Praktikum Anwendungssicherheit
Hochschule der Medien Stuttgart

Protokoll

III: Eingriff in die Programmlogik

Verfasser:	Benjamin Zaiser
E-Mail:	bz003@hdm-stuttgart.de
Studiengang:	Computer Science and Media
Semester:	2
Datum:	19. November 2008

Inhaltsverzeichnis

1 Zum Aufbau des Protokolls 3

2 WebGoat: XSS Attacks 3
2.1 Stored XSS Attack; Quelle: Benutzer . . . 3
2.1.1 Hintergrundinformationen . . . 3
2.1.2 Aufgabenstellung . . . 3
2.1.3 Lösung/Durchführung . . . 4
2.1.4 Erkenntnis . . . 5
2.2 Beheben des Fehlers in der Applikation . . . 5
2.2.1 Aufgabenstellung . . . 5
2.2.2 Lösung/Durchführung . . . 5
2.2.3 Erkenntnis . . . 6
2.3 Stored XSS Attack: Quelle: Datenbank . . . 6
2.3.1 Aufgabenstellung . . . 6
2.3.2 Lösung/Durchführung . . . 7
2.3.3 Erkenntnis . . . 8
2.4 XSS Attack: Quelle: Sucheingabefeld . . . 8
2.4.1 Aufgabenstellung . . . 8
2.4.2 Lösung/Durchführung . . . 9
2.4.3 Erkenntnis . . . 10

3 WebGoat: SQL Injection 10
3.1 Hintergrundinformationen . . . 10
3.2 Denial of service attack . . . 11
3.2.1 Aufgabenstellung . . . 11
3.2.2 Lösung/Durchführung . . . 11
3.2.3 Erkenntnis . . . 13
3.3 SQL Injection um die Authentifizierung zu umgehen . . . 13
3.3.1 Aufgabenstellung . . . 13
3.3.2 Lösung/Durchführung . . . 13
3.3.3 Erkenntnis . . . 14
3.4 Möglichkeit einer SQL Injection Attacke bei Authentifierung unterbinden . . . 14
3.4.1 Aufgabenstellung . . . 14
3.4.2 Lösung/Durchführung . . . 15
3.4.3 Erkenntnis . . . 16
3.5 SQL Injection in POST Parametern . . . 16
3.5.1 Aufgabenstellung . . . 16
3.5.2 Lösung/Durchführung . . . 17
3.5.3 Erkenntnis . . . 17
3.6 Möglichkeit der SQL Injection Attacke in POST-Variablen unterbinden . . . 17
3.6.1 Aufgabenstellung . . . 17

3.6.2 Lösung/Durchführung 17

1 Zum Aufbau des Protokolls

Das Protokoll ist wie folgt aufgebaut: zu jeder Übung gibt es ein Kapitel. Jedes Kapitel hat nach Möglichkeit folgende Unterpunkte:

Hintergrundinformationen Informationen, die für die Durchführung der Übung, bzw. für das Verständnis der Aufgabenstellung erforderlich sind.

Aufgabenstellung Welche Aufgabe soll durchgeführt werden; was ist das Ziel der Übung.

Lösung/Durchführung Wie wurde die Aufgabe gelöst; welche Probleme traten dabei auf.

Erkenntnis Was lernt man aus der Aufgabe; welche Erkenntnis könnte in die Entwicklung einer Web-Applikation einfließen.

2 WebGoat: XSS Attacks

2.1 Stored XSS Attack; Quelle: Benutzer

2.1.1 Hintergrundinformationen

Auf einer einfachen dynamischen Website, wie z.B. einem Gästebuch, kann ein Benutzer in einem Eingabefeld eines Formulars eine Nachricht hinterlassen. Das Formular wird an den Server gesendet und dieser speichert den Inhalt des Eingabefeldes in einer Datenbank. Beim Betrachten des Gästebuchs wird der Inhalt aus der Datenbank ausgelesen, HTML Code generiert und zum Client gesendet, welcher die Website dann darstellt.

Generell kann in eine HTML Seite mithilfe des `<script>` Programmcode, wie z.B. Javascript eingebettet werden. Bei der Darstellung der Seite führt der Browser automatisch (sofern Javascript aktiviert ist) das Skript aus. JavaScript ist eine relativ mächtige Sprache, die viele Möglichkeiten bietet. Unter anderem z.B. das Ändern oder Erweitern des HTML Dokuments, Auslesen von Cookies, Auslesen von Browserfunktionen, etc.

Rein theoretisch könnte man nun in das oben genannte Eingabefeld des Gästebuchs keinen normalen Text, sondern ein in JavaScript geschriebenes Programm, das mit dem `<script>` Tag umschlossen wird eintragen. Das Skript wird automatisch in der Datenbank gespeichert und beim Betrachten des Gästebuchs in die HTML Seite integriert. Der Client-Browser stellt die Seite dar, findet das `<script>` Tag und führt das Skript aus.

