Lab: Exploring Application-Layer Protocols

In this lab, you will explore several key application-layer protocols by interacting with real servers and analyzing packet traces. You will use Python scripts built with scapy and raw sockets to understand how DNS, HTTP, SMTP and POP3 work at the protocol level.

Prerequisites

- Python 3.6+ with a virtual environment (`venv`)
- Scapy (`pip install scapy`)
- Root privileges (`sudo`) for scapy scripts
- Wireshark installed for packet analysis (see https://www.wireshark.org)
- The scripts from this repository

> **Note**
>
> All scapy scripts require root privileges because they craft raw packets. Use `sudo` when running them.

Introduction to Scapy

Before diving into application-layer protocols, you will get familiar with scapy by sending and receiving a simple UDP packet. Scapy is a Python library that lets you build, send, and analyze network packets at a very low level — you control every field in every layer.

Explore the UDP echo script

Read the script `udp/udp_echo.py` and the tutorial `udp/README.rst`. They introduce the key scapy concepts you will use throughout this lab:

- **Layer stacking** with the `/` operator: `IPv6() / UDP() / Raw()`
- **Packet inspection** with `show()` and `show2()`
- **Sending and receiving** with `sr1()`

Pay attention to how the script:

- Resolves a hostname to an IPv6 or IPv4 address
- Builds a UDP packet destined for the echo service (port 7)
- Sends it and waits for the server to echo back the same data

Send a UDP echo request

Run the script against the echo server:

```
sudo python3 udp/udp_echo.py --server echo.computer-networking.info --message "Hello, UDP!"
```

Observe the output:

- What layers does the packet contain?
- What source port was chosen?

- Does the echoed data match what you sent?

Try different options:

- Change the message
- Force IPv4 with `--ipv4-only` — how does the packet differ?
- Capture the exchange with `--tcpdump echo.pcap` and open the file in Wireshark — can you identify the request and response?

> Hint
>
> In Wireshark, use the display filter `udp.port == 7` to show only echo traffic.

DNS

The Domain Name System (DNS) maps human-readable domain names to IP addresses. In this part, you will first build DNS queries by hand using scapy, then observe how iterative resolution traverses the DNS hierarchy from the root servers down to the authoritative servers.

Root DNS Servers

There are 13 root server identities, each operated by a different organization. They are the starting point for iterative DNS resolution.

Server	IPv4	IPv6	Operator
a.root-servers.net	198.41.0.4	2001:503:ba3e::2:30	Verisign
b.root-servers.net	170.247.170.2	2801:1b8:10::b	USC-ISI
c.root-servers.net	192.33.4.12	2001:500:2::c	Cogent
d.root-servers.net	199.7.91.13	2001:500:2d::d	U. of Maryland
e.root-servers.net	192.203.230.10	2001:500:a8::e	NASA
f.root-servers.net	192.5.5.241	2001:500:2f::f	ISC
g.root-servers.net	192.112.36.4	2001:500:12::d0d	US DoD NIC
h.root-servers.net	198.97.190.53	2001:500:1::53	US Army
i.root-servers.net	192.36.148.17	2001:7fe::53	Netnod (Sweden)
j.root-servers.net	192.58.128.30	2001:503:c27::2:30	Verisign
k.root-servers.net	193.0.14.129	2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42	2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33	2001:dc3::35	WIDE (Japan)

Completing the DNS query script

The script `dns/dns_query.py` sends DNS queries to a recursive resolver using scapy. Two parts of the code have been removed and must be completed before the script can run:

1. **Transaction ID**: The variable `transaction_id` must be set to a suitable 16-bit value. Look for the `FIXME` marker in the `send_dns_query()` function. Think about what properties this value should have and why.

2. **Question section**: The `qd` parameter of the `DNS()` constructor is currently set to `None`. You need to construct a `DNSQR` object that specifies which domain name to resolve and which record type to request.

> Hint
>
> Read the docstring at the top of the script carefully — it documents the DNS message format, header fields, and common query types. The scapy `DNSQR` class takes `qname`, `qtype`, and `qclass` parameters.

Once your script works, use it to query the following domain names:

- `google.be`
- `google.tv`
- `blog.computer-networking.info`
- `4ed.computer-networking.info`

For each domain, observe:

- What A (IPv4) and AAAA (IPv6) records are returned?
- Do all domains have both A and AAAA records?
- What is the TTL of each answer?

Iterative DNS resolution

The script `dns/dns_iterative.py` performs iterative resolution: it starts from a root server and manually follows the delegation chain, just like a recursive resolver would internally.

Use it to resolve the following domain names and observe the resolution path:

- `www.whitehouse.gov`
- `github.com`

For each domain, determine:

- How many steps does the resolution take?
- Which root server, TLD server, and authoritative server are contacted?
- Is the domain reachable over IPv4 only, IPv6 only, or both? Use `--qtype A` and `--qtype AAAA` to check.

