UNIVERSITÉ LAVAL

Travail pratique N^o 3

Cours : GLO-3100 Cryptographie et sécurité informatique

1 OpenSSL (2.5 points)

Dans cet exercice, nous allons utiliser la machine virtuelle kali du premier TP pour découvrir d'autres utilisations de OpenSSL. Voici quelques commandes de `openssl`, mais vous devrez les compléter par vous-même pour répondre aux questions demandées.

— Générer une clé privée RSA de "`size`" bits (512, 1024, etc.)
```
$openssl genrsa -out <fichierRsa.priv> <size>
```
— Création d'un clé publique associée à la clé privée "`fichierrRsa.priv`"
```
$openssl rsa -in <fichierrRsa.priv> -pubout -out <fichierRsa.pub>
```
— Chiffrer une clé privée avec l'algorithme DES3 ou autre.
```
$openssl rsa -in <fichierRsa.priv> -des3 -out <fichierOut.pub>
```
— Chiffrer le "`fichier.txt`" avec l'algorithme "`algo`" en utilisant la clé qui se trouve dans la première ligne du fichier "`key`".
```
$openssl enc -algo -in <fichier.txt> -out <fichier.enc> -kfile <key>
```
— Déchiffrer le "`fichier.enc`" avec l'algorithme "`algo`".
```
$openssl enc <-algo> -in <fichier.enc> -d -out <fichier.txt> -kfile <key>
```
— Hacher un fichier avec "`algo`" (sha1, md5, rmd160, etc.)
```
$openssl dgst <-algo> <entree> -out <sortie>
```
— Générer un nombre aléatoire sur "`nbits`" et mettre le résultat dans "`file.key`"
```
$openssl rand -out <file.key> <nbits>
```

On vous demande de faire les opérations suivantes et de prendre des captures d'écran montrant les commandes utilisées et les résultats obtenus :

1. (0.25pts) Générer une clé privée RSA pour Alice tout en la chiffrant avec triple DES (`des3`) en utilisant un mot de passe comme clé symétrique. La clé RSA générée devrait être mise dans le fichier `alice-privatekey.pem`.
2. (0.25pts) Extraire la clé publique de Alice et la mettre dans le fichier `alice-publickey.pem`
3. Faire la même chose pour Bob pour lui générer une paire de clés et les mettre dans `bob-privatekey.pem` et `bob-publickey.pem`.
4. Créer un fichier `message.txt` dans lequel vous écrivez votre nom et votre prénom.
5. (0.5pts) Montrer, via une capture d'écran, les opérations `openssl` qu'Alice fait pour envoyer `message.txt` haché avec SHA1 et signé avec sa clé privée.
6. (0.25pts) Montrer, via une capture d'écran, les opérations *openssl* que Bob fait pour vérifier la signature.
7. Pour qu'Alice envoie `message.txt` d'une manière confidentielle à Bob, il exécute les étapes suivantes :
   a) (0.25pts) Alice génère une clé aléatoire et mets le résultat binaire dans le fichier `key.bin`.
   b) (0.25pts) Alice chiffre `message.txt` avec `key.bin` en utilisant AES-128-CBC et met le résultat dans `protected-message.txt` codé en base64.
   c) (0.25pts) Alice chiffre, en utilisant `rsautl`, la clé `key.bin` avec la clé publique de bob et met le résultat dans `protected-key.bin`
8. (0.5pts) Aider Bob à retrouver le contenu de `message.txt` à partir de `protected-key.bin` et `protected-message.txt`. Montrer les étapes de calculs, via des captures d'écran `openssl`, ainsi que le message.txt obtenu une fois le calcul terminé.

2 Techniques d'authentification et les attaques MITM

L'authentification est omniprésente en sécurité informatique. En effet, toute opération sensible ne devrait se faire qu'après une authentification sécuritaire de son auteur. L'objectif de cet exercice est d'implanter quelques techniques d'authentification et d'analyser leurs vulnérabilités vis-à-vis des attaques MITM (*Man In The Middle*). Pour les deux protocoles qui suivent, les composantes des messages échangés sont séparées par des espaces. Le client est désigné par C et le serveur par S. La notation " $\alpha.\quad A \longrightarrow B : \quad m$ " veut dire qu'à l'étape α du protocole l'utilisateur A envoie à l'utilisateur B le message m. La clé privée d'un utilisateur A est notée par k_a^{-1} et sa clé publique par k_a. La signature d'un message m par une clé privée de A est notée par $\{m\}_{k_a^{-1}}$. Le chiffrement d'un message m par une clé symétrique k est noté par $\{m\}_k$. Le hachage d'un message m par une fonction de hachage H_i est noté par $H_i(m)$.

