Rapport de Projet : CryptoHack

UE Communication Scientifique

Victor Bailleul, Sébastien Leglise

20 octobre 2025

Université de Caen Normandie
Ecole Nationale Supérieure d'Ingénieurs de Caen
Année 2023-2024

UNIVERSITÉ CAEN NORMANDIE

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

Table des matières

1 Introduction

1.1 Contexte du projet

Ce rapport présente une série de challenges de cryptographie réalisés sur la plateforme *CryptoHack* (www.cryptohack.org). L'objectif est de documenter les approchesméthodologiques et les solutions techniques que nous avons mis en oeuvre en binome pour résoudre les défis proposés par la plateforme. En préambule, nous proposons de situer le contexte de ce travail en présentant d'une part le rôle fondamental de la cryptographie en cybersécurité, et d'autre part l'intérêt des plateformes de type *Capture The Flag* (CTF) comme outil d'apprentissage.

1.2 Place de la cryptographie en cybersécurité

La cryptographie est une discipline scientifique qui constitue l'un des piliers de la sécurité des systèmes d'information. Son objet est de développer des techniques permettant de protéger l'information contre toute modification ou accès non autorisé. En cybersécurité, la cryptographie vise à garantir plusieurs principes de sécurité fondamentaux.

La confidentialité Ce principe assure que seules les entités autorisées puissent accéder aux données. L'outil principal pour atteindre cet objectif est le chiffrement, qui consiste à transformer une information (le texte clair) en une forme inintelligible (le texte chiffré) à l'aide d'une clé secrète.

L'intégrité Ce principe garantit que les données n'ont pas été altérées ou corrompues, que ce soit de manière accidentelle ou intentionnelle, durant leur stockage ou leur transmission. Les fonctions de hachage et les codes d'authentification de message (MAC) sont des mécanismes cryptographiques courants pour assurer l'intégrité.

L'authenticité Ce principe permet de vérifier l'identité d'une entité (un utilisateur, un serveur, etc.). Les certificats numériques et les signatures numériques sont des exemples de techniques cryptographiques assurant l'authentification.

La non-répudiation Ce principe empêche une entité de nier avoir effectué une action, comme l'envoi d'un message ou la validation d'une transaction. La signature numérique est le mécanisme de base pour fournir cette garantie.

De la sécurisation des communications sur Internet (protocoles TLS/SSL) à la protection des données stockées, en passant par la sécurisation des transactions financières et la protection de la vie privée, les applications de la cryptographie sont omniprésentes et critiques. L'étude de ses mécanismes et de leurs implémentations est par conséquent essentielle pour toute personne souhait travailler en cybersécurité.

1.3 Les plateformes de CTF dans l'apprentissage

Les challenges de type *Capture The Flag* (CTF) sont des exercices de cybersécurité offensifs et/ou défensifs. Les participants doivent résoudre des épreuves pour trouver une

chaîne de caractères secrète, appelée *flag*, cachée dans un système, un fichier ou encore un chiffré.

Les plateformes de CTF comme *root-me*, *TryHackMe* ou encore *CryptoHack* sont des environnements d'apprentissage pratiques et contrôlés où les utilisateurs peuvent appliquer des connaissances théoriques à travers des exercices thématiques. Cette approche par la pratique favorise une compréhension approfondie des vulnérabilités et des techniques d'exploitation.

Ces plateformes couvrent un large éventail de domaines de la cybersécurité, tels que l'exploitation de binaires, la rétro-ingénierie (*reverse engineering*), l'analyse forensique (*digital forensics*), la sécurité des applications web et la cryptographie.

1.4 Présentation de *CryptoHack*

CryptoHack est une plateforme en ligne dédiée à l'apprentissage de la cryptographie moderne. Elle propose une série de challenges de type CTF qui permettent aux utilisateurs de se familiariser avec les principes, les algorithmes et les attaques cryptographiques. La plateforme propose une approche témathique et progressive dans sa difficulté.

Les challenges sur *CryptoHack* se concentrent sur l'identification et l'exploitation de failles dans des implémentations de protocoles et d'algorithmes cryptographiques largement utilisés, tels que AES, RSA ou Diffie-Hellman.

Chaque challenge est conçu pour illustrer un concept spécifique ou une vulnérabilité connue. Les utilisateurs sont amenés à interagir avec des serveurs distants, à analyser du code source et à développer leurs propres scripts, principalement en Python, pour automatiser les attaques et récupérer les *flags*.

2 Gestion de projet

2.1 Organisation de la charge de travail

Afin d'organiser notre projet, nous avons établi un calendrier de travail s'étendant sur les trois semaines précédant la date de rendu. Cette planification visait à définir un cadre temporel clair et à assurer une répartition équilibrée de la charge de travail.

La première phase a été consacrée à l'exploration du module *General* de la plateforme *CryptoHack*. L'objectif était de nous familiariser avec l'environnement, d'identifier les outils techniques nécessaires et d'évaluer la nature des challenges. Pour optimiser notre progression, nous nous sommes réparti les sous-modules : l'un s'est chargé des sections *Encoding* et *XOR*, tandis que l'autre a traité les sections *Mathematics* et *Data Formats*. Cette étape s'est achevée à l'échéance que nous avions fixée, deux semaines avant le rendu, nous permettant de synchroniser nos avancées.

