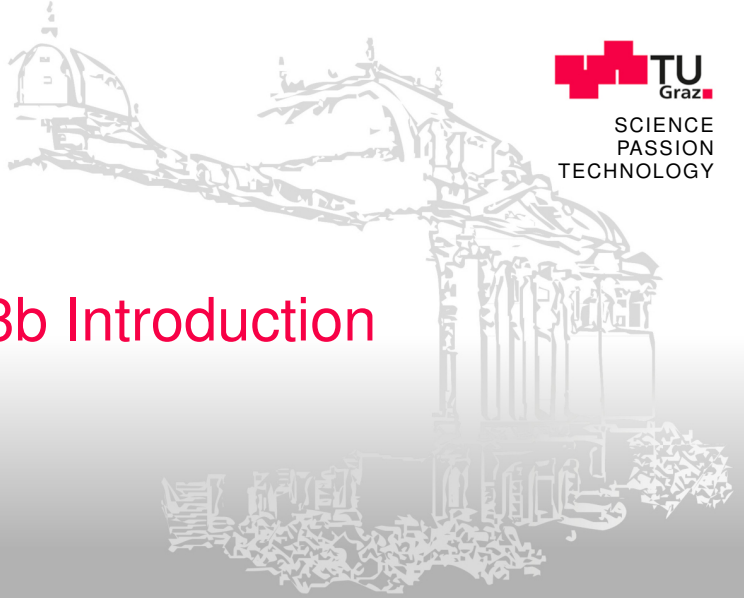
TU Graz

SCIENCE
PASSION
TECHNOLOGY



CON — Task 3b Introduction

Matthias Fischer, Constantin Piber

Your Task

- Receive HTTP/1.1 requests from clients
- Answer requests:
  - Reject invalid packets
  - Parse the request and send the requested file
- Test with browser: `http://localhost:8000`

For this, edit the file `http.cpp`. The Function `handle_connection` is automatically called by the framework.

Your Task

- Receive HTTP/1.1 requests from clients
- Answer requests:
  - Reject invalid packets
  - Parse the request and send the requested file
- Test with browser: `http://localhost:8000`

For this, edit the file `http.cpp`. The Function `handle_connection` is automatically called by the framework.

Your Task

- Receive HTTP/1.1 requests from clients
- Answer requests:
  - Reject invalid packets
  - Parse the request and send the requested file
- Test with browser: `http://localhost:8000`

For this, edit the file `http.cpp`. The Function `handle_connection` is automatically called by the framework.

Your Task

- Receive HTTP/1.1 requests from clients
- Answer requests:
  - Reject invalid packets
  - Parse the request and send the requested file
- Test with browser: `http://localhost:8000`

For this, edit the file `http.cpp`. The Function `handle_connection` is automatically called by the framework.

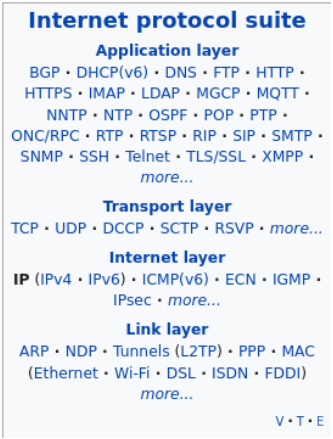
How do packets arrive here?

- Network as a stack: flexible but complex
- Internet Protocol Suite
- From lowest to highest layer:
  1. Link: Physical (e.g. Ethernet)
  2. Internet: Between networks (e.g. IP)
  3. Transport: Connections (e.g. TCP)
  4. Application: Inter-Program (e.g. HTTP)

How do packets arrive here?

- Network as a stack: flexible but complex
- Internet Protocol Suite
- From lowest to highest layer:

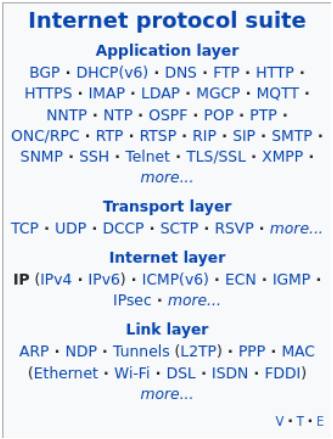
1. Link: Physical (e.g. Ethernet)
2. Internet: Between networks (e.g. IP)
3. Transport: Connections (e.g. TCP)
4. Application: Inter-Program (e.g. HTTP)



How do packets arrive here?