2.1 Authentification via des défis/réponses basés sur des mots de passe (2.25 points)

Le protocole suivant est inspiré de HTTP-Digest-Authentication (RFC-2617). L'échange entre le client et le serveur se fait selon les phases suivantes :

— **Enregistrement (0.5 pt) :**

$$\begin{array}{lllll} E1. & C & \longrightarrow & S & : \quad user\ \ pwd \\ E2. & S & \longrightarrow & C & : \quad 200 \end{array}$$

Pour des raisons de simplicité, nous supposons que le serveur détient, dans un fichier, des enregistrements contenant les informations suivantes : $user$ et $H_1(user : pwd)$.
Pour ne pas envoyer le mot de passe en clair, cet enregistrement devrait se faire à l'intérieur d'un tunnel TLS/SSL. Mais, pour des raisons de simplicité, nous ne créons pas de tunnels durant ce TP.

— **Authentification (1.5 pts) :** pour accéder à une ressource, le client doit prouver son identité comme suit :

$$\begin{array}{lllll} A1. & C & \longrightarrow & S & : \texttt{GET}\ URI \\ A2. & S & \longrightarrow & C & : \texttt{401 Unautorized}\ N_s\ sessionID \\ A3. & C & \longrightarrow & S & : user\ \ N_s\ \ N_c\ \ H_1(H_1(user : pwd) : N_s : N_c : H_1(\texttt{GET} : URI))\ \ SessionID \\ A4. & S & \longrightarrow & C & : code \end{array}$$

— à l'étape $A1$, l'utilisateur demande l'accès à une ressource identifiée par un URI (pour des raisons de simplicité, nous supposons, pour ce TP, que l'URI est toujours égal à : /dir/index.html).
— à l'étape $A2$, le serveur informe le client qu'il n'est pas autorisé à accéder à cette ressource sans authentification en lui envoyant le code `401` suivie de la chaine `Unautorized` suivie d'un nombre aléatoire N_s (qui joue le rôle d'un défi) et de numéro de session $sessionID$.
— à l'étape $A3$, l'utilisateur répond au défi du serveur en envoyant son $user$, le nonce du serveur N_s, son nonce à lui N_c, $H_1(H_1(user : pwd) : N_s : N_c : H_1(\texttt{GET} : URI))$ et $sessionID$.
— à l'étape $A3$, le serveur recalcule $H_1(H_1(user : pwd) : N_s : N_c : H_1(\texttt{GET} : URI))$ et la compare avec la valeur de $H_1(H_1(user : pwd) : N_s : N_c : H_1(\texttt{GET} : URI))$ reçue. Si les deux valeurs sont identiques, le serveur retourne un code 200 à l'étape $A4$ accompagné de la ressource demandée (que nous allons ignorer dans ce TP pour des raisons de simplicité). Sinon, il retourne le code `401`.

— **Intrus (0.25 pt) :** Nous supposons que l'exécution de ce protocole se fait en présence d'un intrus qui a le contrôle du réseau. Il a la capacité de se placer entre le client et le serveur (MITM) pour lire et éventuellement modifier tous les messages échangés. Pour simuler le comportement de cet intrus, votre programme doit donner la possibilité à l'utilisateur de lire et de modifier tout message échangé entre le client le serveur. À chaque fois qu'un message est envoyé durant une étape donnée, votre programme le montre à l'utilisateur et lui demande s'il veut le modifier. Si oui, le programme lit la nouvelle valeur fournie par l'utilisateur et la transmet à la destination comme message de l'étape courante.

Remarque : Pour ce genre de protocoles, un pirate qui réussit à se placer entre le client et le serveur (MITM) peut essayer des attaques passives par dictionnaire. Ayant les valeurs $user$, N_s, N_c et `GET URI` qu'il a interceptées en claire, il peut essayer des mots de passe d'un dictionnaire en calculant $H_1(H_1(user : pwd) : N_s : N_c : H_1(\texttt{GET} : URI))$ et la comparer au hachage intercepté à la troisième étape. Si elles sont égales, cela veut dire que le mot de passe choisi du dictionnaire correspond à celui de l'utilisateur en question. Par ailleurs, si la base de données du serveur est attaquée, le pirate pourrait récupérer $user$ et $H_1(user : pwd)$ et les utiliser pour se connecter au nom de $user$.

2.2 Authentification via des clés publiques (4.25 points)

Dans ce qui suit nous considérons une version simplifiée du protocole WebAuthn (Fido). Un utilisateur peut interagir avec différents serveurs $S_1, \ldots, S_2$ (exemples *www.gmail.com*, *www.facebook.com*, etc.). Pour simplifier l'implantation, nous supposons, dans ce qui suit, que S joue le rôle de tous les serveurs en même temps.