Lors de la deuxième phase, nous avons abordé deux thématiques de la cryptographie asymétrique : RSA et Diffie-Hellman. En conservant une méthodologie de travail parallèle, chaque membre du binôme s'est concentré sur l'une de ces catégories, avec une échéance fixée à une semaine du rendu.

Enfin, la dernière semaine a été entièrement dédiée à la rédaction de ce rapport, nous permettant de synthétiser et de documenter l'ensemble des travaux réalisés.

2.2 Gestion d'un dépôt GitHub

Dès le début du projet, nous avons créé un dépôt GitHub partagé, afin de faciliter l'échange des productions écrites. Ce dépôt nous a permis d'organiser nos scripts de résolution, nos notes et les différents fichiers associés aux challenges.

Chaque membre disposait d'un accès complet au dépôt et pouvait le mettre à jour dès qu'il estimait qu'une contribution était suffisamment stable ou pertinente. Les *commits* étaient accompagnés de messages explicites décrivant les modifications apportées, ce qui facilitait la compréhension de l'évolution du projet.

2.3 Méthodologie GitHub

Pour garantir une collaboration efficace, notre méthodologie de travail s'est appuyée sur une structure de dépôt claire et des pratiques Git rigoureuses.

Chaque script de résolution a été classé dans un répertoire thématique correspondant à sa catégorie (*Encoding/*, *XOR/*, etc.), assurant ainsi une organisation logique des livrables. Nous avons maintenu un rythme de mise à jour régulier, chaque membre étant responsable de pousser ses contributions après validation.

Afin de minimiser les conflits de fusion (*merge conflicts*), nous avons limité le travail simultané sur les fichiers communs. Cette approche a été facilitée par une communication directe pour la coordination et par des choix structurels, comme l'adoption d'une architecture modulaire pour ce rapport en LaTeX plutôt que de travailler sur un fichier unique. Cette méthode a garanti la cohérence et l'intégrité du projet tout au long de son développement.

3 Challenges : General

3.1 Encodage (*Encoding*)

Cette première sous-partie de la catégorie *General* aborde les différentes méthodes de représentation de l'information, essentielles au transport et à l'échange de données. La maîtrise des conversions entre des formats comme le binaire, l'hexadécimal ou le *Base64* constitue un prérequis indispensable pour aborder des défis cryptographiques plus complexes. Il est fondamental de bien distinguer l'encodage du chiffrement : le premier est une transformation de format, publique et réversible, qui ne vise pas à garantir la confidentialité, contrairement au second.

Nous avons décidé de présenter le dernier challenge de cette partie, nommé *Encoding challenge*.

3.1.1 Objectifs

L'objectif de ce challenge consiste à développer un script pour automatiser l'interaction avec un serveur distant de *CryptoHack*. Le processus implique la réception de données encodées selon diverses méthodes (*Base64*, hexadécimal, *ROT13*, *BigInt*, et UTF-8), leur décodage approprié, puis le renvoi de la valeur décodée au serveur.

Pour valider le challenge et obtenir le flag, il est impératif d'exécuter cette séquence de réception, décodage et renvoi avec succès cent fois consécutives. Cette contrainte requiert une solution automatisée, capable de gérer dynamiquement les différents types d'encodage rencontrés.

3.1.2 Méthode

Le challenge met à disposition un script partiel qui présente la manière d'envoyer et recevoir des données avec le serveur, ainsi que le script exécuté côté serveur (cf. Annexe A) pour vérifier les valeurs qui lui ont été transmises. Ces scripts nous ont permis de comprendre le format des données transmises.

La communication avec le serveur s'effectue via l'échange d'objets au format JSON. Pour chaque itération du challenge, le serveur envoie une requête structurée de la manière suivante :

```
{
    "type": "type_d_encodage",
    "encoded": "donnees_encodees"
}
```

Notre script doit alors analyser cette requête, appliquer la méthode de décodage appropriée, et renvoyer la solution au serveur sous le format JSON attendu :

```
{
    "decoded": "donnees_decodees"
}
```

Le script côté serveur vérifie alors la validité des données décodées, puis si cela est valide, crée un nouveau challenge. Si notre script résout cent challenges, la prochaine requête au serveur permettra d'afficher le *flag* dans la sortie standard.

3.1.3 Résultat

FIGURE 1 – Capture d'écran illustrant l'obtention du *flag* après le décodage automatique de cent chaînes de caractères envoyées par le serveur de *CryptoHack*.

Le script présenté en Annexe B du rapport a permis la bonne résolution du challenge (cf. Figure 1) et l'obtention du *flag* suivant :

`crypto{3nc0d3 d3c0d3 3nc0d3}`

3.2 XOR

La deuxième sous-partie de la catégorie *General* est consacrée à l'opération XOR (*ou* exclusif), un concept fondamental en cryptographie constituant l'une des briques de base de nombreux algorithmes de chiffrement. Sa simplicité de mise en œuvre et ses propriétés mathématiques uniques en font un outil puissant pour manipuler l'information.

