Web security for developers

Real threats, practical defense

Injection attacks

- SQL injection

Injection attacks

- SQL injection
  1. Benutze parameterisierte Statements (parameterized statements):

```
sql = „SELECT * FROM users WHERE email = ? and
encrypted_password = ?“;
statement.executeQuery(sql, email, password);
```

Hier kümmert sich der Datenbank-Treiber darum, eventuelle Steuerzeichen sicher zu behandeln.

Injection attacks

- SQL injection
  2. Benutze object-relational mapping (ORM) Bibliotheken

     Aber achte auf eventuelle Hintertüren:

```
def find_user(email, password)
   User.where(„email=‘“ + email + „‘ and
   encrypted_password=‘“ + password + „‘“)
end
```

     Dieses Beispiel wäre angreifbar, wenn man z.B. das Passwort auf „‘ OR 1 = 1,“ setzt:

```
SELECT * FROM users WHERE email = ‚billy@gmail.com‘ AND encrypted_password = ‚‘ OR 1 = 1
```

Injection attacks

- SQL injection
  3. Folge dem Prinzip der geringsten Privilegien

     welches verlangt, dass sämtliche Prozesse und Anwendungen nur mit den Rechten laufen, welche sie benötigen um ihre erlaubten Funktionen auszuführen – und nicht mehr.

     Das schränkt auch den Schaden ein, den ein Angreifer anrichten kann.

Injection attacks

- Command injection

Injection attacks

- Command injection
  1. Steuerzeichen escapen

     Alle Eingaben von HTTP-Requests müssen sorgfältig escaped werden, wenn sie im Code Verwendung finden.

Injection attacks

- Remote code execution

Injection attacks

- Remote code execution

  Ein Angreifer kann remote code execution erreichen, wenn er eine Sicherheitslücke in einem bestimmten Webserver entdeckt und mit diesem Wissen exploit scripts schreibt, um damit alle Webserver dieses Typs anzugreifen.

Injection attacks

- Remote code execution
  1. Deaktiviere code execution während der Deserialisierung. Sobald ein Webserver Deserialisierung verwendet, um eingehende Daten von HHTP-Requests zu verarbeiten, muss er sämtliche verwendeten Serialisierungsbibliotheken entschärfen, indem er jegliche Möglichkeit zur Code-Ausführung deaktiviert – üblicherweise durch entsprechende Konfigurationseinstellungen, die es deinem Webserver erlauben, Daten zu deserialisieren ohne Code auszuführen.

Injection attacks

- File upload vulnerabilities

Injection attacks

- File upload vulnerabilities

  Anatomie eines file upload-Angriffs:

  Erzeuge eine Datei „hack.php“

```
<?php
if (isset($_REQUEST[,cmd‘])) {
   $cmd = ($_REQUEST[,cmd‘]);
   system($cmd);
} else {
   echo „What is your bidding?“;
}
?>
```

  und lade sie als dein Profilbild auf einen Webserver. PHP-Dateien werden vom OS üblicherweise als ausführbare Dateien behandelt, was der Schlüssel zu diesem Angriff ist. Ganz offensichtlich ist eine Datei mit der Endung .php keine gültige Bilddatei, aber der Angreifer kann ja einfach eine JS-Dateityp-Prüfung während des Hochladens deaktivieren.
  Ein Angriff könnte dann so aussehen:

```
https://cdn.example.com/1a2fe/hack.php?cmd=cat+/etc/mysql/my.cnf
```

Injection attacks

- File upload vulnerabilities
  1. Hoste Dateien auf einem abgesicherten System – z.B. Cloudflare oder Akamai. Das schiebt die Aufgabe, auf Sicherheit zu achten, zu einem Dritten.

Injection attacks

- File upload vulnerabilities
  2. Stelle sicher, dass hochgeladene Dateien nicht ausgeführt werden können.

Injection attacks

- File upload vulnerabilities

  3. Validiere den Inhalt hochgeladener Dateien

     Stelle sicher, dass der content-type-header im HTTP request des Uploads dem erwarteten Dateityp entspricht, aber beachte, dass ein Angreifer den header auch leicht fälschen kann.

