Web and E-Mail

Jens Lechtenbörger

Summer Term 2023

Contents

1 Introduction

1.1 Learning Objectives

- Explain communication patterns for Web and e-mail exchanges
  - Perform simple HTTP requests via `telnet` or `gnutls-cli`
  - Interpret E-Mail headers
- Explain the concept of "stateless servers"
- Explain constraints and advantages of caching
- Discuss alternatives to and weaknesses of e-mail security established by secure channels between MUA and MTA

1.2 Previously on CACS ...

1.2.1 Communication and Collaboration

- Communication frequently takes place via the **Internet**
  - Telephony

  - Instant messaging
  - E-Mail
  - Social networks
- Collaboration frequently supported by tools using **Internet** technologies
  - All of the above means for communication
  - ERP, CRM, e-learning systems
  - File sharing: Sciebo, etherpad, etc.
  - Programming (which subsumes file sharing): Git, subversion, etc.
- All of the above are instances of **DSs**

1.2.2 Recall: Internet Architecture

- "Hourglass design"

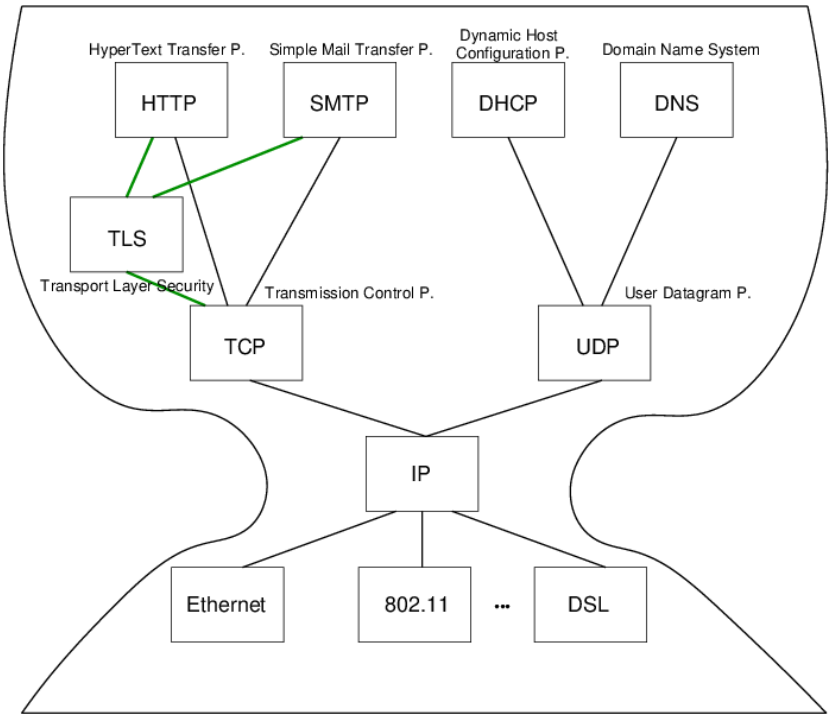

Figure 1: Internet Architecture with narrow waist

- IP is focal point
  - "Narrow waist"
  - Application independent!
    - Everything over IP
  - Network independent!

        - IP over everything
    - No security
        - "IP datagrams are like postcards, written with erasable pencils"

- Today: **HTTP** and **SMTP** at application layer

1.3 Today's Core Questions

- What does your browser do when you enter a URI in the address bar?
- How does e-mail transfer work?

2 Web

2.1 History of the Web (1/2)

- 1945, Vannevar Bush: As we may think
    - Memex for information storage
    - Associative indexing (Hyperlinks)
- 1989, article by Tim Berners-Lee
    - Distributed hypertext system, "»web« of notes with links"
    - Initially for cooperation among physicists at CERN
- May 1991
    - Distributed information system based on HTML, HTTP, and client software at CERN
- August 1991
    - Availability of CERN files announced in `alt.hypertext`

2.2 History of the Web (2/2)

- 1992, NCSA **Web Server** available
    - National Center for Supercomputing Applications, University of Illinois, Urbana-Champaigne
- 1993, Mosaic **browser** created at NCSA
- 1994, World Wide Web Consortium (W3C) founded by Tim Berners-Lee
    - Publication of technical reports and "recommendations"
- Now
    - Web 2.0, Semantic Web (aka Web 3.0 [Hen09]), cloud computing, browser as access device

