

# Audit d'OpenSSL

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

# Table des matières

Introduction . . . . .	9
<b>1 Entropie</b>	<b>10</b>
1.1 Définitions et contexte . . . . .	10
1.1.1 Introduction . . . . .	10
1.1.2 Estimation de l'entropie générée par la source . . . . .	10
1.1.3 Concept d'entropie . . . . .	10
1.1.4 Source d'entropie . . . . .	11
1.1.5 Standards . . . . .	13
1.1.5.1 RFC 4086 . . . . .	13
1.1.5.1.1 Sous Linux . . . . .	13
1.1.5.1.2 Sous Windows . . . . .	14
1.1.5.1.3 Sous OpenBSD et FreeBSD . . . . .	15
1.1.5.1.4 Autres systèmes . . . . .	15
1.1.5.2 FIPS 140 . . . . .	15
1.2 Une backdoor dans nos systèmes cryptographiques ? . . . . .	16
1.2.1 L'affaire Snowden et les documents top secrets de la NSA . . . . .	16
1.2.2 Le NIST et l'algorithme Dual EC DRBG . . . . .	17
1.2.3 ... Et OpenSSL ? . . . . .	18
1.3 Audits . . . . .	18
1.3.1 Audit 1.1 : Le cas Debian 4.0 et OpenSSL 0.9.8 . . . . .	18
1.3.1.1 Normes visées . . . . .	18
1.3.1.2 Description de la faille . . . . .	19
1.3.1.3 Tests . . . . .	20
1.3.1.4 Implémentation . . . . .	20
1.3.1.5 Conclusion . . . . .	21
1.3.2 Audit 1.2 : Le cas LinuxMintDebianEdition sous Android . . . . .	21
1.3.2.1 Normes visées . . . . .	21
1.3.2.2 Description de la faille . . . . .	21
1.3.2.3 Implémentation . . . . .	21
1.3.2.4 Conclusion . . . . .	22
1.3.3 Audit 1.3 : NetBSD 6.0 et OpenSSH . . . . .	22
1.3.3.1 Norme visée . . . . .	22
1.3.3.2 Description de la faille . . . . .	22
1.3.3.3 Implémentation . . . . .	22
1.3.3.4 Conclusion . . . . .	23

1.4	Recommandations sur la conception . . . . .	24
1.4.1	Modélisation et validation . . . . .	24
1.4.2	Source d'entropie absolue . . . . .	24
1.4.3	Bruit de la source . . . . .	24
1.4.4	Composant de conditionnement . . . . .	25
1.4.5	Batterie de tests . . . . .	25
1.4.5.1	Tests sur le bruit . . . . .	25
1.4.5.2	Tests sur le conditionnement . . . . .	26
1.5	Tests effectifs sur l'entropie fournie par les sources d'entropie . . . . .	26
1.5.1	Déterminer si les données sont IID . . . . .	26
1.5.1.1	Tests sur l'indépendance et la stabilité . . . . .	26
1.5.1.1.1	Score de compression . . . . .	29
1.5.1.1.2	Scores <i>Over/Under Runs</i> (2) . . . . .	30
1.5.1.1.3	Score excursion . . . . .	32
1.5.1.1.4	Scores Runs directionnels (3) . . . . .	32
1.5.1.1.5	Score de covariance . . . . .	33
1.5.1.1.6	Score de collision (3) . . . . .	34
1.5.1.2	Test statistique spécifique : $\chi^2$ . . . . .	35
1.5.2	Déterminer l'entropie minimale des sources IID . . . . .	36
<b>2</b>	<b>Génération des clefs</b>	<b>37</b>
2.1	Définitions et contexte . . . . .	37
2.1.1	Clefs cryptographiques . . . . .	37
2.1.1.1	Clefs d'algorithmes symétriques . . . . .	37
2.1.1.2	Clefs d'algorithmes asymétriques . . . . .	38
2.1.2	Méthode de génération et sécurité . . . . .	38
2.1.2.1	Génération de clefs quelconques . . . . .	38
2.1.2.2	Génération de paires de clefs asymétriques . . . . .	38
2.2	Audits . . . . .	39
2.2.1	Audit 2.1 : Génération d'entiers premiers, et de grands nombres . . . . .	39
2.2.1.1	Description de la librairie BIGNUMBER . . . . .	39
2.2.1.2	Normes visées . . . . .	41
2.2.1.3	Implémentation . . . . .	42
2.2.1.4	Conclusion . . . . .	43
2.2.2	Audit 2.2 : Le générateur de Diffie-Hellman . . . . .	43
2.2.2.1	Normes visées . . . . .	43
2.2.2.2	Description de la faille . . . . .	43
2.2.2.3	Implémentation . . . . .	44
2.2.2.4	Conclusion . . . . .	45
2.2.3	Audit 2.3 : Diffie-Hellman Ephémère en mode FIPS . . . . .	45
2.2.3.1	Normes visées . . . . .	45
2.2.3.2	Description de la faille . . . . .	45
2.2.3.3	Implémentation . . . . .	45
2.2.3.4	Conclusion . . . . .	46

<b>3</b>	<b>Chiffrement et Protocoles</b>	<b>47</b>
3.1	Définitions et contexte . . . . .	47
3.2	Audits . . . . .	47
3.2.1	Audit 3.1 : Les "Manger's attack" sur RSA-OAEP . . . . .	47
3.2.1.1	Normes visées . . . . .	47
3.2.1.2	Faible . . . . .	53
3.2.1.3	Implémentation . . . . .	53
3.2.1.4	Conclusion . . . . .	54
3.2.2	Audit 3.2 : Chiffrement SSLv3 ou TLS 1.0 en mode CBC . . . . .	54
3.2.2.1	Normes visées . . . . .	54
3.2.2.2	Description de la faille . . . . .	56
3.2.2.3	Tests . . . . .	56
3.2.2.4	Recommandations . . . . .	57
3.2.3	Audit 3.3 : Non-validation des certificats SSL . . . . .	57
3.2.3.1	Normes visées . . . . .	57
3.2.3.2	Description de la faille . . . . .	57
3.2.3.3	Difficultés du code OpenSSL . . . . .	58
3.2.3.4	Exemple : Trillian . . . . .	59
3.2.3.5	Conclusion . . . . .	59
<b>4</b>	<b>Signature et authentification</b>	<b>60</b>
4.1	Définitions et contexte . . . . .	60
4.2	Audits . . . . .	60
4.2.1	Audit 4.1 : Attaque par injection de fautes sur les certificats RSA . . . . .	60
4.2.1.1	Normes visées . . . . .	60
4.2.1.2	Description de la faille . . . . .	61
4.2.1.3	Implémentation . . . . .	62
4.2.1.4	Conclusion . . . . .	63
4.2.2	Audit 4.2 : Malformation des signatures DSA/ECDSA . . . . .	63
4.2.2.1	Normes visées . . . . .	63
4.2.2.2	Description de la faille . . . . .	64
4.2.2.3	Tests . . . . .	64
4.2.2.4	Implémentation . . . . .	64
4.2.2.5	Conclusion . . . . .	65
<b>5</b>	<b>Protocoles SSL/TLS</b>	<b>66</b>
5.1	Définitions et contexte . . . . .	66
5.2	Audits . . . . .	68
5.2.1	Audit 5.1 : SSL version 2 . . . . .	68
5.2.1.1	Spécifications . . . . .	68
5.2.1.2	Implémentation . . . . .	68
5.2.2	Audit 5.2 : SSL version 3 . . . . .	69
5.2.2.1	Spécifications . . . . .	69
5.2.2.2	Implémentation . . . . .	70
5.2.2.3	Faibles . . . . .	71
5.2.2.3.1	CVE-2013-4353 . . . . .	71

5.2.2.3.2	Attaque sur le padding CBC de Serge Vaudenay . . . . .	71
5.2.2.3.3	CVE-2011-4576 . . . . .	72
5.2.3	Audit 5.3 : TLS version 1 . . . . .	72
5.2.3.1	Spécifications . . . . .	72
5.2.3.2	Implémentation . . . . .	75
5.2.3.3	Faibles . . . . .	76
5.2.3.3.1	Attaque sur le padding CBC de Serge Vaudenay . . . . .	76
5.2.4	Audit 5.4 : TLS version 1.1 . . . . .	76
5.2.4.1	Spécifications . . . . .	76
5.2.4.2	Implémentation . . . . .	77
5.2.4.3	Faibles . . . . .	78
5.2.4.3.1	CVE-2012-2333 . . . . .	78
5.2.4.3.2	Lucky Thirteen CVE-2013-0169 . . . . .	78
5.2.5	Audit 5.5 : TLS version 1.2 . . . . .	79
5.2.5.1	Spécifications . . . . .	79
5.2.5.2	Implémentation . . . . .	79
5.2.5.3	Faibles . . . . .	80
5.2.5.3.1	CVE-2012-2333 . . . . .	80
5.2.5.3.2	CVE-2013-6449 . . . . .	80
<b>6</b>	<b>Observations et alternatives</b>	<b>81</b>
6.1	Code OpenSSL . . . . .	81
6.1.1	Documentation . . . . .	81
6.1.2	Lisibilité du code . . . . .	81
6.2	Construction du projet . . . . .	83
6.3	Alternatives . . . . .	83
	Conclusion . . . . .	84

# Table des figures

1.1	Composants d'une source d'entropie . . . . .	12
1.2	Structure de validation de données . . . . .	29
1.3	Calcul du score de compression . . . . .	30
2.1	Chiffrement symétrique . . . . .	37
2.2	Chiffrement asymétrique . . . . .	38
2.3	Structure d'un BIGNUM . . . . .	40
3.1	Réseau de Feistel . . . . .	48
3.2	Opération de chiffrement EME-OAEP . . . . .	50
3.3	Chiffrement CBC . . . . .	55
3.4	Déchiffrement CBC . . . . .	56
5.1	Schéma global d'une connexion SSL/TLS . . . . .	67

# Liste des tableaux

3.1	Représentations hexadécimales . . . . .	50
5.1	Algorithmes supportés SSLV2 . . . . .	68
5.2	Algorithmes supportés par OpenSSL SSLv2 . . . . .	68
5.3	Algorithmes supportés SSLv3 . . . . .	69
5.4	Algorithmes supportés par OpenSSL, SSLv3 . . . . .	70
5.5	Algorithmes supportés TLS1 . . . . .	73
5.6	Algorithmes supplémentaires AES - RFC3268 TLS1 . . . . .	74
5.7	Algorithmes supplémentaires Camellia - RFC3268 TLS1 . . . . .	74
5.8	Algorithmes supplémentaires SEED - RFC3268 TLS1 . . . . .	75
5.9	Gestion des extensions dans la poignée de main avec OpenSSL . . . . .	77
5.10	Algorithmes supplémentaires RFC 5054 TLSv1.1 . . . . .	77
5.11	Algorithmes supplémentaires - RFC 5289 TLSv1.2 . . . . .	79

# Listings

1.1	md_rand.c . . . . .	20
1.2	ssleay_rand_bytes.c . . . . .	21
1.3	subr_cprng.c(1) . . . . .	23
1.4	subr_cprng.c(2) . . . . .	23
2.1	ossl_typ.h . . . . .	39
2.2	bn.h . . . . .	39
2.3	bn_lib.c . . . . .	40
2.4	dh_gen.c . . . . .	44
2.5	dh_key.c . . . . .	45
3.1	rsa_oaep.c . . . . .	54
4.1	window_search.c . . . . .	62
4.2	fwe.c . . . . .	63
4.3	EVP_VerifyFinal.c . . . . .	64
5.1	patch-cve-2013-4353 . . . . .	71
5.2	patch-cve-2011-4576 . . . . .	72
5.3	constantes protocole TLS . . . . .	75
5.4	Constantes TLS 1.2 . . . . .	79
5.5	patch-cve-2013-6449 . . . . .	80
6.1	Exemple de mauvais code d'OpenSSL . . . . .	82
6.2	Exemple de mauvais test d'OpenSSL . . . . .	82
6.3	Exemple de mauvais test d'OpenSSL corrigé . . . . .	82
6.4	Exemple de mauvais #if 0 d'OpenSSL . . . . .	82



## Introduction

Ce document rapporte plusieurs études que nous avons pu regrouper, analyser, et restituer à propos des différentes implémentations d'OpenSSL, en particulier des failles importantes survenues au cours de ces dernières années.

En se basant sur les demandes du client, nous avons établi plusieurs points critiques dans l'implémentation et/ou l'utilisation d'OpenSSL. La méthodologie d'audit fut alors d'identifier les principales composantes sensibles par ordre d'importance, à savoir la gestion de l'entropie par le système, la génération des clés, le protocole usité pour le chiffrement, le protocole de signature, puis enfin les protocoles SSL/TLS implémentés par OpenSSL.

Pour chacune de ces parties, nous développons tout d'abord une description générale du contexte, suivi des standards et normes lui étant associées. Nous énumérons ensuite quelques failles répertoriées officiellement, ainsi que les correctifs leur ayant été respectivement proposés puis implémentés. Cela nous permet enfin d'auditer OpenSSL et de soumettre nos observations et recommandations.

# 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. Il décrit aussi en particulier les sources de génération de bits aléatoires et leurs 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'il 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 :

$$H(X) = - \sum_x P(X = x) * \log_2(P(X = x))$$

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  $\{v_1, \dots, v_n\}$ , l'entropie de l'expérience vaudra :

$$H(\epsilon) = - \sum_x P(\{a_i\}) * \log_2(P(\{a_i\}))$$

### 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  $H(X) = \log_2(n)$ .

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 trois 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 déterminée 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 trois 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.

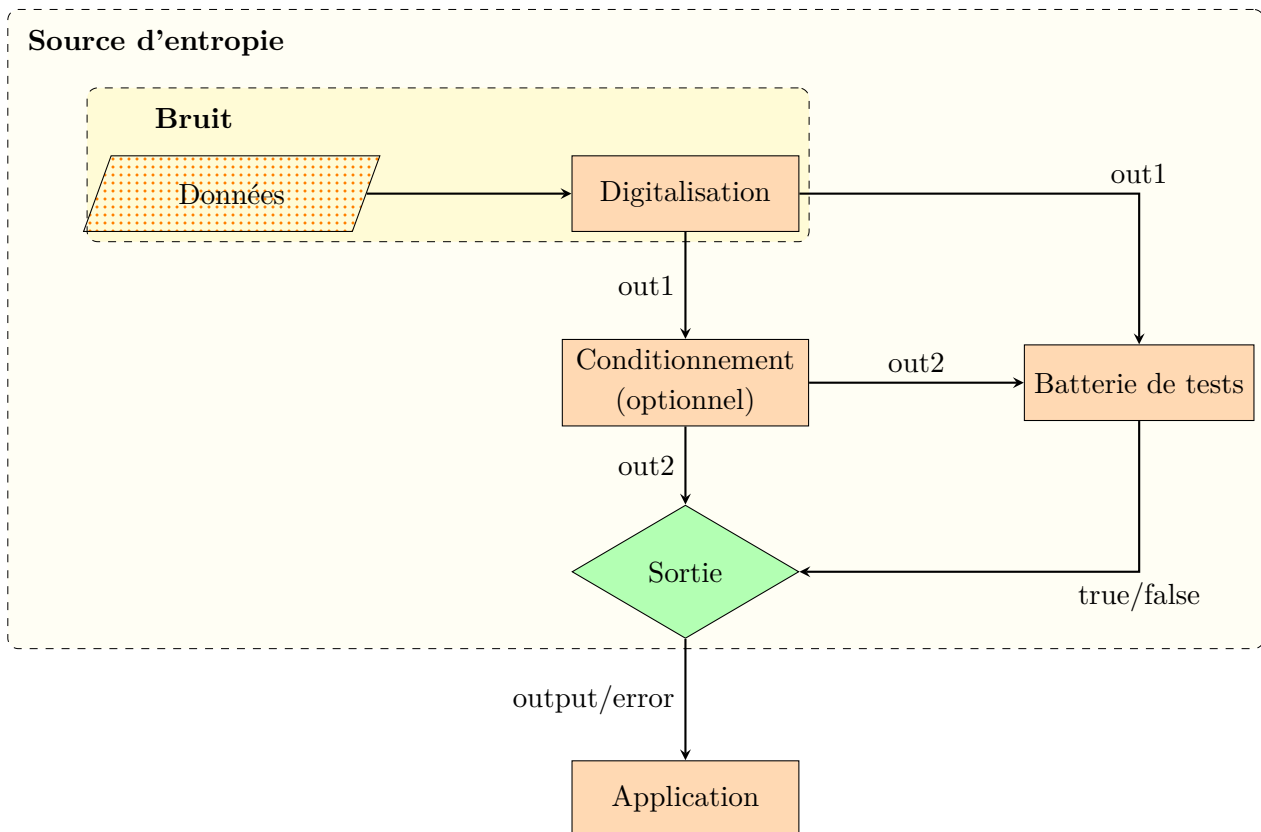


FIGURE 1.1 – Composants d'une source d'entropie. **out1** est une chaîne binaire de taille quelconque et **out2** est une chaîne binaire conditionnée de taille fixe.

### Modèle conceptuel.

Suivant ces sections précédentes, on peut déterminer trois interfaces conceptuelles :

- **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 séquence 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**, chaîne 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

Nos machines utilisent ce qu'on appelle des PRNG (Pseudo Random Number Generator), qui sont des algorithmes qui génèrent une séquence de nombres s'apparentant à de l'aléatoire. En réalité rien n'est aléatoire car tout est déterminé par des valeurs initiales (état du PRNG) et des contextes d'utilisation.

Un bon PRNG se doit d'avoir une très forte entropie (proche de un), afin d'éviter de délivrer de l'information. Comme l'entropie est fournie majoritairement (si ce n'est totalement) par l'OS, il est donc nécessaire de détailler les PRNG les plus utilisés (surtout par les systèmes Linux et BSD - qui sont ceux qui génèrent le plus de certificats SSL).

Nous nous basons sur la RFC 4086 [1] : *Randomness requirements for security* pour le choix des PRNG selon les différents systèmes, voir la section "Normes" pour plus d'informations.

##### 1.1.5.1.1 Sous Linux

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 sont récupérés et XORés dans le *pool*, puis sont "mélangés" 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*.

À 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 - heure et code d'interruption ;
- interruption des complétions du disque - heure de lecture ou écriture ;
- mouvement de la souris - heure et position.

Quand des octets aléatoires sont demandés, la *pool* est hachée avec SHA-1 (20 octets). Si plus de 20 octets sont demandés, le haché est mélangé dans la *pool* pour la rehacher 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 donc pour une génération de clefs de session. Pour générer des clefs cryptographiques demandant une grande entropie, il est recommandé d'utiliser `/dev/random` pour assurer un niveau minimum d'entropie.

`/dev/random` utilise une *pool* d'entropie de 4096 bits (512 octets) génère de l'aléatoire et s'arrête lorsqu'il n'y a plus assez d'entropie ; il attend que la *pool* se remplisse à nouveau.

En conclusion, `/dev/random` doit être utilisé pour une haute qualité d'entropie (i.e. haute sécurité de chiffrement, one-time pad), tandis que `/dev/urandom` doit être utilisé pour des applications non sensibles à des attaques cryptographiques (i.e. jeu en temps réel), car elle génère plus d'entropie que `/dev/random` sur un temps donné, mais s'arrêtera même s'il n'a pas récolté suffisamment d'entropie.

Par exemple, sur un serveur sans souris ni clavier, définir l'entropie avec `/dev/urandom` est très risqué. On recommande donc l'utilisation de `/dev/random` lors de l'audit OpenSSL sur les versions Linux.

Pour connaître l'entropie disponible, il suffit d'entrer la commande suivante :

```
cat /proc/sys/kernel/random/entropy_avail.
```

Désormais, la taille de la *pool* est hardcodée dans le noyau Linux (`/drivers/char/random.c:275`)

Linux offre également la possibilité de récupérer de l'aléatoire depuis un RNG matériel avec la fonction `get_random_bytes_arch`

Un patch est également disponible afin de générer de l'aléatoire avec un débit de 100kB/s [36]. L'entropie est récupérée par le *CPU timing jitter*.

#### 1.1.5.1.2 Sous Windows

Du côté de Microsoft, il est recommandé aux utilisateurs d'utiliser `CryptGenRandom`, [58] qui est un appel système de génération d'un nombre pseudo-aléatoire. La génération est réalisée par une bibliothèque 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 bibliothèque en plus de différentes

données systèmes mais aussi de l'utilisateur telles que :

- l'ID du processus ;
- l'ID du thread ;
- l'horloge système ;
- l'heure système ;
- l'état de la mémoire ;
- l'espace disque disponible du *cluster* ;
- le haché du bloc 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 Sous OpenBSD et FreeBSD

Il faut faire attention aux faux amis, car le `/dev/random` du FreeBSD n'est pas le même que celui de Linux. En fait, il est semblable au `/dev/urandom` de Linux, et est donc tout autant proscrit lors de notre audit.

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 globalement 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 en fait pas vraiment aléatoire et il existe des attaques qui prédisent la sortie de l'algorithme (Attaque de Fluhrer, Mantin et Shamir) .

Sur plusieurs de nos sources (plus anciennes), il est recommandé d'utiliser `/dev/arandom` pour sa rapidité (71 Mb/s) et sa bonne source d'entropie. Ce n'est plus vraiment le cas aujourd'hui.

#### 1.1.5.1.4 Autres systèmes

Nous avons également d'autres RNG comme `/dev/srandom`, `/dev/prandom` ou encore `/dev/wrandom` [34]. `/dev/srandom` est simple et lent, il n'est pas recommandé de l'utiliser.

Certains systèmes ne disposant pas de `/dev/*random`, peuvent utiliser l'EGD (Entropy Gathering Daemon) [57]. Il faut alors utiliser les fonctions OpenSSL `RAND_egd`, `RAND_egd_bytes` et `RAND_query_egd_bytes`. L'EGD est également utilisé par GPG, et peut être utilisé comme *seed*.

#### 1.1.5.2 FIPS 140

Le FIPS 140 (*Federal Information Processing Standards*) est un standard du gouvernement américain spécifique aux modules cryptographiques déployés par des éléments du gouvernement. Il inclut notamment des standards de tests qui permettent de décrire la qualité et/ou valider des générateurs d'entropie ou générateurs pseudo-aléatoires. La version la plus récente est le FIPS 140-2, qui décrit plusieurs niveaux de tests.

Les tests du FIPS 140-1 permettent de s'assurer que les sources d'entropies produisent suffisamment de "bonnes" données, pourvu que les sources d'entropies n'utilisent pas quelques opérations cryptographiques internes. Si une source d'entropie en utilise, alors elle réussira les tests avec quasi-certitude, même si la source d'entropie est faible. De ce fait, ces jeux de tests ne sont pas bons pour tester des générateurs de nombres pseudo-aléatoires cryptographiques, car ceux-ci passeront facilement les tests même si les générateurs d'entropie sont faibles. Par exemple, si l'on hache une suite d'entiers (par pas de 1), les tests seront tous validés bien qu'ils n'aient pas été tirés aléatoirement, ceci en vertu de la fonction de hachage. Pourtant, la donnée est hautement prédictible.

Les tests du FIPS 140-2 ne sont également pas très efficaces, et permettent seulement de détecter si un matériel commence à produire un motif répété. Ils consistent à comparer différentes sorties consécutives d'un générateur. Ainsi, si le "générateur aléatoire" consiste à produire une simple incrémentation de nombres, les tests passeront sans problème.

Il est donc conseillé d'utiliser les tests du FIPS 140-1 et 140-2 pour vérifier uniquement si la source d'entropie produit de bonnes données, au démarrage et périodiquement lorsque c'est possible.

Le FIPS 140-1 définit 4 tests statistiques à lancer sur 20000 bits consécutifs, tests au démarrage ou à la demande :

1. le **test Monobit**, où le nombre de bits à '1' sont comptés. Le test est considéré comme réussi si le nombre de bits à '1' est raisonnablement proche de 10 000 ;
2. le **test Poker**, pour lequel les données sont séparées en une suite consécutive de 4bits pour déterminer combien de fois les 16 configurations de 4 bits apparaissent. Les carrés du résultats sont sommés et permettent de définir si le test passe ou non ;
3. le **test runs**, que nous développerons dans les recommandations du NIST ;
4. le test **long runs**, qui effectue le test de runs sur 34 bits ou davantage.

Le FIPS 140-2 définit enfin des tests continus de sortie (*continuous output tests*). Les données de sortie sont découpées en blocs de 16 octets (ou davantage). Le premier bloc est stocké et comparé au second. S'ils sont identiques, alors le test est échoué. On passe à la paire suivante, le 2e bloc est comparé avec le 3e, etc.

## 1.2 Une backdoor dans nos systèmes cryptographiques ?

### 1.2.1 L'affaire Snowden et les documents top secrets de la NSA

Coup d'éclat en 2012, Edward Snowden, ancien-membre de la NSA et de la CIA, dévoile l'existence des *backdoors* ainsi qu'un lot d'informations conséquent sur la forte influence de la NSA sur le NIST et la RSA Security [32]. Il préleva ainsi plus de 1 700 000 documents à la NSA (d'après un officier de la NSA - 15 décembre 2013), dont 31 000 ultra-confidentiels [59]. Il délivra quelques documents à plusieurs journaux populaires tels que *"The Guardian"* et *"The New York Times"*.

Parmi les documents top secrets rendus publics, un en particulier nous intéresse [40]. Il concerne le contrôle de la NSA sur les systèmes de chiffrement actuels, nom de code BULLRUN, dont voici quelques points importants :



- "Insert vulnerabilities into commercial encryption systems, IT systems, networks, endpoint communications devices used by targets";
- "Influence policies, standards and specification for commercial public key technologies".

Bruce Schneier, un des plus grands cryptologues actuel, et fervent détracteur de la NSA, publie plusieurs articles concernant ce contrôle d'informations sur la question "La NSA a-t-elle réellement placé une *backdoor* au sein d'un nouveau système de chiffrement ?" [46]

Il évoque également le projet BULLRUN, montrant comment la NSA peut placer ses *backdoors*, comment elle peut les choisir, et propose plusieurs stratégies de défense [47] pour les vaincre, notamment :

- les vendeurs doivent rendre au minimum le code de chiffrement publique (spécifications concernant les protocoles inclus). Le reste peut être conservé secret ;
- la communauté des cryptologues doit pouvoir offrir une version compatible et indépendante du système de chiffrement, en *open-source* ou en vente auprès des entreprises privées (pour financer les universités par exemple) ;
- il ne doit y avoir aucun secret. Tout doit être entièrement transparent auprès des clients ;
- l'ensemble des PRNG doivent être rendus conformes avant publication et acceptation ;
- aucune fuite d'information n'est permise, surtout au niveau des protocoles de chiffrement. Ceci afin d'éviter la prédiction de clés privées.