Dies ist die Basis des Cross-Site Scripting oder auch XSS (Der JavaScript Code könnte nun auch Inhalte von einer anderen Seite laden oder Kontakt mit einem externen Server aufnehmen und z.B. Cookie Informationen versenden oder ähnliches).

2.1.2 Aufgabenstellung

In dem gegebenen System kann sich ein Benutzer durch eine Login-Maske anmelden. Mithilfe der Funktion „Edit Profile“ können Angaben zum Profil geändert werden (wie z.B. die Adresse). Es soll nun versucht werden, in das Adress-Feld eine XSS Attacke zu injizieren.

2.1.3 Lösung/Durchführung

Folgender JavaScript Code:

```
<script>alert("You've got hacked"};</script>
```

wird nun in das „Street“ Feld des Benutzers Tom anstatt eines Strassennamens eingefügt:

Abbildung 1: XSS Attacke im Adress-Feld

Nach dem Absenden des Formulars wird der Code in die Datenbank geschrieben und das geänderte Profil wird für die Anzeige gleich wieder zum Client zurückgesendet. Der Browser erkennt das Script-Tag und führt den Programm-Code aus.

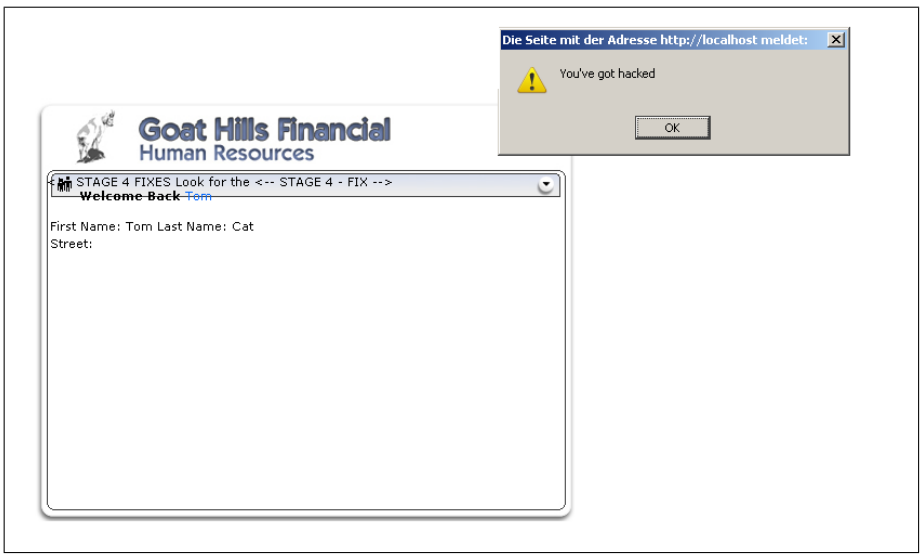

Abbildung 2: XSS Attacke im Adress-Feld

Da das Skript nun in der Datenbank gespeichert ist, wird es immer dann ausgeführt, sobald jemand das Profil von Tom anfordert. Sobald z.B. Jerry das Profil von Tom anschauen will, wird auch bei ihm der Javascript Code ausgeführt.

2.1.4 Erkenntnis

Es wird hier an zwei Stellen auf die Validierung von Inhalten verzichtet. Die erste Stelle befindet sich beim Verarbeiten des gesendeten Formulars, noch bevor die Daten in die Datenbank geschrieben werden. Bereits hier sollte überprüft werden, ob die Eingaben korrekt sind, bzw. nur gültige Zeichenfolgen enthalten. Es sollte an dieser Stelle ein Whitelisting durchgenommen werden. Das bedeutet, es wird festgelegt, welche Zeichen / Zeichenfolgen ein Feld beinhalten darf. Kommen ungültige Zeichen / Zeichenfolgen vor, sollte die Verarbeitung mit einem Fehler abgebrochen werden. Die zweite Stelle ist bei der Verarbeitung des Requests „ViewProfile“, wenn die Daten aus der Datenbank ausgelesen und zum Client gesendet werden. Es kann nicht davon ausgegangen werden, dass die Datenbank nur gültige Zeichenfolgen enthält. Deswegen muss auch hier eine Gültigkeitsprüfung vorgenommen werden.

Der Fehler des Entwicklers ist, dass er den Datenquellen blind vertraut und die Daten keiner Prüfung unterzieht. Die Datenquelle ist in diesem Falle natürlich der Benutzer, der das Formular abgesendet hat, aber auch die Datenbank kann Eingaben des Benutzers enthalten und darf nicht unbeachtet bleiben.

2.2 Beheben des Fehlers in der Applikation

2.2.1 Aufgabenstellung

Der Fehler aus der vorhergehenden Übung soll nun im Programm-Code behoben werden.

