Buy it, use it, break it ... fix it : `Caml Crush`, un proxy PKCS#11 filtrant

Ryad Benadjila, Thomas Calderon et Marion Daubignard
ryad.benadjila@ssi.gouv.fr
thomas.calderon@ssi.gouv.fr
marion.daubignard@ssi.gouv.fr

ANSSI

Résumé Les cartes à puce et les ressources cryptographiques dédiées sont des outils précieux pour renforcer la sécurité des systèmes d'information. Ces périphériques peuvent êtres vus comme un cœur de confiance dans lequel des opérations cryptographiques vont êtres réalisées. L'interface de programmation PKCS#11 est un standard répandu pour dialoguer avec de telles ressources : il est devenu *de facto* le standard préféré de l'industrie. Cependant, des publications récentes ont mis en évidence plusieurs attaques logiques et cryptographiques exploitant des faiblesses de l'interface PKCS#11 pour porter atteinte à la confidentialité ou à l'intégrité des clés cryptographiques stockées dans les ressources concernées. Nous présentons dans cet article une architecture client/serveur ainsi qu'un moteur de filtrage configurable et extensible. La mise en œuvre d'un tel outil pour analyser des commandes PKCS#11 avant de les transmettre à une ressource cryptographique permet de réduire l'exposition de cette ressource à des attaques exploitant des faiblesses de son implémentation de PKCS#11.

Introduction

Les cartes à puce, les jetons d'authentification, les *HSM* (*Hardware Security Modules*) sont des exemples de composants de sécurité dédiés à la protection des valeurs d'objets cryptographiques sensibles, et à leur manipulation sécurisée. Toute notion de confidentialité et d'intégrité est en générale perdue dès la compromission de ces objets, on s'accorde donc facilement sur la nécessité de les gérer correctement. Cependant, en plus de soigner l'implémentation de la cryptographie et des protocoles utilisés, il est fondamental d'examiner l'interface de programmation qui expose les opérations disponibles sur les objets cryptographiques, ce que l'on nomme une **API de sécurité**. Moins populaire que d'autres axes d'attaque, l'exploitation des vulnérabilités portées par les API de sécurité se révèle souvent redoutable : des fuites d'information sur les valeurs des clés utilisées sont monnaie courante. La sécurisation des API de sécurité constitue donc un chantier de recherche incontournable.

Un effort de standardisation substantiel a été fourni par RSA Laboratories, qui a abouti à la définition de l'API *Cryptoki* dans le standard PKCS#11. Cette API est la plus fréquemment utilisée par les systèmes disponibles aujourd'hui. Il paraît donc particulièrement pertinent de s'intéresser à la sécurité offerte par les implémentations dites compatibles avec ce standard. Dans les dix dernières années, les recherches à ce sujet ont démontré l'existence d'attaques critiques laissées ouvertes par le respect des seules exigences du standard, et proposé un ensemble de contre-mesures efficaces pour les compléter. Bien que l'on sache théoriquement pallier aux faiblesses de *Cryptoki*, pour diverses raisons, la plupart des implémentations présentes sur des composants actuellement disponibles dans le commerce ne prennent pas en compte les modifications nécessaires à une sécurisation de l'API qu'ils exposent. De plus, tous les composants déjà présents en milieu de production sont victimes des problèmes découverts. N'ayant pas les moyens de corriger les vulnérabilités présentes dans leurs composants, les utilisateurs sont laissés sans solution viable.

Notre approche répond à ce besoin : elle offre la possibilité de durcir une API de sécurité conforme au standard PKCS#11 en ajustant les restrictions apportées aux contextes d'utilisation de l'API. Dans cet article, nous présentons `Caml Crush`, solution logicielle de proxification de ressources cryptographiques conformes au standard PKCS#11. Fruit d'une architecture sécurisée construite autour d'un moteur de filtrage, le *proxy* est pensé pour être placé en coupure entre les applicatifs et le *middleware* PKCS#11 fourni dans les ressources. Conçu pour être configurable et extensible par ses utilisateurs, `Caml Crush` permet de limiter l'exposition de la ressource cryptographique grâce à un moteur de règles extensible. En plus de la correction de failles connues, `Caml Crush` offre un certain nombre de fonctionnalités additionnelles, dont des propriétés d'isolation entre utilisateurs différents d'une même ressource, l'amélioration de la conformité au standard ou la possibilité de corriger une vulnérabilité découverte *a posteriori*. Nous soulignons que l'intégralité du code source est publiée sur `github` [1].

Le présent article s'articule autour de six parties. La section 1 introduit les éléments principaux de l'interface PKCS#11, permettant d'introduire les motivations de l'approche proposée dans la section suivante 2. La section 3 détaille l'architecture et le fonctionnement du *proxy*, à l'exception du module de filtrage et ses règles extensibles qui sont présentés dans la section 4. Par la suite, la section 5 décrit les avantages de l'approche, tandis que son impact sur les performances et ses limitations sont détaillés en section 6.

1 Description des concepts liés à PKCS#11

1.1 Généralités sur l'interface PKCS#11

Le rôle du standard PKCS#11 est d'offrir une interface générique permettant de dialoguer avec des ressources cryptographiques. En plus d'abstraire les détails d'implémentation variant entre les différents matériels, cette approche permet de faire fonctionner une application compatible avec tout matériel qui respecte le standard.

Le standard PKCS#11 définit une interface de programmation (API) nommée *Cryptoki* (pour *Cryptographic token interface*), dont l'objectif est de permettre à une ou plusieurs applications de réaliser des opérations cryptographiques. Pour ce faire, *Cryptoki* fournit une **vue logique** des périphériques et ressources cryptographiques impliqués. Un conteneur, appelé **slot**, permet de recevoir un **token**. C'est ce dernier qui offre le service cryptographique. Dans l'exemple des cartes à puce, le *slot* correspond au lecteur physique et la carte à puce au *token*.

Le *token* désigne la ressource cryptographique. Néanmoins, son rôle n'est pas exclusivement d'effectuer des opérations cryptographiques. En réalité, il s'agit aussi d'un conteneur qui permet le stockage et l'utilisation sécurisés d'éléments sensibles. De la confidentialité et/ou de l'intégrité de ces éléments dépendent les garanties fournies par la cryptographie. Ces éléments sont abstraits en tant qu'**objets**, qu'on ne manipule pas directement, mais à travers des références nommées **handles**. Ainsi, la raison pour laquelle on utilise un *token* est la confiance en le fait que les objets qu'il contient ne sont pas facilement récupérables et/ou altérables. En particulier, le *token* ne doit pas être duplicable/clonable facilement.

Lors de la connexion d'une application au *token*, une **session** est créée. Quand on veut faire référence à un objet stocké dans le *token* ou à une session, on utilise un *handle* les référençant. *Cryptoki* permet de faire les opérations usuelles de chiffrement et déchiffrement sur des données, mais aussi sur des clés, et dans ce dernier cas on parle d'encapsulation (ou *Wrap*) et de décapsulation (ou *Unwrap*). Les opérations offertes par *Cryptoki* faisant intervenir des clés présentes dans le *token* prennent en argument des *handles* pointant vers ces clés. Elles peuvent avoir pour résultat la sortie de données ou la création d'un *handle*, par exemple dans le cas d'un import de clé suite à une décapsulation. En effet, quand un *token* reçoit une clé encapsulée, il la décapsule (*i.e.* il la déchiffre) et ne l'exporte pas en clair, mais la stocke et renvoie juste un *handle* qui permet de l'utiliser.

La spécification de l'API *Cryptoki* fournit une liste de fonctions à l'aide d'en-têtes en langage C. Les fournisseurs de ressources cryptographiques choisissent généralement la solution consistant à fournir un *middleware* implémenté sous la forme d'une bibliothèque partagée, `DLL` sous Windows et `.so` sous Linux. La bibliothèque exporte les fonctions de l'interface, et porte aussi le nom de *Cryptoki* par abus de langage.

1.2 De l'accessibilité des objets

Selon le standard, un *token* peut contenir trois catégories différentes d'objets : les données, les certificats et les clés. Ces objets peuvent différer par leurs propriétés, telles que leur visibilité, leur durée de vie, la possibilité de les extraire du *token* ou encore de les modifier. Ces caractéristiques sont définies grâce à l'utilisation d'**attributs**. Les différentes classes d'objets utilisent des types d'attributs communs (*public*, *private*, etc.) mais il existe des attributs spécifiques à certains objets. Par exemple une clé RSA utilisera des attributs pour définir son module et ses exposants, attributs qui n'auront pas de sens pour des clés symétriques par exemple. D'une manière générale, on retiendra que quel que soit le type d'une clé cryptographique, sa valeur lui est associée via un ou plusieurs attributs.

La notion de **visibilité** des objets est fondamentale au sein de *Cryptoki*. En effet, il n'est pas nécessaire pour une application de s'authentifier pour manipuler des objets de type *public*. En revanche, les objets de type *private* sur un *token* ne seront utilisables qu'une fois l'utilisateur de la ressource authentifié, traditionnellement sous la forme d'un code PIN. Les objets dits **token objects** sont persistants et visibles de toutes les applications clientes. Ceci n'est pas le cas des **session objects** qui ne sont pas visibles d'une application cliente à l'autre, et n'ont plus d'existence lorsque la **session est terminée**.

Comme précédemment mentionné, la valeur d'une clé est un attribut que l'on ne souhaite pas voir circuler sans restriction en clair à l'extérieur du *token*, même sur demande d'un utilisateur authentifié. Afin de réaliser cela, PKCS#11 introduit des attributs particuliers : les attributs *sensitive* et *extractable*. Ces attributs prennent des valeurs booléennes. L'attribut *sensitive* permet de spécifier qu'un objet est jugé sensible, l'attribut *extractable* définit son aptitude à circuler en dehors du token même chiffré. Selon le standard, un objet possédant l'attribut *sensitive* actif ne doit pas circuler en clair à l'extérieur du *token*, tandis que s'il possède l'attribut *extractable*, on ne doit pas pouvoir extraire l'objet du *token*, même encapsulé par une autre clé. De manière logique, la norme précise qu'une

clé portant *extractable* ne peut avoir *sensitive* positionné à vrai. Nous utiliserons dans la suite de l'article les raccourcis de langage suivants :

- Nous parlerons de clé extractible pour dire que son attribut *extractable* est positionné à vrai.
- Nous dirons qu'un objet est sensible pour signifier que son attribut *sensitive* est positionné à vrai.

L'API *Cryptoki* définit la fonction `C_GetAttributeValue` qui permet de lire les valeurs d'attributs passés en paramètres, ainsi que son pendant `C_SetAttributeValue` qui permet de les définir. Une première contrainte issue de la définition des attributs *extractable* et *sensitive* consiste à ne pas répondre à de telles requêtes de lecture directe de valeur sur des clés ayant l'attribut *sensitive* vrai et *extractable* faux. De manière similaire, pour une clé pour laquelle *extractable* est faux, on ne doit pas répondre à une requête d'encapsulation de la clé. Enfin, ces attributs seraient inutiles s'ils pouvaient être eux-mêmes modifiés pour rendre *sensitive* faux et *extractable* vrai : il faut empêcher de telles modifications par `C_SetAttributeValue`. De telles restrictions sont nécessaires mais ne suffisent pas du tout à assurer qu'une clé sensible ou non-extractible ne soit pas révélée, comme nous allons le voir ci-après.

