




Node HTTPS

Understanding HTTPS/TLS

HTTPS

- Hypertext transfer protocol (Secure)
- Node implements HTTPS client and server
- On top of TLS (Transport Layer Security)

Encryption

- Encryptions are two types
- Symmetric -> You encrypt with key and decrypt with the same key
 - One key 
 - Fast but both client and server must have the same key
- Asymmetric -> You encrypt with a key and decrypt with another
 - Comes in pairs Two keys Private  and Public 
 - Slower but both client and server can have their own public keys
- We always want to encrypt with Symmetric encryptions
- Exchange the symmetric key with asymmetric encryption

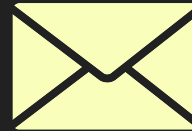
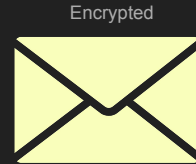
Symmetric Encryption

- Assume both parties have the same key (The most difficult thing)
- User uses the key to encrypt message
- Send it



Symmetric Encryption

- Receiver gets the encrypted message
- Uses the same key to decrypt
- E.g. AES



+

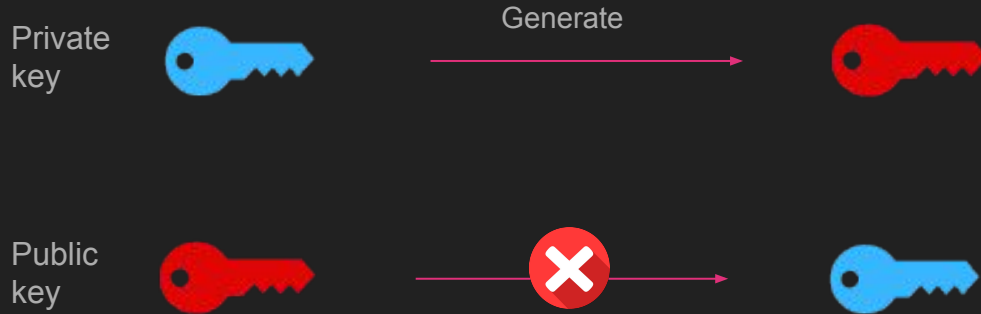


=



Public Key vs Private Key Rules

- Public key private keys are pairs (e.g. Red public, Blue Private)
- Given the Private Key you can generate the Public key
- Given the Public Key you cannot get the Private key



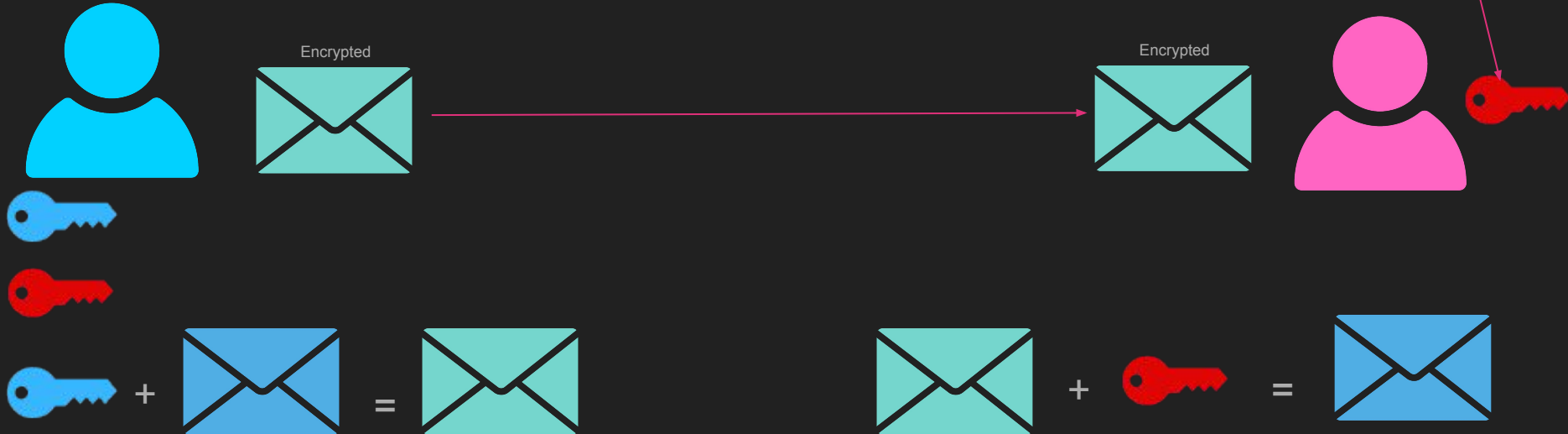
Encrypting with the Public Key

- You can encrypt a message with Public Key
- And only owner of Private key can decrypt it
- Proves Authenticity