2.2.2 Lösung/Durchführung

Zunächst wurde die entsprechende Klasse gesucht, die das Formular mit den eingegebenen Daten des Benutzer verarbeitet und in die Datenbank schreiben wird. In der Funktion `parseEmployeeProfile` werden die einzelnen Werte des Requests den jeweiligen Variablen zugeordnet, die dann in das SQL Statement einfließen. Bei dieser Zuordnung muss nun die Prüfung auf die Gültigkeit hin stattfinden.

Das Überprüfen von Eingabe-Strings auf ungültige Zeichen sollte stets mithilfe von White-Listing erfolgen. Das heißt, es wird festgelegt, welche Daten gültig sind und nicht welche Daten ungültig sind (Black-Listing). Für diese Prüfung eignen sich Regular Expressions hervorragend. Jedoch ist die Schreibweise dieser Expressions sehr ungewohnt und erfordert eine (mitunter langwierige) Einarbeitung.

Mithilfe der folgenden Expression werden Groß- und Kleinbuchstaben, sowie Zahlen in beliebiger Reihenfolge erlaubt:

```
[a-zA-Z0-9]*
```

Durch die Funktion `validate(zu untersuchender String, RegEx)` kann nun die Prüfung vorgenommen werden.

```
protected Employee parseEmployeeProfile(int subjectId, WebSession s) throws ParameterNotFoundException,
        ValidationException
{
    // The input validation can be added using a parsing component
    // or by using an inline regular expression. The parsing component
    // is the better solution.

    HttpServletRequest request = s.getRequest();
    String firstName = request.getParameter(CrossSiteScripting.FIRST_NAME);
    String lastName = request.getParameter(CrossSiteScripting.LAST_NAME);
    String ssn = request.getParameter(CrossSiteScripting.SSN);
    String title = request.getParameter(CrossSiteScripting.TITLE);
    String phone = request.getParameter(CrossSiteScripting.PHONE_NUMBER);

    String address1 = validate(request.getParameter(CrossSiteScripting.ADDRESS1), Pattern.compile("[a-zA-z0-9]*"));
```

Abbildung 3: Überprüfen des Eingabestrings

2.2.3 Erkenntnis

Es muss jeder Eingabestring einer Prüfung mit den entsprechenden Regular Expressions unterzogen werden. Diese Prüfung kann relativ schnell und ohne großen Aufwand in bestehende Software eingebaut werden. Das Schreiben der Regular Expressions benötigt unter Umständen jedoch sehr viel Zeit und Erfahrung.

2.3 Stored XSS Attack: Quelle: Datenbank

2.3.1 Aufgabenstellung

Wie in 2.1.4 bereits dargestellt, reicht es nicht aus die Daten, die vom Benutzer gesendet werden, einer Prüfung zu unterziehen. Es müssen auch die Daten, die aus der Datenbank stammen einer Prüfung unterzogen werden. Wenn z.B. eine böswillige Person Zugriff auf die Datenbank bekommt, könnte sie XSS Attacken einfügen, ohne das Formular zu benutzen.

In der Datenbank befindet sich nun SkriptCode. Trotz des „Security Patches“ aus der vorhergehende Aufgabe, der die Benutzereingaben einer Prüfung unterzieht, kann der Code, der sich noch in der Datenbank befindet eine Attacke ausführen. Die Applikation soll nun dahingehend erweitert werden, dass auch dieses Sicherheits-Loch behoben wird.

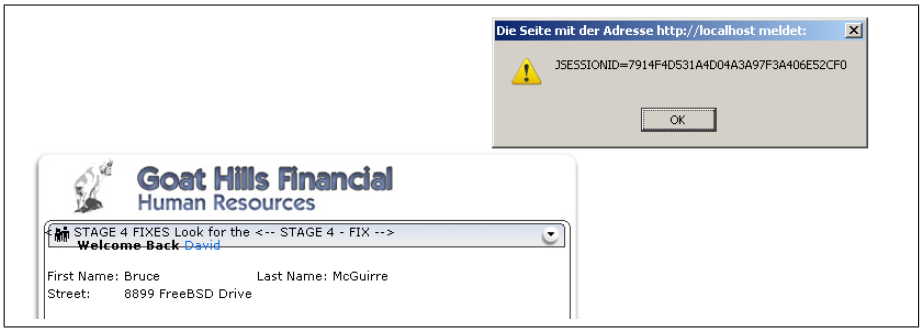

Abbildung 4: XSS Attacke, die noch in der DB gespeichert war

Die Applikation soll nun so erweitert werden, dass Skript-Code, der sich in der Datenbank befindet, beim Auslesen außer Kraft gesetzt wird.

