Standard situation

- **weak authentication:**
  - username and password
    - problem: password snooping
  - IP address (server web; R commands = rsh, rlogin, rcp, ...)
    - problem: IP spoofing
- **other frequent problems:**
  - data snooping / forging
  - shadow server / MITM
  - replay

Channel security

- **authentication (single or mutual), integrity and privacy only during the transit inside the communication channel**
- no possibility of non repudiation
- requires no (or small) modification of applications
Message / data security
- authentication (single), integrity and privacy self-contained in the message
- possibility of non repudiation
- requires modification of applications

Security internal to applications
- each application implements security internally
- the common part is limited to the communication channels (socket)
- possible implementation errors (inventing security protocols is not simple!)
- does not guarantee interoperability

Security external to applications
- the session level would be the ideal one to be used to implement many security functions
- … but it does not exist in TCP/IP!
- a “secure session” level was proposed:
  - it simplifies the work of application developers
  - it avoids implementation errors
  - it is up to the application to select it (or not)
Secure channel protocols

- **SSL / TLS**
  - the most widely used!
- **SSH**
  - it was a successful product (especially in the period when export of USA crypto products was restricted), but today it is a niche product
- **PCT**
  - proposed by MS as an alternative to SSL
  - one of the few fiascos of MS!

SSL (Secure Socket Layer)

- **proposed by Netscape Communications**
- **secure transport channel (session level):**
  - peer authentication (server, server+client)
  - message confidentiality
  - message authentication and integrity
  - protection against replay and filtering attacks
- **easily applicable to all protocols based on TCP:**
  - HTTP, SMTP, NNTP, FTP, TELNET, ...
  - e.g. the famous secure HTTP (https://....) = 443/TCP

Official ports for SSL applications

- **nsilops** 261/tcp # IIOP Name Service over TLS/SSL
- **https** 443/tcp # http protocol over TLS/SSL
- **smtps** 465/tcp # smtp protocol over TLS/SSL (was ssmtp)
- **nntps** 563/tcp # nntp protocol over TLS/SSL (was snntp)
- **imap4-ssl** 585/tcp # IMAP4+SSL (use 993 instead)
- **sshell** 614/tcp # SSLshell
- **ldaps** 636/tcp # ldap protocol over TLS/SSL (was sldap)
- **ftps-data** 989/tcp # ftp protocol, data, over TLS/SSL
- **ftps** 990/tcp # ftp protocol, control, over TLS/SSL
- **telnets** 992/tcp # telnet protocol over TLS/SSL
- **imaps** 993/tcp # imap4 protocol over TLS/SSL
- **ircs** 994/tcp # irc protocol over TLS/SSL
- **pop3s** 995/tcp # pop3 protocol over TLS/SSL (was spop3)
- **msft-gc-ssl** 3269/tcp # MS Global Catalog with LDAP/SSL
SSL – authentication and integrity

- peer authentication at channel setup:
  - the server authenticates itself by sending its public key (X.509 certificate) and by responding to an asymmetric challenge
  - the client authentication (with public key and X.509 certificate) is optional
- for authentication and integrity of the data exchanged over the channel the protocol uses:
  - a keyed digest (MD5 or SHA-1)
  - an MID to avoid replay and cancellation

SSL - confidentiality

- the client generates a session key used for symmetric encryption of data (RC2, RC4, DES, 3DES or IDEA)
- key exchange with the server occurs via public key cryptography (RSA, Diffie-Hellman or Fortezza-KEA)
SSL-3 architecture

<table>
<thead>
<tr>
<th>SSL handshake protocol</th>
<th>SSL change cipher spec protocol</th>
<th>SSL alert protocol</th>
<th>application protocol (e.g. HTTP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSL record protocol</td>
<td>reliable transport protocol (e.g. TCP)</td>
<td>network protocol (e.g. IP)</td>
<td></td>
</tr>
</tbody>
</table>

**Session-id**

Typical web transaction:
- 1. open, 2. GET page.htm, 3. page.htm, 4. close
- 1. open, 2. GET home.gif, 3. home.gif, 4. close
- 1. open, 2. GET logo.gif, 3. logo.gif, 4. close
- 1. open, 2. GET back.jpg, 3. back.jpg, 4. close
- 1. open, 2. GET music.mid, 3. music.mid, 4. close

If the SSL cryptographic parameters must be negotiated every time, then the computational load becomes high.

