Public Key Infrastructures and Cryptographic Protocols

Introduction to Security (192.019)

Simon **Jeanteur**, Mauro **Tempesta**

Security & **Privacy** Research Unit (192-06)
https://**secpriv.wien**

Who Owns a Public Key?

- We have assumed so far that the owner of a public key is known a priori
  - In practice, **everybody** could create a new key pair and claim that the public key belongs to somebody else
  - An **authenticated key exchange** requires **large effort**
    - Communication parties need to meet to exchange their keys
    - Not feasible when they are far apart from each other

- Modern systems use **public key certificates** to bind the public key to its owner

Public Key Certificates

Public Key Certificates

- A public key certificate is a **digital document** used to prove the **authenticity** of a public key
  - Format defined by the standard **X.509**
  - Typically issued by a trusted **certificate authority** (CA)
  - Contains information about the **public key**, the owner (**subject**), **issuer**, intended **key usage**, …
  - Contains a **signature** computed by the CA on the certificate's content

- If the signature is **valid** and the issuer is **trusted**, the key can be used to communicate with the subject specified in the certificate

Common Certificate Content

Key type (e.g., RSA), parameters (e.g., modulus)

Public Key

Information about the key owner (name, address, country, domain name, ...)

Subject

Information about the issuing CA (name, address, country, ...)

Issuer

From / until when the certificate is valid

Validity

Key usage

Key usage (encryption, signing other certificates, ...)

Subject Alternative Names

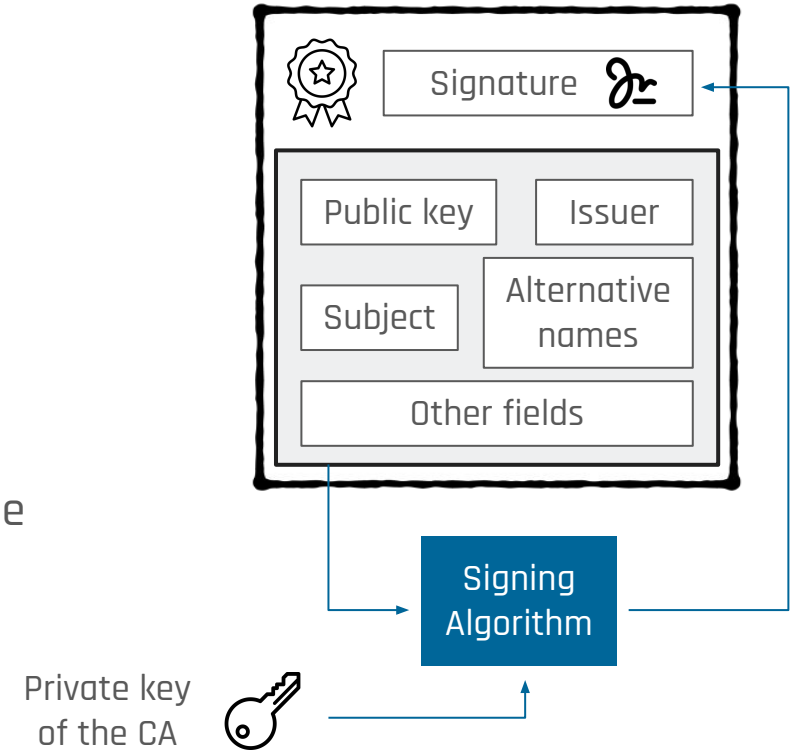
Domains for which the certificate is valid `*.example.com` -> all subdomains of `example.com` (wildcard certificates)

Signature

Algorithms used for computation, signature value

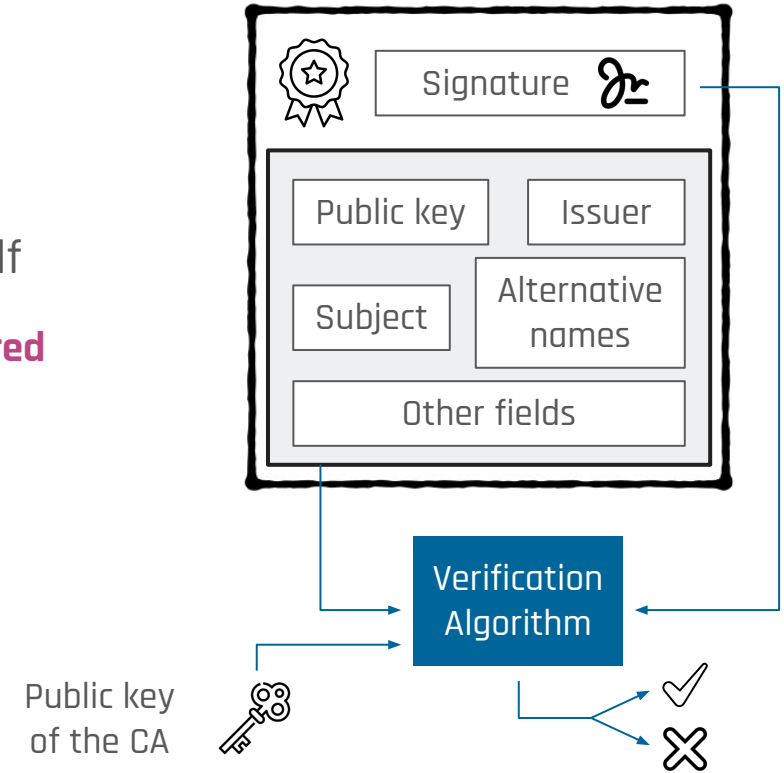
Creation of a Certificate

- The **registration authority** verifies the correctness of the data about the subject requesting the certificate
- The data contained in the certificate is signed with the **private key** of the issuing certificate authority
- The signature is entered into the certificate



Validation of a Certificate

- The **public key** of the issuing certificate authority is used to check whether the signature contained in the certificate matches the content of the certificate itself
  - If not, the content has either been **tampered** or **another entity** created the signature
  - In such a case, the certificate is treated as **invalid**



Authenticity of the CA's Public Key

- When validating the signature in a certificate, the public key of the issuing CA is used to verify the signature
  - How can we verify that the public key of the CA is really **authentic**?
  - By validating the certificate associated to the public key, which can be issued by the same or another CA
  - … and so on!

