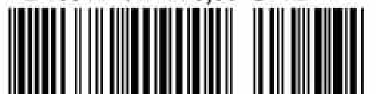
100 % SÉCURITÉ INFORMATIQUE

MISC

Multi-System & Internet Security Cookbook

HORS - SERIE

France METRO : 9 €
DOM : 9 €
CAN : 13,50 $CAD
CH : 15 CHF
BEL : 9,90 €
POL/S : 1100 CFP
POL/A : 1400 CFP



NOVEMBRE DÉCEMBRE 2012 N°6

HISTOIRE

- Mettre de la crypto dans des cartes à puce : toute une histoire
- Un logarithme (trop) discret ?

CODAGE

- Les pièges à éviter quand on programme
- SSL/TLS : un standard qui a aussi ses problèmes

PROGRÈS

- Au cœur du concours qui désignera SHA-3
- Protection contre les fuites par canaux auxiliaires

LES MAINS DANS LA CRYPTOGRAPHIE

DE LA THÉORIE...

...À LA PRATIQUE

APPLICATIONS

- Séquestre de clés : mise en œuvre du partage de secret de Shamir
- Premiers pas en reverse de crypto

CRYPTANALYSES PRATIQUES

- La cryptographie dans RDP : RSA, anecdotes et erreurs d'implémentation
- Mécanisme de contrôle d'authenticité des photographies numériques

CLOUD & IT EXPO

Le salon du Cloud Computing, des Datacenters
et des infrastructures sécurisées

2ÈME ÉDITION

23-24-25 OCTOBRE 2012

PARIS – PORTE DE VERSAILLES

TENUE CONJOINTE

POUR TOUTE DEMANDE D'INFO :
TÉL. : +33 (0)1 41 18 60 63
INFO@CLOUD-AND-IT-EXPO.FR

DEMANDEZ VOTRE BADGE
D'ACCÈS GRATUIT SUR
WWW.CLOUD-AND-IT-EXPO.FR

Un événement

Partenaire officiel

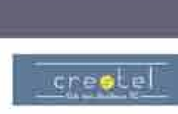
Gold sponsor

Silver sponsors

Partenaire officiel

Partenaires études

Partenaires Institutionnels



OW2 Consortium

Partenaire trophées

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

ÉDITO

Plus c'est long, plus c'est bon

Chers lecteurs et chères lectrices, je vous présente toutes mes excuses. Je réalise combien l'attente a été longue depuis que vous avez, fébriles et émoustillés, tenu entre vos mains le premier volet cryptographique sous vos yeux embués de joie.

Le suspense atteint enfin son paroxysme, son moment clé, et comme le disent Rivest, Shamir et Adelman à ce propos, plus c'est long, plus c'est bon. Je parle évidemment aussi bien des clés que des files d'attente (aussi appelées « queues »), comme celle que vous avez dû subir pour obtenir ce HS06.

Après avoir délesté votre bourse, vous l'avez enfin, et vous pouvez poursuivre avec ce second opus dans le monde fascinant de la cryptographie pour réaliser qu'à l'évidence, la cryptographie n'est pas qu'une histoire de *bits*. Par rapport au précédent, ce numéro porte plus sur « la pratique ». Certes, le lecteur attentif trouvera bien quelques équations au coin d'une page, mais nous avons surtout essayé de montrer qu'entre la théorie et la pratique, ça tient à plus qu'un poil de bit !

Il y a longtemps, la crypto se faisait avec du papyrus, un morceau de bois, une lanière de cuir, bref, trois fois rien, juste de quoi satisfaire quelques sadomasos. De nombreux algorithmes comme César, Polybe ou Vigenère, aujourd'hui complètement désuets et que n'importe quel étudiant sait résoudre de tête (ou presque) passaient pour inviolables à l'époque. Une simple feuille de papier suffisait pour calculer une analyse fréquentielle et casser ces algo. Puis vint la crypto mécanique, fameuse avec son Enigma et les premiers attouchements inwformatiques avec les « bombes » (ne pas les confondre avec de jolies filles, mais les « bomba kryptologiczna » construites en 1936 par les Polonais pour triompher d'Enigma).

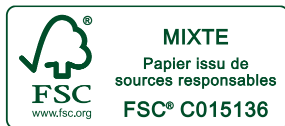
L'informatique a révolutionné pas mal de choses. Maintenant, personne ne peut calculer de tête un AES, un RSA, ou caresser un SHA. Alors quant à cryptanalyser... Encore plus aujourd'hui qu'hier, et contrairement à une idée répandue, la crypto n'assure pas plus de sécurité... quand elle est mal mise en œuvre (au contraire). C'est pour cela qu'il faut prendre ses précautions, ne pas s'improviser cryptographe et se documenter (vous pouvez remercier les auteurs de ce HS pour cela) afin de sortir couvert.

Ne pas oublier non plus que la crypto a pour objectif de protéger les parties prenantes d'un échange. En cela, la crypto n'est pas l'apanage des gentils. D'ailleurs, les auteurs de *ransomwares* progressent eux aussi. Et à l'autre bout de l'échelle de l'évolution du *malware*, Stuxnet et Flame (on ne mentionnera pas DuQu dont on ne voit pas l'arrêt) vont même jusqu'à compromettre des autorités de certification (par piratage ou par cryptanalyse).

Tout ça pour dire, ne boudez pas votre plaisir en en choisissant des longues (les clés, toujours), mais tout ce qui est autour importe aussi beaucoup dans la sécurité réellement apportée.

Bonne lecture,

Fred Raynal
@fredraynal
@MISCRedac



Rendez-vous au 2 novembre 2012 pour le n°64 !

www.miscmag.com

MISC est édité par **Les Éditions Diamond**
B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 - **Fax :** 03 67 10 00 21
E-mail : cial@ed-diamond.com
Service commercial : abo@ed-diamond.com
Sites : www.miscmag.com
www.ed-diamond.com
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution
N° ISSN : 1631-9036
Commission Paritaire : K 81190
Périodicité : Bimestrielle
Prix de vente : 9 Euros

LES ÉDITIONS DIAMOND

Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Frédéric Raynal
Secrétaire de rédaction : Véronique Sittler
Conception graphique : Kathrin Scali
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne
Illustrations : www.fotolia.com
Distribution France : (uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou. Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04
Service des ventes : Distri-médias : Tél. : 05 34 52 34 01

Membre April www.april.org

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans MISC est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à MISC, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire.

Charte de MISC

MISC est un magazine consacré à la sécurité informatique sous tous ses aspects (comme le système, le réseau ou encore la programmation) et où les perspectives techniques et scientifiques occupent une place prépondérante. Toutefois, les questions connexes (modalités juridiques, menaces informationnelles) sont également considérées, ce qui fait de MISC une revue capable d'appréhender la complexité croissante des systèmes d'information, et les problèmes de sécurité qui l'accompagnent. MISC vise un large public de personnes souhaitant élargir ses connaissances en se tenant informées des dernières techniques et des outils utilisés afin de mettre en place une défense adéquate.
MISC propose des articles complets et pédagogiques afin d'anticiper au mieux les risques liés au piratage et les solutions pour y remédier, présentant pour cela des techniques offensives autant que défensives, leurs avantages et leurs limites, des facettes indissociables pour considérer tous les enjeux de la sécurité informatique.

SOMMAIRE

HISTOIRE DE LA CRYPTO

[04] COMMENT LA CRYPTO FUT INTRODUITE DANS LA CARTE À PUCE : QUAND LES ANECDOTES DEVIENNENT DE L'HISTOIRE

[10] LE LOGARITHME DISCRET CONTRE LES TUNNELS SÉCURISES

LA CRYPTO, COMMENT L'UTILISER, POUR QUOI FAIRE ?

[14] J'AI UTILISÉ LA CRYPTOGRAPHIE, ET JE SUIS ENCORE VULNÉRABLE

[24] REVUE D'ATTAQUES DU PROTOCOLE TLS ET DE L'IMPLÉMENTATION OPENSSL

LES AVANCÉES DE LA CRYPTO MODERNE

[34] HACHAGE SÉCURISÉ : LA COMPÉTITION SHA-3

[44] DES BONNES PRATIQUES DE PROGRAMMATION DE LA CRYPTOGRAPHIE CONTRE LES ATTAQUES PAR CANAUX AUXILIAIRES

LA CRYPTO EN PRATIQUE

[56] SÉQUESTRE DE CLÉS : MISE EN PRATIQUE DU « SHAMIR'S SECRET-SHARING SCHEME »

[62] REVERSE ENGINEERING D'UNE CHARMANTE APPLICATION 16 BITS

CRYPTANALYSE PAR LA PRATIQUE

[68] LA CRYPTOGRAPHIE DANS RDP : RSA, ANECDOTES ET ERREURS D'IMPLÉMENTATION

[77] MÉCANISME DE CONTRÔLE D'AUTHENTICITÉ DES PHOTOGRAPHIES NUMÉRIQUES DANS LES REFLEX CANON

ABONNEMENT

[41 / 75 / 76] BONS D'ABONNEMENT ET DE COMMANDE

COMMENT LA CRYPTO FUT INTRODUITE DANS LA CARTE À PUCE : QUAND LES ANECDOTES DEVIENNENT DE L'HISTOIRE

Jean-Jacques Quisquater

mots-clés : CARTE À PUCE / SMART CARD / DES / RSA / COPROCESSEUR / SÉCURITÉ PHYSIQUE

Si aujourd'hui, en 2012, il nous semble simple et évident qu'il y a de la cryptographie forte dans nos smart cards, cela n'a pas toujours été ainsi.

Cet article retrace une grande partie de l'histoire de l'introduction de la cryptographie dans les cartes à puce à partir de la vision personnelle, la plus objective possible, de l'auteur. Nous parlerons aussi un peu de sécurité physique et autre.

1 Introduction

J'ai commencé à travailler sur la carte à puce en janvier 1980 suite à une demande de Philips France pour sécuriser les échanges entre une puce et son lecteur et je n'ai plus vraiment arrêté depuis. Cet article est la suite de ma référence **[15]** qui date de 1997, du chapitre de Louis Guillou, Michel Ugon et moi-même dans l'excellent livre de Gus Simmons **[17]** où le chapitre 12 (*A standardized security device dedicated to public cryptology*) détaille l'ensemble de l'histoire de la carte à puce en plus de 50 pages, et, enfin, du papier historique de Louis Guillou **[9]** en 2004. Notons encore la référence **[8]** de Louis Guillou et Michel Ugon à *CRYPTO* 1986.

La carte à puce a une assez longue histoire et ses premiers développements et brevets n'utilisent pas la cryptographie. Que ce soient les Gondas dans la Nuit des temps de René Barjavel, la première bague de Roland Moreno (voir la référence **[16]** dans la revue *FEB-actualités* qui raconte les débuts de la carte à puce), les premiers brevets japonais, allemands ou américains, rien n'indique que la puce doit avoir des éléments de cryptographie. Il y a cependant une exception : les travaux chez IBM sous la direction de Carl Meyer furent très tôt orientés vers l'utilisation de la cryptographie et une version expérimentale de carte fut testée avec l'algorithme cryptographique Lucifer dans le cadre d'une expérience d'ATM à Londres.

Il y a donc trois questions fondamentales :

- Pourquoi la crypto dans la carte à puce ?
- Est-il possible de mettre de la crypto dans la carte à puce ?
- Nécessité de l'implémentation d'un algorithme suffisamment rapide ?

Nous allons voir maintenant en détail ces trois aspects importants.

2 Pourquoi la cryptographie dans la carte à puce

Et tout d'abord pourquoi faut-il de la crypto dans la carte à puce ? Il est assez facile de voir que si on veut éviter certaines attaques, il faut associer à la puce une certaine puissance de calcul. Prenons le cas simple du mot de passe stocké par la carte et qui sera utilisé pour avoir accès à une ressource (ce fut le cas au début pour la télévision à péage et aussi pour le Minitel). Supposons une carte, qui sera associée à un utilisateur, qui tente de prouver son identité en donnant un mot de passe :

- pour stocker ce mot de passe, il faudra que cette puce ait une mémoire inviolable, et donc, la puce elle-même le sera par extension.
- si le mot de passe (*password*) est transmis tel quel, et donc en clair, il sera facile à copier. Sa transmission ne sera donc pas directe, mais bien en utilisant un détour, par exemple, une question surprise au sujet de ce mot de passe. En pratique, cela se fera par l'usage de nombres aléatoires (*random*), du mot de passe en combinaison avec un algorithme

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

cryptographique (que ce soit une fonction à sens unique, une fonction de chiffrement, un algorithme pour protocole à apport nul de connaissance où le mot de passe ne s'use pas si l'on s'en sert, ...).

À l'époque (1980), il fallait tenir compte en plus des contraintes venant des autorités de contrôle en France (service du chiffre, aujourd'hui ANSSI) : il fallait éviter le détournement (donc, le chiffrement associé au déchiffrement) des objets qui faisaient usage de cryptographie forte. Cela stimula l'innovation : ainsi, les schémas basés sur l'identité, dans le cas du chiffrement à clé secrète, seront pratiquement inventés en 1984 par Louis Guillou, sous le nom de diversification de la clé secrète alors que la même année, Adi Shamir en donnera la description théorique pour les systèmes à clé publique sans aucune proposition de réalisation correcte. Il faudra attendre l'arrivée du protocole de Fiat-Shamir (1986) pour en voir une réalisation pratique, vite suivi par le protocole GQ (Guillou-Quisquater) en 1988.

3 Est-il possible de mettre de la crypto dans la carte à puce ?

Au début, la carte à puce n'offrait pas beaucoup de place : il fallait vraiment innover. De plus, au début, la carte comportait deux puces (voir Figure 1), l'une avec le processeur, l'autre avec la mémoire, connectées avec ce qui tenait fort de la bijouterie mais qui permettait aussi facilement d'espionner les échanges processeur-mémoire où apparaissaient sans aucun doute les éléments secrets. Il fallut attendre l'arrivée de la monopuce pour résoudre ce problème. Des puces multiples ont même été imaginées pour résoudre les problèmes de mémoire, de puissance de calcul, sans jamais atteindre une réalisation (projet octopuce présenté à *Securicom*).

Figure 1

Les premiers algorithmes cryptographiques pour la carte à puce seront donc des algorithmes cryptographiques soit à sens unique, soit inversible, mais toujours à clé secrète. Ce sera la suite de Telepass 1 (voir Figure 2), TDF, Telepass2, Videopass, ...

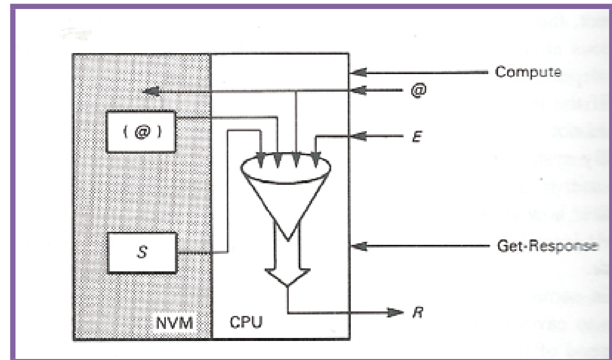


Figure 2

Le premier algorithme cryptographique pour carte à puce semble avoir été imaginé par Honeywell Bull en 1979. Cet algorithme n'a jamais été publié comme les autres ci-dessus d'ailleurs. Tous ces algorithmes étaient bien adaptés à leur contexte restreint en vitesse et en mémoire ainsi qu'aux processeurs 8 bits en usage. Ils utilisaient entre 200 et 300 octets pour leur implémentation complète, ce qui est certainement un exploit. Cependant, la non-publication de ces algorithmes en rendait certainement l'usage plus restreint et la diffusion en dehors de la France difficile.

Le déclic s'est produit à *EUROCRYPT* 1984, à Paris, à la Sorbonne. Louis Guillou **[7]** y présenta la carte à puce et l'accès conditionnel lors d'une session spéciale dédiée aux usages de la carte à puce, une première dans une conférence cryptographique internationale. Lors des questions à la fin de son exposé, des américains demandent quel algorithme cryptographique est utilisé : Louis parle de Telepass et de TDF, puis vu l'insistance de l'assistance, il esquisse au tableau la forme générale de ces algorithmes, ce qui fait sourire certains (rien ne prouve que ces algorithmes sont solides), et grincer certains (là, Louis, tu vas toucher la ligne rouge). À la dernière question, « pourquoi n'utilisez-vous pas le DES ? » Louis répond : « Cela ne sera jamais possible car la ROM est bien trop petite ». Et, de fait, il n'y avait à l'époque que 2048 octets disponibles pour les programmes, tout compris, même si plus tard on trouva aussi l'astuce d'utiliser l'EPROM ou E²PROM pour stocker quelque 500 octets de plus. Les ingénieurs de Philips et moi-même (je travaillais alors au laboratoire Philips PRLB, à Bruxelles) étions dans la salle, et à la sortie, nous nous mîmes à discuter entre nous de ce *challenge* fascinant : mettre le DES dans la carte à puce, sans trop se soucier de nos changes d'obtenir l'autorisation de le faire (il y avait à l'époque pas mal de problèmes d'exportation, entre autres). Nous avions travaillé sur un prototype de modem chiffrant qui utilisait le DES, le même processeur que les cartes à puce (80C51) et ce DES complet tenait dans 1500 octets. Nous arrivons donc dans la saga de la carte à puce avec DES.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

3.1 Le DES dans la carte à puce

Il y avait à l'époque (1984) un projet de transmissions interbancaires sécurisées en Belgique dénommé TRASEC (*TRAnsaction SECurity*) **[18]**. L'idée était de stocker les clés secrètes dans une carte à puce sous le contrôle d'un PIN code et d'exécuter l'algorithme DES avec cette clé dans un ordinateur lui-même non sécurisé. C'était très peu satisfaisant. Je propose alors de lancer un projet visant à implanter le DES dans la carte. Une première étude démarre et nous permet de voir que 1000 octets environ suffiront avec un temps raisonnable d'exécution. Répondant à l'appel des banques belges, Philips propose d'étudier une version plus compacte. Durant plusieurs mois, à plusieurs chercheurs et ingénieurs, nous essayons de réduire le code octet par octet, en déplaçant les permutations en usage dans le DES et en repliant ces mêmes permutations vu leurs propriétés. Finalement, nous parvenons à faire tenir notre DES en 668 octets, chaque octet ayant coûté environ un jour de travail... C'était une première mondiale. Cette version du DES sera longtemps utilisée par les cartes, y compris dans la carte commune Philips-Bull, TB100, et je suis à peu près sûr que l'on en retrouve encore des traces aujourd'hui dans certaines cartes.

Merci à Xavier Darmstaedter (DACOTA Consulting) d'avoir mené à bien la réalisation du projet TRASEC.

Le challenge suivant, le RSA, a une histoire encore plus étonnante.

3.2 Le RSA dans la carte à puce

Il était clair pour chacun que mettre le RSA dans la carte était en quelque sorte le Graal cryptographique de la carte à puce. Cela permettait la signature, la distribution des clés, etc. Paul Barrett invente alors un nouvel algorithme pour le RSA, mais plus spécifiquement dédié à des processeurs pour traitement du signal **[1]**.

Nous faisons quelques tests sur une carte à puce en 1986 et nous arrivons à une première version qui demande près de 100 secondes pour s'exécuter. Puis, nous ajoutons le reste chinois (technique mathématique qui permet de gagner un facteur 4), d'autres astuces liées au stockage de multiples du module public. Finalement, pour une clé RSA de 512 bits, pour un module public à deux facteurs, nous arrivons à fournir une signature en moins de 20 secondes pour un module public général (des formes spéciales permettent d'aller plus vite) ainsi qu'un exposant aléatoire (idem). Il y a cependant plusieurs problèmes : trop lent pour être utilisé (imaginer cela à une caisse de supermarché) et aussi, et même surtout, trop gourmand en ressources de la carte (ROM, RAM, etc.). Il faut donc passer à autre chose.

Le SEPT à Caen commence à imaginer l'usage d'un coprocesseur, ce qui utilise des ressources en surface de silicium qui, comme on le sait, sont aussi limitées vu les possibles flexions de la carte (moins de 20 mm^2). Il est dommage que ce projet n'ait pas été poursuivi.

En 1989, Henri Molko (Philips France et Allemagne) nous arrive avec un projet extraordinaire : ajouter un coprocesseur RSA à un processeur 80C51, en vue de son usage pour réaliser une signature RSA sur 512 bits en moins d'une seconde (nous ferons 0.2 seconde). Les moyens financiers prévus sont quasi nuls, mais nous allons bien utiliser la souplesse d'alors de Philips. Ce sera le projet CORSAIR (*Coprocessor RSA In a Rush*) **[3-14]**. En cinq semaines, plusieurs versions du coprocesseur seront proposées à partir d'un nouvel algorithme que j'avais imaginé entre-temps. Ceci conduira à une première accélération du RSA pour carte à puce par un facteur 500, tenant compte que nous avions réussi à obtenir que le diviseur par deux pour la fréquence d'horloge ne soit pas utilisé. Dominique de Waleffe écrira les simulations et les tests tandis que Jean-Pierre Bournas écrira le microcode. Un premier silicium sera obtenu début 1990 : tout marchera correctement sauf l'écriture dans E^2PROM « trop » optimisée. La Figure 3 est le premier écran d'une simulation de CORSAIR.

Un nouvel algorithme avait donc été imaginé : il est référencé dans la littérature comme l'algorithme de Quisquater **[13]** : ses propriétés principales sont que le coprocesseur est indépendant de la longueur de la clé, que l'exécution est à temps constant de bout en bout car il n'y a aucun test effectué en cours de calculs.

Puis ce sera la course aux coprocesseurs de deuxième génération **[11-12]** parmi les fabricants, chacun utilisant l'algorithme de Montgomery, en commençant par Fortress, sauf Siemens (Sedlak), maintenant Infineon, ... et Philips, bien sûr, (maintenant NXP). De fait, Philips se fait dépasser car étant parti le premier, les autres ont pu profiter de ses expériences.

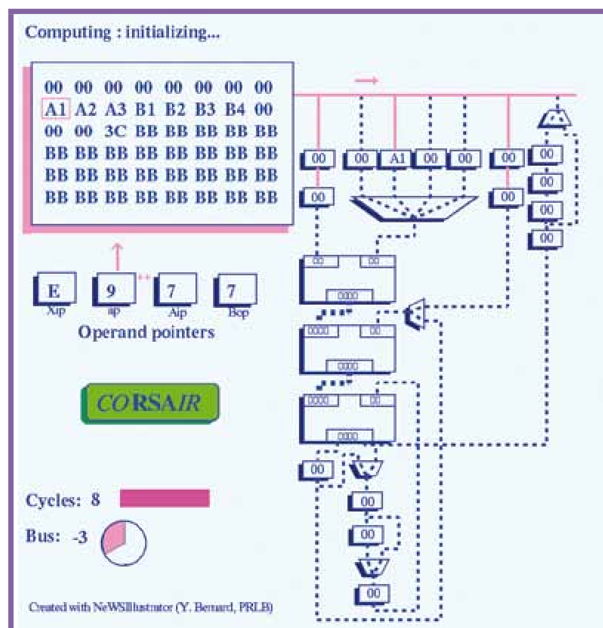


Figure 3

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

En 1995, un nouveau projet aura lieu : FAME **[6]**. Cela conduira l'auteur à une nouvelle version 500 fois plus rapide que celle de CORSAIR. Cette accélération aura été obtenue grâce à un multiplieur d'horloge (16 fois), un multiplieur 32 bits fois 32 bits (au lieu de 8 fois 8), un bus plus large, une mémoire RAM à double accès très large et un meilleur *pipe-line*. Ce coprocesseur incluant des améliorations (ajout des courbes elliptiques) est toujours utilisé dans des variantes optimisées. Au total, en 10 ans, une accélération de 250.000 aura été obtenue entre la première version logicielle 8bit et celle qui résulte de FAME. Bien mieux que la loi de Moore ! La figure 4 décrit brièvement les composants de FAME.

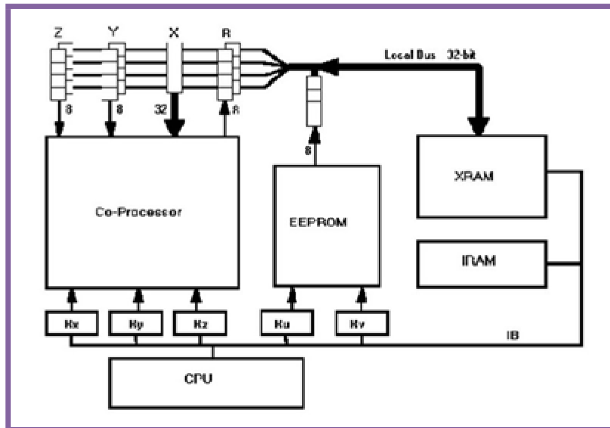


Figure 4

Il y aura encore le projet européen CASCADE (1994-1997) **[2]** sous la direction de Patrice Peyret puis de Pierre Paradinas, tous deux de Gemplus, où on parviendra à incorporer un processeur ARM, d'architecture RISC, à une carte à puce : ce sera aussi l'ancêtre de la Java Card. Nous y apportons des modifications au processeur ARM plutôt orienté traitement du signal en vue d'effectuer efficacement des instructions de type cryptographique. Cela conduira au strongARM.

Notons enfin le projet SCALPS (1996) totalement préparé à l'UCL (Louvain-la-Neuve) et qui résultera en une puce complète incorporant le protocole GQ modifié pour effectuer des paiements sécurisés **[4]**.

4 Les premières attaques

Les premières attaques contre la carte furent liées à la TV à péage dans différents pays. L'inviolabilité supposée de la carte était ainsi mise à rude épreuve. Ces attaques résultaient d'un mélange de véritables attaques physiques, d'erreurs d'implémentation utilisées par les pirates, mais aussi de luttes entre différents acteurs du domaine.

Les attaques par mesure par le temps d'exécution furent connues très tôt (1986) sans être publiques. Paul Kocher **[10]** fut le premier à la publier en 1996, mais dans un contexte qui n'était pas celui de la carte à puce. La première réalisation expérimentale sur une carte à puce ne fut réalisée et publiée qu'en 1998 à *CARDIS* **[5]**.

L'attaque par la simple mesure de la consommation (SPA : *simple power analysis*) était bien connue avant sa publication par Paul Kocher et ceci par plusieurs laboratoires, y compris Philips, TNO (Pays-bas) et IBM. Nous avions effectué deux campagnes de mesure, l'une en 1989, l'autre en 1992. Nos directeurs étaient tellement effrayés qu'ils détruisirent les documents en nous interdisant formellement d'en parler.

En 2000, il y aura la saga des cartes bleues et les « attaques » de Serge Humpich. Je ferai comme à l'époque, je transcris ici les deux textes, publiés en l'an 2000 dans fr.misc.cryptologie, à la demande générale, qui illustrent toute l'histoire.

5 L'étirage des clés

Les tirages des clés ou l'étirage des clés

soutiré à l'as sans ion positif (fin mets de mil)

par Cyreno de Heressac (à dos de prime jeunesse, c'est grand mot dire) (d'après Edmond RoStAnd - 1868-1918, de guerre lasse)

Nous sommes en 1640 (le petit théorème vient juste d'être inventé par Pierre Fermat - 1601-1665 - sa mort a aussi été annoncée en 1653, fosse nouvelle -,
à Toulouse, bien qu'il fisse sûrement ses études à
Bordeaux et Orléans, on n'est jamais trop prudent :
pour le grand, il y a encore de la marge...).

CYRENO
Ah ! non ! elle est un peu courte, jeune homme !
On pouvait tirer... ou mieux !... bien des clés en somme...
En variant à la tonne, -parbleu, tenez
Agressif : "Moi, monsieur, si j'avais une telle clé,
Il faudrait sur-le-champ que je la retirasse !"
Amical : "Mais elle ressemble à votre mot de passe
Pour moins le voir, faites-la passer dans un hachage !"
Descriptif : "Escroc laid !... c'est une pique !... c'est un gage ! Que dis-je, c'est un gage ?... C'est un bénin module !"
Curieux : "De quoi sert ce peu long bidule ?
De critères, monsieur, ou de carte à grimoire ?"
Gracieux : "Aimez-vous à ce point les gros loirs
Que maternellement vous vous préoccupâtes
De tendre ce pair choix à leurs petites pattes ?"
Truculent : "Ça, quand le tirage est augmenté,
La peur du tas bas vous sort-elle la clé
Sans qu'un voisin ne crie aux facteurs publiés ?"
Prévenant : "Gardez-vous, sa tête entraînée
Par son poids, de la fendre en deux sur le champ !"
Tendre : "Faites-lui choix, sire, d'un paramètre tout chant,
Que sa valeur à la lune ne se profane !"
Pédant : "L'algorithme, monsieur, qu'Aestophane
Appelle Marserpentwofishercésixrijndael,
Cent rondes, avec cette clé, ne vaut guère la scytale !"
Cavalier : "Quoi, là mis, ce nombre est à la mode ?
Pour perdre son code secret, c'est vraiment très commode !"
Emphatique : "Ce vent si favorable, clé astrale,

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Guère te factorise, si ne n'est le NISTrAl !"
Dramatique : "C'est l'amère douche quand elle signe !"
Admiratif : "Être au parfum, quel Bond digne !"
Lyrique : "Est-elle quelconque, de quel germe hérite-t-on ?"
Naïf : "Ce module, quand le factorise-t-on ?"
Respectueux : "Souffrez, dame, qu'on vous prête main forte,
C'est là ce qui s'appelle avoir clé sur porte !"
Campagnard : "Hé, ! C'est-y une clé au pâtre? Nanain !
Queuqu'César fainéant ou queuqu'meuh long nain !"
Militaire : "Tirez contre cave Valérie !"
Pratique : "Voulez-vous la mettre en loterie ?
Assurément, en prime, ce sera le gros lot !"
Enfin parodiant PyRSAme en un sanglot
"La voilà donc cette clé qui des traites de son maître
A détruit l'harmonie ! Elle en bleuit, le traître !"
- Voilà ce qu'à peu près, mon cher, vous m'auriez dit
Si vous aviez peu de lettres et plus de chiffres
Mais de chiffre, ô le plus fragmentable des êtres,
Vous n'en eûtes jamais mille bits, et de lettres
Vous n'avez que les trois qui forment le mot : RSA !
Eussiez-vous eu, d'ailleurs, le fin du mot du la
Pour pouvoir là, devant ces nobles galeries,
me servir toutes ces folles plaisanteries,
Que vous n'en eussiez pas articulé le quart
De la moitié du commencement d'un, car
Je me les tire moi-même, avec assez de verve,
Mais je ne permets pas qu'un autre me l'étire.

Toute ressemblance avec des faits réels est purement aléatoire.

Voici le texte complet de la chanson de Roland : certains liens ne sont plus accessibles.

6 La chanson de Roland

Si je peux faire court (oui, mon gars lent) :
--- La chanson de Roland ---
- Au début était la carte à puce, et la carte était Roland Moreno en 1974, voir histoire "officielle" à **http://www.cardshow.com/museum/ex70/a74.html***, qui nous raconte un conte de faits : depuis la nuit des temps, René Barjavel en parlait et d'autres, bien cités par Jean Moulin, à* **http://wwwusers.imaginet.fr/~jmoulin/jmccarte.html***.*
- Dieu trouva que c'était bon et la carte fut incassable en France, pour < 100 F, l'obscurité permit de contourner les lois de la physique et des théorèmes de Shannon, excusez du peu : la cryptographie ne viendra que plus tard
- Paul Kocher n'était pas au courant et trouva le temps de nous la casser, **http://www.cryptography.com/timingattack/** *et surtout* **http://www.cryptography.com/dpa/technical/index.html**
- Une spécification figée en 1984 par ma-gie (Orwell quand tu nous tiens),
Louis Monier, mieux connu aujourd'hui comme l'inventeur d'altavista, avait dans sa thèse d'Orsay affirmé que les nombres de 100 digits ne seraient pas factorisés pour l'an 2000 : un autre bug, un vrai, le pauvre, il n'avait pas prévu l'arrivée de Pollard, un vrai polar...
- Un simple double comme redondance pour la signature RSA, pour éviter les attaques multiplicatives, fut vite écarté par les experts mais retenu car fort simple pour éviter les doubles : l'ISO se chargea de faire mieux, trop vite à mon goût, et ce fut ISO 9796 qui ne passa pas non plus l'an 2000 victime du mou Boo Barkee, du dynamique Deh Cac Can, à l'inverse, et surtout du regretté R. F. Ree qui n'a pas toujours respecté l'anonymat, par de vilains calculs personnels, tous symboliques et coronsternants, **http://dblab.comeng.chungnam.ac.kr/~dolphin/db/journals/jsc/jsc18.html**
Certes, il manquait un bit, vite percé par Don QuiCop de la Manche du Des, et le bit ne fait pas les moines, et François Grieu - EUROCRYPT 2000 - nous montra comment fabriquer de nouvelles signatures à partir de quelques authentiques, est-ce bancal ou bancaire ou banco? Ici, retour à la case Roland et Lino Vatronc sonnant et trébuchant]
- Humpich joue sa carte, croit donner une frousse bleue et perd : un royaume pour mon ticket de métro. Métro, c'est trop. Il n'avait pas misé sur le bon cheval, se croyant richard trop vite.
- Fin de l'épisode : le général Desvignes...
Et c'est ainsi que la carte à puce devint fort peu crédible au pays de la Silicon Valley pour parler francs, non encore chiffrés, c'est clair.
Histoire réécrite en 2000 mais prévisible.
Encore une bataille de perdue, sans cor ni trompette.

Conclusions

L'introduction de la cryptographie semble s'être faite sans beaucoup de recherche. Pour le reste, le cycle d'élaboration des diverses versions de carte à puce suite à des attaques ressemble surtout au cycle de Shiva dans l'hindouisme : c'est le fameux cycle de construction, destruction et purification où la R&D est fort absente. Il est remarquable de noter que la première conférence de recherche dédiée à la carte à puce commence seulement en 1994 (voir Figure 5 : une carte à puce comme carte de participant). Depuis lors, la recherche en cryptographie et carte à puce s'est fortement agrandie pour devenir aujourd'hui majeure. ■

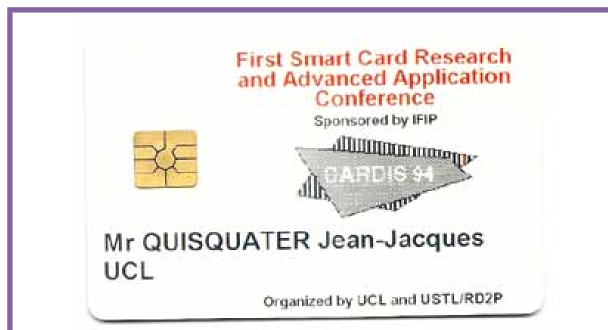


Figure 5

REMERCIEMENTS

L'auteur tient à remercier le général Jean-Louis Desvignes (ancien Chef du Service Central de la Sécurité des Systèmes d'Information) pour les nombreuses et amicales remarques qui ont permis d'améliorer ce texte. Henri Molko et Marc Girault ont aussi relu ce texte.

Les références de cet article sont disponibles sur : **http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6**

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

DEVENEZ

EXPERT EN SÉCURITÉ DE L'INFORMATION

MASTER >>> PARCOURS.1 SÉCURITÉ INFORMATIQUE
PARCOURS.2 MATHÉMATIQUES, CRYPTOLOGIE, CODAGE

www.cryptis.fr

Formation & Recherche

Université de Limoges | FACULTÉ DES SCIENCES ET TECHNIQUES

Master Cryptis - Faculté des Sciences et Techniques
123 Av. Albert Thomas - 87060 Limoges CEDEX
05 55 45 73 23 - cryptis@unilim.fr

LE LOGARITHME DISCRET CONTRE LES TUNNELS SÉCURISÉS

F. Morain - morain@lix.polytechnique.fr - INRIA & LIX École Polytechnique Palaiseau

mots-clés : *LOGARITHME DISCRET / CRYPTANALYSE DU PROTOCOLE DE DIFFIE-HELLMAN / THÉORIE ALGORITHMIQUE DES NOMBRES / COURBES ELLIPTIQUES*

Les tunnels sécurisés sont souvent basés sur des protocoles proches du protocole historique de Diffie et Hellman. La sécurité du tunnel repose sur la résistance de certains groupes finis aux algorithmes de calcul de logarithme discret. Nous faisons le point sur les méthodes de cassage connues.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

1 Le logarithme discret : un problème peu médiatique

Contrairement au problème de la factorisation d'entiers qui attire beaucoup d'attention car il est lié de très près à la sécurité du cryptosystème RSA et du contenu de nos portefeuilles (cf. « Factorisation d'entiers : la voie royale du cassage de RSA » paru dans le hors-série n°5 de *MISC*), celui du logarithme a reçu moins de lumière, alors qu'il est tout aussi important, sinon plus.

Le protocole d'échanges de clés de Diffie et Hellman est rappelé ci-contre. C'est en fait un méta-algorithme, au sens où il doit être instancié par un groupe fini G dans lequel le problème du logarithme discret doit être difficile à résoudre. On se contente généralement d'utiliser un groupe cyclique : si G est cyclique, alors tout élément de G peut s'écrire comme puissance d'un élément générateur g. Le problème du logarithme discret (LD) s'énonce ainsi : étant donné un élément a de G, comment trouver un entier x tel que $a = g^x$? Un tel entier s'appelle le logarithme discret de a en base g.

L'exemple typique est celui des corps finis. Par exemple, dans $(\mathbb{Z}/11\mathbb{Z})^* = \{1, 2, ..., 10\}$ (le groupe des entiers inversibles modulo 11), on constate que tout élément est une puissance de 2 :

$2^0 = 1$, $2^1 = 2$, $2^2 = 4$, $2^3 = 8$, $2^4 = 5$, $2^5 = 10$, $2^6 = 9$, $2^7 = 7$, $2^8 = 3$, $2^9 = 6$.

Nous allons passer en revue quelques algorithmes de résolution de ce problème, que ce soit des algorithmes génériques (qui marchent dans tous les groupes), ou des algorithmes utilisant des relations entre éléments du groupe (quand c'est possible).

2 Élimination des mauvais groupes

Le but du jeu est de trouver des groupes (finis) faciles (temps polynomial) à utiliser et dans lesquels le problème LD est difficile (au moins sous-exponentiel). L'un des groupes à éviter est simplement $(\mathbb{Z}/N\mathbb{Z}, +)$, dans lequel le calcul du logarithme discret revient à effectuer un pgcd...

2.1 La réduction de Pohlig-Hellman

Il est utile d'étudier le problème LD dans un groupe abstrait où on ne saurait faire que la multiplication d'éléments. Soit donc G un groupe cyclique d'ordre n, engendré par a. Pohlig et Hellman ont montré que calculer le logarithme de a modulo n était équivalent à déterminer ce logarithme modulo chaque facteur de n, par application du théorème chinois. Ainsi, si n n'est composé que de petits facteurs premiers, résoudre LD est facile, puisqu'au pire on peut énumérer tous les logarithmes modulo chaque facteur premier.

À titre d'exemple, reprenons le cas de $G = (\mathbb{Z}/11\mathbb{Z})^*$. Comme # $G = 10$, il suffit de calculer le logarithme discret modulo 2 et 5. Par le théorème de Fermat (Lagrange), on a $a^{10} = 1$, ce que l'on peut récrire $(a^2)^5 = 1$, donc tous les nombres a^2 sont d'ordre divisant 5, ou encore des puissances de $g^2 = 2^2$, qui est d'ordre 5. Ce qui veut dire que pour tout nombre a, on a $a^2 = (g^2)^i$ avec $i \in \{0, 1, ..., 4\}$. Il suffit de trouver ce i. Par exemple :

$$9^2 \equiv 81 \equiv 4 \bmod 11 \equiv (2^2)^1 \bmod 11,$$

ce qui conduit à $\log_2 9 \equiv 1 \bmod 5$. De la même façon :

$$9^5 \equiv 1 \bmod 11 \equiv (2^5)^0 \bmod 11$$

donc $\log_2 9 \equiv 0 \bmod 2$. Et le seul entier k modulo 10 vérifiant $k \equiv 1 \bmod 5$ et $k \equiv 0 \bmod 2$ est $k = 6$.

2.2 L'algorithme de Shanks

Pour résoudre $a = g^x$ dans G avec $0 \leqslant x < n$, on écrit $x = c\,u + d$, avec u un entier à déterminer, c et d les quotient et reste de x par u. On réécrit l'équation $a = g^x$ sous la forme :

$$a = (g^u)^c \cdot g^d$$

avec $0 \leqslant d < u$ et $0 \leqslant c < n/u$. Si l'on note g^{-1} l'inverse de g dans G, on en déduit :

$$a\,(g^{-u})^c = g^d.$$

Nous avons ramené le problème à un problème à deux variables séparées. On commence par calculer tous les $b_d = g^d$ pour tous les d possibles, de 1 en 1, appelés pas de bébé. Puis pour chaque valeur de c, on calcule de proche en proche la quantité $a\,(g^{-u})^c$, ce sont les pas de géant. Si pour un c donné, on trouve une identité $(g^{-u})^c = b_d$, on a gagné.

Comment analyse-t-on cet algorithme ? Le nombre d'opérations de groupes est essentiellement u pas de bébé et au pire n/u pas de géant. Le nombre total est minimal pour $u = \sqrt{n}$, conduisant à un nombre $2\sqrt{n}$. L'espace mémoire de stockage pour les b_d est également de $\sqrt{n}$ éléments de G. D'un point de vue pratique, il est vital de stocker les b_d avec une table de hachage, pour garantir des tests d'appartenance rapides aux pas de bébé.

Pour finir, il existe un autre algorithme de calcul de logarithmes discrets en temps $O(\sqrt{n})$, qui utilise peu de stockage, l'algorithme (probabiliste) de Pollard. Les deux algorithmes sont distribuables.

2.3 Conclusion provisoire

D'après ce que l'on vient de voir, un bon groupe ne doit pas avoir un cardinal composé de trop petits facteurs premiers. En outre, sa taille doit être suffisante pour que l'attaque en $\sqrt{n}$ ne soit pas réaliste.

▪ LE PROTOCOLE DE DIFFIE-HELLMAN

Dans cette version simple, Alice et Bob veulent établir une clé commune avec des valeurs qui transitent sur un réseau non protégé. Pour cela, ils choisissent un groupe $G = \langle g \rangle$ et effectuent le protocole suivant. Alice commence par choisir un nombre aléatoire x et envoyer g^x à Bob :

$$A \xrightarrow{g^x} B$$

Bob agit de façon symétrique :

$$A \xleftarrow{g^y} B.$$

À la fin des communications, Alice peut reconstituer la clé commune :

$$A : K_{AB} = (g^y)^x = g^{xy},$$

ce que Bob peut également faire :

$$B : K_{BA} = (g^x)^y = g^{xy}.$$

Le problème de Diffie-Hellman est le suivant : étant donnés (g, g^x, g^y), calculer g^{xy}. Le problème LD est : étant donnés (g, g^x), trouver x.

Il est clair que si on sait résoudre LD, on sait résoudre DH. On sait que la réciproque est vraie dans de nombreux groupes G (travaux de Maurer et Wolf).

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

3 L'algorithme d'Adleman

Dès 1978, L. M. Adleman a proposé un algorithme de calcul sous-exponentiel de calcul de logarithme discret modulo *p*, qui est proche en esprit des algorithmes de factorisation (voir section suivante). Cet algorithme a été généralisé pour de nombreux groupes (corps finis généraux, certaines courbes algébriques), dès qu'une notion de nombres premiers peut être définie dans le groupe.

Cet algorithme n'est pas sans rappeler les algorithmes de factorisation utilisant des combinaisons de congruences. Il procède en deux phases : dans la phase de collection, on accumule les logarithmes discrets de nombres dans une base. Dans la seconde phase, on calcule des logarithmes individuels en s'aidant des résultats de la première phase.

Considérons le cas de $(\mathbb{Z}/101\mathbb{Z})^*$, engendré (lui aussi) par 2. L'idée est de trouver des relations qui nous donneront les logarithmes discrets des nombres {2, 3, 5, 7}, en partant de puissances aléatoires de $g = 2$:

$$2^7 = 128 \equiv 3^3 \bmod 101,\ 2^9 = 7,\ 2^{17} = 3 \cdot 5^2.$$

Comme $\log_2 2 = 1$, on en déduit que les logarithmes de ces nombres en base 2 vérifient :

$$7 = 3 \cdot \log 3 \bmod 100,\ \log 7 = 9,\ \log 3 + 2\log 5 = 17,$$

soit log3 = 69, log5 = 24, log7 = 9.

D'un point de vue algorithmique, on crée un système linéaire entre les logarithmes des $\{p_j\}1 \leqslant j \leqslant k$ qui nous intéressent. Une relation

$$g^{b_i} \equiv \prod_{j=1}^{k} p_j^{\alpha_{i,j}} \bmod p$$

conduit par passage au logarithme

$$b_i \equiv \sum_{i=1}^{k} \alpha_{i,j} \log_g p_j \bmod (p-1),$$

soit une relation linéaire entre les inconnues $\log_g p_j$. Une fois que *k* relations ont été trouvées, on doit inverser ce système défini modulo $p-1$. Ce système est creux, mais contrairement à ce qui se passe pour la factorisation d'entiers, ce système n'est donc pas binaire, et donc plus difficile à résoudre.