**Session-id**

- In order to avoid re-negotiation of the cryptographic parameters for each SSL connection, the SSL server can send a session identifier (that is, more connections can be part of the same logical session)
- If the client, when opening the SSL connection, sends a valid session-id then the negotiation part is skipped and data are immediately exchanged over the secure channel
- The server can reject the use of session-id (always or after a time passed after its issuance)
SSL with session-ID

SSL / TLS sessions and connections

- **SSL session**
  - a logical association between client and server
  - created by the Handshake Protocol
  - defines a set of cryptographic parameters
  - is shared by one or more SSL connections (1:N)
- **SSL connection**
  - a transient SSL channel between client and server
  - associated to one specific SSL session (1:1)

SSL-3 / TLS record protocol

- **application data**
- **fragmentation**: F1, F2
- **compression**
- **computation of MAC**
- **padding**: MAC, P
- **encryption**
- **header**: H
TLS-1.0 record format
- uint8 type = change_cipher_spec (20), alert (21), handshake (22), application_data (23)
- uint16 version = major (uint8) + minor (uint8)
- uint16 length:
  - <= 2**14
    (record not compressed)
  - for compatibility with SSL-2
  - <= 2**14 + 1024
    (compressed records)

SSL – computation of MAC
MAC = message_digest (key, seq_number || type || version || length || fragment)
- message_digest
  - depends on the chosen algorithm
- key
  - sender-write-key or receiver-read-key
- seq_number
  - 64-bit integer (never transmitted but computed implicitly)

SSL-2 problems
- P1) in export version, unnecessarily weakens the authentication keys to 40 bits (same as encryption keys)
- P2) weak MAC (custom keyed-digest)
- P3) padding bytes are included into the MAC computation … but the padding length is not!
  - this allows an active attacker to selectively delete bytes from the end of messages
- P4) ciphersuite rollback attack – as the handshake is not authenticated, an active attacker can change the ciphersuites in the Hello messages to force the use of weak encryption (40 bits minimum, as encryption is mandatory in SSL-2)
SSL-3: solutions to SSL-2 problems
- S1) use 128 bits authentication keys, even with 40 bits encryption keys
- S2) use HMAC (stronger than the custom keyed digest used in SSL-2)
- S3) change the sequence of MAC and padding
- S4) handshake is authenticated (but the Change Cipher Spec message) via the Finished message

SSL-3: new features with respect to SSL-2
- data compression:
  - optional
  - before encryption (after it’s not useful anymore …)
- data encryption is optional:
  - in order to have only authentication and integrity
- possibility to re-negotiate the SSL connection:
  - periodical change of keys
  - change of the algorithms

SSL-3 handshake protocol
- agree on a set of algorithms for confidentiality and integrity
- exchange random numbers between the client and the server to be used for the subsequent generation of the keys
- establish a symmetric key by means of public key operations (RSA, DH or Fortezza)
- negotiate the session-id
- exchange the necessary certificates
Data protection

- key for MAC
- sequence number
- generation of MAC
- data [ compressed ]
- MAC
- padding
- IV
- symmetric encryption
- key for encryption
- protected data

Relationship among keys and sessions
(between a server and the same client)

- pre-master secret (established with PKC)
- master secret
- keys for MAC
- keys for encryption
- IV for encryption
- different for each connection
- client random
- server random
- generated for each connection
- common to several connections
- common to several connections

“Ephemeral ” mechanisms

- one-time keys generated on the fly:
  - to provide authentication they must be signed (e.g. an X.509 certificate must be available)
  - DH suitable, RSA slow
    - compromise for RSA = re-use N times
- perfect forward secrecy:
  - who knows the private key can decrypt all the SSL sessions
  - with ephemeral mechanisms the server’s private key is used only for signing
Client hello

