Ausgabe 01/2020

Java aktuell

Java aktuell

iJUG Verbund
www.ijug.eu

Sicherheit
Schützen Sie Ihre Anwendung vor unbekannten Bedrohungen

DevOps
Worauf es bei einer erfolgreichen DevOps-Kultur ankommt

GraalVM
Der heilige Gral der Compiler?

Immer auf der sicheren Seite

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 01


ORAWORLD
Das e-Magazine für alle Oracle-Anwender!
EOUC
EMEA ORACLE USERGROUP COMMUNITY
• Spannende Geschichten aus der Oracle-Welt
• Technologische Hintergrundartikel
• Leben und Arbeiten heute und morgen
• Einblicke in andere User Groups weltweit
• Neues (und Altes) aus der Welt der Nerds
• Comics, Fun Facts und Infografiken
Jetzt Artikel einreichen oder Thema vorschlagen!
Jetzt e-Magazine herunterladen
www.oraworld.org

Beyond OWASP Top 10 – Unbekanntere Arten von Schwachstellen in Web-anwendungen und APIs

Frank Ully, Oneconsult Deutschland GmbH

Auch wenn bei der Anwendungsentwicklung grundlegende Sicherheitslücken wie Cross-Site-Scripting (XSS) oder SQL-Injections vermieden werden, sind Webapplikationen und Schnittstellen anfällig für teilweise kuriose Schwachstellen. Dieser Artikel stellt ausgewählte, unbekanntere Schwachstellenarten vor, die ein Penetration Tester in der Berufspraxis findet, und gibt Hinweise auf gute Quellen zur Vertiefung der Inhalte.

Sicherheit im Web muss beim Entwerfen und Entwickeln von Webanwendungen und Schnittstellen von Anfang an berücksichtigt werden – dieses Bewusstsein und das notwendige Wissen werden jedoch in Studium und Kursen unzureichend vermittelt, wovon zahlreiche Schwachstellen in verbreiteten Anwendungen zeugen.

Im Dienste der Websicherheit – Open Web Application Security Project (OWASP)

Das Open Web Application Security Project (OWASP) *[1]* ist eine Non-Profit-Organisation, die sich auf die Fahnen geschrieben hat, die Sicherheit von Webanwendungen zu verbessern. Dazu veröffentlichen die zahlreichen Freiwilligen, die zum Projekt beitragen, unter anderem technische Dokumentationen, Softwarebibliotheken und Testwerkzeuge – allesamt kostenfrei nutzbar, unter freien Lizenzen wie Creative Commons, und sowohl bei Dokumenten als auch bei Software in bearbeitbaren Formaten beziehungsweise im Quelltext verfügbar.

Ihre wohl bekannteste Publikation ist die „OWASP Top 10", eine Aufzählung der zehn kritischsten Sicherheitsrisiken in Webanwendungen *[2]*. Die Liste wurde erstmals im Jahr 2003 veröffentlicht, zuletzt 2017 angepasst und ist vielen Entwicklern ein Begriff. Die nächste Aktualisierung der Top 10 steht im Herbst 2020 an *[3]*.

Die „OWASP Top 10" stellt zehn Angriffsarten auf Webanwendungen vor, ihre Ursachen und welche Maßnahmen bei der Entwicklung dagegen helfen. Die Liste beantwortet unter anderem folgende Fragen: Was muss ein Entwickler beachten, wenn er Mechanismen zur Authentifizierung und Autorisierung implementiert? Warum sind Eingabevalidierung und Ausgabekodierung wichtige Grundlagen von sicheren Webanwendungen? Inwiefern kann eine Anwendung von Injektionsschwachstellen betroffen sein? Warum sollte man beim Entwickeln auf verlässliche Komponenten von Dritten achten und darauf, dass die Anwendung möglicherweise sicherheitsrelevante Aktionen loggt?

Manche Anbieter gehen sogar so weit, ihre Software als „sicher" zu bezeichnen, weil sie „gemäß den OWASP Top 10" entwickelt worden sei; nicht immer seriöse Sicherheitsdienstleister bieten Tests nur „gegen die OWASP Top 10" an und stellen danach der Anwendung ein „Sicherheitszertifikat" aus.

Das ist jedoch unaufrichtig, denn die Top 10 sind nach eigener Zielsetzung kein ausführlicher Sicherheitsstandard *[4]*. Die Top-10-Verantwortlichen selbst sehen ihre Liste als Awareness-Dokument, das einem breiten Publikum von Informationssicherheitsverantwortlichen über Softwareprojektmanager und Entwickler bis zu Testern grundlegende Schwachstellen zugänglich machen soll. Deren Ursachen sollten auf jeden Fall in Webanwendungen vermieden werden – denn das Fehlen von Sicherheitsmaßnahmen dagegen grenzt an Fahrlässigkeit.

Wie festgestellt, können die Top 10 also nicht alle Schwachstellenarten abdecken, die bei modernen Webanwendungen und Schnittstellen bei einem technischen Sicherheitsaudit wie einem Web Application Penetration Test aufgespürt werden.