Une fois cela fait, on cherche des *b* tels que :

$$ag^b \equiv \prod_{j=1}^{k} p_j^{\alpha_{i,j}} \bmod p$$

et par passage au logarithme :

$$\log_g a + b \equiv \sum_{i=1}^{k} \alpha_{i,j} \log_g p_j \bmod (p-1)$$

et il ne reste plus qu'à injecter les valeurs des logarithmes trouvés pour en déduire $\log_g a$.

Si l'on veut calculer le logarithme de 17, on trouve par exemple :

$$2^3 \cdot 17 \equiv 5 \cdot 7 \bmod 101$$

ce qui conduit à :

$$3 + \log 17 \equiv \log 5 + \log 7 \equiv 33,$$

d'où log17 = 30.

4 Complexité et records

L'analyse de cet algorithme fait intervenir la fonction

$$L_x[\alpha, c] = \exp\left(c(\log x)^{\alpha}(\log\log x)^{1-\alpha}\right),$$

avec $0 \leqslant \alpha \leqslant 1$, introduite dans le précédent article. Comme pour la factorisation d'entiers, on peut utiliser des corps de nombres, ce qui conduit à la meilleure complexité connue : $O(L_p[1/3, c])$. Notons que la théorie nécessaire est plus complexe que dans le cas de la factorisation des entiers.

Le record actuel, dans $(\mathbb{Z}/p\mathbb{Z})^*$, est pour *p* de 160 chiffres décimaux, obtenu par T. Kleinjung en 2007. Il lui avait fallu 3.3 années sur de PC 3.2 GHz Xeon64, terminant avec une matrice 2,177,226× 2,177,026 avec 289,976,350 coefficients non nuls, inversée en 14 ans de CPU. La partie inversion du système est très technique à faire marcher dans la vraie vie et reste un problème de recherche.

Conclusion

L'investissement théorique dans le logarithme discret a accompagné celui fait pour la factorisation d'entiers. En pratique, il existe moins d'équipes ayant cassé des logarithmes discrets, ce qui explique en partie les records plus petits.

La recherche de nouveaux groupes résistants aux algorithmes à la Adleman est un courant très fort en cryptographie. Les deux seuls candidats qui tiennent la route pour le moment sont ceux formés avec des courbes elliptiques et hyperelliptiques.

Pour les courbes elliptiques, les records de factorisation sont de taille modeste : 112 bits, avec 8.5×10^{16} additions sur 200 PS3 pendant 3 mois et demi. ■

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

DEVENEZ QUELQU'UN DE RECHERCHÉ POUR CE QUE VOUS SAVEZ TROUVER.

FORMATIONS FORENSIQUES

Cours SANS Institute
Certifications GIAC

FOR 408
Investigation Inforensique Windows

FOR 508
Analyse Inforensique et réponses aux incidents clients

FOR 558
Network Forensic

FOR 563
Investigations inforensiques sur équipements mobiles

Dates et plan disponibles
Renseignements et inscriptions
par téléphone +33 (0) 141 409 700
ou par courriel à : formations@hsc.fr

www.hsc-formation.fr

SANS

exclusive de Johann Locatelli (johann@gykroipa.com) le 1 Mars 2015

J'AI UTILISÉ LA CRYPTOGRAPHIE, ET JE SUIS ENCORE VULNÉRABLE

Aris Adamantiadis - aris@badcode.be - Consultant infosec indépendant.
Fondateur du projet libssh (www.libssh.org)

mots-clés : *SÉCURISATION / CHIFFREMENT / SIGNATURE / IMPLÉMENTATION / ALGORITHMES*

La cryptographie est un outil magnifique et est intimement liée à la sécurité informatique. De nombreux algorithmes et protocoles sont disponibles gratuitement pour sécuriser vos applications. Cependant, l'utilisation d'outils cryptographiques ne s'improvise pas, et il y a quelques pièges à éviter.

1 Introduction

Lors de la conception ou de la programmation de votre application, vous serez certainement tenté, à juste titre, d'implémenter des solutions de sécurité fondées sur la cryptographie. La sécurité d'une application doit être évaluée le plus tôt possible, voire même avant l'étape de conception du système. Par sécurité, nous entendons que la confidentialité, l'intégrité des données ainsi que leur disponibilité correspondent aux attentes des utilisateurs. Une autre propriété intéressante est la non-répudiation. La cryptographie permet, dans la plupart des cas, de veiller à ce que ces propriétés soient respectées. Les « briques » suivantes sont disponibles dans la plupart des suites crypto :

- Chiffrement symétrique : cette brique s'occupe de chiffrer des données à l'aide d'une clé secrète, afin qu'un attaquant ne puisse pas en découvrir le contenu. Cette brique s'occupe principalement de la confidentialité.
- Chiffrement asymétrique : ce type de chiffrement utilise une paire de clés (publique, privée) afin de chiffrer des données. Cette brique ne sert qu'à la confidentialité.
- Signature asymétrique : utilisant des outils similaires au chiffrement asymétrique, la signature asymétrique s'assure de la non-répudiation et de l'intégrité de données.
- MAC : les algorithmes de MAC servent principalement à vérifier l'intégrité des données, ou à condenser de manière sécurisée un nombre arbitraire de données afin d'effectuer un contrôle d'intégrité.
- HMAC ou mécanisme de signature symétrique : le HMAC est un algorithme de MAC utilisé en conjonction avec une clé secrète partagée par les différentes parties. Seules les parties possédant la clé privée peuvent contrôler l'intégrité des données. L'avantage de ces algorithmes est qu'ils sont beaucoup plus rapides que les signatures asymétriques.
- PRNG : le PRNG est un outil indispensable à la bonne fonction de certaines briques cryptographiques. Il s'occupe de générer les nombres aléatoires nécessaires à certains algorithmes. La bonne conception et utilisation d'un PRNG est critique pour la sécurité d'un système à cryptographie asymétrique.

L'assemblage de ces dernières briques permet de sécuriser une application. Un protocole tel que TLS emploie tous ces mécanismes pour s'assurer que la connexion est confidentielle (échange de clé, chiffrement), que la connexion n'est pas altérée (signatures, HMAC), et que l'on est bien en train de dialoguer avec le bon serveur (validation de chaînes de CA, non-répudiation). La cryptographie seule ne propose malheureusement pas de solution aux problèmes de disponibilité.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

2 Attaques contre les systèmes informatiques

De nombreuses attaques existent contre les systèmes d'information et sont documentées. On recense deux catégories de vulnérabilités : les vulnérabilités de conception et les vulnérabilités d'implémentation.

Les vulnérabilités de conception sont plus dangereuses, car elles sont difficiles à réparer, et cela implique généralement de forcer la sortie d'une nouvelle version incompatible avec les anciennes. L'absence d'intégrité et de non-répudiation dans le protocole DNS sont des failles de conception qui sont réparées dans DNSSEC. On sait à quel point DNSSEC a du mal à s'imposer et le temps que mettra cette nouvelle version à être généralisée.

Les vulnérabilités d'implémentation sont moins critiques car elles sont « corrigeables » à moindre coût et ne touchent qu'un nombre limité de produits en même temps, mais sont malheureusement plus nombreuses. Lorsqu'une vulnérabilité d'implémentation est découverte, le bug doit être corrigé et la solution diffusée de manière sécurisée vers les systèmes touchés.

Les failles qui entourent la cryptographie dans les applications sont un mélange de ces deux catégories. De manière générale, une vulnérabilité est de conception lorsque l'on n'utilise pas la cryptographie correctement. Une faille est d'implémentation lorsqu'un bug de langage (C, Java, faute de frappe, mise en commentaire de code critique, ...) empêche la crypto de faire correctement son travail. La suite de cet article est divisée entre ces deux catégories.

3 Vulnérabilités de conception

3.1 Le péché capital

La première faute impardonnable est de développer soi-même des algorithmes de crypto. À moins que vous ne soyez un expert reconnu, c'est la recette parfaite pour commettre des erreurs irréparables. Les experts qui ont développé DES, RSA, AES et autres protocoles ont mis des années de travail à profit pour écrire l'algorithme, et leurs productions ont mis encore plus de temps à être relues, vérifiées et attaquées par leurs pairs, avant d'être promues comme standards internationaux. Au cours de ce *peer-review*, de nombreux candidats sont abandonnés en raison de faiblesses découvertes.

Les raisons qui poussent certains à développer leur propre crypto sont la peur que les algorithmes connus soient cassables (voire backdoorés), l'absence d'outils automatiques pour casser leur code et la prétendue sécurité de leur algorithme qui est secret. La première crainte relève de la paranoïa : le choix dans les algorithmes sûrs est assez important pour éviter les *backdoors*, et sera de toute façon plus sûr que de la fabrication maison. L'absence d'outils automatiques (et *hardware*) est un point valide, mais modifier légèrement le résultat d'un algorithme connu peut suffire pour contrer ce risque (au risque par contre d'affaiblir considérablement cet algorithme). La troisième crainte est une violation du principe de Kerckhoffs, qui veut que la sécurité d'un système réside entièrement dans les paramètres variables (càd la clé) et pas dans l'aspect secret de l'algorithme. Le jour où l'algorithme sera révélé, la sécurité de votre système risque de tomber très vite.

N'imaginez pas un instant que cela n'arrive jamais dans l'industrie. Récemment, les deux algorithmes GMR-1 et GMR-2 utilisés dans la téléphonie par satellite ont été analysés et complètement cassés. **[GMR]**

3.2 Chiffrement symétrique

3.2.1 Utilisation de cryptographie faible

De nombreux algorithmes connus sont trop faibles pour être utilisés aujourd'hui, soit parce qu'ils sont cassés ou parce que l'espace des clés est trop faible :

- DES dans sa version classique n'a que 56 bits de sécurité, et est aujourd'hui cassable avec un peu d'efforts.
- Les algorithmes construits sur le chiffre de César et autres xor sont à proscrire car ils sont cassables trivialement.

Un exemple de cryptographie faible est la carte MIFARE (que vous utilisez peut-être pour rentrer dans vos bureaux), qui n'utilise que 48 bits de sécurité. **[MIFARE]**

3.2.2 Non-utilisation de modes

En cryptographie symétrique, les primitives (AES, DES, ...) sont rarement utilisées telles quelles mais sont généralement employées dans des modes. Les modes les plus connus sont ECB, CBC et CTR.

Le mode ECB est le plus simple. Chaque texte clair est chiffré puis émis en sortie sans transformation. C'est donc le mode utilisé par défaut lorsque l'on ne spécifie aucun mode. Les faiblesses de ce chiffrement sont que les morceaux qui reviennent régulièrement (par exemple le *padding*, des signatures, valeurs magiques ou tailles) sont toujours chiffrés de la même manière.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Il est donc possible d'exploiter des attaques par texte clair connu ou par texte clair choisi, et de se constituer un dictionnaire clair/chiffré (d'où le nom de ce mode, *Electronic CodeBook*). Ce mode n'assure pas toujours la confidentialité car il est parfois possible de deviner le contenu uniquement en observant les répétitions.

3.2.3 Utilisation de la même clé sur plusieurs messages chiffrés avec un Stream Cipher

Les *stream ciphers* sont des chiffrements qui visent à se comporter comme un *one-time pad* : à partir d'une graine secrète (la clé), une suite de bits est générée, respectant certaines propriétés mathématiques (la sortie « semble » être aléatoire), et dont la connaissance ne permet pas de récupérer la clé. Chaque bit du message clair est ensuite combiné (xor) avec chaque bit de la sortie du stream cipher, afin de générer le texte chiffré.

Ce chiffrement possède la même faiblesse que le one-time pad : il faut absolument empêcher que deux textes clairs différents soient chiffrés avec la même clé. Dans le cas contraire, on peut extraire facilement la combinaison (xor) entre deux textes clairs via les textes chiffrés.

Wright a documenté dans son livre « Spycatcher » **[WRIGHT]** le cas d'espions russes qui utilisaient des one-time-pad pour sécuriser leurs communications. Ce type de chiffrement est en théorie inviolable, mais dans ce cas, les espions russes se sont mis à recycler des anciennes clés, ce qui a rendu leurs anciens et nouveaux messages déchiffrables.

Prenons deux messages, « Attaque à l'aube » et « Attaque au soir », et chiffrons ces deux messages avec la même clé aléatoire. Le résultat sera deux textes chiffrés, dont la première moitié est totalement identique, car les caractères sont tous chiffrés avec le même flux. Chaque bit des textes chiffrés dans la seconde moitié est lié avec l'autre par un bit de clé. Cela veut dire que si on devine un bit de texte clair dans un des deux textes chiffrés, on obtient automatiquement le bit de texte clair de l'autre. En pratique, il est très facile de retrouver les valeurs ascii et d'obtenir un texte clair qui a du sens pour chaque message avec une clé unique.

3.2.4 Mauvaise utilisation de RC4

RC4 est un stream cipher très simple et très rapide. Cependant, des attaques ont été trouvées contre certaines utilisations de RC4 :

- Les premiers octets générés par RC4 sont biaisés statistiquement et devraient être abandonnés sans être utilisés. Certains auteurs suggèrent que 1024 octets sont suffisants.
- Une attaque permet de déterminer la clé lorsqu'un grand nombre de messages sont chiffrés avec une clé comportant une partie fixe et une partie mobile concaténée à cette clé. **[FMS]**
- Un flux de données généré par RC4 est distinguable d'un flux de données aléatoires avec seulement 2^{25} octets produits par RC4. **[PRENEEL]**

Ces deux faiblesses ont été exploitées avec succès sur le protocole WEP qui se casse aujourd'hui en quelques minutes. De manière générale, RC4 est un chiffrement à éviter en raison de ces différentes vulnérabilités et de ses faiblesses statistiques.

Malheureusement, il n'existe pas d'algorithme aussi simple que RC4 qui puisse le remplacer. Si l'utilisation d'un stream cipher est une obligation, choisissez de préférence une implémentation libre de AES en mode CTR, tout en gardant à l'esprit que les clés et IV doivent changer à chaque utilisation. Une autre possibilité est d'utiliser un des candidats du projet eSTREAM **[ESTREAM]**.

3.2.5 Mauvaise utilisation de modes

Comme vu plus haut, d'autres modes existent et servent à des applications particulières. Ces modes aident non seulement à améliorer la sécurité générale du chiffrement, mais répondent aussi à certains besoins opérationnels :

- Le mode CBC est un mode dans lequel chaque bloc est d'abord combiné au moyen d'un xor avec le texte chiffré précédent, avant d'être chiffré normalement par le *block cipher*. Cela a l'avantage qu'en cas de corruption dans le texte chiffré (par exemple, erreur de transmission), seuls deux blocs sont corrompus après déchiffrement : le bloc corrompu et le suivant. La propagation des erreurs est telle qu'une inversion d'un bit produit un bloc complètement faux ainsi qu'un bit inversé dans le bloc suivant. Dans certains cas, le simple fait de pouvoir inverser un bit précis dans un bloc de données est suffisant pour monter une attaque active (voir paragraphe 3.4.4 - *CBC padding oracle attack*).

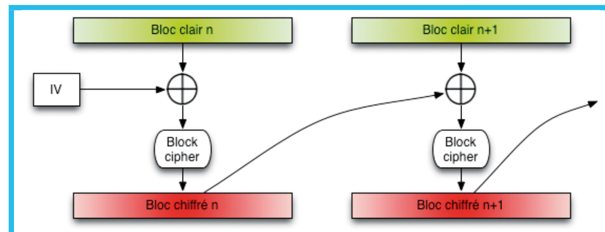


Figure 1 : Chiffrement en mode CBC. Le texte clair est d'abord xoré avec le bloc chiffré précédent (ou l'IV) avant d'être chiffré par le block cipher.

- Le mode CTR est un mode émulant un stream cipher : le block cipher de base est utilisé pour chiffrer un compteur qui s'incrémente à chaque

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

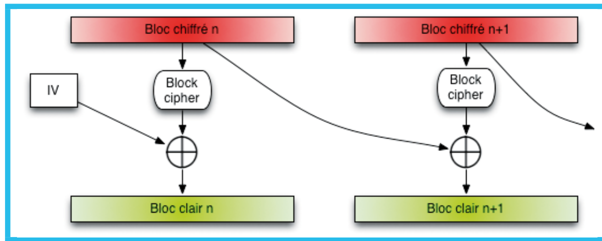


Figure 2 : Déchiffrement en mode CBC. Le texte chiffré est déchiffré par le block cipher et est ensuite xoré au bloc chiffré précédent (ou l'IV).

utilisation, lui-même concaténé à un *nonce* (extrait de l'IV). Le résultat de cette opération est ensuite « xoré » au texte clair pour fournir un texte chiffré. Ce mode propose les avantages des stream ciphers avec la sécurité de block ciphers connus comme AES, mais apporte avec lui les dangers des stream ciphers : il ne faut absolument jamais réutiliser la clé. Il est aussi indispensable de connaître la taille du compteur et de ne jamais dépasser la limite : le compteur reprendrait depuis zéro, et l'algorithme reproduirait la même sortie. Il est aussi important d'utiliser un IV différent lors de chaque utilisation avec une nouvelle clé, afin d'éviter de permettre des attaques potentielles sur le block cipher.

L'avantage principal de ce mode est qu'il est entièrement parallélisable et pré-calculable, ce qui permet d'obtenir de bonnes performances tant en débit qu'en latence.

- D'autres modes existent (PCBC, CFB, OFB, XTS, ...), mais aucun ne fournit de solution définitive à la sécurité des données. Il est impératif de lire les avantages et inconvénients de chaque mode avant de faire son choix.

3.3 Chiffrement asymétrique

Lorsque vous développez une application cryptographique, il y a de grandes chances que vous ayez besoin de chiffrement asymétrique. Historiquement, la plupart des failles contre les chiffrements asymétriques sont des failles d'implémentation. Il y a néanmoins quelques erreurs à ne pas réaliser lorsque l'on développe une application asymétrique.

3.3.1 Taille de clé trop faible

Il n'est pas rare qu'un système de chiffrement permette de générer des clés de différentes tailles, allant de 512 bits à 4096 bits pour du RSA. À l'heure actuelle, les clés privées RSA de taille inférieure à 768 sont considérées comme non sûres, le nombre RSA-768 ayant été factorisé en 2009 **[RSA]**. La taille minimum absolue d'une clé RSA devrait être de 1024 bits, et lorsque cela est possible, une clé de 2048 bits résistera mieux aux avancées technologiques au fil du temps. Lorsque vous choisissez une taille de clé, cette taille doit être adaptée à la sécurité voulue mais aussi au temps de vie de la clé, ainsi que du temps de vie des données qu'elles protègent. **[KEYLEN]** est une bonne référence sur les temps de vie admis pour les types de clés.

En ce qui concerne les clés de type courbes elliptiques, le tableau suivant établit l'équivalence admise entre les tailles de clés RSA et les clés ECC de type NISTECC et BPECC **[IBM]**.

RSA	NISTECC	BPECC
1024 bits	192 bits	160 or 192 bits
2048 bits	224 bits	224 bits
3072 bits	256 bits	256 or 320 bits
7680 bits	384 bits	384 bits
15360 bits	521 bits	512 bits

3.3.2 Signature de blocs arbitraires

Une attaque est possible dans un protocole ne faisant pas assez de vérifications d'intégrité et de validité de données avant de les signer. Dans cette attaque, la partie offensive envoie un bloc de données ou un document à une autorité chargée de le signer. Cet attaquant réutilise ensuite cette signature avec le même bloc de données, mais dans un autre contexte.

Par exemple, si un attaquant arrive à faire signer une CSR (*certificate signing request*) pour un certificat invalide (par exemple un certificat de CA, ou un certificat *wildcard*) par manque de contrôle chez le CA, cet attaquant peut abuser de la signature pour lui-même devenir un CA. Ce type d'attaque peut se réaliser via une erreur d'implémentation au niveau du CA (le certificat à signer n'est pas compris de la même manière par le vérificateur du CA que par les implémentations SSL courantes), ou via une attaque par ingénierie sociale (l'attaquant convainc un membre du personnel de signer le certificat).

Un exemple du premier type de vulnérabilité est la faille présentée à *Blackhat* 2009 **[NULBYTE]** dans laquelle les présentateurs ont acheté un certificat pour un nom de domaine contenant un octet nul entre le nom qu'ils voulaient usurper et leur propre nom de domaine : « www.paypal.com\0.attacker.net ». Le *middleware* du CA était programmé pour vérifier que la partie droite était un nom de domaine valide, alors que la plupart des implémentations SSL du marché (programmées en C) ne faisaient la vérification que de la partie gauche (les routines de gestion des chaînes de caractères en C s'arrêtant aux délimiteur nul).

Une autre manière d'utiliser cette attaque est d'exploiter les vulnérabilités des algorithmes de hachage, tels que MD5, et de générer deux versions d'un même message, qui possèdent le même *hash* md5.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

L'attaquant envoie le premier message, qui est signé par le CA, tandis qu'il envoie le second à la victime. La signature est dans ce cas valable pour les deux messages. Cette faille a été exploitée pour créer de faux certificats CA X.501 **[X501]**, et une attaque de ce type a été implémentée dans le virus Flame pour créer un faux certificat Microsoft Update. **[FLAME]**

3.3.3 Padding non standard

L'algorithme RSA est bien connu. Sa sécurité repose sur des concepts mathématiques : casser RSA revient presque à factoriser un produit de deux grands nombres premiers n (car si on peut factoriser n, on peut casser RSA, l'inverse n'étant pas vrai).

Cependant, cette propriété s'appuie sur l'hypothèse que chaque message n'est jamais chiffré qu'une seule fois, et que ce message n'est pas attaquable par force brute. Imaginons que vous vouliez envoyer un numéro de téléphone chiffré par RSA. Vous prenez les paramètres (e,n) de votre correspondant et générerez le nombre m, qui contient une représentation numérique de votre numéro de téléphone (qui après tout n'est qu'un nombre). Vous envoyez $c = m^e \pmod n$ à votre correspondant.

« e » est l'exposant. La valeur la plus souvent utilisée est 65537 pour ses propriétés binaires (seulement 2 bits à 1), mais parfois cette valeur est plus faible (3).

« n » est le produit de deux nombres premiers p et q.

La première chose que l'on peut remarquer est que si la taille de m (en nombre de bits) est inférieure à la taille de n divisée par e, une simple opération de racine (cubique par exemple) est suffisante pour extraire le message, car m^e est plus petit que n.

La seconde attaque (en imaginant que e est trop grand pour l'attaque précédente) est une attaque par force brute. Puisque l'attaquant possède les mêmes paramètres e et n, il peut lui-même créer tous les messages chiffrés correspondant aux messages m possibles, ce qui est très facile vu le nombre réduits de numéros de téléphone.

Une troisième attaque est possible lorsque le même message est envoyé à plusieurs correspondants et que l'exposant e des trois destinataires est 3. **[ROOTLABS]** Dans cette attaque, le correspondant crée et envoie les messages

$c_1 = m^e \pmod{n_1}$

$c_2 = m^e \pmod{n_2}$

$c_3 = m^e \pmod{n_3}$

En utilisant le théorème des restes chinois, il est possible de calculer le nombre $c = m^e \pmod{n_1\ n_2\ n_3}$. Si e =3, m^e est forcément plus petit que $n_1\ n_2\ n_3$ car m est plus petit que ces nombres. On obtient donc la valeur m avec une simple racine cubique du nombre c.

Ces différentes attaques sont possibles à cause de l'absence de padding. La norme PKCS#1 spécifie comment former les messages RSA (en prenant compte du message initial et de données aléatoires) et comment présenter la sortie de façon non ambiguë. **[PKCS1]**

3.4 Contrôle de l'intégrité

Un principe fondateur des RFC et de l'Internet est le principe de robustesse : « Soyez conservateur dans ce que vous envoyez, soyez libéral dans ce que vous recevez ». Bien que ce principe ait aidé la constitution de piles TCP compatibles, il a aussi contribué à l'introduction de nombreux bugs et problèmes de sécurité (et c'est encore pire dans le monde web).

En cryptographie, ce principe est à proscrire : « soyez conservateur et c'est tout ! ». Le contrôle de l'intégrité des données reçues ou stockées est donc très important, et il est souvent utile de détecter une falsification des données au plus tôt, avant que ces données ne soient utilisées ailleurs. Ce contrôle peut avoir plusieurs formes : contrôle de la syntaxe (validation d'un schéma XML ou de la structure ASN.1), contrôle sémantique (les données ont-elles un sens ? Respectent-elles les règles « métier » ?) et un contrôle cryptographique (ces données ont-elles été signées par un interlocuteur de confiance ?).Voici quelques erreurs à éviter concernant le contrôle de l'intégrité.

3.4.1 Non-vérification de l'intégrité

Cela tombe sous le sens, mais les données qui viennent de l'extérieur doivent être validées, même si elles ont été chiffrées. Dans le cas courant d'une session SSL (par exemple en https), si aucun contrôle d'intégrité n'était présent, un attaquant pourrait injecter ou remplacer des données (par du bruit, vu qu'il ne peut pas chiffrer son *payload*), ou pourrait rejouer des blocs déjà passés sur le réseau (et donc en contrôlant de manière approximative le contenu déchiffré).

3.4.2 Vérification tardive de l'intégrité

Le meilleur moment pour vérifier l'intégrité d'un paquet chiffré est celui qui se rapproche le plus possible de la réception, et celui qui permet à des données corrompues de traverser le moins de code possible. La raison est la suivante : si le paquet corrompu est interprété (par exemple par un parser) avant d'être considéré comme mauvais, il y a des risques qu'il génère des erreurs dans diverses parties du code. Par exemple, dans une chaîne telle que :

Déchiffrement → décompression → contrôle intégrité

on court le risque d'obtenir une erreur de décompression, qui n'est pas directement identifiable comme étant

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

une erreur de sécurité. Ce problème est encore plus grave si l'application renvoie un message d'erreur à l'attaquant, lui expliquant quelle partie du message envoyé était invalide. L'attaque « CBC padding oracle », présentée ci-après, fait partie de cette classe de bug. Le protocole SSH, utilisant des chiffrements de type CBC, est vulnérable à ce type de faille, car le champ « taille » des paquets est utilisé pour lire la suite du paquet sans vérification. **[SSH]**

3.4.3 Contrôle de l'intégrité faible ou insuffisant

Le contrôle de l'intégrité doit être réalisé par un algorithme dont la sécurité est reconnue (par exemple SHA-1 ou SHA-2) et ne doit pas être falsifiable par un attaquant. Il peut être intéressant dans ce cas d'utiliser des fonctions de type HMAC, qui nécessitent une clé symétrique pour pouvoir être utilisées.

Les fonctions du type CRC32, bien qu'utiles au niveau transmission pour découvrir des erreurs matérielles, ne sont pas assez sûres pour des besoins cryptographiques. SSH dans sa première version est à proscrire pour cette raison (entre autres).

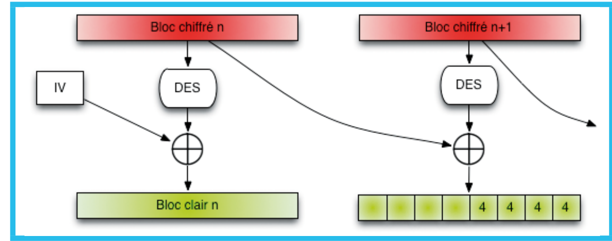
3.4.4 CBC padding oracle attack

L'attaque « CBC padding oracle » a été décrite pour la première fois en 2002 **[CBCPADDING]** et découverte dans différentes applications serveur, dont ASP.NET et Ruby on rails. Bien qu'affectant les protocoles chiffrés et utilisant le mode CBC, cette attaque est un bon exemple des risques encourus lorsqu'aucun contrôle d'intégrité n'est présent.

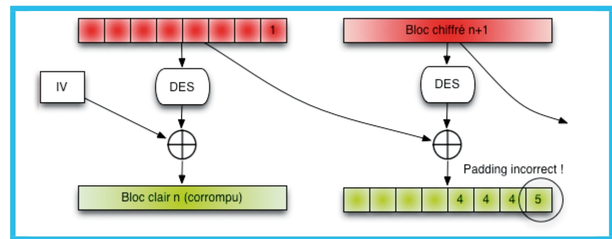
Lors du chiffrement d'une certaine quantité de données en utilisant le mode CBC (le type de block cipher n'a pas d'importance), la norme PKCS #5 est utilisée pour décrire comment les données sont chiffrées et déchiffrées. Cette norme indique aussi ce qu'il faut faire lorsque la quantité de données à chiffrer n'est pas un multiple de la taille de bloc (64 bits pour DES). Pour remplir ce padding, une possibilité aurait été de laisser les octets non utilisés à zéro, mais la norme PKCS #5 a choisi de remplir chaque byte du dernier bloc par le nombre de blocs de padding. Cela devient un problème lorsque l'attaquant peut envoyer des blocs chiffrés et interpréter les messages d'erreur en retour.

Prenons un exemple dans lequel 12 octets sont transmis en utilisant DES-CBC. Les 4 derniers octets du deuxième bloc sont initialisés à 4 et sont utilisés dans le chiffrement. Ce deuxième bloc est d'abord xoré avec le texte chiffré du bloc précédent avant d'être chiffré par DES.

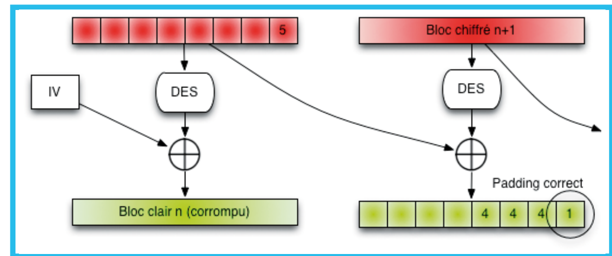
Le déchiffrement suit la procédure inverse : le deuxième bloc est déchiffré avec DES et est ensuite xoré avec le texte chiffré précédent.



Le problème ici est que le texte chiffré précédent est contrôlé par l'attaquant. En modifiant le dernier octet du premier bloc (par exemple en appliquant un xor 1), on est capable de modifier un bit du dernier octet du deuxième bloc. Dans ce cas-ci, le dernier octet devient 5 (4 xor 1) et une erreur de padding est renvoyée à l'attaquant.



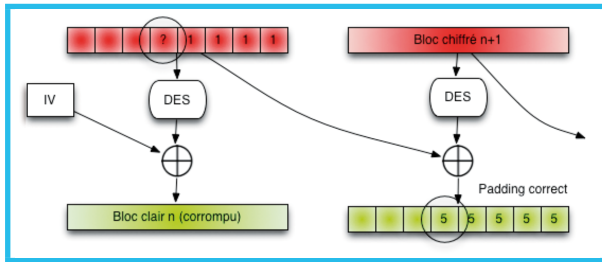
En changeant la valeur xorée à ce dernier octet, on peut finalement tomber sur une valeur de padding valide, c'est-à-dire 1 pour le dernier octet. En effet, le dernier octet n'admet que deux valeurs de padding valide : 1 (unique octet de padding) et 4 (valeur originale). Cette attaque nous permet d'apprendre la taille du padding, égale à (valeur dernier octet) xor (valeur dernier octet ne générant pas d'erreur) xor 1. Nous découvrons que dans notre cas, il y a 4 octets de padding.



L'attaque peut être maintenant généralisée pour découvrir le dernier octet du texte chiffré, c'est-à-dire le quatrième octet du deuxième bloc. Pour ce faire, on doit « xorer » les 4 derniers octets du premier bloc avec la valeur qui nous permet d'obtenir un 5 dans les octets de padding (car nous voulons simuler un padding de 5 octets). Cette valeur est 5 xor 4 = 1.

Nous devons ensuite modifier la valeur qui se trouve dans le 4ème octet du premier bloc, en la « xorant » avec les 255 possibilités, jusqu'à ce qu'on n'obtienne plus d'erreur de padding. À ce moment, nous avons assez d'informations pour déchiffrer la valeur du 4ème octet du deuxième bloc.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



L'attaque est généralisable et permet de déchiffrer le contenu du *cookie*. Voir sur **[ENTROPY]** pour plus de détails.

Afin de se protéger contre ce type d'attaque, plusieurs mesures peuvent être prises :

- Ne pas envoyer d'information à l'attaquant à propos des raisons de l'échec d'ouverture d'une session.
- Ne pas compter sur le padding du chiffrement pour certifier l'intégrité des données, par exemple en ajoutant une signature cryptographique ou un HMAC. Si les données ne rentrent pas dans un bloc complet, utilisez votre propre padding.
- Utiliser comme données de sessions des valeurs aléatoires (*tokens*) enregistrées dans une base de données locale et permettant de récupérer les données de session sans que celles-ci ne transitent par le client (par exemple via une base de données relationnelle, un fichier sur disque ou dans la RAM du serveur web).

3.5 Utilisation de protocoles

Lorsque vous utilisez un protocole de sécurité pour votre application, il est important de se poser quelques questions :

- Quels sont les points forts de ce protocole ?
- Quels en sont les points faibles ?
- Existe-t-il des fonctionnalités du protocole qui nuisent à mon application ?

Les quelques points suivants sont systématiquement à vérifier lorsque l'on implémente ou audite l'utilisation d'un protocole de sécurité tel que SSL.

3.5.1 Non-validation de certificats

Cela peut sembler évident, mais plusieurs logiciels libres **[PIDGIN] [URLLIB2]** (et propriétaires) ont utilisé SSL sans effectuer la moindre validation sur le certificat serveur. Il est important de contrôler si les API client vérifient correctement le certificat, et qu'elles n'acceptent pas de certificat *self-signed* sans signaler d'erreurs. Pour chercher cette faille sur les applications mobiles ou propriétaires, sslstrip **[SSLSTRIP]** peut être un outil très utile.

3.5.2 Origine des certificats dans la chaîne de confiance

En fonction de votre application, les clients et serveurs censés s'interconnecter doivent faire confiance à des certificats CA différents. La liste de certificats des explorateurs courants en contient plus d'une centaine, et la compromission de Diginotar et Comodo **[THREATPOST]** a prouvé que ces CA n'étaient pas inviolables.

Si la sécurité de votre infrastructure est critique, et que les clients censés se connecter sont sous votre contrôle, vous devriez peut-être envisager de déployer votre propre PKI ou de restreindre la liste des CA de vos clients en vous limitant aux CA avec qui vous faites affaire. Après tout, faites-vous confiance à un CA en relation avec le gouvernement chinois ? **[FREEDOMTOTINKER]**

3.5.3 Non-utilisation des protocoles de révocation

Une fonctionnalité importante des PKI est de pouvoir révoquer à tout moment un certificat lorsqu'il y a des soupçons sur la sécurité de celui-ci. Les deux protocoles les plus courants sont OCSP et CRL.

CRL (*Certificate revocation list*) est une liste d'identifiants de certificats révoqués qu'il ne faut plus prendre en compte lors de l'évaluation de la chaîne de confiance. Cette liste est téléchargée par le client, soit lors des mises à jour du logiciel client, soit de manière régulière. Le défaut de cette méthode est qu'elle ne passe pas le facteur d'échelle et que les données peuvent vite devenir volumineuses.

OCSP, quant à lui, est un protocole en ligne, qui permet d'interroger une base de données au sujet du (ou des) certificat(s) qu'il est sur le point d'accepter. Il y a cependant des ennuis liés à la vie privée et des risques de déni de service sur le serveur OCSP (ou sur la route pour y parvenir).

Il est important de vérifier que votre application supporte l'une des deux méthodes et que vous avez une procédure pour révoquer les certificats en cas de besoin.

3.5.4 Attaques par rejeu

Certains protocoles peuvent être victimes d'une attaque par rejeu. Une telle attaque consiste à capturer des paquets envoyés sur le réseau et à rejouer ces paquets entiers ou en partie.

Un exemple de protocole vulnérable à cette attaque est WEP : il n'est pas nécessaire de connaître la clé réseau pour rejouer un paquet. Cette attaque peut

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

être utilisée pour rejouer des paquets ARP *who-has*, qui eux-mêmes génèrent une réponse de la part de la victime. L'augmentation du trafic permet quant à elle d'augmenter les chances de cracker WEP.

Cette attaque peut avoir des répercussions très importantes lorsque la partie du protocole qui peut être rejouée est l'authentification à un service ou l'obtention d'un token d'accès.

La meilleure manière de se protéger contre ce type d'attaques est d'utiliser une clé de session dérivée de données générées par les deux parties (par exemple en utilisant l'échange de clés Diffie-Hellman) et de numéroter chaque paquet sortant avec un numéro de séquence infalsifiable.

4 Vulnérabilités d'implémentation

Il est difficile de généraliser les vulnérabilités d'implémentation, car très souvent elles sont dépendantes de la plateforme et du langage utilisés. Cependant, voici quelques erreurs courantes.

4.1 PRNG

Le PRNG (*Pseudo Random Number Generator*) est une partie critique de chaque système cryptographique. De nombreux algorithmes (tels que DSA, Diffie-Hellman) prennent pour hypothèse que les nombres aléatoires le sont vraiment.

4.1.1 Utilisation de PRNG faibles

Un audit de code ne serait pas complet sans trouver une référence à `random()`, le générateur de nombres aléatoires de la bibliothèque C (ou son équivalent dans le langage utilisé). Ce PRNG est construit sur une formule mathématique simple et très facile à casser. Cela veut dire qu'avec un nombre très limité de données, on est capable de deviner les prochains nombres qui vont sortir. Cela est encore pire lorsque la graine (*seed*) utilisée pour initialiser le générateur est le temps Unix, encore plus facile à deviner.

Le générateur de nombres aléatoires utilisé pour tout ce qui est tokens, cookies aléatoires, mot de passe générés automatiquement et protocoles de cryptographie doit être sécurisé. La méthode la plus efficace est encore d'utiliser une bibliothèque cryptographique qui permet d'obtenir un nombre arbitraire d'octets aléatoires. Sous UNIX, une autre solution est d'utiliser les deux fichiers *device* `/dev/urandom` et `/dev/random`.

Le fonctionnement habituel d'un PRNG est le suivant : Le PRNG possède un état, contenant lui-même une estimation de l'entropie disponible, ainsi qu'un tampon de données supposées aléatoires. Lors de la demande d'une certaine quantité de données par l'application, le PRNG peut aller se « recharger » auprès de l'OS, générer une sortie à partir de son tampon interne (par exemple via un *hash* ou un chiffrement) et réévaluer son entropie.

Certains PRNG (comme `/dev/urandom`) fonctionnent en mode non bloquant : ils fournissent immédiatement la quantité de données aléatoires à la demande, mais la qualité (entropie) diminue si l'utilisation est intensive et que le générateur n'a pas l'occasion de se recharger. D'autres PRNG (comme `/dev/random` sous Linux) garantissent l'entropie de la production, mais ne garantissent pas que ces données soient disponibles lors de la demande (par exemple, la lecture de `/dev/random` bloque si le générateur est vide).

Si votre budget et l'application le permettent, une option est d'obtenir un PRNG hardware, qui permet d'améliorer les performances et la sécurité du PRNG.

4.1.2 Non-vérification des codes d'erreur

Cette remarque s'applique particulièrement aux PRNG, mais est valable pour toutes les API, cryptographiques ou non : il est très important de vérifier les codes d'erreur.

Si l'API permettant de générer des octets aléatoires retourne une erreur au lieu de générer la sortie (par exemple parce que le *pool* d'entropie est vide), il est important de pouvoir le détecter et d'agir en conséquence. Dans le cas contraire, il se peut que votre application utilise des données non initialisées et dont le contenu est prévisible.

4.1.3 État du PRNG déductible

L'une des difficultés lors de l'utilisation d'un PRNG est de tester valablement sa sécurité. Le tristement célèbre bug du PRNG OpenSSL de Debian **[DEBIAN]** montre à quel point une faille même triviale est difficile à trouver. Une étude **[RONWASWRONG]** a mis en évidence qu'une quantité de clés publiques accessibles sur Internet (moins de 1%) partagent des facteurs communs (et donc facilement factorisables). L'hypothèse la plus probable est la faiblesse des PRNG utilisés pour les générer, même si on ignore encore les détails pratiques.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Ce type de risque existe bel et bien si vous utilisez un PRNG, car les PRNG ont tendance à perdre de l'entropie au fur et à mesure qu'on les utilise. Pour minimiser les risques :

- Si votre PRNG offre un mode « normal » et un mode « sécurisé », utilisez le mode normal pour toutes les utilisations de nombres aléatoires moins critiques. La génération de mot de passe, de paires de clés RSA doivent utiliser le mode le plus sécurisé. Idéalement, ce mode devrait permettre au PRNG de générer une erreur ou de patienter si l'entropie disponible est jugée trop faible.

- N'hésitez pas à enrichir l'entropie de votre PRNG s'il le permet en utilisant des informations spécifiques à votre application, variables avec le temps et difficilement prédictibles. Certains PRNG permettent de préciser une estimation de l'entropie que vous ajoutez ; évitez de la surestimer.

- Si votre application est multithreadée, vérifiez que votre bibliothèque cryptographique est *threadsafe* et que les recommandations (pour cette bibliothèque) sont respectées. L'absence de *mutex* pour l'accès au PRNG peut avoir comme conséquence que deux *threads* reçoivent les mêmes données pseudo aléatoires ! Analysez aussi ce que devient l'état du PRNG lorsque votre processus forke (UNIX). Si vous craignez d'être dans ce cas, réinitialisez le PRNG à partir d'une source sûre.

4.2 Cryptographie symétrique

La cryptographie symétrique est généralement mieux comprise que sa sœur asymétrique, probablement parce qu'il y a moins de pièges. Néanmoins, voici deux pièges à éviter.

4.2.1 Déchiffrement de blocs partiels

Si vous implémentez un protocole sécurisé à base de block cipher, il est probable que vous receviez une certaine quantité de données à déchiffrer de la part d'un attaquant potentiel. Il est bon de vérifier que les champs « taille » des paquets entrants soient tous valides, ce qui implique aussi que la taille du texte chiffré doit être un multiple de la taille de bloc de l'algorithme utilisé. Outre les problèmes de padding expliqués plus haut, la plupart des bibliothèques cryptographiques tenteront de lire des blocs complets. Cela peut avoir un impact direct si l'allocation de mémoire pour déchiffrer le paquet a été faite sans prendre en compte le paquet complet.

Concrètement, si vous recevez un paquet de 17 octets chiffré en AES256-CBC et le rentrez tel quel dans les routines d'OpenSSL, il vous sortira un texte clair de 32 octets – qui risque de causer un *overflow*. Ce type de bug est extrêmement difficile à trouver.

4.2.2 Retour sur texte chiffré

Si votre application permet à un attaquant de déchiffrer à la volée des blocs de données, ou inversement de présenter à l'attaquant des données chiffrées qu'il connaît, vous ouvrez la porte à toute une série d'attaques contre la clé ou l'algorithme de chiffrement. Il faut donc bien prendre garde à ne pas laisser de données sensibles (comme des textes clairs ou chiffrés) dans les messages d'erreur (tout comme vous le feriez pour des *stacktrace* Java ou des erreurs SQL).

4.3 Cryptographie asymétrique

Les failles d'implémentation en cryptographie asymétrique sont un domaine de recherche très productif. La plupart des bibliothèques cryptographiques ont souffert dans leur jeunesse d'au moins une attaque sur l'implémentation, et dans la plupart des cas avec comme conséquence l'extraction de la clé privée (sous certaines conditions). Voici les erreurs les plus communes.

4.3.1 DSA/ECDSA avec PRNG faible

DSA est un algorithme de signatures cryptographiques construit sur le problème du logarithme discret dans un groupe fini. ECDSA est un algorithme calqué sur DSA mais utilisant le groupe des courbes elliptiques, ce qui lui donne des propriétés très similaires mais aussi des avantages en terme de complexité des attaques. Cela a pour conséquence que les clés peuvent être plus petites.

La signature d'un message avec DSA et ECDSA implique l'utilisation d'un nombre aléatoire k. Une propriété mathématique de l'algorithme de signature de DSA fait que lorsque deux signatures sur deux messages différents sont réalisées avec le même nombre aléatoire k, la clé secrète est triviale à calculer. Cette propriété a été utilisée pour cracker le DRM de la PlayStation3, utilisant ECDSA et dont plusieurs titres avaient été signés avec le même nombre aléatoire. **[PS3]**

4.3.2 Erreurs dans les constantes

Cela peut paraître anodin, mais la cryptographie utilise de nombreuses constantes, qu'elles soient par convention (par exemple les groupes Diffie-Hellman

définis dans les normes SSH) ou qu'elles fassent partie même des spécifications de l'algorithme (par exemple les S-Box de DES). Il est important de vérifier à ce qu'aucune erreur de transcription ne soit présente. Ce bug de Ruby **[RUBY]** dans la fonction de génération de clés RSA créait de nouvelles clés RSA dont l'exposant e était fixé à 1 (et donc complètement inutile).

4.3.3 Attaques par canaux auxiliaires

Ces attaques sont de loin les plus productives contre la cryptographie asymétrique. Nous avons vu plus haut qu'appliquer à la lettre les formules mathématiques n'est pas suffisant pour garantir la sécurité de RSA : il est important de respecter certaines normes et bonnes pratiques dans la manière de traiter les entrées et les résultats.

Les attaques par canaux auxiliaires s'en prennent quant à elles aux routines mathématiques utilisées pour effectuer des calculs sur des grands nombres. Les opérations les plus courantes (dont l'exponentiation modulo n) doivent absolument être optimisées afin de fournir un résultat en temps raisonnable. Malheureusement, cela implique que le temps et l'énergie utilisés pour le calcul de RSA sont loin d'être constants, en fonction du message et du nombre de bits à un dans la clé privée.

Voici quelques attaques dont les plus connues sont les attaques par timing et par consommation électrique.

