SSL/TLS

Ibéria Medeiros

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa

Public Key Certificates

- **certificate**: data structure that associates a public key with a user

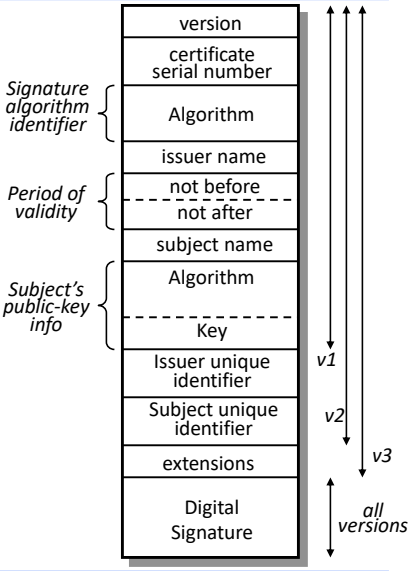
  Ensures:
  - the integrity of the public key (and any other stored information)
  - the correct association between the certificate owner and the public key (and also the rest of the information)

  Examples of certificate formats: X.509, SPKI - Simple Public Key Infrastructure, PGP - Pretty Good Privacy

- **X.509 certificates** : are part of a set of recomendations named X.500, which were developed by the *International Telecommunication Union (ITU)*, that define a directory service
  - the X.509 certificate format is used in a large number of applications, for example, S/MIME, SSL/TLS e SET

Format of a X.509 Certificate

- *serial number:* unique certificate identifier created by the CA
- *issuer name:* X.500 name of the CA
- *subject name:* X.500 name of the user
- *issuer unique identifier:* unique identifier of the CA (*optional, little used)*
- *extensions:* generic mechanism that allows the addition of new fields to the certificate (e.g., extra identifiers)
- *digital signature:* signature made with the private key of the CA (covers all fields of the certificate)



Certification Authority (CA)

- **Function**: creates and associates certificates with a user community; Optionally, can also generate the public-private key pairs
- Notice that the CA is responsible for the certificates for their whole lifetime, namely during creation and revocation

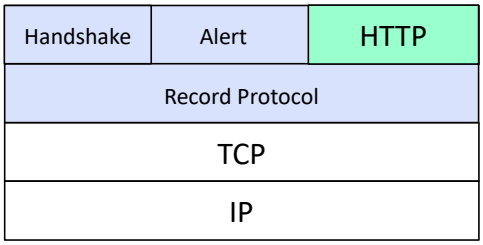
- Creation of a certificate
  - the user sends a request to the CA containing the public key and other personal information
  - the CA checks the personal information (e.g., user identity)
  - the CA creates the certificate, signs it with its private key, and stores it in a database
  - other entities can obtain the certificate, and verify its authenticity and integrity with the public key of the CA

TLS - Transport Layer Security */ SSL - Secure Socket Layer*

- SSL was developed by Netscape; Version 3 (SSLv3) was design with the support of the Internet community and eventually published as a *Internet draft*
- Later on was created the TLS IETF working group, which has the objective of developing the *Internet standard,* whose first version is very similar to SSLv3
- TLS provides a secure channel between two communicating peers, where the only requirement from the underlying transport is a reliable, in-order, data stream
  - ***Authentication***: server is always authenticated; the client side is optionally authenticated; based on asymmetric cryptography (e.g., RSA, ECDSA, EdDSA) or a pre-shared key (PSK)
  - ***Confidentiality***: data encryption + padding
  - ***Integrity***: changes to transmitted data are detected
- These properties should be true even in the face of an attacker who has complete control of the network

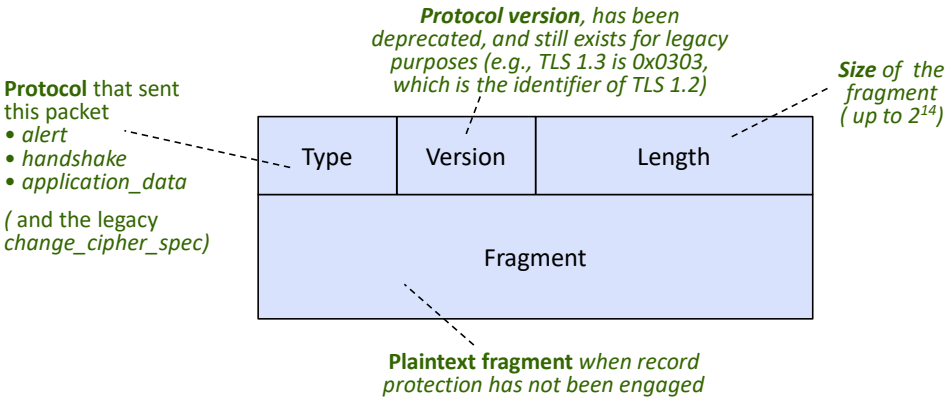
TLS 1.3
(DRAFT VERSION)

TLS - Transport Layer Security / *SSL - Secure Socket Layer*



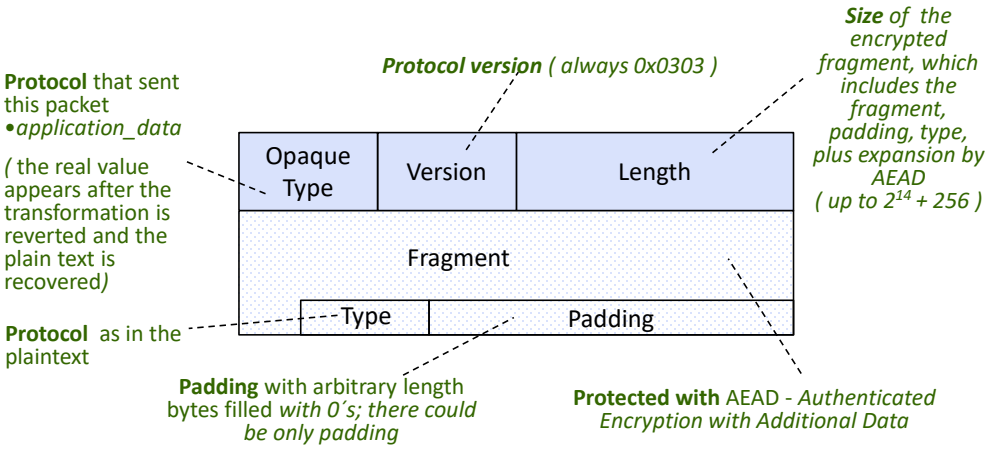
- ***Handshake protocol*** that authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material.
  - designed to resist tampering, where an active attacker should not be able to force the peers to negotiate different parameters
- ***Record protocol*** that uses the parameters established by the handshake protocol to protect traffic between the communicating peers
  - divides traffic up into a series of records, each of which is independently protected using the traffic keys

