FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Sicherheitsanalyse und Handlungsempfehlungen für Ruby on Rails-Anwendungen: Risiken und Best Practices

Autor:
Giorgio Cantavenera, 3576179

Betreuer:
Prof. Dr. Philipp Rohde
M. Sc. Marcel Crolla

Aachen, 14. Dezember 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Sicherheitsanalyse und Handlungsempfehlungen für Ruby on Rails-

Anwendungen: Risiken und Best Practices

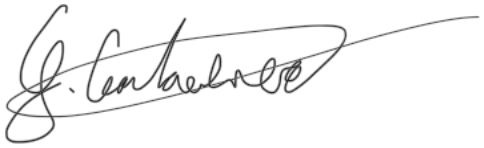
selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Giorgio Cantavenera

Aachen, den 14.12.2024

Unterschrift der Studentin / des Studenten



Abstract

In der heutigen digitalen Landschaft nimmt die Sicherheit von Webanwendungen einen zunehmend zentralen Stellenwert ein. Diese Arbeit untersucht die Sicherheitsrisiken und -maßnahmen in Ruby on Rails, einem populären Framework zur Webentwicklung. Zunächst wird ein Überblick über häufige Sicherheitsrisiken wie Cross-Site Scripting (XSS), SQL-Injection und Cross-Site Request Forgery (CSRF) gegeben, sowie Rails-spezifische Schutzmechanismen analysiert. Der Einsatz von Sicherheits-Gems wie Brakeman, Devise und RuboCop wird diskutiert, um Schwachstellen frühzeitig zu identifizieren und zu beheben.

Darüber hinaus werden die Herausforderungen durch unsichere Konfigurationen und Drittanbieter-Gems beleuchtet, gefolgt von praktischen Handlungsempfehlungen, um Sicherheitsrisiken in der Entwicklung und im Betrieb zu minimieren. Abschließend werden Strategien zur langfristigen Sicherheitsverbesserung und Risikominimierung, wie die Integration von Sicherheitsprozessen in CI/CD-Pipelines und das proaktive Patch-Management, vorgestellt.

Die Arbeit zeigt, dass trotz moderner Sicherheitsmechanismen und Best Practices viele Risiken bestehen bleiben, insbesondere durch komplexe Angriffsvektoren und neue Schwachstellen in Softwarebibliotheken. Ein Ausblick auf zukünftige Entwicklungen verdeutlicht die Notwendigkeit eines kontinuierlichen Sicherheitsmanagements, um der dynamischen Bedrohungslandschaft gerecht zu werden.

Inhaltsverzeichnis

1 Einleitung

Die Sicherheit von Webanwendungen hat angesichts der zunehmenden Häufigkeit und Komplexität von Cyberangriffen an Bedeutung gewonnen. Ruby on Rails, ein populäres Framework für die Webentwicklung, bietet zahlreiche integrierte Sicherheitsfeatures, erfordert jedoch ein fundiertes Verständnis der möglichen Schwachstellen, um diese effektiv einzusetzen. Ziel dieser Seminararbeit ist es, die Sicherheitsrisiken von Ruby on Rails systematisch zu analysieren und Handlungsempfehlungen für eine sichere Entwicklung und den Betrieb von Anwendungen abzuleiten.

Ruby on Rails, eingeführt im Jahr 2004, basiert auf der Programmiersprache Ruby und hat sich durch sein Model-View-Controller-Paradigma sowie seine Entwicklerfreundlichkeit einen festen Platz in der Webentwicklung erarbeitet. Es ermöglicht die schnelle Erstellung von Prototypen und Anwendungen durch umfangreiche Standardfunktionen und eine große Auswahl an Erweiterungen, sogenannten Gems. Diese Flexibilität ist jedoch auch eine potenzielle Schwachstelle, da unsichere Konfigurationen oder veraltete Gems Angriffsflächen bieten können [Har22, OWA22].

Die Relevanz dieses Themas wird durch die zunehmende Nutzung von Webanwendungen unterstrichen, die sensible Daten wie persönliche Informationen oder Zahlungsdetails verarbeiten. Sicherheitsverletzungen können nicht nur finanzielle und rechtliche Konsequenzen haben, sondern auch das Vertrauen der Nutzer erheblich beeinträchtigen. Die Herausforderung liegt darin, nicht nur funktionale Anforderungen zu erfüllen, sondern auch Sicherheitsmaßnahmen proaktiv zu integrieren.

Diese Arbeit beleuchtet typische Sicherheitsrisiken und Schutzmechanismen von Ruby on Rails und bietet praxisnahe Empfehlungen für Entwickler, um eine robuste Sicherheitsstrategie zu implementieren. Ziel ist es, eine Grundlage zu schaffen, die nicht nur aktuelle Sicherheitsanforderungen erfüllt, sondern auch künftigen Bedrohungen standhält.

2 Sicherheitsanalyse in Ruby on Rails

2.1 Überblick über die häufigsten Sicherheitsrisiken

2.1.1 Cross-Site Scripting (XSS)

Laut den OWASP Top Ten gehört Cross-Site Scripting zu den häufigsten Sicherheitsrisiken moderner Webanwendungen. [OWA21] Angreifer schleusen über Sicherheitslücken schädlichen Code in eine Webseite ein, der im Browser des Benutzers ausgeführt wird und beispielsweise dazu genutzt werden kann, Cookies zu stehlen, Sitzungen zu kapern, das Opfer auf eine gefälschte Website umzuleiten, Werbung zum Vorteil des Angreifers anzuzeigen oder Elemente der Webseite zu verändern, um vertrauliche Informationen zu erhalten. Im Folgenden werden verschiedene Angriffszenarien und ihre potenziellen Auswirkungen beschrieben.

Reflected XSS (Reflektierter XSS) Beim Reflektierten XSS-Angriff werden Clientdaten vom Server direkt in die Webseite geladen, ohne das sie gespeichert werden. Dieser Angriff wird häufig per Link auf einer externen Webseite oder in einer E-Mail-Nachricht eingebettet. Wenn ein Benutzer dazu verleitet wird, auf einen bösartigen Link zu klicken, ein speziell gestaltetes Formular abzuschicken oder einfach nur eine bösartige Website zu besuchen, wird der injizierte Code an die anfällige Website gesendet, die den Angriff an den Browser des Benutzers zurückspiegelt. Der Browser führt dann den Code aus, da er von einem „vertrauenswürdigen“ Server stammt.
Als Beispiel könnte ein Angreifer ein Url mit einem eingebettet XSS Payload generieren, wie etwa:

```
1 http://example.com/search?q=<script>alert('XSS!')</script>
```

Stored XSS (Gespeichertes XSS) Stored XSS Angriffe sind solche, bei denen das injizierte Skript dauerhaft auf den Zielservern gespeichert wird, beispielsweise in einer Datenbank, in einem Nachrichtenforum, in einem Besuchsprotokoll oder im Kommentarfeld. Das Opfer ruft dann das bösartige Skript vom Server ab, wenn es die gespeicherten Informationen anfordert.

DOM-based XSS (DOM-basiertes XSS) DOM-based XSS tritt auf, wenn die XSS-Schwachstelle im clientseitigen JavaScript liegt und nicht direkt in der Kommunikation zwischen Client und Server. In diesem Szenario wird das JavaScript auf der Client-Seite so manipuliert, dass schadhafter Code ausgeführt wird, der den DOM des Dokuments verändert. Der Angriff erfolgt durch Manipulation des DOM-Baums und nicht durch eine serverseitige Reaktion.

