Hochschule Wismar
University of Applied Sciences Technology, Business and Design
Fakultät für Ingenieurwissenschaften

Praktikumsarbeit

Erkennen von gefährlichen GET-Anfragen

Eingereicht am: 14. September 2022

von: Leon Kuffner
geboren am 10.06.2002
in Pritzwalk
Matrikelnummer: 360836

Betreuer: Dr. Olaf Hagendorf
Praktikumsbetreuer: Anna-Louise Grensing

Aufgabenstellung

ACHTUNG!
Die ausgehändigte Originalaufgabenstellung (und bei jeder Kopie die entsprechenden Kopie) wird ohne Seitenzahlangabe eingebunden. Bei deutschsprachigen Aufgabenstellungen wird der Titel in englischer Sprache wiederholt.

Für die digitale Fassung der Arbeit ist eine Schilderung der Aufgabenstellung aber durchaus sinnvoll und kann an dieser Stelle verfasst werden.

Kurzfassung

Maximal eine halbe Seite.

Abstract

English version.

Inhaltsverzeichnis

1 Einleitung

Einleitung in die Arbeit.

2 Hauptteil

2.1 Was sind HTTP-Methoden?

2.1.1 Zweck

Um den eigentlichen Kern dieser Arbeit zu erreichen, ist es wichtig, die Rahmenbedingungen, in denen man sich bewegt zu erläutern, zu definieren und abzugrenzen. Deswegen ist eine Definition von HTTP und der Funktionsweise der entsprechenden Methoden unumgänglich. Das Hypertext Transfer Protokoll (HTTP) gehört zur Familie der zustandslosen Response-/Request-Protokolle. Dieses arbeitet auf der Anwendungsebene und stellt eine generische Schnittstelle, um eine flexible Interaktion mit Informationssystemen zu ermöglichen, welche auf Basis des HTTP-Protokolls arbeiten. Dabei wird die eigentliche Implementierung des Systems verborgen. Hierzu stellt das entsprechende System eine einheitliche Schnittstelle, welche immer unabhängig vom bereitgestellten Ressourcentyp ist. Dabei muss der Server nicht einmal wissen, welchen Zweck der Client erfüllt, wichtig ist nur, dass die entsprechende Schnittstelle bekannt ist. Da Anfragen isoliert betrachtet werden, ist es möglich, die Implementierungen so zu gestalten, dass diese in möglichst vielen Kontexten verwendet werden können. Allgemein gesprochen ist das HTTP-Protokoll dafür da, Dokumente zu verschicken. Dabei gibt es zwei Rollen, den Client und den Server. Der Client ist in der Regel derjenige, der eine Konversation startet, indem er eine Anfrage an den Server stellt. Dies geschieht über die einheitliche Schnittstelle. Der Server antwortet dann auf die gestellte Anfrage. HTTP ist Text-basiert, alle Nachrichten sind also an sich nur Bits von Texten. Im Nachrichtenbody ist es dann möglich auch andere Dinge außer Text austauschen. Durch das Verwenden von Text ist es relativ einfach, den Austausch von Client und Server nachzuvollziehen. Eine HTTP-Nachricht besteht aus einem Header und einem Body. Der Body kann auch leer gelassen werden, enthält aber in der Regel die abgefragten Daten. Der Header hingegen enthält Metadaten, wie beispielsweise Verschlüsselungsinformationen. Im Falle einer Request enthält dieser außerdem die entsprechende HTTP-Methode.

2.1.2 Geschichte

Seit der Einführung von HTTP im Jahr 1990 ist es das primäre Übertragungsprotokoll für das World Wide Web. Dabei fing es relativ einfach an. Die einzige Methode war HTTP-GET, welche in der Lage war ein Hypertext-Dokument anzufordern, indem ein Pfad angegeben wurde. Durch das stetige Wachstum des Internets wurde das Protokoll zunehmend komplexer. So wurden unter anderem weitere Methoden hinzugefügt. So konnten zusätzlich auch noch Antworten formuliert werden. Seitdem wurde das Protokoll stetig weiterentwickelt, sodass heute drei Versionen existieren. So wurde in Version zwei eine gemultiplexte Sitzungsschicht auf TLS- und TCP-Protokollen eingeführt. So konnten gleichzeitige HTTP-Nachrichten mit effizienter Komprimierung ausgetauscht werden. HTTP/3 verbesserte diese Gleichzeitigkeit dann noch weiter, da anstelle von TCP das UDP-Protokoll verwendet wird. Alle drei Versionen haben als Basis dieselbe Semantik, unabhängig von der individuellen Weiterentwicklung.