Gute Vorbereitung ist die halbe Miete – Interception Proxy

Die Untersuchung einer Webanwendung oder Schnittstelle beginnt sowohl für wohlmeinende Tester als auch für böswillige Angreifer in der Regel damit, einen Interception Proxy einzurichten. Der Proxy fungiert als Man-in-the-Middle zwischen dem Browser und der Anwendung, die angegriffen wird. Er zeichnet den gesamten HTTP(S)-Verkehr auf und erlaubt es, Anfragen beliebig zu wiederholen oder zu manipulieren. Der bekannteste Vertreter ist der Burp-Proxy *[5]*, wie in *Abbildung 1* gezeigt.

Umgehung von Rate Limiting

In der Regel sollte jede Anwendung Mechanismen gegen Brute-Force-Angriffe enthalten, mindestens an besonders sensiblen Stellen. Andernfalls kann ein Angreifer zum Beispiel versuchen, sich an einer Anmeldemaske mit einem ihm bekannten individuellen Benutzernamen – oder auch naheliegenden Standard-Accountnamen wie „admin" – und Passwörtern aus einer Passwortliste so lange anzumelden, bis er die richtige Kombination gefunden hat.

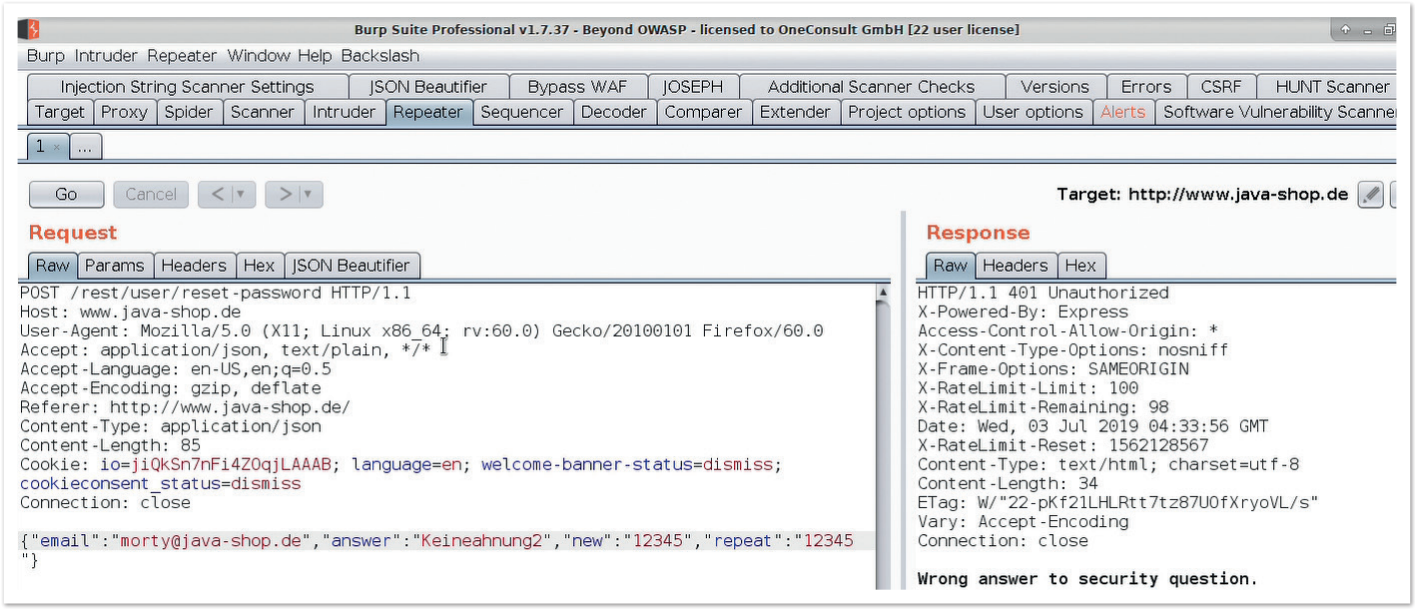

Abbildung 1: Burp-Proxy mit Anfrage (Request) links und Antwort (Response) rechts (Quelle: Frank Ully)

Eine verbreitete Schutzmaßnahme neben CAPTCHAs ist das Rate Limiting *[6]* – auf Deutsch Ratenbegrenzung: Wenn von einer einzelnen IP-Adresse, im Kontext eines Accounts oder innerhalb einer Session zu viele Anfragen in einer bestimmten Zeit gestellt werden, ignoriert die Anwendung weitere Versuche und antwortet mit dem Statuscode `429: Too Many Requests`. Meist senden Endpunkte mit Ratenbegrenzung auch Antwort-Header wie `X-RateLimit-Limit` und `X-RateLimit-Remaining`, die über konfigurierte Grenzen und verbleibende Versuche informieren, wie in *Abbildung 2* dargestellt.