— **Enregistrement (1.5 pts) :** À chaque fois qu'un client s'enregistre chez un serveur, il génère une paire de clés fraîche (une clé publique désignée par k_c et une clé privée désignée par k_c^{-1}). Le client s'enregistre en fournissant son $user$ et sa clé publique k_c ainsi que le domaine du serveur ($domain$). Le serveur sauvegarde le $user$ et sa clé publique du client dans un fichier associé à *domain* (en d'autres termes, à chaque domaine, S détient ses propres utilisateurs dans un fichier séparé.).

$$\begin{array}{llllll} E1. & C & \longrightarrow & S & : & domain\ \ user\ \ k_c \\ E2. & S & \longrightarrow & C & : & domain\ \ N_s \\ E3. & C & \longrightarrow & S & : & domain\ \ \{H_2(N_s)\}_{k_c^{-1}} \\ E4. & S & \longrightarrow & C & : & domain\ \ code \end{array}$$

À l'étape E_1, l'utilisateur envoie une demande d'enregistrement à $domain$ en fournissant son nom d'utilisateur $user$ et une clé publique k_c. Pour s'assurer que le client est bel et bien le propriétaire de la clé publique en question, le serveur lui envoie un défi (un nombre aléatoire N_s fraîchement généré). À l'étape E_2, le client fait la preuve qu'il est le propriétaire de la clé privé associé à k_c en signant $H_2(N_s)$ et en la retournant au serveur à l'étape E_3. À la dernière étape, le serveur retourne le $code$ 200, confirmant le succès de l'authentification, si la signature est correcte. Autrement, il retourne 400 comme $code$.
Les clés privées du client sont sauvegardées localement chez lui et elles ne sont accessibles que via un mot de passe de son choix. À chaque fois que le client a besoin d'utiliser une clé privée, l,application lui demande de fournir son mot de passe. Autrement, pour chaque $domain$ et pour chaque $user$, le client détient un enregistrement dans sa trousse de clés contenant $domain$, $\{user\}_k$ et $\{k_c^{-1}\}_k$. Le $domain$ reste en clair, mais le $user$ la clé privée k_c sont chiffrés séparément par une clé k générée en appliquant une fonction de hachage H_1 sur le mot de passe (pwd) de l'utilisateur (c.-à-d. $k = H_1(pwd)$).

— **Authentification (1.5 pts) :** Une fois l'enregistrement fait, le client peut s'authentifier autant de fois qu'il veut au serveur pour faire certaines opérations (visualiser sa page, envoyer un courriel, etc.). L'authentification se fait d'une manière similaire à l'enregistrement comme le montrent les étapes suivantes. Mais, durant l'authentification, le client n'a plus besoin de soumettre sa clé publique puisqu'elle est déjà connue par le serveur.

$$\begin{array}{llllll} A1. & C & \longrightarrow & S & : & domain\ \ sessionID\ \ user \\ A2. & S & \longrightarrow & C & : & domain\ \ sessionID\ \ N_s \\ A3. & C & \longrightarrow & S & : & domain\ \ sessionID\ \ \{H_2(N_s)\}_{k_c^{-1}} \\ A4. & S & \longrightarrow & C & : & domain\ \ sessionID\ \ code \end{array}$$

— **Transactions (1pt) :** Le client peut répéter autant qu'il veut les opérations suivantes.

$$\begin{array}{llllll} T1. & C & \longrightarrow & S & : & domain\ \ sessionID\ \ operation \\ T2. & S & \longrightarrow & C & : & domain\ \ sessionID\ \ operation\ \ N'_s \\ T3. & C & \longrightarrow & S & : & domain\ \ sessionID\ \ \{H_2(operation.N'_s)\}_{k_c^{-1}} \\ T4. & S & \longrightarrow & C & : & domain\ \ sessionID\ \ code \end{array}$$

À l'étape T_1, le client envoie une $opration$ au serveur. Le serveur demande une preuve que l'opération provient de détenteur du compte $user$ en lui envoyant un défi N'_s à l'étape $T2$. Comme preuve, le client retourne la signature du hachage de l'opération concaténée avec le défi ($H_2(operation.N'_s)$) (le point désigne une concaténation simple sans laisser un espace) à l'étape T_3. Si la preuve est correcte, le serveur exécute l'opération et lui retourne un code 200 sinon, il lui retourne un code 400.

— **Intrus (0.25 pt) :** Comme pour le protocole précédent, votre programme doit donner la possibilité à l'utilisateur de lire et de modifier tout message échangé entre le client le serveur.

