UNIVERSITÉ
CAEN
NORMANDIE

Projet CryptoHack

Rapport de Communication

Master 1 Cybersécurité
Université de Caen – Année 2025-2026

Alexis Debra

Martin Vicquelin

Octobre 2025

Table des matières

1 Introduction

1.1 Contexte

Ce projet a été réalisé dans le cadre du cours de **Communication**, dont l'objectif est de mettre en pratique les outils et méthodes étudiés (LaTeX, Git, gestion de projet agile, présentation orale).

1.2 Objectifs du projet

L'objectif est de résoudre plusieurs **challenges de cryptographie** proposés sur la plateforme CryptoHack, afin de comprendre les principes fondamentaux des systèmes cryptographiques modernes : arithmétique modulaire, RSA, Diffie-Hellman, chiffrement symétrique (AES) et sécurité des protocoles Web (JWT, TLS).

1.3 Présentation de CryptoHack

CryptoHack est une plateforme d'apprentissage de la cryptographie via des défis progressifs en Python, organisés par thématique (Mathématiques, RSA, AES, etc.). Chaque challenge met en lumière un concept cryptographique fondamental, illustré par un exemple concret à résoudre.

Puisque nous n'avons pu vous présenter l'ensemble des challenges que nous avons réalisé, voici le lien vers nos profils respectifs :

— **Alexis Debra — ScaRed**

— **Martin Vicquelin — mv9lagrintaa**

2 Gestion de projet

2.1 Organisation du binôme

Le travail a été réalisé en binôme :

— **Alexis Debra** : General, RSA, Symmetric Ciphers, Mathematics, Diffie-Hellman, Crypto on the Web, Elliptic Curves.

— **Martin Vicquelin** : General, RSA et Symmetric Ciphers.

2.2 Outils utilisés

— **Git & GitHub** : gestion du versionnement et partage du code. Lien du dépôt GitHub : **Projet Communication**

— **LaTeX** : rédaction du rapport et création des slides Beamer.

— **Notion (Kanban)** : suivi agile des tâches et des challenges réalisés.

2.3 Méthodologie agile simplifiée

Le projet a été mené selon une approche **agile légère**, inspirée des méthodes Scrum, adaptée à un projet court (10 heures). Trois mini-sprints ont structuré le travail :

— Sprint 1 : mise en place du dépôt Git et premiers challenges “General”.

— Sprint 2 : réalisation des différents challenges.

— Sprint 3 : rédaction du rapport et préparation de la soutenance.

Un suivi régulier a été effectué sur Notion (Kanban) et documenté sur GitHub.

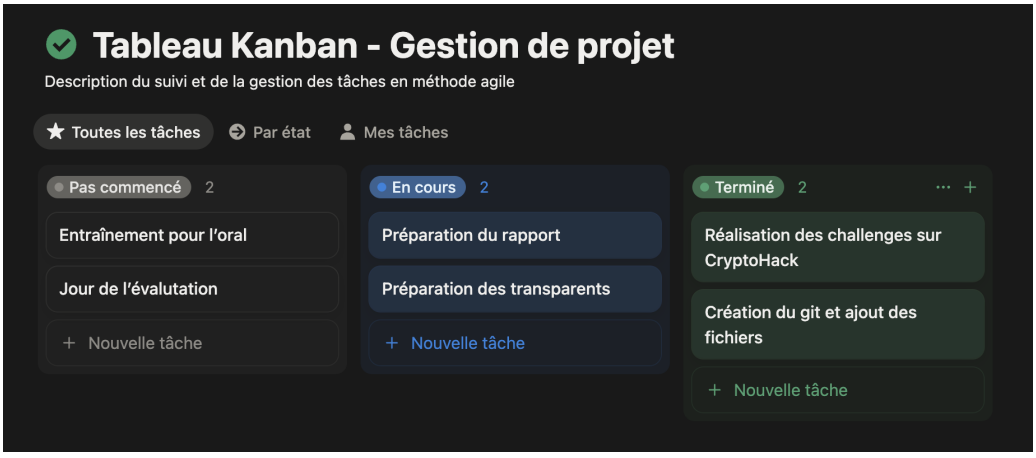

FIGURE 1 – Tableau Kanban Notion utilisé pour le suivi du projet

3 Méthodes

3.1 Approche générale

Chaque challenge a été abordé selon la méthode suivante :

1) Lecture et compréhension du concept cryptographique.
2) Étude des outils mathématiques nécessaires.
3) Implémentation d'un script Python de résolution.
4) Vérification du résultat et documentation du code.