- Dans l'attaque par timing, l'attaquant envoie une quantité de requêtes au serveur, le forçant à effectuer des opérations RSA. Sur certaines implémentations utilisant les optimisations standards, l'attaquant peut obtenir des informations sur la clé secrète en fonction du temps pris pour effectuer l'opération RSA. En compilant de nombreuses requêtes, il peut aller jusqu'à obtenir la clé complète.

- Dans l'attaque par consommation électrique, l'ordinateur ou l'application embarquée contenant la clé secrète est placé(e) sous un *monitoring* très strict. Lors du chiffrement par RSA, on distingue de manière très nette les différentes opérations réalisées par le processeur en fonction des algorithmes en train de tourner, et cela donne des informations précises sur la clé secrète.

- Sur des systèmes partagés, un monitoring précis de l'utilisation processeur ou du cache par d'autres processus peut être utilisé pour deviner la clé privée.

- Le son et les radiations électromagnétiques générés par un ordinateur peuvent également être exploités.

Pour se protéger contre ce type d'attaques, il est indispensable d'utiliser des implémentations sécurisées et bien testées des algorithmes les plus courants. Les moyens techniques à mettre en œuvre pour se protéger ne sont pas triviaux et sortent largement du cadre de cet article. Ils consistent généralement à brouiller les pistes en tentant de rendre les algorithmes le plus constant possible. Les implémentations diffusées librement (par exemple OpenSSL) ont déjà corrigé des attaques par timing, mais sont probablement vulnérables aux autres attaques. Protéger des versions embarquées (matérielles) contre ce type d'attaques est un travail de spécialistes.

Il est également important de noter que ces attaques prennent principalement pour cible la cryptographie asymétrique, mais que les autres briques de base peuvent aussi être analysées sous cette loupe.

Conclusion

La recette la plus efficace pour produire une application crypto « sûre » est à mon avis la suivante :

- Utilisez des algorithmes sûrs et reconnus, ainsi que des implémentations sûres et relues.

- Évitez de réinventer la roue autant que possible – il vaut mieux utiliser des outils conçus par des spécialistes si vous en avez la possibilité, car ces outils viendront avec une documentation sur leurs forces et leurs faiblesses.

- Soyez strict sur ce que vous recevez de l'extérieur. Validez l'intégrité des données à chaque fois que cela est possible et au plus tôt. En cas de problème, enregistrez la défaillance mais ne donnez aucun détail à l'attaquant.

- Le Web est inondé d'informations sur la cryptographie. Avant d'utiliser un protocole ou un algorithme qui ne vous est pas familier, recherchez quelles sont ses forces et faiblesses, et quelles sont les meilleures manières de (ne pas) l'utiliser.

- Aujourd'hui, la sécurité de la cryptographie asymétrique est basée sur la difficulté de la factorisation de grands nombres. Que se passera-t-il lorsque l'ordinateur quantique sera une réalité ?

- N'hésitez pas à faire revoir votre architecture, *design* et implémentation par des experts indépendants. Avec leur recul et leur expérience, ils pourront pointer des faiblesses qui n'apparaissent pas au premier regard.

- Faites de la veille en sécurité. Informez-vous des nouveautés dans le domaine de la cryptographie (nouvelles attaques sur SSL, SSH, ...). Un algorithme ou protocole peut être cassé du jour au lendemain, apprenez des erreurs des autres et concevez des systèmes adaptables rapidement. ■

Les références de cet article sont disponibles sur : http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6/2

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

REVUE D'ATTAQUES DU PROTOCOLE TLS ET DE L'IMPLÉMENTATION OPENSSL

Loïc Ferreira - loic.ferreira@orange.com
Ingénieur en cryptographie. Orange Labs – Applied Cryptography Group

mots-clés : TLS / OPENSSL / CRYPTANALYSE / CANAL AUXILIAIRE / CHIFFREMENT RSA / MODE CBC

La mise en œuvre pratique de mécanismes cryptographiques suppose tout d'abord de les concevoir, de les implémenter et de garantir une utilisation sûre. Au cours de chacune de ces étapes, des difficultés de natures diverses peuvent surgir avec un impact direct sur la sécurité du mécanisme considéré. Mais l'expérience des concepteurs ou le nombre des développeurs ne suffit pas toujours à éviter ces écueils. Illustration avec le protocole TLS et l'implémentation OpenSSL.

1 Description du protocole TLS

Le protocole TLS est constitué d'un ensemble de « sous-protocoles » parmi lesquels le protocole *Record* et *Handshake*.

Le protocole Handshake a pour tâches de négocier les algorithmes cryptographiques, d'échanger les valeurs pour la dérivation des clés de session et d'authentifier le serveur (et, optionnellement, le client).

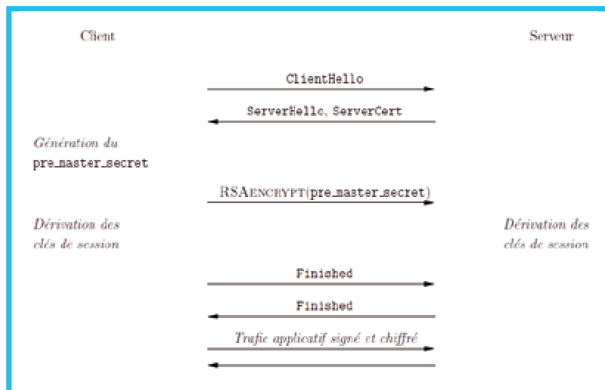


Figure 1 : Description simplifiée du Handshake TLS avec RSA

Le rôle du protocole Record est d'assurer la confidentialité et l'intégrité des données applicatives notamment (le protocole est également sollicité au cours du protocole Handshake).

Le principal mode utilisé au cours du Handshake est sans doute le chiffrement RSA : le client génère pseudo-aléatoirement une valeur appelée `pre_master_secret`. Puis cette valeur est chiffrée avec la clé publique RSA du serveur (cf. Figure 1). Un autre mode d'échange consiste à utiliser le protocole Diffie-Hellman (DH). Dans ce cas, le secret DH négocié entre le client et le serveur fournit le `pre_master_secret` (cf. Figure 2).

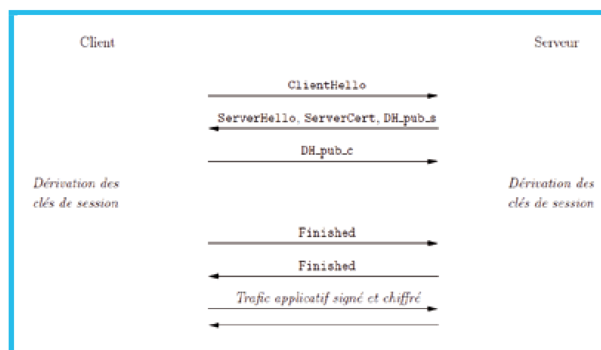


Figure 2 : Description simplifiée du Handshake TLS avec DH

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

À partir du `pre_master_secret` et de deux valeurs publiques générées pseudo-aléatoirement, l'une par le client (`client_random`), l'autre par le serveur (`server_random`), des clés de session sont dérivées pour le chiffrement et la signature des données applicatives.

À l'issue de cette négociation de clé, le message `Finished` est transmis pour sanctionner le bon déroulement de l'échange. Ce message constitue une empreinte cryptographique de tous les messages de la négociation de clé échangés précédemment de part et d'autre. C'est le premier message de la session construit à partir du secret `master_secret` dérivé entre les deux correspondants.

Note

Une description plus détaillée du protocole peut être trouvée dans [8, 9].

Le protocole a été implémenté sous la forme de plusieurs bibliothèques cryptographiques dont la populaire OpenSSL. TLS et OpenSSL sont des objets en évolution régulière. Ces modifications sont motivées par la prise en compte de nouveaux algorithmes correspondant à l'état de l'art (par exemple l'AES pour le chiffrement du trafic applicatif ou les opérations sur courbe elliptique pour la négociation des clés de session). Mais ces changements correspondent également à des correctifs destinés à contrecarrer des attaques de natures différentes.

2 Attaques par canaux auxiliaires

2.1 Gestion des erreurs dans le déchiffrement RSA

2.1.1 Description

Le mode le plus couramment utilisé dans TLS consiste à chiffrer le `pre_master_secret` avec la clé publique RSA du serveur. Le standard utilisé pour ce chiffrement correspond à PKCS #1 v1.5. Le chiffrement d'un message *data* selon ce standard se fait de la manière suivante. Soit k la taille en octet du module RSA et $|data|$ la taille en octet de *data*. Un champ *padding* de $k–3–|data|$ octets non nuls est généré et le bloc m décrit ci-dessous est construit :

$$m = 00\ 02 \mid padding \mid 00 \mid data$$

Puis ce bloc est chiffré avec RSA : $c = m^e \bmod n$.

Le padding a une taille dépendant de la taille de data, néanmoins *padding* a une taille minimale de 8 octets (par conséquent la taille de *data* ne doit pas dépasser k–11 octets). Ainsi, le format de $m = P_0 \mid \ldots \mid P_{k-1}$ est le suivant :

- P_0 = 0x00 ;
- P_1 = 0x02 ;
- P_2 à P_9 : non nul ;
- au moins un octet entre P_{10} et P_{k-1} est nul.

Que se passe-t-il si le message m' obtenu par le serveur après le déchiffrement ne correspond pas au format attendu ? Le serveur envoie au client une alerte spécifique `decode_error` pour signaler cette erreur de format.

Lors de la conférence *Crypto* 1998, Daniel Bleichenbacher a présenté une attaque exploitant ce bit d'information et permettant de retrouver le clair d'un message chiffré avec RSA PKCS #1 v1.5 (*i.e.* : le `pre_master_secret` dans le cas de TLS) **[15]**.

Le principe de l'attaque est le suivant : l'attaquant choisit une valeur s et transmet la valeur $cs^e \bmod n = (ms)^e \bmod n$ à un oracle de déchiffrement (par exemple un serveur TLS). Si l'oracle retourne un message signalant une erreur de format lors du déchiffrement de cs^e, cela signifie que ms ne correspond pas au format d'un bloc clair PKCS #1. Dans le cas contraire, l'attaquant sait que ms respecte ce format.

La chaîne $m = P_0 \mid \ldots \mid P_{k-1}$ a pour valeur $P_0 \times 256^{k-1} + P_1 \times 256^{k-2} + \ldots + P_{k-1} \times 256^0$. Si m est au format PKCS #1, alors nécessairement $P_0 = 0$ et $P_1 = 2$. Les autres octets P_2 à P_{k-1} peuvent avoir (presque) toutes les valeurs entre 0 et 255. Par conséquent, on peut borner les valeurs possibles de m : un message m au format correct PKCS #1 est compris entre :

$$00\ 02\ 00 \ldots 00$$

et

$$00\ 02\ FF \ldots FF$$

Autrement dit : $2 \times 256^{k-2} \leq m < 3 \times 256^{k-2}$. Soit $B = 256^{k-2}$, un message m au format correct vérifie donc $2B \leq m < 3B$.

Dans le cas où le déchiffrement RSA s'est correctement déroulé, la réponse de l'oracle (plus précisément : l'absence de message d'erreur relatif au format de ms) permet à l'attaquant de savoir que

$$2B \leq ms \bmod n < 3B$$

Il en découle qu'il existe une valeur entière r telle que

$$2B \leq ms–rn \leq 3B–1$$

d'où

$$(2B+rn) / s \leq m \leq (3B–1+rn) / s$$

Ainsi, l'intervalle des valeurs possibles pour m a été réduit : on passe d'un intervalle $[2B, 3B–1]$ à un intervalle $[a, b]$ où $a = \max(2B, (2B+rn) / s)$, $b = \min(3B–1, (3B–1+rn) / s)$.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

En répétant ce procédé, on restreint graduellement l'intervalle jusqu'à trouver la valeur *m*. Le principe de l'attaque consiste donc à rechercher *m* de manière exhaustive. Néanmoins, la recherche ne porte pas de manière séquentielle sur chaque bit de *m*. L'information sur le format incorrect du déchiffré permet d'éliminer à chaque étape de l'attaque tout un ensemble de valeurs incorrectes.

2.1.2 Mise en œuvre

L'attaque de Bleichenbacher s'applique de manière générale à PKCS #1 indépendamment du protocole mettant en œuvre ce mécanisme de chiffrement. À titre d'exemple, l'attaque menée pour un module RSA 1024 bits aboutit après 2^{20} appels à l'oracle de déchiffrement.

Vlastimil Klima, Ondrej Pokorny et Tomas Rosa **[16]** ont appliqué une variante de l'attaque de Bleichenbacher à TLS dans le cas d'un échange de clé par RSA. En exploitant la présence du numéro de version dans le `pre_master_secret`, ils parviennent à récupérer cette valeur secrète en moins de 55 heures de connexion avec le serveur TLS (pour RSA 1024). La connaissance du `pre_master_secret` permet à l'attaquant de calculer les clés de session et donc d'avoir accès au trafic applicatif transitant dans le tunnel TLS.

Lors de leurs expérimentations, 2/3 des serveurs TLS testés (choisis aléatoirement sur Internet) étaient potentiellement faillibles à cette attaque (*i.e.* : en cas d'erreur de déchiffrement du `pre_master_secret`, ces serveurs fournissent une réponse faisant d'eux un oracle utilisable pour mener l'attaque). Toutefois, comme le notent Klima *et al.*, l'attaque est – en pratique – limitée par la capacité du serveur TLS à répondre pendant plus de 54 heures à des requêtes erronées (même si le client change son adresse IP régulièrement).

2.1.3 Contre-mesures

La contre-mesure préconisée contre l'attaque de Bleichenbacher consiste à ne pas renvoyer d'erreur explicite en cas de déchiffrement incorrect du `pre_master_secret`. Le serveur TLS doit générer une valeur pseudo-aléatoire de 48 octets et, en cas d'erreur lors du déchiffrement RSA, utiliser cette valeur comme `pre_master_secret` de remplacement (sans notifier d'erreur au client). Le serveur doit alors poursuivre la dérivation des clés de la même manière que si le déchiffrement avait été correctement effectué. La session TLS va s'interrompre d'elle-même lorsque le serveur va constater que le message `Finished` reçu du client à l'issue de la dérivation des clés est incorrect.

Une autre contre-mesure (préconisée par Bleichenbacher) consisterait à utiliser le chiffrement RSA OAEP (PKCS #1 v2.0). À l'heure actuelle (TLS 1.2), cette dernière contre-mesure n'a pas été implémentée pour des raisons de compatibilité avec les versions précédentes. Néanmoins, des recommandations d'implémentation liées au traitement du numéro de version lors du déchiffrement du `pre_master_secret` ont été ajoutées à la description du protocole TLS dans la version 1.2.

2.2 Timing attack par gestion des erreurs lors du déchiffrement d'un record

2.2.1 Description

Obtenir les clés de session TLS (en récupérant le `pre_master_secret` par exemple) permet de déchiffrer toutes les données transitant dans le tunnel correspondant. Toutefois, l'accès aux clés peut être difficile. Par ailleurs, les données échangées peuvent être de valeur inégale aux yeux d'un attaquant. L'intérêt de celui-ci peut ne porter que sur une faible (mais précise) quantité de données, par exemple : un mot de passe. Ainsi, pouvoir déchiffrer une partie d'une communication sans nécessairement être capable d'obtenir les clés de chiffrement correspondantes reste intéressant.

Dans **[17]**, Brice Canvel, Alain Hiltgen, Serge Vaudenay et Martin Vuagnoux présentent une technique permettant de récupérer des données transitant dans un tunnel TLS. La méthode est une extension améliorée d'une technique présentée par Serge Vaudenay.

Dans le protocole TLS, lorsque des données *m* sont chiffrées à l'aide d'un algorithme de chiffrement par bloc, le message effectivement chiffré consiste en un paquet de la forme

$$m \mid mac \mid pad \mid l$$

où :

- *mac* est le résultat du HMAC sur *m* notamment ;
- *pad* est un padding de *l* octets (chacun valant *l*) de sorte que $m \mid mac \mid pad \mid l$ ait une taille multiple de la taille de bloc *b* de l'algorithme de chiffrement.

Après le déchiffrement du paquet, le padding est vérifié (les derniers octets du paquet clair doivent correspondre à $l+1$ octets valant chacun *l*). Puis le *mac* est vérifié. Supposons que, en cas d'erreur sur le padding ou le MAC, une erreur distincte soit émise : respectivement `decryption_failed` et `bad_record_mac`. La nature de cette erreur peut alors être exploitée par l'attaquant pour récupérer *m*.

Par simplification, considérons un bloc $y = \mathrm{ENC}(x)$ obtenu par chiffrement de *x* (de taille *b* d'un bloc de chiffrement) avec un algorithme de chiffrement par bloc ENC. *y* est récupéré par l'attaquant sur le canal de communication et l'objectif est de trouver *x* (un mot de passe par exemple).

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

L'attaquant procède octet par octet en commençant par rechercher les octets les plus à droite. Tout d'abord, il construit $r = L \mid (u_1 \oplus 0x01) \mid (u_0 \oplus 0x01)$ où L est un bloc pseudo-aléatoire de taille b–2. L'attaquant envoie $r \mid y$ au serveur qui procède au déchiffrement en mode CBC en calculant

$$r \oplus ENC^{-1}(y) = (L \mid (u_1 \oplus 0x01) \mid (u_0 \oplus 0x01)) \oplus (x_{b-1} \dots x_0)$$

Les derniers octets de $r \oplus ENC^{-1}(y)$ sont censés correspondre au padding. Si la vérification du padding par le serveur renvoie « correct », cela signifie que (probablement)

$$u_1 \oplus 0x01 \oplus x_1 = 0x01$$

et

$$u_0 \oplus 0x01 \oplus x_0 = 0x01$$

Autrement dit : $x_1 = u_1$ et $x_0 = u_0$. Si la vérification du padding est incorrecte, l'attaquant choisit de nouvelles valeurs u_1, u_0 (cf. Figure 3).

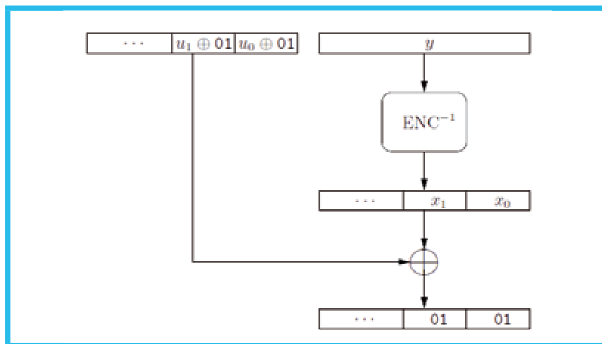


Figure 3 : Déchiffrement CBC de r | y. Ici, le choix u de l'attaquant est correct.

Une fois x_1 et x_0 trouvés, l'attaque se poursuit à l'aide d'un nouveau bloc $r' = L' \mid u$ avec $u = (c \oplus 0x02) \mid (x_1 \oplus 0x02) \mid (x_0 \oplus 0x02)$ où c est un octet. Le serveur procède au déchiffrement de $r' \mid y$. Il calcule $r' \oplus ENC^{-1}(y) = (L' \mid u) \oplus x$ dont les trois derniers octets valent $(c \oplus 0x02 \oplus x_2) \mid 0x02 \mid 0x02$. Si la vérification du padding est correcte, cela signifie alors que $c \oplus 0x02 \oplus x_2 = 0x02$ et donc que $x_2 = c$. Et ainsi de suite pour les autres octets de x.

Ce qui précède correspond à la technique générale de Vaudenay. Toutefois, dans TLS, les messages d'erreur (tels que `decryption_failed`, `bad_record_mac`) ne sont pas envoyés en clair au client mais chiffrés dans le tunnel. Par conséquent, l'attaquant n'a pas explicitement accès à ces messages. Par ailleurs, suite à la publication de l'article de Vaudenay, une erreur dans le padding est désormais traitée de la même manière qu'une erreur sur le MAC (au moins dans OpenSSL depuis la version 0.9.6c). De plus, ces deux erreurs entraînent la fin de la session TLS. Néanmoins, cette attaque peut toujours être mise en œuvre si l'attaquant parvient à distinguer les deux erreurs (padding et MAC). Pour ce faire, Canvel *et al.* utilisent un paquet de la forme $f \mid r \mid y$ où f est une chaîne d'octets pseudo-aléatoires telle que le paquet $f \mid r \mid y$ soit de taille maximale autorisée pour un *record* TLS. Avec un paquet de telle taille, on peut espérer distinguer les deux erreurs par la différence du temps de traitement du paquet. En effet, si le traitement du paquet est bien le suivant :

1. déchiffrement ;
2. vérification du padding ;
3. vérification du MAC ;

alors la session TLS prend fin plus rapidement en cas d'erreur sur le padding qu'en cas d'erreur dans le MAC (f est utilisé pour allonger artificiellement la durée de vérification du MAC).

Comme la session TLS prend fin à chaque choix incorrect de l'attaquant, celui-ci doit poursuivre l'attaque avec un nouveau bloc y' correspondant au chiffrement de x avec de nouvelles clés de session.

2.2.2 Mise en œuvre

Canvel *et al.* ont mis en œuvre leur attaque et ont ainsi pu récupérer le mot de passe utilisé dans l'authentification d'un client Outlook auprès d'un serveur de courrier IMAP. Notons toutefois que les tests réalisés l'ont été dans un réseau local, les perturbations sur le réseau étant donc notablement réduites. L'attaque suppose aussi que l'on puisse se placer en coupure entre le client et le serveur TLS pour injecter les paquets $f \mid r \mid y$ dans le tunnel.

L'attaque tire également parti de l'utilisation d'outils statistiques pour filtrer les réponses reçues du serveur (le temps de réponse en cas d'erreur) et définir si le choix sur les octets de x est correct ou pas. Par ailleurs, la cible étant un mot de passe, usage a été fait d'un dictionnaire de mots de passe, ce qui a également contribué à réduire la complexité générale de l'attaque. À titre d'exemple, un mot de passe de 8 caractères est retrouvé au bout de 166 appels à l'oracle de déchiffrement avec une probabilité de succès de 1/2.

2.2.3 Contre-mesures

Les contre-mesures préconisées par les auteurs sont :

- Égaliser le temps de vérification du padding et du MAC en procédant à la vérification du MAC dans tous les cas.
- Vérifier le MAC avant le padding afin de rendre la vérification du padding négligeable par rapport à celle du MAC. Le paquet à chiffrer est alors de la forme $m \mid pad \mid l \mid mac$ au lieu de $m \mid mac \mid pad \mid l$.

Un patch correspondant à la première proposition a été fourni pour la version OpenSSL 0.9.6e et antérieures. Par ailleurs, des recommandations d'implémentation liées au traitement du MAC ont été ajoutées dans la spécification de TLS à partir de la version 1.1.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

3 Attaque par renégociation

3.1 Description

Le protocole TLS autorise à procéder à des renégociations alors qu'une session est en cours. La renégociation peut être motivée par le souhait de changer d'algorithme de chiffrement, de renouveler les clés de session, de procéder à une authentification d'un niveau plus élevé, par exemple. La renégociation se fait à l'initiative du client (par un message `ClientHello`) ou du serveur (par un message `HelloRequest`). Le nouvel échange de clé s'effectue au travers du tunnel TLS existant. À l'issue de la renégociation, les données applicatives sont protégées avec le nouveau contexte de sécurité.

Du point de vue applicatif, il n'y a pas de discontinuité entraînée par cette renégociation : les données applicatives transmises dans le précédent tunnel et dans le nouveau tunnel sont vues comme un tout. Considérons une session HTTPS **[29]**. Les données

```
GET /pizza?toppings=sausage;address=myaddress HTTP/1.1
```

transmises dans le précédent tunnel et les données

```
Cookie: mycookie
```

transmises dans le nouveau tunnel sont interprétées au niveau applicatif comme

```
GET /pizza?toppings=sausage;address=myaddress HTTP/1.1
Cookie: mycookie
```

Dans **[1]**, Marsh Ray et Steve Dispensa expliquent comment cela peut être mis à profit par un attaquant pour tromper un serveur TLS et abuser d'un client légitime. L'idée générale consiste pour l'attaquant à établir avec le serveur une session TLS à la place du client puis à faire passer une session TLS du client légitime pour une renégociation auprès du serveur.

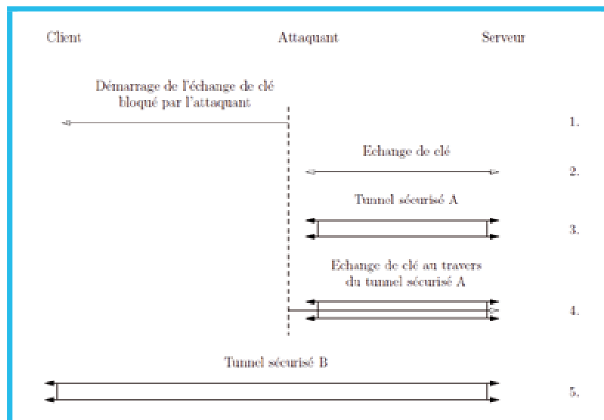


Figure 4 : Attaque par renégociation

Cela permet à l'attaquant de « préfixer » les données envoyées par le client légitime au serveur : le serveur va considérer que les données transmises dans le premier tunnel TLS (établi par l'attaquant) et celles transmises dans le deuxième tunnel (établi par le client légitime) proviennent toutes du client.

La figure 4 illustre une mise en œuvre de l'attaque par renégociation.

Le déroulement de l'attaque est le suivant :

1. Un client débute une session HTTPS pour commander une pizza. L'attaquant, placé en coupure entre le client et le serveur, bloque le paquet `ClientHello` du client. L'attaquant établit alors une session TLS avec le serveur (en utilisant le même canal de communication que le client).
2. L'attaquant procède à un échange de clé avec le serveur.
3. Un tunnel sécurisé A est établi entre l'attaquant et le serveur. Au travers de ce tunnel, l'attaquant peut transmettre tout type de données au serveur. L'attaquant transmet au serveur la requête HTTP suivante :

```
GET /pizza?toppings=pepperoni;address=attackersaddress HTTP/1.1 X-Ignore:
```

   sans retour chariot après `X-Ignore:`.

4. Puis l'attaquant transmet au serveur le `ClientHello` du client légitime bloqué à l'étape 1. Le serveur interprète ce paquet comme une demande de renégociation. Un échange de clé est donc effectué entre le client et le serveur au travers du tunnel établi par l'attaquant. L'attaquant, en coupure entre le client et le serveur, relaie les différents messages TLS entre le client et le serveur.
5. Un tunnel sécurisé B est maintenant établi entre le client et le serveur. L'attaquant n'a pas accès à ce tunnel. Le client peut transmettre au serveur tout type de données de manière sécurisée. Et le client procède à sa commande de pizza par la requête HTTP suivante :

```
GET /pizza?toppings=sausage;address=victimssaddress HTTP/1.1
Cookie: victimscookie
```

6. Le serveur reçoit successivement la requête de l'attaquant et celle du client légitime et les interprète comme :

```
GET /pizza?toppings=pepperoni;address=attackersaddress HTTP/1.1
X-Ignore: GET /pizza?toppings=sausage;address=victimssaddress HTTP/1.1
Cookie: victimscookie
```

Ainsi, l'attaquant commande la pizza... et le client légitime règle la facture.

Le serveur associe les données applicatives du premier tunnel (établi par l'attaquant) avec celles du deuxième tunnel (établi par le client légitime) car le deuxième tunnel est établi sous l'égide du premier. Pourtant, d'un point de vue cryptographique, il n'y a aucune corrélation entre

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

la manière dont ces deux ensembles de données sont protégés (par exemple, les données des deux ensembles pourraient être signées par une même clé symétrique). Il y a donc une discontinuité dans l'authentification entre l'établissement du premier tunnel et l'établissement du deuxième. Et c'est de cette discontinuité que l'attaque tire profit. Par ailleurs, si le client légitime s'authentifie auprès du serveur, cela ne met pas en défaut l'attaque. Paradoxalement, le serveur pourrait considérer que cela correspond même à une authentification *a posteriori* de l'attaquant.

3.2 Mise en œuvre

Anil Kurmus a présenté un scénario possible de cette attaque avec le site Twitter. Tout utilisateur peut modifier ses paramètres de connexion à son compte (mot de passe). L'idée est de « préfixer » le message de l'utilisateur légitime :

```
GET /HTTP/1.1
Cookie: [...]
```

par une requête **POST** :

```
POST /forum/send.php HTTP/1.0
message=GET /HTTP/1.1 [...]
```

Le résultat ? Au lieu d'interpréter la requête comme un changement de mot de passe, le serveur web considère que les données correspondent à un message Twitter qu'il affiche donc sur le site de l'utilisateur offrant ainsi à tous le mot de passe (cf. Figure 5).

Note
Twitter a rapidement corrigé son serveur, rendant maintenant ce scénario inapplicable.

Crédit : Anil Kurmus

```
POST /statuses/update.xml
HTTP/1.1 Authorization: Basic
dmljdGltQGV4YW1wbGUuY29tOnN
User-Agent: curl/7.18.2 (i486-
pc-linux-gnu) l
1 minute ago from API
```

Figure 5 : Mot de passe révélé sur Twitter

3.3 Contre-mesures

L'impact de l'attaque par renégociation dépend de l'application protégée par TLS. Comme illustré plus haut, s'il s'agit de HTTP, l'attaquant peut détourner ou dérober l'accréditation du client légitime (présente dans la figure 4 sous la forme du *cookie*) à son profit et bénéficier d'un service qui sera crédité (et potentiellement facturé) au client légitime.

L'attaque par renégociation met en défaut les mécanismes d'intégrité de TLS puisque l'attaquant peut transmettre des données qui seront considérées par le serveur comme provenant du client légitime. Notons toutefois que, si l'on considère le scénario d'une transaction financière comme plus haut, l'attaque peut être découverte par le client si le serveur lui retourne un résumé de cette transaction (avant ou après validation de cette dernière). Dans ce cas, l'attaquant peut, au mieux, bloquer le message du serveur mais ne peut pas le modifier car il n'a pas accès au tunnel TLS établi entre le client et le serveur.

Si l'intégrité des données échangées entre le client et le serveur peut être mise en défaut, il n'en est pas de même de la confidentialité des données. À ce titre, l'attaque par renégociation est un contournement plus qu'une remise en cause directe de la robustesse cryptographique du protocole TLS.

Une manière (radicale) de contrer cette attaque est d'interdire les renégociations.

Note
Cette correction a été apportée à la version 0.9.8l d'OpenSSL.

Comme TLS supporte les extensions, d'autres corrections sont toutefois possibles. La correction présentée dans **[2]** consiste à ajouter une extension appelée `RenegociationInfo` aux paquets `ClientHello` et `ServerHello`. Celle-ci indique aux correspondants s'il s'agit d'une première négociation et permet de détecter l'attaque.

Par défaut, un serveur TLS doit accepter un `ClientHello` contenant des extensions même s'il n'en comprend pas le sens. Néanmoins, anticipant des problèmes dus à la variété des implémentations, les auteurs de cette correction offrent une alternative à l'extension `RenegociationInfo`. Elle consiste à inscrire dans la liste des suites cryptographiques du `ClientHello` une valeur particulière (`TLS_EMPTY_RENEGOTIATION_INFO_SCSV`) signifiant qu'il s'agit d'un échange de clé initial.

La RFC 5746, décrivant cette correction, a été implémentée dans les versions 0.9.8m et 1.0.0a d'OpenSSL.

4 Mode d'opération : le cas de CBC

4.1 Description

Les algorithmes de chiffrement par bloc peuvent être utilisés selon différents modes. Le mode ECB consiste à chiffrer chaque bloc de données claires indépendamment. C'est le plus simple mais il comporte des inconvénients. Le mode CBC est donc employé

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

dans le protocole TLS. Dans ce mode, le chaînage des blocs permet de masquer d'éventuelles récurrences dans les données claires. Néanmoins, ce mode peut être détourné par un attaquant pour récupérer des données transitant dans le tunnel TLS.

S'appuyant sur des travaux de Gregory Bard, Thai Duong et Juliano Rizzo ont mis en scène un tel scénario en 2011 **[7]** : l'attaque « BEAST ». Le scénario est le suivant : un utilisateur légitime s'authentifie par mot de passe auprès d'un site internet. La transmission du mot de passe se fait par un tunnel TLS. Supposons que l'attaquant ait les capacités suivantes :

- Il peut observer le trafic chiffré échangé entre le client et le serveur TLS.
- Il peut injecter des données claires dans le tunnel TLS (sans toutefois pouvoir récupérer les données claires de l'utilisateur légitime).

Pour découvrir le mot de passe *password*, transmis dans le tunnel TLS par l'utilisateur légitime, l'attaquant procède comme suit :

1. Il récupère le bloc chiffré C_i contenant *password* (par simplification, on supposera que *password* est de la taille d'un bloc de chiffrement, par exemple : 16 octets pour l'AES).
2. Il fait une hypothèse sur la valeur du mot de passe *password'* et injecte un bloc contenant *password'* dans le tunnel TLS.
3. Il récupère le bloc chiffré C_j correspondant à *password'*. Si $C_j = C_i$, alors l'attaquant en déduit que *password* = *password'*.

Autrement dit, l'attaquant se sert du tunnel TLS comme « distingueur » pour mener à bien sa recherche exhaustive sur la valeur du mot de passe.

Néanmoins, deux problèmes se posent :

- Si le client utilise un algorithme de chiffrement par bloc (comme l'AES), le mode CBC est utilisé. Dans ce cas, C_i ne correspond pas au chiffrement de *password* mais à AES($C_{i-1} \oplus$ *password*) où C_{i-1} est le précédent bloc chiffré (cf. Figure 6).
- Si *password* a une taille de 16 octets, la recherche exhaustive correspond à un coût de 2^{128} « chiffrements TLS » (ce « chiffrement TLS » inclut notamment la construction de la trame claire, l'encodage des données, le calcul HMAC, le chiffrement AES). Et une recherche exhaustive sur 128 bits semble un tantinet délicate...

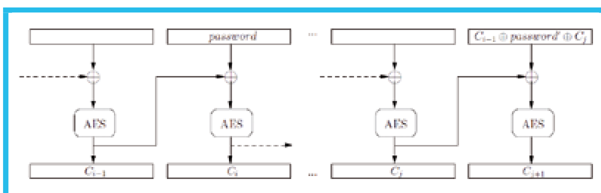


Figure 6 : Chiffrement CBC et recherche de password

Considérons la première difficulté. Elle peut être contournée dans TLS 1.0 car l'IV est prédictible. L'attaquant récupère le bloc chiffré C_i = AES($C_{i-1} \oplus$ *password*). Il veut tester son hypothèse *password'*. Pour cela, il injecte dans le tunnel un bloc $B = C_{i-1} \oplus$ *password'* $\oplus C_j$ où C_j est le bloc chiffré qui aura précédé B. Ainsi, le chiffrement de B aboutit à :

$$\begin{aligned} C_{j+1} &= \text{AES}(B \oplus C_j) \\ &= \text{AES}(C_{i-1} \oplus password' \oplus C_j \oplus C_j) \\ &= \text{AES}(C_{i-1} \oplus password') \end{aligned}$$

Il suffit alors de comparer C_{j+1} à C_i. S'il y a égalité, cela signifie que *password* = *password'*.

Considérons à présent la deuxième difficulté. L'astuce utilisée par Duong et Rizzo consiste à rechercher le mot de passe octet par octet. Soit *password* = $p_0 \mid p_1 \mid ... \mid p_{15}$. La valeur de référence utilisée en pratique par l'attaquant va être C_i = AES($C_{i-1} \oplus$ (*cste* | p_0)) où *cste* est une valeur fixe de 15 octets (par exemple l'en-tête d'une requête HTTP). L'attaquant fait l'hypothèse que le premier octet du mot de passe est p_0' et construit le bloc $B = C_{i-1} \oplus$ (*cste* | p_0') $\oplus C_j$ qu'il injecte dans le tunnel TLS. Il récupère le bloc chiffré correspondant C_{j+1} = AES($C_{i-1} \oplus$ (*cste* | p_0')). Si $C_{j+1} = C_i$, cela signifie que $p_0 = p_0'$. Sinon, un autre bloc B est construit avec la même valeur *cste* et un autre choix pour p_0'. Il y a au plus 256 essais à faire pour trouver l'octet p_0.

Une fois p_0 trouvé, la recherche porte sur p_1 avec le bloc B = *cste'* | p_0 | p_1' où p_1' est l'hypothèse de l'attaquant sur p_1 et *cste'* est une constante de 14 octets (en pratique, *cste'* peut être obtenu à partir de *cste* en effectuant un décalage d'un octet à droite au fur et à mesure que les octets p_k sont trouvés). L'attaquant effectue au plus 256 essais pour trouver p_1 en utilisant comme nouvelle valeur de référence C_i = AES($C_{i-1} \oplus$ (*cste'* | p_0 | p_1)).

En procédant de cette manière, l'attaquant doit effectuer au maximum 256 essais pour chacun des 16 octets de *password*, soient 2^{12} essais au plus, au lieu de 2^{128} essais. L'attaque implique également d'établir un nombre de connexions TLS égal au nombre d'octets de *password* (16 dans notre exemple) nécessaires pour récupérer les différentes valeurs de référence C_i.

Comme l'attaque porte sur les données claires, son coût dépend de la quantité de données que l'on cherche à récupérer. Autrement dit : l'attaque peut être relativement efficace s'il s'agit de retrouver un mot de passe chiffré par TLS mais nettement moins efficace s'il s'agit de déchiffrer des Mo de données.

4.2 Mise en œuvre

Considérons le cas d'une victime disposant d'un compte utilisateur auprès d'un site A (par exemple Paypal). Pour accéder à ce compte, la victime prouve son accréditation en transmettant un mot de passe au travers d'un tunnel TLS 1.0 établi avec A. Les conditions nécessaires à la réalisation de l'attaque sont les suivantes :

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

1. L'attaquant peut observer le trafic chiffré échangé entre le client et le serveur TLS.
2. L'attaquant peut forcer la victime à effectuer des connexions TLS vers un site (légitime) A (en HTTPS par exemple).
3. L'attaquant peut injecter des données claires dans le tunnel TLS établi avec le site A (sans toutefois pouvoir récupérer les données claires de l'utilisateur légitime).

La condition 3 permet à l'attaquant de tester ses hypothèses sur les différents octets du mot de passe, par injection du bloc *B* dans le tunnel TLS.

La condition 2 force la victime à établir (de manière répétée) un tunnel TLS au travers duquel elle transmet son mot de passe au site A. Cela permet à l'attaquant de récupérer les différentes valeurs de références $C_i = \mathrm{AES}(C_{i-1} \oplus (cste \mid p_0 \mid ... \mid p_k))$ et les blocs chiffrés C_{j+1} avec lesquels il vérifie son hypothèse sur les octets p_k.

Selon Duong et Rizzo, les deux dernières conditions peuvent être réunies si la victime charge avec son navigateur une page contenant des liens pointant sur le site A d'une part. D'autre part, l'injection de données dans le tunnel TLS établi entre la victime et le site A peut se faire à l'aide d'un procédé de type *cross-site request forgery* qui permet à l'attaquant de berner le navigateur quant à l'origine des données reçues.

Duong et Rizzo ont présenté une illustration de leur attaque en récupérant le cookie de connexion d'un compte Paypal.

4.3 Contre-mesures

Cette attaque concerne TLS 1.0. Concernant les aspects cryptographiques, il « suffit » d'utiliser TLS 1.1 ou 1.2 puisque dans ces versions l'IV n'est pas prédictible (il est généré pseudo-aléatoirement pour chaque paquet TLS). Par conséquent, l'attaque ne fonctionne pas avec ces versions.

En effet, si l'attaquant ne connaît pas l'IV utilisé lors du chiffrement, il ne peut pas distinguer si son choix est correct ou pas lors de la recherche exhaustive des données claires. Attention : la machine ne doit pas seulement implémenter TLS 1.1 mais refuser de basculer en TLS 1.0 si le correspondant supporte uniquement TLS 1.0. Toutefois, l'utilisation de TLS 1.1 ou 1.2 à l'exclusion de la version 1.0 peut poser des problèmes de compatibilité étant donné que ces versions semblent encore peu supportées (dans les navigateurs notamment) à l'heure actuelle.

Une autre possibilité est de ne pas utiliser le mode CBC pour le chiffrement mais un algorithme de chiffrement par flot tel que RC4. Encore une fois, il s'agit ici d'interdire tout autre algorithme de chiffrement que RC4 (le navigateur ne doit pas opter pour un autre algorithme si le correspondant ne propose pas RC4 dans la suite cryptographique).

> **Note**
> **TLS propose soit des algorithmes de chiffrement par bloc en mode CBC, soit RC4.**

Enfin, l'attaque ne peut fonctionner si les données sont compressées avant chiffrement.

Plusieurs années avant l'attaque BEAST, une correction a été faite dans la bibliothèque OpenSSL (à partir de la version 0.9.6d, 09/05/2002) qui contrecarre de facto l'attaque de Duong et Rizzo. Cette contre-mesure consiste à placer un record vide comme premier élément à chiffrer dans le paquet TLS. À partir de ce record, un HMAC est calculé (incluant un *sequence number* implicite) et le tout est chiffré (après ajout d'un padding fixe). Comme les clés de session (chiffrement et signature) ne sont pas connues de l'attaquant, celui-ci ne peut pas connaître en avance la valeur du champ HMAC ni celle du bloc chiffré obtenu. Ceci permet donc d'obtenir un IV non prédictible qui sera utilisé pour le chiffrement du bloc suivant.

Toutefois, pour des raisons de compatibilité, cette contre-mesure peut être désactivée à partir d'OpenSSL 0.9.6e (30/07/2002). Et elle l'est par défaut...

5 Erreurs d'implémentation

Vous souvenez-vous du « bug » Debian ? Croyant corriger une fuite de mémoire provenant du code OpenSSL, des développeurs du projet Debian ont en réalité drastiquement réduit la source d'entropie utilisée pour générer des valeurs pseudo-aléatoires. La conséquence de ce bug : le nombre de clés RSA et (EC) DSA que le code OpenSSL « corrigé » pouvait générer s'est trouvé limité à un nombre si faible qu'il était possible de générer toutes ces clés. Les clés pouvant être énumérées, toutes les sessions TLS établies à partir de telles clés étaient donc compromises. Pire : les certificats correspondants sont restés potentiellement utilisables pour monter des attaques *man in the middle*, étant données les vérifications parfois partielles qui sont faites (notamment : absence de vérification des listes de révocation).

Néanmoins, d'autres types d'erreurs d'implémentation, plus subtiles, peuvent ruiner la sécurité d'un protocole.

5.1 Description

Imaginons un algorithme mal implémenté qui fournit un résultat correct en général, sauf dans certains cas particuliers lorsque certains paramètres sont utilisés. Cette erreur d'implémentation peut-elle être exploitée ? Oui si l'algorithme en question est sollicité à un moment

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

crucial de la mise en œuvre d'un protocole de sécurité. C'est ce qu'ont montré Brumley, Barbosa, Page et Vercauteren **[18]** en s'appuyant sur des travaux de Biham, Carmeli et Shamir **[19]**.

Biham *et al.* présentent un concept d'attaque tirant profit d'une erreur d'implémentation dans un logiciel ou un composant matériel. Le principe exposé est le suivant : l'opération implémentée retourne un résultat correct dans la grande majorité des cas (*i.e.* : pour la plupart des opérandes engagées dans le calcul) mais, pour certaines valeurs des opérandes, le résultat du calcul est faux. Illustrons ce concept avec l'exemple suivant correspondant à un produit scalaire $[k]P$ selon la formule de Montgomery (algorithme 7).

```
Entrées : point P ∈ E, scalaire k = (k_{l-1}...k_1k_0)_2.
Sortie : point Q = [k]P ∈ E.
RØ = O
R1 = P
for i = l-1 downto Ø step -1
  if k_i = Ø then
    R_1 = R_0 + R_1
    R_0 = [2]R_0
  else
    R_0 = R_0 + R_1
    R_1 = [2]R_1
  end
end
return R_0
```

Figure 7 : Produit scalaire selon la formule de Montgomery (version ladder)

Considérons une valeur P particulière telle que :

- le calcul $[2]R_0$ est incorrect ;
- le calcul $R_0 + R_1$ est correct ;
- le calcul $[2]R_1$ est correct ;
- si le bit k_i vaut 0.

Si un attaquant souhaite recouvrer k_i, il lui suffit de soumettre une telle valeur P à un oracle de calcul $[k]P$. Si le résultat final qui lui est retourné est correct, cela signifie que l'opération $[2]R_0$ n'a pas été effectuée et l'attaquant en déduit que $k_i = 1$. Si le résultat final est incorrect, cela signifie que l'algorithme est passé par l'étape $[2]R_0$ et donc que $k_i = 0$.

Il suffit donc à l'attaquant de construire $\lfloor \log_2(k) \rfloor + 1$ valeurs P_i telles qu'une erreur se produise à l'étape i (et uniquement à cette étape) du calcul $[k]P_i$. Le résultat des différents calculs $[k]P_i$ révèle la valeur des bits k_i.

Brumley *et al.* proposent une mise en œuvre concrète du concept ci-dessus. La version 0.9.8g d'OpenSSL présente en effet une erreur d'implémentation dans la multiplication modulaire $x \times y \bmod p$ où $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$ est l'un des paramètres de la courbe elliptique P-256 du NIST **[20]**. La formule utilisée pour effectuer ce calcul tire parti de la forme particulière de p.

L'attaque a pour but de trouver la valeur k utilisée dans le calcul $[k]P$. Le contexte dans lequel ce calcul est effectué est celui d'un échange de clé Diffie-Hellman statique (cf. Figure 8).

