Certificates, Public Key Infrastructures, and ~~SSL~~/TLS

for infrastructure builders and operators

David Ochel, August 2019

2019-08-31-00

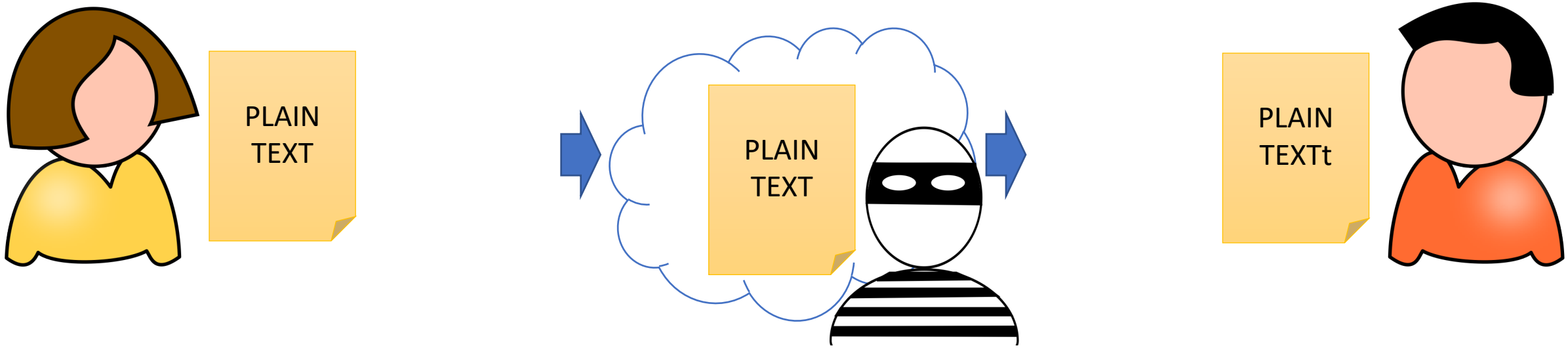
Agenda & Outline

Objective: Align on the practical and security implications of managing infrastructure components that rely on TLS and (server) certificates for enabling transport security.

- Applied crypto basics – 20’
  - Symmetric & asymmetric encryption
  - Hashes and digital signatures
  - What makes crypto hard
- Public Key Infrastructures – 40’
  - Certificates & CAs – and how they are used in TLS
  - Lifecycle considerations for server keys and certificates
- ~~SSL/~~TLS – Configuration Gotchas – 30‘

Thwarting the “Man in the Middle”

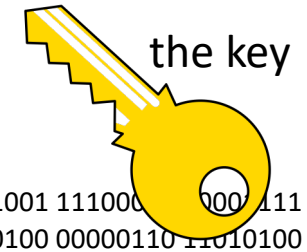
- Communication over (public) networks is not confidential, by design
  - Emails, web traffic, secrets, etc. can be intercepted and/or manipulated
  - Originators of content can be impersonated
- Unless it is protected and authenticated, typically using encryption
  - TLS (as in, HTTPS, FTPS)
  - IPsec, OpenVPN, MACsec, … (or application-specific, like PGP and SSH)



Crypto basics

(because it’s the foundation for everything)

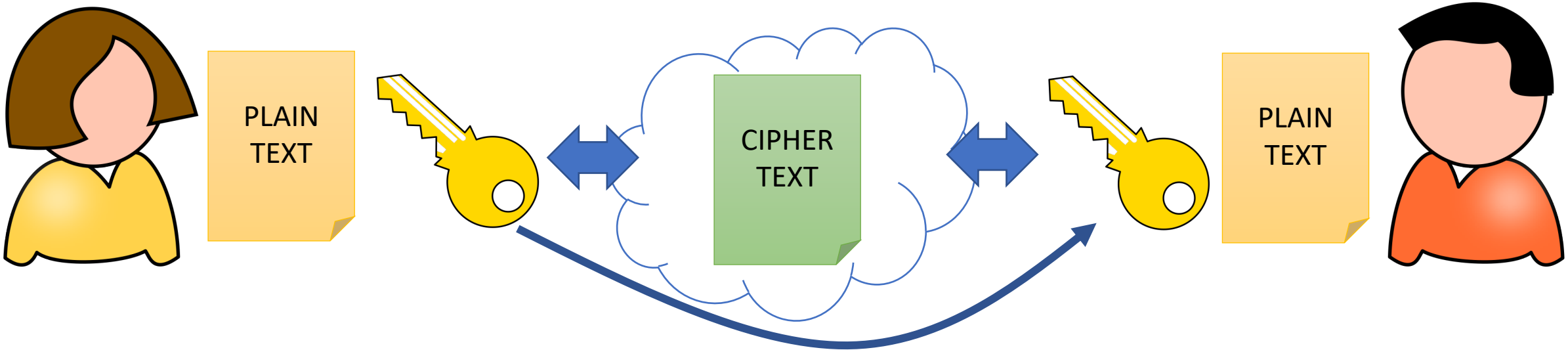
Secret Key – Symmetric Encryption



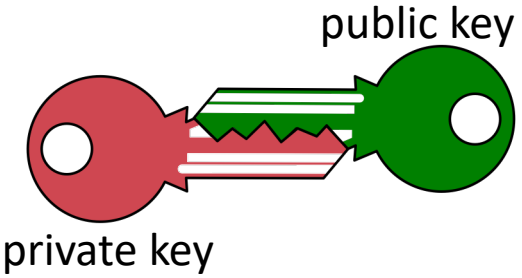
01101001 11100[illegible]000111 10010101
10110100 00000110 11010100 10010011
11001111 10110000 00001010 11110000
10100011 00100001 10001011 01001111

