LB 183 Dokumentation

# Handlungsziel 1:

## Aktuelle Bedrohungen erkennen und erläutern

Um die aktuellen Bedrohungen erkenne zu können, kann man sich an die OWASP Top 10 wenden. OWASP aktualisiert ihre Bedrohungsliste alle 4 Jahre was sie sehr aktuell und nützlich macht. Auf Ihrer Seite findet man folgende Abbildung:



OWASP überprüft Seiten mithilfe von anderen Leuten und wertet die Daten dann aus. Was noch erwähnenswert ist, ist das OWASP mit der Datensammlung für die OWASP Top 10 schon begonnen hat. Bis Dezember 2024 werden Daten gesammelt.

## Informationen beschaffen

Um sich vor möglichen Sicherheitslücken schützen zu können, muss man sein Projekt gründlich geplant haben. Hier kann man sich wieder an die OWASP Top 10 orientieren.

OWASP bietet zu allen Sicherheitslücken in ihrer Liste eine Übersicht, Beschreibung, Massnahmen und sogar Beispielsangriffe. Darunter findet man noch Referenzen und eine Liste von Common Weakness Enumeration (CWE) welche zu dieser Sicherheitslücke gehören.

OWASP bietet zusätzlich noch einen Cheatsheat an: <https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html>

Neben OWASP existieren noch viele weitere Seiten, welche Informationen zur Erkennung und Bekämpfung von Sicherheitslücken bieten.

## Auswirkungen darstellen

Um die möglichen Auswirkungen anzuzeigen, nehme ich SQL injection als Beispiel.   
(Diese Sicherheitslücke werde ich später in der Insecure App auch beheben)

### Schutzziele

Die Schutzziele sind in drei Kategorien unterteil: Vertraulichkeit, Integrität und Verfügbarkeit.

Beurteilen wir, wie stark die Schutzziele von einer SQL injection betroffen wären. 0 steht für gar nicht betroffen, 1 steht für teilweise betroffen und 2 steht für kritisch betroffen.

|  |  |  |  |
| --- | --- | --- | --- |
| Szenario | Vertraulichkeit | Integrität (Daten) | Verfügbarkeit |
| Angreifer gibt Liste von Produkten aus | 0 | 0 | 0 |
| Angreifer gibt Liste von User aus | 2 | 0 | 0 |
| Angreifer meldet sich als Admin an | 1 (2) | 2 | 1 |
| Angreifer löscht Tabellen oder sogar DB | 0 | 2 | 2 |

Angreifer gibt Liste von Produkten aus:

* Vertraulichkeit: Solange die Produkte keine sensiblen Informationen beinhalten, ist die Vertraulichkeit nicht betroffen.
* Integrität: Der Angreifer weiss das SQL injections möglich sind, was die Integrität der Daten in Frage stellt.
* Verfügbarkeit: Die Verfügbarkeit der Produkte ist (noch) nicht betroffen

Zwar werden alle Schutzziele als null gewertet, doch der Angreifer weiss nun, das SQL injections möglich sind und er wird dies höchstwahrscheinlich ausnutzen.

Angreifer gibt Liste von User aus:

* Vertraulichkeit: Alle Informationen zum Benutzer wurden preisgelegt, die Vertraulichkeit ist stark betroffen.
* Integrität: Die Daten wurden (noch) nicht verändert, die Gefahr besteht trotzdem.
* Verfügbarkeit: Die Benutzer sind weiterhin verfügbar.

Angreifer meldet sich als Admin an:

* Vertraulichkeit: Je nach Recht des Admins ist die Vertraulichkeit leicht oder stark betroffen.
* Integrität: Als Admin kann der Angreifer höchstwahrscheinlich Änderungen vornehmen und somit daten bearbeiten und löschen.
* Verfügbarkeit: Sollte der Admin Daten löschen können, so ist die Verfügbarkeit betroffen

Angreifer löscht Tabellen oder sogar DB:

* Vertraulichkeit: Der Angreifer löscht die Tabellen, somit gehen die Daten verloren.
* Integrität: Die Daten werden gelöscht, die Integrität ist stark betroffen
* Verfügbarkeit: Die Daten oder vielleicht sogar die Datenbank wird gelöscht, die Verfügbarkeit ist stark betroffen.

Zusätzlich gibt es noch sekundäre Schutzziele. Ich stelle eines der 4 Szenarios dar.