Doch auch wenn ein Rate-Limiting-Mechanismus implementiert ist, etwa weil dafür eine Security-Drittbibliothek eingebunden wurde, kann dieser öfter als gedacht umgangen werden. Header wie `X-Forwarded-For` *[7]* werden normalerweise von Infrastrukturkomponenten wie Reverse Proxys gesetzt, die damit der Anwendung die eigentliche IP-Adresse einer Anfrage mitteilen. Aber nichts hindert einen Angreifer daran, einen solchen Header in einer Anfrage selbst einzufügen, zum Beispiel `X-Forwarded-For: 127.0.0.1` in einem Interception Proxy. Vertraut die Anwendung diesen Angaben, lässt sich die Ratenbegrenzung einfach umgehen *(siehe Abbildung 3)*.

Dadurch ist die Anwendung wieder anfällig für Denial-of-Service- (DoS) und Brute-Force-Angriffe. Schlimmer noch: Ein Angreifer kann den Rate-Limiting-Mechanismus selbst für DoS missbrauchen, wenn er im Header IP-Adressen einträgt, durch die Organisationen oder Einzelpersonen auf das Internet und damit die Anwendung zugreifen – und diese damit aussperren.

Anfällig für Manipulationen dieser Art sind nicht nur weitere Header in Bezug auf die *Quelle* einer Anfrage wie `True-Client-IP`, sondern auch Header bezüglich des *Ziels* einer Anfrage, etwa `X-Forwarded-Host`. Durch blindes Vertrauen in Ziel-bezogene Header entstehen Schwachstellen wie Web Cache Poisoning *[8]*.

Entdeckt werden können solche Lücken durch automatisierte und manuelle Tests, in denen diese Header gezielt eingefügt werden und beobachtet wird, ob sich das Antwortverhalten der Anwendung ändert. Solche anfälligen Konfigurationen sind auch in verbreiteten Rate-Limiting-Bibliotheken zu finden. Ursächlich ist hier also sowohl der Grundsatz „traue niemals Benutzereingaben" missachtet worden wie auch „kenne und prüfe deine Abhängigkeiten".

Server-Side Template Injection (SSTI)

Injektionen sind der erste Punkt der regulären Top-10-Liste. Solche Schwachstellen entstehen dann, wenn ein Angreifer ungehindert bösartige Daten an die Anwendung senden kann, die unverändert von einem Interpreter verarbeitet werden. Die bekannteste Art sind SQL-Injections, mit denen Datenbankabfragen manipuliert werden können; die Top 10 erwähnt aber auch Injektionen für NoSQL, Betriebssystemkommandos oder XML-Technologien wie XPath *[9]*.

Weniger bekannt hingegen ist, dass eine Anwendung auch von Injektionen in anderen und in der modernen Webentwicklung verbreiteten Komponenten wie Templates betroffen sein kann. Templates dienen dazu, HTML-Seiten einfacher zu gestalten, indem statische Vorlagendateien mit Platzhaltern zur Laufzeit mit den eigentlichen Werten befüllt werden; dadurch sind Programmlogik und Präsentationsschicht einer Webanwendung besser getrennt.

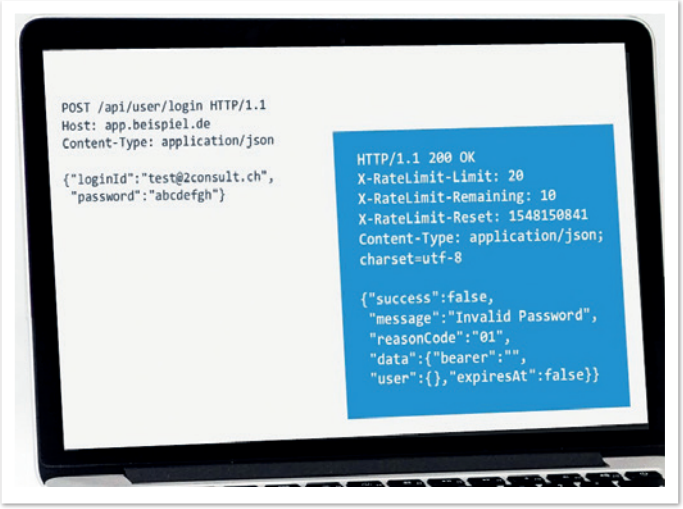

Abbildung 2: Anfrage und Antwort mit Rate-Limiting-Headern (Quelle: Frank Ully)

Verbreitete Template-Sprachen sind FreeMarker oder Velocity unter Java; Jade unter Node.js und Jinja2 unter Python.

Werden jedoch Benutzereingaben ungeprüft direkt in ein serverseitiges Template übernommen, kann ein Angreifer unter Umständen selbst Ausdrücke in der jeweiligen Template-Sprache formulieren – es entsteht eine Server-Side Template Injection (SSTI) *[10]*. Durch sie kann der Angreifer den internen Zustand der Anwendung auslesen und manipulieren oder seine Privilegien erweitern – im schlimmsten Fall direkt Code auf dem Server im Kontext der Anwendung ausführen.