En Septembre 2013, Matthew Green publie un article [21] sur ce vaste problème entre la NSA et la sécurité cryptographique, qui a été salué par plusieurs cryptologues dont B. Schneier.

Il précise cependant que ceci ne reste que des spéculations, mais qu'elles sont nécessaires afin de redoubler d'efforts dans la sécurité de nos communications.

### 1.2.2 Le NIST et l'algorithme Dual EC DRBG

Un rapport de Snowden, indique que la NSA a versé plus de 10 000 000\$ à la compagnie RSA Security [31] pour qu'elle utilise ce dernier, et discutable, algorithme comme générateur. On comprend donc mieux les suspicions autour d'un accord entre le NIST et la NSA pour la publication d'une recommandation de cet algorithme.

Les recommandations du NIST en matière de PRNG (qu'ils appellent plutôt DRNG - Determinist Random Number Generation), débute en 2006, la publication du dernier document sur les DRNG date de Janvier 2012 avec la SP800-90A [26].

Ce document présente quatre algorithmes de PRNG qui sont :

- le Hash\_DRBG basé sur des fonctions de hachage ;
- le HMAC\_DRBG basé également sur des fonctions de hachage ;
- le CTR\_DRBG basé sur du chiffrement par bloc ;
- le *Dual Elliptic Curve Deterministic RBG* (ou *Dual EC DRBG*) basé sur une théorie mathématique.

Les trois premiers sont conventionnels, acceptés par toute la communauté des cryptologues, et s'avèrent efficaces car ils génèrent "suffisamment" d'entropie. Le dernier est très différent des trois autres, dans le sens

où il utilise une fonction de chiffrement à sens unique. Certains cryptologues ont démontré que cet algorithme possède des failles (d'autres indiquent clairement que c'est une backdoor du NIST...). En effet, on peut accepter l'utilisation d'une fonction à sens unique, à condition que le secret utilisé ne soit pas conservé ailleurs (en d'autres termes qu'il soit détruit). Que faire si le NIST garde le secret des algorithmes permettant d'affaiblir considérablement le *Dual EC DRBG*, et rendre l'aléatoire prévisible pour qui s'en donne les moyens ?

En recoupant plusieurs sources, le doute augmente considérablement. En 2006, Berry Schoenmakers et Andrey Sidorenko établissent une cryptanalyse du *DUAL EC DRBG* [48]. En 2007, Dan Shumow et Niels Ferguson furent les premiers à dénoncer le NIST d'avoir placé une *backdoor* délibérément dans cet algorithme [50].

Avant septembre 2013, tout cela n'était que suspicion, mais depuis le NIST a publié un bulletin de nouvelles recommandations pour les DRNG [39], et indique (surtout grâce à un forcing de la communauté cryptologue) que le *Dual EC DRBG* ne doit plus être utilisé pour les raisons suivantes :

- la provenance des points par défaut de la courbe elliptique utilisée n'est pas clairement détaillée ;
- la génération de ces courbes n'est pas digne de confiance.

***"Recommending against the use of SP 800-90A Dual Elliptic Curve Deterministic Random Bit Generation : NIST strongly recommends that, pending the resolution of the security concerns and there - issuance of SP 800-90A, the Dual\_EC\_DRBG, as specified in the January 2012 version of SP 800-90A, no longer be used".***

### 1.2.3 ... Et OpenSSL ?

À présent, il faut rechercher l'utilisation de cet algorithme dans les classes d'OpenSSL, la nouvelle recommandation du NIST faisant foi. Le directeur technique d'OpenSSL Steve Marques a rapporté le 19 décembre 2013 qu' "un bug inusuel [avait] été détecté sur une situation inusuelle". Paul Ducklin apporte une bonne synthèse sur le site NakedSecurity [17], le titre est clair : "Le bug d'OpenSSL nous sauve de l'espionnage".

L'implantation du *DUAL\_EC\_DRBG* dans OpenSSL contient une faille, celle-ci causant un arrêt brutal ou un blocage de programme. Le bug a toujours été là, et il vient seulement d'être détecté. Heureusement, personne n'a pu utiliser cet algorithme, celui-ci est resté dans les phases de tests, les passant tout de même avec succès.

## 1.3 Audits

### 1.3.1 Audit 1.1 : Le cas Debian 4.0 et OpenSSL 0.9.8

#### 1.3.1.1 Normes visées

Les normes visées furent la RFC 4086 [1] et le FIPS 140-2 [55]. La première définit le bon fonctionnement d'un RNG, tandis que la deuxième apporte plutôt des algorithmes réputés sûrs pour la génération d'aléatoire.

### 1.3.1.2 Description de la faille

Le 13 mai 2008, Luciano Bello a découvert une faille critique du paquet d'OpenSSL sur les systèmes Debian [2]. Un mainteneur Debian souhaitant corriger quelques bugs aurait malencontreusement supprimé une grosse source d'entropie lors de la génération des clefs. Il ne restait plus que le PID comme source d'entropie. Comme celui-ci ne pouvait dépasser 32 768 (qui le PID maximal par défaut atteignable), l'espace des clefs a été restreint à 264 148 clefs distinctes.

Le 14 mai 2008, Steinar H. Gunderson démontre qu'en connaissant le secret  $k$  d'une signature, on peut retrouver la clef privée d'un certificat immédiatement [22]. Ce secret  $k$  étant généré avec un PRNG prévisible, on peut stocker deux signatures utilisant le même  $k$ , où le prédire directement.

Une signature DSA consiste en deux nombres  $r$  et  $s$  tels que :

$$r = (g^k[p])[q]$$
$$s = (k^{-1} * (H(m) + x * r))[q]$$

La clé publique =  $(p, q, g)$ , le message en clair =  $m$ , et  $H(m)$  est le *fingerprint* de  $m$  qui est également connu.

– Attaque n° 1 : En connaissant  $k$

$$s * k[q] = (H(m) + x * r)[q]$$
$$\iff (s * k - H(m))[q] = x * r[q]$$
$$\iff ((s * k - H(m)) * r^{-1})[q] = x$$
$$\iff (s * k - H(m)) * r^{-1} = x$$

– Attaque n° 2 : Deux messages possèdent le même  $k$

$$s_1 = (k^{-1}(H(m_1) + x * r))[q]$$
$$s_2 = (k^{-1}(H(m_2) + x * r))[q]$$
$$\iff s_1 - s_2 = (k^{-1}(H(m_1) - H(m_2)))[q]$$
$$\iff (s_1 - s_2) * (H(m_1) - H(m_2))^{-1} = k^{-1}[q]$$
$$\iff \text{On connaît } k \implies \text{Attaque 1}$$

Pour savoir si une clef SSL, SSH, DNSSEC ou OpenVPN est affectée, plusieurs détecteurs de données [12] [13] de clés faibles sont fournies par l'équipe Security de Debian, en même temps que l'avertissement de sécurité [11].

### 1.3.1.3 Tests

Nous avons décidé de tester le nombre de certificats vulnérables causés par le bug OpenSSL de Debian (qui reste le plus populaire), et connaissant la *blacklist* des clefs privées. Les résultats nous montrent que sur 500 000 certificats récupérés, au moins<sup>1</sup> 769 sont vulnérables.

Vous pouvez trouver nos scripts parcourant un fichier contenant un certificat sur chaque ligne, ou un dossier contenant des certificats sous forme de fichiers PEM et nos résultats, dans le dossier consacré à l'audit des clefs cryptographiques.

Le format de nos résultats est :

**COMPROMISED :** <haché\_du\_certificat> <nom\_fichier\_corrompu (sous forme d'adresse IP)>

Évidemment, nous ne mettons pas ces résultats sur le net puisqu'ils indiquent très clairement les adresses IP contenant le certificat friable, et sa clef privée (que l'on peut facilement retrouver parmi la courte blacklist). Pour information, parmi les entreprises vulnérables nous trouvons les géants IBM et CISCO.

### 1.3.1.4 Implémentation

#### Configuration utilisée.

Première version OpenSSL vulnérable : 0.9.8c-1.

Première distribution stable où la vulnérabilité a été corrigé OpenSSL 0.9.8c-4etch3 pour Debian/Etch.

Première distribution de test où la vulnérabilité a été corrigé OpenSSL 0.9.8g-9 pour Debian/Lenny.

Première distribution instable où la vulnérabilité a été corrigé OpenSSL 0.9.8g-9 pour Debian/Sid.

#### Fonction.

La fonction liée à cette norme est accessible sous le paquetage `openssl/crypto/rand/md_rand.c` (cf. Listing 1.1).

```
1 | #ifndef PURIFY
2 | /*
3 |  * Don't add uninitialised data.
4 |  * MD_Update(&m, buf, j); /* purify complains */
5 | */
6 | #endif
7 |
```

Listing 1.1 – md\_rand.c

Analysons plus en détail cette faille. La conséquence est le blocage de la graine (seed) que l'on passe ensuite au PRNG. Cette ligne a été commentée par erreur en voulant corriger un avertissement soulevé par le compilateur Valgrind sur une valeur non initialisée.

---

1. Le logiciel ne prend pas en compte les clés  $\leq$  à 512 bits et celles  $\geq$  à 4096 bits, et ne prend en compte que les certificats RSA

### 1.3.1.5 Conclusion

Tout d'abord, il est fortement recommandé de mettre à jour sa version OpenSSL vers une version stable où le bug a été corrigé. Puis de générer de nouvelles clefs de chiffrement, et de révoquer les certificats corrompus. Il reste malheureusement beaucoup de certificats (+ de 700) touchés par cette faille, la plupart étant valide jusqu'en 2020-2030.

## 1.3.2 Audit 1.2 : Le cas LinuxMintDebianEdition sous Android

### 1.3.2.1 Normes visées

Les normes sont les mêmes que l'Audit 1.1.

### 1.3.2.2 Description de la faille

Récemment, en août 2013 précisément, un patch de sécurité pour les systèmes Android utilisant la version LinuxMintDebianEdition/d' OpenSSL, dévoile une réparation du générateur de nombres pseudo-aléatoires (PRNG) qui ne donnait pas suffisamment d'entropie [27] [33].

Le patch indique que le PRNG de cette version d'OpenSSL utilise dorénavant une combinaison de données plus ou moins prévisibles associées à l'entropie générées par `/dev/urandom`. Mais sachant que le PRNG d'OpenSSL utilise lui-même `/dev/urandom`, on a du mal à comprendre pourquoi en rajouter davantage.

Eric Wong et Martin Boßlet apportent la solution sur leur site [7]. Ils montrent que l'erreur provient d'un bug "à la Debian", une simple ligne diffère de la version officielle d'OpenSSL (utilisant `SecureRandom`) à celle de OpenSSL : `:Random` se situant dans la fonction `ssleay_rand_bytes`. Toutefois, la conséquence n'est pas aussi lourde que celle de Debian, tout d'abord parce que le système Android est rarement utilisé pour du chiffrement de données sensible, et une attaque par prédiction bien que plus rapide qu'une attaque par brute-force reste infaisable. Mais l'erreur est quand même là.

### 1.3.2.3 Implémentation

#### Configuration utilisée.

Système Android utilisant la version 1.0.1e d'OpenSSL au niveau de la classe `SecureRandom`, avec la bibliothèque Java Bridge (fichier JAR) accédant au PRNG d'OpenSSL, la bibliothèque `core/java/com/android/internal/os/ZygoteInit.java` pour la parallélisation des processus (fork).

OS utilisés Linux Mint Debian Edition (LMDE) et Fedora. Le bug concerne l'OS LMDE avec la version OpenSSL 1.0.1e.

#### Fonction.

La fonction d'OpenSSL mise en cause se situe également au niveau de `openssl/crypto/rand/md_rand.c`, dans la fonction `ssleay_rand_bytes(unsigned char *buf, int num, int pseudo)` (cf. *Listing 1.2*).

```
2 | #ifndef PURIFY /* purify complains */  
3 | #if 0  
4 | /* The following line uses the supplied buffer as a small  
   * source of entropy: since this buffer is often uninitialised
```

```
6      * it may cause programs such as purify or valgrind to
8      * complain. So for those builds it is not used: the removal
10     * of such a small source of entropy has negligible impact on
12     * security.
    */
    MD_Update(&m,buf,j);
#endif
#endif
```

Listing 1.2 – ssleay\_rand\_bytes.c

On retrouve le même bug que Debian. Le fameux patch supprimant l'entropie d'OpenSSL... 7 ans auparavant, et corrigé en 2008. Alors que l'équipe de développement d'OpenSSL déclare que l'impact est faible voire inexistant sur la sécurité, les systèmes Android, dérogent à la règle. La raison est que la mémoire n'est pas initialisée sur la version LMDE/OpenSSL 1.0.1e, comparé à Fedora/OpenSSL 1.0.1e. La seule source d'entropie n'est que la valeur du PID du processus courant, très limité.

#### 1.3.2.4 Conclusion

Il est alors recommandé à court terme d'ajouter plus d'entropie à notre PRNG sous Android, soit manuellement comme le script Ruby d'Eric Wong [7], ou avec les outils cités plus haut dans ce document. Sinon, le patch d'OpenSSL [27] répare également l'erreur en rajoutant plus d'aléatoire, bien qu'une recommandation officielle serait la bienvenue, ainsi qu'une meilleure documentation.

### 1.3.3 Audit 1.3 : NetBSD 6.0 et OpenSSH

#### 1.3.3.1 Norme visée

Le noyau du système NetBSD s'appuie sur le RNG CTR du NIST, que l'on peut trouver dans le NIST-SP-800-90A [26] pour définir sa *pool* d'entropie.

#### 1.3.3.2 Description de la faille

Autre faille du même genre sur les systèmes NetBSD 6.0 concernant OpenSSH/OpenSSL et datant de mars 2013 [8] [41]. L'erreur provient d'une insuffisance d'entropie dans le PRNG, qui ne tient que sur 32 ou 64 bits (la taille d'un `sizeof(int)`). Un attaquant peut brute-forcer une clef générée par ce PRNG. Il est probable que les dégâts soient bien moins étendus que lors de l'affaire OpenSSL Debian car les systèmes NetBSD 6.0 sont moins fréquents.

#### 1.3.3.3 Implémentation

##### Configuration utilisée.

La vulnérabilité concerne les versions NetBSD 6.0 et NetBSD 6.0.1. Elle est réparée sur la version NetBSD 6.0.2 et NetBSD 1.0 [38]. Une parenthèse mal placée dans le code du fichier `/src/sys/kern/subr_cprng.c` du système rend prévisible l'entropie fournie pour la génération des clefs.

## Fonctions.

La fonction vulnérable (cf. *Listing 1.3*) est accessible via le chemin `/src/sys/kern/subr_cprng.c`.

```
1 |         if (c->flags & CPRNG_INIT_ANY) {  
2 |             #ifdef DEBUG  
3 |                 printf("cprng %s: WARNING insufficient "  
4 |                     "entropy at creation.\n", name);  
5 |             #endif  
6 |             rnd_extract_data(key + r, sizeof(key - r), RND_EXTRACT_ANY);  
7 |         } else {  
8 |             hard++;  
9 |         }  
10 |     }
```

Listing 1.3 – subr\_cprng.c(1)

Pour la partie concernant la ligne 183 : `rnd_extract_data (key + r, sizeof(key - r), RND_EXTRACT_ANY);`. Le deuxième paramètre devrait être `sizeof(key) - r`. Dans la version 1.15, cet appel a été corrigé, et on utilise une nouvelle fonction nommée `cprng_entropy_try` (cf. *Listing 1.4*) [38] à la place de `rnd_extract_data`.

```
1 | static size_t  
2 | cprng_entropy_try(uint8_t *key, size_t keylen, int hard)  
3 | {  
4 |     int r;  
5 |     r = rnd_extract_data(key, keylen, RND_EXTRACT_GOOD);  
6 |     if (r != keylen && !hard) {  
7 |         rnd_extract_data(key + r, keylen - r, RND_EXTRACT_ANY);  
8 |     }  
9 |     return r;  
10 | }  
11 |
```

Listing 1.4 – subr\_cprng.c(2)

Cette fonction permet de réduire le nombre d'appels de la précédente fonction `rnd_extract_data`. Elle prend en argument le niveau d'entropie à atteindre "hard" ou "soft" afin de donner respectivement de "bons" bits ou "suffisamment" de bits.

### 1.3.3.4 Conclusion

Il est recommandé de passer à une version NetBSD supérieure à 5.1, et de générer de nouvelles données de chiffrement comme les clés SSH ayant été générées avec ce noyau. Il faut également faire attention au patch de sécurité de janvier 2013 qui en tentant de régler le problème a généré une autre erreur produisant le même effet. L'erreur ne provient pas directement du code d'OpenSSL. Mais ici, OpenSSL se contente de récupérer l'entropie fournie sans aucune vérification (fonction `RAND_bytes()`).

Il est également recommandé d'utiliser `/dev/random` plutôt que `/dev/urandom` pour avoir une bonne entropie. On souligne quand même le fait que OpenSSL ne contrôle pas son entropie, si l'entropie du système est quasi-nulle les clefs sont tout de même générées. Un avertissement auprès de l'utilisateur serait un minimum.

## 1.4 Recommandations sur la conception

Le NIST propose des recommandations à plusieurs niveaux concernant la conception de la source d'entropie. Nous n'en reporterons ici que les recommandations génériques. Les éléments ci-dessous sont basés sur le rapport du NIST SP800-90B.

### 1.4.1 Modélisation et validation

La source d'entropie doit suivre les exigences suivantes :

1. le développeur doit documenter complètement la modélisation de la source d'entropie, incluant toutes les interactions entre les composants. De ce fait, la documentation doit pouvoir justifier en quoi la source d'entropie est de confiance ;
2. il doit définir de façon précise les limites conceptuelles de sécurité de la source d'entropie, qui doivent elles-mêmes être équivalentes à un module cryptographique délimitant un périmètre de sécurité (cf. *Cryptographic module boundary* du FIPS 140) ;
3. le développeur doit définir le champ des conditions optimales de fonctionnement du générateur d'entropie ;
4. la source d'entropie doit être possiblement validée suivant les recommandations du FIPS 140, ainsi que les tests s'y référant ;
5. le comportement du bruit de la source doit être documenté, validant en quoi le taux d'entropie ne fluctue pas lors d'une utilisation normale ;
6. dès lors qu'un test de validation n'est pas réussi, la source d'entropie doit immédiatement cesser d'envoyer des données de sortie, et doit notifier à l'application l'erreur rencontrée.

Enfin, une recommandation optionnelle :

7. la source d'entropie doit contenir différents types de bruit pour améliorer le comportement global de la source, et prévenir des tentatives de contrôle externe. Chaque bruit doit vérifier les spécifications du 3.1.

### 1.4.2 Source d'entropie absolue

Certains générateurs de bits aléatoires demandent une source d'entropie absolue, à savoir qu'elle approxime une sortie qui est uniformément distribuée et indépendante des autres sorties.

1. la chaîne de bits générée doit fournir au moins  $(1 - \epsilon)n$  bits d'entropie, où :
  - $n$  est la taille de chaque sortie ;
  - $\epsilon$  tel que :  $0 \leq \epsilon \leq 2^{-64}$ .

### 1.4.3 Bruit de la source

Le bruit fourni par la source doit suivre les recommandations suivantes :

1. le bruit doit avoir un comportement probabiliste et ne doit en aucun cas être définissable par quelconque algorithme ou règle ;



2. le développeur doit pouvoir documenter l'opération sur le fonctionnement du bruit de la source, en montrant en quoi le choix de ce bruit fournit une sortie d'entropie acceptable. Ceci comprend le référencement d'articles de recherche et autre littérature pertinente ;
3. le bruit de la source doit pouvoir être testable, de telle sorte que l'on puisse s'assurer qu'il effectue l'opération attendue. Il doit donc être possible de récupérer des données sur la source de bruit sur lesquelles on puisse lancer la batterie de tests. La récupération des données sur la source de bruit ne doit en aucun cas altérer le comportement du bruit, ou de la sortie ;
4. toute défaillance du bruit doit être rapidement détectable. Les méthodes de détection doivent être documentées ;
5. la documentation de la source du bruit doit également décrire sous quelles conditions le bruit est connu pour mal fonctionner. Ceci consiste ainsi à répertorier les environnements dans lesquels la source peut fonctionner correctement ;
6. la source de bruit doit être protégée au maximum contre toute attaque/tentative de conditionnement ou simple connaissance de fonctionnement de la part d'un adversaire.

#### 1.4.4 Composant de conditionnement

Le composant de conditionnement doit suivre les recommandations suivantes :

1. le développeur doit documenter si la source d'entropie nécessite ou non un conditionnement ;
2. la méthode de conditionnement doit être décrite et argumentée. Elle doit en effet expliciter en quoi elle permet de diminuer l'alignement d'une source de bruit, ou en quoi la sortie créée correspond à l'entropie attendue ;
3. la méthode de conditionnement doit pouvoir être validée par des tests ;
4. le développeur doit pouvoir estimer l'alignement en sortie de conditionnement ;
5. le développeur doit prévoir une documentation expliquant le comportement de la méthode de conditionnement en cas de variation de comportement de la part de la source de bruit.

#### 1.4.5 Batterie de tests

Globalement, les tests doivent suivre les recommandations suivantes :

1. les tests doivent être effectués au démarrage puis de façon continue pour s'assurer que les composants du générateur d'entropie fonctionnent correctement ;
2. tous les tests doivent être documentés, en particulier sur les conditions sous lesquels ils doivent être exécutés, les résultats attendus pour chacun de ceux-ci, et une explication rationnelle indiquant en quoi chaque test est approprié pour la détection de dysfonctionnement de la part de la source d'entropie.

##### 1.4.5.1 Tests sur le bruit

Les tests sur le bruit sont pour la plupart du temps dépendants de la technologie utilisée. Dans la majorité des cas, il faut tester via des procédures traditionnelles de tests (type test monobit, test du  $\chi^2$ , et tests d'exécution) si celui-ci est bien non biaisé et produit des données indépendantes.

1. au minimum, des tests continus doivent être implémentés, et ceci de façon indépendante. Le développeur doit de plus documenter toutes les sources de défaillance ;
2. les tests sont implémentés sur les données binaires récupérées via le bruit source ;
3. la source de bruit doit être testée dans son ensemble (suivant la variation du bruit) ;
4. le bruit généré durant la démarrage ayant passé avec succès les tests de démarrage peut être utilisé pour produire de l'entropie ;
5. lorsqu'un test échoue, le générateur doit en être notifié.

Optionnel :

6. une étude peut être requise sur les bords du bruit généré, dans les cas où le comportement du générateur est altéré.

#### 1.4.5.2 Tests sur le conditionnement

Le rôle du conditionnement est de réduire l'alignement présent chez certaines sources d'entropie afin de s'assurer que l'entropie est construite à un taux acceptable. Les recommandations sont les suivantes :

1. les composants de conditionnements doivent être testés dès le démarrage, afin d'être certains que le générateur fonctionne comme prévu ;
2. le développeur doit documenter les tests implémentés et inclure les conditions d'échecs pour chacun d'entre eux.

### 1.5 Tests effectifs sur l'entropie fournie par les sources d'entropie

#### 1.5.1 Déterminer si les données sont IID

##### Définition.

On dit que des variables sont **IID** (indépendantes et identiquement distribuées) si elles suivent toutes la même loi de probabilité et si elles sont mutuellement indépendantes.

Les tests suivants ont été conçus pour montrer si les données en sortie du bruit et/ou en sortie du conditionnement sont bien IID, ceci en effectuant des tests sur la distribution des données. Le but est de vérifier l'hypothèse  $H_0$ . Pour la vérifier, nous allons considérer ici deux types de tests :

- des tests de répartition aléatoire sur l'ensemble ;
- des tests statistiques.

Si un des tests est passé, alors on passe au suivant. La défaillance d'un seul des tests impliquera le caractère non IID des données.

##### 1.5.1.1 Tests sur l'indépendance et la stabilité

Nous travaillons sur le test statistique bilatéral suivant :

- l'hypothèse  $H_0$  : "Les données sont IID" ;
- l'hypothèse  $H_1$  : "Les données ne sont pas IID".

Suivant un jeu de données ( $|donnees| = n$ ), on divise celui-ci en dix sous-ensembles de même taille ( $\frac{N}{10}$ ). On effectue enfin nos tests sur chacune de ces données. La stratégie de test est effectuée comme suit :

1. on calcule tout d'abord différents scores statistiques parmi les suivants :
  - score de compression, un score par sous-ensemble de données ;
  - score *Over/Under Runs*, deux scores par sous-ensemble ;
  - score excursion, un score par sous-ensemble ;
  - score *Runs* directionnel, trois scores par sous-ensemble ;
  - score de covariance, un score par sous-ensemble ;
  - score de collision, trois scores par sous-ensemble.
2. pour chaque calcul de score :
  - (a) on stocke pour chaque type de score les résultats dans un vecteur de  $J$  scores ;
  - (b) on répète les étapes suivantes 1000 fois :
    - i. on permute les sous-ensembles précédents en utilisant un générateur pseudo-aléatoire, suivant l'algorithme de permutation *Fisher-Yates shuffle* ;
    - ii. on calcule les nouveaux scores suivant cette nouvelle organisation de sous-ensembles ;
    - iii. on récupère ce vecteur de  $J$  scores.
  - (c) on classe les scores de 2a) en le comparant avec tous les scores liés permutés. Par exemple, si le score des données originelles est plus grand que tous ceux des permutations, alors ce premier a le score de 1000, et les répertoires permutés ont un score inférieur à 1000. Il est possible que les données permutées aient le même score. Lorsque le score originel est le même que certains de scores des données permutées, on considère le score en récupérant celui le plus proche de la médiane.

Plus généralement, étant donné :

- un score  $S$  ;
- une liste de scores des données permutées  $L$ .

On détermine le **rang de  $S$**  tel que :

$$Rang(S) = \begin{cases} \max(j) & \text{tel que } L[j] \leq S, & \text{si } L[500] > S & (1) \\ 500 & & \text{si } L[500] = S & (2) \\ \min(j) & \text{tel que } L[j] \geq S, & \text{si } L[500] < S & (3) \end{cases}$$

i. exemple cas (1) :

- soit  $S = 20$  et  $L[500] = 22$ . On est dans le cas (1). On va chercher le  $\max(j)$  tel que  $L[j] \leq S$  ;
- on a  $L[299] = 19$ ,  $L[300] = 20$ ,  $L[301] = 20$ ,  $L[302] = 22$  ;
- on retourne au score 301, ainsi  $Rang(S) = 301$ .

ii. exemple cas (3) :

- A. soit  $S = 20$  et  $L[500] = 18$ . On est dans le cas (3). On va chercher le  $\max(j)$  tel que  $L[j] \geq S$  ;
  - B. on a  $L[599] = 19$ ,  $L[600] = 20$ ,  $L[601] = 20$  ;
  - C. on retourne au score 600, ainsi  $Rang(S) = 600$ .
3. Le rang alors obtenu est considéré comme une  $p$ -value pour un test bilatéral. Pour rappel, une  $p$ -value est la probabilité d'obtenir la même valeur de test si l'hypothèse nulle est vraie. Ainsi, on compte pour chacun des six tests un ensemble de  $10 * J$   $p$ -values.
4. on gère enfin les  $p$ -values de la façon suivante :
- (a) Tous les rangs compris entre 50 et 950 sont relevés. Ces événements ont une probabilité d'arriver de 10% au total ;
  - (b) si le nombre de rangs relevés est supérieur ou égal à 8, on considère le test comme échoué.
    - i. si le test est effectué sur la source du bruit, alors celle-ci n'est pas considérée comme IID ;
    - ii. si le test est effectué sur le conditionnement, alors la source d'entropie n'est pas validée.

