Projet ‘Security for IOT’

Smart Card

Professeur : Yves Roudier

Étudiants :

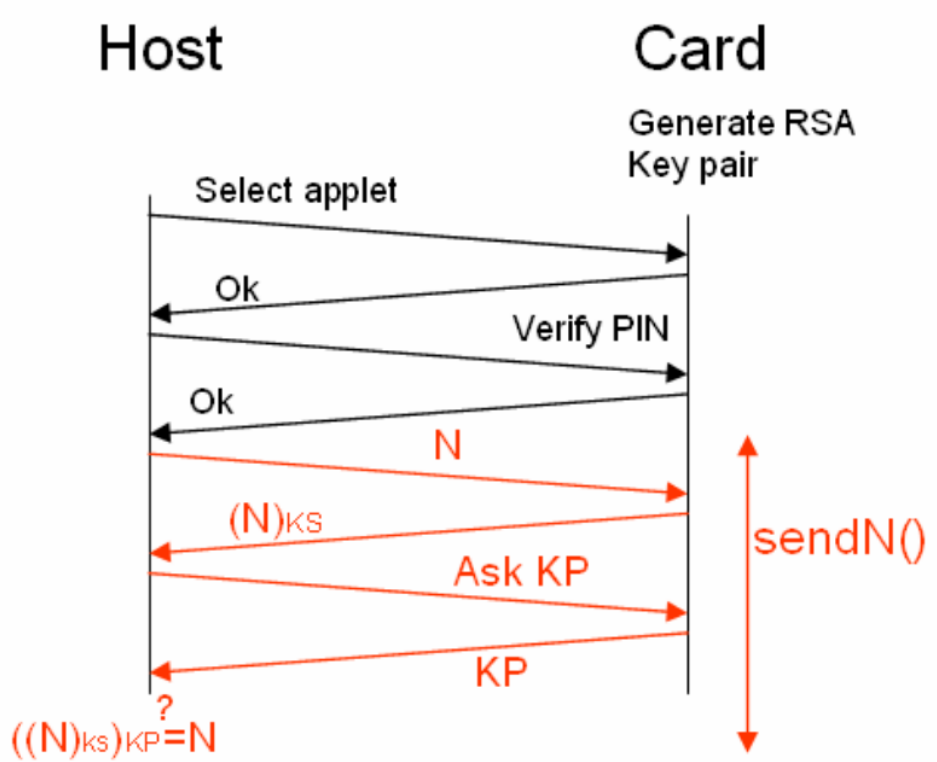
BATISSE Dylann, CARLENS Jean-Philippe, COUSSON Antoine

Introduction

Dans ce projet nous avons travaillé avec des Smart Card et tenté d'effectuer une implémentation d'un processus d'authentification simple à l'aide d'applets Java Card et de signatures RSA. Le but est de pouvoir envoyer du côté client un message qui va être signé par la JavaCard et que l'on pourra vérifier de notre côté grâce à la clé publique de la carte.

Objectifs

1. Protéger la carte à l'aide d'un code PIN.
2. Changer le code pin côté client.
3. Générer une clé de chiffrement RSA 512 bits.
4. Signer une donnée envoyée par un terminal sur demande dans la carte.
5. Renvoyer la clé publique générer par la JavaCard afin de vérifier les signatures.



Technologies utilisées

Nous avons utilisé le framework officiel **Javacard** uniquement du côté de l'applet. Il nous fournit des solutions d'implémentations de **RSA** pour la génération de clés de chiffrement et de **RSA_ALG_SHA_PKCS1** pour les signatures et vérifications.

Du côté du client, nous avons écrit un script python automatisant les intéractions avec la carte. Il utilise le module **pyscard**, un wrapper facilitant les intéractions **APDUs**. Notre script python utilise également **PyCryptodome** afin de reconstruire la clé publique **RSA PKCS1** à partir de l'exposant et du modulus envoyé par la carte.