1.3 Exemples de vulnérabilités classiques sur PKCS#11

Nous avons mentionné l'existence de fonctions d'encapsulation et de décapsulation des clés stockées à l'intérieur d'un *token*. Ces fonctionnalités sont entre autres nécessaires pour partager des valeurs de clés sur un canal public, mais entraînent de nombreuses vulnérabilités, majoritairement dues à des confusions de types d'usage des clés. En effet, il n'existe pas de différence réelle entre un mécanisme d'encapsulation/décapsulation et un mécanisme de chiffrement/déchiffrement au niveau de l'API. En d'autres termes, on ne peut pas décider qu'un mécanisme donné ne sera utilisé que pour chiffrer et l'autre que pour encapsuler. On se retrouve donc avec deux interfaces possibles par opération cryptographique : d'un côté on peut chiffrer ou encapsuler, de l'autre on peut déchiffrer ou décapsuler. On rappelle que déchiffrer ressort un texte clair et que décapsuler crée une clé dans le *token*.

Dans la suite, on notera $\{m\}_k$ indifféremment le chiffrement ou l'encapsulation de m avec k. Illustrons les conséquences possibles d'une confusion avec l'exemple suivant. Supposons qu'une clé k soit sensible et extractible. Elle peut alors être encapsulée pour être envoyée à un autre *token*. Soit k' une autre clé présente sur le token. Lors d'une requête d'encapsulation de k avec une autre clé k', le *token* va répondre la valeur $\{k\}_{k'}$. On

commence par noter que si k' est non-sensible, il n'y a plus qu'à demander la valeur de k' pour pouvoir récupérer k. Supposons alors k' sensible mais utilisable pour déchiffrement. Si l'on soumet une requête au *token* pour un déchiffrement par k', l'opération va se dérouler avec succès. Cette opération va donc ressortir k en clair puisque le même mécanisme a été utilisé, au lieu de créer un *handle* vers k comme l'aurait fait une décapsulation. Soulignons que le *token* ne peut pas discriminer si $\{k\}_{k'}$ est le résultat d'un chiffrement ou d'une encapsulation : d'une part, c'est juste un *blob*, d'autre part les deux sont possibles.

Si les mécanismes ne sont pas liés à un usage dans *Cryptoki*, on peut cependant lier les clés à leur usage. Le standard définit ainsi des attributs *wrap*, *unwrap*, *encrypt* et *decrypt* qui autorisent les opérations correspondantes. La confusion présentée ci-dessus n'est alors possible que si la clé k peut à la fois être utilisée pour encapsuler et déchiffrer. Les attributs *wrap* et *decrypt* – qui ont donné leur nom à cette attaque – sont donc conflictuels et ne devraient pas être activés ensemble. Nous avons aussi vu pourquoi *sensitive* et *extractable* sont conflictuels pour certaines valeurs (une clé non-extractible est forcément sensible).

De nombreuses variantes de ces principes d'attaque peuvent être imaginées. Autant les restrictions à imposer pour ne pas répondre à des requêtes directes d'attributs sont relativement intuitives et (presque) clairement explicites dans le standard, autant la séparation des usages des clés n'avait clairement pas été envisagée comme nécessaire. Preuve en est qu'il a fallu attendre le travail de Clulow [20] qui a mis en lumière ces problèmes, exhibé de nombreuses autres subtilités menant à des attaques de plus en plus élaborées ainsi qu'un intérêt accru vis-à-vis de ce standard.

2 Pertinence et bénéfices de l'usage d'un *proxy* filtrant

Dans cette section nous expliquons les solutions existantes pour contrer les attaques connues sur l'API PKCS#11. Nous discutons pourquoi ces solutions ne sont pas satisfaisantes pour les utilisateurs finaux. Enfin, nous exposons la viabilité du *proxy* PKCS#11 filtrant pour la mise en place de contremesures.

2.1 Obtention d'une API sécurisée

L'interface PKCS#11 est réputée vulnérable à certaines classes d'attaques logiques et cryptographiques. Un travail de modélisation de l'interface et de sa sécurité a été réalisé par Delaune *et al.* dans [21] et

Bortolozzo *et al.* dans [19]. Ce dernier article introduit l'outil *Tookan* [14] qui permet de rechercher et découvrir automatiquement des chemins d'attaques logiques par la résolution de problèmes `SAT`. Suite à ces travaux, de nouvelles variantes de ces attaques ont pu être découvertes sur des *tokens*. En plus de fournir un outil de détection, [19] propose une série de correctifs pour l'interface PKCS#11 et leur implémentation : *CryptokiX* contenant des « fix » appliqués au *token* logiciel *OpenCryptoki* [9].

Tookan et *CryptokiX* sont essentiellement pertinents pour les **fabricants de *tokens***, le premier mettant à l'épreuve les matériels existants et le second fournissant une preuve de concept des contremesures proposées. Un utilisateur soucieux de ces problématiques de sécurité peut bien entendu se servir de *Tookan* pour chercher des vulnérabilités sur son *token*. En revanche, il lui est impossible de mettre en place des correctifs de sa propre initiative. C'est au fabricant de prendre en charge les modifications à apporter sur son produit. Les fabricants mettent en effet à disposition très peu de code source concernant leurs *tokens*, l'utilisateur ayant au mieux des **binaires** mis à jour. Par ailleurs, mettre à jour un *token* est plus ou moins difficile en fonction du type de ressource cryptographique considéré. Par exemple, la mise à jour d'un *token* de type *HSM* est plus facile à effectuer que celle des cartes à puce. De plus, il est peu probable qu'un fabricant fournisse un correctif pour du matériel considéré comme obsolète même si celui-ci est toujours utilisé en production. Ainsi l'utilisateur est dépendant la bonne volonté du fabricant et risque de continuer à utiliser du matériel potentiellement vulnérable.

Nous avons donc souhaité concevoir une solution adaptée aux utilisateurs de *tokens* qui ne disposent pas de correctifs de la part du fabricant. L'approche privilégiée pour tenter de résoudre les problèmes mentionnés est la mise en coupure d'un serveur mandataire, ou *proxy*, entre les applications et le *middleware*. L'avantage de cette solution est d'offrir un point central capable d'appliquer les contremesures de manière dynamique. De fait, il est possible de filtrer les séquences d'appels et les résultats retournés par la ressource cryptographique.

Une autre solution serait de développer un *middleware* de remplacement pour les *tokens*. La complexité d'une telle solution est élevée car il est nécessaire de réaliser une implémentation par *token*. De plus, la gestion bas-niveau des ressources cryptographiques est très souvent propriétaire. Une dernière alternative serait d'empiler une couche PKCS#11 complète au dessus du *middleware* propriétaire. En plus d'un effort de réimplémentation du standard, maintenir un état cohérent entre les deux *middlewares*

serait également nécessaire. Notre approche est donc plus légère et plus portable que les deux alternatives susmentionnées.

Caml Crush remplit son rôle principal : la mise en place de contremesures aux faiblesses de PKCS#11. En effet, le niveau de sécurité atteint est équivalent à celui de *CryptokiX*. Nous tenons néanmoins à souligner que notre solution fait l'hypothèse d'un attaquant pour qui il n'est pas possible de contourner le *proxy* en communiquant directement avec le *middleware* ou le périphérique matériel. Cet aspect sera discuté dans la section 6.2.

2.2 Autres fonctionnalités offertes

L'application des correctifs usuels de l'API PKCS#11 n'est pas suffisante pour une utilisation sécurisée des *tokens*. L'utilisation de Caml Crush permet d'obtenir des propriétés de sécurité supplémentaires qui sont détaillées dans les paragraphes suivants.

Multiples politiques de filtrage. Caml Crush peut être instancié plusieurs fois avec des politiques de filtrage différentes. Ainsi, chaque instance peut être dédiée à un usage précis pour un même *token*. Les applications clientes peuvent être regroupées en fonction de leur niveau de criticité. La figure 1 représente deux applications de niveaux de sensibilité différents qui se partagent l'accès à un même *token*. Les politiques de filtrage peuvent être adaptées à chacune des applications.

Gestion des mécanismes cryptographiques. *Cryptoki* permet l'utilisation de mécanismes cryptographiques faibles. Par exemple, si un *token* supporte le chiffrement DES, rien ne peut empêcher une application de s'en servir. Des règles de filtrage permettent de restreindre les mécanismes cryptographiques autorisés.

Amélioration du modèle d'utilisateur. Le standard PKCS#11 définit deux catégories d'utilisateurs, un administrateur (le *Security Officer*, ou *SO*) et l'utilisateur du *token*. Une même connexion logique peut être utilisée pour réaliser des opérations *SO*, puis des opérations classiques. Pour éviter cela, des instances dédiées à chaque rôle peuvent être mises en place avec des politiques de filtrage distinctes.

Restrictions sur les objets. Le standard PKCS#11 n'offre pas de mécanisme de segmentation des objets sur le *token*. Dès lors qu'un utilisateur est authentifié, il accède à tous les objets. En pratique, une ressource

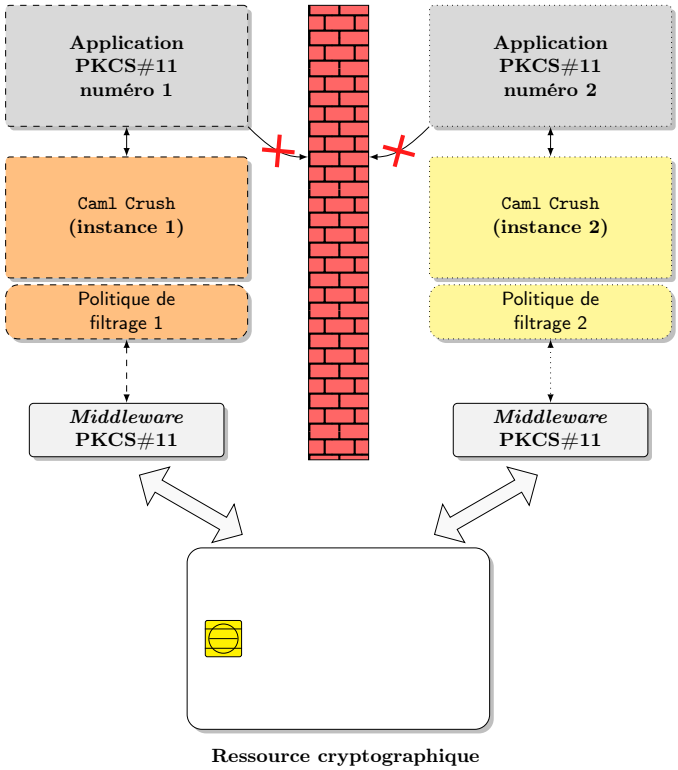

Figure 1. Politiques de filtrage appliquées via diverses instances de `Caml Crush`

cryptographique accessible en réseau va souvent être partagée par des applications différentes. *Cryptoki* ne prévoit pas de moyen de restreindre la visibilité des objets en fonction des applications qui accèdent au *token*. L'utilisation de multiples instances de `Caml Crush` offre une solution simple pour réaliser une telle segmentation : le moteur de filtrage peut être configuré pour n'exposer que les objets correspondants à la politique de filtrage de l'instance.