2.3 WWW/Web

- Standards
  - W3C (HTML 4 Specification)
    - "The World Wide Web (Web) is a network of information resources."
  - HTTP/1.1 Specification (RFC 7230)
    - "The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems."
- Distributed information system
  - Client-Server architecture
    - Web clients (browsers) and servers exchange HTTP messages based on Internet standards
  - Sample Web standards (application layer of Internet architecture)
    - URIs (Uniform Resource Identifiers, next slide)
    - HTTP (this presentation)
    - ((X)HTML, CSS)

2.3.1 URI

- URI = Character string to **identify** entities
  - RFC 3986
    - Format: <scheme>:<scheme-specific-part>
- Examples from RFC 3986 (some containing DNS names)
  - `http://www.ietf.org/rfc/rfc2396.txt`
  - `ldap://[2001:db8::7]/c=GB?objectClass?one`
  - `mailto:John.Doe@example.com`
  - `tel:+1-816-555-1212`
- Scheme-specific part often structured
  - <scheme>://<authority><path>?<query>#fragment
  - E.g., `https://oer.gitlab.io/oer-courses/cacs/Web-and-E-Mail.html?default-navigation#slide-uri`

3 HTTP

3.1 HTTP

- Hypertext Transfer Protocol
  - HTTP/1.1, RFC 7230

        - Plain text messages, discussed subsequently
    - HTTP/2, RFC 7540
        - Adds frame format with compression
    - HTTP/3 also upcoming
        - Uses QUIC over UDP as transport layer protocol

- Request/response protocol
    - Specific message format
    - Several access methods
    - Requires reliable transport protocol
        - Traditionally, TCP/IP with port 80 (or port 443 for HTTPS)

3.2 Excursion: Manual Connections

- HTTP (before HTTP/2) and SMTP are plain text protocols
    - With encrypted variants HTTPS and SMTPS (or STARTTLS)
- Enables experiments on the command line
    - Type (or copy&paste) request, see server response
    - For unencrypted connections, `telnet` can be used (preinstalled or available for lots of OSs)
    - For encrypted connections, `gnutls-cli` can be used (part of GnuTLS, which is free software)
        - TLS = Transport Layer Security
            - Successor to SSL
            - Layer between application layer and TCP, recall Internet architecture
            - Secure channels based on asymmetric cryptography

3.2.1 Warnings

- Next two slides demonstrate how to type HTTP commands (for an improved understanding of the protocol)
    - Subsequent examples with `www.informationelle-selbstbestimmung-im-internet.de` **require** GnuTLS
        - Server redirects from port 80 to port 443
    - If your manual typing is too slow, connections may **time out** (e.g., "Peer has closed the GnuTLS connection")
    - Also, use of backspace or cursor keys may destroy connections
- Suggestion: Type in text editor and copy&paste into command line

3.2.2 telnet

- Original `telnet` purpose: Login to remote host
  - **Insecure** plaintext passwords
  - Nowadays, remote login performed with Secure Shell, `ssh`
- Establish **TCP connection** to destination port
  - `telnet www.google.de 80` (port 80 for HTTP)
    * (For variants without visual feedback possibly followed by `ctrl-+` or `ctrl-]`, `set localecho` [enter] [enter])
    * `GET / HTTP/1.1` [enter]
    * `Host: www.google.de` [enter] [enter]
    * (Screencast)
    * (Context for above lines soon)
  - **Beware**: Buggy telnet implementations may stop sending after first line (use Wireshark to verify)

Here, you see a sample use of `telnet` to open a TCP connection to port 80 on a Google server. You could try out any other number to check on what ports the server is prepared to talk with you. Port 80 is reserved for HTTP, which is slowly phased out in favor of the cryptographically secured variant HTTPS on port 443.

Anyways, once a TCP connection is established successfully, you can send data to the server by typing it. When typing, you need to "speak" the protocol that is expected by the server, here HTTP, and the lines starting with `GET` as well as with `Host` are both part of the HTTP protocol, which is explained on later slides.

Note that you cannot use `telnet` with encrypted connections as you would need to type bytes that setup and use cryptographic protocols then. Thus, while you can *open* a TCP connection to port 443 with `telnet`, it is unlikely that you can *use* that connection by typing the necessary bytes for cryptographic protocols afterwards.

