Cours de cryptologie appliquée de l'EPITA
TLS - partie 2

Manuel Pégourié-Gonnard
mpg@elzevir.fr

ARM France - IoT - mbed TLS

23 novembre 2015

`https://github.com/mpg/cours-tls`

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

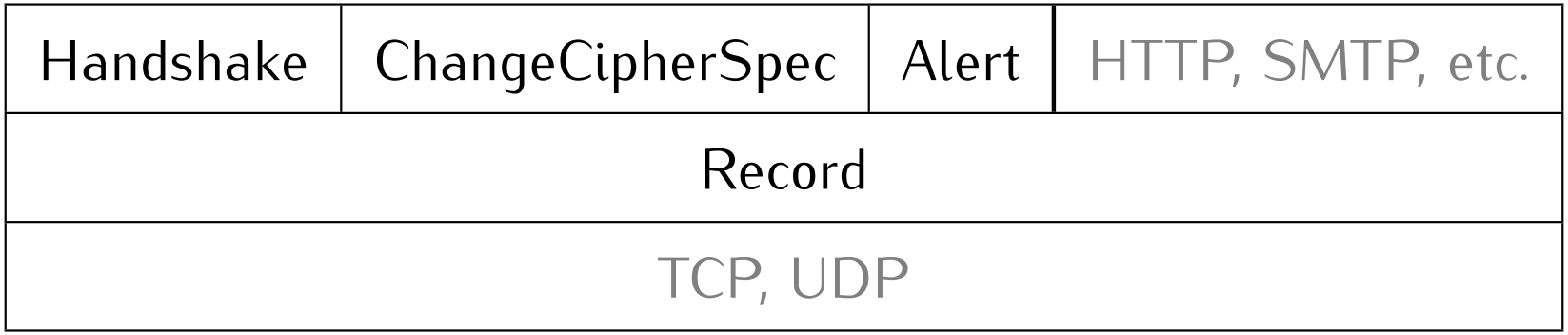
Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Rappel



Handshake

- Négociation *fiable* des paramètres
- Établissement de clé de session *secrètes*
- *Authentification* du serveur, voire du client
- Crypto (en général) asymétrique + symétrique

Record

- *Confidentialité* et *intégrité* des données
- Crypto symmétrique uniquement

Flot général

```
ClientHello                --------->
                                                    ServerHello
                                                   Certificate*
                                             ServerKeyExchange*
                                            CertificateRequest*
                           <--------           ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished#                  --------->
                                              [ChangeCipherSpec]
                           <--------                   Finished#
Application Data#          <--------->     Application Data#
```

* = optionel, # = chiffré

Encodage des messages

T	L1	L2	L3	*(contenu)*

```
struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        [...]
    } body;
} Handshake;

enum {
    client_hello(1), server_hello(2), [...]
    finished(20), (255)
} HandshakeType;
```

ClientHello

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;

struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
opaque SessionID<0..32>;
uint8 CipherSuite[2];
enum { null(0), (255) } CompressionMethod;
```

ServerHello

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;
enum {
    signature_algorithms(13), (65535)
} ExtensionType;
```

Remarques sur les Hello

Négociation

- Le client propose, le serveur dispose
- Le serveur DOIT ignorer silencieusement les valeurs inconnues (version, suites, compression, extensions)

Suites

- TLS_*KEYEX*_WITH_*CIPHER_HASH*
- Ex recommandé : TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- OpenSSL a des noms différents, ex AES128-SHA
- Plus de 300 définies, seulement une poignée recommandée
- `https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4`

Dérivation des clés

- L'échange de clé fourni un secret pré-maître (PMS) de longueur variable
- On en dérive un secret maître (MS) de 48 octets :

```
MS = PRF(PMS, "master secret",
         ClientHello.random + ServerHello.random)
         [0..47];
```

- La fonction pseudo-aléatoire (PRF) est basée sur des hash (MD5+SHA1, SHA-256) et est utilisée comme KDF
- Un bloc de clés est dérivé du MS en appliquant la PRF
- Il est découpé en clés et IV (client/serveur, AES/HMAC)
- RFC 7627 et TLS 1.3 remplacent les random par *session hash* (hash de toute la poignée de main) (cf *triple handshake*)

