AILLAUD Dawson - CLEMENCE Mathéo
HUGUET Théo (et TALHA Adam)

UVSQ université PARIS-SACLAY
IUT de Vélizy-Rambouillet
CAMPUS DE VÉLIZY-VILLACOUBLAY

Compte rendu de SAE :
La carotte électronique

https://github.com/theohuguet01/RT5SA01A

RT3-FA 2025-2026

1) Présentation générale et fonctionnement détaillé (Rubrovitamin)

Rubrovitamin est le **logiciel embarqué de la carte à puce** du système. Il implémente le protocole **ISO 7816 (T=0)** et gère l'ensemble des données sensibles stockées sur la carte, telles que la personnalisation, le PIN/PUK, le solde et le compteur anti-rejeu. Il assure la **sécurité des transactions** grâce à des mécanismes d'authentification, d'anti-rejoue et de protection contre l'arrachement lors des écritures en EEPROM.

1.1) Architecture globale du programme

Le code se structure en 4 "**blocs**" principaux :

1. **Communication bas niveau** (T=0)

- recbytet0() : reçoit un octet depuis l'interface carte <-> lecteur
- sendbytet0() : envoie un octet au lecteur
  Ces fonctions sont externes et écrites en assembleur (fichier **io.s**).

2. **Gestion du protocole carte**

- Envoi de l'**ATR** au reset.
- Lecture des APDUs : 5 octets (CLA INS P1 P2 P3)

3. **Fonctions applicatives**

- Version / personnalisation (CLA 0x81)
- Sécurité / PIN / PUK / compteur / solde (CLA 0x82)

4. **Persistance et robustesse**

- EEPROM pour stocker perso, PIN/PUK, compteurs, solde.
- Mécanisme "anti-arrachement" par journal de transaction (engage() / valide()).
- Mécanisme "anti-rejoue" par un compteur (ee_ctr)

1.2) Démarrage : initialisation + ATR

1.2.1) Initialisation matérielle

Dans **main()**, le microcontrôleur configure différents registres (ports, power reduction, etc.). L'objectif est de préparer l'IO nécessaire à la communication.

1.2.2) Envoi de l'ATR (Answer To Reset)

Juste après l'initialisation, le firmware appelle :

- **atr()**;
- puis **valide()**; (pour finir une transaction interrompue si besoin)

Rôle de l'ATR : c'est la "carte d'identité" envoyée automatiquement par la carte après reset. Il indique :

- la convention (ici 0x3B),
- la structure des octets suivants (paramètres),
- et une zone “historique” (historical bytes) qui contient la chaîne "rubro".

1.3) Format des échanges : APDU en T=0

1.3.1) En-tête APDU (toujours 5 octets)

Le firmware lit systématiquement :

- cla = recbytet0();
- ins = recbytet0();
- p1 = recbytet0();
- p2 = recbytet0();
- p3 = recbytet0();

Interprétation :

- **CLA** : “classe” de commande (groupe fonctionnel)
- **INS** : instruction (action précise)
- **P1/P2** : paramètres (ici servent surtout pour le compteur **anti-rejoue**)
- **P3** : longueur attendue (en émission) ou longueur fournie (en réception), selon la commande

1.3.2) Réponse APDU

Après traitement :

- la carte envoie éventuellement des données (avec sendbytet0)
- puis envoie **toujours** le status :
  - sendbytet0(sw1);
  - sendbytet0(sw2);

Exemples de codes utilisés :

- 90 00 : succès
- 6C XX : longueur incorrecte (XX = longueur attendue)
- 6D 00 : INS inconnu
- 6E 00 : CLA inconnue
- 63 XX : échec (souvent essais restants)
- 69 82/83/84... : conditions de sécurité non satisfaites / blocage / anti-rejoue

1.4) Commande

Le cœur logique du programme est un double switch :

1.4.1) Premier niveau : CLA

- 0x81 : commandes “système” (version, perso)
- 0x82 : commandes “sécurisées” (PIN/PUK/solde/compteur)

1.4.2) Deuxième niveau : INS

Chaque **INS** appelle une fonction dédiée.

1.5) CLA “0x81” : version + personnalisation

1.5.1) version() - CLA=0x81 INS=0x00

But : renvoyer la version de la carte.

- Vérifie p3 == 4 (car "2.00" fait 4 octets)
- Envoie les 4 octets de ver_str
- Met SW=90 00

Si la taille est mauvaise :

- SW1=6C, SW2=size_ver

1.5.2) intro_perso() - CLA=0x81 INS=0x01

But : écrire la “personnalisation” de la carte et initialiser la sécurité.

Étapes :

1. Vérifie **p3 <= 32** (MAX_PERSO)
2. Lit p3 octets de perso dans un buffer RAM perso[]
3. Calcule un PUK (6 chiffres ASCII) via compute_puk_from_perso(perso, len, puk)
4. Réinitialise plusieurs éléments de sécurité :
    - PIN remis au défaut 1,2,3,4
    - compteurs d’essais PIN/PUK remis à max
    - compteur anti-rejeu remis à 0
    - solde remis à 0
5. Écrit tout ça en EEPROM via une transaction engage(...); valide();
6. Met pin_ok=0 et SW=90 00

/!\ Commande qui “repart de zéro” et pose l’identité (perso) et les pin/puk/compteurs.

1.5.3) lire_perso() - CLA=0x81 INS=0x02

But : relire la personnalisation.

