RFC EnSkred

Ce document spécifie le protocole EnSkred, un protocole réseau permettant l'échange sécurisé ou non sécurisé de messages textuels entre applications. Chaque application est identifiée de manière unique par une clé publique RSA de 2048 bits. Les applications communiquent via des connexions TCP établies au sein d'un réseau organisé en graphe.

EnSkred propose deux modes de communication :

- Messagerie ouverte : transmission en clair des messages, suivant le chemin le plus court disponible dans le graphe du réseau.
- Messagerie sécurisée : transmission des messages par chiffrement multicouche afin d'assurer la confidentialité du contenu et de masquer l'identité du destinataire.

I. Format binaire des données et chiffrement RSA

I.1. Format binaire des données

Toutes les valeurs numériques sont représentées en format BigEndian et sont signées.

`BYTE` désigne un octet.

`INT` désigne un entier signé de 4 octet en BigEndian.

`LONG` désigne un entier signé de 8 octet en BigEndian.

`STRING` désigne une chaîne de caractères encodée en UTF-8 au format suivant :

```
STRING = size (INT) + string (BYTES)
```

où `size` est la longueur de la chaîne de caractères en octets et `string` est la chaîne de caractères encodée en UTF-8.

`PUBLIC_KEY` désigne une clé publique RSA 2048 bits encodée au format X.509 comme spécifié dans la RFC 5280. Le format est le suivant :

```
PUBLIC_KEY = size (INT) + public_key (BYTES)
```

où `size` est la longueur en octets de la clé publique encodée en X.509 et `public_key` désigne les octets de la clé publique encodée en X.509 comme défini dans la RFC 5280.

`IPADDRESS` désigne une addresse IPV4 ou IPV6, elle est representer sur le réseau comme une String (vu plus haut) avec la taille de l'adresse en début suivi de la chaine de caractère représentant l'adresse IP.

Une adresse de socket `SOCKETADDRESS` contenant un adresse IP et un numéro de port est représentée un `IPADDRESS` suivi d'un INT valant entre 0 et 65 535.

```
SOCKETADDRESS = adresse IP (IPADDRESS) + numéro de port entre 0 et 65 535 (INT)
```

`LIST(T)` désigne une liste de `T` représentée par un INT valant le nombre d'éléments de la liste suivi des différents éléments de la liste.

Dans le protocole, nous allons devoir échanger des informations sur la topologie du réseau : c'est à dire qui est connecté à qui et sur quelle adresse de socket les différentes applications acceptent de nouvelles connexions.

`NODE` représente la paire formée par une clé publique RSA et une adresse de socket.

```
NODE = PUBLIC_KEY + SOCKETADDRESS
```

`CONNEXION` représente la présence d'une connexion entre deux applications avec les deux clés publiques RSA des deux applications.

```
CONNEXION = PUBLIC_KEY + PUBLIC_KEY
```

I.2. Chiffrement RSA

Dans ce protocole, nous utiliserons des clés publiques RSA de 2048 bits. Le schéma de chiffrement de base sera RSA avec Optimal Asymmetric Encryption Padding (OAEP), dans lequel le processus de *padding* emploie la fonction de hachage SHA-256 ainsi que la fonction de génération de masque (MGF1). Ces procédés de chiffrement sont décrits dans la RFC 8017 et RFC 6234 (pour SHA-256).

Cette méthode permet de chiffrer des séquences d'octets de taille au plus 190 octets et produit en sortie une séquence d'octets de taille 256 octets quelque soit la taille de la séquence d'octets à chiffrer.

Cette méthode garantit que le même message, lorsqu'il est chiffré plusieurs fois avec la même clé, produit des textes chiffrés différents, renforçant ainsi la sécurité contre diverses attaques.

Nous allons maintenant décrire le format pour chiffrer une séquence arbitraire d'octets `payload` avec une clé publique RSA `public_key`. `payload` est une séquence d'octets arbitraire qui se découpe uniquement en une suite de blocs `bloc1` ,…, `blocn` avec tous les blocs (à l'exception du dernier) de 190 octets.

```
RSA(public_key, payload) = nb_blocs (INT) + encrypted_bloc1 (256 BYTES) + ... +
encrypted_blocn (256 BYTES)
```

avec `nb_blocs` le nombre `n` de blocs et `encrypted_bloc1` ,…, `encrypted_blocn` les séquences d'octets de taille 256 produites par le chiffrement RSA de `bloc1` ,…, `blocn` avec la clé publique `public_key` avec le procédé décrit plus haut.

