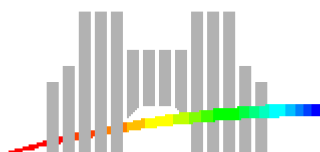


Laboratoire de l'Informatique du Parallélisme
Equipe PLUME - Bureau 319
Ecole Normale Supérieure de Lyon,
46 Allée d'Italie, F-69364 Lyon 7
Tél. 33 (0)4 72 72 86 83
Fax : 33 (0)4 72 72 80 80
www.ens-lyon.fr/LIP/PLUME

Appel d'offres OPPIDUM

"Offre de Procédés et de ProduIts De sécUrisation pour la Mise en œuvre des autoroutes de l'information"

Conventions N°: 00.2.93.0466

Spécification du protocole des signatures légères

Délivrable E1 - 2

Editeur : ENS Lyon

Edition 1.0

Lyon, le 08 décembre 2000

Document 2 : Spécification du protocole des signatures légères

Version 1.0 du 8/12/2000, Par Eric Pommateau et Patrice Dargenton

Table des matières

1 Introduction ... 3
1.1 Généralités ... 3
1.2 Notations ... 3
2 Préparatifs ... 4
2.1 Génération de la séquence de nonces ... 4
2.2 Fonction de signature ... 4
2.3 Format des messages ... 4
2.3.1 Message initial ... 4
2.3.2 Autre message ... 5
2.4 Codes d'erreur ... 5
3 Description du protocole ... 6
3.1 Requête initiale du client ... 6
3.2 Requête suivante du client ... 6
3.3 Réponse du serveur ... 7
3.4 Autres évènements ... 7
3.4.1 TIME_OUT_CLIENT ... 7
3.4.2 TIME_OUT_SERVER ... 7
4 Exemples ... 7
4.1 Fonctionnement normal ... 8
4.2 Perte d'un message émis par le serveur ... 8
4.3 Perte d'un message émis par le client ... 8
4.4 Défaillance du serveur ... 8
4.5 Exemple d'attaque infructueuse ... 9
5 Justifications du protocole ... 9
5.1 Mécanismes cryptographiques ... 9
5.2 Attaques envisageables ... 10
5.2.1 Hypothèses sur les pouvoirs de l'attaquant ... 10
5.2.2 Attaques fatales ... 10
5.2.3 Attaques moyennes ... 10
5.2.4 Dénis de services ... 10
6 Conclusion ... 10

1 Introduction

1.1 Généralités

Ce document décrit la spécification d'un protocole client-serveur qui permet un échange de messages signés et qui n'utilise pas pour chaque message un calcul cryptographique "lourd" de type R.S.A. Ce protocole sera mis en place dans µ-COMM+ entre le SIC (Serveur de l'Intermédiaire de Confiance) et le SD (Serveur de Données). Ce protocole sera réutilisable pour traiter des flux de données au-dessus de la couche U.D.P. (flux de vidéo par exemple).
Le client initie le protocole en envoyant une graine chiffrée avec un algorithme lourd (R.S.A.). Cette graine engendre alors une chaîne représentant des nombres, cette chaîne étant obtenue en itérant une fonction de hachage rapide et surtout à sens unique (MD5, SHA-1, RIPMD-160) sur cette graine. On constitue alors une série de nombres, que l'on désignera comme la séquence de nonces. Ces nonces peuvent alors servir à signer des messages grâce à une fonction de hachage utilisée sur le message concaténé avec un nonce de la séquence.

Une fois la graine déchiffrée par le serveur, celui-ci va signer ses messages avec tous les nonces de rang impair, et le client va signer les siens avec les nonces de rang pair. On commence à prendre les nonces à partir de la fin de la séquence (ce mécanisme est similaire à OTP) afin d'éviter que la compromission d'un nonce ne permette la découverte de tous les nonces suivants de la séquence.

