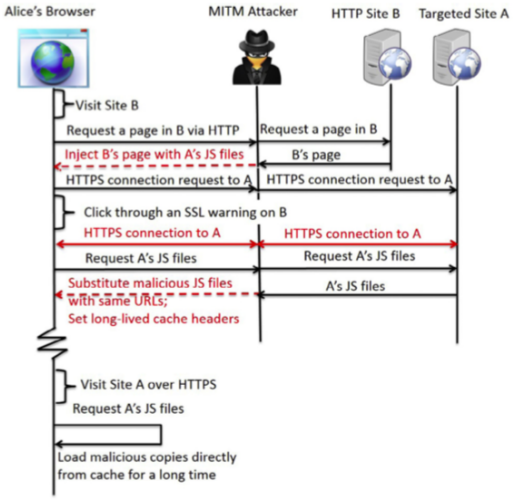

Abbildung 3.3: Der Ablauf einer Cross-Origin-BCP-Attacke (vgl. Yaoqi u. a., [74])

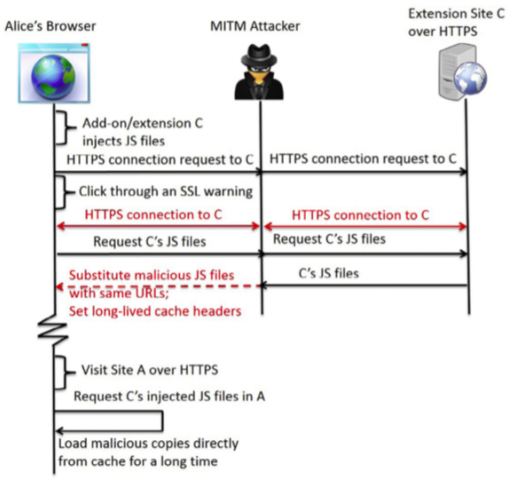

Abbildung 3.4: Der Ablauf einer Cross-Origin-BCP-Attacke unter Verwendung von Browsererweiterungen (vgl. Yaoqi u. a., [74])

Die Kategorie *Extension-assisted* zeichnet sich dadurch aus, dass hier Browsererweiterungen verwendet werden. Viele Browsererweiterungen laden zusätzliche Ressourcen von einem Server und binden sie in die besuchten Seiten ein. Auch hier werden die Ressourcen vom Angreifer abgefangen, verändert oder ersetzt und an das Opfer weitergeleitet. Auch in diesem Fall zeigt der Browser eine Warnung wegen einem ungültigen Zertifikat an. Die kompromittierte Ressource kann in Folge von der Browsererweiterung auf potentiell jeder Seite wiederverwendet werden. Wir sind bis jetzt davon ausgegangen, dass die Browsererweiterungen die weiteren Ressourcen per HTTPS nachladen, doch das ist nicht notwendigerweise der Fall. Wenn Ressourcen mit HTTP auf einer Seite verwendet werden die selbst aber HTTPS verwendet, sollte der Browser das Laden der Ressource blockieren und eine Warnung ausgeben. Jedoch handhaben das nicht alle Browser gleich. Auch dieser Angriff, auf eine durch HTTPS abgesicherte Seite, ist auf Abbildung 3.4 dargestellt Yaoqi u. a., [74].

3.5 Denial of Service

Den Namen dieses Angriffs (abgekürzt DoS) kann man mit Zugriffsverweigerung übersetzen. Das Ziel ist es das Opfer, das einen gewissen Service anbietet, mit Anfragen zu überlasten, sodass reguläre Anfragen nur noch eingeschränkt oder gar nicht beantwortet werden können. Die bösartigen Anfragen können auch von mehreren Computern kommen. In diesem Fall wird dann von einer Distributed Denial of Service (DDoS) Attacke gesprochen. Üblicherweise haben die Hosts,

von denen die bösartigen Anfragen bei einer DDoS Attacke kommen, keine Kenntnis von ihrer Teilnahme an der Attacke, da sie meistens unbemerkt vom Angreifer kontrolliert werden. Der Angreifer kann sich so ein Netzwerk aus vielen Hosts aufbauen, welches Botnet genannt wird. Mittlerweile existieren genügend Tools die es auch einem amateurhaften Angreifer leicht machen solche Angriffe auszuführen. Im Allgemeinen ist das Ziel dem Opfer so finanziell zu schaden. Sich dagegen zu schützen ist schwierig, da die Anfragen nicht einfach von regulären Anfragen unterschieden werden können (vgl. Kirrage, Rathnayake und Thielecke, [31]).
Es gibt viele verschieden Varianten dieses Angriffs, welche sich oft sehr durch ihre Effektivität unterscheiden. Eine Variante auf die hier näher eingegangen wird, ist Regular Expression Denial of Service (ReDoS). Reguläre Ausdrücke (engl. *regular expression*) werden verwendet um zu prüfen ob ein String einem gewissen Muster entspricht. Die meisten Programmiersprachen wandeln diese Ausdrücke in Nondeterministic Finite Automaton (NFA)s um. Daraufhin suchen sie in dem zu prüfenden String nach einer Sequenz, die zu einem Endzustand führt. Ein NFA ist ein Automat, bei dem es in mindestens einem Zustand mehrere Übergangsmöglichkeiten gibt. Diese Tatsache kann in Extremfällen bei der Auswertung eines regulären Ausdrucks zu einer exponentiellen Laufzeit führen. Bei dieser Überprüfung werden verschiedene Reihenfolgen von Übergängen ausprobiert. Das passiert so lange, bis entweder eine Übereinstimmung gefunden wurde oder alle möglichen Reihenfolgen ausprobiert wurden, der zu überprüfende Text aber nicht mit dem regulären Ausdruck übereinstimmt. Hier wird vom Rücksetzverfahren (engl. *Backtracking*) Gebrauch gemacht - sobald der Algorithmus bemerkt, dass die probierte Reihenfolge von Übergängen nicht übereinstimmt, geht der Algorithmus so lange zurück bis eine andere Abzweigung genommen werden kann, die zu einer noch nicht probierten Reihenfolge führt. Dieser Ansatz ist im Allgemeinen effizient, allerdings nur meistens. Die Verwendung von Deterministic Finite Automaton (DFA)s hätte eine stabile Laufzeit, jedoch ist die Erstellung eines solchen Automaten viel aufwändiger, da er weitaus komplexer ist. Ein weiterer Grund ist, dass das Rücksetzverfahren mächtiger ist, im Sinne der akzeptierten Regulären Ausdrücke. Somit werden unter anderem Rückverweise möglich. Abbildung 3.5 zeigt einen Automaten für den ein Backtracking Algorithmus im schlimmsten Fall exponentielle Laufzeit benötigt. Die Laufzeit is exponentiell im Verhältnis zur Eingabegröße, also der Länge des Eingabetexts. Abbildung 3.6 generiert Eingabetexte, die eine exponentielle Laufzeit verursachen. Dadurch, dass die von ihm generierten Texte am Ende immer ein 'b' haben, werden sie vom linken Automaten nicht akzeptiert. Das bedeutet, dass der Matching-Algorithmus alle Zustandsfolgen ausprobiert die mit der gegebenen Eingabegröße möglich sind (vgl. Wüstholz u. a., [73]).

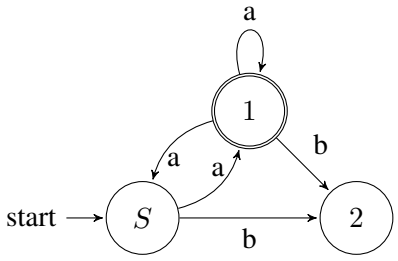

Abbildung 3.5: Beispiel eines anfälligen Automaten

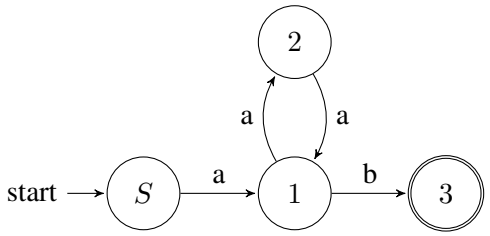

Abbildung 3.6: Automat für bösartigen Input für Abbildung 3.5

3.6 Structured Query Language Injection Attacke

Als Nummer eins in der OWASP TOP 10 Liste von 2017 (OWASP-Foundation, [55]), der häufigsten Sicherheitslücken in Webanwendungen, tritt diese Sicherheitslücke noch immer häufig auf. Erstaunlich ist das, weil sie knapp vor der Jahrtausendwende bereits das erste Mal erwähnt wurde.

Der Ablauf bei dieser Attacke ist folgender: Der serverseitige Code einer Anwendung macht eine Anfrage an die Datenbank und bindet dabei Eingaben des Benutzers mit ein. Für dieses Verhalten gibt es unzählige Beispiele - konkret sind das unter anderem ein Abgleich der Zugangsdaten eines Benutzers, bei dem das bereitgestellte Passwort mit dem Passwort verglichen wird, welches dem anfragenden Benutzer zugeordnet ist. In diesem Beispiel ist somit der Benutzername die verwendete Eingabe. Ein weiterer Anwendungsfall ist eine Produktsuche in einem Online-Shop, bei der nach verschiedenen Parametern der Produkte gefiltert werden kann (vgl. Deepa und Thilagam, [12]). Die vom Server verarbeiteten Daten müssen allerdings nicht zwingendermaßen Eingaben des Benutzers sein, sondern können auch Daten sein welche von der Anwendung selbst erstellt wurden wie zum Beispiel Cookies oder Variablen im Code am Server (vgl. Halfond, Viegas und Orso, [24]). Der im Falle eines Angriffs typische Teil ist die Verwendung von Eingaben durch den Benutzer, die am System ein, von den Betreibern der Anwendung, nicht beabsichtigtes Verhalten verursachen. Das bedeutet, dass der Angreifer das Verhalten der Datenbankabfrage verändert.

3.6.1 Ziele von Angreifern

Die wichtigsten Ziele Structured Query Language Injection Attacke (SQLIA)n sind folgende: Herausfinden welche Parameter einer Datenbankabfrage ausgenutzt werden können, *Database fingerprinting* - Identifizieren des Datenbanktyps (MySQL, SQL Server, Oracle SQL, MongoDB etc.) und der Version, herausfinden des Datenbankschemas - dazu zählen unter anderem Namen von Tabellen, Tabellenspalten und gespeicherten Prozeduren. Weitere Ziele sind das Auslesen, Hinzufügen oder Ändern von Daten sowie die Ausweitung der Rechte des die Abfrage ausführenden Benutzers.
Die Gründe warum die Häufigkeit der Schwachstelle trotz der Bekanntheit so hoch ist, ist einerseits die umfangreiche Verwendung von Benutzereingaben bei Abfragen von benutzerbezogenen Daten. Aus der häufigen Verwendung ergibt sich auch eine höhere Angriffsfläche für mögliche Angreifer. Ein weiterer Grund ist der in der Softwareindustrie, um die Jahrtausendwende sowie heute, verbreitete Versuch Kosten zu sparen, indem wenig Zeit in sicherheitsrelevante Bereiche von Projekten investiert wird. Verantwortlich für diese Lücke sind Projektverantwortliche, Tester und Entwickler der Softwareanwendung, die Daten entgegennimmt und sie in Abfragen miteinbezieht. Sie haben ihren Ursprung in unzureichender Validierung von Benutzereingaben. Diese Lücke ist somit nicht zurückzuführen auf Fehler in der Software der Datenbank, schlechte Eigenschaften oder Einstellungen von Datenübertragungen oder Ähnlichem (vgl. Deepa und Thilagam, [12]).

3.6.2 Second-Order Structured Query Language Injection Attack

Bei dieser Angriffsart werden die bösartigen Benutzereingaben in der Datenbank gespeichert und erst zu einem späteren Zeitpunkt verwendet - meistens werden die Eingaben jedoch sofort verwendet. Ein Beispiel für eine Second-Order SQLIA ist folgendes Szenario: Ein Angreifer legt einen neuen Benutzer mit dem Benutzernamen ”admin’--”. Beim Einfügen des neuen Benutzers wird der Benutzername vom Programm richtig verarbeitet und ”admin’--” gespeichert. In weiterer Folge möchte der Angreifer seinen Benutzernamen ändern, wofür vom Programm die in Listing 3.1 gezeigte Datenbankabfrage verwendet wird. Der Codeausschnit zeigt wie die Datenbankabfrage aussieht, wenn der Angreifer sein Passwort von ’oldPassword’ auf ’newPassword’ aktualisiert.

```
1 updateQuery = "update user set password = '" + newPassword + "'
  ↪ where username = '" + username + "' and password = '" +
  ↪ oldPassword + "';"
```

Listing 3.1: SQL-Abfrage zum Aktualisieren des Benutzerpassworts

Listing 3.2 veranschaulicht wie die Datenbankabfrage mit bösartigen Benutzereingaben aussehen kann.

```
1 updateQuery = "update user set password = 'newPassword' where
  ↪ username = 'admin'--' and password = 'oldPassword';"
```

Listing 3.2: SQL-Abfrage zum Aktualisieren des Benutzerpassworts mit bösartigen Benutzereingaben

Dieses Szenario geht davon aus, dass die Eingaben vor der Update-Anfrage nicht validiert werden. Da die Daten in diesem Fall allerdings bereits in der Datenbank sind, ist es eine realistische Annahme, dass keine erneute Validierung stattfindet. Da '–' die Zeichenfolge für einen einzeiligen Kommentar ist, wird in der obigen Anfrage alles ab dieser Zeichenfolge ignoriert. Somit wird in diesem Fall das Passwort des Benutzers 'admin' auf 'newPassword' gesetzt.

3.6.3 First-Order Structured Query Language Injection Attack

Folgende sind die als am Wichtigsten erachteten Arten von First-Order SQLIAs: Tautology attacks, Piggypacked queries, Illegal/Logically Incorrect Queries, Blind Injection attacks, Alternate Encodings. In diesem Absatz wird auf einige genannte Arten näher eingegangen.

Tautology Attacks sorgen dafür, dass eine Wahrheitsbedingung immer *true* ergibt. Das bekannteste Beispiel hierfür ist eine Abfrage die bei einem Login-Prozess verwendet wird um zu prüfen ob es einen Benutzer mit der bereitgestellten Kombination aus Benutzername und Passwort gibt. Eine solche Abfrage ist bei Listing 3.3 gezeigt.

```
1 getUserQuery = "select * from users where username = '" +
  ↪ userNameInput + "' and password = '" + passwordInput + "';"
```

Listing 3.3: Abfrage für den Abgleich von Benutzername und Passwort im Zuge eines Login-Mechanismus beispielsweise

Es wird davon ausgegangen, dass die Benutzereingaben genau so weiterverwendet werden, wie sie eingegeben wurden. Bei einer falschen Kombination wird das Ergebnis keinen Datensatz beinhalten. Bei einer richtigen Kombination wird das Ergebnis genau einen Datensatz enthalten (unter der Annahme, dass der gleiche Benutzername nicht mehrfach vorkommt). Weiters wird angenommen, dass die Applikation, im Falle eines nicht leeren Ergebnisses, die erste Zeile des Ergebnisses abspeichert. Für die korrekte Funktionsweise ist eine solche Implementierung im Normalfall ausreichend. Ein Angreifer könnte diese Umsetzung ausnutzen indem er als Benutzernamen "maxMusteründ als Passwort *meinPasswort' OR 1 = 1* eingibt. Die Abfrage die in Folge an die Datenbank geschickt wird ist bei Listing 3.4 gezeigt.

```
1 getUserQuery = "select * from users where username = 'maxMuster' and
  ↪  password = 'meinPasswort' OR 1 = 1;"
```

Listing 3.4: Abfrage für den Abgleich von Benutzername und Passwort im Zuge eines Login-Mechanismus beispielsweise

Da der Vergleich *1 = 1* immer wahr ergibt, ist es komplett egal was als Benutzername und Passwort eingegeben wird, da die komplette *Where*-Klausel eine Tautologie ist. In diesem Fall gibt die Datenbankabfrage als Ergebnis alle Benutzer zurück und der Benutzer wird als der Benutzer des ersten Datensatzes aus dem Ergebnis eingeloggt. Illegal/Logically Incorrect Queries werden vor allem verwendet um mehr über die Datenbank selbst und das Schema zu erfahren. Verwendet werden dabei Ausdrücke die Datenbankfehler verschiedenster Arten bei der Ausführung der Abfrage durch die Datenbank erzeugen. Halfond, Viegas und Orso, [24] gibt ein in Listing 3.5 dargestelltes Beispiel.

```
1 SELECT accounts FROM users WHERE login="" AND pass="" AND pin=
  ↪ convert(int,(select top 1 name from sysobjects where xtype="u"))
```

Listing 3.5: Datenbankabfrage für einen Authentifizierungsprozess

Ähnlich wie bei der vorigen Abfrage geht es hier um einen Authentifizierungsmechanismus für den drei Parameter übergeben werden. In diesem Fall hat der Angreifer die ersten beiden Parameter leer gelassen und für den Parameter, der mit der Spalte 'pin' verglichen wird, ein Statement angegeben, welches eine Typumwandlung versucht. *sysobjects* enthält in einer SQL Server Datenbank einen Eintrag für jedes Objekt in der Datenbank. Die *where*-Klausel filtert die Objekte nach Tabellen die von Benutzern erstellt wurden und nicht vom Datenbanksystem selbst. Es wird also versucht den Namen der ersten der herausgefilterten Tabellen in einen *int* umzuwandeln. Die Datenbank liefert in diesem Fall den Fehler "Microsoft OLE DB Provider for SQL Server (0x80040E07) Error converting nvarchar value 'CreditCards' to a column of data type int". Für einen Angreifer liefert diese Fehlermeldung, wenn sie in der Applikation angezeigt wird wertvolle Informationen: die Bestätigung, dass es sich um eine Microsoft SQL Server Datenbank handelt und, dass es eine Tabelle "CreditCards"gibt (vgl. Halfond, Viegas und Orso, [24]).

