Rapport préliminaire – Audit d'OpenSSL

Date	5 février 2014
Rédigé par	Claire Smets, William Boisseleau, Pascal Edouard, Mathieu Latimier, Julien Legras
À l'attention de	Ayoub Otmani

Table des matières

Table des figures

Listings

Introduction

Ceci est l'introduction

Chapitre 1

Entropie

1.1 Définitions et contexte

1.1.1 Introduction

Cette partie explicite les notions d'entropie nécessaires pour la définition d'aléatoire de certains programmes, mais décrit aussi les sources qui de génération de bits aléatoires et les tests liés.

Trois axes principaux sont nécessaires à la mise en place d'un générateur cryptographique aléatoire de bits :

— Une source de bits aléatoires (source d'entropie)
— Un algorithme pour accumuler ces bits reçus et les faire suivre vers l'application en nécessitant.
— Une méthode appropriée pour combiner ces deux premiers composants

1.1.2 Estimation de l'entropie générée par la source

Il est tout d'abord important de vérifier que la source d'entropie choisie produit suffisamment d'entropie, à un taux égalant voire dépassant une borne fixée. Pour ce faire, il faut définir avec précision la quantité d'entropie générée par la source. Il est de plus important de considérer les différents comportements des composants de la source, afin d'éliminer les interactions qu'ils peut y avoir entre les composants. En effet, ceci peut provoquer une redondance dans la génération d'entropie si cela n'est pas considéré. Étant donné une source biaisée, l'entropie générée sera conditionnée et donc plus facilement prévisible/estimable.

La source d'entropie doit donc être minutieusement choisie, sans qu'aucune interaction et conditionnement ne soit possible.

1.1.3 Concept d'entropie

Définition.
Soit X est une V.A. discrète. On définit l'**entropie** de X comme suit :

$$H(X) = -\sum_{x} P(X = x) * log(P(X = x))$$

Le logarithme est dans notre cas de base 2. L'entropie se mesure en shannons ou en bits.

Définition.

On définit le **désordre** (ou incertitude) étant liée à cette expérience aléatoire. Si l'on considère l'ensemble fini des issues possibles d'une expérience $\{v_1, ..., v_n\}$, l'entropie de l'expérience vaudra :

$$H(\epsilon) = -\sum_x P(\{a_i\}) * log(P(\{a_i\}))$$

Propriété.

On constate que l'entropie est maximale lorsque X est équi-répartie. En effet, si l'on considère n éléments de X étant équi-répartie, on retrouve notre entropie de $H(X) = log(n)$.

Ainsi, on comprend qu'une variable aléatoire apporte en moyenne un maximum d'entropie lorsqu'elle peut prendre chaque valeur avec une équiprobabilité. D'un point de vue moins théorique, on considère que plus l'entropie sera grande, plus il sera difficile de prévoir la valeur que l'on observe.

Min-entropy.

La recommandation du NIST propose le calcul de *Min-entropy* pour mesurer au pire des cas l'entropie d'une observation.

Soit x_i un bruit de la source d'entropie. Soit $p(x_i)$ la probabilité d'obtenir x_i. On définit l'entropie au pire des cas telle que :

$$\text{Min-entropy} = -log_2(max(p(x_i))$$

La probabilité d'observer x_i sera donc au minimum $\frac{1}{2^{\text{Min-entropy}}}$.

1.1.4 Source d'entropie

Approche théorique.

La source d'entropie est composée de 3 éléments principaux :

- le **bruit source**, qui est la voûte de la sécurité du système. Ce bruit doit être non déterministe, il renvoie de façon aléatoire des bits grâce à des processus non déterministes. Le bruit ne vient pas nécessairement directement d'éléments binaires. Si ce bruit est externe, il est alors converti en données binaires. La taille des données binaires générées est fixée, de telle sorte que la sortie du bruit source soit déterminé dans un espace fixe.
- le **composant de conditionnement**, qui permet d'augmenter ou diminuer le taux d'entropie reçu. L'algorithme de conditionnement doit être un algorithme cryptographique approuvé.
- une **batterie de tests**, partie également intégrante du système. Des tests sont réalisés pour déterminer l'état de santé du générateur aléatoire, permettant de s'assurer que la source d'entropie fonctionne comme attendu. On considère 3 catégories de tests :
  - Les tests au démarrage sur tous les composants de la source
  - Les tests lancés de façon continue sur le bruit généré par la source
  - Les tests sur demande (qui peuvent prendre du temps)