Dans le cas où le client et le serveur sont désynchronisés (réception d'un message dont l'index n'est pas attendu), plusieurs cas de figure sont à envisager :
- Si une requête se perd dans le sens client-serveur, le client ne recevra pas de réponse du serveur. Il va donc ré-émettre sa requête en précisant que la précédente n'a pas été servie. Si le serveur reçoit cette requête, il va s'apercevoir que le client est en avance et il va le préciser dans sa réponse. Le serveur est obligé dans ce cas de se re-synchroniser avec le client.
- Si une requête se perd dans le sens serveur-client, le client va aussi ré-émettre une requête en précisant que la précédente n'a pas été servie. Pour le serveur, la requête est bien celle attendue (il a servi la précédente), il la traite donc normalement, sans message d'erreur.
- Si un acteur reçoit un message avec un indice trop petit par rapport à ce qu'il attend, il ignore ce message. Le nonce qui a été utilisé pour signer ce message ne pouvant plus être considéré comme actuel.
- Si le client reçoit un message avec un indice trop élevé par rapport à ce qu'il attend, il y a un problème ! Il ré-initialise donc la séquence. En effet, comme c'est le client qui a l'initiative, le processus n'a pas pu avancer sans qu'il le sache.

Comme le nonce sert aussi de critère de fraîcheur, si des messages arrivent signés avec un nonce déjà utilisé, le client comme le serveur doivent ignorer ce message (cela peut être une tentative de fraude ou bien un problème réseau).

1.2 Notations

On notera :
- Clt : le client,
- Srv : le serveur,
- Q_i : un message du client (Q pour Query),
- A_i : une réponse du serveur (A pour Answer),
- IndClt et IndSrv : deux entiers positifs (index) maintenus respectivement par le client et par le serveur,
- N : le nombre maximum de messages que l'on peut envoyer dans la session de signatures légères,
- X, Y : la concaténation du message X avec le message Y,
- P_A, S_A : la paire clé publique - clé privée de l'acteur A (P pour Public, S pour Secrecy),

- $\{X\}_K$: le message X chiffré avec la clé K,
- $H_i(X)$: X chiffré par la fonction de hachage H_i,
- α : une graine crée par Clt,
- E : un code d'erreur du serveur à destination de Clt,
- Sign(X, E, k) : la signature du message X concaténé avec E et l'indice k (voir la formule ci-après au § 2.2).

Un message M émis par l'acteur A vers l'acteur B est noté : A -> B : M

2 Préparatifs

2.1 Génération de la séquence de nonces

Une graine α est créée aléatoirement par le client et envoyée au serveur en utilisant la cryptographie lourde. Le client chiffre la graine d'abord avec sa clé privée pour en garantir l'authenticité, puis avec la clé publique du serveur pour en garantir la confidentialité. Cette graine est employée pour créer la séquence de nonces en itérant une fonction de hachage sur cette graine ; ainsi, on obtient une séquence de la forme :

$$\alpha, H_1(\alpha), H_1^2(\alpha), \ldots, H_1^{2N}(\alpha)$$

On notera $Nonce_i(\alpha)$, le nonce correspondant au i[ème] hachage de α, soit :

$$Nonce_i(\alpha) = H_1^i(\alpha)$$

La séquence peut être soit stockée, soit recalculée à chaque message par le client. Par contre, le serveur doit obligatoirement stocker cette séquence pour des raisons de performances.

2.2 Fonction de signature

La fonction de signature du message est :

$$Sign(C, E, i) = H_2(C, E, Nonce_{2N-1-i}(\alpha))$$

C représente un contenu de message (ou Q_i si le message est envoyé par le client et A_i si le message est la réponse du serveur) et E une erreur associée à un message du protocole.

H_2 permet d'éviter une interception du message qui permettrait à un intrus de changer le contenu et de calculer une signature valide. En pratique, rien n'oblige que H_2 et H_1 soient différentes.