3.7 Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) ist eine Attacke bei der eine bösartige Webanwendung im Namen des Benutzers ungewollte Anfragen bei einer anderen Webanwendung tätigt. Dabei macht sich die Attacke die Tatsache zu nutze, dass der Benutzer auf der anderen Webanwendung eingeloggt ist. Ein passendes Beispiel für dieses Szenario ist ein Benutzer der sich auf einer Webseite eines Online-Shops einloggt. Im gleichen Browser öffnet er ebenso die bösartige Webseite. Diese Webseite sendet ohne eine weitere Interaktion des Benutzers eine Anfrage an den Server des Online-Shops. Auch wenn in diesem Fall die Anfrage nicht von der Logik des Online-Shops kommt, sendet der Browser Cookies die zur ZielDomäne gehören mit der Anfrage mit. Darunter befinden sich meistens Daten zur Authentifizierung, weshalb die Anfrage dem Webserver legitim erscheint. Eine solche Anfrage kann, abhängig von den angewendeten Sicherheitsmechanismen, beispielsweise Kontodaten verändern, Bestellungen tätigen oder den Account selbst löschen. Diese Attacke ist ein Risiko für alle Applikationen bei welchen der Benutzer Aktionen vornehmen kann welche nur im eingeloggten Zustand möglich sein sollten. Aktuelle Online-Banking-Anwendungen und Online-Shops schützen sich daher im Normalfall bereits vor diesem Angriff. Unter Entwicklern ist diese Attacke aber bei weitem weniger bekannt als XSS oder SQLIA (vgl. OWASP-Foundation, [52]).

3.8 Cross-origin Angriff

Durch die immer weiter zunehmende Vernetzung von Internetapplikation zueinander, steigt die Notwendigkeit zwischen den Applikationen Daten auszutauschen. Nun mag es überraschend sein, dass das mit Einschränkungen verbunden ist und es einen grundlegenden Unterschied macht, ob

Daten von einer Informationsquelle benötigt werden, die zur Applikation gehören oder nicht. Außerdem ist es wichtig zu unterscheiden, ob es ausreichend ist Daten an den externen Kommunikationspartner zu senden, oder ob es notwendig ist, Daten von dem Kommunikationspartner zu verarbeiten. Festgelegt werden diese Restriktionen durch die Same-Origin Policy (SOP). Der Begriff bezeichnet keine eigene Spezifikation oder Ähnliches sondern lediglich ein Browser-Feature, das die Kommunikation mit externen Komponenten einschränkt. Das Feature steht im Zusammenhang mit Cross-Origin Resource Sharing (CORS) (Kesteren, [30]) worunter die oben beschriebene Kommunikation unter verschiedenen Applikationen verstanden wird. In Verbindung mit einer CSRF-Sicherheitslücke ist für einen Angriff in den meisten Fällen auch ein weitere Sicherheitslücke in der Applikation notwendig. Im Laufe des Kapitels wird näher auf mögliche Szenarien eingegangen. CORS ist aus dem Problem entstanden, dass die Einschränkungen der SOP von Softwareentwicklern oft umgangen werden, um die Projektanforderungen umzusetzen. Das führt in weiterer Folge in vielen Fällen zu einer Verminderung der Sicherheit der entwickelten Software. Beispielsweise führen die nicht optimalen Umsetzungen oft zu vorher nicht vorhandenen CSRF- und Man-in-the-Middle-Schwachstellen oder der Möglichkeit von Session-Hijacking. Unter dem Begriff *Cross-Origin Injection* versteht man das Einschleusen von Ressourcen in eine Applikation für einen bösartigen Zweck. CORS bietet die Möglichkeit einer saubereren Umsetzung der Kommmunikationsanforderungen. Doch auch die Verwendung von CORS führt durch Fehlkonfigurationen oft zu verschiedensten Sicherheitslücken. SOP verbietet zwar das Lesen von Antworten auf externe Anfragen, allerdings nicht die externen Anfragen selbst. POST-Anfragen (siehe Kapitel 2.1.1) können dabei verwendet werden um unter anderem CSRF-Angriffe auszuführen.
Oft ist es notwendig Daten von einer Adresse abzufragen, die sich von der Adresse der Applikation unterscheidet. Als Beispiel für die Verwendung einer externen Anfrage sei eine Schnittstelle genannt, die eine Berechnungsfunktionalität anbietet. Aber auch externe Skripte wie bei *jQuery* (siehe Kapitel 2.4) oft der Fall, gehören zu dieser Kategorie. Die Anforderung ist, dass mit JavaScript eine Anfrage an die externe Schnittstelle geschickt wird und die Antwort von JavaScript verarbeitet werden kann. Um das zu ermöglichen wurde JSON-P entwickelt, welches die Restriktionen von SOP umgeht. Die Verwendung von JSON-P verursacht wiederum einige Probleme beziehungsweise schränkt die Weiterverarbeitung ebenfalls ein. JSON-P kann nur mit GET-Anfragen (siehe Kapitel 2.1.1) verwendet werden und nicht mit POST-Anfragen. Außerdem ist es nicht möglich die Antwort vor der Weiterverarbeitung zu validieren, da die Antwort als JavaScript interpretiert und sofort ausgeführt wird.

Die Funktionsweise von CORS ist folgendermaßen: Beim Senden einer Anfrage wird der HTTP-Header *origin* mitgesendet. Der Wert dieses Headers ist die Uniform Resource Identifier (URI) der anfragenden Applikation. Die Applikation, die die Anfrage bearbeitet gibt in der Antwortnachricht den *Access-Control-Allow-Origin*-Header an. Der Wert dieses Headers muss dem des *origin*-Headers von zuvor entsprechen. Anschließend vergleicht der, die Antwort verarbeitende Browser, die beiden Header und erlaubt bei einer positiven Überprüfung der Applikation das Lesen der Antwort. Der Wert des *Access-Control-Allow-Origin*-Headers kann auch Platzhalter (engl. *wildcard*) enthalten um sich nicht auf fixe URIs beschränken zu müssen. Bei CORS gibt es bezüglich der Anfrage folgende Unterscheidung: Einfache und nicht-einfache Anfragen. Einfachen Anfragen wird vom Browser vertraut und in Folge sofort abgeschickt. Nicht-Einfache Anfragen benötigen einige zusätzliche Angaben, auf die hier nicht näher eingegangen wird. Die Schwachstellen können in drei Kategorien eingeteilt werden. Die erste Kategorie ist *Unvollständiger Referenzmonitor* worunter in diesem Zusammenhang man das großzügige Zulassen von Anfragen versteht. GET- und POST-Anfragen sind bei einfachen Anfragen standardmäßig erlaubt. Außerdem darf die Anfrage neun vorgegebene Header mitsenden. Ein Problem dabei ist, dass CORS bei keinem Header bis auf *Content Type* Einschränkungen bezüglich des Formats macht. Die Werte des *Accept*-Headers beispielsweise sollten exemplarisch wie folgt aussehen: 'text/html,application/xml'.

Diese Einschränkung wird von Browsern allerdings nicht überprüft. Ein weiteres Beispiel ist die fehlende Überprüfung von Header-Größen. Mit Seitenkanalangriffen (siehe Kapitel 3.9) können Informationen über Cookies beschafft werden. Die zweite Kategorie ist *Riskante Vertrauensabhängigkeiten*. CORS ermöglicht Anwendungen die Nutzung eines Autorisierungskanals mit Domänen welchen vertraut wird. Dieser Kanal lockert die Einschränkungen der SOP und führt somit dazu, dass die vertrauende Domain angreifbar wird. In sehr vielen Fällen sind die Domains denen vertraut wird Subdomains. Schwachstellen in Subdomains können dazu führen, dass die eigentliche Zielapplikation von der Subdomain aus angegriffen wird. Eine andere Möglichkeit ist, dass die HTTPS-Seite der eigenen HTTP-Seite vertraut. Die Seite von Fedex[1] (unter den 500 am meist besuchten Internetseiten laut Alexa[2]) ist ein Beispiel dafür. Die letzte Kategorie ist Fehlkonfigurationen. Ca. 10% der, mit CORS konfigurierten Webseiten, vertrauen unbeabsichtigt Seiten, die potenziell von einem Angreifer kontrolliert werden können. Die Gründe dafür sind folgende: Die Antwortheader werden oft dynamisch zur Laufzeit und somit von der Anwendungslogik selbst angepasst, da die Möglichkeiten, von CORS nicht flexibel genug sind. Außerdem werden die Sicherheitsmechanismen von Entwicklern oft nicht richtig verstanden. Ein weiterer Grund ist, dass die Verwendung gemeinsam mit einem geteilten Cache (zum Beispiel ein Proxyserver) nicht korrekt funktioniert. Der Grund ist, dass die CORS-Header der Nachrichten in der Logik der Caches meist nicht berücksichtigt werden (vgl. Chen u. a., [10]).

3.9 Seitenkanalangriff

Der Begriff *Seitenkanalangriff* ist eine sinngemäße Übersetzung aus dem Englischen (*side-channel attack* - erste Erwähnung von Kocher, [32]). Bei diesem Angriff werden Unterschiede in der Performance, dem Stromverbrauch, der Geräuschentwicklung, oder ähnliche Merkmale ausgenutzt um unautorisiert an Informationen zu gelangen. Oft gehen sie aus Performanceoptimierungen hervor. Angriffe dieser Art gibt es auf verschiedensten Systemen. Sie werden oft auch als Mikroarchitekturelle Attacken (engl. *Mircoarchitectural*) bezeichnet, da sie auf kleine Bestandteile eines Systems abzielt wie beispielsweise Caches. Ein primitives Beispiel ist, dass ein Angreifer bei einer Login-Seite einer Webanwendung herausfinden möchte ob ein Benutzername am System existiert oder nicht. Wenn die Anwendung das Passwort nur prüft falls der Benutzer existiert, kann das zu einem Preformanceunterschied führen, den der Angreifer bemerken und sich zu Nutze machen kann. Im Folgenden wird vor allem auf Angriffe eingegangen welche versuchen auf gespeicherte Daten unauthorisiert zuzugreifen. Bei Seitenkanalangriffen ist das Ziel in erster Linie, unabhängig von der Programmiersprache, meistens Speicheradressen herauszufinden. JavaScript verwendet intern zwar virtuelle Adressen, doch bekommt der Programmierer beziehungsweise das ausführende Programm nichts davon mit. Das bedeutet, dass es in der Sprache keine bereitgestellten Funktionalität gibt um die verwendeten Adressen zu erhalten. Jedoch können ArrayBuffer verwendet werden um die verwendeten Adressen zu rekonstruieren. ArrayBuffer verhalten sich grundsätzlich gleich wie normale Arrays, sind jedoch schneller, da sie eine fixe Größe haben. Normale Arrays haben in JavaScript im Gegensatz zu Java beispielsweise keine fixe Größe.
Für Mikroarchitekturelle Attacken werden die physischen Adressen benötigt, welche Anwendungen eines Betriebssystems nicht offenbart werden. Um diese Adressen trotzdem zu erfahren wird ausgenutzt, wie Browser ArrayBuffer verwenden. Daten werden zwischen Speichern in Seiten übertragen und im Speicher dann auch so abgelegt. Das erste Element eines ArrayBuffers befindet sich immer am Anfang einer Seite. Solche Seiten sind im grundsätzlich 4 Kilobyte (KB) groß. Für das Übertragen großer Datenmengen werden aber auch 2 Megabyte (MB) große Seiten verwendet, welche als Transparent Huge Page (THP)s bezeichnet werden. Seiten werden geladen

[1] https://www.fedex.com/
[2] https://www.alexa.com/siteinfo/fedex.com

sobald Inhalt auf ihnen benötigt wird. Die unterschiedlichen Größen führen zu unterschiedlichen Ladezeiten. Die physikalischen Adressen von 4 KB und 2 MB großen Seiten haben am Ende 15 beziehungsweise 21 Bits die auf '0' gesetzt sind. Beim Iterieren über ein Array kann also bemerkt werden, wann eine neue Seite geladen wird und ob sie eine Seite normaler Größe oder eine THP ist. Bei den meisten Attacken ist ebenfalls eine möglichst genaue Möglichkeit der Zeitmessung notwendig. Je nach Anwendungsfall wird eine Genauigkeit im Millisekunden oder Nanosekunden gebraucht. Die *Performance* Schnittstelle stellt eine Funktion *now* zur Verfügung welche eine Genauigkeit im Mikrosekundenbereich ermöglicht. Falls eine höhere Genauigkeit gebraucht wird, müssen eigene Lösungen verwendet werden. Für diesen Fall gibt es beispielsweise folgende Möglichkeiten: Verwendung von Zählschleifen oder Schnittstellen von Multimediainhalten. Üblicherweise wird keine absolute Zeit benötigt, da nur Zeitspannen berechnet werden müssen.
Mit der Einführung von dem Thread-Konzept in JavaScript (*web worker*) ergaben sich auch neue Möglichkeiten JavaScript-Code auszunutzen. Die Synchronisierung zwischen verschiedenen Threads funktioniert durch Nachrichtenaustausche. Auch wenn die Anwendung so programmiert wurde, dass der Code synchron ausgeführt wird, verwendet der Browser selbst trotzdem viele verschiedene Threads. Es gibt den zentralen Hostprozess welcher mit Hilfe von verschiedenen Threads unter anderem Netzwerk- und lokale Dateizugriffe sowie das Weiterleiten von User Events an Renderingprozesse. Es gibt somit auch verschiedene Renderingprozesse die für das Rendern von Webseiten zuständig sind, die selbst wieder mehrere Threads ausführen. Jeder dieser Prozesse läuft zwar in einer eigenen Sandbox, jedoch kann es sein, dass sich mehrere Webseiten einen Prozess teilen. Die Rendererprozesse müssen mit dem Hostprozess kommunizieren. Beispielsweise werden Netzwerkanfragen vom Hostprozess durchgeführt und auch durch den Benutzer ausgelöste Events, werden vom Hostprozess registriert und an den betroffenen Prozess weitergeleitet. Der Nachrichtenaustausch zwischen den verschiedenen Prozessen beziehungsweise Threads führt zu Events welche in den jeweiligen Threads in First in – First out (FIFO)-Warteschlangen landen und der Reihe nach abgearbeitet werden. Ein Angreifer kann herausfinden wann auf einem Thread Events in der Warteschlange landen. Das kann bewerkstelligt werden in dem der Thread in kurzen, regelmäßigen Abständen Nachrichten an sich selbst schickt. Diese Nachrichten werden an den Host-Prozess weitergeleitet und danach an den ursprünglichen Thread selbst geschickt. Größere Zeitunterschiede treten auf, wenn der Host-Prozess davor noch ein anderes Event verarbeiten muss. Diese Zeitunterschiede können verwendet werden um Webseiten zu identifizieren oder um Keylogger zu implementieren. Außerdem können so zwischen 2 Webseiten auf verschiedenen Tabs im privaten Modus Nachrichten ausgetauscht werden, was dem Sandbox-Prinzip des Browsers widerspricht.
Die letzte hier angeführte Form von Attacken sind solche, die die Sensor API missbrauchen. Diese API wird von Browsern auf mobilen Endgeräten verwendet und bietet Daten über Lichtverhältnisse und Geräusche in der Umgebung des Geräts, die aktuelle Position und Bewegungen des Geräts und viele weitere. Aus dem Standpunkt der Sicherheit werden allerdings nicht alle Sensorschnittstellen gleich behandelt, da unter anderem für Zugriffe auf Global Positioning System (GPS)-Daten die Zustimmung des Benutzers erforderlich ist (vgl. Schwarz, Lipp und Gruss, [60]). Im Gegensatz dazu ist das bei Schnittstellen, die Daten über die Bewegung und Orientierung (*motion and orientation* - Spezifikation[3]) des Geräts liefern nicht der Fall. Das Gefährliche bei den Sensordaten ist, dass sie in vielen Fällen verfügbar sind, obwohl sie aus dem Standpunkt der Sicherheit nicht sein sollten. Ein Beispiel dafür ist eine Webseite die eine andere Webseite mit einem HTML Frame (*iframe*-Element) einbindet. In so einem Fall kann die eingebettete Seite in allen gängigen Browsern auf die zuvor erwähnten Sensordaten zugreifen. Je nach Browser ist es auch möglich, dass selbst Zugriff auf die Daten besteht, wenn ein anderer Browsertab aktiv ist oder gerade eine andere Anwendung aktiv ist. In manchen Browsern (UC Browser, Baidu, Maxthon, Boat, Next -

[3] https://w3c.github.io/deviceorientation/

vgl. Mehrnezhad u. a., [42, Table 4]) ist das sogar der Fall wenn das Gerät gesperrt ist. Mit den Bewegungs- und Orientierungsdaten ist es bei Geräten mit einem Touch-Display möglich Aktionen wie Klicks, Scrollen und Zoomen zu unterscheiden und ebenso ihre Positionen auf dem Gerät festzustellen. Das wird mit Algorithmes bewerkstelligt die auf maschinellem Lernen aufbauen. In weiterer Folge ermöglicht das die Feststellung der Art der verwendeten Anwendung sowie sogar beispielsweise die Erkennung des PINs zur Entsperrung des Geräts (vgl. Mehrnezhad u. a., [42]).

