# INF226 – Software Security

Håkon Robbestad Gylterud

2019–09-23

# The world wide web

# The internet and the web

What happens when we open the browser and type in "www.uib.no"?

# Web protocols, formats and languages

Communication on the world wide web (WWW):

- Domain Name Service (DNS)
- Hyper Text Transfer Protocol (HTTP)
- Uniform Resource Identifier (URI)

# Browsers

- Perform **HTTP requests** on behalf of the users.
- Display pages in Hypertext Markup Language (**HTML**) containing:
    - Images (JPEG, GIF, SVG, PNG, $\cdots$)
    - Video and sound (MP3,MP4,OGG,webm, $\cdots$)
- Style the pages as described in Cascade Style Sheet (**CSS**).
- Executes JavaScript embedded in the pages.

# Web-servers

Web-servers respond to HTTP requests.

- Static websites vs dynamic web sites.
- Dynamic: Any language can be used on the server side.

# HTTP requests

**GET** is the most common requeet type. It fetches a resource at a specific URI.

**HEAD** fetches only the headers for the specified resource.

**POST** Posts content to a specified resource.

# HTTP requests

**GET** is the most common requeet type. It fetches a resource at a specific URI.

**HEAD** fetches only the headers for the specified resource.

**POST** Posts content to a specified resource.

Each request contains headers which specify meta-data about the request:

- Accepted formats/languages
- Cookies
- User agent
- ETag
- . . .

# HTTP responses

The server responds with:

- a status message (200, 404, 500 etc . . . )
- headers
- (possibly) the content of requested resource

# Public key cryptography and authentication

# Public key cryptography

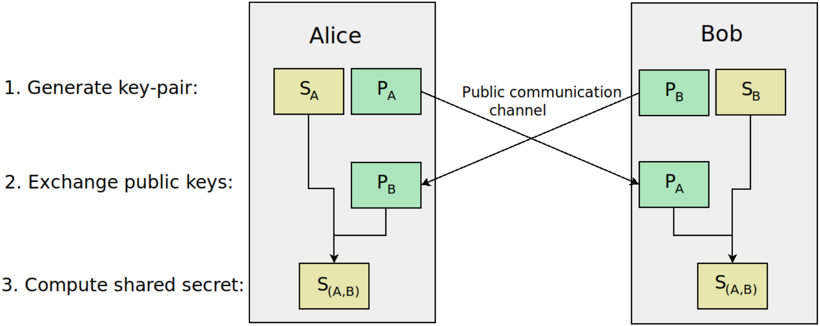


Figure 1: Public key cryptography

## Usage of public key systems

- Encryption
- Messsage authentication codes
- Certificates

# Man-in-the-middle attacks

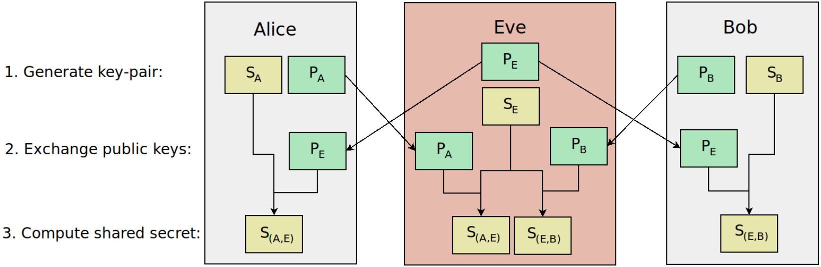


Figure 2: Public key cryptography

# Authentication for public key systems

Public keys must be authenticated.

## Authentication for public key systems

Public keys must be authenticated.

Many different schemes for this:

- Web-of-trust (key-signing)
- Trust upon first use
- Centralised certificate authorities
- . . .

# Trust upon first use

Assumption: The man in the middle does not strike the first time.

# Trust upon first use

Assumption: The man in the middle does not strike the first time.

Mechanism: Trust the public key used in first session. Use that for authentication of later sessions.

# Trust upon first use

Assumption: The man in the middle does not strike the first time.

Mechanism: Trust the public key used in first session. Use that for authentication of later sessions.

Works well for long-lasting trust-relationships. Or when no existing trust relationship exists (i.e. web-site registration).

