



Travaux pratiques

JAVA CARD

TP n°0 – Utilisation de la cryptographie : Réalisation d'une applet Java Card et du client

TP n°1 – Application de fidélité

TP n°2 – Utilisation de la cryptographie et du multi-canal

TP n°3 – Implémentation d'une confusion de type

TP n°4 – Attaques logiques détournement du flot de contrôle

TP n°5 – Récupérer les adresses de l'API pour faire un shell code riche

TP n°6 – Applet mal typée chargée malgré le vérifieur de BC

Nous utiliserons dans cette série de TP l'environnement Eclipse avec le plugin JCOP déjà activé.

Je rappelle le fondement de Java, il n'y a pas d'arithmétique sur les types autres que numériques (et en Java Card sur les entiers). Pensez aux règles de base de génie logiciel, pas de constantes en dur dans le code, nommer correctement vos variables, etc.

Pour tous les TP, nous utiliserons des JCOP.

Pour charger dans la carte, on utilisera suivant les TP et les besoins soit Eclipse directement soit GPShell, soit un programme Java faisant office de terminal. Ce dernier utilise la librairie Opal qui est une implémentation de GP. Vous pouvez charger, télécharger et exécuter des commandes depuis un monde Java.

Procédure de création d'une application :

Donner un nom à votre projet, sélectionner les profils de votre carte (en général JC2.1.1 et OP2.0.1), à votre package et à votre applet. Donner également un AID à votre package et à l'applet (en le dérivant de celui du package).

Écrire le programme Java correspondant à la spécification. Pour ce faire, créer un projet Java Card (un projet par package). Le nom du projet et/ou des packages et applets n'ont pas de réelle importance c'est plus pour vous y retrouver.

Une fois l'application codée, vous la chargez sur la carte au travers du Run Configuration. Double cliquez sur Java Card Application et sélectionner le lecteur de carte comme destination et en avant « Run ».

À l'aide du shell de JCOP vous pouvez échanger avec la carte via des commandes prédéfinies ou directement vos APDU. Vérifiez le bon fonctionnement de votre application.

Si vous rencontrez des erreurs n'hésitez pas à me les signaler :

damien.sauveron@unilim.fr

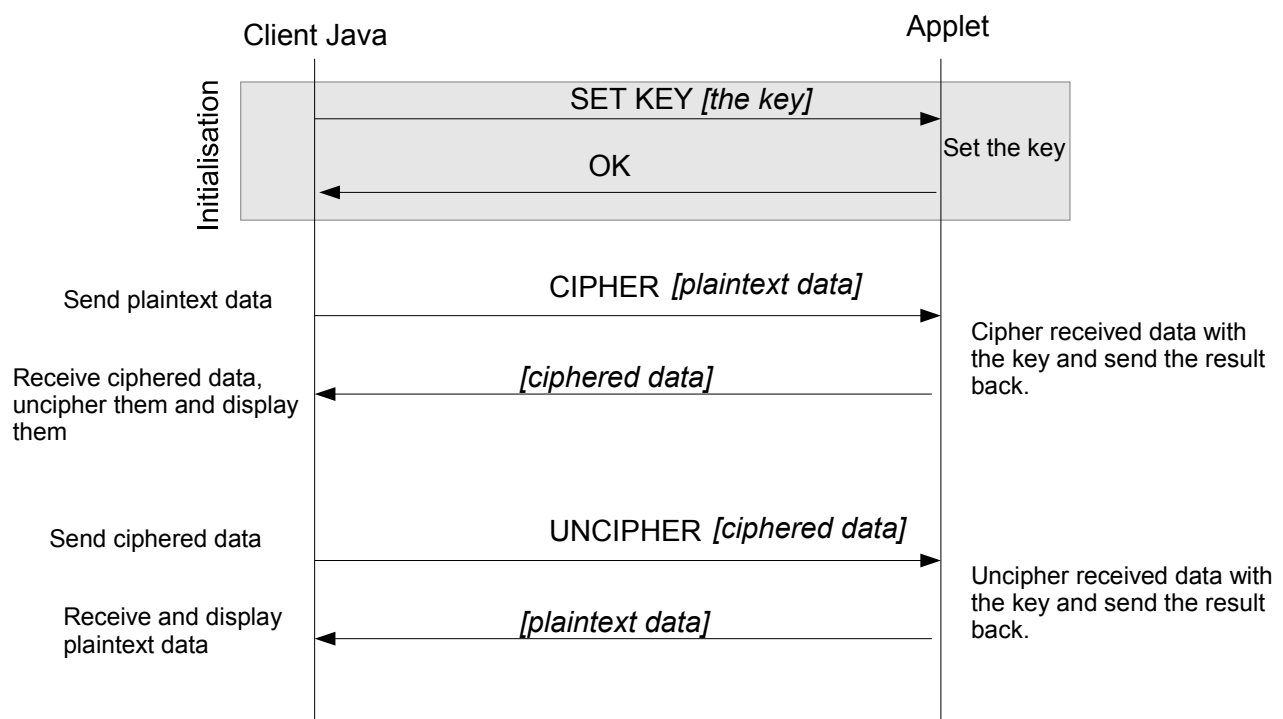
TP n°0 – Utilisation de la cryptographie : Réalisation d'une applet Java Card et du client