Gefunden werden können SSTIs zum Beispiel, indem in Eingabefelder, die in Templates wieder ausgegeben werden, Ausdrücke mit den jeweiligen Metazeichen eingefügt werden, beispielsweise `{{3+3}}`. Gibt der Server als Antwort „6" zurück, wurde der Ausdruck von der Anwendung interpretiert, wie in *Abbildung 4* gezeigt.

In manchen Fällen kann eine ursprünglich als Cross-Site Scripting (XSS) entdeckte und klassifizierte Schwachstelle eigentlich eine Form von SSTI sein. Auch Werkzeuge wie Tplmap *[11]* – benannt in

Abbildung 3: Anfrage mit eingefügtem X-Forwarded-For Header (Quelle: Frank Ully)

Anlehnung an das bekannte SQLmap für SQL-Injections – helfen beim Finden und Ausnutzen von Schwachstellen in Templates.

Entwickler verhindern Injektionen in Templates, indem sie die Korrektheit von Benutzereingaben über eine strikte Eingabevalidierung prüfen und Metazeichen entfernen sowie Funktionen zur Interpretation von Template-Ausdrücken niemals mehrfach aufrufen. Bei besonders hohen Sicherheitsanforderungen empfiehlt es sich, die Template-Umgebung in einem gut gehärteten Docker Container *[12]* vom Rest der Anwendung zu trennen.

Ebenfalls von Injections betroffen sein kann Client-seitiges Templating *[13]* in Frameworks wie React oder Angular. Hier wird jedoch im schlimmsten Fall nicht Code auf dem betroffenen Server ausgeführt, sondern JavaScript (XSS) im Browser des Opfers.

Server-Side Request Forgery (SSRF)

Was hat es mit Server-Side Request Forgery (SSRF) *[14]* auf sich? Ursache dieser Schwachstelle ist, dass eine Anwendung einen Request ausführt, der als Folge eine interne oder externe Ressource aufruft und dessen Ziel der Benutzer ganz oder teilweise vorgeben kann.

Das ist heutzutage nichts Ungewöhnliches – viele Anwendungen oder Schnittstellen nehmen in der Anfrage direkt eine URL entgegen und tun etwas damit, überprüfen beispielsweise den angegebenen Link oder erzeugen aus einem Bildpfad ein Vorschaubild. In anderen Fällen leiten sie Teile einer Anfrage an andere Systeme weiter, etwa zur Authentifizierung oder zum Monitoring *(siehe Abbildung 5)* – oder an ein Zahlungs-Gateway oder ein anderes externes API im Hintergrund.

Oft bestehen dabei Vertrauensbeziehungen, weil Server sich selbst und anderen Systemen im selben Netzwerk vertrauen. Nutzen Angreifer dies aus, können sie somit: Firewalls umgehen und aus dem Internet Dienste erreichen, die eigentlich nur auf dem lokalen Host oder im internen Netz verfügbar sind; das entfernte Netzwerk scannen; sensible interne Daten auslesen. Gegebenenfalls sind sogar reflektierte XSS-Angriffe oder das Ausführen von Code auf einem Server möglich.

In einer Cloud-Umgebung kann SSRF sehr kritisch sein. Wie in *Abbildung 6* gezeigt, können zum Beispiel über Metadaten-URLs wie `http://169.254.169.254/latest/user-data` unter Umständen direkt Zugangsdaten abgefragt werden *[15]*.

Entdeckt werden kann SSRF durch automatisierte Werkzeuge wie den Collaborator *[16]* des kostenpflichtigen Burp-Proxy. Manuell kann es gefunden werden, indem überprüft wird, ob ein Wert eines Request ganz oder teilweise an andere Systeme weitergeleitet wird und dabei manipuliert werden kann. Ebenfalls hilft ein Blick in die *ausgehenden* Firewall-Logs des Servers, um ungewöhnliche Ziele von Anfragen zu entdecken.

Eine unwirksame Schutzmaßnahme gegen SSRF ist Blacklisting, also das Blockieren von mutmaßlich bösartigen Eingaben, die zum Beispiel `127.0.0.1` oder `localhost` enthalten. Jedoch lassen sich IP-Adressen auf unterschiedliche Weise darstellen *[17]*, beispielsweise abgekürzt als `127.1`, in Dezimalnotation `2130706433` oder

