ICR - Labo #1 : *MAC-and-Encrypt and Padding Oracles*

G.Burri

7 novembre 2014

1 Introduction

Le but de ce laboratoire est d'expérimenter le chiffrement symétrique *AES* ainsi que la vérification de l'intégrité des données par *MAC*, de mettre en évidence un problème de sécurité lié à un protocole choisi propre et de proposer une solution.

Nous utiliseront *AES-256* en mode *CBC* pour chiffrer les données ainsi que *HMAC-SHA256* pour l'authentification.

2 Simulation du protocole

2.1 Utilisation du programme

Le code est écrit en langage *Rust*[1] et utilise le système de build *Cargo* qui est livré en standard. Il est conseillé d'utiliser la version *nightly* disponible ici : `http://www.rust-lang.org/install.html`.

Pour construire et lancer l'application il faut se trouver dans le dossier contenant le fichier *Cargo.toml* et lancer la commande suivante :

```
$> cargo run -- <args>
```

Où *<args>* peut valoir :

- *genkey* : génère une clef de 256 bits. Utilisé initialement pour définir la clef d'authentification K_a et la clef de chiffrement K_c ;
- *tests* : effectue un certain nombre de tests pour vérifier le comportement du serveur vis-à-vis du protocole ;
- *oracle-weak* : effectue une attaque sur la première version du serveur ;
- *oracle-fixed* : effectue une attaque sur la version corrigée du serveur.

2.2 Structure du code

Le code est découpé en quatre modules :

- *crypto* : fournit les primitives de chiffrement, déchiffrement, calcul du MAC. Utilise un binding *Rust* vers *OpenSSL* ;
- *packet* : définit le format des paquets et permet leur sérialisation et désérialisation ;

1. `http://www.rust-lang.org`

— *end_point* : permet la création de serveurs et de clients et gère la communication sur *TCP/IP* ;
— *oracle_machine* : implémente l'attaque par padding-oracle.

2.3 Tests du protocole

Un certain nombre de tests sont implémentés dans la fonction `end_point::Client::start_tests(..)`. Il est possible de les exécuter à l'aide de la commande suivante.

```
$> cargo run -- tests
```

La sortie de cette commande est la suivante :

```
   Compiling lab1_rust v0.0.1 (file:///home/gburri/Documents/
       Master/ICR/lab1/lab1_rust)
     Running `target/lab1_rust tests`
Starting server on [::1]:4221...
Server started
===== Test case #1:
Sending a valid packet...
[Client] time: 0. Sending: Command { id: 154, payload(29): "
    ba57cb4a9cc83c9b9027bca2cf9c46f25d0c1608a4044dc878bd474bbd
    " }
[Server] time: 2. Valid command received: Packet { t: Command
    { id: 154, payload(29): "
    ba57cb4a9cc83c9b9027bca2cf9c46f25d0c1608a4044dc878bd474bbd
    " }, timestamp: 1 }
[Server] time: 3. Answer sent: Answer { id: 125, payload(31):
    "
    a88ffbd4758e17d0130cd11c1749149bc33cc818c42edec5fb6edb29352f83
    " }
[Client] time: 4. Command transmitted correctly, answer:
    Packet { t: Answer { id: 125, payload(31): "
    a88ffbd4758e17d0130cd11c1749149bc33cc818c42edec5fb6edb29352f83
    " }, timestamp: 3 }
===== Test passed
===== Test case #2:
[Server] time: 3. Connection closed: EOF
Sending a packet with an unknown type...
[Server] time: 0. Error or invalid packet: Err(
    UnknownPacketTypeError)
===== Test passed
[Server] time: 0. Connection closed: EOF
===== Test case #3:
Sending a packet with an old timestamp...
Error, timestamp mismatch, current timestamp: 0, packet
    received: Packet { t: Command { id: 154, payload(29): "
    ba57cb4a9cc83c9b9027bca2cf9c46f25d0c1608a4044dc878bd474bbd
    " }, timestamp: 0 }
[Server] time: 0. Error or invalid packet: Err(
    InvalidTimestampError)
===== Test passed
[Server] time: 0. Connection closed: EOF
===== Test case #4:
```

```
Sending a packet with altered crypted data (do not alter the
    padding)...
[Server] time: 2. Error or invalid packet: Err(
    MACMismatchError)
===== Test passed
[Server] time: 2. Connection closed: EOF
===== Test case #5:
Sending a packet with too small data...
[Server] time: 0. Error or invalid packet: Err(
    UnconsistentDataSizeError)
===== Test passed
[Server] time: 0. Connection closed: EOF
===== Test case #6:
Sending a packet with too large data...
[Server] time: 0. Error or invalid packet: Err(
    UnconsistentDataSizeError)
===== Test passed
[Server] time: 0. Connection closed: EOF
===== Test case #7:
Sending a packet with wrong padding (all 0)...
[Server] time: 2. Error or invalid packet: Err(PaddingError)
===== Test passed
All tests passed
[Server] time: 2. Connection closed: EOF
```

