istic Informatique Électronique

UNIVERSITÉ DE RENNES 1

BCS

Chiffrement Authentifié

Master 2 Cybersécurité

2018 - 2019

Encadré par :
Patrick DERBEZ

Réalisé par :
Alexis LE MASLE

Table Des Matières

Introduction

Le but de ce projet est de créer un chiffrement authentifié basé sur le chiffrement Midori. Ce chiffrement doit donc avoir un mode de fonctionnement que nous choisissons tel que CBC ou CTR, ainsi qu'un algorithme de signature MAC et de dérivation de clés et d'IV (ou nonce dans le cas CTR par exemple) à partir d'un mot de passe. Ce chiffrement doit pouvoir s'utiliser en ligne de commandes en indiquant le mode chiffrement ou déchiffrement, ainsi qu'un message ou un fichier à chiffrer/déchiffrer accompagné d'un mot de passe donné par l'utilisateur. C'est ce mot de passe qui sera traité par l'algorithme de dérivation de clés afin de produire les différentes clés nécessaire.
Durant ce rapport, le terme "nibble" sera souvent utilisé. Cela désigne un demi octet soit un caractère hexadécimal.

Utilisation

Pour pouvoir utiliser le programme de chiffrement appelé "AE.py" pour "Authenticated Encryption" il faut avoir le fichier Midori.py fourni avec.

Une aide à l'utilisation est fournie en utilisant la commande:

```
./AE.py -h
usage: AE.py [-h] (-e | -d) (-m MESSAGE | --message-file MESSAGE_FILE)
            (-p PASSWORD | --password-file PASSWORD_FILE) [-o OUTPUT]

optional arguments:
  -h, --help            show this help message and exit
  -e, --enc             Mode Chiffrement
  -d, --dec             Mode Dechiffrement
  -m MESSAGE, --message MESSAGE
                        Le message a chiffrer/dechiffrer
  --message-file MESSAGE_FILE
                        Le fichier a chiffrer/dechiffrer
  -p PASSWORD, --password PASSWORD
                        Le mot de passe de chiffrement/dechiffrement
  --password-file PASSWORD_FILE
                        Le fichier contenant le mot de passe de
                        chiffrement/dechiffrement
  -o OUTPUT, --output OUTPUT
                        Le fichier de sortie (sortie standard par défaut)
```

Il faut commencer par choisir le mode de fonctionnement, "--enc" pour "encryption" ou "--dec" pour "decryption". Ensuite le message a chiffrer, cela peut être un message par l'entrée standard avec l'option "-m" ou "--message" suivi du message en un mot ou d'une

phrase entre guillemets. Nous pouvons aussi remplacer l'entrée standard par un fichier à chiffrer avec l'option "--message-file" à la place.
De la même manière le mot de passe peut être entré directement par "-p" ou "--password" pour l'entrée standard, où par un fichier contenant le dit mot de passe avec l'option "--password-file" suivi du nom du fichier.
Enfin nous pouvons préciser un fichier output dans lequel se trouvera le message chiffré ou déchiffré avec l'option "-o" ou "--output". si aucun output n'est précisé, le résultat sera affiché sur la sortie standard.

Voici un exemple de chiffrement d'un message:

```
./AE.py --enc -m "test de chiffrement" -p test1
8b075831a092c016aaae48b0b2108b66814dc083a5948200a7a9c7fa4df7691c
```

Nous venons ici de chiffrer le message "test de chiffrement" avec le mot de passe "test1".

Autre exemple avec un fichier a chiffrer:
Le fichier que nous cherchons à chiffré est le programme AE.py lui même.

```
./AE.py --enc --message-file AE.py -p test1 -o AE.ae
--> AE.ae
```

Le fichier AE.ae est créé et contient donc le fichier AE.py chiffré.

Pour le déchiffrer nous allons maintenant utiliser la commande:

```
./AE.py --dec --message-file AE.ae -p test1 -o AE2.py
--> AE2.py
```

Nous avons déchiffré AE.ae et redirigé la sortie standard dans "AE2.py". Et en utilisant la commande "diff" de Linux nous comparons les deux fichiers pour trouver une différence:

```
diff AE.py AE2.py
```

La commande ne nous renvoie rien, donc aucune différence entre les deux fichiers. Le chiffrement/déchiffrement s'est fait sans encombres.