Comprendre le fonctionnement du XOR est une étape cruciale, car il se situe à la frontière entre l'opération logique et le chiffrement. L'une de ses propriétés essentielles est sa réversibilité : appliquer deux fois la même clé XOR à une donnée permet de retrouver la donnée originale ($A \oplus K \oplus K = A$). Cette caractéristique est au cœur de son utilisation dans des chiffrements à flux comme le *One-Time Pad*.

Nous avons décidé de présenter le challenge le plus représentatif de cette partie, nommé *Lemur XOR*.

3.2.1 Objectifs

Le but de ce challenge est de retrouver le *flag*, à partir de deux images fournies : `lemur.png` et `flag.png` (cf. Figure 2). L'énoncé nous apprend que ces deux images ont été chiffrées avec l'opération XOR en utilisant la même clé secrète.

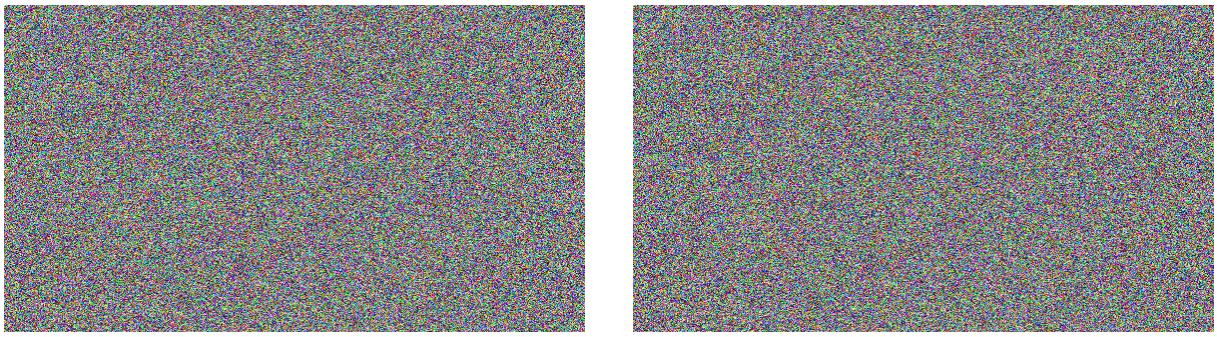

FIGURE 2 – Les deux images chiffrés avec une clé secrète commune. Sur la gauche `flag.png`, sur la droite `lemur.png`.

Le principe de résolution repose sur le fait qu'appliquer deux fois un XOR avec la même clé annule l'opération. En effectuant un XOR entre les deux images chiffrées que nous possédons, la clé secrète commune s'élimine, ne laissant que le résultat du XOR entre les deux images originales. C'est sur cette image combinée que le *flag* devrait devenir visible.

3.2.2 Méthode

Pour résoudre ce challenge, nous avons utilisé un script en Python avec la bibliothèque de manipulation d'images `Pillow`. Nous avons commencé par charger les deux images, `lemur.png` et `flag.png`. Conformément aux instructions, nous avons appliqué l'opération XOR pixel par pixel sur les valeurs de couleur RVB.

Nous avons donc parcouru les deux images simultanément et calculé la nouvelle valeur de chaque pixel en effectuant un XOR sur ses composantes rouge, verte et bleue. Nous avons ensuite utilisé ces nouveaux pixels pour construire une image de sortie de mêmes dimensions que nous avons sauvegardée.

3.2.3 Résultat

FIGURE 3 – Ceci est le schéma explicatif de notre méthode.

Le script présenté en Annexe C du rapport a permis la résolution du challenge en obtenant une image (cf. Figure 3) affichant le *flag* suivant :

`crypto{X0Rly_n0t!}`

3.3 Data formats

Pour présenter cette gatégorie, nous avons choisi le challenge *Transparency*. Pour le résoudre, nous nous appuyons sur le principe de *Certificate Transparency* (CT), une mesure de sécurité imposée aux Autorités de Certification (CA) pour garantir la transparence dans la délivrance des certificats TLS.

Un certificat TLS (souvent appelé certificat SSL ou certificat numérique) remplit deux fonctions principales : il authentifie l'identité d'un site web ou d'un domaine auprès des clients (navigateurs, applications) et il permet d'établir une connexion chiffrée (TLS) entre le client et le serveur, garantissant la confidentialité et l'intégrité des données échangées.

Les *CT logs* sont des bases de données publiques de type *append-only* (où l'on ne peut qu'ajouter des entrées) dans lesquelles les certificats émis par les CA sont enregistrés. Aujourd'hui, les principales CA publient chaque certificat dans au moins deux logs CT publics pour qu'il soit accepté par les navigateurs modernes. Nous exploitons ces journaux pour retrouver des certificats correspondant à une clé publique donnée.

Ces logs étant audités et surveillés, nous pouvons vérifier les entrées, détecter d'éventuels certificats inattendus et contrôler la cohérence de la structure afin de s'assurer qu'aucune entrée n'est dissimulée.

3.3.1 Objectifs

Nos objectifs sont doubles. D'abord, nous voulons retrouver le sous-domaine de cryptohack.org qui utilise la même clé publique que celle fournie dans le fichier `transparency.pem` au sein de son certificat TLS. Ensuite, en visitant ce sous-domaine, nous souhaitons obtenir le flag.