- The validation process stops when a **root certificate** is found

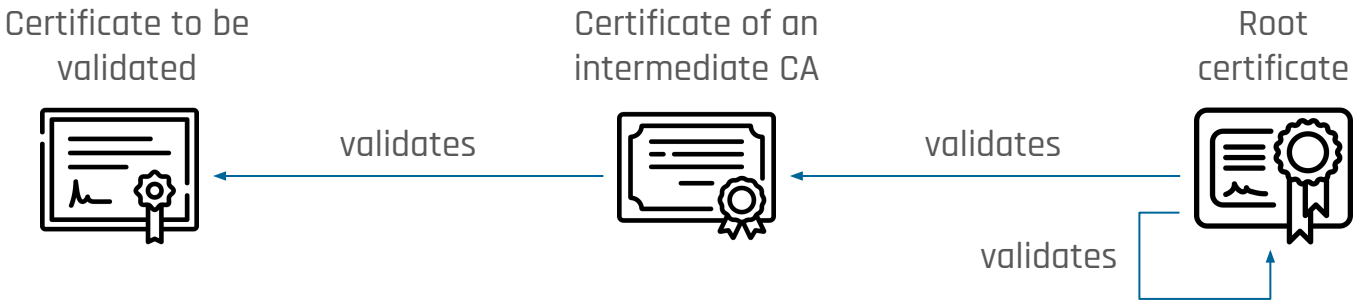
Self-signed and Root Certificates

- A **self-signed certificate** is a certificate in which the subject and the issuer are **the same** entity
  - The signature can be verified with the public key contained in the certificate itself
  - Easy to make, but they do not provide **any trust value**

- A **root certificate** is a self-signed certificate that identifies a **root CA**
  - These certificates are the **trust anchor** of the system
  - Operating systems and other applications using certificates (e.g., some web browsers) ship with **pre-installed root certificates** that are considered trustworthy

Certificate Chains

- A **certificate chain** is the list of certificates used to validate of a certificate:
  - The **first** certificate is the one we would like to validate
  - The validity of a certificate (except the last one) can be verified with the **public key** of the **next certificate** in the list
  - The last certificate is a **root certificate**
- Certificate chains can be arbitrarily long!



Public Key Infrastructures

Public Key Infrastructures

A **Public Key Infrastructure** (PKI) is a framework established to issue, maintain, distribute and revoke digital certificates

Registration authority

Revocation lists

Validation service

Certification authority

Directory service

Certificate policies

PKI - Registration Authority

- The **registration authority** is the entity of the PKI where persons and other subjects (e.g., subordinated CAs) can apply for certificates

- The registration authority verifies the **correctness** of the data contained in the **certificate request**
  - Depending on the required certificate this operation can be performed automatically or manually and can require a different amount of documents from the applicant
  - Upon successful verification, the information in the certificate request is forwarded to the certificate authority

Verification Process for TLS Certificates

- Different strategies corresponding to different trust levels:
  - **Domain Validated Certificates**: only the administrative control over the domain is verified
    - Example: a DNS TXT record for the requested domain must be published and must contain a token received from the registration authority
  - **Organization-Validated Certificates**: the identity of the company owning the domain is also verified
    - This information is included in the issued certificate
  - **Extended Validation Certificates**: compared to the previous class, the registration authority conducts more detailed checks about the company

PKI - Certificate Authority

- The **certificate authority** (CA) is the entity of the PKI that issues certificates for other subjects
  - Commercial CAs: DigiCert, Sectigo, GoDaddy
  - Non-profit CAs: Let's Encrypt
  - Government bodies and large organizations (e.g., Google, Amazon) also run their own CAs

- Other tasks of a CA:
  - Creating and managing **certificates** and (private) **keys** of the PKI to which the CA belongs
  - **Revocation** of certificates before their expiry date

PKI - Certificate Revocation Lists

- A **certificate revocation list** (CRL) contains certificates that have been revoked before their expiration
  - The corresponding private key was **lost** or **leaked**
  - The data contained in the certificate is not valid anymore (e.g., address of the subject has changed)
  - Revocation is **not reversible**!
- CRLs must be **periodically renewed** by the CA
  - Newly revoked certificates are added to the list and revoked, expired ones might be removed
- Disadvantages
  - Revoked certificates are **still accepted** until the list is updated
  - Over time, the size of CRLs can become rather **large** and updating them regularly is **bandwith-costly**

PKI - Validation and Directory Services

- To overcome the limits of CRLs, PKIs usually offer a **validation service** to query the revocation status of a certificate in real time
  - Implemented with standard protocols like **OCSP**
  - Introduces **connection latencies** when certificates are used to setup a TLS connection
  - Possible **privacy concerns**, since the usage of a particular certificate is revealed to another party (albeit trusted)

- PKIs generally offer a **directory service** to browse the certificates issued by them

PKI - Certificate Policies

- A **certificate policy** is a document that describes the different entities that compose a PKI, their roles, duties and working principles
  - Description of the PKI's architecture
  - Registration process and verification modalities
  - Key generation process
  - Implemented mechanisms to protect the PKI
  - Management of certificate revocation lists
  - Legal assurance

- Purpose of the document is to allow outsiders to analyze the **PKI's trustworthiness**

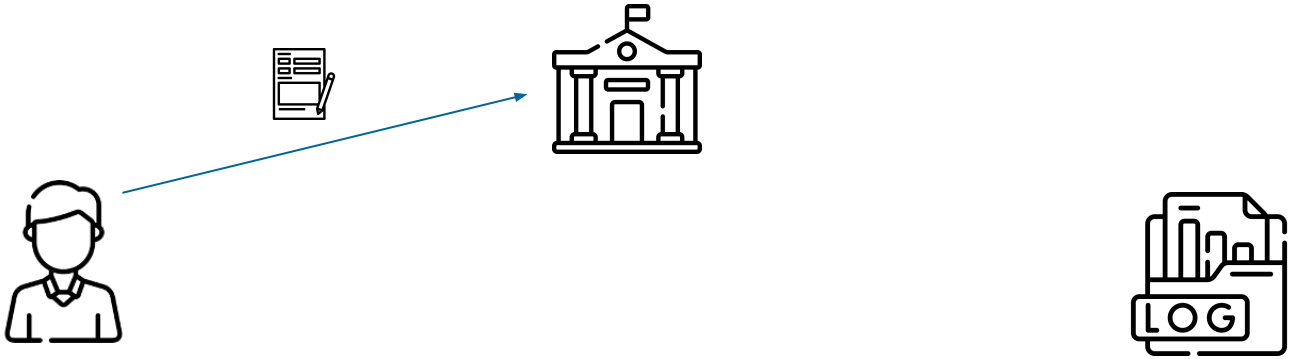
Certificate Transparency

False Issuance of Certificates

- CAs might issue false certificates **by mistake** or as a result of an **attack**
  - Diginotar got hacked and issued ~500 fake certificates that were used to monitor Iranian citizens (2011)
  - Symantec issued ~180 certificates without the domain's owner knowledge, including for `google.com` (2015)
  - Symantec issued over 100 certificates without proper validation (2018)