Objectif : Manipulation d'un algorithme de chiffrement coté Java Card et intégration dans un client Java.

Dans ce TP nous allons utiliser les primitives cryptographiques pour réaliser le chiffrement et le déchiffrement de messages par la carte. Pour cela nous implémenterons un protocole simple dans lequel la carte et le client possèdent une clé symétrique 3 DES de 192 bits qui sera utilisée pour chiffrer et déchiffrer des messages de 8 octets en utilisant l'algorithme ALG_DES_CBC_NOPAD. Cet algorithme utilise la clé de 192 bits et travaille sur des blocs de 64 bits avec un vecteur d'initialisation de même taille que les blocs.

1) Écrire un client Java afin qu'il puisse :

- initialiser l'application avec la clé symétrique utilisée.
- envoyer une donnée de 8 octets en clair à l'applet. Cette donnée sera chiffrée par l'applet et renvoyée au terminal qui la déchiffre et l'affiche.
- envoyer une donnée chiffrée à l'applet qui la déchiffre et retourne la donnée en clair. Cette donnée sera affichée par le terminal.



a) Il faut tout d'abord définir les commandes APDU nécessaires. On peut par exemple utiliser des constantes pour les différentes instructions (respectivement dans l'ordre du schéma 0x20, 0x30 et 0x40).

b) Il faut également importer les packages suivants :

```
import javax.crypto.*;
import javax.crypto.spec.DESedeKeySpec;
import javax.crypto.spec.IvParameterSpec;
```

c) Il faut déclarer un tableau d'octets qui a la taille de la clé et qui contient la valeur de la clé secrète.

d) Il faut déclarer le vecteur d'initialisation initialisé à 0 car l'algorithme du 3DES côté carte repose sur un vecteur d'initialisation qui est toujours égal à 0 à cause de l'espace mémoire limité.

e) Avant les opérations de chiffrement ou de déchiffrement on réalisera les étapes suivantes :

- créer un objet DESedeKeySpec correspondant à la clé secrète qui a été définie,
- créer une fabrique de clés secrètes avec la classe SecretKeyFactory pour l'algorithme 3DES nommé ici "DESede" pour DES Encryption-Decryption-Encryption,
- générer la clé secrète avec la méthode generateSecret(),
- créer un objet de type IvParameterSpec correspondant au vecteur d'initialisation,
- créer un objet de type Cipher en précisant la variante d'algorithme 3DES qui sera utilisé ici, à savoir "DESede/CBC/NoPadding", c'est-à-dire le DESede utilisant un découpage de messages en blocs (Cipher Block Chaining, ici de 64 bits) sans bits de bourrage (No padding).

```
// Prepare key
// Create an array to hold the key
byte[] baKey = { (byte)0x00, (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x05, (byte)0x06,
(byte)0x07, (byte)0x00, (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07,
(byte)0x00, (byte)0x01, (byte)0x02, (byte)0x03, (byte)0x04, (byte)0x05, (byte)0x06, (byte)0x07, };

// Create a DESede key spec from the key
DESedeKeySpec keySpec = new DESedeKeySpec(baKey);

// Get the secret key factory for generating DESede keys
SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DESede");

// Generate a DESede SecretKey object
SecretKey theKey = keyFactory.generateSecret(keySpec);

// Create an initialization vector (necessary for CBC mode)
byte[] baIV = { (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00, (byte)0x00,
(byte)0x00, };
IvParameterSpec iv = new IvParameterSpec(baIV);

Cipher cipher = Cipher.getInstance("DESede/CBC/NoPadding");
```

f) Ensuite pour effectuer une opération de chiffrement ou déchiffrement, on initialisera l'objet Cipher en précisant l'opération à effectuer. L'initialisation nécessite 3 paramètres :

