TU Wien
Institut für Rechnergestützte Automation
Forschungsgruppe Industrial Software
Arbeitsgruppe ESSE
http://security.inso.tuwien.ac.at/

Abgabedokument Lab1

Security for Systems Engineering

183.637 - SS 2015

14.05.2015

Gruppe 05

Name	**MatrNr.**
Marcel Gredler	1325175
Maximillian Moser	1326252
Roman Tonigold	1327192
Rafał Włodarski	1327160

Inhaltsverzeichnis

1 Lab1a

Nachdem wir uns mit dem Webserver verbunden hatten haben wir uns einmal angesehen welche Funktionalitäten auf der Homepage zur Verfügung stehen. Hierbei haben wir die Funktionen ausprobiert und uns zudem den Source-Code aller angesehen. Während dieser Prozedur sind uns zwei Orte aufgefallen an denen eine Schwachstelle verborgen sein könnte und zwar die Such-Funktion und Kontaktierung. // Auf den ersten Blick kannten wir keinen Angriff der auf CGI-Skripts oder MAIL gerichtet war, da wir jedoch den Hinweis hatten das ein erst vor kurzem gefundener Fehler auf der Seite versteckt ist haben wir eine kleine Recherche durchgeführt und sind hierbei auf den „Shellshock“ gestoßen. Natürlich haben wir diesen Angriff nun ausprobiert, um zu sehen ob wir hiermit auf den Server Zugriff erlangen. Mittels

wget –user=user05 –password=d6/0g0Um1zlAmfYF6tA32Q== -U ”() test;;echo \”Content-type: text/plain\”; echo; echo; /bin/cat /etc/passwd” http://localhost:8805/cgi-bin/search

haben wir getestet ob wir auf dem Server etwas auslesen können, und da wir auf diesen Versuch eine Antwort bekommen hatten, konnten wir über dieses Pattern alle notwendigen Informationen erlangen.

1.1 Warum funktioniert Shellshock?

In einem Shellskript ist es möglich Variablen und Funktionen zu definieren. Um diese definierten Funktionen und Variablen verwenden zu können müssen sie aufrufbar sein, was

im Shellskript dadurch gemacht wird, dass Variablen und Funktionen in Umgebungsvariablen exportiert werden, welche später aufgerufen werden können. Wenn nun Parameter übergeben werden, werden diese in eine Umgebungsvariable „geparst“ und eventuell anhängende Funktionen ebenfalls, jedoch in eine mit anderem Namen. Die Funktion selbst ist somit harmlos, die Problematik besteht nun jedoch darin das zusätzlich angehängte Befehle an die Funktion sofort (und ohne Überprüfung) aufgerufen werden, was heißt, dass die Funktion nie aufgerufen werden muss um Schaden anzurichten.

Warum funktioniert dies nun auf dem Webserver?
Die Antwort hierauf ist leicht, der Server verwendet für die Suchfunktion ein CGI-Skript, welches selbst über die Bash aufgerufen wird. Die übergebene Suchinformation wird hierbei als Variable an die Bash übergeben, wodurch wir genau in den oben beschriebenen Fehler hinein gelangen.

1.2 Was kann im System verbessert werden?

Die Schwachstellen im System sind zum einen die Bash-Version (in neuen Versionen wurde ein Update durchgeführt um die Lücke zu schließen) und die fehlende Input Validierung. Würde das System die Eingabe im Suchfeld vor dem Abschicken an das CGI-Skript mittels Whitelisting überprüfen, könnte der Shellshock-Angriff gefiltert und somit vermieden werden, wodurch auch ältere Bash-Versionen keine Probleme mehr bereiten. Eine andere Lösung wäre nichtsdestotrotz das Update der Bash auf dem Server um die Lücke in der Bash selbst zu schließen (hierbei ist das Problem das Updates auf Server oft nur sehr schwer, z.B. viele Sicherheitstests und Trockenläufe, durchführbar sind).

2 Lab1b

3 Lab1c

3.1 app0

3.1.1 Schwachstelle

Die Implementierte Schwachstelle in diesem Code ist „Injection“, genauer gesagt „SQL-Injection“.
Bei „Injection“ Angriffen geht es darum in einem Datenabfrage-Protokoll (wie z.B. SQL) Befehle einzuschläußen, um Attacken durchzuführen. Damit dies möglich ist, muss man in der Lage sein die Meta-Symbole des Protokolls verwenden zu können (z.B. Kommentar, neues Kommando, ...).