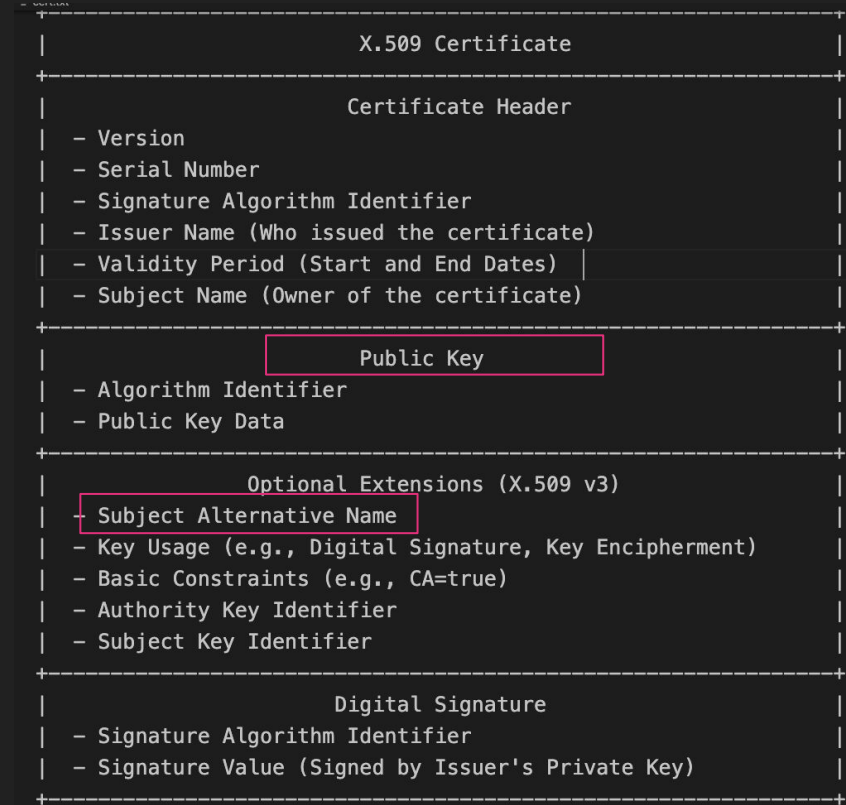
Encrypting with Private Key

- You can encrypt a message with the Private key
 - and only the corresponding Public Key can decrypt it
 - Only owner of the private key could have signed this document
 - Protects confidentiality, nobody could have missed with it



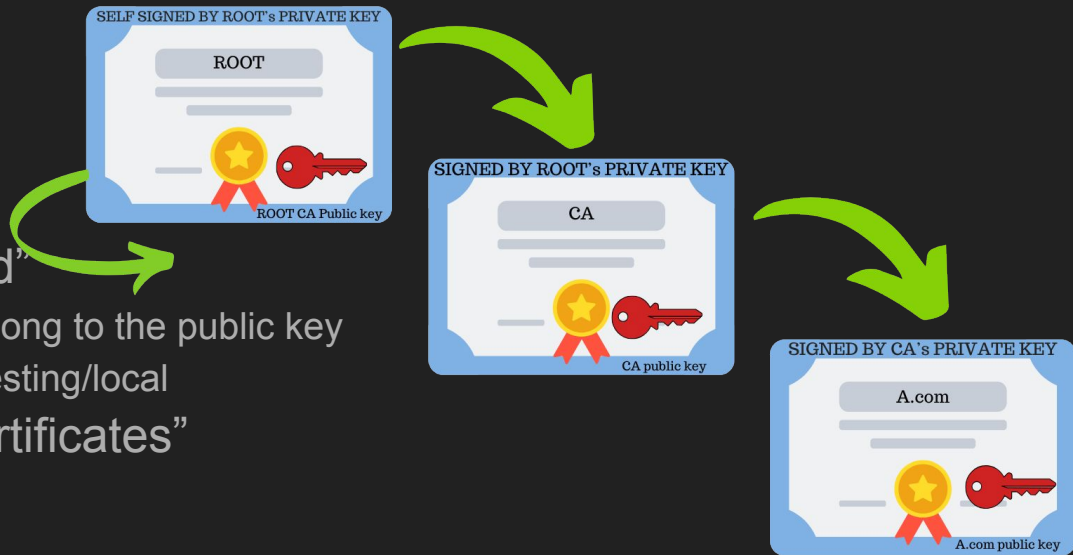
Certificates

- We need a way to proof authenticity
- Generate a pair of public/private key
- Put a public key in a certificate
- Put the website name in the certificate
- Sign the certificate with the private key
- Meet x509