Cross-site scripting attacks

- Stored cross-site scripting attacks

  Durch einen Kommentar in einem Formularfeld kann ein Angreifer JS-code injizieren.

Cross-site scripting attacks

- Stored cross-site scripting attacks
  1. HTML-Zeichen („ &‘ < >) escapen
     Insbesondere Templates escapen üblicherweise interpolierte Zeichen ohne explizit dazu angewiesen zu werden. Alle sicheren templating-Sprachen folgen dem selben Designprinzip: Die templating engine escaped dynamische Inhalte implizit, so lange der Entwickler nicht explizit entscheidet, „raw HTML“ zu erzeugen.

Cross-site scripting attacks

- Stored cross-site scripting attacks
  2. Implementiere eine content security policy

     Moderne Browser gestatten Websites, eine content security policy zu setzen, mit deren Hilfe man JS-Ausführung auf der Website verhindern kann. Durch das Setzen einer csp in den HTTP response headern kannst Du dem Browser befehlen, niemals inline-JS auszuführen, außer wenn es via eines src-Attributs im <script>-Tag importiert wurde.
     Ein typischer csp header sieht folgendermaßen aus:

```
Content-Security-Policy: script-src ,self‘ https://apis.google.com
```

Cross-site scripting attacks

- Stored cross-site scripting attacks

```
Content-Security-Policy: script-src ‚self‘ https://apis.google.com
```

Diese policy definiert, dass Skripte von derselben Domain (‚self‘) und von der Domain apis.google.com importiert werden können, aber inline-JS nicht ausgeführt werden darf.
Die csp einer Site kann auch in einem <meta>-tag im <head>-Element im HTML der Webseiten angegeben werden.
Durch das Whitelisting einer Domain, von welcher der Browser Skripte lädt, definierst Du implizit, dass inline-JS nicht erlaubt ist. Um inline-JS zu gestatten, muss das keyword „unsafe-inline“ in die csp eingefügt werden.

Cross-site scripting attacks

- Reflected cross-site scripting attacks

  Betrachten wir die Google-Suchseite: Wenn Du eine Suche nach „cats“ durchführst, übergibt Google den Suchbegriff als Bestandteil der HTTP in der URL:

  ```
  https://www.google.com/search?q=cats
  ```

  Der Suchbegriff cats wird im Suchfeld über den Suchergebnissen angezeigt. Wäre Google nun eine weniger sichere Firma, wäre es möglich, den cats-Parameter in der URL durch schädliches JS zu ersetzen und diesen Code ausführen zu lassen, wann immer jemand diese URL in ihrem Browser öffnet. Ein Angreifer könnte diese URL als Link per E-Mail an sein Opfer schicken, oder einen Website-Besucher dazu bringen, die URL zu besuchen, indem er sie in einem Kommentar hinterlässt. Das ist die Essenz eines *reflected* cross-site scripting Angriffs: Ein Angreifer (bzw. Opfer) schickt den gefährlichen Code im HTML-Request, und der Server reflektiert ihn dann zurück.

Cross-site scripting attacks

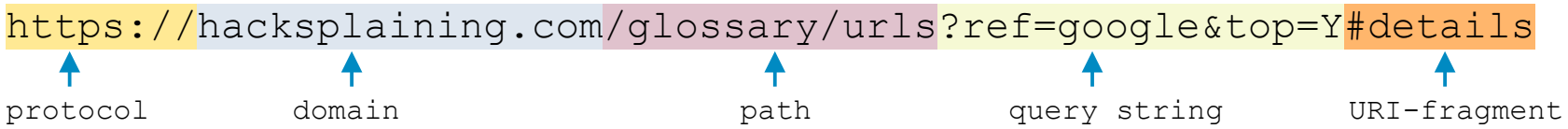
- Reflected cross-site scripting attacks
  1. Escape dynamischen Content von HTTP requests

     Hier greifen die selben Regeln wir bei der Vermeidung von „normalen“ cross-site scripting-Angriffen.