- SSL version preferred by the client
- 28 pseudo-random bytes (Client Random)
- a session identifier (session-id)
  - 0 to start a new session
  - different from 0 to ask to resume a previous session
- list of “cipher suite” (=alg of encryption + key exchange + integrity) supported by the client
- list of compression methods supported by the client
Server hello

- SSL version chosen by the server
- 28 pseudo-random bytes (Server Random)
- a session identifier (session-id)
  - new session-id if session-id=0 in the client-hello or reject the session-id proposed by the client
  - session-id proposed by the client if the server accepts to resume the session
- “cipher suite” chosen by the server
  - should be the strongest one in common with the client
- compression method chosen by the server

Cipher suite

- key exchange algorithm
- symmetric encryption algorithm
- hash algorithm (for MAC)

examples:
- SSL_NULL_WITH_NULL_NULL
- SSL_RSA_WITH_NULL_SHA
- SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
- SSL_RSA_WITH_3DES_EDE_CBC_SHA

Certificate (server)

- certificate for server authentication
  - the subject / subjectAltName must be the same as the identity of the server (DNS name, IP address, ...)
- can be used only for signing or (in addition) also for encryption
  - described in the field keyUsage
  - if it is only for signing then it is required also the phase for server-key exchange
Certificate request
- used for client authentication
- specifies also the list of CAs considered trusted by the server
  - the browsers show to the users (for a connection) only the certificates issued by trusted CAs

Server key exchange
- carries the server public key for key exchange
- needed only in the following cases:
  - the RSA server certificate is usable only for signature
  - anonymous or ephemeral DH is used to establish the master-secret
  - there are export problems that force the use of ephemeral RSA/DH keys
  - Fortezza
- important: this is the only message explicitly signed by the server

Certificate (client)
- carries the certificate for client authentication
- the certificate must have been issued from one CA in the trusted CA list in the Certificate Request message
Client key exchange

- the client generates symmetric keys and send them to the server
- various ways
  - pre-master secret encrypted with the server RSA public key (ephemeral or from its X.509 certificate)
  - public part of DH
  - Fortezza

Certificate verify

- explicit test signature done by the client
- hash computed over all the handshake messages before this one and encrypted with the client private key
- used only with client authentication (to identify and reject fake clients)

Change cipher spec

- trigger the change of the algorithms to be used for message protection
- allows to pass from the previous unprotected messages to the protection of the next messages with algorithms and keys just negotiated
- theoretically is a protocol on its own and not part of the handshake
- some analysis suggest that it could be eliminated
Finished

- first message protected with the negotiated algorithms
- very important to authenticate the whole handshake sequence:
  - contains a MAC computed over all the previous handshake messages (but change cipher spec) using as a key the master secret
  - prevents rollback man-in-the-middle attacks (version downgrade or ciphersuite downgrade)
- different for client and server

---

TLS, no ephemeral key, no client auth

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. client hello (ciphersuite list, client random)</td>
<td>2. server hello (ciphersuite, server random)</td>
</tr>
<tr>
<td></td>
<td>3. certificate (keyEncipherment)</td>
</tr>
<tr>
<td>4. client key exchange (key encrypted for server)</td>
<td>5. change cipher spec (activate protection on client side)</td>
</tr>
<tr>
<td>6. finished (MAC of all previous messages)</td>
<td>7. change cipher spec (activate protection on server side)</td>
</tr>
<tr>
<td></td>
<td>8. finished (MAC of all previous messages)</td>
</tr>
</tbody>
</table>

---

TLS, no ephemeral key, client auth

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. client hello (ciphersuite list, client random)</td>
<td>2. server hello (ciphersuite, server random)</td>
</tr>
<tr>
<td></td>
<td>3. certificate (keyEncipherment)</td>
</tr>
<tr>
<td></td>
<td>4*. certificate request (cert type, list of trusted CAs)</td>
</tr>
<tr>
<td>5*. certificate (client cert chain)</td>
<td>6. client key exchange (encrypted key)</td>
</tr>
<tr>
<td>7*. certificate verify (signed hash of previous messages)</td>
<td>8+9. change cipher spec + finished</td>
</tr>
<tr>
<td></td>
<td>10+11. change cipher spec + finished</td>
</tr>
</tbody>
</table>
### TLS, ephemeral key, no client auth