APDU :
81 02 00 00 **P3** (ici **p3** doit égaler la taille stockée)

Étapes :

1. Lit t = ee_taille_perso
2. Si p3 != t → 6C t
3. Acquitte, puis envoie **t** octets depuis ee_perso[]
4. 90 00

1.6) CLA “0x82” : sécurité + solde + anti-rejeu

1.6.1) État RAM important : pin_ok

La carte utilise un flag RAM :

- pin_ok = 1 après verifier_pin() si le PIN est correct
- ensuite, **chaque opération sensible consomme ce droit** (pin_ok = 0)

Donc le scénario normal est :

1. **VERIFY PIN**
2. faire UNE opération (**lire_solde** OU **credit** OU **debit**)
3. si on veut refaire une opération -> re-**VERIFY PIN**

1.6.2) Vérification/changement/reset PIN

A) verifier_pin() - INS=0x04

APDU : 82 04 00 00 04 **[PIN]**

- Si compteur d’essais à 0 -> 69 83 (bloqué)
- Vérifie la longueur p3 == 4
- Lit 4 octets et compare à ee_pin
- Si OK :
  - met pin_ok=1
  - remet ee_pin_tries = 3
  - 90 00
- Sinon :
  - décrémente ee_pin_tries
  - renvoie 63 XX (essais restants) ou 69 83 si 0

B) changer_pin() - INS=0x05

APDU : 82 05 00 00 08 **[oldPIN(4)][newPIN(4)]**

- Re-vérifie l’ancien PIN

- Si **OK** : écrit le nouveau PIN en **EEPROM** via **transaction**
- Remet les essais PIN à max
- pin_ok est consommé (mis à 0)

C) reset_pin_par_puk() - INS=0x06

APDU : 82 06 00 00 0A **[PUK(6)][newPIN(4)]**

- Si essais PUK à 0 -> 69 83
- Compare les 6 octets au ee_puk
- Si OK :
    - écrit nouveau PIN (transaction)
    - remet essais PIN et PUK à max
    - pin_ok=0
    - 90 00
- Sinon : décrémente essais PUK, 63 XX ou 69 83

1.6.3) Anti-rejeu par compteur (ee_ctr)

Pour credit() et debit(), la carte exige que P1/P2 contiennent **exactement** le compteur EEPROM courant.

Dans check_and_update_ctr() :

- lit ctr_eep = ee_ctr
- construit ctr_req = p1 | (p2<<8) (little endian)
- si différent -> 69 84
- sinon :
    - incrémente ee_ctr
    - écrit en EEPROM
    - OK

Effet : si on rejoue une ancienne commande “**crédit/débit**”, elle échoue car le compteur **ne correspond plus**.

lire_compteur() - INS=0x07

Renvoie le compteur courant sur **2 octets** (little endian).

1.6.4) Gestion du solde (ee_solde)

A) lire_solde() - INS=0x01

APDU : 82 01 00 00 02

- Vérifie pin_ok == 1, sinon 69 82
- Consomme pin_ok

- Vérifie longueur p3==2
- Envoie le solde sur 2 octets (LSB puis MSB)
- 90 00

B) credit() — INS=0x02

APDU : 82 02 P1 P2 02 **[montant LSB][montant MSB]**

- Exige PIN (et consomme pin_ok)
- Exige anti-rejeu via P1/P2
- Lit montant c (little endian)
- Lit solde s, calcule s += c
- Test overflow : si s < c → 61 00
- Écrit ee_solde = s
- 90 00

C) debit() — INS=0x03

APDU : 82 03 P1 P2 02 **[montant LSB][montant MSB]**

- Exige PIN (consomme)
- Exige anti-rejeu
- Lit montant d
- Si d > s -> 61 00 (solde insuffisant)
- Sinon s -= d
- Écrit s via transaction engage(2, &s, &ee_solde, 0); valide();
- 90 00

1.7) Persistance EEPROM + “anti-arrachement”

Objectif

Éviter qu’une coupure (arrachement) pendant une écriture laisse la carte dans un état incohérent.

Fonctionnement

- ee_trans en EEPROM contient :
  - un state (vide/plein),
  - nb_ope opérations,
  - pour chaque opération : taille + adresse destination,
  - un buffer contenant les données à recopier.

Deux phases :

1. engage() prépare en EEPROM le journal (buffer + méta-données) puis met state=plein
2. valide() si state=plein, recopie le buffer vers les adresses finales et met state=vide

2) Relations avec les autres applications et modes de communication

Rubrovitamin ne parle qu'à un **lecteur de carte** via APDU ISO7816 T=0. Les autres briques (Lubiana, Berlicum, Lunar White…) utilisent typiquement une librairie (ex: pyscard) pour envoyer ces APDUs.

Relations principales :

- **Lubiana (personnalisation)** : envoie 0x81/0x01 (intro_perso), lit 0x81/0x02, et peut afficher 0x81/0x00.
- **Borne de recharge (Berlicum)** : lit perso, lit solde, transfère/ajoute du crédit → 0x82/0x02.
  Machine à café (Lunar White) : débite 0.20€ -> 0x82/0x03.
- **Rodelika / PurpleDragon** : n'accède pas directement à la carte

3) Vulnérabilités, dysfonctionnements et attaques possibles

3.1) Attaques "fonctionnelles"

A) Personnalisation non protégée (prise de contrôle)

intro_perso() (CLA 0x81 INS 0x01) :

- ne demande **aucune authentification**
- **réinitialise** solde, compteur, PIN/PUK/essais

Un attaquant avec un lecteur peut :

- "re-personnaliser" une carte (perte des crédits),
- remettre compteur à 0,
- forcer un PUK prédictible.

B) PIN par défaut faible

PIN par défaut = 1 2 3 4 (4 octets) :

- facile à deviner

C) PUK dérivé de la perso (faible cryptographiquement)

Le PUK est calculé via compute_puk_from_perso() :

- c'est une fonction simple, non cryptographique
- si la **perso** est lisible (lire_perso()), un attaquant peut **recalculer** le PUK hors carte.

Conséquence : reset PIN par PUK devient contournable.

D) Anti-rejoue contournable

- Le compteur ee_ctr est **lisible sans PIN** (lire_compteur()), donc un attaquant connaît la valeur attendue.

3.3) Attaques "physiques" / bas niveau

- Interception APDU : le PIN/PUK transitent en clair sur le bus lecteur <-> carte.