Record Format Message: Plain Text



Record Format Message: Cipher Text

- When protection is required, the record protocol takes a plain text fragment and transforms it to a cipher text fragment



AEAD - Authenticated Encryption with Additional Data

- These algorithms ensure confidentiality, integrity and authenticity of the data
- They receive not only the data and key, but also some context information

AEADE-protection = *AEAD(key, nonce, additional_data, fragment_plaintex)*

Either the client_write_key or *server_write_key*

(opaque type || version || length) basically protects the header of the fragment

(sequence number ⊕ iv) where sequence number has 64-bits and is initialized at 0 whenever key changes and iv is either the client_write_iv or server_write_iv

Alert Protocol

- Alert messages convey a *description* of the alert and a legacy *level* field that conveyed the severity of the message in previous versions of TLS
  - in TLS 1.3, the severity is implicit in the type of alert being sent, and the *level* field can safely be ignored
- There are ***closure alerts*** that simply notify the other side that no further data will be transmitted through this side of the connection, and ***error alerts*** that cause the connection to be immediately terminated
- Alert messages are encrypted as specified by the current connection state

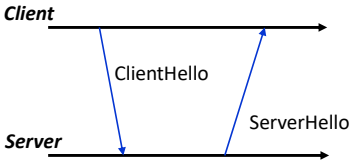
Note: closure alerts are useful to avoid truncation attacks!

Handshake Protocol

- **Function**: allows the authentication of the server and **optionally** of the client, and the negotiation of protocol version and crypto parameters (algorithms and keys)
  - runs before any data is transmitted

- TLS supports three basic key exchange modes:
  - (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves)
  - PSK-only : setup using some secure out of band mechanism or exchanged in a previous connection with the server
  - PSK with (EC)DHE : ensures forward secrecy even with PSK

Phase1: Key Exchange



Establish shared keying material and select the cryptographic parameters; Everything after this phase is encrypted.

- optional messages

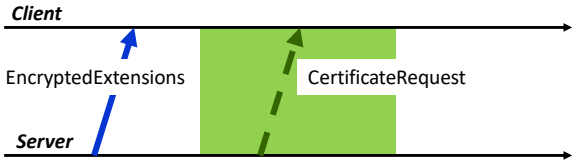
- ClientHello :
  - ***versions*** : list of TLS protocol versions that are supported by the client
  - ***random*** : *nonce* to avoid replay attacks
  - ***cipherSuite*** : list of *symmetric cipher & HKDF* pairs (*HMAC-based Extract-and- Expand Key Derivation Function*)
  - ***key material*** : either the ephemeral *DH key shares* (DH public parameters for different groups) **and/or** *PSK labels* (which identify symmetric keys that should be shared)

Phase1: Key Exchange (cont.)

- ServerHello : selects the crypto parameters
  - ***version*** : largest SSL version supported both by client and server
  - ***random*** : another *nonce* this time generated by the server
  - ***cipherSuite*** : selected crypto algorithms
  - ***key material*** : the ephemeral *DH key share* (for one of the client groups) **and/or** *PSK label* (which indicates the shared key that should be used)

Notice**:** if DH and PSK are used simultaneously, then the derived keys for the protecting of the communications will depend on both keys

Phase 2: Server Parameters



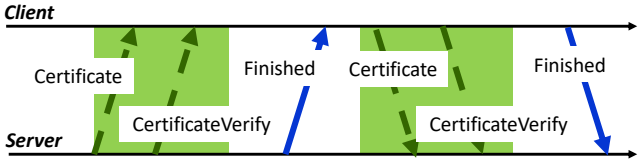
Establish other handshake parameters (whether the client is authenticated, application layer protocol support, etc.)

- optional messages

- encrypted

- EncryptedExtensions :
  - includes responses to the ClientHello extensions that are not required to determine the cryptographic parameters
- CertificateRequest :
  - if required client authentication, asks the client to send its certificates
  - the message has two fields
    - » type of certificate
    - » CAs: list with names of accepted CAs

Phase 3: Authentication



Authenticate the server (and optionally the client) and provide key confirmation and handshake integrity

- optional messages

- encrypted

- Certificate :
  - contains the peer X.509 certificate & supporting chain or a raw public key
  - omitted if PSK is used or if client was not requested to authenticate
- CertificateVerify :
  - signature over entire handshake messages exchanged until this moment using the private key corresponding to the certificate

Signature format: (the use of random in Hellos prevents replay attacks)

encrypt (private key, transcript-hash (Handshake Context, Certificate))

where, transcript-hash *is the hash of concatenation of the included messages*

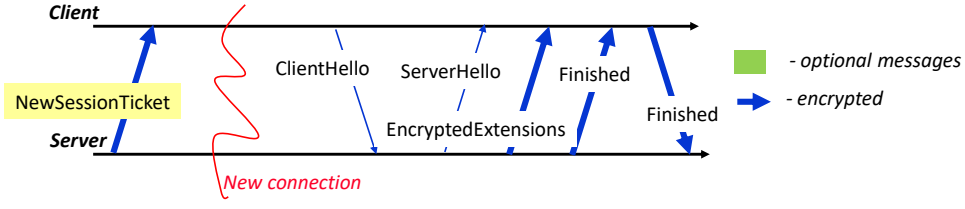
Phase 3 (cont)

Mode	Handshake Context	Base Key for MAC (for Finished)
Server	ClientHello ... later of EncryptedExtensions / CertificateRequest	*server_handshake_traffic_secret*
Client	ClientHello ... later of server Finished	*client_handshake_traffic_secret*

- Finished : terminates the handshake for this peer
  - contains a MAC (Message Authentication Code) over the entire handshake providing
    - » key confirmation,
    - » binds the endpoint's identity to the exchanged keys,
    - » *and* in PSK mode also authenticates the handshake

MAC = MAC (key, transcript-hash(Handshake context || Certificate || CertificateVerify))
NOTE: Certificate and CertificateVerify are only included if they were transmitted

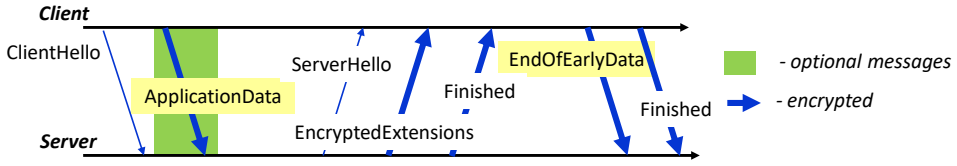
Other Operation Modes: *Resumption*



- Once a handshake has completed, the server can send to the client a **PSK identity** that corresponds to a unique key derived from the initial handshake
- The client can then use that *PSK identity* in future handshakes to negotiate the use of the associated PSK; it can also use (EC)DHE key exchange in order to provide forward secrecy
- This is useful for opening multiple parallel HTTP connections
- NewSessionTicket :
  - includes a *lifetime (max value 1 week)*, a *unique ticket_nonce for all tickets*, and the *ticket* that servers as the PSK identity