3.2 Environnement technique

Les scripts ont été développés en `Python 3`, avec les bibliothèques suivantes :

— `math`, `Crypto.Util.number`, `sympy`, `requests`.

Les fichiers sont organisés dans des sous-répertoires par thème (General, RSA, Symmetric Ciphers, etc.) pour une meilleure lisibilité.

4 Résultats et explications techniques

4.1 Fondamentaux et arithmétique modulaire

Challenges : *Lemur XOR*, *Encoding Challenge*, *Modular Binomials*. Ces exercices ont permis de revoir les principes de l'arithmétique modulaire et des opérations binaires utilisées dans la cryptographie.

Exemple – Lemur XOR Le challenge *Lemur XOR* consistait à appliquer une opération **XOR pixel par pixel** entre deux images afin de révéler une image cachée. Le XOR est une opération fondamentale en cryptographie car elle est réversible :

$$A \oplus B \oplus B = A$$

Le script suivant montre l'implémentation utilisée :

```
from PIL import Image

lemur = Image.open("lemur.png")
flag = Image.open("flag.png")

for x in range(lemur.width):
    for y in range(lemur.height):
        l = lemur.getpixel((x, y))
        f = flag.getpixel((x, y))
        flag.putpixel((x, y), tuple(a ^ b for a, b in zip(l, f)))

flag.save("lemur_xor_flag.png")
```

Ce code utilise la bibliothèque `PIL` pour effectuer le XOR entre les composantes RGB des deux images et sauvegarder le résultat. Il illustre concrètement comment une même opération peut être utilisée à la fois pour chiffrer et déchiffrer un contenu.

4.2 RSA et cryptographie asymétrique

Challenges : *Euler's Totient*, *Private Keys*, *Manyprime*. Ils ont permis de comprendre les fondements du RSA et ses faiblesses potentielles.

Exemple – RSA Signatures Ce défi de cryptographie explique le principe de la **signature numérique RSA** pour garantir l'authenticité d'un message : il faut d'abord calculer l'empreinte du drapeau secret ($\mathbf{H(m)}$) à l'aide de **SHA256**, puis "chiffrer" cette empreinte avec la **clé privée de l'expéditeur** ($\mathbf{d_1}$) modulo $\mathbf{N_1}$ pour obtenir la signature **S**, prouvant ainsi l'origine et l'intégrité du message.

Le script suivant montre l'implémentation utilisée :

```
from Crypto.Hash import SHA256
from Crypto.Util.number import bytes_to_long
N = 152165836548367313276399812241339188558959483740723840508 4...
d = 111759012106430142625482224734495330913788482694905188504 7...
hm = int('259f7a318fa46d43acd6a32ce41c63e91e16d40fc7a8f3dea39dc5df59d1ef31', 16)
hash = SHA256.new(data=b'crypto{Immut4ble_m3ssag1ng}')
S = pow(bytes_to_long(hash.digest()), d, N)
print(hex(S)[2:])
```

Ce code Python est un exemple d'implémentation de la signature numérique RSA. Il ne déchiffre pas de message, mais prouve l'authenticité et l'intégrité d'un message donné en utilisant une clé privée.

4.3 Diffie-Hellman

Challenges : *Generators of Groups*, *Working with Fields*. Ces exercices ont illustré l'échange de clé publique/privée et l'importance des groupes finis.

Théorème Soit $p > 2$ un nombre premier et $\alpha \in \mathbb{Z}_p^*$. Alors α est un **élément primitif modulo** p si et seulement si :

$$\alpha^{\frac{p-1}{q}} \not\equiv 1 \pmod{p} \quad \text{pour tout nombre premier } q \text{ tel que } q \mid (p-1)$$

Implémentation Le script suivant permet de trouver le plus petit générateur du corps $\mathbb{F}_p$ pour $p = 28151$:

```
from primefac import primefac

p = 28151
totient = p - 1
factors = set(primefac(totient))

for g in range(2, p):
    if all(pow(g, totient // q, p) != 1 for q in factors):
        print(f"Le plus petit générateur pour p={p} est : {g}")
        break
```

Explication Le code calcule les facteurs premiers de $p-1$, puis teste chaque candidat g entre 2 et $p-1$. Si, pour tous les facteurs premiers q de $p-1$, on a $g^{(p-1)/q} \not\equiv 1 \pmod{p}$, alors g est un **générateur** (ou élément primitif) du groupe multiplicatif. Ce principe est

essentiel dans le protocole Diffie–Hellman, car un mauvais choix de générateur compromet la sécurité de l'échange de clés.

