Labreport #3

Patrick Eickhoff, Alexander Timmermann

1 HTTP

1.1

Mit dem Befehl `telnet www.uni-hamburg.de 80` öffnen wir eine Verbindung zu der angegeben Adresse über Port 80 (Http Port). Mit der offenen Verbindung erwartet der Host nun unsere Request. Nach der üblichen Form für HTTP-Anfragen, fragen wir die *home.html* an:

```
GET /de/inst/ab/svs/home.html HTTP/1.1
Host:www.inf.uni-hamburg.de
```

Da wir eine HTTP/1.1 Anfrage stellen, müssen wir den Host angegeben, da HTTP/1.1 multiple Domains erlaubt.

Als Antwort erhalten wir jedoch, dass die gesuchte Seite verschoben wurde und nun unter *https://www.inf.uni-hamburg.de/de/inst/ab/svs/home.html* zu finden ist. Da `telnet` jedoch keine SSL-Verbindungen unterstützt, müssen wir per OpenSSL die HTML anfragen: `openssl s_client -connect www.inf.uni-hamburg.de:443` (Port 443 für ssl)

```
GET /de/isnt/ab/svs/home.html
Host: www.inf.uni-hamburg.de
```

Im Kopf der HTML können wir sehen, dass */assets/application-11e3b49e605ff8ba1f01d275bd36850edfdfc1fbbb8c22e55fae1baf643a00d0.css* der Stylesheet ist, den wir suchen. Da SSL jedoch eine sichere Verbindung ist, haben wir kaum Zeit unsere nächste Anfrage zu stellen:

```
GET /assets/application-11e3b49e605ff8ba1f01d275bd36850edfdfc1fbb
b8c22e55fae1baf643a00d0.css
Host:www.inf.uni-hamburg.de
```

2 SMTP(Mail-Spoofing)

2.1

Mittels `netcat mailhost.informatik.uni-hamburg.de 25` verbinden wir uns mit dem SMTP-Server des Informatikums.

```
HELO mailhost.informatik.uni-hamburg.de
MAIL FROM:<123Mustermann@informatik.uni-hamburg.de>
RCPT TO:<123opfer@informatik.uni-hamburg.de>
DATA
From: <123Mustermann@informatik.uni-hamburg.de>
To: <123opfer@informatik.uni-hamburg.de>
Date: Mon, 10 Apr 2016 10:00:00 -0400
Subject: Prank

Its just a prank.

.

QUIT
```

Wenn man nun den Quelltext unserer Fake-Mail und einer normalen Mail vergleicht sieht man einige Unterschiede:
Zum einem ist die Fake-Mail nicht im MIME-Format, wie normalerweise üblich. Sehr gut lässt sich auch erkennen, dass Nachrichten von authentifizierten Usern des RRZ auch als solche im Quelltext sichtbar sind: `(Authenticated sender: 123Mustermann)`. Dies sind nur einige der Unterschiede zwischen einer echten und unserer gefälschten Mail.

3 DNS-Spoofing

3.1

Nach einiger Interaktion mit dem Lizensserver, fällt uns auf, dass keine Authentifikation zwischen Klient und Server gefordert wird. Ausserdem ist die Bestätigung eienr Lizenz vom Server zum Klienten nur der String `SERIAL_VALID=1`. Dies lässt sich leicht fälschen, wenn wir einfach unseren eigenen Server mittels DNS-Spoofing als Lizenzserver ausgeben.

3.2

Um den Lizenclient auszutricksen, müssen wir zuerst sicherstellen, dass er sich mit unserem eigenem Server anstatt dem Lizenzserver verbindet. Obwohl die *LicenseClient.class* nicht einfach auslesbar ist, können wir mittels `strings LicenseClient.class` herausfinden, dass der Klient immer mit der selben Hostadresse *licenseserver* verbindet. Nun müssen wir nur noch in der *hosts*-Datei folgenden Eintrag hinzufügen: *127.0.1.2 licenseserver*, wobei 127.0.1.2 die IP-Adresse ist, auf der wir unseren eigenen Server laufen lassen.

Unseren Server haben wir in *Ruby* geschrieben
(Source:http://www.tutorialspoint.com/ruby/ruby_socket_programming.htm, sh. Appendix A). Wenn der Server angesprochen wird, tut dieser nichts anderes, als irgendeine Eingabe vom Klienten zu nehmen und mit `SERIAL_VALID=1` zu antworten.

3.3

Um sich vor DNS-Spoofing zu schützen hat der Betreiber mehrere Möglichkkeiten: Zum einem können so simple DNS-Angriffe verhindert werden, wenn statt der */etc/hosts* Datei direkt eine DNS-Query verwendet wird. Dies ist aber auch nur begrenzter Schutz, da auch die DNS-Resolver mit sog. *cache-posioning* manipuiliert werden können. Einen besseren Schutz bieten Methoden wie:

1. Zufällig Groß- und Kleinbuchstaben verwenden, da diese nicht im Cache des Resolvers stehen, aber von Name-Servern beim auflösen der IP-Adresse ignoriert werden.
2. Die Query-ID zufällig setzen, sodass Queries nicht gezielt manipuliert werden können.