Isolation du *middleware*. La complexité du standard PKCS#11 entraîne que la qualité du code d'un *middleware* peut varier d'un constructeur à un autre. Déréférencement de pointeurs nuls et dépassement de tampons sont deux exemples de pièges qui peuvent conduire à un *middleware* instable, voire exploitable. Il est paradoxal d'augmenter la surface d'attaque de son système lors de l'utilisation d'une ressource cryptogra-

phique. Un des avantages de l'architecture de `Caml Crush` est de pouvoir isoler aisément le *middleware* d'origine.

Amélioration de conformité. `Caml Crush` permet la rectification de certains écarts de comportement d'un *token* par rapport au standard. Dans la pratique, il s'avère utile d'ajouter des vérifications pour rectifier les défauts de l'implémentation des fonctions critiques.

3 Architecture de la solution

Dans cette section, nous allons décrire l'architecture de `Caml Crush` ainsi que les motivations et choix ayant amené cette architecture.

3.1 Projets de proxification PKCS#11 existant

Comme nous l'avons vu en section 2, l'objectif de `Caml Crush` est de se mettre en coupure entre un client PKCS#11 et un *middleware* qui gère directement une ressource cryptographique. Il existe des projets qui implémentent un *proxy* PKCS#11, dont *GNOME Keyring* [5] et *pkcs11-proxy* [10]. Néanmoins, ces *proxy* mettent l'accent sur le partage réseau (ou local via des sockets *Unix*) d'une même ressource cryptographique, ainsi que la mise en cache des objets de la ressource permettant ainsi un gain en performances. L'optique de ces projets n'est donc pas particulièrement la sécurité : ils posent quelques problèmes vis-à-vis du standard PKCS#11 et des attaques possibles, comme cela est illustré en 3.2.

Nous avons donc décidé de partir d'une nouvelle architecture conçue avec les notions de **défense en profondeur** et de **modularité** en tête. Afin de réaliser cela, nous avons fait certains choix, décrits en 3.2, qui ont été dimensionnant pour l'architecture dont le détail est décrit en 3.3.

3.2 Choix dimensionnant

Modèle Fork. L'accès concurrent à la ressource cryptographique par de multiples applications est un cas d'utilisation classique de PKCS#11. Les solutions de proxification existantes [5,10] listées en 3.1 utilisent un modèle à base de *threads* au niveau du serveur afin de gérer les différents clients qui soumettent des requêtes PKCS#11. Cela signifie donc qu'une unique instance du *middleware* PKCS#11 est chargée dans l'espace mémoire commun partagé par les *threads* qui gèrent les clients. Ainsi, les *handles* de session et d'objets d'un *thread* client sont accessibles par les

autres clients : il serait possible à un client malveillant non authentifié sur la ressource d'« utiliser » des *handles* sur des objets privés ouverts par un autre client authentifié.

Évidemment, une solution serait d'implémenter au niveau du serveur un module de contrôle d'accès filtrant les requêtes des clients pour vérifier qu'un client donné n'essaie pas d'utiliser les *handles* d'un autre client. Cette solution possède néanmoins plusieurs inconvénients :

- Elle nécessite d'implémenter une surcouche de logique qui utilise de la mémoire (l'association entre *threads* et *handles*).
- L'exploitation d'une vulnérabilité détournant le flot d'exécution – par exemple de type *buffer overflow* – permettrait à un client d'outrepasser cette contremesure via l'accès à tout l'espace mémoire.

Au delà de la possible perte de performances qu'elle induirait, la surcouche ajoutée ressemble fortement à ce que qu'un *middleware* PKCS#11 fournit comme service lorsque plusieurs clients discutent avec une même ressource. Les *middlewares* utilisent le **cloisonnement classique** des systèmes d'exploitation entre processus, à savoir l'utilisation de mémoire virtuelle, afin d'isoler les clients. Un serveur utilisant des *fork* pour gérer ses clients semble donc mieux convenir au modèle PKCS#11. Comme nous le verrons en 5, ce modèle de serveur a d'autres avantages, dont la gestion du *multi-middlewares*. Il est par ailleurs possible d'appliquer les principes de défense en profondeur via du cloisonnement plus poussé des processus qui gèrent chaque client, grâce à du *sandboxing* et autre descente de privilèges finement configurée.

Utilisation du langage OCaml. Nous avons choisi de développer les éléments critiques de l'architecture du proxy en OCaml [12]. Celui-ci offre un **typage fort** des données et soulage le développeur des contraintes de gestion mémoire. De plus, de par sa nature fonctionnelle, ce langage est parfaitement taillé pour écrire de manière assez concise un moteur de filtrage, et en décrire les règles telles que celles présentées en 4. Enfin, le langage OCaml autorise l'appel de fonctions C externes, ce qui s'avère indispensable pour interagir avec les *middlewares* PKCS#11.

Utilisation des Sun RPC et du XDR. L'objectif du *proxy* est d'exposer via une socket (TCP ou *Unix*) l'API PKCS#11 afin qu'un client puisse discuter avec le *middleware* au travers du réseau. L'utilisation de RPC (*Remote Procedure Call*) est donc une solution adéquate. Afin de nous assurer un maximum de compatibilité en terme de code, nous avons

décidé d'utiliser le format *Sun RPC* [15]. Celui-ci utilise les spécifications d'interface XDR [16] (*eXternal Data Representation*) pour **sérialiser les données** au travers de la couche de transport. L'avantage du format XDR est sa compatibilité avec plusieurs plateformes (Linux, BSD, Mac OS, Windows etc.). Un second avantage, particulièrement utile dans notre cas, est le fait qu'il ait été porté sous OCaml. Enfin, le recul vis-à-vis de ce standard datant de plusieurs dizaines d'années joue en faveur de la robustesse des implémentations existantes.

Utilisation de générateurs de code. Le standard PKCS#11 compte 68 fonctions avec des prototypes de natures assez différentes. D'une part, la proxification impose d'exporter sur le réseau, via les RPC, ces fonctions. D'autre part, l'utilisation du langage OCaml nécessite l'implémentation d'une couche d'interfaçage avec du code natif écrit en C, afin de « discuter » directement avec le vrai *middleware* qui gère la ressource cryptographique. A la fois XDR (pour les RPC) et CamlIDL (pour l'interface OCaml/C) fournissent un langage descriptif en pseudo-C permettant de générer automatiquement des « *stubs* » :

- `rpcgen` ou `ocamlrpcgen` génèrent du code **client** et **serveur** en C ou en OCaml depuis le fichier XDR. Le but étant de sérialiser des structures de données définies dans le standard (proches des *buffers* et structures en C).
- CamlIDL génère du code C et OCaml, ainsi que des fichiers d'interface OCaml (`.mli`), permettant de **transformer** les types natifs OCaml en types natifs du C et réciproquement. L'avantage est une gestion automatisée des allocations et désallocations en mémoire. CamlIDL utilise une variante de l'IDL (*Interface Definition Language*).

Les avantages d'utiliser des outils de génération automatisée de code sont multiples. D'une part, les fichiers de description d'interface sont simples et de petite taille (au regard du code généré), donc faciles à maintenir et à étendre lors de l'ajout de nouvelles fonctions. D'autre part, le code généré possède des garanties fortes de robustesse, notamment vis-à-vis de la gestion mémoire. Enfin, il est en théorie possible de réutiliser ces mêmes fichiers d'interface pour générer des *stubs* pour d'autres langages qui gèrent les standards IDL et XDR : la réécriture des composants d'interface – vers le *middleware* ou pour la couche réseau – du *proxy* dans d'autres langages qu'OCaml en est simplifiée. Nous verrons néanmoins que cela concerne plus le XDR que l'IDL : 3.3 montre en effet que les *stubs* IDL générés sont

assez profondément modifiés via des patchs conséquents qui ne sont pas forcément évidents à porter.

Conception modulaire. `Caml Crush` a été développé avec la modularité en tête. Comme nous le verrons en 3.3, le projet (*proxy* et bibliothèque cliente) n'est pas monolithique : il a été découpé en modules qui communiquent via des interfaces PKCS#11, avec une API bien définie et/ou de l'interfaçage standardisé (XDR, IDL). Il est donc possible de remplacer chaque module par un autre fonctionnellement équivalent, si tant est qu'il se conforme à ces interfaces. C'est notamment le cas du filtre décrit en 4 doté d'un *backend* et d'un *frontend* qui l'isolent du reste du serveur.

3.3 Architecture détaillée

`Caml Crush` peut être décomposé en plusieurs sous-ensembles, représentés sur la figure 2. Le premier élément de l'architecture est l'interface PKCS#11 en OCaml, ou *binding* ① , qui sert à charger le *middleware* et communiquer avec une ressource cryptographique. Le serveur *proxy* est lui-même composé de plusieurs éléments. Il inclut la logique nécessaire au traitement des connexions *RPC* ③ ④ , le moteur de filtrage ② et s'appuie sur le *binding* pour effectuer des appels PKCS#11 à la ressource cryptographique. Le dernier élément est la bibliothèque cliente qui expose une interface PKCS#11 pour les applications ⑥ et communique avec le serveur *proxy* au moyen de la couche *RPC* ⑤ .

***Binding* PKCS#11 en OCaml** ① Comme nous l'avions introduit en 3.2, le choix du langage OCaml nous a imposé le développement d'une interface OCaml/C pour le standard PKCS#11. En effet, celle-ci n'existait pas parmi les bibliothèques *open source* fournies par la communauté.

En sus de l'utilisation comme composant de `Caml Crush`, ce *binding* PKCS#11 présente plusieurs intérêts du fait de son indépendance du reste du proxy :

- Il ouvre l'intégration et l'utilisation aisée de ressources cryptographiques PKCS#11 – comme les cartes à puce – dans des projets développés en OCaml.
- Le standard PKCS#11 décrit une API « bas niveau » pensée pour du code en C. L'utilisation de OCaml permet de s'affranchir de certaines de ces contraintes, telles que les allocations et désallocations mémoire : comme nous le précisons ci-dessous, cela peut représenter un réel avantage pour le développeur.