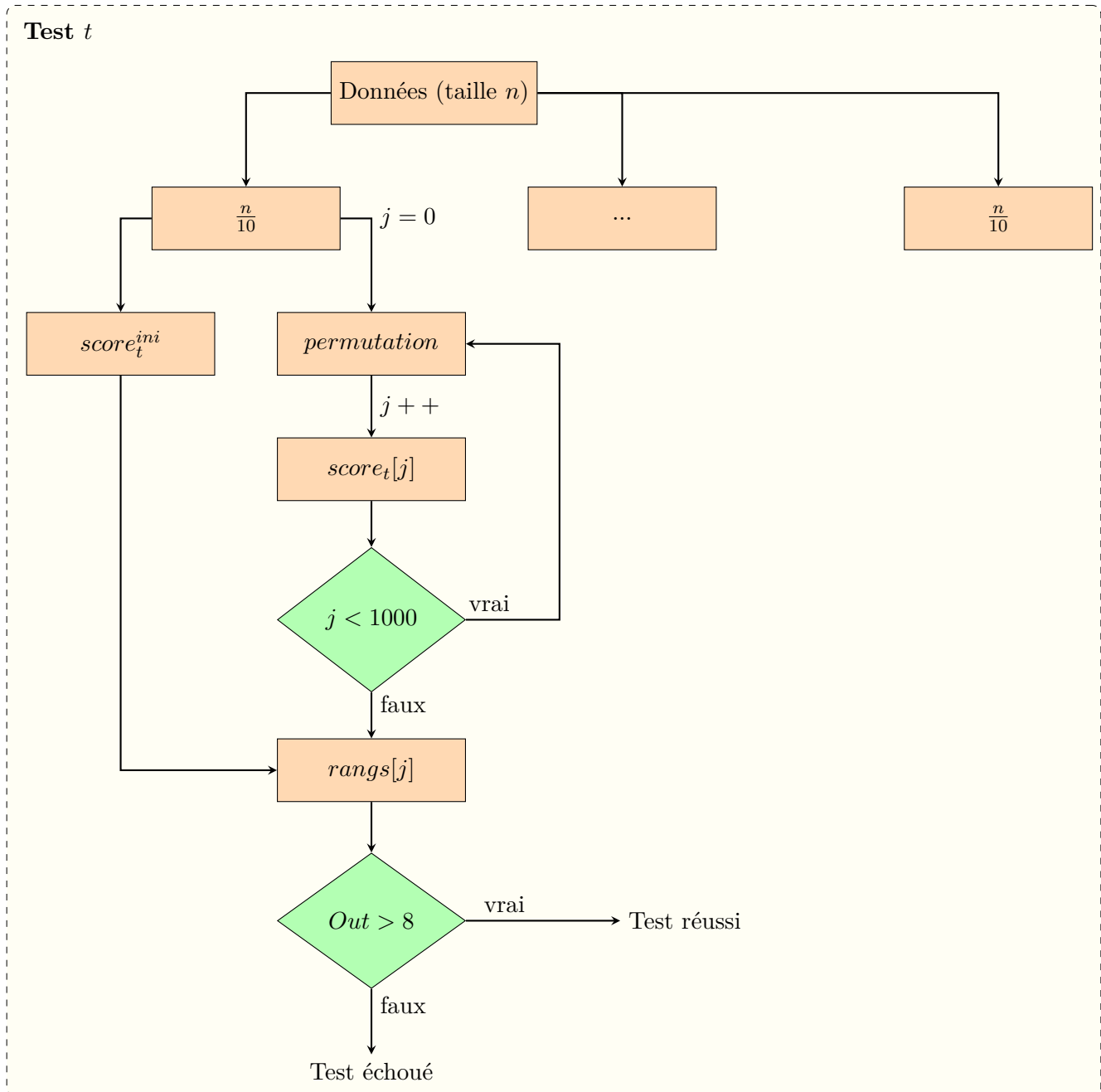


FIGURE 1.2 – Structure de validation de données  $d$  par un test  $t$ .  $Out$  vaut " $|Rangs \leq 50 \ \&\& \ Rangs \geq 950|$ "

#### 1.5.1.1.1 Score de compression

Les algorithmes de compression sont habituellement bien adaptés pour supprimer les données redondantes des chaînes de caractères. Suivant un algorithme de compression choisi, le score de compression est la longueur de la donnée compressée obtenue.

#### Calcul du score.

Le score est calculé de la façon suivante :

1. les sous-ensembles de données sont encodés en chaînes de caractères séparées par une virgule ;

2. la chaîne de caractères est compressée suivant l'algorithme de compression de bzip2<sup>2</sup> ;
3. le score retourné correspond à la longueur de la chaîne de caractères compressée.

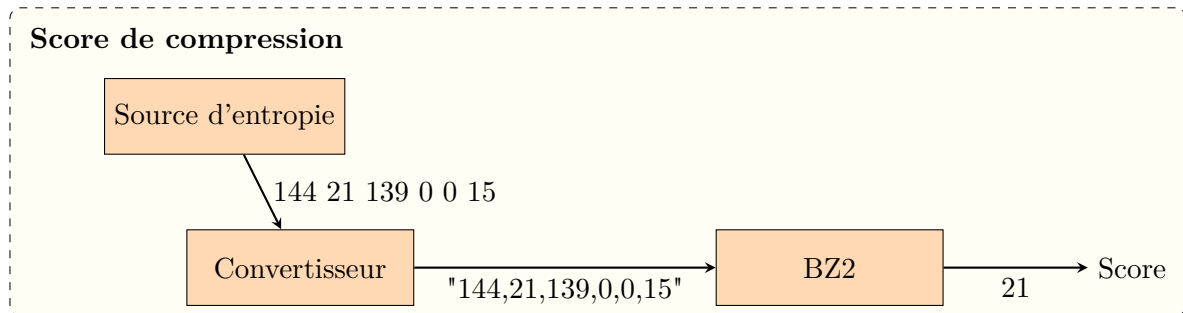


FIGURE 1.3 – Calcul du score de compression

#### 1.5.1.1.2 Scores *Over/Under Runs* (2)

##### Définition.

On définit le **run test** comme la série des valeurs montantes ou la série des valeurs décroissantes. Le terme de **run** peut être expliqué comme la successions d'éléments de la même classe. Le nombre d'augmentations ou de réductions définit la longueur du test. Dans un jeu de données aléatoires, la probabilité que la  $(i + 1)$ ème valeur soit plus grande ou plus petite que la  $i$ ème valeur doit suivre la loi binomiale.

Le *run test* est défini tel que :

- $H_0$  : la séquence a été produite de manière aléatoire ;
- $H_1$  : la séquence n'a pas été produite de manière aléatoire ;
- Test statistique :

$$Z = \frac{R - \bar{R}}{s_R}$$

où :

- $R$  : le nombre de *runs* observés ;
- $\bar{R}$ , la moyenne (le nombre de runs attendus) telle que :

$$\bar{R} = \frac{2N_+N_-}{N} + 1$$

;

- $s_R$  la variance (déviation standard) calculée telle que :

$$s_R^2 = \frac{2N_+N_-(2N_+N_- - N)}{N^2(N - 1)} = \frac{(\bar{R} - 1)(\bar{R} - 2)}{N - 1}$$

;

- niveau significatif :  $\alpha$  ;
- zone critique : Le *run test* rejette l'hypothèse nulle  $H_0$  si  $|Z| > Z_{1-\alpha/2}$  ;

2. cf. [www.bzip.org](http://www.bzip.org)

### Exemple de Run test.

Soit la suite binaire  $X = (x_1, x_2, \dots, x_n)$  sortie d'un générateur aléatoire. Pour tous les  $i$  allant de 1 à  $n - 1$ . On stocke dans un vecteur de taille  $n - 1$  le signe de  $x_{i+1} - x_i$ . Le nombre de *runs* correspond au nombre de changements de signe.

Si  $X = 111001111000110$ , alors son vecteur associé sera  $(+ + - + + + + - + + + -)$ , soit 6 *runs*. Dans le cas des suites binaires, il s'agit simplement de compter le nombre de changements d'éléments (ou classes), l'ensemble de données étant  $\{0, 1\}$ . Si l'élément  $i$  est équivalent au précédent, alors on passe au suivant, sinon on incrémente de un le nombre de *runs*.

Ici on a donc :

- $R = 6$  ;
- $\bar{R} = \frac{2*3*3}{15} + 1 = 2.2$  ;
- $s_R^2 = \frac{1.2*0.2}{14} = 0,017142857$  ;
- d'où  $Z = \frac{6-2.2}{\sqrt{0,017142857}} = 29.0229794$  ;
- on choisit un niveau  $\alpha = 0.05$  ;
- on cherche  $Z_{1-\alpha/2} = \frac{1-\alpha}{2} = 0.475$  soit suivant la courbe normale une valeur de  $Z_{1-\alpha/2} = 1.96$ . Ceci correspond à la valeur critique. La zone de rejet de  $H_0$  est telle que  $|Z| > 1.96$  ;
- étant donné que  $|Z| > 1.96$ , on en conclut que les données ne sont pas réparties aléatoirement au risque de 5%.

### Calcul des scores *Over/Under Runs*.

Pour chaque sous-ensemble, on calcule la médiane des données. On identifie ensuite les données en trois sous-ensembles :

- soit elles sont égales à la médiane ;
- soit elles sont inférieures à la médiane ;
- soit elles sont supérieures à la médiane.

Les sous ensembles inférieurs et supérieurs à la médiane sont susceptibles d'avoir un *run score* relativement faible, si les données sont suffisamment bien IID.

Les deux scores de *over/under runs* sont calculés comme suit :

1. on récupère la médiane de notre sous ensemble de données. Pour des données binaires, la médiane sera de 0,5
2. pour chaque sous ensemble original et chaque sous ensemble permuté, un sous ensemble temporaire est construit comme suit. Pour chaque élément :
  - (a) si l'élément est plus grand que la médiane, on ajoute +1 au vecteur temporaire ;
  - (b) si l'élément est plus petit que la médiane, on ajoute -1 au vecteur temporaire ;
  - (c) si l'élément est égal à la médiane, on passe à l'élément suivant.
3. la plus grande taille du *run* sur les +1 ou les -1 est considérée comme le premier score ;
4. la taille du *run* sur les +1 et les -1 est considérée comme le second score.

### Exemple.

Considérant l'ensemble de taille 7 ayant les données suivantes :  $\{5, 15, 12, 1, 13, 9, 4\}$ .

1. on récupère la médiane : 9 ;
2. création du vecteur temporaire :  $5 < 9 \Rightarrow -1$ .... Vecteur final :  $\{-1, +1, +1, -1, +1, -1\}$ . On note que pour ce vecteur la valeur 9 a été omise ;
3. le *run* le plus long pour le +1 et le -1 est de taille 2, (score 1) ;
4. le *run* global est de taille 5, (score 2).

#### 1.5.1.1.3 Score excursion

Le score d'excursion mesure la déviation en chaque point d'une somme d'éléments suivant leur moyenne.

### Définition.

Étant donné les éléments  $(s_0, s_1, \dots, s_i)$ , et leur moyenne  $\mu_i$ , l'**excursion**  $Esc_i$  est définie telle que :

$$Esc_i = s_0 + s_1 + \dots + s_i - i * \mu$$

Le score retourné est l'excursion maximale en valeur absolue du sous ensemble de données.

### Calcul du score.

Le score est ainsi calculé comme suit.  $\mu$  est défini comme la moyenne des valeurs d'un ensemble donné.

1. pour  $j = 1$  à  $\lfloor \frac{N}{10} \rfloor$  (taille des données du sous ensemble), on calcule  $d_j = |Esc_j| = \left| \sum_{i=0}^j s_i \right|$  ;
2. Score =  $\text{Max}_{j=0, j < \lfloor \frac{N}{10} \rfloor} (d_j)$ .

### Exemple.

Étant donné le sous-ensemble suivant :  $\{2, 15, 4, 10, 9\}$ .

1. calcule de  $\mu$ , ici  $\mu = 8$  ;
2. calcul des  $d_j$  :  
 $d_1 = |2 - 8| = 6$ ,  $d_2 = |2 + 15 - 2 * 8|$ ,  $d_3 = 3$ ,  $d_4 = 1$ ,  $d_5 = 0$  ;
3. Le score est donc de 6.

#### 1.5.1.1.4 Scores Runs directionnels (3)

Le principe du run test reste le même que celui évoqué précédemment à l'exception suivante près : lorsqu'il y a égalité entre deux éléments consécutifs, on le référence comme étant '0' dans le vecteur temporaire plutôt que de passer directement à l'élément suivant.



### Calcul des scores pour des données quelconques.

Le score est calculé de la façon suivante :

1. le nombre total de *runs* (sans considérer le 0 comme un changement de "signe") (score 1) ;
2. la longueur du plus long *run* (les '0' sont également ignorés) (score 2) ;
3. le maximum entre le nombre de données +1 et le nombre de données -1 (score 3).

#### Exemple :

Étant donné l'ensemble suivant :  $\{2, 2, 2, 5, 7, 7, 9, 3, 1, 4, 4\}$ .

1. on calcule son vecteur temporaire lié  $\{0, 0, +1, +1, 0, +1, -1, -1, +1, 0\}$  ;
2. nous comptons 3 séries de run :  $(0, 0, +1, +1, 0, +1)$ ,  $(-1, -1)$  et  $(+1, 0)$  (score 1) ;
3. le run le plus long est de 4 :  $(+1, +1, 0, +1)$  (score 2) ;
4. on compte 4 +1 et 4 -1, le maximum des deux est donc de 4 (score 3).

### Données binaires.

Dans le cadre des données binaires en sortie de source d'entropie, il convient de faire un pré-traitement sur les données :

1. on convertit les bits en bytes ;
2. on calcule le poids de hamming pour ces éléments :  
Pour  $i = 0$  à  $\lfloor \frac{N}{8} \rfloor - 1$ , on stocke dans  $W_i$  son poids de Hamming, tel que  $W_i = \text{hamming\_weight}(s_i, \dots, s_{i+7})$  ;
3. on réitère l'opération des données quelconques à partir de ce vecteur  $W_i$ .

### Poids de Hamming.

Le **poids de hamming** d'un sous ensemble  $(s_i, \dots, s_{i+n})$ , donné par  $\text{hamming\_weight}(s_i, \dots, s_{i+n})$  est défini comme le nombre de '1' de la suite  $(s_i, \dots, s_{i+n})$ .

#### 1.5.1.1.5 Score de covariance

Le score de covariance permet de détecter la relation entre les valeurs numériques successives. En effet, toute relation linéaire entre des paires successives va affecter directement ce score. On pourra alors constater une différence entre la covariance calculée sur le sous-ensemble de données originales et la covariance calculée sur le sous-ensemble de données permutées. La covariance est calculée entre chaque paire consécutive du sous-ensemble  $S$  tel que  $s_i$  est apparié avec  $s_{i+1}$ .

### Calcul du score.

Le score est calculé comme suit :

1. la variable *count* est initialisée à '0' ;
2.  $\mu$ , moyenne des données de  $s_0$  à  $s_{\lfloor \frac{N}{10} \rfloor - 1}$  est calculée ;

3. pour  $i = 1$  à  $\lfloor \frac{N}{10} \rfloor$  on incrémente *count* tel que :

$$count = count + (s_i - \mu)(s_{i-1} - \mu)$$

;

4. on obtient enfin le score :  $Score = \lfloor \frac{count}{\lfloor \frac{N}{10} \rfloor - 1} \rfloor$ .

### Exemple.

Étant donné le sous-ensemble :  $\{15, 2, 6, 10, 12\}$  de 5 éléments :

1.  $\mu = 9$ ;
2. – Pour  $i = 1$ ,  $count = 0 + (2 - 9)(15 - 9) = -42$ ;  
 – Pour  $i = 2$ ,  $count = -42 + (6 - 9)(2 - 9) = -21$ ;  
 – Pour  $i = 3$ ,  $count = -24$ ;  
 – Pour  $i = 4$ ,  $count = -21$ .
3.  $Score = \lfloor \frac{-21}{4} \rfloor = -5$ .

#### 1.5.1.1.6 Score de collision (3)

Une façon naturelle de tester l'entropie d'un générateur est de mesurer le nombre d'essais nécessaires pour obtenir une valeur identique à la première, en d'autres termes, le nombre d'essais nécessaires pour obtenir une collision. Ainsi, notre score de collision va mesurer le nombre d'essais successifs jusqu'à ce qu'un élément identique soit trouvé.

Dans le cas des données binaires, un sous-ensemble de données binaires est converti en séquence de 8 bits avant d'être testé.

### Calcul des scores.

Les trois scores de collisions sont calculés comme suit :

1. *Counts* est une liste d'échantillons nécessaires pour trouver une collision. La liste est vide à l'initialisation;
2.  $pos = 0$ ;
3. tant que  $pos < \lfloor \frac{N}{10} \rfloor$  :
  - (a) trouver le plus petit  $j$  tel que  $s_{pos} \dots s_{pos+j}$  :
    - i. si aucun  $j$  de ce type n'existe, on sort de la boucle tant que ;
  - (b) ajouter  $j$  à la liste *Counts* ;
  - (c)  $pos = pos + j + 1$  ;
4. on retourne les scores suivants :
  - (a) la valeur minimale de la liste *Counts* (score 1) ;
  - (b) la moyenne de la liste *Counts* (score 2) ;
  - (c) la valeur maximale de la liste *Counts* (score 3).

### Exemple.

Considérant les données :  $\{2, 1, 1, 2, 0, 1, 0, 1, 1, 2\}$  de taille 10.

1. on exécute le contenu de la boucle tant que :
  - (a) considérant 2 comme la 0e valeur, la première collision apparaît à  $j = 2$ , pour la valeur 1 ;
  - (b) on passe à une analyse sur le reste de l'ensemble initial non analysé :  $\{2, 0, 1, 0, 1, 1, 2\}$ . La première collision de ce sous-ensemble apparaît en  $j = 3$ , pour la valeur 0 ;
  - (c) on travaille à présent sur l'ensemble  $\{1, 1, 2\}$ . La première collision apparaît en  $j = 1$  ;
  - (d) enfin, on ne trouve aucune collision sur l'ensemble  $\{2\}$ .
2. on retourne les scores :
  - (a)  $\min(Counts) = 1$  (score 1) ;
  - (b)  $\mu_{Counts} = 2$  (score 2) ;
  - (c)  $\max(Counts) = 3$  (score 3).

#### 1.5.1.2 Test statistique spécifique : $\chi^2$

Dès lors que la source d'entropie est considérée comme IID, alors la distribution de ces valeurs peut être considérée comme une distribution indépendante, une distribution multinomiale.

#### Définition.

La **loi multinomiale** est une généralisation de la loi binomiale. On considère  $m$  résultats possibles (2 pour la loi binomiale). Soit  $N_i$  pour  $i \in \{1, \dots, m\}$  la variable aléatoire multinomiale ayant une probabilité  $p_i$ , telle que :

$$\sum_{i=1}^m N_i = 1 \text{ et } \sum_{i=1}^m p_i = 1$$

La densité de probabilité de cette loi s'écrit :

$$\mathbb{P}(N_1 = n_1, \dots, N_m = n_m) = \frac{n!}{n_1! \dots n_m!} p_1^{n_1} \dots p_m^{n_m}$$

Pour respectivement les espérances et variances suivantes :

$$\mathbb{E}[N_i] = np_i \text{ et } \mathbb{V}[N_i] = np_i(1 - p_i)$$

#### Approximation.

Si les variables sont indépendantes,  $\sum_{i=1}^m \frac{(N_i - np_i)^2}{np_i(1-p_i)}$  suit une loi du  $\chi^2$  à  $m$  degrés de liberté. Ainsi, le test du  $\chi^2$  peut ici être utilisé pour tester si des données suivent une distribution multinomiale.

Plus généralement, le test peut être utilisé pour vérifier si des données suivent une distribution particulière, pourvu que celles-ci ne soient pas de trop grande taille. Deux types de tests sont proposés sur la source d'entropie :

- le test d'indépendance entre des données successives obtenues (tests différents pour des données binaires et non binaires) ;

- le test d'ajustement sur les 10 sous-ensembles de données, qui permet de vérifier si le modèle adopté pour les données est satisfaisant, i.e. dans quelle mesure les résidus sont dus au hasard (tests différents pour des données binaires et non binaires).

Nous ne détaillerons pas dans ce rapport ces tests respectifs. Ils sont toutefois décrits dans le document du NIST SP800-90b.

### 1.5.2 Déterminer l'entropie minimale des sources IID

On souhaite à présent estimer l'entropie fournie par une source IID (données indépendantes et identiquement distribuées). Ce test est basé sur le nombre d'observations d'un échantillon le plus courant de la source de bruit ( $p_{max}$ ). Le but est de calculer la borne minimale d'entropie de la source.

#### Calcul de l'entropie minimale.

Étant donné  $N$  échantillons  $\{x_1, \dots, x_n\}$

1. trouver la valeur la plus souvent rencontrée dans le jeu de données ;
2. compter le nombre d'occurrences de cette valeur, stocké dans  $C_{MAX}$  ;
3. calculer  $p_{max} = \frac{C_{MAX}}{N}$  ;
4. calculer  $C_{BOUND} = C_{MAX} + 2.3\sqrt{N * p_{max}(1 - p_{max})}$  ;
5. calculer  $H = -\log_2(\frac{C_{BOUND}}{N})$  ;
6. calculer le nombre d'éléments dans l'échantillon, que l'on stocke dans  $W$  ;
7.  $\min(W, H)$  est l'entropie minimale.

#### Exemple.

Considérant le jeu de données  $\{0, 1, 1, 2, 0, 1, 2, 2, 0, 1, 0, 1, 1, 0, 2, 2, 1, 0, 2, 1\}$  de 20 données :

- la valeur la plus courante du jeu de données est 1 (On compte 6 0, 8 1 et 6 2) ;
- $C_{MAX} = 8$  ;
- $p_{max} = \frac{8}{20} = 0.4$  ;
- $C_{BOUND} = 8 + 2.3\sqrt{20 * 0.4 * 0.6} = 13.04$  ;
- $H = -\log_2(\frac{13.04}{20}) = 0.186$  ;
- $W = 3$  ;
- entropie minimale calculée :  $\min(3, 0.186) = 0.186$ .

## Chapitre 2

# Génération des clefs

### 2.1 Définitions et contexte

#### 2.1.1 Clefs cryptographiques

La génération de clef est une notion fondamentale de cryptographie. En effet, les données sont protégées grâce à des algorithmes (ou des méthodes cryptographiques) et des clefs cryptographiques. Les clefs cryptographiques sont notamment utilisées pour le chiffrement et le déchiffrement de données.

Relativement aux deux types de cryptographie, on compte deux types de clefs :

- les clefs symétriques pour le chiffrement symétrique ;
- les clefs publiques/privées pour le chiffrement asymétriques.

On se considérera dans ce chapitre dans un milieu optimal :

1. l'algorithme cryptographique considéré est optimal ;
2. la génération de bits aléatoire est optimale.

##### 2.1.1.1 Clefs d'algorithmes symétriques

Les algorithmes symétriques utilisent pour leur chiffrement une clef symétrique qui est partagée par chacun des correspondants.

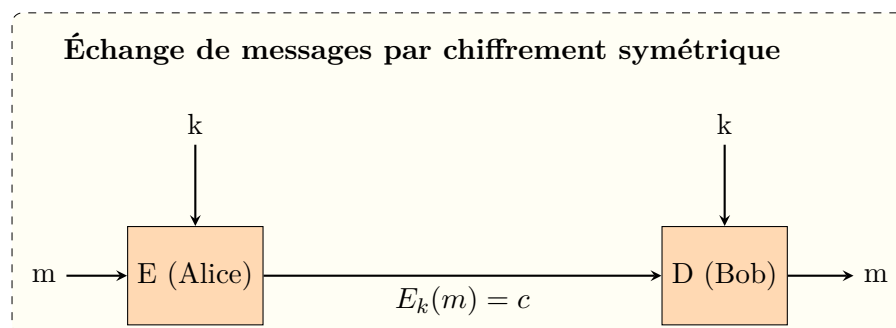


FIGURE 2.1 – Chiffrement symétrique - Alice envoie un message  $c$  chiffré à Bob qui le déchiffre

#### 2.1.1.2 Clefs d'algorithmes asymétriques

Les clefs d'algorithmes symétriques sont de deux catégories, les clefs privées et les clefs publiques. Les clefs publiques sont disponibles pour tous, par demande à l'utilisateur, ou bien souvent par certificat.

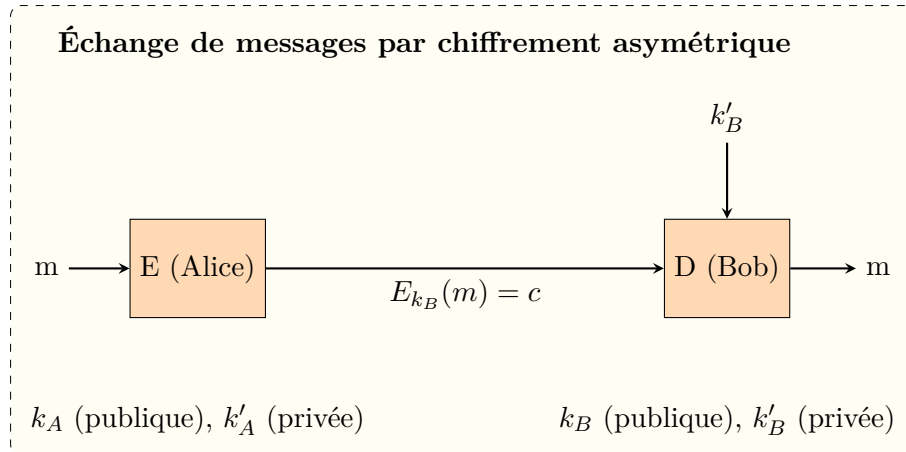


FIGURE 2.2 – Chiffrement asymétrique - Alice envoie un message  $c$  chiffré à Bob avec sa clef publique qui le déchiffre

Comme montré en figure 2.2, Alice chiffre avec la clef publique de Bob un message, qui lui-même déchiffre le message avec la clef privée.