Cross-site scripting attacks

- DOM-basierte cross-site scripting attacks

  Hier schmuggeln Angreifer schädliches JS in die Website eines Users mit Hilfe des URI-Fragemts:



Cross-site scripting attacks

Browser senden von Haus aus keine URI-Fragmente zum Server, wenn der Browser die Seite anfordert. Das bedeutet, dass URI-Fragmente dem Server-seitigen Code nicht zur Verfügung stehen. Durch die Absicherung des Server-seitigen Codes kann man DOM-basierte XSS-Angriffe nicht verhindern. Client-seitiges JS-Code, der URI-Fragmente interpretiert und benutzt muss also vorsichtig dabei sein, wie es den Kontext dieser Fragmente verwendet.

Cross-site scripting attacks

DOM-basiertes XSS ist eine relativ neue Form von Angriffen, und besonders gefährlich, weil die Code-injection vollständig Client-seitig geschieht und nicht dadurch entdeckt werden kann, dass man Server-Logs durchsucht!
Dies bedeutet, dass man sich über diese Angriffsform und seiner Vermeidung besonders im Klaren sein muss, wenn man Code Reviews durchführt.

1. Escape dynamischen Content aus URI-Fragmenten

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs

  Üblicherweise startet ein Angreifer einen CSRF-Angriff durch Ausnutzung von Websites, bei denen GET-Requests den State des Webservers verändern.

  In einer frühen Version von Twitter konnte man Tweets per GET-Request erzeugen. Dieses Versäumnis machte Twitter anfällig für CSFR-Angriffe: Man konnte so URL-Links erzeugen, die – wenn angeklickt – auf der Timeline eines Users einen Post erzeugen:

  ```
  https://twitter.com/share/update?status=in%20ur%20twitter%20CSRF-ing%20ur%20tweets
  ```

  Ein kluger Hacker hat dieses Schlupfloch dann benutzt, um einen viralen Wurm auf Twitter zu erzeugen.

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs

  Da er einen GET-Request verwenden konnte, um einen Tweet zu schreiben, konstruierte er einen bösartigen Link, der – wenn angeklickt – einen obszönen Tweet mit dem selben Link postete. Wenn nun Leser dieses Tweets den Link anklickten, den das erste Opfer getweetet hat, wurden sie selbst zum Opfer dieses Wurms.

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs
  1. Folge den REST-Prinzipien

     Stelle sicher, dass Deine GET-Requests nicht den Zustand des Servers verändern. Deine Website sollte GET-Requests nur verwenden, um Webseiten oder andere Resourcen zu anzufordern.

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs
  2. Implementiere Anti-CSRF-Cookies

     Falls Deine Website sensible Aktionen als Reaktion auf POST-Requests ausführt, musst Du CSRF-Cookies verwenden um sicherzugehen, dass diese Requests sicher von innerhalb Deiner Website initiiert wurden. Sichere Websites verwenden anti-CSRF-Cookies um sicherzustellen, dass POST-Requests von Seiten stammen, die auf derselben Domain gehostet werden. HTML-Seiten dieser Website fügen dasselbe String-Token in einem `<input type="hidden" name="_xsrf" value="5978e29d4ef434a1" />`-Element in jedem verwendeten Formular hinzu, welches POST-Requests erzeugt.

     Schickt ein User das Formular ab, und der _xsrf-Wert des Cookies unterscheidet sich vom _xsrf-Wert im Request-Body, dann verwirft der Server den Request.

     Der Browser sendet das Cookie nur, wenn die Seite von derselben Domain geladen wurde.

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs

  Die meisten modernen Webserver unterstützen Anti-CSRF-Cookies. Hier ein Beispiel für den Tornado-Webserver in Python:

```
<form action="/new_message" method="post">
  {% module xsrf_form_html() %}
  <input type="text" name="message"/>
  <input type="submit" value="Post"/>
</form>
```

  In diesem Beispiel generiert die xsrf-Funktion ein zufälliges Token und schreibt es als hidden input in das HTML-Formular. Der Tornado-Webserver schreibt dieses Token dann in die HTTP-Response-Header als `Set-Cookie: _xsrf=5978…` Wenn der User das Formular abschickt, validiert der Webserver, dass das Token aus dem Formular und im return Cookie Header übereinstimmen. Das Sicherheitsmodell des Browsers sendet das Cookie gemäß der same-origin policy zurück, so dass die Werte nur vom Server gesetzt worden sein können.