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. client hello (ciphersuite list, client random) →</td>
<td></td>
</tr>
<tr>
<td>← 3*. certificate (digitalSignature)</td>
<td>← 4*. server key exchange (signed RSA key or DH exponent)</td>
</tr>
<tr>
<td>5. client key exchange (encrypted key or client exponent) →</td>
<td>6. change cipher spec (activate protection on client side) →</td>
</tr>
<tr>
<td>7. finished (MAC of all previous messages) →</td>
<td>← 8. change cipher spec (activate protection on server side)</td>
</tr>
<tr>
<td>← 9. finished (MAC of all previous messages)</td>
<td></td>
</tr>
</tbody>
</table>

### TLS, data exchange and link teardown

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>(… handshake …)</td>
<td></td>
</tr>
<tr>
<td>N. client data (MAC, [ encryption ]) →</td>
<td>← N+1. server data (MAC, [ encryption ])</td>
</tr>
<tr>
<td>… … …</td>
<td></td>
</tr>
<tr>
<td>LAST-1. alert close notify →</td>
<td>← LAST. alert close notify</td>
</tr>
</tbody>
</table>

### TLS, resumed session

<table>
<thead>
<tr>
<th>CLIENT</th>
<th>SERVER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1*. client hello (…, session-id X) →</td>
<td>← 2*. server hello (…, session-id X)</td>
</tr>
<tr>
<td>3. change cipher spec (activate protection on client side) →</td>
<td>← 5. change cipher spec (activate protection on server side)</td>
</tr>
<tr>
<td>4. finished (MAC of all previous messages) →</td>
<td>← 6. finished (MAC of all previous messages)</td>
</tr>
</tbody>
</table>
TLS 1.0 (SSL 3.1)
- Transport Layer Security
- standard IETF:
  - TLS-1.0 = RFC-2246 (jan 1999)
  - TLS-1.0 = SSL-3.1 (99% coincident with SSL-3)
  - emphasis on standard (i.e. not proprietary) digest and asymmetric crypto algorithms; mandatory:
    - DH + DSA + 3DES
    - HMAC
    - ... that is the ciphersuite
      TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA

TLS 1.1 (SSL 3.2)
- RFC-4346 (april 2006)
- to protect against CBC attacks
  - the implicit IV is replaced with an explicit IV
  - padding errors now use the bad_record_mac alert message (rather than the decryption_failed one)
- IANA registries defined for protocol parameters
- premature closes no longer cause a session to be non-resumable
- additional notes added for various new attacks

TLS 1.2 (SSL 3.3)
- RFC-5246 (august 2008)
- ciphersuite specifies also the PRF (pseudo-random function)
- extensive use of SHA-256 (e.g. in Finished, HMAC)
- support for authenticated encryption (AES in GCM or CCM mode)
- incorporates the protocol extensions (RFC-4366) and the AES ciphersuite (RFC-3268)
- default ciphersuite
  TLS_RSA_WITH_AES_128_CBC_SHA
- IDEA and DES ciphersuites deprecated
TLS evolution (I)

- **ciphersuites / encryption:**
  - (RFC-3268) AES
  - (RFC-4492) ECC
  - (RFC-4132) Camellia
  - (RFC-4162) SEED
  - (RFC-6209) ARIA

- **ciphersuites / authentication:**
  - (RFC-2712) Kerberos
  - (RFC-4279) pre-shared key (secret, DH, RSA)
  - (RFC-5054) SRP (Secure Remote Password)
  - (RFC-6091) OpenPGP

TLS evolution (II)

- **compression:**
  - (RFC-3749) compression methods + Deflate
  - (RFC-3943) protocol compression using LZS

- **altro:**
  - (RFC-4366) extensions (specific and generic)
  - (RFC-4681) user mapping extensions
  - (RFC-5748) renegotiation indication extensions
  - (RFC-5878) authorization extensions
  - (RFC-6176) prohibiting SSL-2
  - (RFC-4507) session resumption w/o server state
  - (RFC-4680) handshake with supplemental data