2.3.2 Lösung/Durchführung

Zunächst wurde wieder nach der entsprechenden Klasse gesucht, die die Daten aus der Datenbank ausliest und das HTML Dokument zusammenstellt. An der entsprechenden Stelle wurde nun einfach das < Zeichen durch ein leeres Zeichen ersetzt. Das HTML Tag ist somit ungültig und wird als normaler Text dargestellt.

```
public Employee getEmployeeProfile(WebSession s, int userId, int subjectUserId) throws UnauthorizedException
{
    Employee profile = null;


    // Query the database for the profile data of the given employee
    try
    {
        String query = "SELECT * FROM employee WHERE userid = " + subjectUserId;
        try
        {
            Statement answer_statement = WebSession.getConnection(s)
                    .createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
            ResultSet answer_results = answer_statement.executeQuery(query);
            if (answer_results.next())
            {

                String address1 = answer_results.getString("address1").replace('<', ' ');

                // Note: Do NOT get the password field.
                profile = new Employee(answer_results.getInt("userid"), answer_results.getString("first_name"),
                        answer_results.getString("last_name"), answer_results.getString("ssn"), answer_results
                                .getString("title"), answer_results.getString("phone"),
                                address1, answer_results.getString("address2"), answer_results
                                .getInt("manager"), answer_results.getString("start_date"), answer_results
                                .getInt("salary"), answer_results.getString("ccn"), answer_results
                                .getInt("ccn_limit"), answer_results.getString("disciplined_date"), answer_results
                                .getString("disciplined_notes"), answer_results.getString("personal_description"));
```

Abbildung 5: Prüfen der „Eingaben“ aus der Datenbank

Abbildung 6: Der Skript-Code wurde erfolgreich außer Kraft gesetzt

2.3.3 Erkenntnis

Diese Lösung gilt vielleicht für die vorliegende Aufgabenstellung, kann aber nicht produktiv eingesetzt werden. Ein Ersetzen der <-Zeichen reicht nicht aus, da XSS Attacken auch auf andere Art und Weise injiziert werden können. Die vorliegende Lösung wird bei der Überprüfung von mehreren Feldern / Strings schnell sehr umfangreich und der Code wird sehr unübersichtlich. Eine Methode, die das ResultSet als Parameter erhält und dann jeden String einer Prüfung mithilfe von Regular Expressions unterzieht wäre von Vorteil.

2.4 XSS Attack: Quelle: Sucheingabefeld

2.4.1 Aufgabenstellung

Die Anwendung bietet eine Suchfunktion, mit der man nach verschiedenen Personen, durch Eingabe eines Namens, im System suchen kann.

Abbildung 7: Suchfeld, das mit JavaScript Code gefüllt wird

Auch dieses Eingabefeld ist anfällig für XSS Attacken.

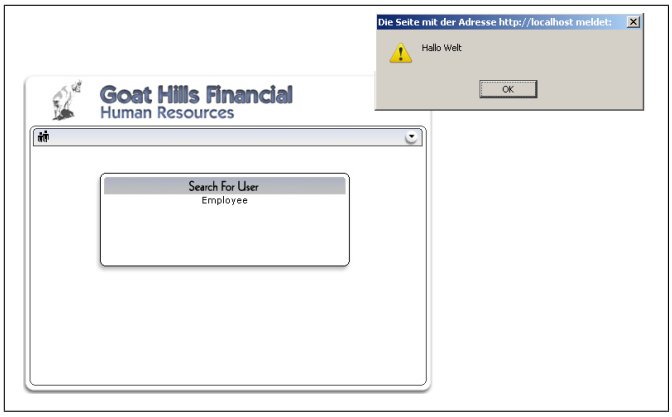

Abbildung 8: Nach Absenden des Formulars, wird der Code ausgeführt

2.4.2 Lösung/Durchführung

Wieder wurde die entsprechende Klasse gesucht, welche den Such-Request bearbeitet. Wie bereits in Aufgabe 1 wird der Suchstring nun mithilfe einer RegularExpression validiert. Bei der Suche nach Namen dürfen nur Groß- und Kleinbuchstaben eingegeben werden. Hierfür reicht folgende Regular Expression aus.

```
[a-zA-Z]*
```

```
public void handleRequest(WebSession s) throws ParameterNotFoundException, UnauthenticatedException,
        UnauthorizedException, ValidationException
{
    if (isAuthenticated(s))
    {
        int userId = getIntSessionAttribute(s, getLessonName() + "." + CrossSiteScripting.USER_ID);

        String searchName = null;
        try
        {
            searchName = getRequestParameter(s, CrossSiteScripting.SEARCHNAME);
            searchName = validate(searchName, Pattern.compile("[a-zA-Z]*"));
```

Abbildung 9: Validierung des Suchstrings mit Regular Expression

2.4.3 Erkenntnis

Es muss grundsätzlich jede Quelle auf gültige Zeichen / Zeichenfolgen hin untersucht werden.