À travers ce challenge, nous visons d'abord à comprendre le fonctionnement d'un certificat TLS et sa structure (clé publique, signature, chaîne de confiance, etc.), ensuite de découvrir le système des *Certificate Transparency logs*, puis d'apprendre à faire correspondre une clé publique à un certificat et enfin d'utiliser des outils d'investigation SSL/TLS et de recherche de certificats.

3.3.2 Méthode

Nous partons d'un fichier au format PEM (*Privacy-Enhanced Mail*), un format standard pour les clés cryptographiques qui utilise l'encodage *Base64*. La clé publique fournie est explicitée Figure 4.

```
    -----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuYj06m5q4M8SsEQwKX+5
NPs2lyB2k7geZw4rP68eUZmqODeqxDjv5mlLY2nz/RJsPdks4J+y5t96KAyo3S5g
mDqEOMG7JgoJ9KU+4HPQFzP9C8Gy+hisChdo9eF6UeWGTioazFDIdRUK+gZm81c1
iPEhOBIYu3Cau32LRtv+L9vzqre0Ollf7oeHqcbcMBIKL6MpsJMG+neJPnICI36B
ZZEMu6v6f8zIKuB7VUHAbDdQ6tsBzLpXz7XPBUeKPa1Fk8d22EI99peHwWt0RuJP
0QsJnsa4oj6C6lE+c5+vVHa6jVsZkpl2PuXZ05a69xORZ4oq+nwzK8O/St1hbNBX
sQIDAQAB
-----END PUBLIC KEY-----
```

Figure 4 – Clé publique au format PEM fournie pour le challenge. Elle constitue le point de départ de notre recherche du certificat associé.

Notre méthode s'est déroulée en deux temps. Dans un premier temps, nous avons analysé la problématique afin de comprendre comment relier l'empreinte SHA-256 d'une clé publique à un certificat TLS complet. Nous avons établi qu'un certificat X.509 contient une clé publique et que sa représentation binaire (DER) sert de base au calcul de l'empreinte utilisée par des services comme `crt.sh`.

Forts de cette analyse, nous avons ensuite conçu un script Python dont les objectifs étaient les suivants : générer l'empreinte SHA-256 de la clé publique fournie, interroger les journaux de transparence des certificats pour retrouver les certificats correspondants, télécharger le certificat identifié, en extraire le nom de domaine, et enfin y accéder pour récupérer le flag.

3.3.3 Résultats

Le script décrit en Annexe D nous a permis d'obtenir le sous-domaine recherché et d'identifier le *flag* (cf. Table 1) permettant la résolution du challenge.

Empreinte (SHA-256)	`29ab37df0a4e4d252f0cf12ad854bede59038fdd9cd652cbc5c222edd26d77d2`
Sous-domaine identifié	`thetransparencyflagishere.cryptohack.org`
Flag obtenu	`crypto{thx_redpwn_for_inspiration}`

Table 1 – Résultats obtenus pour le challenge Transparency.

4 Challenges : Diffie-Hellman

4.1 Introduction à l'échange de clés Diffie-Hellman

Cette catégorie aborde le protocole d'échange de clés de Diffie-Hellman, un mécanisme de cryptographie asymétrique. Proposé en 1976, il a pour objectif de résoudre le problème de la distribution de clés sur un canal de communication non sécurisé.

Le principe de ce protocole repose sur l'utilisation de fonctions mathématiques à sens unique, dont le calcul est aisé dans une direction mais calculatoirement difficile à inverser. Spécifiquement, Diffie-Hellman s'appuie sur la difficulté du problème du logarithme discret dans un groupe fini. Ce procédé permet à deux interlocuteurs d'établir une clé secrète partagée sans transmission préalable de celle-ci, y compris en présence d'un adversaire observant la communication.

Les challenges de cette section visent à analyser les fondements mathématiques de ce protocole, ainsi que les vulnérabilités pouvant résulter d'une implémentation incorrecte ou d'un choix de paramètres inadéquat.

4.2 L'attaque de l'homme du milieu (*Man-in-the-Middle*)

Cette sous-partie examine une vulnérabilité du protocole Diffie-Hellman : l'attaque de l'homme du milieu (*Man-in-the-Middle* ou MitM). Le protocole de base, dans sa forme originelle, ne fournit aucun mécanisme d'authentification des interlocuteurs. Il permet de s'assurer que la clé partagée est secrète, mais pas de vérifier l'identité de la personne avec qui on la partage.

Une attaque MitM exploite cette absence d'authentification. Un adversaire se positionne entre les deux communicants, intercepte leurs messages publics et établit une session Diffie-Hellman distincte avec chacun d'eux. Chaque interlocuteur génère alors une clé secrète partagée avec l'attaquant, tout en croyant communiquer directement l'un avec l'autre.

L'adversaire peut alors déchiffrer les messages, les lire, les modifier, puis les rechiffrer avec la clé de l'autre session avant de les transmettre au destinataire final. Les challenges de cette section illustrent comment cette interception est mise en œuvre et comment elle compromet la confidentialité et l'intégrité de l'échange.

Pour illustrer ce type d'attaque, nous présentons le challenge *Export-Grade*.

4.2.1 Objectifs

L'objectif de ce challenge est de compromettre la confidentialité d'un échange sécurisé par le protocole Diffie-Hellman. Le scénario repose sur une attaque de l'homme du milieu (*Man-in-the-Middle*) durant la phase de négociation des paramètres cryptographiques.