**Example: SSH key trust.**

# Centralized Certificate Authorities

Assumption: We trust a central authority to verfiy public keys for us.

Mechanims: Central authority verifies identity and issues certificates on public keys.

Examples:

- Browsers ship with a list of public keys of trusted Certificate Authorities.
- Organsations can distribute their own certificates for internal use.

# Other schemes

For peer-to-peer authentication:

- one can use preexisting shared secrets (Example: Socialist Millionaire protocol)
- out-of-band communication (verfication of key fingerprints)

# Stream ciphers and Message Authentication Codes

# Stream ciphers

Most modern cryptography is based on **block ciphers**.

- Fixed input and output length (for instance: 128 bits)
- Deterministic: Same key and input gives same output.

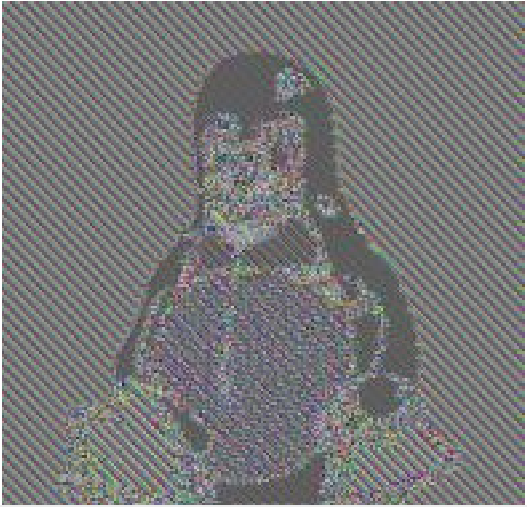
# Stream ciphers

Most modern cryptography is based on **block ciphers**.

- Fixed input and output length (for instance: 128 bits)
- Deterministic: Same key and input gives same output.

**Problem:** Most applications have variable length input/output.

# ECB



## Stream ciphers

# Stream ciphers

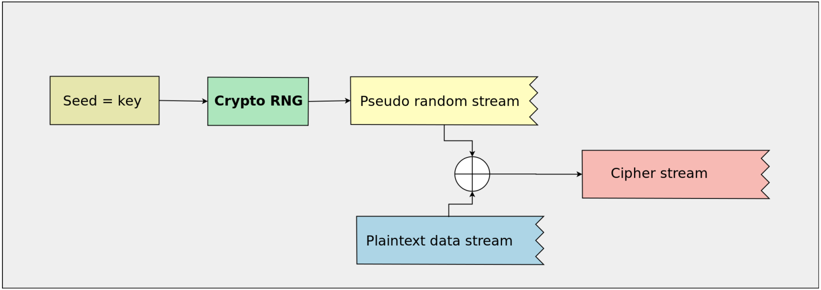


Figure 4: Stream ciphers

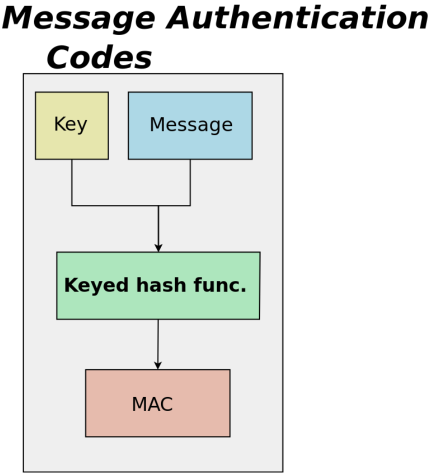
# Stream ciphers are mallable

Stream ciphers:

- based on cryptographic pseudo-random number generators (CPRNGs)
- provides safe extension to arbitrary inputs

BUT: They are mallable!

# Keyed hash functions



# Keyed hash functions

A keyed hash function produces a hash, which depends on the key.

- Used to authenticate messages:
  - Derive a key from shared secret.
  - Sender: Computes keyed hash of encrypted message and attach has.
  - Receiver: Computes keyed hash received message and compare with attached hash.
- Provides both *authenticity* and *integrity*.