4.4 Chiffrement symétrique (AES)

Challenges : *Structure of AES*, *Diffusion through Permutation*, *Bringing It All Together*. Ils ont montré la structure interne d'AES et les principes de confusion et de diffusion.

Fonctions principales Le code étudié implémente plusieurs étapes clés de l'algorithme AES :

- `shift_rows(s)` : décale les lignes de la matrice d'état pour éviter un chiffrement indépendant des colonnes ;
- `inv_shift_rows(s)` : réalise l'opération inverse pour le déchiffrement ;
- `mix_columns(s)` : mélange chaque colonne via des opérations dans le corps de Galois $\mathbb{GF}(2^8)$;
- `inv_mix_columns(s)` : inverse le mélange des colonnes lors du déchiffrement.

Code complet

```
def shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]

def inv_shift_rows(s):
    s[1][1], s[2][1], s[3][1], s[0][1] = s[0][1], s[1][1], s[2][1], s[3][1]
    s[2][2], s[3][2], s[0][2], s[1][2] = s[0][2], s[1][2], s[2][2], s[3][2]
    s[3][3], s[0][3], s[1][3], s[2][3] = s[0][3], s[1][3], s[2][3], s[3][3]

xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

def mix_single_column(a):
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)

def mix_columns(s):
```

```
    for i in range(4):
        mix_single_column(s[i])

def inv_mix_columns(s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v
    mix_columns(s)

state = [
    [108, 106, 71, 86],
    [96, 62, 38, 72],
    [42, 184, 92, 209],
    [94, 79, 8, 54],
]

inv_mix_columns(state)
inv_shift_rows(state)
print(bytes(sum(state, [])))
```

Analyse Ce code illustre le fonctionnement interne d'AES : les opérations `ShiftRows` et `MixColumns` assurent la **diffusion** des bits au sein du bloc de données. En appliquant les fonctions inverses (`inv_mix_columns` et `inv_shift_rows`), on obtient le déchiffrement du bloc. Ces transformations garantissent qu'une petite modification du texte clair entraîne un changement global dans le texte chiffré, rendant AES résistant aux attaques différentielles.

4.5 Cryptographie appliquée sur le Web

Challenges : *JWT Secrets*, *Decrypting TLS 1.2*. Ces deux défis appliquent les notions de cryptographie à des protocoles concrets du Web.

Challenge *JWT Secrets* Ce challenge portait sur la sécurité des jetons d'authentification JSON Web Token (JWT). L'objectif était de découvrir la clé secrète utilisée pour signer un JWT mal configuré. Il illustre les risques liés à l'usage de clés faibles ou d'algorithmes non sécurisés (`HS256` au lieu de `RS256`). Une fois la clé récupérée, il est possible de forger un jeton valide et d'accéder à des ressources protégées.

Challenge *Decrypting TLS 1.2* Ce défi consistait à analyser une capture réseau (`tls2.cryptohack.org.pcapng`) pour déchiffrer une session TLS 1.2 à l'aide de la clé privée du serveur (`privkey.pem`).

Explication du challenge

La connexion utilise la suite de chiffrement `TLS_RSA_WITH_AES_256_GCM_SHA384`. Grâce à la clé RSA du serveur, il est possible de reconstituer le secret partagé et de déchiffrer le flux TLS.

