Master in Computer Engineering (MEI)
Integrated Master in Informatics Engineering (MiEI)

Specialization Profile **CSI**: Cryptography and Information Security

Engenharia de Segurança

jose.miranda@devisefutures.com

Topics

- Applied Criptography
  - Blind signatures
  - Cryptographic protocols / applications
    - SSL/TLS
    - SSH

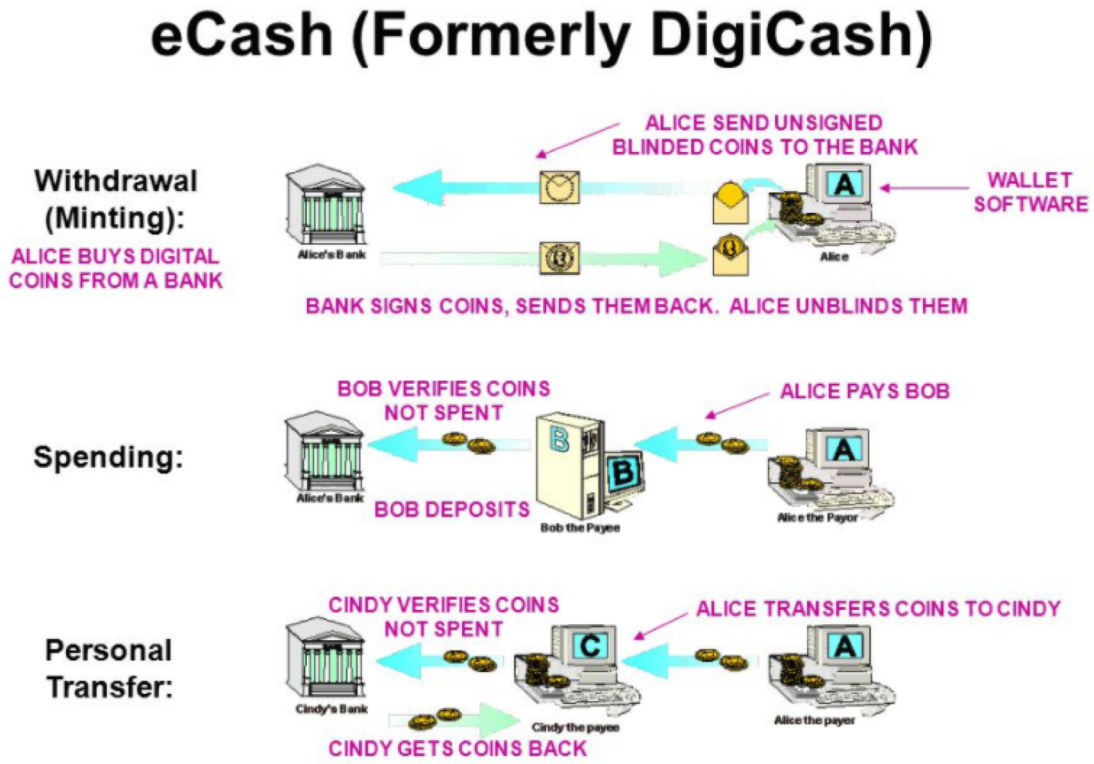
Blind signatures

- Introduced by David Chaum in 1982 for use in the area of currency and electronic payments.
- Blind signature enables an entity to ask a third party to digitally sign a message, without revealing the content of the message;
- Typically the following analogy is made to the physical world:
  - A paper with the message is placed inside an envelope;
  - A carbon paper is inserted in the envelope between the front of the envelope and the paper with the message;
  - The envelope is closed and handed to the subscriber;
  - The subscriber signs the front of the envelope and returns the envelope;
  - The message owner removes the paper from inside the envelope, and it contains the signature of the subscriber.
  - Bottom line: The subscriber did not see the message, but a third party who receives the message can validate the signature.

Blind signatures

When are blind signatures needed?

- Electronic vote, in which ballot papers are signed by the voting system before being deposited in the electronic ballot box, but without the content of the vote being revealed.
- In non-traceable electronic money systems.



Blind signatures

How it works?

- Suppose Alice asks Bob to sign a message **m**, but she does not want Bob to know the content of the message.
- Alice "blinds / obfuscates" the message **m**, with a random number **b** (obfuscation factor): ***blind (m, b)***
- Bob signs the message with his private key **d**, resulting in ***sign (blind (m, b), d)***
- Alice unblinds the message using **b**, resulting in ***unblind (sign (blind (m, b), d), b)***
- The functions used are chosen in such a way that ***unblind (sign (blind (m, b), d), b) = sign (m, d)***, i.e., the signature of **m** by Bob.
- Any entity / person can use Bob's public key to verify that the signature is valid.

Blind signatures

RSA blind signatures

- Let **e** be the RSA public exponent of signer Bob, **d** be the RSA secret exponent of signer Bob, and ***N*** be the RSA modulus.
- Alice chooses a random value **r**, such that **r** is prime in **N** (i.e., ***gcd (r, N) = 1***).
- Alice uses $r^e \bmod N$ as an obfuscation factor
  - Note that since **r** is a random value, $r^e \bmod N$ is also random.
- Let **m** be the message that Alice wants Bob to sign, without knowing its content.

Blind signatures

RSA blind signatures

- Alice obfuscates message **m** with the obfuscation factor

  $$m' \equiv m\, r^e \bmod N$$

  – Note that since $r^e \bmod N$ is random, m' does not reveal any information of m

- Alice sends m' to the signer Bob.
- The signer uses the secret exponent d to calculate the blind signature s'

  $$s' \equiv (m')^d \bmod N$$

- Bob returns s' to Alice.
- Alice removes the obfuscation factor (using r^{-1}), obtaining s, which is the signature of **m** by Bob

  $$s'\, r^{-1} \bmod N \equiv (m')^d\, r^{-1} \bmod N \equiv (m\, r^e)^d\, r^{-1} \bmod N \equiv$$
  $$m^d\, r^{ed}\, r^{-1} \bmod N \equiv m^d\, r\, r^{-1} \bmod N \equiv m^d \bmod N$$

Note that RSA keys satisfy $r^{ed} \equiv r \bmod N$ ança **s**

jose.miranda@devisefutures.com

Blind signatures

RSA blind signatures

- Can be attacked.
- The attacker provides Bob with the blind version of the message ***m'*** obfuscated, i.e., ***m''***

$$\begin{aligned} m'' &= m' r^e \pmod{n} \\ &= (m^e \pmod{n} \cdot r^e) \pmod{n} \\ &= (mr)^e \pmod{n} \end{aligned}$$

- When the attacker unblinds the blind signature of ***m''***, the initial message can easily be obtained.

$m = s' \cdot r^{-1} \pmod{n}$, since

$$\begin{aligned} s' &= m''^d \pmod{n} \\ &= ((mr)^e \pmod{n})^d \pmod{n} \\ &= (mr)^{ed} \pmod{n} \\ &= m \cdot r \pmod{n}, \text{since } ed \equiv 1 \pmod{\phi(n)} \end{aligned}$$

- Solution: Instead of signing the message, sign the message hash.

Blind signatures

Blind signatures based on the *Elliptic Curve Discrete Logarithm Problem* (ECDLP)

- Efficient blind signature system, based on elliptic curve cryptography (ECC), described in http://www.ijimt.org/papers/556-IT302.pdf.
- It has the following phases:
  - Initialization;
  - Obfuscation/Blind;
  - Signature;
  - Unblind;
  - Verification.
- It has three participants:
  - Applicant (Alice),
  - Subscriber (Bob),
  - Checker.

