Die wichtigsten Schwachstellen

Die OWASP Web Top 10 in Java-Anwendungen

Sonderdruck aus
JavaSPEKTRUM 5/2021

Rafael Fedler

Die OWASP Top 10 sind ein De-facto-Standard-Katalog mit den häufigsten in Web-Anwendungen auftretenden Schwachstellen. Diese zu verhindern, gilt in der sicheren Web-Entwicklung als allgemein anerkanntes Ziel. Welche Schwachstellen beinhalten die OWASP Top 10? Welche Risiken gehen von ihnen aus für Anwendungen, Systeme, Nutzer und Daten? Wie können sich diese Schwachstellen in Java-basierten Anwendungen manifestieren? Wie kann man sicher entwickeln und somit seine Anwendungen schützen?

Bereits seit 2003 veröffentlicht OWASP (*O*pen *W*eb *A*pplication *S*ecurity *P*roject), eine Vereinigung von Spezialisten aus dem Bereich IT-Sicherheit, die sogenannten „OWASP Top 10". Hierbei handelt es sich um eine Liste von 10 Schwachstellenkategorien, die am häufigsten in Web-Anwendungen auftreten. Diese Liste wird durch einen Community-Prozess ermittelt und alle paar Jahre aktualisiert. Die aktuellste Version sind die OWASP Top 10 2017 [O1017].

Da die OWASP mittlerweile entsprechende Listen auch für andere Technologien wie zum Beispiel APIs (OWASP API Top 10), Mobil-Anwendungen („Apps"; OWASP Mobile Top 10) und Internet of Things (OWASP IoT Top 10) pflegt und veröffentlicht, spricht man manchmal auch von den OWASP Web Top 10, um diese gegenüber den anderen Katalogen abzugrenzen. Spricht jemand von *den* OWASP Top 10, sind meist die OWASP Web Top 10 gemeint.

Die OWASP Top 10 haben sich in den letzten Jahren zu einem wenn auch nicht formalen, aber De-facto-Standard entwickelt: Die Vermeidung von Schwachstellen der zehn Kategorien gilt in der sicheren Web-Entwicklung als generell anerkanntes Ziel und in sicherheitsbewussten Entwicklerteams als Minimalstandard. Große Organisationen mit ausgereiften Sicherheitsprozessen inventarisieren Schwachstellen in Eigen- und Auftragsentwicklungen nach den OWASP Web Top 10, automatisierte Schwachstellenscanner ordnen Funde ebenfalls den entsprechenden Kategorien zu, und fast alle Sicherheitstestdienstleister richten ihre Sicherheitstests, sogenannte Penetrationstests, inhaltlich nach den OWASP Top 10 aus.

Möchte man sicher entwickeln und somit die Anwendung, das ausführende System, die verarbeiteten und gespeicherten Daten gegenüber Hacker-Angriffen absichern, ist es unabdingbar, zu wissen, was potenziell falsch gemacht werden kann – damit man es dann verhindern kann. Nur wenn man potenziell auftretende Schwachstellen und ihre Ursachen kennt, kann man Gegenmaßnahmen ergreifen. Daher gibt dieser Artikel einen Einblick in die OWASP Top 10.

Die zehn Kategorien der OWASP Top 10

Wie der Name bereits nahelegt, bestehen die OWASP Top 10 aus zehn Kategorien. Vor einer detaillierten Behandlung jeder dieser Kategorien folgt daher in Tabelle 1 eine erste Übersicht. Die ge-

Rafael Fedler ist seit fast 10 Jahren in der IT-Sicherheit tätig. Seit 2015 prüft und stärkt er als Penetrationstester und Red Teamer die Sicherheit von Unternehmen mit offensiven Methoden. Von 2011 bis 2014 war Rafael Fedler in der IT-Sicherheitsforschung aktiv.
E-Mail: rf@nsideattacklogic.de

nannten Kategorien werden in den folgenden Abschnitten im Detail behandelt, inklusive Ursachen, Risiken und wie man seine Anwendungen schützen kann.