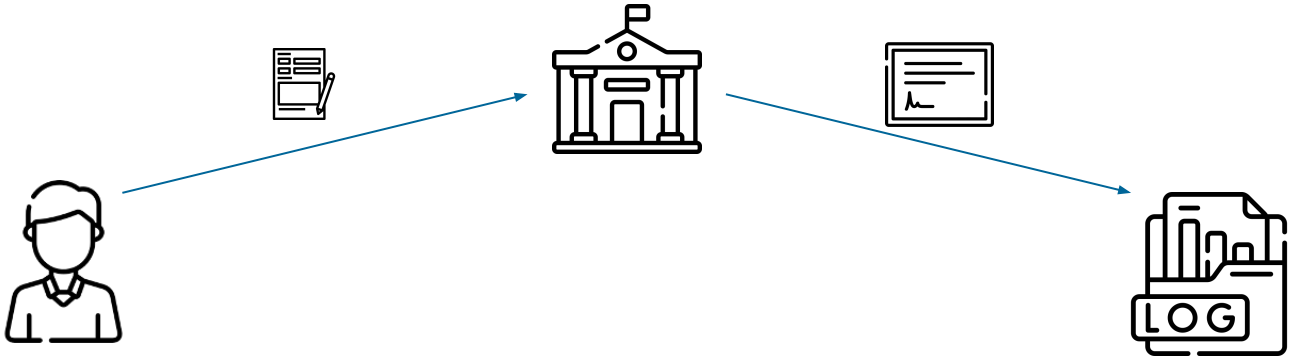
- **Certificate Transparency** (CT) is a standard proposed in 2012 to tackle this problem
  - All certificates issued by trusted CAs are published in **append-only**, **cryptographically protected**, **publicly auditable** logs to spot maliciously or mistakenly issued certificates

Certificate Transparency



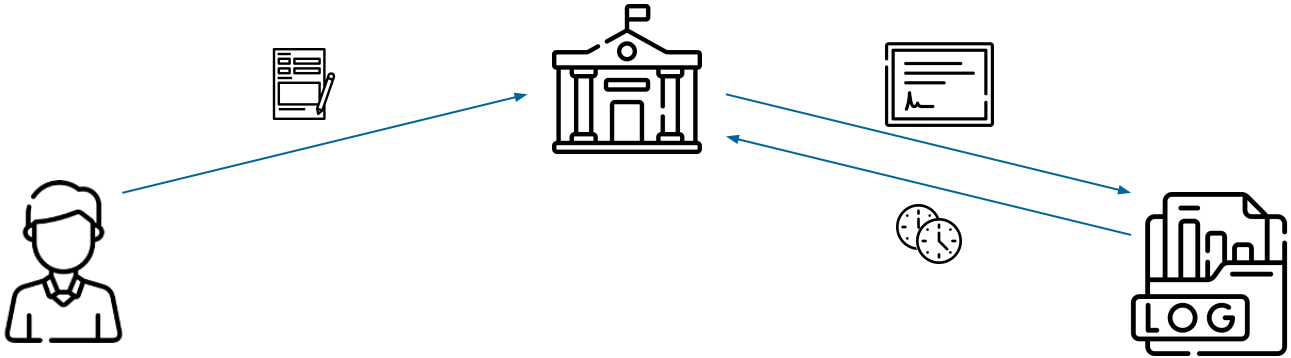
- The domain owner files a request to obtain a valid **certificate** for their domain

Certificate Transparency



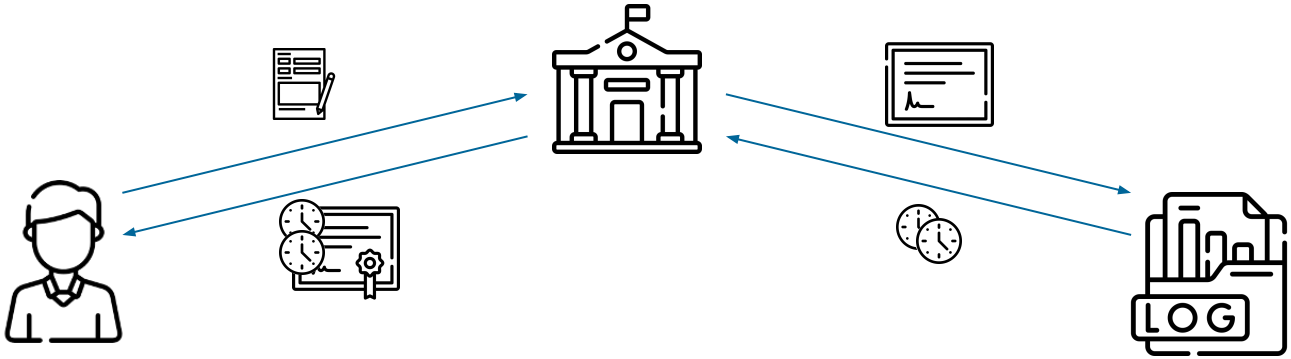
- The CA verifies that the owner is entitled to request the certificate and sends a **pre-certificate** to one or more logs
  - A pre-certificate contains all the information contained in a certificate plus a **poison extension** that prevents it from being accepted as valid

Certificate Transparency



- Each contacted log performs the following operations
  - Verifies the entire **certificate chain**
  - Sends a **signed certificate timestamp** to the CA, a promise to add the pre-certificate to the log within a certain **timeframe** (MMD)
  - Adds the pre-certificate to the log within the MMD

Certificate Transparency



- The CA issues a **certificate** containing the same information as in the pre-certificate and the **SCT** issued by the logs

Certificate Transparency

- Alternative implementations send the SCTs via a TLS extension or the OCSP protocol instead of embedding them into the certificate

- **Monitors** can be periodically run (by companies or individuals) on existing logs to:
  - Notify a website operator about the **issuance** of a new certificate for their domain
  - Check for certificates with **unusual** extensions or permissions, e.g., the ability to sign other certificates
  - Verify the consistency of a log and the correct inclusion of certificates within the log

Enforcement of Certificate Transparency

- Some browsers **display warnings** for **valid certificates** if certain constraints about Certificate Transparency are not satisifed

	SCTs embedded in certificate	**SCTs delivered via TLS extension / OCSP**
Chromium **Google Chrome**	• One SCT from a currently approved log • If the duration of the certificate does not exceed 180 days, the certificate contains at least 2 SCTs from distinct logs (currently or once-approved) • If the duration exceeds 180 days, at least 3 SCTs from distinct logs (currently or once-approved)	• One SCT from a current Google log • One SCT from a current non-Google log
Safari		• 2 SCTs from distinct, currently approved logs
Firefox	**No checks!**	**No checks!**

Cryptographic Protocols

Cryptography is not enough!