TLS and virtual servers: the problem

- **virtual server (frequent case with web hosting)**
  - different logical names associated to the same IP address
  - e.g. home.myweb.it=1.2.3.4, food.myweb.it=1.2.3.4

- **easy in HTTP/1.1**
  - the client uses the Host header to specify the server it wants to connect to

- **… but difficult in HTTPS**
  - because TLS is activated before HTTP
  - which certificate should be provided? (must contain the server's name)
**TLS and virtual servers: solutions**

- collective (wildcard) certificate
  - e.g. CN=*.myweb.it
  - private key shared by all servers
  - different treatment by different browsers
- certificate with a list of servers in subjectAltName
  - private key shared by all servers
  - need to re-issue the certificate at any addition or cancellation of a server
- use the SNI (Server Name Indication) extension
  - in ClientHello (permitted by RFC-4366)
  - limited support by browsers and servers

**DTLS**

- Datagram Transport Layer Security (RFC-4347)
- applies the TLS concepts to datagram security (e.g. UDP)
- doesn’t offer the same properties as TLS
- competition with IPsec and application security
- example – SIP security:
  - with IPsec
  - with TLS (only for SIP_over_TCP)
  - with DTLS (only for SIP_over_UDP)
  - with secure SIP

**Heartbleed**

- RFC6520 = TLS/DTLS heartbeat extension
  - to keep a connection alive without the need to constantly renegotiate the SSL session (DTLS!)
  - also useful in PMTU discovery
- CVE-2014-0160 = openssl bug (buffer over-read)
  - TLS server sends back more data (up to 64kB) than in the heartbeat request
  - see http://xkcd.com/1354/
- attacker can get sensitive data stored in RAM, such as user+pwd and/or server private key (if not using HSM)
HTTP security

- security mechanisms defined in HTTP/1.0:
  - "address-based" = the server performs access control based on the IP address of the client
  - "password-based" (or Basic Authentication Scheme) = access control based on username and password, Base64 encoded
- both schemas are highly insecure (because HTTP assumes a secure channel!)
- HTTP/1.1 introduces “digest authentication” based on a symmetric challenge
- RFC-2617 “HTTP authentication: basic and digest access authentication”

HTTP - basic authentication scheme

GET /path/to/protected/page
HTTP/1.0 401 Unauthorized - authentication failed
WWW-Authenticate: Basic realm="RealmName"
Authorization: Basic B64_encoded_username_password
HTTP/1.0 200 OK
Server: NCSA/1.3
MIME-version: 1.0
Content-type: text/html
<HTML>protected page ... </HTML>

HTTP digest authentication

- RFC-2069
  - technically obsoleted
  - but considered as base case in RFC-2617
- keyed-digest computation:
  - HA1 = md5 ( A1 ) = md5 ( user":" realm ":" pwd )
  - HA2 = md5 ( A2 ) = md5 ( method ":" URI )
  - response = md5 ( HA1 ":" nonce ":" HA2 )
- server uses a nonce to avoid replay attacks
- the authentication server may insert a field "opaque" to transport state informations (e.g. a SAML token) towards the content server
HTTP - digest authentication scheme

GET /private/index.html HTTP/1.1
HTTP/1.0 401 Unauthorized - authentication failed
WWW-Authenticate: Digest realm="POLITO",
nonce="dcd98b7102dd2f0e8b1ld0f600bfbo0c093",
opaque="5ccc069c6403eabaf9f0171e9517f40e41"
Authorization: Digest username="lioy",
realm="POLITO",
nonce="dcd98b7102dd2f0e8b1ld0f600bfbo0c093",
uri="/private/index.html",
response="32e817757c39b4a508060745385edf",
opaque="5ccc069c6403eabaf9f0171e9517f40e41"
HTTP/1.1 200 OK
Server: NCSA/1.3
Content-type: text/html
<HTML>
pagina protetta ...
</HTML>