- nature de l'opération : la constante Cipher.ENCRYPT_MODE si chiffrement ou Cipher.DECRYPT_MODE si déchiffrement.
- la clé secrète
- et le vecteur d'initialisation.

g) L'opération de chiffrement/déchiffrement proprement dite est réalisée grâce à la méthode doFinal().

2) Écrire l'applet Java Card afin qu'elle puisse traiter les commandes APDU envoyées par le terminal.

a) Il faut importer les packages suivants :

```
import javacard.security.DESEKey;
import javacard.security.KeyBuilder;
import javacardx.crypto.Cipher;
```

b) Il faut créer les différents objets nécessaires dans le constructeur de l'applet, soit :

- un objet DESKey via l'utilisation de KeyBuilder.buildKey() avec comme paramètres KeyBuilder.TYPE_DES, KeyBuilder.LENGTH_DES3_3KEY et false (car on ne souhaite pas que la clé soit chiffrée en mémoire).
- un objet Cipher qui implémente l'algorithme souhaité. Pour préciser qu'il s'agit d'un chiffrement/déchiffrement avec l'algorithme 3DES de type CBC No padding on utilisera la constante Cipher.ALG_DES_CBC_NOPAD et d'un paramètre mis à false (car on ne souhaite pas partager cet instance au travers du firewall).

c) Selon la commande APDU reçu :

- la donnée reçue pourra être utilisée comme clé via la méthode setKey(clé, offset)
- la donnée reçue sera chiffrée ou déchiffrée avant d'être renvoyée au terminal. La méthode init() de l'objet Cipher permet de fixer l'opération à effectuer :
 - chiffrement si le deuxième paramètre est égal à Cipher.MODE_ENCRYPT
 - déchiffrement si le deuxième paramètre est égal à Cipher.MODE_DECRYPT.

Le premier paramètre étant l'objet DESKey correspondant à la clé secrète.

On notera que le chiffrement/déchiffrement est effectivement réalisée à l'aide de la méthode doFinal() de l'objet Cipher.

TP n°1 – Application de fidélité

Objectif : Connaissance des mécanismes de partages d'objet en Java Card et mise en place de contrôle d'accès dynamique.

Dans le cas de cette application de fidélité, un des nos partenaire sera le porte-monnaie électronique développé précédemment. Les opérations de débit ou de crédit via le porte-monnaie donneront lieu à une récompense sous forme de points de fidélité à raison de 1 point par tranche de 10 euros (crédités ou débités), le nombre de points étant calculé par le client.

Les spécifications de cette application de fidélité sont les suivantes :

- Elle comportera un compteur de points exprimé sous forme d'un `short` qui devra toujours rester positive et qui ne dépassera pas la limite maximal du format de stockage.
- Elle comportera un PIN qui devra être vérifié pour les opérations de reward (récompense) et de `use_points` (pour utiliser des points). On utilisera pour ce PIN la classe `OwnerPIN` qui fait tout le travail de gestion du PIN pour nous et de manière sécurisée. Nous le fixerons à 8 octets la longueur et à 3 le nombre maximal d'essai consécutifs faux. La valeur du PIN sera fixée lors de l'opération d'installation afin de pouvoir le personnaliser pour chaque carte.
- Elle acceptera 4 commandes `GET_POINTS`, `REWARD`, `USE_POINTS` et `VERIFY` :
 - `GET_POINTS` renverra le solde ;

GET POINTS APDU command						
Command APDU						
CLA	INS	P1	P2	Lc	Data field	Le
0x00	0x50	0x0	0x0	N/A	N/A	2
Response APDU						
Optional data		Status word		Meaning of status word		
Amount of points value		0x9000		Successful processing		

- `VERIFY` permettra de proposer un PIN à vérifier ;