2.3 Format des messages

2.3.1 Message initial

Un message initial du client à la forme suivante :

Acteur, Index, Graine, Signature, Contenu

- Acteur étant la personne qui envoie le message,
- Index représente le rang du nonce en cours d'utilisation (ici Index = 0 étant donné qu'il s'agit du message initial),
- Graine représente la graine α chiffrée avec la clé privée du client puis avec la clé publique du serveur,
- Signature est obtenue par la fonction Sign(Message, '', Index),

- Contenu, enfin, est la première requête du client.

On remarquera l'absence de message d'erreur. En effet, lors du premier message, aucun message n'est sensé avoir transité auparavant entre le client et le serveur.

2.3.2 Autre message

Un message quelconque du client ou du serveur a la forme suivante :

Acteur, Index, Erreur, ComplementErreur, Signature, Contenu

- Acteur étant la personne qui envoie le message (l'identifiant du client par exemple, et pour le serveur un code unique),
- Index représente le rang du nonce en cours d'utilisation,
- Erreur est le code d'erreur sur ce message ou sur un autre évènement du protocole (voir le § 2.4 sur les codes d'erreur),
- ComplementErreur est un champ supplémentaire pour certaines erreurs qui demandent un complément d'information,
- Signature est obtenue par la fonction Sign(Contenu, Erreur, Index),
- Contenu, enfin, est soit la requête du client, soit la réponse du serveur.

2.4 Codes d'erreur

Les codes des erreurs sont les suivants :

- OK

Le protocole se déroule normalement.

- BAD_SIGNATURE

La signature n'est pas valide, le message n'a pu être traité. Complément : en-tête du message du client ayant provoqué cette erreur.

- SERVER_EARLY

Message envoyé par le client si le serveur est décalé vers l'avant. Le message n'est pas pris en compte, on envoie également un LAST_MESSAGE.

- SERVER_LATE

Une réponse du serveur est arrivée en retard, ce message est ignoré par le client. Complément : la différence d'index entre le serveur et le client.

- CLIENT_EARLY

Un certain nombre de requêtes client se sont perdues. Complément : nombre de requêtes client que le serveur n'a pas reçues.

- CLIENT_LATE

Une requête client est arrivée alors que d'autres requêtes d'indice plus élevé dans la séquence ont déjà été traitées. Complément : En-tête de la requête client arrivée en retard.

- TIME_OUT_CLIENT

Le client n'a pas attendu la dernière réponse du serveur. Complément : nombre de TIME_OUT_CLIENT.

- TIME_OUT_SERVER
Le client n'a pas interrogé le serveur depuis trop longtemps, on envoie également un LAST_MESSAGE.

- LAST_MESSAGE
C'est le dernier message émis par le serveur ou par le client demandant de réinitialiser la séquence. Seul l'erreur LAST_MESSAGE peut provoquer la réinitialisation de la séquence.

Le complément d'erreur est mis à 0 si aucune erreur ne nécessite de détail (comme typiquement le code OK).

Toutes les erreurs sont incluses dans la partie signée (chiffrement léger) des messages. Sinon, un intrus pourrait, sans modifier un message, modifier le code d'erreur et provoquer des attaques sérieuses de type DoS (Denial of Service, Déni de service), par exemple en envoyant constamment le code d'erreur LAST_MESSAGE.

3 Description du protocole

Les codes d'erreur et les compléments d'erreur sont volontairement omis dans les messages et dans la partie signée afin d'améliorer la lisibilité de cette description. Chaque erreur rencontrée est stockée localement et renvoyée lors de la prochaine communication. Lorsqu'un message est émis, le code d'erreur et le complément d'erreur de l'acteur ayant émis le message est remis à 0.

3.1 Requête initiale du client

Au début du protocole, IndClt prend la valeur 0.