  PSK = HKDF(resumption_master_secret, "resumption", ticket_nonce, size_key)

Other Operation Modes: *0-RTT Data*



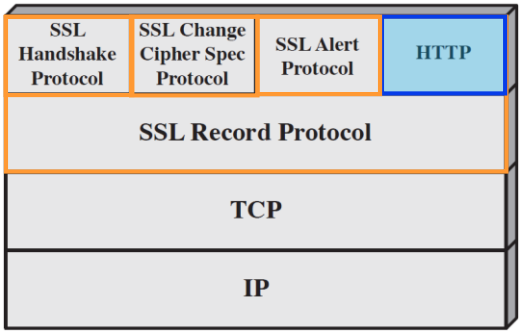
- When clients and servers share a PSK (either obtained externally or via a previous handshake), TLS 1.3 allows clients to send data on the first flight ("early data")
- The client uses the PSK to authenticate the server and to encrypt the early data
- ApplicationData and EndOfEarlyData :
  - ClientHello indicates in an extension the information about crypto parameters and key
  - Messages protected with client_early_traffic_secret from a previous connection
  - No guarantees of non-replay between connections because no new random values are used to generate the key

Note: early data is not forward secret!

TLS 1.2

SSL - Secure Socket Layer / ***TLS - Transport Layer Security***

- Protocol organization



Fundamental Concepts

- ***Session*** : is an association between a client and a server
  - defines a set of cryptographic parameters
  - these parameters can be shared across several ***connections***
  - it is created with the *handshake* protocol
  - in principle there can be several sessions between a client and a server, although this does not occur in practice, since the cryptographic parameters can be re-used in several connections

Parameters :

- **session identifier**: sequence of bytes chosen by the server as identifier
- **peer certificate**: X509.v3 certificate of the peer (can be null)
- **compression method**: compression algorithm (not required)
- **cipher spec**: specifies the hash and encryption algorithms and their attributes
- **master secret**: shared secret between the client and server with 48-bytes
- **is resumable**: flag to indicate if the session can be used to create other connections

Fundamental Concepts (cont)

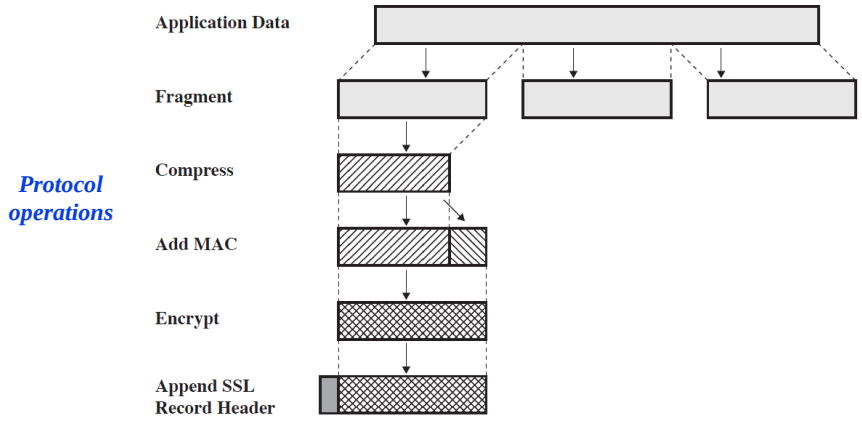
- ***Connection*** : transport level channel between client and server
  - establishes a communication channel between the two peers
  - connections are temporary and are associated with a session

Parameters:

- **Server and client random**: random byte sequences chosen by the peers
- **MAC key of the server**: to protect data sent by the server
- **MAC key of the client**: to protect data sent by the client
- **Encryption key of the server**: to encrypt data from the server
- **Encryption key of the client**: to encrypt data from the client
- **Initialization vectors (IV)** : necessary for the CBC mode of encryption; the last encrypted block serves as the IV of the next message
- **Sequence numbers**: different numbers for each channel direction; they can not exceed the value of $2^{64} - 1$

Record Protocol

- **Function**: ensures the following services in a connection
  - **confidentiality**: based on a symmetric encryption
  - **integrity and authenticity**: based on a MAC



Notes about the operations

- **Fragmentation**: divides the application messages in blocks with a size less or equal to 2^{14} (16384 bytes)
- **Compression**: (optional) the algorithm cannot loose any information of the message, and cannot increase it by more than 1024 bytes
- ***Message authentication code (MAC)*** :

Indirectly protects against replay attacks

*MAC = hash(MAC_secret || pad2 || hash(MAC_secret || pad1|| **seq_num** ||*
SSLCompressed.type || SSLCompressed.length || SSLCompressed.fragment))

protects some of the meta data + data

where:

» *hash* : one of the algorithms MD5 or SHA-1
» *pad1* : byte 0x36 (00110110) repeated 48 times for MD5 and 40 times for SHA
» *pad2* : byte 0x5C (01011100) repeated 48 times for MD5 and 40 times for SHA
» *seq_num* : sequence number for this message
» *SSLCompressed.type, length e fragment* : identifier of the fragmentation algorithm, the length of the compressed fragment, and the compressed fragment

Notes about the operations (cont)

- **Encryption**: the compressed fragment with the MAC is encrypted with a symmetric algorithm

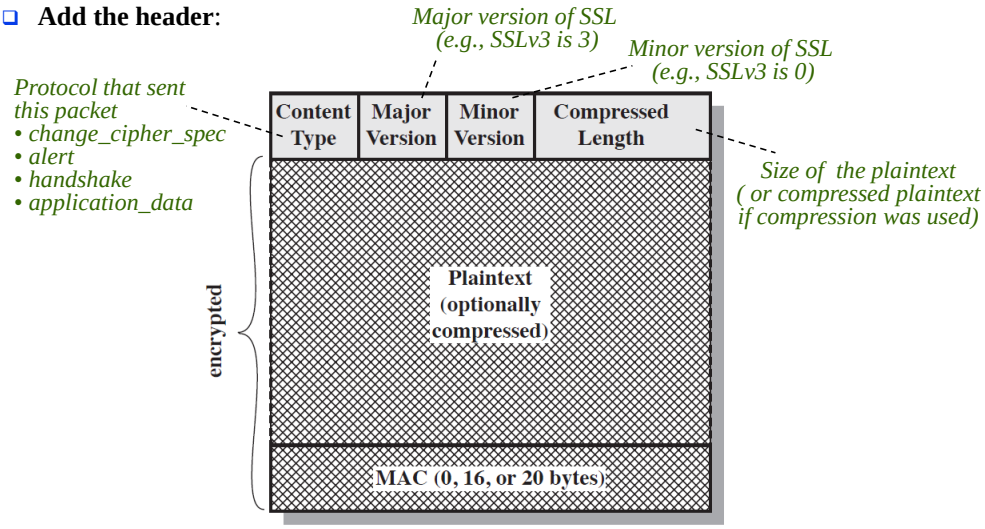
Block Cipher		Stream Cipher	
Algorithm	**Key Size**	**Algorithm**	**Key Size**
IDEA	128	RC4-40	40
RC2-40	40	RC4-128	128
DES-40	40		
DES	56		
3DES	168		
Fortezza	80		
AES	128, 256		

If TLS is incorrectly configured, then you are looking for trouble!