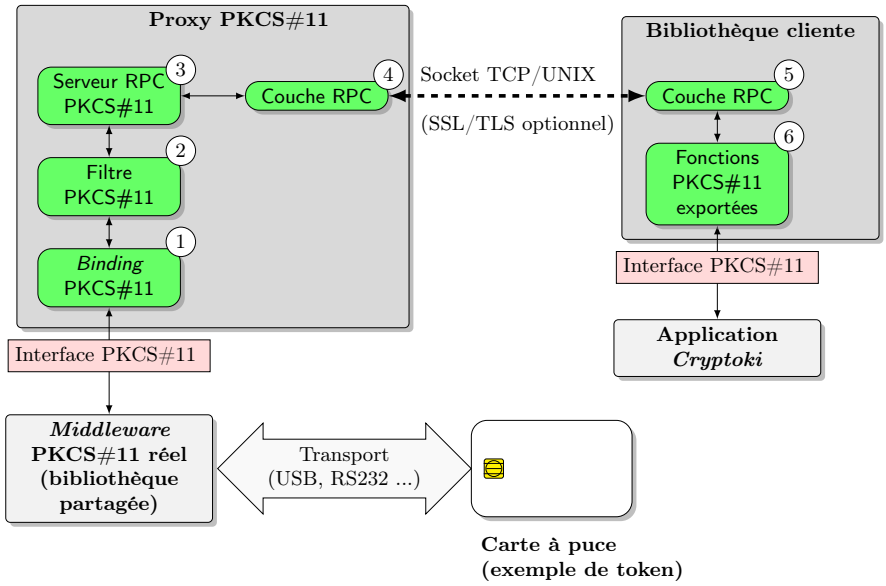

Figure 2. Vue d'ensemble de l'architecture de `Caml Crush`

Tel qu'exposé en 3.2, c'est CamlIDL qui est utilisé pour générer les *stubs* en C permettant de transformer les variables d'un langage à l'autre. La figure 3 présente cette génération dans l'étape (a). Elle comprend :

- L'outil CamlIDL (sous la forme d'un binaire `camlidl`) prend en entrée un fichier au standard IDL et génère des fichiers C et OCaml.
- Le fichier `pkcs11.idl` : il est inspiré du fichier C d'interface `pkcs11.h` publiquement fourni par RSA Laboratories [11]. Y figurent principalement les structures de données (structures proches des `struct` C) et autres `typedef` définis par le standard, ainsi que les déclarations des prototypes des fonctions PKCS#11.
- CamlIDL génère, à partir de `pkcs11.idl`, un fichier `pkcs11_stubs.c` ainsi qu'un fichier `pkcs11.h`. Ce dernier fichier `.h` correspond à un équivalent du fichier fourni en [11]. Le fichier `pkcs11_stubs.c` contient toute la mécanique de *stubbing* permettant de transformer des structures du OCaml en C lors de l'appel d'une fonction PKCS#11 depuis du OCaml, et du C en OCaml lorsqu'il s'agit de rendre un résultat fourni par le *middleware*.
- CamlIDL génère enfin des fichiers OCaml `pkcs11.ml` et `pkcs11.mli` contenant la traduction des structures de données dé-

finies dans l'IDL en OCaml, ainsi que les prototypes des fonctions PKCS#11.

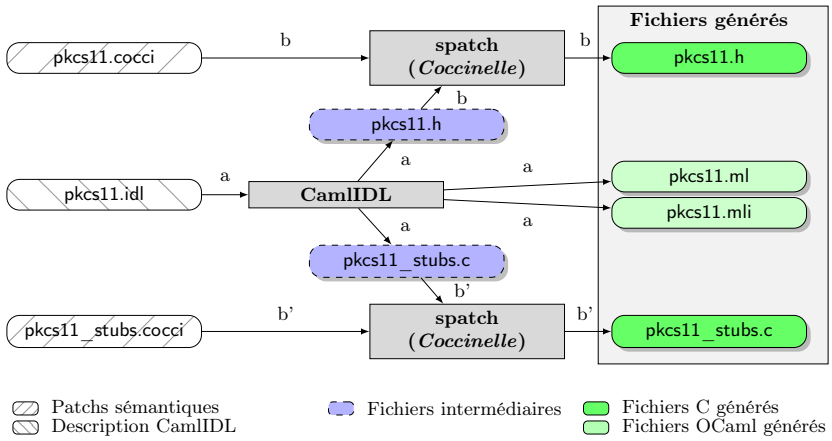

Figure 3. Processus de génération du code de l'interface PKCS#11

Nous prenons en annexe A un exemple concret de traduction d'une structure C simple par l'IDL.

Serveur RPC en OCaml ③ ④ Le serveur RPC est un des composants critiques de l'architecture. En écoute sur une socket, sa conception doit être robuste et sécurisée face à des requêtes de clients légitimes mais aussi hostiles.

Nous utilisons la bibliothèque *OCamlnet* [8] comme base de départ de notre *proxy*. Le but du projet est d'étendre la gestion système d'OCaml et de fournir un support pour des protocoles réseau. Le projet intègre le sous-ensemble *Netplex*, un *framework* dont le rôle est la gestion générique des processus serveur. Il intègre :

- Un parseur de configuration ;
- la gestion des sockets ;
- un support de TLS pour protéger le canal de communication ;
- un module de journalisation ;
- un mécanisme d'administration du processus serveur.

Netplex définit deux modes de gestion des connexions clientes : le modèle *prefork* et le modèle *worker*. Dans le premier cas, chaque connexion cliente est gérée par un processus dédié qui exécute un unique *thread*. Dans le second cas, les connexions clientes sont regroupées et traitées par des

threads au sein d'un processus. Afin d'éviter les problèmes mentionnés en 3.2, *Netplex* sera configuré avec le modèle *prefork*.

Le second avantage de *Netplex* est qu'il peut être étendu pour gérer des protocoles spécifiques. Il dispose d'un support mature de *Sun RPC* dont la sérialisation de données est compatible avec les autres implémentations du standard. Comme mentionné précédemment, il dispose d'un générateur de code OCaml depuis les fichiers d'interface XDR. L'annexe B détaille la syntaxe XDR pour l'exemple de structure `ck_date` précédent.

Par ailleurs, la protection du canal de communication avec le *proxy* est primordiale puisque des informations sensibles comme le code PIN du *token* transitent sur ce lien. `Caml Crush` utilise le support TLS de *Netplex*, fourni par *ocaml-ssl* [7], pour offrir l'authentification mutuelle par certificats. La liste des mécanismes cryptographiques acceptés par la pile TLS du serveur est configurable [1].

Filtre ② Le filtre, développé intégralement en OCaml, est au cœur des mécanismes de filtrage du *proxy*. La section 4 lui étant dédiée, nous ne nous étendons pas ici sur son fonctionnement interne. Lors de l'établissement d'une connexion cliente, le processus parent de *Netplex* affecte la connexion à un processus fils dans lequel les opérations de filtrage sont effectuées. Le moteur de filtrage s'appuie sur *Config_file* [3] pour parser et instancier la politique dans son propre contexte.

Bibliothèque cliente PKCS#11 ⑤ ⑥ Le *middleware* d'origine est remplacé par notre bibliothèque *Cryptoki* de substitution. Son rôle est d'ouvrir un canal de communication avec le serveur *proxy*, d'exporter les symboles PKCS#11 ⑥ pour les applications clientes, et de transmettre les appels de fonctions avec des arguments sérialisés. Le code de la couche de transport ⑤ est généré depuis le fichier d'interface XDR. Notre code [2] peut utiliser les piles *OpenSSL* ou *GnuTLS* pour protéger le canal de communication. Lorsque TLS est utilisé, le client doit s'authentifier auprès du serveur *proxy*. Cette fonctionnalité doit être gérée de manière transparente par la bibliothèque. Le certificat client, la clé privée associée et le certificat de l'autorité de confiance peuvent être définis de trois manières différentes :

- L'emplacement des fichiers à charger compilé dans la bibliothèque.

1. Il faut néanmoins compiler la version de développement de *ocaml-ssl* pour utiliser TLS 1.2 et des paramètres DH/ECDH.
2. Une partie de ce code est repris et adapté des travaux de Richard Jones [13].

- Des variables d'environnement pour indiquer l'emplacement sur le système de fichiers.
- Les éléments d'authentification peuvent être stockés dans la bibliothèque (intégrés dans le binaire généré à la compilation).

Ce composant de l'architecture est développé en langage C [3]. Les fonctions PKCS#11 de substitution réalisent des contrôles basiques sur les requêtes reçues, ceci présente l'avantage de ne pas transmettre des appels invalides au serveur *proxy*. Cependant, la bibliothèque cliente ne joue aucun rôle dans la protection de la ressource cryptographique. Elle n'est pas considérée comme un élément critique pour la sécurité de la solution.

4 Moteur de filtrage PKCS#11 de `Caml Crush`

4.1 Description du moteur

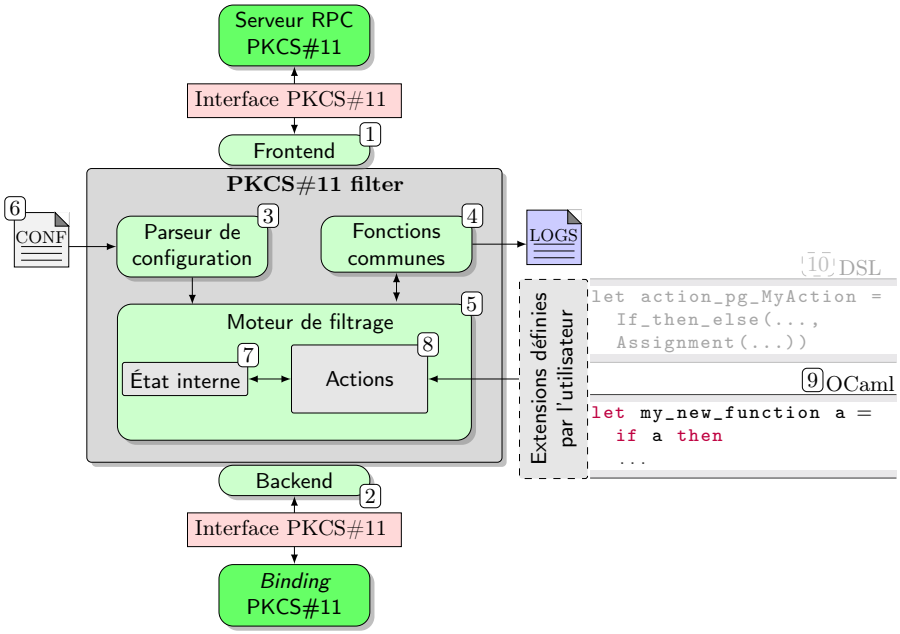

Figure 4. Vue détaillée de l'architecture du filtre

Le moteur de filtrage est au cœur de l'architecture du *proxy*. Il s'agit de la brique essentielle chargée de pallier aux défauts du standard PKCS#11.