2.1.3 Kern Semantik

HTTP bietet eine einheitliche Schnittstelle, welche es ermöglicht, mit einer Ressource zu interagieren. Als Ressource bezeichnet man das Ziel einer HTTP Request. Fast alle Ressourcen werden über einen sogenannten Unique Ressource Identifier (URI) identifiziert. Dabei sollen Ressourcenidentifikation und Anfragensemantik voneinander getrennt werden. Jede Nachricht ist entweder eine Anfrage oder eine Antwort. Dabei erstellt der Client Anforderungen, die dem Server die entsprechenden Absichten mitteilen. Diese Nachrichten werden mittels eines identifizierten Ursprungsservers weitergeleitet. Der Server wartet auf Anforderungen des Clients, analysiert JEDE Nachricht des Clients und analysiert die entsprechende Semantik. Auf Basis dieser Semantik kann der Server dann die entsprechenden Ressourcen identifizieren und eine Antwort an den Client formulieren. Eine Antwort kann dann aus einer oder mehreren Nachrichten bestehen. Empfängt der Client dann diese Nachricht, analysiert dieser die Antwort und guckt, ob die Antwort auch seine Absicht erfüllt hat. Der Client kann dann auf die empfangene Nachricht oder den Statuscode reagieren und entscheiden, was er als Nächstes tut. Da es sich bei HTTP um ein Protokoll handelt, welches nach einem Request-respond Modell arbeitet, sind die Abläufe immer gleich. Auf eine Request folgt immer eine Response, was zum einen der angeforderte Inhalt sein kann oder eben eine Fehlermeldung. Der Client teilt dann dem Server mit, ob er das Dokument auch erhalten hat.

2.2 Welche Methoden sind für uns relevant?

Die Quelle unserer Semantik ist der Token für die Request Methode, dieser teilt dem Server mit, für welchen Zweck diese spezifische Anfrage gestellt worden ist und was der Client erwartet, dass dieser die Antwort als valide erkennt. Anforderungsmethoden rufen eine Aktion auf, welche auf eine Zielressource angewendet werden soll. Diese standardisierten Methoden sollten in der Regel immer dieselbe Semantik haben, unabhängig von der aufrufenden Ressource. An sich entscheidet jede Ressource selber, ob eine bestimmte Semantik angewendet wird. Wichtig sind für diese Arbeit aber nur die HTTP-GET und die HTTP-POST Methoden.

2.2.1 HTTP-GET

Das GET-Verfahren forder die Übertragung einer Ressource an, in einer von der Methode gewünschten Darstellung. Eine erfolgreiche Antwort des Servers wird durch die ßameness"(Gleichheit) beschrieben. Um dies zu überprüfen, wird ein Identifier an unsere GET-Anfrage geknüpft, der wiederum eine Information in Form einer 200 (OK) Response liefern kann. Hierbei handelt es sich um einen Statuscode, welcher angibt, ob die Request erfolgreich war. Der Content, der in dieser 200 Response gesendet wird, hängt dann immer von der jeweiligen Request Methode ab. GET ist der primäre Mechanismus, um Informationen abzurufen. Anwendungen stellen in der Regel für jede wichtige Ressource eine URI, auf diese Weise wird eine Wiederverwendung durch andere Anwendungen gewährleistet. So entsteht eine Art Netzwerkeffekt, sodass das Web immer weiter wächst. Tatsächlich kann man sich die Ressource-identifiers als entfernte Dateisysteme vorstellen. Die Response ist dann sozusagen eine Art Kopie der entsprechenden Zieldatei. HTTP Interfaces für eine Ressource sind meistens so gestaltet wie ein Baum von Objekten. Selbst wenn die Schnittstelle direkt an ein Dateisystem gebunden ist, ist es möglich den Server so zu konfigurieren, dass dieser entsprechend der Anfrage diese Datei heraussucht bzw. auch ausführt. Die Files werden dann nicht direkt gesendet, sondern lediglich deren Output in Form eine Repräsentation. Am Ende jedoch muss nur der Server wissen, wie er die richtige Ressource findet und eine entsprechende Antwort formuliert. Der Inhalt, welcher in einer GET-Anfrage liegt, hat meistens keine eigene Semantik, was dazu führen kann, dass Anfragen abgelehnt werden, oder die Verbindung unterbrochen wird. Der Client sollte niemals Inhalt in einer GET-Request generieren, es sei denn, dies wurde explizit gefordert.

2.2.2 HTTP-POST

Die POST-Methode fragt bei der Zielressource an, ob die in der Methode eingeschlossenen Daten verarbeitet werden können. Diese Daten werden dann gemäß der Semantik der Ressource verarbeitet. Dabei wird POST für folgende Dinge verwendet:

> Bereitstellung von Daten für HTML-Formulare zur Weiterverarbeitung

> Posten einer Nachricht an eine Mailliste, Blog oder Schwarzes Brett

> Erstellen einer neue Ressource, welche dann noch beim Origin Server identifiziert werden muss

> Anhängen von Daten an eine vorhandene Ressource

Zudem gibt der Origin-Server ebenfalls eine Statuscode an. Dieser ist dann abhängig vom Ergebnis der Antwort. Wird eine Post-Request erfolgreich bearbeitet, eine oder mehrere Ressourcen auf dem Server erstellt, so sollte der Server als Antwort den Code 201 (Created) senden. Sollte die Ressource bereits existieren, so ist die Antwort 303 (see other). In diesem Fall erfolgt eine Umleitung auf die vorhandene Ressource. Nicht jede Eingabe ist immer so gestaltet, dass der Server diese beantworten kann. So kann es sein, dass Nutzer oder Angreifer versuchen, diese Methoden gezielt zu manipulieren, um beispielsweise Informationen zu stehlen oder sich Zugriffsrechte zu erschleichen. In diesem Fall würde man von sogenannter Code Injection sprechen, also dem Einschleusen von schädlichem Code. Wie solche Angriffe funktionieren und wie ein Programmierer dies verhindern kann, soll Thematik dieser Arbeit sein.

