
*Introduction à la sécurité des systèmes d'information*

# Devoir 1 : CVE-2016-1494

Corentin COURNAC - Pierre PRÉSENT - Guillaume THIOLLIÈRE

3 Janvier 2017

**Quelle est la faille ou vulnérabilité étudiée ?**

Dans ce rapport, nous étudions la faille identifiée CVE-2016-1494 : RSA package for python implementation.

**Quel est le service ou le programme compromis ?**

La fonction *"verify"* dans le paquet RSA python nommé *"Python-RSA"* avant sa version 3.3 permet d'effectuer une attaque visant à usurper les signatures avec un petit exposant de chiffrement (notation *e*) via un rembourrage de signature, plus communément appelée *"BERserk attack"*.

**Quel est le type de compromission ?**

Le type de compromission est l'usurpation d'identité (via la signature).

**Expliquer la vulnérabilité, décrire le mécanisme permettant de l'exploiter :**

Cette vulnérabilité a pour origine une erreur d'implémentation de la fonction de vérification de signature RSA dans les versions antérieures à la version 3.3 de Python-RSA, qui permet de forger une signature pour des messages arbitraires lorsque l'exposant de chiffrement *e* de la clé publique a une petite valeur, telle que 3, qui reste l'une des valeurs les plus utilisées pour cet exposant.

Pour présenter la vulnérabilité, voyons tout d'abord comment sont encodées ces signatures :

Ces signatures suivent la norme PKCS#1.5, qui préconise l'utilisation de signatures de la forme `00 01 FF FF … FF FF 00 ASN.1 HASH`

Avec `ASN.1` un nombre qui encode le type de hachage ainsi que la taille du message, et `HASH` le hash du message en question. Tous les octets `FF` permettent de remplir ("rembourrer") le message afin que celui-ci ait une longueur égale au module de chiffrement *N.*

Il est impossible sans avoir l'exposant de déchiffrement *d* d'obtenir un nombre qui, lorsqu'on l'élève à la puissance *e*, donnerait une telle signature.

Cependant, la fonction de vérification de Python-RSA ne vérifie pas que la signature soit exactement sous cette forme là. Celle-ci a été codée pour reconnaître le `00 01` du début, puis de sauter jusqu'au prochain octet `00`, avant de lire `ASN.1` et `HASH`. (cf le code source de la fonction `verify()` sur https://bitbucket.org/sybren/python-rsa/src/2baab06c8b867b01ec82b02118d4872a931a0437/rsa/pkcs1.py?at=default&fileviewer=file-view-default#pkcs1.py-279 )

Ainsi, Python-RSA va lire des signatures de la forme `00 01 XX XX … XX 00 ASN.1 HASH` , avec les `XX` des octets quelconques différents de `00`.

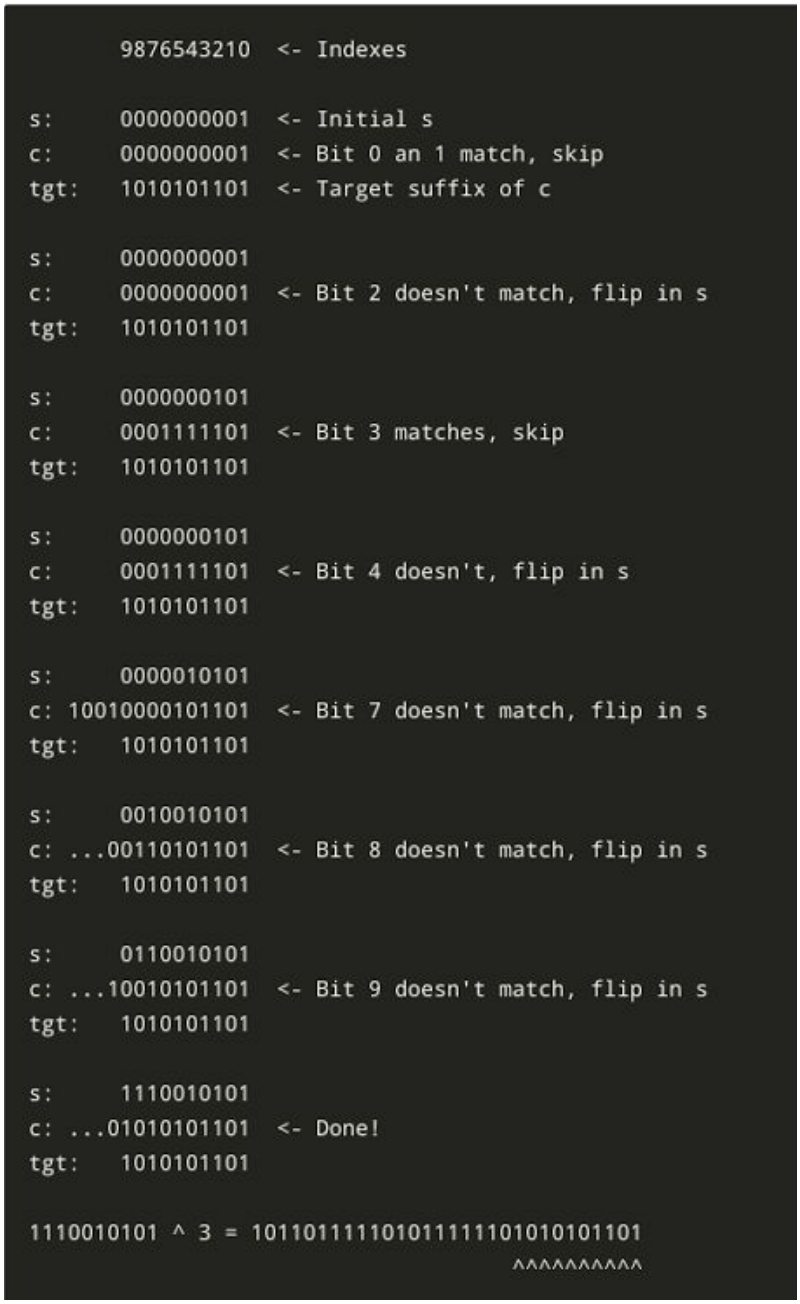
Le but est désormais d'obtenir un message qui, lorsqu'il est mis à la puissance e, avec e petit (e=3 pour la suite), va démarrer par les octets `00 01`, et finir par `00 ASN.1 HASH` du message arbitraire, avec à l'intérieur suffisamment d'octets de rembourrage différents de `00`. Tous ces bits de rembourrage permettent alors de cacher l'imperfection de la mise à la racine cubique du message.