> Hint
>
> ```
> sudo python3 dns/dns_iterative.py --domain www.whitehouse.gov --qtype A
> sudo python3 dns/dns_iterative.py --domain www.whitehouse.gov --qtype AAAA
> ```
>
> Try the `--start-server` option to start from a specific root server from the table above.

Packet capture and analysis

Capture the DNS packets exchanged during an iterative resolution and analyze them in Wireshark.

Capturing packets:

Use the `--tcpdump` option to save a packet capture:

```
sudo python3 dns/dns_iterative.py --domain github.com --tcpdump dns-github.pcap
```

Then open the capture in Wireshark:

```
wireshark dns-github.pcap
```

Useful Wireshark display filters:

Filter	Description
`dns`	Show only DNS packets
`udp.port == 53`	Show all traffic on the DNS port
`dns.qry.name == "github.com"`	Show queries for a specific domain name
`dns.qry.type == 1`	Show only A record queries (type 1)
`dns.qry.type == 28`	Show only AAAA record queries (type 28)
`dns.flags.response == 0`	Show only queries (not responses)
`dns.flags.response == 1`	Show only responses (not queries)
`dns.flags.rcode == 0`	Show only successful responses (no error)
`dns.flags.authoritative == 1`	Show only authoritative answers

Analyze the trace:

- Identify each query/response pair in the delegation chain
- For each step, note which server was queried and what referral was returned
- Look at the Authority and Additional sections in referral responses — what role do glue records play?
- Compare the number of packets exchanged for `www.whitehouse.gov` versus `github.com` — can you explain the difference?

HTTP

HTTP (HyperText Transfer Protocol) is the foundation of the World Wide Web. Like SMTP and POP3, it is a text-based protocol running over TCP. The client sends a request, and the server sends back a response.

The current HTTP/1.1 specification is split across two RFCs: RFC 9110 (semantics: methods, headers, status codes) and RFC 9112 (message syntax: how requests and responses are formatted on the wire).

The most relevant sections are Section 3 of RFC 9112 (request line format), Section 7.2 of RFC 9110 (Host header), and Section 15 of RFC 9110 (status codes).

An HTTP/1.1 request looks like this:

```
GET /path HTTP/1.1\r\n
Host: hostname\r\n
Connection: close\r\n
\r\n
```

Each line ends with CRLF (`\r\n`). The empty line after the headers signals the end of the request.

Complete the HTTP client

The script `http/http_client_v1_student.py` implements a minimal HTTP/1.1 client using raw TCP sockets. The URL parsing, TCP connection, response receiving and response parsing are already implemented.

Your task is to complete the `send_request()` method. Look for the `FIXME` marker — you need to build a valid HTTP/1.1 GET request string containing:

1. A request line with the method, path, and HTTP version
2. A `Host` header with the server hostname
3. A `Connection: close` header
4. An empty line to signal the end of headers

> Hint
>
> Read the docstring of `send_request()` carefully — it contains an example of a valid HTTP/1.1 request. Remember that each line must end with `\r\n`.

Test your client on the test server:

```
python3 http/http_client_v1_student.py http://test.computer-networking.info:8080/
```

You should receive a `200 OK` response with an HTML body.

HTTP redirections

Now try fetching a different URL with your client:

```
python3 http/http_client_v1_student.py http://test.computer-networking.info:8080/old
```

This time, the server does **not** return a `200 OK`. Carefully examine the output:

- What status code does the server return?
- Look at the response headers — which header tells you where the resource has moved?

Read Section 15.4 of RFC 9110 to understand what this status code means and how a client should handle it.

Your task: modify your client so that when it receives a redirection response, it automatically fetches the URL indicated in the response header. Test it again on the same URL — your client should now follow the redirection and display the final page.

> Hint
>
> The `get()` method is a good place to add this logic. After parsing the response, check the status code. If it indicates a redirect, extract the target URL from the headers and make a new request. Be careful with relative paths (starting with `/`) versus absolute URLs.

HTTP cookies

Some web pages require authentication. The server uses **cookies** to remember that a client has already logged in. Cookies are defined in RFC 6265 — see in particular Section 4.1 (`Set-Cookie` header sent by the server) and Section 4.2 (`Cookie` header sent by the client).

Try fetching the following URL with your client:

```
python3 http/http_client_v1_student.py http://test.computer-networking.info:8080/private
```

You should be denied access. Now fetch this URL:

```
python3 http/http_client_v1_student.py http://test.computer-networking.info:8080/login
```

Examine the response headers carefully:

- What header does the server use to give you a cookie?
- What is the name and value of the cookie?

Your task: modify your client so that it can:

1. Store cookies received from the server
2. Send stored cookies back in subsequent requests

Then use your updated client to first fetch `/login` (to obtain the cookie), then fetch `/private` (sending the cookie back). You should now be able to access the private page.

> ### Hint
>
> When the server sets a cookie, it sends a `Set-Cookie` header in the response. When the client sends a cookie back, it uses a `Cookie` header in the request. The simplest approach is to keep a dictionary of cookies and add them to every request to the same host.

SMTP

SMTP (Simple Mail Transfer Protocol, RFC 5321) is the standard protocol for sending email between mail servers. It is a text-based protocol that runs over TCP. Each command and response is terminated by CRLF (`\r\n`).