VERIFY APDU command						
Command APDU						
CLA	INS	P1	P2	Lc	Data field	Le
0x00	0x20	0x0	0x0	Length of the PIN data	PIN data	N/A
• CLA byte denotes the structure of the command • INS byte (0x20) indicates a verify instruction • P1 and P2 are not used, and are both set to 0 • The data field contains the PIN value						
Optional data		Status word		Meaning of status word		
N/A		0x9000		Successful processing		
		0x6300		Verification failed		

- REWARD permettra de créditer des points pour des occasions particulières (anniversaire, etc.) et ne fonctionnera que si le PIN a été vérifié avec succès ;

REWARD APDU command						
Command APDU						
CLA	INS	P1	P2	Lc	Data field	Le
0x00	0x30	0x0	0x0	2	Amount of points	N/A
The data field contains the amount of points						
Response APDU						
Optional data		Status word		Meaning of status word		
N/A		0x9000		Successful processing		
		0x6301		PIN verification required		
		0x6A83		Invalid amount		
		0x6A84		Exceed the maximum amount		

- USE_POINTS permettra de débiter des points (pour les transformer en cadeau) et ne fonctionnera que si le PIN a été vérifié avec succès.

USE_POINTS APDU command						
Command APDU						
CLA	INS	P1	P2	Lc	Data field	Le
0x00	0x40	0x0	0x0	2	Amount of points	N/A
The data field contains the amount of points						
Response APDU						
Optional data		Status word		Meaning of status word		
N/A		0x9000		Successful processing		
		0x6301		PIN verification required		
		0x6A83		Invalid amount		
		0x6A85		Negative balance		

- SET_FRIEND permettra d'informer l'application de l'AID d'une application partenaire et elle ne fonctionnera que si le PIN a été vérifié avec succès ;

SET_FRIEND APDU command						
Command APDU						
CLA	INS	P1	P2	Lc	Data field	Le
0x00	0x10	0x0	0x0	Length of the AID data	AID data	N/A
The data field contains the AID of the partnership application						
Response APDU						
Optional data		Status word		Meaning of status word		
N/A		0x9000		Successful processing		
		0x6301		PIN verification required		
		0x6A87		Partner is not present		

Si cette application est aussi simple que le porte-monnaie électronique, elle doit tout de même exposer une méthode grantPoints(short points) afin de permettre à l'application cliente (le porte-monnaie) de pouvoir ajouter des points. Il faut modifier le porte-monnaie de façon adéquate pour lui permettre de récupérer l'interface partageable.

Pour des raisons de simplicité, si l'ajout de points n'a pas pu se faire parce que le maximum était atteint, aucun SW d'erreur ne sera remonté dans le porte-monnaie initial. On passera l'AID de l'application de fidélité dans les données d'installation. Si celle-ci n'est pas préalablement présente, le porte-monnaie refusera de s'installer et un SW d'erreur sera levé.

Afin de s'assurer que l'appel à la méthode grantPoints(short points) ne peut provenir que de l'application cliente ou de l'application de fidélité elle-même, des vérifications seront ajoutées.

L'objectif de ces vérifications est de s'assurer qu'une application tierce qui aurait récupéré l'objet ne pourra pas en faire un mauvais usage.