Tests de performances

Chiffrement

```
time ./AE.py --enc --message-file 1ko.txt -p password -o 1ko.ae
--> 1ko.ae

real    0m0,091s
user    0m0,071s
sys    0m0,012s
```

```
time ./AE.py -e --message-file 1Mo.txt -p password -o 1Mo.ae
--> 1Mo.ae

real    0m18,066s
user    1m10,905s
sys    0m0,152s
```

```
time ./AE.py -e --message-file 200Mo.txt -p password -o 200Mo.ae
--> 200Mo.ae

real    61m14,817s
user    242m7,995s
sys    0m15,216s
```

Déchiffrement

```
time ./AE.py --dec --message-file 1ko.ae -p password -o 1ko.res
--> 1ko.res

real    0m0,098s
user    0m0,090s
sys    0m0,008s
```

```
time ./AE.py -d --message-file 1Mo.ae -p password -o 1Mo.res
--> 1Mo.res

real    0m18,172s
user    1m11,511s
sys    0m0,119s
```

Nous pouvons chiffrer un fichier de 1 kilo octet en un dixième de seconde et un fichier de 1Mo en 18 secondes environ. Les temps de chiffrement et de déchiffrement sont les mêmes car le chiffrement et le déchiffrement sont traités de la même manière grâce au mode CTR.

Pour 1 Mo nous avons mis 18 secondes, et $18 * 200 = 3600$ soit une heure. C'est le temps qui a été mis pour chiffrer 200Mo. Le chiffrement est donc linéaire et nous pouvons en déduire le temps qu'il faudrait pour chiffrer 1Go.

Le chiffrement d'un fichier de 1 Go est beaucoup trop long pour le tester. Étant donné le temps de chiffrement pour 1 Mo, nous pouvons estimer le temps de chiffrement pour 1 Go.

$1000\ Mo = 1\ Go$ donc $1000 * 18\ secondes = 18\ 000$
$\frac{18\ 000}{3600} = 5\ heures$. Il nous faudrait donc cinq heures pour chiffrer ou déchiffrer un fichier de 1 Go.

Choix de conceptions

Traitement d'un message - Chiffrement/Déchiffrement

Input

Un message à chiffrer peut être saisi par l'entrée standard comme précisé dans la partie "Utilisation" ou par le biais d'un fichier. Il peut tout aussi bien n'être que du texte ou un fichier binaire tel qu'un fichier compressé en zip par exemple.
Ce fichier sera lu puis transformé en hexadécimal pour être ensuite découpé en blocs de taille 16 caractères hexadécimaux (Nibbles). Si le programme est en mode déchiffrement, nous allons commencer par vérifier si les trente-deux derniers caractères représentent la chaîne "bin" hachée avec SHA-3 256 réduit à 32 nibbles, cela nous permet de savoir si le chiffré est un fichier binaire ou non. Ensuite nous vérifions les seize caractères de fin (en supprimant les caractères de "bin" haché si il y'a) qui représentent le MAC. Tous les autres caractères représentent le message chiffré.
Avant de commencer la phase de déchiffrement, nous allons appliquer l'algorithme de calcul du MAC sur le message chiffré pour le comparer au MAC fourni. S'ils ne correspondent pas nous signalons que la signature ou le mot de passe est invalide et nous terminons le programme.
La description de l'algorithme de calcul du mac est décrite dans la partie MAC.

Dérivation de clés

Pour générer les clés utilisées pour l'algorithme de chiffrement Midori, le MAC et le nonce du mode CTR, nous commençons par hacher le mot de passe de l'utilisateur avec SHA-3 256 puis nous opérons un xor de la deuxième moitié du hash avec la première moitié pour le réduire à une taille de 32 caractères.