Clt -> Srv : Clt, 0, $\{\{\alpha\}_{SClt}\}_{PSrv}$, Sign($Q_0$, 0), Q_0

3.2 Requête suivante du client

Soit IndSrv le dernier index reçu par le client en provenance du serveur,
IndClt $\neq$ 0 et IndClt < N :

- IndSrv = IndClt :

Clt -> Srv : Clt, 2*IndClt, Sign(Q_{IndClt}, 2*IndClt), Q_{IndClt}

- IndSrv < IndClt (Vieux messages du serveur) :
Le client ignore cette réponse, Erreur += SERVERLATE

- IndSrv > IndClt :
Erreur += LAST_MESSAGE + SERVER_EARLY

Clt -> Srv : Clt, 2N-2, Sign('Erreur', 2N-2), 'Erreur'

Si IndSrv = 0, le message n'est pas traité (SERVER_LATE). Si IndSrv $\geq$ N, le message est ignoré (signature invalide).

3.3 Réponse du serveur

Au début du protocole, IndSrv prend la valeur 0. Lors du premier message du client, la totalité de la séquence de nonces générée grâce à α est stockée par le serveur.

Suivant la valeur de l'index du client (IndClt) reçue, le serveur répond comme suit :

- IndClt = IndSrv :

Srv −> Clt : Srv, 2*IndSrv+1, Sign(A_{IndSrv}, 2*IndSrv+1), A_{IndSrv}

puis mise à jour des variables : IndSrv++, IndClt++ (lorsque le client reçoit la réponse du serveur).

- IndClt > IndSrv : Erreur += CLIENT_EARLY

Srv −> Clt : Srv, 2*IndClt+1, Sign(A_{IndSrv}, 2*IndClt+1), A_{IndSrv}

puis mise à jour des variables : IndSrv = IndClt+1, IndClt++ (lorsque le client reçoit la réponse du serveur).

- IndClt < IndSrv :
Le serveur ignore cette demande, Erreur += CLIENT_LATE
Aucune mise à jour n'est nécessaire.

3.4 Autres évènements

3.4.1 TIME_OUT_CLIENT

Si le serveur ne répond pas au bout d'un certain temps, le client peut ré-émettre sa requête en la signant avec un nouveau nonce. Il envoie aussi le message TIME_OUT_CLIENT avec comme complément le nombre de fois où il a fait un time-out. Au bout d'un nombre T fixé de time-out sans réponse de la part du serveur, le client réinitialise la séquence (T sera fixé lors des tests de simulation).

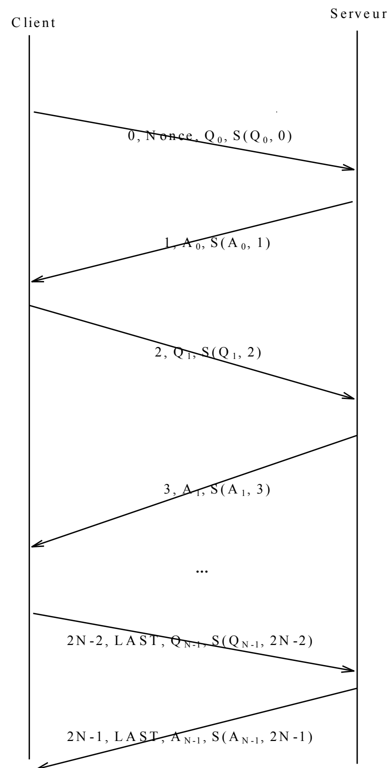
3.4.2 TIME_OUT_SERVER

Le serveur peut vouloir déclarer le protocole obsolète, c'est-à-dire que la session doit être interrompue. Les nonces, ainsi que les clés, ont des durées de vie limitées ; de plus, le serveur peut aussi vouloir faire de temps à autre des révisions. Lors de la prochaine requête client, le serveur enverra alors une erreur TIME_OUT_SERVER ainsi qu'un LAST_MESSAGE. Ce mécanisme est utile aussi lorsque le serveur constate par exemple que la séquence de nonces est compromise.
Il est bien évident que les deux échelles de temps ne sont pas comparables : Le TIME_OUT_CLIENT est de l'ordre de la minute tandis que le TIME_OUT_SERVER est de l'ordre de la semaine, voire du mois.