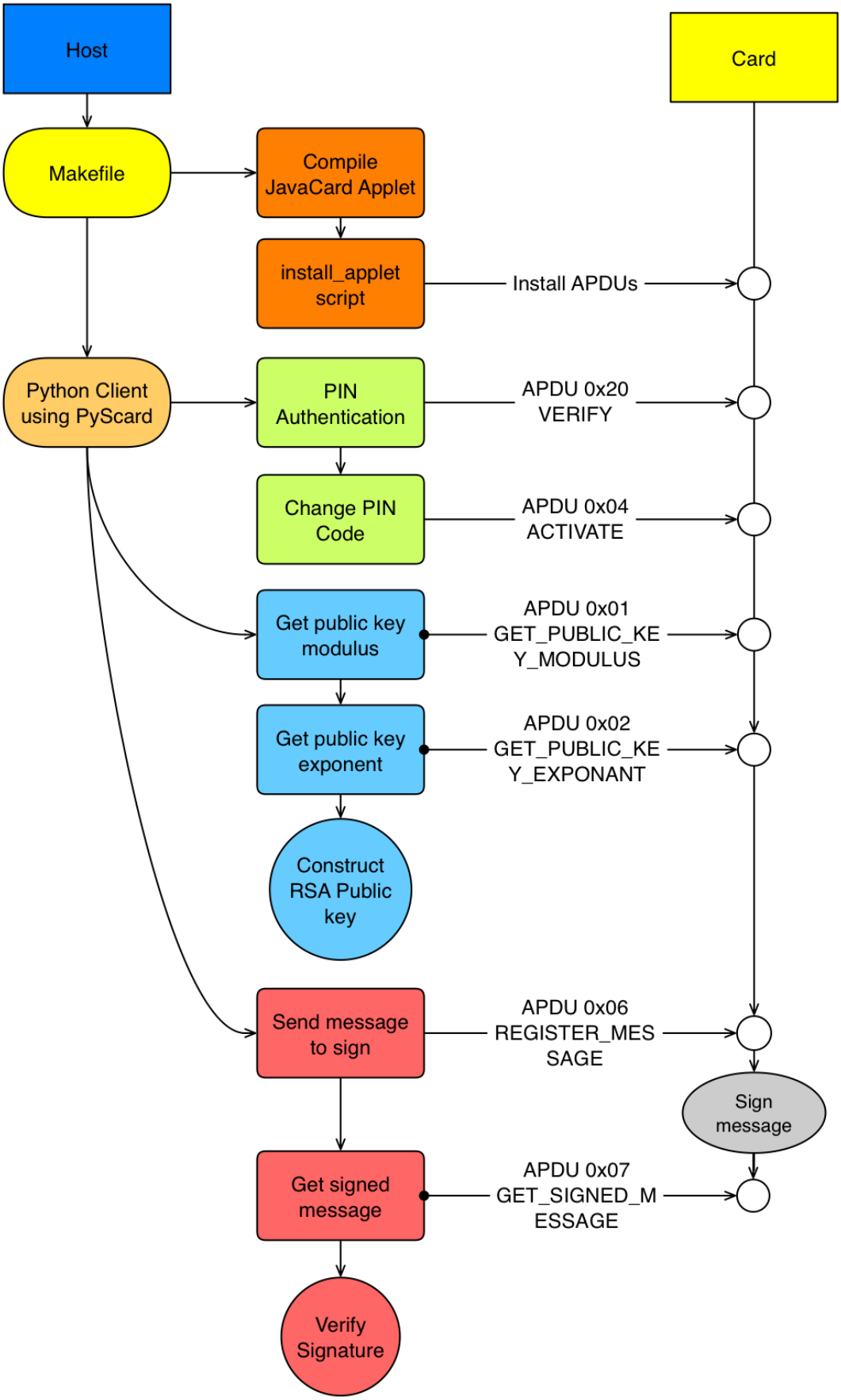
Architecture

Dans la page ci-dessous, nous allons détailler dans quel ordre sont exécutés les différents APDUs enfin d'arriver finalement à la vérification de la signature d'un message par la carte. Les détails des APDU seront aussi donnés dans la partie APDUs.

