Dokumentation

# Inhalt

[Inhalt 2](#_Toc502333403)

[Die Applikation 3](#_Toc502333404)

[Sicherheitsanalyse 4](#_Toc502333405)

[Problembehebung 5](#_Toc502333406)

[Passwort in DB 5](#_Toc502333407)

[Login 6](#_Toc502333408)

[Suche 7](#_Toc502333409)

[Edit 8](#_Toc502333410)

[Testing 10](#_Toc502333411)

# Die Applikation

Unsere Applikation

# Sicherheitsanalyse

|  |  |  |  |
| --- | --- | --- | --- |
| **Schwachstelle** | **Gefahr** | **Klassifizierung** | **Gegenmassnahme** |
| Passwort in DB | Passwort wurde nicht gehasht:  Leichte Beute für Hacker, welche in die DB eindringen | Kann gravierende Folgen haben | Passworthash mit Salt erzeugen. |
| Login | SQL-Injection möglich:  Benutzer kann über Passwortfeld sich selbst zum Admin machen, Passwörter von anderen Benutzern ändern und Benutzer löschen. | Kann teilweise gravierende Folgen haben | Prepared Statements  Serverseitige Feldüberprüfung vor dem Ausführen des SQL-Syntax |
| Suche | SQL-Injection möglich:  Benutzer kann über Suchfeld sich selbst zum Admin machen, Passwörter von anderen Benutzern ändern und Benutzer löschen. | Kann teilweise gravierende Folgen haben | Prepared Statements  Serverseitige Feldüberprüfung vor dem Ausführen des SQL-Syntax |
| Edit | SQL-Injection und XSS möglich:  Kann über die URL, sowie über die Eingabefelder schadhaften SQL-Code einschleusen.  Kann in den Eingabefeldern schadhaften JavaScript-Code einschleusen, welcher dann auch in der DB gespeichert wird. | Kann gravierende Folgen haben | Prepared Statements  Serverseitige Feldüberprüfung vor dem Ausführen des SQL-Syntax |
| Keine Serverseitige Feldüberprüfung | Somit kann auch schadhafter Code in die Datenbank kommen:  XSS im Edit speichert den schadhaften Code in die Datenbank und führt diesem immer beim Neuladen der Seite aus. | Kann gravierende Folgen haben | Strikte Feldüberprüfung und somit diverse Zeichen verbieten. |
| Sensible Eingabefelder nicht genug gesichert. | Passwortfeld wird im Klartext angezeigt. | Kann teilweise gravierende Folgen haben | Zu einem Passwortfeld machen. |

# Problembehebung

## Passwort in DB

Das Passwort in der DB wurde nicht verschlüsselt, es steht im Klartext in der Datenbanktabelle drinnen. Sollte ein Hacker aus der Datenbank Daten entnehmen können, so ist dies danach ohne verschlüsseltes Passwort noch einfacher, die Accounts zu stehlen.



Um es nun den Hackern noch schwerer zu machen, die Accounts zu stehlen, werden wir die Passwörter verschlüsseln und zwar mit der Hash-Funktion. Leider ist ein einfacher Hash des Passworts auch nicht mehr sicher, da man diese Zeichenfolgen auf Webseiten leicht entziffern kann. Vor allem einfache Passwörter wie «abcdefg» oder «1234678» können somit sehr einfach entschlüsselt werden. Damit dies nicht mehr so leicht passieren kann, fügen wir noch einen «Salt» dazu, dies ist ein zusätzlicher Schlüssel, welcher zusammen mit dem Passwort gehasht und in die Datenbank geschrieben wird. Wir nehmen die ID des Benutzers dafür. Wenn nun beim nächsten Mal der Benutzer sich anmelden will, so ist nur das Passwort nicht mehr genügend, weil es nun den Schlüssel nun auch benötigt.