Blind signatures

Blind signatures based on ECDLP

- Parameters of the elliptic curve on the finite Body F_p are defined as follow:

  T= (p, F_p, a, b, G, n, h), where:

  - p is na integer,
  - $a, b \in F_p$ specify the elliptic curve $E(F_p)$ defined by $y^2 = x^3 + ax + b \ (mod\ p)$,
  - $G = (x_G, y_G)$ is a base point of $E(F_p)$,
  - n is a prime number that defines the order of G (i.e., number of points of the subgroup generated by the base point),
  - h is an integer that defines the cofator, $h = \#E(Fp)/n$ (i.e., number of point of the curve, divided by the number of points of the subgroup generated by the base point)

Blind signatures

Blind signatures based on ECDLP

Requester		***Signer***
Blinding Phase 		 ***Initialization Phase***
Calculates r' from the elliptic curve point R' Integers v (randomly selected in the range $(1, n-1)$) Compute $R=v^{-1}R'=(x_0,y_0)$ $r=x_0 \bmod n$ (blinding factor) $m'=H(m)\,r^{-1}r'v \pmod n$ H is the hash function with SHA-256 algorithm	R′ → m′ →	Publish $E_p(a, b)$ Base Point $G=(x, y)$ Integer k (randomly selected in the range $(1, n-1)$) $R'=kG=(x_1, y_1)$ and Compute $r'-x_1 \pmod n$ if $(r' \neq 0)$, Else if choose another k and find r'
Unblinding		***Signing Phase***
$s'=sv^{-1}r'^{-1}r \pmod n$ The unblinding operation is needed to obtain the digital signature (s', R) on message m.	← (s,m′)	*Private key* $= d$ (d randomly selected in the range $(1, n-1)$) *Public key* $= Q = dG = (x_Q, y_Q)$ Check (k, m') in database? If yes, re-select k. Otherwise, compute $s=dm'+kr' \pmod n$
Verifying Phase Any party who has the elliptic domain parameter T of the Signer and the public key of the signer can verify the signature is genuine. $s'G \stackrel{?}{=} QH(m)+Rr$		

Blind signatures

Blind signatures based on ECDLP

- Initialization

```
user@CSI:~/Aulas/Aula3/BlindSignature$ python initSigner-app.py
Output
Init components: 7cb3aebdeaa9723f21df988a3b09fe9475f1e2253011d4cbfda09d62baa1ffe4.b22ad9e44cb42256724e82078ce59
3281bddf440d88267987af3287406bbea30
pRDashComponents: 7cb3aebdeaa9723f21df988a3b09fe9475f1e2253011d4cbfda09d62baa1ffe4.9e7f7b8503308a7565e7650f1648
b61b0b45ad2227928e770520884311d8a87d
```

- Obfuscation/Blind

```
user@CSI:~/Aulas/Aula3/BlindSignature$ python generateBlindData-app.py
Input
Data: Vamos assinar esta mensagem sem o Bob a ver
pRDash components: 7cb3aebdeaa9723f21df988a3b09fe9475f1e2253011d4cbfda09d62baa1ffe4.9e7f7b8503308a7565e7650f164
8b61b0b45ad2227928e770520884311d8a87d
Output
Blind message: 79c597b9d890706b6f48fa6279ecd5fd93ccde333257578bf74d0aed4717e374
Blind components: 2934575c9dd4f8fbbc3595e6af21bbfaa82f8aa02276acce29ee6200ca56b699.ed59ff6e354944d0c8b7d9f93fd3
f287abc56222d92a49994d600a1ccf3744ab
pRComponents: ed59ff6e354944d0c8b7d9f93fd3f287abc56222d92a49994d600a1ccf3744ab.a6157cffafe5fbbb7a41f5cda2aeb858
2568d61a1374f3c2d9cd9ddb37deb611
```

Blind signatures

Blind signatures based on ECDLP

- Signature

```
user@CSI:~/Aulas/Aula3/BlindSignature$ python generateBlindSignature-app.py key.pem
Input
Passphrase:
Blind message: 79c597b9d890706b6f48fa6279ecd5fd93ccde333257578bf74d0aed4717e374
Init components: 7cb3aebdeaa9723f21df988a3b09fe9475f1e2253011d4cbfda09d62baa1ffe4.b22ad9e44cb42256724e82078ce59
3281bddf440d88267987af3287406bbea30
Output
Blind signature: 608d64e75bd4b08733bbc03ed60ad977d063d21943de5875ccea27d0d80e42a667a03f7cb347699164cf91e75b13bb
a47e226f0bd5fdbb3e762686df0cabe104
```

- Unblind

```
user@CSI:~/Aulas/Aula3/BlindSignature$ python unblindSignature-app.py
Input
Blind signature: 608d64e75bd4b08733bbc03ed60ad977d063d21943de5875ccea27d0d80e42a667a03f7cb347699164cf91e75b13bb
a47e226f0bd5fdbb3e762686df0cabe104
Blind components: 2934575c9dd4f8fbbc3595e6af21bbfaa82f8aa02276acce29ee6200ca56b699.ed59ff6e354944d0c8b7d9f93fd3
f287abc56222d92a49994d600a1ccf3744ab
pRDash components: 7cb3aebdeaa9723f21df988a3b09fe9475f1e2253011d4cbfda09d62baa1ffe4.9e7f7b8503308a7565e7650f164
8b61b0b45ad2227928e770520884311d8a87d
Output
Signature: f5efcba14b35e3f6c04a19772fa6af3f95550733ce15e84c687f14adc82b927d
```

Blind signatures

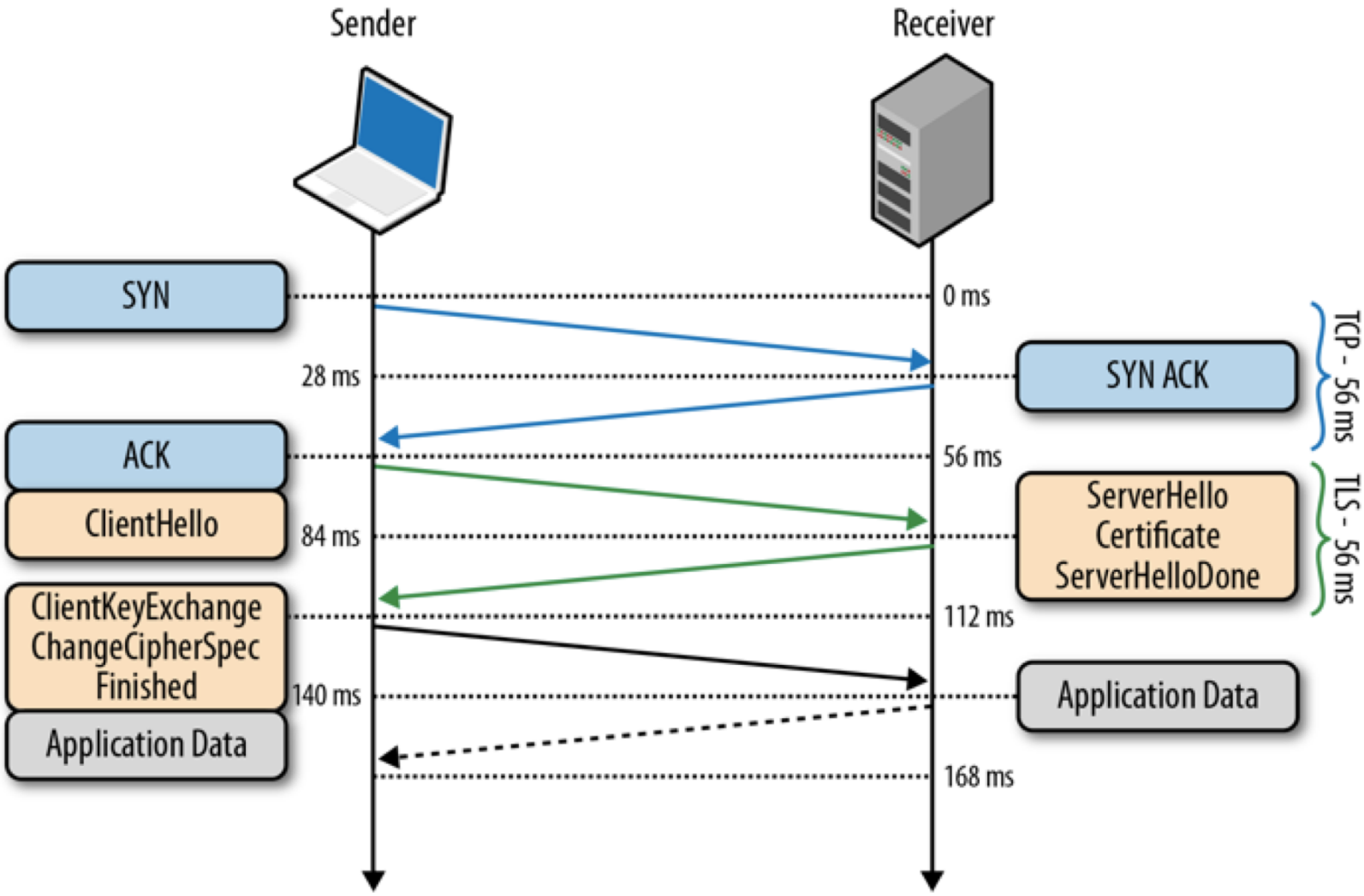
Blind signatures based on ECDLP

- Verification

```
user@CSI:~/Aulas/Aula3/BlindSignature$ python verifySigature-app.py key.crt
Input
Original data: Vamos assinar esta mensagem sem o Bob a ver
Signature: f5efcba14b35e3f6c04a19772fa6af3f95550733ce15e84c687f14adc82b927d
Blind components: 2934575c9dd4f8fbbc3595e6af21bbfaa82f8aa02276acce29ee6200ca56b699.ed59ff6e354944d0c8b7d9f93fd3
f287abc56222d92a49994d600a1ccf3744ab
pR components: ed59ff6e354944d0c8b7d9f93fd3f287abc56222d92a49994d600a1ccf3744ab.a6157cffafe5fbbb7a41f5cda2aeb85
82568d61a1374f3c2d9cd9ddb37deb611
Output
Valid signature
```