3. Un client hybride C/OCaml existe mais n'est pas proposé par défaut.

La figure 4 illustre le détail de ses composants. Dans un souci d'isolation, un **frontend** [1] sépare le *proxy* du moteur de filtrage, tandis qu'un **backend** [2] isole ce dernier de l'interface définie par le *binding* PKCS#11 en OCaml. Le filtre utilise le parseur [3] de *Config_file* pour analyser sa configuration avant de s'en servir pour initialiser une instance du filtre. Les fonctions usuelles de journalisation et de traitement des exceptions sont réunies dans un même module [4]. Enfin, le moteur de filtrage constitue le cœur du filtre [5].

C'est ce dernier module qui remplace le comportement par défaut de la ressource proxifiée lors de l'appel d'une fonction PKCS#11 en fonction de ce qui est défini dans le fichier configuration. A la réception d'un appel à l'API par le *frontend*, celui-ci est transmis au moteur de filtrage, qui peut alors évaluer des conditions sur les entrées, ou utiliser un **état interne** [7] où sont stockées quelques informations utiles à la politique de filtrage. Il peut ensuite effectuer un ou plusieurs appels à la ressource proxifiée grâce au *backend*, pour enfin retourner une valeur de retour. Cette valeur est possiblement différente de celle qui aurait résulté d'un appel direct à la ressource. L'ensemble de ces opérations constitue une **action de filtrage** [8]. Une fois configuré, le moteur de filtrage associe une telle action à chaque fonction disponible selon le standard. Nous soulignons que l'état interne du filtre est propre à chaque instance du filtre, ce qui limite intrinsèquement sa portée.

Le moteur de filtrage est pensé pour être extensible, un utilisateur peut tout à fait définir de nouvelles actions de filtrage, si celles disponibles ne satisfont pas ses besoins. Il faut alors que l'utilisateur ajoute le code de son choix, en OCaml [9], au code actuellement exécuté lorsqu'une fonction de *Cryptoki* est appelée. Cet ajout se fait aisément dans un fichier bien documenté, la fonction ainsi définie par l'utilisateur peut alors être ajoutée à une liste d'actions déjà prédéfinies via le fichier de configuration (sans entrer dans les détails, la priorité des appels aux actions ainsi définies correspond à l'ordre de leur déclaration dans le fichier de configuration).

Dans les paragraphes suivants, nous allons détailler les actions de filtrage actuellement disponibles dans l'implémentation de `Caml Crush`, ou résultant d'une extension à fournir par l'utilisateur à peu de frais (certaines seront bientôt disponibles en [1] pour donner des exemples concrets d'extensions). Nous allons aussi préciser quelles attaques sont corrigées par les actions de filtrage mises en place. Enfin, nous présenterons une nouvelle architecture actuellement en cours de développement pour le moteur de filtrage. Celle-ci est destinée à le rendre plus facilement adaptable aux

besoins spécifiques d'un utilisateur, tout en minimisant les efforts à fournir par celui-ci.

4.2 Fonctionnalités offertes par le filtre et sécurisation de l'API

Blocage de fonctions. Intercepter un appel de fonction et le bloquer en simulant une erreur, tout en restant compatible avec le standard, est peut-être la chose la plus facile à implémenter avec une architecture plaçant le filtre en coupure. Bien qu'assez triviale, cette action de filtrage est très utile, comme exposé dans la section 2.

Durcissement de la conformité au standard. Comme mentionné dans la section 1, même si toutes les mesures à mettre en place pour empêcher un adversaire de récupérer les valeurs d'objets marqués sensibles ou non-extractibles ne sont pas explicitées dans le standard, il existe un certain de nombre de vérifications et de restrictions clairement établies. Ceci ne signifie pas que ces mesures sont correctement implémentées par les ressources cryptographiques disponibles dans le commerce. Typiquement, le travail effectué par Bortolozzo *et al.* dans l'article [19] a mis en évidence que certains *tokens* répondent la valeur de l'objet lors d'une requête directe faite par un appel à `C_GetAttributeValue`, et ce **indépendamment** de la sensibilité de l'objet. De telles erreurs, très préjudiciables pour la sécurité offerte par le *token*, peuvent facilement être corrigées grâce au filtrage. En effet, dans l'exemple ci-dessus, l'action de filtrage appropriée consiste à vérifier la valeur des attributs *sensitive* et *extractable* par des requêtes à la ressource puis, **seulement dans le cas où l'objet est extractible et non sensible**, à procéder à la requête de la valeur demandée.

Des stratégies similaires peuvent être mises en œuvre pour renforcer la confiance placée dans le comportement du *token* pour des requêtes particulièrement sensibles. Ainsi, nous envisageons d'étendre prochainement les contrôles actuellement effectués pas le filtrage de manière à s'assurer que les accès en écriture au *token* ne peuvent être réalisés que dans les circonstances prévues par le standard.

Segmentation des objets par filtrage des *labels*. La notion de visibilité des objets a été introduite en section 1. On rappelle en particulier qu'un utilisateur non-authentifié a accès à tous les objets étiquetés *public*, et qu'un fois authentifié, il peut utiliser **l'ensemble** des objets disponibles

sur un *token*. Ceci peut présenter des inconvénients dans des scénarios d'utilisation d'une même ressource cryptographique par plusieurs utilisateurs ou par des environnements de criticités différentes (par exemple une architecture multi-niveaux). C'est pourquoi nous souhaitons offrir la possibilité de restreindre la visibilité des objets, en créant une instance virtuelle de la ressource cryptographique par niveau grâce au filtre.

Le standard PKCS#11 définit un attribut **label** pour les objets, sans imposer de restrictions particulières. Cet attribut est normalement destiné à nommer les objets selon le choix de l'utilisateur. Il est possible de tirer avantage des *labels* en définissant une politique de nommage propre au moteur de filtrage de manière à pouvoir discriminer les objets selon leur niveau – par exemple le préfixage ou suffixage des *labels* choisis par l'utilisateur par le nom du niveau auquel les objets appartiennent. On peut considérer que tous les objets présents sur le *token* ont des *labels* conformes à cette politique de nommage [4]. C'est la configuration du filtre qui contient la description des *labels* autorisés. Une fois une instance du filtre initialisée avec cette configuration, lors d'une tentative d'utilisation d'un objet, le moteur de filtrage actuel va récupérer son *label* et décider du succès de l'appel en fonction de son appartenance à l'ensemble des *labels* autorisés. Si l'objet possède un *label* interdit, la requête ne sera jamais transmise à la ressource et le moteur de filtrage simulera l'absence sur la ressource de l'objet désigné. Ces éléments sont schématisés sur la figure 5.

Selon la logique développée dans le standard, l'utilisation d'objets est précédée d'une recherche d'objets satisfaisant certains critères, qui résulte en l'obtention de *handles* référençant des objets convenables. Lors d'une demande de recherche d'objets, le moteur de filtrage transmet les appels correspondants, récupère des handles vers l'ensemble des objets disponibles sur la ressource qui satisfont le critère de recherche donné, puis effectue des requêtes pour obtenir leurs *labels* et ne laisse apparaître dans sa réponse que les objets ayant des *labels* autorisés. Cette implémentation est gourmande en appels à la ressource, mais c'est celle choisie pour `Caml Crush`. Des alternatives consistent à entretenir un état local au filtre, mais elles ont le désavantage soit de nécessiter une mécanique beaucoup plus complexe, soit de ne pas bloquer complètement certaines attaques.

4. Cela semble raisonnable d'après l'hypothèse selon laquelle la ressource n'est jamais directement accessible sans une instance du filtre, et si la ressource est bien initialisée.

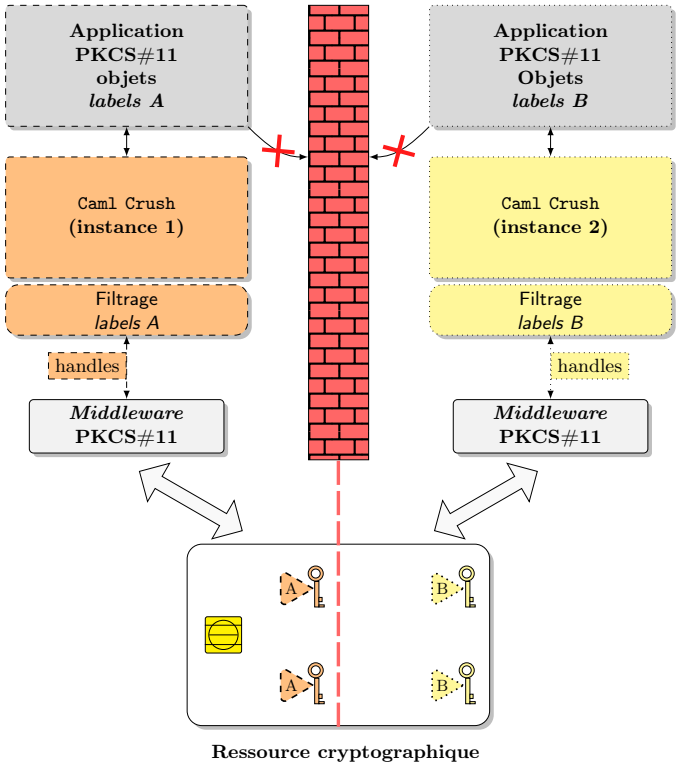

FIGURE 5. Segmentation des objets en fonction de leurs *labels*

4.3 Sécurisation de la ressource contre les attaques classiques

Lors de l'introduction de l'interface *Cryptoki*, nous avons illustré la possibilité de récupérer la valeur d'objets sensibles ou non-extractibles en jouant sur la valeur d'attributs, tout en restant compatible avec le standard.

Dans [19,4], deux stratégies de sécurisation du standard sont proposées. Elles sont survolées ici, pour que l'on puisse évoquer leur implémentation dans le moteur de filtrage. La **première stratégie** consiste en la mise en œuvre simultanée de trois mesures :

- Détecter et empêcher la prise de valeurs conflictuelles pour certains couples d'attributs d'un même objet, à chaque appel permettant le changement de ces valeurs. Typiquement, on a donné l'exemple dans 1 des attributs *wrap* et *decrypt* qui ne peuvent être fixés à vrai tous les deux sans faire courir un risque de fuite d'information. Il

en va de même pour d'autres paires d'attributs, comme *unwrap* et *encrypt*.

- Interdire les changements de valeurs de certains attributs une fois définis, par exemple *sensitive* et *extractable* On souhaitera aussi empêcher la prise de valeurs conflictuelles dans le temps des couples d'attributs cités ci-dessus.
- Modifier le mécanisme d'encapsulation proposé par défaut pour y ajouter, lors de l'export d'une clé, l'export de ses attributs pertinents (ses propriétés, son usage), protégés en intégrité. Ceci implique évidemment la transformation adéquate du mécanisme de décapsulation par l'adjonction des vérifications de l'intégrité et l'initialisation des attributs récupérés. Cette dernière modification n'est pas complètement anodine, dans le sens où la compatibilité avec le standard est perdue.