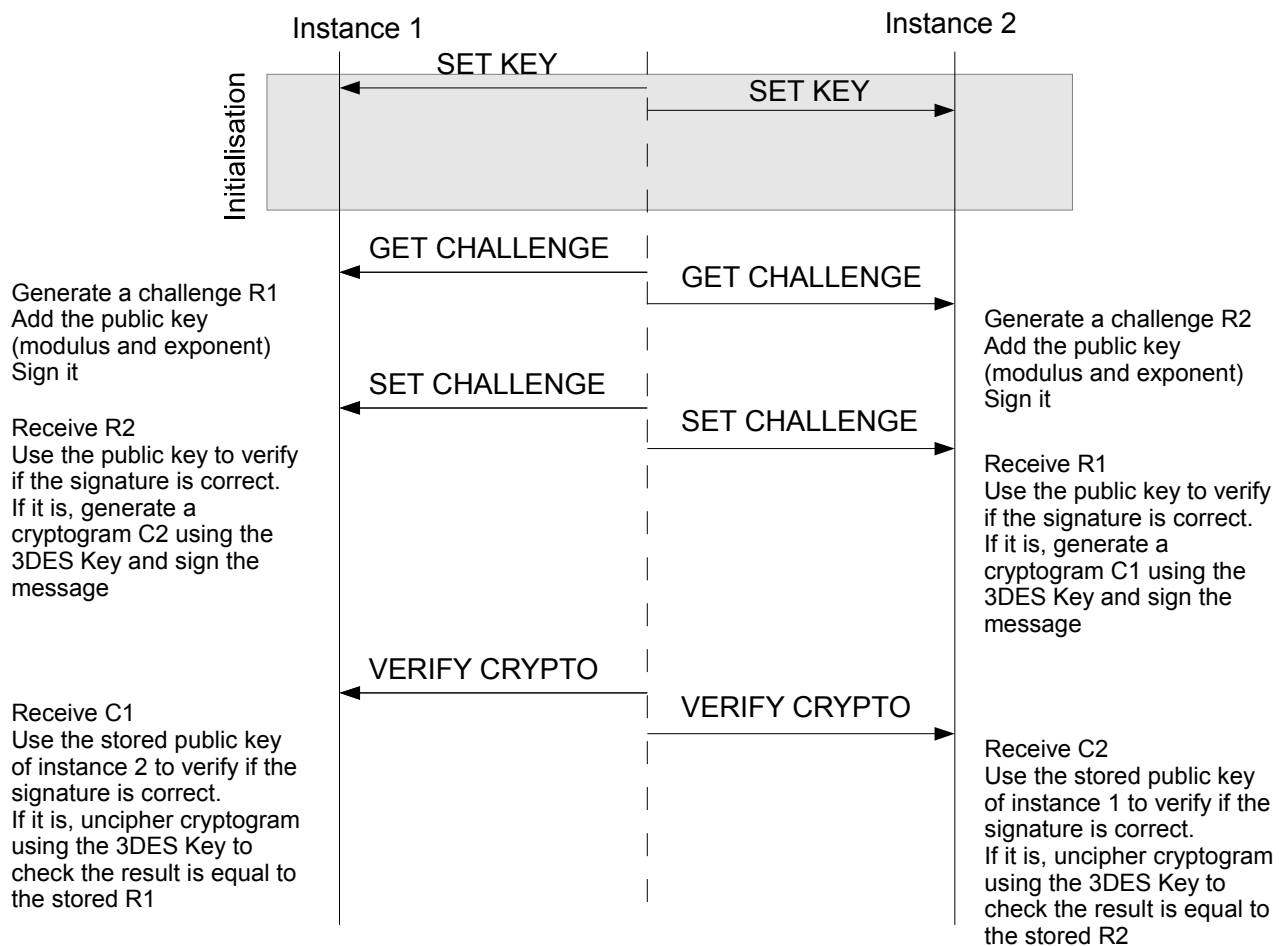
TP n°2 – Utilisation de la cryptographie et du multi-canal

Objectif : Manipulation des primitives de cryptographie et manipulation du multi-canal
L'utilisation de plusieurs instances de la même application en multi-canal nécessite certaines modifications d'une applet classique

Dans ce TP nous allons utiliser les primitives cryptographiques pour réaliser une authentification mutuelle. Pour cela nous implémenterons un protocole *challenge/response* dans lequel au départ les deux parties possèdent une clé symétrique 3 DES de 192 bits qui sera utilisé pour chiffrer les challenges (de 8 octets) en cryptogramme et inversement en utilisant l'algorithme ALG_DES_CBC_NOPAD.

Chaque partie disposera d'un générateur de nombres aléatoires et d'une paire de clés RSA de 512 bits (auto-généré au format CRT) qui sera utilisée pour signer et vérifier les messages avec l'algorithme ALG_RSA_SHA_PKCS1.

Les deux parties seront simulés par 2 instances de la même application.



Utiliser des constantes pour les différentes instructions (respectivement dans l'ordre du schéma 0x10, 0x20, 0x30 et 0x40).

Allez y progressivement : création des différents objets, RNG, RNG+clé pub, RNG+clé pub+signature, ...

À l'issue de l'authentification mutuelle on pourrait poursuivre les échanges en chiffrant avec la clé 3DES et en signant.