A1: Injection

Die erste Kategorie, *A1: Injection*, ist - zusammen mit ihren Sonderfällen *A4* (XML External Entities – XXE) und *A8* (Unsichere Deserialisierung) – gleich eine der gefährlichsten, wenn nicht sogar die gefährlichste: Schwachstellen in dieser Kategorie führen oft oder sogar meistens dazu, dass der Angreifer Code auf dem Server ausführen kann – zum Beispiel Datenbankabfragen, Betriebssystembefehle oder Programmcode. Im letzteren der beiden Fällen spricht man auch von Remote Code Execution (RCE) – im Allgemeinen als die schwerwiegendste Form von Schwachstelle angesehen.

Kategorie	Name	Erläuterung
A1	Injection	Schwachstellen, die es erlauben, Code in die Business-Logik der Anwendung bzw. den Anwendungsserver zu injizieren
A2	Fehlerhafte Authentifizierung	Schwachstellen in der Authentifizierung von Nutzern
A3	Preisgabe sensibler Daten	Unzureichender oder fehlender Schutz von Daten gegen unbefugten Zugriff
A4	XML External Entities (XXE)	Ein Sonderfall von Injection-Angriffen (A1) in XML-Parsern
A5	Fehlerhafte Zugriffskontrolle	Unzureichende, fehlende oder fehlerhafte Mechanismen zur Zugriffskontrolle auf Daten und Funktionen
A6	Sicherheitsrelevante Fehlkonfiguration	Einstellungen mit negativem Einfluss auf die Sicherheit
A7	Cross-Site Scripting (XSS)	Schwachstellen, die es erlauben, Code in die Nutzeroberfläche zu injizieren
A8	Unsichere Deserialisierung	Ein Sonderfall von Injection-Angriffen (A1) in Code, der serialisierte Objekte verarbeitet
A9	Nutzung von Komponenten mit bekannten Schwachstellen	Schwachstellen, die sich aus Drittkomponenten ergeben und bereits öffentlich bekannt sind
A10	Unzureichendes Logging & Monitoring	Fehlende Maßnahmen zur Sichtbarkeit von sicherheitsrelevanten Ereignissen, Handlungen und Anwendungs-/Systemstatus

Tabelle 1: Kategorien der OWASP (Web) Top 10 2017

In die Kategorie *A1: Injection* fallen zum Beispiel SQL und NoSQL Injections, bei denen der Angreifer Datenbankabfragen manipulieren und somit meist auf die ganze Datenbank zugreifen kann. Durch besondere Stored Procedures oder die Möglichkeit, Libraries nachzuladen, können SQL Injections zusätzlich in nicht wenigen Fällen auch zu Remote Code Execution (RCE) führen. Auch gehören zur Kategorie *A1* Buffer Overflows: Der Angreifer kann einen Speicherbereich zum „Überlaufen" bringen und in benachbarte Speicherbereiche beliebige Daten hineinschreiben. Da Nutz- und Kontrolldaten von Programmen meist gemischt nebeneinander im Speicher liegen, ist es oft möglich, auch Kontrolle über das Programm zu übernehmen. Dies ist bei Java allerdings signifikant schwieriger und, sprach- bzw. JVM-bedingt, viel seltener möglich als bei kompilierten, ohne JVM ausgeführten Sprachen.

Abgesehen von den Datenbank-Injections und Buffer Overflows gibt es noch viele weitere Schwachstellentypen in dieser Kategorie, wie zum Beispiel OS Command Injections, bei denen Angreifer Betriebssystembefehle injizieren und ausführen können.

Im Allgemeinen gibt es zwei Lösungsansätze, um Schwachstellen der Kategorie *A1* zu begegnen:

- *Markierung und Auftrennung von Code und Nutzdaten:* Wo durch die darunterliegende Technologie möglich, sollte man eine Trennung von Nutzdaten und Code vornehmen. Im Fall von SQL spricht man zum Beispiel von „Prepared Statements" [JPRST], wenn sämtlicher Input statisch oder in sauber getrennt gekennzeichneten Parametern vorliegt und somit sicher gegen Injektion von SQL-Code ist.
- *Eingabedatenvalidierung, Sanitization & Encoding/Escaping:* Wenn Daten von außerhalb der Anwendung entgegengenommen werden, werden diese für den Zielkontext, in den sie eingefügt werden sollen, auf korrekte Form und Länge validiert, „gesäubert" (sanitized) und ggfs. durch Encoding oder Escaping „entwaffnet". In vielen Fällen muss ein Angreifer aus einem Nutzdatenwert ausbrechen. Dies geschieht oft durch Sonderzeichen (z. B. Anführungszeichen zum Ausbruch aus einem String). Werden diese escapet, kann der Angreifer nicht aus dem Parameter ausbrechen.

