

~~CONFIDENTIEL DÉFENSE~~

~~SPECIAL FRANCE~~



Liberté • Égalité • Fraternité  
RÉPUBLIQUE FRANÇAISE

PREMIER MINISTRE

Secrétariat général de la  
défense et de la sécurité  
nationale

Agence nationale de la sécurité  
des systèmes d'information

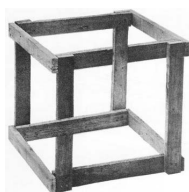
DÉCLASSIFIÉ  
par décision n°15699/ANSSI/SDE/ST/LAM  
du 18 juillet 2018



DOCUMENTATION CLIP  
1305

---

SUPPORT DES CARTES À PUCE SOUS CLIP



Ce document est placé sous la « Licence Ouverte », version 2.0 publiée par la mission Etalab

ANSSI, 51 boulevard de la Tour Maubourg, 75700 Paris 07 SP.

~~CONFIDENTIEL DÉFENSE~~

### Résumé

Ce document décrit l'intégration des cartes à puces dans le système CLIP pour l'authentification des utilisateurs, le déchiffrement de leur espace personnel et l'accès services rendus par les cartes depuis les cages RM par les applications (typiquement, logiciel de messagerie pour la signature et le déchiffrement des messages).

## HISTORIQUE

Révision	Date	Auteur	Commentaire
3.0	01/10/2014	Marion Daubignard	Mise à jour et description du fonctionnement du nouveau proxy
2.0	25/10/2012	ANSSI	Mise à jour majeure et ajout d'une section sur le provisionnement de cartes
1.0	06/01/2011	Benjamin Morin	Version initiale

## Table des matières

<b>1</b>	<b>Architecture générale</b>	<b>6</b>
<b>2</b>	<b>Support des lecteurs de cartes</b>	<b>7</b>
2.1	Paquetage pcsc-lite, drivers ccid . . . . .	7
2.2	Démon pcscd . . . . .	7
2.3	Communication de pcscd avec les lecteurs de cartes . . . . .	7
<b>3</b>	<b>Bibliothèques PKCS#11</b>	<b>8</b>
3.1	Bibliothèque OpenSC . . . . .	8
<b>4</b>	<b>Authentification par carte à puce</b>	<b>8</b>
4.1	Module PAM PKCS#11 . . . . .	8
4.2	Choix du certificat sur la carte . . . . .	8
4.3	Associations certificats / utilisateurs ( <i>mappers</i> ) . . . . .	9
4.4	Gestionnaires d'événements . . . . .	9
4.5	Configuration PAM pour l'ouverture de session . . . . .	9
4.6	Configuration PAM pour le verrouillage de sessions X11 . . . . .	10
4.7	Changement du code PIN du porteur de la carte . . . . .	10
<b>5</b>	<b>Déchiffrement des volumes de données utilisateur</b>	<b>10</b>
5.1	Principe de fonctionnement . . . . .	10
5.2	Outil de déchiffrement . . . . .	11
<b>6</b>	<b>Proxy PKCS#11</b>	<b>11</b>
6.1	Généralités sur le fonctionnement du proxy . . . . .	11
6.2	Filtrage réalisé par le proxy . . . . .	12
<b>7</b>	<b>Du point de vue utilisateur du poste CLIP : éléments de configuration pour utilisation de la carte à puce depuis une cage</b>	<b>13</b>
<b>A</b>	<b>Module PAM PKCS#11</b>	<b>15</b>
A.1	Configuration pam_pkcs11 . . . . .	15
A.2	Configurations PAM . . . . .	15
A.3	Utilitaires pam_pkcs11 . . . . .	16
<b>B</b>	<b>Déchiffrement de données avec OpenSSL</b>	<b>16</b>
<b>C</b>	<b>Enfermement du proxy PKCS#11 dans une cage VServer</b>	<b>17</b>
<b>D</b>	<b>Provisionnement des cartes IAS-ECC</b>	<b>18</b>
D.1	Via PKCS#11 . . . . .	18
D.2	Via PKCS#15 . . . . .	19
<b>E</b>	<b>Fichier de configuration de la carte à puce</b>	<b>20</b>
<b>F</b>	<b>Configuration du proxy PKCS#11</b>	<b>20</b>
F.1	Configuration générale du proxy . . . . .	20
F.2	Configuration du filtrage . . . . .	21
F.2.1	Fichier de configuration pour le filtre du socle . . . . .	22

F.3	Fichier de configuration pour le filtre du socle . . . . .	22
G	Attributs conflictuels	23

## 1 Architecture générale

L'accès aux services d'une carte à puce par une application repose sur un empilement d'au moins deux couches logicielles : la première assure les communications avec le lecteur de carte. Il existe deux logiciels majeurs de ce type sous UNIX, `pcsc` et `openct`. La seconde couche est chargée de la traduction des opérations cryptographiques de haut niveau (demandées par les applications) en commandes APDU<sup>1</sup> envoyées à une carte via un lecteur. Cette couche logicielle se présente sous la forme d'une bibliothèque chargée dynamiquement par les applications. Les fonctions de haut niveau associées aux opérations que peut rendre la carte sont standardisées (standard PKCS#11, aussi connu sous le nom Cryptoki). Il existe généralement une bibliothèque PKCS#11 par modèle de carte, ces bibliothèques étant fournies par les fabricants de cartes. Néanmoins, les cartes répondant à des standards, comme IAS-ECC par exemple, présentent l'avantage de pouvoir être utilisées avec des bibliothèques ``génériques'', libres et dont le code est ouvert, comme `opensc`.

L'intégration des cartes à puces dans CLIP utilise une couche logicielle supplémentaire sous la forme d'un proxy PKCS#11 qui s'intercale entre les processus applicatifs souhaitant accéder à une carte et la bibliothèque PKCS#11 effectivement utilisée. Ce proxy vise à isoler le démon `pcsc-lite` du reste du système, d'une part pour limiter les risques associés à l'exploitation d'une vulnérabilité présente dans la bibliothèque PKCS#11, et d'autre part pour permettre de filtrer<sup>2</sup> les accès aux cartes (certificats, clés, etc.) en fonction de la cage depuis laquelle est émise la requête.

La figure suivante schématise les relations entre ces couches :