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs

  Anti-CSRF-Cookies sollten auch zur Validierung von HTTP-Requests verwendet werden, die mittels JS erzeugt sind. So können auch PUT- und DELETE-Requests abgesichert werden. Der JS-Code muss das Token aus dem HTML parsen und es mit dem HTTP-Request an den Server zurück schicken.

Cross-site request forgery attacks

- Anatomie eines CSRF-Angriffs
  3. Verwende das SameSite-Cookie-Attribut

     Wird das SameSite-Attribut beim Setzen eines Cookies verwendet, dann wird der Browser in einem Request zu Deiner Site alle Cookies entfernen, wenn dieser Request von einer anderen Domain stammt:

     ```
     Set-Cookie: _xsrf=5978e29d4ef434a1; SameSite=Strict;
     ```

     Es ist eine gute Idee, das SameSite-Attribut in allen Cookies zu verwenden, nicht nur solchen, die der CSRF-Absicherung dienen. Allerdings: Wenn Du Cookies für das Session-Management benutzt, wird hiermit das Session-Cookie auch von allen Requests entfernt, die von anderen Websites zu Deiner Website erzeugt werden. Dadurch erfordern dann alle eingehenden Links zu Deiner Seite, dass die Userin sich erneut einloggen muss, auch wenn sie bereits eine Session auf Deiner Website offen hat.

     Daher gibt es den Wert Lax für das SameSite-Attribut, der anderen Websites erlaubt, nur bei GET-Requests Cookies mitzusenden:

     ```
     Set-Cookie: _xsrf=5978e29d4ef434a1; SameSite=Lax;
     ```

Compromising Authentication

- Brute-Force attacks

Die Top 200 der am häufigsten genutzten Passwörter von 2020

Hier sind die schlechtesten Passwörter 2020. In der Liste wird angegeben, wie oft das Passwort kompromittiert und verwendet wurde und wie lange es dauern würde, es zu knacken. Durchsuche die Liste und stell sicher, dass deine Passwörter nicht dabei sind.

Position	Passwort	Anzahl der Benutzer	Benötigte Zeit zum Passwort-Knacken	Anzahl der Hacks
1.	123456	2,068,643	Weniger als eine Sekunde	23,597,311
2.	123456789	1,242,790	Weniger als eine Sekunde	7,870,694
3.	12345678	953,688	Weniger als eine Sekunde	2,944,615
4.	password	858,534	Weniger als eine Sekunde	3,759,315
5.	1234567	837,534	Weniger als eine Sekunde	2,516,606
6.	123123	806,419	Weniger als eine Sekunde	2,238,694

Compromising Authentication

- Brute-Force attacks
  1. Verwende Authentifizierung durch Drittanbieter

Compromising Authentication

- Brute-Force attacks
  1. Verwende Authentifizierung durch Drittanbieter

     Das sicherste Authentifizierungssystem ist dasjenige, welches Du nicht selbst schreiben muss. Überlege, ob Du Deinen Usern nicht ermöglichen möchtest, sich mit ihren Social-Media-Accounts anzumelden. Das ist bequem für die User und entbindet Dich von der Last, jemals User-Passwörter speichern zu müssen.

     Die meisten großen verfügbaren Authentifizierungsdienste basieren auf den open authentication (Oauth) oder OpenID-Standards.

Compromising Authentication

- Brute-Force attacks
  2. Verwende Single Sign-On

Compromising Authentication

- Brute-Force attacks

2. Verwende Single Sign-On

   Wenn Du einen OAuth- oder OpenID-Provider zur Authentifizierung verwendest, dann verwenden Deine User üblicherweise ihre private E-Mail-Adresse als Usernamen.
   Wenn jedoch Deine Zielgruppe Geschäftskunden sind, ziehe besser einen single-sign-on (SSO) Identitäts-Provider in Betracht, wie z.B. Okta, OneLogin oder Centrify. So können die Angestellten sich nahtlos unter ihrer Geschäts-E-Mail-Adresse in Drittanwendungen einloggen. So kann die Firmenadministratorin die ultimative Kontrolle über die verschiedenen Zugriffsrechte behalten.