Remarque : Pour ce genre de protocole, il est difficile pour un pirate de récupérer des informations utiles même s'il se place au milieu. Par ailleurs, même si la base de données du serveur est piratée, la connaissance de son contenu ne devrait pas causer un problème de sécurité. Cependant pour ce protocole, comme pour les précédents, il faut tout le temps s'assurer de l'intégrité de la base de données. Autrement, un pirate qui peut changer la clé publique (ou le mot de passe, ou le mot de passe haché) d'un client par sa propre clé (son mot de passe ou son mot de passe haché) réussira à exécuter des commandes au nom de ce client.

2.3 Autres détails d'implantation

— $domain$ est une chaine de caractères (maximum 50 caractères). Exemples *www.facebook.com* et *www.gmail.com*.
— $user$ est une chaine de caractères (maximum 20 caractères).
— $sessionID$, N_s, N'_s et N_c sont des entiers aléatoires ayant 5 chiffres au maximum.
— $operation$ est un texte fourni par l'utilisateur (maximum 50 caractères) comme ENVOYER UN COURRIEL, LIRE COURRIELS, INVITER UN AMI, etc. Pour ce travail, ces textes ne seront pas analysés par le serveur (pour des raisons de simplicité).
— La fonction de hachage H_1 est MD5.
— La fonction de hachage H_2 est SHA256.
— Pour protéger ses clés privées dans sa trousse le client utilise AES-128 avec le mode CTR.
— Les clés privées et publiques sont de type RSA de 1024 bits.

2.4 Travail demandé

- Votre programme démarre avec le menu suivant :
  ```
  1 : HTTP-Digest
  2 : WebAuthn
  3 : Quitter
  ```
- Une fois, l'utilisateur sélectionne le protocole, il aura le choix d'enregistrer un nouveau compte ou de passer directement à l'authentification.
  ```
  1 : Enregistrer un nouveau compte
  2 : Authentification
  3 : Menu précédent
  4 : Quitter
  ```
- Après une authentification réussie de WebAuthn, on demande à l'utilisateur s'il souhaite faire une opération via le menu suivant :
  ```
  1 : Faire une opération
  2 : Visualiser la trousse de clés
  3 : Menu précédent
  4 : Quitter
  ```
  S'il choisit de faire une opération, il doit fournir une chaîne de caractère indiquant le nom de cette opération. Le menu donne la possibilité aussi de visualiser la trousse de clés déchiffrée par le mot de passe de l'utilisateur.
- Durant chaque étape du protocole, votre programme doit montrer l'échange (nom de l'étape, l'émetteur, receveur et le message envoyé). Par exemple :

$$A1.\quad C \longrightarrow S : \quad www.gmail.com\ \ 12345\ \ Alice$$

Les composantes hachées ou signées des messages doivent apparaitre en format Base64.

- Avant que le message arrive à la destination, le programme donne la possibilité à l'utilisateur de le modifier (pour simuler le rôle de l'intrus).
- À tout moment, l'utilisateur devrait avoir la possibilité de revenir au menu précédent ou bien de quitter le programme.
- Lors d'un enregistrement, votre application donne la possibilité à l'utilisateur de choisir le domaine, le *user* et le mot de passe de la trousse de clé. Le programme dy client `WebAuthn` est celui qui choisit la paire de clés (publique/privé) et la protège avec le mot de passe de l'utilisateur.
- Pour le protocole `WebAuthn`, à chaque fois que le programme client a besoin d'accéder à la trousse, il demande à l'utilisateur le mot de passe permettant de déchiffrer l'information recherchée.

3 Remarques

1. Le travail est individuel.
2. Les langages C, C++ et Java sont permis.
3. Le barème est à titre indicatif et que 10% des points de l'exercice 2 sont réservés aux commentaires.
4. Attention au plagiat ! Faites vos TPs par vous-même.

4 À remettre

— Utiliser le site web du cours pour remettre un seul fichier ".zip" (de taille maximale 40 Mb) qui porte votre nom au complet et qui contient un répertoire par exercice (ne m'envoyez pas vos TPs par courriels s.v.p.).
— Pour l'exercice 1, il faut retourner un fichier ".pdf" ou ".doc" contenant les réponses avec les mêmes numéros que les questions. Les captures d'écran doivent être bien lisibles.
— Pour l'exercice 2, il faut fournir l'exécutable aussi bien que le code source **bien commenté**. Assurez-vous aussi que votre exécutable n'aura besoin d'autres fichiers externes pour qu'il fonctionne sur Kali.

5 Échéancier

Le 12 décembre 2020 avant 14h00. Une pénalité de 0,035% de la note sera appliquée à chaque minute de retard (l'équivalent 2,1% par heure), et ce, pour un maximum de 48 heures. Après 48 heures de retard, la note sera zéro. Par exemple, pour un étudiant qui a eu 9 points, mais avec 5 heures de retard, sa note sera 9 - 9*0,021*5=8,05.