4) Solutions proposées

4.1) Verrouiller la personnalisation (critique)

intro_perso() ne doit être faisable **qu'une seule fois** ou uniquement par un agent autorisé.

Solutions :

1. **Flag EEPROM "carte personnalisée"** (ee_is_perso=0/1)
    - si déjà personnalisée -> **refuser** intro_perso()
2. **Authentification admin** pour classe 0x81 :
    - soit un **PIN admin** distinct

4.2) Anti-rejoue : renforcer le modèle

- Ne pas exposer lire_compteur() sans authentification (ou le retirer).

4.3) PIN/PUK : durcissement

- Forcer le changement du PIN à la première utilisation (pas "1234" permanent).

Lunar White (machine à café)

Lunar White est l'**application de machine à café** qui permet à un utilisateur d'acheter une boisson en débitant une carte à puce Rubrovitamin. Elle communique avec la carte via un

lecteur PC/SC, vérifie le PIN, contrôle le solde et réalise le débit sécurisé. En complément du fonctionnement demandé dans le sujet, Lunar White intègre une **interface web**, un **système de logs** et un **mode connecté à une base de données** pour tracer les transactions.

1) Présentation générale et fonctionnement détaillé

1.1) Rôle de l'application

Lunar White est une application **Flask** qui joue le rôle de "front-end" et "contrôleur" d'une machine à boissons :

- interface web (page /) pour choisir une boisson,
- API JSON (/api/...) pour piloter les étapes : présence carte -> vérification PIN -> achat,
- dialogue avec la carte via **pyscard** (lecteur PC/SC),
- (ajout) journalise les actions dans un fichier log.txt,
- (ajout) enregistre le débit dans une **base MySQL** (Rodelika Web / PurpleDragon).

1.2) Composants du code

- **Flask** : routes + rendu template index.html
- **pyscard** :
  - readers() pour trouver un lecteur
  - conn.transmit(apdu) pour envoyer les APDUs
- **MySQL** :
  - connexion via mysql.connector
  - insertion d'une ligne dans Transactions (Type='DEBIT')
- **Logs locaux** : écriture append dans log.txt

1.3) Données métier

- Prix fixe : PRIX_BOISSON = 20 (centimes, soit 0,20€), conforme au sujet.
- Boissons : dictionnaire BOISSONS (4 boissons proposée)

2) Relations avec les autres applications et modes de communication

2.1) Communication avec Rubrovitamin (carte à puce)

Canal : lecteur PC/SC (pyscard), échanges **APDU ISO7816**.

APDUs utilisés (tous présents dans le code) :

- Lire perso : 81 02 00 00 **P3** (avec gestion de SW1=6C)
- Vérifier PIN : 82 04 00 00 04 **[PIN]**

- Lire solde : 82 01 00 00 02
- Lire compteur anti-rejoue : 82 07 00 00 02
- Débiter : 82 03 P1 P2 02 **[LSB][MSB]**

2.2) Communication avec Purple Dragon / Rodelika Web (BDD)

Canal : MySQL (TCP 3306) vers 172.20.10.3.

Ajout d'une synchronisation : après débit réussi sur la carte, on enregistre un **DEBIT** dans la table Transactions pour l'étudiant, si on arrive à extraire Num_Etudiant depuis la "perso".

/!\ Lunar White fonctionne en "autonomie (pas de base de données)" ; ce mode connecté est comme **optionnel**.

3) Déroulé fonctionnel concret (scénarios)

3.1) Détection carte (API)

POST /api/check_card

1. get_card_connection() :
    - prend le 1er lecteur readers()[0]
    - createConnection().connect()
2. renvoie JSON succès/erreur
3. gère un cas spécial : CARD_DISCONNECTED si carte "unpowered" / erreur PCSC 0x80100067

3.2) Vérification PIN + affichage solde

POST /api/verify_pin avec {pin: "1234"}

1. validation format PIN côté serveur (4 chiffres)
2. connexion carte
3. APDU VERIFY PIN
4. APDU READ SOLDE
5. renvoie solde (centimes)

3.3) Achat d'une boisson (débit + BDD)

POST /api/acheter_boisson avec {boisson_id, pin}
Étapes codées :

1. vérifie boisson_id + PIN format
2. connexion carte
3. **lit le compteur anti-rejoue** (82 07)
4. **vérifie PIN** (1ère fois)
5. **lit le solde** (82 01) -> vérifie suffisant

6. **revérifie PIN** (2ème fois)
   (nécessaire car la carte consomme pin_ok lors de lire_solde)
7. débite la carte (82 03 P1P2 ... 0x14)
8. lit la perso (81 02) et extrait Num_Etudiant
9. enregistre l'opération dans MySQL (si Num_Etudiant valide)
10. renvoie succès + nouveau solde (recalculé localement solde - 20)

4) Ajout par rapport au sujet

Ajouts dans le code

1. **Application Web Flask** (API JSON + page HTML)
2. **PIN/PUK**
3. **Anti-rejoue**
4. **Logs locaux (log.txt)**
5. **Mode connecté à la base MySQL (non obligatoire)**
6. **Lecture de la personnalisation et extraction de Num_Etudiant**
7. **Gestion d'erreurs PCSC "carte déconnectée/unpowered"**
8. **Boissons supplémentaires**

5) Vulnérabilités, dysfonctionnements et attaques possibles (Lunar White)

5.1) Vulnérabilités côté serveur web (Flask)

1. **Absence d'authentification/autorisation sur les endpoints**

- Toute personne sur le réseau peut appeler :
  - /api/get_logs (exfiltration d'infos),
  - /api/acheter_boisson (tentatives de bruteforce PIN),
  - /api/check_card (sonde présence carte).
- Pas de notion de "borne physique autorisée" ou "session utilisateur".