Die Auswirkungen eines erfolgreichen Cross-Site Scripting (XSS)-Angriffs können schwerwiegend sein. Angreifer können sensible Daten wie Session-Cookies oder Login-Credentials stehlen und dadurch die Kontrolle über Benutzerkonten übernehmen. Dies kann zur Manipulation von Benutzerkonten führen, wie dem Ändern von Einstellungen oder dem Auslösen unerwünschter Aktionen. Persistente XSS-Angriffe können zudem Schadcode auf der Anwendung selbst speichern, der alle Nutzer betrifft. Darüber hinaus besteht die Gefahr, dass XSS genutzt wird, um Malware zu verbreiten oder interne Netzwerke des Opfers zu kompromittieren [Fou23a].

2.1.2 SQL Injection

SQL-Injection ist eine Technik, bei der ungesicherte Benutzereingaben in Datenbankabfragen eingebunden werden, um diese absichtlich zu manipulieren und so unbefugten Zugriff auf die Datenbank zu verschaffen. Infolgedessen kann ein Angreifer in der Lage sein, auf besonders geschützte Bereiche der Anwendung zuzugreifen, sämtliche Daten aus der Datenbank abzurufen, vorhandene Daten zu verändern oder sogar schadhafte Befehle auf der Systemebene des Datenbank-Hosts auszuführen. Die Manipulation von Anmeldedaten ist dabei ein häufiges Angriffsszenario. In einem Login-Formular könnte ein Angreifer beispielsweise `admin'--` in das Feld für den Benutzernamen eingeben.

```
1     SELECT * FROM users WHERE username = 'admin'--';
```

`--` markiert den Rest des Codes als Kommentar, was dazu führt, dass der Angreifer ohne Passwortanfrage Zugriff erhält.

Ein weiteres Beispiel ist die sogenannte Union-Based SQL-Injection. Bei einer Anwendung, die für SQL-Injection anfällig ist und bei der die Abfrageergebnisse innerhalb der Antworten der Anwendung zurückgegeben werden, kann das `UNION`-Schlüsselwort verwendet werden, um Daten aus anderen Tabellen innerhalb der Datenbank abzurufen.
Das Schlüsselwort `UNION` ermöglicht es, eine oder mehrere zusätzliche `SELECT`-Abfragen auszuführen und die Ergebnisse an die ursprüngliche Abfrage anzuhängen.

```
1 1 UNION SELECT database(), version(), user();
```

Diese Abfrage gibt nicht nur die gewünschten Daten zurück, sondern auch zusätzliche Informationen über die Datenbank, die Version der Datenbanksoftware und den aktuellen Datenbankbenutzer zurück.

Blind SQL-Injection ist eine subtilere Variante, bei der der Angreifer keine direkten Rückmeldungen vom System erhält. Stattdessen bestimmt der Angreifer die Antwort basierend auf der Reaktion der Anwendung, etwa durch unterschiedliche Ladezeiten. Dieser Angriff wird häufig verwendet, wenn die Webanwendung so konfiguriert ist, dass sie allgemeine Fehlermeldungen anzeigt, den anfälligen Code für SQL-Injection jedoch nicht abgesichert hat. Eine typische Abfrage könnte so aussehen:

```
1 SELECT * FROM users WHERE id = 1 AND IF(1=1, SLEEP(5), NULL);
```

Steigt nun die Antwortzeit signifikant, bestätigt dies dem Angreifer die Verwundbarkeit.

Eine weniger bekannte, aber ebenso gefährliche Methode ist die Second-Order SQL-Injection. Second-Order-SQL-Injection tritt auf, wenn von Benutzern bereitgestellte Daten von der Anwendung gespeichert und später auf unsichere Weise in SQL-Abfragen integriert werden. Dies ist besonders hinterhältig, da die schädliche Wirkung oft erst nach einer gewissen Zeit sichtbar wird. Ein Angreifer könnte beispielsweise eine harmlose Eingabe in ein Profilfeld speichern, die später bei der Erstellung eines Berichts zu einer SQL-Injection führt.

Die Auswirkungen von SQL-Injection sind vielfältig und reichen von Datendiebstahl bis hin zur vollständigen Zerstörung von Datenbanken. Prominente Beispiele wie der Angriff auf Sony Pictures im Jahr 2011 verdeutlichen die Tragweite solcher Sicherheitslücken, bei denen Millionen von Datensätzen preisgegeben wurden. [Ley11] Neben dem Diebstahl sensibler Informationen können Angreifer durch SQL-Injection auch Benutzerrechte erhöhen und administrative Kontrolle über das System erlangen. Dies kann zur Manipulation von Daten oder zur Nutzung der Struktur für weitere Angriffe führen.

2.1.3 Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) ist ein Angriff, bei dem ein authentifizierter Benutzer dazu gezwungen wird, unerwünschte Aktionen in einer Webanwendung auszuführen, ohne dass der Benutzer sich darüber bewusst ist, indem der Angreifer in dessen Namen eine Anfrage absendet. [OWA21]
Ein typischer CSRF-Angriff beginnt damit, dass der Angreifer eine Anfrage konstruiert, die eine ungewollte Aktion wie eine Geldüberweisung oder eine Änderung der Kontoeinstellungen ausführt. Diese Anfrage wird in eine scheinbar harmlose Webseite oder ein Skript eingebettet. Besucht ein authentifizierter Benutzer diese Seite oder führt das Skript aus, sendet der Browser des Benutzers die Anfrage inklusive seiner gültigen Authentifizierungsdaten. Die Zielanwendung kann nicht zwischen einer echten und einer manipulierten Anfrage unterscheiden und führt die Aktion aus.
Ein Beispiel ist der Missbrauch von Online-Banking-Diensten:

```
http://bank.com/transfer?to=attacker&amount=1000
```

Wenn das Opfer, das in sein Bankkonto eingeloggt ist, auf diesen Link klickt, verarbeitet die Bank die Überweisung, als hätte das Opfer sie autorisiert.

2.1.4 Command Injection

Ein Command Injection ist ein schwerwiegender Angriff, bei dem ungesicherte Benutzereingaben direkt in Systembefehle eingebettet werden. Dies könnte zur Folge haben, dass Angreifer schädliche Befehle ausführen können und die Kontrolle über das Serversystem übernehmen. [Fou23a]
Die Hauptursache für Command Injection ist die unzureichende Kontrolle über die Eingabedaten, die oft direkt in Funktionen wie `system()`, `exec()` oder `shell_exec()` eingespeist werden. Diese Funktionen führen Befehle auf Betriebssystemebene aus, was potenziell zu einer vollständigen Gefährdung des Servers führen kann, wenn keine robusten Schutzmaßnahmen implementiert sind.

```
filename = params[:filename]
system("mv /uploads/#{filename} /uploads/processed/#{filename}")
```

Ohne Eingabevalidierung könnte ein Angreifer als `filename` den Wert `file.txt; rm -rf /`. Dadurch würde der Befehl ausgeführt:

```
mv /uploads/file.txt /uploads/processed/file.txt; rm -rf /
```

Dies würde zu einer Löschung aller Daten auf dem Server führen.