3 WebGoat: SQL Injection

3.1 Hintergrundinformationen

Benutzereingaben werden oft für dynamische Datenbankabfragen mit SQL verwendet. Bei einem Login wird z.B. Username und Passwort in eine SQL Abfrage eingefügt, die dann alle Benutzerdaten aus der Datenbank liefert. Die Abfrage könnte so aussehen:

```
SELECT * FROM fe_users WHERE username="$username" AND password="$password";
```

Anstelle von `$password` wird dann das Passwort aus dem Loginfeld eingesetzt. Für Username „foo“ und Passwort „bar“ würde folgendes SQL Statement entstehen:

```
SELECT * FROM fe_users WHERE username="foo" AND password="bar";
```

Man könnte nun aber anstatt des Passworts auch folgenden String eingeben: „bar " AND email="test@test.de“ und folgendes Statement würde generiert werden:

```
SELECT * FROM fe_users

WHERE username="foo" AND password="bar " AND email="test@test.de";
```

Dies wäre eine SQL Injection Attacke. Es wird eine Zeichenkette übermittelt, welches das Subsystem nicht als Text sondern als Steuerzeichen erkennt. Dies ermöglicht einem Angreifer, Zugang zu diversen Daten oder gar Daten zu ändern oder zu löschen.

3.2 Denial of service attack

3.2.1 Aufgabenstellung

Durch ein Login-Formular kann sich ein Benutzer an einem System anmelden. Jedem angemeldeten Benutzer wird eine Datenbankverbindung zugewiesen. Die Applikation hat einen Pool von insgesamt zwei Datenbankverbindungen. Wenn sich nun mehr als drei Benutzer gleichzeitig anmelden, ist die Applikation ausgelastet und steht für die weiteren Besucher nicht mehr zur Verfügung.

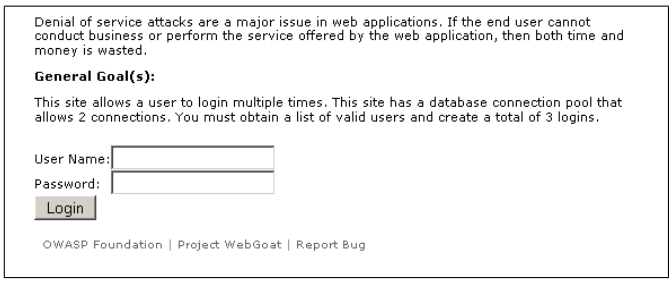

Abbildung 10: Login-Formular für gleichzeitig maximal 2 Benutzer

Die Applikation soll nun „stillgelegt“ werden, indem sich zwei Benutzer gleichzeitig einloggen. Für die Logininformationen soll mithilfe einer SQL Injection Attacke die Datenbank ausgelesen werden.

3.2.2 Lösung/Durchführung

Zunächst wurde versucht sich mit frei erfundenen Login-Informationen anmelden (z.B. Username: root, Passwort: root)

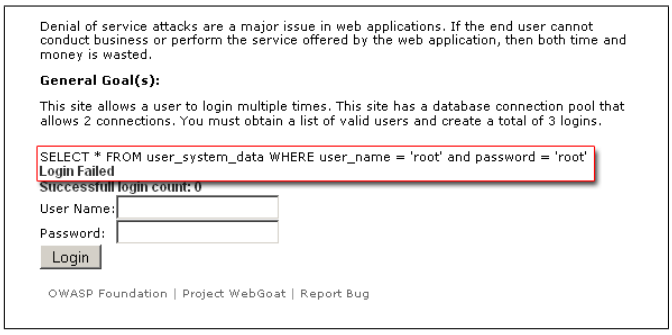

Abbildung 11: Login mit Username: root, Passwort: root

Freundlicherweise gibt die Applikation zusätzlich zu der Information „Login failed“ das ausgeführte SQL Statement zurück.

```
SELECT * FROM user_system_data WHERE user_name = 'root' and password = 'bar'
```

Wie man sieht, enthält der Where-Clause eine Und-Verknüpfung. Sobald diese Bedingung für einen Datensatz in der Tabelle wahr ist, wird dieser zurückgegeben. Erkennt man diese logische Operation als solche und besitzt man weiteres Wissen über logische Operatoren, kann eine SQL Injection Attacke angewendet werden. Ziel ist, dass die Bedingung im Where-Clause wahr wird. Dies ist auch der Fall, wenn man die Und-Bedingung geschickt mit einer Oder-Bedingung erweitert. Man verknüpft also mit dem Oder-Operator die erste Bedingung mit einer zweiten Bedingung, die in jedem Fall wahr ist, wie z.B. „1=1“. Diese Bedingung muss nur noch so verpackt werden, dass die Datenbank den String als Steuerzeichen erkennt:

```
Username: egal
Passwort: egal' or '1=1
```

Dies ergibt folgendes SQL-Statement:

```
SELECT * FROM user_system_data

WHERE user_name = 'egal' and password = 'egal' or '1=1'
```

Die Datenbank führt die Query aus und liefert die gesamte Benutzertabelle mit den Klartext-passwörtern:

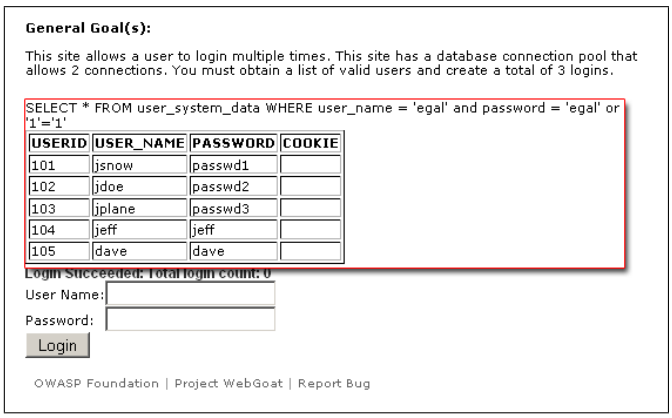

Abbildung 12: Durch SQL Injection auslesen der Benutzerdaten

Nun kann man sich zwei verschiedenen Login-Daten gleichzeitig anmelden und die Applikation akzeptiert keine weiteren Benutzer mehr.

Abbildung 13: Denial of service

3.2.3 Erkenntnis

Das Problem ist, dass durch Eingabe des Steuerzeichens ' Einfluss auf das SQL Statement genommen werden kann. Würde die Benutzereingabe geprüft und entsprechend escaped [1] werden, hätte dies keinen Einfluss mehr auf das SQL Statement. Deswegen sollten immer Eingaben des Benutzers „unschädlich“ gemacht werden. Eine weitere Fahrlässigkeit des Entwicklers ist es die Passwörter im Klartext in der Datenbank zu speichern. Wären die Passwörter verschlüsselt, so hätte der Angreifer keine Verwendung für den Hash-Wert (solange eine gute Hash-Funktion verwendet wird).

3.3 SQL Injection um die Authentifizierung zu umgehen

3.3.1 Aufgabenstellung

Bei dem gegebenen Login-Formular soll die Authentifizierung des Benutzers „Neville“ mithilfe einer SQL Injection Attacke umgangen werden ohne das echte Passwort zu kennen.

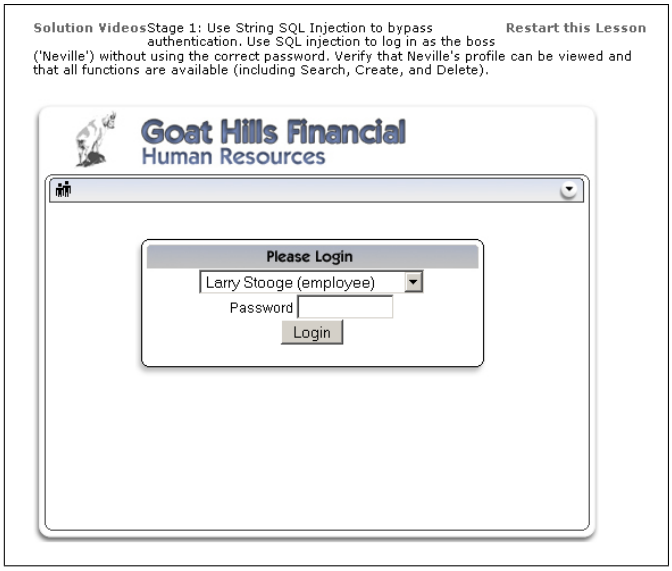

Abbildung 14: Login umgehen durch SQL Injection Attacke

3.3.2 Lösung/Durchführung

Diese Aufgabe kann nach dem gleichen Paradigma wie bereits die vorhergehende Aufgabe gelöst werden. Jedoch ist hierbei noch zu beachten, dass das Eingabefeld auf eine bestimmte Anzahl an Zeichen begrenzt ist. Diese Begrenzung lässt sich jedoch leicht mit der FireBug Extension des Firefox Browsers umgehen. Die „maxlength“ Angabe muss einfach auf einen hohen Wert verändert werden (z.B. 500).

[1] Steuerzeichen werden mit durch ein vorangestelltes \ Zeichen wie normale Textzeichen (CDATA) behandelt

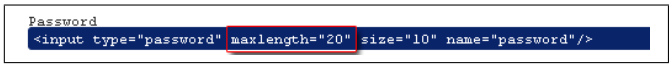

Abbildung 15: Maxlength Angabe beim Passwort Feld verändern

Dann kann mithilfe des folgenden Strings im Passwort-Feld die SQL Injection durchgeführt werden.

```
test' or '1'=1
```

Und der Benutzer wird eingeloggt.