La vulnérabilité exploitée est la capacité pour un attaquant d'influencer le choix des paramètres de groupe utilisés par les deux correspondants. En les contraignant à utiliser un groupe de petite taille (64 bits), le problème mathématique du logarithme discret, sur lequel repose la sécurité du protocole, devient calculatoirement soluble. La résolution de ce problème permet de retrouver une clé privée, et par conséquent la clé secrète partagée utilisée pour chiffrer la communication.

4.2.2 Méthode

Notre attaque s'est déroulée en plusieurs étapes. Premièrement, nous avons activement intercepté la communication initiale d'Alice, qui proposait une liste de groupes cryptographiques. Nous avons altéré ce message en ne conservant que le groupe le plus faible, caractérisé par un module de 64 bits, avant de le transmettre à Bob. L'acceptation de cette unique option par Bob a été relayée à Alice, établissant ainsi un accord sur un canal de communication affaibli.

FIGURE 5 – Capture d'écran de la communication interceptée entre Alice et Bob en tant que *Man-In-The-Middle*.

Une fois cet accord forcé, notre rôle est devenu passif. Nous avons collecté les paramètres publics de l'échange : le module p, le générateur g, et les clés publiques d'Alice (A) et de Bob (B). La faible taille du module p a alors permis de résoudre le problème du logarithme discret. L'échange réalisé avec Alice et Bob est illustré Figure 5.

À l'aide d'un script Python, nous avons calculé la clé privée a d'Alice à partir des valeurs publiques p, g et A. La connaissance de cette clé privée nous a permis de reconstituer la clé secrète partagée ($s = B^a \pmod p$). Cette dernière a servi à dériver la clé de session AES, avec laquelle nous avons déchiffré le message final pour obtenir le *flag*.

4.2.3 Résultat

L'attaque par manipulation des paramètres a réussi. En forçant l'usage d'un groupe faible, nous avons pu calculer la clé privée, reconstituer la clé de session et déchiffrer le message, révélant le *flag* suivant :

`crypto{d0wn6r4d35_4r3_d4n63r0u5}`

Le script développé pour la résolution du challenge est présenté Annexe E.

4.3 Théorie des groupes

Cette sous-partie aborde les structures mathématiques qui sous-tendent de nombreux protocoles de cryptographie asymétrique : les groupes. En algèbre abstraite, un groupe est un ensemble d'éléments muni d'une opération binaire qui satisfait à des axiomes spécifiques (fermeture, associativité, existence d'un élément neutre et d'un inverse pour chaque élément).

La sécurité de protocoles comme Diffie-Hellman ne repose pas sur les nombres en tant que tels, mais sur les propriétés structurelles de ces groupes mathématiques. Des concepts comme l'ordre d'un groupe, l'ordre d'un élément, et la notion de générateur d'un groupe cyclique sont des composantes directes de l'implémentation et de l'analyse de sécurité de ces systèmes.

Les challenges de cette section ont pour objectif d'étudier ces propriétés. Ils illustrent comment les caractéristiques d'un groupe, ou le choix de ses paramètres, peuvent influencer la robustesse d'un schéma cryptographique.

Pour illustrer cette section, nous présentons le challenge *Additive*.

4.3.1 Objectifs

L'objectif de ce challenge est de calculer une clé secrète partagée dans une implémentation du protocole Diffie-Hellman utilisant un groupe additif. Contrairement à l'implémentation classique qui utilise un groupe multiplicatif d'entiers modulo un nombre premier, ce challenge transpose le problème dans une structure où l'opération de groupe est l'addition.

Le principe de sécurité reste fondé sur la difficulté d'inverser une fonction à sens unique. Ici, l'opération équivalente à l'exponentiation est la multiplication scalaire. Le but est de résoudre l'équivalent du problème du logarithme discret dans ce contexte additif afin de retrouver une clé privée, puis de reconstituer la clé secrète partagée, qui constitue le *flag*.

4.3.2 Méthode

Le challenge nous fournit les paramètres publics d'un échange Diffie-Hellman additif : un module premier p, un générateur g, ainsi que les clés publiques d'Alice (A) et de Bob (B). La relation qui lie la clé privée a à la clé publique `A` n'est plus $A = g^a \pmod p$, mais $A = a \cdot g \pmod p$.

Le problème du logarithme discret se traduit ici par la résolution de l'équation $A \equiv a \cdot g \pmod p$ pour trouver l'inconnue a. Cette équation est une congruence linéaire qui se résout efficacement en calculant l'inverse modulaire de g modulo p. Nous avons donc déterminé la clé privée d'Alice en calculant $a = A \cdot g^{-1} \pmod p$.

Une fois la clé privée a obtenue, nous avons pu calculer la clé secrète partagée en appliquant l'opération du groupe avec la clé publique de Bob. L'opération étant la multiplication scalaire, la clé secrète s est obtenue par la formule $s = a \cdot B \pmod p$.