2.3 Wie wird eine Anfrage vom Webserver verarbeitet?

Nachdem alle diese Begriffe geklärt sind, ist es essenziell zu verstehen, wie der Webserver eine solche GET-Request überhaupt verarbeitet. Erst mit diesem Verständnis ist es möglich, in die Thematik der Code Injections einzusteigen. In meinem Fall ist es zu verstehen, welche möglichen Angriffsvektoren und Schwachstellen existieren. Jede Web Request beginnt mit einer URL. Diese beantwortet drei essenzielle Fragen. Um welche Art einer Request handelt es sich? Dies wäre dann das entsprechende Protokoll wie http. Wohin wird die Request gesendet? Das wäre dann der Netzwerkstandort bzw. der Name einer Webseite. Hierbei handelt es sich lediglich um eine

DNS (Domain Name System), hinter der sich eine IP verbirgt, welche dem Webserver zugeordnet ist. Was möchte ich abfragen? Hier wird ein Pfad angegeben und eine entsprechende Query.

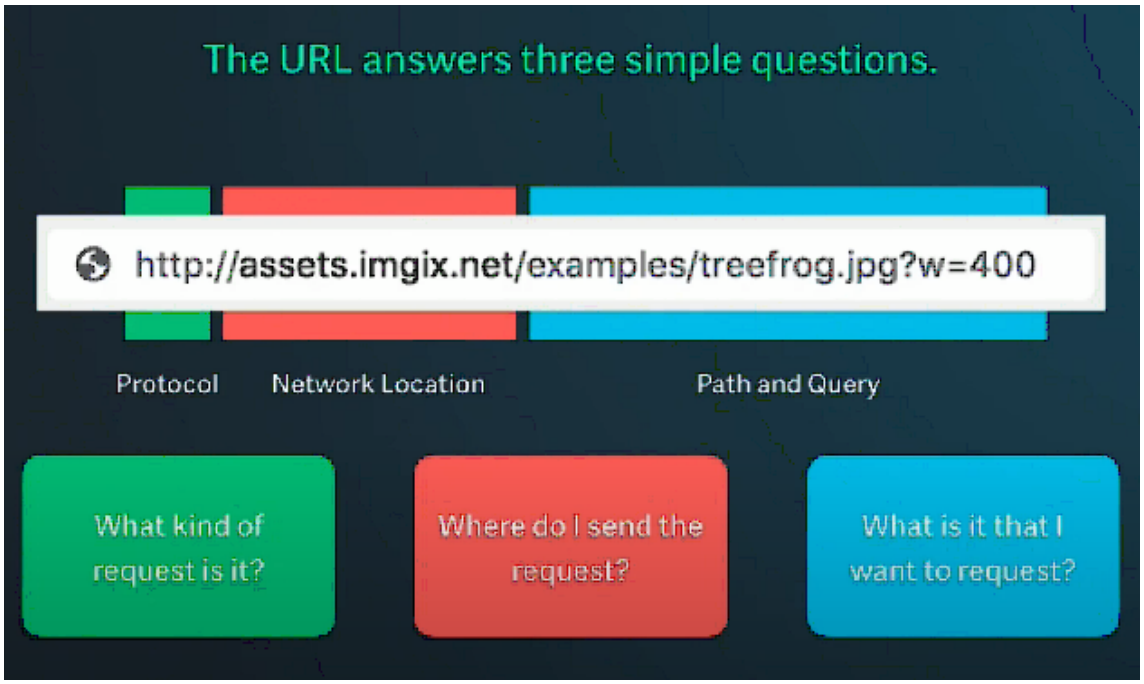

Abbildung 1: Coding Tech. How a Web Request Works, Down to the Atom

Der Browser baut dann aus unserer URL eine GET-Request. Die eigentlich Request beginnt ab dem /Example, daran angeschlossen folgt das entsprechende Protokoll, also in unserem Fall HTTP. Als Host wird dann ganz einfach die Adresse der Webseite angegeben, was dann ungefähr so aussehen würde:

```
GET /examples/treefrog.jpg?w=400 HTTP/1.1
Host: assets.imgix.net
User-AGENT: Curl/7.54.0
Accept: image/webp,*/*
Accept-Encoding: br,gzip,deflate
```

Abbildung 2: HTTP-GET vom Browser

Diese erzeugte Request wird dann vom Browser über das Internet versendet. Mit der entsprechenden Antwort kann dann eine Anzeige generiert werden. Wie das Routing zu dem entsprechenden Server funktioniert, soll nicht Teil dieser Arbeit sein. Wichtig ist jedoch zu wissen, dass zunächst eine Verbindung zum Webserver aufgebaut wird. Dies geschieht durch das TCP-Protokoll. Nachdem der Handshake ausgeführt wurde, kann der Client seine GET-Request versenden. Daraufhin kann

dann der Webserver eine Antwort in Form eines Codes (200) und der Antwort mit der entsprechenden Payload an den Client senden. Beim Server angekommen, wird die Anfrage entsprechend bearbeitet. Am Beispiel eines Warenhauses lässt sich dies ganz gut erklären. Zunächst einmal werden die Kundenwünsche an der Rezeption angenommen, im Falle eines Webservers wäre das dann zum Beispiel Apache oder Nginx. Die Anfrage wird dann entsprechend an Ärbeiter"weitergereicht. Diese sprechen dann eine bestimmte Programmiersprache und bauen dann die "Möbelßusammen. Hierfür werden zunächst alle Teile zusammengesucht, was das Besuchen einer Datenbank sein könnte. Die zusammengebauten Möbel repräsentieren zum Beispiel HTML, CSS oder JavaScript. Diese "Möbelßind dann der finale Output des Webservers. Der Output wird dann entsprechend an den Kunden bzw. den Client zugestellt, wo dieser dann die Seite anzeigen kann.