Nous commençons donc par utiliser un Makefile afin de compiler l'applet JavaCard. Ensuite, nous avons utilisé différents scripts gpshell. Nous avons par exemple un script pour lister les applets, un pour installer la nôtre et un pour désinstaller. Nous allons donc appeler celui d'installation qui qui va communiquer avec la carte pour effectuer le processus.

Ensuite nous utilisons le client Python qui va demander à la carte par l'intermédiaire d'APDUs toujours d'envoyer les paramètres de la clé publique afin de la construire. Ensuite, la vérification du message se fera en deux temps : d'abord nous envoyons le message. Ensuite nous demandons à la carte de renvoyer ce dernier message en le signant. Le client python, disposant de la clé publique de la clé utilisée pour signer peut alors vérifier la signature.

Voir le schéma page suivante :


Host
Card
Makefile
Compile JavaCard Applet
install_applet script
Install APDUs
Python Client using PyScard
PIN Authentication
APDU 0x20 VERIFY
Change PIN Code
APDU 0x04 ACTIVATE
Get public key modulus
APDU 0x01 GET_PUBLIC_KE Y_MODULUS
Get public key exponent
APDU 0x02 GET_PUBLIC_KE Y_EXPONANT
Construct RSA Public key
Send message to sign
APDU 0x06 REGISTER_MES SAGE
Sign message
Get signed message
APDU 0x07 GET_SIGNED_M ESSAGE
Verify Signature

APDUs

Nom de l'APDU	Code INS	Data length	Description	Réponse
HELLO	0x00	0	Demande à l'applet de renvoyer un hello	"Hello" en octets
ACTIVATE	0x04	Entre 4 et 8 octets, le code PIN voulu	Permet de changer le code PIN de l'applet	90 00 (OK)
VERIFY	0x20	Entre 4 et 8 octets, le code PIN à tenter	Permet de tenter une authentification à l'aide du code PIN	90 00 (OK) ou 98 40 (PIN validation not succesfull)
GET_PUBLIC_KEY_MODULUS	0x01	0	Demande d'envoyer le modulus de la clé publique	Le modulus de la clé publique en octets.
GET_PUBLIC_KEY_EXPONENT	0x02	0	Demande d'envoyer l'exposant de la clé publique	L'exposant de la clé publique en octets.
REGISTER_MESSAGE	0x06	Entre 1 et 64 octets, une string	Le message à enregistrer comme "à signer"	90 00 (OK)
GET_SIGNED_MESSAGE	0x07	0	Demande à la carte la signature du message précédemment enregistré	Signature du message en octets.

Le code PIN par défaut vaut "**0123**". Seul l'APDU ***VERIFY*** ne nécessite pas d'être authentifié à l'aide du code PIN au préalable. Par exemple pour envoyer l'APDU REGISTER_MESSAGE, voici le contenu de la requête :

80 06 00 00 05 48 65 6c 6c 6f

80 est l'octet correspondant au CLA. Il indique que nous voulons accéder à une classe propriétaire, dépendante d'un applet.

06 est le code INS de REGISTER_MESSAGE comme indiqué dans le tableau ci-dessus.

00 00 sont P1 et P2, ils définissent des offset qui ne nous sont pas nécessaires.

05 est la taille de la donnée envoyée, ici 5 car "Hello" à 5 caractères.

48 65 6c 6c 6f est "Hello" en hexadécimal.

Applet Java

Tout notre applet réside dans une seule classe. Nous avons en attribut chaques valeurs INS en byte, l'état booléen de l'authentification PIN ainsi que les clés de chiffrement.

Le constructeur de l'applet crée un code PIN par défaut (0123) et génère la clé privée, publique, et l'instance de signature.