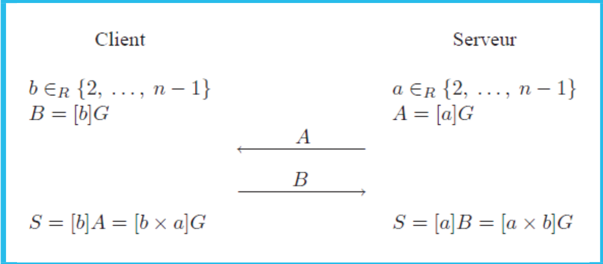


Figure 8 : Échange de clé Diffie-Hellman sur courbe elliptique (ECDH)

Dans le cas d'un échange ECDH statique, le serveur utilise une valeur a constante d'un échange à l'autre. Le fait que a (et donc $A = [a]G$) soit constant permet à l'attaquant d'effectuer de nombreux échanges de clé avec le serveur afin de recouvrer progressivement les digits de a.

La valeur $S = [a]B$ correspond au secret négocié entre le client (l'attaquant) et le serveur. Ce secret commun fournit le `pre_master_secret`, élément principal de la dérivation des clés de session.

Si le calcul $[a]B$ effectué par le serveur est correct, les clés de session seront correctement dérivées et le paquet `Finished` sera correctement construit. Inversement, si le calcul $[a]B$ est incorrectement effectué, l'échange de clé échouera. Par conséquent, l'échange de clé est utilisé par l'attaquant comme distingueur pour savoir si la formule erronée a été mise en œuvre dans le calcul $[a]B$. L'attaquant utilise ce bit d'information pour savoir si son hypothèse sur la valeur de a_i est correcte ou pas.

L'implémentation d'OpenSSL utilise une représentation NAF. Si on considère des digits de taille 4 bits, cela signifie que les composantes a_i de a sont à valeur dans $\{-7, -5, -3, -1, 0, 1, 3, 5, 7\}$. De plus, par définition de la représentation NAF, un digit non nul est nécessairement suivi d'au moins 3 digits nuls. Ainsi, la découverte explicite d'un digit a_i non nul implique la découverte implicite des 3 digits nuls suivants.

Cette attaque peut également être mise en œuvre dans le cas d'un échange Diffie-Hellman éphémère (ECDHE). Lors d'un échange ECDHE, les valeurs secrètes a et b des deux correspondants doivent être renouvelées lors de chaque échange de clé. Toutefois, OpenSSL implémente une « optimisation » assez singulière. Lors d'un échange ECDHE, la valeur secrète a du serveur est effectivement différente d'un client à l'autre, mais cette valeur reste constante si des échanges de clé sont effectués avec le même client !

Note

Cette optimisation est appliquée par défaut...

Cela viole le principe de *forward secrecy* et permet la réalisation de l'attaque.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

5.2 Mise en œuvre

Dans le cas d'un échange ECDHE avec optimisation, la connaissance de a permet à l'attaquant de déchiffrer toutes les communications passées et futures établies entre le serveur et un client donné. En effet, à partir d'une valeur $B = [b]G$ fraîchement générée par le client, l'attaquant est capable de calculer le secret partagé $S = [a]B = [a \times b]G$.

Dans le cas d'un échange ECDH, la valeur a est persistante. Elle a été et sera utilisée par le serveur avec tous les clients. L'attaquant en possession de cette valeur peut donc accéder au clair des communications passées et futures établies entre le serveur et tous ces clients.

L'attaque est possible lorsque l'échange ECDH (ou ECDHE avec optimisation) est effectué avec les courbes elliptiques P-256 et P-384 (mais pas P-192, P-224 ni P-521 pour lesquelles des formules de calcul différentes sont utilisées). L'attaque permet de récupérer le secret a du serveur en 633 échanges de clé ECDH dans le cas P-256.

5.3 Contre-mesures

La formule de calcul erronée a été corrigée à partir de la version 0.9.8h d'OpenSSL. Plus généralement, une manière de prévenir une attaque reposant sur une erreur d'implémentation consiste à procéder à une vérification formelle du code source. Des contre-mesures algébriques peuvent également être déployées pour contrer l'attaque, telles que :

- le *scalar blinding* : $[k]P$ est obtenu par la formule $[k + r \times n]P$ où r est une (petite) valeur choisie pseudo-aléatoirement et $[n]P = O$;
- le *point blinding* : $[k]P$ est obtenu par la formule $[k](P + R) - T$ où R est choisi pseudo-aléatoirement et $T = [k]R$.

Comme une valeur pseudo-aléatoire intervient dans le calcul de $[k]P$, l'attaquant ne peut plus établir de corrélation entre le résultat du calcul et la valeur de k.

Conclusion

Un certain nombre de failles et d'attaques de TLS et OpenSSL a été présenté. La liste n'est sans doute pas exhaustive ni figée. Il est fort possible qu'elle s'étoffera à l'avenir.

Parmi les attaques décrites, certaines (notamment cryptographiques) étaient entièrement nouvelles alors et se sont révélées efficaces contre un protocole qui n'a pas été pensé (conceptuellement) pour y résister, entraînant une modification du protocole lui-même. D'autres attaques ont tiré profit des choix d'implémentation du protocole (par exemple les *timing attacks*). Parfois même, de discrètes erreurs d'implémentation nichées dans un coin obscur de la bibliothèque ont offert à l'attaquant le moyen suffisant pour ruiner la sécurité du protocole. Dans ces deux derniers cas, c'est l'implémentation qui est en cause.

Mais les attaques ne sont pas toutes nouvelles. Ainsi, l'attaque BEAST (2011) s'appuie sur les travaux de Bard (2004, 2006) considérés comme essentiellement théoriques jusque-là. La mise en œuvre de BEAST a été possible grâce aux développements d'un autre champ de la sécurité (réseau) dont les aboutissements ont servi de vecteur pour appliquer l'attaque de Bard.

D'autres attaques, sans être nouvelles dans leur concept, correspondent à la mise en œuvre (avec plus ou moins de succès) dans le monde réseau de techniques largement étudiées et qui ont fait leurs preuves dans d'autres domaines (par exemple : les attaques par canaux auxiliaires, telles que les timing attacks, sur des équipements électroniques comme la carte à puce).

Enfin, il ne faut pas négliger les failles les plus faciles à corriger mais toujours profitables à un attaquant : mauvaise configuration d'un serveur, support d'algorithmes « faibles », etc.

Si la publication d'une attaque ou la détection d'un bug constitue d'utiles piqûres de rappel, cela ne suffit pas forcément à renforcer la sécurité d'un réseau car il y a une latence plus ou moins importante (et un intérêt plus ou moins poussé) avant la prise en compte effective et le déploiement d'un correctif **[21, 22]**.

Pour les administrateurs de serveurs TLS ou autres intéressés à la sécurité du protocole (et de leur réseau), plusieurs moyens sont offerts. Tout d'abord, des études sont régulièrement publiées concernant l'usage du protocole **[23, 24, 25]**, des recommandations d'emploi **[28, 30]**, ses performances **[26]** ou à l'occasion de la découverte de faille ou d'attaque **[27]**. Ces études permettent notamment d'observer l'évolution du déploiement et des usages du protocole (rapidité de prise en compte des correctifs, algorithmes les plus utilisés, etc.).

Enfin, plusieurs sites s'attachent à maintenir une veille sur la sécurité de TLS. Par exemple (mais ce ne sont sans doute pas les seuls) :

- SSL *server survey* de Netcraft : **https://ssl.netcraft.com/ssl-sample-report/index** ;
- SSL *observatory* de l'Electronic Frontier Foundation (EFF) : **https://www.eff.org/observatory** ;
- SSL *Pulse* de SSL Labs : **https://www.trustworthyinternet.org/ssl-pulse/**. ■

Les références de cet article sont disponibles sur : http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6/3

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

HACHAGE SÉCURISÉ : LA COMPÉTITION SHA-3

Ryad Benadjila - ryad.benadjila@ssi.gouv.fr, ANSSI
Olivier Billet - billet@eurecom.fr
Gilles Macario-Rat - gilles.macariorat@orange.com, Orange Labs
Thomas Peyrin - thomas.peyrin@ntu.edu.sg, NTU et SECOSA Pte Ltd

mots-clés : FONCTIONS DE HACHAGE / STANDARD / COMPÉTITION SHA-3

Les fonctions de hachage sont un élément crucial de la quasi-totalité des architectures de sécurité. Après une récente vague d'attaques novatrices importantes, leur construction a dû être revue et un nouveau standard SHA-3 est en cours d'élaboration.

1 Des fonctions de hachage

Une fonction de hachage permet d'associer à chaque élément d'un ensemble de données de taille arbitraire une empreinte ou une étiquette choisie dans un petit ensemble (les chaînes de *n* bits, par exemple). Une telle fonction perd donc nécessairement de l'information : il y a beaucoup moins d'empreintes possibles que de données distinctes et la même empreinte ou étiquette pourra être associée à plusieurs éléments distincts. Cette propriété, parfois associée à d'autres contraintes, est utilisée de multiples façons en informatique et plus particulièrement afin d'apporter des garanties de sécurité.

Dans le cadre de la cryptographie qui nous intéresse ici, les fonctions de hachage constituent un composant essentiel de la quasi-totalité des mécanismes de sécurité, ce qui lui vaut le surnom de « couteau suisse de la cryptographie ». Les fonctions de hachage interviennent en effet aussi bien dans la gestion des mots de passe, que dans le cadre de signatures numériques, dans le chiffrement hybride ainsi que dans la plupart des protocoles de sécurité, notamment en tant que fonction de dérivation de clés secrètes. Nous passons en revue dans la section suivante quelques cas d'usage standards (mais il en existe bien d'autres) qui nous permettent d'introduire, au passage, des propriétés particulières attendues de la fonction de hachage cryptographique. En pratique, la fonction de hachage cryptographique a en général pour objectif de garantir l'ensemble de ces propriétés de façon simultanée.

1.1 À quoi servent les fonctions de hachage ?

1.1.1 Tables de hachage

Un usage (non cryptographique) est la construction d'une structure de données appelée table de hachage. Rechercher un élément parmi un ensemble assez grand nécessite naïvement un temps linéaire et les tables de hachage permettent une telle recherche instantanément (temps constant) pour des éléments d'un format quelconque en associant un haché à chaque élément de la table et en l'enregistrant dans une liste correspondant au haché. Pour rechercher un élément, il suffit de le hacher et de tester sa présence dans la liste correspondante.

Cette technique nécessite une exécution rapide (quelques cycles processeur) ainsi que des listes de faible taille impliquant une fonction de hachage équilibrée. Les fonctions de hachage obtenues dans ce cadre sont beaucoup trop simples pour des usages cryptographiques, mais très utiles en cryptanalyse en raison du grand nombre d'éléments à manipuler.

1.1.2 Mots de passe

Une fonction de hachage cryptographique H doit résister au calcul des pré-images, c'est-à-dire qu'elle doit être à sens unique : il est facile (à traduire par efficace calculatoirement) de calculer $z=H(x)$ à partir d'une donnée quelconque x, mais difficile de déterminer, à partir de z seulement, un y quelconque tel que $H(y)=z$.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Cette propriété est par exemple à la base du mécanisme d'authentification utilisant des mots de passe dans les systèmes d'exploitation : l'empreinte d'un mot de passe salé *H*(password||salt) et le sel *salt* sont enregistrés au préalable (l'opérateur || représente ici la concaténation de chaînes de bits). Le système n'authentifie l'utilisateur que s'il fournit x tel que *H*(x||salt)=*H*(password||salt). Un attaquant, même ayant accès à la valeur enregistrée dans le système, n'est donc pas en mesure de s'authentifier. Le sel de l'exemple évite qu'un attaquant hache tous les mots de passe ayant une structure particulière, comme l'ensemble des mots du dictionnaire dans toutes les langues : bien que coûteuse, cette attaque serait valable sur tous les systèmes faisant usage de la fonction de hachage. Le sel, variable, force à effectuer cette recherche pour chaque mot de passe à retrouver, infligeant un coût parfois prohibitif.

1.1.3 Signatures

La signature numérique (via le célèbre RSA ou la version sur courbe elliptique du DSA) est à la base du commerce électronique, mais est très lente. Les fonctions de hachage permettent de signer le haché *H*(M) de faible taille en lieu et place de M. Évidemment, une telle stratégie nécessite qu'un attaquant ne puisse trouver, étant donné M et sa signature *S*(*H*(M)), une autre donnée W telle que *H*(W)=H(M) car alors *S*(*H*(M))=*S*(*H*(W)) constituerait une signature valide de W, alors que le signataire légitime ne l'a jamais signée ! Cette propriété est la résistance au calcul de seconde pré-images.

La signature ci-dessus requiert une autre propriété en pratique, car la signature doit être non répudiable. Or le signataire, s'il est capable de calculer des collisions pour la fonction *H* (*i.e.* de calculer deux données M et N distinctes telles que *H*(M)=*H*(N)), peut alors arguer que le schéma de signature n'est pas sûr puisqu'en donnant sa signature *S*(*H*(M)) pour la donnée M, quelqu'un pourrait s'en servir comme signature de N, donnée qu'il n'aurait jamais signé. Cette propriété de résistance aux collisions semble peut-être un peu artificielle, mais des chercheurs ont montré dans le cas de MD5 qu'il est possible d'utiliser les collisions à des fins malhonnêtes **[SS08]**.

Il est d'ailleurs inquiétant de voir que le récent malware Flame **[FLM]** semble utiliser une variante des techniques de génération de collisions encore inconnue pour MD5 afin de forger de faux certificats. Ce logiciel espion, très sophistiqué, est ainsi capable de se faire passer pour une mise à jour Windows légitime a priori signée par Microsoft afin d'obtenir des privilèges élevés au sein du système d'exploitation. Flame illustre donc bien l'utilisation pratique d'attaques académiques à des fins malveillantes. Il a aussi révélé, au passage, que Microsoft utilisait encore des certificats à base de MD5 malgré ses faiblesses avérées.

1.1.4 Chiffrements à flot

Si leur exécution est rapide, les fonctions de hachage peuvent générer une suite chiffrante : l'idée est de hacher un compteur et un secret. La suite chiffrante est alors classiquement utilisée comme masque jetable et il est bien connu que celui-ci doit idéalement être parfaitement aléatoire. La propriété requise pour la fonction de hachage est de se comporter « comme une fonction aléatoire, » *i.e.* la garantie de l'absence de toute propriété particulière sur le comportement de la fonction de hachage permettant de la « distinguer » d'une fonction réellement aléatoire.

1.2 Résumé des propriétés désirables

En résumé, et plus formellement, une fonction de hachage cryptographique *H* fait correspondre à toute chaîne de caractères (*i.e.* une chaîne de bits de longueur quelconque) une empreinte de longueur *n* fixe, et doit être à la fois :

- résistante aux pré-images ;
- résistante aux deuxièmes pré-images ;
- résistante aux collisions ;
- rapide à calculer ;

et idéalement :

- indistinguable d'une fonction aléatoire.

Les propriétés mentionnées ci-dessus sont les plus importantes en pratique, mais de nombreux raffinements, pertinents dans certains cas précis, ont été introduits : la résistance à la presque-collision et aux collisions multiples s'avère ainsi utile lors de l'interaction de la fonction de hachage avec d'autres primitives qui lui sont combinées. De plus, les relations entre ces diverses propriétés ont été étudiées.

1.2.1 Résistance au calcul de collisions

Le célèbre paradoxe des anniversaires implique que hacher des données aléatoires $2^{n/2}$ fois permet de trouver une collision avec une forte probabilité. Ainsi, pour obtenir un niveau de sécurité s donné, il est nécessaire de considérer des hachés de longueur au moins *n*=2s.

La résistance maximale de MD5 aux collisions est donc de 2^{64}, celle de SHA-1 de 2^{80} et celle de SHA-256 de 2^{128}. Il s'agit là de simples bornes supérieures et, comme expliqué ultérieurement, un attaquant peut être en mesure de calculer des collisions plus efficacement qu'en les recherchant aléatoirement.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

1.2.2 Résistance au calcul de pré-images

De la propriété d'uniformité de la fonction de hachage découle qu'une pré-image peut être calculée en 2^n hachés. Un niveau de sécurité s requiert donc des hachés de longueur au moins n=s. Il s'agit là aussi d'une simple borne supérieure.

2 L'état actuel des fonctions de hachage

Même si le parallélisme est en général une bonne propriété pour une primitive que l'on souhaite rapide, une fonction de hachage ne peut être construite entièrement parallèle (traitant simultanément tous les blocs de message). En effet, la quantité de mémoire requise par l'état interne de la fonction pour le calcul du haché dépendrait de la taille du message à hacher. Si le message est très long, une utilisation en environnement contraint en mémoire serait impossible. Ainsi, la première grande avancée dans le domaine fut l'algorithme de Merkle-Damgård **[M89, D89]**, qui permit de hacher itérativement tout en traitant les blocs de message séquentiellement, permettant ainsi de traiter les blocs de message comme un flux (sans devoir attendre le dernier bloc), ce qui est très appréciable en pratique. Bien que le procédé date de 1989, la quasi-totalité des fonctions de hachage actuelles y font appel.

Une fonction de hachage itérative est composée de deux transformations : la fonction de compression et la fonction d'extension de domaine. La première est identique à une fonction de hachage, à l'exception de la taille fixe des messages d'entrée. La seconde spécifie l'utilisation de la fonction de compression pour permettre de traiter des entrées de taille variable.

2.1 Les constructions des années 1990-2000

2.1.1 La fonction d'extension de domaine

La construction de Merkle-Damgård est une fonction d'extension de domaine. Dans sa version classique, la fonction de hachage H possède un état interne de n bits (ce qui est le minimum pour espérer construire une fonction de hachage sûre de n bits de sortie), initialisé à une certaine valeur prédéterminée (appelée vecteur d'initialisation ou IV). Le message à hacher est rembourré, de telle manière qu'il puisse être divisé en blocs de taille égale à m bits. Durant la i-ème itération, la fonction de compression h met à jour itérativement l'état interne H_i (aussi appelée variable de chaînage) avec le i-ème bloc de message M_i : $H_{i+1}=h(H_i,M_i)$. Lorsque tous les blocs de message ont été traités, l'état interne représente la valeur du haché final.

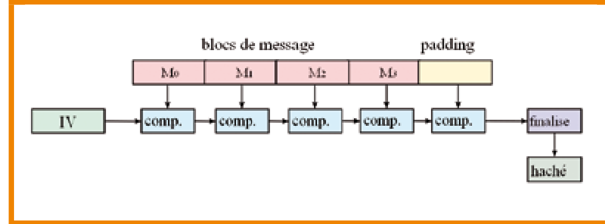


Figure 1 : Construction de Merkle-Damgård

La plupart des fonctions standardisées utilisent cette structure, comme la famille MD (MD4, MD5) et la famille SHA (SHA-1, SHA-256, SHA-512). L'attrait de cette méthode pour les concepteurs de fonctions de hachage est l'élégante preuve de sécurité sous-jacente : si une collision est trouvée pour la fonction de hachage, alors une collision a forcément été trouvée aussi pour la fonction de compression. Cette réduction est extrêmement intéressante car elle permet de transformer un problème complexe (construire une fonction de hachage sûre, *i.e.* pour laquelle il est difficile de trouver une collision) en un problème plus simple (construire une fonction de compression sûre). Il reste bien entendu au concepteur à définir la fonction de compression.

2.1.2 La fonction de compression

La communauté académique a vite constaté que la construction d'un algorithme de chiffrement par bloc (tel l'AES) et la construction d'une fonction de compression sont des problèmes très similaires. Ainsi, quelques fonctions de compression reposent sur l'utilisation d'un algorithme de chiffrement par bloc. La plus connue est la construction de Davies-Meyer : $H_{i+1} = h(H_i,M_i) = E_{M_i}(H_i) \oplus H_i$, où $E_K(M)$ dénote le chiffrement selon l'algorithme E du bloc de message M avec la clé K.

Là aussi, l'intérêt de la construction découle d'une preuve réduisant un problème complexe à un problème plus simple : pour construire une fonction de compression, il suffit de construire un algorithme de chiffrement par bloc, un exercice déjà bien maîtrisé.

Ainsi les fonctions de compression de la famille MD ou SHA utilisent la méthode de Davies-Meyer, avec un chiffrement par bloc interne selon le modèle de Feistel généralisé, ce qui se faisait de mieux lors de leur conception. L'algorithme de chiffrement interne à SHA2, nommé SHACAL-2, fut même recommandé par l'Union Européenne à travers le projet NESSIE. À l'intérieur du Feistel, les opérations utilisées (opérations booléennes, additions modulo 2^{32}, rotations)

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

ont été choisies principalement pour leur rapidité d'exécution sur les processeurs 32/64 bits modernes (nous y revenons dans la partie 4 consacrée aux implantations).

2.2 Le coup de tonnerre de 2004

Des années 90 à 2004, malgré quelques avancées sur MD4 et MD5, les fonctions de hachage restèrent relativement peu analysées par la communauté. La plupart des attaques utilisaient la cryptanalyse différentielle classique (l'étude de la propagation durant le calcul de la fonction d'une certaine différence entre deux entrées jusqu'à la sortie) pour espérer trouver des collisions. À noter que SHA-0 (le premier standard SHA de la NSA, datant de 1993) fut mis à jour par SHA-1 en 1995. La seule différence était l'ajout d'une simple rotation dans l'algorithme et aucune justification ne fut fournie. On découvrit plusieurs années plus tard **[CJ98]** que certaines attaques s'appliquaient à SHA-0 et non à SHA-1...

MD4	128 bits	totalement cassée
MD5	128 bits	totalement cassée
SHA-0	160 bits	totalement cassée
SHA-1	160 bits	cassée théoriquement
SHA-224	224 bits	sécurité précaire
SHA-256	256 bits	sécurité précaire
SHA-384	384 bits	sécurité précaire
SHA-512	512 bits	sécurité précaire

Figure 2 : État de la technique pour les standards de hachage actuels

Durant l'été 2004, un des plus importants résultats de cryptanalyse fut présenté à la « rump session » de la conférence *CRYPTO* : une équipe de chercheurs chinois démontra qu'il est possible de trouver des collisions de manière pratique pour MD4, MD5, SHA-0, ainsi que pour d'autres fonctions de cette famille. Pire, quelques mois plus tard, SHA-1, le standard du NIST en 2004, était lui aussi cassé par ces attaques, même si la complexité (2^{69} opérations) était trop importante pour être réalisable en pratique. Ces attaques, ainsi que les améliorations qui suivirent, eurent un très gros impact sur les techniques de cryptanalyse et les méthodes de construction.

Ces attaques furent le fruit de deux principales innovations. Tout d'abord, les chemins différentiels utilisés pour la cryptanalyse différentielle étaient beaucoup plus complexes (et bien meilleurs) que précédemment. Ensuite, les chercheurs exploitèrent mieux la principale différence entre une fonction de compression et un algorithme de chiffrement par bloc, à savoir les degrés de liberté. En effet, il devient très vite difficile de prédire le fonctionnement interne lors de l'exécution d'un algorithme de chiffrement par bloc, celui-ci impliquant une clé secrète. Une fonction de hachage, à l'inverse, laisse un contrôle total à l'attaquant sur les entrées. L'idée, très complexe à mettre en œuvre, est d'effectuer de petites modifications locales du message afin de réduire la complexité finale de l'attaque.

Ces attaques d'une grande complexité étaient d'autant plus impressionnantes qu'elles furent trouvées « à la main ». Il fallut d'ailleurs plusieurs années à la communauté pour corriger les quelques erreurs que contenait l'attaque originale.

Les fonctions de hachage sont aujourd'hui très étudiées et d'autres propriétés sensibles telles que la résistance au calcul de pré-image ou de seconde pré-image sont analysées. Seuls les standards SHA-256 et SHA-512 résistent, mais le fait qu'ils reposent sur les mêmes techniques de construction que les autres fonctions de la famille MD/SHA a conduit le NIST à lancer en 2008 un appel à soumissions dans l'espoir de trouver le prochain standard SHA-3.

3 Le futur des fonctions de hachage : le concours SHA-3

Le domaine des fonctions de hachage est entré dans une nouvelle ère depuis les attaques de 2004 et le lancement du concours SHA-3 par le NIST en 2008 **[SHA3]**. La recherche s'est intensifiée, permettant une meilleure compréhension des fonctions de hachage.

3.1 Nouvelles attaques

Le NIST a reçu 64 soumissions dont 51 considérées valides. Après une première sélection retenant 14 candidats en juillet 2009, 5 finalistes ont été annoncés en décembre 2010 : BLAKE, GRØSTL, JH, KECCAK et SKEIN. Le choix final sera annoncé *a priori* autour d'octobre 2012.

Le processus de filtrage des candidats a nécessité un gros effort de cryptanalyse de la part de la communauté académique conduisant à de nombreuses avancées. Par exemple, l'attaque par rebond eut raison de nombreux candidats utilisant l'AES (ou ses composants) en trouvant des collisions plus rapidement que l'attaque générique : les différences de l'état interne sont fixées à deux instants du calcul, puis propagées vers l'intérieur de

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

la fonction pour trouver des contraintes et repartent alors vers l'extérieur de la fonction (rebond) jusqu'à atteindre les deux extrémités.

Il est intéressant d'observer que historiquement, le chiffrement par bloc était beaucoup mieux maîtrisé que les fonctions de hachage, les critères de sécurité pour la construction (résistance à la cryptanalyse linéaire et différentielle) provenant des années de cryptanalyse des algorithmes de chiffrement par bloc. L'équilibre est aujourd'hui rétabli puisque certaines attaques dédiées aux fonctions de hachage ont abouti à une meilleure compréhension de l'AES : la toute récente attaque « biclique » **[BKR11]**, descendante directe de l'attaque par rebond et de certaines méthodes de calcul de pré-image, permet de retrouver la clé secrète de l'AES deux fois plus vite que par recherche exhaustive.

3.2 Nouvelles constructions

Le NIST a spécifié que SHA-3 devait prendre une variable de « tweak » en entrée, rendant ainsi l'utilisation de la primitive plus flexible (permettant par exemple de « saler » le hachage de mots de passe).

Des constructions très variées furent soumises au concours, mais la plus grande innovation concerne l'extension de domaine (le mode qui transforme un algorithme de chiffrement par bloc en fonction de compression) suite à l'identification de propriétés potentiellement néfastes dans l'algorithme de Merkle-Damgård. Un grand nombre de correctifs furent proposés : utilisation d'un état interne de $2n$ bits pour un haché de n bits au détriment de la mémoire interne requise (empêchant toute implémentation très compacte en hardware), nouvelle fonction de sortie ou compteur d'itération en entrée de la fonction de compression.

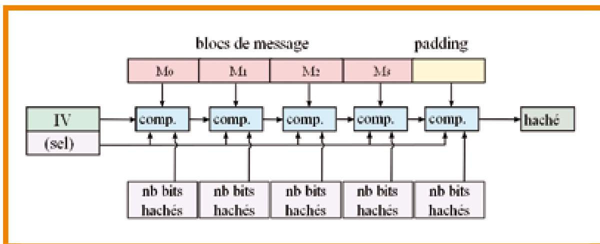


Figure 3 : Fonction d'extension de domaine HAIFA ***[BD06]***

Sur le plan structurel, les fonctions « éponges » n'utilisant qu'une permutation fixe publique (au lieu d'un chiffrement par bloc muni d'une clé secrète) se démarquent par d'intéressantes preuves de sécurité et une certaine flexibilité : elles ont récemment donné des variantes légères adaptées aux environnements très contraints.

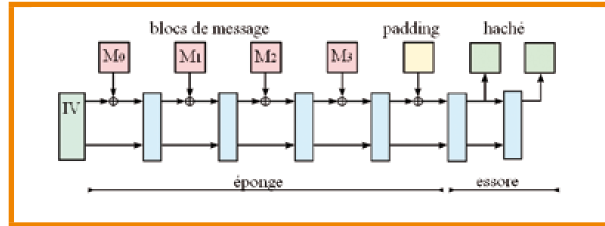


Figure 4 : Fonctions éponges

Enfin, les chiffrements internes proposés restent classiques, renforçant l'impression de maturité des chiffrements par bloc actuels :

- ARX (*Addition-Rotation-XOR*) composés d'additions modulo 2^{32} ou 2^{64}, de rotations et de fonctions booléennes, extrêmement rapides sur processeurs 32 ou 64 bits, mais moins sur d'autres plateformes ; leur inconvénient principal réside dans la complexité de l'analyse de leur sécurité.
- SPN (*Substitution Permutation Network*, tel que l'AES) très souvent accompagnés de preuves de sécurité, mais le calcul des substitutions est généralement coûteux.

BLAKE et SKEIN relèvent du premier groupe, GRØSTL et JH du second, KECCAK se situe entre les deux. Puisque ces 5 candidats sont loin d'être cassés par les meilleures attaques connues, le choix du NIST sera certainement guidé par les performances. BLAKE et SKEIN ont *a priori* un avantage certain pour la dernière ligne droite du concours.

4 Aspects d'implantation

4.1 API du NIST

Afin de faciliter le test des performances, le NIST a imposé une API commune se résumant à trois fonctions principales :

- *Init* : prend en entrée une référence sur l'état interne ainsi que le nombre de bits de sortie de la fonction de hachage (224, 256, 384 ou 512 tels que spécifiés par le NIST).
- *Update* : prend en entrée une référence sur l'état interne, une suite de bits et sa taille. Update fait évoluer l'état interne lors de la réception de blocs de message.
- *Final* : finalise, à la réception du dernier bloc de message, le calcul du haché ; implante en particulier l'opération de rembourrage.

Le hachage d'un message de taille donnée se fera donc via un appel à Init, un ou plusieurs appels à Update, et un appel final à Final pour traiter le dernier bloc et

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

rendre le haché. La forme de l'état interne, le mode opératoire ainsi que les opérations de rembourrage, très spécifiques à chaque candidat, sont abstraites par cette API.

Enfin, pour tester la validité des implantations, le NIST a spécifié un ensemble de vecteurs pour lesquels les hachés sont connus (*Known Answer Test*). Ce type de tests a néanmoins ses limites : il ne révèle en aucun cas les incohérences entre la spécification et l'implantation de référence, comme cela a pu arriver à certains candidats **[SHV]**, les obligeant à soumettre de nouvelles implantations de référence.

La suite eBASH **[EBASH]**, sous-partie du logiciel SUPERCOP développé dans le cadre du projet européen ECRYPT, offre un mécanisme de choix pour mesurer de manière homogène les performances des implantations logicielles des divers algorithmes sur diverses plateformes, et pour divers types de messages (longs, courts). L'API imposée, la vérification systématique des vecteurs de tests, ainsi que l'exhaustivité des tests font de eBASH une source de résultats relativement complète et objective.

Les performances données par eBASH sont en cycles par octet, c'est-à-dire le nombre de cycles CPU nécessaires en moyenne pour hacher un octet du message passé en entrée. Cette mesure a l'avantage de ne pas dépendre de la fréquence du CPU considéré, et d'être homogène par familles de microarchitectures considérées.

Une fonction de hachage standardisée doit idéalement obtenir de bonnes performances sur un large spectre de plateformes, mais également dans les différents scénarios d'utilisation : certains protocoles hachent exclusivement des messages courts de taille fixe (mots de passe, dérivation de clé, mode compteur), alors que d'autres applications condensent de plus longs messages (intégrité de fichiers).

4.2 Implantations logicielles

4.2.1 Les plateformes x86

La plateforme de prédilection pour départager les candidats durant les premiers tours a été x86 (32 bits et 64 bits) : ces CPU sont historiquement au cœur de la plupart des serveurs et des postes clients.

Les fonctions de hachage à base d'ARX s'implantent aisément sur ce type de plateforme. En règle générale, l'état interne de ces fonctions manipule directement des mots de taille 32 ou 64 bits, ce qui permet d'effectuer toutes les opérations (additions, rotations, opérations booléennes) en peu d'instructions et en limitant les transferts entre la mémoire et les registres du CPU. BLAKE et SKEIN tournent par exemple autour de 15 cycles par octet sur x86 en 32 bits, et en dessous des 10 cycles par octet en 64 bits. L'amélioration significative entre 32 et 64 bits tient principalement au doublement de la taille des registres (ainsi que de leur nombre), divisant par deux le nombre d'opérations booléennes séquentielles.

Les fonctions reposant sur un SPN sont plus lentes car elles nécessitent plusieurs « table lookups », soit autant de lectures en mémoire s'ajoutant aux autres opérations effectuées dans les registres. Malgré la faible latence des caches CPU actuels et leur taille élevée (permettant de s'assurer que les tables restent en cache), les opérations de lecture mémoire restent plus coûteuses que les opérations « internes » aux registres (de l'ordre de 4 cycles pour une lecture en cache L1, contre en moyenne 1 cycle pour une opération inter-registres). En l'occurrence, GRØSTL et JH tournent entre 30 et 40 cycles par octet sur architecture 32 bits, et entre 15 et 20 cycles par octet sur architecture 64 bits.

Des implantations logicielles alternatives par rapport aux implantations utilisant les registres généraux permettent parfois de meilleures performances et une meilleure résistance aux canaux cachés grâce à des implantations en temps constant :

- Utilisation des registres xmm de 128 bits (ymm de 256 bits sur les CPU les plus récents) qui permettent de profiter d'un parallélisme accru sur les opérations booléennes et arithmétiques impliquant les octets et les mots de 16/32/64/128 bits. Ces registres disposent aussi d'un jeu d'instructions SSE spécifiques permettant des opérations qui sont par ailleurs complexes dans les registres généraux : l'instruction `pshuffb` permet ainsi d'appliquer une permutation d'octets (« shuffle ») dans un registre xmm en une instruction, autorisant l'implantation de fonctions de substitution directement dans les registres xmm (cette technique a d'ailleurs été appliquée à la SBox de 256 octets de l'AES **[H09]**).
- Les instructions AES-NI récemment introduites par Intel accélèrent le calcul des tours de l'AES (depuis la génération Westmere) ; contrairement aux coprocesseurs de SoC (*System on a Chip*) ou de cartes à puce, l'AES n'est pas implanté en une opération atomique mais décomposé en toutes ses opérations de base. GRØSTL, par exemple, qui n'utilise que la SBox de l'AES, tire parti de ces instructions **[AESNI]**.
- Le « bitslicing » utilise le jeu d'instructions SSE pour remplacer la substitution et les autres opérations linéaires et booléennes par leur forme algébrique sous forme parallèle **[MN07]**.

L'inconvénient de ces implantations alternatives est l'utilisation de jeux d'instructions non standards et donc peu portables.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

4.2.2 Terminaux embarqués avec CPU 32 bits

L'omniprésence de terminaux embarquant des processeurs basse consommation de type RISC (ARM pour les mobiles et MIPS pour les routeurs et les *set-top-box* en tête) impose au futur standard d'avoir des performances décentes sur ces terminaux. Le banc de test **[XBX]**, complémentaire de eBASH, permet de tester les implantations sur ce type de plateformes.

Comme les sur processeurs x86, l'accès à la mémoire (et aux caches) est sensiblement plus lent que les opérations booléennes et arithmétiques effectuées au sein des registres : seul BLAKE permet en moyenne d'être aussi rapide qu'un SHA-256 sur les plateformes testées. L'analogie avec le x86 ne s'arrête pas là puisque les jeux d'instructions vectorielles SIMD (*Single Instruction Multiple Data*, par exemple NEON sur les SoC ARM Cortex) sont largement exploités. Cependant, là où Intel et AMD enrichissent un même jeu d'instructions SSE en x86, les SoC ARM peuvent utiliser des coprocesseurs SIMD assez différents et incompatibles en termes d'opcodes (c'est le cas de NEON et de iwMMXt, la variante MMX introduite par Intel dans ses processeurs ARM xScale).

Parallèlement aux performances pures (cycles par octet), le monde de l'embarqué introduit de nouvelles contraintes en fonction des terminaux considérés : les mobiles tendent vers les PC avec des quantités de RAM et de stockage flash qui augmentent, alors que les routeurs et autres set-top-box ont un environnement fortement contraint en termes de mémoires (RAM et ROM). L'empreinte mémoire des programmes au stockage et à l'exécution est donc aussi critique que leur rapidité. La suite XBX permet de mesurer les empreintes RAM et ROM de chaque code soumis : le candidat BLAKE a là aussi, en moyenne, les meilleurs résultats sur les plateformes 32 bits testées (très bonne compacité en mémoire tout en gardant une bonne rapidité).

4.2.3 Micro-contrôleurs et cartes à puce

Bien que le marché évolue vers l'utilisation de CPU 32 bits « low end » tels que le Cortex-M d'ARM, en général dépourvus de cache, de MMU et de coprocesseurs avancés, de nombreux processeurs 8 bits et 16 bits « basiques » sont encore utilisés dans les micro-contrôleurs et les cartes à puce (comme l'AVR d'Atmel ou les MSP430 de TI).

Malgré leur faible puissance et leur environnement très contraint, l'utilisation de fonctions de hachage sur ce type de plateformes est souvent une nécessité.

Les micro-contrôleurs (utilisés par exemple dans certaines puces RFID) requièrent un standard SHA-3 avec des performances et une empreinte mémoire décentes.

Le cas des cartes à puce est un peu particulier : la cryptographie étant au cœur de ces puces, le fabricant y inclut en général des circuits dédiés aux opérations de chiffrement, de signature et de hachage sous forme de coprocesseurs ASIC. Néanmoins, le standard de chiffrement AES a 15 ans, et ce n'est que depuis peu que les cartes à puce du marché implantent une accélération matérielle dédiée. Il est donc important que le futur standard SHA-3 ne soit pas trop lent dans ses variantes logicielles 8 et 16 bits.

La suite XBX inclut des plateformes caractéristiques des micro-contrôleurs et des cartes à puce, et il y a de fortes chances que le NIST tienne compte des résultats des finalistes sur ces terminaux lors du choix ultime. BLAKE et KECCAK sont en tête sur les plateformes 8 bits, tandis que BLAKE et GRØSTL le sont sur 16 bits.

4.2.4 Autres plateformes

Enfin, des plateformes plus exotiques émergent. C'est le cas des GPU dont la microarchitecture particulière les éloigne assez fortement d'un CPU classique. C'est aussi le cas des processeurs CELL qui, bien qu'embarquant un cœur PowerPC, introduisent de nouvelles facilités de parallélisme SIMD pouvant profiter aux algorithmes cryptographiques (ces processeurs sont notamment utilisés dans les consoles PS3 de Sony). Ces plateformes, vu leur faible usage industriel actuel, n'auront vraisemblablement pas de rôle significatif dans le choix du NIST.

4.3 Implantation matérielle

Les raisons principales pour l'implantation matérielle d'un algorithme sont la performance (à travers un circuit dédié) et la sécurité (plus difficile à accéder et modifier). Ces implantations sont par exemple utilisées comme « root of trust » dans les chaînes de boot sécurisé des PC (via le TPM), ou dans certains SoC dont le firmware doit être protégé (set-top-box avec DRM, ou pour le secret et l'intégrité du *firmware* de terminaux). L'implantation matérielle permet aussi des calculs en temps constant, éliminant certains canaux cachés (une des raisons de l'introduction des AES-NI par Intel, par exemple).

4.3.1 Implantation sur FPGA

Les circuits FPGA (*Field-Programmable Gate Array*) contiennent une matrice de cellules programmables appelées « slices ». Celles-ci comportent des tables LUT (*Look-Up Tables*) et des bascules calculant de petites tables de vérité de fonctions booléennes locales reliées entre elles via du routage et du multiplexage de cellules. N'importe quel circuit numérique est donc implantable via sa réduction en réseau de sous-fonctions booléennes.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Abonnez-vous !

Profitez de nos offres d'abonnement spéciales disponibles au verso !

Économisez plus de 25%*

* Sur le prix de vente unitaire France Métropolitaine

6 Numéros de MISC

par ABONNEMENT : 38€*

au lieu de 51,00 €* en kiosque

Économie : 13,00 €*

*OFFRE VALABLE UNIQUEMENT EN FRANCE MÉTROPOLITAINE
Pour les tarifs hors France Métropolitaine, consultez notre site :
www.ed-diamond.com

Les 3 bonnes raisons de vous abonner :

- Ne manquez plus aucun numéro.
- Recevez MISC dès sa parution chez vous ou dans votre entreprise.
- Économisez 13,00 €/an !

4 façons de commander facilement :

- par courrier postal en nous renvoyant le bon ci-dessous
- par le Web, sur www.ed-diamond.com
- par téléphone, entre 9h-12h et 14h-18h au 03 67 10 00 20
- par fax au 03 67 10 00 21

Bon d'abonnement à découper et à renvoyer à l'adresse ci-dessous

Tournez SVP pour découvrir toutes les offres d'abonnement >>

Édité par Les Éditions Diamond
Service des Abonnements
B.P. 20142 - 67603 Sélestat Cedex
Tél. : + 33 (0) 3 67 10 00 20
Fax : + 33 (0) 3 67 10 00 21

Voici mes coordonnées postales :	
Société :	
Nom :	
Prénom :	
Adresse :	
Code Postal :	
Ville :	
Pays :	
Téléphone :	
e-mail :	

Vos remarques :

....

....

....

En envoyant ce bon de commande, je reconnais avoir pris connaissance des conditions générales de vente des Éditions Diamond à l'adresse internet suivante : **www.ed-diamond.com/cgv** et reconnais que ces conditions de vente me sont opposables.

Tournez SVP pour découvrir toutes les offres d'abonnement

Ce document est la propriété exclusive deJohann Lo

PROFITEZ DE NOS OFFRES D'ABONNEMENT SPÉCIALES POUR LIRE PLUS ET FAIRE DES ÉCONOMIES !

Voici nos offres de couplage

offre 1 — MISC (6 nos) — par ABO : 38€* — au lieu de 51,00€** en kiosque — Economie : 13,00 €

offre 2 — Linux Pratique Essentiel (6 nos) + Linux Pratique (6 nos) — par ABO : 57€* — au lieu de 78,00€** en kiosque — Economie : 21,00 €

offre 3 — GNU/Linux Magazine (11 nos) + Linux Pratique (6 nos) — par ABO : 78€* — au lieu de 121,50€** en kiosque — Economie : 43,50 €

offre 4 — GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) — par ABO : 83€* — au lieu de 130,50€** en kiosque — Economie : 47,50 €

offre 5 — GNU/Linux Magazine (11 nos) + Misc (6 nos) — par ABO : 84€* — au lieu de 133,50€** en kiosque — Economie : 49,50 €

offre 6 — GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) — par ABO : 110€* — au lieu de 169,50€** en kiosque — Economie : 59,50 €

offre 7 — GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Misc (6 nos) — par ABO : 116€* — au lieu de 181,50€** en kiosque — Economie : 65,50 €

offre 8 — GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Misc (6 nos) — par ABO : 143€* — au lieu de 220,50€** en kiosque — Economie : 77,50 €

offre 9 — Linux Pratique Essentiel (6 nos) + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Misc (6 nos) — par ABO : 173€* — au lieu de 259,50€** en kiosque — Economie : 86,50 €

offre 10 — MISC (6 nos) + MISC Hors-Série (2 nos) — par ABO : 44€* — au lieu de 69,00€** en kiosque — Economie : 25,00 €

offre 11 — Linux Pratique (6 nos) + Linux Pratique HS (3 nos) — par ABO : 42€* — au lieu de 63,00€** en kiosque — Economie : 21,00 €

offre 12 — Linux Pratique Essentiel (6 nos) + GNU/Linux Magazine (11 nos) + GNU/Linux Magazine HS (6 nos) + Linux Pratique (6 nos) + Linux Pratique HS (3 nos) + Misc (6 nos) + MISC Hors-Série (2 nos) — par ABO : 199€* — au lieu de 301,50€** en kiosque — Economie : 102,50 €

offre 15 — Linux Pratique Essentiel (6 nos) + Linux Pratique (6 nos) + Linux Pratique HS (3 nos) — par ABO : 72€* — au lieu de 102,00€** en kiosque — Economie : 30,00 €

Vous pouvez également vous abonner sur : www.ed-diamond.com ou par Tél. : 03 67 10 00 20 / Fax : 03 67 10 00 21

Nos Tarifs s'entendent TTC et en euros

	F France Métro	D DOM	T TOM	E1 Europe 1	E2 Europe 2	EUC Etats-Unis Canada	A Afrique	RM Reste du Monde
1 Abonnement MISC	38 €	40 €	44 €	45 €	44 €	46 €	45 €	49 €
2 LPE + LP	57 €	62 €	69 €	71 €	69 €	73 €	71 €	79 €
3 GLMF + LP	78 €	85 €	96 €	99 €	95 €	101 €	98 €	111 €
4 GLMF + GLMF HS	83 €	89 €	101 €	104 €	100 €	105 €	103 €	116 €
5 GLMF + MISC	84 €	90 €	102 €	105 €	101 €	107 €	104 €	117 €
6 GLMF + GLMF HS + Linux Pratique	110 €	119 €	134 €	138€	133 €	140 €	137 €	154 €
7 GLMF + GLMF HS + MISC	116 €	124 €	140 €	144 €	139 €	146 €	143 €	160 €
8 GLMF + GLMF HS + MISC + LP	143 €	154 €	173 €	178 €	172 €	181 €	177 €	198 €
9 GLMF + GLMF HS + MISC + LP + LPE	173 €	186 €	209 €	215 €	208 €	219 €	214 €	239 €
10 MISC + MISC HS	44 €	47 €	53 €	55 €	52 €	56 €	54 €	60 €
11 LP + LP HS	42 €	46 €	52 €	54 €	51 €	55 €	53 €	60 €
12 GLMF + GLMF HS + MISC + MISC HS + LP + LP HS + LPE	199 €	214 €	242 €	250 €	239 €	254 €	247 €	277 €
15 LPE + LP + LP HS	72 €	78 €	88 €	91 €	87 €	93 €	90 €	101 €

- Europe 1 : Allemagne, Belgique, Danemark, Italie, Luxembourg, Norvège, Pays-Bas, Portugal, Suède
- Europe 2 : Autriche, Espagne, Finlande, Grande Bretagne, Grèce, Islande, Suisse, Irlande
- Zone Reste du Monde : Autre Amérique, Asie, Océanie
- Zone Afrique : Europe de l'Est, Proche et Moyen-Orient

* Toutes les offres d'abonnement : en exemple, les tarifs ci-dessus correspondant à la zone France Métro (F) ** Base tarifs kiosque zone France Métro (F)

Mes choix :

Mon 1er choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 2ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
Mon 3ème choix	Je sélectionne le N° (1 à 15) de l'offre choisie :	
	Je sélectionne ma zone géographique (F à RM) :	
	J'indique la somme due : (Total)	€

Exemple : je souhaite m'abonner à l'offre GNU/Linux Magazine + GNU/Linux Magazine Hors-série + MISC (offre 7) et je vis en Belgique (E1), ma référence est donc 7E1 et le montant de l'abonnement est de 144 euros.