Compromising Authentication

- Brute-Force attacks

  Um den Dienst eines single-sign-on Providers zu nutzen, muss man üblicherweise die so genannte Security Assertion Markup Language (SAML) benutzen, die ein etwas älterer und weniger Benutzerfreundlicher Standard ist als Oauth oder OpenID, aber die meisten Programmiersprachen verfügen über ausgereifte SAML-Bibliotheken, die man hierfür verwenden kann.

Compromising Authentication

Sichere dein eigenes Authentifizierungssystem ab

Compromising Authentication

Sichere dein eigenes Authentifizierungssystem ab

Auch wenn die Systeme von Drittanbietern normalerweise sicherer sein dürften als dein eigenes System, gehen diese doch mit Nachteilen einher, denn nicht jeder Nutzer verfügt über einen Social Media- oder ein Gmail-Konto. Daher benötigt man für diese in jedem Fall die Möglichkeit eine Login mit selbst gewähltem Usernamen und Passwort.

Compromising Authentication

Du musst mindestens eine E-Mail-Adresse und ein Passwort speichern. Wird das Userprofil auf der Website angezeigt, ist zusätzlich ein Username erforderlich.
Die einzige 100% sicher Methode ist, eine E-Mail mit einem Verifizierungs-Link an die Adresse zu schicken. Bis dahin sollte man keine weiteren Mails an die betreffende Adresse senden oder sie gar zu Mailinglisten hinzufügen!

Compromising Authentication

Sofern so genannte Wegwerfadressen verwendet werden, um andere User zu belästigen, kann es notwendig sein, auch diese nicht zu erlauben. Hierfür gibt es gut gepflegte Blacklists, um solche Adressen während des Anmeldeprozesses zu erkennen.

Compromising Authentication

Passwort-Resets senden einen Reset-Link an die gespeicherte, validierte E-Mail-Adresse und erlauben dem User, ein neues Passwort einzugeben. Diese Links sollten nach Verwendung verfallen und auch nicht länger als z.B. 30 Minuten gültig sein, um zu verhindern, dass ein Angreifer versucht, alte Links durchzutesten, selbst wenn er Zugang zu einem fremden E-Mail-Account erlangt und nach solchen Reset-Mails sucht.

Compromising Authentication

Verlange sichere Passwörter, die Ziffern und Symbole ebenso enthalten wie Groß- und Kleinbuchstaben. Du solltest zumindest eine Länge von 8 Zeichen erwarten – je länger, umso besser. Studien zeigen, dass die Passwort-Länge wichtiger ist als die Verwendung spezieller Zeichen. Am Ende ist es besser, den User zu komplexen Passwörtern zu bewegen, als diese zu erzwingen – z.B. durch eine Anzeige der „Stärke“ eines gerade eingegebenen Passworts.

Compromising Authentication

Verwende zum Speichern der Passwörter eine schnelle, aber nicht zu schnelle Hashing-Funktion. Schreibe Deinen Hashing-Algorithmus idealerweise so, dass die Anzahl der Durchläufe einfach erhöht werden kann, um die Hashes stärker und zeitaufwändiger zu machen, sobald die Rechenkapazität ausreichend billiger geworden ist. `Bcrypt` ist ein hierfür gut geeigneter Algorithmus:

```
hashed = bcrypt.hashpw(password, bcrypt.gensalt(rounds=14))
```

Compromising Authentication

Zum Schutz vor Rainbow-Attacks, müssen die Hashes „gesaltet“ werden. Verwende idealerweise für jedes Passwort einen neuen Salt-Wert und speichere ihn zusammen mit dem Hash. Damit müsste ein Angreifer für jeden neuen Salt-Wert eine komplett neue Rainbow-Table erzeugen, was diese Art von Angriffen schlichtweg zu zeitaufwändig macht.

Compromising Authentication

Passwort-basierte Systeme sind immer anfällig für Brute-Force-Angriffe. Um die Website wirklich abzusichern, kann man Multifaktor-Authentifizierung (MFA) verwenden, zu welcher ein User sich immer mit Faktoren aus zwei von drei Gebieten ausweisen muss: Etwas, das sie *weiß*, etwas das sie *besitzt* und etwas, das sie *ist*. Ein Beispiel ist ein Geldautomat, welcher die PIN (etwas, das die Kundin weiß) und die Bankkarte (etwas, das sie besitzt) verlangt.