#### 2.1.2 Méthode de génération et sécurité

Les systèmes utilisent des entiers comme clef. Celles-ci sont générées en utilisant des générateurs de nombres aléatoires (RNG) ou des générateurs de nombres pseudo aléatoires (PRNG).

##### 2.1.2.1 Génération de clefs quelconques

Un générateur de bits aléatoires (RNG) approuvé est utilisé pour générer la clef cryptographique. Le RNG doit fournir une entropie complète ou suffisante aux exigences de sécurité du système. Les RNG peuvent permettre de générer complètement et "directement" la clef. C'est le cas pour les clefs privées des algorithmes AES ou DSA. Les RNG peuvent aussi être utilisés comme *seed* pour générer des clefs, suivant des critères spécifiques. C'est le cas pour RSA, où habituellement le *seed* est utilisé comme point de départ pour trouver un nombre premier suivant les critères du FIPS 186-3 [?].

##### 2.1.2.2 Génération de paires de clefs asymétriques

Les algorithmes asymétriques demandent la création de paires de clefs asymétriques (privée/publique), dont chaque clef est associée à une seule entité, le possesseur de la clef. Les clefs sont généralement générées par son possesseur direct, ou par une autorité de confiance qui fournit la clef au possesseur de façon sûre. Il convient au moment de la génération de bien considérer deux paramètres principaux :

1. l'espace de clefs, qui doit être suffisamment grand pour éviter les doublons et les cycles ;
2. Générer les premiers de façon la plus aléatoire possible. Ceci nécessite un bon choix de stratégie d'identification de premiers. On peut par exemple utiliser des algorithmes permettant de détecter le prochain premier, suivant un entier tiré aléatoirement.

## 2.2 Audits

### 2.2.1 Audit 2.1 : Génération d'entiers premiers, et de grands nombres

#### 2.2.1.1 Description de la librairie BIGNUMBER

OpenSSL gère l'arithmétique multi-précision (gestion des grands entiers) avec sa librairie BIGNUMBER. La taille de ces entiers est arbitraire, il n'y a théoriquement pas de limitation de taille. Elle a été spécialement créée afin d'utiliser des systèmes cryptographiques à clefs publiques tels que RSA ou Diffie-Hellman.

Pour le stockage de grands entiers, elle utilise directement et dynamiquement la mémoire comme structure de données. Le code de retour est un entier permettant de vérifier l'état de la mémoire après chaque opération.

Le type de données pour ces grands entiers est une structure ayant pour nom BIGNUM. Cette structure peut être directement utilisée en incluant la bibliothèque `openssl/crypto/bn/bn.h`.

La structure générale d'un BIGNUM est définie dans le fichier `openssl/crypto/ssl_typ.h`, (cf. *Listing 2.1*), et elle est implantée dans le fichier `openssl/crypto/bn/bn.h` (cf. *Listing 2.2*). Un schéma explicatif est défini sur *Listing 2.3*.

```
1 | typedef struct bignum_st BIGNUM;
```

Listing 2.1 – `ssl_typ.h`

```
1 | struct bignum_st
2 | {
3 |     BN_ULONG *d; /* Pointer to an array of 'BN_BITS2' bit chunks. */
4 |     int top; /* Index of last used d + 1. */
5 |     /* The next are internal book keeping for bn_expand. */
6 |     int dmax; /* Size of the d array. */
7 |     int neg; /* one if the number is negative */
8 |     int flags;
9 | };
```

Listing 2.2 – `bn.h`

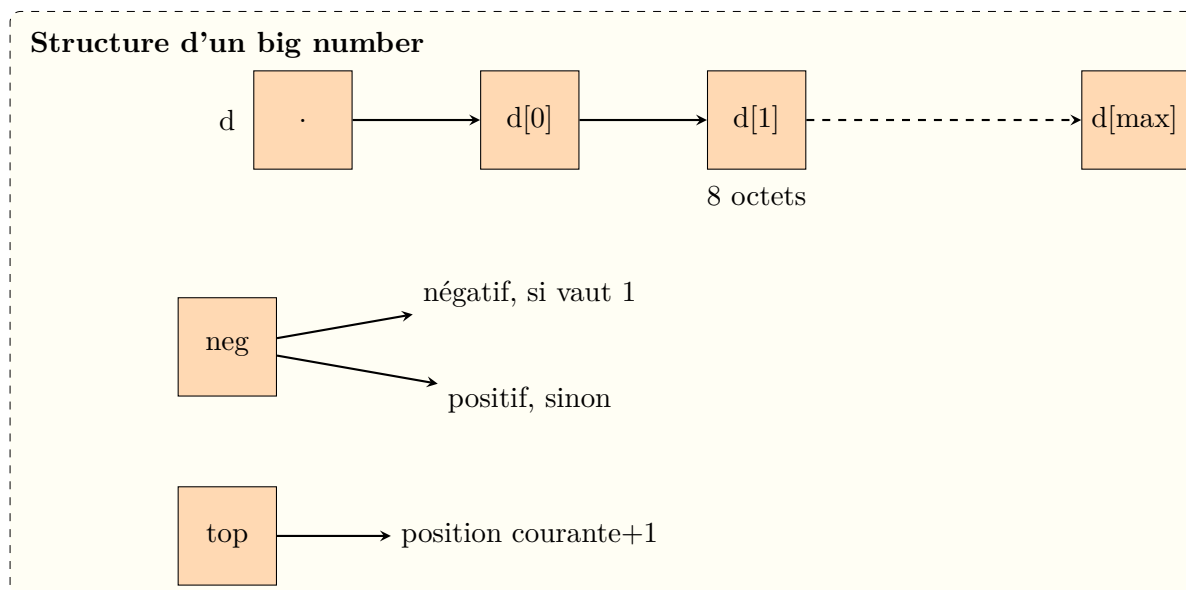


FIGURE 2.3 – Structure d'un BIGNUM

Il existe des structures variantes de BIGNUM pour d'autres cas d'utilisation (Montgomery, Division réciproque, génération lente).

La création d'une nouvelle structure de type BIGNUM se situe dans le fichier `openssl/crypto/bn/bn_lib.h`, la fonction se nomme `BN_new()` (cf. *Listing 2.3*).

```
1  BIGNUM *BN_new(void)
2  {
3      BIGNUM *ret;
4
5      if ((ret=(BIGNUM *)OPENSSL_malloc(sizeof(BIGNUM))) == NULL)
6      {
7          BNerr(BN_F_BN_NEW,ERR_R_MALLOC_FAILURE);
8          return(NULL);
9      }
10
11     ret->flags=BN_FLG_MALLOCED;
12     ret->top=0;
13     ret->neg=0;
14     ret->dmax=0;
15     ret->d=NULL;
16     bn_check_top(ret);
17     return(ret);
18 }
19
```

Listing 2.3 – `bn_lib.c`



### 2.2.1.2 Normes visées

Plusieurs fonctions désignent directement la norme visée, voici quelques exemples sur les thèmes nous concernant pour l'audit.

Sur la génération de groupes d'exponentiation modulaire de Diffie-Hellman allant de 768bits à 8192bits, les RFC visées [?] [?] ont leurs fonctions implantées dans le code :

- `BIGNUM *get_rfc2409_prime_768(BIGNUM *bn);`
- `BIGNUM *get_rfc2409_prime_1024(BIGNUM *bn);`
- `BIGNUM *get_rfc3526_prime_1536(BIGNUM *bn);`
- `BIGNUM *get_rfc3526_prime_2048(BIGNUM *bn);`
- `BIGNUM *get_rfc3526_prime_3072(BIGNUM *bn);`
- `BIGNUM *get_rfc3526_prime_4096(BIGNUM *bn);`
- `BIGNUM *get_rfc3526_prime_6144(BIGNUM *bn);`
- `BIGNUM *get_rfc3526_prime_8192(BIGNUM *bn);`

### RFC 2409 : Internet Key Exchange

La RFC 2409 [?] datant de 1998, donne deux implantations de fonctions récupérant des entiers premiers de plus de 768bits et de plus de 1024bits. Elles se nomment respectivement le premier groupe d'Oakley et le second groupe d'Oakley.

Un groupe MODP (Modular exponentiation group) est un groupe d'exponentiation modulaire.

Pour le groupe MODP sur 768bits (groupe par défaut qui DOIT être le minimum requis) le groupe utilise comme nombre premier :

$$2^{768} - 2^{704} - 1 + 2^{64} * \{[2^{638} pi] + 149686\}$$

Et comme générateur : 2.

Pour le groupe MODP sur 1024bits (groupe alternatif qui DEVRAIT être le groupe utilisé) le groupe utilise comme nombre premier :

$$2^{1024} - 2^{960} - 1 + 2^{64} * \{[2^{894} pi] + 129093\}$$

Et comme générateur : 2.

La RFC 2412 : OAKLEY Key Determination Protocol datant de 1998 également [?] préconise plutôt d'utiliser 22 comme générateur.

### RFC 3526 : More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)

La RFC 3526 [?] complète la RFC 2409. Elle définit les groupes MODP sur 1536, 2048, 3072, 4096, 6144 et 8192 bits.

*Exemple avec 2048 bits :*

Le nombre premier vaut :

$$2^{2048} - 2^{1984} - 1 + 2^{64} * \{[2^{1918} pi] + 124476\}$$

Le générateur vaut : 2

Sur la génération de nombres premiers, nous nous basons sur la RFC 2631 qui utilise les recommandations du FIPS 140-1 que l'on défini dans la section ??

### 2.2.1.3 Implémentation

#### Version visée.

La version audité est OpenSSL 1.0.0l sortie le 6 Janvier 2014.

#### Exemples de fonctions.

Les fichiers `bn_add.c`, `bn_exp.c`, `bn_div.c`, `bn_mul.c`, etc, présents dans le paquetage `openssl/crypto/bn` contiennent des fonctions arithmétiques simples (addition, multiplication, soustraction, exponentiation, etc) utilisant la structure `BIGNUM`.

#### Analyse de la génération d'entiers premiers.

La génération des entiers premiers se fait au sein du fichier `openssl/crypto/bn/bn_prime.c`. Parmi les fonctions intéressantes nous avons :

- `int BN_generate_prime_ex(BIGNUM *ret, int bits, int safe, const BIGNUM *add, const BIGNUM *rem, BN_GENCB *cb);`
- `int BN_is_prime_ex(const BIGNUM *p, int nchecks, BN_CTX *ctx, BN_GENCB *cb);`
- `int BN_is_prime_fasttest_ex(const BIGNUM *p, int nchecks, BN_CTX *ctx, int do_trial_division, BN_GENCB *cb);`

*int BN\_generate\_prime\_ex(BIGNUM \*ret, int bits, int safe, const BIGNUM \*add, const BIGNUM \*rem, BN\_GENCB \*cb);*

- génère un nombre premier aléatoire de *bits* bits;
- si *ret* vaut *NULL* alors l'entier n'est pas stocké, sinon il est stocké sur le descripteur de la structure `BIGNUM`;
- si *cb* n'est pas *NULL* il servira à générer un *callback* lors de la génération du i-ème nombre premier potentiel, lors du test de primalité et lorsqu'un entier premier a été trouvé;
- d'après la RFC 2631 [43] et en particulier les parties concernant le FIPS 186-1 [?], le paramètre *add* permet de vérifier la condition :  $p \bmod add == rem \vee p \bmod add == 1$  Si *rem* == *NULL*) et le paramètre *safe* vaut *true* pour la définition d'un nombre premier sûr, e.g qu'il résiste à certaines attaques cryptanalytiques (i.e un entier premier *p* dont  $(p - 1)/2$  est également premier est sûr);
- attention avant d'utiliser la fonction ci-dessus à bien passer une graine au PRNG;
- la génération d'un nombre premier a une probabilité d'échec négligeable (0.25% d'échecs possibles puissance le nombre de tentatives);
- la fonction retourne '1' si la génération est un succès, '0' sinon.

*int BN\_prime\_ex(const BIGNUM \*p, int nchecks, BN\_CTX \*ctx, BN\_GENCB \*cb);*

- fonction de test de primalité sur l'entier *p*;
- *nchecks* est le nombre de tests à effectuer;
- *cb* s'il n'est pas *NULL* génère un *callback* à chaque j-ème itération dans le code;
- *ctx* s'il n'est pas *NULL* est utilisé pour sauvegarder l'allocation du dessus et libérer la structure;

- la fonction retourne 0 si le nombre est premier avec une probabilité de 100%, 1 si la probabilité est plus faible que  $0.25^n \text{checks}$ , et -1 sinon.

*int BN\_is\_prime\_fasttest\_ex(const BIGNUM \*p, int nchecks, BN\_CTX \*ctx, int do\_trial\_division, BN\_GENCB \*cb);*

- cette fonction fait exactement la même chose que la fonction précédente;
- mais si *do\_trial\_division* vaut '1' il testera avant s'il n'est pas divisible par des petits nombres entiers, s'il vaut '0' cela revient à appeler la fonction *BN\_prime\_ex*

#### 2.2.1.4 Conclusion

Sur la présentation des différentes structures et des différentes opérations arithmétiques revisitées, la documentation est plus complète et plus claire que les autres fonctions auditées en général, même s'il reste à notre avis des efforts sur la mise en forme. Le découpage des fonctions est bien fait au sein du dossier `openssl/crypto/bn` avec des noms de fichiers cohérents, et au sein des fichiers avec des explications sur les différentes structures et des noms de fonctions cohérentes également même si on aurait préféré une ligne d'explications en en-tête de chaque fonction. Lors de l'audit, on retrouvait rapidement le code concernant une fonction.

Sur le contenu, les normes sont bien respectées, notamment pour la génération des nombres premiers qui utilise les tests de primalité de Miller-Rabin, réputés sûrs.

### 2.2.2 Audit 2.2 : Le générateur de Diffie-Hellman

#### 2.2.2.1 Normes visées

Voici l'algorithme de génération de  $g$ , d'après la RFC 2631 [43] datant de 1999 et dérivé de la FIPS-186 :

- 1- Soit  $j = (p - 1)/q$ .
- 2- Choisir  $h \in \mathbb{N}$ , tel que  $1 < h < p - 1$
- 3- Calculer  $g = h^j \mod p$
- 4- Si  $g = 1$  recommencer l'étape 2

Mais depuis 2006, on peut lire comme recommandation dans la RFC 4419 (pour une utilisation SSH) :  
*"It is recommended to use 2 as generator, because it improves efficiency in multiplication performance. It is usable even when it is not a primitive root, as it still covers half of the space of possible residues."*

#### 2.2.2.2 Description de la faille

Lorsque nous avons étudié le code de Diffie-Hellman dans OpenSSL, nous nous sommes penchés sur un choix plutôt étrange. La valeur du générateur est toujours fixé à 2 ou à 5.

Le générateur de Diffie-Hellman n'étant pas une racine primitive dans  $\mathbb{Z}/\mathbb{Z}_p$ , les conséquences sont :

- l'espace des clefs possibles est fortement réduit (Si  $g = 2 \implies$  espace divisé par deux);
- deux clefs privées distinctes pourront avoir une clef publique commune;
- la méthode de cryptanalyse Baby-step Giant-step peut s'en trouver facilitée.

Évidemment, ce choix n'est pas une faille en soit, il n'est juste pas optimal et résulte d'un bon compromis entre vitesse et sécurité. Pour une sécurité optimale, il est conseillé de choisir un générateur qui soit une racine primitive, pour être certain que personne ne puisse signer, déchiffrer des messages à votre place !

### 2.2.2.3 Implémentation

#### Configuration visée.

La version actuelle d'OpenSSL : 1.1.0

#### Fonction.

La fonction liée à cette norme est accessible sous le paquetage `openssl/crypto/dh/dh_gen.c` (cf. *Listing 2.4*). Les commentaires sont très intéressants pour comprendre le choix du générateur par l'équipe de développement.

```
1  if (generator <= 1)
2  {
3      DHerr(DH_F_DH_BUILTIN_GENPARAMS, DH_R_BAD_GENERATOR);
4      goto err;
5  }
6  if (generator == DH_GENERATOR_2)
7  {
8      if (!BN_set_word(t1,24)) goto err;
9      if (!BN_set_word(t2,11)) goto err;
10     g=2;
11 }
12 #if 0 /* does not work for safe primes */
13     else if (generator == DH_GENERATOR_3)
14     {
15         if (!BN_set_word(t1,12)) goto err;
16         if (!BN_set_word(t2,5)) goto err;
17         g=3;
18     }
19 #endif
20 else if (generator == DH_GENERATOR_5)
21 {
22     if (!BN_set_word(t1,10)) goto err;
23     if (!BN_set_word(t2,3)) goto err;
24     /* BN_set_word(t3,7); just have to miss
25      * out on these ones :-( */
26     g=5;
27 }
28 else
29 {
30     /* in the general case, don't worry if 'generator' is a
31      * generator or not: since we are using safe primes,
32      * it will generate either an order-q or an order-2q group,
33      * which both is OK */
34     if (!BN_set_word(t1,2)) goto err;
35     if (!BN_set_word(t2,1)) goto err;
36     g=generator;
37 }
```

Listing 2.4 – dh\_gen.c

#### 2.2.2.4 Conclusion

Ainsi selon la norme RFC-4419, le choix du générateur peut se résumer à un petit générateur qui ne serait pas une racine primitive. La criticité du risque est très grande, car si deux personnes possèdent la même clef publique pour deux clefs privées distinctes, ils pourront alors déchiffrer les messages, et signer à la place l'un de l'autre. Mais la probabilité d'une telle collision est quasiment nulle, il est simplement deux fois plus efficace de générer deux clefs privées possédant la même clef publique, que de retrouver la clef privée en brute force. OpenSSL laisse en plus le choix au développeur de choisir une racine primitive comme générateur, pour des cas d'extrême sécurité.

### 2.2.3 Audit 2.3 : Diffie-Hellman Ephémère en mode FIPS

#### 2.2.3.1 Normes visées

Les normes visées sont les mêmes que celles de l'Audit 2.2 (cf. 2.2.2.1).  
L'activation du mode FIPS entraîne l'utilisation des normes FIPS [?] [55]

#### 2.2.3.2 Description de la faille

Une faille plus grave concerne le mode FIPS (Federal Information Processing Standard) d'OpenSSL, qui peut être compilé avec la commande `./config fipsanisterbuild`. En effet, un attaquant situé entre le client et le serveur connaissant la clef secrète du serveur peut déchiffrer une session SSL/TLS.

L'algorithme EDH/DHE (Diffie-Hellman Éphémère) permet de calculer une nouvelle clef connue uniquement du client et du serveur, donc l'attaquant intermédiaire ne peut plus déchiffrer la session. Cependant, en mode FIPS, OpenSSL ne rejette pas les paramètres P/Q faibles pour EDH/DHE. Lorsque OpenSSL est compilé en mode FIPS, un attaquant en *Man in the Middle* peut donc forcer la génération d'un secret Diffie Hellman prédictible, en modifiant par exemple le trafic réseau. [?] [?].

La faille en elle même n'est pas suffisante pour réaliser l'attaque, elle requiert également une implémentation SSL faible.

#### 2.2.3.3 Implémentation

##### Configuration visée.

La version vulnérable d'OpenSSL est la 0.9.8, en mode FIPS uniquement.

##### Fonction.

La fonction est accessible au chemin `openssl/crypto/dh/dh_key.c`. La figure *Listing 2.5* montre le *diff* entre les deux versions.

```
1  ----- dh/dh_key.c -----
2  - if (!DH_check_pub_key(dh, pub_key, &check_result) || check_result)
3  - {
4  -     DHerr(DH_F_COMPUTE_KEY,DH_R_INVALID_PUBKEY);
5  -     goto err;
```

```
- }
```

Listing 2.5 – dh\_key.c

Cette partie causait un faux positif sous certaines conditions pouvant fragiliser le système.

#### **Solution logicielle.**

Nessus Vulnerability Scanner est un logiciel permettant de tester la configuration du serveur afin d'identifier entre autres de problèmes d'authentification [?]. Le système RedHat préconise son utilisation pour savoir si cette vulnérabilité concerne notre système.

#### **2.2.3.4 Conclusion**

Il est tout d'abord recommandé de passer à la version OpenSSL supérieure, sinon de désactiver le mode FIPS, ou encore de configurer la *ciphersuite* afin de ne pas permettre au serveur d'utiliser Diffie Hellman comme algorithme d'échange de clefs.

## Chapitre 3

# Chiffrement et Protocoles

### 3.1 Définitions et contexte

Nous allons tout d'abord apporter quelques précisions au contexte dans lequel nous travaillons. Plusieurs failles ont été trouvées sur RSA-OAEP ainsi que sur le mode CBC. Nous allons donc décrire ces deux principes et étudier leurs vulnérabilités.

Nous allons voir qu'une combinaison entre un mode de chiffrement et un protocole de chiffrement peut générer des failles, souvent graves (i.e vols de mots de passe), que le padding est également une donnée sensible et qu'il faut donc le choisir intelligemment, et qu'une mauvaise documentation ou un mauvais paramétrage peuvent entraîner des failles au niveau protocolaire.

### 3.2 Audits

#### 3.2.1 Audit 3.1 : Les "Manger's attack" sur RSA-OAEP

##### 3.2.1.1 Normes visées

**OAEP : *Optimal Asymmetric Encryption Padding*.**

Dans les chiffrements par blocs, cela nécessite généralement que tous les blocs soient d'une taille précise. Or ce n'est pas toujours le cas. Pour cela, on rajoute des bits de bourrage (padding).

OAEP est un schéma de remplissage, généralement utilisé avec RSA (en prétraitement). Il a été introduit en 1994 par Mihir Bellare et Phil Rogaway<sup>1</sup>. L'OAEP est une forme de réseau de Feistel qui nécessite une source d'aléa ainsi que deux fonctions de hachage.

RSA-OAEP peut être prouvé sûr dans un modèle théorique idéalisé, celui de l'oracle aléatoire. Il est recommandé par les PKCS.

OAEP a deux buts :

- insérer un élément d'aléatoire qui permet de passer d'un schéma déterministe à un schéma non déterministe (le même message clair chiffré deux fois avec la même clef et le même algorithme n'aura pas le même message chiffré.)
- prévenir un déchiffrement partiel en s'assurant que l'attaquant ne peut retrouver une portion du texte clair sans être capable d'inverser la fonction trapdoor (par exemple la factorisation de deux grands nombres premiers : il est facile de multiplier, mais quand on n'a que le produit il est très difficile de retrouver les facteurs).

Il n'est pas prouvé sûr pour une attaque IND-CCA (attaque à texte chiffré seulement). Victor Shoup a démontré qu'il n'existe pas de preuve générale. Il a montré que dans un cas IND-CCA, quelqu'un qui sait comme inverser partiellement une primitive d'insertion mais ne sait pas comment l'inverser complètement, pourrait bien être en mesure de casser le système. Par exemple, on peut imaginer quelqu'un qui peut attaquer RSAES-OAEP si on sait comment retrouver tous les octets exceptés les 20 premiers d'un entier généré aléatoirement chiffré avec RSAEP (RSA Encryption Protocole). Un tel attaquant n'a pas besoin d'être capable d'inverser entièrement RSAEP, parce qu'il n'utilise pas les 20 premiers octets dans son attaque.

### Réseau de Feistel.

Il est utilisé dans les systèmes de chiffrement par bloc. Un schéma de Feistel repose sur des principes simples dont des permutations, des substitutions, des échanges de blocs de données et une fonction prenant en entrée une clé intermédiaire à chaque étage.

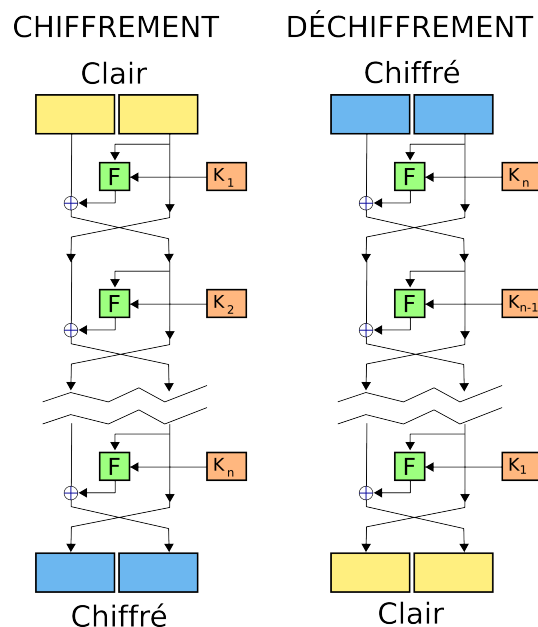


FIGURE 3.1 – Réseau de Feistel

Le chiffrement DES par exemple repose sur ce réseau, et effectue 16 tours. Généralement les deux parties sont équilibrées même si par exemple des algorithmes comme MacGuffin de Bruce Schneier utilisent un réseau non équilibré.

### RSA-OAEP.

Dans la PKCS#1 est décrit le standard de RSA-OAEP. RSAES-OAEP est le terme plus facilement utilisé dans le document : RSA Encryption Scheme OAEP.

Il regroupe les primitives RSAEP et RSADP : respectivement RSA Encryption Protocole et RSA Decryption Protocole.

### RSAES-OAEP-ENCRYPT.

Options :



**Hash** : fonction de hachage (**hLen** contient la longueur en octets de la sortie de fonction de hachage) ;

**MGF** : fonction de génération de masque.

Entrée :

(**n**, **e**) : destinataire de la clef publique RSA (**k** contient la longueur en octets du modulo RSA **n**) ;

**M** : message à chiffrer, une chaîne d'octets de longueur **mLen**, quand  $mLen \leq k - 2hLen - 2$  ;

**L** : champ optionnel à associer au message, la valeur par défaut pour **L**, si **L** n'a aucune condition, est la chaîne vide.

Sortie :

**C** : texte chiffré, une chaîne d'octets de longueur **k**.

Erreurs :

- "message too long" ;
- "label too long".

Précondition :

- la clef publique RSA (**n**, **e**) est valide.

Étapes :

1. vérifier de la longueur :

- si la longueur de **L** est plus grande que la longueur limite en entrée de la fonction de hachage ( $2^{61} - 1$  octets pour SHA-1), renvoyer "label too long" et arrêter ;
- si  $mLen > k - 2hLen - 2$ , le message "message too long" est renvoyé et la fonction est stoppée.