Padding : for block cipher algorithms it is necessary to ensure that the
fragment + MAC + padding + (1 byte) size padding
is a multiple of the cipher block

Notes about the operations (cont)

- **Add the header**:



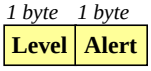
Change Cipher Spec and *Alert* Protocols

- *Change Cipher Spec* : a peer uses this protocol to indicate that is going to change the cryptographic parameters with the pending values (see the *handshake protocol*)

Message format:

- *Alert* : a peer can use this protocol to send alert messages to the other peer

Message format:



Level : has two values, *warning (1)* or *fatal (2)*

Alert : code that indicates the type of alert

Example Alert Codes

- Fatal : causes the termination of the connection and no new connections can be created on the associated session

  unexpected_message : inappropriate message was received

  bad_record_mac : incorrect MAC was received

  handshake_failure : unable to negotiate crypto parameters

  illegal_parameter : incorrect field in message exchanged during handshake

- Warning :

  close_notify : no more messages will be sent through this connection from sender

  no_certificate : cannot provide the requested certificate

  bad_certificate : corrupted certificate

  unsupported_certificate : the received certificate is not supported

  certificate_revoked : the certificate was revoked

Handshake Protocol

- **Function**: allows the authentication of the server and **optionally** of the client, and the negotiation of crypto parameters (algorithms and keys)
  - before any data is transmitted
  - divided in 4 phases with optional messages

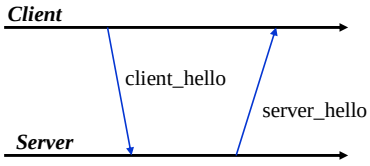
Again, you are looking for trouble!

- Message format:

1 byte	*3 bytes*	*≥0 bytes*
Type	**Length**	**Parameters**

Message Type	Parameters
hello_request	null
client_hello	version, random, session id, cipher suite, compression method
server_hello	version, random, session id, cipher suite, compression method
certificate	chain of X.509v3 certificates
server_key_exchange	parameters, signature
certificate_request	type, authorities
server_done	null
certificate_verify	signature
client_key_exchange	parameters, signature
finished	hash value

Phase1: Establish Security Capabilities



Establish cryptographic algorithms and some configuration parameters

Client_hello:
- ***version*** : largest SSL version understood by the client
- ***random*** : *nonce* to avoid *replay attacks* (4-byte timestamp || 28-bytes random number)
- ***sessionId*** : identifier of the session (= 0 establish a new session/connection; ≠ 0 update crypto parameters or create a new connection)
- ***cipherSuite*** : crypto algorithms that are supported by the client in **decreasing** order of preference
- ***compression method*** : list of supported compression methods

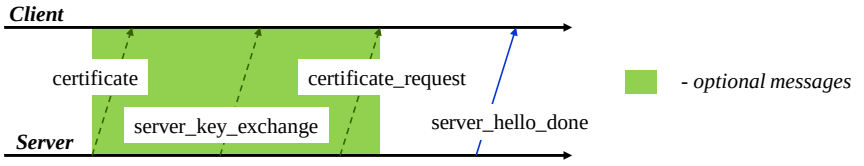
Server_hello:
- ***version*** : largest SSL version supported both by client and server
- ***random*** : another *nonce* this time generated by the server
- ***sessionId*** : new value in case client's id was = 0; otherwise, keeps the same value
- ***cipherSuite*** : selected crypto algorithms
- ***compression method*** : selected compression method

*Opens the door for a **protocol downgrade attack** if server is badly configured*

Main key exchange methods

- RSA : with the **receiver´s RSA public key** – there are two variants, based either *on a RSA encryption or signing key* (called resp. RSA and RSA_EXPORT)*;* there was the need to include the second option to address software export limitations --- a certificate corresponding to a 2048 bits signing key would be transmitted and the key is used to sign a certificate for a (weak) 512 bits encryption key
- Anonymous Diffie-Hellman : each sends his public DH parameters, **without signatures** – *vulnerable to man in the middle attacks*
- Fixed Diffie-Hellman : the **server** has a **signed certificate with the public DH parameters**; the client sends its public parameters either in a **certificate** (if requested by the server) or in one of the exchanged messages – *may result in a **key fixed** for the same peers!!!*
- Ephemeral Diffie-Hellman : the public DH parameters are **signed by the sender**, and certificates are used to provide the public keys to verify the signatures – *more secure because the resulting key is always new!!!*

Phase 2: Server Authentication and Key Exchange



- *certificate* : (not used in Anonymous-DH)
  - server sends its X.509 certificate chain with the RSA public keys (for encryption or signature verification) **or** DH public parameters
- *server_key_exchange* : (not used with Fixed-DH and RSA encryption key)
  - Anonymous-DH : DH parameters (prime q, primitive root a, public value YA)
  - Ephemeral-DH : DH parameters + signature
  - RSA with signing key: new temporary RSA public encryption key + signature

Signature format: (the use of random prevents replay attacks)

encrypt (private key, hash (client_hello.random || server_hello.random || ServerParameters))

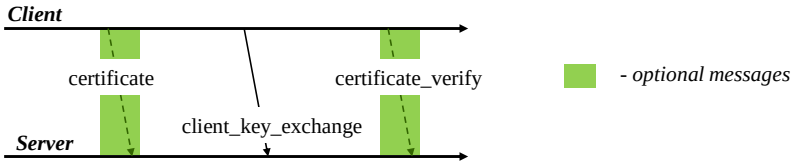
Phase 2 (cont)

- *certificate_request* : (not used with Anonymous-DH)
  - if needed, asks the client to send its certificates
  - the message has two fields
    - » type of certificate:
      - RSA signature only; RSA for Fixed-DH; RSA for Ephemeral-DH
    - » CAs: list with names of accepted CAs
- *server_hello_done* : terminates this phase
  - the message has no parameter

Certificate Validation in a Web Context

- Many attacks have been performed in SSL due to problems related to the **validation of the server (or client) certificate**
- Example tests that need to be carried out
  - the *Subject Distinguished Name* (or one of the extensions like *Subject Alternative Name*) must match the domain of the web server
  - the signature was done correctly with secure algorithms and configurations
  - the certificate has not expired
  - the certificate is being used for the propose that it was made for
  - the CA that made the signature is a trusted CA
  - the certificate has not been revoked (use either a CRL or OCSP)
  - all certificates belonging to the certification path are equally valid