La **seconde stratégie** repose sur la définition de matrices spécifiant des valeurs compatibles d'attributs d'un objet : ce sont les *secure templates*. Les objets instanciés par l'utilisateur de l'API doivent alors adopter pour les valeurs de leurs attributs une des matrices proposées, et ce au moment de leur création. Cette seconde tactique présente l'avantage d'être complètement compatible avec le standard, mais impose des contraintes sur les applications utilisatrices. En effet, l'usage et les propriétés des clés devront bien être fixés par le passage en argument d'une « bonne » matrice à l'appel résultant dans l'instanciation de l'objet. Tout recours ultérieur à un appel à `C_SetAttributeValue` sera impossible.

Ces deux stratégies permettant de « réparer » PKCS#11 sont implémentables assez simplement dans le moteur actuel **sans utiliser de mémoire**.

On mentionnera enfin que l'utilisation de mécanismes cryptographiques jugés non fiables peut être proscrite facilement grâce au filtrage. De plus, la substitution au mécanisme d'encapsulation standard d'un mécanisme non-compatible mais portant un motif d'intégrité permet de prévenir les attaques liées à l'utilisation de la décapsulation *unwrap* comme un **oracle de padding** [18,22,17] par un attaquant.

4.4 Une nouvelle architecture pour le moteur de filtrage

Limites de l'implémentation actuelle. Dans toute l'architecture du projet s'inscrit une volonté de modularité, et le moteur de filtrage n'échappe pas à cette logique. D'une part, son isolation permet de le remplacer facilement par un autre moteur. D'autre part, les actions de filtrage

sont confinées dans un module de manière à limiter le périmètre des interventions nécessaires de la part d'un utilisateur souhaitant réaliser de nouvelles actions de filtrage. Cependant, il reste quand même à un tel utilisateur un certain effort d'appropriation du code environnant - par exemple, les contraintes liées au *binding* qui ont forcé à changer un peu le standard pensé pour du C. Par ailleurs, les extensions utilisateur actuelles utilisent la bibliothèque de *marshaling* de OCaml [6] qui n'est pas *type-safe* : la moindre erreur de développement de la part de l'utilisateur peut être fatale au *runtime* car non repérée par l'inférence de types du compilateur. Pour ces raisons, nous avons souhaité améliorer le moteur de filtrage actuellement disponible en utilisant plus complètement le typage fort propre à OCaml, et en introduisant une possibilité d'extension du filtre demandant moins d'effort.

Nouvelle implémentation en cours de développement. La nouvelle implémentation est en cours de prototypage et d'intégration au *proxy*. D'une part, l'exploitation du typage fort offert par le langage OCaml a été poussée au maximum. D'autre part, la notion d'action de filtrage a été formalisée par l'introduction d'un langage intermédiaire défini spécifiquement pour décrire les opérations à exécuter par le moteur de filtrage. L'idée sous-jacente est de permettre l'extension du filtre à moindre effort.

Ce petit langage (voir [10] sur la figure 4) est construit sur la base d'**actions atomiques** et d'**actions booléennes**. Celles-ci constituent des briques élémentaires que l'on peut combiner entre elles grâce à l'introduction de variables, l'utilisation d'affectations, de séquences et de branchements conditionnels. Par combinaison, on obtient des programmes, nommés **action programs**, qui sont exécutés sur un état local à l'action (différent de l'état interne du filtre). Cet état local est représenté par une valuation, qui associe aux variables définies dans le programme leurs valeurs respectives. Avant qu'un programme ne débute son exécution, le moteur de filtrage initialise la valuation sur laquelle le programme va travailler en associant à la variable spéciale nommée `The_input` la valeur de l'argument passé à l'appel de la fonction PKCS#11 que l'on va traiter. Ensuite, le programme décrivant les opérations de filtrage à faire est exécuté, et cela modifie la valuation au fur et à mesure. Quand il se termine, le programme doit rendre au filtre une valuation dans laquelle la variable spéciale `The_output` est associée à une valeur. C'est cette valeur qui sera considérée par le moteur de filtrage comme la sortie à effectuer en réponse à l'appel PKCS#11 passé.

```
type action_program =
  Assignment of action_program_variable * atomic_action
| Sequence of action_program * action_program
| If_then_else of boolean_action * action_program * action_program

type atomic_action =
  Call_the_\backend of f_P11_functions * action_program_variable
| Mechanism_invalid of f_P11_functions
| etc.

type boolean_action =
  Ck_mechanism_allowed of ck_mechanism_type_t list
| etc.

let action_pg_EncryptInit =
  If_then_else(Ck_mechanism_allowed(list_of_forbidden_mechanisms),
  Assignment(The_output, Call_the_\backend(F_c_EncryptInit, The_input)),
  Assignment(The_output, Mechanism_invalid(F_c_EncryptInit)))
```

Figure 6. Eléments de définition du langage de programmation dédié à la spécification des actions

La figure 6 montre des fragments du code en OCaml servant à définir le petit langage de description des actions. On y trouve un exemple, court mais réaliste, de programme spécifiant les opérations à effectuer par le moteur de filtrage lors d'un appel à la fonction `C_EncryptInit`. Cette fonction est appelée au début d'un chiffrement, et prend entre autres un argument définissant le mécanisme à utiliser. Nous avons évoqué ci-dessus la possibilité d'interdire l'utilisation de certains mécanismes cryptographiques. Pour ce faire, une liste de mécanismes interdits peut être définie lors de la configuration du filtre, chargée à son instanciation. Le moteur de filtrage devra ensuite, lors d'un appel à `C_EncryptInit`, vérifier l'appartenance du mécanisme à cette liste, et en fonction de son autorisation, transmettre l'appel à la ressource cryptographique et récupérer le résultat, ou répondre par un message d'erreur adéquat. Ceci est écrit assez trivialement dans le langage dédié, par l'évaluation d'une action booléenne, suivie soit par une l'action atomique de transfert de l'appel au *backend* puis l'affectation du résultat dans la variable spécifique dédiée, soit par l'action atomique qui affecte au résultat le bon message d'erreur.

On remarque dans cet exemple que pour que le langage dédié soit utilisable, il faut réfléchir à la définition d'actions représentatives des souhaits usuels d'un utilisateur. Evidemment, il est toujours possible à un utilisateur d'écrire lui-même ses propres actions. Si nous supposons d'un côté qu'un tel utilisateur connaît *Cryptoki*, nous souhaitons cependant minimiser l'impact de nos choix architecturaux sur l'effort nécessaire à une telle extension. En particulier, l'exploitation active du typage fort en OCaml implique l'omniprésence de types, de leurs fonctions de construction et de

destruction. Toutes ces fonctions et leurs combinaisons usuelles ont donc été prédéfinies. De plus, l'emploi de stratégies de nommage homogènes prétend rendre la définition de nouvelles actions accessible à l'utilisateur sans qu'il ne prenne réellement connaissance du code environnant.

Enfin, par rapport à la version actuellement disponible, cette nouvelle implémentation possède un état interne un peu plus sophistiqué et qui collecte par défaut, pour chaque appel entrant dans le filtre, son entrée, la sortie produite, et le détail des appels effectués pour construire la réponse. L'ensemble des appels collectés effectivement est configurable par passage d'une variable globale dans la configuration. Ceci permet donc de vérifier des propriétés temporelles simples. Ici encore, le type de l'état est extensible, et on peut envisager de conserver d'autres informations.

5 Avantages de `Caml Crush`

5.1 Support d'architectures multiples

D'une manière générale, les *middlewares* PKCS#11 distribués avec les ressources cryptographiques sont sous forme de binaires. Ces versions sont disponibles pour les systèmes d'exploitation et CPU répandus. De fait, il est impossible d'utiliser le matériel sur une architecture « exotique » (telle MIPS ou SPARC). L'utilisation de *Sun RPC* comme protocole de communication au sein de `Caml Crush` facilite le portage de notre code. La première version de notre architecture était compatible avec Linux. Le portage pour Mac OS X et FreeBSD ne nous as pas demandé d'efforts significatifs. De plus, `Caml Crush` est capable de gérer des connexions clientes avec des *endianness* différentes. Il est par exemple tout à fait possible d'utiliser des clients MIPS ou PowerPC 32-bit pour communiquer avec un *proxy* PKCS#11 s'exécutant sur une architecture `x86_64`. Nous pensons que c'est atout non négligeable pour les plateformes embarquées ou les périphériques mobiles.

5.2 Support de *middlewares* multiples

Comme mentionné en 3.2, `Caml Crush` utilise un processus par connexion cliente pour obtenir une isolation en cohérence avec le standard PKCS#11. Ceci offre l'avantage de pouvoir charger des *middlewares* PKCS#11 différents. Ainsi, une même instance du *proxy* PKCS#11 est capable d'exposer aux clients les fonctionnalités de ressources cryptographiques différentes. Concrètement, un appel RPC particulier permet de

demander le chargement d'un *middleware* particulier. Le moteur de filtrage vérifie la validité de la demande en fonction de la politique. La figure 7 illustre ce principe, un premier client charge un *middleware* configuré pour accéder à une carte à puce, le second client utilisera une ressource de type HSM. Cette fonctionnalité n'est pas essentielle au fonctionnement de `Caml Crush` mais apporte de la flexibilité pour les évolutions futures. Par exemple, la ségrégation des clients en fonction de leur criticité repose aujourd'hui sur des instances multiples du *proxy*. Ceci pourrait évoluer pour qu'une même instance écoute sur des sockets différentes, la politique de filtrage serait associée à une socket et non pas à une instance du proxy. De même, le chargement d'un sous-ensemble de *middlewares* pourrait être autorisé en fonction de la criticité des clients. Une autre utilisation serait de proposer un « concentrateur » PKCS#11 capable de présenter une vue logique unifiée de plusieurs *tokens*. Ceci nécessiterait néanmoins une réimplémentation quasi-complète de PKCS#11, ce qui n'est pas la vocation initiale de `Caml Crush`.