2. **Pas de limitation de débit (rate limiting)**

- Un attaquant peut spammer /api/verify_pin et provoquer :
  - blocage PIN (DoS) en épuisant les essais sur la carte,

5.2) Vulnérabilités BDD / cohérence carte <-> DB

1. **Incohérence forte : débit carte OK mais DB KO**

- Le code le note dans les logs : si l'INSERT échoue (FK, réseau, DB down), la carte est débitée mais la transaction n'est pas enregistrée.
- En mode "connecté", c'est un problème majeur d'audit.

2. **Confiance excessive dans la perso (source non authentifiée)**

- Le Num_Etudiant provient de la carte, mais une carte peut être clonée / falsifiée.
- Si quelqu'un arrive à personnaliser une carte avec un numéro d'un autre étudiant, Lunar White créditera/débitera en DB sous cette identité.

4) Solutions proposées (palliatives + solides)

4.1) Durcissement Flask (indispensable en prod)

Solides

- Mettre derrière un reverse proxy (Nginx) + HTTPS si réseau non totalement isolé.
- Ajouter authentification borne :
    - restreindre aux IPs du réseau des bornes.
- Ajouter **rate limiting** :
    - ex: 5 tentatives PIN/minute/IP, backoff exponentiel.
- Mettre une vraie gestion de session côté UI (CSRF token si formulaires, etc.).

4.2) Cohérence carte <-> base MySQL (mode connecté)

Palliatives

- Si **l'INSERT** échoue, renvoyer un statut "achat ok mais non synchronisé" (au lieu de succès silencieux)..

Solides

- Mettre un identifiant unique de transaction :
    - ex: {Num_Etudiant, ctr, timestamp, boisson} pour éviter doublons.

4.4) Anti-rejoue et concurrence

Palliatives

- Réessayer automatiquement si 69 84 (anti-rejoue) :
    1. relire compteur,
    2. revérifier PIN,
    3. relancer débit.
       (avec limites pour éviter boucles infinies)

4.5) Protection contre DoS / blocage PIN

Palliatives

- Temporisation serveur après échec PIN (ex: +1s, +2s, +4s…).
- Bloquer temporairement l'IP après N échecs.

Solides

- Politique borne : après 1 échec PIN, exiger retrait/réinsertion carte.

Logiciel de personnalisation Lubiana

1. Présentation générale et fonctionnement

Le logiciel Lubiana est un logiciel de personnalisation en ligne de commande (CLI) développé en Python, dont le rôle principal est d'injecter des données dans la carte à puce après son initialisation par le logiciel embarqué Rubrovitamin.

Lubiana agit comme un outil externe d'administration, utilisé par un agent administratif, pour pouvoir :

- attribuer une carte à un étudiant,
- écrire les informations personnelles (numéro étudiant, nom, prénom),
- initialiser le solde de la carte,
- gérer le code PIN,
- consulter les données et le solde de la carte.

Lubiana repose entièrement sur le dialogue APDU norme ISO 7816, en utilisant la bibliothèque pyscard, afin de communiquer directement avec la carte via un lecteur de carte.

2. Fonctionnement détaillé

Le fonctionnement de Lubiana peut être découpé en plusieurs étapes clés.

2.1 Connexion au lecteur et à la carte

Au lancement, Lubiana :

1. détecte les lecteurs de cartes disponibles via `scardsys.readers()`,
2. établit une connexion avec le premier lecteur trouvé,
3. récupère et affiche l'ATR de la carte pour vérification.

Cette étape garantit que :

- un lecteur est bien présent,
- une carte est insérée,
- la communication bas niveau est fonctionnelle.

2.2 Communication par APDU

Toutes les opérations sont réalisées via des APDU structurées selon la norme ISO 7816 (intégrés via Rubrovitamin dans la carte), avec deux classes principales :

- CLA 0x81 : opérations de personnalisation
  (lecture version, lecture/écriture des données personnelles)

- CLA 0x82 : opérations sécurisées
  (PIN, solde, compteur anti-rejoue, crédit)

Exemples d'opérations réalisées par Lubiana :

- Lecture de la version de la carte
- Lecture des données personnelles
- Attribution de la carte (`num;nom;prenom`)
- Vérification du PIN
- Lecture et écriture du solde
- Changement du code PIN

Lubiana ne contient aucune logique métier propre à la carte : il respecte strictement les règles imposées par Rubrovitamin.

2.3 Gestion de la sécurité

Lubiana implémente côté client l'utilisation des mécanismes de sécurité présents sur la carte à puce :

- **PIN :** Toute opération sensible (lecture du solde, crédit, modification) nécessite une authentification préalable par PIN.

Lubiana **n'implémente pas** le mécanisme de blocage du PIN, **il l'utilise** via les règles définies dans Rubrovitamin. Le blocage du PIN est **implémenté dans la carte** (Rubrovitamin), Lubiana ne peut ni le contourner ni le réinitialiser.

En somme, chaque opération sensible déclenche une **vérification explicite du PIN** par APDU.

Exemple dans le code de Lubiana

```
apdu = [0x82, 0x04, 0x00, 0x00, 0x04] + pin_bytes

data, sw1, sw2 = conn_reader.transmit(apdu)
```

Interprétation des réponses possibles :

- `0x90 0x00` → PIN correct
- `0x63 xx` → PIN incorrect, X essais restants
- `0x69 0x83` → PIN bloqué définitivement
- `0x6C` → Erreur de longueur sur les octets transmis

Cet extrait construit une APDU de vérification du code PIN.
L'instruction `0x04` correspond à la commande *VERIFY PIN* définie dans Rubrovitamin.
Les 4 octets du PIN saisis par l'utilisateur sont ajoutés à l'APDU, puis envoyés à la carte via le lecteur.

La carte compare alors le PIN reçu avec celui stocké en EEPROM et renvoie un code de statut (`SW1/SW2`) indiquant si l'authentification est valide, incorrecte ou bloquée.

Lubiana se contente d'interpréter cette réponse et ne prend aucune décision de sécurité côté logiciel.

- **Anti-rejeu :** Pour les opérations de crédit, Lubiana applique strictement le protocole anti-rejeu imposé par la carte.