Idéalement si nos cartes supportent les canaux logiques, chaque instance utilisera un canal logique différent.

Dans ce contexte, il faudra que l'applet dont on créera 2 instances utilise Java Card 2.2.1 et implémente l'interface `MultiSelectable`.

On ajoutera une commande avec l'INS à 0x50 pour pouvoir récupérer le canal logique afin de faire du debug.

On sélectionnera les applications sur le bon canal avec `/select <AID> numero_du_channel` et les commandes suivantes verront leur CLA impacté par le numéro du channel. Toutefois avec la version du plugin JCOP que nous avons, si la carte est en T=0 et qu'elle doit transmettre des données dans le cas 4 des APDU, le GET RESPONSE sera fait avec un CLA à 0x00 (au lieu de contenir le channel), ce qui pose problème (il s'agit d'un bug – si vous faites les opérations à la main avec TestResMan, tout fonctionnera) !

Nous pouvons donc développer un client Java réalisant les opérations proprement en utilisant `openLogicalChannel()` sur un objet `Card`.

Astuce : Si on ne dispose pas de carte supportant les canaux logiques, on peut ne travailler que sur une seule instance ou encore sélectionner et désélectionner les instances pour travailler alternativement avec chacune d'elle :-)

TP n°3 – Implémentation d'une confusion de type

Objectif : Communication inter-applets via le firewall. Implémentation d'un modèle d'attaque. Teste sur des cartes afin de voir si elle fonctionne ou pas. Réflexion sur l'implémentation des objets sur la carte.

Programmer une application serveur exposant une interface partageable avec une méthode recevant un tableau de short.

```
void accArray(short[] array)
```

Programmer une application client souhaitant utiliser l'interface partageable du serveur avec une méthode recevant un tableau de byte.

```
void accArray(byte[] array)
```

1. Comment faire ?
2. Est-ce que si l'application cliente passe un tableau de byte au client, celui-ci pourra s'en servir et comment faire ?
3. Une fois tous les détails fixés, réalisez la confusion de type et essayez avec le serveur de lire toutes les données du tableau de short reçu alors qu'il s'agit d'un tableau de byte et renvoyez les dans une réponse APDU.
Que constatez vous ? Que faire ?
4. Essayez de passer outre cette limitation avec la solution proposée. Introduisez une classe A dans le premier package et changer l'interface.
Essayez d'exploiter cette nouvelle confusion.
Que constatez vous ? Que faire ?
5. Essayez les solutions proposées et que constatez vous ? Y a-t-il d'autres possibilités ?

TP n°4 – Attaques logiques détournement du flot de contrôle

Objectif : Manipulation du CAP file, détournement du flot d'exécution EMAN2

Description: The attack consists in changing the index of a local variable. The specification says that the number of variables that can be used in a method is 255. It includes local variables, method parameters, and in case of an instance method invocation, a reference to the object on which the instance method is being invoked. For that purpose we use two instructions: `sload` and `sstore`. As described in the JCVM Specification, these instructions are normally used in order to load a short from a local variable and to store a short in a local variable. So, if we change the operand of `sstore`, says `sstore 4`, which stores a short into the Local variable 4. Imagine that your program stores at the Java level a short value into the last local variable plus an offset of 2 (here we assume that this local variable contains the return address – note that it depends of the implementation of cards). It means that we try to store into a local that does not exist. Due to the fact that the Byte Code Verifier checks the range of the locals this must be detected during the conversion process. But after this verification if one changes the value of the operand it will not be detect during run time. If the short value represents the address of the first element of an Array, then this manipulation will change the return address of the current method. When exiting from this method, instead of returning to the caller, the program will execute the byte array. Then the content of the byte array must be interpretable by the virtual machine without any stack under or overflow. We are able to execute an arbitrary shell code, hand written at the byte code level.

Experiments: First of all, you need to set up the environment. You need a compiler chain to be installed with the OPAL library. Use the one provided instead of downloading it at <https://bitbucket.org/ssd/opal/downloads/>. Add this library in your classpath project of your IDE. The Java source files and the compilation scripts are available at the following address: <http://cartes.msi.unilim.fr> Download Applets source code, it contains a zip file applets.zip.