2.1.5 Unsichere Session- und Cookie-Verwaltung

Ein zentraler Bestandteil von Webanwendungen ist die Verwaltung von Sessions und Cookies, da diese die Authentifizierung und das Sitzungsmanagement befähigen.
Unsichere Session- und Cookie-Verwaltung stellt erhebliche Sicherheitsrisiken dar, die die Integrität und Vertraulichkeit von Webanwendungen gefährden können. Ein zentrales Problem ist das Session Hijacking, bei dem Angreifer durch das Abfangen von Session-IDs unbefugten Zugriff auf Benutzerkonten erlangen. Dies geschieht oft durch unsichere Verbindungen oder Man-in-the-Middle-Angriffe, insbesondere in unverschlüsselten Netzwerken [Fou23a]. Die Übernahme einer Sitzung ermöglicht es Angreifern, im Namen des Opfers Aktionen durchzuführen oder vertrauliche Daten einzusehen, was schwerwiegende Folgen haben kann.

Ein weiteres Risiko besteht durch unsichere Speicherung von Sitzungscookies auf der Client-Seite. Ohne die Verwendung von Mechanismen wie `HttpOnly` oder `Secure`-Flags können Angreifer Cookies

auslesen oder manipulieren, was zur Kompromittierung der Sitzung führt. Schwache oder fehlende Verschlüsselung erhöht das Risiko zusätzlich, da Angreifer Zugriff auf sensible Daten wie Session-IDs erhalten.

Darüber hinaus gefährdet die unzureichende Invaliderung von Sitzungen die Sicherheit von Anwendungen. Wenn Sitzungen nach einer Abmeldung oder bei Inaktivität nicht ablaufen, bleibt der Zugriff bestehen, was insbesondere bei gemeinsam genutzten Geräten zu Missbrauch führen kann. Dies stellt ein bedeutendes Risiko für die Vertraulichkeit dar und eröffnet Angreifern zusätzliche Angriffsvektoren.

Die aufgeführten Risiken verdeutlichen die Notwendigkeit, Sitzungen und Cookies mit bewährten Sicherheitsmethoden zu schützen, um die Vertraulichkeit und Integrität von Benutzerdaten sicherzustellen.

2.2 Rails-spezifische Sicherheitsfeatures

2.2.1 Schutz vor Cross-Cite Scripting in Ruby on Rails

Ruby on Rails bietet Entwicklern eine Vielzahl von Mechanismen, um die Benutzereingaben sicher zu verarbeiten und so Webanwendungen gegen Cross-Site Scripting (XSS) zu schützen. Im folgenden Abschnitt werden die gezielten Methoden und Strategien erläutert, um XSS-Angriffe effektiv zu verhindern.

Automatisches HTML-Escaping: Der Standart in Rails HTML-Escaping bewirkt, dass alle Zeichenketten, die in den Views implementiert sind, automatisch entschärft werden, bevor sie an den Browser gesendet werden. Dies bedeutet, dass gefährliche Zeichen wie `<`, `>` und `&` in HTML-Entitäten umgewandelt werden, wodurch der Browser sie dann als reinen Text anzeigt, anstatt sie als HTML oder JavaScript auszuführen.

Beispiel für Escaping: Eine Eingabe wie `<script>alert('XSS')</script>` wird folgendermaßen ausgegeben:

```
1 &lt;script&gt;alert('XSS')&lt;/script&gt;
```

Listing 2.1: HTML-Beispiel für Escaping

Automatisches Escaping in ERB (Embedded Ruby) In ERB, der Standard-Template-Engine von Rails, werden automatisch alle Ausgaben entschärft, die mit `<%= %>` eingebunden werden. Somit sind alle Views standardmäßig vor XSS geschützt.

```
1 <%= @user.name %>
```

Ist der Wert von `@user.name` `<script>alert('XSS')</script>`, dann wird der Browser diese Zeichenkette wie im vorherigen Beispiel Listing 2.1 darstellen.

Gezielte Eingabekontrolle mit `sanitize` Es gibt jedoch Situationen, da sind HTML-Inhalte gewünscht. Zum Beispiel bei benutzergenerierten Inhalten wie Blogposts oder Kommentaren. Für solche Fälle stellt Rails die Methode `sanitize` bereit.
Mit `sanitize` können spezifische HTML-Tags und Attribute definiert werden, die erlaubt sind, während alle anderen entfernt werden.

```
<%= sanitize(user.bio, tags: %w[b i u], attributes: %w[class])%>
```

Listing 2.2: Beispiel sanitize

In diesem Beispiel dürfen Benutzer die HTML-Tags `<b>`, `<i>` und `<u>` sowie das Attribut `class` verwenden. Alle anderen Tags und Attribute werden entfernt. Dies verhindert Angriffe, während gewünschte Formatierungen erhalten bleiben.

2.2.2 Schutzt vor SQL-Injection in Ruby on Rails

Gegen SQL-Injection bietet Ruby on Rails durch sein Framework und insbesondere durch Active Record wirksame Maßnahmen, um solche Angriffe zu verhindern. Active Record, das Object-Relational-Mapping (ORM)-Tool von Rails, abstrahiert Datenbankoperationen und bietet Entwicklern eine API, die sowohl die Interaktion mit der Datenbank vereinfacht als auch Schutzmechanismen gegen Angriffe implementiert. Im Zentrum dieser Mechanismen steht die Nutzung von *prepared statements*, die Eingabewerte und SQL-Befehle strikt voneinander trennen. Dies verhindert, dass manipulierte Eingaben die Struktur der SQL-Befehle verändern können [Gui23a].

Ein typisches Beispiel für den Einsatz von *prepared statements* in Active Record zeigt die folgende Abfrage:

```
User.where("email = ?", params[:email])
```

Listing 2.3: Sichere Abfrage mit Prepared Statements

In diesem Beispiel wird der Eingabewert für `params[:email]` durch Rails automatisch escaped. Der Parameter wird als sicherer Platzhalter in die Abfrage eingefügt, wodurch schädliche Eingaben unschädlich gemacht werden. Dadurch wird verhindert, dass ein Angreifer durch Eingaben wie `' OR 1=1 --` Zugriff auf alle Benutzer erhält.

Darüber hinaus bietet Active Record Abstraktionen, die SQL-Injection nahezu vollständig eliminieren können, wenn sie korrekt verwendet werden. Eine typische Methode ist `find_by`, die automatisch Abfragen generiert und die Gefahr unsicherer Konstrukte minimiert:

```
User.find_by(email: params[:email])
```

Listing 2.4: Verwendung von Active Record-Methoden

Neben der Sicherheit sorgt diese Abstraktion auch für eine bessere Lesbarkeit und Wartbarkeit des Codes. Komplexere Abfragen können mithilfe von Active Record ebenfalls sicher gestaltet werden. Ein Beispiel hierfür ist das Sortieren und Filtern von Daten:

```
User.order(created_at: :desc).limit(10)
```

Listing 2.5: Sicheres Sortieren und Filtern mit Active Record

Trotz dieser Sicherheitsmechanismen können Risiken auftreten, wenn Entwickler von der API abweichen und unsichere Praktiken wie die direkte Verwendung von String-Interpolationen anwenden. Das folgende Beispiel zeigt eine gefährliche Implementierung:

```ruby
User.where("email = '#{params[:email]}'")
```

Listing 2.6: Unsicheres Beispiel für SQL-Abfragen

Solche Konstrukte machen Anwendungen anfällig für SQL-Injection, da Eingaben direkt in die SQL-Abfrage eingebettet werden, ohne eine vorherige Validierung oder Escaping [Fou23d].