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond
- ❑ Carte bancaire n° ____ ____ ____ ____

Expire le : ____

Cryptogramme visuel : ___

Votre cryptogramme visuel

Date et signature obligatoire

MISC

Ce document est la prop[...] a.com) le 1 Mars 2015

L'architecture FPGA est à la fois un avantage et un inconvénient : elle est versatile et reconfigurable mais offre de moindres performances et peut s'avérer plus coûteuse que celle d'un ASIC. Elle est d'ailleurs utilisée pour le prototypage d'ASIC. Elle peut aussi être utile dans les cas où le matériel doit être reconfigurable, ou lorsque les volumes de production sont faibles.

Les performances d'un FPGA résident dans sa taille (surface du circuit, mesurée en nombre de slices occupés) et sa rapidité (aussi nommée « throughput », ou nombre de bits du message traités à la seconde ; elle est proportionnelle à la fréquence du circuit et inversement proportionnelle à sa latence).

Un circuit rapide a des unités « pipelinées » fonctionnant en parallèle et donc de nombreux slices alors qu'un circuit compact réutilise au maximum les unités au détriment de la latence globale (et donc du *thoughput*). Il est de plus intéressant qu'un algorithme offre de nombreux compromis rapidité/compacité afin de s'adapter au mieux au cas d'usage.

4.3.2 Implantation ASIC

Les ASIC (*Application Specific Integrated Circuits*) sont des circuits intégrés fondus (non reprogrammables) dédiés à une tâche spécifique. Ils sont à la base des processeurs, micro-contrôleurs, DSP et autres coprocesseurs. Contrairement aux FPGA, ils sont extrêmement coûteux à la conception et à la fabrication et ne sont rentabilisés qu'au travers de gros volumes de production.

À la surface (mesurée en nombre de portes logiques) et la rapidité (mesurée en throughput, comme sur FPGA) s'ajoute la consommation pour caractériser leurs performances.

4.3.3 Les candidats SHA-3 sur FPGA et ASIC

Les performances sur FPGA et ASIC dépendent fortement de la génération et de la technologie utilisées : comparer des résultats ASIC sur technologies de gravure 35 nm et 18 nm a peu de sens.

Le banc de test Athena **[ATHENA]** compare donc les implantations matérielles en VHDL ou Verilog en les synthétisant sur différentes plateformes cibles, assurant la cohérence des résultats. Au-delà du simple classement par throughput et par surface, le ratio entre les grandeurs est donné, permettant d'évaluer à quel « prix » en surface les performances évoluent en moyenne. Cela traduit une réalité industrielle : rien ne sert à un algorithme d'être extrêmement rapide si le coût de fabrication associé à sa surface est prohibitif.

Malgré des résultats très variables en fonction de l'implantation soumise et de la plateforme considérée, une tendance claire se dégage à la fois sur FPGA et sur ASIC : parmi les 5 finalistes, KECCAK est sans conteste la fonction la plus à l'aise en matériel, obtenant de bons résultats à la fois en rapidité et en compacité. En moyenne, GRØSTL, JH et BLAKE arrivent souvent à compenser une faible compacité par un throughput plutôt élevé (ou réciproquement), tandis que SKEIN reste dans la moyenne basse des performances.

4.4 Les canaux cachés

Les attaques par canaux cachés extraient un secret utilisé par un algorithme cryptographique à travers l'information fuyant d'une implantation. La fuite provient par exemple de la consommation électrique, du rayonnement électromagnétique, des latences de calcul, ...

Les fonctions de hachage, utilisées dans certaines constructions de MAC (*Message Authentication Code*), prennent un message M et une clé secrète K en entrée et produisent une valeur H de taille fixe permettant l'authentification du message M auprès des parties possédant la clé K. La récupération de K à travers un canal caché doit être rendue impossible. Plus généralement, les fonctions de hachage manipulant des secrets sont potentiellement menacées. Il ressort de **[BP10]** que les candidats à base d'AES, comme GRØSTL, sont plus facilement attaquables à cause des propriétés de la SBox (permettant à l'attaquant d'extraire la bonne clé en relativement peu de captures). Les attaques contre les ARX peuvent être, elles, moins aisées à mener.

Notons par ailleurs que certains candidats, comme GRØSTL, ont mis l'accent sur l'implantation de versions logicielles **[GSTL]** immunes aux canaux cachés temporels existants dans le cache des CPU **[OS06]**. Ce type de canal caché est en effet à considérer dès lors que l'algorithme cryptographique manipule des tables en mémoire indexées en partie par le secret K (ce qui est le cas pour les SPN lorsque la substitution est implantée sous forme de tables). Une implantation est immune à ces attaques lorsque les opérations manipulant la clé s'effectuent en « temps constant », ce qui est réalisable de diverses manières (voir 4.2.1 : l'utilisation d'instructions AES-NI, de SBox sous forme de permutations SIMD, de *bitslicing*, ...).

Conclusion

Les avancées en cryptanalyse pour les fonctions de hachage ont motivé le lancement par le NIST de la compétition SHA-3. La compétition a apporté une plus grande maîtrise de leur sécurité avec de nouvelles attaques et de nouvelles constructions, celles-ci ayant parfois un impact dans le domaine des chiffrements par bloc. Mais la compétition a aussi permis de mettre en lumière l'importance des aspects industriels dans la construction de primitives cryptographiques : le défi ne tient pas (seulement) dans la seule sécurité de la fonction de hachage, mais bien dans le difficile équilibre sécurité/performance/coût. ■

Les références de cet article sont disponibles sur : http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6/4

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

DES BONNES PRATIQUES DE PROGRAMMATION DE LA CRYPTOGRAPHIE CONTRE LES ATTAQUES PAR CANAUX AUXILIAIRES

Benoit Feix

mots-clés : CRYPTOGRAPHIE EMBARQUÉE / MICROPROCESSEUR / CANAUX CACHÉS / CONTRE-MESURES / IMPLÉMENTATION SÉCURISÉE

Il n'est pas nécessaire aujourd'hui de présenter ce qu'est un microcontrôleur. On en trouve désormais à profusion dans tous les produits de notre quotidien, que ce soit le programmateur de sa machine à laver, sa carte de paiement ou son téléphone portable, etc. Cependant, entre ces différentes utilisations, le niveau de sécurité requis diffère. On parle de produit (élément) de sécurité dans le cas notamment des cartes à puce de paiement et de santé, des passeports électroniques.

Le développement des microprocesseurs sécurisés a longtemps été tiré vers le haut par le monde de la carte à puce. Depuis leur création dans les années 70, ces circuits ont connu bien des améliorations technologiques. Désormais, ils semblent promis à l'avenir à de multiples évolutions afin de remplir de plus en plus leur rôle de coffre-fort au sein des systèmes et circuits électroniques et non pas uniquement sous forme de carte à puce.

Les puces sécurisées ont pour objectif de garantir la confidentialité des clés cryptographiques (et d'autres secrets) qu'ils contiennent malgré le grand nombre de menaces visant à mettre en défaut leur rôle de coffre-fort inviolable. En pratique, afin de s'assurer de la résistance d'un produit, chaque fabricant doit soumettre son composant à des laboratoires d'évaluation. Les experts en sécurité embarquée de ces centres vont alors examiner en détail l'architecture du produit et particulièrement ses mécanismes de sécurité. Puis, lors d'une seconde phase de tests pratiques, ils vont soumettre le circuit aux attaques logiques et physiques connues pour tester sa résistance réelle. Après des mois d'évaluations (et éventuellement des modifications requises), le composant obtient son certificat de sécurité s'il a passé tous les tests avec succès. Un tel certificat est délivré par les instances gouvernementales suite au rapport émis par le laboratoire d'évaluation.

Parmi les tâches scientifiques de sécurisation de ces produits, il en est une peu ou pas connue aux yeux des utilisateurs : « Comment implémenter de manière sûre les algorithmes cryptographiques dans un tel circuit afin de résister à la multitude d'attaques diverses et variées existantes ? Comment les fabricants protègent-ils ces calculs dans leurs produits des nombreuses attaques ? ».

Nous allons dans ces quelques pages vous montrer que sécuriser un tel produit consiste à additionner plusieurs techniques et plusieurs contre-mesures. Celles-ci, à condition d'être correctement implémentées et bien associées, contreront alors efficacement les attaques connues et protégeront ainsi les secrets contenus. Il est aussi nécessaire de mentionner que l'application de ces protections dépend de la technologie utilisée. Elle peut donc varier d'un microprocesseur à un autre. Enfin, dans ce domaine, les techniques d'attaques évoluent quotidiennement et un produit certifié aura toujours une épée de Damoclès au dessus de son silicium.

1 Environnement de développement dans la carte

Dans l'industrie, les langages de programmation les plus utilisés pour programmer sur un microcontrôleur de sécurité sont l'assembleur, le langage C pour les produits dits « natifs » et le langage Java pour les produits JavaCard.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

L'assembleur est fréquemment utilisé pour le développement des algorithmes cryptographiques. Il est même conseillé afin de contrôler chaque instruction du code natif exécuté. De plus, pour se protéger d'attaques simples en analyse de puissance, le développeur écrit son code de sorte que son exécution soit toujours en temps constant, quelles que soient les données manipulées. Par exemple, il interdira dans son code l'usage de tout branchement conditionnel sensible facilement distinguable. Si le programme est écrit en langage C, cette maîtrise de la séquence des op-codes exécutés est difficile à obtenir (et donc dangereuse en termes de sécurité). Il faut par exemple analyser le code désassemblé généré à partir du programme C pour chaque version du compilateur utilisé, et cela peut varier en fonction des options de compilation choisies. En effet, dans ce dernier cas, comme le code natif généré dépend du compilateur, il est difficile de maîtriser la chaîne d'exécution des op-codes du cœur. Certaines optimisations propres au compilateur peuvent même parfois retirer des contre-mesures ; par exemple de la redondance volontaire de code visant à se protéger d'attaques par perturbation (attaques actives).

La plupart des développeurs diront (avec raison) qu'un code cryptographique correctement développé en assembleur est souvent mieux optimisé que son équivalent écrit en C. Un bon développeur fera un meilleur usage de la mémoire et des registres, et la taille du code développé est plus petite pour des calculs également plus rapides.

Bien entendu, l'assembleur utilisé est propre au cœur embarqué dans le circuit. Ainsi, on peut être amené à développer aussi bien sur un cœur CISC 8 bits INTEL80C51 que sur un cœur RISC de type ARM 32 bits. On en déduit donc que le développement sécurisé d'un algorithme cryptographique se fait plus lentement qu'un développement sur un processeur standard.

Une alternative consiste aussi à développer les parties de code les plus sensibles aux attaques et consommatrices de calcul en assembleur et écrire le reste du code de la fonction en C.

2 Les attaques par canaux auxiliaires en cryptographie et leurs contre-mesures

Nous allons brièvement rappeler quelques généralités sur les attaques par canaux auxiliaires. Pour plus de détails, nous conseillons au lecteur de consulter l'article **[D12]** paru dans MISC Hors-Série n°5 qui présente et explique les attaques.

Une carte à puce est potentiellement vulnérable à différentes attaques ; on parle d'attaques logiques et d'attaques physiques. Les attaques logiques sont généralement dédiées au système embarqué complet ; telles des applets malicieuses en Java ou des manipulations « anormales » de commandes non détectées par le système qui peuvent conduire à une fuite d'information.

Cet article étant consacré à la programmation sécurisée d'algorithmes standards de cryptographie, nous allons nous concentrer ici sur les attaques physiques les concernant. Nous utiliserons la terminologie suivante pour évoquer les différentes types d'attaques physiques :

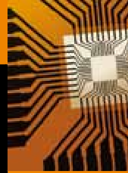
- Les attaques invasives nécessitent du matériel professionnel issu du domaine de l'analyse de défaillance. On parle de stations FIB (Focused Ion Beam), de station de micro-probing et de techniques de rétro-ingénierie hardware utilisant des microscopes électroniques très évolués. De telles techniques impliquent des budgets de plusieurs centaines de milliers d'euros, ce qui n'est pas le cas des attaques suivantes.
- Les attaques passives : il s'agit d'analyser l'énergie consommée (courant) ou les rayonnements émis (électromagnétiques par exemple) par le circuit lors de l'exécution des opérations, et dans notre cas lors des calculs cryptographiques.
- Les attaques actives : le but est de perturber l'exécution du calcul que réalise le microcircuit. En effet, en perturbant le circuit lors d'une opération, celle-ci donne un résultat erroné et donc un résultat cryptographique faux que nous appellerons résultat fauté. En analysant un ou plusieurs de ces calculs erronés, il est possible de retrouver une partie ou la totalité de la clé secrète manipulée.

2.1 Attaques passives

La première publication sur les attaques passives est attribuée à Paul C. Kocher lors de la conférence CRYPTO qui se déroula à Santa Barbara en 1996. Le canal auxiliaire exploité était la mesure du temps d'exécution du calcul. En mesurant le temps nécessaire pour réaliser une exponentiation modulaire m^d modulo n pour différents messages, l'attaquant peut bit après bit retrouver la totalité des bits de l'exposant secret d. Le détail de cette attaque est décrit dans l'article **[K96]**.

En 1998 et en 1999, Paul C. Kocher présente deux nouvelles techniques qui vont marquer l'histoire de la sécurité embarquée. Il s'agit de l'analyse de puissance simple (traduction littérale de *Simple Power Analysis*, SPA), et de l'analyse de puissance différentielle (*Differential Power Analysis*, DPA) sur le DES. Ces deux attaques ont représenté un tournant dans le monde de la carte à puce et ont changé la manière de développer les algorithmes. En 1999, Thomas S. Messerges présente dans **[M99]** la DPA sur l'exponentiation RSA ainsi que les attaques DPA de second ordre **[M00]**.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



Depuis ces articles, de nombreux travaux ont été et sont publiés par de nombreux chercheurs et industriels. À la fois de nouvelles attaques et techniques statistiques d'exploitation des courbes de puissance voient le jour, et d'autre part, de nouvelles contre-mesures pour s'en protéger sont publiées et brevetées. Et à ce fameux jeu du chat et de la souris, sans issue probable, les développeurs doivent empêcher leurs produits de ressembler à du gruyère.

Ainsi, il est possible d'obtenir de l'information exploitable en analysant le courant consommé par le circuit mais aussi ses rayonnements électromagnétiques. Certaines publications scientifiques ont même fait mention d'attaques acoustiques i.e. où la source de la fuite était le signal sonore. Cette dernière source d'information reste aujourd'hui peu exploitée et sa réelle efficacité n'a pas été vraiment démontrée.

Depuis la première DPA, différentes techniques d'attaques plus efficaces ont vu le jour : l'analyse de puissance par corrélation (*Correlation Power Analysis*, CPA) **[BCO04]**, l'analyse d'information mutuelle (*Mutual Information Analysis*, MIA), l'analyse en composantes principales, l'analyse par régression linéaire, l'analyse par collision, par collision corrélation, l'analyse horizontale, l'analyse par dictionnaire de courbes de consommation, plusieurs techniques d'attaques d'ordre supérieur, etc. Chacune de ces techniques a ses propres avantages. Il est même possible d'utiliser les techniques d'analyse passive pour faire de la rétro-ingénierie de circuits et de codes embarqués, on parle alors de SCARE pour *Side Channel Analysis for Reverse Engineering*.

La multiplicité des techniques d'attaques passives rend bien entendu la tâche ardue pour les développeurs lorsqu'il s'agit de vérifier la résistance de leurs produits. Heureusement, il existe des facteurs communs à la plupart de ces menaces :

- L'attaquant doit pour la DPA être capable de prédire une valeur de calcul intermédiaire afin de réaliser ses traitements statistiques.
- La consommation de courant doit être dépendante des données manipulées.
- La consommation de courant doit être dépendante du code exécuté.

Ainsi, si les développeurs parviennent à annuler ces trois composantes critiques de leurs produits, ils pourront rendre inefficaces la plupart des attaques passives.

Mais seulement la « moitié » du travail de sécurisation abordé dans cet article sera faite, puisqu'il reste encore à prendre en compte les attaques actives, et là non plus ce n'est pas une mince affaire.

2.2 Attaques actives

Durant les années 70, les premiers effets des perturbations environnementales sur le comportement des microprocesseurs furent observés dans l'industrie aérospatiale. Mais il fallut attendre 1996 pour que fut publiée leur utilisation à des fins de piratage. Cette année-là, D. Boneh, R. DeMillo et R. Lipton publient la première attaque active (par perturbation) théorique. Leur technique, connue sous le nom de l'attaque Bellcore **[BDL96] [BDL97]**, consiste à perturber plusieurs calculs de signature RSA afin de retrouver, à partir des résultats erronés et des bons calculs, les valeurs des clés secrètes (exposant secret). Puis en 1997, E. Biham et A. Shamir présentent la première analyse différentielle de faute (*Differential Fault Analysis*, DFA) sur l'algorithme DES **[BS97]**.

Depuis 1996, de nombreuses autres attaques par perturbation ont été publiées. On parle aujourd'hui de différents types d'attaques par fautes sur les algorithmes cryptographiques : la DFA, la CFA pour Collision Fault Analysis, l'IFA pour *Ineffective Fault Analysis* et les *Safe Error attacks*, etc.

Début 2000, les moyens d'injections se limitaient à un simple flash d'appareil photo avec une synchronisation peu précise pour injecter la faute, ou un simple générateur de glitches en tension. Désormais, on trouve dans les laboratoires des bancs lasers ou d'injection électromagnétiques perfectionnés avec une synchronisation et localisation spatiale et temporelle extrêmement précises. Les plus perfectionnés de ces outils sont également capables d'injecter plusieurs fautes à différents instants choisis lors de l'exécution d'un calcul par le microcircuit.

Une injection laser visant à modifier une donnée peut entraîner une faute permanente (par exemple sur une donnée en mémoire non volatile) ou une modification transitoire (*transient fault*). Nous considérons ici ce dernier cas qui est le plus fréquent, lorsque la logique du circuit est perturbée ou que des cellules mémoire ou des registres sont stressés. Ainsi, l'effet de la faute va aussi être fonction de la partie du circuit qui est perturbée. Par exemple, injecter une faute lors d'une lecture de la ROM perturbe l'exécution du code, injecter la faute dans une zone de données en RAM modifie les données manipulées, etc.

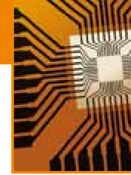
2.3 Définition de contre-mesure

Nous appelons ici par le terme contre-mesure toute technique ayant pour but (et effet) de mettre en défaut une attaque, qu'elle soit active et/ou passive. Il existe plusieurs types de contre-mesures. Nous présentons une classification en trois catégories. Celle-ci ne peut pas être considérée comme standard mais elle résume assez bien les faits.

2.3.1 Trois grandes classes de contre-mesures contre les attaques passives

La première est de type applicatif. En effet, elle est inhérente au type d'application qui utilise l'algorithme

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



cryptographique. Certaines applications ne permettent pas à l'attaquant de choisir le message ou une partie de celui-ci, on peut citer pour illustrer l'application de certains *paddings* lors d'une signature RSA. Il est alors impossible de choisir le message voire de le rejouer. De même, l'insertion de compteurs dans un message à chiffrer induit un effet similaire. L'utilisation de clés de session protège également d'attaques de type DPA car une même clé ne sera utilisée que pour un petit nombre d'exécutions. Néanmoins, qu'advient-il alors de la sécurité du calcul servant à générer la clé de session ?

La deuxième catégorie inclut les contre-mesures ayant pour but de modifier le signal mesuré afin de rendre son interprétation ainsi que son utilisation plus difficiles à l'attaquant. S'il devient difficile à un attaquant d'observer la courbe de consommation de l'algorithme exécuté, il est clair que la tâche est plus ardue pour retrouver les données secrètes. On cherche aussi à désynchroniser les courbes entre elles pour rendre les traitements statistiques de type DPA, CPA, ... moins ou peu efficaces. Bien entendu, de telles contre-mesures (générateur de bruit, scrambleur, ...) ne peuvent pas être considérées comme suffisantes. La vulnérabilité reste résiduelle, elle est présente mais plus difficile à exploiter. Dans cette catégorie, on peut citer également les conceptions de circuit visant à équilibrer la consommation d'un circuit quelles que soient les données manipulées, comme la technique dite du double rail (*dual rail*).

La troisième et dernière catégorie de contre-mesures a pour objectif de décorréler les courbes de consommation de l'exécution de l'algorithme d'une part, d'avec les valeurs des données manipulées (valeurs des calculs intermédiaires) d'autre part. Dans ce cas, il n'est plus possible pour l'attaquant de faire des prédictions (en fonction des hypothèses qu'il fait sur des bits de clé) sur ces valeurs de calculs intermédiaires. On parle alors de masquage et de randomisation des données et des clés.

Remarque : parfois, le fait de s'appuyer sur des algorithmes secrets est perçu comme une contre-mesure. En effet, il est plus difficile d'attaquer un algorithme de structure inconnue qu'un DES ou un AES pour lesquels il est plus envisageable de prédire des valeurs intermédiaires ou d'observer le signal connaissant les opérations réalisées. Cette contre-mesure rentre dans la première catégorie citée ci-dessus. L'utiliser peut rendre la tâche plus ardue pour un attaquant ; cependant, comme pour le fameux principe de Kerckhoffs, il est déconseillé de faire reposer la sécurité sur cet obscurantisme. On peut toujours craindre qu'un jour ou l'autre le secret sera divulgué, par exemple par rétro-conception. Le cas du produit Mifare de NXP et de son algorithme Crypto1 qui a été retrouvé en rétro-conception puis cryptanalysé et cassé par des universitaires néerlandais illustre bien un tel risque.

Dans la suite de cet article, nous allons nous concentrer sur la troisième catégorie de contre-mesures pour présenter des protections qui ne dépendent ni du type d'application visée ni du type de contre-mesures présentes sur le signal... et qui sont donc indépendantes du microprocesseur utilisé.

2.3.2 Contre-mesures et propriété intellectuelle

Parmi les contre-mesures que nous présentons, nombre d'entre elles sont brevetées. En effet, la sécurité est un marché industriel. Savoir protéger ses produits contre les attaques est un savoir et une expertise de valeur.

Les brevets les plus connus sont ceux de Paul C. Kocher et sa société Cryptography Research basée à San Francisco. Avant de publier les attaques SPA et DPA, Paul C. Kocher a astucieusement breveté de nombreuses contre-mesures très efficaces contre ces attaques. Certaines de ses contre-mesures étant incontournables, cela lui a permis de devenir aujourd'hui un acteur majeur du domaine et assuré des revenus probablement substantiels.

Les autres acteurs industriels du domaine ont également d'importants portefeuilles de brevets sur le sujet qui continuent de s'étoffer, chaque nouvelle attaque générant l'invention de nouvelles contre-mesures et de nouveaux brevets.

3 Algorithmes cryptographiques symétriques

Les algorithmes symétriques les plus fréquemment rencontrés dans les produits embarqués sont des algorithmes standards. Si dans certains secteurs d'activités, l'obscurité rime avec sécurité, ce n'est pas le cas dans les cartes à puce... à quelques exceptions près. On retrouve surtout les algorithmes TDES et AES dans la plupart des standards applicatifs, voire pour certains produits, et dans certains pays, on peut être amené à devoir implémenter et sécuriser d'autres standards tels FEAL, RC5, SEED, PRESENT, etc.

3.1 Implémentation

On trouve les implémentations d'algorithmes cryptographiques sous deux formes. La plus fréquente est celle de l'accélérateur de calculs, on parle alors de DES (resp. AES) *hardware* ou de DES *engine* (resp. AES engine). Dans ce cas, un circuit dédié au calcul DES (resp. AES) est intégré au microprocesseur, ce qui permet de réaliser le calcul en seulement quelques cycles d'horloge. La seconde technique est une implémentation purement logicielle, en assembleur, qui s'appuie sur les opérations (logiques et arithmétiques) fournies par le jeu d'instructions du cœur (CPU).

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

3.1.1 Accélérateur de calcul pour algorithmes symétriques

Dans ce cas, tout le calcul cryptographique est réalisé par l'accélérateur. Il suffit de charger la clé et le message dans des registres spécifiques et de lancer le calcul en activant les bits dédiés dans un registre de configuration. Le calcul est alors très rapide : les accès mémoire étant limités ou nuls, les calculs pouvant être parallélisés. On peut ainsi exécuter les 8 substitutions dans les boîtes S du DES en parallèle (idem dans le cas de l'AES pour les 16 substitutions).

Pour l'exemple : on peut obtenir des performances de l'ordre d'un cycle par ronde et donc 16 cycles pour le DES et 48 cycles pour le triple DES. Suivant les choix des fabricants, cela peut bien entendu varier d'un circuit à un autre.

3.1.2 Implémentation purement logicielle

Elle est dans ce cas réalisée en assembleur afin de minimiser les opérations de lecture et d'écriture en mémoire en manipulant au maximum les registres du cœur. Si nous prenons l'exemple d'un processeur ARM (par exemple l'ARM7TDMI), il s'agit d'un cœur RISC (*Reduced Instruction Set Controller*) ayant donc à sa disposition plusieurs registres 32 bits. On développe alors en assembleur un code utilisant au mieux les registres R0 à R15 au détriment des variables en RAM. Les opérations entre registres se font pour la plupart en 1 cycle. Il est également important de tenir compte des étapes du pipeline du cœur afin d'optimiser la séquence d'opération `fetch/decode/execute`. Pour un cœur de type CISC (*Complex Instruction Set Controller*) comme l' INTEL 80C51, il y a uniquement deux registres de 8 bits mais différents types de RAM : la RAM interne (à accès rapide) notée iRAM et la RAM classique. Dans ce cas, nous écrivons un code utilisant au mieux les registres A et B du cœur et la iRAM disponible plutôt que la RAM classique.

De manière générale, il est important d'utiliser peu de RAM et donc d'économiser les variables. En effet, il y a peu de RAM dans une carte à puce (même si nous sommes loin des quelques octets présents sur les premiers microprocesseurs de carte à puce) et celle-ci est majoritairement utilisée par le système d'exploitation. Dans un calcul DES/TDES et AES, l'implémentation ne stocke pas en mémoire toutes les sous-clés des différentes rondes. Elle dérive chaque sous-clé à la volée, c'est-à-dire à chaque ronde, ce qui permettra de ne stocker qu'une seule sous-clé en mémoire.

Une implémentation DES de qualité coûte ainsi entre 1 et 3 kilo-octets de code et nécessite entre 1000 et 10000 cycles pour réaliser le calcul complet. Ces chiffres dépendent de l'architecture du cœur utilisé 8, 16 ou 32 bits et de son efficacité.

3.1.3 Comparaison

L'accélérateur offre des calculs symétriques efficaces car environ 100 à 1000 fois plus rapides que la version logicielle. Le calcul est très rapide et donc plus difficile à attaquer qu'une implémentation logicielle. Cette dernière offre beaucoup plus de cycles à l'attaquant pour réaliser son analyse.

Un accélérateur va cependant grossir la taille du microprocesseur et engendrer une consommation de courant supérieure lors de son fonctionnement. Ces deux « inconvénients » sont largement acceptés aujourd'hui en regard du considérable gain de temps de calcul apporté. Le point le plus délicat reste alors la sécurité de cet accélérateur, soit sa résistance aux attaques. Si une attaque met en défaut l'accélérateur symétrique, le corriger a un coup élevé. Il faut dans ce cas modifier le circuit et envoyer la nouvelle architecture du circuit en fonderie pour la réalisation du nouveau silicium. C'est beaucoup plus coûteux que de corriger une implémentation logicielle, c'est-à-dire corriger le code et le recharger dans le même circuit (*patch*).

3.2 Attaques passives et contre-mesures

Toute implémentation symétrique est menacée par les attaques passives usuelles comme celles énoncées dans l'article **[DI2]**.

3.2.1 Analyse de puissance simple SPA

Considérons que la carte calcule un chiffré C par chiffrement DES d'un message M de 8 octets avec une clé K de 64 bits. Le calcul réalisé est donc : C = DES(M,K).

La première attaque SPA connue sur le DES consistait à analyser une seule courbe de consommation de courant pendant l'exécution de ce calcul. Dans cette courbe, l'attaquant cible en particulier l'opération de dérivation de clé qui s'exécute à chaque ronde afin de retrouver des bits de la clé secrète K.

Considérons une implémentation logicielle du DES (ou TDES) sur une architecture 8 bits. Dans ce cas, le calcul de la sous-clé de ronde comprend des opérations de rotations circulaires et de permutations bit à bit sur deux blocs de 28 bits de clé.

Notons par $K' = (k_{55}\ k_{54} \ldots k_{28}\ k_{27}\ k_{26} \ldots k_{01}\ k_{00})$ la décomposition binaire de la clé K une fois les bits de parité retirés. Lors du calcul de la première sous-clé, ces bits sont partagés en deux blocs de 28 bits B_1 et B_0.

$$B_1 = (k_{55}\ k_{54} \ldots k_{29}\ k_{28})_2$$
$$B_0 = (k_{27}\ k_{26} \ldots k_{01}\ k_{00})_2$$

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Ces blocs vont subir une rotation circulaire à gauche de 1 ou 2 bits suivant le numéro de la ronde exécutée.

La figure ci-dessous détaille le code exécuté pour une rotation à gauche du bloc B_0.

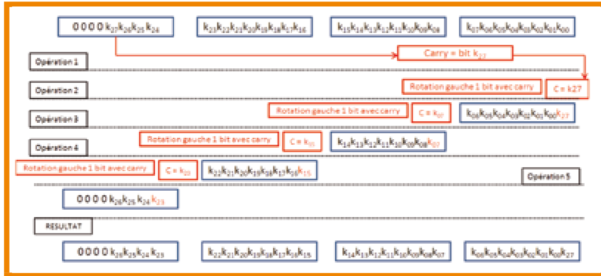


Illustration 1 : Analyse SPA de la dérivation de clé en sous-clé de ronde

Origine de la fuite : sur ce composant, il y a une fuite d'information sur la valeur de la retenue (*carry*). Il est possible de voir une variation de consommation suivant que la retenue vaut 1 ou 0.

Puisque la première opération consiste à mettre dans le bit de carry (retenue) le 28ème bit de clé soit k_{27}, il est possible de retrouver sa valeur sur la courbe de consommation en localisant le moment (le cycle) ou la carry est mise à jour avec le bit k_{27}. Puis la seconde opération met k_{07} dans le bit de retenue qui peut donc lui aussi être retrouvé. Et il en est de même pour le bit k_{15} lors de la troisième opération puis k_{23} lors de la rotation suivante. On vient donc de retrouver les bits k_{07}, k_{15}, k_{23} et k_{27} sur la rotation de B_0. De la même manière, on retrouve les bits k_{35}, k_{43}, k_{51} et k_{55} sur la rotation de B_1.

En analysant ces opérations à chacune des 16 rondes du DES, un attaquant retrouve les 56 bits de la clé K.

3.2.2 Contre-mesure

Cette attaque n'est généralement réalisable que sur une implémentation logicielle et sur une architecture 8 bits ayant une fuite d'information sur la manipulation du bit de retenue. Autrefois réalisable, cette attaque est plus difficile à pratiquer aujourd'hui sur des composants 16 et 32 bits récents. De telles manipulations sont rarement et difficilement distinguables aujourd'hui. Mais cette technique demeure toujours un chemin d'attaque potentiel et il est bon de le garder en mémoire pour ne pas reproduire des erreurs passées.

Cette analyse illustre bien l'efficacité diabolique d'une bonne lecture SPA : être capable de distinguer un type de branchement conditionnel d'un autre, un branchement court (saut à une adresse proche) d'un branchement à une adresse longue (plus de 256 octets par exemple), etc. La SPA s'applique à bien d'autres implémentations et manipulations dans des codes cryptographiques embarqués. Il est donc important qu'un développeur accorde une grande attention à son code pour éviter qu'il ait un comportement différent facilement distinguable en fonction de valeur(s) de bits secrets.

Dans l'exemple précédent, une implémentation sécurisée de cette rotation pourrait consister à remplacer la rotation à gauche par une permutation bit à bit sur chacun des blocs de 28 bits B_0 et B_1 à l'aide d'une table de permutation. De plus, chacune des 56 permutations bit à bit est réalisée dans un ordre aléatoire différent à chaque exécution du DES. Cet ordre aléatoire est déterminé en fonction d'un nombre obtenu à l'aide du générateur d'aléas de la carte (microprocesseur).

Nous venons d'introduire ici un principe fondamental dans la sécurisation des implémentations cryptographiques : l'utilisation de nombres aléatoires connus de la carte uniquement, c'est-à-dire dont les valeurs ne seront jamais communiquées hors du microprocesseur de la carte.

3.2.3 Analyse de puissance différentielle DPA

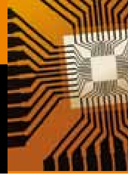
Un attaquant va dans un premier temps obtenir toutes les informations possibles en analyse SPA. Dans le meilleur des cas (d'un point de vue de l'attaquant), il retrouve, comme dans l'exemple précédent, des bits de clé voire le secret complet. Dans le cas contraire, cette analyse lui permet de préparer des attaques plus complexes telles les attaques DPA (voire CPA) présentées précédemment dans l'article **[DI2]** de ce magazine. En effet, on peut en SPA reconnaître l'algorithme implémenté et repérer sur la courbe les différents calculs de l'algorithme que l'on veut exploiter en DPA. Il est possible de distinguer si certaines opérations sont mieux protégées que d'autres et donc de cibler une zone plus faible dans l'algorithme. Un attaquant tente alors de retrouver par DPA la sous-clé utilisée dans ces zones.

Comme mentionné dans l'article **[DI2]**, afin de réaliser son attaque DPA, l'attaquant fait une hypothèse sur une partie des bits de la sous-clé lors de la première (ou dernière) ronde du DES, par exemple les bits (k_5 ... k_0) de la première sous-clé. Ensuite, pour chaque message M (qu'il connaît voire qu'il choisit), il peut calculer la valeur intermédiaire de calcul avant et/ou après la substitution par la première boîte S. Puis il corrèle cette valeur I avec les courbes de consommation qu'il a obtenues des exécutions de la carte (cf. **[DI2]**). La supposition qui conduit à un pic de DPA (le pic le plus significatif) sur les courbes traitées correspond à la valeur des bits de clé recherchée.

3.2.4 Contre-mesure

Pour réaliser la DPA, il faut prédire une valeur intermédiaire (qui dépend des bits de clé) manipulée par l'algorithme à un instant donné du calcul. Pour

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



empêcher l'attaquant de réaliser son attaque, il faut donc réaliser une implémentation rendant impossible cette prédiction. La technique la plus efficace est celle du masquage des données.

Pour l'expliquer, prenons un exemple reposant sur un algorithme simple noté A1.

<u>Algorithme A1:</u>

Entrée : message M de 64 bits, clef K de 64 bits.

Sortie : chiffré C

1. $C_0 \leftarrow P_1(M)$
2. $C_1 \leftarrow C_0 \oplus K$
3. $C_2 \leftarrow \text{RotateLeft}(C_1)$
4. Retourner(C_2)

Cet algorithme applique au message une permutation P_1, puis additionne (en binaire soit un XOR) la clé K au résultat de la permutation, et enfin effectue une rotation circulaire à gauche et retourne le chiffré.

Un attaquant connaissant M peut prédire, pour une hypothèse sur des bits de K, les valeurs des bits correspondants dans les valeurs C_0 et C_1. Ainsi, il est capable de réaliser une DPA et retrouver la clé K.

Nous implémentons ensuite le même algorithme mais résistant à la DPA que nous notons A1.SEC. Utilisons un masquage de données pour le protéger.

<u>Algorithme A1.SEC:</u>

Entrée : message M de 64 bits, clef K de 64 bits.

Sortie : chiffré C

1. $R \leftarrow$ aléa de 64 bits généré par la carte
2. $C_0 \leftarrow M \oplus R$ – masquage du message –
3. $R_0 \leftarrow P_1(R)$
4. $C_1 \leftarrow P_1(C_0)$
5. $C_2 \leftarrow C_1 \oplus K$
6. $C_3 \leftarrow \text{RotateLeft}(C_2)$
7. $R_1 \leftarrow \text{RotateLeft}(R_0)$
8. $C \leftarrow C_3 \oplus R_1$
9. Retourner(C)

L'algorithme A1.SEC renvoie en sortie la même valeur que l'algorithme A1.

Preuve :

$\text{RotateLeft}[\ P_1(M \oplus R)) \oplus K] \oplus \text{RotateLeft}[\ P_1(R)]$

$= \text{RotateLeft}[\ P1(M) \oplus P1(R) \oplus K] \oplus \text{RotateLeft}[\ P_1(R)]$

$= \text{RotateLeft}[\ P1(M) \oplus K] \oplus \text{RotateLeft}[\ P_1(R)] \oplus \text{RotateLeft}[\ P_1(R)]$

$= \text{RotateLeft}[\ P_1(M) \oplus K]$

$= C$

Contrairement à l'implémentation A1 qui est non protégée, l'algorithme A1.SEC n'est pas attaquable en DPA. En effet, à partir du moment où nous masquons le message M avec le masque aléatoire R, un attaquant ne peut plus prédire de valeur intermédiaire. Nous avons ainsi ajouté une seconde inconnue (secrète) R. Ce masque R est tiré aléatoirement à chaque exécution de l'algorithme A1.SEC. Une attaque DPA ne peut plus fonctionner si les valeurs de R sont correctement distribuées, c'est-à-dire que le générateur aléatoire utilisé ne doit pas être biaisé.

L'exemple précédent avec A1 est simple à protéger car les opérations réalisées sont linéaires. Cependant, dans les algorithmes symétriques de type DES et AES, toutes les opérations ne sont pas linéaires, telles les boîtes S de substitution du DES ou celles de l'AES. Cela rend les opérations de masquage difficiles à réaliser. En effet, $\text{Substitution}(M \oplus R \oplus K) \neq \text{Substitution}(M \oplus K) \oplus \text{Substitution}(R)$. Il faut dans ce cas trouver des solutions plus complexes pour masquer ces opérations. Plusieurs techniques de masquage existent, qui permettent de se protéger de la DPA. Nous ne rentrerons pas dans les détails de ces techniques ici.

Il existe d'autres attaques plus puissantes que la DPA : la CPA, la MIA, la PCA, etc. Le masquage protégera efficacement l'implémentation contre ces différentes techniques.

3.3 Attaques actives et contre-mesures

3.3.1 Analyse différentielle de faute(s) DFA

La première attaque DFA sur un algorithme symétrique est publiée par E. Biham et A. Shamir en 1997. Les auteurs présentent comment retrouver la clé secrète d'un DES en utilisant entre 50 et 200 chiffrés erronés. Expliquons le principe de l'attaque dans ce cas précis.

En injectant une faute à un instant « choisi » pendant le calcul (chiffrement) DES d'un message M, l'attaquant obtient un chiffré erroné $C^{\downarrow}$ retourné par le microcircuit à la place du chiffré correct attendu C. En répétant cette opération, il obtient un ensemble de k chiffrés erronés $C_1^{\downarrow}$ ... $C_k^{\downarrow}$ et leurs chiffrés corrects C_1 ... C_k. Par analyse différentielle, il est alors possible de retrouver des bits de la clé voire la clé complète.

L'exemple donné dans le papier **[BCNTW04]** illustre bien la technique DFA. Considérons de même dans la figure qui suit que la dernière ronde du DES a été simplifiée.

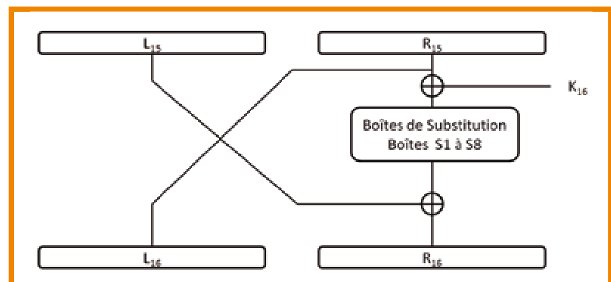
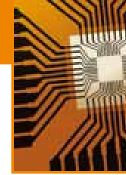


Illustration 2 : Dernière ronde du DES simplifiée

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



La sortie correcte de la dernière (seizième) ronde correspond à :

$$R16 = S(R15 \oplus K16) \oplus L15 = S(L16 \oplus K16) \oplus L15$$

Si la faute est injectée pendant l'exécution de la 15ème ronde, elle engendre une erreur e_{IN} sur R_{15} et donc une valeur incorrecte $R_{15}^{\downarrow}$ en entrée de dernière ronde. On obtient alors un chiffré erroné à la fin du calcul :

$$R_{16}^{\downarrow} = S(R_{15}^{\downarrow} \oplus K_{16}) \oplus L_{15} = S(L_{16}^{\downarrow} \oplus K_{16}) \oplus L_{15}$$

Comme nous connaissons R_{16} (à partir de C) et $R_{16}^{\downarrow}$ (à partir de $C^{\downarrow}$ retourné), on peut calculer l'erreur engendrée sur le résultat :

$$e_{OUT} = R_{16} \oplus R_{16}^{\downarrow} = S(L_{16} \oplus K_{16}) \oplus L_{15} \oplus S(L_{16}\downarrow \oplus K16) \oplus L15$$

et on obtient ainsi la relation à une inconnue K16 :

$$R_{16} \oplus R_{16}^{\downarrow} = S(L_{16} \oplus K_{16}) \oplus S(L_{16}^{\downarrow} \oplus K_{16})$$

En répétant plusieurs fois l'attaque, on obtient ainsi un ensemble de plusieurs relations de la sorte avec plusieurs chiffrés et leurs chiffrés erronés qui vont nous amener à une seule valeur possible pour K_{16}.

Par exemple, si les fautes n'affectent que l'entrée d'une seule boîte S_i - disons la première S_1 - alors la formule précédente nous permettra de retrouver de manière exhaustive les 6 premiers bits de la dernière sous-clé de la ronde en testant les 64 valeurs possibles pour ces 6 bits de K_{16}. Seule la vraie valeur vérifiera toutes les relations obtenues.

En itérant cette opération pour les huit boîtes S_1 à S_8, il est possible de retrouver les 48 bits de K_{16}. Il ne reste plus qu'à rechercher de manière exhaustive les 8 bits manquants.

Il est également possible de réaliser cette attaque en injectant la faute dans une ronde précédente, mais plus la faute est réalisée en amont de la dernière ronde, plus le nombre de chiffrés erronés nécessaires devient (trop) important.

L'efficacité de l'attaque, mesurée par le nombre de chiffrés erronés nécessaires, dépend de l'effet de la faute sur le circuit. Par exemple, s'il est possible de fauter simultanément plusieurs bits en entrée de la substitution (des boîtes S), alors on peut dans le meilleur des cas retrouver la clé à partir de deux chiffrés erronés seulement. Cette attaque a ainsi été réalisée par les auteurs de l'article **[GT04]**.

Ainsi, dans le cas des attaques de type DFA, leur efficacité dépend de l'effet de la faute.

Cette technique dite d'analyse différentielle de faute est applicable avec succès à d'autres algorithmes symétriques, tels l'AES.

3.3.2 Contre-mesure

La contre-mesure la plus simple consiste à vérifier le calcul réalisé, en l'effectuant deux fois.

```
Algorithme protégé DFA:
Entrée: message M de 64 bits, clef K de 64 bits.
Sortie: chiffré C
    1. C ← DES(M, K)
    2. D ← DES (M,K)
    3. Si C ≠ D retourner (attaque détectée)
    4. Si C = D retourner(C)
```

L'inconvénient principal de cette contre-mesure est le doublement du temps de calcul.

La question qui vient à l'esprit est alors : « Pourquoi ne pas faire deux fautes en injectant deux fois ? ». En effet, cela est réalisable même si difficile car il faut dans les deux cas que la faute produise exactement le même effet afin que les deux exécutions retournent le même résultat.

Une autre solution, plus résistante à une double attaque par faute, consiste à calculer C = DES(M,K) puis faire le calcul inverse D = DES-1(C,K) et vérifier que le déchiffré D est bien égal au message d'entrée M.

3.3.3 Réduction du nombre de rondes

Une attaque simple et très efficace quand elle réussit, consiste à réduire le nombre de rondes exécutées par un AES, ou un DES. Cette technique est décrite dans l'article **[CT05]** avec des résultats pratiques.