(it's actually a random string of bits)

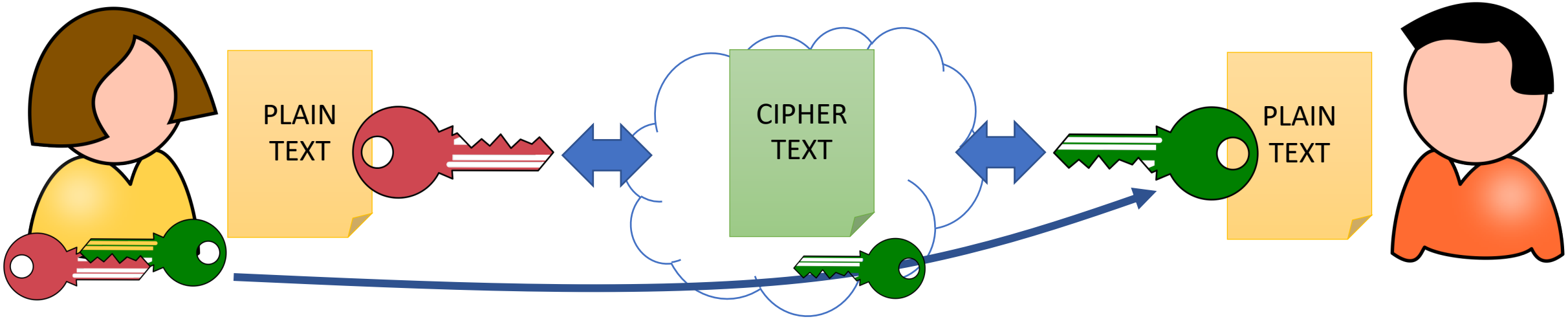
- One key, multiple copies
- Good: fast, straightforward
- Bad: key distribution (requires out-of-band, can't control copies)
- Examples: AES, Blowfish – e.g., 256 bit key length



Public and private keys – Asymmetric Encryption

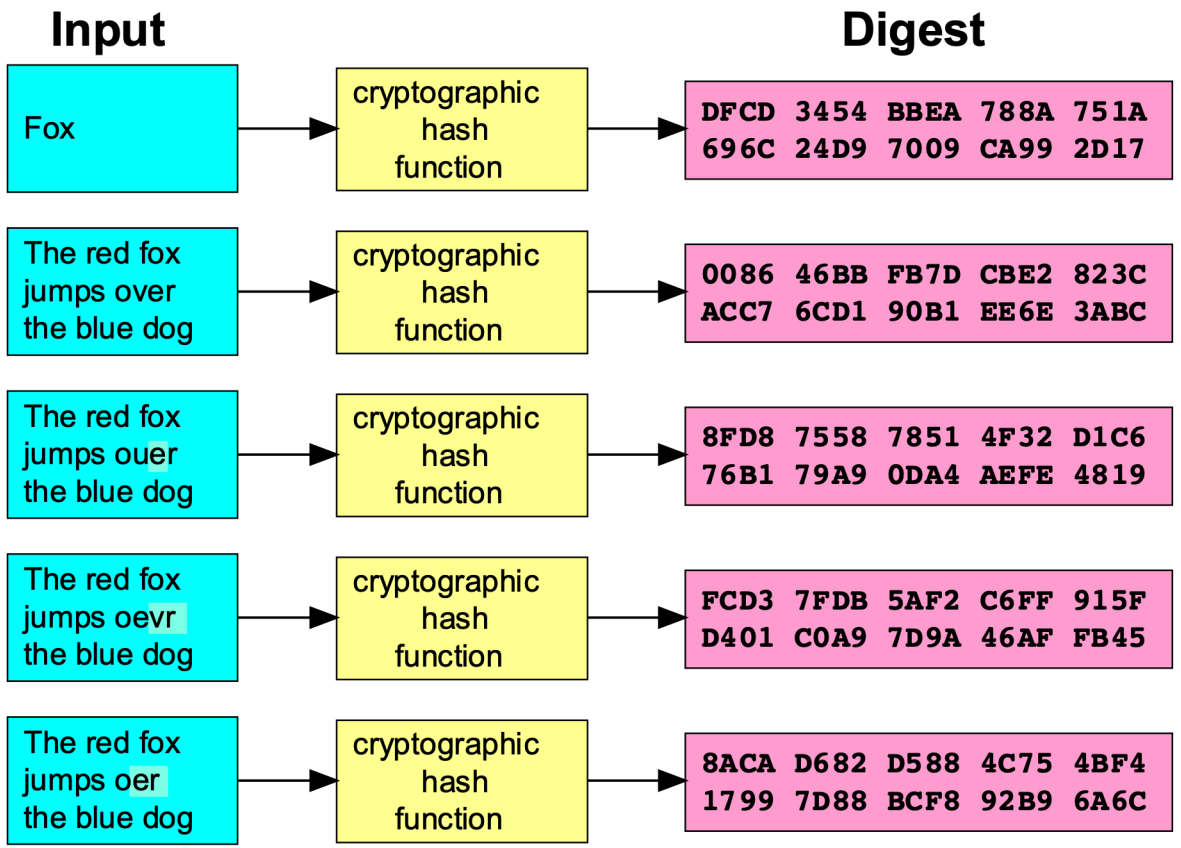


- Mutually complementary pair of keys
- Good: distribution (private key stays with owner, copies of the matching public key can be distributed freely); digital signatures
- Bad: high CPU need, based on (theoretically solvable) math problems
- Examples: RSA, Diffie Hellman – e.g., 2048 bit key length