3.10 Unsichere Deserialisierung

Diese Art der Schwachstelle ist eine weitere aus der Liste der häufigsten 10 Schwachstellen (OWASP-Foundation, [55]). Deserialisierung kommt vor allem beim Datenaustausch zwischen bei Webseiten zwischen dem Browser und dem Server der Anwendung. Ein Beispiel ist eine Buchhaltungswebseite bei der Rechnungen erstellt werden können. Die neue Rechnung wird beim Speichern in einem bestimmten Format an den Server geschickt. Meistens werden die Daten im JavaScript Object Notation (JSON)-Format übertragen, ein Ausnutzen der Deserialisierung ist aber bei verschiedensten Formaten möglich. Eingegangen wird hier allerdings nur auf die Verwendung des JSON-Formats, da es das gängigste Übertragungsformat ist. Der allgemeine Ablauf der Deserialisierung ist folgendermaßen: das Objekt wird im Browser in das zu übertragende Format gebracht - dieser Vorgang wird Serialisierung genannt - und an den Server gesendet. Anschließend wandelt die Anwendung es für die weitere Verarbeitung in ein Objekt der verwendeten Programmiersprache um - dieser Vorgang ist die Deserialisierung. Die Schwachstelle geht aus dem Mechanismus hervor der verwendet wird, um Prinzipien der objektorientierten Programmierung, wie zum Beispiel Polymorphismus, zu unterstützen. Die Bibliotheken, wie beispielsweise *Jackson* für Java oder *Json.Net* für .NET, die das Serialisieren beziehungsweise Deserialisieren übernehmen, erstellen bei der Deserialisierung ein Objekt der erwarteten Klasse und verwenden dann *Setter*, um die entsprechenden Werte des Objekts zu setzen. Je nach der Konfiguration der Bibliotheken sind die Typinformationen in der serialisierten Datei enthalten. Somit ist es auch möglich diese Typinformation zu modifizieren, was dazu führt, dass ein Objekt einer anderen Klasse erstellt wird. Durch eine gezielte Veränderung der übertragenen Daten ist es möglich eine DOS-Attacke auszuführen und sogar eigenen Programmcode am Server auszuführen. Um eine Schwachstelle in der Deserialisierung auszunutzen, benötigt man eine Funktion die ausgeführt wird. Am häufigsten werden *Setter*-Methoden verwendet. Im Gegensatz zu .NET (beziehungsweise C#) gibt es in Java keine richtigen Setter sondern nur Methoden welche üblicherweise verwendet werden, um private Variablen zu setzen. Oft wird von den Bibliotheken auch nicht überprüft ob es eine Variable gibt, die zu der Methode gehört und ob die Namen der Methoden immer das gleiche Muster haben (wie zum Beispiel *setProduct*). Das kann es dem Angreifer ermöglichen Methoden aufzurufen, welche nicht als *Setter* gedacht sind (zum Beispiel *setup*). Solange der Angreifer nicht die Möglichkeit hat eine Typinformation zu übergeben. In diesem Fall muss die Anwendung den Typ selbst herausfinden. Dazu erstellt sie einen Objektgraphen und überprüft die verschiedenen Typen im Graphen mit Hilfe einer Whitelist. Dieses Verfahren ist am sichersten und wird von den bekanntesten Bibliotheken umgesetzt und sie können sicher verwendet werden unter der Voraussetzung, dass die Konfigurationen so beibehalten werden, dass der Angreifer den Objekttyp nicht bestimmen kann. Es wäre notwendig diese Einstellungen zu ändern, wenn man Objekte übertragen will, deren Typ zur Zeit des Kompilierens noch nicht bekannt sind. Anhand des Codebeispiels 3.6 ist ersichtlich, wie die Typflexibilität bösartig ausgenutzt werden kann. Die JSON-Daten können dazu führen, dass die, in der Pfadangabe referenzierte, ausführbare Datei ausgeführt wird.

```
1 {"$type":"System.Configuration.Install.AssemblyInstaller, System.
  ↪ Configuration.Install, Version=4.0.0.0, Culture=neutral,
```

```
↪ PublicKeyToken=b03f5f7f11d50a3a", "Path":"file:///c:/somePath/
↪ MixedLibrary.dll"}
```

Listing 3.6: JSON Daten

Bei Listing 3.7 ist die Methode *set_Path* der Klasse *System.Configuration.Install.AssemblyInstaller* zu sehen, welche verwendet wird um eine Dynamically Linked Library (DLL)-Datei (eine ausführbare Datei in Windows) zu laden und sie zu installieren (vgl. Muñoz und Mirosh, [48]).

```
1 // System.Configuration.Install.AssemblyInstaller
2 public void set_Path(string value)
3 {
4     if (value == null)
5     {
6         this.assembly = null;
7         this.assembly = Assembly.LoadFrom(value);
8     }
9 }
```

Listing 3.7: *set_Path*-Methode

Muñoz und Mirosh, [48]

3.11 Domain Name System-Spoofing

DNS-Spoofing steht im Zusammenhang mit URL-Spoofing, wobei sich der Ablauf grundlegend unterscheidet. Beim URL-Spoofing wird versucht den Benutzer zu täuschen, indem eine bösartige Webseite eine möglichst ähnliche URL verwendet wie die einer vertrauten Seite. Um dem Benutzer vorzutäuschen er befände sich auf der Seite des Zahlungsanbieters PayPal[4], könnte der Angreifer eine möglichst ähnliche URL verwenden. DNS-Spoofing findet allerdings auf einer niedrigeren technischen Ebene statt - der Anwendungsschicht des OSI-Modell (vgl. Tripathi, Swarnkar und Hubballi, [67]).

[4] https://www.paypal.com/

4 Sicherheitsempfehlungen

In diesem Kapitel werden Empfehlungen vorgestellt welche die Möglichkeiten, im Kapitel 3 beschriebene, Schwachstellen auszunutzen, minimieren sollen. Allerdings sollte die Sicherheit in alle Phasen eines Projektes mit einfließen. Während des Entwicklungsprozesses sollten automatisierte Tests durchgeführt werden. Statische und dynamische Codeanalysen können auch schon während des Implementierungsprozesses durchgeführt werden (einen kurze Auswahl an Testsoftware dafür gibt es in Kapitel 2.5). Manuelles Testen wird am Ende der Entwicklung einer Funktionalität durchgeführt. In dieser Arbeit wird trotz der Wichtigkeit in allen Phasen der Softwareentwicklung auf Sicherheit zu achten, nur auf Maßnahmen im Programmcode während der eigentlichen Entwicklung eingegangen (vgl. Deepa und Thilagam, [12]).

4.1 Structured Query Language Injection

Stored Prozeduren oder 'gespeicherte Prozeduren' sind in der Datenbank gespeicherte Anfragen, welche im Normalfall nur einmalig kompiliert werden müssen. Daraus ergibt sich der Vorteil, dass die Verwendung von Stored Prozedures performanter ist, gegenüber Abfragen (dynamische Abfragen), die erst zur Laufzeit kompiliert werden. Außerdem ist die Verwendung von Stored Prozedures die Standardempfehlung um SQL-Injection 3.6 vorzubeugen. Vor SQL-Injections schützen sie allerdings nur, wenn sie richtig umgesetzt werden, denn dann werden die Parameter der Prozedur nicht als Code, sondern nur als eine Eingabe interpretiert und die Logik der Abfrage ist nicht mehr änderbar (vgl. Deepa und Thilagam, [12]). 4.1 zeigt eine Stored Procedure welche für SQL-Injections anfällig ist.

```
1 create procedure dbo.getUserWithCredentialsDynamic
2     @username nvarchar(15) not null,
3     @passwordHash nvarchar(44) not null
4 AS
5     declare @queryString nvarchar(500)
6     set @insertString = 'select username, firstName, lastName from
      ↪ myDatabase.users
7       where username = ' + @username + ' and passwordHash = ' +
        ↪ @passwordHash
8     exec sp_executesql @queryString
9 GO
```

Listing 4.1: Stored Procedure die für SQL-Injections anfällig ist

(vgl. Guyer u. a., [21])

Zwar bieten dynamische Stored Procedures wie diese mehr Flexibilität, welche aber auch von Angreifern ausgenutzt werden kann.
Um SQL-Injection zu verhindern ist Listing 4.2 ein Beispiel, welches die korrekte Erstellung einer Stored Procedure veranschaulicht. Der hier verwendete SQL-Dialekt ist 'Transact-SQL', welcher beim Datenbankmanagementsystem SQL-Server, der Firma Microsoft, verwendet wird.

```
1 create procedure dbo.getUserWithCredentials
2     @username nvarchar(15) not null,
```

```
3     @passwordHash nvarchar(44) not null
4 AS
5     select username, firstName, lastName from myDatabase.users
6     where username = @username and passwordHash = @passwordHash
7 GO
```

Listing 4.2: Empfohlene Schreibweise für Stored Procedures

(vgl. Guyer u. a., [22])

Dieser SQL-Code erstellt eine Prozedur 'getUserWithCredentials' welche als Parameter zwei Zeichenketten erwartet. Hier ist bereits eine implizite Validierung enthalten - erstens wird sichergestellt, dass die Parameterwerte vom richtigen Typ sind, sowie nicht dem Typ *null* entsprechen und die Maximallänge von 15 beziehungsweise 44 nicht überschreiten. 4.3 veranschaulicht wie diese Prozedur in Java-Code aufgerufen werden kann. Die Verbindung zur Datenbank ist nicht Teil des Beispiels.

```
1 User user = null;
2 try (Connection connection = dataSource.getConnection();
3      CallableStatement cstmt = connection.prepareCall("{call dbo.
       ↪ getUserWithCredentials(?, ?)}");) {
4      cstmt.setString(1, username); //username gets already set
       ↪ somewhere before
5      cstmt.setString(2, passwordHash); //passwordHash gets already
       ↪ set somewhere before
6      ResultSet rs = cstmt.executeQuery();
7
8      if(rs.next()) { //only takes the first dataSet - assumed that
       ↪ the username is unique
9   user = new User();
10  user.setUsername = rs.getString("username");
11  user.setFirstName = rs.getString("firstName");
12  user.setLastName = rs.getString("lastName");
13  return user;
14     } else {
15        throw new WrongCredentialsException(username, passwordHash);
16     }
17 }
18 catch(Exception exception) {
19     //handle and log exception
20 }
```

Listing 4.3: Aufruf einer Stored Procedure in Java

(vgl. Guyer u. a., [23])

4.2 Cross-Site Scripting

Im Gegensatz zu 3.6 gibt es gegen XSS keine vergleichsweise einfache Gegenmaßnahme wie 4.1s, doch trotzdem ist eine Absicherung dagegen möglich. Besondere Vorsicht ist geboten mit der Verwendung von Benutzereingaben auf dem clientseitigen Teil der Webanwendung (die Darstellung und Logik im Browser). Das Ziel ist es die Benutzereingaben bei der Anzeige in einem Kontext zu behalten, bei dem kein Code ausgeführt wird. Mit welchen Techniken das erreicht werden kann, wird nachfolgend erläutert.
Im Grunde gibt es 3 verschiedene Möglichkeiten dem Problem zu entgegnen. Die erste Möglich-

keit ist Inputvalidierung. Es ist ratsam nur die notwendigsten Zeichen zu erlauben und die Restriktionen auf jeden Fall am Server zu überprüfen. Eine clientseitige Validierung ist zwar möglich und erhöht möglicherweise die Benutzerfreundlichkeit, sollte aber niemals serverseitige Validierung ersetzen. Außerdem sollte bei der Validierung darauf geachtet werden, dass Zeichenketten mit *whitelists* statt *blacklists* überprüft werden. Bei ersteren wird angegeben welche Zeichen erlaubt sind und jedes andere Zeichen gilt als nicht erlaubt. Bei *blacklists* ist es genau umgekehrt. Das Problem dabei ist, dass meistens nicht erlaubte Zeichen vergessen werden, welche ein Angreifer für bösartige Zwecke nutzen kann (vgl. OWASP-Foundation, [53]). Listing 4.4 Beispiel für die Verwendung einer *Regular Expression*.

```
1 Pattern regex = Pattern.compile("[^a-zA-Z0-9]");
2 System.out.println(regex.matcher("abc123").find()); //false -> no
  ↪ illegal char found
3 System.out.println(regex.matcher("abc-123").find()); //true ->
  ↪ illegal char found
4 System.out.println(regex.matcher("abc 123").find()); //true ->
  ↪ illegal char found
```

Listing 4.4: Verwendung von Regular Expressions in Java

(vgl. Oracle, [50])

Hier wird nach allen Zeichen gesucht die keine Groß- oder Kleinbuchstabe oder eine Zahl sind. In diesem Fall wären auch Umlaute nicht erlaubt.

Die 2. Möglichkeit ist *Entity Encoding*. Da einige Zeichen in HTML eine besondere Bedeutung haben und somit zu unerwartetem Verhalten führen können, werden diese kodiert. Dabei wird von, in HTML vordefinierten, Entitäten Gebrauch gemacht. Das ist notwendig, um diese speziellen Zeichen in ihrer textuellen Form darstellen zu können. HTML-Entitäten beginnen immer mit dem Zeichen '&'. Beispielsweise wird das Zeichen '<' durch '<' ersetzt (vgl. Zalewski, [75, S. 76]). Die Kodierung der Antwort an den Client sollte am Server stattfinden. In Java beispielsweise braucht man dazu eine externe Klasse - 'StringEscapeUtils'[1] der Bibliothek 'Apache Commons Text'. Darin befindet sich die Methode 'escapeHtml4' welche den gewünschten Text kodiert. In vielen anderen Programmiersprachen ist eine vergleichbare Methode standardmäßig enthalten. Um den Benutzer vor XSS zu schützen reicht diese Maßnahme allerdings nur aus, wenn der Text nur als HTML verwendet wird - das bedeutet er darf in keinen *script*-Tags, Attributen von Elementen, Cascading Stylesheet (CSS)-Code Order URLs vorkommen.

Die 3. Methode ist Escaping. Hierbei werden Zeichen die in dem verwendeten Kontext eine besondere Bedeutung haben, durch ein vorangestelltes *Escape*-Zeichen markiert und somit als Text interpretiert. In vielen Sprachen ist der Backslash (\) das *Escape*-Zeichen. Wichtig ist allerdings, dass auch das *Escape*-Zeichen selbst, wenn es im eigentlichen Text vorkommt, markiert wird. Am sichersten ist die Verwendung von Escaping, es ist jedoch auch am umständlichsten. Die 3 genannten Maßnahmen schließen sich gegenseitig jedoch nicht aus und können daher auch gemeinsam eingesetzt werden. Es empfiehlt sich für Escaping eine Bibliothek zu benutzen, die die Möglichkeit bietet potentiell gefährliche Eingaben je nach Verwendungszweck unschädlich zu machen. Für Java ist 'OWASP Java Encoder Project'[2] eine Möglichkeit (vgl. OWASP-Foundation, [53]).