2.4 SQL - Injections (SQLI)

SQL-Injections gehören zur Familie der Code Injections. Hierbei geht es einem Angreifer darum, bestehende SQL-Querys zu erweitern oder diese zu manipulieren. So kann es sein, dass die Anwendung dem Angreifer mit mehr Informationen antwortet als einem vielleicht lieb ist. Dies können zum einen Fehlermeldungen sein, die vom Server oder vom Programm generiert worden sind, der Angreifer kann so darauf schließen, wie das System auf unterschiedliche Eingaben reagiert und zum anderen können dies auch Daten sein, die eigentlich nicht für die Augen von unbefugten Personen bestimmt sind oder eben auch weitreichende Kontrollen über Systeme oder Datenbanken. Deswegen gilt als allererste Faustregel für Anwendungen aller Art, dass Variablen, welche durch Nutzereingaben bestimmt werden, niemals ungeprüft in das Programm übernommen werden dürfen! Ist dies nicht der Fall, können Angreifer diese Schwachstelle ausnutzen, um beispielsweise Authentifizierungen oder Anmeldungen zu umgehen oder um den gesamten Inhalt einer Datenbank auszulesen. Gefährdet sind Webseiten, welche Datenbanken, wie z.Bsp. MYSQL, Oracle, SQL Server oder viele weitere benutzen. SQL-Injections zählen zu den ältesten, am weitesten verbreiteten Sicherheitslücken in Webanwendungen. So stehen SQLI auf Platz eins der Top 10 der häufigsten Sicherheitslücken bei Webanwendungen bei der OWASP (Open Web Application Security Project) Organisation. Wie funktioniert jedoch dieser Angriff? Zunächst einmal suchen Angreifer nach potenziell gefährlichen User Inputs auf einer Webseite oder einer Webapplikation. Eine Anwendung mit einer Schwachstelle benutzt den User Input, um SQL Queries zu generieren. Dies wird dann oftmals "malicious payload"genannt. Diese Payload ist der Schlüssel zur

Attacke. Nachdem dieser Content gesendet wurde, können schädliche SQL Befehle in der Datenbank ausgeführt werden. SQL ist eine Query Sprache, welche dafür entwickelt wurde, Daten in relationalen Datenbanken zu verwalten. SQL kann zum Ändern, abrufen, oder löschen von Daten verwendet werden. Die meisten modernen Webseiten speichern den meisten Inhalt in Datenbanken. In seltenen Fällen ist es sogar möglich, dass man mittels SQL Befehle auf dem Host System ausführen kann. Angreifer können SQL-Injections dazu verwenden, um Nutzer zu imitieren oder gar um Root Rechte zu erlangen. Außerdem ist es möglich, von einer Datenbank aus alle weiteren Datenbanken auf einem Server zu übernehmen. Durch das Löschen oder Stehlen von Daten kann sehr großer Schaden angerichtet werden. Der Worst Case wäre wohl, wenn dadurch das Hostsystem übernommen wird, sodass dann das interne Netzwerk angegriffen werden kann. Mit der Zeit haben sich SQL Attacken ziemlich weit entwickelt, weswegen es viele verschiedene Varianten gibt, um Daten zu extrahieren. Die häufigsten Typen werden in den folgenden Abschnitten erklärt.

2.4.1 Blind SQL-Injection (Boolean)

Die erste Variante ist eine Variante der Blind SQL Injection. Blind SQL Injections kommen dann zum Tragen, wenn eine SQL Schwachstelle vermutet wird, man aber keine Rückgaben zum Beispiel in Form von Fehlermeldungen erhält. Diese Form funktioniert sogar dann, wenn die Datenbank keinen direkten Output generiert. Zum Beispiel bei einem Loginforum wird die Seite lediglich neu geladen, wenn ein SQL-Fehler auftritt. Der Angreifer bekommt also keine Informationen darüber, wie erfolgreich seine Angriffe waren. Es kann jedoch sein, dass die Website auf Basis von Booleschen Werten unterschiedliche Ausgaben generiert. Existiert beispielsweise ein User, dann wird dieser auf der Website begrüßt. Existiert der User nicht bzw. fügt man eine falsche Aussage an das Ende der SQL Injection ein, wird einem mitgeteilt, dass der User nicht existiert. Allein mit dieser Information ist es möglich, Informationen aus der Datenbank zu lesen. Jedoch lassen sich nur 1 Bit große Informationen auslesen. Hier ist der Aufwand zwar groß, jedoch lässt sich folgender Vorgang auch automatisieren.

AND SUBSTRING (password,1,1) = ‘a’

AND SUBSTRING (password,1,1) = ‘b’

Dieser Vorgang wird dann für alle Buchstaben so oft wiederholt, bis ein True zurückkommt. So lassen sich Passwörter mittels SQLI ganz einfach auslesen.