4 Exemples

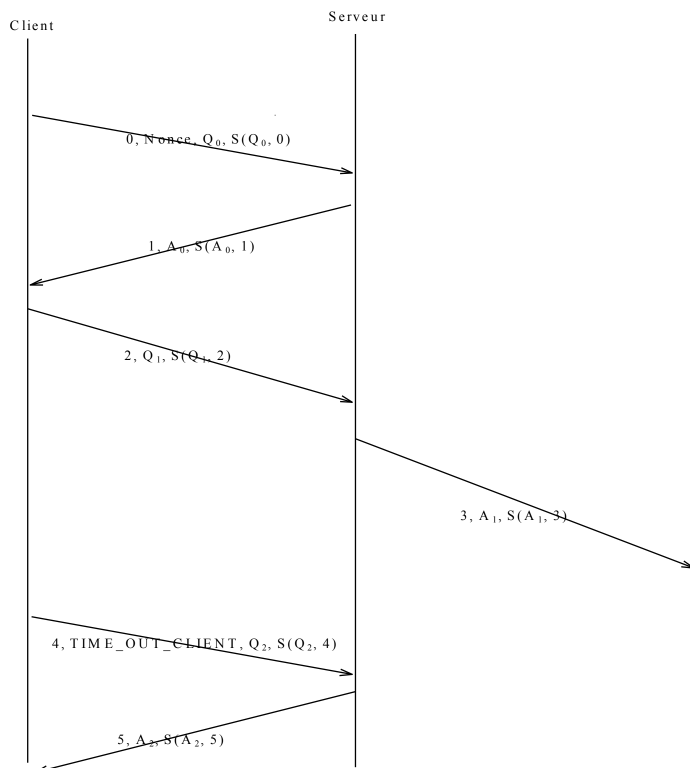
La fonction Sign() est abrégée en S() dans les schémas suivants :