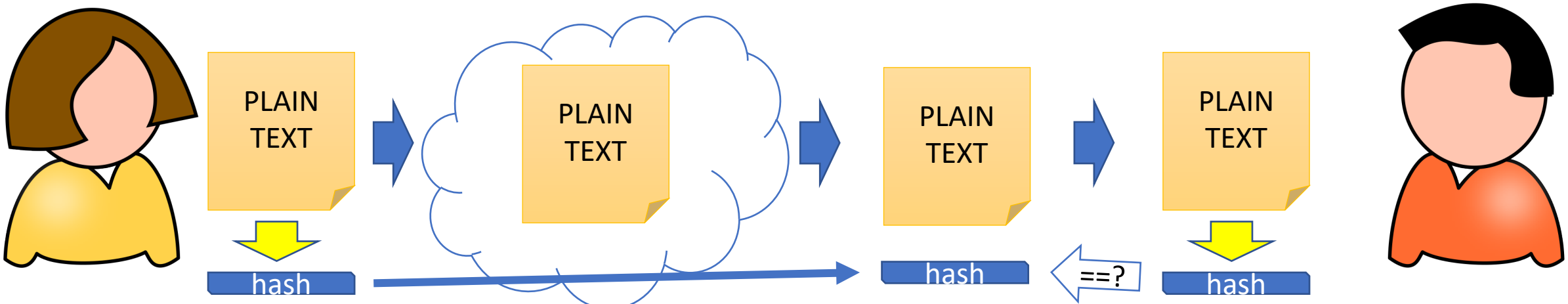


Cryptographic Hashes

- Maps a message into a fixed-size digest (“hash”)
  - Deterministic
  - Collision-resistant
  - Non-reversible (one way), cannot compute message from hash
- Examples: bcrypt, SHA-2

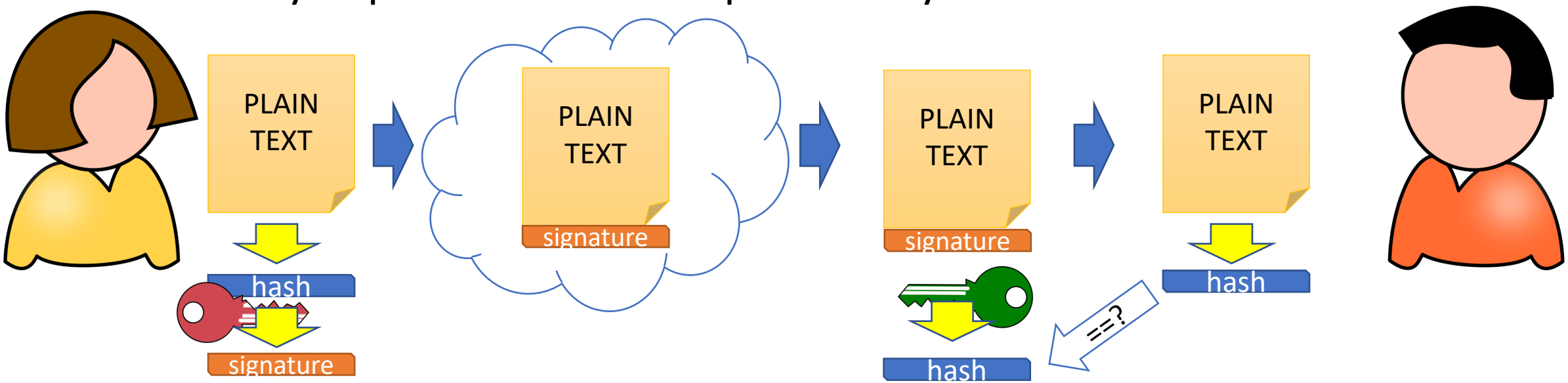


From https://en.wikipedia.org/wiki/File:Cryptographic_Hash_Function.svg



Digital Signatures

- Originator encrypts hash with private key (“digital signature”)
- Recipient decrypts hash with originator’s public key, and computes its own hash over the message
- If the two hashes match, the message must have been signed by somebody in possession of the private key



Typical Crypto Limitations

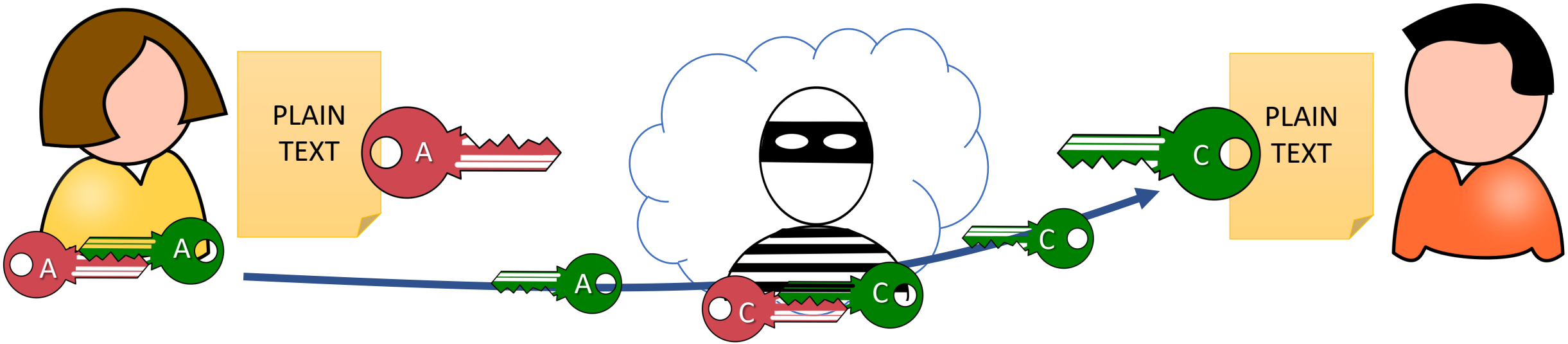
- Implementing crypto algorithms is hard
  - Use approved libraries/products
- Key material “weakens” (risk of eventual compromise increases) over time
  - Rotate keys on defined frequencies
  - Use recommended key lengths and algorithms
- Protecting access to secret and private keys is hard
  - Re-key if somebody with access to a secret key leaves organization
  - Keep keys in as few locations as possible, defend them thoroughly
  - Don’t re-use the same key for different purposes or in different trust settings
- Distributing public keys in a trustworthy manner is still hard
  - Use public key infrastructures!