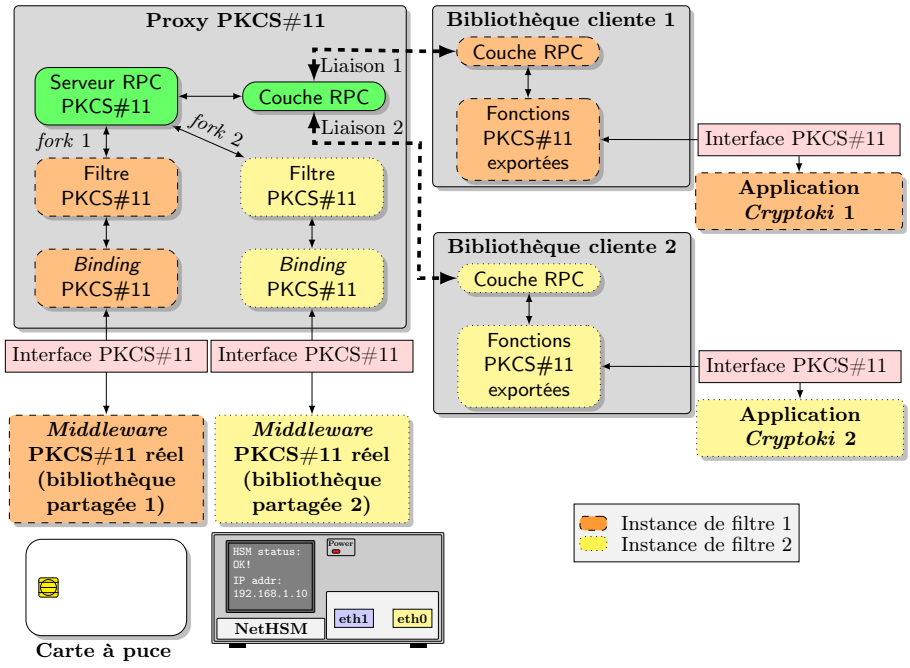

Figure 7. Gestion de *middlewares* multiples

5.3 Sécurité proactive

Les efforts mis en place permettent d'utiliser les ressources cryptographiques dans de meilleures conditions de sécurité. La protection offerte couvre les attaques connues sur le standard. Comme nous l'avons vu en 4, les actions de filtrage peuvent cependant être étendues. Ceci incite à une approche proactive de la sécurité de la part des utilisateurs de `Caml Crush`. En effet, la découverte de vulnérabilités dans les *middlewares* utilisés peut être compensée par la mise en place de règles de filtrage adéquates. Illustrons cela. Si une fonction PKCS#11 s'avère vulnérable à un *buffer overflow* sur une de ses entrées – découvert via du *fuzzing* par exemple –, il suffira à l'utilisateur d'ajouter une règle limitant la taille du *buffer* au niveau du filtre (donc avant que celui-ci n'atteigne le *middleware* possiblement exploitable).

Notons que le développement d'extensions sera d'autant plus facilité lorsque la seconde version du moteur de filtrage sera disponible.

6 Performances et limitations de `Caml Crush`

6.1 Performances

Le fait d'ajouter une surcouche logicielle par dessus une autre induit généralement un impact sur les performances globales. La proxification des appels PKCS#11 n'engendre cependant pas de baisse significative des performances. En effet, le lien utilisé pour communiquer avec le matériel est dépendant du type de *token*, bus série, parallèle ou USB pour les cartes à puce ou réseau pour certains HSM. Ces modes de transport de la donnée sont comparativement plus lents que le temps de traitement par le serveur *proxy*. En revanche, le moteur de filtrage fait appel à la fonction `C_GetAttributeValue` pour obtenir les informations nécessaires à l'action de filtrage. Ce sont ces requêtes PKCS#11 supplémentaires qui augmentent le temps de traitement des opérations. Nous tenons à souligner que l'impact est totalement dépendant de la performance intrinsèque du matériel. Les essais réalisés sur les ressources cryptographiques à notre disposition révèlent que ce compromis reste acceptable.

6.2 Bloquer l'accès direct à la ressource

La sécurité apportée par `Caml Crush` repose sur l'impossibilité pour un attaquant de communiquer directement avec la ressource cryptographique. Il est réaliste de faire l'analogie avec un pare-feu qui protège un réseau

informatique, il ne doit pas être contourné pour que sa politique de filtrage soit utile. Des mesures complémentaires sont donc essentielles : il faut que Caml Crush soit le point de passage obligé des requêtes PKCS#11 vers la ressource cryptographique. La méthodologie sera différente en fonction du type de *token*. Dans le cas des HSM réseau, il est important de les isoler dans un réseau dédié et de s'assurer que seules les instances de Caml Crush peuvent y accéder. Cette mise en œuvre ne comporte pas de difficultés majeures.

Caml Crush peut-être installé sur des postes standard dans le but de sécuriser l'accès à des ressources de type carte à puce. Ce scénario de déploiement demande d'appliquer des mesures de durcissement du poste informatique pour éviter le contournement du proxy filtrant. Prenons l'exemple d'un poste Linux, celui-ci peut être configuré pour utiliser des dispositifs de contrôle d'accès (discrétionnaire ou obligatoire) et des mécanismes de *sandboxing* qui permettent d'isoler le *token*. L'image 8 illustre un déploiement sécurisé de Caml Crush. Dans cet exemple, une application PKCS#11 charge la bibliothèque cliente Caml Crush ⟨1⟩ . Une socket Unix ⟨2⟩ accessible pour les utilisateurs du système est utilisée pour communiquer avec le proxy filtrant. L'instance de Caml Crush est autorisée à charger le *middleware* PKCS#11 de la carte à puce ⟨3⟩ et dispose d'un accès à la socket de communication de PC/SC ⟨4⟩ . Le processus PC/SC est autorisé à se connecter au périphérique USB pour transmettre les ordres propriétaires.

Le système est donc durci avec politique d'accès obligatoire pour restreindre l'accès aux périphériques et aux sockets du système. Premièrement, le processus PC/SC est autorisé à accéder la carte à puce USB, tout autre accès directs est bloqué ⟨6⟩ . Deuxièmement, la politique interdit l'utilisation de la socket PC/SC aux utilisateurs ⟨7⟩ . Enfin, seules les instances de Caml Crush sont autorisées à charger le *middleware* d'origine ⟨8⟩ . Ces mesures de durcissement garantissent que les applications clientes traversent le *proxy* PKCS#11 filtrant, protégeant ainsi les ressources cryptographiques.

6.3 Limites du filtrage

Nous avons abordé en 4 certaines des limites qui nous poussent d'ores et déjà au développement d'une version améliorée du moteur de filtrage. Comme mentionné en détail dans la section 4, un état interne est maintenu pour chaque application *Cryptoki*. Il serait intéressant de pouvoir corréler des informations entre différentes connexions clientes. Cependant, nous rappelons que ces états internes sont maintenus dans des processus

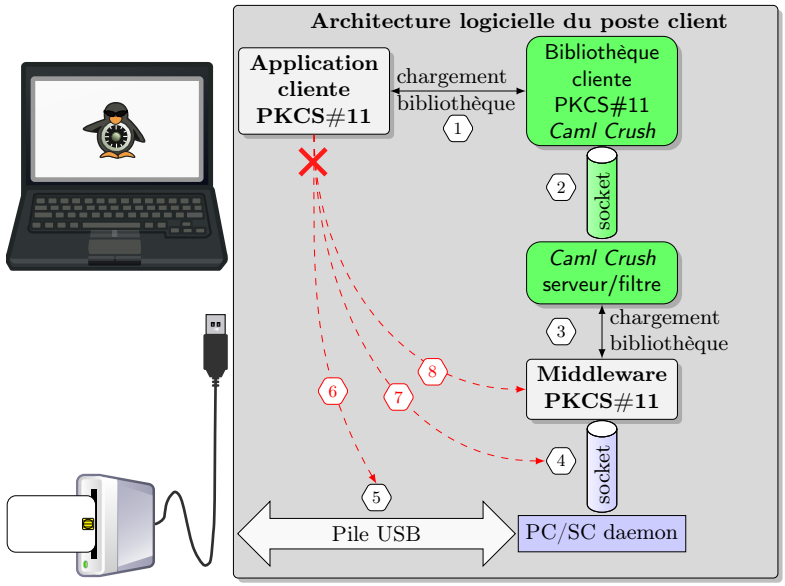

FIGURE 8. Configuration de `Caml Crush` sur un poste durci.

distincts. `Caml Crush` devrait intégrer un mécanisme de communication inter-processus (IPC) pour échanger des messages. Malgré l'intérêt que présente cette fonctionnalité, il faudrait intégrer des primitives de synchronisation et développer un moteur de corrélation, ce qui augmenterait de manière considérable la complexité du code actuel.

Conclusion

Notre solution permet d'appliquer les contremesures aux attaques PKCS#11 de manière dynamique. Les travaux de recherche ont apporté des réponses concrètes pour améliorer la sécurité de PKCS#11. Il n'en reste pas moins que la mise en œuvre de ces correctifs est à la discrétion des fabricants de matériel cryptographique. En pratique, il est contraignant de modifier du matériel déployé à grande échelle, les *tokens* restent donc vulnérables. `Caml Crush` est une alternative pragmatique pour protéger les ressources cryptographiques contre l'ensemble des attaques connues. En effet, le principe de mise en coupure de notre *proxy* filtrant entre les applicatifs et le *middleware* PKCS#11 est simple et efficace. De plus, notre moteur de filtrage est modulaire et autorise l'ajout d'extensions spécifiques (sans nécessiter des compétences avancées en OCaml pour la prochaine version du filtre).

La flexibilité du moteur de filtrage permet d'étendre le champ des possibilités de l'architecture. Ainsi, le déploiement de `Caml Crush` permet d'obtenir des propriétés de sécurité supplémentaires lors de l'utilisation des *tokens*. Les politiques de filtrage peuvent ainsi restreindre les mécanismes cryptographiques, bloquer des appels de fonctions ou encore améliorer la conformité d'un *token*. L'utilisation de `Caml Crush` permet entre autres le partage sécurisé de la ressource cryptographique. Nous sommes convaincus de la pertinence de ces fonctionnalités pour les utilisateurs de matériel compatible, raison pour laquelle le code source de notre projet est immédiatement disponible [1].

Des travaux sont déjà engagés pour offrir une nouvelle mouture du moteur de filtrage. L'utilisation du typage offert par OCaml sera étendue et la syntaxe des règles sera simplifiée. Les développements futurs doivent aussi permettre d'affiner les capacités du moteur de filtrage grâce à la mise en place d'un mécanisme de corrélation entre les différentes instances du filtre. Des attaques plus complexes pourraient ainsi être détectées et bloquées.