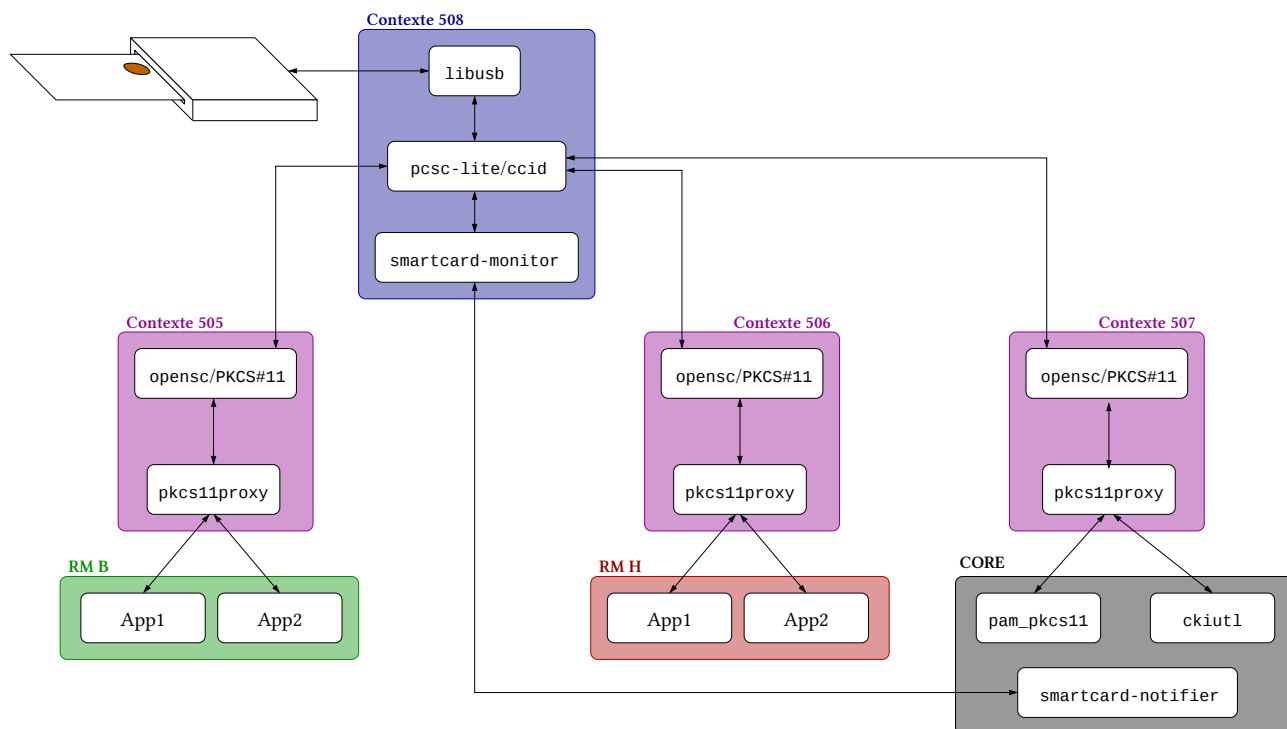


FIGURE 1 – Architecture logicielle de l'intégration des cartes à puce dans CLIP.

Les sections suivantes du document décrivent successivement les éléments logiciels nécessaires aux communications avec les cartes à puce (pilotage des lecteurs de cartes, bibliothèque PKCS#11), la façon dont sont implantées les services attendus des cartes à puce (authentification de l'utilisateur, déchiffrement des volumes de données) et le service de proximité mise à disposition pour traiter les requêtes envoyées aux cartes dans


1. Application Payload Data Unit


2. À noter que la fonction de filtrage du proxy n'est pas encore implémentée.

les cages RM.

## 2 Support des lecteurs de cartes

### 2.1 Paquetage `pcsc-lite`, drivers `ccid`

 `portage-overlay/sys-apps/pcsc-lite`

 `portage-overlay/app-crypt/ccid`

Le paquetage `pcsc-lite` contient un démon, `pcscd`, qui assure la gestion des lecteurs de cartes branchés ainsi que la communication par APDUs avec les cartes qu'ils contiennent. Il se charge en particulier de gérer l'accès de manière concurrente/exclusive à un lecteur, et donc à la carte qu'il contient.

Seul, `pcsc-lite` ne supporte aucun lecteur. Il est nécessaire de lui fournir des drivers qui répondent à l'IFDHandler API pour chaque lecteur que l'on souhaite utiliser, présents sous forme de bibliothèques. Dans CLIP, le driver générique (et libre) `ccid` pour lecteurs USB a été intégré, ce qui implique que tout lecteur compatible avec cette norme est supporté *de facto*. Les lecteurs Omnikey, les modèles Gemalto ainsi que les lecteurs intégrés dans les portables Dell testés sont ainsi supportés.

### 2.2 Démon `pcscd`

Le démon `pcscd` est lancé automatiquement par `start-stop-daemon` au cours du démarrage du poste<sup>3</sup>, qui procède à une descente de privilèges pour que `pcscd` s'exécute dans une cage dédiée (contexte 508). La descente de privilèges est effectuée dans le code du démon `pcscd`, ce qui nécessite l'application d'un patch du code du package `pcsc-lite` spécifique à CLIP.

### 2.3 Communication de `pcscd` avec les lecteurs de cartes

Dans le cas de CLIP, seuls les lecteurs USB sont activés à la compilation. `pcsc-lite` s'appuie sur `libusb` pour énumérer les périphériques USB, identifier les lecteurs de cartes à puce et enfin pour communiquer avec les lecteurs USB supportés. Ces opérations nécessitent de pouvoir :

- traverser et lire les entrées se trouvant sous `/sys/bus/usb` ;
- traverser `/dev/bus/usb` et lire les entrées qui s'y trouvent coorespondant à des lecteurs de cartes à puce.

`pcsc-lite` demande périodiquement (toutes les secondes) à `libusb` de lui énumérer les périphériques USB branchés.



#### Invocation de scripts de création et de suppression des nœuds de périphériques

Pour mémoire, avant le passage en version 1.6.6 et supérieure de `pcscd` et la modification du fonctionnement de `hotplug-clip`, la création et la suppression des périphériques à proprement dit était assurée par les scripts `/sbin/scarddev.ADD.sh` et `/sbin/scarddev.DEL.sh`, respectivement. Ces scripts font partie sur paquetage `portage-overlay-clip/app-clip/smartcard-readers`. Ils sont invoqués soit par le script `/etc/init.d/sccreatedev` (même paquetage) pour la création des périphériques au cours du démarrage du poste, au cas où un lecteur de carte serait déjà branché sur la machine. Les scripts peuvent aussi être invoqués par le module `scardreader.c` du gestionnaire `hotplug-clip`.

3. cf. script `/etc/init.d/pcscd`, créé lors de l'installation du paquetage

### 3 Bibliothèques PKCS#11

Les bibliothèques PKCS#11 contiennent les fonctions cryptographiques normalisées de haut niveau (signature, chiffrement, etc.) et assurent l'envoi des APDUs correspondant à une carte à puce donnée, avec laquelle elles communiquent via pcsc-lite.

#### 3.1 Bibliothèque OpenSC

 portage-overlay/dev-libs/opensc

Le projet *open source* OpenSC permet d'utiliser plusieurs modèles de cartes distincts. Il fournit un certain nombre de fonctions génériques utilisables par les modules associés à chaque type de cartes. Les cartes doivent être compatibles avec le standard PKCS#15 pour être supportées par OpenSC. À défaut, il est nécessaire de recourir à des artifices (parfois peu élégants) pour utiliser les cartes.

La suite logicielle OpenSC permet d'utiliser pleinement l'application PKI des cartes au standard IAS-ECC, y compris pour leur provisionnement, et supporte le Secure-Messaging (établissement d'un canal chiffré pour communiquer avec la carte).

### 4 Authentification par carte à puce

 portage-overlay/sys-auth/pam\_pkcs11

#### 4.1 Module PAM PKCS#11

L'authentification des utilisateurs sur le poste repose sur le module `pam_pkcs11`. Ce dernier charge la bibliothèque PKCS#11 (en réalité, le client du proxy PKCS#11, cf. section 6) qui lui est désignée dans son fichier de configuration (`/etc/pam_pkcs11/pam_pkcs11.conf`), tire un aléa qu'il transmet à la carte pour signature (RSA) et vérifie finalement la signature fournie par la carte. Cette vérification utilise les certificats racine et les listes de révocation présentes sur le poste dans le répertoire `/etc/admin/pkcs11/cacerts|crls`.