Public Key Infrastructures

(or, how do I know that Alice's public key that I grabbed from the web is the one that in fact matches Alice's private key, and not Chuck's, who is pretending to be Alice?)

Remember the Man in the Middle?

- We want to distribute public keys freely, over networks
- But is Alice in fact the one that possesses the private key to the public key that has her name on it?
  - Can I trust that digital signatures signed with the corresponding private key are in fact Alice's signatures, and not somebody who's pretending to be her?
  - Can messages I encrypt with this key only be decrypted by her?



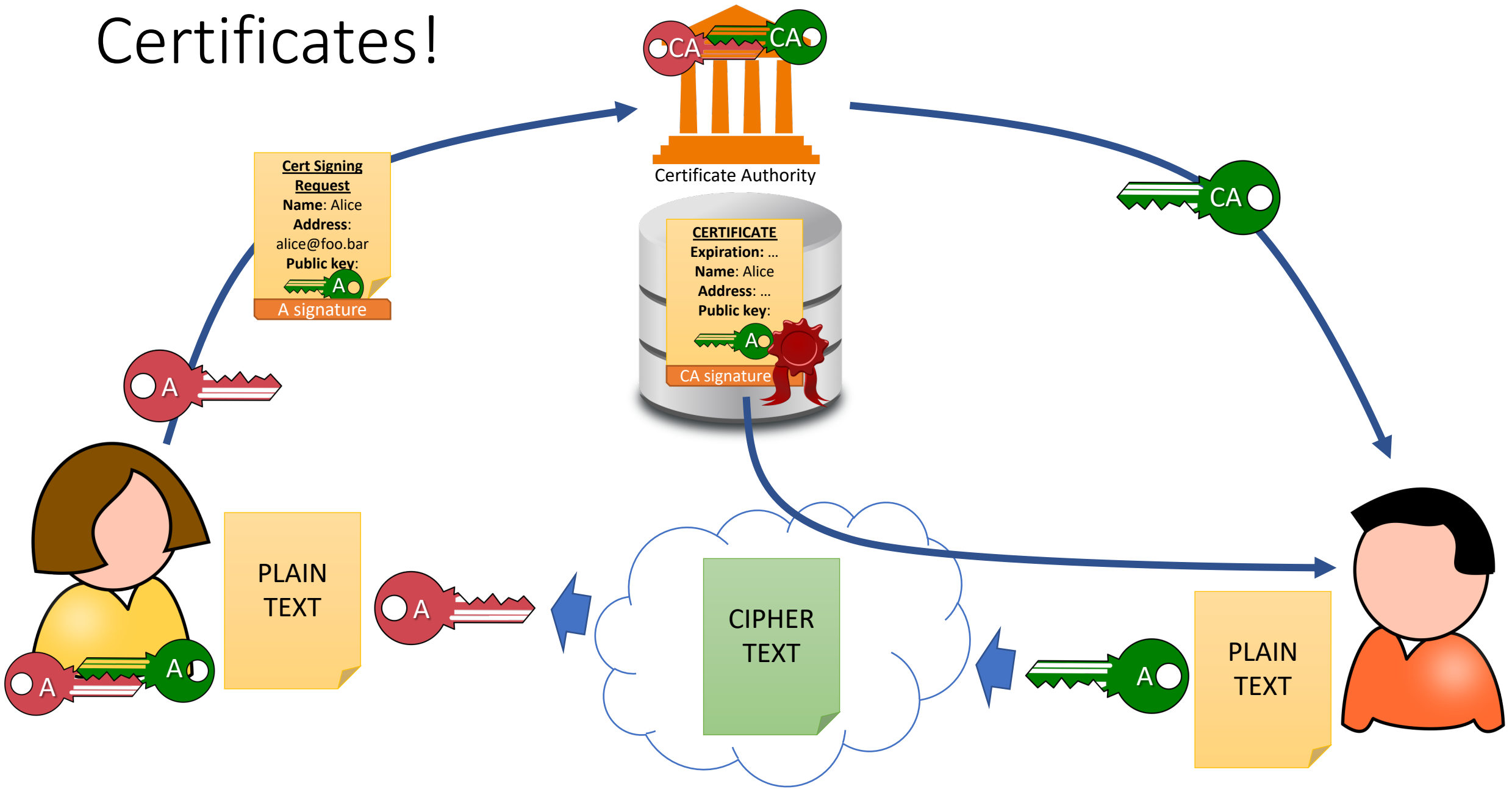
Attesting to a distributed key's authenticity

https://imgs.xkcd.com/comics/responsible_behavior.png

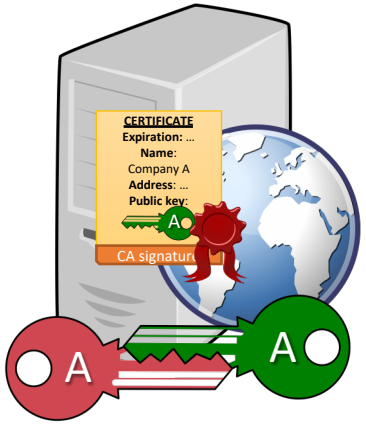
- Crowd-sourced web of trust (e.g., PGP)
  - Users attest to the authenticity of public keys by signing each other's public keys (after having established that they have the same key in front of them)
  - Key Signing Party, anyone?