Die Funktion für die endgültige Verschlüsselung sieht dann so aus:

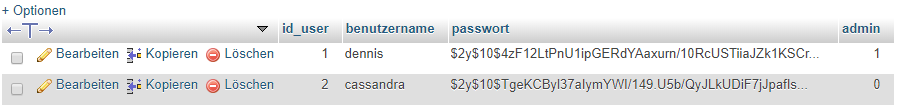
$pw = "panda";

$id = 2;

password\_hash($pw.$id, PASSWORD\_DEFAULT);

‘panda2’ => $2y$10$TgeKCByl37aIymYWI/149.U5b/QyJLkUDiF7jJpafIsvC94.HBpOq

Die Tabelle sieht nun so aus:



## Login

Der darauffolgende SQL-Syntax lässt SQL-Injections zu. Folgende Eingriffe sind dann möglich:

* Benutzer kann sich selbst zum Admin machen
* Passwörter von anderen Benutzern ändern
* Benutzer löschen

Der aktuelle SQL-Query sieht so aus:

$username = ‘cassandra’;

$password = ‘panda’;

$sql = "SELECT id\_user, benutzername, passwort, admin FROM user WHERE benutzername = '$username' AND passwort = '$password'";

Besonders die Variable «password» kann mit einem zusätzlichen SQL-Code ergänzt werden. Die oben genannten Punkte für mögliche Eingriffe werden wir nun alle durchgehen:

Sich selbst zum Admin machen:

$sql = "SELECT id\_user, benutzername, passwort, admin FROM user WHERE benutzername = 'cassandra' AND passwort = 'a'; UPDATE user SET admin = 1 WHERE benutzername = 'cassandra'";

Passwörter von anderen Benutzern ändern:

$sql = "SELECT id\_user, benutzername, passwort, admin FROM user WHERE benutzername = 'cassandra' AND passwort = 'a'; UPDATE user SET passwort = 'hacked' WHERE benutzername = 'dennis'";

Benutzer löschen:

$sql = "SELECT id\_user, benutzername, passwort, admin FROM user WHERE benutzername = 'cassandra' AND passwort = 'a'; DELETE FROM user WHERE benutzername = 'dennis'";

Damit solche Angriffe verteidigt werden können, haben wir mit den Prepared Statements gearbeitet:

$sql = "SELECT id\_user, benutzername, passwort, admin FROM user WHERE benutzername = :username";

$stmt = $db->prepare($sql);

$stmt->execute(array(

'username' => $username

));

Da der Benutzername in der Datenbank ein Unique-Key ist, reicht der Filter für den Benutzernamen. In einer darauffolgenden Überprüfung wird dann auch das Passwort zusammen mit der ID geprüft.

Um dies aber auch noch doppelt abzusichern, haben wir auch noch eine Serverseitige Feldüberprüfung implementiert. Diese wird zusammen mit den Feldeinschränkungen in der Datenbank überprüft:

|  |  |  |
| --- | --- | --- |
| **Name des Feldes** | **Max. zulässige Zeichen** | **Erlaubte Zeichen** |
| Username | 50 | [a-zA-Z][0-9] |
| Password | 128 | [a-zA-Z][0-9][$!?] |

if(!preg\_match("#^[0-9a-zA-Z]+$#", $username)){

header('Location: login.php');

exit;

}

if(!preg\_match("#^[0-9a-zA-Z$!?]+$#", $password)){

header('Location: login.php');

exit;

}

## Suche

Der SQL-Befehl bei der Suche lässt folgendes zu:

* Benutzer kann sich selbst zum Admin machen
* Passwörter von anderen Benutzern ändern
* Benutzer löschen

Der aktuelle SQL-Query sieht so aus:

$sql = "SELECT \* FROM produkte WHERE name LIKE '%$search%'"

Sich selbst zum Admin machen:

$sql = "SELECT \* FROM produkte WHERE name LIKE '%%'; UPDATE user SET admin = 1 WHERE benutzername LIKE '%cassandra%'";

Passwörter von anderen Benutzern ändern:

$sql = "SELECT \* FROM produkte WHERE name LIKE '%%'; UPDATE user SET passwort = 'hacked' WHERE benutzername LIKE '%dennis%'";

Benutzer löschen:

$sql = "SELECT \* FROM produkte WHERE name LIKE '%%'; DELETE FROM user WHERE benutzername LIKE '%dennis%'";

## Edit

Folgende Gefahren sind beim Ändern der Daten möglich:

* Jede Art von SQL-Injection
* Cross Site Scripting

Beim Editieren haben wir die grösste Schwachstelle gefunden. Hier sind nämlich aufgrund des Aufbaus vom SQL-Query alle Art von SQL-Injections möglich und zu gleich kann in den Feldern Schadhafter JavaScript-Code eingeschleust werden und diesen auch noch in der Datenbank speichert.