Log into the web service using the user name `user_n` where `n` is the number written on the smart card, the password is the same (example `user_4` for the card labeled 4). In this file, you will find the applets `Applet1`, `Applet2`, `Applet3`, a `helloworld` applet and four client applications to run on your computer for communicating with the smart card. The Java program `Applet1` has to be used with the applet `Applet1` and so on. The program `Simple_OPAL_Main.java` has to be used with the `hello world` applet. In these client programs, you have to modify the line 95 `installApplet (PACKAGE_ID, APPLET_ID, APPLET_ID, "TODO:YOUR_FILE")`; the string by the name of your CAP file (the Java Card binary file format).

The `gencapfile` is the script (`.bat`) for windows platform or (`.sh`) for linux platform. Extract the zip and store it in your computer. These scripts will be used to compile and convert the Java Card Applet. It will generate the CAP files in each applet directory. You have access to the CapMap through a web service at the following address: <http://cartes.msi.unilim.fr>. With this web service you can Upload a CAP file onto the server and read the binary file clicking on Overview. You can edit each method by clicking on Methods. Choose the method you want to edit on the top left using the lift then click on the Ok button. In the Opcodes area you can modify the opcode, and in the Arguments area you can modify the parameters of the opcodes and once you click either on Submit or Enter your modifications are taken into account. If you read your binary file with the Overview button you can see that your modifications have been taken into account, you can download back your modified file using the Download button. Take care if you modify the opcodes or the arguments in a way the program is no more executable (undefined byte code,...) you will raise an exception.

Step 0: checking the compiling and disassembling chain

Compile and convert using the script with a double click on the bat file (either the .bat or .sh file), the simple applet Hello world to verify your tool chain. It generates CAP files in a new directory called Javacard for each applet. Once compiled and converted use the Simple_OPAL_Main.java program to upload it into the card, send the command apdu and verify the response. Normally it should send the "Hello world" byte array to the card which sends back the same byte array. Reaching that point, you have validated your Java Card tool chain and you are ready to try to hack the card.

Possible troubleshooting

- Wrong version of Java, the script is for the latest version of Java says 1.7.0_07 if you use an earlier version, modify the script file. To know your version use `java -version`.
- User of a 64 bits machine using a 32 bits Java: modify in the batch file the address of the jdk: `C:/program Files (x86)/Java/jdk...`
- User of Eclipse, it could not be able to find the API `smartcardio`: in Project properties, in Errors/Warnings click enable project specific settings and then in Deprecated and restricted API, change Forbidden reference from Error to Warning.

Step 1: retrieving the address of an array

You will have to write your shell in an array. Even if it is not mandatory, the first consists in basic stack manipulation using the CapMap web service.

- Edit the Applet1 applet in your favorite Java editor. Find the method `getMyAddress(...)`. Compile it, verify it, convert it. In fact this step has already been done with the script during the previous step but if you modify the source code you have to redo it.
- Edit the resulting CAP file with the CapMap editor,
- Find the right method (one where you find the instruction `sspush 0xCAFE`), nullify the byte code in order to push on top of the stack the reference of the array, recover the CAP file,
- Load the application into the card, send the adequate APDU and obtain the address of the array.

Step 2: perform an EMAN2 attack to retrieve the address

At that point, you have obtained the linked addresses of the API methods, you need to transform a new applet in order to modify the return address of a called method. The return address will be the reference of your array.

- Edit the Applet2 client in your favorite Java editor, it initializes (sendAPDU) your shell code with the minimum to avoid a crash (a code raising the exception 0x6789), it obtains the address of the array, then sends back the address to the card,
- Edit the Applet2 applet in your favorite Java editor. Three commands corresponding to the three APDU : `INS_SET_SHELLCODE` which initializes the shell code (already done in the array declaration), `INS_GET_SHELLCODE_ADDRESS`: return the array address (Warning you have to redo the step 1), `INS_EXECUTE_SHELL_CODE` that patches the card. Look at the method `executeOurShellCode`. The short array_address is the one sent to the client and obtained at step 1. Compile it, verify it, convert it.
- Edit the resulting CAP file and modify the index of the local variable in order to store it at the return address location of the array which is the $(\text{max local} + \text{arg}) + 1$ (in case of doubt ask the instructor, this could be a cause of the card suicide),
- Load the application into the card, execute the client, it sends the three APDU the last one launching your shell code. You are now able to store in the array any shell code you want.