4.1 Fonctionnement normal



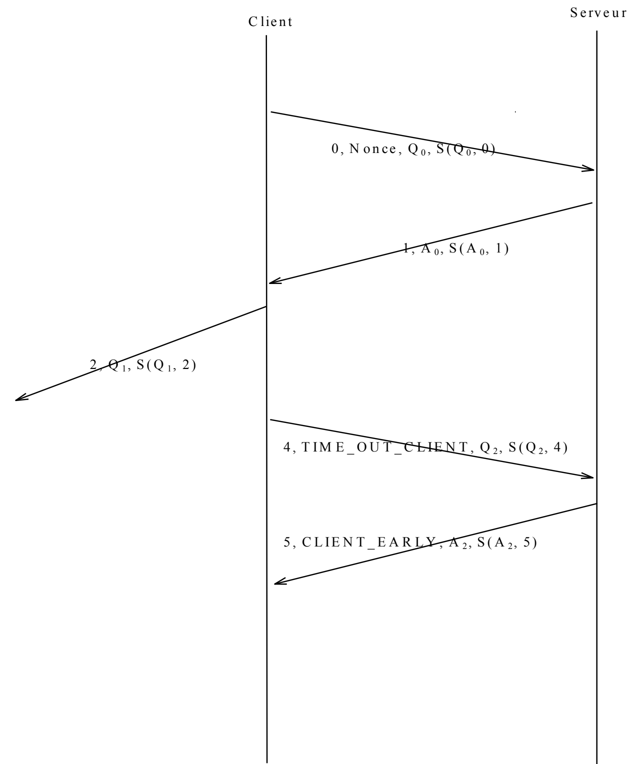
Fonctionnement normal

4.2 Perte d'un message émis par le serveur



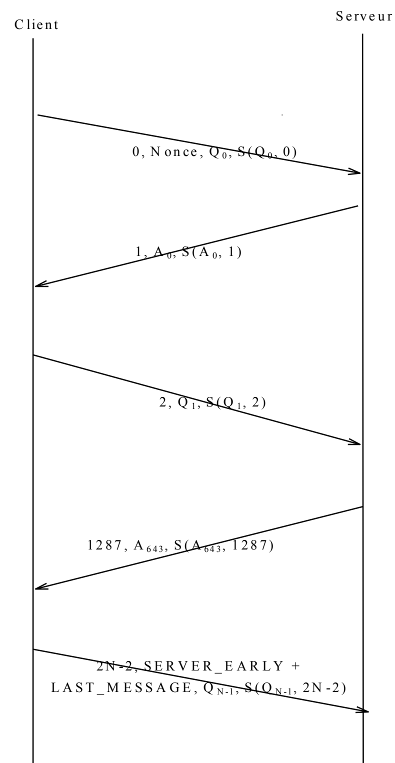
Perte d'un message serveur

4.3 Perte d'un message émis par le client



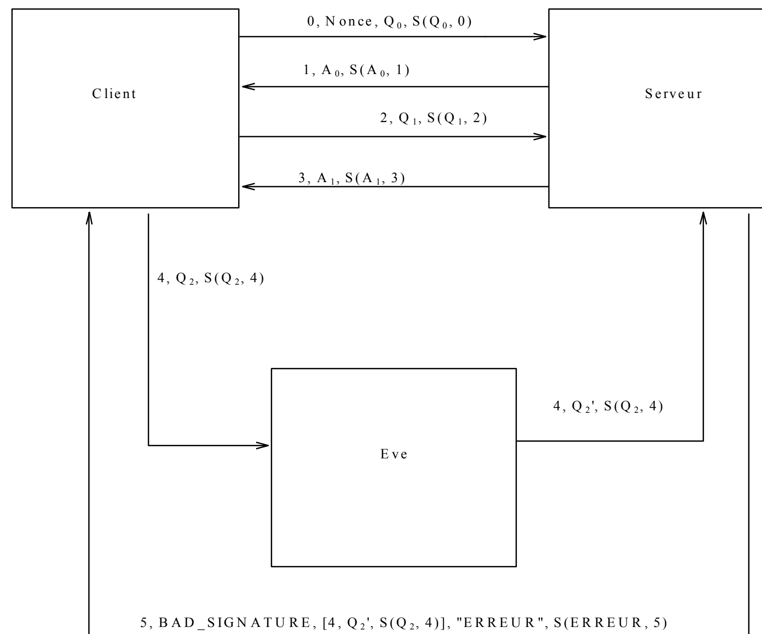
Perte d'un message client

4.4 Défaillance du serveur



Défaillance serveur

4.5 Exemple d'attaque infructueuse



Attaque infructueuse

5 Justifications du protocole

5.1 Mécanismes cryptographiques

Les mécanismes de sécurité utilisés par ce protocole sont :

1. La méthode clé publique-clé privée
Cette méthode assure que nulle autre personne que le serveur ne pourra lire la graine α. Le client et le serveur connaîtront donc à tout moment la totalité de la séquence de nonces.

2. Les fonctions de hachage
H est une fonction impossible à inverser dans la pratique (la connaissance de H(x) ne permet pas la connaissance de x).

3. Le changement de nonce
Le changement de nonce pour chaque message est utile non seulement pour éviter que deux messages émis au client ou au serveur puissent apparaître identiques si leurs contenus sont identiques (afin d'éviter les failles que les logiciels de cryptanalyse savent exploiter), mais surtout pour éviter qu'un espion puisse envoyer un message déjà émis lors d'une stratégie d'attaque.

4. La prise à rebours des nonces de la séquence calculés avec une fonction de hachage
Si la fonction de calcul de la séquence était triviale à calculer, la perte d'un seul nonce entraînerait la possibilité pour l'assaillant de signer n'importe quel message jusqu'à épuisement de la séquence. Toutefois, comme l'indique le qualificatif léger de la signature, le temps de vérification de la signature ne doit pas être trop long. C'est pourquoi on n'emploi pas des fonctions comme 3-DES et l'utilisation d'une fonction de hachage semble alors être un bon compromis. Les nombres de la séquence de nonces sont pris à rebours pour éviter que la découverte d'un nombre n'entraîne la compromission de l'ensemble de la session.

