UNIVERSITÉ LAVAL

Travail pratique N^o 2

Cours : GLO-3100 Cryptographie et sécurité informatique

© M. Mejri, 2016

1 OpenSSL (30 points)

Pour cet exercice, nous vous demandons d'utiliser la machine virtuelle kali du premier TP. 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.)
  `$openssl genrsa -out <fichierRsa.priv> <size>`
- Création d'un clé publique associée à la clé privé "`fichierrRsa.priv`"
  `$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. (3pts) 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. (3pts) 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. (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. (5pts) 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) (3pts) Alice génère une clé aléatoire et mets le résultat binaire dans le fichier `key.bin`.
   b) (3pts) Alice chiffre `message.txt` avec `key.bin` en utilisant AES-128-CBC et met le résultat dans `protected-mess` codé en base64.
   c) (3pts) Alice chiffre, en utilisant `rsautl`, la clé `key.bin` avec la clé publique de bob et met le résultat dans `protected-key.bin`
8. (5pts) Aider Bob à retrouver le contenu de `message.txt` à partir de `protected-key.bin` et `protected-message.` 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, on ne peut autoriser des utilisateurs à lancer des opérations sensibles qu'après une vérification convenable de leurs identités. L'objectif de cet exercice et d'implanter quelques techniques d'authentification et d'analyser leurs vulnérabilités vis-à-vis des attaques MITM (*Man In The Middle*). Les trois techniques d'authentification suivantes pourraient se faire à l'intérieur d'un tunnel TLS/SSL. Mais, dans ce TP, nous n'allons pas nous préoccuper de la création de ce tunnel. Pour les trois 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 " $\alpha.\ \ A \longrightarrow B\ :\ \ m$ " 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 mots de passe en clair (20 points)

L'échange entre le client et le serveur se fait via les phases suivantes :

- **Enregistrement :** Le client s'enregistre chez le serveur en fournissant certaines informations incluant un $userID$, un nom d'utilisateur et un mot de passe. Le serveur sauvegarde toutes les informations sur le client, sauf le mot de passe qui sera remplacé par son hachage.

$$\begin{array}{llllll} E1. & C & \longrightarrow & S & : & userID\ \ nom\ \ pwd \\ E2. & S & \longrightarrow & C & : & 200 \end{array}$$

  Pour des raisons de simplicité, nous supposons que le serveur détient, dans un fichier, des enregistrements contenant les informations suivantes : $userID$, nom et $H_1(pwd)$.
- **Authentification :** Avant de commencer ses transactions, le client doit prouver son identité.

$$\begin{array}{llllll} A1. & C & \longrightarrow & S & : & SessionID\ \ userID\ \ pwd \\ A2. & S & \longrightarrow & C & : & SessionID\ \ code\ \ SetCookie{:}Session{=}N_s \end{array}$$

  Si le hachage du mot de passe fourni par le client correspond au hachage du mot de passe enregistré par le serveur, l'authentification sera acceptée et le serveur retourne le code 200 (OK) accompagné d'un *cookie* de session N_s. Sinon, il retourne le code 401 (utilisateur non autorisé).
- **Transactions :** Le client peut répéter autant qu'il veut les opérations suivantes en envoyant ses commandes ($commande$) accompagnées du *cookie* de la session.

$$\begin{array}{llllll} T1. & C & \longrightarrow & S & : & SessionID\ \ Commande\ \ Cookie{:}session{=}N_s \\ T2. & S & \longrightarrow & C & : & SessionID\ \ code \end{array}$$

Si le N_s fourni par le client correspond à celui envoyé par le serveur, ce dernier retourne le code 200. Sinon, il retourne le code 401.

Remarque : Pour ce genre de protocoles, il est facile de constater qu'un pirate qui réussit à se placer entre le client et le serveur peut facilement voler le $userID$ et le mot de passe d'un utilisateur. Par ailleurs, si la base de données du serveur est attaquée, le pirate va pouvoir récupérer les mots de passe hachés de tous les utilisateurs, ce qui lui permet de briser tous les mots de passe mal choisis.