Welcher Ansatz der richtige ist, hängt von der abzusichernden Technologie ab. Meist ist Trennung von Code und Daten sicherer, jedoch seltener möglich. Mindestens eine der beiden Maßnahmen sollte aber stets zum Einsatz kommen, wo Daten mit Code vermengt werden.

Tabelle 2 zeigt je ein Beispiel für unsicheren und sicheren Code. In der sicheren Variante wurde der Parameter `department` als solcher markiert und das Statement vorbereitet (prepared), sodass kein Code mehr hinzugefügt werden kann. In der unsicheren Variante hingegen können Nutzer die Datenbankabfrage beliebig manipulieren.

A2: Fehlerhafte Authentifizierung

Das Problemfeld der fehlerhaften Authentifizierung, also wie Nutzer angemeldet und nach der Anmeldung weitergehend identifiziert werden, ist ein sehr weites. Prinzipiell kann vieles schiefgehen: Von fehlendem Schutz gegen Passwort-Rate-Angriffe (wie Bruteforce, Wörterbuchangriffe, Credential Stuffing) über schlecht gesicherte Sitzungs-Cookies, welche die Nutzer identifizieren, übersehenen Testnutzerkonten oder Default-Passwörtern, schlechter Entropie, schwachen Anforderungen an Passwörter, bis hin zu mangelnder Zwei- oder Mehrfaktor-Authentifizierung (2FA/MFA) ist hier alles dabei. Auch kann es beispielsweise möglich sein, gültige Nutzernamen anderer Nutzer zu identifizieren (sogenannte Enumeration-Schwachstellen), die dann wiederum für Pass-

unsicher	sicher

`String query =`
`  "select * from employees where department='"`
`  + user_input + "';";`
	`String query =`
	`  "select * from employees where department=?;";`
	`  PreparedStatement prepStmt =`
	`  connection.prepareStatement(query);`

Tabelle 2: Beispiele für sichere und unsichere Konstruktion von Datenbank-Abfragen

wort-Rate-Angriffe verwendet werden können. Aufgrund der Unterschiedlichkeit der Fehler, die in diesem Bereich gemacht werden können, können leider keine Pauschalaussagen darüber getroffen werden, wie diese zu verhindern sind.

Wichtige Anmerkung: Authentifizierung ist nicht mit Autorisierung zu verwechseln. *Authentifizierung* beschäftigt sich mit initialer und fortlaufender Identifikation von Nutzern, während sich *Autorisierung* damit beschäftigt, welche Nutzer welche Berechtigungen zu bestimmten Objekten oder Aktionen haben. Autorisierungsschwachstellen werden in Kategorie *A5: Fehlerhafte Zugriffskontrolle* behandelt.

A3: Preisgabe sensibler Daten

Genau wie Kategorie *A2: Fehlerhafte Authentifizierung* gibt es viele und sehr unterschiedliche Arten und Weisen, wie Sicherheitsrisiken in Kategorie *A3: Preisgabe sensibler Daten* entstehen können. Hierzu gehören Schwächen in der Verschlüsselung (z. B. SSL/TLS), Dateien, die unbeabsichtigt zugänglich gemacht werden durch Ablage an falscher Stelle, vergessene Backup-Archive, aktivierte Verzeichnisdateilisten (z. B. von Upload-Verzeichnissen), Auslieferung von serverseitigen Code-Dateien als Download statt Ausführung, Datenbank-Dumps, Konfigurationsdateien in Web-Verzeichnissen und viele mehr.