For cryptographically secured connections, you may want to use the GnuTLS client as shown on the next slide.

An aside: On the slide, "ctrl-+" means: Press the `ctrl` key and `+` simultaneously. Similarly for other keys.

3.2.3 gnutls-cli

- Establish TLS protected TCP connection with GnuTLS
  - Alternative to `telnet` on previous slide (screencast)
  - `gnutls-cli --crlf www.informationelle-selbstbestimmung-im-internet.de`
    * (HTTPS on port 443 by default)
    * `GET /chaosreader.html HTTP/1.1` [enter]
    * `Host: www.informationelle-selbstbestimmung-im-internet.de` [enter] [enter]
  - SMTP for e-mail, port 587 as alternative to 25
    * `gnutls-cli --crlf --starttls -p 587 secmail.uni-muenster.de`
      · (Type `ehlo localhost`, then `starttls`; press `ctrl-d` to enter TLS mode; needs authentication)

The cryptographic protocol suite TLS is used in two major variants.

1. A special port, e.g., 443, is reserved for cryptographically secured connections. The connecting client (here, GnuTLS) must immediately "talk" a cryptographic protocol.
2. A single port, e.g., 25, supports plaintext as well as cryptographically secured connections. Here, the client starts with plaintext (as with `telnet`), but can issue a specific command (here, `starttls` followed by `ctrl-d`) to switch to a cryptographically secured connection.
   - "ctrl-d" means: Press the `ctrl` key and `d` simultaneously.

In any case, application data is transmitted through secure channels.

3.3 Excursion: Browser Tools

- Modern browsers offer developer tools
  - E.g., press `ctrl-shift-I` with Firefox
  - Tools to inspect HTML, CSS, Javascript
  - Tools to inspect HTTP traffic (Network tab)
    - Live view on browser requests and server responses
      - With details on timing, caching, headers
  - Console with error messages
  - And much more

3.4 HTTP Messages

- Requests and responses
  - Generic message format of RFC 822, 1982 (822→2822→5322)
    - Originally for e-mail, extensions for binary data
      - Lines end with CRLF, `\r\n` below (press `enter`, do not type this)
  - Messages consist of
    - Headers
      - In HTTP always a distinguished start-line (request or status)
      - Then zero or more headers
    - Empty line
    - Optional message body
  - Sample `GET` **request** (does not have a body)
    - ```
      GET /chaosreader.html HTTP/1.1\r\n
      Host: www.informationelle-selbstbestimmung-im-internet.de\r\n
      \r\n
      ```
- Excerpt of sample HTTP **response** to previous `GET` request
  - ```
    HTTP/1.1 200 OK\r\n
    Date: Wed, 08 Apr 2020 13:30:10 GMT\r\n
    Server: Apache\r\n
    Last-Modified: Wed, 24 Jul 2019 12:25:46 GMT\r\n
    ETag: "2cd1-58e6c6898dce2"\r\n
    ```