Compromising Authentication

Für Websites läuft dies normalerweise auf Username und Passwort (etwas, das die Userin kennt) und die Bestätigung, dass ein Authentifikator auf dem Smartphone installiert ist (etwas, das die Userin besitzt).
Damit benötigt ein Angreifer Kenntnis der Anmeldedaten UND Zugriff auf das Smartphone der anzugreifenden Person, was eher eine unwahrscheinliche Kombination ist.

Compromising Authentication

Stelle außerdem sicher, dass Deine User sich auch ausloggen können – viele Geräte werden von verschiedenen Personen genutzt. Deine Logout-Funktion sollte das Session-Cookie im Browser löschen und den Session-Identifier ungültig machen, falls dieser auf dem Server gespeichert wird. Das schützt vor Angreifern, die es schaffen, Session-Cookies zu lesen und damit versuchen, eine Session wieder aufzunehmen.

Compromising Authentication

Stelle außerdem sicher, dass Deine User sich auch ausloggen können – viele Geräte werden von verschiedenen Personen genutzt. Deine Logout-Funktion sollte das Session-Cookie im Browser löschen und den Session-Identifier ungültig machen, falls dieser auf dem Server gespeichert wird. Das schützt vor Angreifern, die es schaffen, Session-Cookies zu lesen und damit versuchen, eine Session wieder aufzunehmen.

Compromising Authentication

Nummeriere Deine User nicht durch, um sie nicht erratbar zu machen. Login-Seiten teilen oft mit, wenn ein Username bereits vergeben ist. Verwende daher immer eine Fehlermeldung, aus der nicht hervorgeht, ob es nun der Username oder das Passwort ist, welches falsch eingegeben wurde.

Compromising Authentication

Auch Timing-Angriffe können auf Login-Seiten durchgeführt werden. Wenn die Website das eingegebene Passwort nur dann überprüft (und hasht), falls der Username korrekt ist, kann dies durch Messung der HTTP-Response-Time erkannt werden.
Errechne also immer das Passwort-Hash, auch wenn der eingegebene Username noch nicht vergeben bzw. ungültig ist.

Compromising Authentication

Und die Passwort-Reset-Funktion sollte nicht mitteilen, ob die E-Mail mit dem Reset-Link erfolgreich versendet wurde, weil so abgefragt werden kann, ob eine E-Mail-Adresse auf einer Website benutzt wird oder nicht. Halte die Nachricht immer neutral, z.B. „Schau in Deinem Postfach nach“.

Compromising Authentication

Zusätzlich kann die Verwendung von CAPTCHAS die meisten Angriffsversuche abschrecken.

Session Hijacking

Wenn ein Hacker auf Session-Informationen zugreifen oder sie fälschen kann, die der Browser schickt, kann er auf jeden Useraccount auf der Website zugreifen. Glücklicherweise haben moderne Webserver sicheren Session-Management-Code, der es einem Hacker praktisch unmöglich macht, eine Session zu manipulieren oder fälschen. Doch selbst wenn es keine Sicherheitslücken in den Session-Management-Fähigkeiten eines Servers gibt, kann ein Hacker die gültige Session einer anderen Person stehlen, während sie noch aktiv ist. Dies nennt man **Session Hijacking**.

Session Hijacking

Um zu verstehen, wie ein Angrefier eine Session entführt, musst Du zuerst verstehen was passiert, wenn ein User und ein Webserver eine Session eröffnen.

Beim Login unter HTTP erteilt der Webserver der Userin einen Session Identifier, den der Browser mit jedem darauffolgenden HTTP-Request übermitteln muss, damit der Server die HTTP-Konservation mit der authentifizierten Userin weiterführen kann.

Der Server speichert den Session State zusammen mit der Session-ID – z.B. den Inhalt des Warenkorbs.