Phase 3: Client Authentication and Key Exchange



- The client starts by validating the server certificate and parameters of *server_hello*
- *certificate* : sends the certificate, in case it was requested by the server; if the client has no certificate, it sends an alert *no_certificate*
- *cliente_key_exchange* :
  - RSA : the client generates a *pre-master secret* with 48-bytes; the value is encrypted with the public key of the server or with the temporary public key
  - Anonymous- or Ephemeral-DH : takes the client DH parameters
  - Fixed-DH : client DH parameters or empty in case the DH parameters were sent in the certificate

Phase 3 (cont)

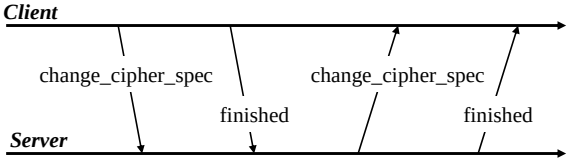
- *certificate_verify* : transmitted only if the client sent a certificate associated with a signing key (i.e., all certificates with the exception of certificates with fixed parameters of DH)
  - allows the server to verify that the client is the real owner of the certificate
  - the client *signs an hash* based on the messages received until this moment

  hash = hfunc(master_secret || pad2 || hfunc(handshake mesg || master_secret || pad1)

  where
  - » *hfunc* : MD5 or SHA-1
  - » *handshake mesg* : sent and received messages with the exception of *client_hello*
  - » *pad1, pad2* : similar to previous pad
  - » *master_secret* : see in front

At this moment, who has performed authentication?

Fase 4: Finish



- The client and server, one after the other, send a message *change_cipher_spec* using the *change cipher spec* protocol, update their cryptographic parameters and send a message *finished* with the **concatenation** of the two hashes, protected with the new cipher algorithms and keys

  MD5(master_secret || pad2 || MD5(handshake msg || sender || master_secret || pad1))
  SHA(master_secret || pad2 || SHA(handshake msg || sender || master_secret || pad1))

  where
  - » *sender* : is the **identifier of the server or client**
  - » *handshake msg* : all previous exchanged messages with the exception of this one

Cryptographic Computations

- Generation of the ***master secret***
  - is a secret with 48-bytes, and is generated from the **pre-master secret**
    - RSA : pre-master secrete created by the client and sent encrypted to the server
    - DH : uses Diffie-Hellman to generate the pre-master secret

master_secret = MD5(pre_master_secret || SHA('A' || pre_master_secret || client_hello.random || server_hello.random)) ||
MD5(pre_master_secret || SHA('BB' || pre_master_secret || client_hello.random || server_hello.random)) ||
MD5(pre_master_secret || SHA('CCC' || pre_master_secret || client_hello.random || server_hello.random))

Cryptographic Computations (cont)

- Generation of cryptographic parameters
  - the master secret is used to generate a **group of secret bits**
  - the number of generated bits must be sufficient to derive keys for: client MAC secret; server MAC secret; encryption key of client and server, and IV of client and server

secret_bits = MD5(master_secret || SHA('A' || master_secret || client_hello.random || server_hello.random)) ||
MD5(master_secret || SHA('BB' || master_secret || client_hello.random || server_hello.random)) ||
MD5(master_secret || SHA('CCC' || master_secret || client_hello.random || server_hello.random)) ||
.....

PITFALLS

What is needed for having secure SSL, namely in the web?

- Certificate potential problems
  - the CA keys have not been compromised
  - the CA does not issue bad certificates, either due to
    - » a malicious or corrupted employee
    - » ineffective validation of the information of the person/organization requiring the certificate
  - the pre-configured certificates in the machine are correct
    - » the browser (or other software) developers select good CA certificates
    - » the browser (or other software) was securely downloaded
    - » the locally stored certificates were not changed (e.g., with a worm)
  - validations are correctly performed on the certificates >>>>
  - compromised certificates are correctly identified
    - » the certificate is revoked once the private key is compromised >>>>
    - » the browser (or other software) duly checks if the certificate was revoked >>>>

What is needed for having secure SSL? (cont)

- Cryptographic algorithms are secure
  - there are no collisions in the hash functions >>>>
  - encryptions/decryptions are not vulnerable >>>>
  - downgrade attacks to use weak crypto primitives >>>>
  - etc …
- In the browser (or other software)
  - the protocol is correctly implemented >>>>
  - does not contain any (remotely) exploitable vulnerabilities >>>>
- The communication might not be between the browser and the server
  - inspection software >>>>
  - CDN for improved performance

GNU security library GnuTLS fails on cert checks [Mar 2014]

According to this Red Hat advisory: "It was discovered that GnuTLS did not correctly handle certain errors that could occur during the verification of an X.509 certificate, causing it to incorrectly report a successful verification. An attacker could use this flaw to create a specially crafted certificate that could be accepted by GnuTLS as valid for a site chosen by the attacker."

As a result, a certificate could be accepted as valid even though it wasn't issued by a trusted CA – for example, an attacker could merely self-sign a bogus certificate.

One commenter in this Reddit thread says the bug is nearly a decade old, having crept into the code back in 2005.

- LINK:
  http://www.theregister.co.uk/2014/03/05/gnu_security_library_gnutls_fails_on_cert_checks_patch_now/

Stolen Code Signing Private Key

SECURITY Mar 19, 2012 4:13 pm

Stolen Encryption Key Compromised Symantec Certificate

By Ellen Messmer, NetworkWorld

When Kaspersky Lab last week spotted code-signed Trojan malware dubbed Mediyes that had been signed with a digital certificate owned by Swiss firm Conpavi AG and issued by Symantec, it touched off a hunt to determine the source of the problem.

SIMILAR ARTICLES:

- How to Protect Yourself From Certificate Bandits
- Digitally Signed Malware Is Increasingly Prevalent, Researchers Say
- Why GlobalSign Made the Right Move to Suspend New Certificates
- F-Secure Finds Malware Signed With Stolen Digital Certificate
- Cybercrime Fight Costing Companies More This Year
- NotCompatible Android Trojan: What You Need to Know

The answer, says Symantec's website security services (based on the VeriSign certificate and authentication services acquisition), is that somehow the private encryption key associated with Conpavi AG certificate had been stolen.

BACKGROUND: Kaspersky Lab spots malware signed with digital certificate

"The private key for Conpavi was exposed," says Quentin Liu, senior director of engineering at the Symantec division. "Someone got hold of the private key." For this type of digital certificate, the private key is held by the certificate owner, in this case, Conpavi. Whether the private encryption key was stolen by an insider at Conpavi or outside attacker isn't known. But the incident points out the risks associated with private encryption keys for this type of digital certificate and the need to safeguard them.

Symantec has revoked the Conpavi certificate that was used to digitally sign the Mediyes malware and is assisting the Swiss firm in analyzing what occurred and helping them prevent this from happening again.