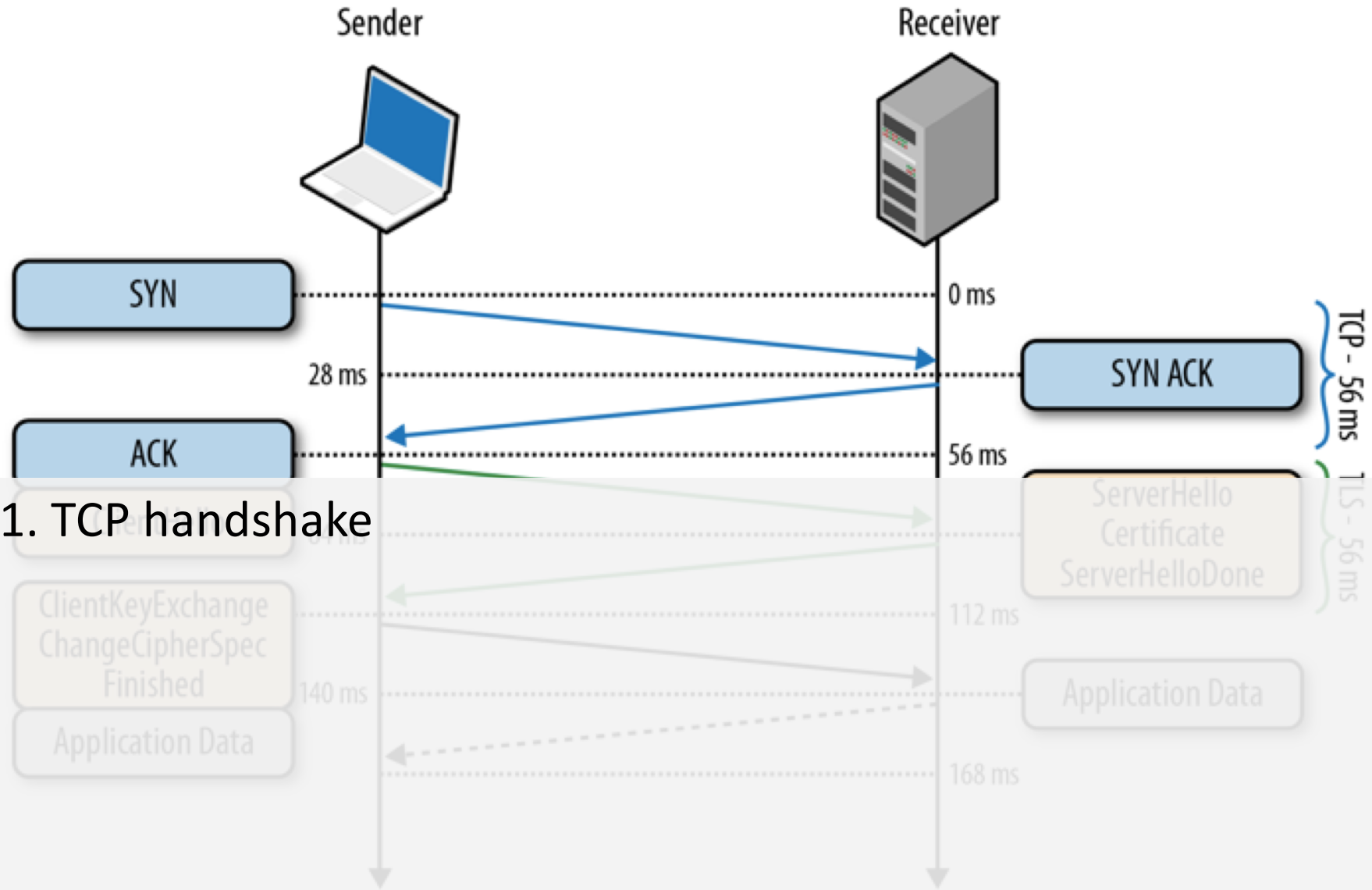
Cryptographic protocols / applications

SSL/TLS (RFC 5246)



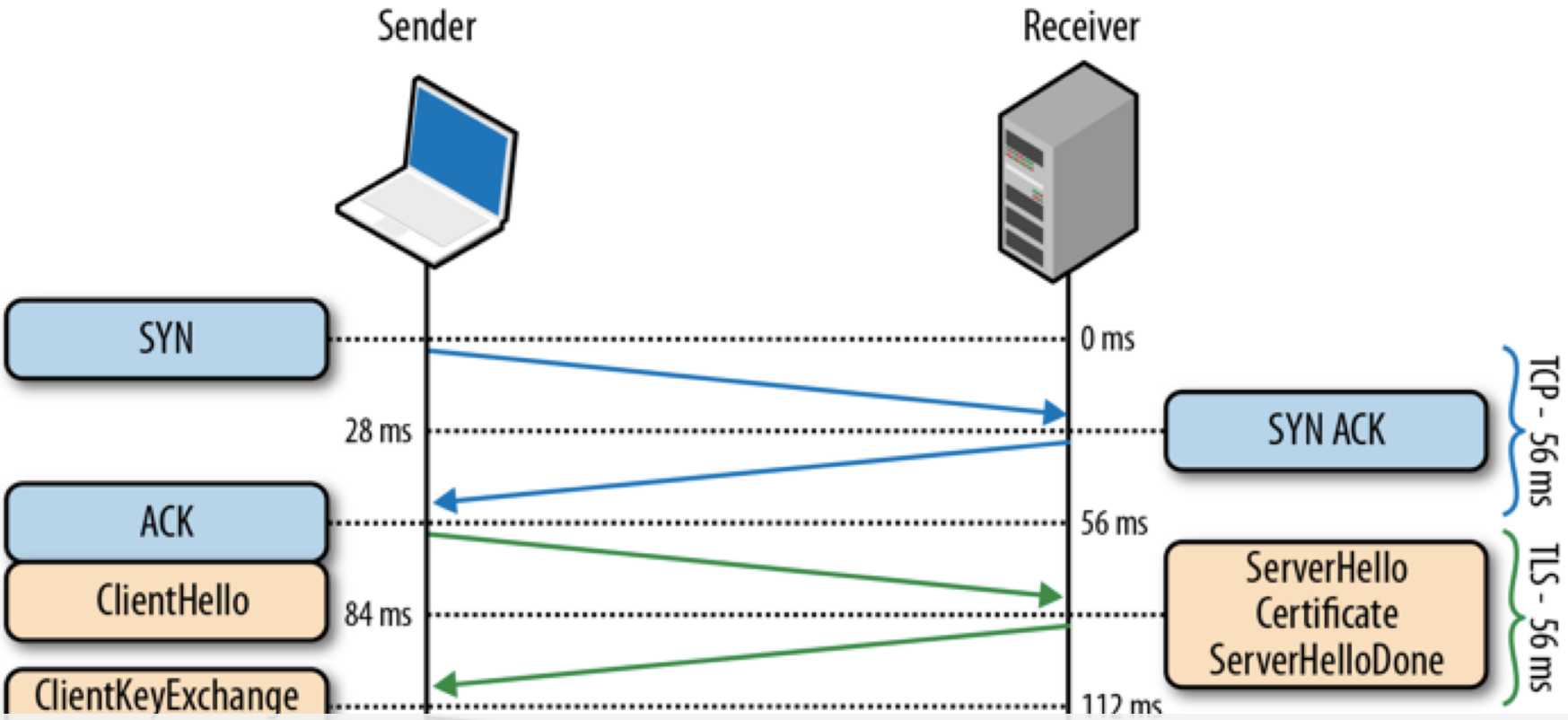
- Note: assuming a 56 ms roundtrip

SSL/TLS (RFC 5246)



