Protocoles Internet
TP 7

Juliusz Chroboczek

26 novembre 2023

On appelle *chiffrage opportuniste* l'approche qui consiste à chiffrer les données même quand on n'a pas authentifié le pair. Le chiffrage pportuniste est vulnérable aux attaques de type *man in the middle* (MiTM), mais il permet de se protéger à faible coût des attaques passives.

Le but de ce TP est d'implémenter un client TCP qui effectue un échange Diffie-Hellman opportuniste puis reçoit un message chiffré. Pour adapter ces techniques à UDP, il faudra soit utiliser un canal de contrôle fiable pour effectuer l'échange Diffie-Hellman, soit prendre soin de réémettre les paquets perdus.

Le protocole implémenté durant ce TP n'est pas conforme à l'état de l'art :

- il effectue un échange Diffie-Hellman sur des entiers de 768 bits, alors qu'il faudrait utiliser au moins 2048 bits en 2023 ;
- il effectue un échange Diffie-Hellman sur un groupe modulaire, en 2023 on utilise plutôt les groupes de courbes elliptiques ;
- il utilise AES-128 en mode CBC, alors qu'en 2023 on préfère le mode GCM.

Je ne suis pas cryptographe, et mon protocole a donc sûrement de nombreuses autres vulnérabilités. Avant de mettre un protocole en production, il est essentiel de consulter un spécialiste.

Exercice 1 (Questions préliminaires)**.**

1. Le chiffrage opportuniste est parfois appelé *better than nothing cryptography (BTN)*. Quelles sont les faiblesses du cryptage opportuniste ? Pourquoi est-il utile quand même ?
2. Lorsque le certificat d'un serveur HTTPS ne peut être validé, le navigateur *web* fait essentiellement du chiffrage opportuniste. Il affiche alors un gros avertissement rouge qui fait peur, et cela alors même qu'il n'affiche pas d'avertissement lors d'une connexion HTTP non chiffrée. Qu'en pensez-vous ? (Vous êtes exceptionnellement autorisés à développer une théorie du complot.)
3. (Pour les crypto.) Quels sont les avantages de ECDH par rapport à DH ? Quels sont les avantages de GCM par rapport à CBC ?

Exercice 2 (Échange Diffie-Hellman)**.** Exécutez le serveur TCP fourni avec l'option `-verbose`. Écrivez un programme qui se connecte au serveur puis :

- tire une chaîne aléatoire de 768 bits et la convertit en un entier $a < 2^{768}$;
- calcule $A = g^a \mod p$ (les valeurs de p et g sont dans le fichier fourni) ;

– envoie A au serveur sous forme d'une chaîne de 768/8 octets ;
– lit une chaîne de 768/8 octets qu'il interprète comme un entier B ;
– vérifie que B n'est pas un élément trivial du groupe $\mathbf{Z}/p\mathbf{Z}$ (les éléments triviaux sont 0, 1 et $p-1$) ;
– calcule l'entier $s = B^a \mod p$.

Vérifiez à chaque étape que votre programme a les mêmes valeurs que le serveur (mettez des `Printf` de partout).

Vous pourrez vous servir de la fonction `crypto/rand.Read`, de la fonction `io.ReadFull`, et des méthodes suivantes du type `math/big.Int` : `SetBytes`, `FillBytes` et `Exp`.

Exercice 3 (Chiffrage)**.** La valeur s obtenue par votre programme est la même sur le serveur et sur le client, et elle n'est pas connue par un observateur passif ; elle peut donc servir à générer une clé de chiffrage opportuniste.

Pour éviter qu'un observateur passif puisse détecter que deux messages sont identiques, nous n'utilisons pas la clé directement dans un algorithme de chiffrage, mais nous la combinons avec un *vecteur d'initialisation* (IV) aléatoire, transmis en clair. La façon de combiner la clé de chiffrage, le IV et le texte chiffré s'appellent un *mode*. Le serveur fourni utilise l'algorithme de chiffrage AES-128 utilisé en mode CBC.

Après l'échange Diffie-Hellman, le serveur envoie :

– 16 octets aléatoires qui servent de vecteur d'initialisation (IV) ;
– 32 octets de texte chiffré.

Modifiez votre programme pour qu'il :

– calcule $h = \mathrm{SHA256}([s])$, où $[s]$ est la représentation de s comme une suite de 768/8 octets ;
– affecte à k les premiers 16 octets de h ; k servira de clé partagée *(key)* ;
– lise 16 octets ; ces 16 octets serviront de vecteur d'initialisation (IV) ;
– lise 32 octets qui serviront de texte chiffré ;
– déchiffre le texte chiffré avec l'algorithme AES-128 en mode CBC avec la clé et le IV obtenus précédemment, enlève les octets valant 0 qui suivent la chaîne, puis convertit le résultat en une chaîne qu'il affichera.

Pour le dernier point, consultez l'exemple donné dans la documentation du paquet `crypto/cipher`. Pour éliminer les octets valant 0, vous pouvez vous servir de la fonction `bytes.TrimRight`.