Hash Length Extension Attack Lab

Nom des étudiants : Chater Ahmed et Daouda David Coulibaly .

Introduction :

Dans ce laboratoire, nous explorerons l'attaque par extension de longueur de hachage, une vulnérabilité qui affecte certains algorithmes de hachage tels que MD5 et SHA1. Cette attaque exploite la manière dont ces algorithmes traitent les données pour permettre à un attaquant d'ajouter des données à un message déjà haché, sans connaître le message original ni la clé secrète utilisée pour le hachage.

2 Lab Environment :

Nous avons configuré un serveur web pour ce laboratoire. Un client peut envoyer une liste de commandes à ce serveur, chaque requête devant inclure un MAC calculé à partir d'une clé secrète et de la liste des commandes. Le serveur exécutera uniquement les commandes de la requête si le MAC est correctement vérifié. Pour ce faire, nous avons utilisé la machine virtuelle hôte comme client et un conteneur pour le serveur web. Afin de configurer l'environnement du laboratoire, nous avons téléchargé le fichier **Labsetup.zip** sur notre machine virtuelle à partir du site web du laboratoire, puis nous l'avons décompressé. Ensuite, nous avons navigué dans le dossier **Labsetup** et utilisé le fichier **docker-compose.yml** pour configurer l'environnement du laboratoire. Une explication détaillée du contenu de ce fichier ainsi que des Dockerfiles associés est disponible dans le manuel utilisateur, accessible via le site web du laboratoire. Si c'est la première fois que vous configurez un environnement SEED en utilisant des conteneurs, il est essentiel de lire attentivement le manuel utilisateur. Pour faciliter l'utilisation, nous avons créé des alias

pour les commandes Docker et Compose dans le fichier **.bashrc** de la machine virtuelle SEEDUbuntu 20.04.

```
version: "3"

services:
    web-server:
        build: ./image_flask
        image: seed-image-flask-len-ext
        container_name: www-10.0.2.80
        tty: true
        cap_add:
                - ALL
        networks:
            net-10.9.0.0:
                ipv4_address: 10.0.2.80

networks:
    net-10.0.2.0:
        name: net-10.0.2.0
        ipam:
            config:
                - subnet: 10.0.2.0/24
```

Tous les conteneurs fonctionnent désormais en arrière-plan. Pour exécuter des commandes sur un conteneur, nous devons souvent ouvrir un shell sur celui-ci. Nous avons d'abord utilisé la commande docker ps pour identifier l'ID du conteneur, puis la commande docker exec pour lancer un shell sur ce conteneur. Des alias ont été créés dans le fichier .bashrc afin de simplifier ces opérations.

Creation des conteneures :

```
[12/22/24]seed@VM:~/.../Labsetup$ docker-compose build
Building web-server
Step 1/4 : FROM handsonsecurity/seed-server:flask
flask: Pulling from handsonsecurity/seed-server

da7391352a9b: Pull complete

14428a6d4bcd: Pull complete

2c2d948710f2: Pull complete

01ee2d1608cf: Pull complete

9f40388d7765: Pull complete

2828455a2ef1: Pull complete

Digest: sha256:75d0de8a7f6b7230f235cfec1105d050f22f053b592f97fdd80
9dbf4c8c69c6c
Status: Downloaded newer image for handsonsecurity/seed-server:fla
sk
 ---> 384199adf332
Step 2/4 : COPY app  /app
 ---> f0de099f9a20
Step 3/4 : COPY bashrc /root/.bashrc
 ---> ab841b6bfa63
Step 4/4 : CMD cd /app && FLASK_APP=/app/www flask run --host 0.
0.0 --port 80 && tail -f /dev/null
 ---> Running in ab5d4f93eec4
Removing intermediate container ab5d4f93eec4
 ---> 96d2b179f935
```

Demarer les conteneures :

```
[12/22/24]seed@VM:~/.../Labsetup$ docker-compose up
Creating network "net-10.0.2.0" with the default driver
Creating www-10.0.2.80 ... done
Attaching to www-10.0.2.80
www-10.0.2.80 |  * Serving Flask app "/app/www"
www-10.0.2.80 |  * Environment: production
www-10.0.2.80 |    WARNING: This is a development server. Do not use it in a
production deployment.
www-10.0.2.80 |    Use a production WSGI server instead.
www-10.0.2.80 |  * Debug mode: off
www-10.0.2.80 |  * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
```

Tous les conteneurs fonctionnent désormais en arrière-plan. Pour exécuter des commandes sur un conteneur, nous devons souvent ouvrir un shell sur celui-ci. Nous avons d'abord utilisé la commande docker ps pour identifier l'ID du conteneur, puis la commande docker exec pour lancer un shell sur ce conteneur. Des alias ont été créés dans le fichier .bashrc afin de simplifier ces opérations.

