The SSL/TLS protocol

Myrto Arapinis
School of Informatics
University of Edinburgh

October 30, 2017
SSL/TLS protocol

Goals: Confidentiality, Integrity, Non repudiation

SSL/TLS use X.509 certificates and hence asymmetric cryptography to exchange a symmetric key. This session key is then used to encrypt subsequent communication. This allows for **data/message confidentiality**, and message authentication codes for **message integrity** and thus, **message authentication**.
SSL/TLS protocol

Google

One account. All of Google.

Sign in to continue to Gmail

Myrto Arapinis
myrto.arapinis@gmail.com

Password

Sign in

Manage accounts on this device

One Google Account for everything Google
SSL/TLS protocol
TCP/IP protocol stack

- TCP/IP provides end-to-end connectivity and is organized into four abstraction layers which are used to sort all related protocols according to the scope of networking involved.
- The SSL/TLS library operates above the transport layer (uses TCP) but below application protocols.
SSL/TLS protocol layers

- **Application Layer**
  - HTTP
  - FTP
  - Telnet
  - Other

- **Handshake Layer**
  - Handshake
  - Change Cipher Spec
  - Alert

- **Record Layer**
  - Record

- **Transport Layer**
  - TCP/IP
SSL/TLS handshake protocol
Client Hello
Server Hello
Certificate

serialNumber: 7377627938844829374
- signature (sha256WithRSAEncryption)
  - issuer: rdnSequence (0)
  - validity
  - subject: rdnSequence (0)
  - subjectPublicKeyInfo
    - algorithm (rsaEncryption)
    - subjectPublicKey: 020210a02828101000aa0c2a0f111b0011132301a5fcdff...
      - modulus: 0x000a00c2a0f111b0011132301a5fcdff7762a8fc0df60a...
      - publicExponent: 65537
  - extensions: 8 items
    - algorithmIdentifier (sha256WithRSAEncryption)
      - AlgorithmId: 1.2.840.113549.1.1.11 (sha256WithRSAEncryption)
www.gmail.com's certificate

Safari is using an encrypted connection to accounts.google.com.
Encryption with a digital certificate keeps information private as it's sent to or from the https website accounts.google.com.

Choose an account

GeoTrust Global CA
Google Internet Authority G2
accounts.google.com

Common Name: Google Internet Authority G2
Serial Number: 7377627938644829374
Version: 3
Signature Algorithm: SHA-256 with RSA Encryption (1.2.840.113549.1.1.1)
Parameters: none
Not Valid Before: Thursday, 6 October 2016 13:59:57 British Summer Time
Not Valid After: Thursday, 29 December 2016 12:28:00 Greenwich Mean Time
Public Key Info
Algorithm: RSA Encryption (1.2.840.113549.1.1.1)
Parameters: none
Public Key: 256 bytes: AA 00 C2 A0 F1 11 B8 01 11 32 30 1A 5F CD FD FF 77 62 A8 FC 0F D6 0A 85 67 FE EF CB FA F7 93 4E 9A CE 49 BC D5 8C 3B 67 B7 4F C6 A7 AB DA F8 EB 04 BA 89 C6 DA 99 EC 3D 42 8C 0E C0 86 0C C4 25 E3 EE 93 AF 17 0C B3 51 88 54 F8 86 71 B8 73 DF B4 CF 0E 3A E1 AB 72 F9 9E 88 78 26 SB AA 4F D8 0D C2 A0 70 79 56 C2 43 07 38 DE BD 15 A5 54 B6 7C A7 AF 1A CD 3B 71 07 AA EC 2E FB 18 EE FE 78 16 A8 5B 37 E4 97 C0 42 D6 00 48 3D 15 64 0B 76 7C B3 8B D9 16 41 3B 4B OD A5 F7 D8 76 D8 7C E7 4B 8A 3E A1 A8 A4 1E 32 58 FB 4C 9D 30 01 FE 7F 90 61 2A EC A3 D1 2A 7D 57 1C 1F A7 E1 48 DF 65 02 75 37 28 C3 52 72 7C B6 5E BE F5 ED 16 D0 BC 7C EF 09 2F 85 9E 57 46 E2 F8 2C D3 ED 4A 14 7A 57 04 E4 07 8E A1 B4 10 FE 27 BC CA 7E E3 E1 B6 27 D7 85 44 52 99 0B 0B 0C 92 3B 47 22 0C E7 19 BD
Exponent: 65537
Key Size: 2048 bits
Key Usage: Encrypt, Verify, Derive

Hide Certificate
OK
Key exchange
Change cipher spec

![Wireshark capture of a Change Cipher Spec message in a TLS handshake]

- **Length**: 70
- **Handshake Protocol**: Client Key Exchange
- **Handshake Type**: Client Key Exchange (16)
  - **Length**: 66
- **EC Diffie-Hellman Client Params**
  - **Pubkey Length**: 65
  - **Pubkey**: 04580c88228565d38a865aa51d1c0de2d75d731d40c5b5b7...

**TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec**

- **Content Type**: Change Cipher Spec (20)
- **Version**: TLS 1.2 (0x0303)
- **Length**: 1

**Change Cipher Spec Message**

```
0000 00 58 56 00 00 01 00 00 03 00 00 04 00 00 05 00 ...]
0010 00 06 07 00 08 09 0a 0b 0c 0d 0e 0f 10 11 12 13 ...
0020 17 2d 80 30 01 02 03 04 05 06 07 08 09 0a 0b 0c ...
0030 0d 10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e ...
0040 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
0050 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
0060 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
0070 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
0080 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
0090 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
00a0 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
00b0 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f ...
```

**TLSv1.2 Record Layer: Handshake Protocol: Multiple Handshake Messages**
A Zoo of TLS attacks

Attacks on TLS that break the intuitive security property of a virtual recreation of a physically secure channel can be categorized along three dimensions.

1. Protocol logic vs. cryptographic design flaw
2. Specification/Standard vs. Implementation errors
3. TLS vs. Context

Flaws in the protocol logic

Attacks targeting the protocol logic may for instance cause the client and server to negotiate the use of weak algorithms even though they both support strong cryptography.

If the faulty negotiation logic conforms to the specification, then the attack is on the specification itself (as, e.g., partially enabled by the False Start modification). If an implementation deviates from the specification to implement a faulty negotiation logic [Dimov, Langley] it is an attack on the implementation. As many aspects of the standard can be underspecified or ambiguous, it is not always possible to distinguish between these two cases.

Another class of protocol logic flaws are state-machine bugs [Early CCS Attack, SMACK Attack].

The attack can also be either an attack on TLS proper, or on its context, e.g. if the attacker can just change the configuration files to deactivate strong cryptography. As the TLS standard does not describe APIs or configuration file formats, context specific attacks are always implementation specific.

The renegotiation attack [TLS Reneg Attack] is a logical attack on the TLS standard, where one peer believes it is running the first handshake on a connection, while the other peer is running a re-handshake. miTLS prevents the renegotiation attack by implementing the renegotiation extension.

More generally, the TLS specification is vague about how applications should handle data coming from consecutive sessions, e.g. whether it is safe to join them and consider them as a single stream, or if the user should be notified of the change of context. The renegotiation extension partially fixes the problem, but it still leaves room for our alert attack, where the attacker can turn any authentic false alert into a warning alert, which gets ignored by default.

Much more seriously, resuming the attacker controlled session on a different connection re-enables the renegotiation attack. This attack is known as the triple handshake attack and is an instance of a larger class of attacks resulting from inadequate channel binding in compound authentication protocols. The miTLS security theorem does not promise channel binding across different connections and is thus not violated by the attack. To be secure, applications making use of miTLS have, however, to be carefully designed to make use of the provided security cues. We give a basic HTTPS client, miHTTPS, as an example for such an application. Subsequently the flaw is also being patched at the TLS level using a new extension.

Cryptographic design flaws

Attacks exploiting cryptographic design flaws may simply result from cryptanalytic progress against the cryptographic building blocks of TLS. They can, however, also result from improper non-blackbox use of otherwise secure cryptographic constructions. An example for this is chosen ciphertext chaining (CBC) mode encryption. Early versions of TLS allow using knowledge of the next initialization vector (IV) to set up adaptive plaintext attacks, see, e.g. OpenSSL archive for a first mention of the ‘BEAST attack’. 
SSL/TLS renegotiation

Client and server are allowed to initiate renegotiation of the session encryption in order to:

- Refresh keys
- Increase authentication
- Increase cipher strength
- ...

Client or server can trigger renegotiation by sending a hello message
SSL/TLS renegotiation weaknesses

- Renegotiation has priority over application data!
- Renegotiation can take place in the middle of an application layer transaction!