- Used to sign a malware code that intercepts browser requests sent to search engines so the attackers could earn money in a fraudulent pay-per-click scheme
- Not very difficult to think how to extend this to browser code!
- LINK: http://www.pcworld.com/businesscenter/article/252099/stolen_encryption_key_compromised_symantec_certificate.html

Problems with checking revoked certificates

Chrome to weed out dodgy website SSL certificates by itself

Ditched online checks like 'seat belt that snaps when you crash'

By **John Leyden** • **Get more from this author**

Posted in Security, 8th February 2012 17:23 GMT

WIN - A free one year, 25 user licence of Microsoft Office 365!

Google will drop online checks for revoked website encryption certificates in future versions of its Chrome browser after it decided that the process no longer offers any tangible benefits.

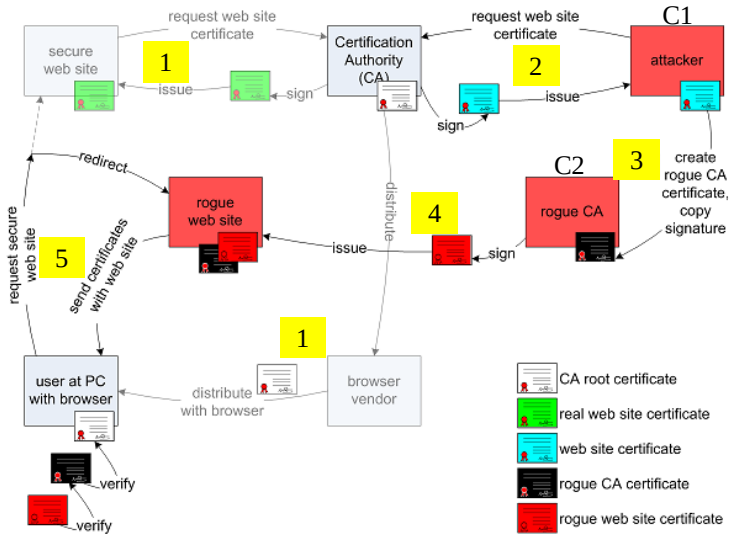
For about a decade now, browsers check the validity of a website's secure sockets layer (SSL) certificate by polling online revocation databases when a user attempts to connect to a secure HTTPS server. A certificate could be cancelled by a Certificate Authority (CA), and thus wind up on a certificate revocation list, if it was faulty or compromised in some way. Cancelling a certificate, or failing to validate it, should therefore warn the visitor to be wary of the site.

However browsers will still establish a connection even if this validation process fails. This behaviour is needed in case users attempt to connect from within heavily firewalled networks, such as public Wi-Fi hotspots and corporate environments: punters might have to sign into an HTTPS site while traffic to other services, including the CAs' verification servers via the online certificate status protocol, are blocked.

Halting access to a HTTPS-secured website if a revocation check failed would leave users unable to connect to sites if the relevant CA was down for any reason, another bad idea.

- When a "**important**" certificate is revoked, a new browser release is pushed out
- Browsers may also call an **online check service** to determine if certificate is OK, but this requires that the answer gets back to the user
- Alternative, the browser can **keep an up-to-date list of revoked certificates**
- LINK: http://www.theregister.co.uk/2012/02/08/chrome_ssl_revocation_checking/

http://www.win.tue.nl/hashclash/rogue-ca/ [Dec 2008]



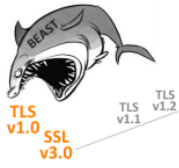
NOTE: attack is possible because certificates C1 and C2 are such that MD5(C1) = MD5(C2) and C2 corresponds to a intermediate CA

BEAST (Browser Exploit Against SSL/TLS) (Sept 2011)

On September 23, 2011 researchers Thai Duong and Juliano Rizzo demonstrated a "proof of concept" called **BEAST** (**Browser Exploit Against SSL/TLS**) using a Java Applet to violate same origin policy constraints, for a long-known Cipher block chaining (CBC) vulnerability in TLS 1.0.[7][8] Practical exploits had not been previously demonstrated for this vulnerability, which was originally discovered by Phillip Rogaway[9] in 2002.

What does the SSL/TLS BEAST exploit mean for my web-based file transfer application?

SEPTEMBER 20, 2011 BY JONATHAN LAMPE



Researchers have discovered a serious vulnerability in TLS v1.0 and SSL v3.0 that allows attackers to silently decrypt data that's passing between a webserver and an end-user browser. This vulnerability can be exploited using a new cookie-based technique called "BEAST" ("Browser Exploit Against SSL/TLS") that takes advantage of block-oriented cipher implementation such as AES and TripleDES.

Which file transfer protocols are affected?

Any interactive HTTPS-based web-based transfer application that relies on SSL/TLS will probably be affected. Web-based "file send" applications will almost certainly be affected. Web services that use cookies to maintain an authenticated session after sign on will also be affected.

At the moment it appears that only protocols that make use of browser cookies are affected. That means that the FTPS and AS2 protocols are safe for now, even if they use TLS v1.0 or SSL v3.0.

POODLE (Oct 2014 and Dec 2014)

Security Labs

SSL 3 is dead, killed by the POODLE attack

Posted by Ivan Ristic in Security Labs on Oct 15, 2014 12:06:32 PM

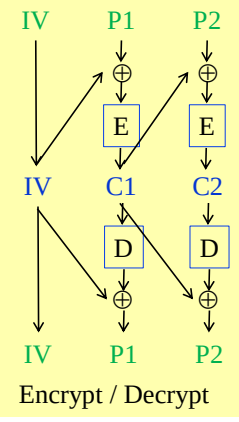
The POODLE Attack (CVE-2014-3566)

Update (8 Dec 2014): Some TLS implementations are also vulnerable to the POODLE attack. More information in this follow-up blog post.

After more than a week of persistent rumours, yesterday (Oct 14) we finally learned about the new SSL 3 vulnerability everyone was afraid of. The so-called POODLE attack is a problem in the CBC encryption scheme as implemented in the SSL 3 protocol. (Other protocols are not vulnerable because this area had been strengthened in TLS 1.0.) Conceptually, the vulnerability is very similar to the 2011 BEAST exploit. In order to successfully exploit POODLE the attacker must be able to inject malicious JavaScript into the victim's browser and also be able to observe and manipulate encrypted network traffic on the wire. As far as MITM attacks go, this one is complicated, but easier to execute than BEAST because it doesn't require any special browser plugins. If you care to learn the details, you can find them in the short paper or in Adam Langley's blog post.