##### Listes de révocation

Actuellement les CRLs ne sont pas prises en compte. Il sera nécessaire de mettre en place une PKI lors du déploiement des cartes à puce.

À noter que les CRL peuvent être locales ou distantes.

L'annexe A fournit des informations complémentaires sur la configuration et les utilitaires de mise au point inclus dans `pam_pkcs11`.

#### 4.2 Choix du certificat sur la carte

Par défaut, `pam_pkcs11` énumère l'ensemble des certificats présents sur la carte à la recherche d'un certificat candidat à utiliser pour l'opération de signature. Un patch a été ajouté pour pouvoir désigner un certificat spécifique à l'aide du *label* associé à ce certificat sur la carte. Il est possible de spécifier le label dans le fichier de configuration de `pam_pkcs11`, ainsi que dans les arguments fournis dans une configuration PAM (en l'occurrence, pour le certificat utilisé pour l'authentification, `label=clip_auth`).



##### Usages des certificats


La vérification de la cohérence du certificat à partir des champs `KeyUsage` vis à vis de l'usage qui en est effectivement fait lors de l'authentification de l'utilisateur n'est pas encore implémentée.



### 4.3 Associations certificats / utilisateurs (*mappers*)

L'association entre un certificat et un utilisateur est réalisée à l'aide de *mappers*. Plusieurs types de *mappers* sont disponibles dans `pam_pkcs11` (`ldap`, `cn`, `krb`, `uid`, etc.); ils se présentent sous la forme de bibliothèques chargées dynamiquement. Certains permettent de faire une association locale (par exemple avec le fichier `/etc/passwd`) ou distante (par exemple, avec les utilisateurs définis au niveau d'un annuaire LDAP). Dans le cas de CLIP, nous optons pour une association locale, via un fichier *ad-hoc*, qui liste les correspondances entre un DN (présent dans les certificats X.509) et un nom d'utilisateur. Ce fichier est stocké dans `/home/etc.users/subject_mapping`.

### 4.4 Gestionnaires d'événements


 `portage-overlay-clip/app-clip/smartcard-monitor`

L'exécutable `pkcs11_eventmgr`, inclus dans `pam_pkcs11`, exécute des opérations paramétrées à la réception d'événements correspondant à l'insertion ou le retrait d'une carte. Les scripts exécutés se trouvent dans les fichiers auxiliaires du paquetage `pam-pkcs11`. Ce gestionnaire d'événements n'est pas utilisé dans CLIP car il s'appuie sur PKCS#11 pour détecter les insertions/retraits de carte, ce qui est inutile et particulièrement lourd si l'on considère qu'il passe également par le proxy PKCS#11. Un second exécutable, `card_eventmgr`, également inclus dans `pam_pkcs11` ne souffre pas de ce travers, mais il n'est plus manifestement plus maintenu par les auteurs. Qui plus est, ces deux utilitaires ne gèrent pas correctement les retraits de lecteur, surtout si lorsque celui-ci contenait une carte à son retrait.

Le package `smartcard-monitor` a donc été développé pour palier aux défauts des utilitaires existants et ainsi gérer proprement les insertions/retraits de lecteurs/cartes. Concrètement, ce package se décompose en deux démons. `smartcard-monitor`, d'une part, qui s'interface directement avec `pcscd` aux côtés duquel il est engagé pour permettre la détection des insertions/retraits de lecteurs/cartes et émettre les notifications correspondantes via une socket UNIX. `smartcard-notifier`, d'autre part, est exécuté dans le socle et se contente de se connecter à la socket UNIX de `smartcard-monitor` afin d'y lire les événements émis et d'invoquer une commande à chaque événement.

Concrètement, `smartcard-notifier` exécute le script `xcardactions.sh`, également fourni dans le package, à chaque événement pour déclencher l'économiseur d'écran et réactiver la saisie du code PIN de l'utilisateur au retrait et à l'insertion de la carte, respectivement.

### 4.5 Configuration PAM pour l'ouverture de session

 `portage-overlay/sys-apps/shadow`

L'extrait de configuration PAM de `system-auth` suivant, restreint à l'étape d'authentification (type `auth`) permet d'utiliser le module `pam_pkcs11` dans un processus d'authentification par carte à puce ou l'authentification par mot de passe (via `pam_tcb`) :

<code>auth</code>	<code>required</code>	<code>pam_env.so</code>
<code>auth</code>	<code>[success=ok default=1]</code>	<code>pam_wheel.so group=pkauth use_login trust</code>
<code>auth</code>	<code>[success=1 default=die]</code>	<code>pam_pkcs11.so</code>
<code>auth</code>	<code>[success=ok default=die]</code>	<code>pam_tcb.so shadow fork prefix=\$2a\$ count=8</code>

La signification des nouvelles directives PAM est rappelée pour mémoire en annexe [A.2](#).

Le module `pam_wheel` permet d'aiguiller la méthode d'authentification de l'utilisateur (par mot de passe ou par carte à puce) en fonction de son appartenance au groupe `pkauth`<sup>4</sup>, ce qui évite de réaliser systématiquement une tentative d'authentification par mot de passe. En cas de succès de `pam_wheel` (`success=ok`), PAM utilise `pam_pkcs11` ; en cas d'échec, PAM saute (`default=1`) directement à `pam_tcb`.

4. L'ajout d'un utilisateur à ce groupe est réalisée à la création ou la modification d'un compte utilisateur.

**Modification de pam\_wheel**

Une modification de `pam_wheel` est nécessaire pour réaliser le choix du mode d'authentification (portage-overlay/sys-libs/pam, fichier `pam_wheel.c`).

Cette configuration commune est complétée par la configuration spécifique de `slim` suivante (contenue dans le paquetage `shadow`), qui permet de mémoriser le mot de passe ou le code PIN saisi par l'utilisateur pour permettre le déchiffrement des partitions utilisateur au cours de l'étape session de PAM :

```
auth [success=done default=die] pam_exec_pwd.so
```

#### 4.6 Configuration PAM pour le verrouillage de sessions X11



portage-overlay-clip/sys-auth/pwcheckd

La réauthentification des utilisateurs suite au verrouillage d'une session X11 est assurée sous CLIP par le démon *pwcheckd* (c.f. [1], §4.4). Ce démon utilise PAM pour vérifier le mot de passe des utilisateurs. Il suffit donc de modifier la configuration PAM de *pwcheckd* pour utiliser le module `pam_pkcs11` pour réauthentifier les utilisateurs par carte à puce.

La pile PAM utilisée pour le déverrouillage est similaire à la configuration commune donnée dans la section précédente :

```
auth [success=ok default=1] pam_wheel.so group=pkauth use_login trust
auth [success=1 default=die] pam_pkcs11.so
auth [success=ok default=die] pam_tcb.so shadow fork prefix=$2a$ count=8
```

#### 4.7 Changement du code PIN du porteur de la carte



portage-overlay-clip/app-clip/userd-server

### 5 Déchiffrement des volumes de données utilisateur



portage-overlay-clip/app-clip/clip-user-mount

#### 5.1 Principe de fonctionnement

Le déchiffrement des volumes de stockage de l'utilisateur intervient après son authentification (cf. section précédente). L'utilisateur dispose d'un bicle RSA pour le déchiffrement de ses partitions. À noter que ce bicle est différent de celui utilisé pour l'authentification de l'utilisateur. Il dispose à ce titre d'un label spécifique, `clip_disk`, différent de celui du bicle utilisé pour l'authentification. La partie publique est utilisée pour chiffrer une clé maître symétrique, qui est impliquée dans le chiffrement des différentes partitions. La partie privée est quant à elle stockée dans la carte de l'utilisateur.