Prepared Statements sind nicht nur ein Schutzmechanismus, sondern auch eine performante Lösung, da die Abfragen vom Datenbankmanagementsystem (DBMS) vorab kompiliert und mehrfach mit unterschiedlichen Parametern ausgeführt werden können. Rails gewährleistet somit nicht nur Sicherheit, sondern optimiert auch die Performance der Datenbankzugriffe.

Die Sicherheitsmechanismen von Rails zeigen, wie durch die konsequente Nutzung sicherer Abfragemethoden und Abstraktionen Sicherheitsrisiken erheblich reduziert werden können. Dennoch liegt ein Teil der Verantwortung bei den Entwicklern, die bereitgestellten Werkzeuge korrekt zu nutzen und unsichere Konstrukte zu vermeiden. [Fou23d, Gui23a]

2.2.3 Rails-Schutz gegen Cross-Site Request Forgery (CSRF)

Cross-Site Request Forgery (CSRF) ist ein Angriff, bei dem ein authentifizierter Benutzer ohne sein Wissen dazu gebracht wird, unerwünschte Aktionen in einer Webanwendung auszuführen. Dieser Angriff nutzt die Tatsache aus, dass ein Browser bei einer Anfrage automatisch Authentifizierungsdaten wie Cookies oder Sitzungsinformationen mitsendet. Ruby on Rails begegnet dieser Bedrohung durch die standardmäßige Implementierung von **CSRF-Tokens**, die die Validität einer Anfrage sicherstellen und so CSRF-Angriffe wirksam verhindern [OWA21, Gui23b].

Funktionsweise von CSRF-Tokens in Rails

CSRF-Tokens werden von Rails für jede Benutzersitzung automatisch generiert und in Formularen oder AJAX-Anfragen eingefügt. Das Token wird kryptografisch sicher erstellt und ist an die aktuelle Sitzung des Benutzers gekoppelt. Jede Anfrage, die Änderungen auf der Serverseite auslöst (z. B. POST-, PUT-, PATCH- oder DELETE-Anfragen), muss ein gültiges CSRF-Token enthalten. Der Server prüft bei eingehenden Anfragen, ob das übermittelte Token mit dem serverseitig gespeicherten übereinstimmt. Fehlt das Token oder ist es ungültig, lehnt Rails die Anfrage ab und gibt eine Fehlermeldung wie `ActionController::InvalidAuthenticityToken` zurück [Har22, Gui23b].

Ein Beispiel für die automatische Einbindung von CSRF-Tokens in Rails-Formulare sieht wie folgt aus:

```erb
<%= form_with(url: '/transfer', method: :post) do |form| %>
  <%= form.text_field :recipient %>
  <%= form.text_field :amount %>
  <%= form.submit 'Transfer' %>
<% end %>
```

Listing 2.7: CSRF-Token in einem HTML-Formular

Rails generiert hierbei ein verstecktes Eingabefeld mit dem CSRF-Token und fügt es automatisch in das Formular ein. Bei der Verarbeitung der Anfrage überprüft der Server das Token, um sicherzustellen, dass die Anfrage tatsächlich vom Benutzer stammt, für den die Sitzung gültig ist.

Auch AJAX-Anfragen sind durch diesen Mechanismus geschützt. Rails bindet das CSRF-Token mithilfe des `meta`-Tags ein, sodass es von JavaScript ausgelesen und mit der Anfrage als Header gesendet werden kann:

```
<meta name="csrf-token" content="<%= csrf_meta_tags %>">
```

Listing 2.8: CSRF-Token für AJAX-Anfragen

Verbindung zum Praxisbeispiel

Wie im vorherigen Abschnitt beschrieben, könnte ein Angreifer versuchen, eine Überweisung im Namen eines authentifizierten Benutzers durchzuführen, indem er eine manipulierte URL erstellt. Ohne Schutzmechanismen wie CSRF-Tokens würde der Server die Anfrage ausführen, da sie die Cookies des Benutzers enthält. Dank der standardmäßigen Verwendung von CSRF-Tokens in Rails ist ein solcher Angriff jedoch nicht möglich. Das manipulierte Beispiel:

```
http://bank.com/transfer?to=attacker&amount=1000
```

scheitert, da das Token in der Anfrage fehlt oder ungültig ist. Die Bankanwendung erkennt die Anfrage als unautorisiert und lehnt sie ab. Dies verdeutlicht, wie effektiv CSRF-Tokens nicht nur einfache Angriffe abwehren, sondern auch für eine hohe Sicherheit in komplexen Anwendungsszenarien sorgen.

Durch diesen Schutz stellt Rails sicher, dass Angreifer keine Aktionen im Namen eines authentifizierten Benutzers durchführen können, selbst wenn sie Zugriff auf bestimmte API-Endpunkte oder URLs haben. CSRF-Tokens bieten somit eine essenzielle Schutzschicht, die in modernen Webanwendungen unverzichtbar ist.

2.2.4 Rails-Schutz gegen Command Injection

Command Injection stellt eine schwerwiegende Sicherheitslücke dar, bei der Angreifer durch manipulierte Eingabewerte schädliche Systembefehle ausführen können. Wie im vorherigen Abschnitt 2.1.4 gezeigt, resultiert ein solcher Angriff oft aus unzureichender Eingabevalidierung und der direkten Verwendung von Eingabedaten in Funktionen wie `system()` oder `exec()`. Diese Sicherheitsrisiken werden in Ruby on Rails durch verschiedene Mechanismen behandelt, insbesondere durch die Nutzung von Sicherheits-Gems und strikter Input-Validierung, die Angriffsvektoren effektiv minimieren [Fou23a, oRG23a].

Sicherheits-Gems zur Prävention

In der Rails-Community existieren mehrere Sicherheits-Gems, die speziell dafür entwickelt wurden, potenzielle Command-Injection-Schwachstellen zu vermeiden. Gems wie `Shellwords` bieten Methoden zur sicheren Handhabung von Eingabedaten, die in Systembefehlen verwendet werden. Das folgende Beispiel zeigt die Anwendung von `Shellwords.escape`, um sicherzustellen, dass die Eingabedaten keine schädlichen Befehle enthalten:

```
require 'shellwords'

filename = Shellwords.escape(params[:filename])
system("mv /uploads/#{filename} /uploads/processed/#{filename}")
```

Listing 2.9: Sicherer Umgang mit Eingabedaten mithilfe von Shellwords

Durch die Nutzung von `Shellwords.escape` werden Sonderzeichen im Eingabewert maskiert, sodass keine ungewollten Befehle eingebettet werden können. Dies verhindert effektiv Angriffe wie das Ausführen des Kommandos `rm -rf /` im vorherigen Beispiel.

Input-Validierung und Whitelisting

Neben der Verwendung von Sicherheits-Gems setzt Rails auf robuste Eingabevalidierung, um potenziell gefährliche Daten frühzeitig zu erkennen und abzulehnen. Entwickler können hierfür Validierungsregeln definieren, die die Eingaben auf erlaubte Muster beschränken. Im obigen Szenario könnte die Validierung sicherstellen, dass der Dateiname nur aus alphanumerischen Zeichen und bestimmten Sonderzeichen besteht:

```
1 if params[:filename] =~ /\A[a-zA-Z0-9._-]+\z/
2   system("mv /uploads/#{params[:filename]}
       /uploads/processed/#{params[:filename]}")
3 else
4   raise "Ungueltiger Dateiname"
5 end
```

Listing 2.10: Eingabevalidierung zur Verhinderung von Command Injection

Dieser Ansatz basiert auf einem Whitelisting, das sicherstellt, dass nur explizit erlaubte Zeichen in den Eingaben enthalten sind. Dadurch werden Angriffe wie im vorherigen Abschnitt beschrieben wirkungsvoll abgewehrt.