- Network as a stack: flexible but complex
- Internet Protocol Suite
- From lowest to highest layer:

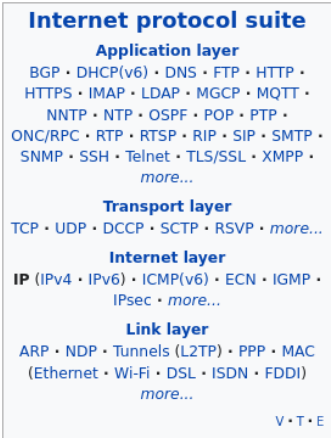
1. Link: Physical (e.g. Ethernet)
2. Internet: Between networks (e.g. IP)
3. Transport: Connections (e.g. TCP)
4. Application: Inter-Program (e.g. HTTP)



How do packets arrive here?

- Network as a stack: flexible but complex
- Internet Protocol Suite
- From lowest to highest layer:

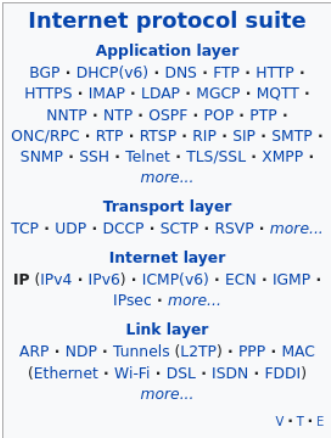
1. Link: Physical (e.g. Ethernet)
2. Internet: Between networks (e.g. IP)
3. Transport: Connections (e.g. TCP)
4. Application: Inter-Program (e.g. HTTP)



How do packets arrive here?

- Network as a stack: flexible but complex
- Internet Protocol Suite
- From lowest to highest layer:

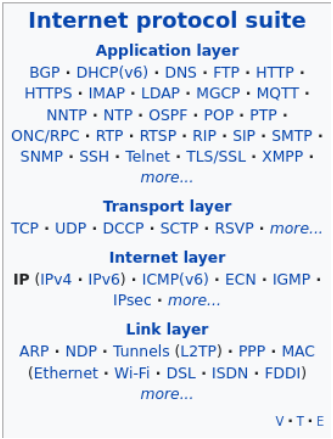
1. Link: Physical (e.g. Ethernet)
2. Internet: Between networks (e.g. IP)
3. Transport: Connections (e.g. TCP)
4. Application: Inter-Program (e.g. HTTP)



How do packets arrive here?

- Network as a stack: flexible but complex
- Internet Protocol Suite
- From lowest to highest layer:

1. Link: Physical (e.g. Ethernet)
2. Internet: Between networks (e.g. IP)
3. Transport: Connections (e.g. TCP)
4. Application: Inter-Program (e.g. HTTP)



How do packets arrive here?

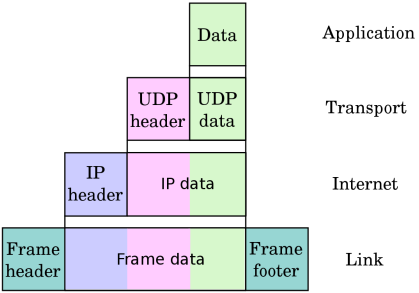


Figure: Example: UDP over IP over Ethernet. [1]

[1] Source https://commons.wikimedia.org/wiki/File:UDP_encapsulation.svg

How do packets arrive here?