- 1. TCP handshake

SSL/TLS (RFC 5246)



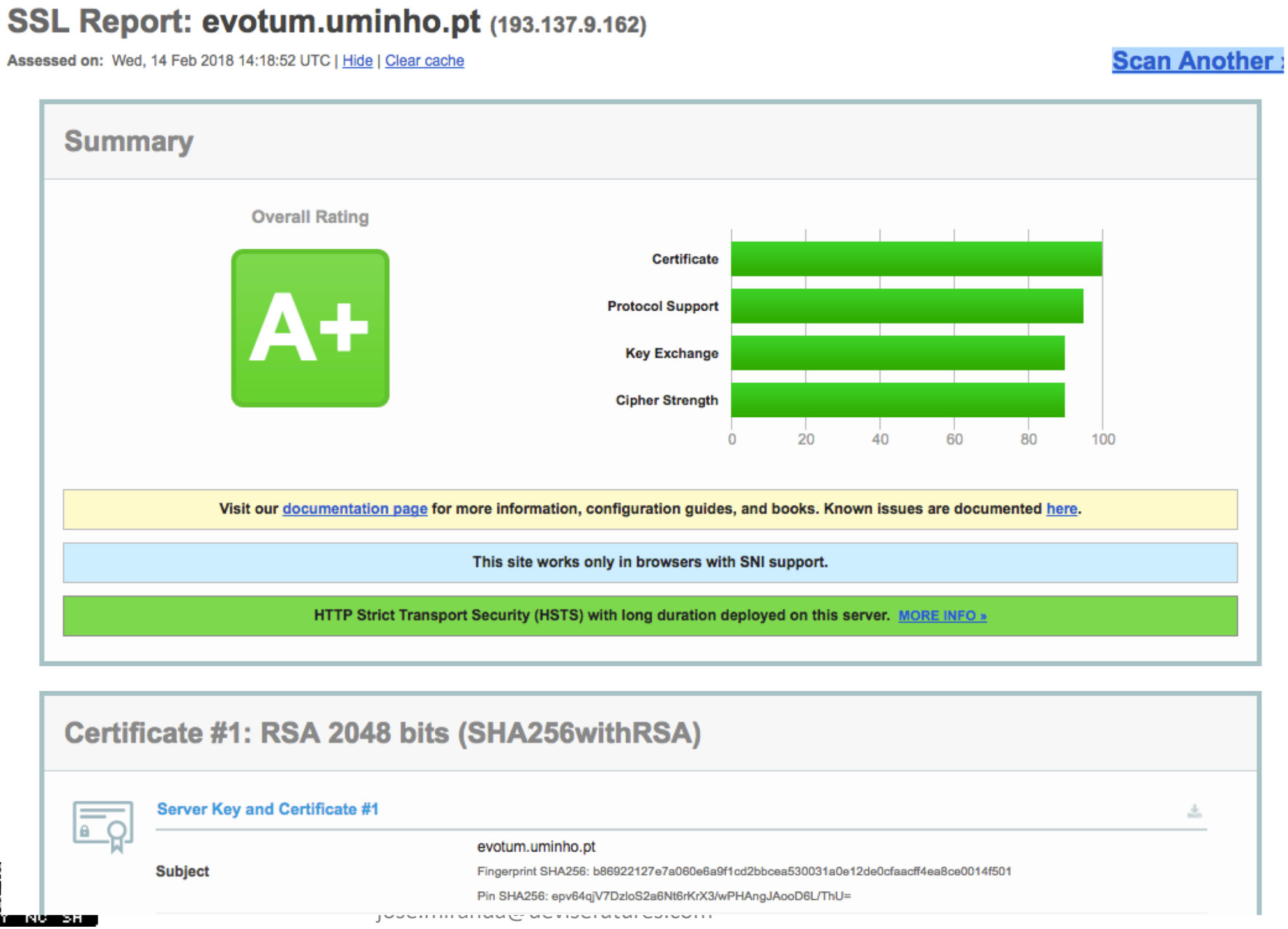
- 2a. Client sends data in plaintext, such as version of the TLS protocol, list of supported ciphers (for example, TLS_RSA_WITH_AES_256_CBC_SHA256) and other TLS options ;
- 2b. The server obtains the version of the TLS protocol for communication, chooses the cipher to use from the list provided by the client, adds its certificate and sends the response to the client. Optionally requests the client certificate.

SSL/TLS (RFC 5246)

- 3a. Assuming that both parties have negotiated a common version of the protocol and ciphers, and that the client "accepts" the certificate provided by the server, the client initiates the key exchange (RSA or Diffie-Hellman), used to establish the symmetric key for the session;
- 3b. The server processes the key exchange parameters sent by the client, validates the integrity of the message by checking the MAC, and returns the message "Finished" encrypted.

- 3c. The client decrypts the message with the negotiated symmetric key, verifies the MAC, and if everything is correct, establishes the tunnel (all communication is encrypted by the negotiated symmetric key) and the application data is sent.

SSL/TLS – teste (www.ssllabs.com)



SSL/TLS – teste (www.ssllabs.com)

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 currently support draft version 18.

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_256_CBC_SHA384 (0xc028) ECDH secp256r1 (eq. 3072 bits RSA) FS	256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) ECDH secp256r1 (eq. 3072 bits RSA) FS	256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x9f) DH 4096 bits FS	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256 (0x6b) DH 4096 bits FS	256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x39) DH 4096 bits FS	256
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x88) DH 4096 bits FS	256

SSL/TLS – teste (www.ssllabs.com)

Handshake Simulation

Android 2.3.7 No SNI [2]	Incorrect certificate because this client doesn't support SNI RSA 2048 (SHA256) | TLS 1.0 | TLS_DHE_RSA_WITH_AES_128_CBC_SHA | DH 4096		
Android 4.0.4	RSA 2048 (SHA256)	TLS 1.0	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Android 4.1.1	RSA 2048 (SHA256)	TLS 1.0	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Android 4.2.2	RSA 2048 (SHA256)	TLS 1.0	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Android 4.3	RSA 2048 (SHA256)	TLS 1.0	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Android 4.4.2	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Android 5.0.0	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Android 6.0	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Android 7.0	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Chrome 49 / XP SP3	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Chrome 57 / Win 7 R	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Firefox 31.3.0 ESR / Win 7	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Firefox 47 / Win 7 R	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA ECDH secp256r1 FS
Firefox 49 / XP SP3	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Firefox 53 / Win 7 R	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Java 6u45 No SNI [2]	Client does not support DH parameters > 1024 bits RSA 2048 (SHA256) | TLS 1.0 | TLS_DHE_RSA_WITH_AES_128_CBC_SHA | DH 4096		
Java 7u25	RSA 2048 (SHA256)	TLS 1.0	TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA ECDH secp256r1 FS
Java 8u31	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 ECDH secp256r1 FS
OpenSSL 0.9.8y	RSA 2048 (SHA256)	TLS 1.0	TLS_DHE_RSA_WITH_AES_256_CBC_SHA DH 4096 FS
OpenSSL 1.0.1l R	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
OpenSSL 1.0.2e R	RSA 2048 (SHA256)	TLS 1.2	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Safari 9 / iOS 9 R	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Safari 9 / OS X 10.11 R	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Safari 10 / iOS 10 R	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS
Safari 10 / OS X 10.12 R	RSA 2048 (SHA256)	TLS 1.2 > http/1.1	TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 ECDH secp256r1 FS

SSL/TLS – teste (openssl)

```
user@CSI:~$ openssl s_client -state -servername evotum.uminho.pt -connect evotum.uminho.pt:443
CONNECTED(00000003)
SSL_connect:before SSL initialization
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS read server hello
```

```
---
Certificate chain
 0 s:/C=PT/ST=Braga/L=Braga/O=Universidade do Minho/OU=SCOM/CN=evotum.uminho.pt
   i:/C=NL/ST=Noord-Holland/L=Amsterdam/O=TERENA/CN=TERENA SSL CA 3
 1 s:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Assured ID Root CA
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Assured ID Root CA
 2 s:/C=NL/ST=Noord-Holland/L=Amsterdam/O=TERENA/CN=TERENA SSL CA 3
   i:/C=US/O=DigiCert Inc/OU=www.digicert.com/CN=DigiCert Assured ID Root CA
---
```