  L'objectif principal de ces tests est d'être capable d'identifier rapidement des échecs de génération d'entropie, ceci avec une forte probabilité. Il est donc important de déterminer une bonne stratégie de détermination d'échec pour chacun de ces tests.

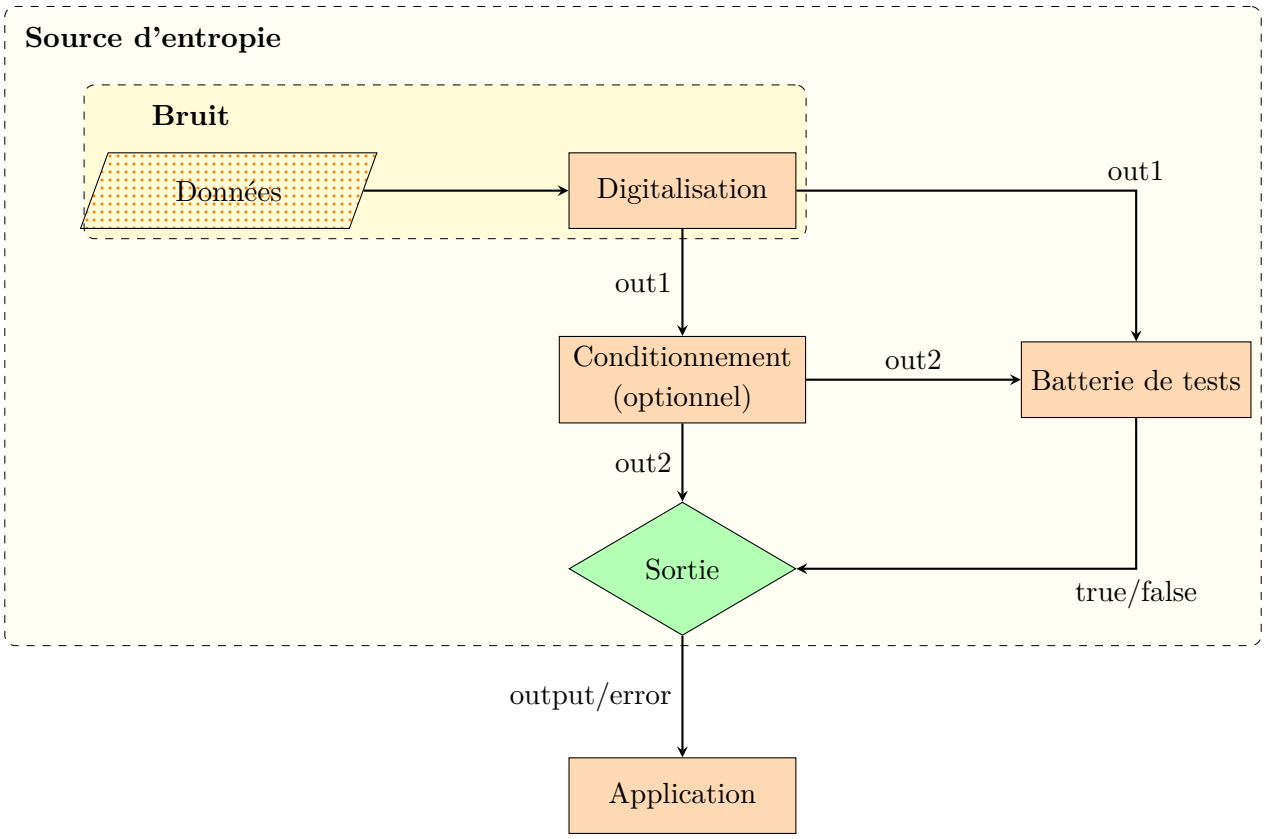

FIGURE 1.1 – Composants d'une source d'entropie. `out1` est une chaîne binaire de taille quelconque et `out2` est une chaîne binaire conditionnée de taille fixe.