2.4 Quelle est la stratégie recommandée en pratique parmi les trois listées ci-après ?

— *MAC-and-Encrypt* : $Enc(M)|MAC(M)$;
— *MAC-then-Encrypt* : $Enc(M|MAC(M))$;
— *Encrypt-then-MAC* : $Enc(M)|MAC(Enc(M))$.

D'après [3], la stratégie *Encrypt-then-MAC* est la plus sûre dans le cadre du chiffrage authentifié. L'article de *M. Bellare and C. Namprempre* [1] évalue ces trois stratégies.

2.4.1 Quelle stratégie est utilisée par *TLS* ?

TSL utilise la deuxième version (*MAC-then-Encrypt*). À noter que le *MAC* est optionnel.

Une proposition [2] existe afin d'utiliser du *Encrypt-then-MAC* pour *TSL*.

2.4.2 Quelle stratégie est utilisée par *SSH* ?

SSH utilise la même méthode que celle utilisée dans ce laboratoire, c'est-à-dire *MAC-and-Encrypt*.

2.5 Quel est le rôle du timestamp en terme de sécurité ?

Il permet de minimiser certaines attaques comme l'attaque par rejeu (*replay attack*)[6] où un attaquant réutilise tel-quel tout ou une partie d'un message intercepté au préalable.

2. https ://tools.ietf.org/html/draft-ietf-tls-encrypt-then-mac-02

Dans notre cas un attaquant ne pourra pas rejouer une commande telle quelle, car elle serait rejetée par le serveur ayant un *timestamp* supérieur. Si l'attaquant essaie de renvoyer un paquet avec un timestamp modifié, alors les données décodées ne seront plus validées par le *MAC* car le vecteur d'initialisation utilisé (IV) lors du déchiffrement est composé en partie par le *timestamp*.

2.6 Y a-t-il un moyen d'effectuer une attaque de type *denial-of-service* sur notre dispositif ?

Via une *replay attack* en modifiant le *timestamp* pour qu'il soit valide, le dispositif va devoir déchiffrer les données puis calculer le *MAC* avant de se rendre compte que le paquet est invalide et envoyer une réponse qui sera chiffrée et authentifiée. Dans ce cas on peut faire travailler énormément le dispositif en lui envoyant autant de paquets à déchiffrer que le permet le débit du moyen de communication utilisé. Cela peut amener le dispositif à être surchargé.

2.7 À la place d'utiliser un *IV* aléatoire, le mode *CBC* implémente une approche basée sur un *nonce*. Que peut-on dire de sa sécurité ?

Cet article de *P. Rogaway*[2] suggère d'éviter de suivre le standard qui consiste à chiffrer le nonce avec la même clef utilisée pour les données.

2.8 Remarques concernant la sécurité de notre protocole

A priori nous n'avons pas choisi la stratégie la plus recommandée en terme de sécurité. Comme nous le verrons par la suite, ce protocole est vulnérable à une attaque de type *padding-oracle*.