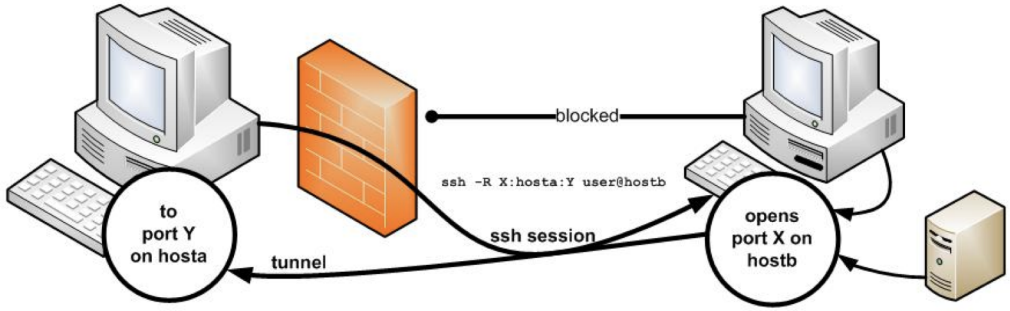
SSL/TLS – teste (openssl)

```
user@CSI:~$ openssl s_client -state -servername evotum.uminho.pt -connect evotum.uminho.pt:443
CONNECTED(00000003)
SSL_connect:before SSL initialization
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS write client hello
SSL_connect:SSLv3/TLS read server
---
Certificate chain
 0 s:/C=PT/ST=Braga/L=Braga/O=Univ
   i:/C=NL/ST=Noord-Holland/L=Amst
 1 s:/C=US/O=DigiCert Inc/OU=www.d
   i:/C=US/O=DigiCert Inc/OU=www.d
 2 s:/C=NL/ST=Noord-Holland/L=Amst
   i:/C=US/O=DigiCert Inc/OU=www.d
---
SSL handshake has read 4262 bytes and written 327 bytes
Verification: OK
---
New, TLSv1.2, Cipher is ECDHE-RSA-AES256-GCM-SHA384
Server public key is 2048 bit
Secure Renegotiation IS supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
SSL-Session:
    Protocol  : TLSv1.2
    Cipher    : ECDHE-RSA-AES256-GCM-SHA384
    Session-ID: 17595A5243D0ED2AEA2345C32E5E6392B9B20C0AA664906F7C784825F9C4F818
    Session-ID-ctx:
    Master-Key: 81905ADC8F48E67A0EFA2B3F87B1C7F1D8B3D4517087F7DED26425B4B10EB0479FDBA0ED57E8B03AD59
DACC68DB8102F
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    TLS session ticket lifetime hint: 300 (seconds)
    TLS session ticket:
    0000 - 2d 4a e6 31 8b 37 d3 f4-2e 68 c2 ad 70 0e 5b e2   -J.1.7...h..p.[.
    0010 - 89 41 0f 70 7f 13 2f d1-9e 57 b8 0c a7 4c f3 bf   .A.p../..W...L..
    0020 - 2b 9a 51 48 a9 b4 3e 4e-1f 53 46 9d b2 ce 13 7f   +.QH..>N.SF.....
    0030 - 23 4e af a1 bc 4b 5b 0c-79 82 5d 3f 48 66 de 65   #N...K[.y.]?Hf.e
    0040 - 6c e0 44 2c a9 2d 85 58-43 eb 37 30 7a fa 7f cc   l.D,.-.XC.70z...
    0050 - f5 4e 06 ab a6 44 b4 52-39 fb 94 3f 36 50 f0 0c   .N...D.R9..?6P..
    0060 - 80 32 54 8a 03 f3 51 5a-62 4e 27 f1 e7 fe ef df   .2T...QZbN'.....
    0070 - de 36 bb 5e 8b b8 d6 3a-96 95 25 bb 37 e5 ba bc   .6.^...:..%.7...
    0080 - 54 42 3d 29 4d fd c1 e0-9c 06 ca 2a 0c 50 8a a2   TB=)M......*.P..
    0090 - 27 b7 0e db 43 12 a6 8e-8f 63 6a 16 32 d2 18 d1   '...C....cj.2...
    00a0 - 85 d0 7a 17 4a d9 aa 05-3b 3b 07 85 91 ab fc f8   ..z.J...;;......
    00b0 - 16 b6 2b b4 18 ae e5 7c-52 25 23 3f 7d ca 19 c4   ..+....|R%#?}...

    Start Time: 1518960431
    Timeout   : 7200 (sec)
    Verify return code: 0 (ok)
    Extended master secret: no
```

SSH (Secure Shell)

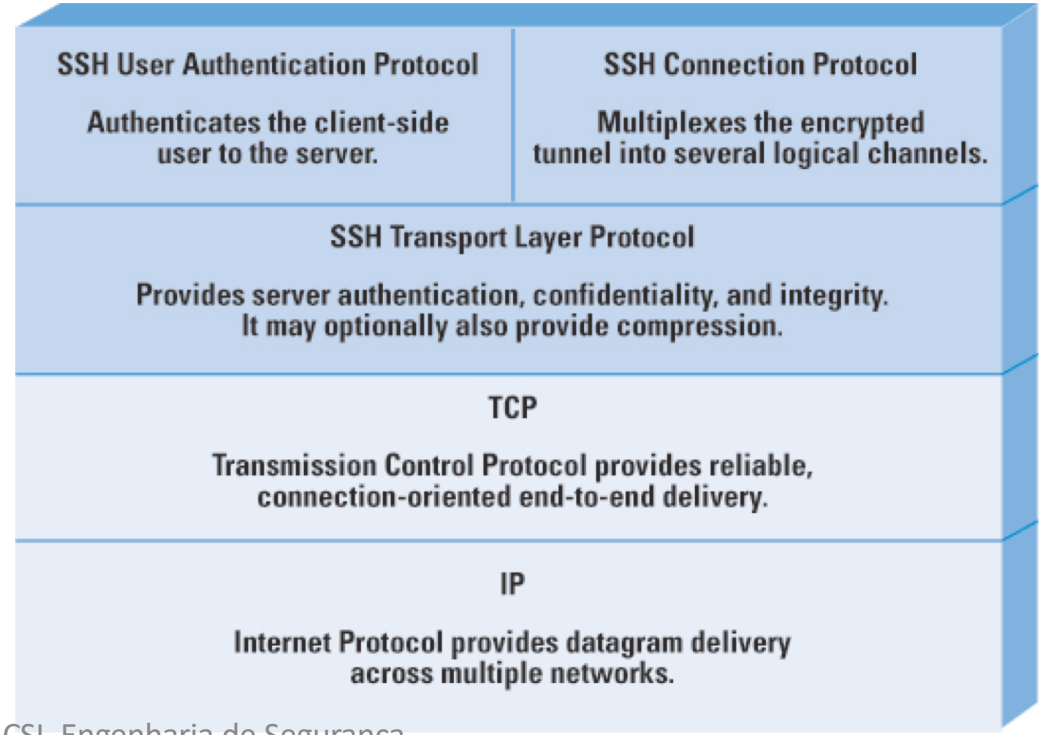
- **Network cryptographic protocol** used, among others, to:
  - Access remote computer,
  - Tunneling - for example to create an encrypted channel to carry non-encrypted protocols (e.g. SMB),



  - TCP port and X11 sessions forwarding,
  - File transfer via SSH file transfer (SFTP) or secure copy (SCP),
  - VPN creation (for routing packets between different networks or bridging between broadcast domains)
- Uses public-key encryption to authenticate the remote computer as well as the user, if necessary.