Was bei Java-Anwendungen oft besonders schwer wiegen kann: WAR- beziehungsweise JAR-Dateien, Manifests, Anwendungsserver- (Tomcat, JBoss, WebLogic ...) oder Anwendungskonfigurationsdateien (inkl. eventueller Passwörter wie zum Beispiel für Datenbankverbindungen, API-Keys und andere Secrets), welche in Webverzeichnissen liegen – also an Pfaden, die für Nutzer zugänglich sind. Aufgrund von manuellen und automatischen Deployment-Prozessen, die in der Praxis anzutreffen sind, passiert dies nicht so selten. Darüber können dann unberechtigte Parteien wie Nutzer und Angreifer Zugriff zum Code der Anwendung erlangen, der im Falle von Java sehr leicht dekompilierbar ist und womit dann Schwachstellen, Passwörter, Datenbankverbindungsparameter und andere sensible Informationen erbeutet werden können. Auch ist es möglich, die Anwendung dann auf Sicherheitslücken zu analysieren, die mit diesem White-Box-Zugriff weitaus einfacher und schneller zu finden sind als mit Black-Box-Zugriff, also ohne Zugriff auf den Anwendungscode.

Kategorie *A3* ist abzugrenzen von Kategorie *A5: Fehlerhafte Zugriffskontrolle*. Schwachstellen in *A5* betreffen Autorisierungsprobleme auf Objekt- und Funktionsebene, die meistens durch Anwendungscode selbst gesichert werden (sollten). *A3* hingegen beschäftigt sich mit Daten und Dateien auf dem Übertragungsweg oder im Dateisystem, ist also eine Abstraktionsebene unterhalb der Anwendung selbst.

A4: XML External Entities (XXE)

Bei Kategorie *A4: XML External Entities* handelt es sich eigentlich um einen Sonderfall von Kategorie *A1: Injection*. In dieser Kategorie finden sich Schwachstellen, die aus dem fehlerhaften Umgang von XML-Parsern mit von außerhalb der Anwendung stammenden Definitionen von sogenannten „Entities". Dies sind Zeichenketten, die teils vom XML-Standard vorgegeben werden, aber auch in der sogenannten DTD, der Document Type Definition, selbst definiert werden können.

Vereinfacht können XML-Entities als „Variablen+" verstanden werden, da XML-Parser bestimmte Entity-Typen auflösen und mit externen Inhalten befüllen können. Und hier liegt auch das Risiko: Ist der XML-Parser oder die XML-Parsing-Library so konfiguriert, dass sie beliebige Definitionen solcher Entities (sogenannte External Entities) auflöst, kann ein Angreifer über diese auf Ressourcen außerhalb der betroffenen Anwendung zugreifen und diese in die Server-Antwort integrieren. Hierzu gehört das Auslesen von lokalen Dateien, Netzwerkdateifreigaben und teils sogar Netzwerk-Ressourcen, die zum Beispiel vom XML-Parser beziehungsweise der Library per HTTP angesprochen werden können. In letzterem Fall kann man somit Netzwerkpakete an andere Systeme im Netz des verwundbaren Systems schicken und damit eventuell sogar interne Funktionalität aufrufen, die nicht zugänglich sein sollte (sog. SSRF – Server-Side Request Forgery).

Ein alternativer Angriff, der über XXE-Lücken durchgeführt werden kann, ist ein sehr effektiver und leicht durchzuführender Denial-of-Service-Angriff: Die sogenannte „Billion Laughs"-Attacke wird durchgeführt, indem mehrere XXE-Definitionen rekursiv auf einander verweisen und bei deren Auflösung den Server überlasten. Ein Beispiel für eine gefährliche XML External Entity-Definition in einer DTD findet sich im Code-Abschnitt in Listing 1, wobei die Entity den Inhalt einer lokalen Datei, /etc/password, in das Ergebnis integriert.

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

Listing 1: Beispiel für XXE-Angriff

Alles in allem lassen sich, durch den darüber möglichen Dateisystem- oder Netzwerkzugriff, XXE-Lücken oft zu gefährlichen Angriffsketten aufbauen, über die der Angreifer das betroffene System oder dessen Daten vollständig unter seine Kontrolle bringen kann. Der Schutz gegen diese gefährlichen Lücken hingegen ist relativ einfach: In XML-Parsern beziehungsweise Parsing-Bibliotheken sollte die Konfiguration entweder global oder bei jeder Instanziierung so gewählt werden, dass extern definierte Entities nicht aufgelöst oder ausgewertet werden. Diese einfache Einstellung verhindert die genannten Angriffe. Mehr Infos bei OWASP unter [OWXXE].

