Exploit de la vulnérabilité CVE-2016-1494 Contrefaçon de signature dans le package python-rsa

Valérian Baillet, Matthias Beaupère

1 Introduction

La vulnérabilité CVE-2016-1494 a été soumise par Filippo Valsorda le 5 janvier 2016. Il s'agit d'une possibilité de falsifier des signatures à exposant faible dans la module rsa de python. Ce faille est présente dans les versions antérieures à 3.3.

2 Les signatures RSA

Pour signer un message en RSA, on joint ce message d'un hash du message, chiffré avec la clé privé de l'expéditeur comme ceci :

$$m^e \ mod \ N = sou(e,n)estlaclpubliquedudestinataire$$

Avec m le hash du message à signer, e l'exposant et N le modulo de la clé privé de l'expéditeur. On obtient s la signature.

Lorsque le correspondant reçoit la signature s, il effectue l'opération suivante :

$$s^d \ mod \ N = m$$

Avec d la clé privée de l'expéditeur.

Il calcule ensuite le hash du message et le compare à la signature déchiffrée m.

Si les deux hash du message ne sont pas identiques, le destinataire sait que le message n'a pas été envoyé par le bon expéditeur.

3 Faille

3.1 Système compromis

Le service compromis est la librairie python proposé pour le chiffrage rsa. Si l'exposant de chiffrement e est trop faible, il est possible de pouvoir imiter la signature d'une personne à partir d'un message intercepté. Tous les systèmes informatiques utilisant cette bibliothèque peuvent être compromis. C'est-à-dire que toutes les machines clientes ou serveur peuvent être compromis.

3.2 Vulnérabilité

La faille se trouve dans la fonction verify() de la librairie python-rsa. Cette fonction permet de vérifier l'identité du destinataire grâce à sa signature rsa (voir **??**).
Pour toute cette partie, nous allons utiliser la clé publique (3,n). En regardant le code de cette fonction, on peut voir que cette fonction accepte une certaine forme de message de signature (voir **??**)
Cette forme est la suivante :

00 01 XX XX ... XX XX 00 ASN.1 HASH

On peut voir que les chiffres centraux ont peu d'importances dans la signature. Sachant qu'il est pratiquement impossible de trouver la clé secrète du destinataire, nous allons plutôt essayer de faire une approximation de cette signature.

A partir de ce moment, on comprend que pour falsifier la signature interceptée, il suffit de reproduire le préfixe et le suffixe de cette signature. Plus spécifiquement, il faut créer une signature qui, lorsqu'elle est élevée à la puisse e, on retombe sur le préfixe et le suffixe de la signature copié.

Comme dit précédemment, les chiffres centraux de la signature ont peu d'importance donc pour notre signature. Nous allons donc utiliser la racine cubique pour le préfixe de notre signature. Cela permettra d'avoir les bons chiffres pour le préfixe avec une approximation sur les bits centraux.

La partie la plus difficile se trouve dans la façon de trouver le suffixe. Il faut trouver un message qui, élevé au cube, donne le suffixe de la signature à imiter. Posons d'abord les notations. Notons :

1. s pour le message source

2. f pour pour le message source élevé à la puissance 3
3. c pour le suffixe cible à recréer

Nous allons comparer bit à bit f et c. A chaque fois que le bit de f diffère de celui de c, nous allons changer le bit dans s et recréer c en cubant le nouveau message s. Cela devrait permettre d'avoir un message qui, une fois à la puissance trois, donne le même suffixe que la signature à copier. Les bits de poids plus fort ne sont pas un souci car cela reste une approximation de la signature.

Prenons un exemple. Nous allons essayer de créer un message qui possède le suffixe '100101011'. Nous prenons donc un message s égal à '000000001'. Par conséquent f vaut '000000001'.

Nous comparons les premiers bits de f et c. Ils sont identique donc nous ne faisons rien. Le deuxième bit de chaque message ne sont pas identique. Nous changeons donc le deuxième bit dans s et recréons f en cubant s. A ce stade, nous avons les valeurs suivantes :

1. s = '000000011'
2. f = '000011011'
3. c = '100101011'

Nous continuons notre algorithme en comparant le nouveau f créé et le c. Les bits 3 et 4 sont identiques donc nous ne faisons rien. Le bit 5 diffère donc nous changeons le bit 5 de s et nous recréons c. Nous avons les valeurs suivantes (des zéros ont été ajoutés pour plus de visibilité) :

1. s = '0000000010011'
2. f = '1101011001011'
3. c = '0000100101011'

Comme nous ne touchons pas aux bits de poids faible de s, nous savons que le début du suffixe ne va pas changer. Nous réintérons se processus jusqu'à avoir un message f avec un un suffixe identique à c. C'est à dire :

1. s = '0000000100110011'
2. f = '1000000100101011'
3. c = '0000000100101011'

Pour vérification, nous avons bien $s^3 = 110111001100000010010101 1$ qui possède le bon suffixe.

En concaténant le préfixe (avec rajout de bits de poids faibles pour avoir la longueur de la signature nécessaire) et le suffixe nous avons un signature qui est accepter par la fonction verify.Il faut juste vérifier que la concaténation du suffixe ne modifie pas les bits de poids forts. Pour cela, il suffit de prendre un suffixe assez petit.

Pour finir, il ne faut pas que les bits centraux soient des couples de zéros. Pour éviter cela, on réitère l'algorithme avec des messages s différents.

4 Annexes

4.1 Annexe 1 - Code de la fonction verify

```
def verify(message, signature, pub_key):
    blocksize = common.byte_size(pub_key.n)
    encrypted = transform.bytes2int(signature)
    decrypted = core.decrypt_int(encrypted, pub_key.e, pub_key.n)
    clearsig = transform.int2bytes(decrypted, blocksize)

    # If we can't find the signature marker, verification failed.
    if clearsig[0:2] != b('\x00\x01'):
        raise VerificationError('Verification failed')

    # Find the 00 separator between the padding and the payload
    try:
        sep_idx = clearsig.index(b('\x00'), 2)
    except ValueError:
        raise VerificationError('Verification failed')

    # Get the hash and the hash method
    (method_name, signature_hash) = _find_method_hash(clearsig[sep_idx+
    message_hash = _hash(message, method_name)

    # Compare the real hash to the hash in the signature
    if message_hash != signature_hash:
        raise VerificationError('Verification failed')
```

```
    return True

def _find_method_hash(method_hash):
    for (hashname, asn1code) in HASH_ASN1.items():
        if not method_hash.startswith(asn1code):
            continue

        return (hashname, method_hash[len(asn1code):])

    raise VerificationError('Verification failed')

HASH_ASN1 = {
    'MD5': b('\x30\x20\x30\x0c\x06\x08\x2a\x86'
             '\x48\x86\xf7\x0d\x02\x05\x05\x00\x04\x10'),
    'SHA-1': b('\x30\x21\x30\x09\x06\x05\x2b\x0e'
               '\x03\x02\x1a\x05\x00\x04\x14'),
    'SHA-256': b('\x30\x31\x30\x0d\x06\x09\x60\x86'
                 '\x48\x01\x65\x03\x04\x02\x01\x05\x00\x04\x20'),
    'SHA-384': b('\x30\x41\x30\x0d\x06\x09\x60\x86'
                 '\x48\x01\x65\x03\x04\x02\x02\x05\x00\x04\x30'),
    'SHA-512': b('\x30\x51\x30\x0d\x06\x09\x60\x86'
                 '\x48\x01\x65\x03\x04\x02\x03\x05\x00\x04\x40'),
}
```