Cette solution est préférable à celle consistant à chiffrer individuellement chacune des clés symétriques associées aux volumes utilisateur (`rm_b`, `rm_h` et `user`) car elle permet de diviser par trois le nombre d'opérations de déchiffrement effectuées par la carte <sup>5</sup> à l'ouverture d'une session. Cette clé maître est ensuite utilisée pour déchiffrer les clés de chiffrement des volumes utilisateur, sans recours à la carte.

La clé maître chiffrée est stockée dans `/home/keys/<user>.masterkey`. Elle est déchiffrée par la carte à puce de l'utilisateur dans la fonction `do_mounts()` du script `mount.clip` puis exportée dans la variable

5. Selon la longueur des modules RSA et le type de carte utilisé, chaque opération de déchiffrement peut prendre plusieurs secondes.

d'environnement MASTERKEY pour être accessible aux scripts qui sont invoqués par la suite, en particulier `mount.crypt`. Le déchiffrement de la clé maître est conditionnée par l'appartenance de l'utilisateur au group `pkauth`.



#### Déchiffrement de la clé maître

L'opération de déchiffrement de la clé maître par la carte a lieu dans le script `/sbin/mount.clip` plutôt que `mount.crypt` car l'ouverture de session donne lieu à autant d'invocations de `mount.crypt` par `mount.clip` que de partitions à déchiffrer. Le recours à la carte ne doit avoir lieu qu'à la première invocation ; les invocations suivantes s'appuient sur la valeur de MASTERKEY, qui serait « oubliée » à chaque nouvelle invocation de `mount.crypt` par `mount.clip` si elle avait lieu dans `mount.crypt`.

Les clés de déchiffrement des volumes sont stockées chiffrées (avec la clé maître) sur le poste dans les sous-répertoires de `/home/keys/`, avec l'extension `.key.enc` (pour les distinguer des clés chiffrées avec une clé dérivée du mot de passe de l'utilisateur, qui ont l'extension `.key`).

## 5.2 Outil de déchiffrement



`portage-overlay-clip/sys-auth/ckiutl`

Les versions initiales de CLIP intégrant les cartes à puce pour le déchiffrement des volumes de données de l'utilisateur s'appuyaient sur OpenSSL. Cette configuration est donnée pour mémoire en annexe B. Compte-tenu de la lourdeur de configuration d'OpenSSL dans les interactions avec les cartes à puces, un outil *ad-hoc* de déchiffrement, `ckiutl`, a été développé et est utilisé dans les versions actuelles de CLIP.

L'outil `ckiutl` s'appuie sur la bibliothèque d'abstraction des fonctions PKCS#11, `libp11`. Cet outil se contente actuellement d'appeler la fonction de déchiffrement d'une carte à puce en utilisant la clé privée du bclé portant le bon label (`clip_disk`).

## 6 Proxy PKCS#11



`portage-overlay-clip/app-crypt/pkcs11-proxy`

### 6.1 Généralités sur le fonctionnement du proxy

Le chargement dynamique de bibliothèques PKCS#11 potentiellement non maîtrisées rend leur isolation du reste du système préférable sur le plan de la sécurité, d'autant que ces bibliothèques peuvent être utilisées par des processus privilégiés (PAM en particulier).

CLIP utilise un service de proximité pour émettre des requêtes PKCS#11 afin d'isoler le code des bibliothèques PKCS#11, tout en permettant aux applications qui en ont besoin de communiquer avec les cartes à puce. Les applications concernées correspondent non seulement à celles qui sont exécutées dans les cages utilisateur RM (logiciel de messagerie, navigateur, etc.), mais aussi dans le socle (modules PAM d'authentification, de déchiffrement des volumes utilisateur et de déverrouillage de l'écran).

Le proxy PKCS#11 est une architecture client/serveur(démon). Les communications sont réalisées via une socket UNIX. La partie cliente est une bibliothèque partagée (`libp11client.so`<sup>6</sup>) qui expose une interface PKCS#11 pour les applicatifs. Elle fonctionne en suivant le modèle RPC (Remote Procedure Call) et sérialise les appels de fonction, les paramètres et résultats de manière transparente vis-à-vis des applications souhaitant accéder aux services de la carte.

6. L'usage du nom historique de la librairie `pk11proxy.so` est toujours possible grâce à un lien symbolique dans le socle vers la librairie actuelle.

La partie serveur est isolée du reste du système dans une cage VServer. Le démon, nommé `pkcs11proxyd` est en attente de connexions émanant de la partie cliente sur la socket. Lorsqu'un client se connecte, le middleware PKCS#11 natif est chargé par `pkcs11proxyd`. Puisque le démon doit pouvoir dialoguer avec `pcscd`, il faut donc exposer la socket UNIX pour échanger avec celui-ci.

En pratique, il existe autant de démons PKCS#11 que de contextes d'utilisation de la carte à puce qu'il est souhaitable de cloisonner. Ainsi, dans la configuration actuelle de CLIP, trois cages sont utilisées pour isoler les démons PKCS#11. Deux pour les cages utilisateurs RMH et RMB et une troisième pour les applicatifs du socle. La configuration des cages doit être adaptée pour accéder à la socket de communication avec `pkcs11proxyd`. Les paramètres des cages VServer correspondantes sont définies dans `/etc/jails/p11proxy_J` où J correspond au nom d'une cage. Pour information, le procédé d'enfermement du démon est donné en annexe C. La socket UNIX de communication avec `pcsc-lite` est exportée dans chacune des cages de démons PKCS#11 par un montage `bind` de `/var/run/pcscd`).

Le proxy actuellement disponible dans CLIP est implémenté en OCaml. Le projet est disponible sur le GitHub de l'ANSSI [?] et l'architecture du projet est documentée dans l'article *Buy it, use it, break it ... fix it : Caml Crush, un proxy PKCS#11 filtrant* [?]. Sans entrer dans les détails, on relève un choix architectural dimensionnant dans l'implémentation : au contraire des solutions de proxyfication existantes qui utilisent des *threads* permettant aux applications concurrentes d'accéder à une ressource cryptographique via une unique instance du middleware PKCS#11 sous-jacent, le choix d'un modèle *fork-exec* permet d'isoler les applications les unes des autres. Ceci améliore la sécurité fournie par la solution globale, en ce sens que l'accès aux objets cryptographiques obtenu par une application après authentification n'est pas possible à une application concurrente malveillante. Les modifications de Caml Crush dans CLIP se limitent à l'ajout des éléments liés à l'enfermement des instances du proxy lancées dans des cages.

## 6.2 Filtrage réalisé par le proxy

Le proxy comporte un composant dédié aux fonctionnalités de filtrage. Tous les appels soumis au démon `pkcs11proxyd` traversent le filtre. Ce dernier peut les bloquer ou les modifier avant de relayer à la couche inférieure. Les sorties correspondantes sont à leur tour soumises au filtre, qui peut leur appliquer un post-traitement avant de finalement renvoyer une réponse définitive à l'applicatif appelant.

L'implémentation du filtre permet de mettre en oeuvre les politiques de sécurité suivantes.