[1] https://commons.apache.org/proper/commons-text/javadocs/api-release/org/apache/commons/text/StringEscapeUtils.html

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

Listing 4.5 stellt die Verwendung der genannten Java-Bibliothek in unterschiedlichen Kontexten dar.

```
1 <body><%= Enocode.ForHtml(userInput) %></body> //use user input as
  ↪ usual html
2 <div style="width:<%= Encode.forCssString(userInput) %>"> //use user
  ↪  input as css attribute value
3 <div style="background:<%= Encode.forCssUrl(userInput) %>"> //use
  ↪ user input as css url
4 <script> //use userInput in a JavaScript variable
5     var msg = "<%= Encode.forJavaScriptBlock(userInput) %>";
6     alert(msg);
7 </script>
8 <a href="/search?value=<%= Encode.forUriComponent(userInput) %>&
  ↪ order=1"> //use userInput in a search link
```

Listing 4.5: Verwendung des OWASP Java Encoder Project in verschiedenen Kontexten

(vgl. OWASP-Foundation, [54])

4.3 Maßnahmen für Cookies

Da Cookies oft für Angreifer wertvolle Informationen, wie zum Beispiel Daten zur Authentifizierung, enthalten, ist es wichtig auch hier möglichst gute Sicherheitsvorkehrungen zu treffen. Ein wichtiger Parameter der bei Cookies gesetzt werden kann ist *domain*. Der Parameter legt fest in welchen Domänen sich ein, mit dem Browser kommunizierender, Server befinden muss, damit der Browser den Cookie bei einer Anfrage an den Webserver mitsendet. Wenn der Parameter nicht gesetzt wird, wird der Hostname des Servers genommen, der den Cookie erstellte. Unterdomänen der angegebenen Domäne gelten als erlaubt und bekommen den Cookie. Bei der Angabe der Domäne *example.com* erhält auch ein Server der Domänen *sub.example.com* den entsprechenden Cookie. Die einzige Ausnahme ist der Fall, dass der Domäne Parameter nicht gesetzt wird. Dann sind Unter-Domänen ausgeschlossen. Die Domäne kann nicht auf eine Unter-Domäne der Domäne des ausstellenden Servers gesetzt werden. Umgekehrt darf der Wert auch nicht zu allgemein sein, wie zum Beispiel *.com*.
Die womöglich wichtigsten Angaben bei einem Cookie sind *HttpOnly* und *Secure*, welche vom Server nur entweder gesetzt oder nicht gesetzt werden können. *HttpOnly* legt fest, dass der Cookie nur für HTTP verwendet werden soll und somit von JavaScript nicht ausgelesen werden kann. Die *Secure*-Option gibt an, dass der Cookie nur über HTTPS-Verbindungen gesendet werden darf. Trotzdem gibt es mit JavaScript noch die Möglichkeit den für die entsprechende Domäne reservierten Speicher auszulasten und den Cookie erneut zu speichern, allerdings ohne die *secure*-Option zu setzen (vgl. Zalewski, [75, S. 149f.]).

4.4 HTTP zu HTTPS Weiterleitung

Auf Grund der Wichtigkeit HTTPS statt HTTP zu verwenden ist es ratsam sicherzustellen, dass die Kommunikation über das sicherere Protokoll abgewickelt wird. Um die Erreichbarkeit der Ressourcen eines Servers über HTTP trotzdem zu ermöglichen, werden am Server beide Protokolle angeboten, die Anfragen an den HTTP-Service aber an den HTTPS-Service weitergeleitet. Listing 4.6 zeigt die gängigste Methode gezeigt um das auf einem Apache-Webserver zu erreichen.

```
1 NameVirtualHost *:80
2 <VirtualHost *:80>
```

```
3     ServerName www.example.com
4     Redirect permanent / https://secure.example.com/
5 </VirtualHost>
6
7 <VirtualHost _default_:443>
8     ServerName secure.example.com
9     DocumentRoot /usr/local/apache2/htdocs
10    SSLEngine On
11 # etc...
12 </VirtualHost>
```

Listing 4.6: HTTP zu HTTPS-Weiterleitung mit einem Apache-Webserver

(vgl. Apache Software Foundation, [4])

4.5 HTTP Strict Transport Security

Das HTTP Strict Transport Security (HSTS)-Protokoll[3] ist eine Erweiterung von HTTPS und soll die Möglichkeiten für Angriffe auf HTTPS-Verbindungen reduzieren. Eine der Schwachstellen der 4.4 ist die Tatsache, dass die Umleitung erst am Server passiert und somit die initialen HTTP-Anfragen unsicher sind. Wenn die Information der Weiterleitung nicht vom Client gespeichert wird, landet ein neuerlicher Aufruf der Webseite mit HTTP als HTTP-Anfrage beim Server.
Mit HSTS ist es möglich, dass der Browser selbst HTTP-Anfragen in HTTPS-Anfragen umwandelt. Dazu sendet der Server einen zusätzlichen Antwortheader an den Browser um mitzuteilen, dass Anfragen nur über HTTPS erfolgen sollen. Der zusätzliche Header kann folgende Parameter enthalten:

- max-age (erforderlich): Gibt an wie lange der Browser die Information maximal speichern soll; die Zeitangabe erfolgt in Sekunden
- includeSubdomains (optional): Wenn gesetzt, wird HSTS auch auf UnterDomänen angewendet
- preload (optional): Wenn gesetzt, wird angenommen, dass die Domäne in die *Preload*-Liste aufgenommen werden soll.

(vgl. Santos u. a., [59])

Die *Preload*-Liste ist eine Auswahl an Domänes welche auch beim ersten Aufruf im Browser mit HTTPS angesprochen werden. Diese Liste ist fix in der Software des entsprechenden Webbrowsers hinterlegt. Um in dieses Liste aufgenommen werden zu können, müssen einige Sicherheitsstandards erfüllt werden[4]. Ein Eintrag in dieser Liste behebt somit das Problem, dass die Server der Domäne über HTTP angesprochen werden können, wenn aus verschiedenen, möglichen Gründen die HSTS-Information im Browser nicht vorhanden ist (vgl. Chromium-Team, [11]).
Auch für diese Maßnahme ist, mit Listing 4.7 ein Teil einer Konfigurationsdatei, eines Apache Servers, zur Veranschaulichung enthalten:

```
1 <VirtualHost *:443>
2 ...
```

[3] https://tools.ietf.org/html/rfc6797
[4] https://hstspreload.org

```
3 Header always set Strict-Transport-Security "max-age=63072000;
  ↪ includeSubdomains; preload";
4 ...
5 </VirtualHost>
```

Listing 4.7: HSTS-Konfiguration mit einem Apache-Webserver

(vgl. Elst, [15])

4.6 Content Security Policiy

Content Security Policy (CSP) ist ein Sicherheitsmechanismus des Browsers, welcher die Ausführbarkeit von JavaScript-Skripten einer Webseite im Browser einschränkt. Sie beugt Injektion-Attacken im Allgemeinen, vor allem aber XSS vor. CSP-Optionen können in einem zusätzlichen HTTP-Header an den Client gesendet werden. CSP ermöglicht den Schutz vor XSS, Clickjacking und Mixed content. Wie alle Sicherheitsmaßnahmen, die in Browsern umgesetzt werden, ist die Wirksamkeit der Maßnahme abhängig von der Implementierung des Browsers. Daher sollte diese Maßnahme nicht als alleinige Maßnahme gegen die genannten Attacken verwendet werden, sondern zusätzlich zu den Maßnahmen aus Kapitel 4.2. Bei der Umsetzung einer CSP ist es häufig notwendig bestehende Teile der Anwendungsimplementierung umzuschreiben, da sie mit der CSP nicht kompatibel wären.
CSP bietet die Möglichkeit von welchen Quellen Ressourcen geladen werden dürfen. Es ist möglich für verschiedene Typen von Ressourcen verschiedene Regeln zu setzen. Dabei kann man unter anderem zwischen Bildern, Skripten (engl. *script-files*) und Stylesheets unterscheiden. Die am häufigsten verwendete Umsetzungsvariante sind *whitelists* in denen die Domänen von denen Ressourcen geladen werden dürfen angegeben sind. Das Problem bei diesem Ansatz ist jedoch, dass eine Schwachstelle bei einem der angegebenen Domänen genutzt werden kann um die gesamte CSP zu umgehen. Abbildung 4.1 verdeutlicht, dass mit einer steigenden Anzahl an gelisteten Domänen das Sicherheitsrisiko stark ansteigt.

Daher wird ein CSP-Ansatz empfohlen, der ohne *whitelists* auskommt: *Nonces*. Hierbei wird im CSP-Header *script-src 'nonce-randomValue'* angegeben. Der *randomValue* sollte für jede Anfrage des Browsers neu generiert werden. Um diese CSP-Policy zu erfüllen, muss bei jeder inline in einem HTML-Dokument referenzierte Ressource der gleiche generierte Wert in einem *nonce*-Attribut angegeben sein. Es ist empfehlenswert die genaue Umsetzung der nonce-basierten Policy einem Framework (wie Django oder React) zu überlassen, doch auch Webpack ist dazu in der Lage. Mit dem beschriebenen Ansatz gibt es allerdings das Problem, dass Scripts, die im Browser erstellt werden, nicht vertraut wird. Um dieses Problem zu beheben, wird dem Header ein weiterer Wert *'script-dynamic'* hinzugefügt. Dadurch werden Skripts erlaubt, die dynamisch im Browser erstellt werden. Das Erteilen dieser Erlaubnis ist ohne Probleme möglich, da nur Skripte denen bereits vertraut wird weitere Skripte erstellen können. Es gilt jedoch zu beachten, dass die empfohlenen Veränderungen Injektion-Attacken nicht unmöglich machen, sondern nur die Möglichkeiten einschränken sollen. Beispielsweise ist es bei der Verwendung der *'script-dynamic'*-Option möglich bösartige, externe Skripte einzubinden, wenn der Angreifer einen Einfluss auf die URL der Ressource hat (vgl. Weichselbaum u. a., [71]).

4.7 Subresource Integrity

Eine weitere Maßnahme, um seine Webanwendungen gegen manipulierte und eingeschleuste Ressourcen zu schützen, ist Subresource Integrity (SRI). Hierbei wird dem Element, welches eine externe Quelle referenziert, ein *integrity*-Attribut hinzugefügt. Der Wert des Attributs ist einer oder mehrere Hashwerte. Der Hashwert wird anhand des Inhalts erstellt, den die Ressource enthalten

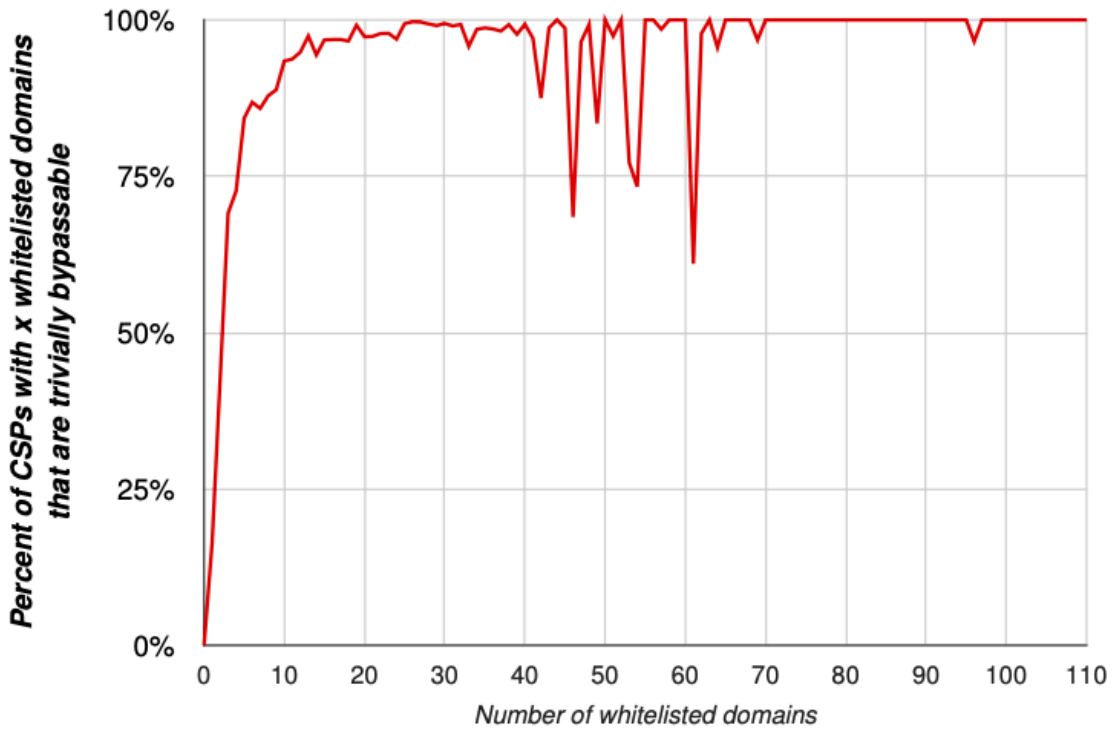

Abbildung 4.1: Verhältnis zwischen umgehbarer CSPs und der Anzahl an verwendeten Domänes in der *whitelist* (vgl. Weichselbaum u. a., [71])

soll. Das Konzept wird durch die Verwendung des *integrity*-Attributs bei einem *script*-Tag verdeutlicht in Listing 4.8 verdeutlicht.

```
1 <script src="https://cdn.com/exampleLibrary/version_3_15.js"
  ↪ integrity="sha384-dOTZf16X8p34q2/
  ↪ kYyEFm0jh89uTjikhnzjeLeF0FHsEaYKb1A1cv+Lyv4Hk8vHd
2 sha512-
  ↪ Q2bFTOhEALkN8hOms2FKTDLy7eugP2zFZ1T8LCvX42Fp3WoNr3bjZSAHeOsHrbV1F
  ↪  u9/A0EzCinRE7Af1ofPrw==" />
```

Listing 4.8: SRI-Konfiguration auf einem Apache-Webserver

Der Wert des *integrity*-Attributs muss eine Liste sei,n die Zeichenfolgen folgender Struktur enthält: *Algorithmenbezeichung-Hashwert*. Der *Hashwert* ist Base64[5]-kodiert. Browser, die diesen Mechanismus unterstützen, müssen zumindest die Algorithmen *SHA-256, SHA-384* und *SHA-512* anbieten. Derzeit wird die Verwendung des Attributs lediglich bei den *script*- und *link*-Elementen unterstützt. SRI kann auch gemeinsam mit CSP verwendet werden. Dafür wird folgender Wert dem CSP-Header beigefügt: *require-sri-for script*. Damit werden JavaScript-Ressourcen nur vom Browser ausgeführt, wenn SRI bei der Ressource verwendet wird und erfolgreich überprüft wurde. Diese Anforderung kann auch für Ressourcen verwendet werden, die mit dem *style*-Tag inkludiert werden (vgl. MDN-Contributors, [41]).

[5] https://tools.ietf.org/html/rfc4648

4.8 Same Origin Policy

Die SOP ist ein zentrales Sicherheitskonzept von Webapplikationen, welches von Browsern umgesetzt wird. Im Gegensatz zu den vorherigen Sicherheitsmechanismen wird diese Maßnahme von Browsern standardmäßig umgesetzt. Wobei anzumerken ist, dass auch hier das Verhalten der verschiedenen Browser nicht in jedem Fall gleich ist. Die SOP gliedert sich in verschiedene Bereiche wobei es immer darum geht, verschiedene Applikationen voneinander zu trennen und die Kommunikation untereinander zu unterbinden. Wann eine Applikation als verschieden von einer anderen gesehen wird, hängt von dem Anwendungsbereich der SOP ab. Im Allgemeinen kann aber davon ausgegangen werden, dass zwei Applikationen als verschieden gesehen werden, wenn sich das Protokoll, der DNS-Name oder die Portnummer unterscheiden.
Bei der Betrachtung der SOP sollte man folgende Bereiche unterscheiden: LocalStorage, Cookies, das DOM sowie Netzwerkanfragen (vgl. Zalewski, [75, S. 141-149]). In dieser Arbeit wird lediglich auf die Netzwerkanfragen im Kontext von SOP näher eingegangen. Standardmäßig erlaubt ein Browser Zugriffe eines Skripts einer Seite mit der URL *https://www.expampleSite.com* auf Server einer Seite mit der URL *https://exampleSite.com:400* nicht. In diesem Fall wäre der Grund dafür der unterschiedliche DNS-Name sowie eine andere Portnummer. Beim Fehlen der Portnummer wird der Standardport für das jeweilige Protokoll verwendet (bei HTTPS entspricht das 443). Da es manchmal notwendig ist trotzdem auf andere Server zuzugreifen, gibt es das Konzept CORS, welches im Kapitel 3.8 bereits erwähnt wurde. Bei CORS können Webserver folgend auf eine Clientanfrage der Antwort den *Access-Control-Allow-Origin (ACAO)*-Header anfügen, welcher die Verarbeitung der Antwort im Browser erlaubt. Durch den *origin*-Header in der Anfragenachricht an den Server weiß der Server zu welcher Domäne die im Browser ausgeführte Logik gehört. Um festzulegen welchen Applikationen diese Erlaubnis erteilt wird, gibt es die Möglichkeit den ACAO-Header mit folgenden möglichen Werten anzugeben: *'null'*, *'*'* oder eine spezifische Domäne. Die Verwendung der ersten beiden Werte wird nicht empfohlen, da im Falle eines *origin*-Header-Werts von *'null'* die Verarbeitung der Antwort erlaubt wird. Bei der Verwendung des Werts *'*'* wird jeder Domäne vertraut. Die einzige sichere Methode ist eine Domäne anzugeben. Der ursprüngliche CORS-Standard von W3C[6] erlaubt eine Liste von Domänes. Mit der Zeit hat sich jedoch der WHATWG Fetch Standard[7] etabliert, in welchem nur eine Domäne erlaubt ist. Falls die Nutzung der Ressourcen von Browserkontexten verschiedener Domänen möglich sein soll, ist es somit notwendig den Header zur Laufzeit auf die anfragende Domain zu setzen. Hierbei wird empfohlen, SubDomänes nicht zu erlauben. Das bedeutet, dass nicht auf ein Vorkommen der Domäne im *origin*-Wert gesucht werden, sondern ein exakter Vergleich durchgeführt werden soll. Um das zu verdeutlichen dient folgendes Beispiel: Der *origin*-Header ist *https://sub.expample.domain.com:500*. Die erlaubte Domäne ist *example.domain.com*. Hier würde *sub.example.domain.com* auf Gleichheit mit *example.domain.com* überprüft werden. In diesem Fall sollte keine Erlaubnis gegeben werden, was am besten erreicht wird, in dem eine Fehlerantwort geschickt wird. Auch das Protokoll und die Portnummer können, je nach Ansprüchen an Flexibilität und Sicherheit, in die Überprüfung einbezogen werden. Dem *orign*-Wert *'null'* sollte ebenso nicht vertraut werden (vgl. Chen u. a., [10]).