Wo befindet sich der Fehler im Code?

Wie man im Listing 1 sehen kann, werden die Übergebenen Argumente einfach gelesen und verwendet (Zeilen 8 und 30). Da die Meta-Symbole in diesen Zeilen nicht entfernt werden und die Abfrage in Zeile 30 erst erstellt wird, können weitere Befehle eingeschläußt werden, bzw. der derzeitige Befehlt modifiziert werden.

Wie kann der Fehler behoben werden?

Um den Fehler zu beheben könnte man anstelle von normalen Statements, Prepared-Statements verwenden. Eine andere Möglichkeit wäre es die erhaltenen Informationen zu „Whilelisten“, um dadurch alle Meta-Symbole zu entfernen.

3.1.2 Ausnutzen der Schwachstelle

Um die Schwachstelle ausnutzen zu können müssen die folgenden Schritte getätigt werden:

- (H2-Datenbank muss gestartet und DB erstellt sein)
- anstatt einfach den Namen einer Person anzugeben, kann man ein SQL Kommando übergeben. Zum Beispiel könnte man statt „Manro“, „D"); DROP TABLE person –“ verwenden
- der Name muss beim Starten des Programms angegeben werden

java -jar app0a.jar <name-exploit>

3.1.3 Korrektur der Schwachstelle

Die Schwachstelle im Programm ist aufgehoben worden, indem im Code anstatt von Statements, PreparedStatements verwendet wurden (dies ist in Listing 2 zu sehen). Wieso ist dieser Code nun sicher?
Bei PreparedStatements sind die Abfragen bereits erstellt, zu dem Zeitpunkt indem die Parameter übergeben werden. Die Abfragen wurden beim Erstellen des PreparedStatements bis auf die „?“ bereits optimiert und es werden anschließend nur noch Primitive erlaubt.

3.1.4 Wo ist diese Schwachstelle aufgetreten?

- Im Oktober 2014 wurde eine Schwachstelle für SQL-Injection im Playstation Network gefunden. Obwohl diese Schwachstelle nicht ausgenutzt worden ist (wurde von einem Sicherheitsexperten gefunden), gab es vor dem Fix dieser Schwachstelle die Möglichkeit mittels einfachem Injection über den Browser die Nutzerdaten von Millionen-Usern auszulesen.
  Quelle: http://www.golem.de/news/sql-injection-sicherheitsluecke-erlaubt-zugriff-auf-sony-kundendaten-1410-110199.html (abgerufen am 14.05.2015 um 16:35)

- Im April 2015 haben australische Forscher eine Schwachstelle in der MedicalApp von SAP gefunden, mit der es möglich ist SQL-Injection durchzuführen. Ein derartiger Angriff würde es einem erlauben die Daten aus der Datenbank auszulesen, oder neue Daten einzufügen (neue Patientendaten). Zu diesem Zeitpunkt gibt es glücklicherweise noch keine Mitteilungen das diese Schwachstelle auf bösartige Weise ausgenutzt worden wäre.
  Quelle: http://customstoday.com.pk/researchers-found-sql-injection-flaw-in-sap-medical-app-allow-other-apps-to-get-access-to-emr-unwired-database-3/ (abgerufen am 14.05.2015 um 16:45)

- Aufgrund einer Schwachstelle in Magento (April 2015) sind 100K Webseiten gefährdet. Diese Schwachstelle erlaubt es mittels SQL-Injection einen neuen Admin-User in die Datenbank einzufügen und mittels diesem Kontrolle über eben jene zu erhalten. Seit dem Bekanntwerden dieser Schwachstelle wurden bereits einige Angriffe auf Webseiten gestartet, welche diese Schwachstelle angreifen.
  Quelle: http://arstechnica.com/security/2015/04/potent-in-the-wild-exploits-imperil-customers-of-100000-e-commerce-sites/ (abgerufen am 14.05.2015 um 16:50)

- Im April 2015 wurde die Firma AussieTravelCover mittels SQL-Injection angegriffen. Mithilfe dieses Angriffs ist es den Angreifer möglich gewesen die Daten von mehr als 750K Kunden zu erhalten. Um den Fehler zu beheben wurde die Webseite für einen guten Monat offline genommen, zudem sind die Behörden eingeschalten worden, jedoch bis jetzt ohne Festnahmen.
  Quelle: http://news.softpedia.com/news/Aussie-Travel-Cover-Hacked-Over-750-000-Customer-Records-Exposed-470583.shtml (abgerufen am 14.05.2015 um 17:05)

3.2 app2

3.2.1 Schwachstelle

Art der Schwachstelle:

CWE-930: OWASP Top Ten 2013
Category A2 - Broken Authentication and Session Management
CWE-256: Plaintext Storage of a Password

Kurze Beschreibung:

Speichern der Passwörter in Plain-Text kann zu Systemkompromitierung führen.

