Techniques d'attaque - chiffrement RSA

Tristan BILOT, Nora DELFAU, Enzar SALEMI, Madushan THAMBITHURAI
EPITA

31 Mai 2021

Abstract

Vous travaillez dans le département de criminologie informatique du département de surveillance des cyber-territoires. Les services secrets ont intercepté un message chiffré (texte chiffré) envoyé par une personne (sous surveillance) à son ami. Le message a été chiffré à l'aide de l'algorithme de chiffrement symétrique AES-256-CBC. La clé avec laquelle ce message a été chiffré a été dérivée d'un mot de passe envoyé chiffré par l'algorithme de chiffrement asymétrique RSA. Vous devez casser la clé et trouver le message initial (texte brut). Selon les premières informations dont vous disposez, l'outil qui a été utilisé pour générer la paire de clés asymétriques a été mal configuré. Par conséquent, la qualité des nombres premiers choisis était médiocre. Le module de chiffrement a été calculé par la multiplication de deux nombres premiers (un grand p et un q relativement petit) choisis au hasard à condition que leur produit soit de la taille souhaitée (2048 bits dans cet exemple).

1 Notes

1.1 RSA

RSA est l'algorithme de chiffrement asymétrique le plus utilisé au monde, surtout pour les communications sur Internet. Voici son fonctionnement:

- Choisir p et q, deux nombres premiers distincts
- Calculer leur produit $n = pq$, appelé module de chiffrement
- Calculer $\phi(n) = (p-1)(q-1)$ (c'est la valeur de l'indicatrice d'Euler en n)
- Choisir un entier naturel e premier avec $\phi(n)$ et strictement inférieur à $\phi(n)$, appelé exposant de chiffrement
- Calculer l'entier naturel d, inverse de e modulo $\phi(n)$, et strictement inférieur à $\phi(n)$, appelé exposant de déchiffrement ; d peut se calculer efficacement par l'algorithme d'Euclide étendu.

Dans le cas où $n = p * q$ utilise des p et q grands et de taille proche (ex: p=100, q=100), l'algorithme est stable et il est en pratique aujourd'hui impossible de factoriser n en temps polynomial. Cependant, si le choix de p et q utilise des nombres trop petits et différents (p=20, q=100), il va être possible d'effectuer

la factorisation sur le plus petit des deux et le chiffrement sera donc cassable. Toute la robustesse du chiffrement RSA tient sur le fait que quand p et q sont des nombres premiers très grands, il est très compliqué de trouver la factorisation de $\phi(n)$, qui est nécessaire d'être connu afin de déchiffrer le message.

2 Implémentation avec openSSL

Afin de pouvoir implémenter RSA sur notre propre machine, nous utiliserons openSSL. Un chiffrement/déchiffrement d'un message via RSA peut être effectué en seulement 4 étapes:

- Générer une clé publique
- Générer une clé privée grâce à la clé publique
- Chiffrer un message
- Le déchiffrer

Chaque étape peut être synthétisé en une ligne de commande openSSL:

```
openssl genrsa -out PrivateKeyTristan 2048
openssl rsa -in PrivateKeyTristan -pubout -out PublicKeyTristan
openssl rsautl -encrypt -pubin -inkey public.key -in plaintext.txt -out encrypted.txt
openssl rsautl -decrypt -inkey private.key -in encrypted.txt -out plaintext.txt
```

Dans le cas précédent, nous commençons par générer une clé privée PrivateKeyTristan de taille 2048bit, puis nous créons une clé publique PublicKeyTristan grâce à la clé privée. Le modulus est donné son forme hexadécimale, il va donc falloir le convertir en décimal avant de pouvoir travailler dessus. Il ne reste plus qu'à créer un fichier d'exemple (ici plaintext.txt) que l'on va chiffrer grâce à la clé publique. Le déchiffrement s'effectuera quant à lui grâce à la clé privée.

3 Factoriser une clé RSA

A présent, partons d'une clé publique donnée:

```
o cat public.pem
-----BEGIN PUBLIC KEY-----
MIIBJTANBgkqhkiG9w0BAQEFAAOCARIAMIIBDQKCAQQAjaVpGbUm1FIlrO1L5kUi
zvBKY5ELn2/+prESVUEBO+RdSLb7JnG3VA5qTgtV46nkxFqNX1SgaZxlMtShKH+s
sAixxW411gHcKp4uZlGJ6qNdIte+olLB7PJwMatlfVs16CzecPglnS4U6YbzYuPo
bnvY5IEqUvLozC9puLDJWXeP2yQKjhfLlXJFcBLYO2xycpDlC459oo/r36v1I9oD
uUuUMir0IfnKAq3mBNqrks2cKCRENvEV/b7XbSyFAHrKf85JiTr4DnlVYy7HuJ9W
+rAYduD+iCmaNzQNQ5yy4Bs8B+YMiEdCG/4EnVmVQGsm9KCLINFJs8YarqNTHWL4
+NHihwIDAQAB
-----END PUBLIC KEY-----
```

