Documentation CLIP

1206

Générateur d'aléa noyau

Version	Date	Auteur	Commentaires
1.0	04/09/2008	Vincent Strubel	Version initiale, à jour pour *clip-kernel-2.6.22.24-r9*.

Table des matières

Introduction....4
1 Générateur d'aléa standard....5
1.1 Taille des pools augmentée....5
1.2 Polynômes générateurs adaptés....6
1.3 Adaptations mineures de l'ajout à un pool....7
2 Générateur avec retraitement CCSD....8
3 Mise en oeuvre....9
3.1 Utilisation par la couche noyau....9
3.2 Utilisation par la couche utilisateur....9
3.3 Initialisation et sauvegarde de l'état....10
Annexe A Références....12

Introduction

Les postes CLIP utilisent des nombres aléatoires pour de nombreuses opérations, notamment cryptographiques, dont la sécurité dépend de la qualité de l'aléa utilisé. Le générateur d'aléa noyau constitue la seule source d'aléa pour ces opérations sensibles, aussi bien en couche noyau qu'en couche utilisateur. Le générateur standard du noyau Linux est normalement utilisé, avec des modifications mineures apportées principalement par le *patch grsecurity*. Ce générateur étant largement documenté par ailleurs, le présent document se borne à décrire (en section 1) les divergences de la version déployée dans CLIP par rapport au modèle standard. Alternativement à ce générateur standard, les postes CLIP intègrent un générateur d'aléa noyau réalisant un retraitement CCSD de la sortie du générateur standard. Ce générateur, décrit en section 2 du présent document, n'est à ce stade pas employé au sein des systèmes CLIP.

1 Générateur d'aléa standard

La principale source d'aléa utilisée au sein d'un système CLIP est le générateur d'aléa standard du noyau, qui est dans une large mesure conforme à la description qui en est donné dans la section 2 du document de référence [LRNG]. Cette section se limite donc à la description des différentiels entre cette description de référence et l'implémentation mise en oeuvre dans CLIP.

1.1 Taille des *pools* augmentée

L'option *grsecurity GRKERNSEC_RANDNET* (cf. [CLIP_1203]), qui est activée dans les noyaux CLIP, multiplie par quatre les tailles des *pools* d'entropie, aussi bien primaire que secondaire et *urandom*. Les tailles effectivement utilisées dans un noyau CLIP sont donc :

- 128 mots de 4 octets pour les *pools* secondaire et *urandom*, au lieu de 32.
- 512 mots de 4 octets pour le *pool* primaire, au lieu de 128.

Cette augmentation de taille entraîne aussi une modification de la formule d'extraction d'aléa d'un *pool*, par rapport à celle donnée dans la figure 2.3 du document de référence [LRNG]. La fonction d'extraction consiste à réaliser une transformation *sha_transform()* (*lib/sha1.c*), correspondant à une étape de hachage SHA1[1], pour chaque bloc de 64 octets du *pool* source, suivie de la réinjection d'un mot de 4 octets du résultat de ce hachage dans le *pool* source. La formule générique est ainsi la suivante, pour l'extraction de 5 octets d'aléa d'un *pool* quelconque :

```
extract_pool_512(pool, u32 out[16]) :
/* Fonction symbolique d'extraction directe de 64 octets d'aléa.
   En pratique, ce traitement est inclus dans le dernier appel
   add_pool()
*/

j <- position_courante(pool)
n <- taille(pool) /* Taille en octets */

for (i = 0; i < 16; i = i + 1)
      out[i] = pool[ (j - i) mod n ]

position_courante(pool) = (j - 16) mod n

/* 16 octets d'entropie retournés dans out[] */


extract_step(pool) :

u32 buf[5]  /* 5 mots de 4 octets */

SHA1_init(buf)

/* taille(pool) : taille en mots de 4 octets, soit 128 pour secondaire et
```