Kontext zur Rails-Sicherheitsarchitektur

Rails betont durch seine Sicherheitsrichtlinien, dass Eingabedaten niemals direkt in systemkritischen Funktionen verwendet werden sollten. Der Schutz vor Command Injection ergänzt andere Mechanismen wie CSRF-Schutz, um eine umfassende Sicherheitsarchitektur zu gewährleisten. Im Unterschied zu CSRF, bei dem Angreifer auf Sitzungsdaten abzielen, zielt Command Injection direkt auf die Ausführung schädlicher Befehle. Beide Angriffstypen verdeutlichen jedoch, wie wichtig eine sorgfältige Kontrolle und Verarbeitung von Benutzereingaben ist.

Durch die Kombination von Sicherheits-Gems, strenger Input-Validierung und bewährten Praktiken bietet Rails einen robusten Schutz gegen Command Injection. Diese Mechanismen sorgen dafür, dass Anwendungen selbst bei komplexen Eingabeszenerien sicher bleiben und Serverressourcen vor unautorisiertem Zugriff geschützt sind.

2.2.5 Sichere Session- und Cookie-Verwaltung in Rails: Nutzung von Secure Cookies, HTTP-only Flags und Konfigurationsoptionen

Die Verwaltung von Sessions und Cookies spielt eine wesentliche Rolle in der Sicherheit von Webanwendungen. Unsichere Praktiken in der Handhabung von Sessions und Cookies können zu schweren Sicherheitslücken führen, darunter Angriffe wie Session Hijacking und die unbefugte Verwendung von Sessions. Ruby on Rails bietet zahlreiche Mechanismen, um die Integrität und Vertraulichkeit von Cookies und Sessions zu gewährleisten und vor solchen Angriffen zu schützen [Fou23c, oRG23b].

Secure Cookies und HTTPS

Ein grundlegender Sicherheitsmechanismus in Rails für die Verwaltung von Cookies ist die Verwendung des `Secure`-Flags. Mit diesem Flag wird sichergestellt, dass Cookies nur über verschlüsselte Verbindungen (HTTPS) übertragen werden. Dadurch wird verhindert, dass Cookies in unverschlüsselten Netzwerken abgefangen werden. In Rails kann dies durch die Konfiguration des Cookie-Speichers in der `session_store`-Einstellung erreicht werden. Hierbei wird das `secure`-Attribut aktiv gesetzt, um Cookies nur in einer HTTPS-Verbindung zu übertragen:

```
Rails.application.config.session_store :cookie_store, key:
    '_your_app_session', secure: Rails.env.production?
```

Listing 2.11: Konfiguration von Secure Cookies in Rails

Durch diese Konfiguration wird sichergestellt, dass in der Produktionsumgebung alle Cookies nur über HTTPS verschickt werden, was die Sicherheit erhöht und das Risiko von Man-in-the-Middle-Angriffen reduziert.

HTTP-only Cookies

Das `HTTP-only`-Flag stellt sicher, dass Cookies nicht über clientseitige Skripte zugänglich sind, was einen effektiven Schutz gegen Cross-Site Scripting (XSS)-Angriffe bietet. Rails setzt dieses Flag standardmäßig für Session-Cookies, sodass bösartige Skripte im Browser des Nutzers nicht auf die Cookies zugreifen können. Dies verhindert, dass Angreifer beispielsweise mit JavaScript sensible Informationen wie Session-Tokens extrahieren können, wenn die Anwendung anfällig für XSS ist [Fou23b].

SameSite Cookies

Ein weiterer wichtiger Sicherheitsmechanismus ist die Verwendung des `SameSite`-Attributs für Cookies. Dieses Attribut verhindert, dass Cookies bei Cross-Site-Requests gesendet werden. Es kann auf `Strict` oder `Lax` gesetzt werden, je nach den Anforderungen der Anwendung. In Rails kann dieses Attribut leicht konfiguriert werden, um zusätzliche Schutzmechanismen gegen Cross-Site Request Forgery (CSRF) und ähnliche Angriffe zu implementieren. Die folgende Konfiguration setzt das `SameSite`-Attribut auf `Lax`:

```
Rails.application.config.session_store :cookie_store, key:
    '_your_app_session', same_site: :lax
```

Listing 2.12: SameSite-Attribut in Rails

Mit dieser Einstellung werden Cookies nur dann bei Anfragen von anderen Seiten gesendet, wenn die Anfrage als ”lax”betrachtet wird (z. B. bei Navigationen), was für die meisten Webanwendungen ausreichend ist, um die Sicherheit zu erhöhen und gleichzeitig eine gute Benutzererfahrung zu gewährleisten.

Verschlüsselung von Session-Daten

Rails bietet standardmäßig die Verschlüsselung von Session-Daten, die im Browser des Benutzers gespeichert werden. Dies schützt vor Manipulation und sorgt dafür, dass sensible Daten nicht im Klartext gespeichert werden. Die Verschlüsselung erfolgt mit einem geheimen Schlüssel, der entweder in der `config/secrets.yml`-Datei oder in Umgebungsvariablen definiert wird. Dieser Schlüssel wird verwendet, um die Integrität der gespeicherten Daten sicherzustellen und zu verhindern, dass Angreifer die Session-Daten ändern können:

```
production:
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Listing 2.13: Konfiguration des geheimen Schlüssels

Durch diese Verschlüsselung wird gewährleistet, dass die in der Session gespeicherten Informationen sicher und unveränderlich bleiben, auch wenn ein Angreifer Zugriff auf die Cookies erlangen sollte.

2.2.6 Nutzung von Sicherheits-Gems wie Brakeman, Devise und RuboCop

Sicherheits-Gems wie **Brakeman**, **Devise** und **RuboCop** können erheblich dazu beitragen, den Sicherheitsstandard und die Codequalität zu erhöhen. Jedes dieser Gems erfüllt spezifische Rollen in der Sicherheitsstrategie von Rails-Anwendungen und ist ein unverzichtbares Werkzeug für Entwickler.

Brakeman: Sicherheitsprüfung durch statische Codeanalyse

Brakeman ist ein statisches Analysetool, das speziell für Rails-Anwendungen entwickelt wurde. Es analysiert den Quellcode auf Schwachstellen, ohne die Anwendung auszuführen, und bietet umfassende Berichte über potenzielle Sicherheitsrisiken wie SQL-Injections, Cross-Site Scripting (XSS), unsichere Redirects und Massenzuweisung. Brakeman ist besonders effektiv in Continuous Integration (CI)-Pipelines, da es Entwickler frühzeitig auf Sicherheitsprobleme aufmerksam macht. [Col24]

Ein Beispiel: Brakeman kann unsichere Konstruktionen wie die folgende identifizieren:

```
User.where("email = '#{params[:email]}'")
```

Das Tool empfiehlt die Verwendung sicherer Abfragemethoden wie:

```
User.where(email: params[:email])
```

Diese Art der Überprüfung schützt vor SQL-Injections, einer der häufigsten Sicherheitslücken in Webanwendungen.

Devise: Authentifizierung und Benutzerverwaltung

Devise ist ein Gem, das die Implementierung von Authentifizierungs- und Benutzerverwaltungsfunktionen in Rails-Anwendungen stark vereinfacht. Es bietet Module für Passwortverschlüsselung, Sitzungsmanagement, Konto-Sperrung nach wiederholten Fehlversuchen, Multi-Faktor-Authentifizierung und mehr. Devise verwendet `BCrypt`, eine bewährte Methode zur sicheren Passwort-Hashing, und minimiert damit die Gefahr durch Passwortdiebstahl und Brute-Force-Angriffe. [Hea24]

Ein Beispiel für die Benutzeranmeldung mit Devise:

```
class User < ApplicationRecord
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable
end
```

Die einfache Konfiguration sorgt dafür, dass selbst komplexe Authentifizierungsanforderungen leicht implementiert werden können.

Zusätzlich ermöglicht Devise die nahtlose Integration mit Sicherheitsfeatures wie rollenbasierten Zugriffskontrollen, indem es mit anderen Gems wie `Pundit` oder `CanCanCan` kombiniert wird.

RuboCop: Sicherer und konsistenter Code

RuboCop ist ein statisches Analysetool, das nicht nur die Einhaltung von Coding-Standards gewährleistet, sondern auch sicherheitskritische Aspekte überprüft. RuboCop analysiert den Code auf potenziell unsichere Muster, wie etwa die direkte Verwendung von gefährlichen Methoden oder unsichere Konfigurationsoptionen. Es hilft Entwicklern, nicht nur die Wartbarkeit und Lesbarkeit des Codes zu verbessern, sondern auch potenzielle Angriffsvektoren zu eliminieren. [Tea24b]

Ein Beispiel: RuboCop warnt vor der Nutzung unsicherer Methoden wie `eval` oder ungesicherter Benutzer-Eingaben und gibt Vorschläge für sicherere Alternativen. So kann ein unsicheres Konstrukt wie:

```
eval("2 + 2")
```

durch sicherere Ansätze ersetzt werden, die keine dynamische Codeausführung erfordern.

2.3 Sicherheitslücken durch unsichere Konfigurationen und Drittanbieter-Gems

Unsichere Konfigurationen und die Nutzung von Drittanbieter-Gems stellen ein erhebliches Risiko für die Sicherheit von Ruby on Rails-Anwendungen dar. Während Rails als Framework viele Sicherheitsmechanismen standardmäßig implementiert, können unachtsame Konfigurationen oder unsichere Abhängigkeiten die Anwendung anfällig für Angriffe machen.

2.3.1 Probleme durch veraltete Gems und Abhängigkeiten

Drittanbieter-Gems sind ein zentraler Bestandteil des Ruby-on-Rails-Ökosystems und erleichtern die Entwicklung durch vorgefertigte Funktionalitäten. Diese Abhängigkeiten bergen jedoch Risiken, insbesondere wenn veraltete oder unsichere Versionen verwendet werden. Angriffe wie die sogenannte *Dependency Confusion* nutzen Schwachstellen in der Verwaltung von Abhängigkeiten aus und können dazu führen, dass ein Angreifer schädlichen Code einschleust. Eine unzureichende Aktualisierung von Gems, beispielsweise durch Vernachlässigung von Sicherheitsupdates, erhöht das Risiko für bekannte Schwachstellen wie SQL-Injection oder Cross-Site Scripting (XSS). Tools wie *Bundler Audit* können hier Abhilfe schaffen, indem sie Abhängigkeiten auf bekannte Sicherheitslücken prüfen [OWA24, Tea24a].

2.3.2 Absicherung von Authentifizierungsprozessen

Die Sicherheit von Authentifizierungsprozessen ist ein kritischer Bestandteil jeder Webanwendung. Rails bietet mit der Integration von Bibliotheken wie `bcrypt` eine sichere Möglichkeit zur Speicherung von Passwörtern. `bcrypt` nutzt eine adaptive Hashing-Funktion, die es ermöglicht, den Aufwand für die Passwort-Hashing-Berechnung zu erhöhen, wodurch Brute-Force-Angriffe erheblich erschwert werden. Ein korrekt implementierter Passwort-Hashing-Prozess könnte wie folgt aussehen:

```
password = "user_password"
hashed_password = BCrypt::Password.create(password)
```

Listing 2.14: Sichere Passwort-Hashing-Implementierung mit bcrypt

Neben der Verwendung von sicheren Hashing-Algorithmen ist es essenziell, Authentifizierungsprozesse durch Mechanismen wie Multi-Faktor-Authentifizierung (MFA) oder Ratenbegrenzungen gegen Brute-Force-Angriffe zu schützen. Die Kombination von `bcrypt` mit einem Gem wie *Devise* ermöglicht eine robuste und flexible Authentifizierungslösung.

2.3.3 Risiken durch unsichere Standardwerte und Einstellungen in Rails

Rails bietet standardmäßig sinnvolle Voreinstellungen, doch bestimmte Konfigurationen erfordern Anpassungen, um maximale Sicherheit zu gewährleisten. Beispielsweise sind standardmäßige Einstellungen für die Caching-Strategie oder Cross-Origin Resource Sharing (CORS) potenziell unsicher und sollten je nach Anwendung angepasst werden. Ein prominentes Beispiel ist die Aktivierung von CORS für alle Domains, was es Angreifern erleichtert, bösartige Anfragen an die Anwendung zu stellen. Eine sichere Konfiguration könnte wie folgt aussehen:

```
1 Rails.application.config.middleware.insert_before 0, Rack::Cors do
2   allow do
3     origins 'trusted-domain.com'
4     resource '*', headers: :any, methods: [:get, :post, :patch,
          :put]
5   end
6 end
```

Listing 2.15: Beispiel für eine sichere CORS-Konfiguration

Weitere potenzielle Risiken bestehen bei der Konfiguration von Session- und Cookie-Optionen. Beispielsweise sollten Cookies immer mit den Flags `HttpOnly` und `Secure` versehen werden, um sie vor unbefugtem Zugriff und Übertragung in unsicheren Netzwerken zu schützen.

Die Kombination aus bewusster Konfiguration, der regelmäßigen Aktualisierung von Gems und der Implementierung bewährter Sicherheitspraktiken reduziert das Risiko, das durch unsichere Konfigurationen und Drittanbieter-Gems entsteht, erheblich.

3 Handlungsempfehlungen und Best Practices

3.1 Konkrete Sicherheitsmaßnahmen und Best Practices für Ruby on Rails

Ruby on Rails bietet eine Vielzahl von Mechanismen und Best Practices, die es Entwicklern ermöglichen, Sicherheitslücken frühzeitig zu vermeiden und die Integrität von Webanwendungen zu gewährleisten. Die konsequente Anwendung bewährter Sicherheitspraktiken ist entscheidend, um die Angriffsfläche von Rails-Anwendungen zu minimieren und die Sicherheit aufrechtzuerhalten.

Die sichere Speicherung von Passwörtern spielt eine zentrale Rolle im Sicherheitskonzept jeder Webanwendung. Rails bietet durch die Integration von `bcrypt` einen leistungsfähigen Mechanismus zur sicheren Speicherung von Passwörtern. Dieser Algorithmus wird durch die Methode `has_secure_password` in Active Record Modellen genutzt und sorgt dafür, dass Passwörter nicht im Klartext gespeichert werden, sondern sicher gehasht und verschlüsselt. Zusätzlich bieten Sicherheits-Gems wie Devise erweiterte Funktionen für die Authentifizierung, wie Multi-Faktor-Authentifizierung und die Sicherstellung der Sicherheit von Benutzersitzungen [Pla23].

Neben der sicheren Speicherung von Passwörtern spielt der Einsatz von spezialisierten Sicherheits-Gems eine wichtige Rolle bei der Identifikation und Behebung von Schwachstellen. `Brakeman` führt eine statische Code-Analyse durch und erkennt potenzielle Sicherheitslücken wie SQL-Injection, Cross-Site Scripting (XSS) und unsichere Konfigurationen. Brakeman lässt sich problemlos in Continuous Integration (CI) und Continuous Deployment (CD) Pipelines integrieren, wodurch automatisierte Sicherheitsberichte generiert werden und frühzeitig auf Schwachstellen hingewiesen wird [Col]. `RuboCop` ergänzt die Sicherheitsüberprüfung durch die statische Analyse des Codes und hilft dabei, unsichere Codemuster zu identifizieren. Dies sorgt für eine kontinuierliche Verbesserung des Codes und reduziert das Risiko von Sicherheitslücken, die durch inkonsistente oder fehlerhafte Implementierungen entstehen könnten [Tea23].

Rails bietet zudem integrierte Schutzmechanismen, die Angriffe wie Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF) und SQL-Injection verhindern. Um XSS zu vermeiden, wird standardmäßig jede Benutzereingabe, die in Views gerendert wird, durch das `escape_html`-System entschärft, wodurch das Ausführen schadhafter Skripte verhindert wird. Entwickler sollten dennoch sicherstellen, dass sie auf die Verwendung von Methoden wie `raw()` verzichten, die Sicherheitslücken öffnen können. Der Schutz gegen CSRF wird durch die automatische Generierung von CSRF-Tokens in Formularen realisiert, die bei jeder POST-, PUT-, DELETE- oder PATCH-Anfrage überprüft werden, um sicherzustellen, dass die Anfrage von der legitimen Quelle stammt. Im Hinblick auf SQL-Injection schützt Rails durch die Verwendung von Active Record automatisch vor SQL-Injection-Angriffen. Entwickler sollten daher immer auf die Verwendung von parametrisierten Abfragen und die Active Record Query Interface zurückgreifen, anstatt dynamische SQL-Statements zu verwenden [Fou23a].

Neben der Anwendungssicherheit ist auch die sichere Konfiguration der Produktionsumgebung von entscheidender Bedeutung. Um die Vertraulichkeit von Benutzerdaten zu wahren, sollte immer HTTPS verwendet werden, um die Datenübertragung zu verschlüsseln. Rails unterstützt diese Anforderung durch die Middleware `force_ssl`, die sicherstellt, dass alle Verbindungen über HTTPS

erfolgen. Weiterhin sollten Content Security Policy (CSP)-Header eingesetzt werden, um den Zugriff auf externe Ressourcen zu kontrollieren und somit die Angriffsfläche für XSS-Angriffe zu verringern. Ebenso ist es wichtig, dass in der Produktionsumgebung Debugging-Informationen und Stack-Traces deaktiviert sind, um Angreifern keine weiteren Informationen über die Anwendungsstruktur zu bieten. Rails ermöglicht eine strikte Trennung der Umgebungen (`development`, `test`, `production`), was sicherstellt, dass sensible Konfigurationen nur in der richtigen Umgebung angewendet werden.

Ein proaktives Patch-Management stellt sicher, dass bekannte Sicherheitslücken in Gems und Framework-Versionen schnell geschlossen werden. Tools wie `Dependabot` automatisieren den Prozess der Erkennung veralteter Gems und bieten direkt aktualisierte Versionen an, wodurch Entwickler auf dem neuesten Stand bleiben können [Git23].

3.2 Langfristige Sicherheitsstrategie

Eine langfristige Sicherheitsstrategie für Ruby on Rails-Anwendungen ist entscheidend, um sicherzustellen, dass Sicherheitspraktiken kontinuierlich umgesetzt werden und potenzielle Bedrohungen rechtzeitig abgewehrt werden. Dies kann durch die systematische Integration von Sicherheitsprozessen in den Entwicklungszyklus, die regelmäßige Sensibilisierung der Entwickler sowie die Förderung einer Kultur der sicheren Softwareentwicklung erreicht werden.

3.2.1 Integration von Sicherheitsprozessen in die CI/CD-Pipeline

Die kontinuierliche Integration und Bereitstellung (CI/CD) bietet eine ideale Plattform, um Sicherheitsprozesse frühzeitig und automatisiert in den Entwicklungszyklus zu integrieren. Dies umfasst die regelmäßige Durchführung von automatisierten Sicherheitsprüfungen während des gesamten Entwicklungsprozesses. Tools wie Brakeman für statische Code-Analyse, RuboCop für die Einhaltung von Sicherheitsrichtlinien sowie Automatisierung von Sicherheitstests sollten in die CI/CD-Pipeline eingebunden werden. Diese Tools ermöglichen die frühzeitige Identifikation von Sicherheitslücken und tragen dazu bei, dass potenzielle Schwachstellen in der Anwendung behoben werden, bevor diese in die Produktion gelangen [Col24, Tea24b]. Automatisierte Tests auf Sicherheitslücken und das regelmäßige Überprüfen von Abhängigkeiten und Systemkonfigurationen gewährleisten einen proaktiven Schutz vor Angriffen.

3.2.2 Regelmäßige Code-Reviews und Sicherheitsaudits

Code-Reviews durch erfahrene Entwickler sind ein essenzieller Bestandteil der langfristigen Sicherheitsstrategie. Dabei sollten nicht nur funktionale Aspekte berücksichtigt werden, sondern auch sicherheitsrelevante Fragen wie die Validierung von Benutzereingaben, die Verwendung sicherer Bibliotheken und die ordnungsgemäße Behandlung von sensiblen Daten. Code-Reviews können dazu beitragen, potenzielle Sicherheitslücken frühzeitig zu erkennen und zu beheben. Zusätzlich sollten regelmäßig Sicherheitsaudits durchgeführt werden, bei denen externe Experten die Anwendung auf Schwachstellen prüfen. Solche Audits helfen, Sicherheitslücken zu identifizieren, die möglicherweise von den Entwicklern übersehen wurden, und bieten eine zusätzliche Sicherheitsebene, um die Robustheit der Anwendung zu erhöhen [OWA21].

3.2.3 Förderung einer Kultur der sicheren Softwareentwicklung