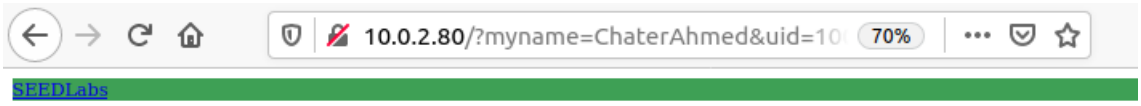
```
[12/22/24]seed@VM:~/.../Labsetup$ dockps -a
256323bf2a44  www-10.0.2.80
[12/22/24]seed@VM:~/.../Labsetup$ docksh 256323bf2a44
root@256323bf2a44:/#
```

Nous avons configuré le serveur web en utilisant le domaine www.seedlab-hashlen.com pour héberger le programme serveur. Dans notre machine virtuelle (VM), nous avons associé ce nom d'hôte au conteneur du serveur web (10.0.2.80). Cette association a été réalisée en ajoutant l'entrée correspondante dans le fichier /etc/hosts

```
  GNU nano 4.8
127.0.0.1       localhost
10.0.2.80       www.seedlab-hashlen.com
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Nous avons effectué l'étape consistant à envoyer une requête typique du client au serveur. La requête inclut l'argument uid, que le serveur utilise pour récupérer la clé MAC depuis le fichier LabHome/key.txt. Dans notre exemple, la commande est lstcmd, avec une valeur de 1, demandant au serveur de lister tous les fichiers. L'argument final est le MAC, calculé à l'aide de la clé secrète partagée entre le client et le serveur, ainsi que des arguments de la commande. Avant d'exécuter la commande, le serveur vérifie le MAC pour garantir l'intégrité de la requête.

http://www.seedlab-hashlen.com/?myname=ChaterAhmed&uid=1001&lstcmd=1&download=secret.txt&mac=dc8788905dbcbceffcdd5578887717c12691b3cf1dac6b2f2bcfabc14a6a7f11



Hash Length Extension Attack Lab

Sorry, your MAC is not valid

3 Tasks :

3.1 Task 1: Send Request to List Files :

j'ai envoyé une requête bénigne au serveur pour analyser sa réponse. J'ai utilisé le modèle de requête suivant :

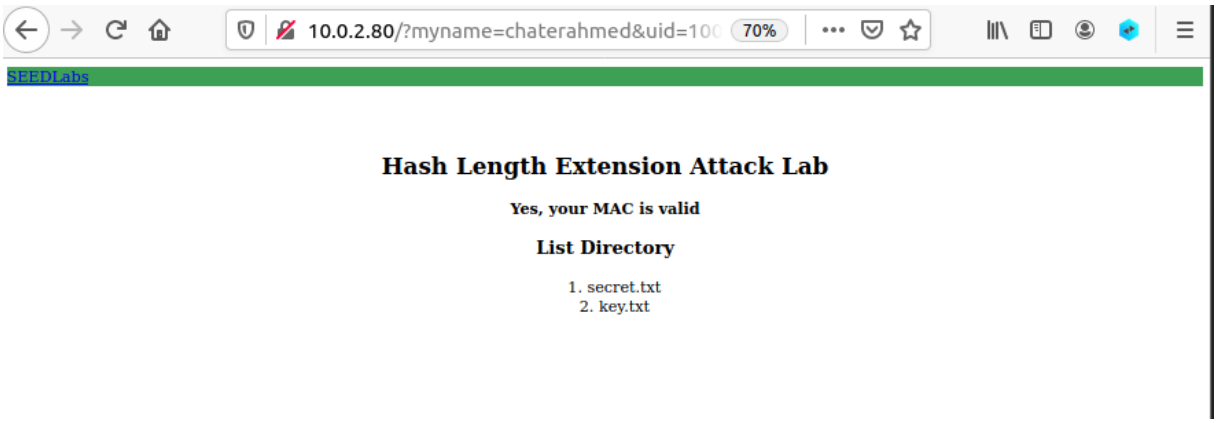
http://www.seedlab-hashlen.com/?myname=<name>&uid=<need-to-fill>&lstcmd=1&mac=<need-to-calculate>

Pour compléter cette requête, j'ai renseigné mon nom réel dans le champ myname=chaterahmed. Ensuite, j'ai choisi un uid=1002 et sa clé associée à partir du fichier key.txt situé dans le répertoire LabHome, qui contient des paires uid:key=983abe.

J'ai ensuite généré le hash SHA-256 avec la commande suivante :