2. coder EME-OAEP :

- si l'étiquette **L** n'est pas spécifiée, laisser **L** à une chaîne vide. Laisser  $lHash = Hash(L)$ , une chaîne d'octets de taille **hLen** ;
- générer une chaîne d'octets **PS** consistant en  $k - mLen - 2hLen - 2$  d'octets zéro. La taille de **PS** peut être zéro ;
- concaténer **lHash**, **PS**, un unique octet avec la valeur hexadécimale *0x01*, et le message **M** pour former un bloc de données **DB** de longueur  $k - hLen - 1$  octets, tel que :  $DB = lHash || PS || 0x01 || M$  ;
- générer une chaîne d'octets aléatoires de longueur **hLen** ;
- laisser  $dbMask = MGF(seed, k - hLen - 1)$  ;
- laisser  $maskedDB = DB \text{ xor } dbMask$  ;
- laisser  $seedMask = MGF(maskedDB, hLen)$  ;
- laisser  $maskedSeed = seed \text{ xor } seedMask$  ;
- concaténer un unique octet avec la valeur hexadécimale *0x00*, **maskedSeed**, et **maskedDB** pour former un message chiffré **EM** de longueur **k** octets tel que  $EM = 0x00 || maskedSeed || maskedDB$ .

### 3. chiffrement RSA :

- convertir le message codé EM en un entier représentatif du message  $m$  :  $m = \text{OS2IP}(EM)$  ;
- appliquer la primitive de chiffrement RSAEP avec la clef RSA publique  $(n, e)$  pour produire un entier  $c$  représentatif du message chiffré :  $c = \text{RSAEP}((n, e), m)$  ;
- convertir le texte chiffré représentatif  $c$  en un texte chiffré  $C$  de taille  $k$  octets :  $C = \text{I2OSP}(c, k)$ .

### 4. envoyer en sortie le texte chiffré $C$ .

#### Note.

Si  $L$  est une chaîne vide, la valeur du hash correspondante  $lHash$  a la représentation hexadécimale suivante pour différents choix de hash :

SHA-1 :	(0x)da39a3ee	5e6b4b0d	3255bfef	95601890	afd80709	
SHA-256 :	(0x)e3b0c442	98fc1c14	9afb4c8	996fb924	27ae41e4	649b934c
	a495991b	7852b855				
SHA-384 :	(0x)38b060a7	51ac9638	4cd9327e	b1b1e36a	21fdb711	14be0743
	4c0cc7bf	63f6e1da	274edebf	e76f65fb	d51ad2f1	4898b95b
SHA-512 :	(0x)cf83e135	7eeb8bd	f1542850	d66d8007	d620e405	0b5715dc
	83f4a921	d36ce9ce	47d0d13c	5d85f2b0	ff8318d2	877eec2f
	63b931bd	47417a81	a538327a	f927da3e		

TABLE 3.1 – Représentations hexadécimales

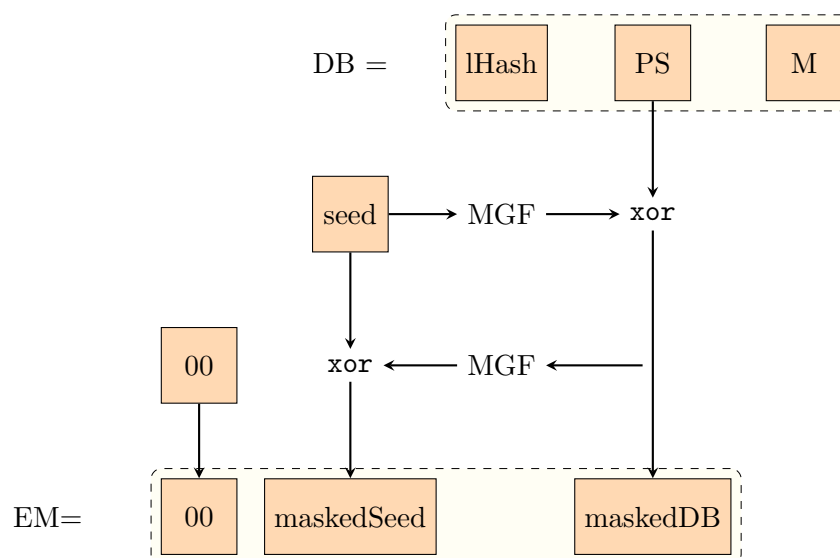


FIGURE 3.2 – Opération de chiffrement EME-OAEP

$lHash$  est le hash de l'étiquette optionnelle  $L$ . L'opération de déchiffrement suivant inverse les étapes pour retrouver  $M$  et vérifier  $lHash$  et  $PS$ .

## RSAES-OAEP-DECRYPT (K, C, L).

Options :

**Hash** : fonction de hachage (**hLen** contient la longueur en octets de la sortie de la fonction de hachage) ;

**MGF** : fonction de génération du masque.

Entrée :

**K** : destinataire de la clef privée RSA (**k** contient la longueur en octets du modulo RSA **n**) ;

**C** : texte chiffré à déchiffrer, une chaîne de caractères de taille **k**, où  $k = 2hLen + 2$  ;

**L** : champ optionnel dont l'association avec le message doit être garantie ; la valeur par défaut pour **L** est, si pas de spécification, une chaîne vide.

Sortie :

**M** : message, une chaîne d'octets de longueur **mLen**, où  $mLen \leq k - 2hLen - 2$ .

Erreur :

– "decryption error" ;

Étapes :

### 1. Vérification des longueurs :

- Si la longueur de **L** est supérieure à la taille limite en entrée de la fonction de hachage ( $2^{61}-1$  octets pour SHA-1), renvoie "decryption error" et s'arrête ;
- Si la longueur du texte chiffré **C** n'est pas de **k** octets, renvoie "decryption error" et s'arrête ;
- Si  $k < 2hLen + 2$ , renvoie "decryption error" et s'arrête.

### 2. déchiffrement RSA :

- Convertir le texte chiffré **C** en un entier **c** représentatif du message chiffré :  $c = \text{OS2IP}(C)$  ;
- Appliquer la primitive de déchiffrement **RSADP** à la clef privée RSA **K** et au message chiffré représentatif **c** pour produire un entier **m** représentatif du message clair :  $m = \text{RSADP}(K, c)$  ; si **RSADP** renvoie "ciphertext representative out of range" (signifie que  $c \geq n$ ), renvoie "decryption error" et s'arrête ;
- convertir le message représentatif **m** en un message déchiffré **EM** de longueur **k** octets :  $EM = \text{I2OSP}(m, k)$ .

### 3. déchiffrement EME-OAEP :

- si l'étiquette **L** n'est pas spécifiée, laisser **L** à une chaîne vide, laisser  $lHash = \text{Hash}(L)$ , une chaîne d'octets de longueur **hLen** ;
- séparer le message encodé **EM** dans un seul octet **Y**, une chaîne d'octets **maskedSeed** de longueur **hLen**, et une chaîne d'octets **maskedDB** de longueur  $k - hLen - 1$  telle que  $EM = Y || \text{maskedSeed} || \text{maskedDB}$  ;
- laisser  $\text{seedMask} = \text{MGF}(\text{maskedDB}, hLen)$  ;
- laisser  $\text{seed} = \text{maskedSeed} \text{ xor } \text{seedMask}$  ;

- (e) laisser `dbMask = MGF(seed, k - hLen - 1)` ;
- (f) laisser `DB = maskedDB xor dbMask` ;
- (g) séparer `DB` en une chaîne d'octet `lHash` de longueur `hLen`, une chaîne de padding (possiblement vide) `PS` consistant en des octets hexadécimaux de valeur `0x00`, et un message `M` tel que `DB = lHash || PS || 0x01 || M` ;  
s'il n'y a pas d'octet avec la valeur hexadécimale `0x01` pour séparer `PS` de `M`, si `lHash` n'est pas égal à `lHash`, ou si `Y` n'est pas une sortie non nulle, renvoyer "decryption error" et s'arrêter.

4. renvoyer le message `M`.

#### Note.

Il faut faire attention qu'un adversaire ne puisse distinguer les différentes erreurs dans les conditions de l'étape 3, que ce soit par un message d'erreur, ou un temps de réponse différent, ou, plus généralement, apprendre une information partielle à propos du message en clair `EM`. Sinon un adversaire peut être en mesure d'obtenir des informations utiles sur le déchiffrement du texte chiffré `C`, conduisant à une attaque à chiffré choisi telle que celle observée par Manger.

#### **RSAES-PKCS1-v1\_5.**

`RSAES-PKCS1-v1_5` combine les primitives `RSAPB` et `RSADP` avec la méthode de codage `EME-PKCS1-v1_5`. Il est mathématiquement équivalent au schéma de chiffrement dans la `PKCS # 1 v1.5`. `RSAES-PKCS1-v1_5` peut fonctionner sur des messages de longueur supérieure à `k - 11` octets (`k` est la longueur en octets du modulo `RSA`), bien qu'il faille faire attention aux attaques portant sur les faibles exposants `RSA` menée par Coppersmith, Franklin, Patarin, and Reiter quand les longs messages sont chiffrés.

En règle générale, l'utilisation de ce schéma pour chiffrer un message arbitraire, en opposition à une clef générée aléatoirement, n'est pas recommandée.

Il est possible de générer des textes chiffrés `RSAES-PKCS1-v1_5` valides sans connaître les messages clairs correspondants, avec une probabilité raisonnable de réussite.

Cette possibilité peut être exploitée dans une attaque à chiffré choisi. Par conséquent, si `RSAES-PKCS1-v1_5` doit être utilisé, certaines contre mesures faciles à implémenter devraient être mises en place afin de contrecarrer l'attaque trouvée.

Des exemples typiques comprennent l'ajout de la structure des données à encoder, le contrôle rigoureux de la conformité des `PKCS# 1v1.5` (et d'autres redondances) dans les messages déchiffrés, et la consolidation des messages d'erreur dans un protocole client-serveur basé sur `PKCS # 1 v1.5`. Ils peuvent tous être des contre-mesures efficaces et n'entraînent pas de changement à un protocole n°1 sur la base de `v1.5-PKCS`. Il a été récemment montré que la sécurité du protocole `SSL / TLS handshake`, qui utilise `RSAES-PKCS1-v1_5` et certaines contre-mesures, peut être liée à une variante du problème `RSA`.

#### Note.

Les passages suivants décrivent des recommandations concernant l'utilisation de `RSAES-PKCS1-v1_5`. Les recommandations de la version 1.5 de ce document sont incluses ainsi que de nouvelles recommandations motivées par les avancées de cryptanalyses durant les années suivantes.

- il est recommandé que les octets pseudo-aléatoires soient générés indépendamment pour chaque processus de chiffrement, en particulier si la même donnée est en entrée pour plus d'un processus de chiffrement. Les résultats de Haastad sont une des motivations pour cette recommandation ;
- la chaîne de padding PS est d'une longueur d'au moins 8 octets, ce qui est une condition de sécurité pour les opérations sur les clefs publiques, et qui rend difficile pour les attaquants de récupérer les données en essayant tous les blocs chiffrés possibles.
- les octets pseudo aléatoires peuvent aussi aider à contrecarrer une attaque grâce à Coppersmith et al. quand la taille du message à chiffrer est gardé petit. L'attaque marche sur les petits exposants RSA quand des messages similaires sont chiffrés avec la même clef publique. Plus spécifiquement, une façon peut être, quand deux entrées RSAEP correspondent sur une large portion de bits (8/9) et qu'un petit exposant RSA est utilisé ( $e = 3$ ) pour chiffrer les deux, il peut être possible de retrouver les entrées avec l'attaque. Une autre façon d'attaquer est couronnée de succès pour déchiffrer un seul texte chiffré, quand une large proportion (2/3) des entrées de RSAEP est déjà connue. Pour des applications typiques, le message à chiffrer est court (par exemple une clef symétrique de 128 bits) donc peu d'informations seront connues ou en commun entre deux messages pour permettre l'attaque. Cependant, si un long message est chiffré, ou si une partie du message est connu, alors l'attaque peut fonctionner. Dans tous les cas, le schéma RSAES-OAEP surmonte l'attaque.

### 3.2.1.2 Faille

RSA-OAEP peut être soumis à une attaque nommée "Mangers Attack" selon son implantation [?]. OpenSSL semble être vulnérable à une attaque de ce type, à base de "prédictions" par injections de fautes. La vulnérabilité semble être très récente puisqu'elle fonctionne sous OpenSSL 1.0.0.

Le padding OAEP devait palier le problème d'insécurité que causait le padding PKCS#1 v1.5 (attaque à chiffré choisi) [?]. OpenSSL a tout de même pris en compte cette vulnérabilité et a placé des contre-mesures efficaces. La Technische Universität Darmstadt (Allemagne) explique en détails comment sont implémentées ces contre-mesures et montre que dans certains cas l'attaque reste possible. Enfin, elle apporte ses propres contre-mesures.

On peut noter que plusieurs bibliothèques sont vulnérables à une attaque de Manger qui consiste à contrôler la taille des paramètres à hacher, mais que l'implantation de RSA-OAEP d'OpenSSL ne le permet pas. La raison est que le décodage OAEP est linéaire quelle que soit la taille des paramètres et les erreurs survues. Il semble également y avoir un problème avec l'OAEP\_padding sur le chiffrement RSA. Bill Nickless recommande l'utilisation de PKCS\_padding. [?]

### 3.2.1.3 Implémentation

#### Configuration visée.

L'étude a été réalisée sur la bibliothèque OpenSSL-1.0.0.

#### Fonction.

La fonction audité se nomme `RSA_padding_check_PKCS1_OAEP()` (cf. *Listing 3.1*) et est accessible à partir du chemin `openssl/crypto/rsa/rsa_oaep.c`.

```
1 | lzero = num -flen;
2 | if (lzero < 0)
3 | {
4 |     /* signalling this error immediately after detection might allow
5 |      * for side-channel attacks (e.g. timing if 'plen' is huge
6 |      * -- cf. James H. Manger, "A Chosen Ciphertext Attack on RSA
7 |      * Optimal
8 |      * Asymmetric Encryption Padding (OAEP) [...]", CRYPTO 2001),
9 |      * so we use a 'bad' flag */
10 | bad = 1;
11 | lzero = 0;
12 | flen = num; /* don't overflow the memcpy to padded_from */
13 | }
```

Listing 3.1 – rsa\_oaep.c

Le développeur n'a pas considéré qu'il y avait un grand danger dans le code de contre-mesures. Pourtant l'étude confirme qu'il y a un décalage de temps possible, certes léger mais qui peut entraîner une attaque en "branch prediction" (qui peut se traduire en prédiction par dérivation).

#### 3.2.1.4 Conclusion

Il n'y a pas vraiment de quoi s'alarmer, cette attaque est en pratique infaisable sur un serveur car il y a suffisamment de variations de délais (différences de CPU, opérations multi-tâches, connexions réseaux, etc...) pour éviter une attaque par timing. Cependant, sur des systèmes embarqués l'attaque peut être réalisable, et il serait plus prudent de palier ce problème.

### 3.2.2 Audit 3.2 : Chiffrement SSLv3 ou TLS 1.0 en mode CBC

#### 3.2.2.1 Normes visées

##### SSL version 3

Cette partie référence la RFC 6101 [19] du protocole SSL version 3, concernant le chiffrement par bloc. Dans ce chiffrement, chaque bloc de texte clair est chiffré dans un bloc de donnée chiffrée. Comme il est très peu probable que le texte clair (qu'importe la donnée à envoyer) ne se découpe exactement en taille de blocs nécessaire (qui est de 64 bits normalement), on doit alors rajouter du bourrage à la fin de ces petit blocs avec des valeurs régulières, normalement des '0'.

Lorsqu'un chiffrement CBC est utilisé, un vecteur d'initialisation (IV) est maintenu pour chaque clef. Cette valeur est initialisée pour la première fois par la poignée de main du protocole SSL. Par la suite, le dernier bloc du texte chiffré de chaque tour est préservé comme IV pour le tour suivant.

```
block-ciphered struct {  
    opaque content[SSLCompressed.length];  
    opaque MAC[CipherSpec.hash_size];  
    uint8 padding[GenericBlockCipher.padding_length];  
    uint8 padding_length;  
} GenericBlockCipher;
```

- content : le contenu du bloc de message.
- MAC : la valeur du haché MAC.
- padding : le bourage qui est utilisé pour forcer la taille du bloc.
- padding\_length : la taille du bourage utilisé qui doit être plus petit que la taille du bloc chiffré et il peut être à '0'.

La taille des données chiffrées (`SSLCiphertext.length`) est égale à la somme de `SSLCompressed.length`, `CipherSpec.hash_size` et `padding_length + 1`.

### CBC : Cipher Block Chaining

C'est un mode de chiffrement qui a été très utilisé : enchaînement des blocs. Sur chaque bloc, un OU exclusif avec le chiffrement du bloc précédent est appliqué. Un vecteur d'initialisation est lui aussi utilisé. Contrairement au mode ECB, les blocs identiques ne seront pas chiffrés de la même façon. On ne pourra donc pas repérer de chaîne de caractères récurrentes aussi facilement. Ce mode de chiffrement possède plusieurs inconvénients :

- un chiffrement de plusieurs blocs en parallèles est impossible (puisque chaque bloc dépend du chiffrement du précédent). Déchiffrer avec un IV incorrect entraînera une corruption dans le premier bloc en clair, mais les blocs suivants seront corrects. C'est parce qu'un texte clair peut être récupéré grâce à deux blocs adjacents du texte chiffré. Le déchiffrement, contrairement au chiffrement, peut donc être parallélisé. À noter que si un seul bit change dans le texte chiffré, le bloc clair correspondant est complètement corrompu.
- Si une erreur se produit sur un bloc, elle sera répercutée sur tous les suivants. La propagation d'erreur n'est pas limitée.

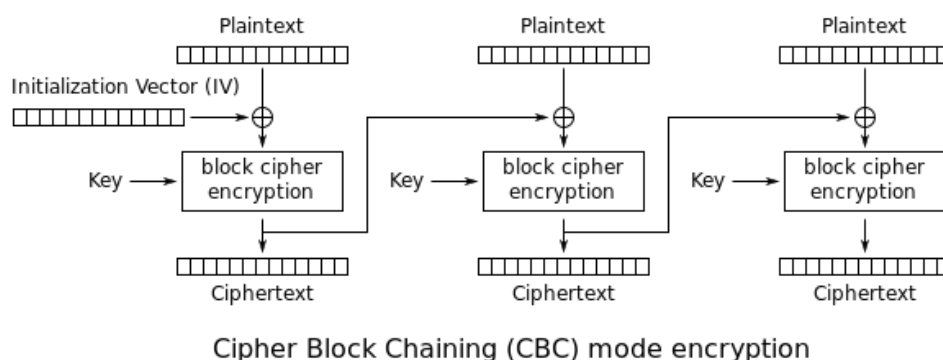


FIGURE 3.3 – Chiffrement CBC

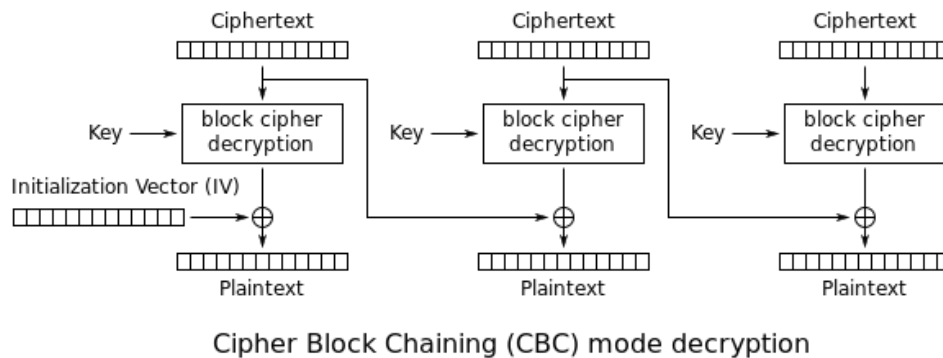


FIGURE 3.4 – Déchiffrement CBC

Il a d'abord été défini par le NIST dans le FIPS 81 (<http://www.itl.nist.gov/fipspubs/fip81.htm>). Le standard a été publié en 1981.

### 3.2.2.2 Description de la faille

En Septembre 2011, une attaque en *man in the middle* très efficace a vu le jour contre les protocoles SSLv3 et TLS 1.0. [?] [?] [?] [?]. L'attaque est à clair choisi. Le but étant d'insérer des morceaux de texte clair grâce au navigateur dans la requête chiffrée avec ces protocoles, ceci afin de récupérer les cookies de session.

La technique est basique, un individu enregistre plusieurs cookies de session auprès de divers sites officiels (banques, messageries, etc...). Puis, il clique malencontreusement sur du code Java malveillant (publicité, image, etc...). et l'attaque se déroule automatiquement. L'ensemble des cookies est envoyé au serveur malveillant qui n'a plus qu'à déchiffrer les clés de session.

La cause viendrait du mode de chiffrement choisi : CBC. SSL/TLS est un protocole qui chiffre un canal de communication. De ce fait il ne chiffre pas un fichier unique, mais une série d'enregistrements. Il y a deux façons d'utiliser le mode CBC dans ce cas précis :

- prendre chacun de ces enregistrements indépendamment des autres. Générer un nouveau vecteur d'initialisation à chaque fois ;
- traiter ces enregistrements comme un seul objet en les concaténant. Le vecteur d'initialisation est donc choisi aléatoirement pour le premier enregistrement et pour les autres, il aura pour valeur le dernier bloc de l'enregistrement précédent.

SSLv3 et TLS 1.0 utilisent ce deuxième choix, cela soulève un lourd problème de sécurité. En 2004, Moeller [?] trouve une méthode pour exploiter ce mauvais choix afin de récupérer des morceaux de textes clairs. Il y a certes une faille immense, mais peu exploitable. Les grandes entreprises savent (normalement) qu'il ne faut pas utiliser le mode CBC pour du chiffrement SSL/TLS. Et, dans tous les cas, plusieurs navigateurs ne permettent pas ce type d'attaque (c'est le cas de Chrome par exemple).

### 3.2.2.3 Tests

Nous n'avons pas repris les tests du logiciel BEAST qui s'avère être introuvable sur le Web (celui-ci étant un projet universitaire, développé par un étudiant de l'Université de Versailles). Mais une vidéo de l'exploit



est accessible sur YouTube au lien ci-dessous :

**Lien YouTube :** <http://www.youtube.com/watch?v=ujz4SXzWK9o>

#### 3.2.2.4 Recommandations

La faille existe tant que l'association de ces protocoles avec le mode de chiffrement CBC existe. Même si l'attaque est infaisable sur les navigateurs les plus répandus (Chrome, Firefox, IE, Safari, ...), OpenSSL devrait pouvoir interdire cette association, et ne pas laisser le travail aux navigateurs. Mais rien n'empêche l'utilisation de ce chiffrement par un navigateur plus léger, nous pourrions tester cette vulnérabilité lors de notre partie 3 si nous trouvons un navigateur acceptant ce type de chiffrement.

### 3.2.3 Audit 3.3 : Non-validation des certificats SSL

#### 3.2.3.1 Normes visées

##### RFC 5246

La RFC 5246 [16] stipule que le serveur doit toujours envoyer un message certificat dès lors que la méthode d'échange des clefs a été acceptée. Le certificat DOIT être approprié à la suite des chiffrements utilisés pour les échanges de clefs. La structure du message est :

```
opaque ASN.1Cert<1..224-1>;
struct {
    ASN.1Cert certificate_list<0..224-1>;
} Certificate;
```

La liste de certificats (`certificate_list`) est une séquence de certificats. Le certificat de l'expéditeur DOIT venir en premier. Les suivants sont ceux qui forment la chaîne de certification. Les certificats auto-signés qui spécifient un certificat racine peuvent être omis de la chaîne, sous condition que le certificat à l'extrémité de la chaîne soit dans tous les cas validé.

Le même message type et la même structure seront utilisés pour la réponse au client du message de requête de certificat.

#### 3.2.3.2 Description de la faille

Six chercheurs des universités de Stanford et d'Austin au Texas, analysent une attaque en Man in the Middle autour des certificats SSL sans utilisation d'un navigateur. Le titre est sans appel "Le code le plus dangereux du monde" [?].

SSL doit permettre d'être sécurisé en toutes circonstances, que le cache DNS soit empoisonné, que les attaquants contrôlent les points d'accès et les routeurs, etc. Il assure théoriquement trois grands principes de la cryptologie : la confidentialité, l'intégrité et l'authentification. Nous connaissons certaines failles au niveau du navigateur et de l'implantation SSL (voir ci-dessus). Mais il existe également d'autres cas d'utilisation du protocole SSL. Par exemple :

- administration à distance basé sur le cloud, stockage sécurisé sur le cloud en local ;
- transmissions de données sensibles (ex : e-commerce) ;
- services en ligne comme les messageries électroniques ;

- authentification via applications mobiles comme Android et iOS.

L'étude montre que la validation des certificats SSL est basée sur plusieurs applications et bibliothèques dont :

- OpenSSL
- JSSE
- CryptoAPI
- NSS
- GnuTLS
- etc...

En fait, un attaquant en *Man In The Middle* peut intercepter le secret entre un client et un serveur utilisant une connexion SSL. Il peut ainsi récupérer des numéros de carte bancaire, avoir accès à une messagerie, récupérer des mots de passe, etc. La cause principale vient du fait que les développeurs retouchent les bibliothèques cryptographiques à leur façon. En voulant réparer un bug ou en souhaitant rendre SSL compatible avec leurs API, ils injectent de nouvelles vulnérabilités. De plus, l'application est souvent propriétaire et payante ce qui rend le débogage difficile.

Que ce soit accidentel ou intentionnel, l'une des conséquences les plus graves est la non-validation de certificat sur des contextes où la sécurité est primordiale (e.g. paiement en ligne). La faute ne revient pas directement au code d'OpenSSL, mais à une mauvaise utilisation des différentes fonctions et options.

Voici quelques exemples concrets concernant différentes API :

- les services comme Amazon's Flexible Payments Service PHP et PayPal Payments Standard PHP passent le paramètre `CURLOPT_SSL_VERIFYHOST` à *true* alors que la valeur doit être passée à 2. La conséquence est la désactivation de la validation du certificat ;
- lynx, un navigateur textuel très connu et souvent utilisé dans le développement d'applications, vérifie les certificats auto-signés seulement si la fonction de validation de certificat GnuTLS retourne une valeur négative. Malheureusement, dans certains cas la fonction peut retourner 0 pour certaines erreurs (dont les certificats signés par une autorité sans confiance) ;
- la bibliothèque SSLSocketFactory de JSSE, très réputée, ne fait pas de vérification si la cypher suite du client vaut NULL ou est une chaîne vide ;
- vulnérabilités sur Apache HttpClient, WebSockets, Android, ... ;
- autres causes célèbres : non reconnaissance des expressions régulières, non vérification du résultat de la validation, désactivation de l'authentification.