II. Création et évolution du réseau

Dans cette section, nous allons décrire comment les applications rejoignent le réseau et comment le réseau évolue au cours du temps.

Pour cela, nous allons décrire les informations maintenues par les applications, comment les applications transmettent des informations à tout le réseau puis comment les applications rejoignent le réseau et enfin comment elles le quittent.

Toutes les trames du protocole sont préfixées par un octet (BYTE) donnant l'opcode de la trame. Dans la présentation les trames ont des noms et les opcodes correspondant sont indiqués dans une table à la fin de ce document.

II.1. Information maintenue par les applications

Dans tout le protocole, une application est uniquement identifiée par sa clé publique RSA qui est supposée unique au sein du réseau. Chaque application va maintenir une vue de la topologie du réseau: c'est à dire les adresses d'écoute des applications qu'elle connaît et les connexions entre les différentes applications. Cette vue locale peut différer d'une application à l'autre. Par exemple, une application peut être au courant d'une connexion entre deux applications alors que les autres applications ne le sont pas encore.

Plus précisément, chaque application maintient à jour plusieurs informations :

- une association entre les applications du réseau et les adresses de socket sur lesquelles elles acceptent de nouvelles connexions.
- un graphe non-orienté connexe représentant les connexions entre les applications.
- pour chacune de ces connexions, l'application correspondante.

II.2. Broadcast d'information de bas niveau

Avant présenter la manière dont les applications rejoignent le réseau et le quittent, nous allons montrer quelle information les applications échangent entre elles pour maintenir l'état du réseau.

La primitive de base pour maintenir l'état du réseau est le broadcast d'information de bas niveau. Cette primitive permet à une application de transmettre une information (BYTES) à toutes les autres

applications du réseau.

Pour communiquer une information `payload` (BYTES) à toutes les autres applications du réseau, une application transmet un trame de la forme suivante à toutes les applications à laquelle elle est connectée :

```
   BROADCAST = opcode (BYTE) + id_sender (PUBLIC_KEY) + message_id (LONG) + size (INT)
+ payload (BYTES)
```

avec `id_sender` la clé publique RSA de l'application qui transmet, `message_id` un identifiant unique de la trame, `size` la longueur en octets de `payload` et `payload` désigne la séquence d'octets représentant l'information à transmettre.

Quand une application reçoit une trame de ce type, elle va procéder ainsi :

1. Si elle a déjà reçu une trame avec le même `id_sender` et le même `message_id` , elle ignore la trame.
2. Sinon, elle interprète `payload` comme décrit dans les sections suivantes et elle transmet cette trame à l'identique à toutes les applications auxquelles elle est connectée (sauf l'application qui vient de lui transmettre la trame).

II.3. Connexion d'une application au réseau

Un application `sender` qui désire se joindre au réseau doit être en possession d'une paire de clés publique/privée RSA 2048 bits. La clé publique est utilisée pour identifier l'application et sera désignée par `id_sender` dans cette section. L'application doit aussi accepter de nouvelles connexions sur une socket d'écoute dont l'adresse sera désignée par `socket_address_sender` dans cette section.

1. L'application `sender` établit une connexion TCP avec une application du réseau que nous appellerons `receiver` .
2. L'application `sender` envoie une trame de type `PRE_JOIN` à l'application avec laquelle elle a établi la connexion et dans laquelle elle indique sa clé publique RSA.

```
PRE_JOIN = opcode (BYTE) + id_sender (PUBLIC_KEY) + socket_address_sender (SOCKETADDRESS)
```

3. En réponse à cette trame `PRE_JOIN`, l'application `receiver` envoie un challenge qui vise à garantir que l'application qui rejoint le réseau possède bien la clé privée RSA correspondant à la clé publique RSA indiquée dans la trame `PRE_JOIN`.

Le challenge est un LONG tiré aléatoirement par l'application `receiver` et chiffré avec la clé publique RSA de l'application `sender`. L'idée est que seule l'application `sender` qui possède la clé privée RSA correspondant à la clé publique RSA peut déchiffrer le challenge.

```
CHALLENGE_PUBLIC_KEY = opcode (BYTE) + challenge (RSA(id_sender, challenge))
```

4. En réponse, l'application `sender` envoie une trame de type `RESPONSE_CHALLENGE` dans laquelle elle indique le LONG obtenu en déchiffrant le challenge avec sa clé privée RSA.

```
RESPONSE_CHALLENGE = opcode (BYTE) + challenge_decrypted (LONG)
```

5. À la réception de cette trame, l'application `receiver` vérifie que `challenge_decrypted` est égal au challenge initialement généré par l'application `receiver`. Si ce n'est pas le cas, elle ferme la connexion.

   a. L'application `receiver` transmet une trame de type `JOIN_RESPONSE` à l'application `sender` dans laquelle elle indique sa clé publique RSA `id_receiver` ainsi que la topologie du réseau correspondant à la vue locale de `receiver`.