Le principe est le suivant :

1. Lire le compteur de transaction sur la carte
2. Insérer ce compteur dans l'APDU de crédit
3. La carte vérifie que le compteur est valide et attendu

Exemple dans le code de Lubiana (lecture compteur)

```
apdu = [0x82, 0x07, 0x00, 0x00, 0x02]

data, sw1, sw2 = conn_reader.transmit(apdu)
```

```
ctr = int(data[0]) | (int(data[1]) << 8)
```

Cette APDU permet de lire le compteur de transaction interne à la carte.
L'instruction `0x07` correspond à la commande *READ COUNTER* définie dans Rubrovitamin.

La carte renvoie la valeur actuelle du compteur sur 2 octets (format little-endian).
Lubiana reconstruit cette valeur sous forme d'entier afin de l'utiliser lors d'une opération de crédit.

Ce compteur représente l'état logique de la dernière transaction validée par la carte.

Exemple dans le code de Lubiana (crédit avec compteur)

```
apdu = [0x82, 0x02, p1, p2, 0x02, montant_lsb, montant_msb]
```

Toute tentative de rejouer une ancienne APDU de crédit est refusée par la carte !
(`SW1 = 0x69, SW2 = 0x84`).

Cette APDU réalise une opération de crédit sur la carte.
Les paramètres `p1` et `p2` contiennent le compteur précédemment lu, qui sert de preuve de fraîcheur de la transaction.

La carte vérifie que le compteur transmis correspond exactement à la valeur attendue.
Si ce n'est pas le cas (rejeu d'une ancienne trame, incohérence, attaque), la transaction est refusée.

Ce mécanisme empêche toute tentative de duplication ou de rejouement d'une APDU de crédit.

3. Relations avec les autres applications et modes de communication

Lubiana est au cœur de la phase administrative du projet.

3.1 Relation avec Rubrovitamin

- Rubrovitamin définit :
    - la mémoire EEPROM,

    - les instructions APDU,
    - les règles de sécurité (PIN, anti-rejeu, anti-arrachement).
- Lubiana exploite ces règles, pour personnaliser la carte, sans jamais les modifier.

La communication se fait exclusivement par APDU norme ISO 7816 via lecteur PC/SC.

3.2 Relation avec Berlicum et la base de données

Lubiana n'interagit pas directement avec la base de données.
Son rôle se limite volontairement à la personnalisation et à l'administration de la carte.

Les aspects transactionnels, bonus et recharges sont gérés ultérieurement par :

- Lunar White
- Berlicum (CLI et Web),
- la base de données MySQL.

Cette séparation limite les risques et clarifie les responsabilités des logiciels.

4. Vulnérabilités, dysfonctionnements et attaques possibles

Malgré les mécanismes de sécurité, plusieurs risques existent.

4.1 Attaques par force brute sur le PIN

Un utilisateur malveillant pourrait tenter :

- des essais répétés de PIN.

Contre mesure existante sur la carte (via **Rubrovitamin**) :
Le code PIN est limité en nombre d'essais (3) et peut être bloqué par la carte.

4.2 Rejeu d'APDU de crédit

Sans mécanisme de protection, un attaquant pourrait :

- rejouer une ancienne trame de crédit qu'il aurait intercepté.

Contre mesure existante (**Rubrovitamin** et **Lubiana**) :
L'anti-rejeu basé sur le compteur empêche toute réutilisation d'une APDU déjà utilisée.

4.3 Arrachement de carte pendant une transaction

Si la carte est retirée brutalement pendant une écriture :

- les données pourraient devenir incohérentes.

Contre mesure implémentée (**Rubrovitamin**) :
Le mécanisme d'anti-arrachement garantit que la transaction est soit entièrement réalisée, soit annulée.

5. Conclusion sur Lubiana

Lubiana joue un rôle clé mais volontairement limité dans le projet :

- Il ne définit pas la sécurité, il l'applique.
- Il ne gère pas la logique métier, il personnalise.
- Il constitue un pont sécurisé entre l'agent administratif et la carte à puce.
- Lubiana n'implémente aucune logique métier
- Lubiana ne stocke aucune donnée sensible
- Lubiana ne fait que dialoguer avec la carte

Cela réduit la surface d'attaque, les risques de compromission et les responsabilités du logiciel.

Logiciel de recharge Berlicum (CLI & Web)

1. Présentation générale et fonctionnement

Le logiciel Berlicum est la borne de recharge du projet *La Carotte Électronique*.
Il est destiné aux étudiants, et permet d'assurer le lien entre :

- la **carte à puce** (porteuse du solde),
- la **base de données Purple Dragon** (bonus, recharges),
- et l'utilisateur final.

Contrairement à **Lubiana**, qui est un outil administratif, **Berlicum est un logiciel utilisateur**.
Il existe sous **deux implémentations distinctes** :

- **Berlicum CLI** : version en ligne de commande (Python)
- **Berlicum Web** : version web (Flask + HTML/JS), développée en bonus

Ces deux versions implémentent **la même logique fonctionnelle**, mais avec des interfaces et des surfaces d'attaque différentes.

Rôles principaux de Berlicum

Berlicum permet à un étudiant de :

- consulter ses informations personnelles stockées sur la carte,
- consulter les bonus disponibles en base de données,
- transférer ses bonus de la base vers la carte,
- consulter le solde disponible sur la carte,
- recharger sa carte via une **simulation de paiement par carte bancaire**.

2. Fonctionnement détaillé

Le fonctionnement de Berlicum repose sur une **double interaction critique** :

- **Carte à puce** → via APDU norme ISO 7816
- **Base de données MySQL** → via requêtes SQL et procédures stockées dans la Base de données

2.1 Lecture des informations depuis la carte

Au démarrage d'une opération, Berlicum lit les données personnelles stockées sur la carte :

- Numéro étudiant
- Nom
- Prénom

Ces informations sont stockées sur la carte sous la forme :

num;nom;prenom

Exemple de lecture côté Berlicum :

perso = _read_perso_raw()

parts = perso.split(";")

etu_num = parts[0].zfill(8)

Explication :

- `_read_perso_raw()` envoie une APDU de lecture à la carte.
- Les données sont découpées à partir du séparateur “`;`”.
- Le numéro étudiant est normalisé sur 8 caractères (`zfill(8)`) afin de correspondre au format attendu en base de données.