A5: Fehlerhafte Zugriffskontrolle

In Kategorie *A5: Fehlerhafte Zugriffskontrolle* werden verschiedene Schwachstellen zusammengefasst, die mit Sicherheitslücken in der Autorisierung zu tun haben, also darin, wie eine Anwendung den Zugriff auf Objekte und Funktionen regelt: Wer darf welche Funktionen aufrufen und wer darf auf welche Daten und Objekte zugreifen?

Die trivialste Lücke aus dieser Kategorie ist die sogenannte „Insecure Direct Object Reference" (IDOR). Bei dieser kann auf Objekte direkt über ihren Schlüssel (zum Beispiel eine numerische ID)

zugegriffen werden, obwohl dies nicht möglich sein sollte. Angenommen, dass beispielsweise ein Bewerber in einer Bewerbungsplattform über den Link `https://jobs.firma.de/CV?id=123` seinen hochgeladenen Lebenslauf prüfen kann, könnte er die numerische ID ersetzen, um – falls dies nicht verhindert wird – auf die Lebensläufe anderer Bewerber zuzugreifen. Dies ist die trivialste Schwachstelle aus dieser Kategorie, aber dennoch nicht selten anzutreffen. Dies gilt vor allem deswegen, weil in modernen Web-Anwendungen an vielen Stellen auf Objekte über ihre Schlüssel zugegriffen wird und die Entwickler konsistent jede einzelne Stelle absichern müssen.

Andere Schwachstellen aus dieser Kategorie betreffen den Funktionszugriff, Rollenschemata, Rechteerhöhungsmöglichkeiten, generell fehlendende serverseitige Berechtigungsprüfungen, administrative Funktionen, die nur in der Nutzeroberfläche ausgeblendet wurden, aber mit Netzwerkanfragen dennoch angesprochen werden können, und weitere.

Schützen kann man sich gegen Lücken aus dieser Kategorie, indem man *konsistent* Berechtigungsprüfungen für alle Funktionen, Unterseiten, Objektzugriffe (egal ob lesend oder schreibend) und Endpunkte umsetzt. Dies kann entweder durch gute Software-Patterns geschehen, wo jeder Request erst an eine Autorisierungs-Middleware mit Rollen-zu-Rechten-Mappings umgeleitet wird, durch Annotationen wie zum Beispiel die von Spring Method Security (Beispiele: `@Secured`, `@RoleAllowed`, `@PreAuthorize` usw., s. a. [SPSEL]) und andere Verfahren, die Konsistenz vereinfachen. Wichtig ist auch, dass die darunterliegenden Rollen und Berechtigungen sinnvoll an Hand des „Principle of Least Privilege" gewählt werden.

A6: Sicherheitsrelevante Fehlkonfiguration

Auch diese Kategorie ist inhaltlich sehr breit. Hierzu gehören alle Formen der Konfiguration, entweder der Anwendung, des Anwendungsservers oder des Webservers, die die Sicherheit beeinträchtigen können. Einige wenige Beispiele betreffen beispielsweise die sogenannten HTTP-Sicherheits-Header: Header wie die sogenannte Content Security Policy legen fest, wie der Browser mit verschiedenen potenziell gefährlichen Inhalten wie JavaScript-Code umgeht. Sichere Werte für diesen Header können Angriffe (vor allem aus Kategorie *A7: Cross-Site Scripting*) massiv erschweren oder gleich verhindern; unsichere Werte (z. B.: `script-src: 'unsafe-inline'`) hingegen können Angriffe enorm vereinfachen.

Die sogenannten CORS-Header (Cross Origin Resource Sharing) legen fest, ob und wie Daten einer Web-Anwendung durch andere Web-Anwendungen abgerufen werden können. Auch hier gilt: Gute Werte können die Sicherheit stärken, unsichere Werte hingegen, die zugleich leider auch die Entwicklung erleichtern und somit häufiger vorkommen, können potenziell alle Nutzerdaten gegenüber anderen Seiten preisgeben und sind somit sehr gefährlich.

Dies sind nur zwei Beispiele – aufgrund der verschiedenen beteiligten Komponenten gibt es noch viele weitere. Leider bleibt in diesem Bereich aufgrund der Verschiedenheit nur, sich über die Best Practices bezüglich Security des eingesetzten Tech Stacks zu informieren und die Konfigurationswerte angemessen zu wählen. Infos zu Sicherheits-Headern finden sich unter [OWHED].