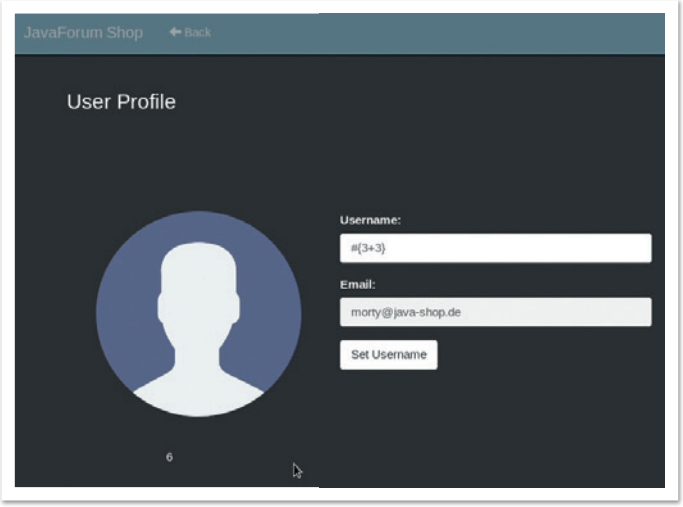

Abbildung 4: Server-Side Template Injection (Quelle: Frank Ully)

Abbildung 5: Anfrage mit externer URL, potenziell anfällig für Server-Side Request Forgery (Quelle: Frank Ully)

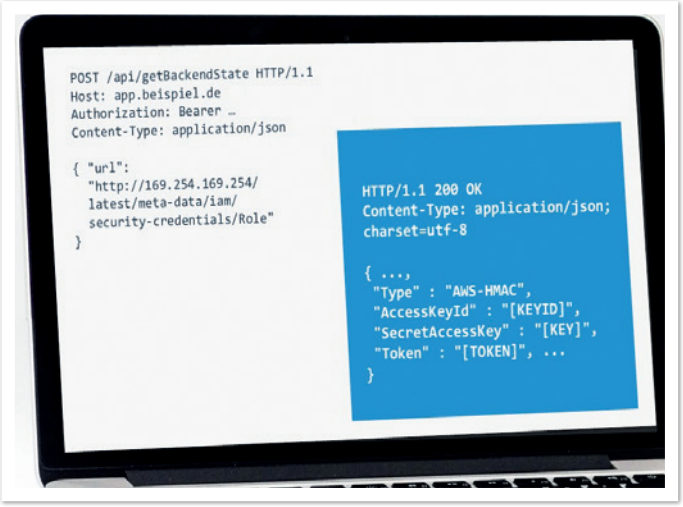

Abbildung 6: Anfrage mit veränderter URL, Server-Side Request Forgery wurde ausgenutzt (Quelle: Frank Ully)

hexadezimal `0x7F000001`. Darüber hinaus kann auch ein entsprechend konfigurierter Domainname wie `9zlhb.xip.io` als aufgelöste IP-Adresse `127.0.0.1` zurückmelden.

Zur Prävention von SSRF ist es wie immer wichtig, die vom Benutzer – oder Angreifer – gelieferten Daten so zu validieren, dass nur erwartete Werte akzeptiert werden, zum Beispiel über entsprechende Maßnahmen für Eingabevalidierung *[18]*. Idealerweise findet jedoch ein Whitelisting statt: Explizit erlaubt ist nur der Zugriff auf eine Liste mit bekannten Ressourcen – nicht nur bezogen auf Hostnamen, sondern auch auf Protokoll und Port. Als weiterer Schutz können Firewall-Regeln für ausgehenden Verkehr dienen, die einschränken, mit welchen entfernten Servern und auch mit welchen Ports auf dem lokalen System(!) die Anwendung kommunizieren kann. Schließlich sollten alle Dienste in einem Netzwerk, wie Monitoring-Tools oder Admin-Oberflächen, immer mit einer Anmeldemaske gesichert sein.

Cross-Origin Resource Sharing (CORS)

Normalerweise sind Webanwendungen von der Same Origin Policy (SOP) *[19]* geschützt, die vom Browser durchgesetzt wird.

Die SOP erlaubt Client-seitigen Skriptsprachen wie JavaScript nur dann uneingeschränkten Zugriff auf Daten in einer zweiten Webseite, wenn beide vom selben Ursprung (Englisch: *Origin*) stammen. Zwei Webseiten oder Ressourcen haben dann denselben Ursprung, wenn Protokoll, Domain und Port übereinstimmen. Beispielsweise kann ein unter `https://beispiel.de/verzeichnis1` eingebettetes JavaScript auf Elemente unter `https://beispiel.de/verzeichnis2` zugreifen, nicht aber auf `https://beispiel.de/`, weil hier eine andere Domain steht.

Cross-Origin Resource Sharing (CORS) *[20]* ist ein Mechanismus, um die Same Origin Policy gezielt aufzuweichen, und erlaubt über den Browser Kommunikation zwischen Webanwendungen, die auf unterschiedlichen Origins laufen. CORS als neuerer Standard wurde notwendig, weil mit moderner Microservice-Architektur und externen APIs Zugriffe auf entfernte Ressourcen zunehmen und andernfalls nur mit eher unschönen Abhilfen möglich sind.