Prenons le code DES suivant :

```
Algorithme DES:
Entrée: message M de 64 bits, clef K de 64 bits.
Sortie: chiffré C
C ← Permutation_Initiale(M)
K0 ← K
Compteur_de_ronde ← 0 ;
Tant que (Compteur_de_ronde < 16)
        Ki ← sous clef dérivée de la clef Ki-1
        C ← RondeDES(C,Ki)
        Compteur_de_ronde++
C ← Permutation_Finale(C)
Retourner(C)
```

Un attaquant injecte ici une faute sur la valeur du compteur `Compteur_de_ronde` afin de sortir de la boucle alors que les 16 rondes n'ont pas été exécutées. Ainsi, une seule ronde pourrait être exécutée. Suivant le nombre de rondes exécutées (par exemple une ou trois), à partir d'un ou plusieurs chiffrés corrompus, il est possible de retrouver la clé K. Il est possible d'appliquer une cryptanalyse sur ce DES au nombre de tours réduit, pour une complexité bien moindre que sur un plus grand nombre de rondes.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

3.3.4 Contre-mesure

La contre-mesure précédente (DFA) s'applique ici. Mais une technique plus simple et plus rapide consiste à s'assurer que le nombre de rondes exécutées est bien égal à 16 (ou 48 dans un triple DES).

```
Algorithme DES "protégé" contre réduction de rondes
C ← Permutation_Initiale(M)
K0 ← K
Compteur_de_ronde ← 0
Compteur_de_ronde_bis ← 16
Tant que (Compteur_de_ronde < 16)
        Ki ← sous clef dérivée de la clef Ki-1
        C ← RondeDES(C,Ki)
        Compteur_de_ronde++
        Compteur_de_ronde_bis--
C ← Permutation_Finale(C)
Si (Compteur_de_ronde ≠ 16) retourner (attaque détectée)
Si (Compteur_de_ronde_bis ≠ 0) retourner (attaque détectée)
Sinon Retourner(C)
```

4 Algorithmes cryptographiques asymétriques

4.1 Implémentation

Les algorithmes asymétriques les plus fréquemment utilisés dans les produits de sécurité de type cartes à puce sont employés à des fins de signature. Ce sont le RSA (et sa version utilisant le théorème des restes chinois, le RSA CRT), le DSA (*Digital Standard Algorithm*) ainsi que son équivalent à base de courbes elliptiques, l'ECDSA pour les opérations de signatures. Les protocoles d'échange de clés de Diffie-Hellman (DH) et l'*Elliptic Curve DH* (ECDH) sont présents également et de plus en plus utilisés.

4.1.1 Différents types d'arithmétiques modulaires et de coprocesseurs arithmétiques

Il n'est pas possible, même avec un cœur 32 bits, d'implémenter un de ces algorithmes sans la présence d'un accélérateur de calcul pour les opérations sur les grands entiers. En effet, réaliser une exponentiation RSA 2048 nécessite en moyenne 3072 opérations de multiplication modulaires de type a·b modulo n où a, b et n sont des nombres de 2048 bits. Pour remédier à tout souci de performance, les fabricants de microcircuits modernes ajoutent un coprocesseur arithmétique conçu pour accélérer l'opération de multiplication modulaire sur de tels grands entiers. Suivant le fabricant, le coprocesseur et le choix de l'arithmétique modulaire varient. Par exemple, certains accélérateurs sont basés sur la réduction de Barrett, d'autres sur la multiplication modulaire de Montgomery, et d'autres variantes existent également.

4.1.2 Exponentiation RSA

Soit n un module RSA (public) de j bits, par exemple prenons j égal à 2048. Le module n est égal au produit de deux grands nombres premiers secrets p et q de tailles égales, soit 1024 bits. Notons e l'exposant public choisi généralement petit pour accélérer le calcul public. Par exemple, prenons e égal à $2^{16}+1$. Enfin, notons d l'exposant secret défini par : $d = e^{-1}$ modulo (p-1)·(q-1).

Nous avons de la sorte défini la clé publique (e, n) ainsi que la clé secrète d.

La signature S d'un message m (choisi inférieur à n) est : $S = m^d$ modulo n. La vérification de la signature consiste à s'assurer que Se modulo n est égal à m.

Nous utiliserons pour la suite la notation suivante pour la décomposition d'un nombre de j bits en binaire :

$$d = (d_{j-1}\, d_{j-2} \ldots d_1\, d_0)_2.$$

Algorithme d'exponentiation standard

Entrée: message *m*, exposant *d*, module *n*

Sortie: m^d modulo *n*

1. $r \leftarrow 1$
2. Pour i de (*j*-1) à 0 faire
   a. $r \leftarrow r^2$ modulo *n*
   b. Si $d_i = 1$ alors $r \leftarrow r \cdot m$ modulo *n*
3. Retourner (*r*)

Comme d est un grand nombre (ex. 2048 bits), l'opération de signature est très coûteuse malgré la présence d'un coprocesseur. Mais il est possible de l'accélérer théoriquement d'un facteur quatre en utilisant le théorème des restes chinois, on parle alors de RSA CRT (*Chinese Remainder Theorem*).

Pour ce faire, on calcule l'exponentiation modulo p et modulo q, puis on utilise le théorème des restes chinois pour revenir au résultat modulo p·q soit n.

On définit les paramètres secrets suivants qui nécessitent d'être stockés en mémoire dans le microcontrôleur :

d_p = d modulo (p-1)

d_q = d modulo (q-1)

$i_q = q^{-1}$ modulo p

On calcule alors la signature S de la manière suivante :

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

S_p = (m modulo p)^{dp} modulo p (= S modulo p)
S_q = (m modulo q)^{dq} modulo q (= S modulo q)
S = ((S_p – S_q)·i_q modulo p)·q + S_q

L'exponentiation modulo n est ainsi remplacée par deux exponentiations de taille moitié. La complexité de l'exponentiation étant d'ordre cubique en la taille de n, une exponentiation manipulant des éléments deux fois plus petits sera donc 2^3=8 fois plus rapide. Et comme nous effectuons deux exponentiations au lieu d'une, l'exponentiation CRT est finalement quatre fois plus rapide qu'une exponentiation classique. Les opérations de recombinaison sont théoriquement d'une complexité négligeable en regard des exponentiations.

4.2 Attaques passives et contre-mesures

4.2.1 Analyse SPA

Comme pour les algorithmes symétriques, les techniques d'attaques SPA, DPA, CPA s'appliquent efficacement sur les implémentations asymétriques afin de retrouver une partie ou la totalité des bits de la clé secrète. Nous ne pouvons détailler dans cet article toutes les attaques connues sur les algorithmes asymétriques, nous allons nous concentrer sur le RSA et sa version CRT.

Considérons par exemple l'analyse SPA de la courbe de consommation de courant d'une exécution d'exponentiation RSA implémentée classiquement. Il est possible, tel qu'expliqué dans **[D12]**, de distinguer les opérations d'élévation au carré des opérations de multiplication et ainsi de retrouver sur cette seule courbe tous les bits de l'exposant secret.

4.2.2 Contre-mesure

La contre-mesure suivante, nommée « atomicité », est une implémentation sécurisée efficace contre la SPA.

Algorithme atomique d'exponentiation

Entrée: message m, exposant d, module n

Sortie: m^d modulo n

1. $r_0 \leftarrow 1$
2. $r_1 \leftarrow m$
3. $k \leftarrow 0$, $i \leftarrow j$-1
4. Tant que ($i \geq 0$) faire
   a. $r_0 \leftarrow r_0 \cdot r_k$ modulo n
   b. $k \leftarrow k \oplus d_i$
   c. $i \leftarrow i - \neg k$
5. Retourner (r_0)

On observe dans cet algorithme que les opérations d'élévation au carré r_0^2 sont désormais effectuées par un algorithme de multiplication. La trace en courant de l'opération est donc similaire à celle d'une multiplication. De plus, les manipulations sur les bits d_i de l'exposant se font avec un code identique quelle que soit la valeur de d_i. Il n'y a donc pas de branchement conditionnel qui pourrait être également une source de fuite d'information.

Deux autres algorithmes d'exponentiation peuvent être utilisés : l'algorithme de « Carré et Multiplication Toujours » et « L'échelle de Montgomery (Montgomery Ladder) ».

Algorithme carré et multiplication toujours

Entrée: message m, exposant d, module n

Sortie: m^d modulo n

1. $r_0 \leftarrow 1$
2. $r_1 \leftarrow m$
3. Pour i de j-1 à 0 faire
   a. $r_0 \leftarrow r_0 \cdot r_0$ modulo n
   b. $r_{1-d_i} \leftarrow r_0 m$ modulo n
4. Retourner (r_0)

Algorithme "Echelle de Montgomery"

Entrée: message m, exposant d, module n

Sortie: m^d modulo n

1. $r_0 \leftarrow 1$
2. $r_1 \leftarrow m$
3. Pour i de j-1 à 0 faire
   a. $r_{1-d_i} \leftarrow r_0 \cdot r_1$ modulo n
   b. $r_{d_i} \leftarrow r_{d_i}^2$ modulo n
4. Retourner (r_0)

Ces deux algorithmes ont pour inconvénient de requérir j opérations de carré et j opérations de multiplication quand l'algorithme atomique ne nécessite que 3j/2 multiplications au total en moyenne. Si l'on considère que la complexité du carré modulaire est de l'ordre de 0.8 fois celle de la multiplication (soit donc S = 0.8M), on obtient les complexités suivantes pour ces trois algorithmes :

- Version atomique = 1.5·j·M
- Carré et Multiplication Toujours = 1.8·j·M
- Echelle de Montgomery = 1.8·j·M

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Un autre algorithme atomique utilisant uniquement des opérations de carré a été proposé récemment dans **[CFGRV11]** et offre des résultats équivalents à la version d'exponentiation atomique présentée ci-dessus.

4.2.3 Analyse DPA

Choisir un de ces algorithmes pour implémenter l'opération d'exponentiation protège ainsi de la SPA, mais pas de la DPA. Il est donc nécessaire d'ajouter d'autres protections.

Randomisation du message

- Randomisation additive :

- remplacer m par $m^* = m + r_1 \cdot n$	avec r_1 un nombre aléatoire de u bits (u = 32 bits ou plus).
- remplacer n par $n^* = r_2 \cdot n$	avec r_2 un nombre aléatoire de u bits (ou fixé à 2^u -1)

- calculer $S^* = m^{*d}$ modulo n^*
- calculer $S = S^*$ modulo n

Le résultat S calculé est bien égal à la valeur attendue une fois S^* réduit modulo n. Les valeurs r_1 et r_2 sont générées aléatoirement à chaque nouvelle opération d'exponentiation.

- Randomisation multiplicative :

- calculer $S^* = (r^e \cdot m)^d$ modulo n	avec r un nombre aléatoire de u bits
- calculer $U = r^{-1}$ modulo n	
- calculer $S = S^* \cdot U$ modulo n	

Randomisation de l'exposant

- Par addition d'un multiple de l'ordre du groupe :

- $d^* = d + r \cdot \varphi(n) = d + r \cdot (p-1) \cdot (q-1)$	φ étant la fonction indicatrice d'Euler
- calculer $S^* = m^{d^*}$ modulo n	(ou $S = S^* = ((m^*)^{d^*}$ modulo $n^*)$ modulo n)

- Additive :
  - $d^* = d - r$
  - calculer $S^* = m^{d^*}$ modulo n
  - calculer $U = m^r$ modulo n
  - calculer $S = S^* \cdot U$ modulo n

Ces contre-mesures de randomisation du message et de l'exposant offrent (si correctement implémentées) une protection efficace contre les attaques DPA, CPA, etc., sur l'exponentiation modulaire RSA et RSA CRT.

Il existe d'autres contre-mesures (ainsi que d'autres attaques) sur le RSA et le RSA CRT ainsi que sur les autres algorithmes à clé publique. Nous ne pouvons tout détailler dans cet article et ce n'est pas notre but. Les principes d'attaques et de protection sont fondamentalement identiques, seule la mise en œuvre varie car propre à chaque algorithme.

4.3 Attaques actives et contre-mesures

4.3.1 DFA sur RSA CRT

L'attaque la plus connue est la DFA sur le calcul du RSA en mode CRT. Elle est très efficace et d'une grande simplicité.

Dans le calcul RSA CRT expliqué précédemment, nous pouvons écrire S comme la combinaison linéaire :

$S = a \cdot S_p + b \cdot S_q$ pour deux entiers a et b tels que :

- a modulo p = 1	- b modulo p = 0
- a modulo q = 0	- b modulo q = 1

L'attaque consiste à exécuter la signature RSA CRT du message m deux fois de suite mais en fautant une des deux exponentiations lors du second calcul. Le premier calcul non « fauté » correspond donc à : $S = a \cdot S_p + b \cdot S_q$. Lors du second calcul, nous avons fauté S_q en $S_q^{\downarrow}$ qui est le fauté erroné.

$S^{\downarrow} = a \cdot S_p + b \cdot S_q^{\downarrow}$

soit $u = S - S^{\downarrow} = b \cdot (S_q - S_q^{\downarrow})$

or b est nul modulo p, donc b est un multiple de p.

On peut donc écrire u sous la forme : $u = v \cdot p \cdot (S_q - S_q^{\downarrow})$

si q ne divise pas $(S_q - S_q^{\downarrow})$

comme $n = p \cdot q$

alors $pgcd(S - S^{\downarrow}, n) = pgcd(u, n) = p$.

On retrouve ainsi le facteur premier secret p de n, puis en divisant n par p, on retrouve évidemment q.

Ainsi, en deux exécutions, dont une « fautée », du RSA CRT pour un message m, on vient de retrouver directement les composantes secrètes de la clé RSA.

Variante de Lenstra

Une variante a été proposée par A. Lenstra, qui ne nécessite que l'exécution fautée et donc la valeur de $S^{\downarrow}$. L'attaquant injecte une faute de la même manière que précédemment et calcule :

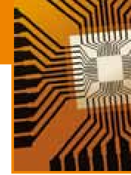
$p = pgcd(m - (S^{\downarrow})^e, n)$

Ici, il n'est pas nécessaire d'exécuter deux fois le calcul pour retrouver le premier secret p. L'attaque est identique pour retrouver q, il suffit de fauter la première exponentiation S_p en $S_p^{\downarrow}$.

4.3.2 Contre-mesure

De même que dans les algorithmes symétriques, pour se protéger de cette attaque, on rejoue le calcul une seconde fois et on vérifie que les deux valeurs obtenues pour S sont identiques. Cependant, vu le temps de calcul nécessaire au RSA CRT (pour 2048 bits entre

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



50 et 200 ms suivant le fabricant), rejouer ce calcul est très pénalisant. Or nous avons vu précédemment que l'exposant public e est généralement choisi petit, de 17 bits dans le cas où e = $2^{16}+1$.

Si la valeur de e est connue du microcontrôleur, on peut calculer l'opération inverse et vérifier si cette opération inverse conduit bien au message m initial. Cette technique est très efficace et largement utilisée aujourd'hui.

Algorithme RSA CRT protégé contre la DFA:
Entrée: message *M* de *j* bits
Clef: p,q,d_p,d_q,i_q,e de *j*/2 bits.
Sortie: signature *S*

1. $S \leftarrow$ RSA CRT (m)
2. $D \leftarrow m^e$ modulo ($p \cdot q$)
3. Si $m \neq D$ retourner (attaque détectée)
4. Si $m = D$ retourner(S)

Il existe également des attaques par faute DFA sur l'exponentiation RSA classique, mais la réalisation est plus compliquée en pratique. Ce calcul peut être protégé de la même manière par une vérification avec l'exposant public e.

4.3.3 Safe error sur l'exponentiation « square and multiply always »

Dans le cas où l'algorithme choisi pour l'exponentiation est le « Carré et Multiplication Toujours », l'implémentation est alors vulnérable aux attaques par faute dites « Safe Error ». En effet, une faute injectée dans une multiplication $r_1 \leftarrow r_0 \cdot m$ ne modifiera pas le résultat qui sera retourné par la carte. Cela donne ainsi comme information à l'attaquant que la multiplication fautée était une « fausse » multiplication et que le bit de l'exposant à cet instant valait donc 0.

Cette attaque ne s'applique pas à l'échelle de Montgomery, ni à l'exponentiation atomique.

5 Attaques plus complexes

Nous avons présenté les principales attaques par faute ou par analyse de puissance avec quelques unes des contre-mesures permettant aux implémentations de s'en prémunir. Cet article n'est cependant pas exhaustif et d'autres attaques plus complexes non présentées existent, ainsi que d'autres contre-mesures.

Malgré de telles protections, les attaquants et autres chercheurs en sécurité ont amélioré les attaques usuelles afin de les mettre en défaut. On parle notamment d'attaques d'ordre supérieur qui vont combiner plusieurs instants sur une même courbe afin de supprimer les effets des masques. Les développeurs vont ainsi devoir utiliser plusieurs masques rendant les développements plus compliqués.

Il est également possible de combiner les attaques actives et passives en une même attaque. Par exemple, une injection de faute modifie l'exécution du code ou la valeur de certaines données afin de créer lors de la suite de l'exécution une fuite en analyse de signal de type SPA.

Conclusion

Nous avons présenté quelques-unes des techniques d'implémentation sécurisée des algorithmes cryptographiques dans un microprocesseur embarqué. Développer un produit de sécurité n'est pas une chose aisée puisque cela nécessite une bonne connaissance de l'ensemble des attaques qui pourraient menacer les algorithmes. Cette connaissance des menaces permet ensuite de coder les protections adéquates qui souvent doivent composer avec des ressources mémoire limitées, comme c'est le cas dans la carte à puce.

Cela nécessite également de combiner intelligemment ces contre-mesures. En effet, chacune protège d'une ou plusieurs attaques mais ne suffit pas à elle seule à sécuriser tout le produit. Par ailleurs, il faut, pour réaliser cela, disposer d'une bonne compréhension du comportement physique de l'élément à sécuriser. La couverture de protection se doit d'être la plus exhaustive possible puisque un attaquant cherchera inévitablement à exploiter le point le plus faible du produit.

Pour réaliser ces tâches, il est nécessaire que les développeurs aient une bonne maîtrise de toutes les attaques connues. Cela requiert de maintenir ses connaissances en sécurité à l'état de l'art dans un domaine en perpétuelle évolution.

Toutefois, si ce travail est réalisé avec soin, si l'efficacité des contre-mesures a été vérifiée et correctement testée sur produit réel, alors le produit réalisé obtient son certificat et peut être mis sur le terrain avec un bon niveau de confiance.

Ce domaine scientifique s'applique bien entendu à tout type de composant électronique. Il est fort probable que, de par l'évolution rapide des objets électroniques, et des besoins de sécurité dans les usages quotidiens, développeurs et attaquants n'aient pas fini de jouer aux gendarmes et aux voleurs dans les années à venir. ■

REMERCIEMENTS

Je remercie Christophe Clavier, Christophe Giraud et Hugues Thiebeauld pour leurs remarques et conseils lors de la rédaction de cet article.

Les références de cet article sont disponibles sur : http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6/5

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

SÉQUESTRE DE CLÉS : MISE EN PRATIQUE DU « SHAMIR'S SECRET-SHARING SCHEME »

Philippe Teuwen - phil@teuwen.org

mots-clés : SSSS / SECRET-SHARING / OPENPGP / LIVE CD / KEY ESCROW

Ou comment livrer vos secrets à la postérité tout en vous prémunissant des faux frères grâce à un astucieux partage de clé secrète de Shamir exécuté depuis un live CD et accompagné d'un chouia de procédures.

1 Introduction

Nos pauvres petits neurones conservent tant bien que mal toujours plus de sésames. Mais nous ne sommes pas éternels et il serait dommage qu'avec nous disparaissent les accès à un héritage numérique, par exemple le mot de passe d'un disque chiffré ou d'un hypothétique site web commençant par « F » ou « G » qui contiendrait nos courriels, notre agenda, nos photos, nos fichiers, voire des musiques, films et jeux chèrement acquis.

Au sein d'une PME, la perte subite d'un collaborateur pourrait même devenir dommageable à l'entreprise peu prévoyante qui verrait alors la continuité de ses activités (*business continuity*) mise à mal. Nous continuerons à utiliser l'exemple de la PME par la suite, mais les principes de base restent transposables à un cercle familial ou d'amis.

Une réponse à cette problématique est ce qu'on appelle le séquestre de clé (*key escrow*), qui permet à un tiers « sous certaines conditions » d'accéder à ces clés. Mais quel tiers, sous quelles conditions, et comment lui accorder notre confiance morale mais aussi technique ? L'autorité de séquestre doit en effet être en mesure de garantir en toute sécurité la confidentialité des clés sous séquestre.

2 Clé de séquestre asymétrique

La première pièce du puzzle sera assez logiquement la création d'une paire de clés publique et privée propre à l'autorité de séquestre (entité qui reste à définir pour l'instant). Ainsi, toute personne désireuse de confier ses sésames n'aura qu'à les chiffrer elle-même avec cette clé publique et à les mettre à disposition voire à les rendre publiquement accessibles vu que seule l'autorité possède la clé privée correspondante.

La clé privée de séquestre sera protégée par une *passphrase* sur laquelle repose donc toute la sécurité du système. Mais qui sera le détenteur de cet ultime secret ?

3 Règle des deux hommes

Cette règle (*two-man rule*) est utilisée dans des domaines sensibles tels la commande d'envoi de missiles nucléaires afin d'éviter tout dérapage accidentel ou malicieux. En cryptographie, les Américains emploient la locution « two-person integrity » (TPI) quand il s'agit d'empêcher qu'une personne seule ait accès aux clés cryptographiques assurant la sécurité des communications (*COMSEC*).

Voici donc un concept intéressant qui nous aiderait à résoudre ces problèmes de confiance et de sécurité envers l'autorité de séquestre. En exigeant que deux individus collaborent pour révéler les données sous séquestre, nous nous mettons à l'abri d'un acte malveillant isolé, voire d'une faille de sécurité isolée.

Divisons donc la passphrase de la clé de séquestre et remettons les morceaux à un groupe de personnes de confiance constituant ce que nous appellerons dorénavant la *business continuity team* (BCT).

Comment morceler cette passphrase ? Répartir simplement *N* morceaux entre les *N* membres de la BCT les oblige à se réunir tous ensemble pour pouvoir

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

utiliser la clé privée de séquestre, mais si l'un d'eux est indisponible ou est justement le collaborateur disparu dont on souhaite récupérer les sésames, comment faire ?

4 Shamir's Secret-Sharing Scheme

Nous allons donc être amenés à répartir la passphrase de la clé privée de séquestre entre les personnes de confiance qui constituent la BCT, mais de sorte qu'il suffisse de réunir une fraction d'entre elles pour accéder à nouveau à la clé privée.

A priori, le problème n'est pas aisé. Pourtant, il a été résolu élégamment par Adi Shamir dès 1979 **[SHAMIR]** et la solution peut s'illustrer par une simple parabole.

Non, non, il ne s'agit pas d'allégorie, mais bien de mathématiques et de polynômes.

Deux points sont suffisants pour définir une ligne, trois pour définir une parabole, quatre pour une cubique, etc. Si maintenant nous voulons partager un secret, disons la valeur 1234, entre six individus et que trois d'entre eux soient nécessaires pour retrouver le secret, nous choisirons au hasard une parabole parmi celles passant par le point (0, 1234) et nous donnerons les coordonnées de six de ses points à ces six individus (cf. Figure 1).

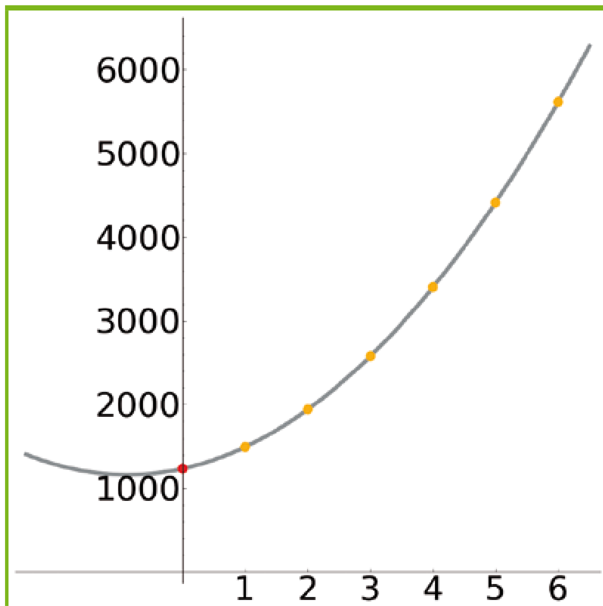


Figure 1 : Parabole passant par (0, 1234) et six de ses points.

Si seulement deux d'entre eux, les n° 2 et 4, venaient à partager leurs coordonnées, ils ne pourraient retrouver la parabole originale et donc la valeur du point secret en $x=0$ (cf. Figure 2).

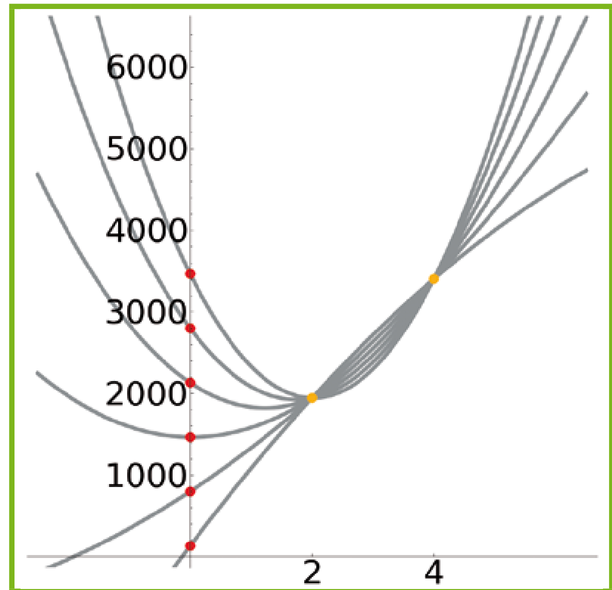


Figure 2 : Paraboles passant par les points des participants n°2 et 4.

Il faut donc bien qu'un troisième individu accepte de partager ses coordonnées pour parvenir à définir une et une seule parabole et révéler la valeur secrète 1234 (cf. Figure 3).

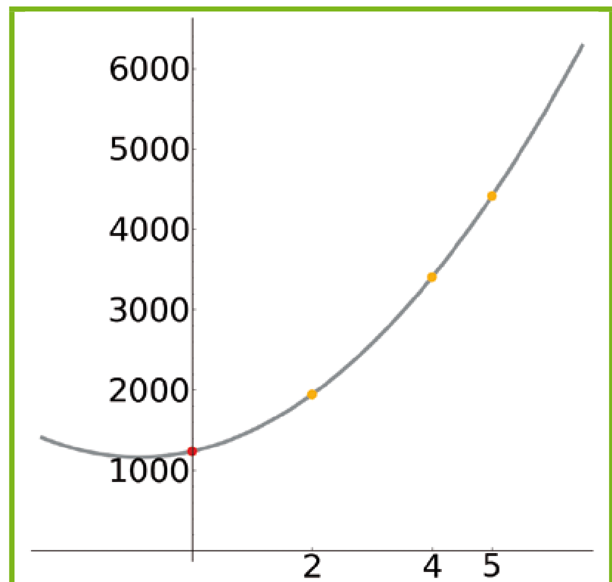


Figure 3 : Une seule parabole peut passer par les points des participants n°2, 4 et 5.

Les amateurs retrouveront les équations correspondant à cet exemple sur Wikipédia **[WIKIPEDIA]**.

Cet exemple illustré donne une idée du principe de ce protocole qui utilise en réalité des polynômes dans un corps fini, non représentable dans le plan.

Une implémentation libre existe **[SSSS]** et est disponible dans les principales distributions, notamment dans Debian/Ubuntu par un simple `apt-get install ssss`.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Deux commandes sont alors disponibles : **ssss-split** et **ssss-combine**.

Pour partager le secret « 1234 » de l'exemple précédent en six morceaux dont trois sont nécessaires à sa reconstruction, on exécutera **ssss-split** avec les options **-n 6** (six morceaux) et **-t 3** (dont trois nécessaires) :

```
$ ssss-split -t 3 -n 6
Generating shares using a (3,6) scheme with dynamic security
level.
Enter the secret, at most 128 ASCII characters: 1234
1-44eb879e
2-36abde94
3-43726a38
4-8eabfa4b
5-fb724ef5
6-893217db
```

Et pour le reconstituer, on utilisera **ssss-combine** avec les morceaux numéros 2, 4 et 5 :

```
$ ssss-combine -t 3
Enter 3 shares separated by newlines:
Share [1/3]: 2-36abde94
Share [2/3]: 4-8eabfa4b
Share [3/3]: 5-fb724ef5
Resulting secret: 1234
```

Mais pour mettre le tout en musique, il nous faut encore quelques procédures et moyens techniques garantissant qu'à aucun moment quelqu'un n'ait un accès intégral à la clé privée de séquestre ou à sa passphrase.

5 Assemblage

Afin d'éviter toute fuite, l'ensemble des opérations sensibles s'effectue sur un ordinateur sans unité de stockage ni réseau qui démarre sur un live CD contenant les quelques scripts nécessaires. Toutes les communications digitales se feront via des mémoires flash USB labellisées USBPUBLIC, USBSECURE1 et USBSECURE2. Ces dernières sont dédoublées pour offrir une redondance matérielle et ne pourront jamais être utilisées que via le live CD.

5.1 Génération de la clé de séquestre

Les clés publiques des membres de la BCT sont copiées sur USBPUBLIC après vérification de leurs signatures ou de leur empreinte. Ensuite, le live CD est démarré, les trois mémoires flash sont insérées et les clés publiques sont copiées de USBPUBLIC vers les deux mémoires USBSECURE.

Une passphrase aléatoire est générée et partagée en quatre secrets de Shamir (**ssss-split -t 2 -n 4**), un par membre de la BCT. Chaque membre chiffre son secret partagé de Shamir avec un mot de passe de son choix. Les secrets chiffrés sont alors sauvés sur les mémoires USBSECURE tandis que les mots de passe sont chiffrés à l'aide des clés publiques des membres correspondants et sauvés sur USBPUBLIC et USBSECURE.

La paire de clés de séquestre peut enfin être créée et la clé privée protégée par la passphrase aléatoire. La paire de clés est sauvée sur les USBSECURE et la clé publique sur USBPUBLIC.

L'empreinte (*fingerprint*) de la clé publique de séquestre est recopiée manuellement sur un papier et un certificat de révocation est généré et sauvé sur les trois mémoires.

Toutes ces étapes sont bien entendu semi-automatisées par un script idoine, les interactions avec les membres étant limitées à la saisie des mots de passe et l'affichage de l'empreinte (cf. Figure 4).

Figure 4 : Les étapes initiales de la création de la clé de séquestre

Le live CD est arrêté et USBPUBLIC est insérée dans un autre ordinateur. La clé publique de séquestre en est récupérée, son empreinte extraite, imprimée quatre fois et comparée à la transcription manuelle. Les copies imprimées sont remises à chaque membre de la BCT pour qu'il puisse signer en toute confiance cette nouvelle clé. Les mots de passe chiffrés sont expédiés aux membres correspondants. Le certificat de révocation est imprimé deux fois après quoi USBPUBLIC est effacée de manière sécurisée.

Deux copies du live CD, les mémoires flash USBSECURE1 et USBSECURE2 et les certificats de révocation sont répartis dans deux coffres dont les accès doivent être réglementés par une procédure adéquate.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

La clé publique de séquestre est signée par l'ensemble des membres de la BCT et mise à disposition de toute personne qui en aurait besoin.

5.2 Mise sous séquestre

L'individu désireux de placer sous séquestre une information récupère la clé publique de séquestre.

Il la valide en vérifiant que la clé est bien signée par les membres de la BCT et qu'il existe une chaîne de confiance entre ces membres et sa propre clé.

```
$ gpg --check-sigs --list-options show-uid-validity escrow@
ma_petite_entreprise.com
```

La sortie doit indiquer `[marginal]` (voire `[full]` s'il est lui-même membre de la BCT) devant l'uid.

S'il ne possède pas de clé OpenPGP ou que l'entreprise n'est pas adepte du *Web-of-Trust*, il peut toujours vérifier la clé de séquestre par son empreinte.

```
$ gpg --fingerprint escrow@ma_petite_entreprise.com
```

Il peut alors chiffrer en toute confiance l'information souhaitée et la placer là où la BCT l'aura convenu.

Les détails de la mise sous séquestre de chaque type d'information sensible (clé OpenPGP, certificat Outlook, mot de passe Truecrypt, *slot* d'une partition LUKS, etc.) doivent être documentés et expliqués aux utilisateurs ainsi que les conventions de nommage et de stockage choisies. Si certaines informations doivent d'abord être déchiffrées avant séquestre, il faut être très prudent : ne pas laisser de trace sur les disques, privilégier les *pipes* en ligne de commandes et s'il est inévitable de créer un fichier temporaire, l'effacer de manière sécurisée (avec wipe, shred, srm ou encore Heidi Eraser sous Windows).

Nous approfondirons le cas des clés OpenPGP, mais voyons d'abord comment se déroule l'opération inverse.

5.3 Récupération de données mises sous séquestre

La donnée à récupérer, donc chiffrée avec la clé de séquestre, sera placée sur la mémoire flash USBPUBLIC ainsi qu'éventuellement la clé publique du récipiendaire de cette donnée si on veut éviter que la donnée ne circule en clair.

Le live CD est démarré sur un ordinateur dépourvu d'unité de stockage et de réseau. La clé privée de séquestre et les secrets de Shamir chiffrés sont lus depuis une des deux mémoires USBSECURE. Deux membres de la BCT introduisent tour à tour le mot de passe qu'ils avaient choisi lors de la génération de la clé de séquestre. S'ils l'ont oublié, ils peuvent toujours se rafraîchir la mémoire en consultant la copie chiffrée qu'ils avaient reçue avant de venir participer à la procédure de récupération. La passphrase de la clé de séquestre est reconstituée et la donnée à récupérer est déchiffrée. Si nécessaire, elle sera re-chiffrée avec la clé publique du récipiendaire avant d'être copiée sur USBPUBLIC.

5.4 Particularités de la mise sous séquestre d'une clé OpenPGP

Si vous êtes amené à utiliser des clés OpenPGP dans un contexte professionnel avec une mise sous séquestre, il est préférable de ne pas utiliser votre clé personnelle mais de créer une nouvelle paire réservée à cet effet, voire deux paires : une pour le chiffrement et une pour la signature. Seule la clé de chiffrement est mise sous séquestre, et ce, sans passphrase. Ainsi, en cas de nécessité, vos communications chiffrées seront déchiffrées sans que personne n'apprenne votre passphrase ni ne soit en mesure de signer des messages en votre nom.

La mise sous séquestre se fera de la sorte :

```
$ gpg --export-secret-subkey --armor --export-options \
export-reset-subkey-passwd export-minimal alice@ma_petite_entreprise.com | \
gpg --encrypt --armor --recipient escrow@ma_petite_entreprise.com
--trust-model always > \
gnupg_private_yourname_pgpID_date.gpg
```

Si la clé devait être extraite du séquestre et rechiffrée avec la clé publique de la personne habilitée, celle-ci exécuterait :

```
$ gpg --decrypt gnupg_private.gpg |gpg --import --allow-secret-key-import
```

N'importe quel message se déchiffre alors le plus simplement du monde, sans passphrase :

```
$ gpg --decrypt anyfile.gpg
```

À noter que si vous êtes obligé par exemple par un tribunal de fournir le moyen de déchiffrer un message OpenPGP précis, ne donnez jamais votre clé secrète, mais bien la clé de session symétrique de ce message :

```
$ gpg --show-session-key --output /dev/null message.gpg
You need a passphrase to unlock the secret key for
user: "Philippe Teuwen (Doegox) <phil@teuwen.org>"
2048-bit ELG-E key, ID 9A4A59B9, created 2002-05-05 (main key ID 9AD7E3DB)

gpg: encrypted with 2048-bit ELG-E key, ID 9A4A59B9, created 2002-05-05
      "Philippe Teuwen (Doegox) <phil@teuwen.org>"
gpg: session key:
`9:8B4AD21DBF4AAACF3166DC0449211A2D523FAA2ED1F9C3EEBF7EC332C5B18A60'
```

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

La chaîne obtenue est alors utilisée pour déchiffrer directement le message :

```
$ gpg -override-session-key \
'9:8B4AD21DBF4AAACF3166DC0449211A2D523FAA2ED1F9C3EEBF7EC332C5B18A60' \
message.gpg
```

5.5 Cas particuliers

La procédure et les scripts doivent également couvrir certains aspects de maintenance.

Un changement de constitution de la BCT amène à la création d'un nouveau partage et à l'invalidation de l'ancien, le tout en deux temps pour éviter qu'une erreur sur le nouveau partage n'efface à jamais l'accès à la passphrase : seule la mémoire USBSECURE1 sera modifiée et la procédure de récupération de données testée, et en cas de succès, le contenu de USBSECURE2 remplacera celui de USBSECURE1. L'effacement sécurisé des anciens secrets de Shamir sur les USBSECURE suffit à invalider l'ancienne BCT puisqu'il n'y a pas d'autre copie disponible.

D'autres cas particuliers sont par exemple la perte du mot de passe chiffré d'un des membres de la BCT dont la version chiffrée est alors simplement récupérée d'un des USBSECURE et recopiée sur USBPUBLIC.

Enfin, il est sage de tester régulièrement les différentes procédures, par exemple des essais de récupération de données de test mises sous séquestre.

Conclusions et derniers conseils

Le séquestre de clé en entreprise est avant tout destiné à assurer la continuité des activités, il serait regrettable qu'une fausse manœuvre, un support défectueux ou un bug ruine ce bel échafaudage. En pratique, chaque étape intégrera la redondance et les validations nécessaires pour éviter toute perte ou corruption de données. En outre, il est intéressant de prévoir une trace de l'ensemble des opérations effectuées sur le live CD qui sera retranscrite sur chacune des mémoires USB en fin de session.

La création d'un tel séquestre n'est pas qu'une affaire d'ingénieurs, de crypto et de live CD. C'est avant tout un élément à intégrer à la vie de l'entreprise et à faire accepter aux collaborateurs. Sur ces aspects une documentation claire et exhaustive des procédures à l'usage des membres de la BCT mais aussi de tout un chacun est primordiale et comme on dit : « le diable se cache dans les détails ».

Quelle information placer sous séquestre, quand, sous quelle forme, avec quelle étiquette, à quel endroit ? Qui peut activer la BCT pour récupérer une donnée ? En quelles circonstances ? Quelles procédures appliquer à l'accès physique aux live CD et USBSECURE si on souhaite garantir la règle des deux hommes? Autant de questions qui dépendent de l'environnement dans lequel le déploiement est envisagé.

Nous voici à la fin de cet article qui a tenté de vous montrer une solution de séquestre dont l'autorité est décentralisée ou du moins oligarchique grâce à un partage de Shamir qui devrait gagner la confiance de ses utilisateurs tout en garantissant une certaine résilience quant à son exécution.

Les scripts du live CD sont à disposition dans un projet Google **[ESSSSCROW]** créé pour l'occasion. Toute collaboration ou retour d'expérience sont les bienvenus. ■

Note sur les générateurs d'aléa :

Un ordinateur dépourvu d'unité de stockage et de réseau n'a que peu de sources d'entropie à sa disposition s'il n'a pas de composant matériel spécifique (TRNG) pour créer la clé de séquestre. Il peut donc être utile d'intégrer au live CD le paquet haveged. Il s'agit d'une implémentation de l'algorithme HAVEGE (*HArdware Volatile Entropy Gathering and Expansion*) [HAVEGE] qui collecte un maximum d'entropie, notamment par une mesure du temps d'exécution d'une séquence d'instructions, sensible aux prédictions de branchement, à l'état des caches et de divers états internes hautement volatiles du processeur.

RÉFÉRENCES

[SHAMIR] Shamir, Adi (1979), «How to share a secret», Communications of the ACM 22 (11): 612–613

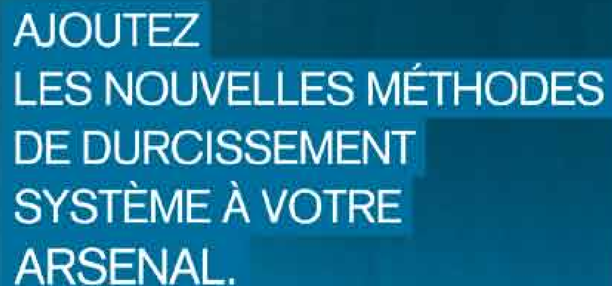
[WIKIPEDIA] https://fr.wikipedia.org/wiki/Partage_de_clé_secrète_de_Shamir

[SSSS] http://point-at-infinity.org/ssss/

[HAVEGE] http://www.irisa.fr/caps/projects/hipsor/

[ESSSSCROW] https://code.google.com/p/esssscrow/

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015



FORMATIONS SÉCURISATION

Cours SANS Institute
Certifications GIAC

SEC 505
Sécuriser Windows

SEC 506
Sécuriser Unix & Linux

DEV 522
Durcissement des applications Web

Dates et plan disponibles
Renseignements et inscriptions
par téléphone +33 (0) 141 409 700
ou par courriel à : formations@hsc.fr

www.hsc-formation.fr

SANS

GIAC

HSC

exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

REVERSE ENGINEERING D'UNE CHARMANTE APPLICATION 16 BITS

Philippe Laulheret - philippe.laulheret@supelec.fr

Éleve ingénieur Supélec

mots-clés : REVERSE ENGINEERING / 16 BITS / ATTAQUE PAR CLAIR CONNU

Petit flashback. Nous sommes à la fin des années 90, Windows Millenium est sur le point de sortir, les baladeurs mp3 commencent à se multiplier, et dans l'indifférence générale, les New Executable (NE) de Microsoft périclitent. Heureusement, quelques braves logiciels à la pointe de la technologie persistent à assurer la compatibilité avec Windows 3.x. Nous allons nous intéresser aujourd'hui à l'un d'entre eux, qui porte le doux nom, ô combien évocateur, de charme.exe.

1 Présentation du problème

1.1 Disclaimer

Cet article se veut avant tout didactique et dans la mesure du possible divertissant. Il n'est pas attendu du lecteur une connaissance poussée en RE. Les fichiers sur lesquels portent cette étude peuvent être trouvés ici **[PCFun39]**.

1.2 Description du logiciel

Le logiciel `charme.exe` est un binaire 16 bits dont le but est de déchiffrer des fichiers pour adultes en échange d'un code obtenu par minitel. Ces fichiers chiffrés portent l'extension `.XXX` et sont stockés dans le répertoire `ADULTES`.

Des images en clair sont également disponibles pour attirer le chaland. Elles portent l'extension `.ICA`. Il est amusant de noter que des images bonus sont également présentes, sous l'extension `.IBA`. Elles sont accessibles une fois un premier déchiffrement effectué et ne différent en rien des images précédentes. Il est facile alors de changer leur extension pour les rendre disponibles immédiatement...

1.3 Petit tour du propriétaire

Avant de plonger tête baissée dans l'étude de l'exécutable, il peut être intéressant d'observer de plus près les différents contenus.

En ouvrant à l'éditeur hexadécimal les fichiers images « en clair », on peut voir qu'ils commencent par le texte `IMAGES` ; quelques octets plus loin, on peut voir le nom du fichier en `.bmp` puis des données quelconques. Au vu de la taille des fichiers (entre 75 et 150 ko) et l'aspect aléatoire des données présentes, il est clair que l'on a affaire ici à des données compressées. En comparant divers fichiers `.ICA`, on peut également remarquer que les 20 premiers octets sont toujours identiques.

```
494D 4147 4553 0000 00B8 F950 00FF FFFF   IMAGES...,ùP.ÿÿÿ
FF01 3F5C 8001 0000 0028 40F6 BFB8 F950   ÿ.?\€....(@ö¿,ùP
00A0 3C5C 805F 0FBC A424 0000 00BE 3C5C   . <\€_.¼¤$...¾<\
8000 0086 0200 005F 0F86 02F6 A4A4 0BA7   €..†..._.†.ö¤¤.§
4086 025F 0F48 0B63 3ADC 4106 0AAC A442   @†._.H.c:ÜA..¬¤B
1900 0000 005F 0F0B 414E 4E45 2D30 332E   ....._..ANNE-03.
424D 505F 2000 644F AC24 36B4 0400 E16C   BMP_ .dO¬$6´..ál
```

Figure 1 : En-tête d'un fichier image en clair « .ICA ». En rouge, la zone invariante parmi tous les fichiers images.

Malheureusement, la structure des images chiffrées est différente. Les fichiers `.XXX` commencent par des informations de *copyright*, suivies du nom de l'archive (destinée à être saisie au minitel), la liste des fichiers présents, puis un bloc d'environ 240 octets presque

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

tout le temps nuls, parsemé de quelques valeurs différentes de zéro. Nous y reviendrons dans quelques instants. Enfin, le reste du fichier contient les données à proprement parler, chiffrées/compressées.

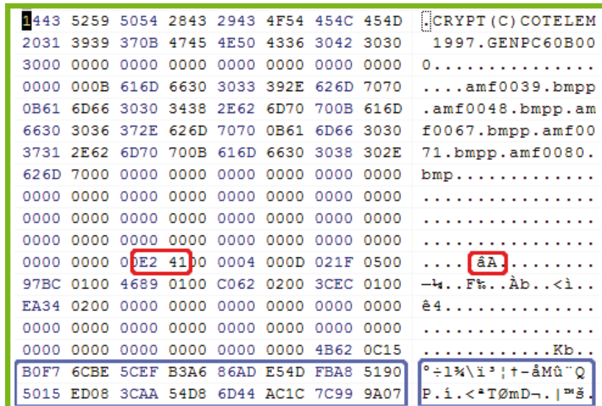


Figure 2 : Début d'un fichier archive. En rouge, le checksum ; en bleu, le début des données cryptées.

