TP3 - Modes de chiffrement

30 Septembre 2021

L'objectif de ces exercices est de vous familiariser avec les vulnérabilités de différents modes de chiffrement (ou leur utilisation), et d'explorer les solutions possibles.

> Pour ce TP, pas besoin de `sage`. Tout peut être fait en python, ou dans le langage de votre choix.

Préliminaires

Pour les deux exercices suivants, des interactions avec un serveur sont nécessaires. Pour éviter tout problème de gestion et d'éventuel problème de réseau, un script python faisant tourner un serveur local (sur `localhost:8000` par défaut) est mis à disposition avec le sujet.

Les deux exercices étant indépendants, le serveur n'assure pas les mêmes fonctionnalité, et il faudra lancer le bon pour chaque exercice.

Tous les scripts mis à disposition tournent en **python 3**, donc assurez vous d'utiliser la bonne version en cas de problème. Pour les opérations cryptographiques, le package `pycryptodome` est utilisé. Vous pouvez l'installer avec un simple `pip3 install pycryptodome`.

Pour lancer le serveur, exécutez le script `server.py` dans un terminal, et connectez vous via le port affiché pour interagir avec (en cas de soucis pour se connecter, ne perdez pas de temps et demandez de l'aide). Le script `client.py` peut vous servir de base pour l'exercice.

Le contenu des scripts autres que client.py est évidemment considéré secret (bien que vous y aillez accès pours des raisons de simplicité) et les informations contenues ne doivent pas être utilisées pour la réalisation de l'exercice. En revanche, je vous encourage à y jeter un oeil et ajouter des instructions de debug pour vous aider.

Exercice 1 : CVE-2020-1472 Zerologon

La CVE-2020-1472, appelée Zerologon en référence à la vulnérabilité exploitée et au nom du protocol, permet à un attaquant de s'authentifier sur un *Domain Controller* sans connaître les identifiants. L'objectif de cet exercice est d'exploiter la vulnérabilité cryptographique rendant le bypass de l'authentification possible. Nous considérerons seulement une version simplifié du protocole, et l'exploitation à partir du *Domain Controller* est hors sujet ici.

Commencez par lire le *white paper* décrivant la vulnérabilité (notamment les pages 3 à 5).

Ici, le serveur local n'effectue pas la dernière étape du protocole (*Signing + sealing* avec la clé de session), qui empêche l'attaque. En pratique, cette étape peut être désactivée via un flag sur certaines implémentations.

Implémentez l'attaque permettant de s'authentifier. Ici, le secret (correspondant typiquement à un hash salé du mot de passe utilisateur) est généré aléatoirement au lancement du serveur.

> Servez vous du code disponible dans le dossier `ex1-zerologon_material`. Le fichier `client.py` est une bonne base pour commencer le script d'attaque.

Le serveur local est assez basique : une fois lancé, il attend que le client s'y connecte en envoyant 2 séquences d'octets (type `bytes` en python). Le première correspond on challenge du client, auquel le serveur répond en envoyant son challenge. La seconde requête attendue est le validation du challenge, auquel le serveur répond par `b'You are successful authenticated!'` ou `b'Authentication failed'`.
Ne cherchez pas à compliquer les choses, l'exploit tient en 10 lignes de Python.

Exercice 2 : Oracle de padding avec le mode CBC

Le mode de chiffrement CBC a longtemps été utilisé de pair avec l'AES pour assurer la confidentialité des données. En revanche, de par son design, il est *malléable*. C'est à dire qu'un attaquant peut opérer des modifications contrôlées sur le contenu du message clair, uniquement à partir du chiffré.

De plus, s'agissant d'un mode de chiffrement par bloc, un padding est systématiquement ajouté à la fin du message pour que la taille soit multiple de la taille d'un bloc (128 bits pour AES). Si la taille du message est déjà multiple de la taille du bloc, un bloc entier de padding est ajouté.