Dazu verwendet CORS Header. Der `Origin`-Header wird bei Cross-Origin Requests vom Browser oder Client selbst gesetzt. Die Webanwendung kann nun mit `Access-Control-Allow-`-Headern antworten und damit dem Client signalisieren, woher (`Access-Control-Allow-Origin`) welche HTTP-Methoden (`Access-Control-Allow-Methods`) erlaubt sind, welche Header in der Anfrage enthalten sein dürfen (`Access-Control-Allow-Headers`) und ob der Client vorhandene Credentials über Cookies oder HTTP-Authentifizierung mitsenden soll (`Access-Control-Allow-Credentials`).

Anfällig *[21]* wird eine Webanwendung nun, wenn sie blindlings jeden empfangenen `Origin`-Anfrage-Header im Antwort-Header `Access-Control-Allow-Origin` reflektiert und zudem in der Antwort den Header `Access-Control-Allow-Credentials: True` setzt. In diesem Fall kann ein bösartiges JavaScript von der Domain eines Angreifers, die ein Opfer ansurft, auf einer verwundbaren Webanwendung oder API im Namen des dort angemeldeten Opfers Aktionen durchführen. Aber auch in subtileren Fällen bestehen Gefahren: Wenn das `null`-Origin erlaubt ist oder das Origin nur in Bruchstücken validiert wird und etwa – statt der Domain `beispiel.de` – `keinbeispiel.de` als erlaubter Ursprung angesehen wird *(siehe Abbildung 7)*.

Auch Anwendungen ohne `Access-Control-Allow-Credentials` sind möglicherweise anfällig, wenn als Origin die Wildcard `*` oder

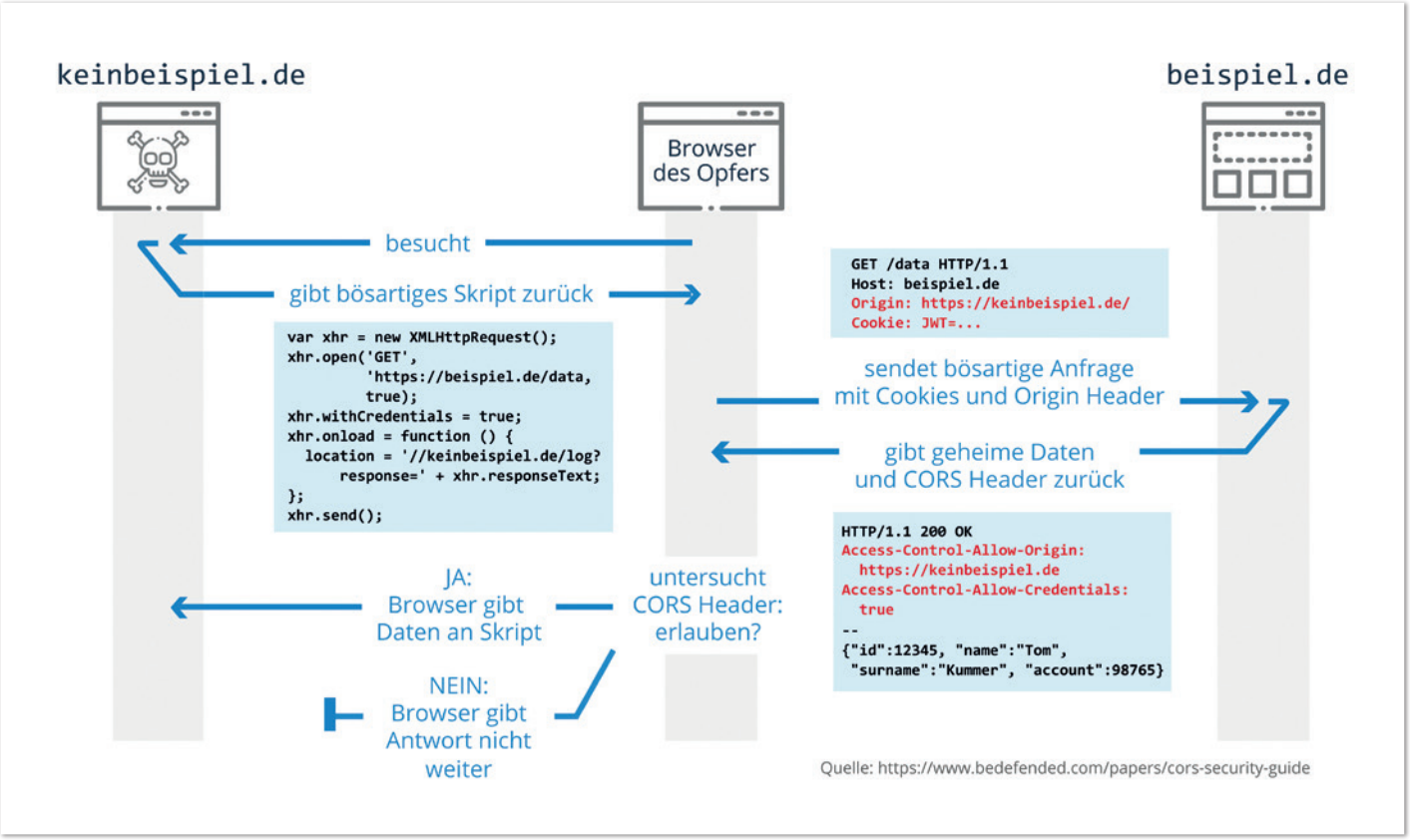

Abbildung 7: Schematischer Ablauf der Ausnutzung einer Schwachstelle im Cross-Origin Resource Sharing (in Anlehnung an [21])