4.3.3 Résultat

L'analyse de la structure de groupe additive a permis d'identifier la méthode de résolution appropriée. Le calcul de l'inverse modulaire nous a donné accès à la clé privée, menant directement à la reconstitution de la clé secrète partagée. Nous avons obtenu le *flag* suivant :

`crypto{cycl1c_6r0up_und3r_4dd1710n?}`

5 Challenges : RSA

5.1 Introduction au cryptosystème RSA

Dans cette catégorie, nous abordons le cryptosystème à clé publique RSA, décrit pour la première fois en 1977. C'est l'un des systèmes de chiffrement asymétrique les plus connus et les plus utilisés dans le monde.

Nous nous appuyons sur le principe fondamental de RSA : la difficulté de factoriser de grands nombres composés en leurs facteurs premiers. Cette propriété mathématique

nous permet de garantir la sécurité du système, à condition de choisir et d'implémenter correctement les paramètres.

Le cryptosystème RSA possède deux usages principaux. Le premier est le chiffrement, où nous publions une clé publique permettant à d'autres de nous envoyer des messages chiffrés, que nous pouvons ensuite déchiffrer grâce à notre clé privée. Le second est la signature numérique, qui nous permet de signer un message avec notre clé privée, afin que quiconque puisse vérifier l'authenticité et l'intégrité du message à l'aide de la clé publique correspondante.

Les challenges de cette section ont pour objectif de nous faire explorer les fondements du cryptosystème RSA, tout en mettant en évidence les erreurs courantes et les failles d'implémentation qui ont, dans certains cas, conduit à des attaques réelles.

5.2 RSA Multi-Factor Attack

Dans le challenge *ManyPrime*, nous disposons d'un fichier contenant un module RSA n, un exposant public e et un ciphertext ct. L'énoncé nous donne deux indices explicites : l'auteur indique qu'il utilisera "over 30" facteurs premiers et renvoie vers la méthode des courbes elliptiques (ECM). Ces indications orientent notre stratégie d'attaque : extraire un grand nombre de facteurs premiers relativement petits grâce à l'ECM, puis retrouver la clé privée pour déchiffrer le message.

5.2.1 Objectifs

Notre objectif est de récupérer le flag contenu dans le ciphertext. Concrètement, nous devons factoriser n en l'ensemble de ses facteurs premiers, puis calculer $d = e^{-1} \pmod{\varphi(n)}$ où $\varphi(n) = \prod_i (p_i - 1)$, et enfin déchiffrer ct pour obtenir le *plaintext* (le *flag*).

5.2.2 Méthode

Notre méthode de résolution exploite une vulnérabilité dans la conception du module RSA : sa construction à partir d'un grand nombre de petits facteurs premiers. Cette approche affaiblit considérablement sa résistance aux algorithmes de factorisation modernes comme l'ECM (*Elliptic Curve Method*), qui sont particulièrement efficaces pour trouver des facteurs de taille modeste.

Pour la mise en œuvre pratique, nous avons configuré un environnement Python avec les bibliothèques `primefac`, pour son implémentation de l'ECM, et `pycryptodome` pour les opérations arithmétiques modulaires. Le processus de factorisation a été réalisé via une boucle itérative qui invoque `primefac.ecm()` pour extraire progressivement chaque facteur premier p du module n, jusqu'à sa décomposition complète.

Une fois l'ensemble des facteurs premiers obtenu, nous avons validé leur produit pour nous assurer qu'il correspondait bien au module n original. Nous avons ensuite calculé la fonction indicatrice d'Euler $\varphi(n) = \prod(p_i - 1)$ et déterminé l'exposant privé d en résolvant l'équation $e \cdot d \equiv 1 \pmod{\varphi(n)}$. Finalement, le message original a été restauré en déchiffrant le ciphertext ct par exponentiation modulaire ($pt = ct^d \pmod{n}$).

5.2.3 Résultat

Notre script, présenté en Annexe F extrait plusieurs facteurs premiers et les affiche sur le terminal. Les valeurs obtenues sont présentées Table 2.

$p_1 = 17281246625998849649$	$p_2 = 9389357739583927789$	$p_3 = 10638241655447339831$
$p_4 = 16656402470578844539$	$p_5 = 14100640260554622013$	$p_6 = 9303850685953812323$
$p_7 = 17174065872156629921$	$p_8 = 14963354250199553339$	...

Table 2 – Facteurs premiers extraits par ECM

Après les calculs de vérification et de déchiffrement, nous obtenons le message en clair suivant :

`crypto{700_m4ny_5m4ll_f4c70r5}`

5.3 Cryptanalyse RSA avec exposant faible

Dans cette sous-partie, nous présentons l'analyse du challenge *Vote for Pedro* qui illustre une vulnérabilité cryptographique classique dans l'implémentation du chiffrement RSA. Ce challenge met en avant les dangers liés à l'utilisation d'un exposant public faible, particulièrement lorsque combiné avec des messages de petite taille et l'absence de mécanisme de padding approprié. En le résolvant, nous montrons comment il est possible de forger des signatures sans disposer de la clé privée, compromettant ainsi l'intégrité du système d'authentification.

5.3.1 Objectifs

Notre objectif principal dans ce challenge est d'obtenir un flag en votant pour Pedro avec une signature valide émise par Alice. Le serveur vérifie la signature du vote en utilisant la clé publique d'Alice et retourne le flag uniquement si le message déchiffré correspond exactement à la chaîne **VOTE FOR PEDRO**. La difficulté réside dans l'impossibilité d'accéder à la clé privée d'Alice pour signer le message légitimement.