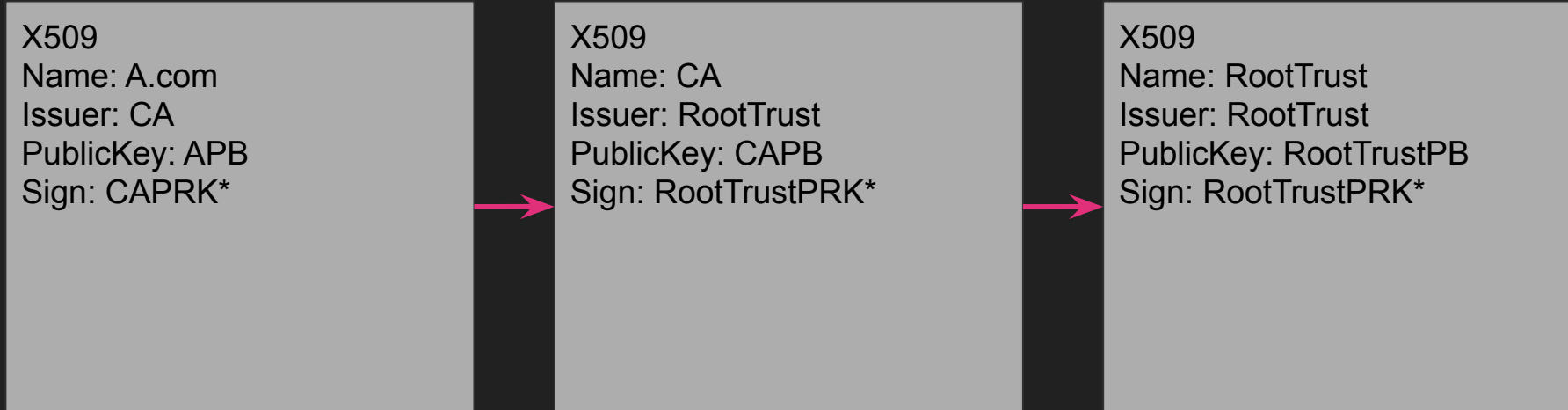


Certificates

- Certificates can be “self signed”
 - The private key signing the cert belongs to the public key
 - Usually untrusted and used for testing/local
- Certificates can sign “other certificates”
 - Creating a trust chain
 - Issuer name is who issued it
 - Lets encrypt
- Ultimately a ROOT cert is found
 - ROOT certs are always self signed
 - They are trusted by everyone
 - Installed with OS root (certificate store)



Certificate Verification



Client receives the full chain, wants to verify A.com cert signature which as been signed by CA public key issuer, so it gets the CA and gets the CAPUB to verify, but also it needs to trust the CA cert so it, verifies that by getting the RootTrust public key and verifies it, but the RootTrust is self signed so it looks up its local cert store. If it is there it is trusted. Else rejected.


TLS


- Transport Layer security
- Encrypt using the same key on both client and server
- For that we need to exchange the key
- We use public key encryption to exchange key
- We share certificate for authentication

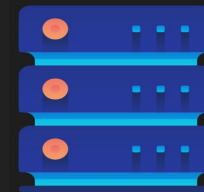
TLS 1.2 (RSA)

open



RSA Public key 

RSA Private key 



Client hello

Server hello (cert)

Change cipher, fin

Change cipher, fin

GET /

Headers+
index.html

<html>...
....

close

Problems with that approach

- Encrypting the symmetric key with public key is simple
- But its not perfectly forward
- Attacker can record all encrypted communications
- If the server private key is leaked (heart bleed)
- They can go back and decrypt everything
- We need ephemeral keys! Meet Diffie Hellman