- The attacker can circumvent cryptography and break the security goals of the protocol by simply intercepting, duplicating, sending back the messages in transit on the network, without need to break the encryption scheme

- In the following, we assume that cryptography is a fully reliable black box and focus on how cryptography is used

Attacker threats

Even though attacks are often surprising and hard to predict, they can be roughly classified according to the kind of interaction between the attacker and the honest parties

Interleaving and reflection attack

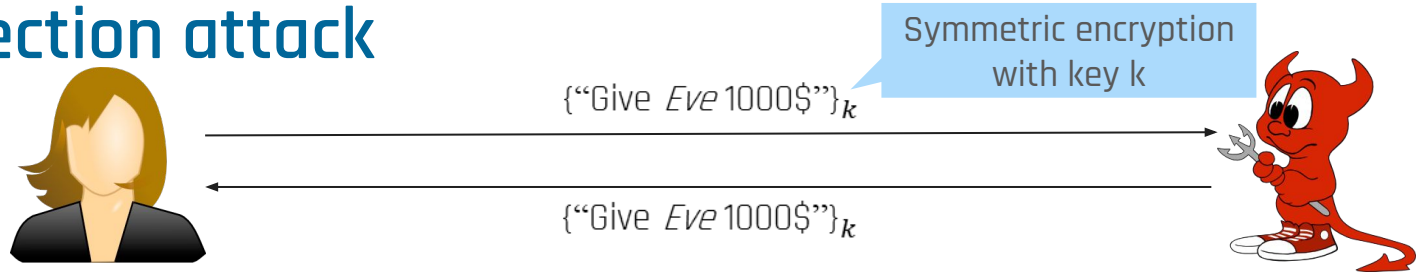
Interleaving attack

- a message generated in a protocol session is exploited by the attacker to interact with another simultaneously ongoing session

Reflection attack

- an interleaving attack where a message is sent back to its generator

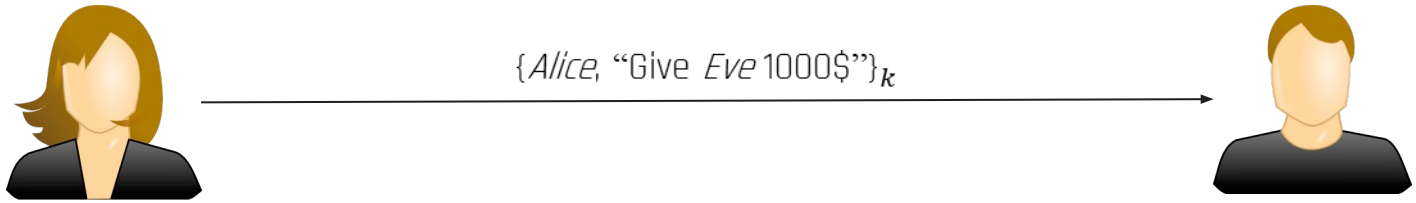
Reflection attack



- **Reflection attack**: an attack in which a message is sent back to its generator
  - works when the victim is playing multiple roles in the protocol, possibly in different sessions

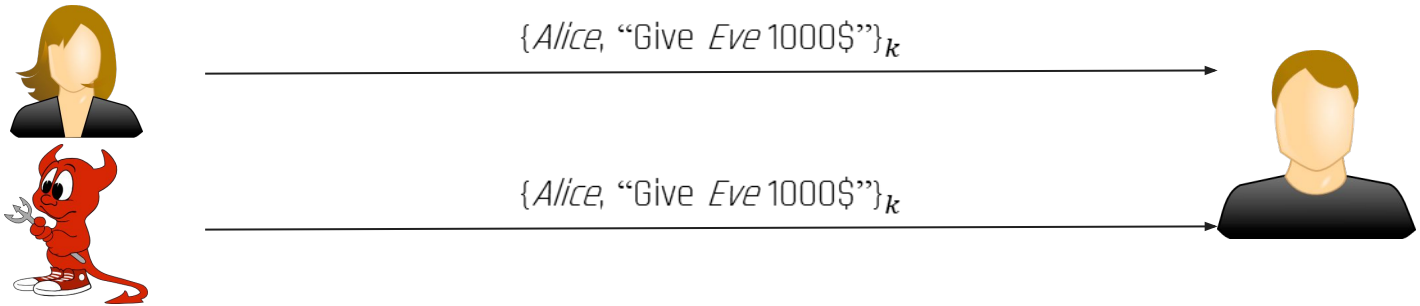
- The attacker intercepts the transfer request and sends it back to Alice, who recognizes the message as generated by Bob and performs the transfer
- The symmetric nature of the encryption key k does not allow Alice to verify whether the ciphertext has been generated by Bob or by herself

Avoiding reflection attacks



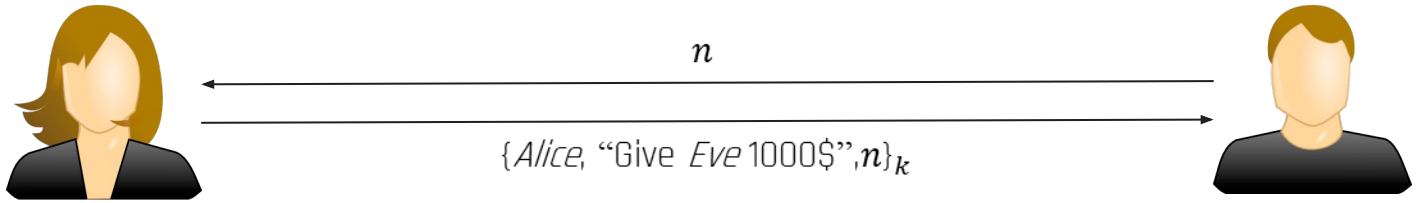
A solution is to break the symmetry of the cryptographic scheme by inserting the originator's identifier (or the intended receiver's one)

Replay attacks



The same message is duplicated and sent several times to the intended recipient

Avoiding replay attacks



A solution is to exploit a **challenge-response nonce handshake**

- A nonce is a **randomly generated number** n, used in a single protocol session and then discarded
- An authentication request is accepted only if no authentication request with the same nonce has previously been accepted

An Example

Transport Layer Security (TLS)