SSH (Secure Shell)

- Organized in three protocols on top of TCP
  - SSH Transport Layer Protocol
    - Note: It has the *forward secrecy* property (i.e., if a key is compromised during a session, this knowledge does not affect the security of previous sessions)
  - SSH User Authentication Protocol
  - SSH Connection Protocol

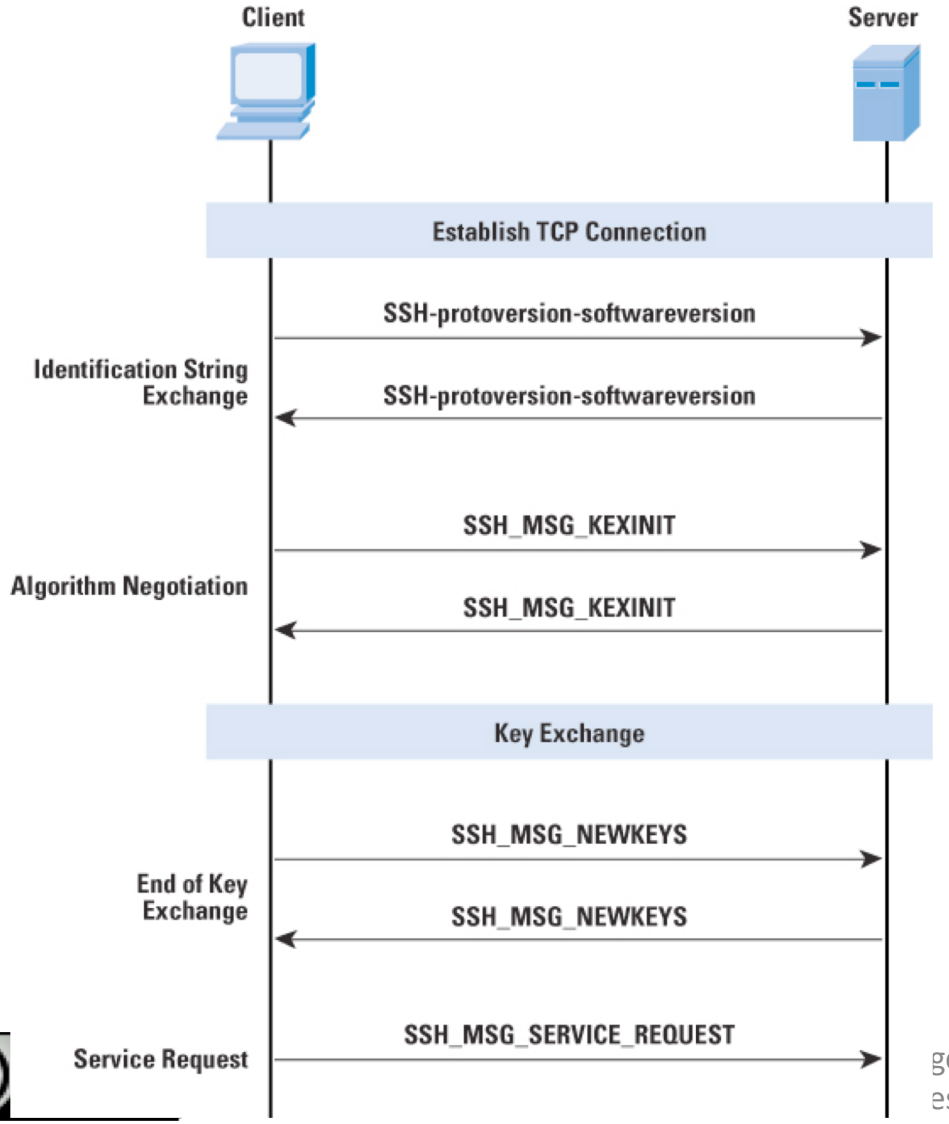


SSH (Secure Shell)

- SSH Transport Layer Protocol (RFC 4253)

SSH Transport Layer Protocol

Provides server authentication, confidentiality, and integrity. It may optionally also provide compression.



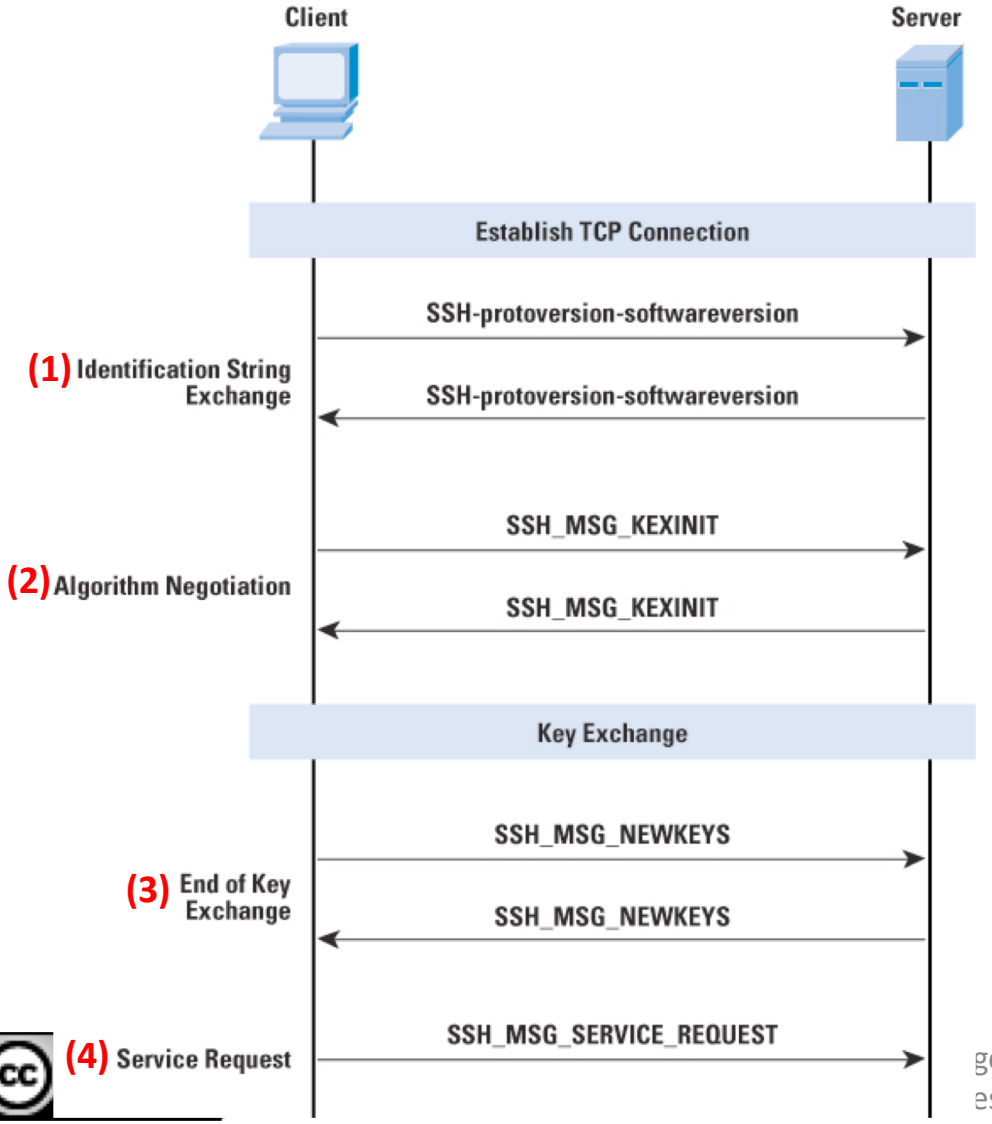
- Server authentication is performed based on the server's public-private key pair;
- For this authentication to be possible, the client must know (previously) the public key of the server
  - Typically, the client has a local database with the public keys of the known servers, not requiring a central administration infrastructure or coordination with a third party (there is an alternative that the public key is certified by a Certification Authority and in this case the client has to save the root certificate of the CA to validate the certificates with the public key of the servers);

SSH (Secure Shell)

- SSH Transport Layer Protocol (RFC 4253)

SSH Transport Layer Protocol

Provides server authentication, confidentiality, and integrity. It may optionally also provide compression.



Step 1: Identification exchange, in the format *SSH-protoversion-softwareversion SP comments CR LF*

Step 2: Negotiation of algorithms (key exchange - DH -, cipher - 3des-cbc, aes128-cbc, ... -, MAC - hmac-sha1, ... - and compression - none or zlib -), by order of preference (chosen the first algorithm proposed by the client that the server also supports, for each type of algorithm)

Step 3: Diffie-Hellman key exchange, according to RFC 2409, setting the session key and ensuring server authentication (server used his private key to sign/encrypt the DH key exchange). From this moment on, traffic will be encrypted, according to the next slide.