5.3.2 Méthode

Le challenge nous fournit deux éléments essentiels : le script du serveur et la clé publique d'Alice. En analysant le script serveur (cf. Annexe G), nous révélons le mécanisme de vérification des signatures. Le serveur reçoit un vote signé sous forme hexadécimale, applique la vérification RSA en calculant le vote à la puissance de l'exposant public modulo N, puis convertit le résultat en bytes. Le message est ensuite extrait en prenant la partie suivant le dernier octet nul avant d'être comparé avec la chaîne attendue.

La clé publique d'Alice présente une particularité cruciale : l'exposant public $e = 3$. Cette valeur faible, combinée avec l'absence de padding robuste, crée une vulnérabilité exploitable en permettant une attaque par extraction de racine cubique. En effet, lorsque le message original est inférieur à la racine cubique de N, l'opération de signature peut être inversée sans recours au modulo, rendant possible la forge de signature.

Le principe mathématique de notre attaque repose sur l'équation de vérification RSA standard, $\text{vote}^e \pmod N$. L'exposant public étant faible ($e = 3$), si le message M est suffisamment petit pour que $M < \sqrt[3]{N}$, l'opération modulo devient superflue. L'équation se simplifie alors en $\text{vote}^3 = M$, nous permettant d'obtenir la signature par un simple calcul de racine cubique. Le message **VOTE FOR PEDRO** n'étant pas un cube parfait, notre stratégie a été de trouver un entier S tel que S^3 partage les mêmes bits de poids faible que le message, ce qui est suffisant car le serveur ignore les bits de poids fort lors de la vérification.

Pour ce faire, nous avons implémenté un algorithme, `cube_root_2_pow`, qui calcule cette racine cubique bit par bit. En procédant par raffinement successif, il garantit la convergence vers une solution valide. Sur le plan technique, notre script établit une connexion socket avec le serveur, convertit le message cible en entier, puis exécute l'algorithme pour calculer la signature forgée. Celle-ci est ensuite convertie en hexadécimal et transmise dans un payload JSON pour validation.

5.3.3 Résultat

L'exécution de notre script de résolution, présenté dans l'Annexe .. produit le résultat suivant après connexion au serveur :

```
Place your vote. Pedro offers a reward to anyone who votes for him!
{"flag": "crypto{y0ur_v0t3_i5_my_v0t3}"}
```

6 Conclusion

Annexes

A - Script exécuté côté serveur pour le challenge *Encoding challenge*

```
from Crypto.Util.number import bytes_to_long, long_to_bytes
from utils import listener # this is cryptohack's server-side module and not part of python
import base64
import codecs
import random

FLAG = "crypto{????????????????????}"
ENCODINGS = [
    "base64",
    "hex",
    "rot13",
    "bigint",
    "utf-8",
]
with open('/usr/share/dict/words') as f:
    WORDS = [line.strip().replace("'", "") for line in f.readlines()]


class Challenge():
    def __init__(self):
        self.no_prompt = True # Immediately send data from the server without waiting for user input
        self.challenge_words = ""
        self.stage = 0

    def create_level(self):
        self.stage += 1
        self.challenge_words = "_".join(random.choices(WORDS, k=3))
        encoding = random.choice(ENCODINGS)

        if encoding == "base64":
            encoded = base64.b64encode(self.challenge_words.encode()).decode() # wow so encode
        elif encoding == "hex":
            encoded = self.challenge_words.encode().hex()
        elif encoding == "rot13":
            encoded = codecs.encode(self.challenge_words, 'rot_13')
        elif encoding == "bigint":
            encoded = hex(bytes_to_long(self.challenge_words.encode()))
        elif encoding == "utf-8":
```

```
            encoded = [ord(b) for b in self.challenge_words]

        return {"type": encoding, "encoded": encoded}

    #
    # This challenge function is called on your input, which must be JSON
    # encoded
    #
    def challenge(self, your_input):
        if self.stage == 0:
            return self.create_level()
        elif self.stage == 100:
            self.exit = True
            return {"flag": FLAG}

        if self.challenge_words == your_input["decoded"]:
            return self.create_level()

        return {"error": "Decoding fail"}


import builtins; builtins.Challenge = Challenge # hack to enable challenge to be run locally, see https://cryptohack.org/faq/#listener
listener.start_server(port=13377)
```

B - Script de résolution du challenge *Encoding challenge*

```
from pwn import * # pip install pwntools
from Crypto.Util.number import *
import codecs
import base64
import json

r = remote('socket.cryptohack.org', 13377, level = 'debug')

def json_recv():
    line = r.recvline()
    return json.loads(line.decode())

def json_send(hsh):
    request = json.dumps(hsh).encode()
    r.sendline(request)


for i in range(100):
    received = json_recv()

    print("Received type: ")
    print(received["type"])
    print("Received encoded value: ")
    print(received["encoded"])

    type = received["type"]
    encoded = received["encoded"]

    if type == "base64":
        decoded = base64.b64decode(encoded).decode()
    elif type == "hex":
        decoded = bytes.fromhex(encoded).decode()
    elif type == "rot13":
        decoded = codecs.decode(encoded, 'rot_13')
    elif type == "bigint":
        decoded = long_to_bytes(int(encoded, 16)).decode()
    elif type == "utf-8":
        decoded = bytes(encoded).decode()

    to_send = {
    "decoded": decoded
    }

    json_send(to_send)

json_recv()
```