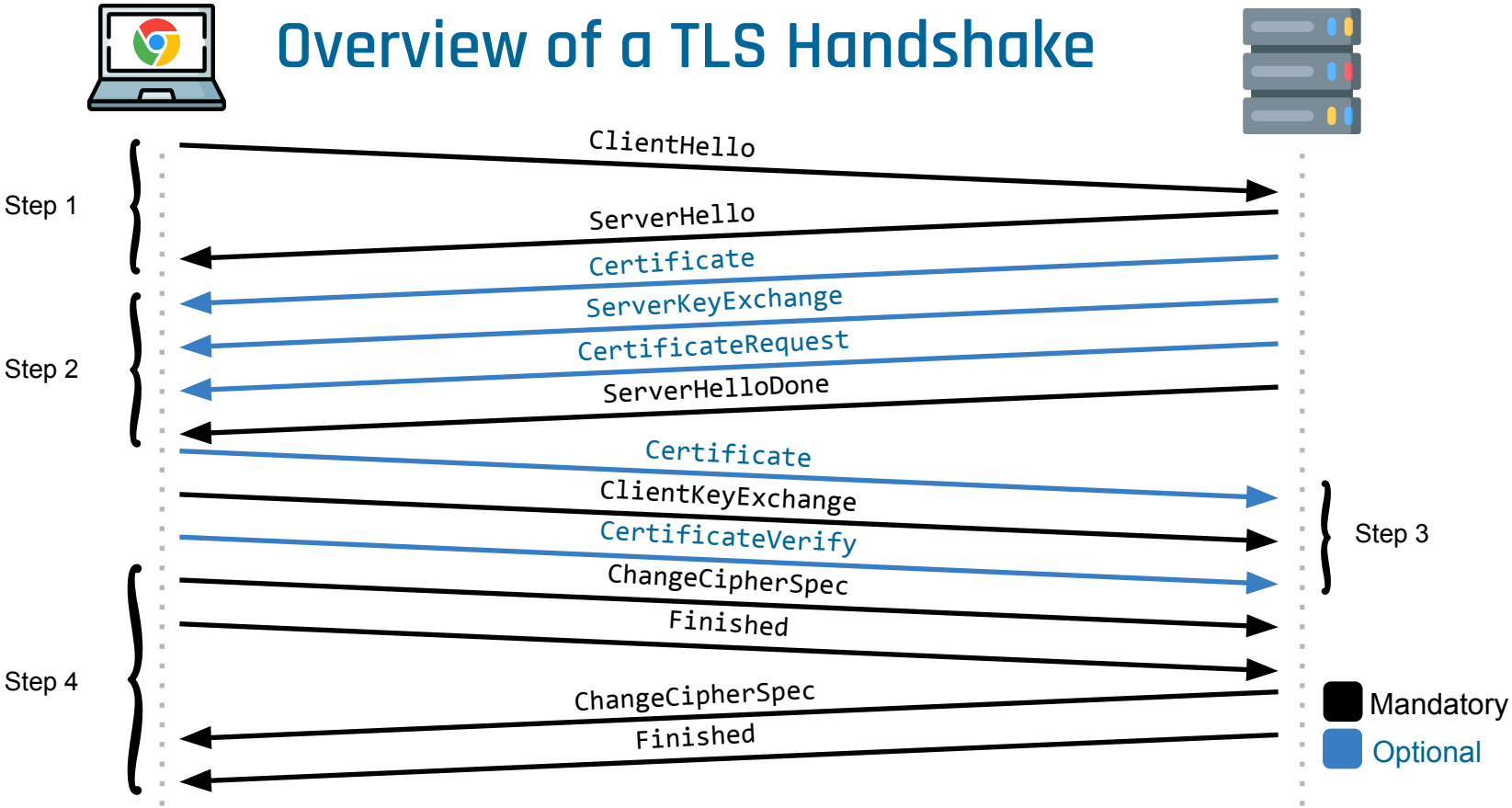
TLS Protocol

- The **TLS protocol** (Transport Layer Security) is the standard protocol for secure data transmission on the Internet
  - Previously known as **SSL** (Secure Sockets Layer)
  - Latest version: TLS v. 1.3 (2024, support 64.1%)
  - Most widespread: TLS v. 1.2 (2008, support 99.9%)
- **Security guarantees** provided by TLS:
  - **Confidentiality** of the transmitted data by using symmetric encryption algorithms
  - **Integrity** of the transmitted data using MACs
  - **Authentication** of the communication partners using asymmetric algorithms and certificates
    - Optional, but **almost always** enabled for servers
    - Optional and rarely used for clients (if used, server must be authenticated as well)

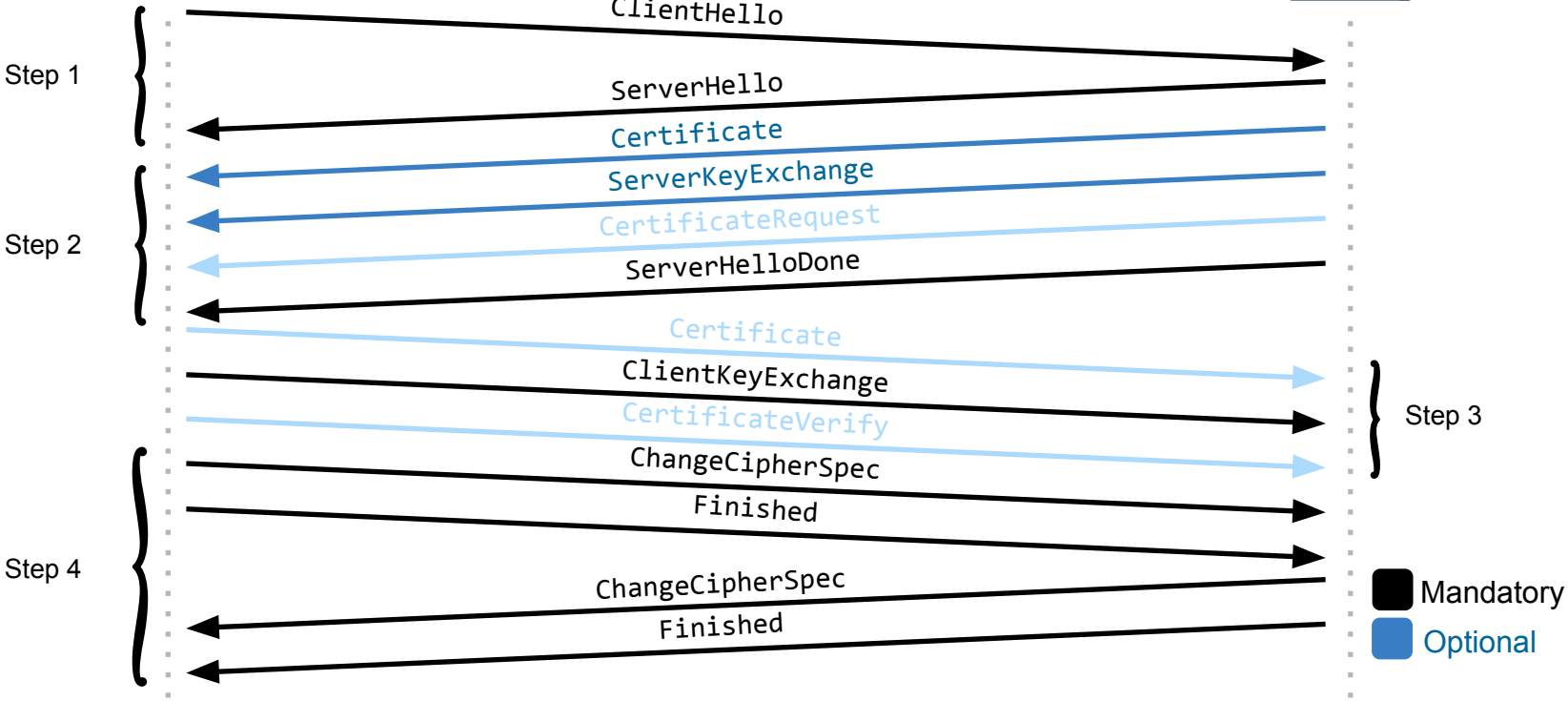
TLS Handshake (TLS v. 1.2)