2 Plateforme de test et gros pixels

2.1 Où est-ce que j'ai bien pu ranger mes disquettes ?

IDA Pro désassemble sans souci le logiciel, mais les debuggers usuels se cassent les dents sur ce format 16 bits, il est donc temps de partir au grenier chercher les disquettes d'installation de Windows 3, car c'est l'heure de la leçon rétro de rétroingénierie !

La première étape est donc d'installer notre OS sur une machine virtuelle (Virtual Box, par exemple). Une fois la vague de nostalgie passée (et les 5 ou 6 changements de disquette), notre machine virtuelle est prête, et n'attend plus qu'on lui présente un gentil debugger. Le plus efficace pour notre propos est sans doute Turbo Debugger de Borland (**`TDW.exe`**), que l'on peut trouver, entre autres, avec Turbo Pascal pour Windows.

2.2 Segmentation ?

L'utilisation de TDW est assez intuitive, mais certains automatismes ne sont pas forcément les mêmes que sur une architecture x86-32bits usuelle. L'un des éléments les plus déroutants est de prime abord l'utilisation d'une mémoire segmentée **[Seg]**.

Une adresse va être de la forme **`selecteur:index`**. Imaginons que l'on veuille placer un *breakpoint* sur une fonction que nous montre IDA, au hasard **`sub_919C`**. Dans notre désassembler fétiche, en appuyant sur *cltr+espace* avec le curseur au début de la fonction, IDA va afficher dans la console **`cseg03:11BC`**. C'est l'adresse qu'il faudra entrer dans TDW. Le mapping en mémoire des segments varie à chaque exécution, pour connaître la valeur de **`cseg03`**, le plus rapide est de trouver une fonction appelée en début de programme qui se trouve dans ce segment et noter directement la valeur correspondante.

Il est aussi à noter que lorsque une adresse est passée en argument à une fonction, on va voir deux **`push`**, d'abord le segment, ensuite l'index.

```
push ds
push di
call foo
```

Maintenant que nos outils sont en place, il est temps de s'amuser un peu.

3 Étude

3.1 Approche bad boy

Une première approche consiste à chercher la routine qui affiche « Cette clé n'est pas la bonne » et à changer l'instruction de branchement. La voici :

```
loc_2AF8:                    ; Mot de passe
lea     di, [bp+var_102]
push    ss
push    di
mov     di, offset byte_4D310 ; Nom archive
push    ds
push    di
call    sub_9336
or      al, al
jz      short loc_2B28
```

Figure 3 : Le message classique « bad boy » qui n'est pas le seul obstacle.

TDW nous permet de voir que la fonction appelée prend en paramètres le mot de passe saisi et le nom du fichier archive.

Remplacer le **`jz`** par un **`jmp`** lance en effet la décompression des images, mais nous gratifie d'une erreur « Le fichier archive est corrompu ». Nous n'avons cependant pas tout perdu, l'application nous autorise maintenant à voir les images bonus...

3.2 Approche moins naïve

Si l'application est capable de nous dire quasi immédiatement que la clé n'est pas la bonne, peut-être que nous pouvons en tirer parti pour trouver ce graal : étudions donc cette routine.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

```
call    @TFileStream@Create$q6String4Word ; TFileStream::Create(String,Word)
mov     word ptr [bp+var_20A], ax
mov     word ptr [bp+var_20A+2], dx
push    0
push    0
push    0
les     di, [bp+var_20A]
push    es
push    di
les     di, es:[di]
call    dword ptr es:[di+8] ; Sans doute seekPos(0)
mov     di, 19A4h
push    ds
push    di
push    0
push    17Ch
les     di, [bp+var_20A]
push    es
push    di
les     di, es:[di]
call    dword ptr es:[di] ; ReadBuffer(ds:19A4,17C)
mov     [bp+var_205], 0
mov     di, 1AA0h
push    ds
push    di
push    80h ; 'Ç'
lea     di, [bp+var_202] ; Le mdp
push    ss
push    di
call    decrypt
```

Figure 4 : Chargement du fichier

La fonction ouvre un **TFileStream** et lit **0x17C** octets à l'adresse **dseg19:19A4**, puis passe une partie de ce qui a été lu à la fonction **decrypt**.

On peut considérer pour le moment la fonction **decrypt** comme une boîte noire, voyons la suite du traitement.

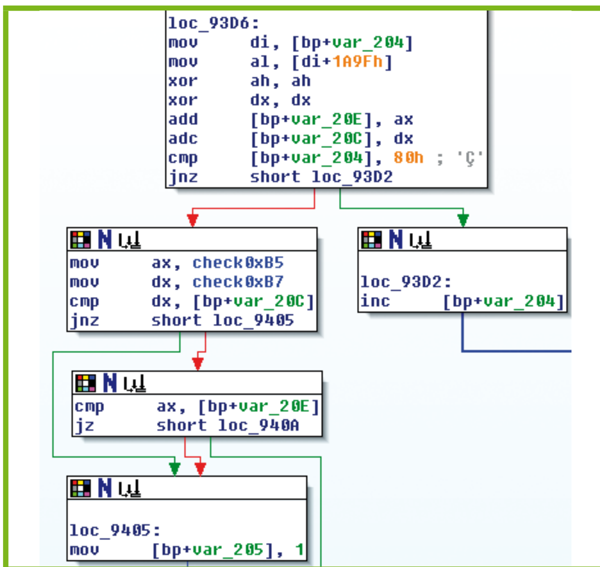


Figure 5 : Calcul d'un checksum rudimentaire

Les opérations suivantes forment un *checksum* très basique : on somme octet par octet le résultat du déchiffrement, et on le compare à une valeur stockée dans le fichier archive.

Connaissant la valeur du checksum, une approche par bruteforce pourrait être envisagée. Le mot de passe est long de 14 caractères, et l'alphabet est de 26+10-3=33 caractères (certains caractères ont été retirés pour éviter les confusions comme le O et le 0). Le nombre de combinaisons possibles est donc 33^14, ce qui fait penser que notre solution est non optimale.

```
000A0 0000 0000 0000 0000 0000 0000 0000 0000  ................
000B0 0000 0000 00E2 4100 0004 000D 021F 0500  .....âA.........
000C0 97BC 0100 4689 0100 C062 0200 3CEC 0100  —¼..F‰..Àb..<ì..
000D0 EA34 0200 0000 0000 0000 0000 0000 0000  ê4..............
000E0 0000 0000 0000 0000 0000 0000 0000 0000  ................
000F0 0000 0000 0000 0000 0000 0000 4B62 0C15  ............Kb..
00100 B0F7 6CBE 5CEF B3A6 86AD E54D FBA8 5190  °÷l¾\ï³¦†-åMû¨Q
00110 5015 ED08 3CAA 54D8 6D44 AC1C 7C99 9A07  P.í.<ªTØmD¬.|™š.
00120 3643 7396 FFDF 81D0 7562 DFF5 F822 3DB7  6Cs–ÿß ÐubßõøÖ"=·
00130 3ED4 447A 350B E157 CF58 7386 6578 A107  >ÔDz5.áWÏXs†ex¡.
00140 47FC D285 9721 2D18 5BAF 882A C134 3C16  GüÒ…—!-.[¯ˆ*Á4<.
00150 9555 02C8 C65B BE15 ECF1 B93D 5636 0F6F  •U.ÈÆ[¾.ìñ¹=V6.o
00160 D35C 2DD0 EAD4 81C1 B5DD DDF5 8DDC 4CD5  Ó\-ÐêÔ ÁµÝÝõ ÜLÕ
00170 927A FE13 0B8A 4295 68E4 8404 D6A8 0448  'zþ..ŠB•hä„.Ö¨.H
00180 3353 80A0 096D C1A0 21CC 49F0 78A3 782E  3S€ .mÁ !ÌIðx£x.
00190 7601 3A87 8403 2F46 5D53 9361 8B5B A805  v.:‡„./F]S“a‹[¨.
001A0 0C97 0538 BBAB 8C3B 0A3D 7315 741D BB75  .—.8»«Œ;.=s.t.»u
001B0 9B65 33F0 9CBD 328B 3241 7658 BBEB 0740  ›e3ðœ½2‹2AvX»ë.@
001C0 B49B 0B61 B1DF A4BC 8C30 37A4 BA30 0E83  ´›.a±ß¤¼Œ07¤º0.ƒ
001D0 852D 3CB1 3CA4 0017 76E3 A30A F3B2 63A8  …-<±<¤..vã£.ó²c¨
001E0 B1A0 A2B6 DA28 20DF B5BE 2BFF 73F4 DA06  ± ¢¶Ú( ßµ¾+ÿsôÚ.
```

Figure 6 : En rouge, le checksum. En bleu, les données sur lesquelles il porte. En vert, le début de la première image cryptée.

3.3 Étude de la fonction de déchiffrement

On peut décomposer la fonction de déchiffrement en deux sous-fonctions. Nous allons voir que toutes deux sont inversibles (ce qui est bien heureux pour des fonctions de chiffrement/déchiffrement).

3.3.1 Permutation

Voici la première étape, appliquée au mot de passe saisi par l'utilisateur :

```
                mov     al, 1
                cmp     al, [bp+var_13]
                ja      short loc_917F
                mov     [bp+var_count], al
                jmp     short loc_9158
; ---------------------------------------------------------------

loc_9155:                               ; CODE XREF: permutation+53↓j
                inc     [bp+var_count]

loc_9158:                               ; CODE XREF: permutation+29↑j
                mov     al, [bp+var_count]
                xor     ah, ah
                mov     di, ax
                mov     al, [di+463h]
                xor     ah, ah
                les     di, [bp+arg_mdp]
                add     di, ax
                mov     dl, es:[di]
                mov     al, [bp+var_count]
                xor     ah, ah
                mov     di, ax
                mov     [bp+di+var_12], dl
                mov     al, [bp+var_count]
                cmp     al, [bp+var_13]
                jnz     short loc_9155
```

Figure 7 : Fonction de permutation

On initialise **var_count** à 1 et on va l'incrémenter jusqu'à avoir atteint la fin du mot de passe. Puis on regarde la valeur à **cs : [var_count + 0x463]**, que l'on va noter k et on copie dans un *buffer* à la position **var_count** la valeur du mot de passe en k.

Pour information, la permutation effectuée et son inverse sont les suivants :

```
Sigma = ( 0, 10, 2, 7, 5, 1, 3, 9, 8, 6, 4, 11, 12, 13, 14,15,16 )
invSigma = ( 0, 5, 2 , 6, 10, 4, 9, 3, 8, 7, 1, 11, 12, 13, 14,15,16 )
```

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

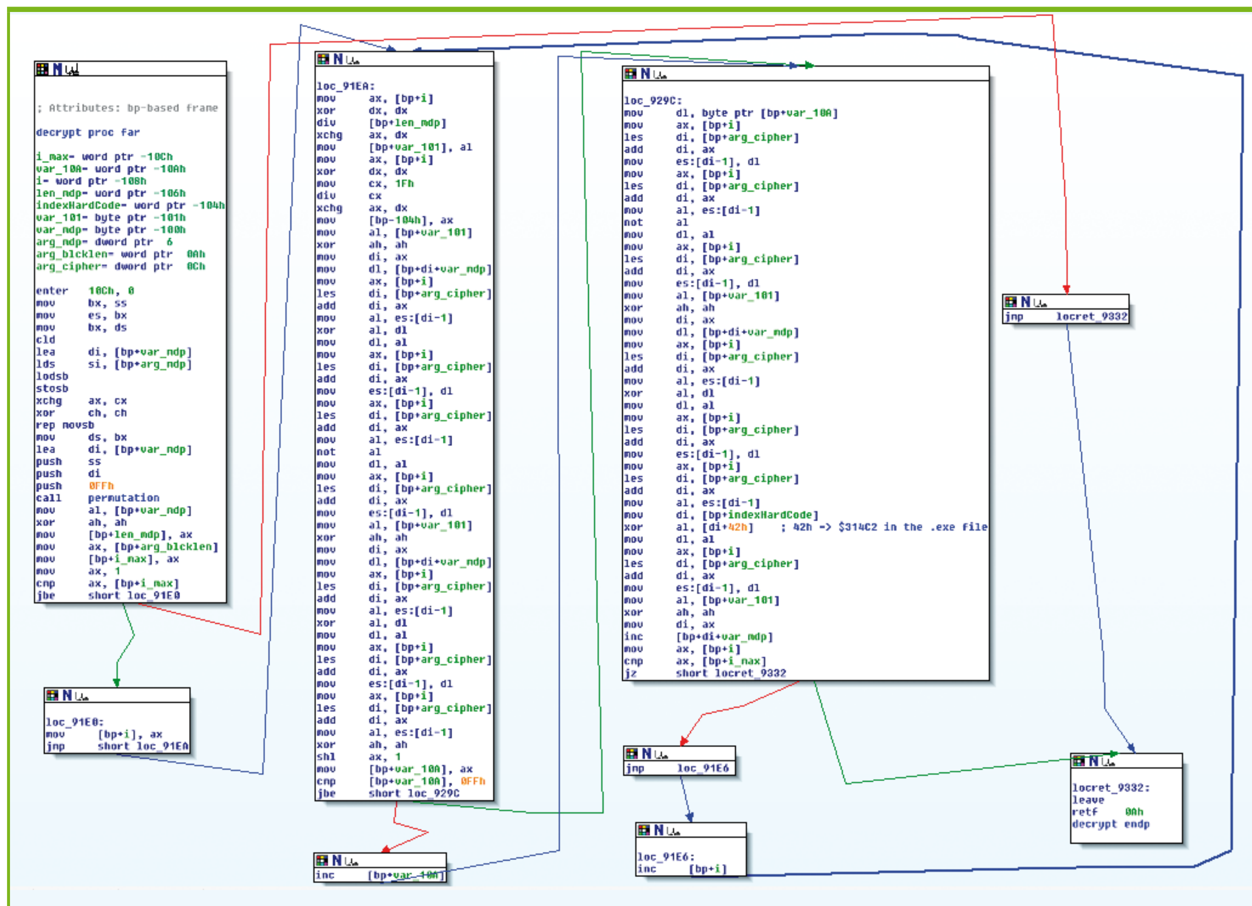


Figure 8 : Fonction de déchiffrement

Cette permutation est indépendante des données entrées, et n'accroît pas la « force » cryptographique de l'algorithme : supposons que l'on puisse casser l'algorithme sans la permutation, il suffit de faire subir la permutation inverse, notée **invSigma**, au mot de passe trouvé pour casser l'algorithme avec permutation. Elle évite juste qu'il reste en clair en mémoire.

Une fois que le mot de passe a été permuté, il est fourni au reste de l'algorithme pour accomplir le déchiffrement.

3.3.2 Déchiffrement

La routine de déchiffrement étant un peu longue, la voici dans sa version assembleur : voir Figure 8.

Nous pouvons néanmoins nous intéresser à son équivalent c, plus compacte :

```
for (int i=1; i < blockLen; i++) {
   int k = i % mdpLen;
   int indexHard = i % 0x1F;
   char cipher;
   cipher = myCipher[i-1];
   cipher = mdp[k] ^ cipher;
   cipher = ~cipher;
   cipher = cipher ^ mdp[k]  ;
   unsigned short sh = cipher;
   sh = sh << 1;    // sh = 2*sh;
   unsigned short comp = 0x0FF;
   if ((sh > comp)) sh++;
   cipher = (char) sh;
   cipher = ~ cipher;
   clear[i-1] = cipher ^ mdp[k] ^ hardKey[i] ;
   mdp[k]++ ;
}
```

hardKey - *no pun intended* - est une clé codée en dur dans le logiciel.

On remarque que la plupart des opérations sont relativement simples, consistant en des XOR et des NOT. La partie du code mise en relief est la plus intéressante, il s'agit d'un SHL (les 0 et les 1 de la représentation binaire du caractère sont décalés d'une place vers la gauche). La comparaison avec 0xFF peut sembler déroutante de prime abord. En réalité, il s'agit de simuler un SHL d'un octet sur un registre 16 bits : un SHL sur un **char** nous ferait perdre la valeur du bit de poids fort (remplacé par un 0 au bit

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

de poids faible). En utilisant un `short`, on peut voir si la valeur après le SHL est supérieure à 255, et si c'est le cas, on met à 1 le bit de poids faible (`sh++`), puis on cast le `short` en `char` pour se débarrasser des bits superflus.

Il est rassurant de noter que comme annoncé précédemment, toutes les opérations effectuées sur les données son inversibles...

D'autre part, il est important de remarquer que pour déchiffrer un octet en position `i-1`, les seules variables utiles sont le caractère du mot de passe (après permutation) à la position `i % mdpLen`, et un des caractères de la clé codée en dur (à la position `i % 0x1F`) : si notre mot de passe fait 10 caractères de long, changer exactement un de ses caractères ne va changer qu'un seul octet tous les 10 octets du résultat final.

3.4 Be smart for fun and boobies

3.4.1 You've been a bad boy II

Savoir que les opérations effectuées sur les données sont inversibles, c'est bien, mais malheureusement on ne connaît ni le mot de passe, ni les données en clair. Il nous faut donc chercher ailleurs.

Reprenons donc le raisonnement de tout à l'heure : faisons un `jmp` par-dessus le message d'erreur indiquant un mauvais mot de passe, et regardons comment le programme fait pour nous annoncer que notre archive est corrompue.

Une recherche dans IDA nous montre que la décision est prise à la `loc_879C` :

```
lea     di, [bp+var_2C8]
push    ss
push    di
mov     di, offset unk_4BC5E
push    ds
push    di
push    6
call    @$basg$qm6Stringn4Char4Byte ; operator=(String &,Char *,Byte)
call    @$bsub$qm6Stringt1 ; operator-(String &,String &)
jz      short loc_87D1
mov     di, offset aLeFichierArchi ; "Le fichier archive est comrrompu"
```

Figure 9 : Mince ! Notre archive est corrompue.

Ce code n'est pas vraiment clair, et une étude dynamique peut se montrer intéressante : voir Figure 10

Comme on peut le voir en traquant les variables du segment de données, le mot-clé `IMAGES` réapparaît. Pour tester l'intégrité de l'archive une fois décryptée, l'auteur du programme cherche dans le résultat obtenu après déchiffrement le même en-tête que celui présent dans les fichiers en clair que nous avions vu dans notre étude préliminaire. D'un point de vue génie logiciel, il semble logique que l'auteur ait conservé le format

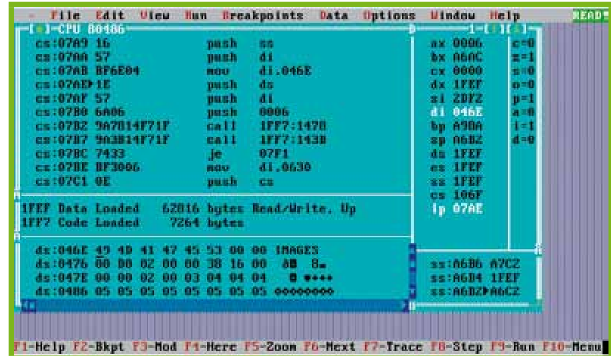


Figure 10 : TDW nous montre quelque chose d'intéressant.

d'image qu'il utilisait pour encoder les images en clair, en l'encapsulant dans une couche cryptographique. C'est pour cela que l'on retrouve le même en-tête que précédemment.

La suite s'impose d'elle-même...

3.4.2 Owned ! Plain text attack

Pour résumer, nous savons que les fichiers commençant par IMAGES contiennent les images en clair, et débutent tous par la même série d'une vingtaine de caractères. D'autre part, on peut voir que les données dans les archives chiffrées commencent à être déchiffrées juste après le checksum.

Nous savons donc précisément que la zone en rouge de la figure 1 est le clair de la zone verte de la figure 6.

Une résolution analytique est sans doute possible, mais au vu de la complexité du problème, un simple brute force est tout à fait envisageable : le nombre d'essai est au pire de 33*14 pour un mot de passe de longueur 14.

Il est bon de noter que la complexité de notre brute force est passée d'exponentielle à linéaire : nous avons vu que chaque octet déchiffré ne dépend que d'un seul octet de la clé de déchiffrement.

La mise en place du brute force revient essentiellement à placer une boucle `for` extérieure autour de la fonction de déchiffrement et d'essayer toutes les valeurs possibles pour le caractère du mot de passe. Une remarque néanmoins : le mot de passe peut avoir une longueur variable, entre 10 et 16 caractères. Il existe une méthode simple pour déterminer sa longueur correcte : lorsque `charme.exe` charge le mot de passe en mémoire, il lui ajoute un caractère '`\n`' (qui ne fait pas partie des caractères que l'utilisateur peut entrer). Durant le brute force, on va donc rencontrer un caractère « indéchiffrable » qui sera en réalité chiffré avec le caractère spécial '`\n`'.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Et heureuse coïncidence, ce caractère est toujours en position 0 du mot de passe permuté. On sait donc à ce moment que l'on a fait un tour modulo la longueur du mot de passe.

```
#define MAGIC_OFFSET 0x17C
#define mdpLen 16
#define possibleLen 36// pas de  L O

int permutation[mdpLen +1] = {0, 10, 2, 7, 5, 1, 3, 9, 8, 6, 4,
11, 12, 13, 14, 15, 16 };
int rev[mdpLen+1] = { 0, 5, 2 , 6, 10, 4, 9, 3, 8, 7, 1, 11, 12,
13, 14,15,16 }   ;
char mdp[mdpLen +1];
int  indexMdp[mdpLen +1];
char possibleValue[possibleLen] =
"L6OABCDEFGHIJKMNPQRSTUVWXYZ012345789";
char hardKey[] = "\xE6\xA1\x2F\x07\x9D\x15\xC4\x37\x0A\x20\x20\
x20TO YQD-80 07TO YOUR\0\0\0\0";
char valueToRecognize[]="IMAGES\x00\x00\x00\xB8\xF9\x50\x00\xFF\
xFF\xFF\xFF\x01\x3F\x5C\x80";
char myCipher[mdpLen];
int breakAt = mdpLen;  //mdp can be shorter than 16 char

   /*  [...]
   fseek(myFilz, MAGIC_OFFSET, SEEK_SET);
   fread(myCipher,1,mdpLen,myFilz)
   [...] */

void bruteforce() {
  for (int i=1; i < mdpLen +1; i++) {
    int trouve =0;
    int k = i % mdpLen;
    int indexHard = i % 0x1F;
    char cipher;

    for (int cur=0; cur <possibleLen; cur++) {
      mdp[k] = possibleValue[cur];
      cipher = myCipher[i-1];
      cipher = mdp[k] ^ cipher;
      cipher = ~cipher;
      cipher = cipher ^ mdp[k]   ;
      unsigned short sh = cipher;
      sh = sh << 1;
      unsigned short comp = 0x0FF;
      if ((sh > comp)) sh++;
      cipher = (char) sh;
      cipher = ~ cipher;
      cipher = cipher ^ mdp[k] ^ hardKey[i] ;
      if ((cipher == valueToRecognize[i-1])) {
        trouve=1;
        break;
      }
      if (!trouve){   // in charme.exe we have mdp[0] == '\n'
          mdp[k]='x';  //which is not a valid char as a user input
          if (breakAt == mdpLen) breakAt = k;
      }
    }
  }
```

```
  printf("Da pass iz : ");
  for (int i = 1; i < breakAt+1;i++) printf("%c", mdp[rev[i]]);
}
```

4 Le mot de la fin

4.1 Remarque

En parcourant le code désassemblé, on peut trouver des messages d'erreur qui ne sont jamais affichés dans le programme. D'autre part, plusieurs versions du logiciel ont circulé à la fin des années 90. On peut penser que les différentes techniques de checksum, la présence de la clé en dur dans le programme s'expliquent par une évolution progressive de celui-ci, car déjà à l'époque, certains magazines publiaient des programmes en Basic pour contourner des protections par XOR...

Conclusion

La protection rudimentaire de `charme.exe` peut faire rire aujourd'hui, mais à l'époque, les outils et les techniques n'étaient pas forcément aussi développés qu'ils le sont maintenant (bien qu'une personne motivée aurait pu venir à bout de l'algorithme avec simplement TDW et un outil de *dead listing*). L'affaiblissement de la sécurité permettant une attaque par clair connu est néanmoins regrettable.

Il est intéressant de voir qu'il y a plus de 10 ans déjà, ceux qui protègent les données, et ceux qui tentent d'y accéder de manière illégitime jouaient déjà au chat et à la souris en complexifiant toujours plus les algorithmes de protection.

Enfin, pour faire écho au dossier de *MISC* **[Misc57]** dédié à l'industrie pornographique, on peut saluer ce logiciel qui était une vraie innovation technologique/stratégique à l'époque. Le minitel rose d'aujourd'hui n'est plus en 8 couleurs, mais la protection de contenu à l'aide de primitive cryptographique reste une question d'actualité. ■

RÉFÉRENCES

[PCFun39] http://www.abandonware-magazines.org/download.php?num=2614

[Seg] http://en.wikipedia.org/wiki/X86_memory_segmentation

[Misc57] Sexe, Drogue et Sécurité informatique – MISC n°57 Septembre/Octobre 2011

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

LA CRYPTOGRAPHIE DANS RDP
RSA, ANECDOTES ET ERREURS D'IMPLÉMENTATION

Jérome Gabriac - jgabriac@quarkslab.com et Sébastien Kaczmarek - skaczmarek@quarkslab.com (@deesse_k)

mots-clés : RDP / RSA-512 / FACTORISATION

Le protocole RDP est aujourd'hui utilisé quotidiennement dans les entreprises et reste un des incontournables pour la prise en main à distance. D'un point de vue cryptographique, des faiblesses de conception ont été relevées il y a maintenant plusieurs années mais restent toujours d'actualité sur certains systèmes, notamment Windows XP et les premières versions de Windows 2003 Server. Cet article propose de revenir sur ces vulnérabilités en étudiant deux implémentations différentes, celle de Microsoft et celle de Xrdp. En conséquence directe de nos analyses, nous présenterons un exemple pratique de la factorisation d'un modulus RSA 512 bits.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

1 Contexte

Les spécifications du protocole RDP **[1]** sont officielles depuis environ 5 ans. Malgré une simplicité supposée standard, les spécifications sont composées d'environ une trentaine de PDF de 50 pages en moyenne et le protocole comprend en réalité de nombreuses extensions : impression, transports d'objets DirectX, codec remoteFx, canaux USB, ... Tout autant de points d'entrée que pourraient exploiter un attaquant en cas de présence d'une faille critique.

Cependant, dans cet article, nous nous focaliserons essentiellement sur les premières phases du protocole, c'est-à-dire lors de l'établissement de la connexion et de la phase de négociation des clés de chiffrement pour la suite des échanges. Les vulnérabilités décrites par la suite sont connues depuis de nombreuses années mais permettront d'illustrer concrètement une méthodologie pratique de factorisation de grands nombres (plus de 150 chiffres décimaux) et de mettre en avant quelques pièges à éviter lors du développement d'un protocole s'appuyant sur des primitives cryptographiques.

Pour plus de précisions, le lecteur intéressé pourra parcourir un dossier de recherche **[2]** retraçant l'histoire de l'implémentation de la cryptographie dans le protocole RDP ainsi que plusieurs analyses de sécurité le concernant.

2 RDP et cryptographie dans Windows

2.1 Historique

Le support du chiffrement est natif depuis la version 4.0 introduite dans Windows NT. À l'époque, un seul mode de sécurité était disponible : *Standard RDP security*, encore utilisé aujourd'hui. Il repose sur l'échange d'une clé RC4 (40, 56 ou 128 bits), et sur RSA.

Par la suite, à partir de Windows 2003 SP1 jusqu'à Windows 7, de nouvelles fonctionnalités de sécurité ont vu le jour, il s'agit du *Enhanced Security Mode* :

- arrivée de TLS ;
- CredSSP (authentification NTLM/Kerberos basée sur TLS), d'autres packages d'authentification peuvent être utilisés.

Le niveau FIPS avec chiffrement 3DES a également été ajouté par la suite dans *Standard Security Mode* mais s'avère peu utilisé en pratique en raison des contraintes d'utilisation.

2.2 Les différents modes

RDP offre principalement 2 modes de sécurité :

- Standard security mode : le mode évoqué précédemment. Sa principale faiblesse est qu'il repose sur une clé RSA-512 bits lors de la phase d'échange de clés, et ceci jusque dans les versions Windows XP et server 2003.
- Enhanced security mode : ce mode introduit notamment TLS (`schannel.dll`) et CredSSP qui permet d'utiliser une authentification de type NTLM/Kerberos encapsulée dans TLS.

Par la suite, nous proposons d'analyser plus en détail le « standard security mode », nous rappellerons ainsi comment il est possible d'attaquer en pratique ce mode, utilisé encore par défaut sous Windows XP SP3 avec une clé RSA de 512 bits.

2.3 Le cas de standard security

Lors d'un échange utilisant le protocole RDP avec Standard Security Mode, le client et le serveur doivent se mettre en accord sur trois clés de chiffrement distinctes à utiliser pour tenter de sécuriser le canal de communication :

- la clé de chiffrement client/déchiffrement serveur ;
- la clé de déchiffrement client/chiffrement serveur ;
- la clé d'authentification des paquets qui correspond à un algorithme de type MAC propriétaire permettant d'assurer l'intégrité et l'authenticité des paquets.

Il existe donc deux clés de session symétriques pour l'échange des données chiffrées, ces dernières sont renouvelées après l'envoi de 4096 paquets chiffrés.

2.3.1 Négociation du protocole

Dans un premier temps, le client et le serveur négocient le protocole à utiliser. Lorsqu'un client tente une connexion sur un serveur RDP, il lui spécifie la liste des protocoles de sécurité, des algorithmes et des tailles de clés de chiffrement qu'il est capable de supporter (il s'agit du paquet *MCS Connect Initial PDU*).

Le serveur répond en spécifiant le protocole qu'il a choisi dans le but de sécuriser l'échange. Si le serveur est une machine tournant sous Windows XP, Windows Server 2003 ou inférieur, il s'agira du Standard Security Mode.

2.3.2 Échange de clés

Nous détaillons ci-dessous le mode opératoire utilisé.

Le client et le serveur vont générer chacun, de façon aléatoire, 32 octets, qui vont permettre, une fois réunis, de dériver les trois clés précédemment citées.

Nous appellerons **RS** et **RC** les 32 octets générés respectivement par le serveur et le client.

Pour le serveur, la première étape consiste à envoyer **RS** ainsi qu'une clé publique RSA de 512 bits au client. À noter que cette clé RSA est signée par une autre clé de type RSA également, nous reviendrons sur ce point dans la section suivante.

Une fois en possession des valeurs **RS** et **RC**, le client les dérive pour produire les clés de chiffrement.

L'algorithme de dérivation n'est pas très complexe. Un bloc temporaire de 48 octets est calculé en fonction de **RS**, **RC** et des fonctions de hachage **SHA1** et **MD5**. Les 16 premiers octets de ce bloc temporaire représentent la clé d'authentification des paquets, soit 128 bits.

Le reste de ce bloc temporaire est dérivé à nouveau à l'aide cette fois de **RS**, **RC** et uniquement **MD5**. Il en résulte les deux clés de chiffrement client/serveur, dont la taille est 128 bits.

Il est possible pour le serveur de spécifier des tailles de clés plus petites (40 ou 56 bits) même si celles-ci ne garantissent pas la sécurité des données. Si tel est le cas, une partie des 128 bits sera figée puis ignorée pour obtenir la taille de clé voulue.

Le processus de dérivation peut se schématiser de la façon suivante :

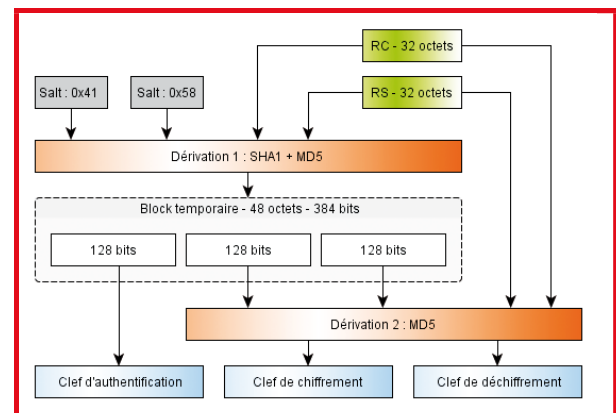


Figure 1 : Processus de dérivation des clés de chiffrement et d'authentification des paquets

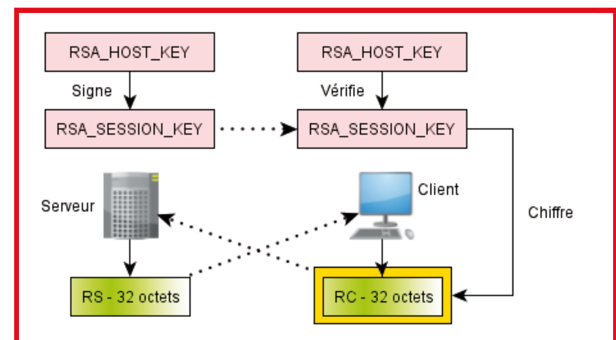


Figure 2 : Signature des valeurs RC et RS

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Le serveur devant lui aussi réaliser cette opération de calcul des clés, le client lui envoie **RC**, chiffré avec la clé RSA 512 bits envoyée précédemment par le serveur. Le serveur, en possession de la clé RSA privée, déchiffre la valeur envoyée par le client et effectue les mêmes calculs que ceux évoqués précédemment.

Le protocole d'échange des valeurs **RC** et **RS** repose sur RSA, ce processus est globalement symétrique : voir Figure 2, page précédente.

2.4 Stockage des clés

Il est également intéressant de noter comment les clés privées **RSA_SESSION_KEY** et **RSA_HOST_KEY** sont stockées sur le serveur RDP.

La clé **RSA_HOST_KEY** est fixée en dur dans la DLL **mstlsapi.dll**, il s'agit d'une première erreur de conception, en effet cette clé privée est commune à toutes les installations de Windows XP et Server 2003. Or c'est elle qui assure l'intégrité de la clé publique **RSA_SESSION_KEY** lorsqu'elle est transmise sur le réseau. Pour anecdote, Microsoft l'a même désormais incluse dans les spécifications officielles du protocole.

Concernant la clé **RSA_SESSION_KEY**, la partie publique est stockée dans un BLOB au niveau du registre dans **HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\TermService\Parameters\Certificate**. Dans ce BLOB figure également la signature de cette même clé générée avec **RSA_HOST_KEY**, ce qui évite de recalculer la signature à chaque initialisation d'une nouvelle connexion.

La clé privée **RSA_SESSION_KEY** est quant à elle stockée sous forme d'un secret LSA **[3]** nommé **L$HYDRAENCKEY**. Ce secret est stocké dans la ruche **SECURITY** du registre sous forme chiffrée avec l'algorithme DES ou AES256 depuis Windows Vista.

La clé **RSA_SESSION_KEY** est régénérée au démarrage du système d'exploitation si la clé de registre n'est pas présente. Nous pouvons également noter que la clé privée est accessible uniquement par un administrateur aussi bien au niveau du registre qu'en mémoire dans le processus **svchost.exe** concerné.

2.5 Man-In-The-Middle

On peut donc directement constater que la sécurité de l'ensemble de la communication dépend de la taille de la clé RSA envoyée par le serveur (**RSA_SESSION_KEY**), mais également de la taille de la clé RSA qui a été utilisée pour la signer (**RSA_HOST_KEY**).

Si l'une des clés RSA est cassée, il est possible pour un attaquant de réaliser une attaque de type *Man-In The-Middle* :

- De manière passive et active si la clé **SESSION** est compromise, cela permettrait de déchiffrer toutes les communications à partir d'un simple fichier **.pcap**, par exemple, ou à la volée.
- De manière active seulement si la clé **HOST** est compromise. Dans ce cas, un attaquant pourrait lors d'une attaque de type ARP *spoofing*, par exemple, substituer la clé **SESSION** par la sienne (il détient évidemment la partie privée) et la signer avec la clé privée **HOST**.

Or les 2 clés utilisées ont une taille de 512 bits, ce qui est bien trop faible de nos jours et comme nous le verrons dans la section 4, il est possible de factoriser un modulus RSA de 512 bits en à peine une semaine avec quelques machines disposant d'une configuration matérielle standard.

Dans la suite de l'article, nous proposons d'étudier l'implémentation du protocole RDP dans le projet open source Xrdp pour Linux. Nous détaillerons alors la mise en œuvre d'une attaque Man-In-The-Middle passive.

3 Le cas d'une implémentation tierce

3.1 Le serveur Xrdp

Le projet open source Xrdp **[4]** permet de partager une session graphique sur un système Linux, il implémente le protocole RDP, notamment le mode classique « Standard RDP security ». L'outil se base sur les bibliothèques **libvnc** ou **librdp** selon la méthodologie utilisée pour manipuler l'espace graphique X11.

Par ailleurs, la bibliothèque **librdp** repose sur la version 4 du protocole, ce qui laisse penser que le serveur est vulnérable à MS02-051 **[5]** (ce qui permettrait d'espionner les frappes clavier), ainsi qu'aux attaques Man-In-The-Middle expliquées précédemment.

Dans la suite de l'article, nous appuierons nos analyses sur la version 0.6.0 du serveur.

3.2 Génération des clés

Avant toute première utilisation, il est nécessaire de générer une clé RSA (la clé **RSA_SESSION_KEY** étudiée précédemment) et la stocker dans le fichier **/etc/xrdp/rsakeys.ini**. C'est le rôle du binaire **keygen** dont les sources correspondent à **\keygen\keygen.c**.

Par défaut, les clés utilisées ont une taille de 512 bits et la valeur ne peut être changée sans modification des sources :

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

```
#define MY_KEY_SIZE 512

[...]

g_writeln("");
g_writeln("Generating %d bit rsa key...", MY_KEY_SIZE);
g_writeln("");
if (error == 0)
{
  error = ssl_gen_key_xrdp1(MY_KEY_SIZE, e_data, e_len, n_data, n_len,
                            d_data, d_len);
```

L'implémentation libre est donc vulnérable également aux attaques Man-In-The-Middle passives et actives, un attaquant pourrait alors déchiffrer intégralement toutes les communications.

3.3 Attaque passive de la session RDP

Afin d'attaquer une session RDP donnée, nous allons supposer que le trafic d'une session a pu être capturé par un *sniffer* réseau dans un fichier au format PCAP.

Afin de calculer les deux clés de chiffrement (client → serveur et serveur → client), nous devons être en possession des données suivantes :

- les valeurs **RS** et **RC** ;
- la clé privée **RSA_SESSION_KEY**.

La valeur **RS** est transmise en clair par le serveur lors de la phase « Basic Settings Exchange » avec le paquet *MCS Connect Response PDU*. Il est donc très facile de la capturer.

Ce paquet contient également la clé publique **RSA_SESSION_KEY** apposée avec sa signature réalisée à partir de **RSA_HOST_KEY**. En utilisant la méthode de factorisation évoquée dans la section 4, nous pouvons récupérer la clé privée en un temps raisonnable.

Il nous reste à obtenir la valeur **RC** qui est envoyée chiffrée sur le réseau après usage de la clé publique **RSA_SESSION_KEY**. Or nous pouvons la factoriser, ce qui permet de déchiffrer la valeur de **RC** qui est transmise par le client lors de la phase « RDP Security Commencement » et l'envoi du paquet *Security Exchange PDU*.

Il reste ensuite à dériver les valeurs obtenues pour obtenir les deux clés de chiffrement souhaitées, mais également la clé d'authentification des paquets que nous n'utiliserons cependant pas par la suite.

Nous avons développé un outil, plus qu'expérimental, qui automatise cette tâche, il suffit de lui fournir en entrée un fichier au format PCAP, ce qui permet de récupérer les clés de chiffrement et déchiffrer les paquets :

```
[+] Server MCS Connect Response PDU found
  -> SC_SECURITY len = 236 bytes
  -> TS_UD_SC_SEC1::encryptionMethod = FLAG_128BIT_ENCRYPTION
  -> TS_UD_SC_SEC1::serverRandomLen = 32 bytes
  -> TS_UD_SC_SEC1::serverCertLen = 184 bytes

  -> Server Random =
000000: 5e 59 d2 e4 80 25 ed dc  79 fe 00 0f 6b 19 b5 6d  ^Y...%..y...k..m
000010: 90 ff 1d 3c 8c cb cc fd  a4 75 0b 91 42 73 b9 b8  ...<.....u..Bs..

  -> Server RSA_SESSION_KEY =
000000: 01 00 00 00 01 00 00 00  01 00 00 00 06 00 5c 00  ..............\.
000010: 52 53 41 31 48 00 00 00  00 02 00 00 3f 00 00 00  RSA1H.......?...
000020: 01 00 01 00 35 dd 21 86  99 25 c7 bf 8e 3e c6 f9  ....5.!..%...>..
000030: e7 c2 8f 6c 37 71 26 cf  e4 52 86 52 12 b2 61 ab  ...l7q&..R.R..a.
000040: 2d 25 83 2a 6e 0c 5c ef  9e be d9 a0 86 ba 58 17  -%.*n.\.......X.
000050: 29 36 aa c9 2f a2 a3 8d  35 f9 6c 5a a8 16 91 80  )6../...5.lZ....
000060: d8 54 d2 e0 00 00 00 00  00 00 00 00 08 00 48 00  .T............H.
000070: 3e ec c5 23 8f 13 fd bb  3a b7 53 1c 6b e4 1d e9  >..#....:.S.k...
000080: a9 6a fe 94 4a 5a c7 3d  f6 64 2d 73 38 4d dd b1  .j..JZ.=.d-s8M..
000090: 72 dd 10 d6 56 b4 2a 41  27 53 fb 50 55 c5 3c 45  r...V.*A'S.PU.<E
0000a0: 2a b6 97 80 2d f5 10 69  fd 67 a1 10 9b 9a 40 31  *...-..i.g....@1
0000b0: 00 00 00 00 00 00 00 00                           ........

[+]Security exchange PDU found
  -> TS_SECURITY_PACKET::length = 72 bytes
  -> Encrypted client Random =
000000: 51 19 aa bf 29 2c 3e d7  45 06 96 25 53 dc 3e 19  Q...),>.E..%S.>.
000010: b2 2f f3 e0 00 4f 03 25  45 6f 01 6c af fc f3 26  ./...O.%Eo.l...&
000020: 52 5f 88 2e c9 89 cb 86  50 84 0a 59 9f 91 95 07  R_......P..Y....
000030: f1 13 ca 3e c5 f5 1e 2d  d9 b5 e2 8b 3f 23 7b 0f  ...>...-....?#{.

[+] Decrypt client random
  -> Client Random =
000000: d1 f4 ae 40 37 8e c7 3f  3e f2 68 cf ce 1d b9 4e  ...@7..?>.h....N
000010: 7a 43 04 49 30 e4 59 c8  36 2b 9e db 8f 85 2c 43  zC.I0.Y.6+....,C

[+] Computing keys
  -> Authent key : fda2ca93dbe6d403b020e4f852cb333d
  -> Encrypt key : 94223724d3b23a5376c7970f54f1443b
  -> Decrypt key : 5a23293387ca55d13b235939a3d33790

[+] Decrypt packet No 1
000000: 0c 04 0c 04 bb 47 03 00  00 00 1c 00 10 00 00 00  .....G..........
000010: 00 00 00 00 41 00 64 00  6d 00 69 00 6e 00 69 00  ....A.d.m.i.n.i.
000020: 73 00 74 00 72 00 61 00  74 00 65 00 75 00 72 00  s.t.r.a.t.e.u.r.
000030: 00 00 77 00 30 00 30 00  74 00 77 00 30 00 30 00  ..w.0.0.t.w.0.0.
000040: 74 00 00 00 00 00 00 00  02 00 1c 00 31 00 39 00  t...........1.9.
000050: 32 00 2e 00 31 00 36 00  38 00 2e 00 32 00 33 00  2...1.6.8...2.3.
000060: 37 00 2e 00 31 00 00 00  40 00 43 00 3a 00 5c 00  7...1...@.C.:.\.
000070: 57 00 69 00 6e 00 64 00  6f 00 77 00 73 00 5c 00  W.i.n.d.o.w.s.\.
000080: 73 00 79 00 73 00 74 00  65 00 6d 00 33 00 32 00  s.y.s.t.e.m.3.2.
000090: 5c 00 6d 00 73 00 74 00  73 00 63 00 61 00 78 00  \.m.s.t.s.c.a.x.
0000a0: 2e 00 64 00 6c 00 6c 00  00 00 c4 ff ff ff 50 00  ..d.l.l.......P.
0000b0: 61 00 72 00 69 00 73 00  2c 00 20 00 4d 00 61 00  a.r.i.s.,. .M.a.
0000c0: 64 00 72 00 69 00 64 00  00 00 00 00 00 00 00 00  d.r.i.d.........
0000d0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000e0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
0000f0: 0a 00 00 00 05 00 03 00  00 00 00 00 00 00 00 00  ................
000100: 00 00 50 00 61 00 72 00  69 00 73 00 2c 00 20 00  ..P.a.r.i.s.,. .
000110: 4d 00 61 00 64 00 72 00  69 00 64 00 20 00 28 00  M.a.d.r.i.d. .(.
000120: 68 00 65 00 75 00 72 00  65 00 20 00 64 00 19 20  h.e.u.r.e. .d..
000130: e9 00 74 00 e9 00 29 00  00 00 00 00 00 00 00 00  ..t...).........
000140: 00 00 00 00 03 00 00 00  05 00 02 00 00 00 00 00  ................
000150: 00 00 c4 ff ff ff 01 00  00 00 07 00 00 00 00 00  ................
000160: 64 00 00 00
```

Ci-dessus, nous observons par exemple en clair les identifiants de l'administrateur dans une structure au début du paquet (Client Info PDU). Les identifiants se retrouvent en effet dans ce paquet si l'utilisateur les a renseignés en utilisant le client classique MSTSC et non lors du *login* interactif une fois la session graphique RDP initialisée. Dans ce dernier cas, il serait nécessaire de reconstruire le mot de passe en exploitant les structures **TS_INPUT_EVENT** envoyées par le client.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

L'application Cain **[6]** est en mesure également d'exploiter la vulnérabilité initialement découverte par Erik Forsberg **[7]**.