Cette lecture permet :

- d’identifier l’étudiant,
- de lier les opérations carte ↔ base de données,
- d’éviter toute saisie manuelle côté utilisateur.

2.2 Consultation des bonus en base de données

Les **bonus** sont stockés dans la base de données (Purple Dragon), dans la table `Transactions`.

Un bonus est défini par :

- `Type = 'CREDIT'`
- `Commentaire` commençant par `"Bonus"`
- non encore marqué comme transféré

Exemple de requête utilisée :

SELECT COALESCE(SUM(Montant), 0)

FROM Transactions

```
WHERE Num_Etudiant = %s
  AND Type = 'CREDIT'
  AND Commentaire LIKE 'Bonus%'
  AND Commentaire NOT LIKE '%transféré%'
```

Explication :

- La requête calcule le **total des bonus disponibles**.
- `COALESCE` garantit un retour à `0` si aucun bonus n'est présent.
- Le marquage textuel `"transféré"` permet de conserver un historique sans supprimer les données.

Cette logique permet :

- de conserver un **historique complet**,
- d'éviter la suppression de données,
- d'assurer la traçabilité des bonus.

2.3 Transfert des bonus vers la carte

Le transfert des bonus est une **opération critique**, car elle touche :

- la carte,
- la base de données,
- et doit respecter la cohérence globale.

La logique appliquée est la suivante :

1. Lire les bonus disponibles en base
2. Créditer la carte à puce
3. Marquer les bonus comme transférés en base

Exemple côté carte :

```
credit_card_amount(montant_bonus)
```

Explication :

- Cette fonction déclenche une authentification par PIN.
- Elle lit le compteur anti-rejeu.
- Elle envoie une APDU de crédit sécurisée à la carte.

Exemple côté base :

```
UPDATE Transactions

SET Commentaire = CONCAT(Commentaire, ' (transféré)')

WHERE Num_Etudiant = %s
```

Explication :

- Les bonus ne sont pas supprimés (UPDATE).
- Ils sont simplement marqués comme transférés.
- Cela empêche un double transfert et conserve l'historique.

! **Attention** !
La carte et la base ne partagent pas de transaction distribuée → une désynchronisation est possible en cas d'erreur intermédiaire.

2.4 Recharge par carte bancaire (simulation)

La recharge par carte bancaire est **volontairement simulée** dans le cadre du projet.

Le principe est :

- demander un montant,
- afficher des messages simulant le paiement,
- créditer la carte,
- créditer la base de données via une procédure stockée.

Exemple d'appel de procédure stockée :

```
cursor.callproc(
    "CrediterCompte",
    [etu_num, float(montant), "Recharge CB Berlicum"]
)
```

Qu'est-ce qu'une procédure stockée ?

Une **procédure stockée** est une fonction SQL définie directement dans la base de données, exécutée côté serveur, appelée depuis l'application cliente.

Avantages en termes de sécurité et de robustesse

L'utilisation de procédures stockées permet :

- de centraliser la logique métier sensible,
- de limiter les requêtes SQL dynamiques côté application,
- de réduire les risques d'injection SQL,
- d'imposer des contrôles stricts (existence du compte, montants valides),
- de faciliter les audits et le contrôle des accès.

Dans Berlicum, la procédure `CrediterCompte` garantit que :

- le compte étudiant existe,
- le montant est valide,
- la transaction est atomique côté base.

Cette approche de procédure stockées permet :

- de centraliser la logique métier en base,
- de garantir la cohérence des écritures,
- de faciliter les contrôles et audits.

3. Relations avec les autres applications

3.1 Relation avec la carte à puce (Rubrovitamin)

Berlicum communique avec la carte exclusivement via **APDU ISO 7816**.

Il exploite les mécanismes définis dans Rubrovitamin :

- PIN obligatoire pour les opérations sensibles,
- anti-rejeu via compteur,
- anti-arrachement via transaction EEPROM.

Berlicum **ne définit aucune règle de sécurité**, il les applique strictement.

3.2 Relation avec la base de données (Purple Dragon)

Berlicum est le **seul logiciel utilisateur** avec Rodelika à interagir directement avec la base.

Il utilise :

- des requêtes SQL contrôlées,
- des procédures stockées,
- une journalisation logique via commentaires.

Cette centralisation permet :

- la traçabilité,
- l'analyse a posteriori,
- la détection d'anomalies.

4. Différences entre Berlicum CLI et Berlicum Web

Berlicum CLI

- Interface simple et directe
- Moins de surface d'attaque
- Utilisé principalement pour les tests et démonstrations
- Exécution locale maîtrisée

Berlicum Web

- Interface utilisateur plus réaliste
- Accès via navigateur
- Introduction de nouvelles menaces :
    - injections,
    - détournement de requêtes,
    - désynchronisation client/serveur

La version Web est une **implémentation alternative**, développée en bonus, conservant la même logique métier que la CLI.

5. Vulnérabilités, dysfonctionnements et attaques possibles

5.1 Désynchronisation carte / base

Scénario :

- carte créditée,
- erreur BDD (ex : compte inexistant),
- incohérence globale.

Impact :

- solde carte ≠ solde BDD.

5.2 Attaques par rejeu

Tentative :

- rejouer une ancienne APDU de crédit.

Contre mesure :

- anti-rejeu basé sur compteur (Rubrovitamin),
- APDU refusée avec `0x69 0x84`.

5.3 Vulnérabilités spécifiques au Web

- manipulation des requêtes HTTP,
- modification du montant côté client,
- appels multiples à l'API.