Le message Finished

```
struct {
    opaque verify_data[verify_data_length];
} Finished;

verify_data
   PRF(master_secret, finished_label, Hash(handshake_messages))
      [0..verify_data_length-1];
```

- En pratique `verify_data_length` est 12 octets
- Le but est d'assurer la fiabilité de la négociation : un attaquant qui essaie d'influencer la négociation sera détecté
- Sert aussi de confirmation de clé, ce qui simplifie les preuves de sécurité

Serveur intolérants et attaque en downgrade

- Des serveurs codés avec les pieds rejettent les ClientHello avec une version « trop » élevée, ou avec des extensions
- Certaines middlebox font de même
- Les clients qui veulent quand même communiquer dans ce cas font un *fallback* avec un ClientHello.version plus faible
- Ce fallback hors-protocole n'est pas protégé par le Finished
- Un serveur intolérant est indistiguable d'un attaquant
- Ce fallback n'est donc *pas sûr* (ex. partie DLE de POODLE)
- On a un compromis utilisabilité-sécurité (pour changer !)
- Le RFC 7507 améliore la situation avec une *signaling ciphersuite value* (SCSV) indiquant le fallback

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Principe

```
ClientHello                -------->
                                                ServerHello
                                                Certificate
                           <--------       ServerHelloDone
ClientKeyExchange
[ChangeCipherSpec] ...
```

```
  struct {
      ProtocolVersion client_version;
      opaque random[46];
  } PreMasterSecret;
  struct {
      public-key-encrypted PreMasterSecret pre_master_secret;
  } EncryptedPreMasterSecret;
  struct {
      EncryptedPreMasterSecret;
  } ClientKeyExchange;
```

L'attaque de Bleichenbacher (1)

Oracle de padding sur PKCS#1 v1.5

- Rappel : $\mathrm{RSA\_PKCS1\_15}(m) = \mathrm{RSA}(\mathrm{PKCS1\_15}(m))$
- $\mathrm{PKCS1\_15}(m) =$ | 00 | 02 | pad | 00 | m |
- Autrefois, les serveurs envoyaient une erreur spécifique si le padding était incorrect
- L'attaquant envoie un chiffré c' arbitraire, observe la réponse
- Si pas d'erreur de padding (une fois sur environ 2^{16}), alors $m' = \mathrm{RSA}^{-1}(c')$ commence par 00 02.
- Autrement dit, dans ce cas, $2 \cdot 2^{l-16} \leq m' < 3 \cdot 2^{l-16}$ où l est la longueur en bits de la clé.

Propriété algébrique de RSA

RSA est un homomorphisme : $\mathrm{RSA}(m_1 m_2) = \mathrm{RSA}(m_1)\,\mathrm{RSA}(m_2)$

L'attaque de Bleichenbacher (2)

Principe de l'attaque

- On connaît c et on va trouver m en consultant l'oracle
- On pose $B = 2^{l-16}$, on sait déjà que $m \in [2B, 3B-1]$
- On cherche s_1 tel que $c \cdot s_1^e$ soit accepté par l'oracle ; on a

$$m \cdot s_1 \mod n \in [2B, 3B-1]$$
$$m \cdot s_1 \in \cup_{r \in \mathbb{N}} [2B + rn, 3B - 1 + rn]$$
$$m \in \cup_{r \in \mathbb{N}} \left[\frac{2B + rn}{s_1}, \frac{3B - 1 + rn}{s_1} \right] \cap [2B, 3B-1]$$

- On itère (voir gribouillis au tableau)
- Au total quelques millions de requêtes suffisent : faisable

L'attaque de Bleichenbacher (3)

Comment traiter EncryptedPreMasterSecret

- Naïf : si mauvais padding, erreur ; si mauvaise version, erreur

L'attaque de Bleichenbacher (3)