Pour le code PIN nous avons utilisé l'implémentation **javacard.framework.OwnerPIN**. Nous nous sommes basés sur la documentation officielle Oracle. Elle gère déjà tout ce qui est par exemple les tentatives restantes et *reset*.

```
private void activate(APDU apdu) {
    if (!pin.isValidated())
        ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);
    byte[] buffer = apdu.getBuffer();
    byte byteRead = (byte) (apdu.setIncomingAndReceive());
    pin.update(buffer, ISO7816.OFFSET_CDATA, byteRead);
}
```

La fonction **process** récupère les APDUs. Elle va ensuite parser l'octet INS de l'APDU afin de pouvoir appeler les fonctions gérant chacune d'entre elles. Si l'INS est inconnu, une erreur **INS_NOT_SUPPORTED** est renvoyée.

Ensuite, chaque fonction reçoit l'objet APDU contenant les diverses données de la requête. Pour lire les données, il faut récupérer le buffer à l'aide de **apdu.getBuffer()**, qui renvoie un tableau d'octets. Les données de l'utilisateur commencent donc à partir de l'index **ISO7816.OFFSET_CDATA**. Pour écrire et donc envoyer une réponse, il faut également écrire

dans le buffer reçu à l'aide de **apdu.getBuffer()** puis soumettre la réponse à l'aide de **apdu.setOutgoingAndSend()**, dont il faut bien spécifier la taille des données que nous envoyons.

L'implémentation **javacard.security.Signature** permet de facilement signer des strings. Elle possède une méthode sign() qui prend un buffer source en argument, à signer, et un buffer destinataire dans lequel écrire la signature. Elle retourne la taille du tableau d'octet signé. Nous prenons donc la string à signer, la signons, puis Il suffit ensuite de simplement envoyer en réponse la signature comme expliqué au-dessus.

Voici par exemple le code de l'APDU signant un string :

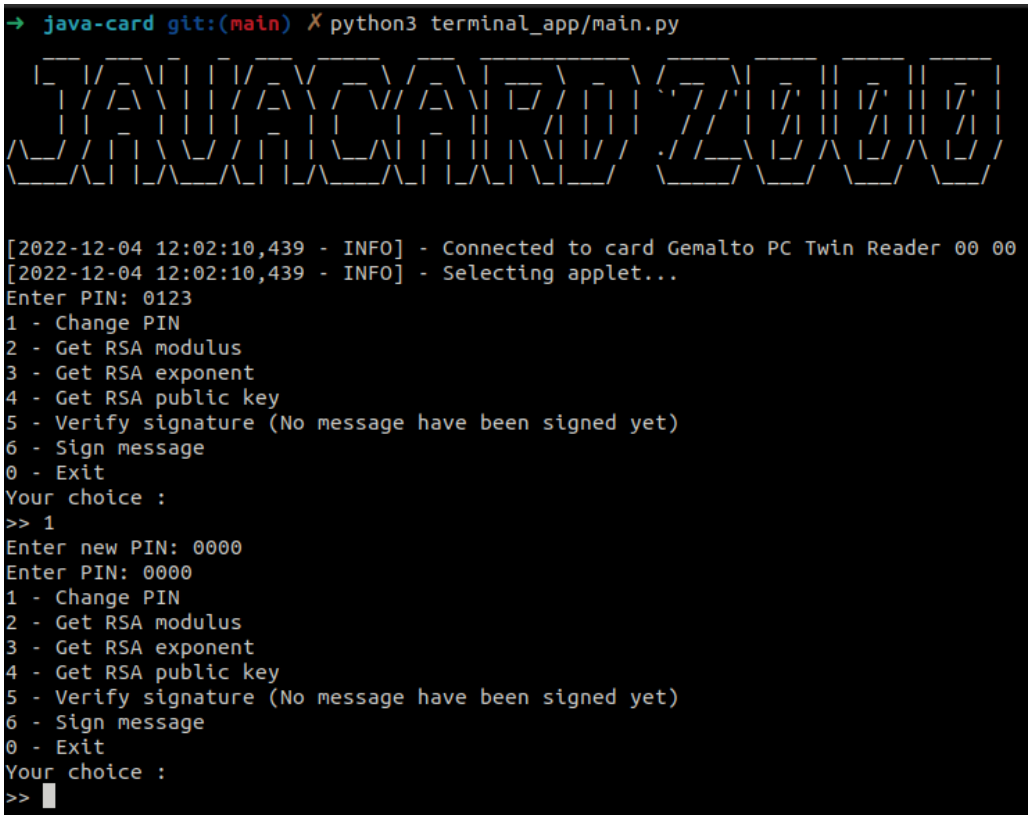
```
private void sendSignedRegisteredMessage(APDU apdu) {
    if (!pin.isValidated())
        ISOException.throwIt(SW_PIN_VERIFICATION_REQUIRED);

    byte[] buffer = apdu.getBuffer();
    short numBytes = (short) currentMessageToSign.length;
    Util.arrayCopyNonAtomic(currentMessageToSign, (short) 0, buffer, (short) 0, numBytes);

    short length = m_signature.sign(
            currentMessageToSign,
            (short) 0,
            (short) numBytes,
            buffer,
            (short) 0) ;

    apdu.setOutgoingAndSend((short) 0, length);
}
```