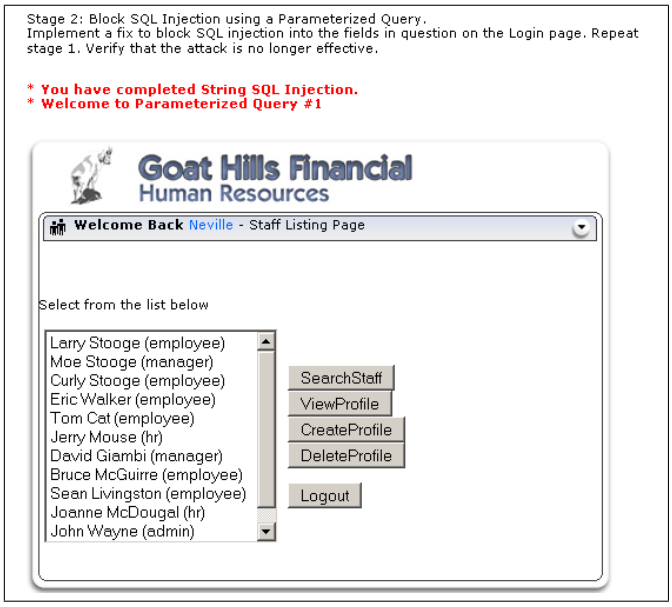

Abbildung 16: Erfolgreicher Login ohne Passwort-Kenntnis

3.3.3 Erkenntnis

Interessant ist, dass man mit dieser eigentlich einfachen Attacke Zugriff auf Benutzerkonten (sogar Administrator-Konten) bekommen kann. Umso wichtiger ist es für den Entwickler, dass gerade bei Login-Formularen die Benutzereingaben überprüft und validiert, bzw. sog. Prepared Statements verwendet werden (mehr dazu später).

3.4 Möglichkeit einer SQL Injection Attacke bei Authentifierung unterbinden

3.4.1 Aufgabenstellung

Die Applikation soll nun so verändert werden, dass die zuvor beschriebene SQL Injection Attacke bei der Authentifizierung unterbunden wird.

3.4.2 Lösung/Durchführung

Zuerst wurde wieder die entsprechende Klasse gesucht, die den vom Client abgesendeten Request mit den Credentials verarbeitet. Es wurde festgestellt, dass die eigentliche Authentifizierung aber in einer anderen Methode stattfindet.

```
public void handleRequest(WebSession s) throws ParameterNotFoundException, ValidationException
{
    // System.out.println("Login.handleRequest()");
    getLesson().setCurrentAction(s, getActionName());

    List employees = getAllEmployees(s);
    setSessionAttribute(s, getLessonName() + "." + DBSQLInjection.STAFF_ATTRIBUTE_KEY, employees);

    String employeeId = null;
    try
    {
        employeeId = s.getParser().getStringParameter(DBSQLInjection.EMPLOYEE_ID);
        String password = s.getParser().getRawParameter(DBSQLInjection.PASSWORD);

        // Attempt authentication
        boolean authenticated = login(s, employeeId, password);
```

Abbildung 17: handleRequest Methode

Die Signatur der Methode, die die Authentifierzung durchführt lautet:

`boolean: login(Session, EmployeeId, Password)`

In der Methode wurde dann der Fehler entdeckt, der die SQL Injection Attacke ermöglicht: der Wert des Passwort Feldes wird, so wie er ist, in ein SQL Statement eingefügt. Es bietet sich nun an, ein sogenanntes „Prepared Statement“ für die SQL Abfrage zu verwenden. Die Variablen im Statement werden mit einem „?“ ersetzt. Danach wird jede Variable der Reihe nach von links nach rechts mit der Variable aus dem Request entsprechend gesetzt. Ein Integer-Wert wird mit der Methode „setInt“, ein String mit der Methode „setString“, usw. gesetzt.

```
public boolean login(WebSession s, String userId, String password)
{
    // System.out.println("Logging in to lesson");
    boolean authenticated = false;

    try
    {
        //String query = "SELECT * FROM employee WHERE userid = " + userId + " and password = '" + password + "'";
        // System.out.println("Query:" + query);

        PreparedStatement query = WebSession.getConnection(s).prepareStatement("SELECT * FROM employee WHERE userid = ? and password = ?",
                ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
        query.setString(1, userId);
        query.setString(2, password);

        try
        {
            //Statement answer_statement = WebSession.getConnection(s)
            //      .createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
            //ResultSet answer_results = answer_statement.executeQuery(query);
            ResultSet answer_results = query.executeQuery();
```

Abbildung 18: Prepared Statement

Die Java-VM setzt nun den String `test' or '1'=1` in das Statement ein, behandelt dabei aber jedes Zeichen als Text-Zeichen (CDATA) und nicht als Steuerzeichen. So wird die SQL Injection Attacke erfolgreich außer Kraft gesetzt.

Abbildung 19: Attacke wurde abgewehrt

Bei dem Prepared Statement muss unbedingt beachtet werden, bestimmte Konfigurationen des ursprünglichen Statements zu übernehmen, wie z.B. „ResultSet.TYPE_SCROLL_INSENSITIVE“ und „ResultSet.CONCUR_READ_ONLY“. Diese wurde anfangs nicht beachtet und der Request endete in einer Fehlermeldung.