- **Filtrage de mécanismes cryptographiques jugés faibles.** Même s'il est disponible et implémenté par la ressource cryptographique sous-jacente, un mécanisme filtré ne sera pas visible ou utilisable par les applications.
- **Filtrage des objets présents dans la ressource.** Le filtre permet de dissimuler tous les objets présents sur une ressource ne disposant pas d'un label et/ou d'un identifiant ("ID") parmi ceux autorisés.
- **Restrictions sur les fonctions autorisées.** On peut mettre sur liste noire des fonctions de l'interface qu'on ne souhaite pas exposer aux applications. Il existe également la possibilité d'interdire toute modification des objets de la ressource (i.e. n'autoriser que les opérations spécifiées par le standard comme étant disponible dans une session ouverte pour lecture seule). De plus, il est possible d'interdire systématiquement toute opération d'administration (au sens d'opération disponible pour un *Security Officer* enregistré sur la ressource).
- **Actions de filtrage permettant de bloquer des attaques logiques sur la ressource cryptographique.** Un jeu de règles (`patchset`) implémentant des corrections potentielles est disponible. Lorsque celui-ci est activé, la création, modification ou duplication des objets non-générés sur la ressource est bloquée. L'existence sur la ressource d'objets présentant des attributs conflictuels est rendue impossible. La liste des attributs jugés conflictuels est fournie en annexe G. De plus, des attributs (comme celui qui qualifie le caractère sensible d'une clé) sont rendus non-modifiables une fois qu'ils ont pris certaines valeurs faisant des objets qu'ils qualifient des objets à protéger. Les mécanismes d'injection et l'export de clés sont modifiés de manière à autoriser le transport avec l'objet cryptographique de ses attributs, la co-

existence dans une ressource d'objets avec des attributs conflictuels étant une source d'attaque classique.

Le moteur de filtrage est géré au moyen d'un fichier de configuration. Il existe dans CLIP un fichier par instance du démon (actuellement socle, RMH, RMB). Le détail des configurations est fourni pour référence en annexe F, seul un bref survol des options activées est fourni ici. Dans la configuration du socle, le patch est activé, et la seule autre restriction imposée concerne les labels des objets. Seuls les besoins de l'authentification locale justifient l'exposition d'une carte à puce dans le socle. Ainsi, la configuration par défaut impose que les objets exposés portent les labels `clip_auth` et `clip_disk`, qui sont les labels par défaut des objets servant à l'authentification locale dans le fichier `smartcards` (cf section E). On note donc que toute modification de configuration devra être homogène sur ce point, ce qui est détaillé dans la section 7.

En ce qui concerne les configurations par défaut pour les cages, un certain nombre de fonctions sont interdites : en bref, on peut déchiffrer ainsi que créer ou vérifier une signature avec une clé de la ressource. Les opérations liées à l'administration sont interdites, et seul l'accès en lecture est possible. Enfin, dans la cage RMB (respectivement RMH), les objets exposés doivent porter un label préfixé par `rm_b` (respectivement `rm_h`).

## 7 Du point de vue utilisateur du poste CLIP : éléments de configuration pour utilisation de la carte à puce depuis une cage

Dans un premier temps, il faut faire en sorte d'activer le lancement du proxy pour la cage voulue. Ceci peut se faire de deux manières : soit grâce à l'interface graphique disponible au travers du menu "Configuration et Administration" de la barre de confiance (accessible à un utilisateur possédant le rôle ADMIN), en sélectionnant "Périphériques", puis "Attribution des périphériques". Dans l'onglet "Cartes à puce" de la fenêtre ouverte, le champ "Exposition de la carte à puce" permet de sélectionner les cages dans lesquelles on souhaite exposer une carte. Modifier ce fichier résulte en la mise à jour d'une variable d'environnement nommée `PKCS11_JAITS`, définie dans le fichier nommé `smartcards`<sup>7</sup>, accessible et modifiable plus amplement depuis un terminal Admin, dans le répertoire `/etc/admin/conf.d/smartcards`. Concrètement, cocher une cage via l'interface graphique ou ajouter manuellement le nom de la cage à `PKCS11_JAITS` va provoquer le lancement d'une instance du proxy pour cette cage *lors du prochain démarrage du service* `/etc/init.d/pkp11proxy`. Toute édition de ce fichier sur un poste non-instrumenté nécessite un redémarrage pour sa prise en compte. Le bon lancement du proxy se traduira par un message dans les journaux de démarrage.

Dans un second temps, il est nécessaire de configurer le proxy pour chacune des cages qui l'utilise. Les éléments de configuration du proxy sont disponibles dans le répertoire `/etc/admin/conf.d/pkcs11proxyd-conf/`. Ils sont accessibles et modifiables depuis un terminal Admin. Ce répertoire contient un répertoire pour chacune des cages pouvant utiliser une instance du proxy : `core/`, `rm_h/` et `rm_b/`. Ces trois répertoires ont un contenu identique par défaut. Ils contiennent deux sous-répertoires, `conf/` et `filter/`, contenant chacun un fichier. Le premier contient des éléments de configuration liés à l'architecture du proxy, le second des éléments liés au dispositif de filtrage qui modifie les appels PKCS#11 transmis à la couche inférieure. L'utilisateur classique ne sera certainement intéressé que par la modification des paramètres suivants liés au filtrage :

- le champ `modules` spécifie la bibliothèque chargée suivant la chaîne de caractères demandée par le client. Par défaut, cette chaîne est vide et on tente de charger la bibliothèque spécifiée correspondant à la chaîne vide ;
- le champ `allowed_labels` permet de spécifier les labels visibles.

De plus amples détails sur le contenu des fichiers sont fournis dans l'annexe F.

Enfin, s'il s'agit du socle et de la mise en place de l'authentification locale par carte à puce, il faut s'assurer que les labels des objets utilisés pour les opérations de signature et déchiffrement décrites dans les sections 4 et 5 sont bien ceux des objets conservés dans la carte à puce. Pour cela, il faut récupérer les bons labels dans

7. Le contenu complet de ce fichier de configuration est détaillé en annexe E.

la carte à puce (manuellement, si ces informations ne sont pas données lors du provisionnement). Ensuite, en lançant un terminal Admin, on peut accéder au fichier de configuration `/etc/admin/conf.d/smartcards`, et vérifier que le label de la clé de signature est celui qui est affecté à la variable `PKCS11_AUTH_LABEL`, et que celui de la clé de déchiffrement est affecté à la variable `PKCS11_DISK_LABEL` (le tout accolé au = et sans guillemets).

Enfin, lors de mises à jour ou de changements de version de CLIP, les fichiers de configuration correspondant aux éventuelles nouvelles configuration fournies ne sont pas adoptés d'office comme nouvelles configurations par défaut. Ceci évite de rendre inopérant le système et de perdre les informations de configuration pertinentes. Les nouveaux fichiers apparaîtront aux mêmes emplacements que leurs homologues, avec un nom préfixé par un `.` et le suffixe `.confnew`. Il appartient alors à l'administrateur du système de personnaliser et renommer ces nouveaux fichiers de configuration. Dans tous les cas, les fichiers portant les noms spécifiés plus haut (`smartcards` et `*.conf`) seront ceux utilisés par le filtre.