Script python client

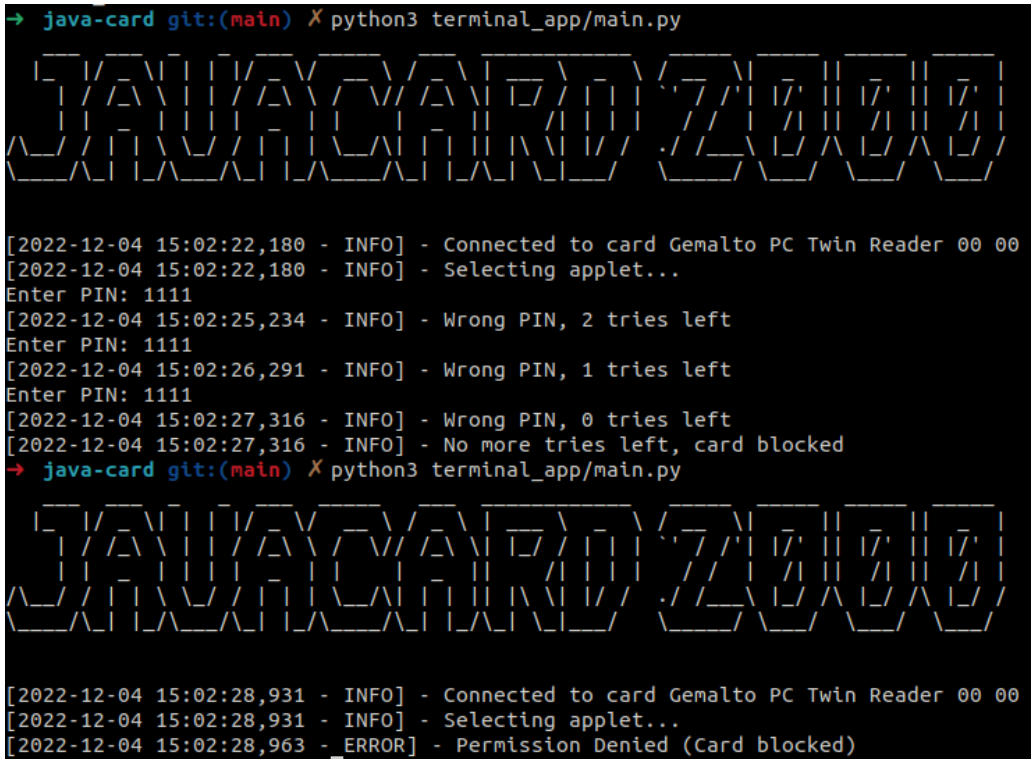
L'application python se présente sous la forme d'un terminal interactif à choix multiples.
Une fois la carte insérée, le pin propre à la carte devra être saisi pour la déverrouiller et accéder aux services de la carte.