Session Hijacking

- Server-Side Sessions

  Im traditionellen Modell des Session Management hält der Server den Session State im Speicher, und beide – der Webserver und der Browser übermitteln den Session Identifier hin und her. Dies nennt man eine Server-Side Session.
  Historisch haben Webserver damit experimentiert, die SessionID auf verschiedenen Wegen zu übermitteln:

  - In der URL
  - Als HTTP-Header
  - Im body eines HTTP-Request

Session Hijacking

- Server-Side Sessions

  Der mit Abstand häufigste und zuverlässigste Mechanismus ist jedoch die Übermittlung der SessionID als Session Cookie. Cookies wurden bereits 1995 von Netscape als Teil des HTTP eingeführt. Server-Side Sessions sind stark verbreitet und üblicherweise sehr sicher. Sie haben jedoch eine Limitierung in der Skalierbarkeit, weil der Webserver den Session State im Speicher vorhalten muss. Das bedeutet, dass zum Zeitpunkt der Authentifizierung nur ein Server von der aufgebauten Session weiß. Geht ein Folgerequest für die gleiche Userin an einen anderen Server, muss dieser in der Lage sein, die wiederkehrende Userin zu erkennen. Daher greifen Webserver üblicherweise auf einen Shared Cache oder eine Datenbank zu, von welcher jeder Server bei jedem Request den gecachten Session State lesen kann. Beide Varianten sind Zeit- und Ressourcenlastig und limitieren so die Responsiveness von Websites mit großen Userzahlen.

Session Hijacking

- Client-Side Sessions

  Aufgrund der schlechten Skalierbarkeit für große Websites wurden Client-Side Sessions entwickelt. Dabei wird der gesamte State im Session-Cookie (meist als JSON) gespeichert, was die vorgenannten Probleme der Server-Side Sessions behebt.
  Client-Side Sessions lassen sich natürlich hervorrragend manipulieren. Daher müssen die Daten darin derart geschützt werden, dass Manipulationen nicht gelingen.

Session Hijacking

- Client-Side Sessions

  Z.B. indem die Daten verschlüsselt werden, bevor der Server sie an den Client schickt. Dadurch werden die Daten Client-seitig komplett intransparent, und Manipulationen können vom Server sofort erkannt und der User ausgeloggt werden.

  Eine andere Maßnahme ist das mitsenden einer Prüfsumme. Dabei können die Daten jedoch immer noch von einem neugierigen Hacker gelesen werden. Achte also darauf, auf solche Art keine Daten zu speichern, die von Userinnen oder Hackern nicht gelesen werden können sollen.

Session Hijacking

- Cookie-Diebstahl

  Cookies können aus dem Cookie-Header einer authentifizierten Userin gestohlen werden. Dies kann auf drei verschiedene Arten geschehen.

Session Hijacking

- Cookie-Diebstahl
  1. Cross-Site Scripting

     Angreifer nutzen hierbei JS-Code, der in den Browser der Userin injiziert wurde, um ihre Cookies auszulesen und an einen vom Hacker kontrollierten externen Server zu schicken. Der Hacker erntet dann diese Cookies sobald sie auftauchen und kopiert sie in eine Browser-Session oder fügt sie einem Skript hinzu, um Aktionen unter der Session der Userin durchzuführen.

Session Hijacking

- Cookie-Diebstahl
  1. Cross-Site Scripting

     Abhilfe gegen diese Art von Angriff schafft die Markierung aller Cookies als `HttpOnly` im `Set-Cookie` header. Dadurch verhindert der Browser, dass JS-Code auf die Cookies zugreifen kann:

     ```
     Set-Cookie: session_id=278283910977381992837; HttpOnly
     ```

Session Hijacking

- Cookie-Diebstahl
  2. Man-in-the-Middle Angriff

     Dabei sitzt der Angreifer zwischen dem Browser und dem Webserver und kann den gesamten Netzwerkverkehr mitlesen.