The most relevant sections of RFC 5321 are Section 4.1.1 (commands) and Section 4.2 (reply codes).

A typical SMTP session consists of:

1. Client connects, server sends a `220` greeting
2. Client sends `HELO` (or `EHLO`) to identify itself
3. Client sends `MAIL FROM:<sender>` to specify the envelope sender
4. Client sends `RCPT TO:<recipient>` to specify the envelope recipient
5. Client sends `DATA`, then the message body, terminated by a lone `.`
6. Client sends `QUIT` to close the session

SMTP response codes: `2xx` = success, `3xx` = intermediate (e.g. 354 ready for data), `4xx` = temporary failure, `5xx` = permanent failure.

Start the local SMTP server

A simple SMTP server is provided in `smtp/server-smtp.py`. It uses the `aiosmtpd` library and logs all received emails to the terminal.

Start it in a separate terminal:

```
cd smtp
python3 server-smtp.py
```

The server listens on `localhost:8025` by default. Keep it running for the exercises below.

Understand the helper functions

Before writing any SMTP code, read the two helper functions defined at the top of `smtp/smtp_client_student.py`:

- `read_line(sock)` — reads one CRLF-terminated line from the socket and returns it as a string. How does it detect the end of a line?
- `send_cmd(sock, cmd)` — sends a command string to the server with the proper CRLF termination. What encoding does it use and why?

Make sure you understand how these functions work — you will use them in the next exercise.

Complete the SMTP client

The script `smtp/smtp_client_student.py` implements a minimal SMTP client. The first three steps (TCP connection, server greeting, and HELO command) are already implemented. **Your task is to complete steps 4 through 8** by following the comments in the code.

For each step, the comments describe:

- What SMTP command to send
- What format the command uses
- What response code to expect

Look at how steps 1–3 are implemented and follow the same pattern for the remaining steps. Use the `send_cmd()` and `read_line()` functions.

Test your client by sending a simple message:

```
python3 smtp_client_student.py alice@example.com bob@example.com "Hello Bob!"
```

Check the server terminal — the email should appear there.

Test with a sample email file from the `examples/` directory:

```
python3 smtp_client_student.py alice@example.com bob@example.com \
    "$(cat examples/simple.txt)"
```

The `examples/` directory contains several email files with valid headers:

- `examples/minimal.txt` — only essential headers (From, To, Subject, Date)
- `examples/simple.txt` — standard headers with MIME-Version
- `examples/with-cc.txt` — email with Cc recipients
- `examples/with-reply-to.txt` — email with Reply-To header
- `examples/multiline.txt` — longer email with multiple paragraphs

Try sending different example files and observe what the server receives. Pay attention to the difference between the **envelope addresses** (MAIL FROM / RCPT TO) and the **header addresses** (From: / To: in the message) — they do not have to match.

POP3

POP3 (Post Office Protocol version 3, RFC 1939) is the standard protocol for retrieving email from a mail server. Like SMTP, it is a text-based protocol running over TCP.

POP3 responses begin with `+OK` (success) or `-ERR` (failure). Some commands (such as `RETR` and `LIST`) return multi-line responses that are terminated by a line containing only a single dot (`.`).

The most relevant sections of RFC 1939 are Section 4 (AUTHORIZATION state: USER, PASS), Section 5 (TRANSACTION state: STAT, LIST, RETR, DELE), and Section 6 (UPDATE state: QUIT).

Start the local POP3 server

A POP3 server is provided in `pop/pop-server.py`. It uses the Twisted framework and comes pre-populated with three sample email messages.

Start it in a separate terminal:

```
cd pop
python3 pop-server.py
```

The server listens on `localhost:1100` (a non-privileged port, so no `sudo` is needed). Default credentials:

- Username: `student`
- Password: `password`

Understand the helper functions

The script `pop/pop-client-student.py` uses the same `read_line()` and `send_cmd()` helper functions as the SMTP client. Read them again and note how POP3, like SMTP, uses CRLF-terminated lines over TCP.

Complete the POP3 client

The script `pop/pop-client-student.py` connects to the POP3 server and sends the `USER` command. **Your task is to complete** the remaining steps:

1. **Authentication**: send the `PASS` command to complete the login. Use `getpass.getpass()` to prompt the user for their password.
2. **Mailbox exploration**: once authenticated, use POP3 commands to discover and read the messages in the mailbox:
   - How many messages are in the mailbox and what is the total size?
   - What is the size of each individual message?
   - Retrieve and display each message.

   Remember that some POP3 responses are multi-line — they end with a lone `.` on a line by itself. You will need a loop to read all lines until you encounter this terminator.
3. **Close the session** properly.

> Hint
>
> The docstring at the top of the script lists all POP3 commands with their corresponding RFC sections. Read Section 5 of RFC 1939 for the exact syntax and expected responses.

Test your client:

```
python3 pop-client-student.py localhost student -p 1100
```

You should be able to see all three pre-populated messages.