Une fois le mot de passe haché, nous concaténons aux 16 premiers caractères du hash la valeur hexadécimale de ‘1’, et aux 16 derniers caractères la valeur hexadécimale de ‘2’.
Nous hachons le résultat des 16 premiers nibbles concaténé à ‘1’ pour obtenir ce qui sera la clé utilisé dans Midori et de taille 64 nibbles.
Nous hachons le résultat des 16 derniers nibbles concaténés à ‘2’ et nous ne gardons que les 24 premiers nibbles pour obtenir ce qui servira de nonce. Nous ne gardons que 8 nibbles car nous concaténons au nonce le compteur de CTR qui est un entier décris en hexadécimal sur 8 nibbles pour obtenir un bloc de 16 nibbles.
Pour finir nous concaténons au hash du mot de passe la valeur ‘3’ puis nous hachons le résultat pour obtenir ce qui servira de clé pour HMAC.

Découpage en blocs

Le découpage d’un message en blocs de 16 nibbles peut être long selon la taille de la chaîne de caractères, pour cette raison, de la parallélisation a été utilisée. Dans le cas où le message à découper est plus grand que quatre blocs de 16 caractères, quatre threads sont créés et se partagent le message en quatre part égales. Si le message est plus grand que quatre fois la taille d’un bloc soit $4 * 16 = 64$, un cinquième thread est créé et chargé de traiter cette portion restante du message.

Chaque thread va subdiviser sa portion du message en blocs de 16 caractères hexadécimaux sous forme de liste et chacune des listes de blocs calculées seront ensuite réunie en une seule liste finale comportant le résultat du découpage du message.
Maintenant que le message est découpé en blocs, nous appliquons un padding sur le dernier bloc, commençant par un ‘1’, suivi d’entre zéro et quatorze ‘0’ et terminé par un ‘1’. Pour plus d’informations, voir la partie Padding.

Chiffrement/Déchiffrement des blocs

La liste de blocs sera ensuite transmise à la fonction de parallélisation du chiffrement par bloc CTR. La fonction CTR est écrit de manière à prendre en paramètre un “nonce”, le bloc à chiffrer sous la forme d’un tableau de caractères hexadécimaux et une clé de chiffrement.

La fonction de parallélisation est similaire à celle de découpage des blocs, si la liste de blocs contient plus de quatre blocs, quatre threads seront créés et vont chacun traiter une part égale de la liste de blocs. Chaque thread s’occupera de “nombre de bloc modulo quatre” et un cinquième thread peut être créé pour traiter les éventuels blocs restants.

Formatage du message

Si le programme est en mode chiffrement nous créons une chaîne de caractères qui est la concaténation de tous les blocs chiffrés. Si le programme est en mode déchiffrement, nous allons concaténer tous les blocs en une chaîne de caractères à laquelle nous supprimons le padding de fin.
En mode chiffrement, nous allons calculer la signature MAC du chiffré pour la concaténer au message chiffré. Nous allons donc fournir à notre algorithme de HMAC, une clé générée dans l’algorithme de dérivation de clés ainsi que le message chiffré lui même.

Une fois que nous avons obtenu notre signature MAC nous avons notre chaîne composée de notre message chiffré concaténé avec notre signature MAC et potentiellement du hash du mot “bin” si le fichier chiffré était un binaire.

Output

Pour finir, si l’utilisateur n’a précisé aucune sortie avec l’option ‘-o’ du programme, le résultat sera affiché sur la sortie standard. Dans le cas où une sortie est précisée, nous allons vérifier son type. Si nous avions vu à l’input le hash de la chaîne “bin” en fin de chiffré, nous savons alors que le fichier doit être considéré comme un fichier binaire. Nous allons donc transformer notre message déchiffré de la forme hexadécimal à la forme binaire, si ce n’est pas un fichier binaire alors nous convertissons notre chaîne hexadécimal en chaîne de caractères. Dans le cas d’un chiffrement, nous écrivons dans le fichier la chaîne hexadécimal du message chiffré et du MAC. Si nous chiffrons un fichier binaire nous allons en plus ajouter la chaîne “bin” hachée et réduite à 32 nibbles à la fin du fichier chiffré pour préparer le déchiffrement.

Mode de chiffrement/déchiffrement

Pour l’implémentation de ce programme, j’ai choisi d’utiliser le mode de chiffrement par blocs “CTR” pour “CouTeR mode” pour sa simplicité. En effet, le mode CTR permet de chiffrer et de déchiffrer avec la même fonction, il n’y a pas besoin d’écrire de fonction inverse. De plus, chaque chiffrement de bloc étant indépendant, nous pouvons paralléliser le calcul pour pouvoir accélérer et gagner du temps.