Die schädlichste Art, einen SQL-Query im PHP zu schreiben und den wir in der Applikation verwenden wäre folgender:

$sql = "SELECT \* FROM produkte WHERE id\_produkt = $id";

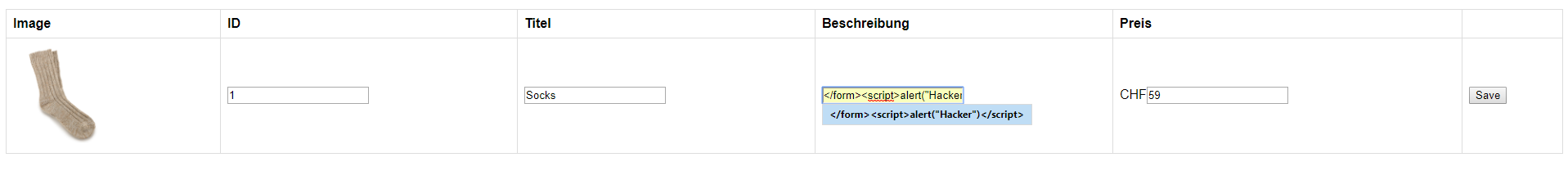
Der SQL-Query ist nämlich so schädlich, weil die Variabel am Ende steht und dazu auch noch ein Zahlenwert verlangt wird. Dadurch muss der Hacker nicht auf die Hochkommas achten und kann nach einem Semikolon einfach einen SQL-Query nach Wahl anhängen. Als Beispiel kann der Hacker einen neuen Benutzer erstellen:

$sql = "SELECT \* FROM produkte WHERE id\_produkt = 1; INSERT INTO user (benutzername, passwort, admin) VALUES ('hacker', 'hacked', 1)";

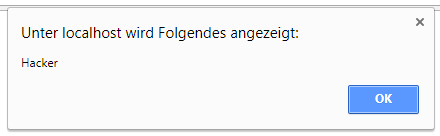
Und Worst-Case:

$sql = "SELECT \* FROM produkte WHERE id\_produkt = 1; DROP DATABASE 'm181\_produktverwaltung'";

Bei der zweiten Gefahr ist es möglich schadhaften JavaScript-Code in die Datenbank zu speichern.



Wie oben im Bild abgebildet, kann man hier einen HTML- und JavaScript-Code per Eingabefeld injiziert werden. Der Grund, warum das Cross Site Scripting hier funktioniert, ist wegen der Datenausgabe im Value. Dadurch wird dies als HTML-Code interpretiert. Der Endtag des Formulars ist zwingend, denn es soll danach der Script-Tag eingeschleust werden. Das HTML erkennt somit dass das Formular zu Ende ist. Im Script drinnen kann man jede Art von Angriff implementieren, wir haben jetzt das einfache Beispiel mit dem Alert gemacht.



Diese Box erscheint nun und jedes Mal, wenn der Benutzer die Seite neu lädt, wird das Script ausgeführt, weil es in der Datenbank gespeichert wurde.

Damit auch dies nicht mehr möglich ist, haben wir bestimmte Zeichen ausgeschlossen:

|  |  |  |
| --- | --- | --- |
| **Name des Feldes** | **Max. zulässige Zeichen** | **Verbotene Zeichen** |
| Name | 50 | <> |
| Beschreibung | - | <> |
| Preis | - | <> |

if(preg\_match("/[<>]/", $titel)){

header('Location: edit.php?id='.$id);

exit;

}

if(preg\_match("/[<>]/", $beschreibung)){

header('Location: edit.php?id='.$id);

exit;

}

# Testing