Erweiterte Beschreibung:

Probleme mit Passwort-Management erscheinen wenn ein Passwort in den App-Eigenschaften oder Konfiguration bzw. in der Datenbank als Plain-Text gespeichert werden.
Jede Person, die den Lesezugriff zu solchen Dateien/Informationen hat bzw. bekommt, kann auch die passwortgeschützen Ressourcen zugreifen.

Zeit der Einfügrung:

Architektur und Design

Modi der Einführung:

Anwender glauben manchmal, dass sie ihre Anwendungen von einer Person nicht schützen können, die den Zugriff zur Konfiguration/Datenbank hat, aber diese Einstellung macht die Arbeit des Angreifers einfacher.

Ort des Fehlers im Code:

Wie man im Listing 3 sehen kann, werden in der Datenbank nur der Username und das Plain-Text-Passwort gespeichert (Zeilen 2 und 3).

Wie kann der Fehler behoben werden?

1. Passwort hashen (z.B. SHA-256)
2. Salt hinzufügen (random-Zeichen pro Benutzer und Passwort z.B. 24 bit lang)
3. „Langsames Hashen“: Passwort-Stretching (z.B. PBKDF2, min. 10000 Iterationen)

3.2.2 Ausnutzen der Schwachstelle

Um die Schwachstelle auszunutzen, gibt es folgende möglichkeiten:

- Die Webseite: http://wlodarski.at/secsyseng-ss2015/05/app2/src-vuln/lab1c.php zu besuchen.
- Das Program (app2/src-vuln/lab1c.php) auf einem beliebigen Server mit PHP auszuführen.

Dann sollen der Username und das Passwort angegeben werden und der Button „Registrieren“ gedrückt werden.

3.2.3 Korrektur der Schwachstelle

Wie man im Listing 4 sehen kann, ist die Schwachstelle aufgehoben worden, indem man in Code zusätzlich randomisierte Salts generiert (Zeile 52) und den PBKDF2 Algorithmus (Zeilen 1-36) mit SHA-256 Hashverfahren (Zeile 39 sowie 16) verwendet.

Das Programm ist nun sicher, da es keine Plain-Text Passwörter mehr gespeichert werden und der Algorithmus für das langsame Hashen (Zeilen 20-30) mit Salt und sicherem SHA-256 gilt heutzutage als eine sichere Methode für das Speichern der Passwörter.

3.2.4 Wo ist diese Schwachstelle aufgetreten?

- Im August 2013 hat sich herausgestellt, dass Google Chrome die Passwörter in Plain-Text Form speichert. Falls eine unbefugte Person den Zugriff zum Computer des Opfers bekommt, dann kann sie alle in Chrome gespeicherten Passwörter problemlos lesen - ohne Master-Passwort und einfach im Einstellungen-Panel. Wenn das nicht genug wäre, hat der Dev-Chef von Chrome gesagt, er wüsste über diese Schwachstelle und es gäbe keine Pläne, das System zu ändern.
  Es gibt noch keine bekannten Auswirkungen und Ausnutzungen von dieser Schwachstelle. Quellen: http://www.theguardian.com/technology/2013/aug/07/google-chrome-password-security-flaw
  https://nakedsecurity.sophos.com/2013/08/08/chrome-firefox-display-plain-text-passwords-with-a-few-clicks/