```
Content-Length: 11473\r\n
more headers omitted
Content-type: text/html; charset=utf-8\r\n
\r\n
HTML code as body
```

3.5 HTTP Methods

- Case-sensitive (capital letters)
  - `GET` (Request for resource, see section 4.3.1)
  - `HEAD` (Request information on resource, see section 4.3.2)
  - `POST` (Transfers entity, see section 4.3.3)
    - Annotations, postings, forms, database extensions
  - `PUT` (Creates new resource on server, see section 4.3.4)
  - `DELETE` (Deletes resource from server, see section 4.3.5)
  - `CONNECT` (Establish tunnel with proxy, see section 4.3.6)
  - `OPTIONS` (Asks for server capabilities, see section 4.3.7)
  - `TRACE` (Tracing of messages through proxies, see section 4.3.8)

3.6 Conditional GET

- `GET` under conditions
  - Requires (case-insensitive) request header
    - (Can be used by browser to check if cached version still fresh)
    - Different types, e.g.: `If-Modified-Since`, `If-Match`, `If-None-Match`
- Example
  - Request
    - ```
      GET /chaosreader.html HTTP/1.1
      Host: www.informationelle-selbstbestimmung-im-internet.de
      If-None-Match: "2cd1-58e6c6898dce2"
      ```
  - Response
    - ```
      HTTP/1.1 304 Not Modified
      Date: Wed, 08 Apr 2020 14:07:31 GMT
      additional headers
      ```

Please revisit the response for an earlier HTTP request. Note that the response contains a `Last-Modified` date and an `ETag`. Both pieces can be used for conditional gets. While the date is probably self-explanatory, the ETag is some version identifier provided by the server. Changed page contents are reflected in changed ETag values (but not necessarily the other way round).

On this slide, you see a conditional GET with the ETag value `"2cd1-58e6c6898dce2"` from the previous response. As the server's ETag value did not change, it responds with status code 304, indicating that no modification took place. Hence, a cached result would still be fresh and usable, saving bandwidth and reducing transmission delays.

3.7 Sample Status Codes

- Three digits, first one for class of response
  - 1xx: Informational - Request received, continuing process
    - 100: Continue - Client may continue with request body
  - 2xx: Successful - Request successfully received, understood, and accepted
    - 200: OK
  - 3xx: Redirection - Further action necessary to complete request
    - 302: Found (temporarily under different URI)
    - 303: See Other (redirect to different URI in `Location` header)
    - 304: Not Modified (previous slide)
  - 4xx: Client Error - Request with bad syntax or cannot be fulfilled
    - 403: Forbidden
    - 404: Not Found
  - 5xx: Server Error - Server failed for apparently valid request

3.8 Review Question

- Did you execute `GET` requests and conditional `GET` requests on the command line? Any surprises?
  - Note that examples with `www.informationelle-selbstbestimmung-im-internet.de` require GnuTLS (server redirects HTTP requests on port 80 to HTTPS port 443).

4 Server State and Cookies

4.1 State Models

- **Stateless**: Server does not maintain client state
  - Advantages
    - Simplified server design, reduced resource usage
    - State changes on server do not require client notifications
    - Recovery (restart after server crash) "simple": No client state to restore
  - E.g.: HTTP, DNS
    - Server forgets client after request
    - No session
- **Stateful**: Server maintains client state
  - E.g., file server with table of pairs (Client, File) for caching
    - Keep track which client has current version
    - Performance improvement via locality

  - Recovery requires to restore consistent state

Two major state models exist for servers in distributed systems, namely stateful and stateless ones. Don't worry if the idea of a stateless server seems surprising. Being stateless means being simple, much simpler than what we usually use, but with some advantages shown here.

On the negative side, if a stateless server receives the same request multiple times in short sequence from the same client, it will happily provide the same, potentially long and costly answer again and again. In contrast, a stateful server might save time and resources with a brief reply: "Hey, my previous answer is still valid."