2.4.2 Blind SQl-Injection (Time)

Eine weitere Variante der Blind SQL Injection verläuft zeitgesteuert. Hier wird ganz einfach die Zeit gemessen, die eine Website benötigt, um auf eine SQL-Abfrage zu reagieren und um den entsprechenden Inhalt zu generieren. Auch hier kann wieder nur ein Bit Information auf einmal ausgelesen werden. TRUE würde bedeuten, dass die Anfrage lange gebraucht hat, ein FALSE hingegen bedeutet, dass die Antwort relativ kurz ist. Wie in dem Beispiel zuvor müssen wir eine Bedingung an das SQL Statement knüpfen, in diesem Fall ist es jedoch zeitgesteuert. Ein solcher Funktionsaufruf für eine MySQL Datenbank wäre zum Beispiel die SLEEP (time) Funktion. Da wir keinerlei Informationen geliefert bekommen, haben wir also noch die Möglichkeit zu schauen, ob eine Time Based SQL Injection möglich ist. Ein Angriff könnte dann in etwa so aussehen:

AND IF (1=1, SLEEP (10), NULL)

Sollte die Antwort dann tatsächlich zehn Sekunden lange dauert, hat man eine Time Based Schwachstelle entdeckt, welche dann nur noch ausgenutzt werden muss. So kann man versuchen, die unterschiedlichen Antwortzeiten zu interpretieren, um so auf Passwörter oder andere Dinge in der Datenbank zu schließen.

2.4.3 Out of Band Data Exfiltration

Out of Band SQL-Injections sind eine wenig verbreitete Variante der SQL-Injections, weil diese darauf basiert, dass auf dem Datenbankserver gewisse Funktionen aktiviert sind, welche dann von der Webanwendung benutzt werden. Diese Variante wird dann verwendet, wenn der Angreifer nicht über denselben Kanal, den er verwendet, um anzugreifen, auch die Informationen extrahieren kann. Die Out-of-Band Methode bietet hier eine Alternative zu den Zeitbasierten Angriffen. Insbesondere bei instabilen Datenbankservern ist diese Methode sinnvoll, da zeitbasierte Angriffe unzuverlässig sein können. Out of Band SQLI verlässt sich auf die Fähigkeit des Datenbankservers, DNS- oder aber auch HTTP-Requests zu stellen. Auf diese Weise können Daten an den Angreifer geliefert werden. So gibt es beim Microsoft SQL Server beispielsweise den Befehl ”xp_dirtree”, welcher es ermöglicht DNS Anfragen an einen Server zu stellen. Dieser Server kann dann vom Angreifer kontrolliert werden, sodass dieser die Anfrage einfach auslesen kann. Im Falle einer Oracle Datenbank gibt es das Paket ÜTL_HTTP”. Dieses kann verwendet werden, um HTTP Anfrage an einen Server zu senden, welcher dem Angreifer gehört.

2.4.4 Error based SQL-Injection

Die letzte Variante, die ich hier vorstellen möchte, ist eine Fehlerbasierte SQL Injection. Dabei werden Fehlermeldungen clever ausgenutzt, sodass diese statt eines Fehlers Daten aus der Datenbank zurückgeben. So kann ganz leicht überprüft werden, ob zum Beispiel das Passwort eines Nutzers falsch ist, bzw. das richtige kann ganz einfach ausgegeben werden. Diese Variante gilt als eine Sonderform der SQL-Injection, da man sich hier Fehlermeldungen in Form von Stacktraces oder Errot Messages zunutze macht. Auch hier werden diese Fehlermeldungen ungeprüft an die Datenbank, die Applikation oder den Webserver weitergereicht. Diese Fehlermeldungen werden dann durch das Senden von falschen Eingaben erzwungen. Die Fehlermeldungen können dann Informationen darüber enthalten, welches Datenbanksystem verwendet wurde, wie dieses intern strukturiert ist oder wie die Applikation aufgebaut ist. Mit diesen Informationen kann der Angreifer seinen Angriff planen oder auf weitere mögliche Angriffsvektoren schließen. Die Informationen können von Tabellennamen oder Spaltennamen bis hin zu reinen Informationen wie Zugangsdaten reichen. Oftmals reicht bereits ein einziger Angriff, um Daten zu ex filtrieren oder um die gesamte Datenbank einfach zu übernehmen. Zwar sind Fehlermeldungen in der Entwicklung sehr hilfreich, sollten aber in der Live-Version deaktiviert sein. Das entsprechende Logging von Fehlermeldungen sollte ausschließlich in einem geschütztem Bereich vorgenommen werden, bzw. die entsprechenden Logfiles sollten Zugangsbeschränkungen unterliegen. Diese Variante ist die neben Blind- und Union-based SQL-Injections die am häufigsten auftretende Variante. Werden mehrere Abfragen ineinander verschachtelt, um auf diese Weise an Fehlermeldungen zu gelangen, so spricht man von sogenannten Double Query SQL-Injections.