### 3.2.3.3 Difficultés du code OpenSSL

OpenSSL ne déroge pas à la règle.

Voici quelques vulnérabilités du code :

- les contraintes de nom x509 ne sont pas correctement validés ;
- les applications DOIVENT fournir elles même leur code de vérification de nom d'hôte. Or, des protocoles comme HTTPS, LDAP ont chacun leurs propres notions de validations. Ainsi, Apache Libcloud utilise les bibliothèques Python, elles-même utilisant des commandes OpenSSL. Et sa méthode de vérification du nom d'hôte comporte des vulnérabilités pouvant causer des attaques en *Man in the Middle*

(e.g "google.com" et "oogle.com" vérifie la même expression régulière) ;

- un programme utilisant OpenSSL peut exécuter la fonction `SSL_connect` pour le handshake SSL. Bien que certaines erreurs de validation soient signalées par `SSL_connect`, d'autres ne peuvent être vérifiées qu'en appelant la fonction `SSL_get_verify_result`, alors que `SSL_connect` se contente de retourner "OK".

#### 3.2.3.4 Exemple : Trillian

Trillian est une messagerie cliente instantanée reliée à OpenSSL pour la sécurisation de l'établissement de connexion. Par défaut OpenSSL ne soulève pas d'exception en cas de certificat auto-signé ou de non-confiance auprès de la chaîne de vérification. À la place, il envoie un drapeau. De plus, il ne vérifie jamais le nom d'hôte. Si l'application appelle la fonction `SSL_CTX_set` pour initialiser le drapeau `SSL_VERIFY_PEER`, alors `SSL_connect` se ferme et affiche un message d'erreur lorsque le certificat n'est pas valide. Mais Trillian n'initialise jamais ce drapeau. Par conséquent, `SSL_connect` va retourner 1 et le statut de la validation du certificat peut être connu en appelant la fonction `SSL_get_verify_result`. Encore une fois, Trillian n'appelle pas cette fonction. Les conséquences sont très lourdes : vols de mots de passe, compromissions de services, révélations des paramètres de sécurité, etc...

L'étude montre que l'attaque est possible sur la version 5.1.0.19 et antérieure de Trillian.

#### 3.2.3.5 Conclusion

Les chercheurs nous donnent alors plusieurs leçons à retenir, dont voici quelques points :

- premièrement, les vulnérabilités doivent être trouvées et réparées lors des phases de tests. Certaines se trouvent très facilement si les procédures de tests sont bien réalisées ;
- deuxièmement, la plupart des bibliothèques SSL ne sont pas **sûres par défaut**, laissant le choix de la sécurité aux applications de plus haut niveau avec choix des options, choix de la vérification de l'hôte, choix d'interprétation des résultats ;
- troisièmement, même les bibliothèques SSL sûres par défaut peuvent être mal utilisées par des développeurs changeant les paramètres par défaut par des paramètres non sécurisés. La cause peut venir d'une **mauvaise documentation** ou d'une mauvaise formalisation de la part de l'API. Les API devraient entre autres proposer des abstractions de haut niveau pour les développeurs comme des tunnels d'authentification, plutôt que de les laisser traiter des détails de bas niveau comme la vérification du nom d'hôte.

Nous conseillons surtout une meilleure documentation d'OpenSSL, et des rapports d'erreurs d'interfaces plus simples et plus consistants afin d'éviter les erreurs d'interprétation. L'idée des chercheurs de proposer des abstractions de haut niveau pour les applications semblent être une très bonne idée.

## Chapitre 4

# Signature et authentification

### 4.1 Définitions et contexte

Une signature électronique utilise le concept de la traditionnelle signature sur papier et la tourne en une empreinte électronique. Cette empreinte est un message encodé et est unique pour chaque document et chaque signataire. La signature permet alors de garantir l'authenticité du signataire pour son document. Toute modification dans le document après l'avoir signé rend la signature invalide, ce qui protège alors contre les fausses informations et la contrefaçon des signatures.

De ce fait, il est important de faire attention à toutes les formes de vulnérabilités des signatures afin de comprendre les attaques possibles sur la contrefaçon des signatures. Cette partie est consacrée à la bonne compréhension et l'implémentation qui sont définies dans la RFC, afin de se prémunir des différentes attaques possibles. On verra, cependant, qu'il existe quand même des failles au niveau des injections, surtout lorsque la librairie dépend trop du matériel.

### 4.2 Audits

#### 4.2.1 Audit 4.1 : Attaque par injection de fautes sur les certificats RSA

##### 4.2.1.1 Normes visées

Dans la RFC 3447 [25], la signature est décrite telle une primitive de signature qui produit la représentation de la signature depuis un message sous le contrôle d'une clef privée. La vérification se fait alors en récupérant la représentation du message depuis la représentation de la signature sous le contrôle de la clef publique correspondante.

La signature se déroule en deux opérations qui sont la génération et la vérification. L'opération de génération consiste donc à générer une signature depuis un message avec la clef privée de l'utilisateur (signataire) et l'opération de vérification consiste à vérifier la signature en se basant sur le message en utilisant la clef publique du signataire. Ce schéma peut être utilisé dans de multiples applications telles que les certificats X.509.

La norme spécifie deux types de schéma de signature qui sont :

- RSASSA-PSS ;

– RSASSA-PKCS1-v1\_5.

Même si aucune attaque n'est connue contre RSASSA-PKCS1-v1\_5, RSASSA-PSS est recommandé dans l'intérêt d'augmenter la robustesse d'un système. RSASSA-PKCS1-v1\_5 est toujours incluse pour des raisons de compatibilité avec les applications existantes et, même si elle est toujours appropriée dans les nouvelles applications, une transition vers RSASSA-PSS est encouragée.

La norme décrit un modèle général à suivre (qui est également utilisé pour IEEE Std 1363-2000) combinant les primitives de signature et de vérification avec une méthode d'encodage sur les signatures. L'opération de génération de signature applique une opération d'encodage de message à un message pour produire un message encodé, qui est ensuite converti en un entier représentatif du message. Une primitive de signature est alors appliquée sur la représentation du message pour produire la signature.

Dans le sens inverse, l'opération de vérification de signature applique une primitive de vérification de signature à la signature pour récupérer la représentation du message, qui est alors convertie dans un message de chaîne de caractères encodé. L'opération de vérification est appliquée au message et au message encodé pour déterminer s'ils consistent bien en l'un et l'autre.

#### **RSASSA-PSS :**

RSASSA-PSS combine les primitives RSASP1 et RSAVP1 avec la méthode d'encodage EMSA-PSS. La longueur du message sur laquelle RSASSA-PSS peut travailler est soit illimitée, soit contrainte par une très grande valeur, dépendant de la fonction de hachage. Contrairement à RSASSA-PKCS1-v1\_5, un identificateur de fonction de hachage n'est pas incluse dans le message encodé par EMSA-PSS. De ce fait, en théorie, il est possible pour un attaquant de substituer une autre fonction de hachage (potentiellement plus faible) que celle sélectionnée par le signataire. Il est alors recommandé que la fonction de génération de masquage d'EMSA-PSS soit basée sur la même fonction de hachage. De cette façon, l'encodage de tout entier dépendra de la fonction de hachage et il sera plus difficile pour un adversaire de substituer une autre fonction que ce qui a été sélectionné par le signataire.

La comparaison entre fonctions de hachage est seulement utilisée pour empêcher la substitution de fonction de hachage, et n'est pas nécessaire si la fonction de hachage est substituée d'une autre façon (e.g., le vérificateur n'accepte qu'une fonction de hachage désignée).

Ce qui est différent pour RSASSA-PSS des autres méthodes de signature RSA, c'est qu'il est probabiliste plutôt que déterministe, du fait de l'incorporation d'un aléa. Cette valeur d'aléa augmente la sécurité de la méthode. Cependant, le fait que la valeur soit aléatoire n'est pas critique pour la sécurité. Dans les situations où l'aléatoire n'est pas possible, une valeur fixe ou une séquence de nombres peut être employée plutôt, et avec une sécurité similaire.

#### **4.2.1.2 Description de la faille**

L'Université du Michigan a réussi l'exploit de récupérer la clef privée d'un certificat RSA en un peu plus de 100h [42] [20]. L'attaque fonctionne par injection de fautes [29] sur la méthode d'authentification. La technique est donc très poussée, mais le résultat en vaut la chandelle. L'injection de fautes doit se faire sur quelques bits pour ne pas faire dysfonctionner le système tout entier. Les signatures erronées produites

révéleront de l'information sur la clef privée. Avec le bon matériel et 100h d'attente, la clef peut être reforgée.

La technique consiste à renseigner de fausses signatures afin de vérifier les fautes avec la clef publique de la machine. Lorsque la signature est identifiée comme fausse, elle est envoyée à un analyseur contenant l'algorithme du *Listing 4.1*

```
1      window\_search (m, s, e, win\_size, win\_idx) {  
2          found = 0;  
3          for(d[win\_idx] in [0..2^win\_size-1];  
4              sqr\_iter in [0..win\_size-1];  
5              fault in [0..card(bits(d))-1]) {  
6              found += test\_equation 10(m, s, e,  
7                  win\_idx, d[win\_idx], sqr\_iter, fault loc)  
8          }  
9          if (found == 1)  
10             return d[win\_idx]  
11         else  
12             return -1  
13     }
```

Listing 4.1 – window\_search.c

$d$  : fenêtre,  $win\_size$  : taille fixée sur la fenêtre,  $win\_idx$  : index de la fenêtre. Et  $m, s, e$  des entiers pour l'opération d'exponentiation modulaire :

$$m^e[s]$$

En contrôlant le voltage, on arrive à savoir qu'un bit en particulier est mauvais. Bit après bit on reconstruit la clef privée, avec une signature erronée pour chacun des exposants de toutes les fenêtres. L'étude montre que 650 signatures corrompues suffisent pour retrouver 100% d'une clef privée RSA de 1024 bits.

#### 4.2.1.3 Implémentation

##### Configuration utilisée.

Le logiciel utilisé pour l'étude est OpenSSL 0.9.8i. L'attaque se fait sur la librairie d'authentification OpenSSL sous un système SPARC Linux qui implémente FPGA pour les systèmes cartes à puces.

##### Fonction Fixed-window modular exponentiation.

Cette fonction est accessible dans plusieurs fonctions de chiffrement :

- RSA ;
- ElGamal ;
- DSA ;
- Diffie-Hellman ;
- etc.

Elle garantit des opérations en temps constant afin d'éviter des attaques par timing, et elle reste très performante. Elle s'apparente à la technique *square-and-multiply* à la seule différence qu'elle utilise des fenêtres de largeur  $w$  bits, et partitionne l'exposant dans ces fenêtres au lieu d'examiner l'exposant pour différentes opérations comme *square-and-multiply*.

C'est le fait que la fenêtre soit fixe qui la rend insensible à une attaque par timing.

L'algorithme FWE est donnée en *Listing 4.2*

```
1  FWE(m, d, n, win_size) {  
2      num_win = card(bits(d)) / win_size;  
3      acc = 1;  
4      for(win_idx in [num_win-1..0])  
5          for(sqr_iter in [0..win_size-1])  
6              acc = (acc * acc) mod n;  
7      d[win_idx] =  
8          bits(d, win_idx * win_size, win_size);  
9      acc = (acc * m^d[win_idx]) mod n;  
10     return acc;  
11 }
```

Listing 4.2 – fwe.c

L'inconvénient de cet algorithme est qu'il utilise plus de 1 000 multiplications. Or, il est connu que pour une attaque par injection de fautes, la multiplication est l'opération la plus sensible en cas de dégradation du microprocesseur. *"The fixed-window exponentiation algorithm in the OpenSSL library does not validate the correctness of the signature produced before sending it to the client, a vulnerability that we exploit in our attack"*

#### 4.2.1.4 Conclusion

Lorsque le système est vulnérable, OpenSSL ne le détecte pas forcément. Le risque est donc très fort, et les contre-mesures sont parfois difficiles à trouver dans les phases de tests. Toutefois, cette étude soulève un choix de programmation qui semble à première vue anodin, mais qui peut avoir de lourdes conséquences.

Toutefois, il faut pouvoir contrôler la machine (en ayant un accès au BIOS par exemple) pour pouvoir exploiter cette faille car il faut pouvoir toucher directement à l'alimentation de la faille.

Cependant, cette erreur n'est pas à prendre à la légère, car une attaque à base de faisceaux lumineux est en cours de développement afin de réaliser cette attaque à distance.

## 4.2.2 Audit 4.2 : Malformation des signatures DSA/ECDSA

### 4.2.2.1 Normes visées

La RFC R979 [?] définit l'utilisation de DSA (*Digital Signature Algorithm*) et ECDSA (*Elliptic Curve Digital Signature Algorithm*) de façon déterministe. DSA et ECDSA sont deux standards de signature électronique qui offrent l'intégrité et authenticité dans de nombreux protocoles.

Une caractéristique de DSA et ECDSA est qu'ils ont besoin de produire, pour chaque génération de signature, une valeur aléatoire toute fraîche (**k**). Pour une bonne sécurité, **k** doit être choisi aléatoirement et uniformément depuis un groupe d'entiers modulaires, en utilisant un processus cryptographiquement sûr. Même une petite erreur dans le processus peut devenir une attaque sur la méthode de signature. De ce fait,

un système qui génère mal ou pas suffisamment d'entropie lors de la génération d'un nombre aléatoire peut poser de grosses failles dans le déploiement du schéma de signature DSA et ECDSA.

Cette méthode d'utilisation de l'aléatoire avec DSA et ECDSA fait son implémentation plus difficile à tester. Les tests automatiques ne sont pas fiables lorsqu'il s'agit de détecter si l'implémentation utilise une source aléatoire de grande qualité. De ce fait, le processus d'implémentation est alors plus vulnérable à un échec catastrophique, souvent découvert après le déploiement du système et suite à des attaques réussies.

Il est possible de retourner DSA et ECDSA en une utilisation déterministe en utilisant un processus déterministe de génération d'une valeur "aléatoire",  $k$ . Ce processus doit remplir quelques caractéristiques cryptographique afin de maintenir les propriétés de vérifiabilité et d'infalsifiabilité attendues par cette méthode de signature. De ce fait, pour une personne ne connaissent pas la clef privée de la signature, la transformation du message en valeur correspondante,  $k$ , doit être calculatoirement indiscernable du retour de la fonction aléatoire et uniforme.

#### 4.2.2.2 Description de la faille

En 2008, une vulnérabilité sur la malformation des signatures survient sur OpenSSL (re-analysé en Novembre 2012) [49] [54].

La cause vient de plusieurs fonctions implémentant la fonction `EVP_VerifyFinal()`. Elles valident de fausses signatures au lieu de retourner des erreurs, parmi les signatures corrompues peuvent se trouver :

- des signatures DSA ;
- des signatures ECDSA.

En 2009, un cas similaire a été trouvé dans un autre protocole (NTP) avec la même fonction `EVP_VerifyFinal` [53].

La conséquence est très grave, car cette faille permet une attaque en *Man in the Middle*, en faisant par exemple une attaque par *phishing* en HTTPS où la validation de la chaîne des certificats serait valide.

#### 4.2.2.3 Tests

Si la faille est toujours exploitable, les conséquences sont très graves. Nous pouvons tester sur d'anciennes versions OpenSSL si cette faille persiste.

#### 4.2.2.4 Implémentation

##### Configuration visée.

La faille concerne toutes les versions antérieures à OpenSSL 0.9.8j, lorsqu'un client SSL/TLS utilise des clefs DSA/ECDSA pour s'authentifier sur un serveur.

##### Fonction.

La fonction `EVP_VerifyFinal()` (cf. *Listing 4.3*) est accessible sous le paquetage `openssl/crypto/evp/p_verify.c`.

```
1 | int EVP_VerifyFinal(EVP_MD_CTX *ctx, const unsigned char *sigbuf,  
2 |                   unsigned int siglen, EVP_PKEY *pkey)  
3 | {  
4 |     unsigned char m[EVP_MAX_MD_SIZE];  
5 |     unsigned int m_len;
```



```
7      int i,ok=0,v;
8      MS_STATIC EVP_MD_CTX tmp_ctx;
9
10     for (i=0; i<4; i++)
11     {
12         v=ctx->digest->required_pkey_type[i];
13         if (v == 0) break;
14         if (pkey->type == v)
15         {
16             ok=1;
17             break;
18         }
19     }
20     if (!ok)
21     {
22         EVPerr(EVP_F_EVP_VERIFYFINAL,EVP_R_WRONG_PUBLIC_KEY_TYPE);
23         return(-1);
24     }
25     EVP_MD_CTX_init(&tmp_ctx);
26     EVP_MD_CTX_copy_ex(&tmp_ctx,ctx);
27     EVP_DigestFinal_ex(&tmp_ctx,&(m[0]),&m_len);
28     EVP_MD_CTX_cleanup(&tmp_ctx);
29     if (ctx->digest->verify == NULL)
30     {
31         EVPerr(EVP_F_EVP_VERIFYFINAL,EVP_R_NO_VERIFY_FUNCTION_CONFIGURED);
32         return(0);
33     }
34     return(ctx->digest->verify(ctx->digest->type,m,m_len,
35                               sigbuf,siglen,pkey->pkey.ptr));
36 }
37
```

Listing 4.3 – EVP\_VerifyFinal.c

La fonction retourne '1' si la signature est valide, '0' si la signature est incorrecte et '-1' pour toute autre raison. Mais dans certains cas cette fonction retournait toujours '0'.

#### 4.2.2.5 Conclusion

Ici, la faille persistera tant que le serveur et le client resteront à une version antérieure à OpenSSL 0.9.8j ; les clefs, quant à elles, ne sont pas vulnérables, et peuvent être conservées. Malheureusement, le nombre de serveurs tournant sous OpenSSL 0.9.8 et versions antérieures est très élevé.

Il est également recommandé aux développeurs utilisant OpenSSL de faire des audits réguliers de la fonction EVP\_VerifyFinal() pour s'assurer que les vérifications sont bonnes. Les tests étant assez simples à effectuer.

## Chapitre 5

# Protocoles SSL/TLS

### 5.1 Définitions et contexte

SSL et TLS (successeur de SSL) sont des protocoles de sécurisation des échanges sur internet. Les versions 2 et 3 de SSL ont été développées par Netscape puis le brevet a été racheté par l'IETF en 2001 qui a publié une évolution de ce protocole en TLS. Ce protocole fonctionne selon un mode client-serveur et fournit les objectifs de sécurité suivants :

- authentification serveur/client ;
- confidentialité des données échangées ;
- intégrité des données échangées.

Du point de vue réseau, ce protocole se situe dans la couche session du modèle OSI et entre transport et application dans le modèle TCP.

Pour simplifier la compréhension des parties suivantes, le schéma 5.1 représente de manière large l'établissement d'une connexion SSL/TLS. Les données entre accolades sont chiffrées avec la clef indiquée en indice. La *masterkey* est la clef principale qui sera dérivée pour chiffrer chaque message.

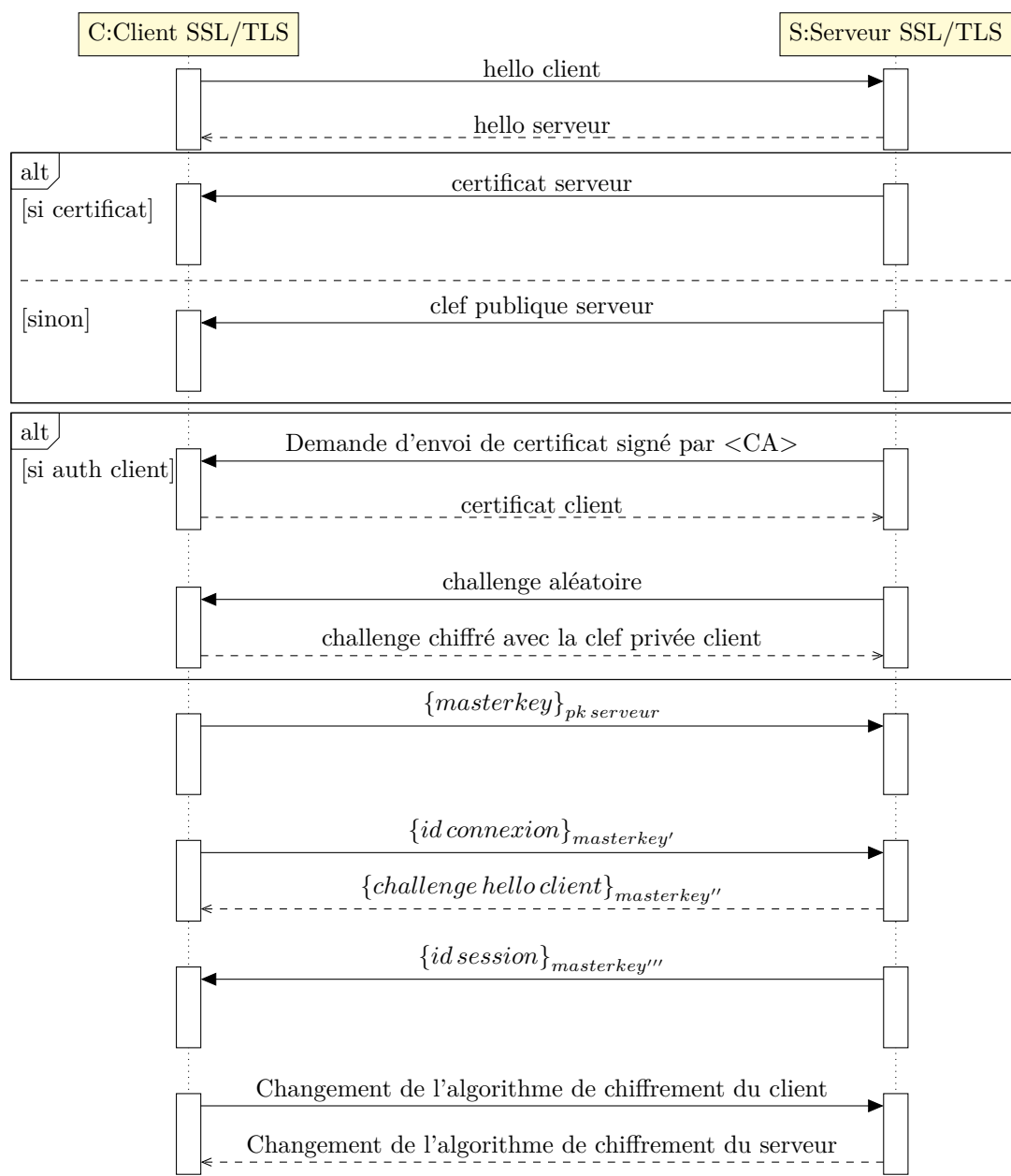


FIGURE 5.1 – Schéma global d'une connexion SSL/TLS

**hello client** Version du protocole SSL avec laquelle le client souhaite communiquer, challenge, algorithmes de chiffrement supportés par le client, méthodes de compressions supportées par le client.

**hello serveur** Version du protocole SSL calculée par le serveur (plus haute version du serveur supportée également par le client), challenge, id de session, algorithmes de chiffrement supportés par le serveur, méthodes de compression supportées par le serveur.

## 5.2 Audits

### 5.2.1 Audit 5.1 : SSL version 2

#### 5.2.1.1 Spécifications

Il n'existe pas de RFC pour SSL version 2. En effet, ce protocole a été pensé et développé par la société Netscape Communications. Cette version est sortie en 1994. Toutefois, on trouve des morceaux d'informations dans la RFC 6176 [52] et le draft de Hickman [23].

Identifiant	KeyExch	Authn	Enc	MAC
SSL_CK_RC2_128_CBC_WITH_MD5	RSA	RSA	RC2.128 CBC	MD5
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	RSA.512	RSA	RC4.40 CBC	MD5
SSL_CK_IDEA_128_CBC_WITH_MD5	RSA	RSA	IDEA.128 CBC	MD5
SSL_CK_DES_64_CBC_WITH_MD5	RSA	RSA	DES.56 CBC	MD5
SSL_CK_DES_192_EDE3_CBC_WITH_MD5	RSA	RSA	3DES.168 CBC	MD5
SSL_CK_RC4_128_WITH_MD5	RSA	RSA	RC4.128	MD5
SSL_CK_RC4_128_EXPORT40_WITH_MD5	RSA.512	RSA	RC4.40	MD5

TABLE 5.1 – Algorithmes supportés SSLV2

**Remarque** Le CK signifie CIPHER-KIND.

#### 5.2.1.2 Implémentation

Dans le code d'OpenSSL, cette version du protocole SSL se trouve dans les fichiers commençant par `s2_` du répertoire `ssl/`. Les constantes sont déclarées dans le fichier `ssl2.h`, on retrouve bien les algorithmes du draft comme montré sur le tableau 5.2.

Identifiant	Constante OpenSSL
SSL_CK_RC2_128_CBC_WITH_MD5	SSL2_CK_RC2_128_CBC_WITH_MD5
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	SSL2_CK_RC2_128_CBC_EXPORT40_WITH_MD5
SSL_CK_IDEA_128_CBC_WITH_MD5	SSL2_CK_IDEA_128_CBC_WITH_MD5
SSL_CK_DES_64_CBC_WITH_MD5	SSL2_CK_DES_64_CBC_WITH_MD5
SSL_CK_DES_192_EDE3_CBC_WITH_MD5	SSL2_CK_DES_192_EDE3_CBC_WITH_MD5
SSL_CK_RC4_128_WITH_MD5	SSL2_CK_RC4_128_WITH_MD5
SSL_CK_RC4_128_EXPORT40_WITH_MD5	SSL2_CK_RC4_128_EXPORT40_WITH_MD5

TABLE 5.2 – Algorithmes supportés par OpenSSL SSLv2

On y trouve également des constantes non définies dans le draft avec des commentaires très succincts :

- `SSL2_CK_NULL_WITH_MD5 /* v3 */`
- `SSL2_CK_DES_64_CBC_WITH_SHA /* v3 */`
- `SSL2_CK_DES_192_EDE3_CBC_WITH_SHA /* v3 */`
- `SSL2_CK_RC4_64_WITH_MD5 /* MS hack */`
- `SSL2_CK_DES_64_CFB64_WITH_MD5_1 /* SSLeay */`
- `SSL2_CK_NULL /* SSLeay */`

Les constantes commentées avec v3 sont présentes pour des raisons de rétro-compatibilité depuis SSL v3. Celles commentées par SSLeay sont des vestiges de l'ancêtre d'OpenSSL : SSLeay. Elles sont sûrement conservées pour la rétro-compatibilité avec des vieux logiciels utilisant SSLeay. La MS hack est spécifique à Windows.