`null` erlaubt sind oder der Origin nur unzureichend überprüft und dann reflektiert wird – zum Beispiel für das Umgehen von IP-Beschränkungen oder Client-Side Cache Poisoning.

Man kann solche Schwachstellen finden, indem man mit einem Proxy in Anfragen einen `Origin`-Header einfügt und wie zuvor beschrieben verändert. Auch automatisierte Werkzeuge wie CORScanner *[22]* helfen dabei, Fehlkonfigurationen aufzudecken.

Häufig sind CORS-Einstellungen in Frameworks fehlerhaft. Verhindern kann man solche Schwachstellen oft schon, indem man die Standard-Konfigurationen überprüft und CORS nur aktiviert wird, wenn es wirklich benötigt wird. Dann sollte das Origin gegen eine Whitelist strikt geprüft werden – und nicht mit regulären Ausdrücken. Weitere Maßnahmen wie das Einschränken der erlaubten Methoden in `Access-Control-Allow-Methods` oder das Setzen des Headers `Vary: Origin` erhöhen die Sicherheit zusätzlich.

OAuth und JWT – weitere Quellen unbekannterer Schwachstellen

Es gibt noch zahlreiche weitere unbekanntere und durchaus kritische Schwachstellenarten. Besonders hervorzuheben sind Schwachstellen in neueren, als besonders sicher angesehenen Authentifizierungsmechanismen wie OAuth 2.0 *[23]* und beim Einsatz von JSON Web Token (JWT) *[24]* statt normaler Sitzungstoken. Sind JWT unzureichend kryptographisch gesichert oder die Implementierung oder Konfiguration ihrer Prüfung fehlerhaft, können Angreifer Daten für Authentifizierung und Autorisierung manipulieren und so ihre Privilegien erweitern. Dagegen schützen können sich Entwickler beispielsweise, indem sie die „Best Current Practices" zu OAuth *[25]* und JWT *[26]* berücksichtigen. Diese sind derzeit allerdings noch im Entwurfsstadium.

Websicherheit grundsätzlich – weiterführende Quellen

Wie oben beschrieben, beleuchten die OWASP Top 10 – und auch dieser Artikel – nur schlaglichtartig einzelne Schwachstellen. Eine Veröffentlichung, die sich systematisch dem Entwickeln von sicheren Webanwendungen widmet, ist der „Application Security Verification Standard" (ASVS) *[27]*. Er gibt in verschiedenen Kategorien detaillierte Sicherheitsanforderungen für Anwendungen vor, die von Stakeholdern genutzt werden können, um zu bestimmen, was eine sicher entwickelte Anwendung ist und wie das erreicht und getestet werden kann.

Der „OWASP Testing Guide" *[28]* beschreibt ein umfassendes Framework für Penetrationstests, das Organisationen an ihre Bedürfnisse anpassen können, und enthält einen detaillierten und sehr technischen Leitfaden, anhand dessen Tester einzelne Sicherheitsprobleme aufdecken können. Während der Testing Guide die Überprüfung von Webanwendungen in *Grey-Box*-Szenarien beschreibt, also mit nur teilweise verfügbaren Informationen, gibt der „Code Review Guide" *[29]* Hinweise, wie bei *White-Box*-Analysen mit vorliegendem Quelltext Lücken gefunden werden können.

Analog zu den Top 10 mit deren Blick auf Schwachstellen aus Sicht der Angreifer beschreiben die „Top 10 Proactive Controls" *[30]* aus Sicht der Verteidiger, welche Sicherheitsmaßnahmen Anwendungen in jedem Fall enthalten sollten. Zu jedem Punkt – unter anderem Eingabevalidierung und Ausgabekodierung, Verschlüsselung und Zugriffskontrolle – wird dargestellt, was darunter zu verstehen ist, wie er umgesetzt werden könnte, welche Schwachstellen dadurch verhindert werden und welche Tools und Quellen dabei helfen.

Selbst ausprobieren – Werkzeuge und Anwendungen

Am einfachsten lernt man durchs Machen. Um sich eine Übungsumgebung für einfache bis fortgeschrittene Webapp-Lücken einzurichten, egal ob man Projektleiter, Entwickler oder Tester ist, bietet sich eine Kombination von Interception Proxy und absichtlich verwundbarer Software an.