[1] Sans ré-initialisation du contexte, c'est-à-dire en utilisant le résultat du hachage précédent comme vecteur d'initialisation. Une telle étape est ainsi équivalente à un appel *update()* de la fonction de hachage SHA1 standard du noyau (cf. [CLIP_1205]).

```
urandom, et 512 pour primaire */

for (i = 0; i < taille(pool); i = i + 16)
      SHA1(pool [i <-> i+16], buf) /* Hachage de 512 bits de pool dans buf */
      add_pool(pool, buf[ i % 5], 1)

add_pool(pool, buf[ i % 5], 1)
u32 data[16]
extract_pool_512(pool, data)

SHA1(data, buf)

/* "Folding" du résultat (20 octets) sur 10 octets */
buf[0] = buf[0] ^ buf[3]
buf[1] = buf[1] ^ buf[4]
buf[2] = buf[2] ^ roll32 (buf[2], 16) /* buf2 décalé de 16 bits (gauche) */

résultat extrait <- 10 premiers octets de buf
```

Dans cette description, '^' représente l'opération XOR bit à bit de deux mots, *SHA1(data, buf)* correspond à une étape de hachage de *data* (64 octets) dans *buf* (20 octets), et *add_pool(pool, w, 1)* la fonction d'injection d'un mot *w* dans *pool*, telle qu'elle est détaillée en section 1.3.

On notera que cette augmentation de taille complexifie sensiblement l'attaque théorique générique sur la *forward security* présentée en section 3.1 de l'article [LRNG]. En effet, le calcul par force brute de l'état avant extraction d'un *pool,* à partir de la connaissance de cet état à l'instant suivant l'extraction, nécessite de deviner (en notant *j* la position courante du *pool* avant l'extraction, comme dans l'article) :

- les mots aux positions *j, j-1, ..., j-8* dans le cas des *pools* secondaire et *urandom*, soit 2^{288} possibilités (au lieu des 2^{96} cités dans l'article).
- les mots aux positions *j, j-1, ..., j-32* dans le cas du *pool* primaire, soit 2^{1056} possibilités[2].

L'augmentation de taille a par ailleurs un effet similaire sur l'attaque 'optimisée' en 2^{64} calculs présentée par les auteurs pour certaines valeurs de *j* : l'optimisation décrite permet uniquement de deviner la valeur précédente à l'index *j*, ce qui laisse une complexité d'ordre 2^{256} pour les *pools* secondaire et *urandom*, et 2^{1024} pour le *pool* primaire.

Etant donné ces degrés de complexité, et le fait que ces attaques supposent un accès en lecture de l'attaquant à une zone mémoire réservée au noyau, immédiatement après la génération d'un nombre aléatoire sensible, et avant toute ré-injection d'entropie par un événement extérieur, les attaques sur la *forward security* du RNG Linux présentées dans [LRNG] sont considérées comme non significatives.

1.2 Polynômes générateurs adaptés

La modification des tailles de *pools* par *GRKERNSEC_RANDNET* entraîne une modification correspondante des polynômes utilisés pour mettre à jour ces *pools* (cf. [LRNG], section 2.5). Les polynômes utilisés dans les noyaux CLIP sont ainsi :

[2] L'attaque en $O(2^{96})$ sur le *pool* primaire, décrite dans [LRNG], est fausse, résultant d'une application erronée de la formule d'extraction simplifiée établie par les auteurs pour un *pool* de 32 mots. La complexité réelle de cette attaque, pour un *pool* primaire de 128 mots, est en fait de 2^{288}.

- Pour le *pool* primaire :

$$x^{512} + x411 + x^{308} + x^{208} + x^{104} + x + 1$$

- Pour les *pools* secondaire et *urandom* :