## 5.2.2 Audit 5.2 : SSL version 3

### 5.2.2.1 Spécifications

La version 3 du protocole SSL est décrite dans la RFC 6101 [19]. On y trouve notamment en section A.6 la liste des algorithmes de chiffrement pouvant être utilisés avec cette version, référencés sur la tableau 5.3.

Identifiant	KeyExch	Authn	Enc	MAC
SSL_NULL_WITH_NULL_NULL	NULL	NULL	NULL	NULL
SSL_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5
SSL_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA1
SSL_RSA_EXPORT_WITH_RC4_40_MD5	RSAsign	RSAsign	RC4.40	MD5
SSL_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4.128	MD5
SSL_RSA_WITH_RC4_128_SHA	RSA	RSA	IDEA.128	SHA1
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSAsign	RSAsign	RC2.40 CBC	MD5
SSL_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA.128 CBC	SHA1
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	RSAsign	RSAsign	DES.40	SHA1
SSL_RSA_WITH_DES_CBC_SHA	RSA	RSA	DES.56 CBC	SHA1
SSL_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES.168 CBC	SHA1
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	DH	DSS	DES.40 CBC	SHA1
SSL_DH_DSS_WITH_DES_CBC_SHA	DH	DSS	DES.56 CBC	SHA1
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH	DSS	3DES.168 CBC	SHA1
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	DH	RSA	DES.40 CBC	SHA1
SSL_DH_RSA_WITH_DES_CBC_SHA	DH	RSA	DES.56 CBC	SHA1
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH	RSA	3DES.168 CBC	SHA1
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DHE.512	DSS	DES.40 CBC	SHA1
SSL_DHE_DSS_WITH_DES_CBC_SHA	DHE	DSS	DES.56 CBC	SHA1
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE	DSS	3DES.168 CBC	SHA1
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	DHE.512	RSA	DES.40CBC	SHA1
SSL_DHE_RSA_WITH_DES_CBC_SHA	DHE	RSA	DES.56 CBC	SHA1
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE	RSA	3DES.168 CBC	SHA1
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	DH.512	None	RC4.40	MD5
SSL_DH_anon_WITH_RC4_128_MD5	DH	None	RC4.128	MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	DH.512	None	DES.40 CBC	SHA1
SSL_DH_anon_WITH_DES_CBC_SHA	DH	None	DES.56 CBC	SHA1
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	DH	None	3DES.168 CBC	SHA1
SSL_FORTEZZA_KEA_WITH_NULL_SHA	FRTZA	KEA	None	SHA1
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	FRTZA	KEA	FRTZA	SHA1
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	FRTZA	KEA	RC4.128	SHA1

TABLE 5.3 – Algorithmes supportés SSLv3

### 5.2.2.2 Implémentation

Dans le code d'OpenSSL, cette version du protocole SSL se trouve dans les fichiers commençant par `s3_` du répertoire `ssl/`. Les constantes sont déclarées dans le fichier `ssl3.h`, on y retrouve les algorithmes de la RFC, référencés en table 5.4.

Identifiant	Constante OpenSSL
SSL_NULL_WITH_NULL_NULL	
SSL_RSA_WITH_NULL_MD5	SSL3_CK_RSA_NULL_MD5
SSL_RSA_WITH_NULL_SHA	SSL3_CK_RSA_NULL_SHA
SSL_RSA_EXPORT_WITH_RC4_40_MD5	SSL3_CK_RSA_RC4_40_MD5
SSL_RSA_WITH_RC4_128_MD5	SSL3_CK_RSA_RC4_128_MD5
SSL_RSA_WITH_RC4_128_SHA	SSL3_CK_RSA_RC4_128_SHA
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	SSL3_CK_RSA_RC2_40_MD5
SSL_RSA_WITH_IDEA_CBC_SHA	SSL3_CK_RSA_IDEA_128_SHA
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3_CK_RSA_DES_40_CBC_SHA
SSL_RSA_WITH_DES_CBC_SHA	SSL3_CK_RSA_DES_64_CBC_SHA
SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSL3_CK_RSA_DES_192_CBC3_SHA
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3_CK_DH_DSS_DES_40_CBC_SHA
SSL_DH_DSS_WITH_DES_CBC_SHA	SSL3_CK_DH_DSS_DES_64_CBC_SHA
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	SSL3_CK_DH_DSS_DES_192_CBC3_SHA
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3_CK_DH_RSA_DES_40_CBC_SHA
SSL_DH_RSA_WITH_DES_CBC_SHA	SSL3_CK_DH_RSA_DES_64_CBC_SHA
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	SSL3_CK_DH_RSA_DES_192_CBC3_SHA
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3_CK_DHE_DSS_DES_40_CBC_SHA
SSL_DHE_DSS_WITH_DES_CBC_SHA	SSL3_CK_DHE_DSS_DES_64_CBC_SHA
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	SSL3_CK_DHE_DSS_DES_192_CBC3_SHA
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3_CK_DHE_RSA_DES_40_CBC_SHA
SSL_DHE_RSA_WITH_DES_CBC_SHA	SSL3_CK_DHE_RSA_DES_64_CBC_SHA
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	SSL3_CK_DHE_RSA_DES_192_CBC3_SHA
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	SSL3_CK_ADH_RC4_40_MD5
SSL_DH_anon_WITH_RC4_128_MD5	SSL3_CK_ADH_RC4_128_MD5
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	SSL3_CK_ADH_DES_40_CBC_SHA
SSL_DH_anon_WITH_DES_CBC_SHA	SSL3_CK_ADH_DES_64_CBC_SHA
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	SSL3_CK_ADH_DES_192_CBC3_SHA
SSL_FORTEZZA_KEA_WITH_NULL_SHA	SSL3_CK_FZA_DMS_NULL_SHA
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	SSL3_CK_FZA_DMS_FZA_SHA
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	SSL3_CK_FZA_DMS_RC4_SHA

TABLE 5.4 – Algorithmes supportés par OpenSSL, SSLv3

**Attention** Les 3 algorithmes FORTEZZA sont commentés dans OpenSSL depuis le commit 89bbe14c506b9bd2fd00e6bae22a99ef1ee7ad19 de 2006.

**Remarque** OpenSSL déclare d'autres constantes pour utiliser SSL 3 avec Kerberos 5 :

- SSL3\_CK\_KRB5\_DES\_64\_CBC\_SHA

- SSL3\_CK\_KRB5\_DES\_192\_CBC3\_SHA
- SSL3\_CK\_KRB5\_RC4\_128\_SHA
- SSL3\_CK\_KRB5\_IDEA\_128\_CBC\_SHA
- SSL3\_CK\_KRB5\_DES\_64\_CBC\_MD5
- SSL3\_CK\_KRB5\_DES\_192\_CBC3\_MD5
- SSL3\_CK\_KRB5\_RC4\_128\_MD5
- SSL3\_CK\_KRB5\_IDEA\_128\_CBC\_MD5
- SSL3\_CK\_KRB5\_DES\_40\_CBC\_SHA
- SSL3\_CK\_KRB5\_RC2\_40\_CBC\_SHA
- SSL3\_CK\_KRB5\_RC4\_40\_SHA
- SSL3\_CK\_KRB5\_DES\_40\_CBC\_MD5
- SSL3\_CK\_KRB5\_RC2\_40\_CBC\_MD5
- SSL3\_CK\_KRB5\_RC4\_40\_MD5

### 5.2.2.3 Failles

#### 5.2.2.3.1 CVE-2013-4353

Cette faille découverte par Anton Johansson [24] permet de faire planter OpenSSL avec un déréférencement de pointeur NULL et peut ainsi causer des dénis de service. Cette faille a été corrigée le 7 janvier 2014 par Stephen Henson. Voici le patch de correction dans la fonction `ssl3_take_mac` du fichier `ssl/s3_both.c` :

```
diff --git a/ssl/s3_both.c b/ssl/s3_both.c
index 8de149a..0a259b1 100644
--- a/ssl/s3_both.c
+++ b/ssl/s3_both.c
@@ -203,7 +203,11 @@
 {
     const char *sender;
     int slen;

-    /* If no new cipher setup return immediately: other functions will
+    /* If no new cipher setup return immediately: other functions will
+     * set the appropriate error.
+     */
+    if (s->s3->tmp.new_cipher == NULL)
+        return;
     if (s->state & SSL_ST_CONNECT)
     {
         sender=s->method->ssl3_enc->server_finished_label;
```

Listing 5.1 – patch-cve-2013-4353

#### 5.2.2.3.2 Attaque sur le padding CBC de Serge Vaudenay

Cette attaque [56] fait partie de la famille des attaques par canaux auxiliaires et plus particulièrement des timing attacks. En effet, lorsqu'OpenSSL déchiffre en mode CBC, le temps varie en fonction de la longueur du message et du padding. Pour que l'attaque fonctionne, il faut utiliser un oracle de padding qui valide ou non le padding d'un chiffré.

Ben Laurie a appliqué un correctif du code OpenSSL le 28 janvier 2013 qui rend le décodage CBC constant, que le padding soit correct ou non. Le code a été placé dans une nouvelle fonction `tls1_cbc_remove_padding`

du fichier `ssl/s3_cbc.c`

### 5.2.2.3.3 CVE-2011-4576

Cette faille permettait de récupérer des données d'un déchiffrement précédent. En effet, le buffer n'était pas réinitialisé. Cette faille a été identifiée [28] et corrigée par Adam Langley le 4 janvier 2011, voici le patch appliqué :

```
----- ssl/s3_enc.c -----
2 diff --git a/ssl/s3_enc.c b/ssl/s3_enc.c
  index 0ddfe19..c5df2cb 100644
  --- a/ssl/s3_enc.c
  +++ b/ssl/s3_enc.c
  @@ -512,6 +512,9 @@
 
      /* we need to add 'i-1' padding bytes */
      l+=i;
10 +     /* the last of these zero bytes will be overwritten
11 +      * with the padding length. */
12 +     memset(&rec->input[rec->length], 0, i);
      rec->length+=i;
14     rec->input[l-1]=(i-1);
  }
```

Listing 5.2 – patch-cve-2011-4576

## 5.2.3 Audit 5.3 : TLS version 1

### 5.2.3.1 Spécifications

La version 1 de TLS est décrite dans la RFC 2246 [14]. Ce protocole est assez similaire à SSL 3 mais il y a quelques différences notables. Il y est notamment prévu un mécanisme de rétro-compatibilité vers SSL 3. La plus grosse différence est que TLS, contrairement à SSL, permet de commencer une connexion non chiffrée sur un port usuel tel que le 80 et bascule en mode chiffré avec la commande STARTTLS.

Du point de vue des algorithmes de chiffrement, on a globalement la même liste que SSL 3 et les algorithmes de Fortezza ont été retirés, comme on peut le constater sur le tableau 5.5.



Identifiant	KeyExch	Authn	Enc	MAC
TLS_NULL_WITH_NULL_NULL	NULL	NULL	NULL	NULL
TLS_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5
TLS_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA1
TLS_RSA_EXPORT_WITH_RC4_40_MD5	RSaex	RSaex	RC4.40	MD5
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4.128	MD5
TLS_RSA_WITH_RC4_128_SHA	RSA	RSA	IDEA.128	SHA1
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSaex	RSaex	RC2.40 CBC	MD5
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA.128 CBC	SHA1
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	RSaex	RSaex	DES.40	SHA1
TLS_RSA_WITH_DES_CBC_SHA	RSA	RSA	DES.56 CBC	SHA1
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES.168 CBC	SHA1
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	DH	DSS	DES.40 CBC	SHA1
TLS_DH_DSS_WITH_DES_CBC_SHA	DH	DSS	DES.56 CBC	SHA1
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DH	DSS	3DES.168 CBC	SHA1
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	DH	RSA	DES.40 CBC	SHA1
TLS_DH_RSA_WITH_DES_CBC_SHA	DH	RSA	DES.56 CBC	SHA1
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	DH	RSA	3DES.168 CBC	SHA1
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DHE.512	DSS	DES.40 CBC	SHA1
TLS_DHE_DSS_WITH_DES_CBC_SHA	DHE	DSS	DES.56 CBC	SHA1
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DHE	DSS	3DES.168 CBC	SHA1
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	DHE.512	RSA	DES.40CBC	SHA1
TLS_DHE_RSA_WITH_DES_CBC_SHA	DHE	RSA	DES.56 CBC	SHA1
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	DHE	RSA	3DES.168 CBC	SHA1
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	DH.512	None	RC4.40	MD5
TLS_DH_anon_WITH_RC4_128_MD5	DH	None	RC4.128	MD5
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	DH.512	None	DES.40 CBC	SHA1
TLS_DH_anon_WITH_DES_CBC_SHA	DH	None	DES.56 CBC	SHA1
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	DH	None	3DES.168 CBC	SHA1

TABLE 5.5 – Algorithmes supportés TLS1

**Algorithmes supplémentaires de la RFC 3268** La RFC prévoit la possibilité d'étendre cette liste. Ainsi, la RFC 3268 [9] apporte des nouvelles constantes avec AES et SHA-1, listés sur le tableau ??.

Identifiant	KeyExch	Authn	Enc	MAC
TLS_RSA_WITH_AES_128_CBC_SHA	RSA	RSA	AES 128 CBC	SHA1
TLS_DH_DSS_WITH_AES_128_CBC_SHA	DH	DSS	AES 128 CBC	SHA1
TLS_DH_RSA_WITH_AES_128_CBC_SHA	DH	RSA	AES 128 CBC	SHA1
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	DHE	DSS	AES 128 CBC	SHA1
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	DHE	RSA	AES 128 CBC	SHA1
TLS_DH_anon_WITH_AES_128_CBC_SHA	DH	NULL	AES 128 CBC	SHA1
TLS_RSA_WITH_AES_256_CBC_SHA	RSA	RSA	AES 256 CBC	SHA1
TLS_DH_DSS_WITH_AES_256_CBC_SHA	DH	DSS	AES 256 CBC	SHA1
TLS_DH_RSA_WITH_AES_256_CBC_SHA	DH	RSA	AES 256 CBC	SHA1
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	DHE	DSS	AES 256 CBC	SHA1
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	DHE	RSA	AES 256 CBC	SHA1
TLS_DH_anon_WITH_AES_256_CBC_SHA	DH	NULL	AES 256 CBC	SHA1

TABLE 5.6 – Algorithmes supplémentaires AES - RFC3268 TLS1

**Algorithmes supplémentaires de la RFC 4132** La RFC 4132 [35] apporte également une liste supplémentaire utilisant l'algorithme de chiffrement Camellia (cf. tableau 5.7).

Identifiant	KeyExch	Authn	Enc	MAC
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	RSA	RSA	Camellia 128 CBC	SHA1
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	DH	DSS	Camellia 128 CBC	SHA1
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	DH	RSA	Camellia 128 CBC	SHA1
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	DHE	DSS	Camellia 128 CBC	SHA1
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	DHE	RSA	Camellia 128 CBC	SHA1
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	DH	NULL	Camellia 128 CBC	SHA1
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	RSA	RSA	Camellia 256 CBC	SHA1
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	DH	DSS	Camellia 256 CBC	SHA1
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	DH	RSA	Camellia 256 CBC	SHA1
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	DHE	DSS	Camellia 256 CBC	SHA1
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	DHE	RSA	Camellia 256 CBC	SHA1
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	DH	NULL	Camellia 256 CBC	SHA1

TABLE 5.7 – Algorithmes supplémentaires Camellia - RFC3268 TLS1

**Algorithmes supplémentaires de la RFC 4162** La RFC 4162 [30] ajoute l'algorithme de chiffrement SEED, dont la liste est consultable sur le tableau 5.8.

Identifiant	KeyExch	Authn	Enc	MAC
TLS_RSA_WITH_SEED_CBC_SHA	RSA	RSA	SEED CBC	SHA1
TLS_DH_DSS_WITH_SEED_CBC_SHA	DH	DSS	SEED CBC	SHA1
TLS_DH_RSA_WITH_SEED_CBC_SHA	DH	RSA	SEED CBC	SHA1
TLS_DHE_DSS_WITH_SEED_CBC_SHA	DHE	DSS	SEED CBC	SHA1
TLS_DHE_RSA_WITH_SEED_CBC_SHA	DHE	RSA	SEED CBC	SHA1
TLS_DH_anon_WITH_SEED_CBC_SHA	DH	NULL	SEED CBC	SHA1

TABLE 5.8 – Algorithmes supplémentaires SEED - RFC3268 TLS1

**Extensions TLS** Des extensions TLS du hello client sont décrites dans les RFC 3546 [5], 4366 [6], 4492 [4] et 4507 [45] :

- **Server Name Indication** : permet d'indiquer au serveur quel est le nom du serveur qu'il demande, cela est utilisé lorsqu'il y a plusieurs hôtes virtuels sur une même machine ;
- **Maximum Fragment Length Negotiation** : sans cette extension, TLS spécifie une taille maximale fixe de fragment à  $2^{14}$  octets. Cette extension permet aux clients d'adapter cette taille en fonction des limites de mémoire ou de bande passante. Valeurs autorisées :  $2^9, 2^{10}, 2^{11}, 2^{12}$ . Si la valeur n'est pas dans cette liste, le serveur doit interrompre la poignée de main ;
- **Client Certificate URLs** : sans cette extension, TLS spécifie que lors de l'authentification client, ce dernier doit envoyer son certificat pendant la poignée de main. Grâce à cette extension, le client peut économiser de l'espace disque en stockant son certificat à une autre adresse ;
- **Trusted CA Indication** : indique les clefs de CA racines possède le client ;
- **Truncated HMAC** : permet de tronquer le code MAC à 10 octets pour économiser de la bande passante ;
- **Certificate Status Request** : indique que le client souhaite vérifier la validité du certificat du serveur avec une requête OCSP par exemple ;
- **Supported Elliptic Curves** : indique les courbes elliptiques supportées par le client ;
- **Session Ticket** : permet de rétablir une session précédente.

### 5.2.3.2 Implémentation

Pour les constantes représentant les algorithmes disponibles, tout est dans `ssl/tls1.h`. On y trouve également les algorithmes utilisant les courbes elliptiques décrits dans le draft <http://tools.ietf.org/html/draft-ietf-tls-ecc-12> :

```

1 #define TLS1_CK_ECDH_ECDSA_WITH_NULL_SHA 0x0300C001
2 #define TLS1_CK_ECDH_ECDSA_WITH_RC4_128_SHA 0x0300C002
3 #define TLS1_CK_ECDH_ECDSA_WITH_DES_192_CBC3_SHA 0x0300C003
4 #define TLS1_CK_ECDH_ECDSA_WITH_AES_128_CBC_SHA 0x0300C004
5 #define TLS1_CK_ECDH_ECDSA_WITH_AES_256_CBC_SHA 0x0300C005
6
7 #define TLS1_CK_ECDHE_ECDSA_WITH_NULL_SHA 0x0300C006
8 #define TLS1_CK_ECDHE_ECDSA_WITH_RC4_128_SHA 0x0300C007
9 #define TLS1_CK_ECDHE_ECDSA_WITH_DES_192_CBC3_SHA 0x0300C008
10 #define TLS1_CK_ECDHE_ECDSA_WITH_AES_128_CBC_SHA 0x0300C009
11 #define TLS1_CK_ECDHE_ECDSA_WITH_AES_256_CBC_SHA 0x0300C00A
12
13 #define TLS1_CK_ECDH_RSA_WITH_NULL_SHA 0x0300C00B

```

```

15 #define TLS1_CK_ECDH_RSA_WITH_RC4_128_SHA 0x0300C00C
#define TLS1_CK_ECDH_RSA_WITH_DES_192_CBC3_SHA 0x0300C00D
#define TLS1_CK_ECDH_RSA_WITH_AES_128_CBC_SHA 0x0300C00E
17 #define TLS1_CK_ECDH_RSA_WITH_AES_256_CBC_SHA 0x0300C00F

19 #define TLS1_CK_ECDHE_RSA_WITH_NULL_SHA 0x0300C010
#define TLS1_CK_ECDHE_RSA_WITH_RC4_128_SHA 0x0300C011
21 #define TLS1_CK_ECDHE_RSA_WITH_DES_192_CBC3_SHA 0x0300C012
#define TLS1_CK_ECDHE_RSA_WITH_AES_128_CBC_SHA 0x0300C013
23 #define TLS1_CK_ECDHE_RSA_WITH_AES_256_CBC_SHA 0x0300C014

25 #define TLS1_CK_ECDH_anon_WITH_NULL_SHA 0x0300C015
#define TLS1_CK_ECDH_anon_WITH_RC4_128_SHA 0x0300C016
27 #define TLS1_CK_ECDH_anon_WITH_DES_192_CBC3_SHA 0x0300C017
#define TLS1_CK_ECDH_anon_WITH_AES_128_CBC_SHA 0x0300C018
29 #define TLS1_CK_ECDH_anon_WITH_AES_256_CBC_SHA 0x0300C019

31 #define TLS1_CK_RSA_WITH_AES_128_SHA 0x0300002F
#define TLS1_CK_DH_DSS_WITH_AES_128_SHA 0x03000030
33 #define TLS1_CK_DH_RSA_WITH_AES_128_SHA 0x03000031
#define TLS1_CK_DHE_DSS_WITH_AES_128_SHA 0x03000032
35 #define TLS1_CK_DHE_RSA_WITH_AES_128_SHA 0x03000033
#define TLS1_CK_ADH_WITH_AES_128_SHA 0x03000034

37 #define TLS1_CK_RSA_WITH_AES_256_SHA 0x03000035
39 #define TLS1_CK_DH_DSS_WITH_AES_256_SHA 0x03000036
#define TLS1_CK_DH_RSA_WITH_AES_256_SHA 0x03000037
41 #define TLS1_CK_DHE_DSS_WITH_AES_256_SHA 0x03000038
#define TLS1_CK_DHE_RSA_WITH_AES_256_SHA 0x03000039
43 #define TLS1_CK_ADH_WITH_AES_256_SHA 0x0300003A

45 #define TLS1_CK_RSA_WITH_CAMELLIA_128_CBC_SHA 0x03000041
#define TLS1_CK_DH_DSS_WITH_CAMELLIA_128_CBC_SHA 0x03000042
47 #define TLS1_CK_DH_RSA_WITH_CAMELLIA_128_CBC_SHA 0x03000043
#define TLS1_CK_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA 0x03000044
49 #define TLS1_CK_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA 0x03000045
#define TLS1_CK_ADH_WITH_CAMELLIA_128_CBC_SHA 0x03000046

```

Listing 5.3 – constantes protocole TLS

Pour ce qui est de la poignée de main, OpenSSL utilise la même fonction que pour SSL 3 : `ssl3_connect` (`s3_clnt.c`)/`ssl3_accept` (`s3_srvr.c`) et la gestion des extensions se trouve dans la fonction `ssl_scan_clienthello_tlsext` (`tl1_lib.c`).

### 5.2.3.3 Failles

#### 5.2.3.3.1 Attaque sur le padding CBC de Serge Vaudenay

Voir 5.2.2.3.2

### 5.2.4 Audit 5.4 : TLS version 1.1

#### 5.2.4.1 Spécifications

La version 1.1 de TLS est spécifiée par la RFC 4346 [15]. Différences avec la version 1.0 :

Extension	Implémentée
Server Name Indication	oui (t1_lib.c)
Maximum Fragment Length Negotiation	non trouvée
Client Certificate URLs	non trouvée
Trusted CA Indication	non trouvée
Truncated HMAC	non trouvée
Certificate Status Request	oui (t1_lib.c)
Supported Elliptic Curves	oui (t1_lib.c)
Session Ticket	oui (t1_lib.c)

TABLE 5.9 – Gestion des extensions dans la poignée de main avec OpenSSL

Identifiant	KeyExch	Authn	Enc	MAC
TLS_SRP_SHA_WITH_3DES_EDE_CBC_SHA	SRP SHA1	SRP SHA1	3DES CBC	SHA1
TLS_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA	SRP SHA1	RSA	3DES CBC	SHA1
TLS_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA	SRP SHA1	DSS	3DES CBC	SHA1
TLS_SRP_SHA_WITH_AES_128_CBC_SHA	SRP SHA1	SRP SHA1	AES 128 CBC	SHA1
TLS_SRP_SHA_RSA_WITH_AES_128_CBC_SHA	SRP SHA1	RSA	AES 128 CBC	SHA1
TLS_SRP_SHA_DSS_WITH_AES_128_CBC_SHA	SRP SHA1	DSS	AES 128 CBC	SHA1
TLS_SRP_SHA_WITH_AES_256_CBC_SHA	SRP SHA1	SRP SHA1	AES 256 CBC	SHA1
TLS_SRP_SHA_RSA_WITH_AES_256_CBC_SHA	SRP SHA1	RSA	AES 256 CBC	SHA1
TLS_SRP_SHA_DSS_WITH_AES_256_CBC_SHA	SRP SHA1	DSS	AES 256 CBC	SHA1

TABLE 5.10 – Algorithmes supplémentaires RFC 5054 TLSv1.1

- l'IV implicite est remplacé par une IV explicite (protection contre les attaques sur CBC : <http://www.openssl.org/~bodo/tls-cbc.txt>);
- utilisation de l'alerte `bad_record_mac` plutôt que `decryption_failed` lors des erreurs de padding;
- les sessions fermées prématurément peuvent être reprises.

**Algorithmes supplémentaires de la RFC 5054** La RFC 5054 [51] ajoute l'échange de clef/authentification SRP, listé sur le tableau 5.10

**Extensions TLS** Une extension TLS du hello client est décrite dans la RFC 5054 :

- **srp** : permet d'indiquer le support d'algorithmes d'échange de clefs/authentification SRP ;

#### 5.2.4.2 Implémentation

On retrouve les constantes dans `ssl/tls1.h` :

```
/* SRP ciphersuites from RFC 5054 */
#define TLS1_CK_SRP_SHA_WITH_3DES_EDE_CBC_SHA 0x0300C01A
#define TLS1_CK_SRP_SHA_RSA_WITH_3DES_EDE_CBC_SHA 0x0300C01B
#define TLS1_CK_SRP_SHA_DSS_WITH_3DES_EDE_CBC_SHA 0x0300C01C
#define TLS1_CK_SRP_SHA_WITH_AES_128_CBC_SHA 0x0300C01D
#define TLS1_CK_SRP_SHA_RSA_WITH_AES_128_CBC_SHA 0x0300C01E
```

```
#define TLS1_CK_SRP_SHA_DSS_WITH_AES_128_CBC_SHA    0x0300C01F
#define TLS1_CK_SRP_SHA_WITH_AES_256_CBC_SHA        0x0300C020
#define TLS1_CK_SRP_SHA_RSA_WITH_AES_256_CBC_SHA    0x0300C021
#define TLS1_CK_SRP_SHA_DSS_WITH_AES_256_CBC_SHA    0x0300C022
```

### 5.2.4.3 Failles

#### 5.2.4.3.1 CVE-2012-2333

Un *integer underflow* permettait de causer un déni de service dans les protocoles TLS 1.1, 1.2 et DTLS. Cette faille [10] a été publiée le 10 mai 2012 par Codenomicon et affecte les versions d'OpenSSL suivantes :

- < 0.9.8x
- 1.0.0x < 1.0.0j
- 1.0.1x < 1.0.0c

Cette faille a été trouvée grâce à l'outil Fuzz-o-Matic développé par Codenomicon.