   ```
   JOIN_RESPONSE = opcode (BYTE) + id_recipient (PUBLIC_KEY) + LIST(NODE) + LIST(CONNEXION)
   ```

   b. L'application `receiver` transmet une trame de `BROADCAST` contenant le payload `NEW_NODE` à toutes les autres applications du réseau comme indiqué dans la section précédente.

   ```
   NEW_NODE = opcode_payload (BYTE) + id_sender (PUBLIC_KEY) + socket_address_sender (SOCKETADDRESS) + id_receiver (PUBLIC_KEY)
   ```

uand une application reçoit un broadcast contenant le payload `NEW_NODE`, elle met à jour sa vue du réseau en ajoutant le nœud et la connexion correspondants au payload.

6. L'application `sender` qui rejoint le réseau met à jour sa vue du réseau en ajoutant le nœud et la connexion correspondants aux informations de la trame `JOIN_RESPONSE`. À partir de ce moment, l'application est considérée comme membre du réseau.

7. Si c'est possible, l'application `sender` se connecte à une autre application (différente de `receiver`) du réseau.

II.4. Établissement d'une nouvelle connexion par une application membre du réseau

Dans cette section, nous allons décrire comment une application `sender` peut établir une nouvelle connexion avec une application `receiver`. Les deux applications doivent être membres du réseau. Autrement dit, elle doivent avoir rejoint le réseau en utilisant la procédure décrite dans la section précédente.

1. L'application `sender` établit une connexion TCP avec l'application `receiver` sur la socket `socket_address_receiver` où `receiver` accepte de nouvelles connexions. **sender envoie un paquet SecondJoin à receiver (semblable au PreJoin).**
   SECOND_JOIN = opcode (BYTE) + id_sender (PUBLIC_KEY) + socket_address_sender (SOCKET_ADDRESS)
2. L'application `receiver` va verifier que `sender` apparaît bien dans sa liste de nœuds. Si ce n'est pas le cas, `receiver` attend d'être notifié de la présence de `sender` dans le réseau. Cette situation peut survenir car l'information sur la présence de `sender` peut mettre du temps à se propager dans le réseau.

3. Une fois que `sender` apparaît dans la topologie de `receiver`, `receiver` envoie un challenge à `sender` pour vérifier que `sender` possède bien la clé privée RSA correspondant à la clé publique RSA indiquée dans la trame `CHALLENGE_PUBLIC_KEY`. L'application `sender` répond à ce challenge avec une trame de type `RESPONSE_CHALLENGE` comme dans la séquence de connexion au réseau. Si la réponse n'est pas correcte, `receiver` ferme la connexion, sinon elle envoie un CHALLENGE_OK à sender.
   CHALLENGE_OK = opcode (BYTE) + id_receiver (PUBLIC_KEY)
4. L'application `sender` broadcast une trame `BROADCAST` contenant le payload `NEW_CONNECTION` à toutes les autres applications du réseau.

```
NEW_CONNECTION = opcode_payload (BYTE) + id_sender (PUBLIC_KEY) + id_receiver
```

```
(PUBLIC_KEY)
```

5. Les applications `sender` et `receiver` mettent à jour leur topologie du réseau avec cette nouvelle connexion.

II.5. Déconnexion d'une application du réseau

Dans cette section, nous allons décrire comment une application `leaver` quitte le réseau. Les applications auxquelles `leaver` est directement connecté seront appelées `direct_neighbors`. Une application dont l'un des voisins est en train de quitter le réseau ne peut pas quitter le réseau tant que cette déconnexion n'est pas terminée. Pour cela, on entend une application qui a renvoyé une trame positive à la demande de déconnexion d'un autre de ses voisins (étape 2 ci-dessous).