Notamment, si l'attaquant dispose d'un oracle de format sur le message clair (un oracle de padding typiquement), il est en mesure d'abuser de cette malléabilité pour retrouver le contenu du message.
Si vous n'êtes pas familier avec ce type d'attaque, je vous invite à regarder ce lien, qui illustre le concept, et à poser des questions en cas d'incompréhension. Cette vulnérabilité est très connue et bien documentée, vous pouvez trouver de nombreuses explications différentes sur internet, n'hésitez pas à en regarder plusieurs si vous avez du mal à comprendre le concept.

> Le padding utilisé n'est pas standard (afin d'éviter que vous ne repreniez des solutions toutes faites).

Le padding utilisé par l'oracle de cet exercice est le suivant : `pad = [i for i in range(padding_len)]`
Par exemple, pour un message nécessitant 1 octet de padding, `pad = 0x00`. Pour deux octets, `pad = 0x00 01`, ..., pour 16 octets, `pad = 0x00 01 .. 0f`.

L'idée de l'attaque est toutefois identique : retrouver le message clair un octet à la fois, en partant de l'octet le plus à droite.

1. En envoyant des requêtes à l'oracle niveau 1, retrouvez le contenu du message (le message clair est évidemment disponible dans le fichier `oracle.py`, mais supposez que vous n'y avez pas accès).
   Pour appréhender le fonctionnement du script, vous pouvez ajouter des informations de debug si vous le souhaitez.
   (a) Notez que cet oracle attend au moins 2 blocs de 16 octets (IV + chiffré), et retourne des codes d'erreur clairs en fonction de la validité du padding.
   Implémentez une fonction `query_oracle` qui effectue une requête à l'oracle et retourne un booléen représentant la validité du padding. Vous pouvez tester avec un message quelconque (qui devrait être invalide) et le message chiffré original (qui devrait être valide).
   (b) Implémentez une fonction `decrypt_char` qui prend en entrée deux blocs `c0` et `c1` ainsi que l'indice `i` du caractère à retrouver.
   Cette fonction va faire varier le $\texttt{i}^{ime}$ octet de `c0` jusqu'à ce que le déchiffrement de `c0 || c1` donne un padding valide.
   La fonction retourne le $\texttt{i}^{ime}$ caractère du bloc `c1` déchiffré.
   (c) Implémentez une fonction `decrypt_block` qui prend en entrée le block à déchiffrer `c1` et le block précédent `c0`, et utilise la fonction `decrypt_char` sur les indices i = 15, ..., 0 pour retrouver les caractères de `c1` un par un.
   La fonction retourne le contenu de en clair de `c1`.
   Indice : Lorsque vous changez d'indice, n'oubliez pas de mettre à jour le bloc intermédiaire `c0'` pour que le padding corresponde.
   (d) En prenant l'IV et le chiffré donnés, parcourez les blocs de droite à gauche, en déchiffrant chacun d'eux à l'aide de la fonction `decrypt_block`.

2. Modifiez la fonction `query_oracle` pour exploiter l'oracle de niveau 2. Si vous ne comprenez pas ce qui vous permet de distinguer la validité du padding, vous pouvez aller voir le code dans `oracle.py`. Le reste du code ne devrait pas changer puisque la vérification est similaire.
3. Quel mécanisme(s) supplémentaire(s) pourrait-on coupler au mode CBC pour éviter cette vulnérabilité ? En vous inspirant de l'oracle niveau 2, décrivez les précautions particulières à prendre pour éviter que la vulnérabilité puisse encore être exploitée.
4. Afin d'éviter de reposer sur une solution "bancale" (qui demande une implémentation extrêmement minutieuse pour éviter des conséquences désastreuses), que suggéreriez-vous pour remplacer ce mode de chiffrement. Pourquoi ?

À RETENIR

Les modes de chiffrement les plus répandus ne sont pas forcement les plus sécurisés et les plus fiables. Certains, tel que le mode CBC, sont conservés principalement pour des raisons de rétrocompatibilité, et leur sécurité repose sur un assemblage parfois complexe de mécanisme ad-hoc permettant de palier leur(s) problème(s).

De manière général, il est préférable de se tourner vers une solution qui assure les besoins de sécurité requis par défaut (chiffrement authentifié avec le mode GCM par exemple) .

S'il n'y a pas le choix, il vaut mieux faire appel à un expert pour éviter d'introduire une vulnérabilité dans le mécanisme mis en place.