## A Module PAM PKCS#11

### A.1 Configuration pam\_pkcs11

L'extrait du fichier de configuration suivant correspond à la définition d'un module PKCS#11 :

```
pkcs11_module p11proxy {
    module = /usr/lib/p11proxy.so
    description = "P11 proxy client module";
    slot_description = "none";
    ca_dir = /etc/pam_pkcs11/cacerts;
    crl_dir = /etc/pam_pkcs11/crls;
    support_threads = true;
    cert_policy = ca,signature;
    label={\${1}};
}

use_mappers = subject;
mapper_search_path = /usr/lib/pam_pkcs11;

mapper subject {
    debug = false;
    module = internal;
    ignorecase = false;
    mapfile = file ://home/etc.users/subject_mapping;
}
```

On peut utiliser de façon exclusive `slot_description` ou `slot_num` pour désigner le slot à utiliser. Privilégier `slot_description` car la spécification PKCS#11 ne garantit pas d'ordre des slots. La valeur `none` prend le premier slot trouvé.

`cert_policy` précise la politique de vérification (plusieurs valeurs peuvent être spécifiées) : `none`, `ca` (CA check), `crl_online` (téléchargement de CRL), `crl_offline` (utilisation des CRL locales), `crl_auto` (tentative online ; offline en cas d'échec de connexion), `signature` (vérification de signature).

`use_mappers` : liste des mappers (`cert` ↔ `login`) à utiliser. Un certain nombre sont supportés (`generic`, `subject`, `openssh`, `opensc`, `pwent`, `null`, `ldap`, `cn`, `mail`, `ms`, `krb`, `uid`, `digest`). Certains nécessitent une bibliothèque pour opérer le mapping et/ou un fichier pour spécifier explicitement le mapping, d'autres s'appuient sur la comparaison entre un champ du certificat X509 (e.g., `CN`) avec les logins définis sur la machine (`pwent`).

### A.2 Configurations PAM

Pour mémoire, la syntaxe des fichiers de configuration PAM a changé (tout en assurant la compatibilité de l'ancienne syntaxe, cf. document de configuration de PAM). La signification des codes de retour des modules PAM utilisés dans les fichiers de configuration de CLIP sont les suivants :

- `success` désigne un code de retour `PAM_SUCCESS` du module concerné ;
- `auth_err` désigne un code de retour `PAM_AUTH_ERR` du module concerné ;
- `default` désigne tous les codes de retour autres que ceux précédemment énumérés.

La signification des actions associées à ces codes de retour sont les suivantes :

- un entier  $n$  (dans ce cas,  $n = 1$ ) signifie qu'il faut sauter les  $n$  modules PAM suivant dans la pile de modules ;

- ignore signifie que le code de retour du module ne sera pas pris en considération dans le retour global de PAM.
- ok signifie que le code de retour devrait être pris en considération pour le résultat global de PAM (ce code de retour remplacerait un code précédent égal à PAM\_SUCCESS, mais pas un code de retour équivalent à un échec);
- done est similaire à ok, mais termine immédiatement le processus d'authentification;
- die termine immédiatement le processus d'authentification de PAM avec un code de retour d'échec de l'authentification.

### A.3 Utilitaires pam\_pkcs11

Plusieurs utilitaires de test sont fournis avec pam\_pkcs11 :

- pkcs11\_listcerts : énumère les certificats présents sur la carte et vérifie leur validité ;
- pklogin\_finder : tente d'associer les données contenues dans les certificats de la carte avec un login utilisateur (permet de tester le module PAM sans nécessairement entrer dans le processus de login) ;
- pkcs11\_inspect : affiche le contenu des certificats (utile au cours de la mise au point de la configuration des associations certificats / utilisateurs) ;
- pkcs11\_make\_hash\_link : outil de création des *hash* des certificats et CRLs.

Outils les plus utiles : pkcs11\_listcerts pour vérifier que les certificats sont bien présents sur la carte et pklogin\_finder pour vérifier que l'association avec les utilisateurs du système existe.

Les certificats des autorités de certification de confiance doivent être présents dans /etc/pam\_pkcs11/cacerts, ainsi que les liens symboliques dont les noms correspondent aux hachés des certificats. Ces derniers peuvent être créés avec l'outil make\_hash\_link.sh, fourni avec pam\_pkcs11 (dans /usr/share/...).

L'ajout du certificat de l'autorité de certification de confiance utilisé pour l'authentification des utilisateurs par carte à puce sera réalisée lors de l'installation du poste (le certificat fera partie des éléments de configuration présents sur les supports d'installation).

## B Déchiffrement de données avec OpenSSL

OpenSSL dispose de *moteurs (engines)* permettant de déléguer des opérations cryptographiques à une carte à puce, par le biais d'une bibliothèque PKCS#11. Le moteur PKCS#11 pour OpenSSL s'appuie sur libp11 (couche d'abstraction à une bibliothèque PKCS#11 quelconque).

La commande suivante (sur la ligne de commande de openssl) charge le module :

```
engine -t dynamic -pre SO_PATH :usr/local/lib/engines/engine_pkcs11.so -pre ID pkcs11  
-pre LIST_ADD 1 -pre LOAD -pre MODULE_PATH :usr/local/lib/lib0csCryptoki.so
```

Le moteur à charger au démarrage d'openssl peut également être défini dans son fichier de configuration /etc/ssl/openssl.cnf pour pouvoir invoquer directement l'opération de signature/déchiffrement :

```
openssl_conf          = openssl_def  
  
[openssl_def]  
engines = engine_section  
  
[engine_section]  
pkcs11 = pkcs11_section
```



```
[pkcs11_section]
engine_id = pkcs11
dynamic_path = /usr/local/lib/engines/engine_pkcs11.so
MODULE_PATH = /usr/local/lib/lib0csCryptoki.so
init = 1
PIN = 1234

[req]
distinguished_name = req_distinguished_name

[req_distinguished_name]
```

Alternativement, le PIN peut être passé par une variable d'environnement en remplaçant la ligne PIN=1234 par la suivante :

```
PIN = $ENV :PIN_VAR
```

L'emplacement du fichier de configuration de openssl peut être spécifié en positionnant la variable d'environnement OPENSSL\_CONF (la commande openssl rsautl ne supporte pas l'option -config).

Les commandes suivantes envoient une requête de signature d'une donnée résidant dans /tmp/grumf :

```
rsautl -engine pkcs11 -sign -keyform engine -in /tmp/grumf -inkey slot_0-label_bmorin
rsautl -engine pkcs11 -sign -keyform engine -in /tmp/grumf -inkey label_bmorin
```

Pour voir les différentes manières de désigner une cle, saisir une option syntaxiquement erronée :

```
rsautl -engine pkcs11 -sign -keyform engine -in /tmp/grumf -inkey bmorin
```

## C Enfermement du proxy PKCS#11 dans une cage VServer

Les adaptations du démon en vue de son enfermement dans un vserver s'inspirent des opérations analogues effectuées dans les démons cryptsd et syslog.

En résumé, il existe au moins deux possibilités pour enfermer un processus dans une cage VServer à l'aide des fonctions de la bibliothèque clip-libvserver :

- invoquer la fonction `clip_jailself(...)`<sup>8</sup> en passant entre autres paramètres le numéro de contexte (xid), les capacités, la racine de la cage, etc., ou
- définir les paramètres de configuration de la cage dans un répertoire ad-hoc (généralement présent dans /etc/jails), préparer le contexte (setup) à l'aide de `vsctl`, et faire entrer le processus proxy au sein de la cage via la fonction `clip_enter_context(...)`.