1. L'application `leaver` envoie une trame de type `LEAVE_NETWORK_ASK` à ces `direct_neighbors`.

```
LEAVE_NETWORK_ASK = opcode (BYTE)
```

2. Une application qui reçoit une trame `LEAVE_NETWORK_ASK` répond par une trame de type `LEAVE_NETWORK_RESPONSE` avec un octet valant 1 si elle accepte la demande de quitter le réseau et 0 sinon. Elle accepte si elle n'est pas entrain de participer à la déconnexion d'un autre de ses voisins. Si elle a initié une demande de déconnexion simultanément à la réception de la trame `LEAVE_NETWORK_ASK`, elle répond par une trame de type `LEAVE_NETWORK_RESPONSE` avec un octet valant 0 si sa clé publique RSA est plus petite que celle de `leaver` et 1 sinon. Si elle répond 1, elle annule sa demande de déconnexion auprès de ses voisins.

```
LEAVE_NETWORK_RESPONSE = opcode (BYTE) + answer (BYTE)
```

3. L'application `leaver` attend les réponses de tous ses `direct_neighbors`.

   a. Si elle reçoit au moins un réponse négative, elle annule sa demande de déconnexion. Pour cela, elle envoie une trame de type `LEAVE_NETWORK_CANCEL` à ses `direct_neighbors`.

```
LEAVE_NETWORK_CANCEL = opcode (BYTE)
```

b. Si elle reçoit une réponse positive de tous ses `direct_neighbors`, elle envoie une trame de type `LEAVE_NETWORK_CONFIRM` à ses `direct_neighbors`.

```
LEAVE_NETWORK_CONFIRM = opcode (BYTE) new_connections (LIST(PUBLIC_KEY + SOCKETADDRESS))
```

où `new_connections` est une liste de couples contenant une clé publique RSA et l'adresse de socket à laquelle l'application qui reçoit la trame doit se connecter de manière à maintenir la connexité du réseau. Le protocole ne précise pas comment l'application `leaver` calcule cette liste. La seule contrainte est que le réseau doit rester connecté au départ de `leaver`. Un possibilité est de demander à chaque voisin de `direct_neighbors` de se connecter à chaque autre voisin de `direct_neighbors`. Cette solution n'est pas optimale mais elle est simple à mettre en œuvre. La liste `new_connections` peut être vide.

4. Une application qui reçoit une trame `LEAVE_NETWORK_CONFIRM` va établir chacune des connexions indiquées dans la trame en suivant la procédure d'établissement de connexion décrite dans la section précédente. Une fois que toutes les connexions sont établies, l'application renvoie une trame de type `LEAVE_NETWORK_DONE` à l'application qui lui a envoyé la trame `LEAVE_NETWORK_CONFIRM`.

```
LEAVE_NETWORK_DONE = opcode (BYTE)
```

5. L'application `leaver` attend d'avoir reçu une trame de type `LEAVE_NETWORK_DONE` de chacun de ses `direct_neighbors`. Puis elle broadcast à tout le réseau une trame `BROADCAST` contenant le payload `LEAVE_NETWORK_BROADCAST` le payload `REMOVE_NODE` à toutes les autres applications du réseau comme indiqué dans la section précédente.

```
REMOVE_NODE = opcode_payload (BYTE) + id_leaver (PUBLIC_KEY)
```

Quand une application reçoit une trame contenant le payload `REMOVE_NODE`, elle met à jour sa vue du réseau en supprimant le nœud et les connexions correspondantes au payload.

6. L’application `leaver` ferme toutes les connexions TCP avec ses `direct_neighbors` .

III. Communication ouverte

Dans cette section, nous allons décrire comment une application `sender` peut envoyer un message à une application `receiver` en utilisant le protocole de communication ouverte.

1. L’application `sender` envoie une trame de type `OPEN_MESSAGE` à un de ses voisins ayant la plus petite distance dans le graphe du réseau à l’application `receiver` .

```
OPEN_MESSAGE = opcode (BYTE) + id_sender (PUBLIC_KEY) + id_receiver (PUBLIC_KEY) + message (STRING)
```

2. Quand une application reçoit une trame de type `OPEN_MESSAGE` , si elle n’est pas la destinataire, elle transmet cette trame à sa voisine ayant la plus petite distance à `receiver` dans le graphe du réseau. Si `receiver` n’apparaît pas dans la topologie de l’application, il y a deux possibilités :

- `receiver` n’est jamais apparu dans le réseau. Dans ce cas, l’application attend d’être notifié de la présence de `receiver` dans le réseau. Comme le réseau est connexe, `receiver` finira par apparaître dans la topologie de l’application.
- `receiver` n’est plus dans la topologie de l’application car il a quitté le réseau. Dans ce cas, l’application ignore la trame.

NB : Le protocole garantit que le message sera reçu par `receiver` sauf dans le cas où `receiver` quitte le réseau.

