

Laborübung:

Einstieg in die Socket Programmierung

1. Einleitung

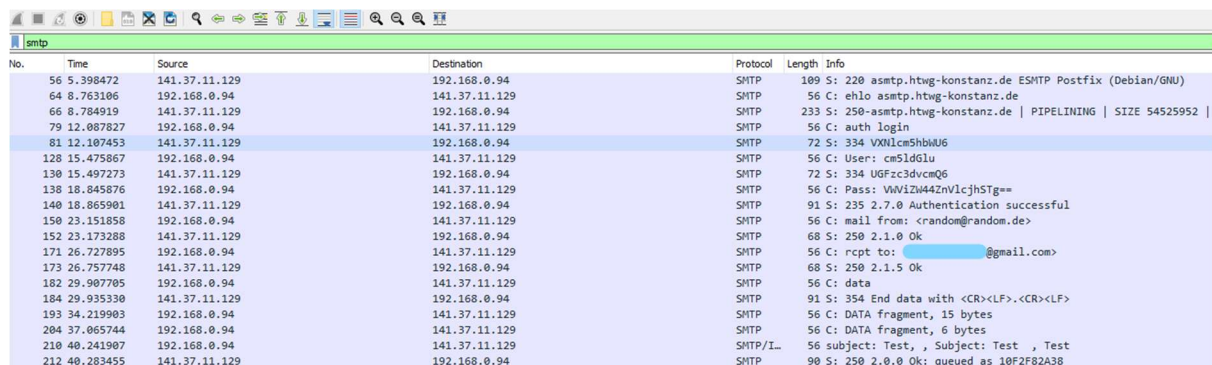
➔ Einleitung

2. Vorbereitung

➔ Einleitung

3. Mail

3.1 SMTP über telnet



No.	Time	Source	Destination	Protocol	Length	Info
56	5.398472	141.37.11.129	192.168.0.94	SMTP	109	S: 220 asmtg.htwg-konstanz.de ESMTD Postfix (Debian/GNU)
64	8.763186	192.168.0.94	141.37.11.129	SMTP	56	C: ehlo asmtg.htwg-konstanz.de
66	8.784919	141.37.11.129	192.168.0.94	SMTP	233	S: 250-asmtg.htwg-konstanz.de PIPELINING SIZE 54525952
79	12.087827	192.168.0.94	141.37.11.129	SMTP	56	C: auth login
81	12.107453	141.37.11.129	192.168.0.94	SMTP	72	S: 334 VNWlcmShbWU6
128	15.475867	192.168.0.94	141.37.11.129	SMTP	56	C: User: cm5ldGlu
130	15.497273	141.37.11.129	192.168.0.94	SMTP	72	S: 334 UGFzc3dvcmQ6
138	18.845876	192.168.0.94	141.37.11.129	SMTP	56	C: Pass: VNWlcmShbWU6
140	18.865901	141.37.11.129	192.168.0.94	SMTP	91	S: 235 2.7.0 Authentication successful
150	23.151858	192.168.0.94	141.37.11.129	SMTP	56	C: mail from: <random@random.de>
152	23.173288	141.37.11.129	192.168.0.94	SMTP	68	S: 250 2.1.0 Ok
171	26.727895	192.168.0.94	141.37.11.129	SMTP	56	C: rcpt to: *****@gmail.com>
173	26.757748	141.37.11.129	192.168.0.94	SMTP	68	S: 250 2.1.5 Ok
182	29.907705	192.168.0.94	141.37.11.129	SMTP	56	C: data
184	29.935330	141.37.11.129	192.168.0.94	SMTP	91	S: 354 End data with <CR><LF>.<CR><LF>
193	34.219903	192.168.0.94	141.37.11.129	SMTP	56	C: DATA fragment, 15 bytes
204	37.065744	192.168.0.94	141.37.11.129	SMTP	56	C: DATA fragment, 6 bytes
210	40.241907	192.168.0.94	141.37.11.129	SMTP/I	56	subject: Test, Subject: Test, Test
212	40.283455	141.37.11.129	192.168.0.94	SMTP	90	S: 250 2.0.0 Ok: queued as 10F2F82A38

Der oben zu sehende Screenshot zeigt einen WireShark Trace (Filter auf „smtp“) bei welchem die folgenden Schritte ausgeführt wurden, um eine Mail mittels SMTP Protokoll zu versenden:

1. Starten von „telnet“ in einer cmd-Line
2. Verbindungsaufbau zum Mail-Server mittels:
open asmtg.htwg-konstanz.de 587
3. Senden des „ehlo“ Kommand (siehe No. 64 & 66) mittels:
ehlo asmtg.htwg-konstanz.de
4. Authentifizierung des Login (No. 79 & 81) mittels:
auth login
5. Übermitteln des Nutzer als base64-kodierte Zeichenketten mittels:
cm5ldGlu
6. Übermitteln des Passwort als base64-kodierte Zeichenketten mittels:
VNWlcmShbWU6
7. Übermitteln des „Mail-From“ Feld mittels:
mail from: <rnetin@htwg-konstanz.de>
8. Übermitteln des „Mail-To“ (rcpt) Field mittels:
rcpt to: <*****@gmail.com>
9. Ankündigung der Datenübermittlung mittels:
data
10. Übermitteln des Betreff mittels:
Subject: Test
11. Übermitteln der Nachrichtendaten:
Test
12. Beenden der Übertragung mittels:
Carriage Return + Line Feed
.
Carriage Return + Line Feed

➔ Siehe Folgeseite

Betrachten Sie die Aufzeichnung in WireShark. Was fällt Ihnen auf?

Alle ausgeführten Schritte finden sich in dem WireShark Trace. Da die Daten unverschlüsselt zum SMTP Server übertragen werden sind alle Daten (auch Passwörter) als Klartext lesbar (zwar kodiert lassen sich jedoch leicht decoden)

Schreiben Sie noch eine Mail an ihren Email-Account. Verwenden Sie willkürliche Email-Adressen für MAIL-FROM sowie für das „from:“-Feld in der Mail. Siehe auch Wikipedia Beispiel. Lesen Sie die Mail in ihrem Postfach. Was fällt Ihnen auf?

Es ist mittels des SMTP Server möglich Emails auch von anderen Absendern zu versenden (Änderung des *local-part*, vor @ möglich) jedoch scheint der Server den *domain-part* der Mailadresse (nach dem @) zu prüfen. So ist es möglich eine Email von angela.merkel@rnetin.de zu senden jedoch nicht von angela.merkel@bundesregierung.de.

3.2 SMTP in Python

Hier wurden die oben händisch ausgeführten Schritte durch ein Python Skript automatisiert ausgeführt (das Resultat bleibt das Gleiche).

Python Code siehe: smtp_client.py im zugehörigen Unterordner

4. Rechner Server

4.1 Lokale Kommunikation

1. für jedes gesendete Paket bestimmen, welcher Befehl in welchem Skript (Client/Server) dafür verantwortlich ist, dass das Paket gesendet wird.

Client	Server	
	Wartet auf Client	<code>sock.listen(1)</code>
Verbindung zu Server aufbauen		<code>sock.connect((Server_IP, Server_PORT))</code>
	Verbindung akzeptieren	<code>conn, addr = sock.accept()</code>
	Warten auf Daten [BLK 1]	<code>data = conn.recv(1024)</code>
Senden von Daten [FREE 1]		<code>sock.send(MESSAGE.encode('utf-8'))</code>
Warten auf Server OK [BLK 2]		<code>msg=sock.recv(1024).decode('utf-8')</code>
	Senden von OK [FREE 2]	<code>conn.send(data[:-1])</code>
	Schließen TCP Verbindung	<code>conn.close()</code>

2. für jeden blockierenden Befehl bestimmen, die Ankunft welches Pakets dafür verantwortlich ist, dass die Ausführung des Befehls vervollständigt wird.

→ Siehe oben

4.2 Netzwerk-Kommunikation

1. Wie können Sie im Client Python-Skript die IP-Adresse und Port-Nummer des verwendeten lokalen Sockets bestimmen („bestimmen“ im Sinne von herausfinden)?

```
sock.getsockname()
```

```
4  Server_IP = '127.0.0.1'
5  Server_PORT = 50000
6  MESSAGE = 'Hello, World!'
7
8  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
9  #print("Local socket name is", sock.getsockname())
10 sock.settimeout(10)
11 print('Connecting to TCP server with IP ', Server_IP, ' on Port ', Server_PORT)
12 sock.connect((Server_IP, Server_PORT))
13
14 print("Local socket name is", sock.getsockname())
15
16 print('Sending message', MESSAGE)
17 sock.send(MESSAGE.encode('utf-8'))
18 try:
19     msg=sock.recv(1024).decode('utf-8')
20     print('Message received; ', msg)
21 except socket.timeout:
22     print('Socket timed out at',time.asctime())
23 sock.close()
```

2. Wann (in welcher Code-Zeile) und woher erhält ein Client seine IP-Adresse und Port-Nummer?

In Zeile nachdem sich der Client mit dem Server verbunden wurde???. Die abfrage in Zeile 9 führt zu einem Fehler.

3. Wie können Sie im Client-Skript die IP-Adresse und Port-Nummer des Sockets setzen?

Mittels:

```
sock.bind(('127.0.0.2', 50001))
```

Der Server zeigt die Änderung wie folgt:

```
Listening on Port 50000 for incoming TCP connections
Listening ...
Incoming connection accepted: ('127.0.0.2', 50001)
received message: Hello, World! from ('127.0.0.2', 50001)
Connection closed from other side
Closing ...
```

Die Verbindung über eine IP aus einem andern Netz ist nicht möglich:

```
sock.bind(('179.0.0.2', 50001))
```

4. Warum müssen Sie Timeouts verwenden und wie funktioniert try ... except? Mit welchem Befehl können Sie einen gemeinsamen Timeout für alle Sockets setzen?