4.9 Cross-Site-Request-Forgery-Token

In diesem Kapitel wird eine Gegenmaßnahme zur, in Kapitel 3.7 beschriebenen, CSRF-Attacke vorgestellt. Es müssen lediglich die Endpunkte des Webservers abgesichert werden, bei denen der Status eines Objekts verändert wird. Im Normalfall sind das POST-Anfragen (siehe Kapitel

[6] https://www.w3.org/TR/cors/
[7] https://fetch.spec.whatwg.org/ Kapitel *core-protocol*

2.1.1). Die gängigste Variante um CSRF zu verhindern ist, die Verwendung eines zufällig generierten Tokens. Der Token wird am Server erstellt und in den normalen HTML-Ressourcen an den Browser gesendet. Hierbei muss der erstellte Wert vor allem pro Benutzer pro Session einmalig sein. Die Sicherheit wird erhöht, wenn der Token außerdem pro Serveranfrage einmalig ist. In diesem Fall führt das Laden der vorigen Seite im Browser mittels des *Zurück*-Buttons aber zu einem veralteten Token. Am Server wird, bei der Anfrage für eine statusverändernde Aktion, der Token auf eine Übereinstimmung überprüft. Falls die Werte nicht übereinstimmen muss der Server die Ausführung der Aktion abbrechen. Das bedeutet, dass der Server den zuvor generierten Token bis zum Zeitpunkt der Überprüfung Zwischenspeichern muss. Um diese Maßnahme zu implementieren empfiehlt es sich eine Bibliothek zu verwenden, die dafür gedacht ist. Wenn die am Server verwendete Programmiersprache Java ist, kann unter anderem Spring Security[8] oder OWASP CSRFGuard[9] dafür verwendet werden. Um eine Absicherung gegen CSRF möglich zu machen ist es auch notwendig, sich gegen XSS-Attacken abzusichern (vgl. OWASP-Foundation, [52]). Ausführliche Informationen dafür gibt es im Kapitel 4.2.

[8] https://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/
[9] https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project

5 Schadsoftware Poisontap

Poisontap ist eine Schadsoftware, welche eine Reihe von verschiedenen Attacken durchführt. Es werden ohne das Wissen des Benutzers zahlreiche, häufig verwendete, Webseiten aufgerufen und Teile der Ressourcen verändert oder ersetzt und für viele Jahre lang im Cache des Browsers gespeichert. Außerdem wird am DNS-Cache des angegriffenen Computers ein DNS-Rebinding durchgeführt. Das bedeutet, dass die gespeicherte IP-Adresse zu einer bestimmten Domain verändert wird. Weiters wird mit dem Server des Angreifers eine dauerhafte WebSocket-Verbindung hergestellt. Das führt dazu, dass auch nach dem eigentlichen Angriff Zugriffe auf den Computer möglich sind und weitere Angriffe gemacht werden können. Erzielt wird das durch die Ausnutzung verschiedener Sicherheitslücken der Webanwendungen.

5.1 Funktionsweise

In diesem Kapitel wird die allgemeine Vorgangsweise näher erläutert.
Um Poisontap ausführen zu können wird ein Gerät benötigt, welches Internetzugriffe simulieren kann, wie zum Beispiel ein Raspberry Pi Zero[1]. Sobald das Gerät an einem Computer angeschlossen ist und erkannt wurde gibt es vor, dass alle möglichen IPv4-Adressen in seinem lokalen Netzwerk enthalten sind. Das führt dazu, dass alle anschließenden Netzwerkanfragen an Poisontap geschickt werden, selbst wenn es bereits bestehende Verbindungen per Ethernet oder Wireless Local Area Network (WLAN) gibt. Der Grund dafür ist, dass ein Subnetz eines Netzwerks mit niedriger Priorität, der Standardadresse (Adresse des Routers) eines Netzwerks mit höherer Priorität vorgezogen wird. Dieser Ablauf funktioniert unabhängig davon ob der Computer gesperrt ist oder nicht.
Falls auf dem Computer in einem Browser Webseiten geöffnet sind, ist die Wahrscheinlichkeit hoch, dass Netzwerkanfragen gemacht werden, da die meisten Webseiten Netzwerkanfragen zu verschiedensten Zwecken senden, auch wenn keine Benutzeraktivität vorhanden ist. Wenn kein lokaler DNS-Server verwendet wird ist es für Poisontap möglich, die Anfrage direkt zum Poisontap-Webserver umzuleiten. Da alle Anfragen, bei denen der Empfänger im lokalen Netzwerk liegt, gegenüber dem Poisontap-Netzwerk priorisiert werden, wird die Zieladresse richtig aufgelöst. Im Falle einer Zieladresse außerhalb des lokalen Netzwerks wird anschließend allerdings trotzdem die Anfrage an Poisontap und den Webserver darin gesendet. Als Antwort sendet der Webserver eine Antwort die als HTML oder JavaScript interpretiert werden kann. X-Frame-Options sind in diesem Fall nutzlos, da der Poisontap-Webserver entscheidet, welche Header an den anfragenden Browser gesendet werden. Die Antwort des Poisontap-Webservers erstellt, in dem Fall, dass sie im Browser ausgeführt wird, viele nicht sichtbare *iframes*, welche Anfragen an die meistverwendeten Webseiten[2] senden. Das Schema der angefragten URLs sieht folgendermaßen aus: *Protokoll://hostname/PoisonTap*, zum Beispiel *http://nfl.com/PoisonTap*. Anfragen von Webseiten beinhalten oft Cookies die verwendet werden können um den Benutzer zu authentifizieren, oder andere zustandsbehaftete, benutzerbezogene Daten zu speichern. Somit werden bei den Webseiten, die Cookies zur Authentifizierung benutzen, aktive Sessions weiterverwendet. Auch die *HttpOn-*

[1] https://www.google.com/search?client=safari&rls=en&q=raspberry+pi+zero&ie=UTF-8&oe=UTF-8
[2] https://www.alexa.com/topsites

ly-Option ist in diesem Fall nutzlos, da die Cookies über HTTP gesendet werden. Ebenso nutzlos ist in diesem Fall die Same-Origin Policy, da es für den Browser aussieht als würde der richtige Server kontaktiert werden. Falls HTTPS verwendet wird, aber bei Cookies nicht die *Secure*-Option gesetzt wurde, kann HTTPS umgangen werden und die Cookies an Poisontap gesendet werden.
Poisontap antwortet auf die Anfragen der *iframes* mit in HTML eingebettetem JavaScript-Code, der eine persistente WebSocket-Verbindung zu dem Server des Angreifers (im Internet, nicht der WebServer am Poisontap-Gerät) erstellt. Poisontap wählt die Caching-Optionen dieser Antworten so, dass sie für einen sehr langen Zeitraum (viele Jahre) im Browsercache gespeichert bleiben.

5.2 Gegenmaßnahmen

In diesem Kapitel wird darauf eingegangen, welche Maßnahmen Entwickler von Webanwendungen implementieren können, um Angriffe von Poisontap zu verhindern. Die Benutzer der Seiten haben zum Schutz vor den genannten Angriffen folgende Möglichkeiten: Schließen der Browser beim Verlassen des Computers sowie das Deaktivieren der Anschlüsse die für das Gerät mit Poisontap verwendet werden können (Universal Serial Bus (USB), Thunderbolt, ...). Eine Möglichkeit ist dafür zu sorgen, dass die Daten auf der Festplatte verschlüsselt werden, sobald sich der Computer im Ruhezustand befindet. Die Notwendigkeit eines Passworts, um die Daten zu entschlüsseln führt dazu, dass der Browser im Ruhezustand keine Netzwerkanfragen mehr machen kann.
Poisontap ist möglich, wenn die Webanwendung Zugriffe per HTTP (und nicht nur HTTPS) erlaubt oder die *Secure*-Option bei Cookies der Seite nicht aktiviert ist. Eine weitere Möglichkeit ist, dass die Webanwendung Zugriffe auf externe Seiten ohne ausreichende Überprüfung der Ressourcen erlaubt. Die Möglichkeit des Ladens von Ressource kann mittels einer CSP (siehe Kapitel 4.6) eingeschränkt werden. Eine weitere Möglichkeit ist die Überprüfung der geladenen Inhalte auf ihre Integrität mittels des SRI-Mechanismus (siehe Kapitel 4.7). Der Grund dafür ist, dass Cache Poisoning (siehe Kapitel 3.4) von Ressourcentypen auf die SRI angewendet wird, somit nicht mehr möglich ist.

6 Browsererweiterung

Im Rahmen dieser Arbeit wurde eine Browsererweiterung erstellt, welche den Einsatz der Schadsoftware Poisontap (Kapitel 5) erkennen soll. Die Browsererweiterung mit dem Namen *Poison-Protect* wurde für den Browser Google Chrome entwickelt. Daher beziehen sich Informationen zu Browsererweiterungen in diesem Abschnitt nur auf selbigen Browser.

6.1 Anforderungen

Die Browsererweiterung soll in der Lage sein zu erkennen, wann Poisontap verwendet wird und die Verbindungen zu unterbrechen. Die Browsererweiterung ist momentan nur für den Browser Chrome verfügbar. Angenommen auf einem Computer sind die Browser Firefox und Chrome geöffnet und ein Poisontap-Gerät wird an denComputer angeschlossen. In diesem Fall kann die Browsererweiterung zwar die Verbindung in Chrome unterbrechen, jedoch nicht in Firefox. Erwartet wird von der Browsererweiterung, dass sie in der Lage ist Poisontap so früh zu erkennen und zu stoppen, sodass in weiterer Folge keine bösartige Verwendung mit Poisontap möglich ist. Das macht es erforderlich, dass folgende Funktionen verhindert werden: das Speichern dauerhafter Einträge im Browser Cache, das Senden von Cookies an Poisontap, sowie das Erstellen von Möglichkeiten für nachträgliche Zugriffe über das Internet.
Da es in Chrome einfach möglich ist, die Erweiterung zu deaktivieren, wird empfohlen, den Computer nicht unbeaufsichtigt im entsperrten Modus zu lassen.

6.2 Konzept

Eine Browsererweiterung ist ein kleines Programm, das im Browser installiert werden kann und verschiedenste Funktionalitäten anbieten kann. Es ist beispielsweise möglich das Aussehen von Webseiten zu verändern, Informationen zu Netzwerkanfragen des Browsers zu erhalten und zu blockieren und Daten offline, im Browser zu speichern. Die aufgerufenen Seiten haben allerdings keinen Zugriff auf den Code oder die Daten von Browsererweiterungen. Ausführlichere Informationen zu Browsererweiterungen bietet eine Webseite von Google[1]. PoisonProtect benötigt Informationen zu Netzwerkanfragen aller URLs. Konkret wird die IP-Adresse und die Domäne einer Anfrage ausgelesen. Außerdem hat PoisonProtect die Möglichkeit Anfragen zu blockieren. Ansonsten erhält die Erweiterung unbegrenzte Speicherkapazität, konkret benötigt sie zwischen 100 und 200MB.
Um die Ausführung von Poisontap zu verhindern, wurden mehrere Abwehrmechanismen in betracht gezogen, wobei alle versuchen das DNS-Spoofing zu erkennen. Dabei wurde vor allem darauf Wert gelegt, dass die Maßnahme möglichst schwer zu umgehen ist. Die erste Möglichkeit ist eine verschlüsselte Variante des DNS-Protokoll zu nutzen. Das wäre beispielsweise mit einem Google DNS-Server möglich[2]. Das Problem dabei ist allerdings, dass man hierbei davon abhängig, dass die jeweilige Domäne ebenfalls eine sichere Variante von DNS verwendet. Eine weitere Möglichkeit ist das Ausnutzen der Tatsache, dass der Server, der sich bei Poisontap als der Ziel-

[1] https://developer.chrome.com/extensions
[2] https://developers.google.com/speed/public-dns/docs/secure-transports

server der Netzwerkanfrage ausgibt, immer dieselbe IP-Adresse verwendet. Es ist möglich, dass verschiedenen Domänen dieselbe IP-Adresse zugeordnet ist. Allerdings ist es je nach Anzahl der besuchten, verschiedenen Webseiten unwahrscheinlich, dass alle den gleichen Zielserver haben. Diese Annahme wird noch durch die Beobachtung von Butkiewicz, Madhyastha und Sekar, [7] dass Webseiten mit einem hohe Rankingplatz Anfragen an mehrere verschiedene Server schicken. Die letzte Möglichkeit ist, die korrekten IP-Adressen der Seiten, die Poisontap aufruft, zu speichern und zu erkennen, wenn die Absendeadresse der Antwort von der gespeicherten Adresse abweicht.

6.3 Implementierung

Für die Umsetzung wurde die Verwendung des Abwehrmechanismus gewählt, bei dem die Antwort-IP-Adresse mit der korrekten Adresse verglichen wird. Der Grund dafür ist, dass im Gegensatz zur ersten Möglichkeit keine Abhängigkeit von den Betreibern der Domänen besteht. Der zweite Vorteil ist, dass die Maßnahme wirksam bleibt, selbst wenn der Poisontap-Server mehrere verschiedene IP-Adressen verwendet.
Um den Domänen ihre richtigen IP-Adressen zuzuordnen, wurde ein Python-Skript verwendet. Das Skript (Kapitel 9.4) arbeitet eine Liste der wichtigsten 1 Million Webseiten ab, da diese Liste ebenso von Poisontap verwendet wird. Die Adressauflösung erfolgt durch den *acDNSResolver* (Kapitel 2.2) des ausführenden Computers. Die Zuordnungen werden durch PoisonProtect in einer Offline-Datenbank (IndexedDB[3]) im Browser gespeichert. Die Buchstaben der Domänen werden in umgekehrter Reiheinfolge in die Ausgabedatei geschrieben. Beispielsweise wird *www.google.com* als *moc.elgoog.www* abgespeichert. Der Grund ist, dass Subdomains so leichter erkannt werden. Diese Tatsache wird bei der genaueren Erklärung der Implementierungslogik ersichtlich werden.
In der Datei *manifest.json* (Kapitel 9.2) befinden sich Konfigurationsdaten wie der Name der Anwendung, die Berechtigungen sowie die Verknüpfung zur Logik des Programms - *background.js* (Kapitel 9.3). Folgend wird näher auf genannten JavaScript-Code eingegangen. Sobald PoisonProtect das erste Mal ausgeführt wird, wird zuerst die Funktion *initializeDataBase* ausgeführt. In ihr werden die vom Python-Skript erstellten DNS-Daten eingelesen und in der Datenbank gespeichert. Die Daten werden asynchron aus 10 verschiedenen JSON-Dateien geladen, in ein gemeinsames Array gespeichert und anschließen sequentiell in die Datenbank hinzugefügt. Die Datensätze bestehen aus dem Namen der Domäne welcher als Schlüssel dient. Außerdem ist eine Liste an IP-Adressen zugeordnet, sowie ein Zeitstempel der letzten Änderung. Beim Erstellen der Datenbank wird eine Maxmialanzahl an Einträgen als die Initialgröße zuzüglich 10% festgelegt. Beim Erreichen dieser Anzahl wird beim Einfügen gleichzeitig der älteste Eintrag gelöscht. Die Größe der Datenbank kann sich verändern wenn eine Anfrage getätigt wird dessen Domäne nicht in der Datenbank vorhanden ist und der Zustand die Anwendung zur Zeit nicht im blockierenden Zustand ist.
Im Anschluss an das Erstellen der Datenbank werden außerdem *WebRequest-Listener* gesetzt. Bei jeder Netzwerkanfrage werden die IP-Adresse und die Domäne ausgelesen und mit der Datenbank abgeglichen. Wenn die Logik feststellt, dass zu viele ungültige Anfragen gemacht wurden, werden alle folgenden Anfragen für einen gewissen Zeitraum verweigert. Die Tatsache, dass eine IP-Adresse nicht mit der in der Datenbank gespeicherten Adresse übereinstimmt, muss nicht zwingendermaßen DNS-Spoofing als Ursache haben, da sich IP-Adressen auch legitimerweise ändern können. Im Array *lastDistinctRequests* werden für einige Minuten alle Anfragen protokolliert. Es wird gespeichert, ob die IP-Adresse mit der Datenbank übereinstimmte und wann die

[3] https://developers.google.com/web/ilt/pwa/working-with-indexeddb

Anfrage getätigt wurde. Ebenso ist der Initiator der Anfrage in der Information enthalten. Solange der Anteil an falschen IP-Adressen in dieser Liste 10% nicht übersteigt, werden die Adressen den jeweiligen Datensätzen in der Datenbank hinzugefügt. Bei Domänen mit häufig falschen Adressen werden die falschen Adressen dauerhaft ignoriert. In *lastDistinctRequests* wird pro Domäne nur ein Eintrag gespeichert. Die Liste wird alphabetisch nach den Domänennamen des Initiators sortiert wodurch Unter-Domänen erkannt und entfernt werden können. Durch die Tatsache, dass die Domäne des Initiators gespeichert wird und nicht die Ziel-Domäne ist die Browsererweiterung weniger anfällig für zu viele Updates von IP-Adressen.
Bei jeder neuen Anfrage wird *lastDistinctRequests* neu evaluiert. Somit wird nach einem erkannten Angriff der Zustand nach einer gewissen Zeit wieder zurück gesetzt. Außerdem kann der Benutzer den Zustand ebenso über das Icon der Browsererweiterung zurücksetzen. Beim Zurücksetzen werden die Objekte des Arrays *lastDistinctRequests* so aktualisiert, dass so getan wird als ob alle Anfragen der Liste die korrekte IP-Adresse gehabt hätten. Das wird getan um ein sofortiges, erneutes Blockieren weiterer Anfragen zu verhindern. Die Interaktion mit dem Benutzer ist minimal gehalten - der einzige Teil der als Benutzerschnittstelle bezeichnet werden kann ist das Icon der Browsererweiterung.