Parallélisation

J’ai utilisé la bibliothèque “multiprocessing” de python 3 pour paralléliser la fonction de chiffrement CTR ainsi que le découpage du message en blocs de taille 8 octets, soit 16 nibbles. Avant d’utiliser la parallélisation, le chiffrement de 1Mo se faisait en 37 secondes, maintenant cela se fait en 18 secondes.

MAC

Pour signer notre chiffrement, HMAC semblait tout indiqué.
Pour générer la signature MAC, nous fournissons une clé “mackey” ainsi que le message chiffré. La clé “mackey” a été générée lors de la phase de dérivation de clés à partir du mot de passe utilisateur.
L’utilisation d’un simple hachage du message chiffré comme signature est inefficace car si un attaquant intercepte le message, il peut modifier des bits et hacher de nouveau le message. Lors du déchiffrement pendant la vérification du MAC le programme verra que le hash du message chiffré correspond au MAC fourni et ne verra pas que le fichier a été modifié. C’est pour cela que l’on doit utiliser une clé.
L’algorithme de signature HMAC créé va d’abord concaténer le message chiffré avec la clé “mackey”, nous hachons ce message et cette clé avec SHA-3 256 et nous faisons un xor

entre les deux moitiés du hash de manière a obtenir un hash de taille 32 et nous le refaisons une nouvelle fois de manière à obtenir un hash de taille 16.
Nous avons obtenue notre signature MAC de taille 16 nibbles soit 8 octets.
Grâce à cette méthode, si un attaquant intercepte le message et le modifie, il ne pourra pas générer la signature MAC car elle est générée à partir du mot de passe de l'utilisateur donné lors du chiffrement initial.
Nous venons donc de garantir l'authentification et l'intégrité du message car au moment du déchiffrement, le programme va envoyer la partie du fichier correspondant au message chiffré et l'envoyer de nouveau dans la fonction de calcul de HMAC puis le comparer à la signature MAC fourni. S'ils ne correspondent pas, la programme va signaler que la signature est invalide, cela peut être causé par un intégrité compromise ou un mauvais mot de passe de déchiffrement.

Padding

Pour signaler la fin d'un message, nous utilisons du padding. Il existe différentes formes de padding, et dans notre cas il s'agit du padding commençant et terminant par un "1" séparé par aucun ou plusieurs "0". Le message à chiffrer est divisé en blocs de taille 16 nibbles, si le dernier bloc n'est pas complet car le message n'est pas de la taille d'un multiple de 16, nous allons ajouter un '1' dans ce bloc, le compléter par des '0' et le finir par un dernier '1'.
Si le bloc n'avait plus que deux caractères manquant nous allons mettre les deux '1' sans '0' et si le bloc n'avais plus qu'une place libre nous allons mettre un '1', créer un nouveau bloc rempli de "0" et finir de nouveau par un '1'.
Si la taille du message était un multiple de 16 alors tous les blocs sont remplis. Nous allons alors en créer un nouveau composé de un '1', suivi de quatorze "0" et un dernier '1' pour finir le bloc. De cette manière nous sommes sur que notre message se termine par un padding. Tous ces blocs sont ensuite chiffrés avec le padding. Durant la phase de déchiffrement, nous savons alors que le dernier caractère doit être un "1", une suite ou non de '0' et un nouveau '1'. Si ce n'est pas le cas alors le déchiffrement a eu un problème et le programme ne retourne rien car le message déchiffré est faux.

Conclusion

Le programme "AE.py" permet de chiffrer des messages ou des fichiers tout en garantissant leur intégrité et l'authentification par l'utilisation d'un mot de passe et d'une signature HMAC. Comme décrit précédemment, si un adversaire modifie un caractère hexadécimal du message chiffré, le programme de déchiffrement signalera que la signature est invalide. Nous saurons alors que l'intégrité ou l'authentification du message est compromise.
Le programme permet de chiffrer des messages de 1 Mo en 18 secondes environ soit 1 Go en cinq heures environ. Le temps de calcul semble être linéaire car le chiffrement de 200 Mo met une heure et c'est aussi le résultat obtenu si nous calculons $200 * 18 = 3600$ où 18 est le temps de calcule pour 1Mo.