- Trusted Third Parties
  - Certificate authorities establish a hierarchical chain of trust into public keys
  - All users trust the same CA, thus any public key signed by the CA is extended trust (without the need for a secure channel to obtain individual public keys)

Certificates!

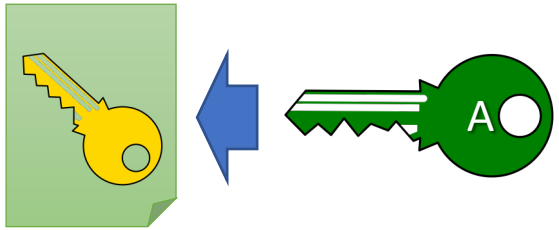


Transport Layer Security – crypto (simplified)



1: Client requests server certificate, validates it against CA cert

2: Client generates symmetric session key*



3: Client & server exchange traffic encrypted with the session key

*This is a much simplified view of how key generation with RSA works. Authentication of the server with RSA is implicit. DH/ECDH works differently.

Lifecycle of a server certificate

1. Public/private keypair management
2. Certificate request
3. Certificate issuance
4. Certificate use and revocation
5. Certificate consumption by clients

(1) Managing server public/private keys

- Private keys should never leave the server (need to keep confidential)
  - Use Certificate Signing Requests instead of letting the CA generate the server's public/private key pair
  - Do not share private keys across trust domains / between different services (lest compromise of the weakest link will compromise them all)
- Beware of aging keys
  - Limit validity of certificates to an appropriate number of days or years
  - Generate new keys when "renewing" a cert instead of getting a new cert for the same old key
  - Consider new keys & certs if an operator with access leaves the organization
- Self-signed certificates

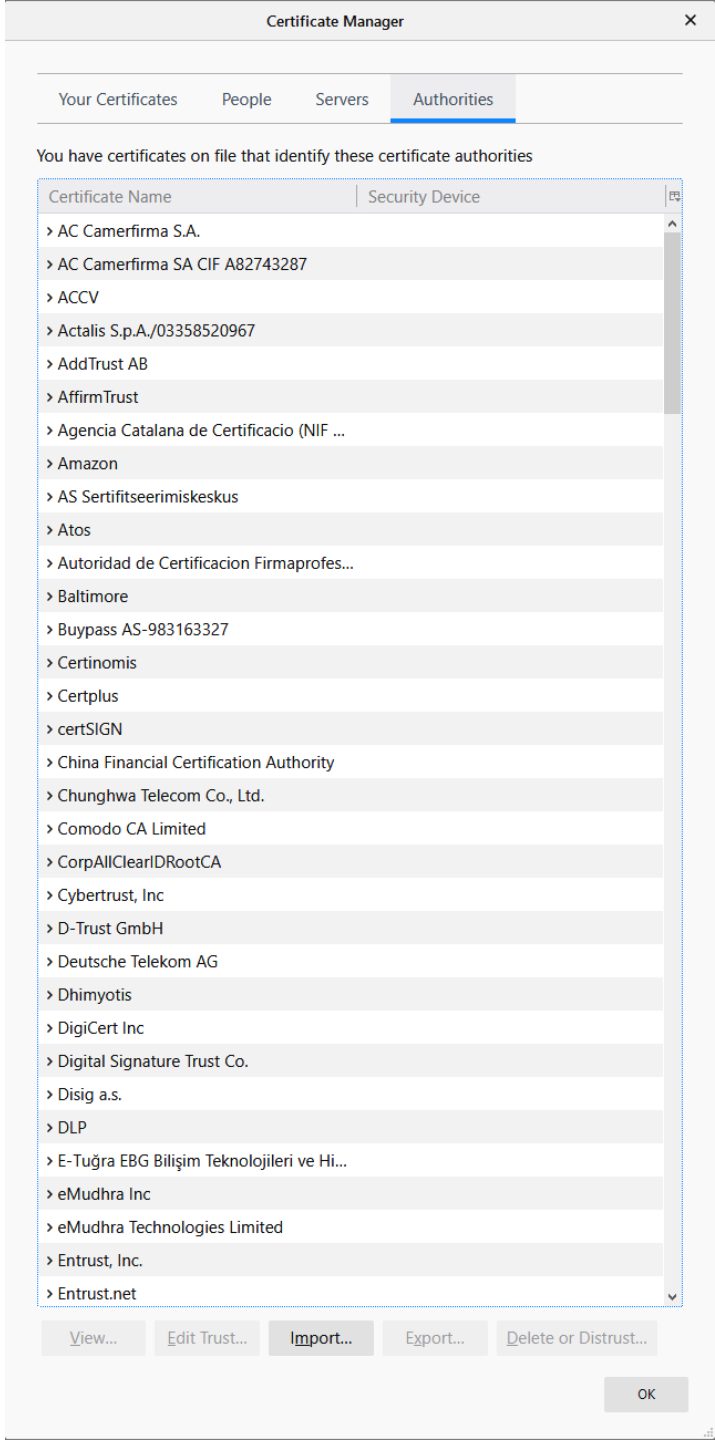
(2) Requesting a certificate

- Prefer FQDN subject names over wildcards
  - Wild cards encourage proliferation of the same keys across different services / trust domains
  - Consider multiple Subject Alternative Names instead