3 Utilisation du serveur comme d'un oracle de déchiffrement

3.1 Historique de l'attaque par oracle à l'aide du remplissage

L'attaque originelle a été publiée en 2002 par *Serge Vaudenay*. En 2010, cette attaque a été mise en pratique contre plusieurs frameworks web tels que *JavaServer Faces*, *Ruby on Rails* et *ASP.NET*. En 2012, il a été montré qu'elle était efficace contre certain appareils hardware.

Il existe une nouvelle variante, publiée en 2013, nommée *the Lucky Thirteen attack*, permettant d'attaquer des implémentations ayant été corrigées. En février 2013, les personnes en charge des implémentations de *TLS* travaillaient à la réalisation d'un correctif à cette attaque.

L'attaque la plus récente utilisant un *padding-oracle* est *POODLE*[3] qui a été dévoilée en septembre 2014.

Cette section est largement inspirée de l'article de *Wikipedia* sur la *padding-oracle attack* [5].

3. `http://en.wikipedia.org/wiki/POODLE`

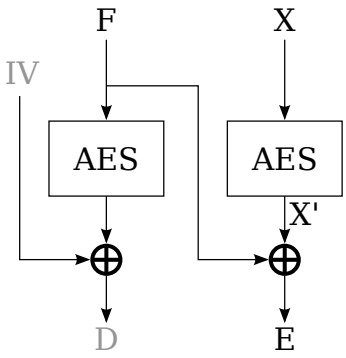

FIGURE 1 – *Décryptage par un oracle, AES-CBC.*

3.2 Explication de l'attaque pour notre cas

Le but est de faire décoder par un oracle tout ou une partie d'un message chiffré intercepté. Le décryptage se fait par bloc de 16 octets et nécessite le bloc chiffré le précédant ou l'IV dans le cas du premier bloc. Pour notre test nous partons du principe que l'attaquant a intercepté un paquet chiffré, qu'il en a compris la structure et qu'il a deviné que l'IV correspondait au *timestamp*.

Nous utilisons une attaque basée sur l'information renvoyée par l'oracle concernant la présence d'un bourrage valide. D'après le protocole un *MAC* est calculé à partir des données non-bourrées, puis le bourrage est ajouté pour obtenir une taille multiple de 16, et finalement les données et le bourrage sont chiffrés. Lors du traitement par l'oracle, les données sont d'abord déchiffrées puis le bourrage est contrôlé. S'il n'est pas valide un paquet d'erreur est renvoyé au client (*CryptError*). Si le bourrage est correct, alors celui-ci est retiré et les données restantes sont authentifiées à l'aide du *MAC*. Si l'authentification échoue alors un paquet d'erreur est renvoyé au client (*AuthError*).

La valeur des octets du bourrage correspond à sa taille, par exemple un bourrage de longueur trois est représenté par *[0x03, 0x03, 0x03]*. Si les données avant bourrage sont déjà multiple de 16, alors un bourrage de longueur 16 est ajouté de sorte qu'un bourrage soit toujours présent.

La figure 1 illustre la structure de l'attaque. IV et D n'ont pas d'importance dans notre cas. X est le bloc à décrypter (*ciphertext*), *X'* est le bloc à décrypter après avoir été décodé par *AES* mais avant d'avoir été « xoré » par F. F correspond à un bloc qui sera forgé par nos soins durant le décryptage de X. De plus C, qui n'est pas illustré sur le schéma, correspond au bloc précédent X ou à l'IV si X est le premier bloc et R correspond au message décrypté (*plaintext*).

Dans un premier temps nous allons chercher le premier octet b de F noté F_1 en itérant celui ci de 0 à 255. Pour chaque itération un paquet de commande est envoyé à l'oracle comprenant en guise de données chiffrées : $F + X$. Le paquet d'erreur renvoyé va nous indiquer si le bourrage est correct (*AuthError*) ou s'il ne l'est pas (*CryptError*). Pour le premier octet nous allons chercher le bourrage *[0x01]*, pour le deuxième le bourrage *[0x02, 0x02]* et ainsi de suite jusqu'à l'octet 16.