A7: Cross-Site Scripting (XSS)

Kategorien *A1*, *A4* und *A8* beinhalten Schwachstellen, bei denen Code in den Server injiziert und dort ausgeführt werden kann. Kategorie *A7: Cross-Site Scripting (XSS)* hingegen beinhaltet Schwachstellen, bei denen Code entweder in den Server injiziert oder von diesem an Nutzer zurückgespiegelt wird – und dann nicht serverseitig, sondern im Browser der Anwender zur Ausführung kommt. Der Name der Kategorie stammt daher, dass meistens Skript-Code zur Ausführung kommt und dieser eine Sicherheitsmaßnahme des Browsers umgeht, die verhindert, dass Code einer Seite auf einer anderen Seite ausgeführt werden kann.

Diese Schwachstellenkategorie ist deswegen besonders gefährlich, weil der Angreifer den Code der Webseite, der vom Browser des Nutzers ausgeführt wird, beliebig ändern kann. Er kann somit Darstellung und Funktionalität beliebig manipulieren und dadurch beispielsweise Zugangsdaten stehlen, Schadsoftware-Downloads injizieren, Funktionen im Nutzerkontext des Opfers und somit mit dessen Rechten ausführen, auf Daten des Nutzers zugreifen und vieles mehr. Schwachstellen dieser Kategorie ergeben sich vor allem dann, wenn Nutzereingaben dynamisch aus der letzten Anfrage oder aus der Datenbank in Webserver-Ausgaben integriert werden, beispielsweise wie folgt:

```
String resp = "<html><head>[...]<body>[...]<h3>Your name is: "
+ request.getParameter("name") + "[...]";
```

Hier könnte ein Angreifer nun einen Link wie folgt konstruieren: `https://anwendung.de/userinfo?name=<script>alert()</script>` und somit Skript-Code in die Webseite injizieren. Klickt der Nutzer auf diesen Link, wird der Skript-Code des Angreifers serverseitig integriert und kommt im Browser des Nutzers zur Ausführung.

Verhindern kann man XSS-Schwachstellen, indem man Werte, die dynamisch in Webseiten integriert werden, einem Output Encoding und Escaping unterzieht. Dies bedeutet: Zeichen, die einen HTML- oder JavaScript-Kontext manipulieren können, sollten von ihrer „Code-Form" in ihre reine Darstellungsform umcodiert werden. Beispielsweise kann das öffnende Tag-Zeichen < von HTML bei jedem Vorkommen in `<` umcodiert werden. Dem Nutzer wird dieses noch immer als offene spitze Klammer < angezeigt, durch den Browser jedoch nicht als Tag-öffnendes Zeichen interpretiert. Dies muss für alle im jeweiligen Zielkontext (HTML, JavaScript, Angular Templates...) gefährlichen Sonderzeichen entsprechend durchgeführt werden, wobei es Library-Funktionen gibt, auf die zurückgegriffen werden kann.

A8: Unsichere Deserialisierung

Auch hierbei handelt es sich um einen Sonderfall der Kategorie *A1*, der besonders gravierend ist: Java unterstützt das direkte Exportieren und Importieren von Objekten aus beziehungsweise in den Speicher. Dabei werden allerdings durch die JVM bestimmte Methoden, die per Standard immer bei der Deserialisierung aufgerufen werden müssen, ausgeführt, zum Beispiel `readObject`.

Bei bestimmten Objekten ist es nun so, dass durch geschickt gewählte Werte der Member-Variablen die Aufrufe bei der Deserialisierung so verkettet werden können, dass beliebiger Java-Code, der im zu deserialisierenden Objekt als Werte eingeführt wurde, zur Ausführung kommt. Die Folge ist vollständige Kontrolle über die Anwendung und oft, wenn auch nicht immer, über den gesamten Server. Da sowohl die Angriffe wie auch Schutzmaßnahmen komplizierter sind, werden sie an dieser Stelle nicht genauer behandelt. Mehr Infos finden sich unter [OWDES].

A9: Nutzung von Komponenten mit bekannten Schwachstellen

Hierbei handelt es sich um eine der konzeptionell einfachsten Kategorien der OWASP Top 10: In Kategorie *A9* werden Schwachstellen erfasst, die sich dadurch ergeben, dass Drittkomponen-

ten mit eigenen, bereits bekannten Schwachstellen eingesetzt werden. Es handelt sich also um veraltete Komponenten, die nicht selbst entwickelt wurden und deren Version nicht aktuell ist oder die keine Sicherheitsaktualisierungen von ihren jeweiligen Entwicklern mehr erhalten (d. h. EOL, End Of Life wurde erreicht).