Die Schaffung einer Sicherheitskultur innerhalb des Entwicklerteams ist ein langfristiger und nachhaltiger Ansatz, um Sicherheitslücken in Anwendungen zu minimieren. Dies erfordert eine kontinuierliche Sensibilisierung der Entwickler für Sicherheitsaspekte und die Integration von

Sicherheitspraktiken in den Entwicklungsprozess. Ein proaktives Risikomanagement, das auf der frühzeitigen Identifikation von Bedrohungen und der Anwendung von Best Practices in der Softwareentwicklung basiert, sollte von der Geschäftsführung und dem Management unterstützt werden. Zu den bewährten Methoden gehört das regelmäßige Einführen von Schulungen zu Sicherheitsthemen sowie das Fördern von sicherheitsbewusstem Verhalten im Alltag der Entwickler. Eine Sicherheitskultur umfasst auch die kontinuierliche Aktualisierung von Sicherheitsrichtlinien, um mit den sich ständig ändernden Bedrohungslandschaften Schritt zu halten

4 Fazit und Ausblick

Die vorliegende Seminararbeit hat sich mit der Sicherheitsanalyse und den Handlungsempfehlungen für Ruby on Rails-Anwendungen auseinandergesetzt. Dabei wurden die häufigsten Sicherheitsrisiken, spezifische Sicherheitsfeatures von Rails sowie Best Practices und Handlungsempfehlungen untersucht. Im Zentrum der Betrachtungen standen typische Schwachstellen wie Cross-Site Scripting (XSS), SQL-Injection, Cross-Site Request Forgery (CSRF), Command Injection und unsichere Konfigurationen, die durch Rails-eigene Mechanismen und externe Sicherheits-Gems wie `Brakeman`, `Devise` und `Rubocop` behandelt werden können.

Ergebnisse und Erkenntnisse

Eine wesentliche Erkenntnis dieser Arbeit ist, dass Ruby on Rails von Haus aus zahlreiche Schutzmechanismen bietet, die viele der gängigen Sicherheitsrisiken signifikant reduzieren können. Dennoch zeigt die Analyse, dass ein Großteil der Sicherheitsverantwortung weiterhin bei den Entwicklern liegt. Unsichere Konfigurationen, der Einsatz veralteter Drittanbieter-Gems und die Missachtung bewährter Praktiken stellen nach wie vor erhebliche Gefahren dar. Ein Beispiel hierfür sind veraltete Versionen von Bibliotheken, die bekannte Sicherheitslücken enthalten und somit Angreifern eine potenzielle Angriffsfläche bieten.

Der Einsatz spezialisierter Sicherheits-Gems, eine konsequente Umsetzung von Input-Validierung sowie die Absicherung sensibler Prozesse wie der Authentifizierung und Autorisierung wurden als zentrale Maßnahmen identifiziert. Ergänzend wurden Best Practices wie regelmäßige Code-Reviews, die Integration von Sicherheitschecks in CI/CD-Pipelines und ein stringentes Patch-Management hervorgehoben. Diese Maßnahmen tragen nicht nur zur Reduktion bestehender Risiken bei, sondern fördern auch eine langfristige Sicherheitskultur in Entwicklungsteams.

Kritische Betrachtung und Grenzen der Arbeit

Trotz der umfassenden Analyse kann diese Arbeit keinesfalls alle Sicherheitsrisiken und Best Practices vollständig abdecken. Die Komplexität moderner Webanwendungen sowie die kontinuierliche Weiterentwicklung von Angriffstechniken und Abwehrmechanismen machen es nahezu unmöglich, eine endgültige und vollständige Übersicht zu liefern. So wurden beispielsweise spezifische Bedrohungen wie die Absicherung von APIs, Angriffe auf Websockets oder die Sicherheitsanforderungen in hochverfügbaren Systemen nicht im Detail behandelt. Eine weitergehende Betrachtung dieser Themen wäre für zukünftige Arbeiten von großem Interesse.

Auch ist zu berücksichtigen, dass sich Sicherheitsmaßnahmen in der Praxis nicht immer ohne Weiteres umsetzen lassen. Faktoren wie Zeitdruck, Ressourcenknappheit oder mangelndes Sicherheitsbewusstsein können dazu führen, dass Sicherheitsmaßnahmen vernachlässigt werden. Eine Kultur der sicheren Softwareentwicklung muss daher von allen Beteiligten aktiv gefördert werden.

Ausblick

Die Sicherheit von Ruby on Rails-Anwendungen bleibt ein hochrelevantes und dynamisches Forschungsfeld. Zukünftige Arbeiten könnten sich intensiver mit der automatisierten Erkennung und Behebung von Schwachstellen befassen oder die Integration von maschinellem Lernen zur Verbesserung von Sicherheitsanalysen untersuchen. Darüber hinaus bietet die zunehmende Nutzung von Microservices und Cloud-Infrastrukturen neue Herausforderungen, die in Kombination mit Rails-spezifischen Sicherheitsmaßnahmen betrachtet werden sollten.

Abschließend lässt sich festhalten, dass Sicherheit ein kontinuierlicher Prozess ist. Die hier dargestellten Risiken und Maßnahmen sind ein wichtiger Schritt, um Rails-Anwendungen resilienter gegen Angriffe zu machen. Dennoch bleibt es eine Aufgabe der Entwickler und Organisationen, sich fortlaufend über neue Bedrohungen und Technologien zu informieren und ihre Sicherheitsstrategien entsprechend anzupassen.

A Literaturverzeichnis

[Col] Justin Collins. Brakeman: Static analysis security tool for ruby on rails. https://brakemanscanner.org/. Zugriff am 11. Dezember 2024.

[Col24] Justin Collins. Brakeman: A static analysis security tool for ruby on rails applications. https://brakemanscanner.org/, 2024. Zugriff am 11. Dezember 2024.

[Fou23a] OWASP Foundation. Command injection. https://owasp.org/www-community/attacks/Command_Injection, 2023. Abgerufen am 07. Dezember 2024.

[Fou23b] OWASP Foundation. Cookie security. https://owasp.org/www-project-secure-coding-practices/cookie_security.html, 2023.

[Fou23c] OWASP Foundation. Session management cheatsheet. https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html, 2023.

[Fou23d] OWASP Foundation. Sql injection prevention. https://owasp.org/www-project-top-ten/, 2023. Zugriff am 8. Dezember 2024.

[Git23] GitHub. Dependabot: Automate dependency updates. https://github.com/dependabot/dependabot-core, 2023. GitHub Documentation.

[Gui23a] Rails Guides. Active record basics. https://guides.rubyonrails.org/active_record_basics.html, 2023. Zugriff am 8. Dezember 2024.

[Gui23b] Rails Guides. Security guide: Csrf. https://guides.rubyonrails.org/security.html#csrf-countermeasures, 2023. Zugriff am 8. Dezember 2024.

[Har22] M. Hartl. *Ruby on Rails Tutorial: Learn Web Development with Rails*. Addison-Wesley, 2022.

[Hea24] Heartcombo. Devise: Flexible authentication solution for rails. https://github.com/heartcombo/devise, 2024. Zugriff am 11. Dezember 2024.

[Ley11] John Leyden. Sony hack: A timeline. *The Register*, 2011.

[oRG23a] Ruby on Rails Guides. Security guide: Common vulnerabilities in rails. https://guides.rubyonrails.org/security.html, 2023.

[oRG23b] Ruby on Rails Guides. Security guide: Cookies and sessions. https://guides.rubyonrails.org/security.html#cookies-and-sessions, 2023.

[OWA21] OWASP Foundation. Owasp top ten 2021. https://owasp.org/www-project-top-ten/, 2021. Zugriff am 4. Dezember 2024.

[OWA22] OWASP Foundation. Ruby on rails security guide. https://owasp.org, 2022. Zugriff am 4. Dezember 2024.

[OWA24] OWASP. Dependency security: Risks and recommendations. https://owasp.org/, 2024. Zugriff am 11. Dezember 2024.

[Pla23] Plataformatec. Devise: Flexible authentication solution for rails with warden. https://github.com/heartcombo/devise, 2023. GitHub Repository.

[Tea23] RuboCop Team. Rubocop: A ruby static code analyzer and formatter. https://github.com/rubocop/rubocop, 2023. GitHub Repository.

[Tea24a] Bundler Audit Team. Bundler audit: Scanning ruby applications for vulnerable dependencies. https://github.com/rubysec/bundler-audit, 2024. Zugriff am 11. Dezember 2024.

[Tea24b] RuboCop Core Team. Rubocop: A ruby static code analyzer and formatter. https://rubocop.org/, 2024. Zugriff am 11. Dezember 2024.