6.4 Ergebnisse

Die Browsererweiterung ist im Stande die Verwendung von Poisontap beziehungsweise DNS-Spoofing im Allgemeinen zu erkennen und zu verhindern. Voraussetzung dafür ist lediglich, dass nur der Browser (derzeit lediglich Chrome) geöffnet ist, in dem PoisonProtect installiert ist und dass PoisonProtect eingeschaltet ist.
Die korrekte Funktionsweise der Browsererweiterung wurde durch die Testausführung des Servers der auf auf dem Raspberry Pi Zero gestartet wird, sichergestellt. In diesem Fall wurde der Server allerdings lokal auf dem Computer gestartet der für die Testausführung benutzt wurde. Das verwendete Betriebssystem war *Kali Linux*[4]. Vor dem Starten des Servers wird mit dem *iptables*-Befehl von dem Computer ausgehende Anfragen auf den Ports 80 und 443 (das sind die Standardports der Protokolle HTTP und HTTPS) an den Port weitergeleitet der von dem lokalen Server verwendet wird. Außerdem wird der Befehl *dnsspoof* verwendet um DNS-Anfragen abzufangen und zu beantworten. Es wird vorgetäuscht, dass die angefragt Domäne zu der IP-Adresse des lokalen, verwendeten Computers gehört. Nach dem Starten des Servers wartet dieser auf Anfragen des Browsers. Im Anschluss an eine Anfrage startet der eigentliche Angriff. Nach wenigen Sekunden werden die Anfragen des Browsers von der Browserweiterung blockiert.

Bei der Verwendung von der Browsererweiterung ist es wahrscheinlich, dass die Erweiterung fälschlicherweise in den blockierenden Zustand wechselt. Die Anzahl dieser *false positives* wird sich mit der Zeit verringern, da Seiten die besucht werden in die Datenbank eingetragen werden und bei bereits vorhandenen Einträgen neue IP-Adressen eingetragen werden. Darum erhöht es die Genauigkeit von PoisonProtect wenn es bereits möglichst früh vor einem Angriff unter normalen Umständen verwendet wird. Ein Grund für die genannten *false positives* ist, dass darauf geachtet wurde, dass im Falle eines Angriffs, dieser möglichst früh erkannt wird. Es ist somit ein Kompromiss zwischen Sicherheit und Benutzerfreundlichkeit. Außerdem hat natürlich auch die notwendige Abarbeitung der Implementierungslogik einen negativen Einfluss auf die Geschwindigkeit mit der eine Webseite im Browser geladen wird. Diesen negativen Faktor muss der Benutzer jedoch nicht zwingendermaßen als Unterschied wahrnehmen. Der Flaschenhals in der Logik von PoisonProtect sind die Zugriffe auf die Datenbank. Im Bezug auf die Datenbank ist eine wei-

[4] https://www.kali.org

tere Auffälligkeit der erforderliche Speicherplatz. Nach der Installation benötigt die Anwendung schon circa 200 MB. So gut wie der komplette Speicherbedarf entsteht durch die Datenbank. Der Speicherbedarf könnte langfristig verbessert werden, wenn die Implementierung Datensätze löschen würde, auf die seit einem längeren Zeitraum nicht mehr zugegriffen wurde. Dazu würde die Information des letzten Zugriffs jedoch zusätzlich gespeichert werden müssen. Das Problem dabei wäre vor allem, dass bei jedem Lesevorgang eines Datensatzes dieser auch aktualisiert werden müsste. Diese Notwendigkeit würde die Performance in größerem Ausmaß negativ beeinflussen. Der Grund dafür ist, dass Daten einer Tabelle gleichzeitig von mehreren Anfragen gelesen werden können, jedoch zu einem Zeitpunkt nur eine Anfrage gewährt wird, die die Tabelle aktualisiert.
Eine weitere mögliche Verbesserung der Logik wäre, in dem *Array lastDistinctRequests* sowohl den Initiator einer Anfrage, sowie die angefragt Domäne zu speichern und beide Informationen verschieden zu verarbeiten. Eine weitere mögliche Verbesserung wäre, eine Logik zu implementieren die alte IP-Adressen einer Domäne aus dem Datensatz der Domäne löscht. Dafür wäre es jedoch notwendig in jedem Datensatz zu jeder IP-Adresse einen Zeitstempel der letzten Verwendung zu speichern. Aufgrund dieser Notwendigkeit wäre es aus Gründen der Performance und des erforderlichen Speicherplatzes möglicherweise nicht realisierbar. Im Zusammenhang mit dem vorigen Punkt wäre es auch ratsam es möglich zu machen, dass die Information, dass sich die Adresse eines Datensatzes häufig ändert, regelmäßig neu zu evaluieren und gegebenenfalls wieder zu entfernen. Der Grund dafür ist, dass diese Funktionalität einen negativen Einfluss auf Zuverlässigkeit Angriffe zu erkennen haben kann.
Anzumerken ist außerdem, dass diese Browsererweiterung lediglich vor der aktuellen Implementierung von Poisontap schützt. Es ist jedoch durchaus möglich die Implementierung von Poisontap so umzuschreiben, dass der Angriff durch die Browsererweiterung nicht erkannt wird. Das kann beispielsweise erreicht werden indem der Poisontap-Server bei der Antwort auf eine Anfrage eine für die angefragt Domäne korrekte IP-Addresse verwendet.

7 Related Work

Bestehende wissenschaftliche Arbeiten in Verbindung mit dieser Arbeit können wie folgt kategorisiert werden. Die gegebene Einteilung ist lediglich eine von verschiedenen Möglichkeiten, relevante Arbeiten zu gruppieren.

7.1 Schwachstellen und Angriffsszenarien

Auf Grund der hohen Nutzung des Internets und der komplexen Interaktion der Webanwendungen mit verschiedenen Komponenten und Protokollen, ergibt sich eine große Angriffsfläche. Daraus und auf Grund der Tatsache, dass immer mehr finanzielle Transaktionen über das Internet durchgeführt werden, gibt es eine große Anzahl an verschiedensten Angriffstechniken gegen Webanwendungen. Attacken auf NODE.js Server (ReDoS) sowie deren Erkennung werden in Staicu und Pradel, [65] analysiert. Da aus guten Gründen vermehrt auf die Verwendung des sicheren Protokolls HTTPS gesetzt wird, gerät das gezielte Unterbrechen dieser sicheren Verbindung mehr in den Vordergrund. Durumeric u. a., [14] widmete sich dieser Thematik. In Yaoqi u. a., [74] wird gezeigt wie bösartige Ressourcen, trotz der Verwendung von HTTPS, mit Hilfe einer Man-in-the-Middle-Attacke, im Browser Cache gespeichert werden können. Die Autoren des Texts geben ebenso Empfehlungen für verschiedene, involvierte Parteien um solche Angriffe zu verhindern. Chen u. a., [10] befasst sich mit dem Problem von gefährlichen Notlösungen (engl. *workarounds*) im Zusammenhang mit der SOP, die oft zu Zugriffen auf eigene Ressourcen, durch Clients denen nicht vertraut wird, führen. Deshalb werden in der erwähnten Arbeit Veränderungen der Browserimplementierungen sowie der Spezifikation von CORS vorgeschlagen. Eine sehr bekannte Angriffsart, welche bereits seit vielen Jahren in verschiedensten Formen existiert, ist XSS. Gupta und Gupta, [19] und Hydara u. a., [28] geben einen Überblick der verschiedenen Angriffsformen und deren Gegenmaßnahmen. Ein Problem, welches in so gut wie allen Programmiersprachen existiert, ist die Verwendung veralteter Softwarebibliotheken, wobei das Problem vermehrt in der Frontend-Programmierung auftritt, da dort üblicherweise mehr Bibliotheken verwendet werden. In Lauinger u. a., [35] wird näher auf dieses Thema eingegangen, sowie auf die Notwendigkeit besserer Software für das Management und die Wartung von Softwareabhängigkeiten. Schwarz, Lipp und Gruss, [60] befassen sich mit Side-channel attacken in JavaScript und schlagen ein verbessertes Modell für die Erlaubnis der Ausführung von JavaScript-Code durch den Browser, vor. Lian, Shacham und Savage, [36] zeigen, dass bösartiger Code in bereits kompilierten Code eingeschleust werden kann. Gezeigt wird das bei einem aktuellen Compiler des Browsers Chrome.

7.2 Sicherheitsmaßnahmen und Best Practices

Der Großteil der erwähnten Arbeiten beinhaltet bereits Empfehlungen, um den jeweiligen Sicherheitslücken vorzubeugen. Die in diesem Abschnitt ausgewählte Literatur jedoch setzt den Schwerpunkt vermehrt auf das Verhindern von Angriffen und geht auf Best Practices für verschiedene, in Softwareprojekten involvierte Akteuere, ein.
In Zalewski, [75], Liang, [37] und Vittie, [69] wird auf die Herausforderung Webanwendungen sicherer zu machen eingegangen. Diese Arbeiten bieten einen Überblick über die bedeutendsten Sicherheitsprobleme sowie deren Ursachen und Gegenmaßnahmen. In Deepa und Thilagam, [12] wird auf XSS- and SQL-injection, sowie Sicherheitslücken durch Fehler in der Programmlogik

eingegangen. Gupta und Gupta, [20] widmet sich ebenso Injection-Attacken. Der Fokus liegt bei Anwendungen die für die Frontend-Programmierung JavaScript und HTML5 verwenden. In der Arbeit wird ein Framework vorgestellt, welches in einer Browsererweiterung ausgeführt werden kann und Injektion-Attacken verhindern kann. Bhargavan u. a., [5] geht darauf ein wie TLS mit aktuellen Best-Practices umgangen werden kann. Als Folge stellen sie neue Best-Practice-Lösungen sowie Empfehlungen für Änderungen der Implementierung von TLS vor. In Maisuradze, Backes und Rossow, [40] geht es um *Constant-Blinding*. Das ist eine von Browsern angewandte Methode bei der Konstanten in kompiliertem JavaScript-Code, die von einem potentiellen Angreifer kontrolliert werden, entfernt werden. Um die Vollständigkeit dieser Sicherheitsmaßnahme zu testen, wird in der erwähnten Arbeit ein Fuzz testing-Framework vorgestellt. Auf Grund nicht zufriedenstellender, vorhandener Lösungen wird eine neue Lösung vorgeschlagen: die Verwendung einer Komponente (Web Proxy), welche Konstanten ersetzen und ungefährlich machen soll bevor die Ressourcen den Browser erreichen. Die grundlegende Logik des Codes soll dadurch nicht verändert werden.

7.3 Sicherheitstests

Sicherheitstests sind ein Thema in dem viel Forschung betrieben wird, da automatisierte Software zum Finden von Sicherheitslücken, Entwicklern und Testern viel Zeit ersparen kann. Außerdem könnten auch Schwachstellen gefunden werden, die zuvor nicht gefunden wurden. Ein Problem ist allerdings, dass der Prozess der Schwachstellensuche nicht sehr gut funktioniert Votipka u. a., [70]. Peroli u. a., [56] stellt ein Testframework vor welches die Lücke zwischen Penetrations-Tests und Modellbasierten Tests schließen soll. Hübler, [27] schlägt eine Lösung vor, die maschinelles Lernen verwendet um die Analyse und Evaluierung automatisierter Tests durchzuführen, da diese Phase, auf Grund der Datenmenge, sehr zeitaufwendig und fehleranfällig ist. Da Webanwendungen im Allgemeinen dynamischer geworden sind ist statische Codeanalyse nicht ausreichend für aktuelle Applikationen. Ein Ansatz ist die dynamische und statische Analyse miteinander zu verbinden Alhuzali u. a., [3]. Um die Testabdeckung zu erhöhen präsentiert Huang u. a., [26] einen Schwachstellenscanner für Webanwendungen, der automatisch Testdaten generiert und kombinatorische Verschleierungstechniken (engl. *combinative evasion techniques*) einsetzt um neue Sicherheitslücken zu finden. Staicu, Pradel und Livshits, [66] behandelt das serverseitige *NODE.js* und schlägt eine Technik vor durch die angreifbare Module zur Laufzeit sicher ausgeführt werden können. Das wird erreicht durch eine Kombination statischer Analyse und die Durchsetzung von Sicherheitsregeln. Melicher u. a., [43] konzentriert sich auf DOM XSS, eine spezielle Form von XSS die an Bedeutung gewinnt. Auch hier wird in verschiedenen Phasen dynamische und statische Analyse angewendet.

7.4 IT-Forensik

Wenn es um das Theam IT-Forensik geht, ist üblicherweise bereits ein Angriff passiert. Das bedeutet jedoch nicht, dass nichts getan werden muss. Es gibt Arten von Angriffen die über einen großen Zeitraum lang unentdeckt in einem System bleiben. Ma, Zhang und Xu, [38] benutzt ein Spurenverfolgungssystem (engl. *tracing system* das Logdaten und *Tainting* um nicht vertrauenswürdige verfolgen zu können. Die Autoren von Vadrevu u. a., [68] entwickelten einen Browser welcher Logging-Fähigkeiten verbessert. Bei wichtigen Interaktionen zwischen dem Benutzer und der aktuellen Webseite erstellt der Browser einen Screenshot der geänderten Seite und speichert diesen gemeinsam mit dem Zustand der Seite sowie dem aktuellen DOM-Baum.

7.5 Browsererweiterung

Um die Sicherheit von JavaScript Code, im Speziellen mit der Verwendung von externen Programmbibliotheken, zu erhöhen stellt Magazinius, Hedin und Sabelfeld, [39] eine Methode vor um im auszuführenden Code Sicherheitsüberprüfungen auszuführen bevor der Code ausgeführt wird. Eine Möglichkeit diese Funktionalität verfügbar zu machen ist mittels Browsererweiterungen. Heule u. a., [25] konzentrieren sich auf existierende Probleme und Gefahren die von Browsererweiterungen ausgehen.

Diese Arbeit versucht ein umfangreiches Themengebiet mit verschiedenen Sicherheitslücken von Webanwendungen abzudecken ohne sich dabei zu sehr in den technischen Details zu verlieren und bietet Informationen über gut bekannte Sicherheitsmaßnahmen als auch über solche die noch weniger bekannt sind. Es stellt eine Browsererweiterung vor welche die Schadsoftware Poisontap (Kamkar, [29]) an der Ausführung hindern kann.