2.5 Cross Site Scripting (XSS)

Laut der OWASP Foundation befindet sich die SQL-Injection auf Platz Eins der am häufigsten auftretenden Angriffe auf Webapplikationen. Nicht weniger gefährlich, wenn nicht noch gefährlicher, befindet sich auf Platz drei (Stand 2020) XSS. XSS ist eine Abkürzung für das Cross Site Scripting (CSS, um eine Verwechslung mit der Styling-Sprache CSS zu vermeiden). Wie SQLI gehört XSS zur Familie der Code Injection, also auch hier wieder das Einschleusen von schädlichem Code in Webanwendungen. Grund dafür ist auch hier wieder das nicht ausreichende Überprüfen von Variablen, welche hauptsächlich durch den Nutzer eingegeben werden. Eingeschleust wird in diesem Fall JavaScript Code, welcher durch den Webbrowser weiter

interpretiert wird. So ist es einem Angreifer möglich, JavaScript Code im Browser des Opfers auszuführen. Auf diese Weise können Sitzungen vollständig übernommen werden, Websites gefälscht, Phishing Angriffe durchgeführt oder sogar Browser benutzt werden, um Malware zu installieren. Das Gefährliche dabei ist, dass JavaScript Zugriff auf das DOM hat, weswegen es kaum Grenzen gibt, was mit einem solchen Angriff möglich ist. In diesem Fall ist nicht nur eine Datenbank gefährdet und indirekt auch die Nutzer, sondern wirklich jeder, der die Website besucht. Die Browser führen den Code nämlich automatisch aus, es sei denn die Ausführung von JavaScript wurde in den Browsereinstellungen deaktiviert. Ziemlich typisch für XSS ist es, dass Angreifer ihren Code meistens so positionieren, dass dieser bei Zugriffen automatisch ausgeführt wird, sodass ein Besucher der Webseite nichts davon bemerkt. Eines der häufigsten Ziele ist es, die Session-ID oder Cookies zu stehlen, sodass ganze Sitzungen mithilfe der Session-ID übernommen werden können. Um diese Thematik besser zu erläutern, hier vielleicht einmal ein kleines Beispiel. Ein Angreifer versucht zum Beispiel den Nutzer dazu zu bringen, auf einen Link zu klicken, der entweder auf der gefälschten Webseite platziert worden ist oder per E-Mail versendet wurde. Dieser Link wurde mithilfe von XSS auf der Webseite eingeschleust. Nun zeigt die Seite eine Fehlermeldung oder ein Login Fenster an, welches den Nutzer auffordert seine Daten einzugeben. Klickt das Opfer nun auf einen Link oder gibt Logindaten an, erfolgt eine Umleitung auf eine andere Webseite. Diese gefälschte Seite läuft dann höchstwahrscheinlich auf einem Webserver, der dem Angreifer gehört. Hier kann dann der Code ganz bequem im Hintergrund ausgeführt werden. XSS ist sogar möglich, wenn die Verbindung durch SSL verschlüsselt ist. Dies ist nur möglich, wenn der Code bereits vor dem Verbindungsaufbau in einer Webseite platziert wird. Auf diese Weise ist es sogar möglich, die Cookies zu modifizieren. Generell gilt, dass mit XSS kein Webserver angreifbar ist, sondern lediglich der Besucher von einer Webseite. Wie bereits bekannt, hat JavaScript Zugriff auf das Document Object Model. Im DOM sind alle Informationen über die aktuell angezeigten Dokumente enthalten. So kann man beispielsweise mit folgendem Script sämtlich Cookies auslesen und diese in einem Textdokument speichern.

<script> fs.writeFile('cookies.txt', document.cookie) </script>

2.5.1 Server XSS

Serverseitiges XSS tritt dann auf, wenn nicht vertrauenswürdige Nutzereingaben in die vom Server generierten HTTP-Responses eingebunden werden. Diese Daten

können zum einen aus der Request stammen, aber auch von einem Speichermedium. Hier kann man dann von Reflected Server XSS und auch von Stored Server XSS sprechen. In diesem Fall befindet sich die Schwachstelle auf der Seite des Servers, bzw. in dessen Code. Der Browser gibt dann einfach die erzeugte Antwort wieder, inklusive aller Skripte, die darin eingebettet sind.

2.5.2 Client XSS

Client-XSS tritt auf, wenn nicht vertrauenswürdige Nutzereingaben verwendet werden, um das DOM mit unsicheren JavaScript-Aufrufen zu updaten. Ein JavaScript Aufruf wird als unsicher bezeichnet, wenn dieser dazu verwendet werden kann. Um validen JavaScript-Code in das DOM Element einzufügen. Die Datenquelle hierbei könnte das DOM selber sein, oder vom Server gesendet worden sein. Dies kann dann eine wiederum ein AJAX Aufruf sein oder das einfache Laden einer Seite. Ultimativ können diese Daten dann aus einer Request stammen oder von einer festen Datenquelle. Wichtig ist hierbei, dass dies vom Client als auch vom Server stammen kann. Deswegen kann man hier von Reflected Client XSS und Stored Client XSS sprechen. Durch diese neue Einführung ändert sich jedoch nicht die Definition von DOM-based XSS. Beim Dom-based XSS handelt es sich nur um eine Teilmenge von Clien-XSS. Datenquelle ist hierbei der DOM und nicht der Server. Sowohl Server-XSS als auch Client-XSS können Stored als auch Reflected sein, weswegen das Ganze in der folgenden Abbildung einmal dargestellt ist, um eine Verwirrung zu vermeiden.