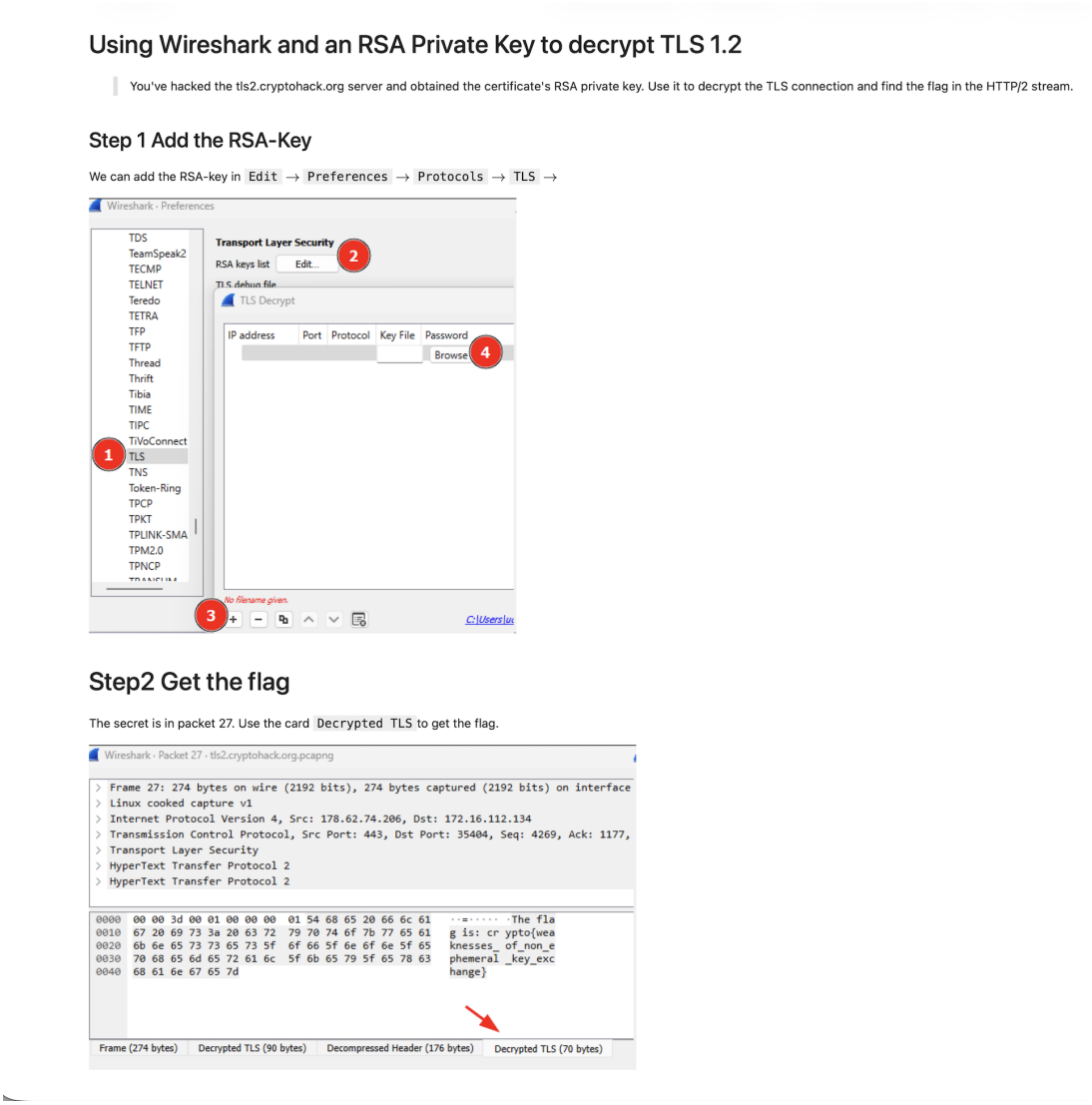

FIGURE 2 – Explication du fonctionnement de TLS 1.2 et de son déchiffrement.

Procédure avec Wireshark

Analyse Ce challenge démontre la dépendance du chiffrement TLS 1.2 à la sécurité des clés RSA utilisées pour l'échange. Une clé compromise ou trop courte permettrait

de déchiffrer des communications censées être confidentielles. L'évolution vers TLS 1.3 renforce considérablement la protection des données en rendant ces échanges éphémères et chiffrés dès le début de la connexion.

5 Bilan et apprentissages

5.1 Difficultés rencontrées

— Compréhension des mathématiques (totient, groupe, inverse mod n).

— Gestion du temps pour équilibrer théorie et mise en pratique.

— Quelques problèmes techniques lors de l'exécution de scripts RSA.

5.2 Compétences acquises

— Compréhension approfondie des algorithmes RSA, AES et Diffie-Hellman.

— Utilisation efficace de Git, Notion et LaTeX pour la gestion de projet.

— Développement de scripts Python pour des cas concrets de cryptographie.

6 Conclusion et perspectives

Ce projet a permis de relier la théorie de la cryptographie à la pratique à travers la résolution de défis variés. Il a favorisé la collaboration, l'autonomie et la rigueur documentaire.

Pour la suite :

— Approfondir la cryptographie moderne (ECC, post-quantique).

— Continuer à pratiquer pour renforcer la maîtrise des mathématiques appliquées à la sécurité.