- Choose your CA wisely
  - Consider reputation of different CAs, even though browsers might trust them all
  - Consider validation and support needs (e.g., *Let's Encrypt* – domain validation only)

- Identity Verification
  - Registration Authorities – (in-person) offices of the CA for high-trust situations (e.g., validating somebody's passport in person - not common in TLS server scenarios
  - Domain Validation – does the requesting organization control the domain that the cert is for?
  - Organization Validation – can the requesting organization prove its name/DBA/address as listed in the CSR?
  - Extended Validation – subject name matches organization's incorporation doc

- Certificate Signing Request (CSR) – standardized format to request a certificate
  - Does not include the user's|server's private key, but is signed by its private key to demonstrate possession of it

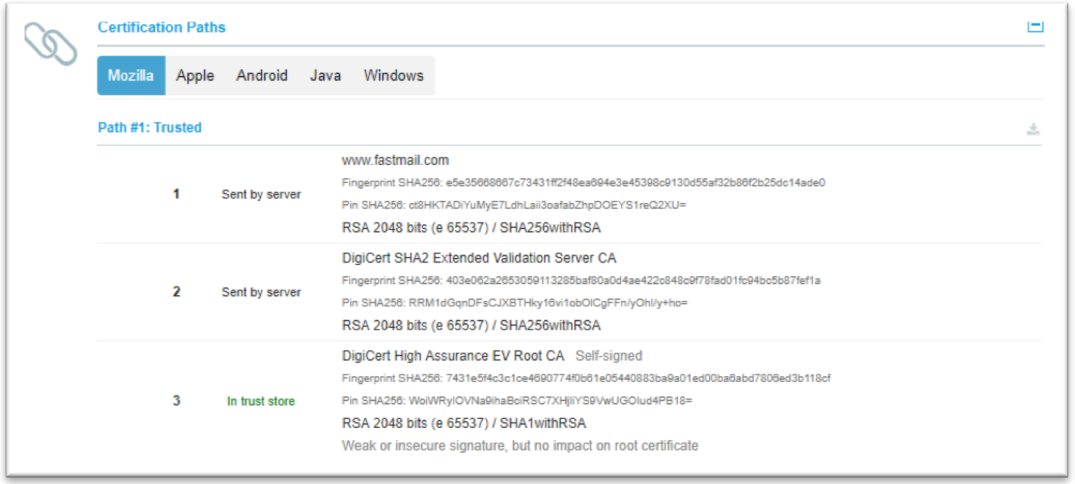


(3) Issuing a certificate

- Do not use self-signed certs
  - Teaching users to click through warnings in their browsers desensitizes them against actual threats
  - Internal CAs for internal services are OK *if* they are managed properly
- Mitigate risk of CA compromise
  - The root CA should be highly protected, offline, and only used to issue certs to certificate-issuing CAs lower down in the hierarchy
  - Consider using different cert-issuing CAs for different cert purposes / environments
- Automation is OK…
  - ...if cert requests are properly authenticated & authorized