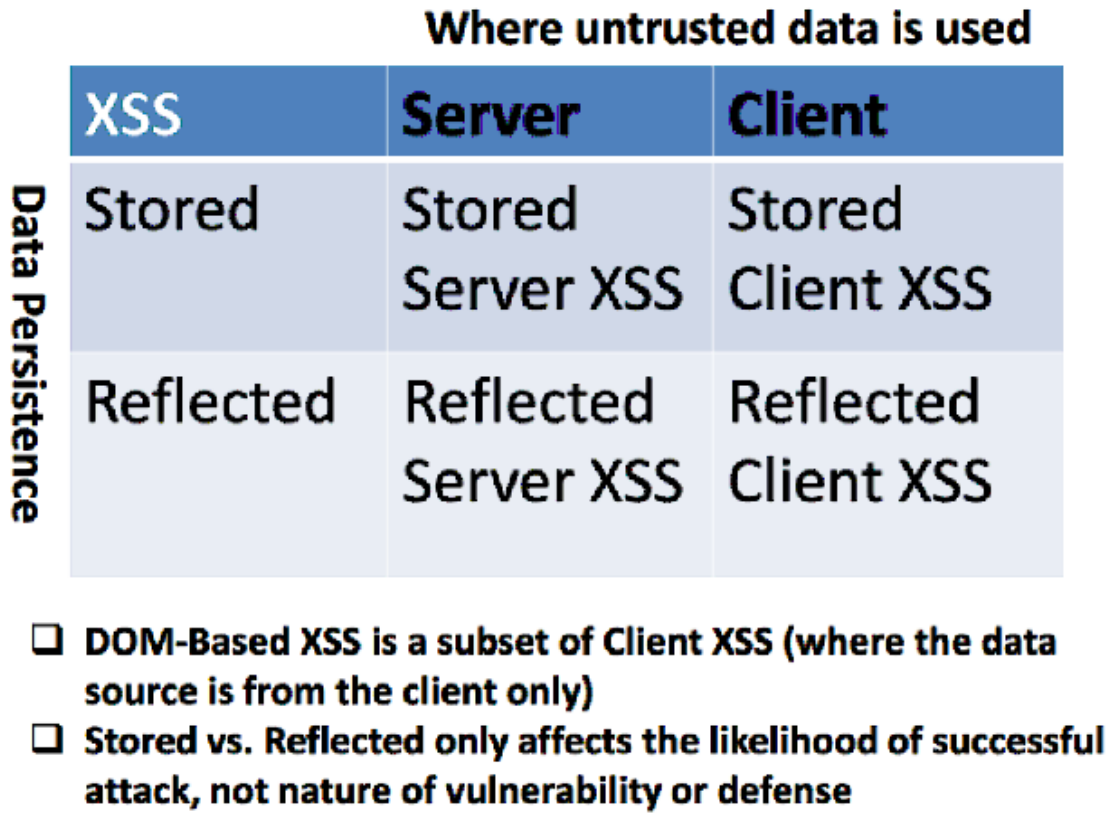

Abbildung 3: OWASP.org. Types of Cross Site Scripting

2.5.3 Reflected XSS

Reflektierte (nicht persistente) XSS Angriffe können auftreten, wenn eine schadhafte Payload an eine angreifbare Webanwendung gesendet wird. Diese Anfrage wird dann reflektiert, die HTTP-Response vom Server besteht dann aus der Payload. Angreifer nutzen hierfür Social Engineering Techniken wie Phishing, um Opfer dazu zu bringen Schadcode auszuführen. Dieser Schadcode wird dann an die Anfrage an den Webserver angehangen. Der Browser des Opfers führt anschließend den Schadcode in Form einer HTTP-Response aus. Reflected XSS wird als nicht persistent bezeichnet, da jedes Opfer die Anfrage mit dem schadhaften Payload selber senden muss. Von daher wird versucht, so viele Opfer wie möglich abzudecken, sodass ein Erfolg mit der Menge der Opfer quasi garantiert ist. Angriffe dieser Form zielen vor allem darauf ab, dass das Opfer Eingaben in Eingabeformulare tätigt. Um eine große Abdeckung zu gewährleisten sind die häufigsten Ziele Nachrichtenforen, Fehlermeldungen oder Ergebnisseiten von Suchmaschinen. Diese Art erfordert es nicht, dass der Angreifer eine Lücke in einer Webanwendung findet und diese entsprechend

ausnutzt. Bösartige Skripte werden also nicht dauerhaft eingeschleust. Perfekt nachgebildete Loginseiten können so ganz einfach Nutzerdaten stehlen.