# Keyed hash functions

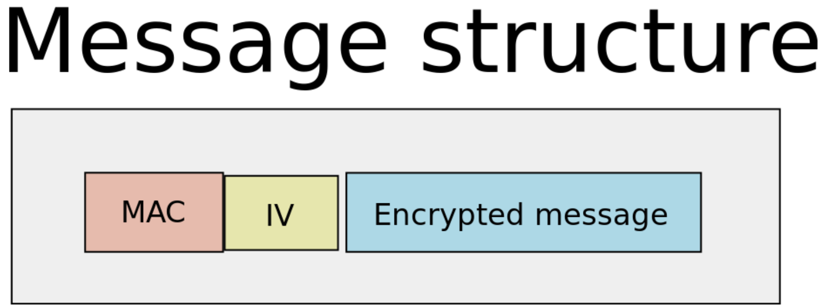


Figure 6: Keyed hash function

Question: Why is it important to encrypt first, not last?

# TLS

*Transport Layer Security* (TLS) are protocols providing communication security.

- Current version (TLS 1.3)
- Previous versions include *weak ciphers*.
- Provides:
  - Confidentiality
  - Authentication (Via X.509 certificates)
  - Forward secrecy

If you need cryptographic transport security: use TLS 1.3.

# Ciphers in TLS 1.3

TLS version 1.3 has drastically reduced the number of supported ciphers:

- AES in counter mode and CBC-MAC
- ChaCha20 and Poly1305 MAC

No more DES or RC4.

# HTTPS

HTTP can be transmitted over TLS (HTTPS). Authentication provided by Certificate Authorities (such as Let's Encrypt):

- Same-origin protocol separates HTTP from HTTPS.
- Many sites still serve content over plaintext HTTP.

# Logged in, and then what?

# Logged in, and then what?

- User actions are often given in separate requests from the authentication request.
- How do we ensure that each request comes from a valid user?

## Example: Webmail

1. `/login`
   - User requests login form, and enters password
2. `/inbox`
   - User posts login details to the inbox page
   - Server responds with inbox, listing messages, after checking password
3. `/delete?messageid=123`
   - User requests a message deleted
   - **How can the server know the user is the same?**

# Session IDs

The standard way solution is to use a **session ID**, which identifies the user in the following session.

Requires:

- Entropy: Session ID must not be guessable (random, 128 bits)
- Secrecy: Session ID must not be leaked:
  - HTTPS
  - Debugging modes often leak session IDs
  - Cross-site-scripting (Cookies: `HttpOnly`, `SameSite`).

# Common pitfall: Lacking entropy

Special care needs to be taken when generating random salts or secret keys.

- Entropy is a finite resource on any system.
- Not all random number generators are suitable for cryptographic use.

Use the recommended source of randomness on your system!

# java.util.Random

- java.util.Random is a Linear Congruential Generator (LCG).

**Using `java.util.Random` is a very insecure source of cryptographic randomness**:

- By observing only a few bytes of output from an LCG, one can completely determine the rest of the sequence.

(LCGs are well suited for statistical work and Monte-Carlo simulations)

# Generating secure random bytes in Java

You can use SecureRandom as a general purpose source of entropy:

Code

```
import java.security.SecureRandom;
...
SecureRandom random = new SecureRandom();

final byte[] token = new byte[32];
random.nextBytes(token);
```

## Generating secure random keys in Java

Different ciphers have different `KeyGenerator` implementations in Java. For instance AES:

```
javax.crypto.KeyGenerator;
javax.crypto.SecretKey;
...
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(256); // Specifying the key-size
SecretKey secretKey = keyGen.generateKey();
```

# Structure of a user authentication scheme based on passwords

1. Provide a way for user to authenticate server (ex: HTTPS w/valid certificate)
2. Establish a secure communication channel (ex: HTTPS)
3. User transmits password
4. Server verifies password:
   - Salted (128 bit)
   - Run through an expensive key derivation function (ex: SCrypt)
5. Server responds with a secure session ID
6. Client program stores session ID as securely as possible

# Structure of a user authentication scheme based on passwords

1. Provide a way for user to authenticate server (ex: HTTPS w/valid certificate)
2. Establish a secure communication channel (ex: HTTPS)
3. User transmits password
4. Server verifies password:
    - Salted (128 bit)
    - Run through an expensive key derivation function (ex: SCrypt)
5. Server responds with a secure session ID
6. Client program stores session ID as securely as possible

- Are there alternatives to sending the password to the server?
- Two-factor would be better.