Comment traiter EncryptedPreMasterSecret

- Naïf : ~~si mauvais padding, erreur~~ ; si mauvaise version, erreur
- Attaque de Bleichenbacher (1998) : oracle de padding
- TLS 1.0 : Si mauvais padding, on génère un PMS aléatoire, erreur reportée au Finished ; (si mauvaise version erreur)

L'attaque de Bleichenbacher (3)

Comment traiter EncryptedPreMasterSecret

- Naïf : ~~si mauvais padding, erreur~~ ; ~~si mauvaise version, erreur~~
- Attaque de Bleichenbacher (1998) : oracle de padding
- TLS 1.0 : Si mauvais padding, on génère un PMS aléatoire, erreur reportée au Finished ; (~~si mauvaise version erreur~~)
- Attaque de Klima (2003) : oracle de version
- TLS 1.1 : Si mauvais padding, on génère un PMS aléatoire et on l'utilise ; si mauvaise version pareil

L'attaque de Bleichenbacher (3)

Comment traiter EncryptedPreMasterSecret

- Naïf : ~~si mauvais padding, erreur~~ ; ~~si mauvaise version, erreur~~
- Attaque de Bleichenbacher (1998) : oracle de padding
- TLS 1.0 : Si mauvais padding, on génère un PMS aléatoire, erreur reportée au Finished ; (~~si mauvaise version erreur~~)
- Attaque de Klima (2003) : oracle de version
- TLS 1.1 : ~~Si mauvais padding, on génère un PMS aléatoire et on l'utilise~~ ; si mauvaise version pareil
- TLS 1.2 : On génère un PMS aléatoire ; si mauvais padding ou mauvaise version, ou l'utilise
- Usenix 2014 : new Bleichenbacher side channels
- Voir `ssl_parse_encrypted_pms()` dans mbed TLS

L'attaque de Bleichenbacher (3)

Comment traiter EncryptedPreMasterSecret

- Naïf : ~~si mauvais padding, erreur~~ ; ~~si mauvaise version, erreur~~
- Attaque de Bleichenbacher (1998) : oracle de padding
- TLS 1.0 : Si mauvais padding, on génère un PMS aléatoire, erreur reportée au Finished ; (~~si mauvaise version erreur~~)
- Attaque de Klima (2003) : oracle de version
- TLS 1.1 : ~~Si mauvais padding, on génère un PMS aléatoire et on l'utilise~~ ; si mauvaise version pareil
- TLS 1.2 : On génère un PMS aléatoire ; si mauvais padding ou mauvaise version, ou l'utilise
- Usenix 2014 : new Bleichenbacher side channels
- Voir `ssl_parse_encrypted_pms()` dans mbed TLS
- La « vraie » solution serait d'utiliser OAEP

Concept de (Perfect) Forward Secrecy

Problème avec le transport de clé RSA

- Un attaquant enregistre plein de communications
- Plus tard, il obtient la clé RSA du serveur (faille sur le serveur, national security letter, factorisation, ...)
- Il peut alors déchiffer toutes les communications enregistrées

Forward Secrecy

- C'est quand l'attaque précédente n'est pas possible :-)
- Offerte par les suites basées sur Diffie-Hellman

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Rappel : crypto sur les courbes elliptiques

Log discret dans les corps finis (FFDL)

- $s = g^a \mod p$, trouver a
- Échange de clé : (FF)DH
- Signatures : DSA
- Complexité $\approx 2^{c\sqrt[3]{n}}$

Log discret dans les courbes elliptiques (ECDL)

- $S = a \cdot G \in E$, trouver a
- Échange de clé : ECDH
- Signatures : ECDSA
- Complexité $\approx 2^{n/2}$

Factorisation des entiers

- $n = pq$, trouver p et q
- Signature : RSA
- Chiffrement : RSA
- Complexité $\approx 2^{c\sqrt[3]{n}}$

Tailles équivalentes

Sym.	FF/RSA	ECC
112	2432	224
128	3248	256
256	15424	512

DHE-RSA, ECDHE-RSA, ECDSA

```
ClientHello (+ext)   -------->
                                                ServerHello
                                                Certificate
                                          ServerKeyExchange
                               <--------      ServerHelloDone
Certificate
ClientKeyExchange
[ChangeCipherSpec] [...]
```