Figure 1: Clé publique que l'on souhaite casser

Nous allons extraire des informations sur la méthode de chiffrement: notamment le modulus et l'exponent. Le modulus est en fait la variable n et l'exponent e. Il ne nous manque plus qu'a trouver $\phi(n) = (p-1)(q-1)$, donc essayer toutes les valeurs possibles de p et q. L'extraction des variables peut

```
 o openssl rsa -in public.pem -pubin -text -noout
Public-Key: (2072 bit)
Modulus:
    00:8d:a5:69:19:b5:26:d4:52:25:ac:ed:4b:e6:45:
    22:ce:f0:4a:63:91:0b:9f:6f:fe:a6:b1:12:55:41:
    01:3b:e4:5d:48:b6:fb:26:71:b7:54:0e:6a:4e:0b:
    55:e3:a9:e4:c4:5a:8d:5f:54:a0:69:9c:65:32:d4:
    a1:28:7f:ac:b0:08:b1:c5:6e:35:d6:01:dc:2a:9e:
    2e:66:51:89:ea:a3:5d:22:d7:be:a2:52:c1:ec:f2:
    70:31:ab:65:7d:5b:35:e8:2c:de:70:f8:25:9d:2e:
    14:e9:86:f3:62:e3:e8:6e:7b:d8:e4:81:2a:52:f2:
    e8:cc:2f:69:b8:b0:c9:59:77:8f:db:24:0a:8e:17:
    cb:95:72:45:70:12:d8:3b:6c:72:72:90:e5:0b:8e:
    7d:a2:8f:eb:df:ab:f5:23:da:03:b9:4b:94:32:2a:
    f4:21:f9:ca:02:ad:e6:04:da:ab:92:cd:9c:28:24:
    44:36:f1:15:fd:be:d7:6d:2c:85:00:7a:ca:7f:ce:
    49:89:3a:f8:0e:79:55:63:2e:c7:b8:9f:56:fa:b0:
    18:76:e0:fe:88:29:9a:37:34:0d:43:9c:b2:e0:1b:
    3c:07:e6:0c:88:47:42:1b:fe:04:9d:59:95:40:6b:
    26:f4:a0:8b:20:d1:49:b3:c6:1a:ae:a3:53:1d:62:
    f8:f8:d1:e2:87
Exponent: 65537 (0x10001)
```

Figure 2: Extraction du modulus et de l'exponent

se faire grâce à openssl sur la clé publique. Nous savons que la génération de la clé a été faussée par le choix d'un p petit, il nous sera donc simple dans ce cas de factoriser, mais bien évidemment, dans un cas réel p et q seront tellement grands qu'il nous faudrait des décennies avant de pouvoir factoriser sur nos meilleurs ordinateurs.

```
public static void main (String[] args){
    var n = new BigInteger( val: "2999962175617872927568262512400737446
    var p = new BigInteger( val: "2");
    var e = new BigInteger( val: "65537");

    while (n.mod(p).compareTo(BigInteger.valueOf(0)) != 0) {
        if (p.isProbablePrime( certainty: 1))
            System.out.println(p);
        p = p.add(BigInteger.valueOf(1));
    }
    var q = n.divide(p);
    var pMin1 = p.subtract(BigInteger.valueOf(1));
    var qMin1 = q.subtract(BigInteger.valueOf(1));
    var phiOfN = pMin1.multiply(qMin1);
    var d = e.modInverse(phiOfN);

    System.out.println("Values found:");
    System.out.println("d = " + d);
    System.out.println("q = " + q);
    System.out.println("p = " + p);
}
```

Figure 3: Algorithme de factorisation

4 Format ASN.1 DER

Tout d'abord, créons un fichier private.txt contenant nos variables au format ASN.1: A partir de ce fichier private.txt, nous pouvons générer un fichier der:

```
asn1=SEQUENCE:rsa_key

[rsa_key]
version=INTEGER:0
modulus=INTEGER:29999621756178729275682625124007374402258736442
publicExponent=INTEGER:65537
privateExponent=INTEGER:27772203034339492454195785106224251732 8
prime1=INTEGER:10038779
prime2=INTEGER:29883735617826360432561196061799322808340273695 4
exponent1=INTEGER:4104691
exponent2=INTEGER:13729181923449866278041183640153131977910452 1
coefficient=INTEGER:2852744
```

Figure 4: Clé RSA au format ASN.1

`openssl asn1parse -genconf private.txt -out private.der`

Puis vérifier que le certificat est valide en exportant toutes ces variables:

`openssl rsa -in private.der -inform der -text -check`

Il nous est ensuite possible de reconstituer la clé privée avec:

`openssl rsa -in private.der -inform der -out private_key.priv`

5 Déchiffrement

Grâce à cette clé privée, nous allons pouvoir déchiffrer la clé AES que nous avons:

```
 o openssl rsautl -decrypt -inkey private_key.priv -in decoded -out SOLUTION
tristanbilot at root in ~/Documents/dev/epita/secu
 o cat SOLUTION
unbreakablePass
```

Figure 5: Déchiffrement de la clé AES

Maintenant que la clé AES est en clair, il va être possible de l'utiliser pour déchiffrer le message originel. Il est important d'ajouter les mêmes paramètres de déchiffrement que ceux utilisés pour le chiffrement: c'est à dire aes-256-cbc sans sel. Le message est enfin déchiffré.

```
┌──(madu㉿maduKAli)-[~/Bureau/technique_attaque/tp2]
└─$ openssl enc -d -base64 -aes-256-cbc -in msg_a_decrypter -out message_decrypted -k "unbreakablePass" -nosalt
*** WARNING : deprecated key derivation used.
Using -iter or -pbkdf2 would be better.

┌──(madu㉿maduKAli)-[~/Bureau/technique_attaque/tp2]
└─$ cat message_decrypted
Cible: coffre fort de l'Epita
Ce soir à 23:00
```

Figure 6: Déchiffrement du message avec la clé AES