Betroffen sein können Bibliotheken (DLL/SO/JAR), Backend- und Frontendkomponenten (d. h. auch JavaScript) oder Programme, die aufgerufen werden. Die sich ergebenden Risiken entsprechen genau denen, die sich durch die jeweiligen Schwachstellen ergeben, die in der jeweiligen betroffenen Drittkomponente vorhanden sind. Sie reichen also von „vernachlässigbar gering" bis „hochkritisch".

Der Schutz ist theoretisch einfach: Es sollten keine veralteten oder EOL-Komponenten zum Einsatz kommen, sondern diese regelmäßig und häufig aktualisiert werden. Veraltete Komponenten zu identifizieren, kann durch verschiedene Tools gut automatisiert werden, zum Beispiel in einem Artifact Repository, Code Repository wie GitHub/GitLab oder in einer CI/CD-Pipeline.

A10: Unzureichendes Logging & Monitoring

Ein oft anzutreffendes Problem ist, dass Betreiber einer Anwendung nur wenig Einblicke in die Geschehnisse ihrer Anwendung haben. Dies bedeutet insbesondere, dass nicht auffällt, wenn Angriffe oder unerlaubte Aktionen durchgeführt werden oder, falls diese bereits geschehen sind, nicht rekonstruiert werden kann,

- was die Folgen von diesen waren,
- was die Ursache war,
- wer der Angreifer war und
- wie dieser sich unbefugten Zugriff verschafft hat.

Alles in allem ist der Besitzer oder Betreiber einer Anwendung also oft blind und kann weder Angriffe erkennen, noch Gegenmaßnahmen ergreifen oder diese später aufklären.

Die Lösung hierzu ist, ein „angemessenes" Logging und Monitoring zu implementieren. Hierbei müssen verschiedene Dinge abgewogen werden: Granularität, Datenschutzgesetze, Datenspeicherdauer, Platzbedarf und weiteres. Alles in allem ist ein gutes Logging und Monitoring im Ernstfall jedoch essenziell – und viele merken dies erst, wenn es zu spät ist.

Schlusswort

Das Thema IT-Sicherheit und sichere Softwareentwicklung ist ein unglaublich vielseitiges und vielschichtiges, wie man auch an den hier vorgestellten OWASP (Web) Top 10 sieht, wobei nur ein kleiner Ausschnitt der Probleme und Lösungen behandelt werden konnte. So unterschiedlich die Technologien und Komponenten sind, sind auch die potenziellen Sicherheitslücken und deren Lösungen.

Es braucht daher auch ein Vorgehen von mehreren Seiten an das Problem der Sicherheit: Einerseits durch automatisierbare Maßnahmen (wie SAST und DAST – Static/Dynamic Application Security Testing, Scans, Dependency-Überprüfungen u. a.), andererseits durch menschliche Intelligenz, zum Beispiel durch Sicherheitstests, Code Audits und weitere. Vor allem ist Kenntnis dessen, was alles sicherheitstechnisch schief gehen kann, essenziell für die sichere Entwicklung – man kann sich nicht schützen gegen Dinge, die man nicht kennt oder versteht.

Der Autor hofft, mit diesem Artikel das Bewusstsein hierfür gestärkt und einen oberflächlichen Überblick über die wichtigsten Themengebiete der sicheren Web-Anwendungsentwicklung gegeben zu haben.

Literatur und Links

[JPRST] Oracle Java Documentation, Prepared SQL Statements in Java, https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html

[O1017] OWASP Web Top 10 2017, https://owasp.org/www-project-top-ten/

[OWDES] OWASP Cheat Sheet on Deserialization, https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

[OWHED] OWASP Page on HTTP Security Headers, https://owasp.org/www-project-secure-headers/#div-headers

[OWXXE] OWASP Cheat Sheet on XXE Prevention, https://cheatsheetseries.owasp.org/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.html

[SPSEL] Spring Security Documentation on Expression-Based Access Control, https://docs.spring.io/spring-security/site/docs/3.0.x/reference/el-access.html