While some stateless protocols, such as DNS, are used to provide stateless services, other stateless protocols, such as HTTP, are used as building block for stateful applications. This is what we see whenever we log in to some Web site, with some examples on the next slide.

How state can be tracked via the stateless HTTP with session IDs, is addressed afterwards.

Also, you already saw that client-side state can be used to form conditional GET requests: When the browser receives a reply, it remembers the resource's ETag or modification date and can use this information in subsequent conditional requests, which is the basis for HTTP caching later on.

Importantly, in all cases the HTTP server acts statelessly.

4.2 Stateful Web Applications

- HTTP is stateless
  - Yet, Web applications often maintain client state
    - E.g., personalized session after login
      - Virtual shopping cart
      - Shopping history, preferences
      - Exercises in Learnweb
  - Solution for stateful applications
    - Manage state of related requests as session outside HTTP
      - Usually in database systems
    - Server-side code
      - (Not part of Web server, but of separately programmed application)
      - Gets executed for incoming HTTP requests
      - Manages state and uses HTTP messages to transfer session IDs (next slide)

4.3 Session IDs

- Session ID = Identifier to connect subsequent/related requests and responses
  - Typical variant: Client-side storage of IDs in browser
    - ID sent by server S, stored by browser (cookie or local storage)
    - Browser includes IDs set by S for every subsequent visit of S
      - Think of automatic ID card (whose contents you do not understand)
      - My browsers remove cookies and clear local storage upon exit
  - Alternative: Server-side, session ID embedded in dynamically generated URIs

        - May hinder caching
            - URI does not identify resource any longer

4.3.1 Cookies (1/2)

- RFC 6265: HTTP State Management Mechanism
    - Idea
        - Client stores data sent by server
        - Client sends this data with subsequent requests
            - Without understanding that data at all
    - Details
        - Cookie is **named byte string**
        - Server transfers cookie in `Set-Cookie` (2) header in response
            - `Set-Cookie`: Version 0/Netscape and RFC 6265
            - `Set-Cookie2`: Version 1, RFC 2965
            - (Besides, JavaScript may create cookie at client)
        - Client sends cookie in `Cookie` header in requests

4.3.2 Cookies (2/2)

- Note: Sometimes you may read that cookies are text files
    - That is usually wrong, misleading, and irrelevant
    - Modern browsers store cookies as rows in a relational database
        - Storage in filesystem or database is an implementation detail
- Cookies have name, value, optional attributes/flags, e.g.:
    - `Expires`, `Max-Age`
        - Determine lifetime of cookie
        - If both missing: "Session" cookie to be deleted when browser exits
    - `Domain`
        - DNS domain of servers to which the cookie should be sent

5 Caching

5.1 HTTP Caching

- Caching reduces latency and server load for identical requests

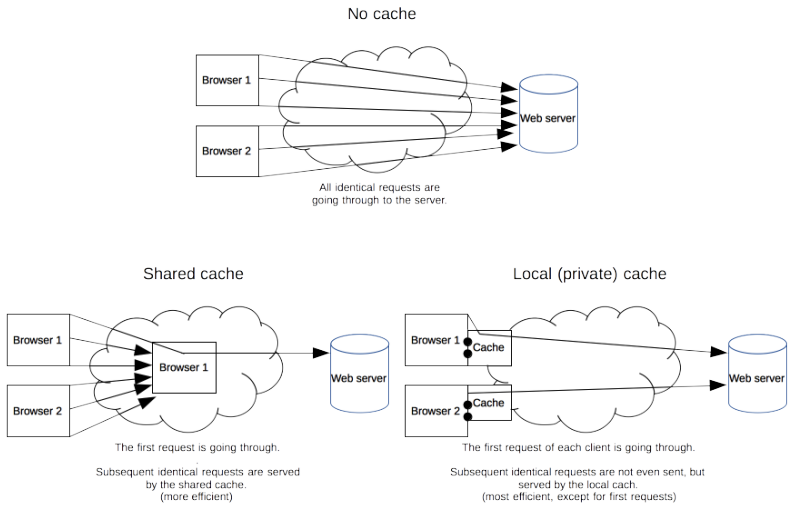

Figure 2: "HTTP cache types" by Mozilla Contributors under CC BY-SA 2.5; from MDN web docs

- HTTP caching assumptions
  - URI identifies resource, stability, client-independence
- Semantic transparency
  - Caching is not visible to users
  - Response from cache is equivalent to hypothetical one from server

The upper part of the figure shows a situation without caching, where identical requests for the same resources need to be transmitted to the server again and again.

With caching, two major variants exist as shown in the lower part: On the right, each browser has its own private cache (which is the case for usual browsers), which means that identical requests from the same browser can now be answered from the cache. Clearly, this reduces (a) latency as no network transmission is necessary and (b) server load. Even on Web pages with dynamic contents, such caching is useful as lots of resources such as images, scripts, and style information remain stable for longer periods of time.

The lower left shows a cache that is shared by multiple browsers. Usually, such caches are implemented by so-called proxy servers, for example in company or university settings. Web proxies intercept and inspect outgoing requests. They forward requests for unknown resources to origin servers to obtain responses which they then store locally to deliver cached responses for subsequent identical requests.