HTTP and SSL/TLS

- two approaches:
  - “TLS then HTTP” (RFC-2818 – HTTP over TLS)
  - “HTTP then TLS” (RFC-2817 – upgrading to TLS within HTTP/1.1)
- note: “SSL then HTTP” is in widespread use but it is undocumented
- the two approaches are not equivalent and have an impact over applications, firewall and IDS
- concepts generally applicable to all protocols:
  - “SSL/TLS then proto” vs. “proto then TLS”

SSL client authentication

- via client authentication it’s possible to identify the user that opened the channel (without asking for his username and password)
- some web servers support a (semi-)automatic mapping between the credentials extracted from the X.509 certificate and the users of the HTTP service and/or the OS
Authentication in web applications

- The earlier the access control, the smaller the attack surface
- No need to repeat the authentication (the ID may be propagated)

![Diagram showing the flow of authentication in web applications]

What about forms requesting user/pwd?

- Technically speaking, it’s not important the security of the page containing the form
  - http://www.ecomm.it/login.html
- ... Because the actual security depends on the URI of the method used to send username and password to the server
  - `<form ... action="https://www.ecomm.it/login.asp">`
- ... But psychologically, it is very important the security of the page containing the form because few users have the technical knowledge to verify the URI of the HTTP method used to send user/pwd

E-payment systems

- Failure of the digital cash, for technical and political problems (e.g. the DigiCash failure)
- Currently the most widely used approach is transmitting a credit card number over a SSL channel...
- ... But this is no guarantee against fraud: VISA Europe declares that Internet transactions generate about 50% of the fraud attempts, although they are just 2% of its total transaction amount!
Security of the credit card transactions

- STT (Secure Transaction Technology)
  VISA + Microsoft
- SEPP (Secure Electronic Payment Protocol)
  Mastercard, IBM, Netscape, GTE, CyberCash
- SET = STT + SEPP (Secure Electronic Transaction)

SET

- SET is not a payment system but a set of protocols to use inside an open untrusted network the existing infrastructure for credit card payments
- uses X.509v3 certificates dedicated only to SET transactions
- protects the user privacy because it shows to each part only the relevant and pertinent data

Features of SET

- version 1.0 (may 1997)
- digest: SHA-1
- symmetric encryption: DES
- key exchange: RSA
- digital signature: RSA con SHA-1
- www.setco.org (doesn't exist any more)
A web-based payment architecture

1. offer
2. order
3. redirect
4. enter c.c.
5. credit card
6. c.c. data
7. old
8. payment is OK

virtual
POS

payment
gateway

financial
world

Internet

cardholder

merchant

0.00

Web-based payment architecture

- baseline:
  - the buyer owns a credit card
  - the buyer has a SSL-enabled browser

- consequences:
  - the effective security depends upon the configuration of both the server and the client
  - the payment gateway has all the information (payment + goods) while merchant knows only info about the goods

PCI DSS

- Payment Card Industry Data Security Standard
- required by all credit card issuers for Internet-based transactions
- much more detailed technical prescriptions compared to other security standards (e.g. HIPAA = Health Insurance Portability and Accountability Act)
- https://www.pcisecuritystandards.org
  - v2.0 = oct 2010
  - v3.0 = nov 2013
PCI DSS prescriptions (I)

- **design, build and operate a protected network:**
  - R1 = install and maintain a configuration with firewall to protect access to the cardholders’ data
  - R2 = don’t use pre-defined system passwords or other security parameters set by the manufacturer

- **protect the cardholders’ data:**
  - R3 = protect the stored cardholders’ data
  - R4 = encrypt the cardholders’ data when transmitted across an open public network

PCI DSS prescriptions (II)

- **establish and follow a program for vulnerability management**
  - R5 = use an antivirus and regularly update it
  - R6 = develop and maintain protected applications and systems

- **implement strong access control**
  - R7 = limit the access to the cardholders’ data only to those needed for a specific task
  - R8 = assign a single unique ID to each user
  - R9 = limit physical access to the cardholders’ data

PCI DSS prescriptions (III)

- **regularly monitor and test the networks**
  - R10 = monitor and track all accesses to network resources and cardholders’ data
  - R11 = periodically test the protection systems and procedures

- **adopt a Security Policy**
  - R12 = adopt a Security Policy