2.5.4 Stored XSS

Bei einem Stored XSS Angriff, empfängt eine Webanwendung Nutzereingaben aus nicht vertrauenswürdigen Quellen und speichert diese. Diese Inhalte werden dann in spätere HTTP-Responses aufgenommen und entsprechend ausgeführt. Um einen solchen Angriff durchzuführen, muss ein Angreifer zunächst eine Schwachstelle ausmachen. Diese Schwachstelle befindet sich im Backend einer Anwendung, sodass eine Ausführung von bösartigen Anfragen möglich wird. Ist man erst einmal in der Anwendung, so muss gar keine Methode entwickelt werden, um nicht vertrauenswürdige Eingaben an den Server zu übermitteln. Stored XSS wird als Typ 1 oder als persistenter XSS Angriff bezeichnet und beruht auf nicht sanitisierten Nutzereingaben für Skripte, welche dauerhaft auf dem Zielserver gespeichert sind. Böswillige Nutzer können auf diese Weise die Kontrolle darüber erlangen, wie der Browser des Opfers ein Skript ausführt. Die Auswirkungen solcher Angriffe können von leichten bis zu vollständigen Kompromittierungen reichen. Dies hängt meistens davon ab, wieviele Rechte der betroffene Nutzer auf dem Server hat. Folgen eines solchen Angriffs können Session-Hijacking, Privilegienerweiterungen, Spoofing von Webinhalten und vieles mehr sein. Der Unterschied zum Reflected XSS ist, dass der Server den bösartigen Inhalt ausführt und das Ergebnis an eine HTTP-Response anfügt. Beim Stored XSS wird beliebiger Code auf einem Webserver gespeichert. Im Gegensatz zu anderen XSS Varianten, welche voraussetzen, dass der Nutzer zum Zeitpunkt des Angriffes angemeldet ist, wird beim Stored XSS, die auf dem Webserver gespeicherte schädliche Payload vom Browser für JEDEN Benutzer der sich anmeldet ausgeführt. Somit ist diese Variante wesentlich gefährlicher.

2.5.5 DOM Based XSS

DOM-basiertes XSS ist eine Cross Site Scripting Schwachstelle, welche es dem Angreifer ermöglicht, die Browser Umgebung eines Clients so zu ändern, dass Schadsoftware eingeschleust werden kann. Da diese Variante auf dem DOM basiert, ist die Ausführung des Codes unmittelbar nach dem Laden einer Seite. HTML Quellcode und die HTTP-Response bleiben hierbei unverändert, was eine Erkennung wesentlich schwieriger macht. Die Schadsoftware wird in der Browser-Umgebung eines Clients

gespeichert, außerdem ist der Schadcode deswegen nicht in der Response des Webservers enthalten. Angriffe dieser Art können nur erkannt werden, wenn das DOM und die clientseitigen Skripte noch zur Laufzeit überprüft werden. Angriffe erfolgen hauptsächlich auf Anwendungen mit einem ausführbaren Pfad, welcher Daten von einer Quelle zu einer sogenannten Senke ßinkümleitet. Quelle können hierbei JavaScript Attribute sein. Diese dienen als Speicherort für schädliche Eingaben. Der entsprechende Code wird dann in den ßinksäusgeführt. Beispiele hierfür sind ëval", ßetTimeoput", ".innerHTMLünd viele weitere.

2.6 Härtungsmaßnahmen

Hier dann die Fortsetzung mit weiteren Unterpunkten

2.6.1 Härtungsmaßnahmen gegen SQL-Injections

2.6.2 Härtungsmaßnahmen gegen Cross Site Scripting

3 Zusammenfassung und Ausblick

Rückblick, Bewertung, Ausblick über mögliches Fortführen der Arbeit

A Beispielanlage

Beispieltext.

Literaturverzeichnis

[1] Justin Clarke. *SQL Hacking*. unbekannt. Franzis Verlag GmbH, 2016. ISBN: 978-3-645-60466-6.

[2] Michael Kofler. *Hacking und Security*. 2. Auflage. Rheinwerk Computing, 2022. ISBN: 978-3-8362-7191-2.

[3] Claudia Eckert. *IT-Sicherheit*. 10. Auflage. Gruyter, Walter de GmbH, 2020. ISBN: 978-3-1105-5158-7.

[4] Kevin Beaver. *Hacken für Dummies*. 4. Auflage. Wiley-VCH, 2017. ISBN: 978-3-5278-1904-1.

[5] R. Fielding. *RFC 9110*. 2022. URL: https://www.rfc-editor.org/rfc/rfc9110.html.

[6] Ian Muscat. *Out of Band SQLI*. 2022. URL: https://www.acunetix.com/blog/articles/sqli-part-6-out-of-band-sqli/.

[7] St172317. *Error Based SQL Injection*. 2022. URL: https://it-forensik.fiw.hs-wismar.de/index.php/Error_Based_SQL_Injection.

[8] mootoki(Github). *Types of Cross Site Scripting*. 2022. URL: https://owasp.org/www-community/Types_of_Cross-Site_Scripting.

[9] Crashtest Security. *Reflektierte Cross-Site-Scripting-Schwachstelle*. 2022. URL: https://crashtest-security.com/de/reflektierter-xss-angriff/.

[10] Crashtest Security. *Was ist Stored XSS und wie kann man es verhindern*. 2022. URL: https://crashtest-security.com/de/stored-xss-luecke/.

Bildverzeichnis

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.
Die Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Ich erkläre ferner, dass ich die vorliegende Arbeit in keinem anderen Prüfungsverfahren als Prüfungsarbeit eingereicht habe oder einreichen werde.

Die eingereichte schriftliche Fassung entspricht der auf dem Medium gespeicherten Fassung.

Ort, Datum Unterschrift

Thesen

Praktikumsarbeit
Erkennen von gefährlichen GET-Anfragen

Eingereicht am:	14. September 2022
von:	Leon Kuffner geboren am 10.06.2002 in Pritzwalk
Matrikelnummer:	360836
Betreuer:	Dr. Olaf Hagendorf
Praktikumsbetreuer:	Anna-Louise Grensing

- These 1
- These 2
- …