8 Zusammenfassung

Derzeit Softwareentwickler mit der Tatsache konfrontiert, dass in Implementierungen bereits sehr bekannt Fehler gemacht werden die zu gravierenden Sicherheitslücken führen können. Das ist auf Faktoren wie Zeitmangel, fehlende Kompetenz, die steigende Komplexität der Anwendungen, fehlendes Verständnis der Wichtigkeit sicherheitsrelevanter Maßnahmen bei Entwicklern selbst oder Vorgesetzten, zurückzuführen. Darum hatte diese Arbeit als Ziel das Auftreten der häufigsten und schwerwiegendsten Fehler bei dynamischen Webanwendungen zu reduzieren. Dazu wurden zuerst die wichtigsten Angriffsarten und ihre Voraussetzungen vorgestellt, sowie anschließend auf ihre Gegenmaßnahmen eingegangen. Dabei wurde besonderer Wert auf die Verständlichkeit gelegt sowie auf die Notwendigkeit lediglich über Informatikgrundkenntnisse zu verfügen, gelegt.
In Abschnitt 9.1 befindet sich zudem eine kurze Aufstellung der erwähnten Sicherheitsmaßnahmen. Diese Liste soll als Kontrollmöglichkeit bei der Entwicklung von Webanwendungen dienen. Es ist jedoch anzumerken, dass die Liste nicht insofern vollständig ist, als dass sie alle möglichen relevanten Sicherheitsmaßnahmen enthält. Es handelt sich hierbei lediglich um eine Auswahl an wichtigen Maßnahmen. Durch das Zurverfügungstellen der Liste wird die Entwicklungszeit verbessert, da die wichtigsten Maßnahmen für die jeweiligen Schwachstellen bereits angeführt sind. Daher besteht für den Entwickler weniger die Notwendigkeit nach wichtigen Maßnahmen zu suchen.
In weiterer Folge wurde die Schadsoftware Poisontap vorgestellt welche unter anderem einen Browser-Cache Poisoning Angriff ausführt. Außerdem werden Cookies von Webseiten an den Server des Angreifers weitergeleitet und eine dauerhafte Websocket-Verbindung mit selbigem Server erstellt die der Angreifer nach dem eigentlichen Angriff weiterhin für Zugriffe und Manipulationen verwenden kann. Die dabei angegriffenen Webseiten sind die 1 Million am meisten aufgerufenen Webseiten laut Alexa des Unternehmens Amazon. Als Schutz vor dieser Schadsoftware wurde PoisonProtect, eine Browsererweiterung für den Browser Google Chrome entwickelt. PoisonProtect ist in der Lage die Verwendung von Poisontap zu erkennen und zu unterbinden. Die korrekte Funktionsweise von PoisonProtect wurde mit der testweisen Ausführung der von Poisontap verwendeten, für die Testausführung relevanten Befehle verifiziert. Mehr Informationen zu den durchgeführten Tests befinden sich in Kapitel 6.4.

Literatur

Wissenschaftliche Literatur

[1] Joint Technical Commitee ISO/IEC JTC 1. *Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 4: Management framework*. Techn. Ber. 7498-4. CH-1211 Genf, Schweiz: Joint Technical Commitee ISO/IEC JTC 1, Nov. 1989.

[2] Adida, Barth und Jackson. "Rootkits for JavaScript environments". In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. WOOT'09. Berkeley, CA, USA: USENIX Association, 2009, S. 4–4.

[3] Alhuzali u. a. "NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, S. 377–392.

[5] Bhargavan u. a. "Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS". In: *2014 IEEE Symposium on Security and Privacy*. Mai 2014, S. 98–113.

[6] Braden. *Requirements for Internet Hosts - Application and Support*. Request for Comments 1123. Internet Engineering Task Force, Network Working Group, Okt. 1989.

[7] Butkiewicz, Madhyastha und Sekar. "Understanding Website Complexity: Measurements, Metrics, and Implications". In: *2011 ACM SIGCOMM conferenc on Internet measurement conference*. Nov. 2011.

[8] Carpenter und Brim. *Middleboxes: Taxonomy and Issues*. Request for Comments. The Internet Society, Feb. 2002.

[10] Chen u. a. "We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, S. 1079–1093.

[12] Deepa und Santhi Thilagam. "Securing web applications from injection and logic vulnerabilities: Approaches and challenges". In: *Information and Software Technology* 74 (2016), S. 160–180.

[13] Dierks und Allen. *The TLS Protocol Version 1.0*. Request for Comments. The Internet Society, 1999.

[14] Durumeric u. a. "The Security Impact of HTTPS Interception". In: *Network and Distributed System Security Symposium*. Jan. 2017.

[16] Fette und Melnikov. *The WebSocket Protocol*. Techn. Ber. Internet Engineering Task Force (IETF), Dez. 2011.

[17] Fielding, Nottingham und Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. Request for Comments. Internet Engineering Task Force, 2014.

[18] Gettys u. a. *Hypertext Transfer Protocol – HTTP/1.1*. Request for Comments. The Internet Society, Network Working Group, Juni 1999.

[19] Gupta und Gupta. "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art". In: *International Journal of System Assurance Engineering and Management* 8.1 (Jan. 2017), S. 512–530.

[20] Gupta und Gupta. “JS-SAN: defense mechanism for HTML5-based web applications against javascript code injection vulnerabilities”. In: *Security and Communication Networks* 9.11 (2016), S. 1477–1495.

[24] Halfond, Viegas und Orso. “A classification of SQL-injection attacks and countermeasures”. In: *International Symposium on Secure Software Engineering (ISSSE 2006)*. Jan. 2006.

[25] Heule u. a. “The Most Dangerous Code in the Browser”. In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015.

[26] Huang u. a. “Web Application Security: Threats, Countermeasures, and Pitfalls”. In: *Computer* 50.6 (2017), S. 81–85.

[27] Hübler. *Automated Detection of Security Vulnerabilites Using Machine Learning for Automated Testing*. 2014.

[28] Isatou Hydara u. a. “Current state of research on cross-site scripting (XSS) – A systematic literature review”. In: *Information and Software Technology* (2015), S. 170 –186.

[31] Kirrage, Rathnayake und Thielecke. “Static Analysis for Regular Expression Denial-of-Service Attacks”. In: *Network and System Security*. Hrsg. von Javier Lopez, Xinyi Huang und Ravi Sandhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 135–148.

[32] Kocher. “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In: *Koblitz N. (eds) Advances in Cryptology — CRYPTO ’96*. Bd. 1109. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, S. 104–113.

[34] Kristol und Montulli. *HTTP State Management Mechanism*. Techn. Ber. 2109. Internet Engineering Task Force, Network Working Group, Feb. 1997.

[35] Lauinger u. a. “Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web”. In: *CoRR* (2017).

[36] Lian, Shacham und Savage. “A Call to ARMs: Understanding the Costs and Benefits of JIT Spraying Mitigations”. In: *The Network and Distributed System Security Symposium*. Jan. 2017.

[37] Liang. *JavaScript Security Learn JavaScript security to make your web applications more secure*. Packt Publishing Ltd., 2014.

[38] Ma, Zhang und Xu. “ProTracer: Towards Practical Provenance Tracing by Alternating Between Logging and Tainting”. In: *Network and Distributed System Security Symposium*. Jan. 2016.

[39] Magazinius, Hedin und Sabelfeld. “Architectures for Inlining Security Monitors in Web Applications”. In: *Engineering Secure Software and Systems*. Hrsg. von Jan Jürjens, Frank Piessens und Nataliia Bielova. Springer International Publishing, 2014, S. 141–160.

[40] Maisuradze, Backes und Rossow. “Dachshund: Digging for and Securing (Non-)Blinded Constants in JIT Code”. In: *Network and Distributed System Security Symposium*. Jan. 2017.

[42] Mehrnezhad u. a. “TouchSignatures: Identification of User Touch Actions and PINs Based on Mobile Sensor Data via JavaScript”. In: *CoRR* abs/1602.04115 (2016).

[43] Melicher u. a. “Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting”. In: *Network and Distributed System Security Symposium*. Jan. 2018.

[44] Miller, Fredriksen und So. “An Empirical Study of the Reliability of UNIX Utilities”. In: *Commun. ACM* 33.12 (Dez. 1990), S. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[45] Milner. “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), S. 348 –375. ISSN: 0022-0000. DOI: https://doi.org/10.1016/0022-0000(78)90014-4. URL: http://www.sciencedirect.com/science/article/pii/0022000078900144.

[46] Mockapetris und Dunlap. “Development of the Domain Name System”. In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM ’88. ACM SIGCOMM. New York, NY, USA: ACM, 1988, S. 123–133.

[47] Modadugu und Rescorla. “The Design and Implementation of Datagram TLS”. In: (Dez. 2003).

[48] Muñoz und Mirosh. “Friday the 13th JSON Attacks”. In: *BlackHat Conferenc 2017*. Juli 2017.

[49] Netzer und Miller. “What Are Race Conditions? Some Issues and Formalizations”. In: *ACM Lett. Program. Lang. Syst.* 1.1 (März 1992), 74–88. ISSN: 1057-4514. DOI: 10.1145/130616.130623. URL: https://doi.org/10.1145/130616.130623.

[56] Peroli u. a. “MobSTer: A model-based security testing framework for web applications”. In: *Software Testing, Verification and Reliability* 28.8 (2018), e1685.

[58] Ross und Gondrom. *HTTP Header Field X-Frame-Options*. Techn. Ber. Internet Engineering Task Force (IETF), Okt. 2013.

[59] Santos u. a. “Implementation State of HSTS and HPKP in Both Browsers and Servers”. In: Nov. 2016, S. 192–207. ISBN: 978-3-319-48964-3. DOI: 10.1007/978-3-319-48965-0_12.

[60] Schwarz, Lipp und Gruss. “JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks”. In: *Network and Distributed System Security Symposium*. Jan. 2018.

[61] Shah und Mehtre. “An overview of vulnerability assessment and penetration testing techniques”. In: *Journal of Computer Virology and Hacking Techniques* 11 (Feb. 2014), S. 27–49. DOI: 10.1007/s11416-014-0231-x.

[62] Marc Shapiro. “Structure and Encapsulation in Distributed Systems: the Proxy Principle”. In: *Int. Conf. on Distr. Comp. Sys. (ICDCS)*. Int. Conf. on Distr. Comp. Sys. (ICDCS). IEEE. Cambridge, MA, United States, 1986, S. 198–204. URL: https://hal.inria.fr/inria-00444651.

[64] Spafford. “The Internet Worm Program: An Analysis”. In: *SIGCOMM Comput. Commun. Rev.* 19.1 (Jan. 1989), 17–57. ISSN: 0146-4833. DOI: 10.1145/66093.66095. URL: https://doi.org/10.1145/66093.66095.

[65] Staicu und Pradel. “Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, S. 361–376.

[66] Staicu, Pradel und Livshits. “SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS”. In: *Network and Distributed System Security Symposium*. Jan. 2018.

[67] Tripathi, Swarnkar und Hubballi. “DNS Spoofing in Local Networks Made Easy”. In: *IEEE International Conference on Advanced Networks and Telecommunications Systems*. Okt. 2017. DOI: 10.1109/ANTS.2017.8384122.

[68] Vadrevu u. a. “Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots”. In: *Network and Distributed System Security Symposium*. 2017.

[69] Mac Vittie. *Web Application Security is a Stack : How to CYA (Cover Your Apps) Completely*. IT Governance Publishing, 2015.

[70] Votipka u. a. “Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. Bd. 00. Mai 2018, S. 374–391.

[71] Weichselbaum u. a. “CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy”. In: *23rd ACM Conference on Computer and Communications Security*. Juli 2016.

[73] Wüstholz u. a. “Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions”. In: *TACAS*. 2017.

[74] Yaoqi u. a. “Man-in-the-browser-cache: Persisting HTTPS attacks via browser cache poisoning”. In: *Computers and Security* 55 (2015), S. 62 –80.

[75] Zalewski. *The Tangled Web - A Guide to Securing Modern Web Applications*. Hrsg. von William Pollock. Pollock, William, 2012.

Online-Referenzen

[4] ASF Intrabot Apache Software Foundation. 2019. URL: https://cwiki.apache.org/confluence/display/HTTPD/RedirectSSL.

[9] CERN-ComputerSecurityTeam. 2019. URL: https://security.web.cern.ch/security/recommen\-dations/en/codetools/rats.shtml.

[11] Chromium-Team. 2019. URL: https://hstspreload.org.

[15] Remy van Elst. 2016. URL: https://raymii.org/s/tutorials/HTTP_Strict_Transport_Security_for_Apache_NGINX_and_Lighttpd.html.

[21] Guyer u. a. März 2017. URL: https://docs.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-executesql-transact-sql?view=sql-server-ver15.

[22] Guyer u. a. Sep. 2017. URL: https://docs.microsoft.com/en-us/sql/t-sql/statements/create-procedure-transact-sql?view=sql-server-ver15.

[23] Guyer u. a. Aug. 2019. URL: https://docs.microsoft.com/en-us/sql/connect/jdbc/data-source-sample?view=sql-server-ver15.

[29] Kamkar. *PoisonTap - siphons cookies, exposes internal router and installs web back- door on locked computers.* Feb. 2019. URL: https://samy.pl/poisontap/ (besucht am 16. 11. 2016).

[30] Van Kesteren. *Cross-Origin Resource Sharing*. Jan. 2014. URL: https://www.w3.org/TR/cors/.

[33] Koppers und Webpack contributors. 2020. URL: https://github.com/webpack/webpack.

[41] MDN-Contributors. 2019. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity.

[50] Oracle. *Matcher (Java Platform SE 7)*. 2018. URL: https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html.

[51] OWASP-Foundation. Juni 2019. URL: https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

[52] OWASP-Foundation. 2019. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html.

[53] OWASP-Foundation. 2019. URL: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[54] OWASP-Foundation. *OWASP Java Encoder*. 2020. URL: https://www.owasp.org/index.php/OWASP_Java_Encoder_Project.

[55] OWASP-Foundation. *OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks*. Okt. 2017. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[57] Rapid7-LLC. 2016. URL: https://www.rapid7.com/products/metasploit/download/editions/.

[63] SOURCEFORGE. Juni 2015. URL: http://findbugs.sourceforge.net/bugDescriptions.html.

[72] WHATWG. Juni 2010. URL: https://html.spec.whatwg.org/multipage/workers.html#workers.

9 Anhang

9.1 Checkliste Sicherheitsmaßnahmen

◯ SQL-Injetions: Verwendung von Stored Procedures; keine dynamischen Abfrage

◯ XSS: Inputvalidierung mit Whitelists statt Blacklists; nur die notwendigsten Zeichen erlauben

◯ XSS: Verwendung von Entity Encoding

◯ XSS: Escaping mit einer dafür gedachten Biblbiothek

◯ Cookies: Setzen der Cookie-Header *HttpOnly* und *Secure*

◯ HTTP zu HTTPS Weiterleitung

◯ Verwendung von HSTS

◯ Anwendung einer CSP ohne Whitelists sondern mit Nonces

◯ Verwendung von SRI

◯ Anwendung einer CSP; im Header weder einen Wildcard-Wert oder *null* angeben; beim dynamischen Setzen des Headers Unter-Domäneen nicht vertrauen

◯ Verwendung eines CSRF-Tokens mit Hilfe einer dafür gedachten Bibliothek

9.2 manifest.json

```
{
  "name": "PoisonProtect",
  "version": "1.0",
  "description": "Protects from the Poisontap attack",
  "manifest_version": 2,
  "permissions": ["unlimitedStorage", "webRequest", "
  ↪ webRequestBlocking", "<all_urls>"],
  "background": {
    "scripts": ["background.js"],
    "persistent": true
  },
  "browser_action": {
    "default_title": "Reset to non-blocking"
  }
}
```

9.3 background.js

```
async function initializeDataBase(){
    let objStore = database.createObjectStore("hosts", {keyPath: "hostName"
    ↪ });
    createdIndex = await objStore.createIndex("lastModified", "lastModified"
    ↪ , {unique:false});
    printMessage("start data storage procedure");
    loadJSON(loadJsonCallback, 'dnsData0.json', 0);
}

function loadJsonCallback(data, i){
    let entryList = Object.entries(data);
    let transaction = database.transaction(["hosts"], "readwrite").
    ↪ objectStore("hosts");
    for(let j = 0; j < entryList.length; j++){
        let newHostData = {
            "hostName": entryList[j][0],
            "ips": entryList[j][1].ips,
            "lastModified": new Date()
        };
        let callback = null;
        if(j === entryList.length - 1){
            callback = function () {
                currentDBSize += entryList.length;
                if(i + 1 === filesToLoad){
                    maxDBSize = currentDBSize * 1.1;
                    printMessage("database update complete");
                    setWebRequestListeners();
                } else {
                    loadJSON(loadJsonCallback, "dnsData" + ++i + ".json", i)
                    ↪ ;
                }
            }
        }
        saveHostData(newHostData, transaction, callback, false);
    }
}