Angreifer meldet sich als Admin an

* Authentizität: Nicht mehr garantiert. Bei einem Admin könnte es sich um einen Angreifer handeln, der sich als Admin ausgibt.
* Nachweisbarkeit: Sind Logging und Triggers implementiert, kann nachgewiesen werden, welcher Admin kompromittiert wurde. Ansonsten nicht betroffen.
* Nicht-Abstreitbarkeit: Sind Logging und Trigger korrekt implementiert, können Operationen nicht abgestritten werden.
* Anonymität: In diesem Fall nicht relevant. (Person, welche Handlungen ausführt, soll nicht identifiziert werden können)

### Risiko

Bewerten wir nun das Risiko einer SQL injection   
(Diese Tabelle habe ich von PR\_183\_Grundbegriffe kopiert)

Injection befindet sich auf den Top 3 (OWASP Top 10), was die Eintrittswahrscheinlichkeit sehr hoch macht.

Der Schaden kann verheerend sein (Ausgabe von persönlichen Daten, CRUD-Operationen)

In diesem Fall soll man das Risiko vermeiden

Niedrig Hoch

Schaden (Impact)

Niedrig Hoch

Eintrittswahrscheinlichkeit  
(Likelihood)

Risiko reduzieren

Risiko reduzieren

Risiko vermeiden

Risiko akzeptieren

## Angriffsfläche

Die Angriffsfläche bezeichnet die Punkte, welcher ein Angreifer ausnutzen kann, um in das System einzudringen. Diese Fläche möchte man so klein wie möglich halten. In Bezug auf Injections müsste man nur so viele User eingaben ermöglichen wie nur nötig. Oder Usereingaben mit vordefinierten Antworten ersetzen.

# Handlungsziel 2

Ich werde die Sicherheitslücke «SQL-Injections» genauer betrachten. Ich werde zeigen, wie man die Ursachen erkennen kann und welche Gegenmassnahmen man vornehmen kann.

## Sicherheitslücke und deren Ursache erkennen

Wie schon erwähnt, werde ich SQL-Injection behandeln. Doch wie kann meine diese Sicherheitslücke erkennen?

Eine Methode ist, selbst probieren. Bei der Insecure app muss man sich mit einem Benutzer anmelden.

Die Zweite Methode: Überprüfen, ob alle Queries parametrisiert wurden.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Oder

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Nach dem clicken auf den login Button melde ich mich mit dem Administrator an.

Ein Bild, das Text, Schrift, Zahl, Software enthält.

Automatisch generierte Beschreibung

Warum funktioniert das? Mithilfe von SSMS lässt sich das zeigen.



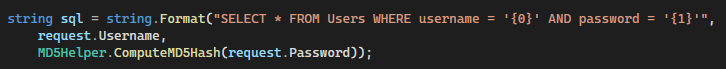
SELECT \* FROM Users 🡪 Hole alle Spalten von Tabelle Users

WHERE username = ‘administrator’ 🡪 Wo der username = administrator ist.

Standardmässig würde noch die folgende Anweisung folgen

AND password = ‘HASHED PASSWORT’

Was auffällt, ist das der letzte Teil mit dem Passwort auskommentiert ist. Wie konnte es dazu kommen? Das Problem: Die Query wurde nicht parametrisiert.



Die eingegebenen Werte können weiterhin, als SQL verstanden werden.

Injection step-by-step:

Administrator’ 🡪 Ich gebe den Usernamen ein und schliesse den String

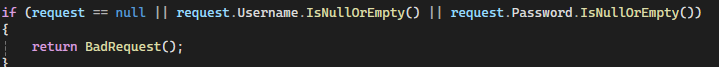
Wenn ich den Namen des Benutzers kenne, kann ich nun die restlichen Anweisungen mithilfe von -- auskommentieren.

Kenne ich den Namen des Benutzers nicht kann ich folgendes anwenden.

OR 1=1 -- 🡪 Oder 1=1, ermöglicht es uns, uns trotzdem anzumelden, auch wenn der Username unbekannt/falsch ist.

Auch hier kommentieren wird die restlichen Anweisungen mit -- aus.

(Beim Passwort ist trotzdem eine Eingabe nötig, denn die Eingabe darf nicht null sein)



Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Dies ist möglich, weil im Backend der erste oder Default Benutzer zurückgegeben wird. Auch wenn man mehrere Benutzer erhält, ist man mit einem garantiert angemeldet.

Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

## Gegenmassnahme Vorschlagen und umsetzen

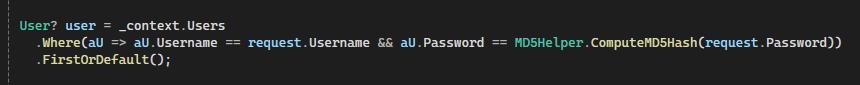
Wie schon erwähnt, besteht diese Sicherheitslücke, weil die Query nicht parametrisiert wurde.