Diffie Hellman

- Let us not share the symmetric key at all
- Let us only share parameters enough to generate it
- Each party generate the same key
- Party one generates X number (private)
 - Also generates g and n (public, random and prime)
- Party two generates Y number (private)

Public g, n

Private X



Private Y



Diffie Hellman

- Party 1 sends g and n to Party 2
- Anyone can sniff those values fine.
- Now both has g and n

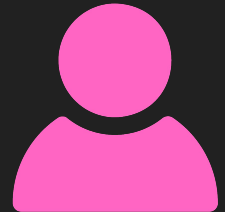
Public g, n

Private X



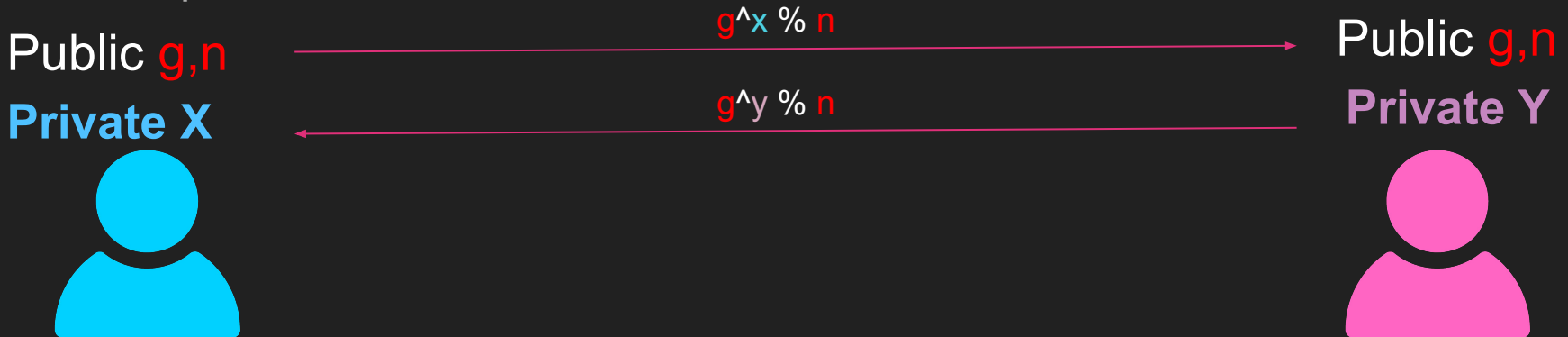
Public g, n

Private Y



Diffie Hellman

- Party 1 takes g to the power of $X \% n$
 - $g^X \% n$ is now a public value
 - Cannot be broken to get X !
- Party 2 does the same with Y
 - $g^Y \% n$ is now a public value
 - Cannot be broken to get Y
- Both parties share the new values



Diffie Hellman

- Party 1 takes Y's value and raise it to X
 - $(g^y \% n)^x = g^{xy} \% n$
- Party 2 takes X's value and raise it to Y
 - $(g^x \% n)^y = g^{xy} \% n$
- Both now has the same value $g^{xy} \% n$
- This is the used as a seed for the key $g^{xy} \% n$