Session Hijacking

- Cookie-Diebstahl
  2. Man-in-the-Middle Angriff

     Abhilfe gegen diese Art von Angriff schafft die Verwendung von HTTPS. Nachdem Du HTTPS aktiviert hast, solltes Du Deine Cookies als „secure“ markieren, damit der Browser sie niemals unverschlüsselt über HTTP verschickt:

     ```
     Set-Cookie: session_id=278283910977381992837; Secure
     ```

Session Hijacking

- Cookie-Diebstahl
  3. Cross-Site Request Forgery

     Ein Angreifer, der CSRF anwendet, benötigt keinen Zugriff auf das Session-Cookie einer Userin. Statt dessen bringt er sie dazu, einen Link zu Deiner Website anzuklicken. Sofern die Userin bereits eine Session zu Deiner Website offen hat, schickt der Browser ihr Session-Cookie zusammen mit dem HTTP-Request, wodurch ggf. eine sensible Aktion ausgeführt werden kann – z.B. Likes für ein Produkt zu vergeben, das der Angreifer anpreisen möchte.

Session Hijacking

- Cookie-Diebstahl
  3. Cross-Site Request Forgery

     Um diese Art von Angriffen zu verhindern, vershehe Deine Cookies mit dem `SameSite`-Attribut:

     ```
     Set-Cookie: session_id=278283910977381992837; SameSite=Strict
     ```

     Damit werden sämtliche Cookies von allen HTTP-Requests entfernt, die von externen Websites aus getriggert werden -
     was störend sein kann, wenn Deine Userin Deinen Content über Social Media teilen möchte, da jeder, der diesen Link anklickt, gezwungen wird, sich erneut anzumelden.

Session Hijacking

- Cookie-Diebstahl
  3. Cross-Site Request Forgery

     Diese Unannehmlichkeit lässt sich beheben, indem Du den Browser so konfigurierst, nur bei GET-Requests von externen Websites das Cookie nicht zu enfernen:

     ```
     Set-Cookie: session_id=278283910977381992837; SameSite=Lax
     ```

     Und wir haben ja gelernt, dass GET-Requests den Zustand des Servers nicht verändern sollten, womit auch Session-Cookies in diesem Fall keinen Schaden anrichten können.

Session Hijacking

- Cookie-Diebstahl

  Session Fixation

  In den frühen Tagen des Internet beherrschten viele Browser keine Cookies. Daher wurde gerne URL-Rewriting verwendet, um die SessionID an jede besuchte URL anzuhängen. So können sie in Logfiles landen – manche Server mögen vielleicht aufgrund veralteter Konfigurationen solche SessionIDs noch zulassen.

Session Hijacking

- Cookie-Diebstahl

  Session Fixation

  Dann gibt es noch die so genannte Session Fixation. Wenn ein dafür anfälliger Webserver auf eine unbekannte SessionID trifft, bittet er die Userin, sich zu authentifizieren, um dann eine Session mit dieser ID zu erstellen. Dadurch kann ein Angreifer eine solche Session im Voraus fixieren, um anschließend einen verführerischen Link mit dieser fixierten SessionID zu verbreiten. Sobald dies geschieht, kann der Angreifer die selbe URL in seinem Browser verwenden.

Session Hijacking

- Cookie-Diebstahl

  Session Fixation

  Die Unterstützung von URL-Rewriting sollte also in jedem Fall abgeschaltet werden. Sie dient keinem Zweck und erlaubt Angriffe durch Session Fixation.

Session Hijacking

- Cookie-Diebstahl

  Schwache SessionIDs

  Echte Zufallszahlen sind durch Software schwer zu erzeugen. Die meisten Zufallsgeneratoren verwenden Umgebungsfaktoren wie die Systemzeit als Seeds, um die Zufallszahlen zu generieren. Wenn ein Angreifer eine ausreichende Zahl von Seedwerten bestimmen (oder sie auf einen eingeschränkten Bereich möglicher Werte beschränken) kann, kann er potenziell valide SessionIDs aufzählen und sie auf Deinem Server ausprobieren.

Session Hijacking

- Cookie-Diebstahl

  Schwache SessionIDs

  Konsultiere die Dokumentation Deiner Server-Software, um sicherzustellen, dass der Server ausreichend lange, nicht erratbare SessionIDs verwendet, die durch einen starken Zufallsgenerator erzeugt werden.

  Und lies regelmäßig Security Advisories, damit Du weißt, wann Du Verwundbarkeiten Deines Web-Stacks patchen musst.

Kommende Themen:

Permissions
Information Leaks
Encryption
Third Party Code
XML-Attacks
Don’t be an accessory
DOS-Attacks