Modèle conceptuel.

Suivant ces sections précédentes, on peut déterminer 3 interfaces conceptuelle :

- `getEntropy` qui retourne
  - `entropy_bitstring`, une chaîne de bits de l'entropie demandée
  - `assessed_entropy`, entier indiquant le nombre de bits d'entropie de `entropy_bitstring`
  - `status`, booléen renvoyant `true` si la requête est satisfaite, `false` sinon.

- `getNoise` qui prend en entrée :
  - `number_of_sample_requested`, entier indiquant le nombre d'éléments demandés en retour à la source de bruit

  et en sortie :
  - `noise_source_data`, la séquences d'éléments demandée, ayant la taille `number_of_sample_requested`.
  - `status`, booléen renvoyant `true` si la requête est satisfaite, `false` sinon.

- `HealthTest`, élément test de la batterie de tests, qui prend en entrée :
  - `type_of_test_requested`, chaine de bits déterminant le type de tests que l'on souhaite effectuer (peut différer suivant le type de source)

  et en sortie :
  - `pass-fail_flag`, booléen qui renvoie `true` si la source d'entropie a réussi le test, `faux` sinon.

1.1.5 Standards

1.1.5.1 RFC 4086

1.1.5.1.1 Sous Linux

Suivant la RFC 4086 intitulée *Randomness requirements for security* , Il existe plusieurs niveaux de récupération d'aléatoire sous linux :

— Point primaire (*Primary pool*) :
512o (128 mots de 4o) + ajout d'entropie
— Point secondaire (*Secondary pool*) :
128o pour générer le fichier `/dev/random`. Un autre point secondaire existe : `/dev/urandom`

L'entropie est récupérée par exemple lorsqu'un événement apparaît (telle qu'une interruption du disque dur), la date et l'heure de l'événement est récupéré et XORée dans le *pool*, puis est "mélangée" avec une primitive polynomiale possédant un degré de 128. Le *pool* devient ensuite une boucle où de nouvelles données sont XORées ("mélangées" encore par la primitive polynomiale) tout le long du *pool*.

A chaque appel qui rajoute de l'entropie dans le *pool*, celui-ci calcule une estimation de la probabilité d'une réelle entropie des données. Le pool contient alors l'accumulation des estimations de l'entropie totale contenue dans le pool.

Les sources d'entropie sont les suivantes :
— Interruption Clavier
— Interruption des complétions du disque
— Mouvements de la souris
Quand des octets aléatoires sont demandés, la *pool* est haché avec SHA-1 (20 octets). Si plus de 20 octets est demandé plus que 20 octets, le haché est mélangé dans la pool pour rehacher la pool ensuite etc. À chaque fois que l'on prend des octets dans la pool, l'entropie estimée est décrémentée. Pour assurer un niveau minimum d'entropie au démarrage, la *pool* est écrite dans un fichier à l'extinction de la machine.

`/dev/urandom` fonctionne selon le même principe sauf qu'il n'attend pas qu'il y ait assez d'entropie pour donner de l'aléatoire. Il convient pour une génération de clefs de session. Pour générer des clefs cryptographiques de longue durée, il est recommandé d'utiliser `/dev/random` pour assurer un niveau minimum d'entropie.