```
[12/22/24]seed@VM:~/Desktop$ echo -n "983abe:myname=chaterahmed&uid=1002&lstcmd=1" | sha256sum
d3b56ca3b05bfa64809efb51eafab3adc6c0036cc50baeaea0735f3eb861b0e7  -
```

http://www.seedlab-hashlen.com/?myname=chaterahmed&uid=1001&lstcmd=1&mac=7d5f750f8b3203bd963d75217c980d139df5d0e50d19d6dfdb8a7de1f8520ce3



Hash Length Extension Attack Lab

Yes, your MAC is valid

List Directory

1. secret.txt
2. key.txt

Task 3: The Length Extension Attack :

Nous avons accompli la tâche consistant à générer un MAC valide pour une URL sans connaître la clé MAC. En supposant que nous disposions du MAC d'une requête valide R et que nous connaissions la taille de la clé MAC, nous avons réussi à forger une nouvelle requête basée sur R, tout en calculant un MAC valide.

Partant du message original M = "This is a test message" et de sa valeur MAC, nous avons démontré comment ajouter un message supplémentaire, "Extra message", à la fin de M après son remplissage (padding), et calculer le nouveau MAC sans connaître la clé secrète MAC. Cette tâche a mis en pratique les concepts liés à la manipulation de MACs et a renforcé notre compréhension des vulnérabilités potentielles dans les systèmes de vérification d'intégrité.

```
echo -n "This is a test message" | sha256sum
```

```
6f3438001129a90c5b1637928bf38bf26e39e57c6e9511005682048bedbef906
```

```
#include <stdio.h>
#include <arpa/inet.h>
#include <openssl/sha.h>
int main(int argc, const char*argv[])
{
int i;
unsigned char buffer[SHA256_DIGEST_LENGTH];
SHA256_CTX c;
SHA256_Init(&c);
for(i=0; i<64; i++)
        SHA256_Update(&c, "*", 1);
// MAC of the original message M (padded)
c.h[0] = htole32(0x6f343800);
c.h[1] = htole32(0x1129a90c);
c.h[2] = htole32(0x5b163792);
c.h[3] = htole32(0x8bf38bf2);
c.h[4] = htole32(0x6e39e57c);
c.h[5] = htole32(0x6e951100);
c.h[6] = htole32(0x5682048b);
c.h[7] = htole32(0xedbef906);
// Append additional message
SHA256_Update(&c, "Extra message", 13);
SHA256_Final(buffer, &c);
for(i = 0; i < 32; i++){
printf("%02x", buffer[i]);
}
printf("\n");
                    [ Read 29 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File ^\ Replace   ^U Paste Text^T To Spell
```

Task 4: Attack Mitigation using HMAC :

Nous avons accompli la tâche visant à corriger une erreur de calcul du MAC commise par le développeur. Jusqu'à présent, nous avons observé les risques liés à un calcul non sécurisé du MAC, où la clé et le message étaient simplement concaténés. Pour remédier à cette faille, nous avons adopté la méthode standard qui consiste à utiliser HMAC.

Nous avons modifié la fonction verify_mac() du programme serveur, située dans le fichier lab.py, afin qu'elle utilise le module hmac de Python pour calculer le MAC. En fournissant une clé et un message (tous deux sous forme de chaînes de caractères), nous avons utilisé HMAC pour garantir un calcul sécurisé du MAC. Cette amélioration renforce l'intégrité et la sécurité des communications entre le client et le serveur.

```
real_mac = hmac.new(bytearray(key.encode('utf-8')),
msg=message.encode('utf-8', 'surrogateescape'),
digestmod=hashlib.sha256).hexdigest()
```

```
def verify_mac(key, my_name, uid, cmd, download, mac):
    download_message = '' if not download else '&download=' + dow
    message = ''
    if my_name:
        message = 'myname={}&'.format(my_name)
    message += 'uid={}&lstcmd='.format(uid) + cmd + download_mess
    payload = key + ':' + message
    app.logger.debug('payload is [{}]'.format(payload))
    real_mac = hashlib.sha256(payload.encode('utf-8', 'surrogatee
    app.logger.debug('real mac is [{}]'.format(real_mac))
    if mac == real_mac:
        return True
    return False
```

Nous avons accompli la tâche de correction du calcul du MAC en utilisant HMAC avec l'algorithme SHA-256. À la suite des modifications, nous avons configuré la fonction verify_mac() dans le fichier lab.py pour qu'elle utilise digestmod=hashlib.sha256 afin de calculer le MAC et retourner son résultat en utilisant .hexdigest().

Après avoir apporté ces modifications, nous avons arrêté tous les conteneurs, les avons reconstruits, puis redémarrés pour que les changements prennent effet.

Ensuite, nous avons répété la tâche 1 en envoyant une requête pour lister les fichiers, tout en calculant le MAC à l'aide de HMAC avec la clé 123456. Cela nous a permis de confirmer que le calcul du MAC

était désormais conforme aux standards de sécurité et garantissait l'intégrité des requêtes transmises au serveur.

l'utilisation de HMAC protège contre les attaques par extension de longueur, car elle sépare efficacement la clé et le message, et le processus cryptographique garantit que seul un utilisateur possédant la clé secrète peut produire un MAC valide. Toute requête malveillante tentant de manipuler ou d'ajouter des données échouera donc à la vérification.