Dès qu'un paquet d'erreur *AuthError* est reçu alors nous pouvons calculer $X'_1 = F_1 \oplus b$ puis le premier octet de notre message décrypté $R_1 = X'_1 \oplus C_1$.

Avant de passer à l'octet suivant $b' = b + 1$ il faut s'assurer que les b premiers octets de E vaudront bien b' lors du décryptage par l'oracle. Pour ce faire on met à jour F comme ceci : $\forall i \in [1, \ldots, b], F_i = b' \oplus X_i$.

Une subtilité existe pour la recherche du premier octet : il est possible que le paquet d'erreur *AuthError* corresponde, avec une faible probabilité, à un autre bourrage que *[0x01]*. Pour prévenir ce cas il faut, pour ce premier octet, envoyer un paquet de commande pour toutes les valeurs de F_1 et compter le nombre de paquets d'erreur *AuthError* reçus. Si ce nombre est égal à 1 alors on peut passer à b', sinon il faut recommencer en modifiant $F_2 = (F_2 + 1) \, mod \, 256$.

Le code correspondant à cette attaque peut être exécuté par la commande suivante :

```
$> cargo run --release -- oracle-weak
```

La sortie est la suivante :

```
The oracle machine has found the plain block!:
   Expected block: [242, 93, 12, 22, 8, 164, 4, 77, 200, 120,
       189, 71, 75, 189, 2, 2]
   Decrypted block: [242, 93, 12, 22, 8, 164, 4, 77, 200, 120,
        189, 71, 75, 189, 2, 2]
```

3.3 Calcul de la complexité moyenne de l'attaque en terme de nombre de requêtes effectuées auprès de l'oracle

Sans prendre en compte la particularité du premier octet illustré à la section précédente, la complexité moyenne pour le décryptage d'un bloc de 16 octets est de $16 * 256/2 = 2048$ requêtes.

Dans l'exemple présenté dans le code, le nombre de requêtes est de 2099. La durée d'exécution est de 180 ms, cette relative longue durée est certainement due à un overhead engendré par les couches réseau *TCP/IP*.

4 Correction du protocole

4.1 Description

Le correctif proposé consiste à authentifier également le bourrage et non plus que les données. Cela a pour conséquence de vérifier en premier l'authenticité du contenu avant de procéder à la validité du padding. Les deux messages d'erreur, *CryptError* et *AuthError*, font toujours parties du protocole.

Le code correspondant à ce correctif peut être exécuté par la commande suivante :

```
$> cargo run --release -- oracle-fixed
```

5 Conclusion

Ce laboratoire a permis de mettre en évidence une attaque par *padding-oracle* sur un protocole utilisant des normes de sécurité standards et éprouvées telles que *AES* en mode *CBC*, une authentification par *HMAC-SHA256* et la

stratégie *MAC-and-Encrypt*. Une solution a ensuite été proposée et testée face à l'attaque précédemment citée.

Références

[1] M. Bellare and C. Namprempre. Authenticated encryption : Relations among notions and analysis of the generic composition paradigm. *UC San-Diego*, 2007.

[2] Phillip Rogaway. Evaluation of some blockcipher modes of operation. *University of California*, 2011.

[3] Wikipedia. Authenticated encryption — Wikipedia, the free encyclopedia, 2014.

[4] Wikipedia. Block cipher mode of operation — Wikipedia, the free encyclopedia, 2014.

[5] Wikipedia. Padding oracle attack — Wikipedia, the free encyclopedia, 2014.

[6] Wikipedia. Replay attack — Wikipedia, the free encyclopedia, 2014.

[7] Wikipedia. Transport layer security — Wikipedia, the free encyclopedia, 2014.