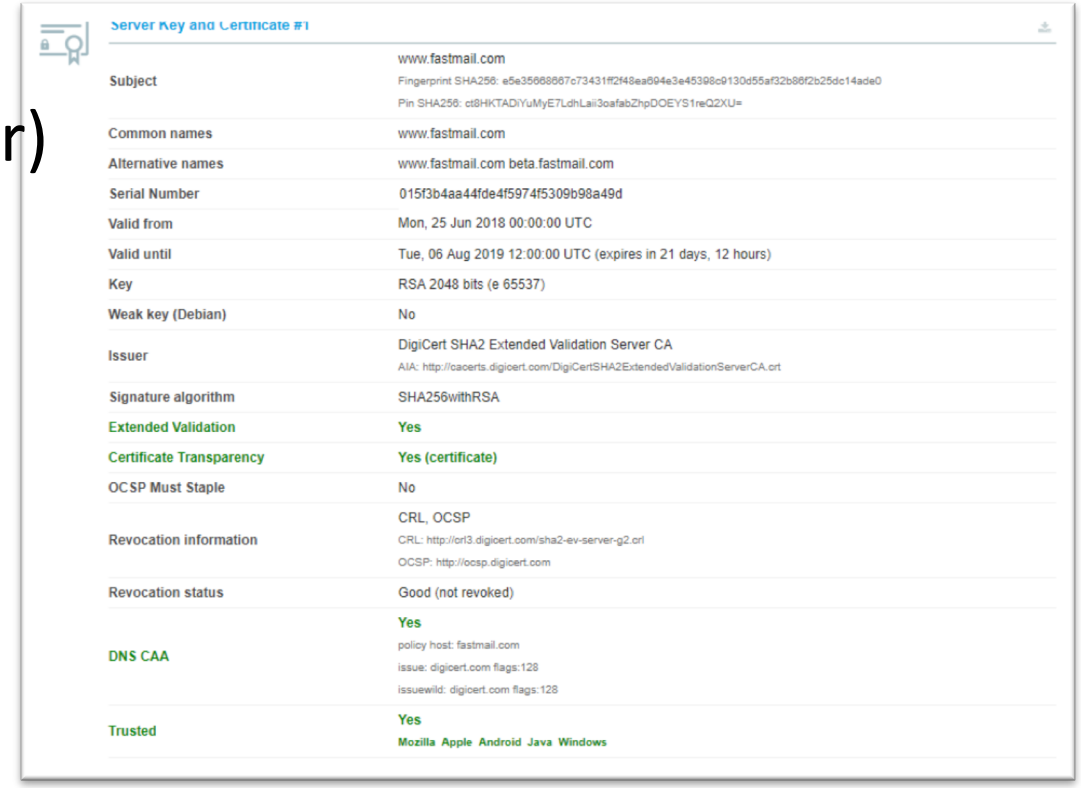
(4) Hosting and revoking certificates

- Serve certificate chains with your certificate
  - Clients may break if intermediate certs are not served by your server, since they may only have the root CA certs in their trust stores
- Monitor for certificate expiration
- Revoke certificates for compromised or retired keys



Certificates – the devil is in the details

- Expiration Date
- Purpose (domain validation, cert issuer)
- Subject Alternative Names
- Wildcard Certs vs. FQDN
- Key Usage



Other uses of certificates & trust chains

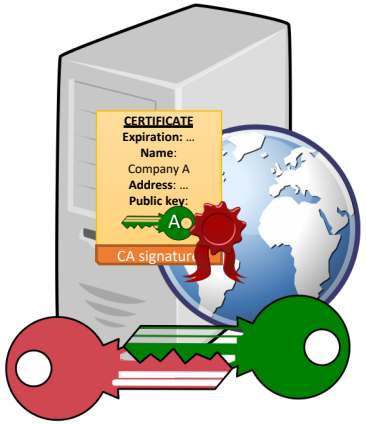
- Code signing
- Windows updates
- Europe: electronic signatures
- Device authentication
- User authentication
- S/MIME
- DNSSEC
- PGP Web of Trust

Using certificates for authentication

- Authentication factors: what you have; what you know; what you are
  - Certs (i.e., private keys!) are “what you know”
    - Same issue as passwords – can be cloned onto multiple devices, picked up by malware, …
  - Can be used to implement “what you have” if protected from cloning
    - E.g., on a smart card, HSM, secure enclave
- Do certs authenticate users or devices?
  - Certs don’t authenticate anything, it’s the private key that corresponds to the public key in the cert that does the authentication
  - Arguably, the CA’s digital signature on the certificate provides for authenticity of the certificate’s content
- Do keys (SSH keys, keys related to device certs, …) authenticate users or devices?
  - It depends – on usage, protection of the keys, context, …

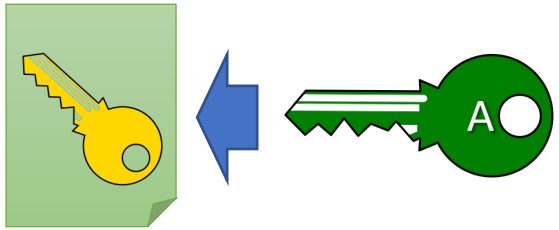
~~SSL/~~TLS – Configuration Gotchas

Transport Layer Security – crypto (simplified)



1: Client requests server certificate, validates it against CA cert

2: Client generates symmetric session key*



3: Client & server exchange traffic encrypted with the session key

*This is a much simplified view of how key generation with RSA works. Authentication of the server with RSA is implicit. DH/ECDH works differently.