- Die Uber App hat den Usern die E-Mails mit Plain-Text-Passwörtern geschickt, dank denen die Hackers die Kontos angreifen konnten. Im letzten Monat (Mai 2015) hat Isabelle Berner aus den USA die Rechnungen für Taxi-Fahrten im Großbritannien bekommen, wobei sie gar nicht in Großbritannien war. Ihr Konto wurde gehackt und zwar hat Frau Berner das Passwort geändert, aber das war trotzdem kein Problem für den Hacker, ihr Konto nochmals auszunutzen. Danach hat sie ein E-Mail von Uber-Support bekommen, indem sie ihr neues Passwort - im Plain-Text - bekommen hat. Schließlich hat Frau Berner ihr Geld von Uber zurückbekommen.
  Quellen: http://motherboard.vice.com/read/ubers-response-to-hacked-accounts-is-more-bad-security
  http://www.itpro.co.uk/security/24631/uber-sends-hacking-victim-new-password-in-plain-text-email
  https://nakedsecurity.sophos.com/2015/05/19/uber-in-hot-water-again-over-plaintext-passwords-in-emails/

- Im Juni 2014 gab es 31 964 Super-Mikro Motherboards mit hard-kodierten Plain-Text Passwörtern in ihren Controllers. Dabei hatten 3 296 von diesen die Default-Kombinationen. Es wurde ein Path veröffentlicht, aber es muss im System geflashed werden, was nicht immer eine Möglichkeit ist.
  Es gibt keine bekannten Auswirkungen und Ausnutzungen von dieser Schwachstelle.
  Quelle: http://www.pcworld.com/article/2366020/alert-issued-over-plain-text-passwords-in-some-super-micro-motherboards.html

4 Beispiele

4.1 Source Code formatieren

Es folgen einige Beispiele wie Sourcecode in diesem Dokument formatiert und referenziert werden kann (siehe Listing 5 auf Seite 13 und siehe Listing 6 auf Seite 8).

Ebenso können kurzer Code oder kurze Befehle direkt in der Zeile in einem `lstinline Block` mit typengleicher Schrift formatiert werden.

```bash
#!/bin/bash
echo "Bash version ${BASH_VERSION}..."
for i in {0..10..2}
  do
     echo "Welcome $i times"
 done

echo "some very very very very very very very very very very very very very very very very very very very very long string"

exit 0;
```

Listing 6: Example bash script

4.2 Bilder

Es folgen einige Beispiele wie Bilder in diesem Dokument eingefuegt werden koennen (siehe Abbildung 1 auf Seite 8).

Abbildung 1: ESSE Logo

```
public static void main(String[] args){

    System.out.println(args[0]);
        if(args.length < 1){
            System.err.println("Usage: java -jar SQLInjection <name to add>");
            System.exit(-1);
        }
        String name = args[0];
        Connection conn = null;
        String add_person_to_db = "INSERT INTO person(name) VALUES (";

        try{
            Class.forName("org.h2.Driver");
        } catch (ClassNotFoundException e) {
            System.err.println("Cant' find Driver for H2-Database");
            System.exit(-1);
        }
        try {
            conn = DriverManager.getConnection("jdbc:h2:tcp://localhost/~/app0",
                "admin", "");
        } catch (SQLException e) {
            System.err.println("Cannot create connection from Manager.");
            System.exit(-1);
        }

        if(conn != null){

            try {

                Statement stat = conn.createStatement();
                boolean retval = stat.execute(add_person_to_db+"'"+name+"')");
                System.out.println("Added Person:<"+name+"> to database.");
                stat.close();
            } catch (SQLException e) {
                System.out.println("Invalid input \nUsage: java -jar SQLInjection
                    <name to add>\n"+e.getMessage());
                System.exit(-1);
            }

        }

        if(conn != null){

            try {
                conn.close();
            } catch (SQLException e) {
                System.err.println("Couldn't close DB-connection.");
            }

        }

        System.exit(1);
    }
```

Listing 1: Code mit Schwachstelle

```
1  public static void main(String[] args){
3          if(args.length != 1){
               System.err.println("Usage: java -jar SQLInjection <name to add>");
5              System.exit(-1);
           }
7          String name = args[0];
           Connection conn = null;
9          String add_person_to_db = "INSERT INTO person(name) VALUES (?)";
11         try{
               Class.forName("org.h2.Driver");
13         } catch (ClassNotFoundException e) {
               System.err.println("Cant' find Driver for H2-Database");
15             System.exit(-1);
           }
17         try {
               conn = DriverManager.getConnection("jdbc:h2:tcp://localhost/~/app0", ↩
                   "admin", "");
19         } catch (SQLException e) {
               System.err.println("Cannot create connection from Manager.");
21             System.exit(-1);
           }
23
           if(conn != null){
25
               try {
27
                   PreparedStatement stat = conn.prepareStatement(add_person_to_db);
29                 stat.setString(1,name);
                   boolean retval = stat.execute();
31                 System.out.println("Added Person:<"+name+"> to database.");
                   stat.close();
33             } catch (SQLException e) {
                   e.printStackTrace();
35                 System.exit(-1);
               }
37
           }
39
           if(conn != null){
41
               try {
43                 conn.close();
               } catch (SQLException e) {
45                 System.err.println("Couldn't close DB-connection.");
               }
47
           }
49
           System.exit(1);
51     }
```