OWASP bietet den kostenfreien Zed Attack Proxy (ZAP) an *[31]*, der teilweise auch automatisiert Schwachstellen findet. Als Alternative kann der Burp-Proxy in der kostenlosen Community Edition *[5]* dienen, mit allerdings deutlich eingeschränktem Funktionsumfang.

Ziel der Übungsangriffe ist am besten eine Webanwendung, in die mit Vorsatz Lücken eingebaut wurden. Davon gibt es inzwischen eine unübersehbare Menge – entweder lokal als Installationspaket, virtuelle Maschine oder Docker-Image *[32]* eingesetzt oder direkt online im Internet *[33]*.

Hervorzuheben ist der „OWASP Juice Shop" *[34]*, der lokal wie auch in Cloud-Umgebungen vielfältig installiert werden kann. Dieser sprichwörtliche Saftladen ist nach modernem Ansatz als JavaScript-lastige Frontend-Anwendung (Angular) entwickelt, die auf REST-APIs (Node.js mit Express) zugreift. Der Juice Shop wird regelmäßig mit aktuellen Schwachstellen bestückt, etwa im Zuge des „Google Summer of Code". Er bietet eine sehr umfangreiche Dokumentation *[35]*, die hilft, das Anzapfen des Saftladens vorzubereiten und Schwachstellen systematisch zu finden, und enthält zudem einen ausführlichen Lösungsleitfaden.

Quellen

[1] *https://www.owasp.org/index.php/Main_Page*
[2] *https://www.owasp.org/images/9/90/OWASP_Top_10-2017_de_V1.0.pdf*
[3] *https://github.com/OWASP/Top10/milestones*
[4] *https://www.owasp.org/index.php/Top_10-2017_Foreword*
[5] *https://portswigger.net/burp*
[6] *https://nordicapis.com/everything-you-need-to-know-about-api-rate-limiting/*
[7] *https://portswigger.net/kb/issues/00400110_spoofable-client-ip-address*
[8] *https://portswigger.net/blog/practical-web-cache-poisoning*
[9] *https://www.owasp.org/index.php/Top_10-2017_A1-Injection*
[10] *https://portswigger.net/blog/server-side-template-injection*
[11] *https://github.com/epinna/tplmap*
[12] *https://dev.to/petermbenjamin/docker-security-best-practices-45ih*
[13] *https://portswigger.net/kb/issues/00200308_client-side-template-injection*
[14] *https://portswigger.net/web-security/ssrf*
[15] *https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Server%20Side%20Request%20Forgery*

[16] *https://portswigger.net/burp/documentation/collaborator*
[17] *https://www.psyon.org/tools/ip_address_converter.php?ip=127.0.0.1*
[18] *https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Input_Validation_Cheat_Sheet.md*
[19] *https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy*
[20] *https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS*
[21] *https://www.bedefended.com/papers/cors-security-guide*
[22] *https://github.com/chenjj/CORScanner*
[23] *https://github.com/koenbuyens/Vulnerable-OAuth-2.0-Applications/blob/master/authorizationcode_tester.md*
[24] *https://cheatsheetseries.owasp.org/cheatsheets/JSON_Web_Token_Cheat_Sheet_for_Java.html*
[25] *https://tools.ietf.org/html/draft-ietf-oauth-security-topics-13*
[26] *https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp-04*
[27] *https://github.com/OWASP/ASVS*
[28] *https://www.owasp.org/index.php/OWASP_Testing_Project*
[29] *https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project*
[30] *https://www.owasp.org/index.php/OWASP_Proactive_Controls*
[31] *https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project*
[32] *https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project/Pages/Offline*
[33] *https://github.com/geeksonsecurity/vuln-web-apps*
[34] *https://www.owasp.org/index.php/OWASP_Juice_Shop_Project*
[35] *https://bkimminich.gitbooks.io/pwning-owasp-juice-shop/*

Frank Ully
Oneconsult Deutschland
frank.ully@oneconsult.com

Frank Ully schloss sein Studium an der Hochschule Karlsruhe als Diplom-Technikredakteur ab. Er baute die Dokumentationsabteilung eines Softwareherstellers auf und leitete sie einige Jahre. Später war er dort für IT-Sicherheit verantwortlich. Berufsbegleitend absolvierte er den Masterstudiengang Security Management an der TH Brandenburg und zertifizierte sich zum Offensive Security Certified Expert (OSCE) sowie Professional (OSCP); er erwarb zudem weitere Zertifikate. Im November 2017 begann er seine Arbeit bei Oneconsult Deutschland, seit April 2018 ist er Senior Penetration Tester und Security Consultant.

Data Analytics 2020

28. & 29. April | in Düsseldorf

DOAG

analytics.doag.org

Programm online

Early Bird
bis 21. Januar 2020

JavaLand 2020

17. - 19. März 2020 in Brühl bei Köln

Ab sofort Ticket & Hotel buchen!

www.javaland.eu

Präsentiert von: DOAG Heise Medien

Community Partner: iJUG Verbund

www.javaland.eu