2.2 Authentification via des défis/réponses basés sur des mots de passe (25 points)

Le protocole suivant est inspiré de Ms-CHAP-v2 de Microsoft. L'échange entre le client et le serveur se fait selon les phases suivantes :

- **Enregistrement :** Elle se fait de la même manière que le protocole précédent sauf que le serveur sauvegarde directement les mots de passe et non pas leurs hachages. Autrement le serveur détient des enregistrements de la forme : $userID$, nom et pwd.

– **Authentification :** Avant de commencer ses transactions, le client doit prouver son identité comme suit :

$$\begin{array}{llllll} A1. & C & \longrightarrow & S & : & SessionID\ \ userID \\ A2. & S & \longrightarrow & C & : & SessionID\ \ N_s \\ A3. & C & \longrightarrow & S & : & SessionID\ \ H_1(SeesionID.N_s.H_2(pwd))\ \ N_c \\ A4. & S & \longrightarrow & C & : & SessionID\ \ code\ \ H_1(SeesionID.N_c.H_2(pwd))\ \ SetCookie\text{:}Session = N'_s \end{array}$$

À l'étape $A3$, le serveur calcule la valeur attendue de $H_1(SeesionID.N_s.H_2(pwd))$ et il la compare avec la valeur reçue. En cas d'erreurs, il retourne le code 411, sinon il retourne le code 200 avec un *cookie* de la session et $H_1(SeesionID.N_c.H_2(pwd))$. Dans ce protocole, le client a le moyen aussi de s'assurer qu'il est en train d'échanger avec le bon serveur (même si le protocole TLS/SSL n'est pas utilisé) en comparant la valeur attendue de $H_1(SeesionID.N_c.H_2(pwd))$ avec la valeur reçue. H_1 et H_2 sont deux fonctions de hachage et les points séparent les différentes composantes signifient une concaténation (sans laisser d'espace entre les composantes).

– **Transactions :** Le client peut répéter autant qu'il veut les opérations suivantes en envoyant ses commandes ($commande$) accompagnées du *cookie* de la session.

$$\begin{array}{llllll} T1. & C & \longrightarrow & S & : & SessionID\ \ Commande\ \ Cookie\text{:}session = N'_s \\ T2. & S & \longrightarrow & C & : & SessionID\ \ code \end{array}$$

Si le $Cookie : session\ N_s$ fourni par le client correspond à celui envoyé par le serveur, ce dernier retourne le code 200. Sinon, il retourne le code 401.

Remarque : Pour ce genre de protocoles, bien qu'il soit difficile de récupérer le mot de passe de l'utilisateur, un pirate qui réussi à se placer entre le client et le serveur peut facilement modifier la commande ou réutiliser le *cookie* de la session pour injecter ses propres commandes. Par ailleurs, si la base de données du serveur est attaquée, le pirate pourrait récupérer les mots de passe de tous les utilisateurs s'ils ne sont pas bien protégés.

2.3 Authentification via des clés publiques (25 points)

Dans ce qui suit nous considérons une version simplifiée du protocole UAF de FIDO-Alliance (Google, Microsoft, Samsung, PayPal, etc.).

– **Enregistrement :** 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 chez le serveur en fournissant certaines informations incluant son $userId$, son nom et sa clé publique k_c qui sera utilisée exclusivement avec le serveur et les $UserID$ en question. Le serveur sauvegarde toutes les informations sur le client.

$$\begin{array}{llllll} E1. & C & \longrightarrow & S & : & userID\ \ nom\ \ k_c \\ E2. & S & \longrightarrow & C & : & 200 \end{array}$$

Les clés privées du client sont sauvegardées localement chez lui et elles ne sont accessibles que via un mot de passe connu seulement par lui. À chaque fois qu'il a besoin d'utiliser une clé privée, il doit fournir son mot de passe. Autrement, pour chaque serveur S, le client détient un enregistrement dans sa trousse de clés contenant le nom du serveur S et $\{k_c^{-1}\}_k$, avec $\{k_c^{-1}$ est la clé privée associée à la clé k_c et k est une clé générée à partir de son mot de passe (pwd) en appliquant une fonction de hachage H_1 ($k = H_1(pwd)$). De son côté, le serveur détient des enregistrement contenant les informations suivantes : $userID$, nom et k_c.