$$x^{128} + x^{103} + x^{76} + x^{51} + x^{25} + x + 1$$

1.3 Adaptations mineures de l'ajout à un *pool*

La fonction d'addition d'entropie à un *pool* diffère elle aussi de celle décrite dans [LRNG], cette fois du fait des évolutions du noyau Linux elles-mêmes, qui ajoutent une rotation des mots ajoutés d'un nombre de bits *input_rotate* variable. Cette fonction prend ainsi la forme suivante :

```
twist[8] := {     0x00000000, 0x3b6e20c8, 0x76dc4190, 0x4db26158,
                  0xedb88320, 0xd6d6a3e8, 0x9b64c2b0, 0xa00ae278 }

input_rotate := 0; /* Initialement */

pool primaire :
(a1, a2, a3, a4, a5) := ( 411, 308, 208, 104, 1 )

pools secondaire et urandom :
(a1, a2, a3, a4, a5) := ( 103, 76, 51, 25, 1 )

add_pool_one(pool, j, word)
      n <- taille(pool)
      w <- rol32(word, input_rotate)
      /* Rotation de input_rotate bits vers la gauche */

      if (j)
            input_rotate <- (input_rotate + 21) & 31
      else
            input_rotate <- (input_rotate + 14) & 31

      w <- w ^ pool [ (j + a1) mod n ]
      w <- w ^ pool [ (j + a2) mod n ]
      w <- w ^ pool [ (j + a3) mod n ]
      w <- w ^ pool [ (j + a4) mod n ]
      w <- w ^ pool [ (j + a5) mod n ]
      w <- w ^ pool [ j ]

      pool [ j ] <- (w >> 3) ^ twist [ w & 7 ]

add_pool(pool, words, nwords)
      n <- taille(pool)
      j <- position_courante(pool)

      for (/* chaque mot word dans words (nwords étapes) */)
            j = (j - 1) mod n
            add_pool(pool, j, word)
```

2 Générateur avec retraitement CCSD

Alternativement au générateur standard (sauf modifications *grsecurity*) du noyau Linux, les noyaux CLIP intègrent un générateur d'aléa basé sur un retraitement par CCSD ([CCSD]) de la sortie du générateur standard. Ce générateur fournit uniquement une interface à la couche utilisateur (à travers un *device* en mode caractère */dev/crandom*, de majeur 1 et mineur 16), et n'inclut pas d'interface interne au noyau, équivalente à *get_random_bytes()*[3]. Ce générateur CCSD est fourni par le module *ccsd* réalisant l'intégration de CCSD à la couche noyau (cf. [CLIP_1205]), lorsque ce dernier est compilé avec l'option *CONFIG_CCSD_RNG*.

La génération d'aléa est réalisée par des appels à la fonction *CC_GenereAlea()*, portant sur une session CCSD dédiée. Celle-ci est initialisée au chargement de module avec un bruit *noise* de la taille correspondant à la caractéristique *MinNoise()* de la bibliothèque CCSD, et produit par un appel *get_random_bytes()*[4], ainsi qu'avec un pointeur de fonction de lecture d'aléa supplémentaire (argument *noisehook* de *CC_Initialise()*) correspondant elle aussi à *get_random_bytes()*. Dans un tel cas, la sortie de *CC_GenereAlea()* correspond à un mélange du pseudo-aléa extrait du *pool urandom* du noyau par *get_random_bytes()* et de celui produit par le générateur interne de la bibliothèque à partir de la "graine" *noise*.

Afin d'éviter des accès concurrents potentiellement non supportés aux fonctions CCSD, le générateur d'aléa protège les appels à *CC_GenereAlea()* par la prise d'un *mutex*. L'aléa lu sur */dev/crandom* est produit par tranches de 16 octets (boucle d'appels à *CC_GenereAlea()* avec une longueur 16, jusqu'à atteindre la longueur demandée). Le *mutex* est pris et relâché à chaque itération, de manière à éviter qu'un processus ne puisse causer un déni de service sur le générateur en lisant un aléa arbitrairement long.

L'ouverture du *device crandom* est gérée par le *hook clsm_inode_memdev_open()* du LSM CLIP (cf. [CLIP_1201]), lorsque ce dernier est compilé avec l'option *CONFIG_CCSD_RNG*. Ce *device* n'est à ce stade pas utilisé en pratique comme source d'aléa dans les systèmes CLIP. Il pourrait cependant, au terme d'une étude plus poussée de sa sécurité, être amené à remplacer */dev/urandom* pour certains usages.