Step 4: the client sends an SSH_MSG_SERVICE_REQUEST packet, requesting the *SSH User Authentication* or *SSH Connection* protocol

SSH (Secure Shell)

- SSH packets in the TCP data segment

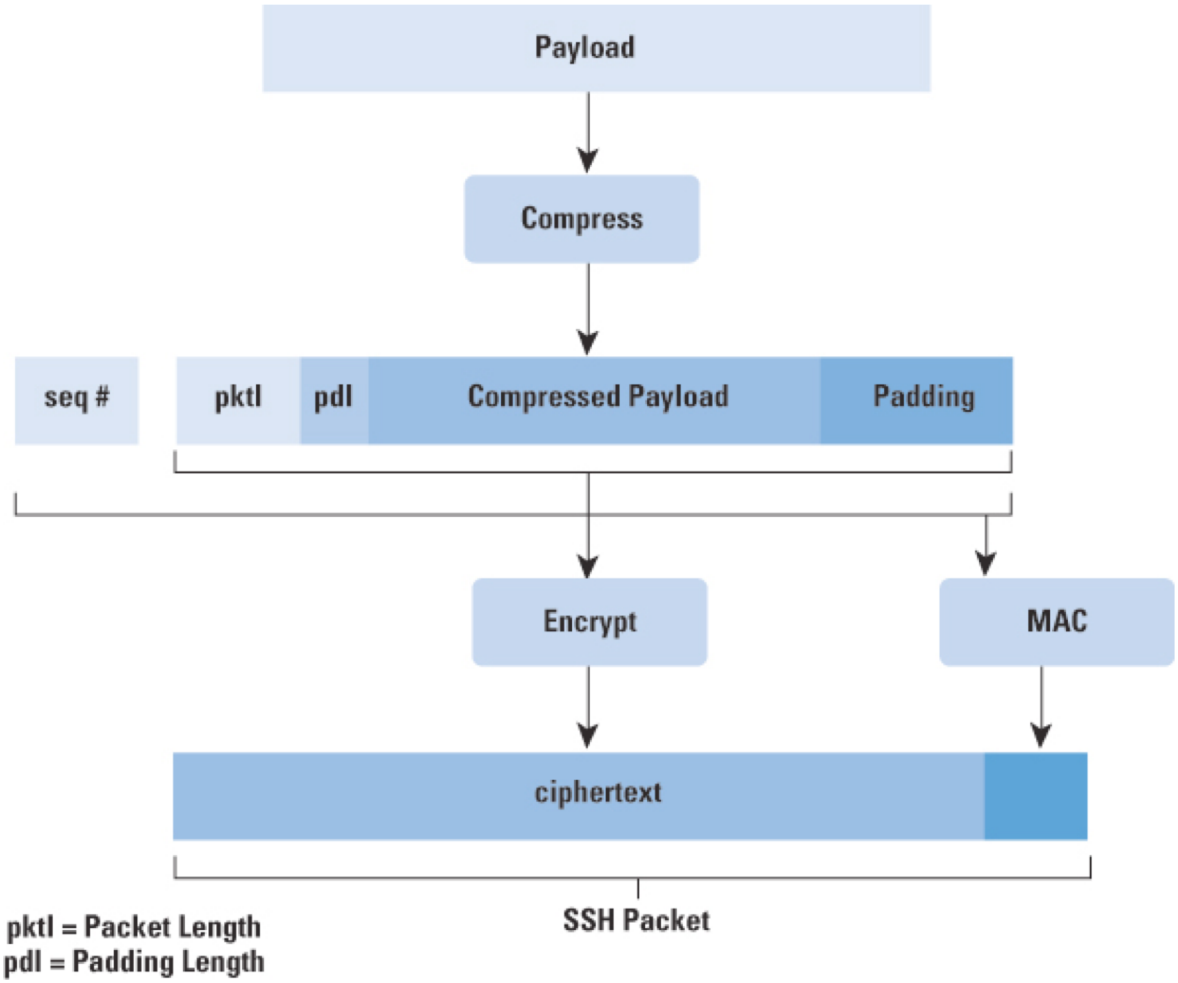
Packet length: size of the SSH packet in bytes, not including the *packet length* and MAC fields..

Padding length: size of the Padding field.

Payload: useful content of the SSH packet.

Padding: Contains random padding bytes, so that the total size of the SSH packet (excluding the MAC) is multiple of the cipher block size, or 8 bytes if it is a stream cipher.

Message Authentication Code (MAC): The MAC is calculated over the entire SSH packet and a sequential number, excluding the MAC field. The sequential number is the sequential (32-bit) number of the packet, initialized to zero for the first packet and incremented for each subsequent packet. The sequence number is not included in the information sent.



SSH (Secure Shell)

- SSH User Authentication Protocol (RFC 4252)

SSH User Authentication Protocol

Authenticates the client-side user to the server.

  - User authenticated by the server
    - *Username* – identity that the client claims
    - *Service name* – service that the customer wants to access (typically o *SSH Connection Protocol*),
    - *Method name* – authentication method (*publickey*, *password* ou *hostbased*).

byte	SSH_MSG_USERAUTH_REQUEST (50)
string	username
string	service name
string	method name
....	method-specific fields

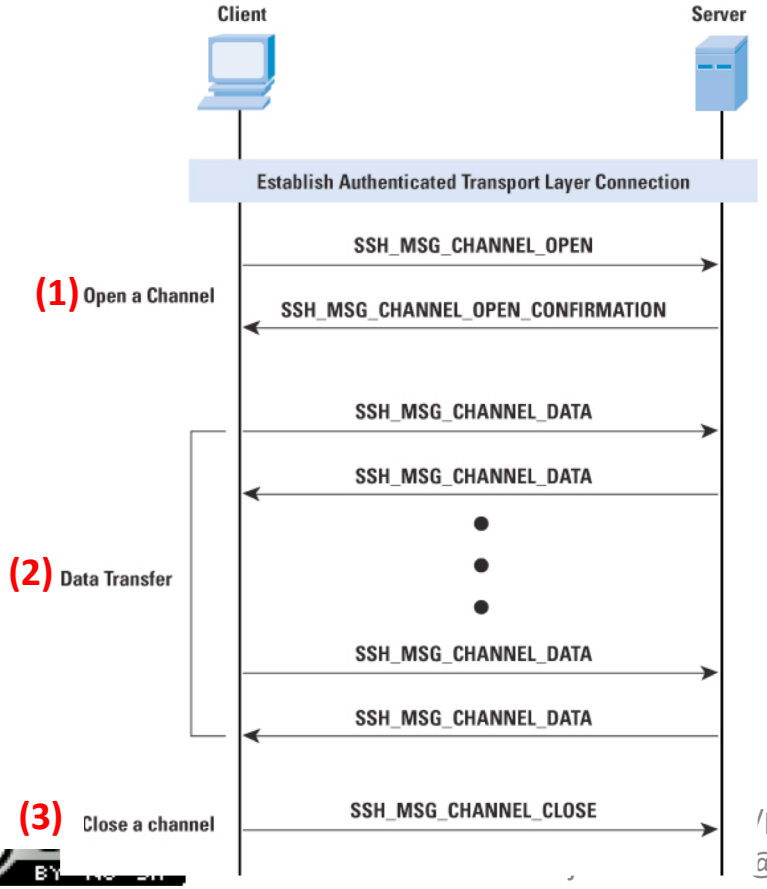
SSH (Secure Shell)

- SSH Connection Protocol (RFC 4254)

SSH Connection Protocol

Multiplexes the encrypted tunnel into several logical channels.

  - Executes on top of the *SSH Transport Layer Protocol* and assumes that the connection is secure and authenticated (called a tunnel)
  - The *Connection Protocol* allows the creation of several logical channels over the same tunnel



Step 1: The opening of a channel identifies the type of application

- session – remote execution of a program (e.g., shell, scp, sftp, ...),
- X11 – allows the application to run on the server, but the graphical view is performed on the client,
- forwarded-tcpip – remote port forwarding,
- direct-tcpip – local port forwarding

for this channel and establish the channel identification/number.

Step 2: While the channel is open, data is transferred in both directions.

Step 3: When one side decides to close the channel, it sends the message SSH_MSG_CHANNEL_CLOSE.