Pour obtenir le préfixe `00 01 XX XX ...`, l'opération est simple : comme *e* est petit, la mise à la puissance *e* ne va pas dépasser le module. Prendre la racine cubique (rappel : *e*=3), bien qu'imparfaite, d'un tel nombre suffira à obtenir un message qui, élevé à la puissance *e*, donnera un message de cette forme.

Le suffixe `00 ASN.1 HASH` peut être calculé en itérant sur chacun des bits du message forgé, jusqu'à ce que le message en question, élevé au cube, ait un suffixe égal à ce suffixe.

Le principe de base de cet algorithme est que, lorsque l'on inverse le bit N sur le message forgé, alors cela va inverser le bit N sur ce message élevé au cube, sans affecter les bits de 0 à N-1. Ainsi, on va itérer sur chacun des bits du message forgé, en les inversant lorsque le bit de même indice du message élevé au cube est différent de celui que l'on cible, en partant du bit d'indice 0, et jusqu'à obtenir un message élevé au cube dont les derniers bits sont égaux à `00 ASN.1 HASH`.

Un exemple est présent sur l'image ci-dessous :




On peut de cette manière obtenir un message qui, élevé au cube, possède le suffixe `00 ASN.1 HASH`.

Nous avons désormais deux messages : un qui, élevé au cube, a comme préfixe `00 01 XX XX …` avec tous les `XX` différents de `00`, et un qui, élevé au cube, a comme suffixe `00 ASN.1 HASH`.

Il s'agit maintenant de coller ces messages de façon à obtenir un message qui combine ce préfixe et ce suffixe. On va tronquer autant de bits à la fin du message contenant le préfixe que le nombre de bits du message contenant le suffixe, puis simplement concaténer ces messages, de façon à obtenir un message de longueur *N* (le module de chiffrement), dont le début contient le début du message préfixe, et la fin contient entièrement le message suffixe.

Nous avons vu précédemment que, lorsque l'on change certains bits, tous les bits de poids plus faible que celui-ci ne seront pas changés lorsque l'on élèvera au cube le message. Le suffixe de ce message au cube sera donc toujours `00 ASN.1 HASH`.

Ensuite, les bits du suffixe sont suffisamment peu nombreux pour qu'ils n'aient pas d'influence sur les 2 premiers octets du message élevé au cube.

On obtient alors un message forgé qui, élevé au cube, est de la forme `00 01 XX XX … XX XX 00 ASN.1 HASH`.

Cependant, après concaténation, certains des octets `XX` peuvent être égaux à `00`.

Pour résoudre ce problème, il suffit tout bonnement de recalculer le préfixe en prenant la racine cubique d'un autre nombre de la même forme, en changeant les valeurs des `XX` aléatoirement avant le passage à la racine cubique. En répétant suffisamment ce processus, nous obtiendrons finalement un message de la forme voulue, et n'ayant aucun bit égal à `00` dans le rembourrage.