Versions

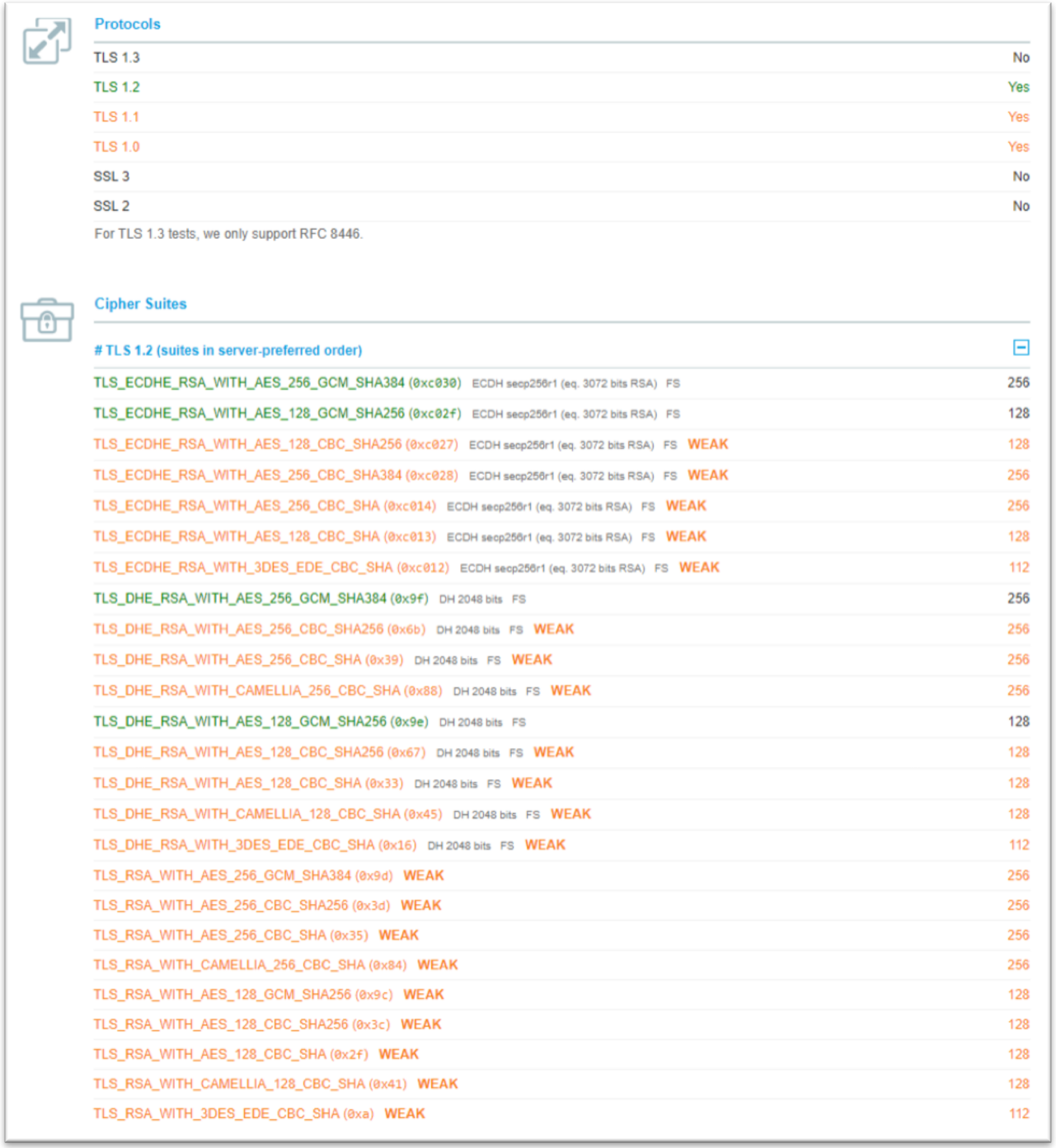
- SSL is outdated and insecure
  - ~~v2 – 1994~~ (Netscape Navigator 1.1 ☺)
  - ~~v3 – 1995~~
- TLS
  - ~~1.0 – 1999~~ (RFC 2246), similar to SSLv3
  - 1.1 – 2006, uses MD5 and SHA-1 (on it's way out)
  - 1.2 – 2008
  - 1.3 – 2018

Cipher Suites

- Determine the mix and match of crypto algorithms a client and server can agree on for:
  - Authentication
  - Key exchange
  - Encryption algorithm & key sizes
  - Etc.

➢Disable weak cipher suites

➢Prefer them in order of strength



Protocols

TLS 1.3	No
TLS 1.2	Yes
TLS 1.1	Yes
TLS 1.0	Yes
SSL 3	No
SSL 2	No

For TLS 1.3 tests, we only support RFC 8446.

Cipher Suites

 # TLS 1.2 (suites in server-preferred order)

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) ECDH secp256r1 (eq. 3072 bits RSA) FS	256
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f) ECDH secp256r1 (eq. 3072 bits RSA) FS	128
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027) ECDH secp256r1 (eq. 3072 bits RSA) FS WEAK	128
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028) ECDH secp256r1 (eq. 3072 bits RSA) FS WEAK	256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) ECDH secp256r1 (eq. 3072 bits RSA) FS WEAK	256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) ECDH secp256r1 (eq. 3072 bits RSA) FS WEAK	128
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012) ECDH secp256r1 (eq. 3072 bits RSA) FS WEAK	112
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x9f) DH 2048 bits FS	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x6b) DH 2048 bits FS WEAK	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x39) DH 2048 bits FS WEAK	256
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x88) DH 2048 bits FS WEAK	256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x9e) DH 2048 bits FS	128
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256 (0x67) DH 2048 bits FS WEAK	128
TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x33) DH 2048 bits FS WEAK	128
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) DH 2048 bits FS WEAK	128
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) DH 2048 bits FS WEAK	112
TLS_RSA_WITH_AES_256_GCM_SHA384 (0x9d) WEAK	256
TLS_RSA_WITH_AES_256_CBC_SHA256 (0x3d) WEAK	256
TLS_RSA_WITH_AES_256_CBC_SHA (0x35) WEAK	256
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) WEAK	256
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c) WEAK	128
TLS_RSA_WITH_AES_128_CBC_SHA256 (0x3c) WEAK	128
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f) WEAK	128
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) WEAK	128
TLS_RSA_WITH_3DES_EDE_CBC_SHA (0xa) WEAK	112

Other Tidbits

- Perfect Forward Secrecy
  - Session keys don’t depend on protection by the server’s private key
  - A result of using DH/ECDH (instead of RSA) for key agreement
  - If RSA is not supported by a web server, MITM traffic inspection will break ;-)
- Various other, security-relevant configuration settings. For example:
  - Prevent protocol version downgrade attacks (TLS_FALLBACK_SCSV)
  - Reject insecure renegotiation (support the Renegotiation Indication extension, RFC 5746)
  - Tell clients to always use TLS (Strict Transport Security (HSTS))
  - Only use the TLS NULL compression method
  - Support OCSP stapling (RFC 6066)
  - Support session resumption

Potential Client Issues

- Mismatch of supported protocol versions
- Mismatch of supported ciphersuites
- Certificate Pinning
  - Will interfere with MITM attempts to de- and re-encrypt traffic (e.g., for inspection)

Fun Tools

- Dealing with certs
  - OpenSSL
- Checking TLS configs
  - sslyze
  - www.ssllabs.com
  - observatory.mozilla.org
- References
  - https://www.ssllabs.com/projects/documentation/