Listing 2: korregierter Code

```php
function echoInsert($username, $password) {
    echo "SQL-Statement:<br/>";
    echo "<font style=\"font-family: monospace\"><b>INSERT INTO</b> users(username, ↩
        password)<br/><b>VALUES</b>(<br/>    $username,<br/> ↩
            $password<br/></font>)";
}

if(isset($_POST['submit'])) {
    $username = htmlspecialchars($_POST["user"]);
    $password = htmlspecialchars($_POST["pass"]);
    $cpassword = htmlspecialchars($_POST["cpass"]);

    if ($password != $cpassword) {
        echo "<font color=\"red\"><b>Die beiden Passw\"orter m\"ussen gleich ↩
            sein!</b></font>";
    } else {
        echoInsert($username, $password);
    }
}
```

Listing 3: app2 mit Schwachstelle

```
function pbkdf2($algorithm, $password, $salt, $count, $key_length, $raw_output =
    false) {
    $algorithm = strtolower($algorithm);
    if(!in_array($algorithm, hash_algos(), true))
        trigger_error('PBKDF2 ERROR: Invalid hash algorithm.', E_USER_ERROR);
    if($count <= 0 || $key_length <= 0)
        trigger_error('PBKDF2 ERROR: Invalid parameters.', E_USER_ERROR);

    if (function_exists("hash_pbkdf2")) {
        // The output length is in NIBBLES (4-bits) if $raw_output is false!
        if (!$raw_output) {
            $key_length = $key_length * 2;
        }
        return hash_pbkdf2($algorithm, $password, $salt, $count, $key_length,
            $raw_output);
    }

    $hash_length = strlen(hash($algorithm, "", true));
    $block_count = ceil($key_length / $hash_length);

    $output = "";
    for($i = 1; $i <= $block_count; $i++) {
        // $i encoded as 4 bytes, big endian.
        $last = $salt . pack("N", $i);
        // first iteration
        $last = $xorsum = hash_hmac($algorithm, $last, $password, true);
        // perform the other $count - 1 iterations
        for ($j = 1; $j < $count; $j++) {
            $xorsum ^= ($last = hash_hmac($algorithm, $last, $password, true));
        }
        $output .= $xorsum;
    }

    if($raw_output)
        return substr($output, 0, $key_length);
    else
        return bin2hex(substr($output, 0, $key_length));
}

function echoInsertSecure($username, $salt, $password) {
    $securePassword = pbkdf2(sha256, $password, $salt, 20000, 24);
    echo "SQL-Statement:<br/>";
    echo "<font style=\"font-family: monospace\"><b>INSERT INTO</b> users(username,
        salt, password)<br/><b>VALUES</b>(<br/>
            $username,<br/>     $salt,<br/>
            $securePassword<br/></font>)";
}

if(isset($_POST['submit'])) {
    $username = htmlspecialchars($_POST["user"]);
    $password = htmlspecialchars($_POST["pass"]);
    $cpassword = htmlspecialchars($_POST["cpass"]);

    if ($password != $cpassword) {
        echo "<font color=\"red\"><b>Die beiden Passw\"orter m\"ussen gleich
            sein!</b></font>";
    } else {
        $salt = mcrypt_create_iv(24, MCRYPT_DEV_URANDOM);
        echoInsertSecure($username, $salt, $password);
    }
}
```

Listing 4: app2b korrigiert

```c
/*
 * Just an example C-file.
 */

#include <stdio.h>

int global_variable = 1;
#ifdef DEBUG
int another_global_variable = 1;
#endif

/*
 * Some comment
 */
int main(void)
{
  temp_variable = 4711;
  another_variable = 0815;

  printf("foo bar baz %02d", temp_variable);

  return 1;
}
```

Listing 5: Example C/C++ file