Der Timeout wird genutzt damit ein Client nicht endlos auf eine Antwort vom Server wartet sondern nach einer gewissen Zeit (engl.: Timeout, tada) terminiert. Dies wird durch den try, catch Block realisiert. Das Programm/Thread versucht solange eine Antwort zu erhalten bis der socket eine Timeout Exception wirft. Diese wird von except „gefangen“ und die Instruktion entsprechen ausgeführt.

5. Finden Sie experimentell heraus, ob Sie einen Server betreiben können, der ECHO-Anfragen auf dem gleichen Port für UDP und TCP beantwortet?

Dies ist möglich wenn ein Server mit UDP & TCP Socket erstellt wird welche in unterschiedlichen Threads auf anfragen wartet. Dies wurde mittels den Python Skripten in Order 4/2/5 getestet.

4.3 Unterstützung mehrere Clients

➔ Siehe Python Skripte im zugehörigen Unterordner

5. Port Scanner

5.2 Versuch

Hierzu wurde erneut ein Pythonskript genutzt um mittels Threading und den oben beschriebenen Verfahren die offenen Ports zu ermitteln. Das Skript ist unten aufgeführt und im zugehörigen Unterordner zu finden.

```
import socket
from threading import Thread
import time

Server_IP = '141.37.168.26'
MESSAGE = 'hello'

def scan_port(port):

    tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    tcp_client.settimeout(10)

    try:
        print('Connecting to TCP server with IP ', Server_IP, ' on Port ', port)
        tcp_client.connect((Server_IP, port))
    except socket.timeout:
        print("Connection timed out at",time.asctime(), "on port", port)
    except ConnectionRefusedError:
        print('Connection refused [WinEr: 10061] at',time.asctime(), 'on port', port)
    except socket.error:
        print('Socket error at',time.asctime(), 'on port', port)
    finally:
        tcp_client.close()

if __name__ == '__main__':
    for i in range(1, 51):
        t=Thread(target=scan_port, args=(i,))
        t.start()

    #scan_port(7)
```

5.3 Fragen

1. Geben Sie die Liste der offenen TCP und UDP Ports an.

Derzeit sind alle im Bereich liegenden Ports geschlossen bzw. eine Verbindung wird durch den Server explizit zurückgewiesen (Port 22 & 23 siehe WireShark Trace bzw. Bild)

The image displays a Wireshark packet capture interface. The top pane shows a list of network packets. Packet 64 is highlighted, showing a TCP segment from 141.37.168.26 to 20.111.1.3 on port 22. The packet details pane on the left shows the structure of this packet: Ethernet II, Internet Protocol Version 4, and Transmission Control Protocol. The TCP section is expanded, showing a Reset (RST) flag and a sequence number of 65530. The packet bytes pane on the right shows the raw data of the packet, including the Ethernet header and the TCP segment.

```
> Frame 57: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on Interface \Device\NPF_{2B4B1594-3EB8-49DC-B8DF-C0A30599AE96},
> Ethernet II, Src: 06:9e:20:00:02:01 (06:9e:20:00:02:01), Dst: 03:00:04:00:00:00 (03:00:04:00:00:00)
> Internet Protocol Version 4, Src: 141.37.168.26, Dst: 141.37.206.29
> Transmission Control Protocol, Src Port: 22, Dst Port: 65530, Seq: 1, Ack: 1, Len: 0
  Source Port: 22
  Destination Port: 65530
  [Stream index: 23]
  [Conversation completeness: Incomplete (37)]
  [TCP Segment Len: 0]
  Sequence Number: 1 (relative sequence number)
  Sequence Number (raw): 65344011
  [Next Sequence Number: 1 (relative sequence number)]
  Acknowledgment Number: 1 (relative ack number)
  Acknowledgment number (raw): 2659932488
  0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x014 (RST, ACK)
  Window: 65280
  [Calculated window size: 65280]
  [Window size scaling factor: -1 (unknown)]
  Checksum: 0xc46b [unverified]
  [Checksum Status: Unverified]
  Urgent Pointer: 0
  > [Timestamps]
  > [SEQ/ACK analysis]
```

2. Wählen Sie für TCP und UDP jeweils einen offenen und einen geschlossenen Port und erklären Sie die entsprechende Paketsequenz, die Sie in WireShark aufgezeichnet haben.

➔ Derzeit nicht möglich

3. Auf Port 7 des Servers läuft ein ECHO-Dienst. Testen Sie ihr Client-Script mit dem ECHO-Server. Versuchen Sie das TCP und das UDP Script.

➔ Derzeit nicht möglich