- A TLS connection in version 1.2 is established in 4 steps:
  1. Negotiation of **security parameters**
  2. **Server authentication** and **key exchange**
  3. **Client authentication** (optional) and continuation of the **key exchange**
  4. Completion of the handshake
- After a successful handshake, the encrypted and integrity-protected data transmission takes place
- In the following, we discuss only the most relevant fields of the messages involved in the handshake
  - **Assumption**: a fresh session is started (resumption is also supported by TLS)

Overview of a TLS Handshake



Overview of a TLS Handshake



Step 1 - Negotiation of Security Parameters

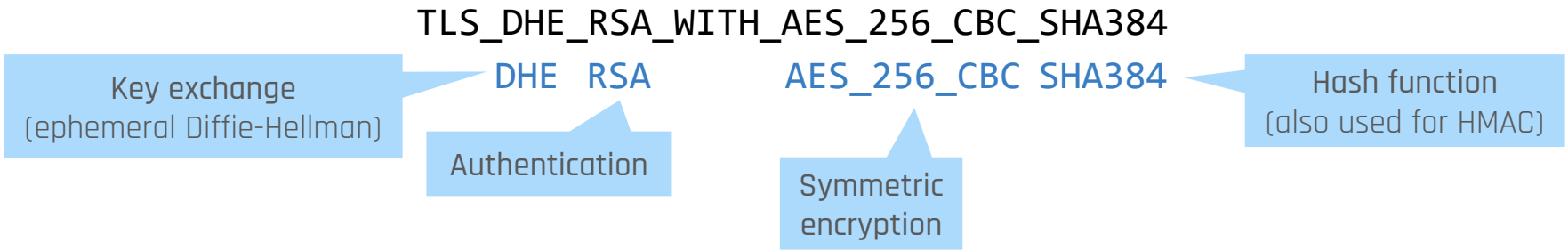
- `ClientHello` (Client ⟶ Server)
  - **Latest TLS version** supported by the client
  - List of supported **cipher suites** and compression methods
  - **Random byte sequence** for computing the master secret

- `ServerHello` (Server ⟶ Client)
  - Selected **TLS version** (here 1.2), **cipher suite** and compression methods to be used for transmission
  - **Random byte sequence** for computing the master secret

- Connection is terminated if:
  - the server **does not support** any of the TLS versions, cipher suites or compression methods suggested by the client
  - the server selects a cipher suite or TLS version **not supported** by the client

Cipher Suites

- A cipher suite defines a combination of 4 **cryptographic mechanisms**:
  - The **key exchange** algorithm
  - The **authentication** mechanism for clients and servers
  - The **symmetric encryption** algorithm used to encrypt data (together with key size and mode of operation)
  - A **hash function** used as basis for key-derivation and, in some cases, for the computation of a **HMAC** for **data integrity**



Forward Secrecy

- Some of the cipher suites supported by TLS 1.2 (and all those supported by TLS 1.3) provide **forward secrecy**
  - Session keys of past encrypted communications **cannot be compromised** even if the attacker obtains the private key of the server!
  - Typically implemented with an **ephemeral Diffie-Hellman key exchange**
    - For every connection, both client and server randomly generate a **fresh Diffie-Hellman private key** to perform the key exchange
    - The ephemeral Diffie-Hellman private key is **different** from the server's private key used for authentication (whose corresponding public key is stored in the certificate)

Step 2 - Key Exchange and Server Auth.

- `Certificate` (Server ⟶ Client, optional)
  - Sent if server authentication is enabled, contains the **server's certificate chain**
  - The certificate must be **appropriate** for the key exchange algorithm in the selected cipher suite:
    - `DHE_RSA`: the certificate contains a RSA public key and the key usage *`digitalSignature`* is permitted (we use RSA to sign)

- `ServerKeyExchange` (Server ⟶ Client, optional)
  - Sent only if **additional information** is needed to establish a premaster secret (e.g., when the ephemeral Diffie-Hellman key exchange is used, but not for Diffie-Hellman with static keys or RSA)
  - Content of the message in case of ephemeral Diffie-Hellman:
    - **Prime modulus** and **generator** g (chosen by the server)
    - Diffie-Hellman **public value** $g^{\text{🔑}} \bmod P$ (where the private value 🔑 $\in [2; P-2]$ is randomly generated)
    - If server authentication is enabled, a **signature** computed over the above values and the random sequences sent by client and server in step 1

Step 2 - Key Exchange and Server Auth.

- `ServerHelloDone` (Server → Client)
  - Denotes the end of the second phase
  - If server authentication is required, the validity of the certificate chain should now be verified

Validation of the Server's Certificate - Fundamental Steps

- For every certificate in the chain:
  - The **signature** can be verified with the public key contained in the **next certificate** of the chain (or in the same certificate for root certificates)
  - The certificate has not been used before/after its **validity period**
  - Possibly, it is checked that the certificate has not been **revoked**
    - Details depend on the client performing the verification
  - Each certificate (except the first) can be used to **sign** certificates
- For the first certificate of the chain:
  - One of the domains in the certificate **matches the domain** that the client wants to connect to

The certificate is valid if all the above operations are successful and the client trusts the root certificate of the chain

Step 3 - Key Exchange (and optionally Client Auth.)