– **Authentification :** Avant de commencer ses transactions, le client doit prouver son identité, via les étapes suivantes :

$$\begin{array}{llllll} A1. & C & \longrightarrow & S & : & SessionID\ \ userID \\ A2. & S & \longrightarrow & C & : & SessionID\ \ N_s \\ A3. & C & \longrightarrow & S & : & SessionID\ \ \{H_1(N_s)\}_{k_c^{-1}} \\ A4. & S & \longrightarrow & C & : & SessionID\ \ code \end{array}$$

À l'étape A_3, le serveur utilise la clé publique du client pour retrouver $H_1(N_s)$ et il le compare avec le hachage de la valeur de N_s envoyée à l'étape $A2$. Si les deux hachages coïncident, il retourne 200 comme $code$, sinon, il retourne le code 401.

– **Transactions :** Le client peut répéter autant qu'il veut les opérations suivantes.

$$\begin{array}{lllll} T1. & C & \longrightarrow & S & : \ Commande \\ T2. & S & \longrightarrow & C & : \ Commande \ \ N'_s \\ T3. & S & \longrightarrow & C & : \ \{H_1(Commande.N'_s)\}_{k_c^{-1}} \\ T4. & S & \longrightarrow & C & : \ code \end{array}$$

À la différence des protocoles précédents, le client confirme sa commande à l'étape $T3$ en la signant avec sa clé privée. Plus précisément, à l'étape 3, le client voit sur son écran une demande de confirmation de la commande qu'il doit approuver en fournissant son mot de passe.

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.