1.1.5.1.2 Sous Windows

Du coté de Microsoft, il est recommandé aux utilisateurs d'utiliser `CryptGenRandom`, qui est un appel système de génération d'un nombre pseudo-aléatoire. La génération est réalisée par une librairie cryptographique (*Cryptographic service provider library*). Celle-ci gère un pointeur vers un *buffer* en lui fournissant de l'entropie afin de générer un nombre pseudo aléatoire en retour avec en plus, le nombre d'octets d'aléatoires désirés.

```
BOOL WINAPI CryptGenRandom(
  _In_     HCRYPTPROV hProv,
  _In_     DWORD dwLen,
```

```
  _Inout_  BYTE *pbBuffer
);
```

Le service *provider* sauvegarde une variable d'état d'un sel pour chaque utilisateur. Lorsque `CryptGenRandom` est appelé, celui-ci est combiné avec un nombre aléatoire généré par la librairie en plus de différentes données systèmes mais aussi de l'utilisateur tels que l'ID du processus du *thread*, l'horloge système, l'heure système, l'état de la mémoire, l'espace de disque disponible du *cluster* et le haché du block d'environnement mémoire de l'utilisateur.

Le tout est envoyé à la fonction de hachage SHA-1 et le nombre en sortie est utilisé comme sel de clef RC4. Cette clef est enfin utilisée pour produire des données pseudo-aléatoires et mettre à jour la variable d'état du sel de l'utilisateur.

1.1.5.1.3 OpenBSD

Dans OpenBSD, on trouve des sources d'aléatoire supplémentaires par rapport à Linux. On trouve notamment `/dev/arandom` qui génère de l'aléatoire selon une version leakée de RC4 : `ARC4` (Alleged RC4). Pour rappel, RC4 était un projet commercial de RSA Security et un hacker anonyme a publié un code qui faisait la même chose, code légitime identifié par ARC4. De nos jours, il est fortement conseillé de ne plus utiliser RC4 car le flux de données aléatoires n'est pas vraiment aléatoire et il existe des attaques qui prédisent la sortie de l'algorithme (Attaque de Fluhrer, Mantin et Shamir) .

1.2 Audits

1.2.1 Audit 1 : X

1.2.1.1 Norme visée

1.2.1.2 Faille

1.2.1.3 Implémentation

Version OpenSSL.

Fonction.

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```c
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

```

```
13      for (i = 0; i < N; i++)
        {
15          puts("LaTeX is also great for programmers!");
        }
17
        return 0;
19  }
```

Listing 1.1 – codeAleatoire.c

Audit.

1.3 Recommandations générales

FIGURE 1.2 – Titre de Figure 1.1

Chapitre 2

Génération des clés

2.1 Définitions et contexte

2.2 Audits

2.2.1 Audit 1 : X

2.2.1.1 Norme visée

2.2.1.2 Faille

2.2.1.3 Implémentation

Version OpenSSL.

Fonction.

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

    for (i = 0; i < N; i++)
    {
        puts("LaTeX is also great for programmers!");
    }

    return 0;
}
```

||

Listing 2.1 – codeAleatoire.c

Audit.

2.3 Recommandations générales

FIGURE 2.1 – Titre de figure 2.1

Chapitre 3

Poignée de main

3.1 Définitions et contexte

3.2 Audits

3.2.1 Audit 1 : X

3.2.1.1 Norme visée

3.2.1.2 Faille

3.2.1.3 Implémentation

Version OpenSSL.

Fonction.

La fonction liée à cette norme est accessible sous le paquetage `bla/bla/bla`, dont les composantes principales sont listées en *listing* 3.1.

```
#include <stdio.h>
#define N 10
/* Block
 * comment */

int main()
{
    int i;

    // Line comment.
    puts("Hello world!");

    for (i = 0; i < N; i++)
    {
        puts("LaTeX is also great for programmers!");
    }

    return 0;
}
```

||

Listing 3.1 – codeAleatoire.c

Audit.

3.3 Recommandations générales

FIGURE 3.1 – Titre de figure 3.1

Conclusion

Ceci est la conclusion

Bibliographie

[1] 3rd, D. E., Schiller, J., and Crocker, S. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.

[2] Fluhrer, S., Mantin, I., and Shamir, A. Weaknesses in the key scheduling algorithm of rc4. In *PROCEEDINGS OF THE 4TH ANNUAL WORKSHOP ON SELECTED AREAS OF CRYPTOGRAPHY* (2001), pp. 1–24.