Annexe A : détails sur CamlIDL

Nous prenons ci-dessous un exemple concret de traduction d'une structure C simple par l'IDL. La structure `ck_date` définie dans le standard PKCS#11 contient trois *buffers* de tailles statiques, comme illustré en listing 1.1.

```
struct ck_date {
  unsigned char year[4];
  unsigned char month[2];
  unsigned char day[2];
};
```

Listing 1.1. Structure `ck_date` définie dans `pkcs11.h`

CamlIDL traduit cette structure en type `record` contenant trois champs qui sont des `char arrays`, voir listing 1.2.

```
type ck_date = {
  year: char array;
  month: char array;
  day: char array;
}
```

Listing 1.2. Structure `ck_date` traduite en record OCaml par CamlIDL

Afin de passer de la structure C au record OCaml et réciproquement, CamlIDL génère des *stubs* en C qui interagissent avec le **runtime Caml** (voir les listings 1.3 et 1.4). OCaml gère la taille de ses tableaux dynamiquement, le fait que les `char arrays` de `ck_date` ne puissent pas dépasser une certaine taille est implémenté au sein des fonctions de *stubbing* : ces fonctions lèvent une erreur au runtime Caml – au sens exception OCaml – s'il y a un problème.

```
/* Fonction de transformation de ck_date du monde OCaml vers le C */
void camlidl_ml2c_pkcs11_struct_ck_date(value _v1, struct ck_date *
    _c2, camlidl_ctx _ctx)
{
  /* "Value" OCaml allouée et gérée par le runtime Caml
     _v3 contient un record OCaml ck_date                    */
  value _v3;
  ...
  /* Récupération du premier champ year du record ck_date  */
  _v3 = Field(_v1, 0);
  /* Récupération de la taille du char array du champ year */
  _c4 = Wosize_val(_v3);
  /* Vérification de la taille: elle doit être égale à 4   */
  if (_c4 != 4) invalid_argument("struct ck_date");
  /* Copie des octets du char array OCaml vers le buffer C
     Le stub suppose que la fonction appelante a alloué la
```

```
    structure _c2                                                    */
  for (_c5 = 0; _c5 < 4; _c5++) {
    _v6 = Field(_v3, _c5);
    (*_c2).year[_c5] = Int_val(_v6);
  }
  ...
}
```

Listing 1.3. *Stubbing* IDL OCaml vers C de la structure `ck_date`

```
/* Fonction de transformation de ck_date du monde C vers OCaml */
value camlidl_c2ml_pkcs11_struct_ck_date(struct ck_date * _c1,
    camlidl_ctx _ctx)
{
  /* Value OCaml qui va recevoir le record créé à partir
     de la structure C ck_date *_c1 donnée en entrée          */
  value _v2;
  ...
  /* Allocation dans le runtime Caml d'un char array qui      */
  /* sera le champ year du record _v2                         */
  _v3[0] = camlidl_alloc_small(4, 0);
  /* Copie octet par octet du buffer year dans le champ
     year du record _v2
     La taille de la copie est fixée à 4                      */
  for (_c4 = 0; _c4 < 4; _c4++) {
    Field(_v3[0], _c4) = Val_int((*_c1).year[_c4]);
  }
  Field(_v2, 0) = _v3[0];
  ...
  return _v2;
}
```

Listing 1.4. *Stubbing* IDL C vers OCaml de la structure `ck_date`

En plus des structures de données partagées entre le C et l'OCaml, CamlIDL génère aussi tous les prototypes de fonctions définies par l'API PKCS#11 (les *stubs* maniant les structures de données sont en fait appelés aux entrées et sorties de ces fonctions). Ces prototypes doivent néanmoins être adaptés à la syntaxe IDL. Prenons l'exemple de la fonction `C_Encrypt` qui est définie dans le *header* standard [11] par le prototype :

```
CK_RV C_Encrypt(/*in*/  ck_session_handle_t session,
                /*in*/  unsigned char *data,
                /*in*/  unsigned long data_len,
                /*out*/ unsigned char *encrypted,
                /*out*/  unsigned long *encrypted_len)
```

Ce prototype contient en argument des éléments qui sont à la fois des entrées (le *handle* de session, le pointeur sur les données `data` à chiffrer, leur taille `data_len`), et des sorties (un pointeur sur le *buffer* recevant les

données après chiffrement `encrypted`, ainsi qu'un pointeur sur un entier recevant sa taille `encrypted_len`).

L'un des objectifs du *binding* OCaml/C étant de simplifier l'utilisation de PKCS#11, il nous a paru intéressant de bien séparer en OCaml les entrées des sorties de fonctions (via l'utilisation de *tuples* comme valeurs de retour). Par ailleurs, l'IDL doit capturer le fait qu'OCaml gère les tailles de ses types dynamiquement. Voici dans le langage IDL une manière de redéfinir la fonction `C_Encrypt` :

```
CK_RV C_Encrypt([nativeint, in]/*in*/ ck_session_handle_t session,
    [size_is(data_len), in]/*in*/ unsigned char data[],
    [nativeint]/*in*/ unsigned long data_len,
    [size_is(*encrypted_len),out]/*out*/ unsigned char encrypted[],
    [ignore]/*out*/ unsigned long *encrypted_len)
```

Nous spécifions bien, via les métadonnées CamlIDL entre crochets, les éléments qui sont des entrées `in` et ceux qui sont des sorties `out` de la fonction OCaml. Certains éléments ne sont ni des entrées ni des sorties : les tailles des *buffers* sont implicites pour les `arrays` OCaml. Il faut par contre spécifier à CamlIDL que ces tailles sont bien les tailles des *buffers* C qui seront manipulés par le *stub* de `C_Encrypt` : cela est fait via la directive `size_is`. Enfin, la directive `nativeint` précise à CamlIDL qu'il faut traiter les entrées concernées, qui sont des types `long int` du C, en tant que type `nativeint` OCaml. Notons que l'utilisation des types entiers basiques d'OCaml n'est pas adaptée du fait de la taille de 31 bits (resp. 63 bits) sur plateformes 32-bit (resp. 64-bit), le bit de poids fort étant utilisé par le *garbage collector* au *runtime*.

Au final, le résultat de la définition d'interface CamlIDL précédente fournit le prototype OCaml suivant :

```
external C_Encrypt :
        ck_session_handle_t -> char array -> ck_rv_t * char array
        = "camlidl_pkcs11_ML_CK_C_Encrypt"
```

Comme nous pouvons le voir, ce prototype est assez « naturel » pour le programmeur : il prend en entrée un *handle* de session (qui est un `nativeint`) ainsi qu'un `char array` qui représente les données à chiffrer, et rend un couple contenant une valeur de retour sous forme de `nativeint` ainsi qu'un `char array` contenant les données chiffrées par `C_Encrypt`. Le mot clé `external` précise que la fonction dont le prototype est donné est un *stub* natif dont le symbole est `camlidl_pkcs11_ML_CK_C_Encrypt`.

CamlIDL nous a énormément simplifié la création du *binding* OCaml/C. Il présente néanmoins quelques limitations vis-à-vis de l'interface PKCS#11 qu'il nous a fallu contourner. En effet, PKCS#11 définit

certains appels particuliers permettant de récupérer les tailles prévisionnelles qu'un *buffer* aura après une opération. Cela est indispensable pour le programmeur C qui alloue les données au fur et à mesure des opérations effectuées. Si l'on reprend l'exemple de `C_Encrypt`, le paramètre `unsigned char *encrypted` permet, lorsqu'il vaut `NULL`, de récupérer dans la sortie `unsigned long *encrypted_len` la taille que doit avoir le *buffer* après l'opération de chiffrement. Le programmeur peut alors allouer cette taille et appeler une seconde fois `C_Encrypt`. Ce mode opératoire n'a plus vraiment d'intérêt en OCaml vu que le programmeur n'a plus à gérer les allocations mémoire. Il nous a donc fallu « cacher » au programmeur ces opérations « bas niveau » effectuées dans les *stubs*. Malheureusement, la syntaxe de CamlIDL n'avait pas assez d'expressivité pour traduire ce comportement : une solution possible, que nous avons adoptée, est de **patcher** les *stubs* **après leur génération**. C'est ce qui est représenté par les étapes (b) et (b') sur la figure 3. Néanmoins, il n'est pas aisé de patcher du code généré du fait de sa nature : il suffit qu'une nouvelle version de CamlIDL change par exemple le nom de ses variables dans les *stubs* pour qu'un patch classique ne s'applique plus. Une solution qui répond à ce besoin est l'utilisation d'un outil de **patch sémantique**, en l'occurrence *Coccinelle* [2].

Annexe B : détails sur XDR

Le listing 1.5 reprend l'exemple de structure `ck_date` détaillé pour CamlIDL en Annexe A. L'outil `ocamlrpcgen` se chargera de générer le code de sérialisation et de valider le typage de cette structure lors des communications entre les clients et le serveur. Le fichier XDR est également utilisé pour définir les fonctions distantes disponibles. Dans notre cas il s'agit de l'ensemble des fonctions PKCS#11 et de quelques fonctions propriétaires. Ces dernières sont utilisées pour la détection de l'architecture du client (32/64-bit et *endianness*) et le support du *multi-middlewares*.

```
struct rpc_ck_date {
  opaque rpc_ck_date_year <4>;
  opaque rpc_ck_date_month <2>;
  opaque rpc_ck_date_day <2>;
};
```

Listing 1.5. Définition de `ck_date` dans notre interface XDR

Le listing 1.6 illustre la déclaration de la fonction `C_Encrypt` dans notre fichier d'interface. Le standard XDR spécifie que les fonctions RPC ne peuvent retourner qu'une seule valeur, il est alors nécessaire d'encapsuler les résultats dans des structures lorsque plusieurs arguments de sortie sont nécessaires. C'est le cas dans cet exemple, puisqu'il faut renvoyer le code de retour et les données suite à l'opération de chiffrement.

```
/* Structure en sortie de la fonction RPC C_Encrypt */
struct ck_rv_c_Encrypt {
  rpc_ck_rv_t c_Encrypt_rv;  /* Code de retour */
  opaque c_Encrypt_value <>; /* Donnees de taille variable */
};
...
/* En entrée, un handle de clé, et les données à chiffrer */
ck_rv_c_Encrypt c_Encrypt(rpc_ck_session_handle_t, opaque_data) =
    51;
...
```

Listing 1.6. Définition de `C_Encrypt` dans notre interface XDR

Références

1. Caml Crush. https://github.com/ANSSI-FR/caml-crush.
2. Coccinelle. http://coccinelle.lip6.fr/.
3. Config_file. http://config-file.forge.ocamlcore.org/.
4. CryptokiX. http://secgroup.dais.unive.it/projects/security-apis/cryptokix/.
5. GNOME Keyring. http://live.gnome.org/GnomeKeyring.
6. Marshal. http://caml.inria.fr/pub/docs/manual-ocaml/libref/Marshal.html.
7. OCaml-SSL. https://github.com/savonet/ocaml-ssl.
8. Ocamlnet. http://projects.camlcity.org/projects/ocamlnet.html.
9. openCryptoki. http://sourceforge.net/projects/opencryptoki/.
10. pkcs11-proxy. http://floss.commonit.com/pkcs11-proxy.html.
11. pkcs11.h. ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs11.h.
12. Projet OCaml. http://caml.inria.fr/.
13. SunRPC and XML-RPC over GnuTLS (SSL, TLS) and ssh. http://people.redhat.com/~rjones/secure_rpc/.
14. Tookan. http://secgroup.dais.unive.it/projects/security-apis/tookan/.
15. Sun RPC RFC 1057. http://www.ietf.org/rfc/rfc1057.txt, 1988.
16. XDR RFC 4506. http://tools.ietf.org/html/rfc4506, 2006.
17. Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology - CRYPTO 2012*, pages 608–625, 2012.
18. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In *Advances in Cryptology - CRYPTO '98*, pages 1–12, 1998.
19. Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *ACM Conference on Computer and Communications Security*, pages 260–269. ACM Press, October 2010.
20. Jolyon Clulow. On the security of pkcs#11. In *CHES 2003*, pages 411–425, 2003.
21. Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal security analysis of pkcs#11 and proprietary extensions. *Journal of Computer Security*, 18(6) :1211–1245, 2010.
22. Serge Vaudenay. Security flaws induced by cbc padding - applications to ssl, ipsec, wtls ... In *Advances in Cryptology - EUROCRYPT 2002*, pages 534–546, 2002.