La première solution, qui correspond à celle mise en œuvre dans cryptd, est plus simple que la seconde, qui correspond à celle mise en œuvre dans syslog.

Le proxy PKCS#11 requiert au moins l'accès aux fichiers présents dans le répertoire /var/run/pcscd/ pour communiquer avec la carte. Il est donc nécessaire d'ajouter un montage bind de ce répertoire dans la cage du proxy. Ce montage bind peut être effectué avant d'enfermer le proxy par un appel à `clip_jailself`, mais ceci « pollue » le VFS du socle avec des montages supplémentaires. De ce point de vue, il peut être préférable de démarrer la cage en paramétrant ses montages externes (`fstab.external`) de sorte qu'ils n'apparaissent pas dans le socle. De plus, le processus enfermé dans sa cage selon la solution à base de `clip_jailself` subsiste

---

8. cf. `server.c` de cryptd

dans le contexte au sein duquel il a été créé, ce qui n'est pas nécessaire. On privilégiera donc la seconde solution. On peut s'inspirer pour ce faire du script de création de la cage audit (cf. `core-services` dans les sources, ou `/etc/init.d/clip-audit` sur le poste) au sein de laquelle est enfermé le démon `syslog`.

## D Provisionnement des cartes IAS-ECC

L'utilisation des cartes à puce dans CLIP pour l'authentification des utilisateurs et le déchiffrement des clés des partitions utilisateurs nécessite de pouvoir également charger sur les cartes les certificats X.509 et des clés privées adéquats, et avec les bons labels (`core_auth` pour l'authentification, `core_disk` pour le déchiffrement).

Être capable de faire faire des opérations de signature ou de chiffrement à une carte, via des outils libres (`opensc`) ou propriétaires, n'implique pas *de facto* que l'on puisse y charger ce que l'on souhaite. Néanmoins, nous avons constaté de manière empirique que l'application PKI des cartes IAS-ECC que nous avons eues entre les mains (Gemalto MultiApp, Sagem/Morpho S3, Oberthur) nous permet de réaliser le chargement de certificats X.509 et de clés privées avec les utilitaires fournis avec `opensc` (version patchée par Viktor Tarazov).

Le *provisioning* peut être réalisé de deux manières : soit via l'API PKCS#11, soit via l'API PKCS#15 pour les cartes qui la supportent. Les procédures de provisioning correspondantes sont détaillées, pour les cartes IAS-ECC dans les sections suivantes, dans lesquelles on suppose que les fichiers suivants ont été préalablement générés :

- `core_auth_cert.{pem|der}` : certificat contenant la clé publique d'authentification de l'utilisateur ;
- `core_auth_key.{pem|der}` : la clé privée d'authentification de l'utilisateur, correspondant à la clé publique précédente ;
- `core_disc_cert.{pem|der}` : certificat contenant la clé publique de déchiffrement des clés de chiffrement des partitions de l'utilisateur ;
- `core_disc_key.{pem|der}` : la clé privée de déchiffrement des clés de chiffrement des partitions de l'utilisateur, correspondant à la clé publique précédente ;
- `ca.crt` : certificat de l'autorité racine contenant la clé publique correspondant à la clé privée ayant servi à signer les certificats précédents.

### D.1 Via PKCS#11

L'utilitaire `pkcs11-tool` fourni par `opensc` permet de réaliser les opérations de provisioning, mais il nécessite que les éléments à charger sur la carte soient au format DER.

Les commandes suivantes permettent de charger les éléments sur la carte, et de leur associer les bons labels et attributs :

```
MODULE="--module /usr/lib/opensc-pkcs11.so"

pkcs11-tool $MODULE -l -w clip_auth_cert.der -y cert -a clip_auth
pkcs11-tool $MODULE -l -w clip_auth_key.der -y privkey
pkcs11-tool $MODULE -l -w clip_disk_cert.der -y cert -a clip_disk
pkcs11-tool $MODULE -l -w clip_disk_key.der -y privkey
```

Pour énumérer les objets présents sur une carte :

```
MODULE="--module /usr/lib/opensc-pkcs11.so"

pkcs11-tool $MODULE -l -0
```



Le chemin du MODULE peut varier selon les installations.

## D.2 Via PKCS#15

L'utilitaire `pkcs15-init` fourni par `opensc` permet de réaliser les opérations de provisioning, et supporte en entrée les formats DER et PEM.

Pour les cartes IAS-ECC, il est nécessaire de préciser l'identifiant (AID) de l'application PKI car ce n'est pas l'application par défaut utilisée lorsque qu'aucun AID n'est précisé.

Pour effacer une carte IAS-ECC :

```
AID='e828bd080fd25047656e65726963'

OPTS="--aid $AID"

pkcs15-init $OPTS -E
```

Pour re-crée la structure PKCS#15 sur une carte effacée :

```
AID='e828bd080fd25047656e65726963'

OPTS="--aid $AID"

pkcs15-init $OPTS -C
```

Cette commande peut échouer sur certaines cartes qui réalisent cette opération automatiquement suite à leur effacement avec la commande précédente. Selon la carte, l'option `aid` peut poser problème, il suffit alors d'essayer sans...

Les commandes suivantes permettent de charger les éléments sur la carte, et de leur associer les bons labels et attributs :

```
AID='e828bd080fd25047656e65726963'

OPTS="--aid $AID"

pkcs15-init $OPTS -X clip_auth_cert.pem -l clip_auth
pkcs15-init $OPTS -S clip_auth_key.pem --auth-id 02-u decrypt
pkcs15-init $OPTS -X clip_disk_cert.pem -l clip_disk
pkcs15-init $OPTS -S clip_disk_key.pem --auth-id 02 -u decrypt
```



Le paramètre `auth-id` doit correspondre à l'objet sur la carte correspondant au PIN utilisateur.

L'outil `pkcs15-tool` fourni par `opensc` permet entre autre d'énumérer les objets présents sur une carte IAS-ECC :

```
AID='e828bd080fd25047656e65726963'

pkcs15-tool --aid $AID -D
```

## E Fichier de configuration de la carte à puce

On trouve ci-dessous le fichier smartcards (accessible en lecture et écriture depuis un terminal ADMIN dans le répertoire /etc/admin/conf.d/).

```
# Set to no to entirely disable smartcard support
SMARTCARD_SUPPORT=yes

# List of jails for which PKCS#11 proxying will
# be enabled (none, "core", "rm_h", "rm_b", or
# any combination thereof)
PKCS11_JAILS=core

# List of jails for which OpenPGP smartcards will
# be supported (none, "rm_h", "rm_b", "rm_h rm_b")
OPENPGP_JAILS=

#Label of the certificate used for authentication
#by pam_pkcs11
PKCS11_AUTH_LABEL=clip_auth

#Label of the certificate used for homedir encryption
#by pam_pkcs11
PKCS11_DISK_LABEL=clip_disk
```