In any case, resources to be cached are identified by their URIs. For caching to be effective, URIs and their resources should be stable for some time. For shared caches, URIs must be client-independent, i.e., all clients must receive the same resource for the same URI.

As indicated on the next slide, servers may include expiration dates in their responses, beyond which resources should not be cached. Also, conditional GET requests can be used to validate whether cached results are still current.

5.2 HTTP Caching Mechanisms

- Expiration
  - Server may indicate expiration date in Expires or Cache-Control header

- Validation
  - After expiration date, cache must check whether resource still usable
  - May return new expiration date
    - Conditional `GET` ("Slow hit")

5.3 HTTP Caching Rules

- Complex rules, lots of details
  - (Some details on Cache Control header)
- Server may limit caching
  - `no-store`, `no-cache`, `must-revalidate`
- Client may
  - enforce validation
    - `no-cache`
  - forbid caching
    - `no-store`

6 Proxies

6.1 Web Proxies

- Web proxy server is intermediary between client and server
  - Acts as server to client
    - Proxy accepts request from client
      - Then acts as client to server to obtain response
    - Proxy delivers response to client
  - Acts as client to server
    - Proxy sends request of real client to server
      - Server just sees some client request
    - Proxy obtains response from server

6.2 Sample Proxy Applications

- Cache
- Firewall/Content filter/Ad blocker
- Anonymizer, e.g., Tor
  - My privacy policy recommends surfing via Tor
- Debugging tool
  - E.g., intercept and analyze app network data

- Surrogate/Reverse proxy, Content Delivery Network (CDN)
  - Replicated contents, inbound messages intercepted and redirected, e.g.:
    * Load balancing
    * Geographical diversity (reduced latency, increased availability)

6.3 Review Questions

- What is a stateless protocol? Given that HTTP is a stateless protocol, how can lots of applications that apparently require state be implemented on top of HTTP?
- Where are HTTP caches typically located? What impact might HTTPS have on caching?

7 E-Mail

7.1 E-Mail Basics

- Among oldest Internet applications
- Message format
  - Based on RFC 822, 1982 (later taken up in HTTP)
  - Extended with Multipurpose Internet Mail Extensions (MIME)
    * `Content-Type` (type of data contained in message)
    * `Content-Transfer-Encoding` (how data in message body is encoded)
- Plaintext messages
  - E-mail is like **postcard**, written with **erasable pencil**
    * Neither confidentiality nor integrity
  - Learn self-defense, use GnuPG if you don't like this
    * SSL/TLS insufficient approach, recall end-to-end security

7.2 Message Transfer

- Terminology
  - **Mail User Agent** (MUA): Your mail reader
    * E.g., browser, Thunderbird, Emacs
  - **Mail Transfer Agent** (MTA): Mail server/daemon
    * E.g., sendmail, exim, postfix

- **Simple Mail Transfer Protocol**, 1982 (SMTP, RFC 821→2821→5321)

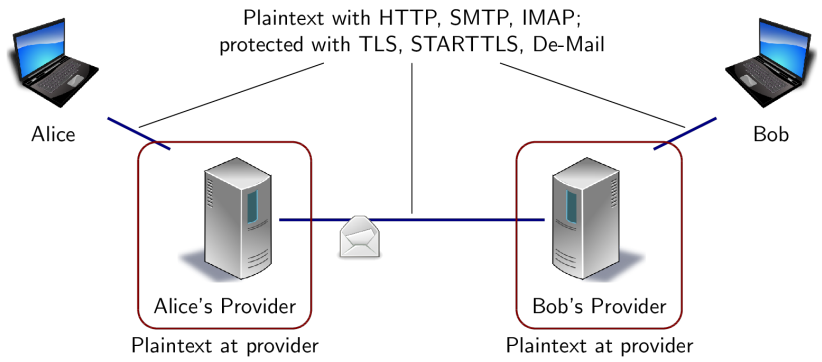

Figure 3: Hop-to-hop security of e-mail

  - Outgoing messages, MUA-to-MTA, MTA-to-MTA
    - Plaintext (TCP/IP, port 25)

The figure visualizes hop-by-hop transmission of e-mail with SMTP. When sending an e-mail, from an Internet point of view, your client (aka Mail User Agent) performs an end-to-end SMTP exchange with some mail server (mail servers are also called Mail Transfer Agents). A sample exchange is shown on the next slide. Usually, that server resides at the organization that gave you your e-mail address; if it is not responsible for the recipient's address, it performs another end-to-end SMTP exchange with a suitable server.

Finally, the recipient picks up the e-mail in another end-to-end exchange. Note that SMTP is only used to deliver e-mail to servers, not to users. So, the final exchange uses some other protocol such as IMAP in mail clients or HTTP in Web clients.

Nowadays, the exchanges between hops are usually encrypted, while the bulk of e-mail is plaintext, which is available for inspection and modification at each participating server.

7.3 SMTP