Il est possible de demander la clé public de la carte,
De signer un message et de vérifier la signature de ce message,

La taille limite des messages pouvant être signé est de 128 caractères pour des raisons matérielles, en effet la RAM de la carte étant de 128 bytes si nous essayons d'envoyer un message de cette taille il ne sera pas transmis entièrement à la carte et elle renverra un code d'erreur 6F 00



```
→ java-card git:(main) ✗ python3 terminal_app/main.py
 ___  ___  _   _  ___  _____  ___  _____________  _____  _____  _____  _____
|_  |/ _ \| | | |/ _ \/  __ \ / _ \ | ___ \  _  \ /  __ \|  _  ||  _  ||  _  |
  | / /_\ \ | | / /_\ \ /  \// /_\ \| |_/ / | | | `' / /'| |/' || |/' || |/' |
  | |  _  | | | |  _  | |    |  _  ||    /| | | |   / /  |  /| ||  /| ||  /| |
/\__/ / | | \ \_/ / | | | \__/\| | | || |\ \| |/ /  ./ /___\ |_/ /\ |_/ /\ |_/ /
\____/\_| |_/\___/\_| |_/\____/\_| |_/\_| \_|___/   \_____/ \___/  \___/  \___/


[2022-12-04 12:02:10,439 - INFO] - Connected to card Gemalto PC Twin Reader 00 00
[2022-12-04 12:02:10,439 - INFO] - Selecting applet...
Enter PIN: 0123
1 - Change PIN
2 - Get RSA modulus
3 - Get RSA exponent
4 - Get RSA public key
5 - Verify signature (No message have been signed yet)
6 - Sign message
0 - Exit
Your choice :
>> 1
Enter new PIN: 0000
Enter PIN: 0000
1 - Change PIN
2 - Get RSA modulus
3 - Get RSA exponent
4 - Get RSA public key
5 - Verify signature (No message have been signed yet)
6 - Sign message
0 - Exit
Your choice :
>>
```



```
→ java-card git:(main) ✗ python3 terminal_app/main.py
 ___  ___  _   _  ___  _____  ___  _____________  _____  _____  _____  _____
|_  |/ _ \| | | |/ _ \/  __ \ / _ \ | ___ \  _  \ /  __ \|  _  ||  _  ||  _  |
  | / /_\ \ | | / /_\ \ /  \// /_\ \| |_/ / | | | `' / /'| |/' || |/' || |/' |
  | |  _  | | | |  _  | |    |  _  ||    /| | | |   / /  |  /| ||  /| ||  /| |
/\__/ / | | \ \_/ / | | | \__/\| | | || |\ \| |/ /  ./ /___\ |_/ /\ |_/ /\ |_/ /
\____/\_| |_/\___/\_| |_/\____/\_| |_/\_| \_|___/   \_____/ \___/  \___/  \___/


[2022-12-04 15:02:22,180 - INFO] - Connected to card Gemalto PC Twin Reader 00 00
[2022-12-04 15:02:22,180 - INFO] - Selecting applet...
Enter PIN: 1111
[2022-12-04 15:02:25,234 - INFO] - Wrong PIN, 2 tries left
Enter PIN: 1111
[2022-12-04 15:02:26,291 - INFO] - Wrong PIN, 1 tries left
Enter PIN: 1111
[2022-12-04 15:02:27,316 - INFO] - Wrong PIN, 0 tries left
[2022-12-04 15:02:27,316 - INFO] - No more tries left, card blocked
→ java-card git:(main) ✗ python3 terminal_app/main.py
 ___  ___  _   _  ___  _____  ___  _____________  _____  _____  _____  _____
|_  |/ _ \| | | |/ _ \/  __ \ / _ \ | ___ \  _  \ /  __ \|  _  ||  _  ||  _  |
  | / /_\ \ | | / /_\ \ /  \// /_\ \| |_/ / | | | `' / /'| |/' || |/' || |/' |
  | |  _  | | | |  _  | |    |  _  ||    /| | | |   / /  |  /| ||  /| ||  /| |
/\__/ / | | \ \_/ / | | | \__/\| | | || |\ \| |/ /  ./ /___\ |_/ /\ |_/ /\ |_/ /
\____/\_| |_/\___/\_| |_/\____/\_| |_/\_| \_|___/   \_____/ \___/  \___/  \___/


[2022-12-04 15:02:28,931 - INFO] - Connected to card Gemalto PC Twin Reader 00 00
[2022-12-04 15:02:28,931 - INFO] - Selecting applet...
[2022-12-04 15:02:28,963 - ERROR] - Permission Denied (Card blocked)
```

```
1 - Change PIN
2 - Get RSA modulus
3 - Get RSA exponent
4 - Get RSA public key
5 - Verify signature (No message have been signed yet)
6 - Sign message
0 - Exit
Your choice :
>> 6
Message to sign: This message will be signed
[2022-12-04 14:14:27,930 - INFO] - Signature from SmartCard: 39d461c0f4b32a074495e66d1b12f600ac0b0ddb28c0
088680b0eea68dcc30cf1711091969f837b145d292d60a7d5eed8f91e8d240a64a3114b96f0e9c4c4650
1 - Change PIN
2 - Get RSA modulus
3 - Get RSA exponent
4 - Get RSA public key
5 - Verify signature (Last signature: 39d461c0f4b32a074495e66d1b12f600ac0b0ddb28c0088680b0eea68dcc30cf171
1091969f837b145d292d60a7d5eed8f91e8d240a64a3114b96f0e9c4c4650)
6 - Sign message
0 - Exit
Your choice :
>> 5
Message to verify: This message will be signed
[2022-12-04 14:14:32,246 - INFO] - Checking "This message will be signed" signature
[2022-12-04 14:14:32,246 - INFO] - No public key loaded, asking for RSA public key to SmartCard...
[2022-12-04 14:14:32,287 - INFO] - The signature is valid.
1 - Change PIN
2 - Get RSA modulus
3 - Get RSA exponent
4 - Get RSA public key
5 - Verify signature (Last signature: 39d461c0f4b32a074495e66d1b12f600ac0b0ddb28c0088680b0eea68dcc30cf171
1091969f837b145d292d60a7d5eed8f91e8d240a64a3114b96f0e9c4c4650)
6 - Sign message
0 - Exit
Your choice :
>>
```