Contre mesures :

- validations côté serveur,
- confirmation utilisateur,
- logique critique uniquement côté backend.

6. Conclusion sur Berlicum

Berlicum est le **cœur transactionnel** du projet :

- il relie la carte, la base et l'utilisateur,
- il met en œuvre les règles de sécurité sans les définir.

La coexistence des versions **CLI** et **Web** permet :

- de comparer deux architectures,
- d’illustrer les impacts de l’interface sur la sécurité.

Base de données Purple Dragon

1. Extraits du schéma SQL (MySQL)

Les extraits ci-dessous proviennent du fichier `carote_electronique.sql` et illustrent la structure réellement exploitée.

```sql
CREATE TABLE users (
  Num_Etudiant   CHAR(8)      NOT NULL,
  Nom            VARCHAR(60)  NOT NULL,
  Prenom         VARCHAR(60)  NOT NULL,
  Email          VARCHAR(120) NOT NULL UNIQUE,
  Tel            VARCHAR(20),
  UNIQUE (Num_Etudiant),
  PRIMARY KEY (Num_Etudiant)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE Agents (
  id            INT           NOT NULL AUTO_INCREMENT,
  Identifiant   VARCHAR(60)   NOT NULL UNIQUE,
  Nom           VARCHAR(60)   NOT NULL,
  Prenom        VARCHAR(60)   NOT NULL,
  Password_Hash VARCHAR(255)  NOT NULL,
  Role          ENUM('ADMIN','AGENT') NOT NULL DEFAULT
'AGENT',
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE Compte (
  Num_Etudiant   CHAR(8)       NOT NULL,
  Solde_Actuel   DECIMAL(10,2) NOT NULL DEFAULT 0.00,
  Derniere_MAJ   DATETIME      NOT NULL DEFAULT
CURRENT_TIMESTAMP,
  PRIMARY KEY (Num_Etudiant),
  FOREIGN KEY (Num_Etudiant) REFERENCES users(Num_Etudiant)
    ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE Transactions (
  id              BIGINT       NOT NULL AUTO_INCREMENT,
  Num_Etudiant     CHAR(8)      NOT NULL,
  Type            ENUM('BONUS','DEPENSE') NOT NULL,
  Montant         DECIMAL(10,2) NOT NULL,
```

```
  Date_Transaction DATETIME     NOT NULL DEFAULT
CURRENT_TIMESTAMP,
  Commentaire      VARCHAR(255),
  PRIMARY KEY (id),
  INDEX (Num_Etudiant),
  FOREIGN KEY (Num_Etudiant) REFERENCES users(Num_Etudiant)
    ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

> Remarque : une table `Carte` est définie dans le schéma, mais elle n'est pas utilisée dans l'implémentation (aucun flux applicatif ne s'appuie dessus).

2. Rôle et objectifs

La base de données **Purple Dragon** (MySQL) constitue le socle de persistance des données du système. Elle centralise l'ensemble des informations nécessaires à l'exploitation de la carte à puce, notamment :

- l'identification des utilisateurs,
- les données métiers nécessaires au fonctionnement des applications Rodelika.

3. Structure générale

Le schéma de la base, défini dans le fichier `carote_electronique.sql`, repose sur un modèle relationnel structuré. Chaque table répond à une responsabilité précise (principe de séparation des données) et les relations sont assurées par des clés primaires et étrangères garantissant l'intégrité référentielle.

Cette structuration permet :

- une cohérence forte des données,
- une exploitation simultanée par plusieurs applications,
- une évolutivité du modèle sans remise en cause de l'existant.

Logiciel Rodelika – Version CLI

1. Présentation et rôle

Rodelika CLI est une application Python en ligne de commande destinée aux opérations techniques et administratives. Elle permet aux agents d'interagir directement avec la base de données Purple Dragon afin de consulter et gérer les informations liées aux comptes étudiants.

2. Principe de fonctionnement

Le fonctionnement de Rodelika CLI repose sur trois éléments principaux :

- une connexion sécurisée à la base de données MySQL Purple Dragon,
- une authentification des agents à l'aide de mots de passe hachés,
- l'exécution d'actions métiers simples (consultation, ajout de bonus, suivi des transactions).

Logiciel Rodelika – Version Web

1. Présentation et rôle

Rodelika Web est une application web développée en Python (framework Flask). Elle fournit une interface graphique permettant une utilisation plus accessible du système, sans passer par la ligne de commande.

2. Principe de fonctionnement

L'application web met en œuvre :

- une authentification par session utilisateur,
- un contrôle d'accès aux différentes pages,
- l'affichage des données issues de la base Purple Dragon (comptes, transactions, statistiques).

La version Web est destinée à l'usage quotidien et à la démonstration fonctionnelle du projet.

3. Présentation et rôle

Rodelika Web est une application web développée en Python (Flask). Elle fournit une interface graphique destinée aux utilisateurs, facilitant l'exploitation quotidienne des données sans recours à la ligne de commande.

4. Fonctionnement essentiel

L'application met en œuvre :

- une authentification par session,
- un contrôle d'accès aux routes,
- des requêtes SQL pour l'affichage des informations et statistiques.