```
telnet wi 25
Trying 128.176.159.139...
Connected to wi.uni-muenster.de.
Escape character is '\^]'.
220 wi-vm700.wi1.uni-muenster.de Microsoft ESMTP MAIL Service ready at Tue, 27 Oct 2009 11
HELO mouse.nix
250 wi-vm700.wi1.uni-muenster.de Hello [128.176.159.107]
MAIL From: micky@mouse.nix
250 2.1.0 Sender OK
RCPT To: lechten@wi.uni-muenster.de
250 2.1.5 Recipient OK
DATA
354 Start mail input; end with <CRLF>.<CRLF>
Received: from mx1.disney.com ([192.195.66.20]) by smtp.mouse.nix Super Duper SMTP Server;
To: 42@universe.com
From: micky@mouse.nuix
Subject: Don't panic

Somebody Else's Problem!  (This is the message body after the empty
```

```
line.  Note that headers preceding the empty line have also been
entered manually.  They are ignored by SMTP, but displayed to user.)

.

250 2.6.0 <b13a2a36-f56b-43ec-ad81-41ec44190e6a@wi-vm700.wi1.uni-muenster.de> Queued mail
```

This is a real SMTP exchange via `telnet`, which demonstrates dramatic weaknesses in protocol design. Without going into too many details, note the following:

1. `MAIL From` is an SMTP command to indicate who sent the e-mail. I was allowed to enter any value, which was acknowledged to be OK.
2. `RCPT To` is an SMTP command to indicate who should receive the e-mail.
3. `DATA` is an SMTP command to say how the e-mail should look like. As stated earlier, e-mails have got header lines which are separated by an empty line from the body.

   As part of the data, I started with a fake "Received" header. Then, I entered fake "To" and "From" headers. By accident, I made a typo in the "From" header.

7.4 SMTP MUA Header

```
Microsoft Mail Internet Headers Version 2.0
Received: from wi-vm700.wi1.uni-muenster.de ([128.176.158.92]) by wi-vmail2005.wi1.uni-mue
Received: from mouse.nix (128.176.159.107) by wi-vm700.wi1.uni-muenster.de (128.176.159.13
Received: from mx1.disney.com ([192.195.66.20]) by smtp.mouse.nix Super Duper SMTP Server;
To: 42@universe.com
From: <micky@mouse.nuix>
Subject: Don't panic
MIME-Version: 1.0
Content-Type: text/plain
Message-ID: <b13a2a36-f56b-43ec-ad81-41ec44190e6a@wi-vm700.wi1.uni-muenster.de>
Return-Path: micky@mouse.nix
Date: Tue, 27 Oct 2009 11:22:28 +0100
X-OriginalArrivalTime: 27 Oct 2009 10:22:35.0473 (UTC) FILETIME=[66C35410:01CA56EF]
```

Most mail clients allow to see e-mail headers. This is what was shown for the e-mail from the previous slide.

Note that the recipient is *not* `42@universe.com`. SMTP does not care about the e-mail contents following the `DATA` command from the previous slide. The real recipient *was* shown on the previous slide, but once the message is delivered, no visible trace of that is left for the recipient.

Also note the "Received" headers. Some are real, one was typed on the previous slide.

Those must have been happy times when people designed and implemented such protocols without thinking of negative consequences.

7.5 Review Questions

- Who found the previous e-mail in his or her inbox?
- What parts of header data are trustworthy (to what degree)?

7.6 Concluding Questions

- What did you find difficult or confusing about the **contents** of the presentation? Please be as specific as possible. For example, you could describe

your current understanding (which might allow us to identify misunderstandings), ask questions in a Learnweb forum that allow us to help you, or suggest improvements (maybe on GitLab). Most questions turn out to be of general interest; please do not hesitate to ask and answer in the forum. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

8 Conclusions

8.1 Summary

- Web browsers and servers talk HTTP
  - Simple message format
  - Stateless request/response protocol
    - State via cookies
  - Different connection types
  - Caching for performance
- E-Mail transferred via SMTP

8.2 Outlook

- HTTP used for various applications
  - Web services (e.g., SOAP messages)
  - Ad-hoc request/reply protocols
- REST
  - Representational State Transfer
  - Software architecture for distributed hypermedia systems
    - Generalization of Web
    - Defining constraints
      - Client/Server
      - Stateless
      - Cacheable
      - Uniform interface, may use: URIs, MIME types, HTTP methods
      - Layered System
      - (Code on demand)

Bibliography

[Hen09] Jim Hendler. "Web 3.0 Emerging". In: *Computer* 42.1 (2009), pp. 111–113. DOI: 10.1109/MC.2009.30.

License Information