[3] Cette absence d'interface interne permet de conserver la "modularisabilité" de l'intégration CCSD au noyau. Elle est aussi justifiée par la plus grande lenteur de la génération avec retraitement CCSD (en moyenne de l'ordre de 12 fois plus lente que */dev/urandom* pour la couche utilisateur), qui peut être plus gênante en couche noyau.

[4] On notera de plus que ce bruit est différent de celui utilisé pour les transformations cryptographiques basées sur CCSD, telles qu'elles sont décrites dans [CLIP_1205].

3 Mise en oeuvre

3.1 Utilisation par la couche noyau

Le noyau CLIP utilise exclusivement l'interface *get_random_bytes()*, puisant dans le *pool urandom*, pour la génération d'aléa nécessaire à ses traitements. Celle-ci est principalement utilisée pour :

- Initialiser le bruit des sessions CCSD décrites en [CLIP_1205], et le cas échéant de celle du générateur décrit en section 2.
- Fournir au besoin l'aléa complémentaire requis par les transformations CCSD décrites en [CLIP_1205].
- Générer des vecteurs d'initialisation (IVs) aléatoires pour le chiffrement IPsec en confidentialité ([CLIP_1205]).
- Fournir les *offsets* aléatoires intervenant dans la randomisation des projections mémoire, telle qu'elle est décrite en [CLIP_1203].
- Fournir les numéros de séquence TCP, et numéros de ports éphémères pour la couche IP.

3.2 Utilisation par la couche utilisateur

Au sein de la couche utilisateur CLIP, le générateur non-bloquant (*/dev/urandom, pool urandom*) est utilisé de préférence au générateur bloquant (*/dev/random*, *pool* secondaire), afin de ne pas exposer les fonctions de sécurité critiques qui nécessitent la lecture d'aléa à des dénis de service par épuisement de l'entropie disponible dans le *pool* secondaire. En particulier, le générateur bloquant */dev/random* est réservé au socle CLIP, et n'est exposé dans aucune des cages du système, où il est remplacé par un lien symbolique vers le générateur non-bloquant (cf. [CLIP_1304] et [CLIP_1401]). Même au sein du socle, le */dev/random* n'est en pratique pas utilisé.

Les deux usages principaux de la génération d'aléa dans la couche utilisateur CLIP sont d'une part la génération de clés cryptographiques durables (qui restent utilisées après un redémarrage, et sont stockées sous une forme ou une autre sur un support de stockage, local ou amovible), et d'autre part la génération de clés de session, ainsi que des éléments de clés et *nonces* qui servent le cas échéant à négocier de telles clés. Quelques usages complémentaires, non liés aux clés cryptographiques, sont aussi listés à la fin de la présente section.

Génération de clés cryptographiques permanentes

Le générateur */dev/urandom* est utilisé, par lecture directe ou lecture directe suivie d'un retraitement, pour générer les clés permanentes suivantes :

- Clés symétriques de chiffrement USB (lecture directe) ([CLIP_DCS_15088]),
- Clés RSA de chiffrement et signature USB (lecture et vérification de primalité par *ssh-keygen*) ([CLIP_DCS_15088]),
- Clés symétriques utilisées pour chiffrer les clés RSA USB lors de leur export sur un support

amovible (lecture directe),

- Clés des partitions utilisateur (lecture directe, mais utilisation avec un retraitement par hachage) ([CLIP_1302]),
- Clés RSA pour l'authentification SSH des accès aux rôles ADMIN (CLIP et RM) et AUDIT (lecture et vérification de primalité par *ssh-keygen*) ([CLIP_1302]),
- Clés GPG éventuelles des utilisateurs (lecture et retraitement par *gpg*).

Clés de session

Les clés et éléments de clés temporaires suivants sont aussi générés par lecture de */dev/urandom* :

- Clé de chiffrement du *swap,* renouvelée à chaque démarrage (lecture directe, puis retraitement par hachage par *cryptsetup*) ([CLIP_1301]),
- Nonces (lecture directe) et éléments de clés (négociation CCSD, lecture indirecte à travers la fonction de lecture d'aléa fournie à CCSD) pour les négociations IKEv2 ([CLIP_1502]),
- Nonces et éléments de clés (négociation *Diffie-Hellman*) pour l'établissement des clés de session SSH (lecture directe) ([CLIP_1302]),
- Nonces et éléments de clés (négociation *Diffie-Hellman*) pour l'établissement des clés de session utilisées par les connexions SSL/TLS établies pour le téléchargement de mises à jour ou par les applications utilisateur (lecture directe).