Ici nous pouvons voir que nous envoyons le message “This message will be signed” à la Java Card pour qu’elle le signe de son côté avec sa clé privée. Une fois le message signé, la Java Card nous renvoie la signature du message que l’on va pouvoir ensuite vérifier.

Le client n’a pas encore construit la clé publique pour pouvoir vérifier la signature donc une demande à la Java Card est effectué pour récupérer le module et l’exposant pour ainsi la reconstruire.

Nous vérifions ensuite la signature du message en utilisant les algorithmes de cryptographie utilisés par la JavaCard avec RSA et SHA-1.

```
>> 4
-----BEGIN PUBLIC KEY-----
MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAL2mzuckZsi3xdhf+iM8YSeEZVJliL3s
guRSeOB2FotVDBtD2R10ICS222kgjzFGVChQ6xBRcjDrh+Yy38iavgECAwEAAQ==
-----END PUBLIC KEY-----
```

Conclusion

Nous avons réussi à mener à bien ce projet, nous avons découvert la technologie Java Card, pris en main les APDUs. Ce projet à été intéressant et différent de nos projets précédemment réalisés.

En effet nous avons pu implémenter de manière concrète sur des outils originaux les mécanismes de cryptographie que nous avons étudiés en cours. La facilité d'accès nous a agréablement surpris et nous réfléchissons même à essayer d'acheter nos propres JavaCard, si possible dans des versions plus récentes pour essayer de nous amuser avec (Essayer de refaire un semblant de YubiKey ?). Nous avons pu mêler IoT et sécurité.

Annexe

GitHub : https://github.com/Smart-Card-TEAM/java-card