IV. Communication sécurisée

Dans cette section, nous allons décrire comment une application `sender` peut envoyer un message à une application `receiver` en utilisant le protocole de communication sécurisée.

IV.1. Format des trames de communication sécurisée

Les trames pour la communication sécurisée sont définies récursivement comme suit :

```
SECURE_MESSAGE(recipient) = opcode (BYTE) + RSA(recipient, LIST(INSTRUCTION))

INSTRUCTION = PASS_FORWARD | MESSAGE

PASS_FORWARD = opcode_instruction (BYTE) + next_recipient (PUBLIC_KEY) +
SECURE_MESSAGE(next_recipient)
MESSAGE = opcode_instruction (BYTE) + sender (PUBLIC_KEY) + id_message (LONG) +
message (STRING)
```

La trame `SECURE_MESSAGE(recipient)` est une trame de type `SECURE_MESSAGE(recipient)` contient une liste d'`INSTRUCTION` encodée avec la clé publique RSA de `recipient`.

Une `INSTRUCTION` est soit de type `PASS_FORWARD` ou `MESSAGE`.

- `PASS_FORWARD` est une instruction qui contient la clé publique RSA de la prochaine application destinataire avec une trame `SECURE_MESSAGE(next_recipient)` encodée avec la clé publique RSA de `next_recipient`.
- `MESSAGE` est une instruction qui contient `PUBLIC_KEY` de l'application qui envoie le message, un identifiant de message (LONG) et le message (STRING).

IV.2. Comportement à la réception d'une trame SECURE_MESSAGE

Quand une application reçoit une trame de type `SECURE_MESSAGE(recipient)`, elle déchiffre la liste d'`INSTRUCTION` avec sa clé privée RSA. Elle exécute les instructions dans l'ordre.

IV.2.1. Instruction PASS_FORWARD

Quand une application reçoit une instruction de type `PASS_FORWARD`, il y a deux possibilités :

- `next_recipient` est la clé publique RSA d'une application avec laquelle elle a une connexion. Dans ce cas, la trame `SECURE_MESSAGE(next_recipient)` est transmise à cette application.
- `next_recipient` n'est plus connecté à l'application. Dans ce cas, la trame est ignorée.

IV.2.2. Instruction MESSAGE

A la réception d'une instruction de type `MESSAGE`, l'application décode le message.

IV.3. Transmission d'un message sécurisé

Comme les applications peuvent se déconnecter après l'envoi d'un trame `SECURE_MESSAGE`, il est possible que le message ne soit pas transmis à son destinataire final car l'une des applications sur le chemin planifié pour cette trame a quitté le réseau.

Les trames `SECURE_MESSAGE` sont donc prévues pour permettre la mise en place d'un système sécurisé d'acquittement de la réception.

L'application `sender` va choisir un chemin dans le graphe aussi complexe qu'elle le souhaite qui va passer par le destinataire final et revenir à `sender`. Ce message contiendra le message pour le destinataire final et un message d'acquittement pour `sender`.

V. Problèmes connus

- L'expéditeur des messages ouverts ou sécurisés peuvent être forgés. Ce problème est résolu par la signature cryptographique des messages.
- Il n'y a aucune limite sur la taille des messages.
- Le protocole ne donne pas de règle à suivre pour le choix des chemins pour la communication sécurisée.
- Le challenge sur la clé publique RSA est vulnérable par une attaque de type man-in-the-middle.
- L'utilisation de RSA pour chiffrer de longs messages est

inefficace et peut réaliste en pratique.

VI. Table des opcodes

Type de trame	Opcode
BROADCAST	1
PRE_JOIN	2
CHALLENGE_PUBLIC_KEY	3
RESPONSE_CHALLENGE	4
JOIN_RESPONSE	5
LEAVE_NETWORK_ASK	6
LEAVE_NETWORK_RESPONSE	7
LEAVE_NETWORK_CANCEL	8
LEAVE_NETWORK_CONFIRM	9
LEAVE_NETWORK_DONE	10
OPEN_MESSAGE	11
SECURE_MESSAGE	12

Type de trame	Opcode
SECOND_JOIN	20
CHALLENGE_OK	30

VI.1. Table des opcodes pour les payloads

Type de payload	Opcode
NEW_NODE	100
NEW_CONNECTION	101
REMOVE_NODE	102

VI.2. Table des opcodes pour les instructions

Type d’instruction	Opcode
PASS_FORWARD	200
MESSAGE	201
STOP	202