- **POODLE** stands for *Padding Oracle On Downgraded Legacy Encryption*
- Man-in-the-middle exploit which takes advantage of the protocol negotiation phase to downgrade (or fallback) to SSL 3.0
- If attackers successfully exploit this vulnerability, on average, they only need to make 256 SSL 3.0 requests to reveal 1 byte of encrypted messages

Attack on CBC of TLS 1.0



Known:
• Padding: append *p* bytes with value *p-1* and the last byte has the length value *p-1 (e.g., no padding => add at the end 1 byte with 0x00)*
• Attacker: captured two consecutive cipher blocks, C1 and C2

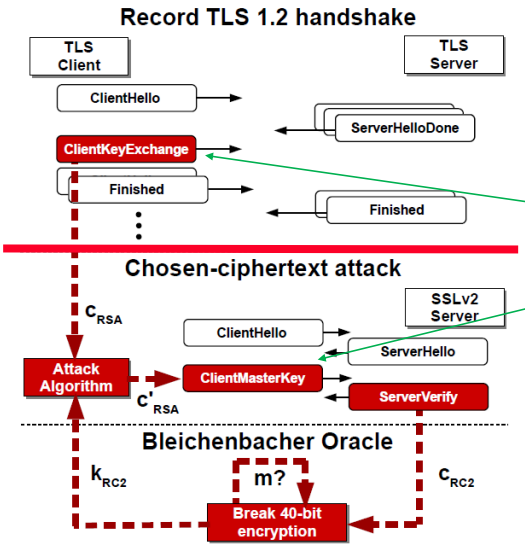
Objective: ***decrypt last byte of block C2 by guessing its value (and checking the guess)***

Method:
- create a fake message with IV* | C_1* | … | C_n*, where the last two blocks
  C_{n-1}* = random bytes | (guess ⊕ 0x00)
  C_n* = C2
- the recipient will decrypt the message by
  $P_n = C_{n-1}* \oplus D(C_n*) = C_{n-1}* \oplus D(C2)$
  = (random bytes | (guess ⊕ 0x00)) ⊕ (some bytes | last_byte(D(C2)))
- if we concentrate on the last byte,
  which is (guess ⊕ 0x00) ⊕ last_byte(D(C2)), then the result is:

➢ 0x00 if guessed correctly (i.e., guess = last_byte(D(C2))), and therefore padding is valid and the returned error is ***bad_record_mac***
➢ incorrect_byte if guessed incorrectly, and therefore padding is invalid and returned error is ***decryption_failed***
• If guessed correctly, then the original decrypted byte of C2 is **[guess ⊕ last_byte(C1)]**

DROWN: Decrypting RSA with Obsolete and Weakened eNcryption (Mar 2016)

Details: D. Aviram et al., *DROWN: Breaking TLS using SSLv2*



TLS 1.2 server and SSLv2 server share the same public/private keys!

1. The attack is probabilistic, where in ~1000 TLS RSA key exchange messages, it is able to decrypt one of them; and is cross protocol as it attacks TLS 1.2 by exploring a problem in SSLv2

2. convert the ClientKeyExchange of TLS that contains a RSA encrypted 48-byte premaster secret **to** a valid ClientMasterKey of SSLv2 also containing a secret, which is encoded in RSA PKCS#1 v1.5 and is vulnerable to attacks based on an *oracle*

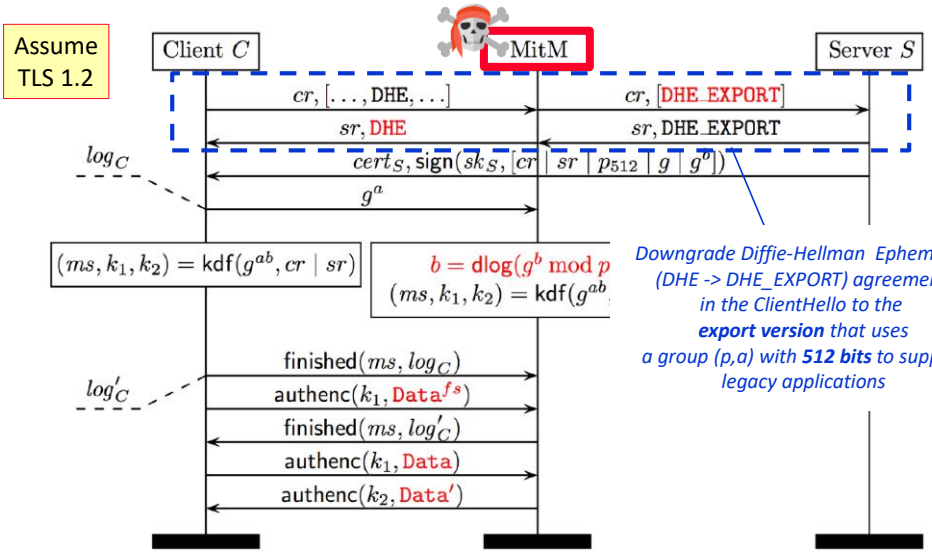
3. perform a modified version of Bleichenbacher's padding oracle attack on the ClientMasterKey, and decrypt the message after ~40000 oracle queries

4. use the decrypted plaintext of the ClientMasterKey to get the content of the ClientKeyExchange, and then decrypt the rest of the TLS messages for that session

Longjam Downgrade Attack

Details: D. Adrian et al., *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*, CCS 2015

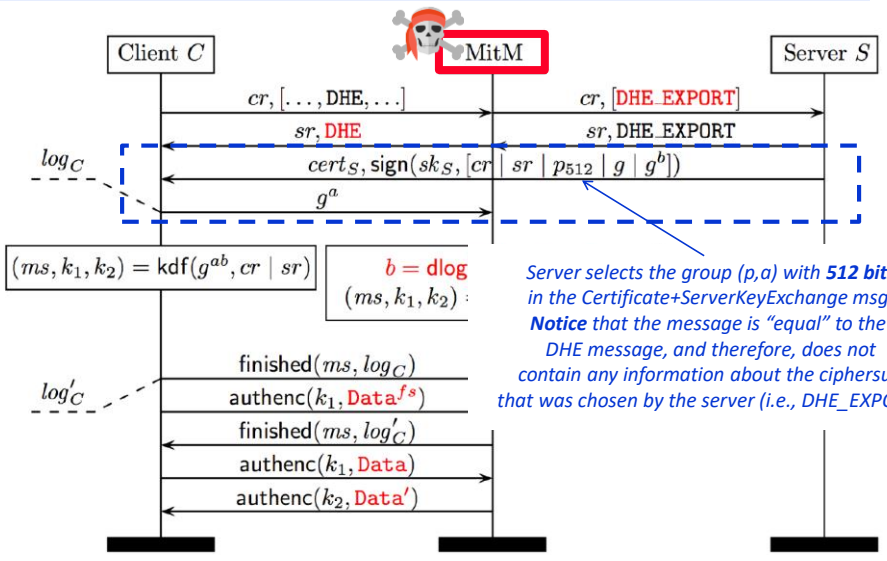
Assume TLS 1.2



*Downgrade Diffie-Hellman Ephemeral (DHE -> DHE_EXPORT) agreement in the ClientHello to the **export version** that uses a group (p,a) with **512 bits** to support legacy applications*