Autres usages

Enfin, l'aléa fourni par */dev/urandom* sert aussi pour les mécanismes suivants :

- Génération d'un "canari" Propolice/SSP de 4 octets ([CLIP_1101]) lors de chaque appel *exec()*.
- Génération des sels utilisés pour le hachage des mots de passe utilisateur, aussi bien pour l'authentification que pour la production des clés permettant le déchiffrement des clés de partitions utilisateur ([CLIP_1302]).
- Génération des sels pour les hachages *openssl,* utilisés notamment pour la dérivation des clés de chiffrement des clés de partitions utilisateur ([CLIP_1302]), et pour la dérivation des mots de passe des clés RSA (USB et SSH).

3.3 Initialisation et sauvegarde de l'état

L'état du générateur d'aléa noyau est initialisé à chaque démarrage noyau par l'injection d'une "graine" sauvegardée avant l'arrêt précédent. Cette initialisation est gérée de la manière standard pour un poste Linux, par le script de démarrage *urandom* (cf. [CLIP_1301]) :

- Lors de l'arrêt du système, une graine est lue sur */dev/urandom*, et sauvegardée dans */var/run/random-seed* (sans protection particulière, à part les permissions discrétionnaires, qui en limitent l'accès à *root* - et le cloisonnement des cages qui en limite l'accès au seul socle CLIP).
- Au démarrage du système, le contenu de ce fichier *random-seed* est écrit sur */dev/urandom*, ce qui entraîne son injection dans le *pool* primaire du noyau.

- Immédiatement après cette injection, une nouvelle graine est lue sur */dev/urandom*, et sauvegardée dans */var/run/random-seed*, afin d'éviter de réutiliser la même graine au prochain démarrage, si le poste venait à être arrêté de manière non nominale (perte d'alimentation, etc).

La seule différence notable par rapport à la procédure standard des distributions Linux tient à la taille de cette graine, qui est de 2048 octets dans les systèmes CLIP, au lieu des 512 normalement utilisés, afin d'adapter cette initialisation aux *pools* d'entropie quatre fois plus grands des systèmes CLIP.

Annexe A Références

[CLIP_1001] *Documentation CLIP – 1001 - Périmètre fonctionnel CLIP*

[CLIP_1002] *Documentation CLIP – 1002 – Architecture de sécurité*

[CLIP_1003] *Documentation CLIP – 1003 – Paquetages CLIP*

[CLIP_1101] *Documentation CLIP – 1101 – Génération de paquetages CLIP*

[CLIP_1201] *Documentation CLIP – 1201 – Patch CLIP-LSM*

[CLIP_1203] *Documentation CLIP – 1203 – Patch Grsecurity*

[CLIP_1205] *Documentation CLIP – 1205 – CCSD en couche noyau*

[CLIP_1301] *Documentation CLIP – 1301 – Séquences de démarrage et d'arrêt*

[CLIP_1302] *Documentation CLIP – 1302 – Fonctions d'authentification locale*

[CLIP_1304] *Documentation CLIP – 1304 – Cages et socle CLIP*

[CLIP_1401] *Documentation CLIP – 1401 – CagesRM*

[CLIP_1502] *Documentation CLIP – 1502 – Racoon2*

[CLIP_DCS_15088] *Conception de l'étude de la problématique de stockage sur support amovible, CLIP-DC-15000-088-DCS*

[CCSD] CELAR, *Couche Cryptographique pour la Sécurité de Défense – Document d'Interface Client version 3.2*

[LRNG] Zvi Gutterman *et al.*, *Analysis of the Linux Random Number Generator,* IEEE Symposium on Security and Privacy, 2006