Public g, n
Private X



$$(g^y \% n)^x = g^{xy} \% n$$

$$g^x \% n$$

$$g^y \% n$$

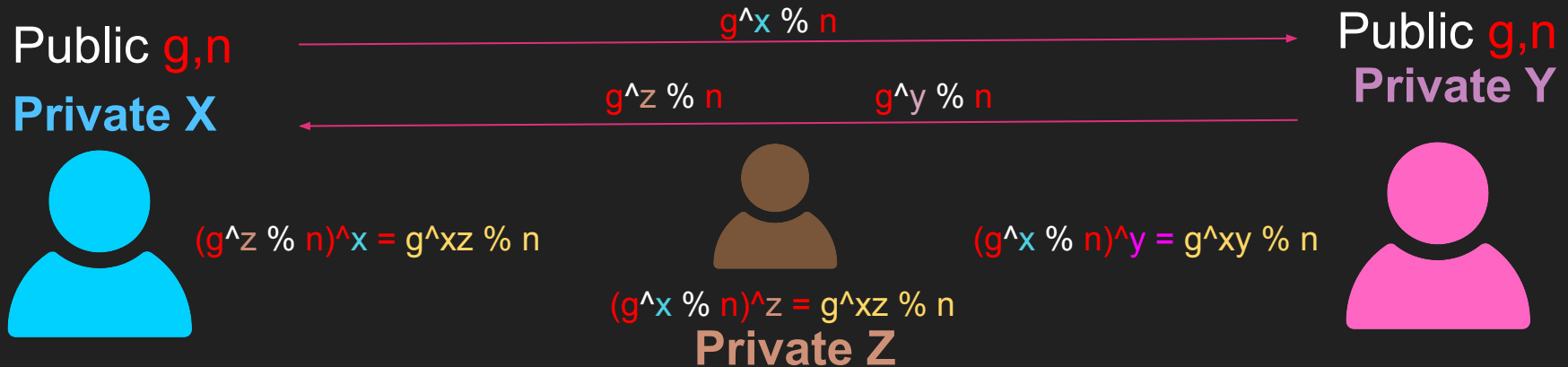
Public g, n
Private Y



$$(g^x \% n)^y = g^{xy} \% n$$

More problems! MITM

- This solves perfect secrecy
- But what if someone intercepts and put their own DH keys
- MITM replace Y's parameter with their own
- X doesn't know that happened (it's just numbers)



Solved with signing

- We bring back public key encryption
- But only to sign the entire DH message
- With certificates

TLS1.3

open

X

g, n

$(g^y \% n)$



$(g^y \% n)^x = g^{xy} \% n$



Client verifies the message is indeed signed by Y by using public key in certificate

close

g, n

$(g^x \% n)$

Entire message
Signed with private
key! (we don't send
PK)



$(g^y \% n)$



GET /



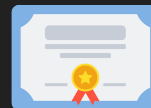
index.html

`<html>...`

Public key



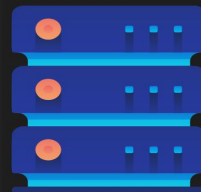
Private key



Y

$(g^x \% n)$

$(g^x \% n)^y = g^{xy} \% n$



There is more to TLS

- More stuff is sent in the TLS handshake
- TLS extensions
 - ALPN
 - SNI
- Cipher algorithms
- Key generation algorithms
- Key size
- Digital signature algorithms
- Client side certificates

Node HTTPS

- Node HTTPS Server requires a certificate and private key
- The rest is the same
- More work though!
- Requests gets hit with additional cost
- Responses gets hit with additional cost


Generate Private key and Certificate with OpenSSL


- OpenSSL is a library for cryptographic operations
- Generate private key
 - `openssl genrsa -out private-key.pem 2048`
- Generate Certificate x509 (which contains public key)
 - `openssl req -new -x509 -key private-key.pem -out certificate.pem -days 365`
 - Answer questions to fill the the 509 fields
 - Most important is common name , subject alternative which is the website

Example Server

16-https > JS 161-raw-https-server.js > ...

```
1 const https = require('node:https');
2 const fs = require('fs');
3
4 // Define certificate and private key
5 const options = {
6   key: fs.readFileSync('private-key.pem'),
7   cert: fs.readFileSync('certificate.pem')
8 };
9
10 // Create an HTTPS server
11 const server = https.createServer(options, (req, res) => {
12   res.writeHead(200, { 'Content-Type': 'text/plain' });
13   res.end('Hello , on HTTPS!\n');
14 });
15
16 // Start the server
17 server.listen(8443, () => {
18   console.log(`HTTPS server is running on https://localhost:8443`);
19 });
20
21 //consume like curl https://localhost:8443 --insecure
22
```

We pass in our private keys
(secret don't leak it!) 

We pass the certificate 

Request is decrypted before we
get here

Response is encrypted right after
this

Example Client

16-https > JS 160-raw-https.js > ...

```
1  const http = require("node:https");
2  const req = http.request("https://example.com", { "method": "GET" });
3
4  req.on("response", res => {
5      console.log(res.headers)
6      console.log(res.statusCode);
7      //set the encoding
8      res.setEncoding('utf-8')
9      res.on("data", data => console.log("some data" + data))
10  })
11
12  req.end(); // must call it to actually send the request
13  // (end the stream //we will discuss this more on the stream lecture)
14  let x = req.getHeaders();
15  console.log(x)
16
```

New module that knows how to do https

https scheme

Response is decrypted before we get here

Request is encrypted right after this

Summary & Demo

- Encryptions
- TLS
- Certificates
- Node HTTPS
- Code exercise →
 - raw https request
 - Create a certificate/private/public key
 - Raw https server