Incorrect implicit assumption: the client doesn’t change through renegotiation
Marsh Ray’s plaintext injection attack on HTTPS

Attacker:
GET /pizza?toppings=pepperoni;address=attacker
str HTTP/1.1
X-Ignore-This:(no carriage return)

Victim:
GET /pizza?toppings=sausage;address=victim
str HTTP/1.1
Cookie:victim
cookie

Result:
GET /pizza?toppings=pepperoni;address=attacker
str HTTP/1.1
X-Ignore-This:GET /pizza?toppings=sausage;address=victim
str HTTP/1.1
Cookie:victim
cookie

⇒
Server uses victim’s account to send a pizza to attacker!
Marsh Ray’s plaintext injection attack on HTTPS

Attacker:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)

Victim:
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_str

Result:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_str

⇒
Server uses victim's account to send a pizza to attacker!
Attacker:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)

Victim:
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie

⇒
Server uses victim’s account to send a pizza to attacker!
Marsh Ray’s plaintext injection attack on HTTPS

Attacker:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)

Victim:
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie

Result:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie

⇒
Server uses victim’s account to send a pizza to attacker!
Marsh Ray’s plaintext injection attack on HTTPS

Attacker:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:(no carriage return)

Victim:
GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie

Result:
GET /pizza?toppings=pepperoni;address=attacker_str HTTP/1.1
X-Ignore-This:GET /pizza?toppings=sausage;address=victim_str HTTP/1.1
Cookie:victim_cookie

⇒ Server uses victim’s account to send a pizza to attacker!
Anil Kurmus’ plaintext injection attack on HTTPS
Anil Kurmus’ plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=

Victim:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password...

⇒

the attacker gets the user name and password!
Anil Kurmus’ plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=
Anil Kurmus’ plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=

Victim:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password...
Anil Kurmus’ plaintext injection attack on HTTPS

Twitter status updates using its API by posting the new status to http://twitter.com/statuses/update.xml, as well as the user name and password

Attacker:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password
User-Agent: curl/7.19.5
Host: twitter.com
Accept:*/*
Content-Length: 140
Content-Type: application/x-www-form-urlencoded
status=

Victim:
POST /statuses/update.xml HTTP/1.1
Authorization: Basic username:password...

⇒ the attacker gets the user name and password of the victim!
The SAML Single Sign On (SSO) protocol
SAML SSO protocol
SAML SSO protocol
SAML SSO protocol
SAML SSO protocol (OASIS 2005)

A1. C, IdP, AuthReq(ID, SP), URI

A2. C, IdP, AuthReq(ID, SP), URI

IdP builds an authentication assertion

A3. Response(ID, SP, IdP, {AA}_{K_{idp}^{-1}}), URI

A4. Response(ID, SP, IdP, {AA}_{K_{idp}^{-1}}), URI

S2. Resource
Google’s SAML-based Single Sign-On for Google Applications deviates from the above protocol for a few, seemingly minor simplifications in the messages exchanged:

**G1.** *ID* and *SP* are not included in the authentication assertion, *i.e.* \( AA = \text{AuthAssert}(C; IdP) \) instead of \( \text{AuthAssert}(ID; C; IdP; SP) \);

**G2.** *ID*, *SP* and *IdP* are not included in the response, *i.e.* \( Resp = \text{Response}\{\{AA\}\_{K_{ldP}}^{-1}\} \) instead of \( \text{Response}(ID; SP; IdP; \{AA\}\_{K_{ldP}}^{-1}) \).
Attack Google’s SSO implementation

[A. Armando, R. Carbone, L. Compagna, J. Cullar, L. Tobarra, "Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for google apps", (FMSE'08)]

![Diagram showing an attack on Google's SSO implementation.](image)

Legend:
- $A \xrightarrow{M_{ch}} B$: $A$ sends $M$ on $ch$ confidential to $B$
- $A \xrightarrow{M_{ch}} B$: $A$ sends $M$ on $ch$ authentic for $A$
- $A \xrightarrow{M_{ch}} B$: $M$ is sent on $ch$ authentic for $A$ and confidential to $B$
SAML SSO protocol (OASIS 2012)

C → IdP → SP

S1. GET URI

A1. HTTP 302 IdP?SAMLRequest=AuthReq(ID, SP)&RelayState=URI

SAML Authentication Protocol

A2. GET IdP?SAMLRequest=AuthReq(ID, SP)&RelayState=URI

A3. HTTP 200 Form(...)

IdP builds an authentication assertion

\[ AA = AuthAssert(ID, C, IdP, SP) \]

A4. POST SP?SAMLResponse=Response(ID, SP, IdP, \{AA\}_{K_{IdP}})&RelayState=URI

S2. HTTP 200 Resource(URI)
Attack SAML SSO protocol (OASIS 2012)

[A. Armando, R. Carbone, L. Compagna, J. Cullar, G. Pellegrino, A. Sorniotti, ”From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure?”, Chapter in Future Challenges in Security and Privacy for Academia and Industry]

⇒ XSS attack on SAML-base SSO for Google Apps