OWASP:

The preferred option is to use a safe API, which avoids using the interpreter entirely, provides a parameterized interface, or migrates to Object Relational Mapping Tools (ORMs).  
**Note:** Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data or executes hostile data with EXECUTE IMMEDIATE or exec().

Als Entwickler Weiss ich nun auch, dass parametrisierte Queries nicht zu 100% sicher sind. Sollte ich jemals stored procedures verwenden muss ich weitere Sicherheitsaspekte in Betracht nehmen.

Mithilfe von LINQ können parametrisierte Queries erstellt werden.



Dank dieser Umsetzung können SQL-Injections nicht mehr ausgeführt werden.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Mit dem richtigen Passwort kann man sich wie gewohnt anmelden   
(Error stammt von den oberen Versuchen)

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Zahl, Schrift, Software enthält.

Automatisch generierte Beschreibung

# Handlungsziel 3

## JWT-Token

JWT-Tokens dienen zum Autorisieren und sie werden üblicherweise erstellt, wenn sich der Benutzer anmeldet.

Damit JWT implementiert werden kann, muss zuallererst das Program.cs angepasst werden und es muss noch ein Package geholt werden.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Ein JWT-Token besteht ausfolgenden Eigenschaften

* Issuer 🡪 Herausgeber
* Audience 🡪 Für wenn der Token gedacht ist
* Claims 🡪 identifiziert den Benutzer anhand von «Behauptungen» oder Claims
* notBefore 🡪 JWT wurde nicht vor diesem Datum ausgehändigt
* expires 🡪 Ablaufdatum des JWT
* signingCredentials 🡪 Geheime Signatur

Ein Bild, das Text, Screenshot, Software, Betriebssystem enthält.

Automatisch generierte Beschreibung

Schlussendlich wird der Token als JWE oder als JWS serialisiert.

Nun wird der Token erstellt, jedoch wird er dem User noch nicht übergeben. Dies ist der nächste Schritt.

Aktuell ist es so, dass wenn sich der User anmeldet, wird sein ganzer Benutzer im local storage gespeichert. Dies ist unsicher, denn mithilfe von XSS könnte ein Angreifer die Werte aus dem local storage auslesen und so an sensible Informationen gelangen.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Da wir nun den JWT-Token erstellen, müssen wir ihn nur noch dem User übergeben.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Dieser ajax call fragt ab, ob sich der Benutzer korrekt angemeldet hat. Am Schluss erhält er den JWT zurück, diesen wollen wir nun speichern. Wo früher die saveUser Methode aufgerufen wurde, steht nun die StoreJwt Methode.

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

Diese speichert den JWT im localstorage des Benutzers ab und beinhaltet keine sensiblen Informationen. Nun wurde der Jwt abgespeichert, aber der Benutzer kann so noch nicht autorisiert werden. Der Token muss noch abgefragt werden können.

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

Diese Methode holt den JWT anhand von der Id

(Beim local storage handelt es sich um ein Key-Value-Storage)

Da wir den Benutzer nicht mehr im local storage speichern, müssen wir einige bestehende Methoden anpassen und eine hinzufügen.

Die Methoden getUserid, getUsername und isAdmin griffen auf den User zu. Nun müssen wir dies mit dem JWT machen.

Zuallererst müssen wir den JWT parsen, damit wir die Werte auslesen können.

Ein Bild, das Text, Schrift, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Nun können die Properties ausgelesen werden.

Als nächstes möchten wir, dass sich der User automatisch anmeldet, sollte er einen JWT besitzen. Dafür müssen wir die isLoggedIn Methode anpassen. Hierzu versuchen wir den JWT, anhand von der Id, zu holen.

Ein Bild, das Text, Schrift, Screenshot, Reihe enthält.

Automatisch generierte Beschreibung

Wenn der Benutzer auf Seiten zugreift, möchten wir kontrollieren, ob er angemeldet ist:

Ein Bild, das Text, Screenshot, Software, Display enthält.

Automatisch generierte Beschreibung

Zuletzt kommt noch das Auslesen des Jwt headers.

Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung

Anhand von dieser Methode können wir Ajax calls absichern und die Autorisierung des Benutzers prüfen.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Zuletzt müssen wir noch die REST-Points absichern, mithilfe vom Authorize Attribut

Ein Bild, das Schrift, Text, Screenshot, Grafiken enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Screenshot, Software, Computersymbol enthält.

Automatisch generierte Beschreibung