Maintenant que nous avons une signature qui est reconnue comme valide par Python-RSA 3.3, plus rien ne nous empêche d'envoyer des messages compromis contenant des maliciels à un serveur qui vérifiait ses sources.

**Cette faille concerne-t-elle des machines clientes ou des machines serveurs ?**

Cette faille peut autant usurper la signature d'un message client, que celle d'un message serveur. Les deux sont donc concernés.

**Décrire une architecture typique du système d'information qui pourrait être impliquée dans l'exploitation de ces failles :**

Pour effectuer une mise à jour système, un OS doit se connecter à un serveur de référence, connu et de confiance, dans le but de récupérer les nouvelles mises à jour. Si un utilisateur malveillant réussit à intercepter puis usurper la signature d'un serveur de mise à jour, il lui est tout à fait possible de transférer au client n'importe quel exécutable malveillant.

**Que préconiseriez-vous pour limiter l'impact de l'exploitation de ces failles ?**

Il n'y a pas de limitation possible une fois que la signature a été usurpée. La seule solution pour utiliser le paquet Python-RSA avant 3.3 tout en limitant l'exploitation de cette faille, est de prendre un exposant de chiffrement assez grand (ex : e=65537), pour que cette attaque ne soit pas réalisable.

**Que préconiseriez-vous pour empêcher qu'elles ne puissent être exploitées ?**

La solution la plus simple et la plus efficace est d'utiliser au minimum la version 3.3 de Python-RSA, car c'est à ce numéro de version que la faille a été corrigée (référence du commit ici), pour tous les systèmes communicants concernés.

**Proposer une expérimentation permettant de mettre en évidence la vulnérabilité et son exploitation :**

On simule une application qui reçoit des messages depuis internet et qui vérifie la source du message. L'exemple affichera sur la sortie standard si le message a bien été signé et affichera qu'il n'a pas confiance dans le message sinon. L'exemple ne fera confiance qu'en la clef publique valant 3.

```
while True:
	message = sock.recv(4096)
	sig = sock.recv(4096)
	if rsa.verify(message, sig, key):
		print("I trust your message:", message.decode("ASCII"))
	else:
		print("I don't trust your message")
```

Pour exploiter la faille, on va envoyer un message et créer une signature :

```
message_hash = hashlib.sha256(message).digest()

ASN1_blob = rsa.pkcs1.HASH_ASN1['SHA-256']
suffix = b'\x00' + ASN1_blob + message_hash

sig_suffix = 1
for b in range(len(suffix)*8):
	if get_bit(sig_suffix ** 3, b) != get_bit(from_bytes(suffix), b):
		sig_suffix = set_bit(sig_suffix, b, 1)

while True:
	prefix = b'\x00\x01' + os.urandom(2048//8 - 2)
	sig_prefix = to_bytes(cube_root(from_bytes(prefix)))[:-len(suffix)] + b'\x00' * len(suffix)
	sig = sig_prefix[:-len(suffix)] + to_bytes(sig_suffix)
	if b'\x00' not in to_bytes(from_bytes(sig) ** 3)[:-len(suffix)]: break
```

Puis on va l'envoyer:

```
UDP_IP = "127.0.0.1"
UDP_PORT = 5005

sock = socket.socket(socket.AF_INET, # Internet
		socket.SOCK_DGRAM) # UDP

sock = socket.socket(socket.AF_INET, # Internet
		socket.SOCK_DGRAM) # UDP
sock.sendto(message, (UDP_IP, UDP_PORT))
sock.sendto(sig, (UDP_IP, UDP_PORT))
```

Dans cet exemple, si le serveur utilise une version de python-rsa inférieure à 3.3, alors le serveur affichera le message construit par le pirate.

**Le code source complet est disponible à l'adresse suivante https://github.com/thiolliere/exemple_securite**

# Glossaire

**Exposant de chiffrement e :** Entier naturel e premier avec φ(n) et strictement inférieur à φ(n) tel que φ(n) = (p - 1)(q -1) avec p et q deux nombres premiers distincts (Wikipedia)

**Exposant de déchiffrement d :** Entier naturel d, inverse (modulaire) de e modulo φ(n), et strictement inférieur à φ(n) (Wikipedia)

**Inversion de bit :** Changer la valeur d'un bit (0 → 1, 1 → 0)

**OS :** Operating System, ou Système d'Exploitation (SE) en français.

**Signature :** Élément d'un message permettant d'authentifier son auteur.

# Références


https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1494

https://blog.filippo.io/bleichenbacher-06-signature-forgery-in-python-rsa/

https://bitbucket.org/sybren/python-rsa/pull-requests/14/security-fix-bb06-attack-in-verify-by/diff

https://bitbucket.org/sybren/python-rsa/src/2baab06c8b867b01ec82b02118d4872a931a0437/rsa/pkcs1.py?at=default&fileviewer=file-view-default#pkcs1.py-279