SSH (Secure Shell)

- Usage:
  - Remote access:
    - The first time you need to pay attention to the displayed fingerprint, since from there the remote machine is validated

```
[jepm@ProOne ~]$ ssh jepm@algo.paranoidjasmine.com
The authenticity of host 'algo.paranoidjasmine.com (163.172.150.117)' can't be established.
ECDSA key fingerprint is SHA256:hHqIkUw3XmEUnVKmFqP4ajGeFtw0P6QJnsOwEN2DZXE.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'algo.paranoidjasmine.com' (ECDSA) to the list of known hosts.
```

    - How can I validate the RSA key fingerprint?
      - Verify if it is published by the remote system administrator or physically access the remote system and issue the following command:

```
jepm@algo:~$ sudo ssh-keygen -l -f /etc/ssh/ssh_host_ecdsa_key
256 SHA256:hHqIkUw3XmEUnVKmFqP4ajGeFtw0P6QJnsOwEN2DZXE root@DF.server01 (ECDSA)
```

SSH (Secure Shell)

- Usage:
  - Remote access with user authentication
    - Generate the pair of personal keys (on the local machine) so that you do not need to enter a password in the remote server (although it is necessary to enter the passphrase of the created private key):

```
jmiranda:0 ~ 1001 $ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/jmiranda/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/jmiranda/.ssh/id_rsa.
Your public key has been saved in /home/jmiranda/.ssh/id_rsa.pub.
The key fingerprint is:
56:6a:a0:7f:a3:2e:22:89:fd:9b:98:f0:05:21:a7:a0 jmiranda@clients01.
```

    - Copy content from ~/.ssh/id_rsa.pub to the remote machine and add (on the remote machine) to ~/.ssh/authorized_keys
    - On the remote machine perform

```
chmod 700 ~/.ssh
chmod 600 ~/.ssh/authorized_keys
```

in such a way that:

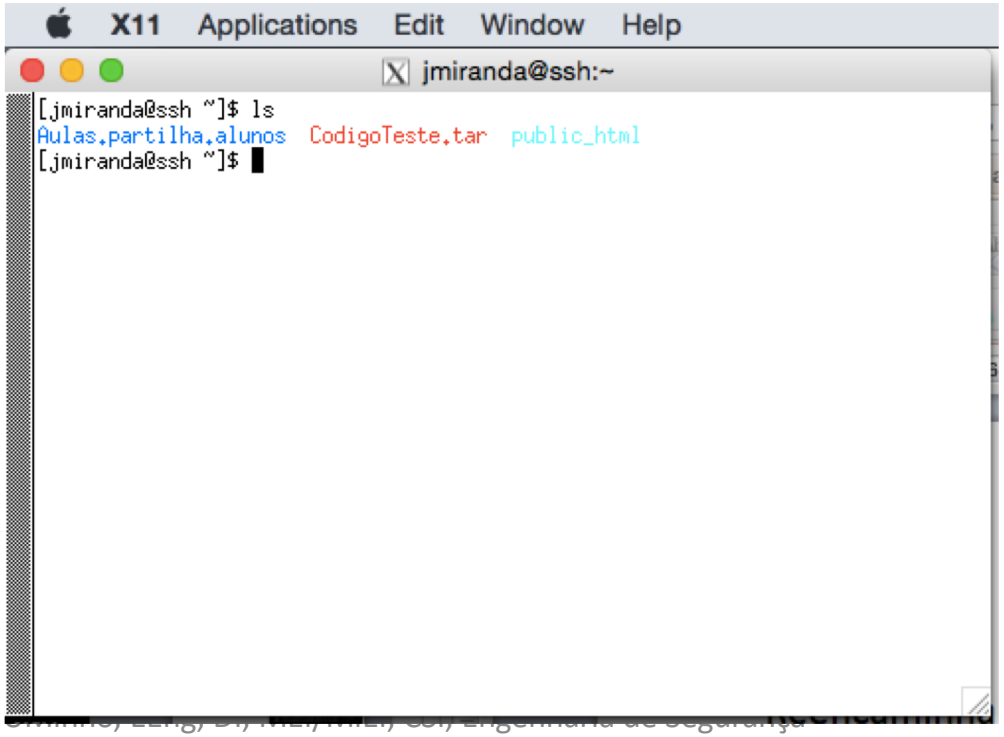
```
[jmiranda@ssh .ssh]$ ls -la
total 8
drwx------ 2 jmiranda dcc   28 Nov 26 20:29 .
drwx------ 5 jmiranda dcc 4096 Dec 15 17:19 ..
-rw-r--r-- 1 jmiranda dcc  393 Nov 26 20:29 authorized_keys
```

SSH (Secure Shell)

- Usage:
  - X11 session forwarding

```
$ ssh -X jmiranda@ssh.alunos.dcc.fc.up.pt
```

```
[jmiranda@ssh ~]$ ls
Aulas.partilha.alunos  CodigoTeste.tar  public_html
[jmiranda@ssh ~]$ xterm&
```



SSH (Secure Shell)

- Usage:
  - TCP port forwarding (*local port forward*)
    - For example, accessing the remote mail server through a local port
      - Note: -L (means local port) and has the argument
        <local-port>:<connect-to-host>:<connect-to-port>
    - On the local machine perform:

```
$ ssh -L 8025:smtp.alunos.dcc.fc.up.pt:25 jmiranda@ssh.alunos.dcc.fc.up.pt
```

```
$ telnet localhost 8025
Trying ::1...
Connected to localhost.
Escape character is '^]'.
220 smtp.alunos.dcc.fc.up.pt ESMTP Postfix (2.2.8) (Mandriva Linux)
```

SSH (Secure Shell)

- Usage:
  - Inverted TCP port forwarding (*reverse/remote port forward*)
    - For example, connect to the remote port and start listening on port 8080 to access the proxy through the local machine and access the intraWeb
      - Note: -R (means remote port) and has the argument

        <remote-port>:<connect-to-host>:<connect-to-port>

    - On the remote machine perform:

```
[jmiranda@ssh ~]$ ssh -R 8080:192.168.0.1:8080 jmiranda@clients.mycryptovault.com
```

    - On the local machine (clientes.mycryptovault.com) perform:

```
jmiranda:0 ~/.ssh 1019 $ curl --proxy localhost:8080 web.alunos.dcc.fc.up.pt
<html>
<body>
<!-- $Id: index.html 92365 2007-09-23 14:04:27Z oden $ -->
<!-- $HeadURL: svn+ssh://svn.mandriva.com/svn/packages/cooker/apache-conf/current/SOURCES/index.html $ --
>
<h1>It works!</h1>
</body>
</html>
```

      - Notice that

```
jmiranda:0 ~/.ssh 1020 $ curl web.alunos.dcc.fc.up.pt
curl: (6) Couldn't resolve host 'web.alunos.dcc.fc.up.pt'
```

SSH Audit (ssh-audit)

```
user@CSI:~/Tools/ssh-audit$ python ssh-audit.py ssh.dcc.fc.up.pt
# general
(gen) banner: SSH-2.0-OpenSSH_6.6.1
(gen) software: OpenSSH 6.6.1
(gen) compatibility: OpenSSH 6.5-6.6, Dropbear SSH 2013.62+ (some functionality from 0.52)
(gen) compression: enabled (zlib@openssh.com)

# key exchange algorithms
(kex) curve25519-sha256@libssh.org          -- [info] available since OpenSSH 6.5, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp256                    -- [fail] using weak elliptic curves
                                            `- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp384                    -- [fail] using weak elliptic curves
                                            `- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62

# host-key algorithms
(key) ssh-rsa                              -- [info] available since OpenSSH 2.5.0, Dropbear SSH 0.28
(key) ecdsa-sha2-nistp256                  -- [fail] using weak elliptic curves
                                           `- [warn] using weak random number generator could reveal the key
                                           `- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62

# encryption algorithms (ciphers)
(enc) aes128-ctr                           -- [info] available since OpenSSH 3.7, Dropbear SSH 0.52
(enc) aes192-ctr                           -- [info] available since OpenSSH 3.7
(enc) aes256-ctr                           -- [info] available since OpenSSH 3.7, Dropbear SSH 0.52
(enc) arcfour256                           -- [fail] removed (in server) since OpenSSH 6.7, unsafe algorithm

# message authentication code algorithms
(mac) hmac-md5-etm@openssh.com              -- [fail] removed (in server) since OpenSSH 6.7, unsafe algorithm
                                            `- [warn] disabled (in client) since OpenSSH 7.2, legacy algorithm
                                            `- [warn] using weak hashing algorithm
                                            `- [info] available since OpenSSH 6.2
(mac) hmac-sha1-etm@openssh.com             -- [warn] using weak hashing algorithm

# algorithm recommendations (for OpenSSH 6.6.1)
(rec) -diffie-hellman-group14-sha1         -- kex algorithm to remove
(rec) -diffie-hellman-group-exchange-sha1  -- kex algorithm to remove
(rec) -diffie-hellman-group1-sha1          -- kex algorithm to remove
(rec) -ecdh-sha2-nistp256                  -- kex algorithm to remove
(rec) -ecdh-sha2-nistp521                  -- kex algorithm to remove
(rec) -ecdh-sha2-nistp384                  -- kex algorithm to remove
(rec) -ecdsa-sha2-nistp256                 -- key algorithm to remove
(rec) +ssh-ed25519                         -- key algorithm to append
```