La correction suivante a été appliquée le même jour par l'équipe d'OpenSSL :

```
diff -u ssl/tl_enc.c
index 201ca9a..f7bdeb3 100644
--- a/ssl/tl_enc.c
+++ b/ssl/tl_enc.c
@@ -889,6 +889,8 @@
     if (s->version >= TLS1_1_VERSION
        && EVP_CIPHER_CTX_mode(ds) == EVP_CIPH_CBC_MODE)
     {
+        if (bs > (int)rec->length)
+            return -1;
         rec->data += bs;    /* skip the explicit IV */
         rec->input += bs;
         rec->length -= bs;
```

Ici, la variable **bs** est la taille de bloc utilisée dans l'algorithme de chiffrement. On constate que, sans vérification, il était possible de déplacer les pointeurs **rec->data**, **rec->input** et **rec->length** au-delà de la longueur du paquet et ainsi créer une erreur de segmentation.

#### 5.2.4.3.2 Lucky Thirteen CVE-2013-0169

Cette attaque [37] a été trouvée par Nadhem Alfaridan et Kenny Paterson de l'Université de Londres le 5 février 2005 et a été corrigée par Adam Langley et Emilia Kasper. Versions touchées :

- ≤ 1.0.1c
- ≤ 1.0.0j
- 0.9.8x

Cette attaque reprend le principe de l'attaque de Vaudenay (5.2.2.3.2).

Lorsqu'un message chiffré incorrect est reçu, un message d'erreur fatale est renvoyé vers l'expéditeur. Cependant, le temps de génération de ce message d'erreur dépend du nombre d'octets valides, utilisé par un haché MAC. Un attaquant peut donc injecter des messages chiffrés erronés dans une session TLS/DTLS en mode CBC, et mesurer le temps nécessaire à la génération du message d'erreur, afin de progressivement



déterminer le contenu en clair de la session.

Il faut  $2^{23}$  sessions TLS pour retrouver un bloc en clair. Pour mener l'attaque, le client TLS doit alors continuer en permanence à ouvrir une nouvelle session, dès que la précédente s'est terminée en erreur fatale.

## 5.2.5 Audit 5.5 : TLS version 1.2

### 5.2.5.1 Spécifications

La version 1.2 de TLS est décrite dans la RFC 5246 [16].

**Algorithmes supplémentaires de la RFC 5289** La RFC 5289 [44] ajoute les codes MAC SHA256 et SHA384 ainsi que le mode GCM (Galois Counter Mode), rapportés dans le tableau 5.11.

Identifiant	KeyExch	Authn	Enc	MAC
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	ECDHE	ECDSA	AES 128 CBC	SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	ECDHE	ECDSA	AES 256 CBC	SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	ECDH	ECDSA	AES 128 CBC	SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	ECDH	ECDSA	AES 256 CBC	SHA256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	ECDHE	RSA	AES 128 CBC	SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	ECDHE	RSA	AES 256 CBC	SHA384
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	ECDH	RSA	AES 128 CBC	SHA256
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	ECDH	RSA	AES 256 CBC	SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	ECDHE	ECDSA	AES 128 GCM	SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	ECDHE	ECDSA	AES 256 GCM	SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	ECDH	ECDSA	AES 128 GCM	SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	ECDH	ECDSA	AES 256 GCM	SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	ECDHE	RSA	AES 128 GCM	SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	ECDHE	RSA	AES 256 GCM	SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	ECDH	RSA	AES 128 GCM	SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	ECDH	RSA	AES 256 GCM	SHA384

TABLE 5.11 – Algorithmes supplémentaires - RFC 5289 TLSv1.2

### 5.2.5.2 Implémentation

On retrouve les constantes dans `ssl/tls1.h` :

```

2  /* ECDH HMAC based ciphersuites from RFC5289 */
3  #define TLS1 CK_ECDHE_ECDSA_WITH_AES_128_SHA256      0x0300C023
4  #define TLS1 CK_ECDHE_ECDSA_WITH_AES_256_SHA384      0x0300C024
5  #define TLS1 CK_ECDH_ECDSA_WITH_AES_128_SHA256      0x0300C025
6  #define TLS1 CK_ECDH_ECDSA_WITH_AES_256_SHA384      0x0300C026
7  #define TLS1 CK_ECDHE_RSA_WITH_AES_128_SHA256      0x0300C027
8  #define TLS1 CK_ECDHE_RSA_WITH_AES_256_SHA384      0x0300C028
9  #define TLS1 CK_ECDH_RSA_WITH_AES_128_SHA256      0x0300C029
10 #define TLS1 CK_ECDH_RSA_WITH_AES_256_SHA384      0x0300C02A
11
12 /* ECDH GCM based ciphersuites from RFC5289 */
13 #define TLS1 CK_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 0x0300C02B

```

```
14 #define TLS1_CK_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 0x0300C02D
    #define TLS1_CK_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 0x0300C02E
16 #define TLS1_CK_ECDHE_RSA_WITH_AES_128_GCM_SHA256 0x0300C02F
    #define TLS1_CK_ECDHE_RSA_WITH_AES_256_GCM_SHA384 0x0300C030
18 #define TLS1_CK_ECDH_RSA_WITH_AES_128_GCM_SHA256 0x0300C031
    #define TLS1_CK_ECDH_RSA_WITH_AES_256_GCM_SHA384 0x0300C032
```

Listing 5.4 – Constantes TLS 1.2

### 5.2.5.3 Failles

#### 5.2.5.3.1 CVE-2012-2333

Voir 5.2.4.3.1

#### 5.2.5.3.2 CVE-2013-6449

Cette faille a été découverte en novembre 2013 et a été corrigée le 19 décembre 2013 par l'équipe d'OpenSSL. Elle affecte les versions < 1.0.2. Elle consiste à créer un déni de service du serveur en envoyant une structure de données mal formée. Le crash se situe dans la fonction `ssl_get_algorithm2` du fichier `ssl/s3_lib.c`. Le patch correctif :

```
1 index bf832bb..c4ef273 100644 (file)
   --- a/ssl/s3_lib.c
   +++ b/ssl/s3_lib.c
   @@ -4286,7 +4286,7 @@ need to go to SSL_ST_ACCEPT.
    long ssl_get_algorithm2(SSL *s)
    {
    7     long alg2 = s->s3->tmp.new_cipher->algorithm2;
   -     if (TLS1_get_version(s) >= TLS1_2_VERSION &&
    9 +     if (s->method->version == TLS1_2_VERSION &&
        alg2 == (SSL_HANDSHAKE_MAC_DEFAULT|TLS1_PRF))
    11         return SSL_HANDSHAKE_MAC_SHA256 | TLS1_PRF_SHA256;
        return alg2;
```

Listing 5.5 – patch-cve-2013-6449



## Chapitre 6

# Observations et alternatives

### 6.1 Code OpenSSL

#### 6.1.1 Documentation

Nos observations sur la documentation d'OpenSSL va se développer sur trois axes : dans le code, le manuel et les commentaires git.

La documentation d'OpenSSL est quasi inexistante dans le code lui-même. Ce problème vient principalement du fait que le code était originellement développé par Eric A. Young et Tim Hudson seulement, et ce pendant environ 6 ans. Puis ces deux développeurs ont rejoint RSA Security en 1998, léguant le code à la communauté pour devenir OpenSSL. Ce code était très peu commenté au vu des premiers commits d'OpenSSL.

Actuellement, OpenSSL est principalement en développement réactif, c'est-à-dire qu'il y a surtout des corrections ou des implémentations de nouveaux standards. En revanche, les corrections de failles sont souvent accompagnées de commentaires dans le code et cela est très appréciable.

Au-delà de la documentation dans le code, le manuel d'utilisation n'est pas très bien fourni non plus, regroupant très souvent plusieurs fonctions dans une même entrée *man* et oubliant certaines fonctions. Un exemple très intéressant est la vérification du pair connecté. En effet, lorsqu'on fait un `SSL_accept`, il est important de vérifier l'identité du pair connecté avec la commande la fonction `SSL_set_verify_result`, cela est assez connu lorsqu'on lit des recommandations en ligne. Or, le manuel de `SSL_accept` n'en fait pas mention et tout novice de la libssl peut se limiter à cette fonction, croyant que la connexion est sûre.

Une autre source de documentation est les commentaires des commits git. Ces derniers sont en général assez clairs et bien développés lorsque cela est nécessaire.

#### 6.1.2 Lisibilité du code

Au niveau de la lisibilité du code, notre analyse va se porter sur : l'architecture du projet, les conventions de nommage, le *coding style* et l'utilisation des constantes et des macros C.

Pour ce qui est de l'architecture, bien que le code soit assez conséquent (plus de 450 000 lignes de code dont 66% de C – <http://www.ohloh.net/p/openssl>), il est relativement bien organisé, notamment les

dossiers `crypto/` et `ssl/`. On accède vite au fichier recherché grâce à des noms de dossiers assez explicites et une convention de nommage des fichiers.

En effet, les noms de fichiers sont souvent précédés d'un indicateur de quelques caractères qui permettent d'identifier rapidement de quel morceau un fichier fait partie. Par exemple, les fichiers commençant par `s3_` sont des fichiers `.c` relatifs à SSL version 3. On retrouve un peu la même logique dans le nommage des fonctions. Ainsi pour la partie chiffrement, on aura un préfixe `EVP_`, pour SSL version 3 `SSL3_` etc. Il arrive de trouver quelques incohérences comme dans le fichier `s3_cbc.c` qui gère donc le mode CBC de SSL version 3, on retrouve la fonction `tls1_cbc_remove_padding` ou `tls1_md5_final_raw` alors qu'il aurait fallu créer un nouveau fichier `t1_cbc.c` ou changer le nom du fichier pour regrouper plusieurs protocoles.

Pour ce qui est du *coding style*, il se rapproche beaucoup du *Whitesmiths* mais peut être s'agit-il d'une variante. Ce style est peut être moins clair que celui de GNU, du Kernel Linux ou même du K&R mais ce choix reste à l'appréciation des mainteneurs du projet.

Les constantes sont assez largement utilisées et déclarées massivement dans les *headers* C en code hexadécimal. Cela dit, on retrouve beaucoup de tests comparant octet par octet, avec des valeurs hexadécimales écrites en dur dans le code. Un bon exemple de ce mauvais code est dans le fichier `s23_srvr.c` qui gère la rétro-compatibilité des versions de SSL/TLS. On y retrouve la ligne

```
|| n=((p[0]&0x7f)<<8)|p[1];
```

Listing 6.1 – Exemple de mauvais code d'OpenSSL

où un commentaire, une constante avec un nom explicite pour `0x7f` et même des espaces auraient été largement appréciés. Dans ce même fichier, des comparaisons d'octets sont faites également sans constante avec un nom explicite comme ce test :

```
1  /*
2  * SSLv2 header
3  */
4  if ((p[3] == 0x00) && (p[4] == 0x02))
```

Listing 6.2 – Exemple de mauvais test d'OpenSSL

Ici un petit commentaire nous met sur la voie mais le code suivant aurait été plus clair :

```
1  /*
2  * SSLv2 header
3  */
4  if ((p[3] == SSL2_VERSION_MAJOR) && (p[4] == SSL2_VERSION_MINOR))
```

Listing 6.3 – Exemple de mauvais test d'OpenSSL corrigé

Ces constantes existent dans le *header* `ssl2.h` mais n'ont pourtant pas été utilisées.

Du côté des macros C, on regrettera les nombreux morceaux de code mis en commentaire avec des macros :

```
1  #if 0
2      SSLerr(SSL_F_SSL23_GET_CLIENT_HELLO, SSL_R_RECORD_TOO_SMALL);
3      goto err;
4  #else
```

```
|| v[1] = TLS1_VERSION_MINOR;  
| #endif
```

Listing 6.4 – Exemple de mauvais `#if 0` d'OpenSSL

Il y en a énormément et cela complique beaucoup la lecture du code car on croit parfois être dans du code alors que ce code est commenté. Certains morceaux de code sont enlevés à cause de failles mais ils restent présents pour une raison que nous ignorons.

## 6.2 Construction du projet

La construction d'OpenSSL se fait avec *autotools* qui permet de générer les makefile nécessaires à la compilation. Il existe une page wiki officielle à ce sujet mais elle n'est pas toujours très précise. Ils n'expliquent d'ailleurs pas très bien comment compiler les bibliothèques `libssl.so` et `libcrypto.so`. Il faut en fait spécifier les options suivantes :

```
$ ./config shared zlib-dynamic  
$ make
```

On regrettera ce type de construction qui vieillit. De nos jours, beaucoup de projets migrent vers *cmake* (comme MySQL par exemple) qui est plus simple à configurer et génèrent des sorties claires et colorées qui améliore le suivi de la compilation. De plus, il permet de compiler les binaires dans un dossier séparé, ce qui permet de ne pas perturber l'arborescence de travail qui ne contient que les sources.

## 6.3 Alternatives

Il serait intéressant de se pencher sur la sécurité des alternatives à OpenSSL. Voici une liste non exhaustive de projet open source :

- cryptlib (ne gère pas DTLS) ;
- CyaSSL ;
- GnuTLS ;
- MatrixSSL ;
- Network Security Services (utilisé dans Firefox, successeur du code développé par Netscape, ne gère pas DTLS) ;
- PolarSSL (ne gère pas DTLS) ;
- Java Secure Socket Extension (ne gère pas DTLS).

Chaque alternative a ses points forts et ses points faibles, il vaut mieux alors choisir le bon outil selon le travail à effectuer.

Il semble que OpenSSL soit un peu au dessus du lot en terme de support d'algorithmes et de méthodes en tout genre mais cela s'explique par l'âge et la taille du code.

En revanche, les autres se démarquent en désactivant des méthodes dépréciées alors qu'OpenSSL souhaite sans doute garder une rétro-compatibilité.

## Conclusion

Cette étude fut l'occasion d'effectuer une première approche d'audit en analysant de façon générale des failles liées à un des programmes majeurs dans le domaine de la cryptographie, OpenSSL. Nous avons pu déterminer une stratégie globale d'audit suivant les exigences et suggestions du client, puis nous avons effectué un approfondissement dans plusieurs domaines, répartis de façon modulaire entre les membres du projet.

Nous avons pu notamment constater que les failles identifiées sont principalement sur des anciennes versions d'OpenSSL et qu'elles sont, par conséquent, pour la plupart aujourd'hui corrigées. Toutefois, un utilisateur du logiciel possédant une version non à jour est vulnérable -suivant sa version- à des attaques référencées dans ce rapport.

L'audit de deux semaines nous a permis d'établir un état général sur les faiblesses de ce logiciel. Nous aurions aimé avoir davantage de temps pour entrer au plus près du code. Comme nous l'avons rappelé plusieurs fois, le projet OpenSSL n'a pas un code facilement lisible, et le manque de documentation nous a considérablement freiné dans nos analyses. De plus, le code en lui-même est de taille considérable, avec plus de 400 000 lignes de code.

De ce fait, nous considérons que le projet OpenSSL mériterait un rafraîchissement complet, en clarifiant à la fois la structure et l'implémentation, mais aussi en ajoutant une documentation digne de ce nom, afin d'en faciliter la compréhension et d'éviter au maximum toute mauvaise utilisation.

# Bibliographie

- [1] 3RD, D. E., SCHILLER, J., AND CROCKER, S. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005.
- [2] AMAURY, AND MICHEL, B. Debian : Découverte d'une faille de sécurité critique dans openssl de debian. <http://linuxfr.org/news/d%C3%A9couverte-dune-faille-de-s%C3%A9curit%C3%A9-critique-dans-openssl-de-deb>, Mai 2008.
- [3] ARCHLINUX. Random number generation. [https://wiki.archlinux.org/index.php/Random\\_Number\\_Generation](https://wiki.archlinux.org/index.php/Random_Number_Generation).
- [4] BLAKE-WILSON, S., BOLYARD, N., GUPTA, V., HAWK, C., AND MOELLER, B. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFCs 5246, 7027.
- [5] BLAKE-WILSON, S., NYSTROM, M., HOPWOOD, D., MIKKELSEN, J., AND WRIGHT, T. Transport Layer Security (TLS) Extensions. RFC 3546 (Proposed Standard), June 2003. Obsoleted by RFC 4366.
- [6] BLAKE-WILSON, S., NYSTROM, M., HOPWOOD, D., MIKKELSEN, J., AND WRIGHT, T. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), Apr. 2006. Obsoleted by RFCs 5246, 6066, updated by RFC 5746.
- [7] BOBLET, M., AND WONG, E. Openssl prng is not (really) fork-safe. <http://emboss.github.io/blog/2013/08/21/openssl-prng-is-not-really-fork-safe/>, Août 2013.
- [8] BUCKSH, B., KUEBART, J., MAXWELL, G., AND THOR-LANCELOT, S. Netbsd security advisory 2013-003. <http://ftp.netbsd.org/pub/NetBSD/security/advisories/NetBSD-SA2013-003.txt.asc>, Mars 2013.
- [9] CHOWN, P. Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS). RFC 3268 (Proposed Standard), June 2002. Obsoleted by RFC 5246.
- [10] CODEMICON. Cve-2012-2333. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2333>, 2012.
- [11] DEBIAN-SECURITY. Bulletin d'alerte debian : Générateur de nombres aléatoires prévisibles. <https://www.debian.org/security/2008/dsa-1571>.
- [12] DEBIAN-SECURITY. dowkd.pl. <http://security.debian.org/project/extra/dowkd/dowkd.pl.gz>.
- [13] DEBIAN-SECURITY. openssl-blacklist.deb. <https://packages.debian.org/fr/sid/openssl-blacklist>.
- [14] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.
- [15] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), Apr. 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176.
- [16] DIERKS, T., AND RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.
- [17] DUCKLIN, P. The openssl software bug that saves you from surveillance! <http://nakedsecurity.sophos.com/2013/12/22/the-openssl-software-bug-that-saves-you-from-surveillance/>, Décembre 2013.
- [18] 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.

- [19] FREIER, A., KARLTON, P., AND KOCHER, P. The Secure Sockets Layer (SSL) Protocol Version 3.0. RFC 6101 (Historic), Aug. 2011.
- [20] GOODIN, D. Severe openssl vuln busts public key crypto. [http://www.theregister.co.uk/2010/03/04/severe\\_openssl\\_vulnerability/](http://www.theregister.co.uk/2010/03/04/severe_openssl_vulnerability/), Mars 2010.
- [21] GREEN, M. On the nsa. <http://blog.cryptographyengineering.com/2013/09/on-nsa.html>, Septembre 2013.
- [22] GUNDERSON, S. H. Some maths. [http://plog.sesse.net/blog/tech/2008-05-14-17-21\\_some\\_maths.html](http://plog.sesse.net/blog/tech/2008-05-14-17-21_some_maths.html), Mai 2008.
- [23] HICKMAN, K. E. The ssl protocol. <http://tools.ietf.org/html/draft-hickman-netscape-ssl-00>, 1995.
- [24] JOHANSSON, A. Cve-2013-4353. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-4353>, 2013.
- [25] JONSSON, J., AND KALISKI, B. Public-Key Cryptography Standards (PKCS) #1 : RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational), Feb. 2003.
- [26] KELSEY, J., AND BARKER, E. Recommendation for random number generation using deterministic random bit generators. <http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf>, Janvier 2012.
- [27] KLYUBIN, A. Some securerandom thoughts. <http://android-developers.blogspot.de/2013/08/some-securerandom-thoughts.html>, Août 2013.
- [28] LANGLEY, A. Cve-2011-4576. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4576>, 2011.
- [29] LAWSON, N. Advances in rsa fault attacks. <http://rdist.root.org/2008/03/10/advances-in-rsa-fault-attacks/>, Mars 2008.
- [30] LEE, H., YOON, J., AND LEE, J. Addition of SEED Cipher Suites to Transport Layer Security (TLS). RFC 4162 (Proposed Standard), Aug. 2005.
- [31] MANDALIA, R. Report : Nsa paid rsa \$10m to use dual ec drbg as preferred random number generator. <http://www.techienews.co.uk/973955/report-nsa-paid-rsa-10m-use-dual-ec-drbg-preferred-random-number-generator/>, Décembre 2013.
- [32] MENN, J. Snowden disclosures prompt warning on widely used computer security formula. <http://www.reuters.com/article/2013/09/23/us-usa-security-snowden-rsa-idUSBRE98M06Q20130923>, Septembre 2013.
- [33] MICHAELIS, K., MEYER, C., AND SCHWENK, J. Randomly failed! the state of randomness in current java implementations. [http://www.nds.rub.de/media/nds/veroeffentlichungen/2013/03/25/paper\\_2.pdf](http://www.nds.rub.de/media/nds/veroeffentlichungen/2013/03/25/paper_2.pdf), 2013.
- [34] MIROS. Miros manual : arandom, prandom, random, srandom, urandom, wrandom. <https://www.mirbsd.org/htman/i386/man4/arandom.htm>, Octobre 2013.
- [35] MORIAI, S., KATO, A., AND KANDA, M. Addition of Camellia Cipher Suites to Transport Layer Security (TLS). RFC 4132 (Proposed Standard), July 2005. Obsoleted by RFC 5932.
- [36] MUELLER, S. Entropy generator with 100 kb/s throughput. <http://lkm1.iu.edu/hypermail/linux/kernel/1302.1/00479.html>, Février 2013.
- [37] NADHEM ALFARDAN, K. P. Cve-2013-0169. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0169>, 2013.
- [38] NETBSD. Diff for /src/sys/kern/subr\_cprng.c between version 1.14 and 1.15. [http://cvswb.netbsd.org/bdweb.cgi/src/sys/kern/subr\\_cprng.c.diff?r1=1.14&r2=1.15&only\\_with\\_tag=MAIN&f=h](http://cvswb.netbsd.org/bdweb.cgi/src/sys/kern/subr_cprng.c.diff?r1=1.14&r2=1.15&only_with_tag=MAIN&f=h), Janvier 2013.
- [39] NIST. Nist opens draft special publication 800-90a, recommendation for random number generation using deterministic random bit generators, for review and comment. [http://csrc.nist.gov/publications/nistbul/it1bul2013\\_09\\_supplemental.pdf](http://csrc.nist.gov/publications/nistbul/it1bul2013_09_supplemental.pdf), Septembre 2013.
- [40] NSA-TOP-SECRET. Ts//si//nf. <http://s3.documentcloud.org/documents/784159/sigintenabling-clean-1.pdf>, 2013.

- [41] PATRICK. Journal faille de sécurité critique dans le générateur pseudo-aléatoire de netbsd 6.0. [http://linuxfr.org/users/patrick\\_g/journaux/faille-de-securite-critique-dans-le-generateur-pseudo-aleatoire-de-netbsd-6-0](http://linuxfr.org/users/patrick_g/journaux/faille-de-securite-critique-dans-le-generateur-pseudo-aleatoire-de-netbsd-6-0), Mars 2013.
- [42] PELLEGRINI, A., BERTACCO, V., AND AUSTIN, T. Fault-based attack of rsa authentication. <http://web.eecs.umich.edu/~valeria/research/publications/DATE10RSA.pdf>, 2010.
- [43] RESCORLA, E. Diffie-Hellman Key Agreement Method. RFC 2631 (Proposed Standard), June 1999.
- [44] RESCORLA, E. TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM). RFC 5289 (Informational), Aug. 2008.
- [45] SALOWEY, J., ZHOU, H., ERONEN, P., AND TSCHOFENIG, H. Transport Layer Security (TLS) Session Resumption without Server-Side State. RFC 4507 (Proposed Standard), May 2006. Obsoleted by RFC 5077.
- [46] SCHNEIER, B. Did nsa put a secret backdoor in new encryption standard? [http://www.wired.com/politics/security/commentary/securitymatters/2007/11/securitymatters\\_1115](http://www.wired.com/politics/security/commentary/securitymatters/2007/11/securitymatters_1115), Novembre 2007.
- [47] SCHNEIER, B. Defending against crypto backdoors. [https://www.schneier.com/blog/archives/2013/10/defending\\_again\\_1.html](https://www.schneier.com/blog/archives/2013/10/defending_again_1.html), Octobre 2013.
- [48] SCHOENMAKERS, B., AND SIDORENKO, A. Cryptanalysis of the dual elliptic curve pseudorandom generator. <http://s3.documentcloud.org/documents/786216/cryptanalysis-of-the-dual-elliptic-curve.pdf>, Mai 2006.
- [49] SECURITY, O. Openssl security advisory - cve-2008-5077. [https://www.openssl.org/news/secadv\\_20090107.txt](https://www.openssl.org/news/secadv_20090107.txt), Janvier 2009.
- [50] SHUMOW, D., AND FERGUSON, N. On the possibility of a back door in the nist sp800-90 dual ec prng. <http://rump2007.cr.yp.to/15-shumow.pdf>, 2007.
- [51] TAYLOR, D., WU, T., MAVROGIANNOPOULOS, N., AND PERRIN, T. Using the Secure Remote Password (SRP) Protocol for TLS Authentication. RFC 5054 (Informational), Nov. 2007.
- [52] TURNER, S., AND POLK, T. Prohibiting Secure Sockets Layer (SSL) Version 2.0. RFC 6176 (Proposed Standard), Mar. 2011.
- [53] US-CERT/NIST. Vulnerability details : Cve-2009-0021. <http://www.cvedetails.com/cve/CVE-2009-0021/>, Janvier 2009.
- [54] US-CERT/NIST. Vulnerability summary for cve-2008-5077. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-5077>, Juillet 2009.
- [55] U.S.\_DEPARTMENT\_OF\_COMMERCE. Fips publication 140-2 : Security requirements for cryptographic modules. <http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf>, Mai 2001.
- [56] VAUDENAY, S., 2002.
- [57] WARNER, B. Egd : The entropy gathering daemon. <http://egd.sourceforge.net/>, Juillet 2002.
- [58] WIKIPEDIA. Cryptgenrandom. <http://en.wikipedia.org/wiki/CryptGenRandom>.
- [59] WIKIPEDIA. Révélations d'edward snowden. [http://fr.wikipedia.org/wiki/R%C3%A9v%C3%A9lations\\_d%27Edward\\_Snowden](http://fr.wikipedia.org/wiki/R%C3%A9v%C3%A9lations_d%27Edward_Snowden), 2013-2014.