- ServerKeyExchange contient :
  - les paramètres choisis (p et g pour FFDH, ou un identifiant de courbe elliptique)
  - la part du serveur g^a ou aG
  - la signature du tout (plus les random) par le clé associée au certificat du serveur (authentification du serveur)
- ClientKeyExchange contient la part g^b ou bG du client

Propriétés de (EC)DHE-(RSA|ECDSA)

- Les trois échanges offrent la *forward secrecy*
- Le « E » de (EC)DHE signifie éphémère : les parties génèrent un nouvel exposant secret à chaque fois
- DHE-RSA est environ trois fois plus coûteux que RSA côté serveur, et 20 fois plus côté client
- ECDHE-RSA est seulement 15% plus cher que RSA côté serveur (3 fois côté client)
- Chiffres de 2011, les courbes elliptiques progressent encore
- ECDSA est moins cher que RSA côté serveur, mais plus cher côté client
- ECDHE-(RSA|ECDSA) sont les plus recommandés
- ECHDE-RSA est actuellement le plus courant car les certificats ECDSA ne sont pas disponibles depuis longtemps

Problèmes avec FFDH (1)

Validation des paramètres

- Un serveur malicieux peut choisir des mauvais paramètres
- Le client ne peut pas les valider (trop coûteux)
- Utilisé dans une variante de l'attaque *triple handshake*
- Solution : draft-ietf-tls-negotiated-ff-dhe

Taille des paramètres

- Certains clients (Java 6) ne supportent pas plus de 1024 bits
- Le serveur ne peut pas savoir
- Compromis sécurité-interopérabilité
- Solution : draft-ietf-tls-negotiated-ff-dhe

Problèmes avec FFDH (2)

Réutilisation des paramètres

- Après un gros précalcul pour un p donné, casser des log discrets $\mod p$ devient très rapide (cf. `https://weakdh.org/` point 2)
- C'est probablement tout juste possible pour des gouvernements de casser quelques p de 1024 bits.
- Beaucoup de serveurs ont un p de 1024 bit en commun :
  - Le plus courant est utilisé par 18% des serveurs HTTPS
  - Les deux plus courants donnent 60% des VPN et 26% des serveurs SSH
- Solutions : utiliser un p assez grand, ou unique

Plus généralement sur les attaques groupées, voir :
`http://blog.cr.yp.to/20151120-batchattacks.html`

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Principe

- Le serveur présente un certificat contenant une clé publique et une identité (et des méta-donnés)
- Il prouve qu'il possède la clé privée associée en déchiffrant le PMS ou en signant le ServerKeyEchange avec cette clé
- L'association entre l'identité et la clé est garantie par une autorité de certification (CA) qui a signé le certificat
- Cette CA peut elle-même avoir été signée par une autre CA, mais on ne fait que déplacer le problème
- Au bout d'un moment on doit arriver à une racine de confiance
- Il suffit de distribuer quelques racines de confiance avec les navigateurs pour authentifier des millions de sites web

Premiers problèmes

Il faut vérifier les certificats (nom et signature)

- Le nom dans le certificat doit correspondre au nom attendu
- Le certificat doit être signé correctement
- La liste des racines de confiance doit être censée et à jour
- Certaines implémentations ne vérifient rien par défaut
- Certains développeurs sont juste paresseux, ou se disent que c'est pas grave pour les tests, puis oublient en prod

Aparté : les trois états, dont un maudit, en sécurité

1. Ça ne marche pas (et c'est pas sécurisé)
2. Ça marche mais c'est pas sécurisé
3. Ça marche et c'est sécurisé

Premiers problèmes

Il faut vérifier les certificats (nom et signature)

- Le nom dans le certificat doit correspondre au nom attendu
- Le certificat doit être signé correctement
- La liste des racines de confiance doit être censée et à jour
- Certaines implémentations ne vérifient rien par défaut
- Certains développeurs sont juste paresseux, ou se disent que c'est pas grave pour les tests, puis oublient en prod

Aparté : les trois états, dont un maudit, en sécurité

1. Ça ne marche pas (et c'est pas sécurisé)
2. ~~Ça marche mais c'est pas sécurisé~~ JAMAIS! mieux vaut 1.
3. Ça marche et c'est sécurisé

Le problème de la révocation

Quand une clé est compromise, le certificat associé doit être révoqué.