- `ClientKeyExchange` (Client → Server)
  - Content depends on the selected key exchange mechanism
    - Ephemeral Diffie-Hellman: DH **public value** $g^{🔑💻} \mod P$ where the DH **private value** $🔑_{💻} \in [2; P-2]$ is chosen by the client and g, P come from the server (*ServerKeyExchange*, step 2)

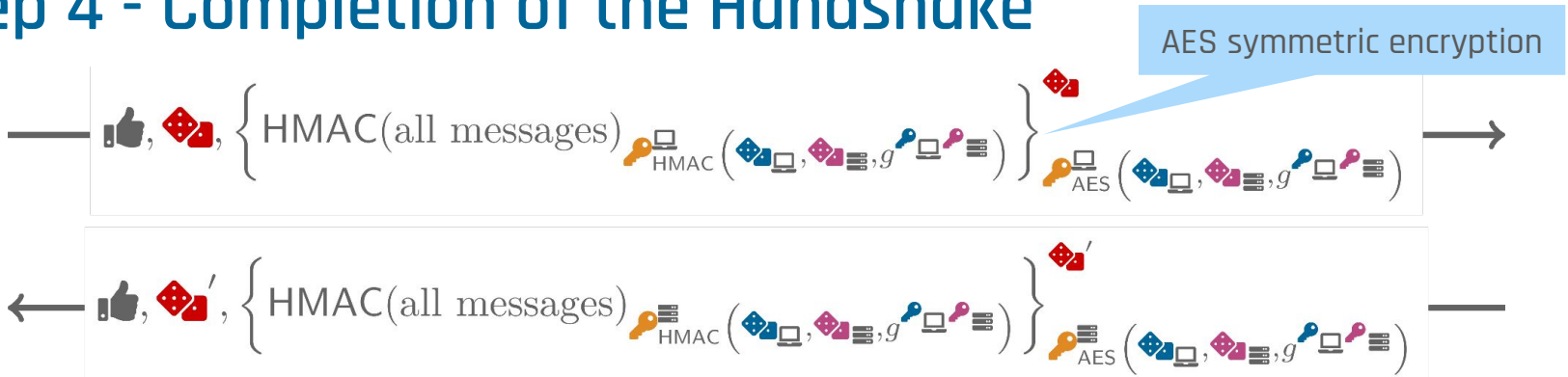
Computation of Premaster and Master Secret

- **Premaster secret** (depends on the key exchange) 
  - Diffie-Hellman: Server can use the client's public value and its own private value to compute the premaster secret (analogous for the client)

- **Master secret** 

  - Derived from the **premaster secret** and the **randomly generated byte sequences** of client and server (step 1)
  - All keys (encryption keys, keys for HMAC calculation, ...) are derived from the master secret

Step 4 - Completion of the Handshake



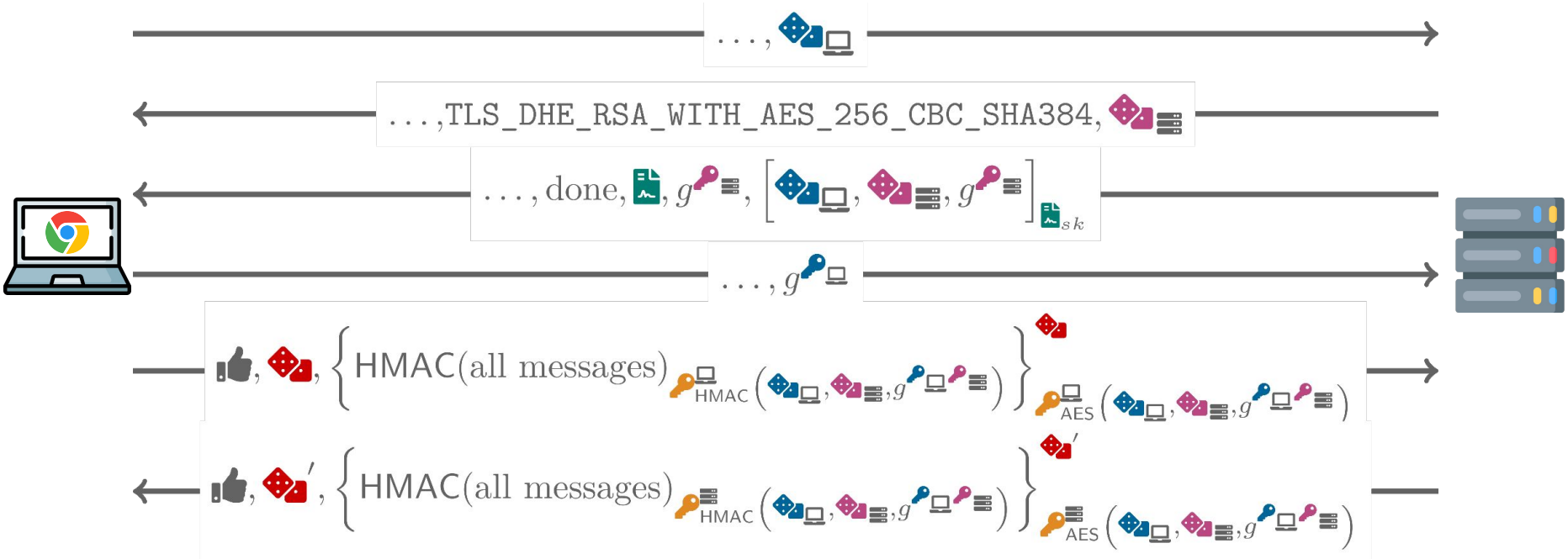
- First the client, then the server send the following messages
  - `ChangeCipherSpec`: communicates that the negotiated algorithms and parameters will be used from now on
  - `Finished`: first message encrypted with the negotiated symmetric algorithm
    - Contains the HMAC computed over all previous handshake messages
    - In case of decryption failures HMAC mismatch, the connection is terminated
    - The keys for the encryption and the HMAC is computed from the **Master Secret** (i.e., from the DH secrets and the nonces)