C - Script de résolution du challenge *Lemur XOR*

```
from PIL import Image
import sys
from io import BytesIO

def get_rgb_bytes(png_path):
    with Image.open(png_path) as img:
        img = img.convert('RGB')
        return img.tobytes()

lemur = get_rgb_bytes("lemur_ed66878c338e662d3473f0d98eedbd0d.png")
flag = get_rgb_bytes("flag_7ae18c704272532658c10b5faad06d74.png")

xored_result = []
for i, c in enumerate(lemur):
    xored_byte = c ^ flag[i % len(flag)]
    xored_result.append(xored_byte)

with Image.open("lemur_ed66878c338e662d3473f0d98eedbd0d.png") as img:
    width, height = img.size

result_img = Image.frombytes('RGB', (width, height), bytes(xored_result))
result_img.save("xored_result.png")
```

D - Script de résolution du challenge *Transparency*

```
from Crypto.PublicKey import RSA
from OpenSSL.crypto import load_certificate, FILETYPE_PEM
import hashlib
import json
import requests

f = open("transparency_afff0345c6f99bf80eab5895458d8eab.pem")
key = RSA.import_key(f.read()).public_key()


sha256 = hashlib.sha256(key.exportKey(format="DER"))
fp = sha256.hexdigest()
print("fingerprint:", fp)

user_agent = 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/40.1'
url = "https://crt.sh/?spkisha256={hash}&output=json"

req = requests.get(url.format(hash=fp), headers={'User-Agent': user_agent})
content = req.content.decode('utf-8')
data = json.loads(content)
id = data[0]["id"]
download_url = "https://crt.sh/?d={id}"
req = requests.get(download_url.format(id=id), headers={'User-Agent': user_agent})
PEMcert = req.content.decode('utf-8')
    #obtenir le nom du sous domaine
cert = load_certificate(FILETYPE_PEM, PEMcert)
CN = cert.get_subject().commonName
print("Nom du domaine: ", CN)

flag_url = "https://" + CN
req = requests.get(flag_url, headers={'User-Agent': user_agent})
flag = req.content.decode('utf-8')
print("Flag:", flag)
```

E - Script de résolution du challenge *Export grade*

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
import hashlib
import json
def is_pkcs7_padded(message):
    padding = message[-message[-1]:]
    return all(padding[i] == len(padding) for i in range(0, len(
   padding)))


def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    # Derive AES key from shared secret
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    # Decrypt flag
    ciphertext = bytes.fromhex(ciphertext)
    iv = bytes.fromhex(iv)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)

    if is_pkcs7_padded(plaintext):
        return unpad(plaintext, 16).decode('utf-8')
    else:
        return plaintext.decode('utf-8', errors='replace')

p_hex = "0xde26ab651b92a129"
g_hex = "0x2"
A_hex = "0x674f9bbabc48cd8d"
B_hex = "0xb0474b841afb4d6e"
iv = "42ec73835ae43797fa73803c035158fb"
encrypted_flag = "
   a398f814b44b69a83d2c9a0d8c35eaf65bc39b405af361a1f6ebeaec967c5809
   "

p = int(p_hex, 16)
g = int(g_hex, 16)
public_A = int(A_hex, 16)
public_B = int(B_hex, 16)

from sympy.ntheory.residue_ntheory import *
a = discrete_log(p, public_A, g)
print(a)
shared_secret = pow(public_B, a, p)
print(decrypt_flag(shared_secret, iv, encrypted_flag))
```

F - Script de résolution du challenge *ManyPrime*

```
from Crypto.Util.number import inverse, long_to_bytes
import primefac
import math

n = 58064239189884319292956385687089779 9...
e = 65537
ct = 32072149053462443414999372352732297...

factors = []
current = n
while current > 1:
    if primefac.isprime(current):
        factors.append(current)
        break
    else:
        p = primefac.ecm(current)
        print(f"p:{p}")
        factors.append(p)
        current //= p

product = 1
for p in factors:
    product *= p
assert n == product

phi = 1
for p in factors:
    phi *= (p - 1)

d = inverse(e, phi)
pt = pow(ct, d, n)
decrypted = long_to_bytes(pt)
print(decrypted)
```

G - Script de résolution du challenge *Vote for Pedro*

```
from Crypto.Util.number import bytes_to_long, long_to_bytes
import socket
import json

host = 'socket.cryptohack.org'
port = 13375
N = 22266616657574989868109324252160663470925207690694094953...
e = 3

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
data = sock.recv(1024)
print(data)

T = b'VOTE FOR PEDRO'
c = bytes_to_long(T)

def cube_root_2_pow(c, k_max):
    s = c % 8
    for k in range(3, k_max):
        diff = s**3 - c
        d = diff // (2**k)
        t = (-d) % 2
        s = s + t * (2**k)
    return s

s = cube_root_2_pow(c, len(T)*8 + 8)
sign_hex = long_to_bytes(s).hex()

payload = {
    "option": "vote",
    "vote": sign_hex
}

sock.send(json.dumps(payload).encode())
flag = sock.recv(1024)
print(flag)
```