## F Configuration du proxy PKCS#11

### F.1 Configuration générale du proxy

A titre d'exemple, voici le contenu par défaut du fichier pkcs11proxycd\_core.conf. Pour les cages RMH et RMB, les fichiers sont sensiblement identiques. Les paramètres importants choisis sont explicités plus bas.

```
netplex {
  controller {
    socket_directory = "/var/lib/p11proxy_core/tmp";
    max_level = "debug";    (* Log level *)
    logging {
      type = "syslog";      (* Log to syslog *)
      facility = "daemon";
    };
    logging {
      type = "syslog";      (* Log to syslog *)
      facility = "daemon";
      subchannel = "filter";
    };
  };
  service {
    name = "PKCS#11 Filtering Proxy - Core";
    (* Do NOT change conn_limit, this would be a serious SECURITY ISSUE *)
```

```
conn_limit = 1;
protocol {
    (* This section creates the socket *)
    name = "rpc_pkcs11";
    address {

        type = "local";
        path = "/var/run/p11proxy/socket";
    };
};
processor {
    (* This section specifies how to process data of the socket *)
    type = "rpc_pkcs11";
    filter_config="/etc/pkcs11proxyd/filter_core.conf";
    use_ssl = false;
    cafile = "/unused/ca.crt";
    certfile = "/unused/server.crt";
    certkey = "/unused/server.key";
};
workload_manager {
    type = "dynamic";
    max_jobs_per_thread = 1; (* Everything else is senseless *)
    min_free_jobs_capacity = 1;
    max_free_jobs_capacity = 1;
    max_threads = 100;
};
}
```

Dans ce fichier, on retrouve entre autres :

- une section controller, qui permet d'agir sur l'emplacement de la collecte d'informations dans les journaux. On voit dans ce fichier que les informations générées par le proxy apparaîtront dans `daemon.log`.
- une section service, qui paramètre le service fourni par le proxy. Parmi les paramètres, on note que `conn_limit` est fixé à 1 et ne doit pas être modifié, et il en est de même pour `max_jobs_per_thread` définie dans la section `max_jobs_per_thread`. En effet, le proxy est architecturé selon un modèle *fork-exec* de manière à fournir une isolation entre les différentes applications clientes. Le chemin de montage de la socket de communication est dans la variable `path`.
- une section processor, on peut définir le fichier de configuration du filtre chargé lors de la création de l'instance, et configurer un éventuel lien SSL entre parties cliente et serveur du proxy (ceci n'est pas utilisé dans le cas de CLIP).

## F.2 Configuration du filtrage

On notera que dans les fichiers de configuration suivants, les symboles compris entre `(*` et `*)` sont considérés comme des commentaires.

A l'exception des variables debug (variant de 0 à 2 pour augmenter la précision des messages journalisés) et `log_subchannel` (pour configurer le canal de journalisation) la syntaxe adoptée pour la configuration du filtrage est la suivante. Chaque variable contient une liste (entre crochets) formée d'une paire (entre parenthèses) constituée d'une expression régulière définissant des chaînes de caractères et d'une valeur à prendre.

La chaîne de caractères correspond à celle passée en paramètres par l'application cliente. Par défaut, elle sera vide. Si on compte utiliser plusieurs applications clientes nécessitant plusieurs profils de filtrage différents, il faudra choisir deux chaînes distinctes. On portera attention à ce qu'en cas de duplication d'une ligne ou d'un élément dans une liste, seul la dernière occurrence est prise en compte.

Pour illustrer ces propos, on détaille l'exemple suivant :

```
modules = [("", "/usr/lib/lib1.so"), ("toto", "/usr/lib/lib2.so")]
allowed_labels= [(".", "test.*")]
```

La première ligne va provoquer le chargement de la bibliothèque lib1.so (resp. lib2.so) quand l'application cliente passe en paramètre la chaîne de caractères vide (resp. "toto"). La seconde ligne va, quelle que soit la chaîne de caractères envoyée par l'application cliente, rendre invisible tout objet dont le label ne commence pas par test.

### F.2.1 Fichier de configuration pour le filtre du socle

Ci-dessous le contenu du fichier filter\_core.conf.

```
debug = 0
log_subchannel = filter
modules = [("", "/usr/lib/opensc-pkcs11.so")]
forbidden_mechanisms = [(".", [])]
allowed_labels = [(".", ["clip_disk", "clip_auth"])]
allowed_ids = [(".", [".*"])]
forbidden_functions = [(".", [])]
enforce_ro_sessions = [(".", no)]
forbid_admin_operations = [(".", no)]
(* Fixing PKCS#11 with patchset 1 *)
filter_actions_post = [ (".*",
    [
        (***** This is optional :key usage segregation
            *****)
        (* (C_Initialize, do_seggregate_usage), *)
        (... autres actions de filtrage ...)
        (C_DeriveKey, sanitize_creation_templates_patch), (
            C_UnwrapKey, sanitize_creation_templates_patch)
    ]
    )
]
```

### F.3 Fichier de configuration pour le filtre du socle

Ci-dessous le contenu du fichier filter\_rm\_b.conf.

```
debug = 0
log_subchannel = filter
modules = [("", "/usr/lib/opensc-pkcs11.so")]
forbidden_mechanisms = [(".", [])]
allowed_labels = [(".", ["rm_b.*"])]
```

```
allowed_ids = [(".", "*"), [(".", "*")]]
forbidden_functions = [(".", "*"), [ C_InitToken,
(...autres fonctions ...)
    C_SeedRandom,
]]
forbid_admin_operations = [(".", "*"), yes]]
enforce_ro_sessions = [(".", "*"), yes]]
remove_padding_oracles = [(".", "*"), [wrap, unwrap, encrypt]]]
```

## G Attributs conflictuels

Dans la version actuellement implémentée du proxy, les combinaisons d'attributs considérées comme conflictuelles sont les suivantes.

- cKA\_WRAP = cK\_TRUE et cKA\_DECRYPT = cK\_TRUE ;
- cKA\_UNWRAP = cK\_TRUE et cKA\_ENCRYPT = cK\_TRUE ;
- cKA\_SENSITIVE = cK\_FALSE et cKA\_EXTRACTABLE = cK\_FALSE ;
- cKA\_SENSITIVE = cK\_TRUE et cKA\_ALWAYS\_SENSITIVE = cK\_FALSE ;
- cKA\_EXTRACTABLE = cK\_FALSE et cKA\_NEVER\_EXTRACTABLE = cK\_FALSE ;
- cKA\_WRAP = cK\_TRUE et cKA\_SENSITIVE = cK\_FALSE ;
- cKA\_WRAP = cK\_TRUE et cKA\_ALWAYS\_SENSITIVE = cK\_FALSE ;

De plus, lorsque la séparation des usages des clés est demandée, les couples suivants sont ajoutés aux attributs conflictuels :

- cKA\_ENCRYPT = cK\_TRUE et cKA\_SIGN = cK\_TRUE ;
- cKA\_ENCRYPT = cK\_TRUE et cKA\_SIGN\_RECOVER = cK\_TRUE ;
- cKA\_DECRYPT = cK\_TRUE et cKA\_VERIFY = cK\_TRUE ;
- cKA\_DECRYPT = cK\_TRUE et cKA\_VERIFY\_RECOVER = cK\_TRUE ;
- cKA\_ENCRYPT = cK\_TRUE et cKA\_VERIFY = cK\_TRUE ;
- cKA\_ENCRYPT = cK\_TRUE et cKA\_VERIFY\_RECOVER = cK\_TRUE ;
- cKA\_DECRYPT = cK\_TRUE et cKA\_SIGN = cK\_TRUE ;
- cKA\_DECRYPT = cK\_TRUE et cKA\_SIGN\_RECOVER = cK\_TRUE.

## Références

[1] ANSSI. Fonctions d'authentification clip. Documentation CLIP.