Les solutions qui marchent moyen

- Certificate revocation list (CRL) : volume, latence
- Online Certificate Status Protocol : que faire si le serveur ne répond pas ? Attaque ou simple problème de disponibilité ?
- Les CAs n'ont pas d'incitation économique directe à investir beaucoup ici dans la révocation

De meilleures solution partiellement déployées

- OCSP stapling : le serveur attache une réponse OCSP dans la poignée de main TLS
- CRLsets : les fabricants de navigateurs distribuent des diffs de CRL (Chrome)

Les autorités « de confiance »

Le maillon faible

- Plus d'une centaine d'autorités dans les navigateurs
- Chacune peut certifier n'importe quel nom
- Le total est aussi fiable que la *moins* fiable des racines !

Les autorités ne sont pas toujours fiables

- En 2011, DigiNotar est compromis et émet des centaines de certificats frauduleux, qui seront utilisé pour MitM plus de 300000 utilisateurs iraniens de Gmail.
- Racines retirées des navigateurs, compagnie en faillite
- Plusieurs incidents similaires au cours des années
- Problème : certains acteurs sont *too big to fail*
- Problème : a-t-on détecté tous les incidents ?

Solutions partielles émergentes (1)

DANE (RFC 6698)

- Distribution de certificates ou clés via le DNS
- S'ajoute aux CA traditionnelles ou les remplace
- Problème majeur : dépend de DNSSEC !

Certificate Transparency (RFC 6962)

- Tous les certificats sont publiés dans un journal append-only vérifiable (arbre de Merkle)
- Les serveurs doivent fournit des preuves d'inclusion, ou bien les clients peuvent bavarder entre eux
- Tout le monde peut inspecter les journaux
- N'évite pas les problèmes, permet seulement de les détecter

Solutions partielles émergentes (2)

HTTP Public-Key Pinning (7469)

- Le serveur peut imposer des clés qui devront être présentes dans la chaine lors des prochaines connections
- Complique l'administration lors du changement des clés
- Ne marche que pour HTTPS
- Ne sécurise pas la première connection
- En fait si pour certains sites et certains navigateurs (pinning pré-configurés)

Rapports de forces

Exemple récent : l'affaire Google vs Symantec
`https://googleonlinesecurity.blogspot.fr/2015/10/sustaining-digital-certificate-security.html`

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Le problème square-and-multiply

```
a = 1
for i from 0 to bitlength(b) - 1 do:
    if b_i == 1 then:
        a = a * g
    a = a^2
```

- RSA, (EC)DH et ECDSA comportent tous une étape cruciale qui s'implémente naïvement par square-and-multiply
- Le temps d'éxécution, à taille d'exposant donné, fuite directement le poids de Hamming de l'exposant secret
- Si on la corrige en square-and-multiply-always, si on ne fait pas gaffe on fuite la longueur de l'exposant secret (pour ECDSA c'est catastrophique)
- Des attaques plus sophistiquées peuvent observer les variations en fonction des valeurs de g

C'est exploitable en pratique

1996 « Timing attacks on [...] RSA [...] » en local uniquement

2003 « Remote timing attacks are practical »

- Récupère la clé RSA complète du serveur
- Mis en pratique sur un réseau local

2011 « Remote timing attacks are still practical »

- Récupère une clé ECDSA (sur certaines courbes dans OpenSSL)
- Mis en pratique sur un réseau local

2009 « Opportunities and limits of remote timing attacks » : 100 ns sur un réseau local, 15–100 micro secondes sur l'internet

Autres attaques et défenses

Autres canaux auxiliaires

- Attaque par cache, en local ou à distance
- Avec un peu d'accès physique :
  `https://www.tau.ac.il/~tromer/acoustic/`
  `https://www.tau.ac.il/~tromer/handsoff/`
- Avec plus d'accès physique : puissance, ondes EM, etc.