Longjam Downgrade Attack

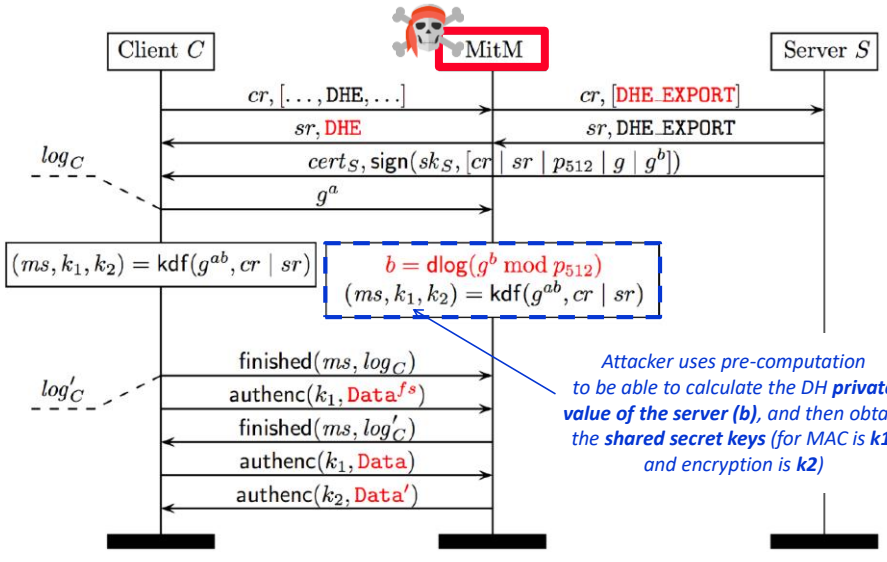
Details: D. Adrian et al., *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*, CCS 2015



*Server selects the group (p,a) with **512 bits** in the Certificate+ServerKeyExchange msg. **Notice** that the message is "equal" to the DHE message, and therefore, does not contain any information about the ciphersuit that was chosen by the server (i.e., DHE_EXPORT)*

Longjam Downgrade Attack

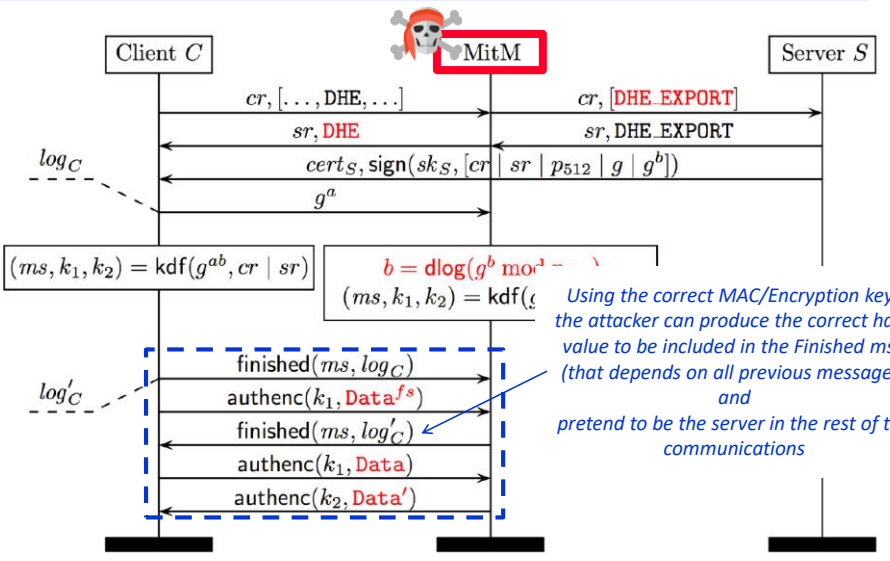
Details: D. Adrian et al., *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*, CCS 2015



*Attacker uses pre-computation to be able to calculate the DH **private value of the server (b)**, and then obtain the **shared secret keys** (for MAC is **k1** and encryption is **k2**)*

Longjam Downgrade Attack

Details: D. Adrian et al., *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*, CCS 2015



Using the correct MAC/Encryption keys the attacker can produce the correct hash value to be included in the Finished msg (that depends on all previous messages) and pretend to be the server in the rest of the communications

Apple Update for OS X Addresses Critical SSL Flaw (Feb 2014)

Vulnerability Summary for CVE-2014-1266

Original release date: 02/22/2014

Last revised: 03/05/2014

Source: US-CERT/NIST

Overview

The SSLVerifySignedServerKeyExchange function in libsecurity_ssl/lib/sslKeyExchange.c in the Secure Transport feature in the Data Security component in Apple iOS 6.x before 6.1.6 and 7.x before 7.0.6, Apple TV 6.x before 6.0.2, and Apple OS X 10.9.x before 10.9.2 does not check the signature in a TLS Server Key Exchange message, which allows man-in-the-middle attackers to spoof SSL servers by (1) using an arbitrary private key for the signing step or (2) omitting the signing step.

This was called the "gotofail" bug because Apple left an extraneous "goto" command in the code that validated SSL certificates

- LINK:

  http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266

  http://www.computerworld.com/s/article/9246593/Apple_patches_critical_gotofail_bug_with_Mavericks_update?taxonomyId=17

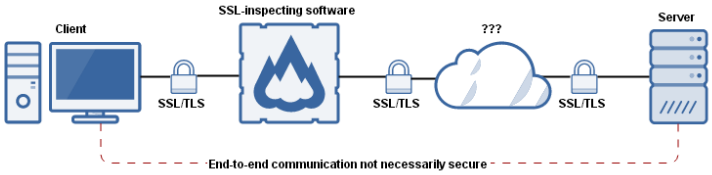
Heartbleed Bug (April 2014)

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

The vulnerability is as a buffer overflow that allows more data can be read than should be allowed in the keep alive message exchange

SSL Inspection Software



- There are network-level applications, devices, and appliances that do SSL inspection (secure web gateways, firewalls, data loss prevention products)
- The client can **verify only** that it is communicating with the SSL-inspecting software
  - the client is unaware of what technique the SSL-inspecting software is using for validating SSL certificates
  - how many additional points exist between the inspecting software and the target
  - because of this lack of transparency, the client must assume that the SSL inspecting software is doing everything perfectly, but unfortunately, this might not be the case

Bibliography

- W. Stallings, Cryptography and Network Security (6th Edition), 2014 : chapter 17 (pag 545 – 563)
- T. Polk, S. Chockhani, K. McKay, Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations, NIST Special Publication 800-52, Revision 1, September 2013
- Kaufman et al, Network Security: Private Communication in a Public World (2 Edition), 2002: chapter 19 (477-498)