```

```
34 function getHostData(hostname, details, callback) {
35     checkTemporaryData(function(hostname, details, callback){
36         let request = database.transaction(["hosts"]).objectStore("hosts").
           ↪ get(hostname);
37         request.onsuccess = function(event){
38             if(event.target.result === undefined){
39                 callback(hostname, details);
40             } else {
41                 callback(event.target.result, details);
42             }
43         };
44         request.onerror = function(){
45             callback(null);
46         }
47     }, hostname, details, callback);
48 }
49
50 function saveHostData(hostData, objStore = null, callback = null,
   ↪ loggingEnabled = true){
51     checkTemporaryData(function(hostData, objStore = null, callback = null){
52         if(objStore === null){
53             objStore = database.transaction(["hosts"], "readwrite").
               ↪ objectStore("hosts");
54         }
55         let request = objStore.put(hostData);
56         request.onsuccess = function(){
57             if (loggingEnabled) {
58                 printMessage('saved hostdata with key - ' + hostData.
                   ↪ hostName);
59             }
60             if (callback !== null){
61                 callback();
62             }
63             if(++currentDBSize >= maxDBSize) { //delete oldest entry
64                 deleteOldestEntry(objStore);
65             }
66         };
67         if (callback !== null){
68             request.onerror = function(){
69                 callback();
70             }
71         }
72     }, hostData, objStore, callback);
73 }
74
75 function deleteOldestEntry(objStore){
76     let cursorRequest = createdIndex.openCursor(null, "prev");
77     cursorRequest.onsuccess = function(event) {
78         let cursor = event.target.result;
79         let deleteRequest = objStore.delete(cursor.key);
80         deleteRequest.onsuccess = function(){
81             currentDBSize--;
82             printMessage('Deleted oldest entry, key was: ' + cursor.key);
83         }
84     }
85 }
86
87 function loadJSON(callback, fileName, i) {
88     let request = new XMLHttpRequest();
89     request.overrideMimeType("application/json");
90     request.open('GET', "./" + fileName, true);
91     request.onreadystatechange = function () {
```

```
92          if (request.readyState === 4 && request.status === 200) {
93              callback(JSON.parse(request.responseText), i);
94          }
95      };
96      request.send(null);
97  }
98
99  function getMaxPercentageOfWrongIPRequests(requestNumber){
100     if(requestNumber > 16){
101         return detectWrongIPsThreshold;
102     }
103     let startValue = 0.5;
104     if(requestNumber > 2){
105         startValue -= (requestNumber * 0.25).toFixed(2);
106     }
107     return startValue;
108 }
109
110 function checkSpoofingValueForUpdate(requestWithWrongIP = false) {
111     let spoofingDetectedOld = spoofingDetected;
112     let additionalInfo = '';
113     let updateRequestCount = 0, nonUpdateRequestCount = 0;
114     if(lastDistinctRequests.length > 0) {
115         updateRequestCount = lastDistinctRequests.filter(request => request.
            ↪ ipUpdate).length;
116         nonUpdateRequestCount = lastDistinctRequests.length -
            ↪ updateRequestCount;
117     }
118     if(!requestWithWrongIP){
119         if (spoofingDetected && updateRequestCount === 0) { //spoofing has
            ↪ been detected but no suspicious request has been made lately
120             spoofingDetected = false;
121             additionalInfo += 'no updates have been made recently';
122         }
123     } else {
124         if(updateRequestCount > maxNumOfInitialWrongIPs &&
            ↪ nonUpdateRequestCount === 0){ //only update requests
125             spoofingDetected = true;
126             additionalInfo += 'only updateRequests have been made without
                ↪ any requests with correct IP';
127         }
128         else if(updateRequestCount > maxNumOfInitialWrongIPs &&
129             updateRequestCount > lastDistinctRequests.length *
                ↪ getMaxPercentageOfWrongIPRequests(lastDistinctRequests.length
                ↪ )) {
130             spoofingDetected = true;
131             additionalInfo += 'too many updates have been made recently';
132         }
133     }
134     if(spoofingDetectedOld !== spoofingDetected){
135         enableExtensionPopUp(spoofingDetected);
136         printMessage('spoofingdetected: ' + spoofingDetected + '; ' +
            ↪ additionalInfo, logLevel.warning);
137     }
138 }
139
140 function verifyDomainIP(hostInfo, details) {
141     if(hostInfo === null){
142         return;
143     }
144     let requestLogObject = {
145         "ipUpdate": !spoofingDetected,
```

```
146             "timestamp": new Date()
147         };
148         if(typeof hostInfo === 'string') { //hostInformation not present in the
            ↪ DB; block unknown origins when spoofingDetected
149             requestLogObject.hostName = getDomainFromUrl(details);
150             checkSpoofingValueForUpdate();
151             printMessage('hostname ' + hostInfo + ' not present in the database'
                ↪ , logLevel.warning);
152             if(!spoofingDetected){
153                 printMessage('add hostObject ' + JSON.stringify(hostInfo) + ' -
                    ↪ ' + details.ip + ' to the database');
154                 saveHostData({
155                     "hostName": hostInfo,
156                     "ips": [details],
157                     "lastModified": new Date()
158                 });
159             }
160         } else if(typeof hostInfo === 'object') {
161             requestLogObject.hostName = getDomainFromUrl(details);
162             if(hostInfo.ips.includes(details.ip) || (hostInfo.frequentUpdates
                ↪ !== undefined && hostInfo.frequentUpdates)){
163                 requestLogObject.ipUpdate = false;
164                 let message = hostInfo.ips.includes(details.ip) ? 'hostname ' +
                    ↪ hostInfo.hostName + ' present in the database and IP (' +
                    ↪ details.ip + ') correct' :
165                     'hostname ' + hostInfo.hostName + ' present in the database;
                        ↪  ignored wrong IP (' + details.ip + ') due to regularly
                        ↪ frequent updates';
166                 printMessage(message, hostInfo.ips.includes(details.ip) ?
                    ↪ logLevel.info : logLevel.warning);
167                 checkSpoofingValueForUpdate();
168             } else {
169                 //update if no spoofing detected
170                 checkSpoofingValueForUpdate(true);
171                 printMessage('hostname ' + hostInfo.hostName + ' present in the
                    ↪ database and IP (' + details.ip + ') NOT correct', logLevel.
                    ↪ warning);
172                 if(!spoofingDetected) {
173                     hostInfo.ips.push(details.ip);
174                     printMessage('add IP ' + details.ip + ' to the ipList of
                        ↪ host ' + hostInfo.hostName);
175                     let timeStampArray = domainUpdatesCount[requestLogObject.
                        ↪ hostName] || [];
176                     timeStampArray.push(new Date());
177                     domainUpdatesCount[requestLogObject.hostName] =
                        ↪ timeStampArray
178                         .filter(timestamp => diffMinutes(timestamp, new Date())
                            ↪ <= requestsSaveDurationMinutes);
179                     if (domainUpdatesCount[requestLogObject.hostName].length >=
                        ↪ 2) {
180                         hostInfo.frequentUpdates = true;
181                     }
182                     saveHostData(hostInfo);
183                 }
184             }
185         }
186         lastDistinctRequests = lastDistinctRequests.filter(request =>
            ↪ diffMinutes(request.timestamp, new Date()) <=
            ↪ requestsSaveDurationMinutes);
187         addRequestToTempStorage(requestLogObject);
188     }
189
```

```
190 function addRequestToTempStorage(requestLogObject){
191     if (requestLogObject.hostName === undefined){
192         printMessage('tried to add invalid object to tmpStorage (hostName is
            ↪  missing) - ' + JSON.stringify(requestLogObject), logLevel.error)
            ↪ ;
193     } else {
194         if (requestLogObject.hostName.endsWith('.www')){
195             requestLogObject.hostName = requestLogObject.hostName.replace('.
                ↪ www','');
196         }
197         let hostNameIndex = lastDistinctRequests.findIndex(request =>
            ↪ request.hostName === requestLogObject.hostName);
198         if(hostNameIndex !== -1){
199             let oldSameObject = lastDistinctRequests[hostNameIndex];
200             if((oldSameObject.ipUpdate && !requestLogObject.ipUpdate) ||
201                 JSON.stringify(oldSameObject) === JSON.stringify(
                    ↪ requestLogObject)){
202                 return;
203             }
204             lastDistinctRequests.splice(hostNameIndex, 1);
205         }
206         lastDistinctRequests.push(requestLogObject);
207         lastDistinctRequests.sort();
208         hostNameIndex = lastDistinctRequests.indexOf(requestLogObject);
209         removeLowerPriorityObject(hostNameIndex + 1, hostNameIndex,
            ↪ requestLogObject);
210         removeLowerPriorityObject(hostNameIndex - 1, hostNameIndex,
            ↪ requestLogObject);
211     }
212 }
213
214 function removeLowerPriorityObject(oldIndex, newIndex, requestLogObject){
215     let oldObject = lastDistinctRequests[oldIndex];
216     if(oldObject === undefined || oldObject === requestLogObject){ //index
        ↪ out of bounds
217         return;
218     }
219     if(oldObject.hostName.includes(requestLogObject.hostName)){
220         if(oldObject.ipUpdate){
221             lastDistinctRequests.splice(newIndex, 1);
222         } else {
223             lastDistinctRequests.splice(oldIndex, 1);
224         }
225     }
226 }
227
228 function diffMinutes(timeStamp1, timeStamp2){
229     let diffMilliSeconds = 0;
230     if(timeStamp1 < timeStamp2) {
231         diffMilliSeconds = timeStamp2 - timeStamp1;
232     } else if(timeStamp2 < timeStamp1) {
233         diffMilliSeconds = timeStamp1 - timeStamp2;
234     }
235     return diffMilliSeconds / 60000;
236 }
237
238 function beforeRequestCallback(){
239     lastDistinctRequests = lastDistinctRequests.filter(request =>
        ↪ diffMinutes(request.timestamp, new Date()) <=
        ↪ requestsSaveDurationMinutes);
240     checkSpoofingValueForUpdate();
241     return {
```

```
242         cancel: spoofingDetected
243     };
244 }
245
246 function getDomainFromUrl(details){
247     let url = details.initiatior == undefined || details.initiatior == null
    ↪ ? details.url : details.initiatior;
248     try{
249         return new URL(url).hostname.split("").reverse().join("");
250     } catch (exception) {
251         printMessage("could not url for " + url + '; reason: ' + exception.
        ↪ message, logLevel.error);
252         return details.url;
253     }
254 }
255
256 function responseStartedCallback(details){
257     if(!spoofingDetected){
258         let hostname = getDomainFromUrl(details);
259         //test for hostname validity and that the returned IP is an Ipv4
        ↪ address as Ipv6 is not supported
260         if(!hostname.includes(".") || !ipv4Regex.test(details.ip)){
261             return;
262         }
263         getHostData(hostname, details, verifyDomainIP);
264     }
265 }
266
267 function printMessage(message, messageLogLevel = logLevel.info){
268     if(lastLogMessage !== undefined && lastLogMessage === message){
269         return;
270     }
271     lastLogMessage = message;
272     message += " " + new Date().toLocaleString();
273     if(messageLogLevel === logLevel.error){
274         console.error(message);
275     } else if(messageLogLevel === logLevel.warning){
276         console.warn(message);
277     } else {
278         console.log(message);
279     }
280 }
281
282 function checkAndCallFunction(functionParameter, args){
283     if(functionParameter !== null && typeof functionParameter === 'function'
    ↪ ){
284         functionParameter(...args);
285     }
286 }
287
288 function resetSpoofingDetected(){
289     console.clear();
290     if(spoofingDetected) {
291         printMessage('reset spoofing state');
292         spoofingDetected = false;
293         enableExtensionPopUp(false);
294         lastDistinctRequests = lastDistinctRequests.filter(request =>
        ↪ diffMinutes(request.timestamp, new Date()) <=
        ↪ requestsSaveDurationMinutes);
295         for(let i = 0; i < lastDistinctRequests.length; i++){
296             lastDistinctRequests[i].ipUpdate = false;
297         }
```

```
298         }
299     }
300
301     function enableExtensionPopUp(enable){
302         printMessage((enable ? 'enable' : 'disable') + ' extension browserAction
            ↪ ');
303         if(enable){
304             chrome.browserAction.enable();
305         } else {
306             chrome.browserAction.disable();
307         }
308     }
309
310     function setWebRequestListeners(){
311         chrome.webRequest.onBeforeRequest.addListener(beforeRequestCallback,
            ↪ urlFilter, beforeRequestOptions);
312         chrome.webRequest.onResponseStarted.addListener(responseStartedCallback,
            ↪  urlFilter, headersReceivedOptions);
313     }
314
315     function checkTemporaryData(callback = null, ...args){
316         if(database === undefined){
317             chrome.browserAction.onClicked.addListener(resetSpoofingDetected);
318             enableExtensionPopUp(false);
319             spoofingDetected = false;
320             lastDistinctRequests = [];
321             domainUpdatesCount = {};
322             let openDBRequest = indexedDB.open("dnsDatabase", 1);
323             openDBRequest.onblocked = function(){
324                 printMessage("Please close all other tabs with this site open!",
                    ↪  logLevel.warning);
325             };
326             openDBRequest.onsuccess = function(event){
327                 database = event.target.result;
328                 let objStore = database.transaction(["hosts"]).objectStore("
                    ↪ hosts");
329                 let dbSizeRequest = objStore.count();
330                 dbSizeRequest.onsuccess = function(){
331                     currentDBSize = dbSizeRequest.result;
332                     if(currentDBSize > 0){ //db has been created already
333                         maxDBSize = currentDBSize * 1.1;
334                         setWebRequestListeners();
335                     }
336                     try{
337                         createdIndex = objStore.index("lastModified");
338                     } catch (exception) {} //data and index not created yet
339                     checkAndCallFunction(callback, args);
340                 };
341             };
342             openDBRequest.onupgradeneeded = function(event){
343                 printMessage("update database");
344                 database = event.target.result;
345                 database.onerror = function(event){
346                     printMessage("Database error: " + event.target.errorCode,
                        ↪ logLevel.error);
347                 };
348                 initializeDataBase();
349             };
350         } else {
351             checkAndCallFunction(callback, args);
352         }
353     }
```

```
354
355 let database, createdIndex, lastDistinctRequests, domainUpdatesCount;
356 let spoofingDetected; //will be true if certain amount of request has non
    ↪ listed IPs
357 let currentDBSize = 0, maxDBSize = Number.MAX_SAFE_INTEGER, lastLogMessage;
358 const ipv4Regex = /^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}$/;
359 const requestsSaveDurationMinutes = 8, detectWrongIPsThreshold = 0.2,
    ↪ maxNumOfInitialWrongIPs = 5;
360 const logLevel = {
361     info: 0,
362     warning: 1,
363     error: 2
364 };
365 const filesToLoad = 10;
366
367 const urlFilter = {
368     urls: ["<all_urls>"]
369 };
370 const headersReceivedOptions = ["extraHeaders"];
371 const beforeRequestOptions = ["blocking"];
372 checkTemporaryData();
```

9.4 findTop1MillionIPs.py

```
1  import socket
2  import concurrent.futures
3  import threading
4  from datetime import datetime
5
6
7  def resolveLine(hostname):
8      global lineCount, start, resolved, unresolved
9      if "." not in hostname:
10         unresolved.append(hostname)
11         return
12     try:
13         thread_local.addressInfo = socket.gethostbyname_ex(hostname)
14         thread_local.ipList = '", "'.join(thread_local.addressInfo[2]).strip
           ↪ ()
15         thread_local.addresses = thread_local.addressInfo[1]
16         thread_local.hostValue = f'{{"ips": ["{thread_local.ipList}"]}}'
17         resolved.append(f'"{hostname[::-1]}":{thread_local.hostValue}') #
           ↪ reverse the domain name before appending it
18     except (socket.gaierror, socket.herror):
19         unresolved.append(hostname)
20     lineCount += 1
21     if lineCount % 1000 == 0:
22         thread_local.checkpoint = datetime.now() - start
23         print(f'{lineCount / 10000}% {int(thread_local.checkpoint.seconds /
           ↪ 60):02d}'
24               f':{thread_local.checkpoint.seconds % 60:02d}')
25
26
27 def internetConnection():
28     try:
29         socket.gethostbyname('google.com')
30         return True
31     except:
32         return False
33
34
35 if internetConnection():
```

```
36      file = []
37      resolved = []
38      unresolved = []
39      with open('top-1m.csv', 'r') as inputFile:
40          for row in inputFile:
41              row = row.strip().split(',')
42              hostname = row[len(row) - 1]
43              file.append(hostname)
44      outputFileName = f'dnsData{int(10 - (len(file) - 1) / 100000)}.json'
45      with open('top-1m.csv', 'w') as outputFile:
46          for i in range(len(file) - 1, -1, -1):
47              if i >= 100000:
48                  outputFile.write(file.pop() + "\n")
49
50      print('read done')
51      start = datetime.now()
52      thread_local = threading.local()
53      lineCount = 0
54
55      with concurrent.futures.ThreadPoolExecutor(max_workers=64) as pool:
56          results = list(pool.map(resolveLine, file))
57
58      print(f'resolved {len(resolved)} hosts')
59      if len(resolved) > 0:
60          print(f'could not resolve {len(unresolved)} hosts: {unresolved}')
61
62      with open(outputFileName, 'w+') as outputFile:
63          outputFile.write('{')
64          lineCount = 0
65          for element in resolved:
66              outputFile.write(element.replace("'", '"'))
67              lineCount += 1
68              if lineCount < len(resolved):
69                  outputFile.write(',')
70          outputFile.write('}')
71  else:
72      print("no internet connection!")
```