```python
session["agent_id"] = agent["id"]
```

```python
@login_required
def index():
    return render_template("index.html")
```

Extraits de code (Python / Flask)

Configuration de l'application, secret de session et connexion MySQL (`rodelika_web.py`):

```python
DB_CONFIG = {
    "host": "purple-dragon-db",
    "port": 3306,
    "user": "rodelika",
    "password": "rodelika",
    "database": "carote_electronique",
}
```

Contrôle d'accès par session (décorateur `login_required`) :

```python
def login_required(f):
    @wraps(f)
    def wrapper(*args, **kwargs):
        if "agent_id" not in session:
            flash("Veuillez vous connecter.")
            return redirect(url_for("login"))
        return f(*args, **kwargs)
    return wrapper
```

Authentification Web (récupération de l'agent en base + vérification bcrypt) :

Python

```python
def login():
    if request.method == "POST":
        identifiant = request.form.get("identifiant",
"").strip()
        password = request.form.get("password", "")

        if not identifiant or not password:
            flash("Identifiant et mot de passe requis.")
            return render_template("login.html")

        try:
            cnx = get_db()
            cursor = cnx.cursor(dictionary=True)
            cursor.execute("SELECT * FROM Agents WHERE
Identifiant=%s", (identifiant,))
            agent = cursor.fetchone()
        except Exception as e:
            flash(f"Erreur base de données : {e}")
            return render_template("login.html")
        finally:
            try:
                cnx.close()
            except:
                pass

        if not agent:
            flash("Identifiant incorrect.")
            return render_template("login.html")

        if not bcrypt.checkpw(password.encode(),
agent["Password_Hash"].encode()):
            flash("Mot de passe incorrect.")
            return render_template("login.html")

        session["agent_id"] = agent["id"]
        session["agent_ident"] = agent["Identifiant"]
        session["agent_role"] = agent["Role"]

        flash("Connexion réussie.")
        return redirect(url_for("index"))

    return render_template("login.html")
```

Exemple de requêtes du tableau de bord (compteurs + transactions récentes) :

Python

```
def index():
    stats = {
        "nb_etudiants": 0,
        "nb_comptes": 0,
        "solde_total": 0.0,
        "nb_transactions": 0,
    }
    transactions = []

    try:
        cnx = get_db()
        cursor = cnx.cursor(dictionary=True)

        cursor.execute("SELECT COUNT(*) AS total FROM users")
        stats["nb_etudiants"] = cursor.fetchone()["total"]

        cursor.execute("SELECT COUNT(*) AS total FROM Compte")
        stats["nb_comptes"] = cursor.fetchone()["total"]

        cursor.execute("SELECT SUM(Solde_Actuel) AS total FROM
Compte")
        stats["solde_total"] = cursor.fetchone()["total"] or
0.0

        cursor.execute("SELECT COUNT(*) AS total FROM
Transactions")
        stats["nb_transactions"] = cursor.fetchone()["total"]

        cursor.execute(
            """
            SELECT t.id, t.Num_Etudiant, u.Nom, u.Prenom,
t.Type, t.Montant, t.Date_Transaction, t.Commentaire
            FROM Transactions t
            JOIN users u ON u.Num_Etudiant = t.Num_Etudiant
            ORDER BY t.Date_Transaction DESC
            LIMIT 20
            """
        )
        transactions = cursor.fetchall()

    except Exception as e:
        flash(f"Erreur lors du chargement du tableau de bord :
{e}")
```

```
    finally:
        try:
            cnx.close()
        except:
            pass

    return render_template("index.html", stats=stats,
transactions=transactions)
```

5. Présentation générale

Rodelika Web (`rodelika_web.py`) constitue l'interface applicative destinée aux utilisateurs finaux. Il s'agit d'une application web permettant une exploitation plus accessible et ergonomique des données issues de la carte à puce.

6. Architecture

L'application repose sur :

- un backend Python,
- une connexion sécurisée à la base Purple Dragon,
- des routes applicatives dédiées aux différentes fonctionnalités.

Cette architecture permet une séparation claire entre la logique métier, l'accès aux données et la présentation.

7. Fonctionnalités principales

Rodelika Web permet notamment :

- l'affichage structuré des données utilisateur,
- une interaction simplifiée sans recours à la ligne de commande.

Déploiement et orchestration Docker

1. Objectifs du déploiement

La conteneurisation via Docker constitue une **infrastructure commune** à l'ensemble des projets développés dans le cadre de la SAE. Elle est utilisée pour déployer de manière homogène et reproductible toutes les applications : **Rodelika**, **Berlicum**, **Lubiana**, **Lunar-White** et **Rubrovitamin**, ainsi que les services techniques associés (base de données, services système).

2. Organisation du projet

Le dépôt est structuré de façon modulaire, chaque application disposant de son propre répertoire, tandis que les éléments transverses (Docker, base de données) sont mutualisés :

- `rodelika/` : application carte à puce (CLI et Web)
- `berlicum/` : application CLI et Web
- `lubiana/` : application CLI
- `lunar-white/` : application Web (Flask)
- `rubrovitamin/` : code bas niveau (C / AVR) exécuté dans un conteneur dédié
- `db/` : schéma SQL de la base Purple Dragon
- `docker-compose.yml` et `Dockerfile` : orchestration et construction des images

Cette organisation facilite la lisibilité du projet et la gestion multi-applications.

3. Architecture conteneurisée

Chaque composant logiciel est déployé dans son propre conteneur Docker. L'architecture observée en exécution comprend notamment :

- un conteneur **MySQL** hébergeant la base de données Purple Dragon,
- des conteneurs applicatifs **CLI** (Rodelika CLI, Berlicum CLI, Lubiana CLI),
- des conteneurs applicatifs **Web** (Rodelika Web, Berlicum Web, Lunar-White),
- un conteneur **Traefik** assurant le routage HTTP,
- des conteneurs techniques dédiés (**pcscd**, **rubrovitamin**).

Chaque conteneur est isolé tout en communiquant via des réseaux Docker internes.

4. Orchestration avec Docker Compose

L'ensemble des services est orchestré à l'aide de Docker Compose. Le fichier `docker-compose.yml` permet :

- le démarrage coordonné de tous les projets de la SAE,
- la définition des ports exposés pour les applications Web,
- la gestion des dépendances entre services (base de données, services système),
- la persistance des données via des volumes Docker.

Une seule commande permet ainsi de lancer l’intégralité de l’environnement de travail.

5. Images et services techniques

Les applications Python sont construites à partir d’images basées sur un environnement Debian/Python standardisé. Les services spécifiques, comme **pcscd** (gestion du lecteur de carte à puce) ou **Rubrovitamin** (exécution de code bas niveau), sont isolés dans des conteneurs dédiés afin de respecter le principe de séparation des rôles.