Défense de base : codage en temps constant

- Pas de branche qui dépend d'une donnée secrète
- Pas d'accès mémoire dont l'adresse dépend d'un secret
- Pas d'instruction à temps variable sur des secrets
- `https://cryptocoding.net/`

Autre stratégie de défense : le *blinding*

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Autres échanges de clé

Clé pré-partagé : PSK

- Sans crypto asymétrique, pour les environnement *très* restreints
- Plus de travail pour la gestion des clés (il FAUT une clé par client)
- Pas de forward secrecy

Divers, peu utilisés

- (EC)DH non-ephemère : léger gain de perf, perte de la FS
- DH_anon : léger gain de perf, vulnérable à un MitM
- DSS : aka DSA, nécessite certificat spécifique
- RSA_export, DH_export : limité à 512 bits, cf diapo suivante

Attaques par downgrade sur les suites export

FREAK (2015)

1. L'attaquant modifie le ClientHello : RSA_export seulement
2. Le serveur offre alors une clé RSA de 512 bits
3. La clé RSA de 512 bits est cassée par l'attaquant (100$)
4. L'attaquant peut modifier les Finished pour effacer ses traces
5. Solution évidente : désaciver RSA_export !
6. C'est lent : `https://freakattack.com/`

Logjam (2015)

- Attaque similaire avec DH_export
- Un pré-calcul pour un p donné permet de casser DH en direct pendant la poignée de mains

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Attaques génériques

Stripping

- L'attaquant supprime une redirection HTTP →HTTPS
- L'attaquant supprime l'annonce de STARTTLS
- `http://www.thoughtcrime.org/software/sslstrip/`
- Solution partielle pour HTTP : HTTP-Strict-Transport-Security (RFC 6797)

Dénis de service

- Contre le serveur : l'attaquant envoie un ClientHello puis ne répond plus. Le serveur calcule une signature et un demi DH.
- Contre un tiers, en DTLS : un serveur mal configuré peut être utilisé comme amplificateur ; les cookies DTLS sont conçus pour l'empêcher.

Renégociation

Consiste à effectuer une deuxième poignée de mains au sein d'une même connection (rafraichissement des clés, certificat client).

Renégociation non sécurisée (2009)

- Les deux poignées de main ne sont pas liées cryptographiquement
- Un attaquant peut injecter du traffic avant le début d'une connection légitime
- Solution : RFC 5746

Triple handshake (2014)

- Attaque sophistiquée permettant à un serveur de « voler » l'authentification du client auprès d'un autre serveur
- Trouvées en essayant de prouver TLS sûr…
- Solution : RFC 7627

Bugs dans les implémentations

Sans doute plus de 80% des failles de sécurité (au doigt mouillé) !
Quelques exemples célèbres :

- La faille Debian/OpenSSL : de 2006 à 2008, mauvais patch, nombres aléatoires générés avec seulement 16 bits d'entropie
- Hearbleed (OpenSSL) : de 2012 à 2014, buffer overread, fuite d'information silencieuse, potentiellement clé secrète `http://heartbleed.com/`
- Early CCS (OpenSSL) : de 2012 à 2014, erreur de machine à état, MitM possible `https://www.imperialviolet.org/2014/06/05/earlyccs.html`
- Double goto fail d'Apple : (2014) `if(...) goto fail; goto fail; if(...)` ; permet d'utiliser des certificats sans leur clé

Liste non exhaustive !

Poignée de mains : généralités

Transport de clé RSA

Échange de clé Diffie-Hellman (elliptique)

Infrastructure à clé publique X.509

Attaques par canaux auxiliaires

Autres échanges de clé

Autres attaques

Pratique

Pour la prochaine séance

- Télécharger mbed TLS `https://tls.mbed.org/`
- Le compiler (cf Readme)
- Lire `https://tls.mbed.org/high-level-design`
- Lire `https://tls.mbed.org/kb/how-to/mbedtls-tutorial`