Summary - The symbols

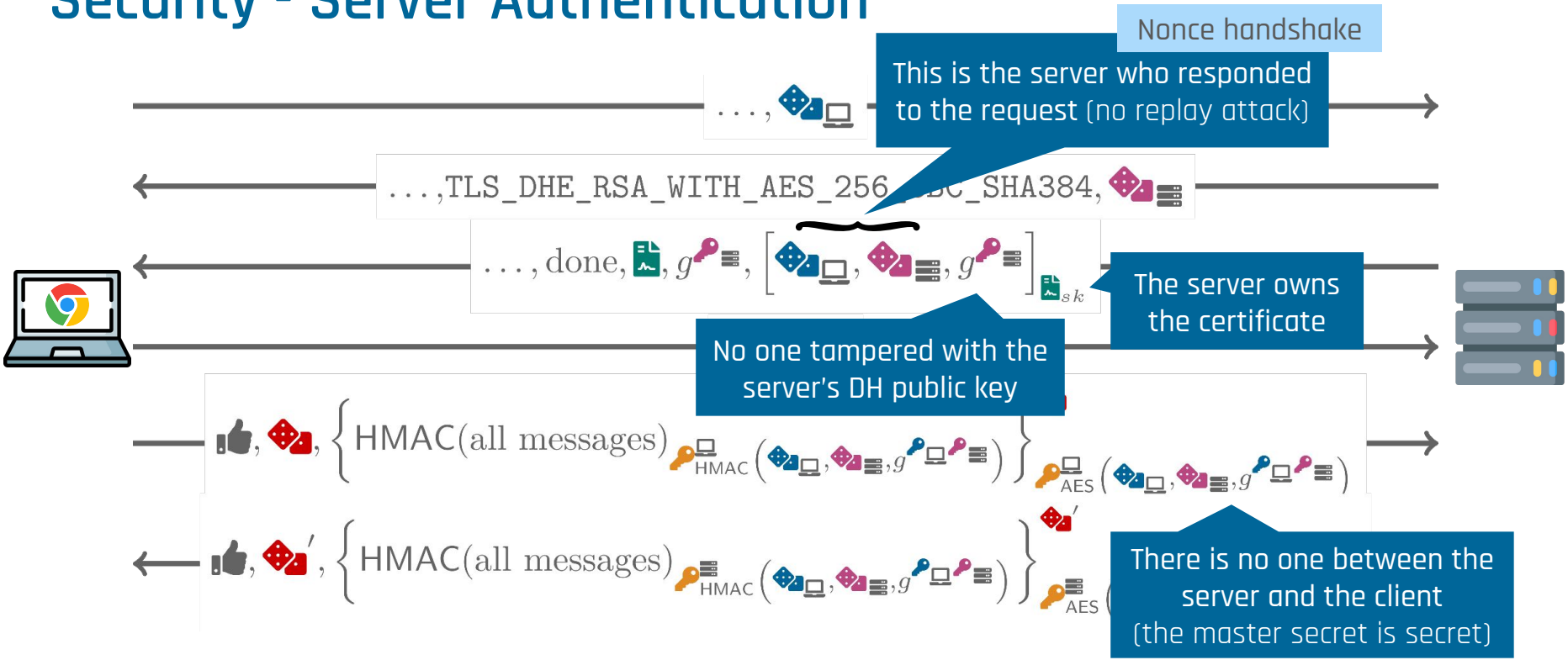
-  : A **random** value
-  : A **secret** value (e.g., a secret key)
-  : A **certificate**
  -  : The (secret) **signing key** of the certificate

- $[m]_k$: **Signature** of m with the key k (in our example, this is an RSA signature)
- $\{m\}_k^r$: Randomized **symmetric encryption** of m with key k and seed r (in our example, this is **AES256** where r is the initial IV of the CBC mode)
- g^{ab} : **Diffie-Hellman common secret** generated with the secrets a and b

Summary - The protocol



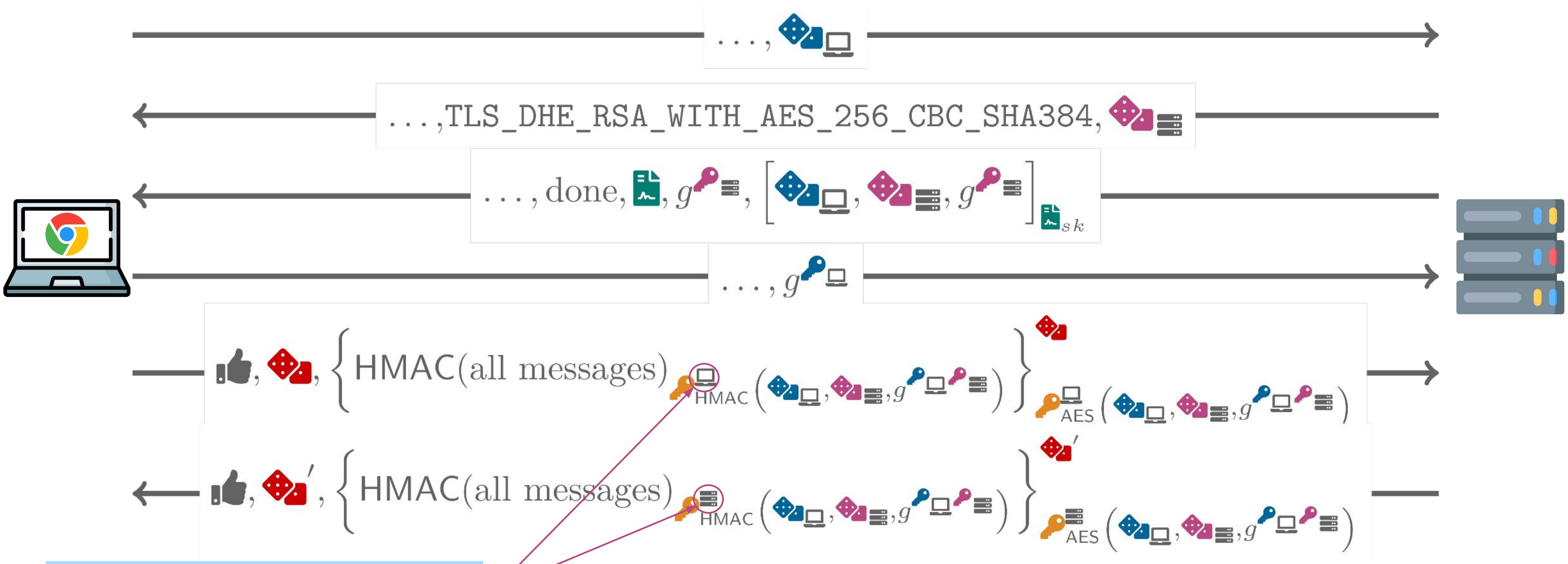
Security - Server Authentication



Security - Questions

Who is the client?

The “owner” of



What about reflection attacks?

The client and the server use different symmetric keys