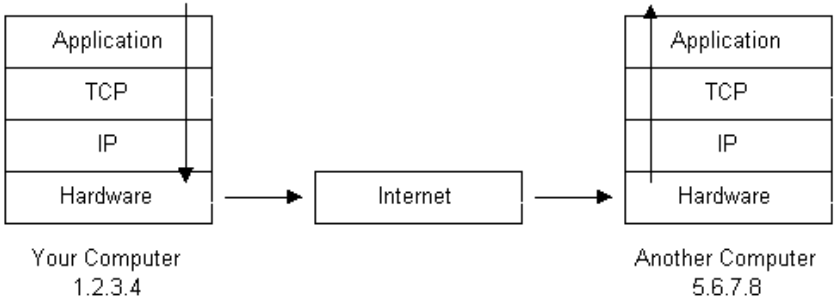


Figure: Example: Two Computers, TCP over IP over Ethernet. [1]

[1] Source https://medium.com/@anna7/internet-protocol-layers-in-internet-protocol-suite-tcp-ip-abe038c0adde

About network stability

Why do we need to `read` multiple times? – Slow networks! Big requests!

About network stability

Why do we need to `read` multiple times? – Slow networks! Big requests!

Why do we need to `write` multiple times? – The operating system handles slow connections for us *but* can be interrupted

HTTP

- Base HTTP is request & response protocol

HTTP

- Base HTTP is request & response protocol
- Client requests with method: `GET`, `HEAD`, `OPTIONS`, `POST`, `PUT`, `DELETE`, `TRACE`, `CONNECT`, `PATCH`
- Download data from server, or send data to server
- Example:

```
GET /index.html HTTP/1.1
...
```

HTTP

- Base HTTP is request & response protocol
- Server responds: `200 OK`, `400 Bad Request`, ...
- Codes: `1xx` informational response, `2xx` success, `3xx` redirection, `4xx` client errors, `5xx` server errors
- Example:

```
HTTP/1.1 200 OK
...
```

HTTP

- Base HTTP is request & response protocol
- Some requests and responses include a body
- Uploading data to the server, or downloading data from the server

HTTP

- Base HTTP is request & response protocol
- Both directions use headers to send metadata, e.g. `Host`, `Connection`, `Content-Type`, `Content-Length`, `Range`
- Client-Headers tell: Who, from where, ...
- Server-Headers answer: What, How much data, ...

HTTP

- Full Example:

```
GET /lorem.txt HTTP/1.1
Host: localhost:12345
```

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/plain
Content-Length: 17

lorem ipsum dolor
```

HTTP

```
❯ make run
./server
Server startup ok - listening on port 8000.
Incoming connection from 127.0.0.1:47690...
Requesting file ./webroot/lorem.txt
HEADER Host: localhost:12345
Request headers OK.
Sending file from start=0 to end=17
```

```
❯ nc localhost 8000
GET /lorem.txt HTTP/1.1
Host: localhost:12345

HTTP/1.1 200 OK
Connection: close
Accept-Ranges: bytes
Content-Length: 17
Content-Type: text/plain

lorem ipsum dolor
```

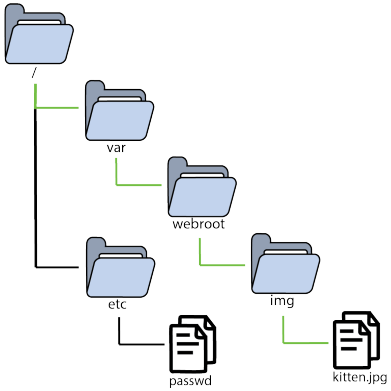
Figure: Example request

Webroot escape attacks

- The server “serves” static files, i.e. lets clients request files from the disk
- Privacy: Only allow access to public folder (webroot)
- Client requests relative path, server must make sure it is inside webroot while resolving full path
- Relative path elements: current directory `./`, parent directory `../`

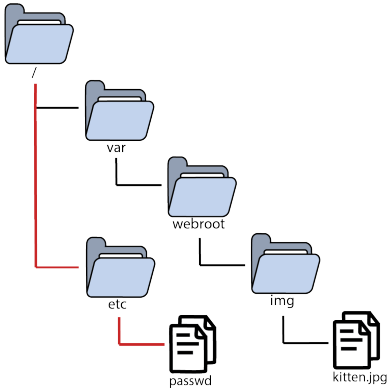
Webroot escape attacks

- webroot: `/var/webroot`
- valid access:
  - `/index.html` → `/var/webroot/index.html`
  - `/img/kitten.jpg` → `/var/webroot/img/kitten.jpg`
  - `/img/../lorem.txt` → `/var/webroot/lorem.txt`



Webroot escape attacks

- webroot: `/var/webroot`
- invalid access:
  - `/../../etc/passwd` → ~~`/etc/passwd`~~
  - `/img/../../personal.txt` → ~~`/var/personal.txt`~~



Webroot escape attacks