TP n°5 – Attaques du linker

Objectif : Mise en évidence de la non protection des byte codes sensibles

Description: In the previous hand on session you have been able to divert the control flow using the return address. We provide you only one API address: `throwIt()` i.e. `invokestatic 0x08c6`. You will latter have to write linked shell code in the previously specified array. To obtain the addresses of the methods use the API discovering process.

- Edit the Applet3 applet in your favorite Java editor. Look at the methods `getTheISOExceptionThrowIt()` and `getTheAPDUSetOutgoingAndSendAddress()`. Compile it, verify it, convert it.
- Edit the resulting CAP file and remove the tokens in order to push on top of the stack only the addresses that will be resolved by the linker,
- Run the client program Applet3 which sends the APDU to the card store the result. You have obtained the address of the two API methods. If you want other addresses for this card you have just to code the method.

Step 1: execute your rich shell code

Reaching that point you have hooked the card and you are able to execute any arbitrary array in the card. Dump the memory.

- In the array builds a linked application that gets a static value, stores it in a buffer and increments the address of the static.
- When the buffer of 250 byte is full, store it in the APDU buffer, send the response, build the adequate structure to dump the memory from the address xxx to yyy,
- Run the external program xxxx which sends the APDUs to the card and stores the result.

Step 2: read the content of the array

Great you have dump the Java Card memory!!! Now pay attention to the structure. Use an hexadecimal editor and find the array stored at previous hand on session one. What can you remark?

Find the countermeasure...

Step 3: find our own linked code in the memory

Now you know that addresses are scrambled when push on top of the stack. It does not matter the VM descrambled them for you for free. Nevertheless each time something is easy to do shall be protected in the card. Can you reverse a code?

Use an hexadecimal editor and find our own code.

Look at the structure of the stored byte array far from what you sent to card?

At that point you have learn how to dump the memory of the card, you saw that smart card manufacturers try to hide information but you can bypass it, you are able to find applet structures and modify other binary applications by removing, adding code. Keep in mind you have played with development card not a product. Do you believe a product is far from that? What you can do with the shell code is just limited to your imagination. Enjoy!

TP n°6 – Applet mal typée et acceptée

Objectif : Attaque mixte nécessitant pour la contrer une pile typée mais bien typée !

Description: The EMAN2 attack aims at modifying the function's return address upon an instruction which accesses to the frame's header.

On the step 2 of TP n°4, we succeeded this attack with an applet not compliant with the Java Card security rules. A verification with BCV detects this no authorized behavior and the application will be refused.

To bypass the BCV, a way is to modified the application inside the card by a laser beam injection. For this hand on session, the effect of a laser beam is simulated. An applet, checked by the off-card BCV, is modified with CapMap. Let us focus on the Applet4 applet. This applet shall not be recompiled. You should use the provided CAP file.

- The method 3 contains the couple `jsr/ret` instructions and a piece of code located after the return instruction. From the method CFG, there is no path to access to the instructions put after the return instruction. This fragment of code is ill-typed and is not verified by the BCV;
- Check the Applet4 with the off-card BCV with the command:
 - (windows): `verifyApplet.bat applet4/javacard/applet4_jsr.cap`
 - (linux): `verifyApplet.sh applet4/javacard/applet4_jsr.cap`
- Simulate a fault in the `if_scmpeq_w` instruction's parameter with CapMap. The widely used fault model allows to set, reset or change a precise byte to a random value. Here, assume that the `if_scmpeq_w` operand can be modified to `0x0005` instead of `0xFF05`.
- Install and execute this code on the card. Keep in mind that the instruction `if_scmpeq_w 0xFF05` comes the card to an infinite loop.

With this manipulation, we succeeded in bypassing the offcard BCV. Pointed out by Stata et al. [1], an unreachable code is not checked by the Java BCV. The Java Card BCV has the same behavior. Instead of the Java platform, an application can be modified by an external fault injected on the smart card's chip. As effect, a code, different from the installed one, may be executed.

[1] R. Stata and M. Abadi, "A type system for java bytecode subroutines," ACM Trans. Program. Lang. Syst., vol. 21, no. 1, pp. 90–137, 1999.