3.4.3 Erkenntnis

Um solche Schwachstellen in der Software-Architektur gleich von Anfang an zu unterbinden, sollte man stets Prepared Statements für SQL Abfragen einsetzen. Prepared Statements sind nicht nur unter Java verfügbar, sondern auch in vielen anderen Programmiersprachen. Und falls nicht, so sollte der Datenbankzugriff stets durch einen Abstraktionslayer gekapselt werden. In diesem Layer kann dann das Behandeln der Inputparameter (notfalls von Hand) stattfinden.

3.5 SQL Injection in POST Parametern

3.5.1 Aufgabenstellung

Nachdem sich ein Benutzer erfolgreich angemeldet hat kann mithilfe der Funktion „ViewProfile“ die Daten eines anderen Benutzers angezeigt werden. Dies ist aber nur dann möglich, wenn der angemeldete Benutzer auch das Recht hat, die Daten des ausgewählten Benutzers anzeigen zu lassen. Die POST-Parameter lauten wie folgt:

```
employee_id=101&action=ViewProfile
```

Es soll nun ein Profil eines Benutzers angezeigt werden, für den der angemeldete Benutzer eigentlich keine Berechtigung besitzt.

3.5.2 Lösung/Durchführung

So wie beim Login-Formular in der vorhergehenden Übung, könnten auch die POST-Parameter anfällig für eine SQL Injection Attacke sein. Zunächst muss wieder die Abfrage der employee_id umgangen werden. Dies geschieht mithilfe der OR Verknüfung mit einer wahren Bedingung. Um aber nun den gewünschten Benutzer zu erhalten, muss noch das „ORDER BY“ Schlüsselwort angefügt und die entsprechende Spalte erraten werden. Die modifizierten POST-Parameter sehen dann so aus:

```
employee_id=101 OR 1=1 ORDER BY salary DESC &action=ViewProfile
```

Der angemeldete Benutzer Larry erhält so Einblick in das Profil von Benutzer Neville (der, mit dem höchsten Gehalt).

Abbildung 20: Zugriff auf Profil außerhalb des eigentlichen Rechte-Sets

3.5.3 Erkenntnis

Es muss grundsätzlich *jede* Eingabe auf mögliche Attacken hin untersucht werden, nicht nur die Formulare. Kurz: alles was auf irgend eine Art und Weise vom Client kommt, muss besonders behandelt werden.

3.6 Möglichkeit der SQL Injection Attacke in POST-Variablen unterbinden

3.6.1 Aufgabenstellung

Das in der vorhergehenden Aufgabe beschriebene Security-Problem soll nun behoben werden.

3.6.2 Lösung/Durchführung

Die Lösung ist analog zur der Behebung der SQL Injection bei dem Login-Formular. Das SQL Statement wird mithilfe eines Prepared Statements gekaspelt.

```
public Employee getEmployeeProfile(WebSession s, String userId, String subjectUserId) throws UnauthorizedException
{
    Employee profile = null;

    // Query the database for the profile data of the given employee
    try
    {
        //String query = "SELECT employee.* "
        //      + "FROM employee,ownership WHERE employee.userid = ownership.employee_id and "
        //      + "ownership.employer_id = " + userId + " and ownership.employee_id = " + subjectUserId;

        PreparedStatement query = WebSession.getConnection(s).prepareStatement("SELECT employee.*"
                + "FROM employee,ownership WHERE employee.userid = ownership.employee_id and "
                + "ownership.employer_id = ? and ownership.employee_id = ?",
                ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
        query.setString(1, userId);
        query.setString(2, subjectUserId);
        try
        {
            //Statement answer_statement = WebSession.getConnection(s)
            //      .createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
            //ResultSet answer_results = answer_statement.executeQuery(query);
            ResultSet answer_results = query.executeQuery();
```

Abbildung 21: Zugriff auf Profil außerhalb des eigentlichen Rechte-Sets

Dadurch wird die Attacke auch bei POST-Parametern unterbunden und die Anfrage wird mit einer Fehlermeldung abgebrochen.

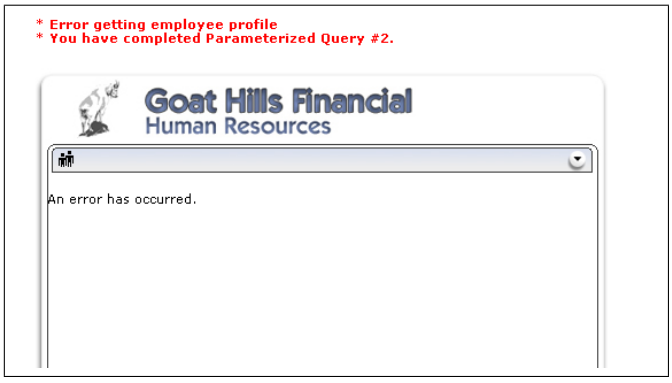

Abbildung 22: Anfrage wird mit Fehlermeldung abgewiesen