Labreport 02

Tronje Krabbe, Julian Deinert

25. Mai 2016

Inhaltsverzeichnis

Aufgabe 1 HTTP

1.1 Telnet

Wir haben versucht uns mit dem Befehl `telnet` mit dem angegebenen Server zu verbinden.

```
$ telnet www.inf.uni-hamburg.de 80
Trying 134.100.56.130...
Connected to www.inf.uni-hamburg.de.
Escape character is '^]'.

GET /de/inst/ab/svs/home.html HTTP/1.1
```

Als Antwort auf unseren GET-Request erhalten wir eine Website mit Returncode `302 Found`, die uns sagt, dass das Dokument nur mittels *https* erreichbar ist. Da Telnet kein https kann, greifen wir auf openssl zurück.

```
$ openssl s_client -connect www.inf.uni-hamburg.de:443

GET /de/inst/ab/svs/home.html HTTP/1.1
```

Wir erhalten den HTTP-Fehlercode `400 Bad Request` zurück. Dementsprechend können wir auch keine CSS-Dateien anfordern.

Aufgabe 2 SMTP

2.1 Mail Spoofing

Wir verbinden uns mittels *Netcat* mit dem Mailserver `mailhost.informatik.uni-hamburg.de` auf Port 25. Nach dem wir die Felder `FROM`, `RCPT TO` sowie `DATA` gesetzt haben wird unsere Mail erfolgreich versendet. Der Empfänger kann anhand des Quelltextes erkennen, dass die Mail nicht von einem *Authenticated sender* geschickt wurde.
Es gibt keinen Unterschied ziwschen einer gespooften gmail oder informatik.uni-hamnburg Adresse, da beide Mails direkt beim Mailhost eingereicht wurden.

Aufgabe 3 License Server

3.1 DNS Spoofing

Wir haben uns mit *netcat* zum Server verbunden und werden aufgefordert einen von 4 validen Commands einzugeben. Diese sind:

- help
- serial
- version
- quit

Wenn wir eine Serial angeben, die zufällig keine gültige ist, bekommen wir die Meldung `SERIAL_VALID=0` zurück. Anhand dieser Information erstellen wir unseren TCP server so, dass bei jeder Serial die Meldung `SERIAL_VALID=1` zurückgegeben wird.

3.2 Eigener License Server

```
#!/usr/bin/env python
import socketserver as ss

class LeetTCPHandler(ss.BaseRequestHandler):
    def handle(self):
        self.data = self.request.recv(1024).strip()
        print(self.data)
        if self.data == b'VERSION':
            self.request.sendall(b"Numeric Serial Server Validation System 2.1a")
        elif self.data[:6] == b'SERIAL':
            self.request.sendall(b"SERIAL_VALID=1")
        else:
            self.request.sendall(b"Invalid command")


if __name__ == "__main__":
    HOST, PORT = "localhost", 1337

    server = ss.TCPServer((HOST, PORT), LeetTCPHandler)

    server.serve_forever()
```

Außerdem ergänzen wir unser Host-File um diese Zeile:
`127.0.0.1 license-server.svslab localhost`
Nun können wir das Java-Programm ausführen, und es verbindet sich mit unserem eigenen TCP-Server, und akzeptiert eine beliebige Zahl als License-Key:

```
Your license has expired - please enter new license key!
To order your new key for just 999.99$ call +1-555-we-rule

Key: 123
Numeric Serial Server Validation Sy
Thanks for purchasing a new license!
```

3.3 Verhinderung des Angriffs

Eine naive Möglichkeit, den Spoof zu entdecken, wäre einfach, ein weniger offensichtiliches Schlüsselwort als Ausgabe nach erfolgreicher Überprüfung zu wählen. Momentan gibt der Server *SERIAL_ VALID=0* zurück, wenn der Key falsch ist. Daraus ist leicht zu folgern, dass *SERIAL_ VALID=1* die Ausgabe ist, die bei einem richtigen Key gegeben wird. Dies könnte man ändern um den Angriff zu erschweren.

Eine sicherere Methode wäre, Public-Key-Cryptographie zu nutzen. Das Programm wird mit dem public-key des Servers sowie seinem eigenen private-key kompiliert. Der Server kennt den public-key des Programms. Die Kommunikation geschieht so, dass alles, was das Programm an den Server sendet, mit seinem public-key verschlüsselt ist. Der Server entschlüsselt die erhaltenen Daten mit seinem private-key, und verschlüsselt seine Ausgabe mit dem public-key des Programms.

Auch hier liegt leider eine Schwierigkeit; hat jede Kopie des Programms das gleiche Key-Paar? Wenn nicht, woher weiß der Server, welchen Key er benutzen muss? Außerdem könnte man den Speicher des Programms untersuchen, während es läuft, oder einen Decompiler benutzen, um die Keys zu finden.

4 Aufgabe 4 License-Server

4.1

Unser Programm ist im Anhang zu finden.

4.2

Es gibt mehrere Möglichkeiten, so einen Service gegen Brute-Force Angriffe zu schützen. Von Anfang an ist es eine gute Idee, Authentifizierungsversuche zeitlich zu begrenzen. Wenn nur eine Validierung pro Sekunde zugelassen wird, merkt ein User mit einem validen Key praktisch keinen Unterschied, ein Brute-Force Angriff hingegen dauert dann deutlich länger. Weiterhin kann man IPs, die eine gewissen Anzahl Fehlversuche hinter sich haben, weiter verlangsamt bedienen, oder gar ganz blockieren.

4.3

Wir haben folgende Zahlen haben wir rausgefunden: 90877300, 31337000, 21935900, 62674000, 18802200. Wenn man den größten gemeinsamen Teiler dieser Zahlen berechnet, erhält man 3133700. Und tatsächlich akzeptiert der Server alle Zahlen, die nach der Formel $(n \cdot 3133700) \mod 10^8$ generiert wurden.

5 Aufgabe 5 Implementieren eines TCP-Chats

5.1

Die vom Server ausgestrahlten URLs sind `http://www.oracle.com/technetwork/java/socket-140484.html` und `https://code.google.com/archive/p/example-of-servlet/source/default/source`.

5.2

Wir kompilieren die Quelltext-Dateien `ClientWorker.java`, `SocketThrdServer.java` und `SocketClient.java`, um den Server und die Client ausführen zu können. Wenn wir in den Quelltext schauen, finden wir raus, dass der Server auf Port 4444 servt. Wenn wir ihn mit `telnet` ansprechen, wird unser Input immer noch einmal zurückgeschickt, und im GUI des Servers angezeigt. Da wir die gethreadete Variante nehmen, können wir auch gleichzeitig mit dem `SocketClient` Programm 'chatten'.