```
❯ ./server
Server startup ok - listening on port 8000.
Incoming connection from 127.0.0.1:48144...
Requesting file ./webroot/../secret.txt
webroot path: /var/webroot :: request path: /var/secret.txt
HEADER Host: localhost:12345
Request headers OK.
Sending file from start=0 to end=18
```

```
❯ nc localhost 8000
GET /../secret.txt HTTP/1.1
Host: localhost:12345

HTTP/1.1 200 OK
Connection: close
Accept-Ranges: bytes
Content-Length: 18
Content-Type: text/plain

Super secret file
```

Figure: Requesting a file outside webroot

Webroot escape attacks

Preventing such attacks:

- Use `realpath` to resolve request and webroot and compare
- Reject requests with relative path elements

Tips

- Read data line by line
- *Always* assume an invalid request!
- Use the hints in the footnotes – `man` pages!
- Start server using `make run`, check for leaks with `make valgrind`
- Test your implementation using `netcat`
- Print debug data about the request, its parts and function calls
- Close both the connection *and* the file at the end, deallocate memory

Tips

- Read data line by line
- *Always* assume an invalid request!
- Use the hints in the footnotes – `man` pages!
- Start server using `make run`, check for leaks with `make valgrind`
- Test your implementation using `netcat`
- Print debug data about the request, its parts and function calls
- Close both the connection *and* the file at the end, deallocate memory

Tips

- Read data line by line
- *Always* assume an invalid request!
- Use the hints in the footnotes – `man` pages!
- Start server using `make run`, check for leaks with `make valgrind`
- Test your implementation using `netcat`
- Print debug data about the request, its parts and function calls
- Close both the connection *and* the file at the end, deallocate memory

Tips

- Read data line by line
- *Always* assume an invalid request!
- Use the hints in the footnotes – `man` pages!
- Start server using `make run`, check for leaks with `make valgrind`
- Test your implementation using `netcat`
- Print debug data about the request, its parts and function calls
- Close both the connection *and* the file at the end, deallocate memory

Tips

- Read data line by line
- *Always* assume an invalid request!
- Use the hints in the footnotes – `man` pages!
- Start server using `make run`, check for leaks with `make valgrind`
- Test your implementation using `netcat`
- Print debug data about the request, its parts and function calls
- Close both the connection *and* the file at the end, deallocate memory

Useful functions

C

- Compare strings: `strcmp` / `strcasecmp` (case insensitive)
- Find character: `strchr` (first) / `strrchr` (last)
- Allocate space dynamically: `malloc` (and `free` after use)
- Convert string to number: `atol` (to long) / `strtol` (more control)

Useful functions

C

- Copy data: `memcpy` (data) / `strcpy` (strings) / `strncpy` (max length)
- Files: `stat` (file information) / `fopen` (open file) / `fseek` (move in file) / `fread` (read from file) / `fclose` (close file)
- Output: `printf` (write to stdout) / `dprintf` (write to connection)

Useful functions

C++

- Strings Class: `std::basic_string`
- Read file: `std::basic_ifstream`
- Convert string to number: `std::stol`
- Memory: `new` / `new[]`, `delete` / `delete[]`

`std::string` members: `size`, `append`, `substr`, `find_first_of`, etc.

Code Examples

All examples can be found on the course website and on discord.
We want to reads in words (separated by space), split each word by `:` and reply to client.

- `ping_simple.cpp` Simplistic implementation, no multi-read
- `ping.cpp` Do multiple reads to get all data
- `ping_cpp.cpp` C++ implementation of the above

Code Examples

All examples can be found on the course website and on discord.
We want to reads in words (separated by space), split each word by `:` and reply to client.

- `ping_simple.cpp` Simplistic implementation, no multi-read
- `ping.cpp` Do multiple reads to get all data
- `ping_cpp.cpp` C++ implementation of the above

Examples

Questions?