5.2 Attaques envisageables

Afin de modéliser la sécurité du protocole, on admet les hypothèses suivantes, qui ne sont pas exagérées pour évaluer les conséquences d'une attaque.

5.2.1 Hypothèses sur les pouvoirs de l'attaquant

L'attaquant peut :
- Lire tous les messages du réseau,
- Envoyer des messages sur le réseau comme s'il s'agissait d'un message d'un autre acteur (imposture),
- Bloquer des messages,
- Rétro-concevoir (désassembler) les logiciels et donc retrouver les algorithmes utilisés dans le protocole.

Par contre, il ne peut normalement pas en un temps raisonnable (selon les hypothèses cryptographiques) :
- Casser une clé R.S.A.,
- Inverser une fonction de hachage.

5.2.2 Attaques fatales

Les 3 actions suivantes sont fatales pour le protocole :
- Perte de α, qui permettrait à l'attaquant de calculer la séquence de nonces et d'émettre autant de messages signés qu'il le souhaite,
- Perte de SSrv, qui permettrait l'imposture du serveur,
- Perte de SClt, qui permettrait l'imposture du client.

5.2.3 Attaques moyennes

On peut différencier deux sortes d'attaque de crytpanalyse :

1. En-ligne : Un attaquant récupère et bloque la progression d'un message. Il essaie de falsifier ce message avant que l'acteur ne génère un time-out. On se protège contre ce genre d'attaque par la fonction de hachage. Le temps mis pour inverser cette fonction par force brute est normalement bien supérieur au temps de TIME_OUT_CLIENT.

2. Hors-ligne : L'attaquant ayant récupéré un message, il dispose de plus de temps pour calculer de nouveaux nonces. La prise à rebours des nonces dans la séquence empêche l'assaillant d'exploiter sa connaissance de ce message. Il devra de toute façon agir avant la clôture du protocole (TIME_OUT_SERVER). Il serait également étonnant qu'une personne achète un super-calculateur pour pirater des micro-paiements ; cependant des utilisateurs pourront bientôt utiliser des réseaux de PC disponibles, par exemple ceux mis à la disposition des étudiants.

5.2.4 Dénis de services

L'attaquant peut provoquer des embouteillages au niveau du serveur, en envoyant par exemple des messages invalides ou en essayant de modifier des codes d'erreur. On prévient le déni de service en signant les codes d'erreur et en ne répondant pas à un message initial invalide.

6 Conclusion

L'intérêt du protocole des signatures légères est la répartition judicieuse entre le cryptage "lourd" et "léger" afin de permettre des performances compatibles avec l'utilisation fréquente correspondante aux micro-paiements. Le point critique à vérifier dans ce protocole est que l'allègement des calculs cryptographiques ne remet pas en cause la sécurité du télé-commerce. Pour cela, la spécification semi-formelle du protocole, telle qu'elle est présentée dans ce document, a justement pour but de préparer à la

prochaine étape : la formalisation. Par l'analyse formelle, nous allons essayer de montrer les propriétés du protocole, par exemple que la compromission de tel message, information, session ou de l'ensemble du protocole implique la perte de telle clé, graine, nonce et/ou la falsification de tel message. L'objectif est de démontrer l'absence, en cas d'attaque, de phénomènes de cascade entraînant la compromission de l'ensemble du protocole. Si une attaque est possible, nous devons montrer que ses conséquences doivent être limitées et que sa possibilité d'occurrence est suffisamment faible pour que la sécurité du système soit considérée comme raisonnable et acceptable.
Un prototype est en cours de développement en langage C en utilisant la librairie OpenSSL, qui a l'avantage de présenter un code source ouvert. Ce prototype va nous permettre de tester en pratique les hypothèses soulevées par l'analyse formelle. Le cas de la mobilité sera envisagé afin de déterminer comment celle-ci est compatible suivant les contraintes de sécurité mises en évidence par la formalisation.