Am einfachsten wäre es vermutlich innerhalb der LicenseClient-Datei den Namen mit Groß- und Kleinschreibung zu randomisieren.
Für weitere Informationen siehe Quelle: *http://www.esecurityplanet.com/network-security/how-to-prevent-dns-attacks.html*

4 License-Server(Bruteforce-Angriff)

4.1

Den Bruteforce-Angriff haben wir als Python-Skript geschrieben (sh. Appendix B) basierend auf dem Passwort-Bruteforce. Als problematisch erwies sich jedoch, dass der LicenseServer bei zu vielen Anfragen bzw. Versuchen die Verbindung geschlossen und einen Neuaufbau für ein gewisses Zeitfenster abgelehnt hat. Aus diesem Grund haben wir sobald keine Antwort mehr vom Server zurückkommt, eine Wartezeit von 10s eingeführt, bis wir wieder eine Verbindung zum Server aufbauen. Dies funktioniert zwar, jedoch treibt es auch die Berechnungszeit enorm in die Höhe.

4.2

Eine Möglichkeit für den Betreiber sich gegen Bruteforceangriffe zu schützen, ist einen größeren Zeitabstand zwischen Anfragen zu fordern. Auf diese Weise würde ein Bruteforce-Angriff einen enormen Zeitaufwand haben. Eine weitere Möglichkeit ist, bei hoher Anzahl subsequenter Anfragen von der selben IP-Adresse, diese zu sperren (Blacklisting).

4.3

5 Iplementieren eines TCP-Chats

5.1

Mithilfe unserer *UDPReceiver.java* (siehe Appendix C) können wir die einzelnen UDP-Packete empfangen und auf der Konsole ausgeben. Da die Packete jedoch nur Teile der

beiden URL's enthalten, müssen wir die einzelnen Packete selbst zusammen puzzeln. Die URL's lauten: *http://www.oracle.com/technetwork/java/socket-140484.html* und *https://code.google.com/archive/p/example-of-servlet/*. Die 2. URL wurde anders übermittelt, da jedoch der Project-Hosting-Service von Google Anfang 2016 eingestellt wurde, findet man dies nun im Archiv.

5.2

Nach Zusammenfassen und etwas umschreiben des Tutorials von Oracle, läuft unser Server dann über *localhost* auf dem Port 4444. Wenn wir diesen jetzt über `telnet localhost 4444` ansprechen, können wir per Konsole Daten an den Server übertragen. Ohne das wir die Funktionalität des Servers geändert haben, gibt er dem Klienten jedoch nur seine Eingabe als Ausgabe zurück.

5.3

Die ursprüngliche Implementation von Server und Klient, reicht nicht aus, um einen tatsächlichen Chatraum mehrerer User zu verwirklichen. Jeder der User, deren Verbindung jeweils über einen eigenen Thread verwaltet wird, muss nun mit den anderen Usern kommunizieren können, um Nachrichten auszutauschen. Da Kommunikation zwischen einzelnen unabhängigen Threads nicht einfach zu realisieren ist, soll der Server die Kommunikation verwallten.
Hierzu haben wir ein Beobachtermuster implementiert, sodass der Server alle Threads in einer Liste speichert und die Threads dem Server mitteilen , wenn sie eine Nachricht senden. In der dafür vorgesehen `notify-` Methode, wird dann an alle registrierten Threads der `print`-Befehl weitergereicht.

```
public void notify(String message, int ID, String User) {
        for (ClientThread t : connections) {
                if (t.ID != ID) {
                        t.print(User + ":" + message);
                }
        }
}
```

Ein Thread, der seine Client-Verbindung schließt, wird aus der Liste *connections* entfernt.

5.4

Die *Useradmin.java* aus Aufgabenblatt2 liefert uns eine einfache Möglichkeit eine Authentifikation zu implementieren. Nachdem wir die Klasse importiert haben, können wir auf die `checkUser` Methode zugreifen, um zu prüfen, ob der User in unserer Datenbank vorliegt und das eingegebene Passwort korrekt ist.
Bei Verbindungsaufbau eines Klienten, sendet der Server eine Anfrage nach Username und Passwort. Damit wir das Passwort nicht als String im Speicher stehen haben, lesen wir den Inputstream Byte für Byte aus und konvertieren diese zu Charactern, die wir

einer Liste hinzufügen, bis wir das Terminierungszeichen
`r` lesen. Diese Liste lässt sich dann in ein char-Array umwandeln.

```
BufferedReader in = null;
        try {
                PrintWriter out = new PrintWriter(
                    client.getOutputStream(), true);
                in = new BufferedReader(new InputStreamReader(
                                client.getInputStream()));
                out.println("Username: ");
                String username = in.readLine();
                out.println("Password:");
                List<Character> l = new ArrayList<Character>();
                do {
                        char c = (char) in.read();
                        if (c == '\r') {
                                break;
                        }
                        l.add(c);
                } while (true);
                char[] password = new char[l.size()];
                int i = 0;
                for (Character c : l) {
                        password[i] = c.charValue();
                        i++;
                }
                l = null;
```

Der Server prüft intern, ob die User-Passwort-Kombination gültig ist. Ist sie es nicht, wird sofort die Verbindung geschlossen. Wenn der User sich erfolgreich authentifiziert, trägt der Server den User in die Liste eingeloggter User ein. So wird verhindert, dass sich verschiedene Personen mit dem selben Benutzer anmelden.