2.4 Autres détails d'implantation

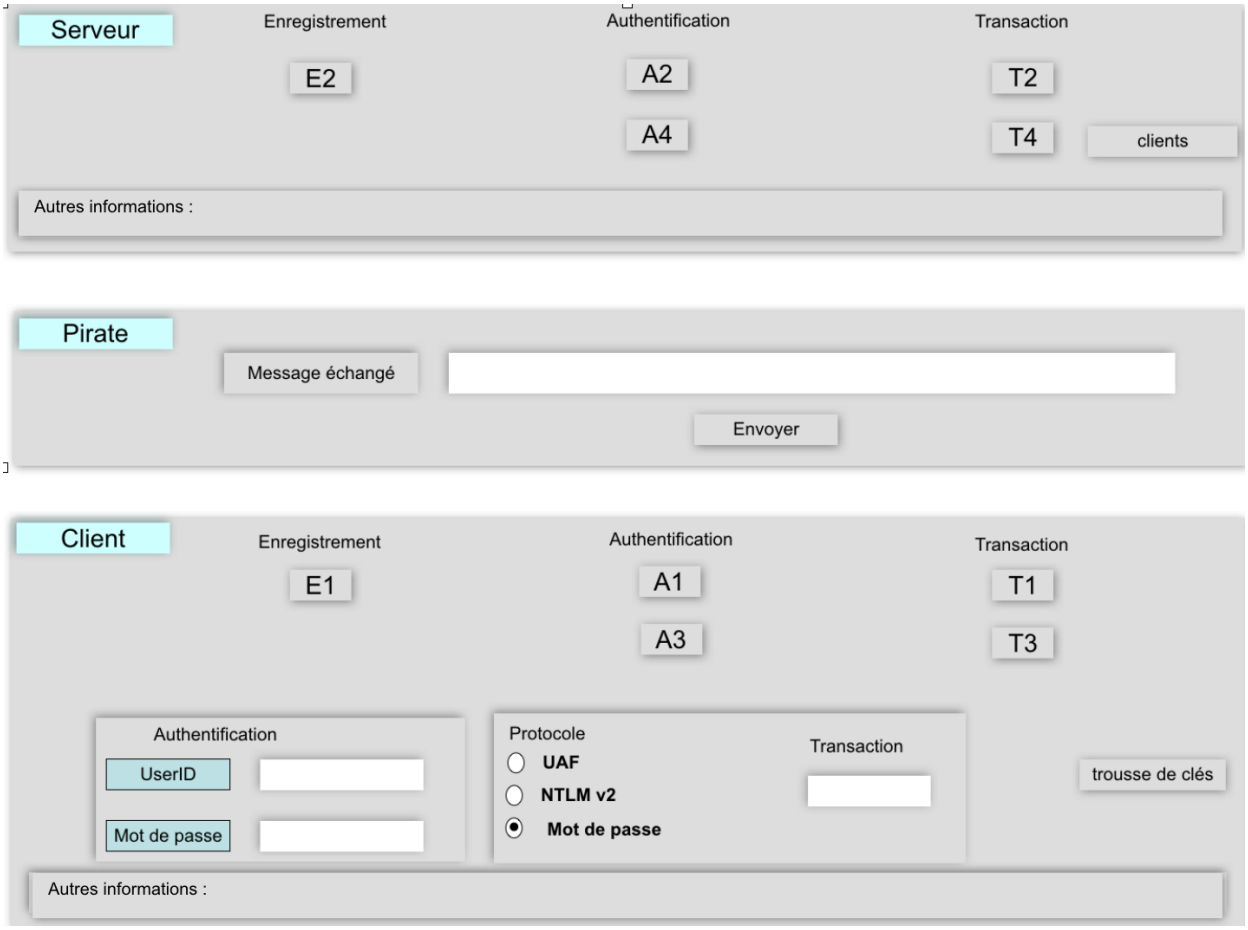
– $UserId$ est une chaine de caractères (maximum 10 caractères).
– nom est une chaine de caractères (maximum 10 caractères).
– $SessionID$, N_s, N'_s et N_c sont des entiers aléatoires ayant 5 chiffres au maximum.
– La commande est un texte fourni par l'utilisateur (maximum 10 caractères) comme TRANSFERT, PAYER, 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 SHA1.
– La fonction de hachage H_2 est MD4.
– Pour protéger ses clés privées dans sa trousse le client utilise AES-128 avec le mode CBC.
– Les clés privées et publiques sont des clés RSA de 1024 bits.
– Les composantes hachées ou signées doivent être encodées en Base64 avant d'être envoyées sur le réseau.

2.5 Travail demandé

On vous demande d'écrire un programme (en C, C++, C#, Java ou JavaScript) qui simule les échanges entre un client et un serveur, et ce, en présence d'un pirate qui a le contrôle total du réseau. Pour chaque message envoyé et avant d'être acheminé à sa destination, on donne la possibilité au pirate de le modifier. Évidemment, le receveur n'accepte pas le message s'il a le moyen de savoir qu'il a été modifié. L'interface doit contenir trois partis : une pour le client, la deuxième pour le pirate et la troisième pour le serveur comme dans la figure suivante. Avant qu'un message ne puisse se rendre à sa destination, il doit apparaître dans la partie "Pirate" qui donne la possibilité à l'utilisateur (Pirate) de le modifier. Une fois la modification terminée, on appuie sur le bouton "Envoyer" pour envoyer la nouvelle version du message à sa destination.

Les boutons $E1$ et $E2$ permettent d'enregistrer un client chez le serveur, les boutons $A1$, $A2$, $A3$ et $A2$ lui permettent de faire son authentification et les boutons $T1$, $T2$, $T3$ et $T4$ lui permettent de faire ses transactions.

Le bouton "clients" permet d'afficher, dans la zone "autres informations", les enregistrements des clients sauvegardés par le serveur. Le bouton "trousse de clés" permet d'afficher, dans la zone "autres informations", la trousse des clés du client.



3 Remarques

1. Le travail est individuel.
2. Le barème est à titre indicatif et que 10% des points de l'exercice 2 sont réservés aux commentaires.
3. 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'aucun autre fichier externe pour pouvoir fonctionner.

5 Échéancier

Le 14 décembre 2016 avant 14h00. À noter que les TPs remis en retard ne seront pas acceptés.