4 RSA et factorisation en pratique

4.1 Rappel

Casser un RSA nécessite de factoriser le modulus (N) de la clé publique en un produit de deux facteurs premiers. Il existe plusieurs algorithmes capables de factoriser des grands nombres, mais le plus connu et aussi un des plus efficace est sûrement GNFS (*General Number Field Sieve*). Il est notamment capable de factoriser des nombres composés de plus de 100 chiffres et c'est donc exactement ce dont nous avons besoin ici.

En effet, un RSA 512 bits représente un nombre composé de 155 chiffres, et le casser est aujourd'hui devenu trivial. Le durée du calcul dépend bien entendu de la puissance qui lui est dédiée, mais même avec de modestes moyens, quelques semaines peuvent suffire, tout au plus. À titre d'exemple, il est aujourd'hui possible de factoriser n'importe quel modulus de 512 bits en une semaine avec 4 processeurs Intel Core i7 cadencés à 2.8 Ghz. À noter qu'il s'agit principalement de l'avènement des processeurs à plusieurs cœurs qui a rendu cette performance possible, ainsi 4 processeurs de ce type permettent d'utiliser 4*2*4 = 32 *threads* au total.

4.1.1 L'algorithme GNFS

GNFS est basé sur l'algorithme du crible quadratique **[8]** et le but est d'essayer d'établir une congruence de carrés modulo N (le modulus), qui bien souvent conduit à la factorisation de celui-ci. Le calcul peut être décomposé en trois étapes :

- La sélection polynomiale :
  Il s'agit de la recherche d'un polynôme qui, selon la valeur de ses coefficients, affectera le temps de calcul total de façon non négligeable. Il optimise l'algorithme du crible quadratique en lui permettant d'établir plus de relations. Un bon polynôme est caractérisé par le fait que les coefficients des relations trouvées sont les plus petits possible.

  Chaque polynôme possède ainsi un score de Murphy représentant en fait la probabilité que les relations aient de petits coefficients. Il faut donc choisir le polynôme ayant le score de Murphy le plus élevé afin d'obtenir de bonnes performances. Cette étape est parallélisable en scindant l'intervalle de recherche des polynômes, l'algorithme le plus naïf consistant à le scinder séquentiellement.
- Le *sieving* : Il s'agit de la première étape de l'algorithme du crible quadratique, qui collecte des informations (les relations) qui peuvent conduire à une congruence de carrés. Cette étape est également facilement parallélisable en découpant l'intervalle de recherche en sous-intervalles, séquentiellement par exemple.
- L'algèbre linéaire : C'est la seconde étape de l'algorithme du crible quadratique qui consiste à exploiter les relations trouvées précédemment en les plaçant dans une matrice, pour ensuite la résoudre et trouver une congruence de carrés. Cette étape peut également être parallélisée mais avec un coût de mise en œuvre plus élevé, il est raisonnable d'exécuter cette portion de l'algorithme de manière séquentielle, quelques heures suffisant, par exemple, à retrouver les facteurs d'un nombre de 155 chiffres, ce qui est négligeable face à la durée totale du sieving. Cela nécessite toutefois un espace mémoire assez important.

4.1.2 Les outils

Les implémentations que nous allons utiliser pour réaliser ces trois étapes sont les suivantes :

- Msieve **[9]** : Il s'agit d'un outil open source comprenant une suite d'algorithmes utiles à la factorisation de grands nombres, nous allons l'utiliser pour la sélection polynomiale et l'algèbre linéaire.
- GGNFS **[10]** : Implémentation libre et vieillissante d'outils de factorisation mais possédant une meilleure implémentation de l'algorithme de sieving (*Lattice Siever*).

Les sources de ces outils sont accessibles librement en ligne sur les sites des auteurs respectifs, mais pour des raisons pratiques, des versions pré-compilées pour Windows et pour certains types de processeurs sont disponibles **[11]**.

Par la suite, cet article couvrira donc la manipulation des outils sous Windows uniquement, mais toutes les sources étant disponibles librement, il sera bien entendu possible de la reproduire sur un système d'exploitation différent, ce qui impliquera très certainement des modifications mineures.

Nous recommandons au lecteur de visiter Mersenneforum **[12]** où nous retrouvons une communauté importante d'utilisateurs de ce type d'outils, mais également d'autres espaces liés à des problèmes mathématiques différents, cependant souvent liés à la recherche de nombres premiers.

4.2 Factorisation du modulus

4.2.1 La sélection polynomiale

Nous réalisons cette étape avec l'outil msieve. Une fois téléchargé puis extrait, il faut s'assurer que le nom du binaire principal est bien `msieve.exe`, pour

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

des raisons pratiques que nous verrons par la suite. Nous devons ensuite créer un fichier contenant notre nombre N à factoriser, nous le nommerons **modulus.ini**, pour le donner en entrée à msieve. Voici la commande classique permettant de démarrer la sélection polynomiale :

```
msieve -i modulus.ini -s poly.dat -l poly.log -nf poly.fb -v -np
```

La signification des différents paramètres est la suivante :

- **-i** : Fichier d'entrée contenant le nombre à factoriser.
- **-s** : Fichier de sortie où msieve stockera une liste de polynômes.
- **-l** : Fichier de sortie où msieve écrira ses lignes de log en plus de celles affichées dans la console.
- **-nf** : Fichier de sortie où msieve stockera le meilleur polynôme trouvé à la fin du calcul.
- **-v** : Active le mode verbeux, pour afficher les différentes étapes du calcul.
- **-np** : Indique à msieve de n'effectuer que la sélection polynomiale.

Lorsque la commande précédente est exécutée, msieve détermine un intervalle de calcul total et l'affiche dans les premières lignes du fichier de log (**poly.log** dans notre cas) :

```
searching leading coefficients from 1 to 10784745
```

D'après nos expérimentations, msieve ne va pas parcourir entièrement, mais aléatoirement cet intervalle et prendre le premier polynôme qu'il juge acceptable. Il est vrai que le parcourir entièrement peut s'avérer très long. Cependant, il est possible de forcer msieve à parcourir entièrement un intervalle donné. Cette option permet aussi de paralléliser la sélection polynomiale, et donc de gagner du temps si plusieurs machines sont à notre disposition.

Pour cela, il suffit d'ajouter au paramètre **-np** les bornes de l'intervalle à parcourir. Exemple, en découpant en 4 l'intervalle total :

```
msieve -i modulus.ini -s poly1.dat -l poly1.log -nf poly1.fb -v -np 1,2696186
msieve -i modulus.ini -s poly2.dat -l poly2.log -nf poly2.fb -v -np 2696186,5392372
msieve -i modulus.ini -s poly3.dat -l poly3.log -nf poly3.fb -v -np 5392372,8088558
msieve -i modulus.ini -s poly4.dat -l poly4.log -nf poly4.fb -v -np 8088558,10784745
```

Contrairement au comportement précédent, msieve se force à parcourir entièrement ces intervalles et écrit le meilleur polynôme de chacun d'entre eux dans le fichier **.fb** correspondant. Si on découpe le calcul en X intervalles, on aura X polynômes, chacun écrit dans le fichier **polyX.fb** pour suivre l'exemple donné ci-dessus.

Puisqu'au final, il ne faut qu'un seul polynôme, il va falloir sélectionner le meilleur manuellement, et c'est là que nous allons avoir besoin du score de Murphy expliqué précédemment.

Lorsqu'un intervalle de sélection polynomiale est parcouru, msieve écrit les détails du meilleur polynôme trouvé dans le fichier de log correspondant, par exemple :

```
polynomial selection complete
R0: -6313732992482317107990
R1: 91612524781
A0: -4237649597664914144225379830
A1: -3609918412527578186900
A2: -535903781932150
A3: -1611006400
A4: 828
skew 1123927.49, size 2.160e-012, alpha -5.689, combined =
4.553e-008 rroots = 2
```

Le score de Murphy correspond ici à la valeur **combined**, soit : 4.553e-008.

Il suffit donc de conserver le fichier **polyX.fb** correspondant au polynôme ayant la valeur **combined** la plus élevée. La sélection polynomiale est désormais achevée.

4.2.2 Le sieving

À ce stade, nous avons notre nombre N à factoriser, et le fichier **.fb** produit par msieve contenant un bon polynôme. Nous avons maintenant besoin d'un outil supplémentaire qui est GGNFS afin de réaliser la deuxième étape qu'est le sieving.

Rappelons que la troisième étape expliquée par la suite, l'algèbre linéaire, sera réalisée par l'outil msieve.

Ces deux dernières étapes sont assez complexes à mettre en œuvre manuellement, pour des raisons évidentes :

- Deux outils différents sont utilisés, et effectuer la transition des données de l'un à l'autre peut s'avérer complexe.
- Paralléliser le calcul est indispensable pour factoriser notre nombre dans un temps raisonnable.

Heureusement, Brian Gladman a développé un script en Python, **factmsieve.py** **[13]**, afin de simplifier au maximum toutes ces étapes. Pour le développer, il repose sur le script Perl **factmsieve.pl**, qui à l'origine était le script officiel permettant de piloter GGNFS.

Voici les différentes fonctionnalités qui seront traitées par le script :

- conversion de fichier de polynôme **.fb** au format GNFS **.poly** ;
- gestion de la puissance de calcul en fonction du processeur ;
- gestion du calcul parallélisé avec X clients ;
- compression des fichiers de relations pour gagner de l'espace disque.

Le script doit ensuite être configuré, il s'agit principalement de définir les chemins des binaires msieve et ceux de la suite ggnfs ainsi que la configuration des *cores* du processeur :

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

```
# Chemins des binaires msieve et ggnfs
GGNFS_PATH = 'C:\\Users\\USERNAME\\Desktop\\MISC\\ggnfs\\'
MSIEVE_PATH = 'C:\\Users\\USERNAME\\Desktop\\MISC\\msieve\\'

# Configuration du processeur pour le calcul parallélise
# Exemple avec utilisation d'un Core i7 avec 4 cores et 2 threads par core
NUM_CORES = 4
THREADS_PER_CORE = 2
```

Il convient ici de renseigner les détails fournis dans les spécifications Intel du processeur concerné.

Une fois les modifications effectuées, on crée un nouveau fichier, appelé **modulus.n**, dans lequel on va y stocker notre nombre N à factoriser avec la syntaxe suivante :

```
n: MODULUS
```

MODULUS est le nombre N à factoriser en base 10.

Puis nous ajoutons à ce même répertoire le fichier **.fb** contenant notre polynôme, nommons le **modulus.fb**.

Désormais, nous avons un répertoire contenant les fichiers suivants :

- **factmsieve.py** : script Python effectuant les deux dernières étapes de l'algorithme ;
- **modulus.n** : fichier contenant le nombre à factoriser avec la syntaxe GGNFS ;
- **modulus.fb** : fichier contenant le résultat de la sélection polynomiale.

Il faut maintenant déterminer le nombre de machines qui participeront au calcul.

Le client N°1 est la machine « maître », c'est elle qui recevra les fichiers de relations trouvées par les autres et produira le résultat de la factorisation.

Pour chaque machine, il faut bien entendu répéter les étapes de configuration du script Python avec des paramètres qui lui sont propres.

Tout est prêt pour lancer les calculs, nous prendrons par la suite 5 clients pour illustrer le principe de la parallélisation.

Voici les commandes à exécuter sur chaque client :

```
Client No 1: #python factmsieve.py modulus 1 5
Client No 2: #python factmsieve.py modulus 2 5
...
Client No 5: #python factmsieve.py modulus 5 5
```

Le script exécute GGNFS pour effectuer le sieving, c'est-à-dire chercher des relations en utilisant le polynôme que nous avons trouvé. Un nombre minimum de relations est estimé et une fois atteint, le script appelle automatiquement l'outil msieve pour vérifier que le nombre de relations est suffisant. Si ce n'est pas le cas, la phase de sieving sera automatiquement reprise.

Pour le client maître, les fichiers produits par GGNFS seront automatiquement compressés en gzip par le script Python.

En revanche, pour les autres clients, les relations trouvées seront accumulées dans un fichier. Par exemple, pour le client N, le fichier **spairs.add.N** sera créé. On pourra à tout moment déplacer ce fichier sur le client N°1, dans le répertoire où se trouve le script Python. Celui-ci le chargera automatiquement afin d'ajouter à son calcul les relations trouvées par le client N à la prochaine vague de sieving.

4.2.3 L'algèbre linéaire

Lorsque le nombre de relations sera suffisant, msieve démarrera la dernière étape qui est l'algèbre linéaire.

Les fichiers contenant les relations représentent des gigaoctets de données, d'où l'intérêt, certes limité au vu de la taille des disques durs actuels, de bénéficier de la compression GZip.

Le principe de l'algèbre linéaire est, rappelons-le, de placer ces relations dans une matrice pour ensuite la résoudre et trouver une congruence de carrés menant à la factorisation du modulus.

Ces données seront donc placées en mémoire pendant le calcul, ce qui implique qu'il faudra assez de RAM disponible (quelques gigaoctets pour un nombre de 150 chiffres) pour passer cette étape.

Lorsque l'algèbre linéaire est terminé, le résultat de la factorisation est stocké dans le même répertoire que le script :

```
-> Factorization summary written to g155-modulus.txt
```

Conclusion

Au travers de cet article, nous avons illustré le fait que la cryptographie n'est pas toujours une fin en soi et le simple fait d'en utiliser des primitives n'implique pas nécessairement un gain en sécurité. Le développement d'un protocole sécurisé n'est pas une tâche facile lorsqu'on part de rien, cela explique certainement le choix de Microsoft d'utiliser aujourd'hui le protocole TLS qui est plus sûr et qui a déjà subit de nombreuses analyses de sécurité.

Pour les utilisateurs soucieux de se protéger, nous conseillons l'usage exclusif de « Enhanced Security mode » et des certificats. Cela renvoie cependant les administrateurs et les décideurs à des contraintes opérationnelles parfois difficiles à mettre en œuvre : utiliser une PKI, distribuer les certificats, ... ■

REMERCIEMENTS

Nous tenons à remercier toute l'équipe Quarkslab, ses accros à l'opéra et ses pongistes les plus fourbes :)

Les références de cet article sont disponibles sur : http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6/6

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Complétez votre collection des anciens numéros de...

Les 4 façons de commander !

Par courrier
En nous renvoyant ce bon de commande.

Par téléphone
Entre 9h-12h & 14h-18h au 03 67 10 00 20 (paiement C.B.)

Par le Web
Sur notre site : www.ed-diamond.com.

Par fax
Au 03 67 10 00 21 (C.B. et/ou bon de commande administratif)

MISC

N° 52 — du numéro... ...au numéro — N° 57, N° 58, N° 59, N° 60, N° 61, N° 62

Bon de commande MISC

Réf.	Désignation	Prix / N^{os}
MISC N°52	4 Outils indispensables pour tester votre sécurité !	8,00 €
MISC N°53	La sécurité du wi-fi, des paroles en l'air ?	8,00 €
MISC N°54	Anonymat sur internet : Risque ou nécessité ?	8,00 €
MISC N°55	Au coeur des technologies sécurité de MICROSOFT	8,00 €
MISC N°56	Forensics : Les nouveaux enjeux	8,00 €
MISC N°57	Sexe, Drogue et sécurité informatique	8,00 €
MISC N°58	Sécurité et architecture PC : L'impossible confiance ?	8,00 €
MISC N°59	Ingénierie sociale sur internet	8,00 €
MISC N°60	Cloud Computing et sécurité : Une difficile cohabitation ?	8,50 €
MISC N°61	Sécurité des bases de données	8,50 €
MISC N°62	Sécurité des applications sous Linux	8,50 €

HS N° 4 — du numéro... ...au numéro — HS N° 5

Bon de commande MISC Hors-Série

Réf.	Désignation	Prix / Nos
MISC Hors-Série N°4	À l'assaut du WEB - Nouveaux usages ... Nouveaux enjeux	8,00 €
MISC Hors-Série N°5	Cryptographie : vos secrets sont-ils bien gardés ?	9,00 €

Bon de commande

à remplir (ou photocopier) et à retourner aux Éditions Diamond – MISC – BP 20142 – 67603 Sélestat Cedex

	Référence	Prix / N°	Qté	Total
EXEMPLE :	MISC N°61	8,50 €	1	8,50 €
		TOTAL :		
		FRAIS DE PORT FRANCE MÉTRO. :		+3,9 €
		FRAIS DE PORT HORS FRANCE MÉTRO. :		+6 €
		TOTAL :		

Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Téléphone :

e-mail :

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond
- ❑ Carte bancaire n° ⎿⎿⎿⎿ ⎿⎿⎿⎿ ⎿⎿⎿⎿ ⎿⎿⎿⎿

Expire le : ⎿⎿⎿⎿

Cryptogramme visuel : ⎿⎿⎿

Votre cryptogramme visuel

Date et signature obligatoire

- ❑ Je souhaite recevoir des infos des Edtions Diamond
- ❑ Je souhaite recevoir des infos des partenaires des Editions Diamond

Retrouvez les sommaires et commandez tous nos magazines sur notre site :
http://www.ed-diamond.com

MISC Multi-System & Internet Security Cookbook

e document est la propriété exclusive deJohann Locatelli (johann@g… .com) le 1 Mars 2015

Complétez votre collection de

au tarif promotionnel de 4 €TTC par numéro* !

Les 4 façons de commander !

Par courrier
En nous renvoyant ce bon de commande.

Par téléphone
Entre 9h-12h & 14h-18h au 03 67 10 00 20 (paiement C.B.)

Par le Web
Sur notre site : www.ed-diamond.com.

Par fax
Au 03 67 10 00 21 (C.B. et/ou bon de commande administratif)

Vous recherchez un numéro spécifique ?

Rendez-vous sur www.ed-diamond.com pour consulter le sommaire détaillé de chaque magazine !

Choisissez vos numéros dans le tableau ci-dessous :

- ❑ N°1 Les vulnérabilités du Web !
- ❑ N°2 Windows et la sécurité
- ❑ N°4 Internet un château construit sur du sable? ...ou les protocoles réseaux en question
- ❑ N°6 Sécurité du wireless ?
- ❑ N°7 La guerre de l'information - évaluation, riques, enjeux
- ❑ N°8 Honeypots - Le piège à pirate !
- ❑ N°9 Que faire après une intrusion ?
- ❑ N°10 VPN - Virtual Private Networt - Créez votre réseau sécurisé sur internet
- ❑ N°12 La faille venait du logiciel
- ❑ N°14 Reverse Engineering - Retour au sources
- ❑ N°16 Télécoms - Les risques des infrastructures
- ❑ N°17 Comment lutter contre - Le spam, les malwares, les spywares ?
- ❑ N°18 Dissimulation d'information
- ❑ N°19 Les Dénis de Services - La menace rôd
- ❑ N°20 Cryptographie malicieuse : quand les vers et virus se mettent à la crypto
- ❑ N°21 Limites de la sécurité
- ❑ N°23 De la recherche de faille à l'exploit
- ❑ N°25 Bluetooth, P2P, Messageries instantanées : Les nouvelles cibles
- ❑ N°26 Matériel, mémoire, humain, multimédia : Attaques tous azimuts
- ❑ N°28 Exploits et correctifs : Les nouvelles protections à l'épreuve du feu
- ❑ N°29 Sécurité du coeur de réseau IP : un organe critique
- ❑ N°30 Les protections logicielles
- ❑ N°32 Que penser de la sécurité selon Microsoft ?
- ❑ N°33 RFID - Instrument de sécurité ou de surveillance ?
- ❑ N°34 Noyau et rootkit
- ❑ N°36 Lutte informatique offensive - Les attaques ciblées
- ❑ N°37 Déni de service
- ❑ N°38 Code malicieux - Quoi de neuf ?
- ❑ N°39 Fuzzing - Injectez des données et trouvez les failles cachées
- ❑ N°40 Sécurité des réseaux - Les nouveaux enjeux
- ❑ N°41 LA CYBERCRIMINALITÉ ...ou quand le net se met au crime organisé
- ❑ N°42 LA VIRTUALISATION : Vecteur de vulnérabilité ou de sécurité ?
- ❑ N°43 La sécurité des web services
- ❑ N°44 Compromissions électromagnétiques
- ❑ N°45 La sécurité de Java en question
- ❑ N°46 Construisez et validez votre sécurité
- ❑ N°47 La lutte antivirale, une cause perdue ?
- ❑ N°48 Comment se protéger contre la peste spam ?
- ❑ N°49 Vulnérabilités Web et XSS - Des ennemis que vous sous-estimez !
- ❑ N°50 La sécurité des jeux
- ❑ N°51 Sécurité des OS mobiles

Numéros MISC épuisés :
N°3, N°5, N°9, N°11, N°13 à N°15, N°22, N°24, N°27, N°31, N°35 et N°41

*dans la limite des stocks disponibles.

Numéros MISC Hors-Série épuisés : HS N° 1 à 3

Bon de commande

à remplir (ou photocopier) et à retourner aux Éditions Diamond – MISC – BP 20142 – 67603 Sélestat Cedex

Quantité	Prix / N°	Total
	x 4,00 €	=
	FRAIS DE PORT FRANCE MÉTRO. :	+ 3,90 €
	FRAIS DE PORT HORS FRANCE MÉTRO. :	NOUS CONSULTER
	TOTAL :	

Je choisis de régler par :

- ❑ Chèque bancaire ou postal à l'ordre des Éditions Diamond
- ❑ Carte bancaire n° |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expire le : |_|_|_|_|

Cryptogramme visuel : |_|_|_|

Date et signature obligatoire

Voici mes coordonnées postales :

Nom :

Prénom :

Adresse :

Code Postal :

Ville :

Téléphone :

e-mail :

- ❑ Je souhaite recevoir des infos des Édtions Diamond
- ❑ Je souhaite recevoir des infos des partenaires des Éditions Diamond

Ce document est la propriété exclusive de Johann … lli (johann@gykroipa.com) le 1 Mars 2015

MÉCANISME DE CONTRÔLE D'AUTHENTICITÉ

DES PHOTOGRAPHIES NUMÉRIQUES DANS LES REFLEX CANON

Laurent Clévy - lclevy@free.fr

mots-clés : PHOTOGRAPHIE NUMÉRIQUE / AUTHENTICITÉ DES DONNÉES / TIFF

Comment prouver qu'une photographie numérique n'a pas été modifiée avant publication ? En ajoutant des données d'authenticité à l'image, fonction disponible depuis plusieurs années chez Canon et Nikon. Dmitry Sklyarov de la société ElcomSoft a décrit en novembre 2010 [1] le principe général utilisé par les boîtiers de la marque rouge, mais sans livrer suffisamment de détails pour recréer ces données, appelées ici « Original Decision Data », ou ODD. Cet article se propose de décrire comment trouver les secrets de son appareil (sans fer à souder) et comment recalculer ces données. Le constructeur a depuis janvier 2011 supprimé cette fonctionnalité, sans doute en réaction à la présentation précédemment citée, et émis un avis produit [2]. Les pages suivantes se veulent didactiques et en prolongement de la présentation d'ElcomSoft et ne sauraient en aucun cas la remplacer. Elles décrivent des expérimentations faites début 2010, puis achevées en novembre 2010, grâce à la pièce manquante du puzzle...

1 Le format RAW/CR2 et l'usage du tag ODD

Les photographies numériques sont écrites sur la carte mémoire en format RAW (basé sur le format TIFF) ou au format Jpeg. Malgré les efforts de Dave Coffin avec Dcraw **[3]**, Phil Harvey avec ExifTool **[4]** et de votre serviteur **[5]** pour le format CR2 de Canon, une partie des t*ags* TIFF (ou Jpeg) ne sont pas documentés : vous ne maîtrisez pas complètement de quelle manière sont archivés les originaux de vos souvenirs de vacances. En l'occurrence, c'est le cas du tag « Original Decision Data », qui porte l'identifiant unique 0x0083 (131 en décimal). Lorsque cette option est activée, via le menu, l'appareil ajoute ce « sceau » à chaque image. L'authenticité des images peut ensuite être vérifiée grâce à un kit vendu par Canon **[6]**. Il est même possible de chiffrer les images sur les appareils haut de gamme uniquement, en insérant dans le boîtier une carte Secure Mobile dédiée, mais l'algorithme sous-jacent n'est pas décrit à ce jour. Les images au format Jpeg contiennent aussi des métadonnées, et peuvent également contenir un tag ODD, de format quasi-identique.

2 Que peut-on apprendre par observation du tag ODD ?

Pour commencer, analysons le contenu du tag ODD pour le modèle 20D, car son format (version 2) est plus simple que sur les appareils récents (version 3). Il ne s'agit pas dans cette première partie d'expliquer complètement le mécanisme de contrôle d'authenticité (*Message Authentication Code*, MAC dans la langue de Benny Hill), mais d'offrir un simple aperçu.

ExifTool nous donne l'*offset* du tag dans le fichier image :

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

```
$ exiftool.exe -H -U -OriginalDecisionDataOffset 20d/IMG_0102_20d.CR2
0x0083 Original Decision Data Offset    : 7259830
```

Voici donc les données stockées à l'offset 7259830, (0x6ec6b6 en hexadécimal), quelque peu mises en forme. Les lignes sont numérotées pour faciliter le commentaire. Se reporter à l'encart pour appréhender le format TIFF/CR2 et comment localiser les données ODD dans un fichier image issu du 20D.

```
1:ffffffff 02000000
2:803f025b 99f65111 cc76f6fe c5180bc8 8fe0a342
3:04000000
4:00000000 ca340900 ec916500 f2980ecd fb706dad 20e04cb2 1620374c 061f2325
5:01000000 00000000 6e000000 4925704c a36ccdef 55a6f7ac 3c1038ca 5ff91ada
6:02000000 72000000 08030000 de69a0d4 b6a2fb99 937599cc 82d30d21 284b975d
7:03000000 7e030000 4c310900 4c428724 cbf0810c e516462f a06aa0a2 8b1e9527
```

Les lignes 2, 4, 5, 6 et 7 semblent contenir chacune une empreinte SHA1 (les 20 derniers octets, à grande entropie), ou plutôt des données de contrôle d'authenticité HMAC-SHA1 **[7]** puisque le rôle de ces données est d'être capable d'authentifier les données images. Les 4 dernières lignes sont au même format, commençant par un entier sur 32 bits (long) au format Intel (**http://en.wikipedia.org/wiki/Endianness**), sans doute le numéro de la « section ». La ligne 3 dénombre ces enregistrements identiques, toujours en *Little Endian*.

Revenons au format TIFF : l'offset des données de l'image principale et leur taille sont données par :

```
exiftool.exe  -v  20d/IMG_0102_20d.CR2
[...]
  + [IFD3 directory with 5 entries]
  | 0)  Compression = 6
  | 1)  StripOffsets = 603338
  | 2)  StripByteCounts = 6656492
[...]
```

respectivement 603338 (0x934ca) et 6656492 (0x6591ec). On retrouve (au format Intel) ces 2 valeurs dans le Tag ODD à la ligne 4, colonnes 2 et 3. La colonne 2 contient donc l'offset vers l'image principale et la 3e colonne la taille de cette « section ». Enfin, on peut penser que les supposés HMAC-SHA1 sont calculés à chaque fois à partir de l'offset (colonne 2) et pour la taille indiquée en colonne 3.

Pourquoi donc avoir besoin de plusieurs HMAC au lieu d'un unique code d'authenticité pour le fichier ? C'est pour authentifier également les métadonnées. Même si le tag est à la fin du fichier (0x934ca + 0x6591ec = 0x6ec6b6), il faut respecter le format TIFF et écrire l'offset vers les données du tag ODD dans le répertoire TIFF, précisément entre les offsets 0x37a (ligne 6, 0x308+0x72) et 0x37e (ligne 7). La dernière zone exclue du calcul de la signature correspond à la valeur du tag « Orientation » (paysage ou portrait), que l'utilisateur peut modifier ultérieurement à l'enregistrement de l'image sur la carte mémoire.

En résumé, une empreinte garantissant l'intégrité est calculée sur chacune des 4 régions du fichier TIFF. La ligne 2 ne décrivant pas de région, elle concerne certainement le fichier dans son ensemble, c'est-à-dire, on peut le supposer, en agrégeant les quatre HMAC-SHA1 des lignes 4 à 7. Tout ceci est illustré dans la figure 1.

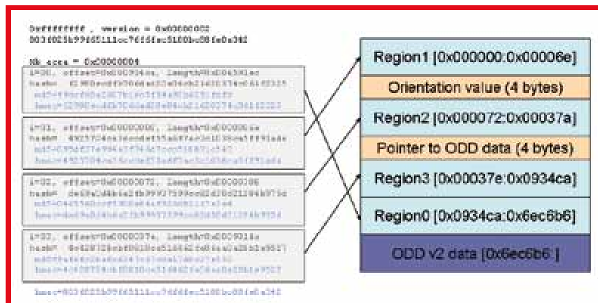


Figure 1

Il n'est malheureusement pas possible de comprendre la totalité de l'algorithme sans analyser le logiciel interne (*firmware*) de l'appareil photo. C'est ce qu'a fait Dmitry Sklyarov : reportez-vous à sa présentation **[1]** pour les détails, à l'exception toutefois des clés HMAC. Pour la version 2 du tag ODD, une clé unique est utilisée pour tous les boîtiers d'un même modèle. Ceci n'est plus vrai pour la version 3 du tag : chaque clé HMAC est unique par boîtier. J'ai pu reproduire les résultats décrits dans cette première partie et dans **[1]**, en disposant de la clé utilisée par les boîtiers 20D, voir les valeurs calculées qui sont en bleu dans la figure 1.

Voyons donc dans la suite de cet article comment retrouver l'algorithme complet pour la version 3 du tag ODD, et surtout récupérer cette clé secrète et unique par boîtier, comme avec le 60D.

3 Exécutons, traçons du code sur notre appareil photo

Depuis 2009, grâce aux bidouilleurs du forum CHDK **[8]** et autour du firmware alternatif Magic Lantern **[9]**, il est possible d'exécuter du code sur les reflex numériques Canon. On peut « sauvegarder » la mémoire vive et non volatile (Flash) sur la carte mémoire. Le processeur interne est le DIGIC, incluant un cœur ARM. Le système d'exploitation temps-réel de Canon s'appelle DryOS **[10]**.

L'analyse statique du firmware avec FindCrypt **[11]** ou SignSrch **[12]** révèle entre autres des constantes liées aux implémentations SHA1, HMAC-SHA1, MD5 et SHA256. Les suppositions faites précédemment se renforcent...

Des fonctions de débogage sont même activables **[13]**. Ci-dessous, nous pouvons voir un extrait du journal de démarrage du 60D :

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

```
Sat Feb 19 20:25:15 2011
     0:    11.679 [STARTUP]
K287 ICU Firmware Version 1.0.8 ( 3.3.1 )
     1:    11.741 [STARTUP]
ICU Release DateTime 2010.11.08 08:40:51
[...]
     4:    13.402 [PTPCOM] Magic Lantern
     5:    13.420 [PTPCOM] Built on 2010-12-23 20:54:26 by user@
ubuntu1004desktop
```

avec des traces pour les fonctions « MAC », pour Message Authentification Code :

```
   794:   719.845 [MAC] MAC_Initialize
[...]
   799:   720.641 [MAC] Key=0x4 Board=0x820fd801 Body=0x22970ba2
```

Les trois valeurs en ligne 799 sont respectivement **`KeyId`**, **`BoardId`** et **`BodyId`**, nous y reviendrons. Cette dernière valeur est également appelée **`SerialNumber`** par ExifTool, et elle est imprimée sous le boîtier de l'appareil.

Enfin, lorsqu'une photo est prise avec l'option « Original Decision Data » activée via le menu :

```
  8660: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x60000, 0x10000000 OffLen:0xb566,
  8661: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x90000, 0x0 OffLen:0xe39b, 0x1)
  8662: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x70000, 0x20000000 OffLen:0xe39e,
  8663: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x90000, 0x0 OffLen:0x1b61e9, 0x3)
  8664: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x80000, 0x1000000 OffLen:0x1b61ec
  8665: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x10000, 0x10000 OffLen:0x4e63fa,
  8666: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x30000, 0x0 OffLen:0x6e, 0x4)
  8667: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x40000, 0x0 OffLen:0xb2f6, 0x108)
  8668: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x50000, 0x0 OffLen:0x5c0, 0x4)
  8669: [MAC] macSetArea(Ref:0xffffffff, 0x0 Set:0x0, 0x0 OffLen:0xcc8, 0x260)
```

dans le log apparaît un calcul de *digest* sur 32*8 = 256 bits. Il s'agit peut être d'un calcul SHA256, une amélioration de SHA1, dont les constantes ont été détectées dans le firmware et qui produit une valeur sur 256 bits.

```
  8686: 34693.045 [ENGP] [HASH] KaiserCompleteCBR
  8687: 34693.085 [ENGP] [HASH] CrawReadCompleteCBR
  8688: 34693.162 [ENGP] [HASH] digestA 0xa48a690b
  8689: 34693.187 [ENGP] [HASH] digestB 0x8b3753f5
  8690: 34693.210 [ENGP] [HASH] digestC 0xfe6bb8c2
  8691: 34693.227 [ENGP] [HASH] digestD 0x18a08b9
  8692: 34693.249 [ENGP] [HASH] digestE 0x20e4919c
  8693: 34693.271 [ENGP] [HASH] digestF 0x26c42332
  8694: 34693.293 [ENGP] [HASH] digestG 0x2e0440aa
  8695: 34693.315 [ENGP] [HASH] digestH 0x606beefc
```

En analysant les chaînes de caractères du firmware, on trouve également ceci (la première colonne est l'offset dans le firmware) :

```
  91604 MAC_SetArea RAW [THM] [%#x]
  91644 MAC_SetArea RAW [THM Padding] [%#x]
  91668 MAC_SetArea : RAW [DISP] [%#x]
  91aa4 MAC_SetArea : RAW [TWAIN] [%#x]
  91aec MAC_SetArea : RAW [MAIN] [%#x]
  91b30 MAC_SetArea : RAW [ORIENT] [%#x]
  91b58 MAC_SetArea : RAW [USER] [%#x]
  91b78 MAC_SetArea : RAW [FLASH] [%#x]
  91b9c MAC_SetArea : RAW [MAKER] [%#x]
```

4 Reconstitution des données du tag ODD, version 3

Comme précédemment, observons le contenu du tag ODD (version 3), produit par notre 60D :

```
01: ffffffff 03000000
02: 14000000 fd86c9ed c422e369 14a68036 ba9825e8 40064c9b
03: 14000000 ac0fd27d 41e00cbb 76800050 0231e8df c9b6c6cf
04: 28020000 04000000 089b5a74 03000000 991dce01 03000000 04000000 01d80f82
rand=320ef068
05:  08000000
06: [01000000 04000000 *6881ee0b
07:  14000000 63569076 b9283b0e 64c5a7a7 042c1f42 871d0fc9 01000000 fa634e00 9db97f01]
08: [02000000 04000000 *0dfc84d5
09:  14000000 026db83c b6181d6b bbf77ef9 5002282b 4d26ef11
10:   09000000
11:    00000000 6e000000, 72000000 c6050000, 3c060000 04070000
12:    a00f0000 cca40000, 74b50000 02220000, e80d0100 0e000000
13:    f6e33300 02000000, f8634e00 02000000, 971dce01 02000000]
14: [03000000 04000000 *6b252675
15:  14000000 23a93a3a a7d3a8bb e3753ecf bc571166 e196ba0e 01000000 6e000000 04000000]
16: [04000000 04000000 *e394364d
17:  14000000 0a245bdd 4c366931 cb08e16e da749e4d dc1ea6a0 01000000 6cb40000 08010000]
18: [05000000 04000000 *fef182b5
19:  14000000 03a013ab 309475c2 10d2aa9f 0a3d5641 209f2ee3 01000000 38060000 04000000]
20: [06000000 04000000 *f4fb870e
21:  14000000 aa374cb9 8d24bbae 9e004fa4 62ded429 7a6f5e63 01000000 76d70000 72360000]
22: [07000000 04000000 *fba7fe65
23:  14000000 954a9ee5 009dcfe9 e8d1872d 514baf05 fb2e6491 01000000 f60d0100 00d63200]
24: [08000000 04000000 *51c5f451
25:  14000000 cbd2fef0 fbd60871 e31c9812 e2ba5b71 308b357c 01000000 f8e33300 00801a00]
[...]
```

Il ne faut pas chercher dans un premier temps à tout comprendre. Nous retrouvons cette fois-ci 8 régions (entre crochets) avec deux différences : une valeur qui semble aléatoire sur 32 bits (précédée d'un ***** pour la lecture) et la région est définie (offset, puis taille) sur les deux dernières colonnes. Autres différences, les lignes 2 et 3 possèdent chacune un HMAC-SHA1 sans région ni aléa, enfin, la région 2 semble découpée en neuf sous-parties et utilise un seul aléa.

ExifTool nous indique que l'image principale est à l'offset 5137400 (0x4e63f8) et d'une taille de 25147809 (0x17fb9a1). Cela est cohérent avec les deux dernières valeurs de la région 1 (la différence de 4 octets est à négliger).

Sans rentrer dans les détails, les huit régions sont appelées *Area* dans le firmware et calculées par la fonction **`MAC_SetArea()`**, se trouvant à partir de l'offset 0x91600 dans le firmware. Nous avons vu précédemment les messages générés dans le journal par cette fonction. Voici maintenant l'algorithme à appliquer.

5 L'algorithme de signature, pour les boîtiers récents

L'algorithme suivant est utilisé pour la version 3 du tag et pour un sous-type vhash>1. Pour le 60D, vhash=3,

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

il s'agit de la sixième valeur, ligne 4. Curieusement, l'aléa associé à chaque région n'est pas utilisé ici.

```
file_sha256 = sha256.new()
Pour chaque région:
   Si region2:
      region2_sha256 = sha256.new()
      pour chaque sous-partie:
         region2_sha256 = sha256.update( region )
   sinon:
      region_sha256 = sha256.new( region ).digest()
   signature = hmac-sha1(region_sha256.digest(), hmac_key) //lignes
7,9,15,17,19,21,23,25
   file_sha256 = sha256.update(region_sha256) // sha256 accumulation
odd_sha256 = sha256(tag_odd).digest()
odd_sign = hmac-sha1(odd_sha256, hmac_key)   // hmac de la ligne 3
file_sha256 = sha256.update(odd_sha256)
file_sign =  hmac-sha1(file_sha256.digest(), hmac_key) // hmac de la ligne 2
```

En résumé, un SHA256 puis HMAC-SHA1 sur chaque région et le contenu du tag ODD (en ignorant les 56 premiers octets, sinon il y a un problème de poule et d'œuf). Enfin un HMAC-SHA1 sur l'accumulation des SHA256 calculés aux étapes précédentes.

Comme il faut pouvoir calculer le tag ODD de chaque image, à raison de 6,3 d'entre elles par seconde, le calcul du SHA256 est effectué par accélération matérielle, nommée « Kaiser », et intégré au Digic. Pour le boîtier 450D, l'aléa par région est bien utilisé (sous-type vhash=1), et l'algorithme est basé sur MD5 au lieu de SHA256, mais je n'en connais pas les détails. L'algorithme ci-dessus a été vérifié également sur un 7D (vhash=3) et un 50D (vhash=2).

6 Calcul de hmac_key

Cette clé est calculée comme suit, dans la fonction **____MAC_Start()** :

```
v1 = IKBoardID | BodyId | Hmac_rand
```

Soit la concaténation de 3 valeurs :

- **IKBoardID**, une clé unique au boîtier de 32 octets, générée ou injectée en usine.
- Un identifiant public du boîtier (**BodyId**), ici 0x22970ba2. Nous l'avons aperçu dans le log système, et il est disponible dans le tag TIFF 0x000c du *MakerNote* (appelé **SerialNumber** par ExifTool).
- Et l'aléa **hmac_rand**, dans notre exemple, la valeur est 0x68f00e32, ligne 4 et générée par la fonction **Rand()** de la bibliothèque C, appelée 2 fois successivement sur 16 bits.

```
Hmac_key = sha1sums(v1)
```

avec

```
def sha1sums(m):
  sh = sha1()
  sh.update(m)
  sh.update('\x01')
  r1 = sh.digest()
  sh = sha1()
  sh.update(m)
  sh.update('\x02')
  r2 = sh.digest()
  return r1+r2[0:12]     // les 8 derniers octets de r2 sont ignorés
```

Il y a deux manières de récupérer la valeur unique **IKBoardID**, en mémoire vive ou via une version stockée de manière persistante.

Au passage, le slide 23 de M. Sklyarov **[I]** intervertit **BodyID** et **Hmac_rand**, ce qui est incorrect.

7 Récupération du secret boîtier IKBoardId

7.1 En mémoire vive

La fonction **macdispinf()**, pour « MAC display info », utilise une structure qui contient des valeurs intéressantes :

```
        LDR   R5, =0x5AE8       ; 60D version 1.0.8
        LDR   R4, [R0]
        LDR   R0, [R5,#8]       ; R0 = adresse de la structure
        LDR   R1, [R0,#8]       ;  BoardID
        ADR   R0, aBoardid0x08xD ; "BoardID=0x%08x(%d)\n"
        MOV   R2, R1
        BL    printf
...
        ADR   R0, aIkboardid  ; "IKBoardID ="
        BL    printf
        MOV   R4, #0
 loop   LDR   R0, [R5,#8]
        ADD   R0, R0, R4
        LDRB  R1, [R0,#0x2C] ;boucle affichant 32 octets depuis l'offset 0x2c
        ADR   R0, a02x_0      ; "%02x "
        BL    printf
        ADD   R4, R4, #1
        CMP   R4, #0x20
        BCC   loop
        ADR   R0, asc_FF1A5BC8 ; "\n"
        BL    printf_maybe
```

À l'offset 8 se trouve **BoardID** et à l'offset 0x2c (44 en décimal) notre **IKBoardID** sur 32 octets. Canon a dû estimer que le risque de récupération en mémoire vive était faible, et celui en mémoire persistante plus probable, d'où une meilleure protection, que nous allons détailler.

7.2 Stockée dans la mémoire flash

Les « propriétés » de DryOS sont des structures nommées, permettant de stocker en mémoire et sur la mémoire flash (ici, à l'adresse 0xff800000) la plupart des données applicatives, comme la configuration de l'appareil. On retrouve **BoardID** via la propriété 0x1060002 (stockée en Little Endian), et **FKBoardID** dispersée dans les propriétés 0x01060004, 0x01060000, 0x0100000B et 0x10000008.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

FKBoardID (F pour *Forward*) est une version obfusquée de **IKBoardID** (I pour *Inverse*). Il faut appliquer la fonction **CRP_IObfunc()** du firmware pour retrouver notre valeur en clair : en clair il faut inverser obfuscation.

```
$ python grep_odd.py 60d/FF800000.BIN
0x01000006,  4 = 0x2d0078,  a20b9722 (BodyID)
0x01000008, 16 = 0x2d0030,  f5d7799d9a35575e6dc479294bdc3f79 (FKBoardID_16)
0x0100000b,  8 = 0x2d00ac,  17da42cdaffbefcb (FKBoardID_8)
0x01060000,  4 = 0x52f6cc,  62bcf06c (FKBoardID_4)
0x01060001,  4 = 0x52f6d8,  04000000 (KeyID)
0x01060002,  4 = 0x52f6e4,  01d80f82 (BoardID)
0x01060004,  4 = 0x52f718,  4f492fe6 (FKBoardID_0)
FKBoardID=4f492fe662bcf06c17da42cdaffbefcbf5d7799d9a35575e6dc479294bdc3f79

$ python iobfunc.py
4f492fe662bcf06c17da42cdaffbefcbf5d7799d9a35575e6dc479294bdc3f79
ce5969180f38bbab2a28f08b44b536dc407cd9ee54843226505ec35a[820fd801]
```

Le script **grep_odd.py** parcourt la zone de la mémoire flash où sont stockées les propriétés, à la recherche des propriétés qui nous intéressent. Le format de stockage d'une propriété en mémoire est : **property_length+8** (32 bits), **property_id** (32 bits), **property_value**. Tout ceci est toujours stocké au format Intel.

En sortie, la première colonne est l'identifiant de la propriété, le 2e sa longueur, l'offset dans le *dump*, et enfin, sa valeur, octet par octet. On peut noter que l'on retrouve la valeur **BoardID** dans les 4 derniers octets de **IKBoardID**, ce qui confirme que cette dernière est la version en clair de **FKBoardID**.

8 Démo

Avec **IKBoardID** = 'ce5969180f38bbab2a28f08b44b536dc407cd9ee54843226505ec35a820fd801', nous pouvons calculer la clé **hmac_key**, et finalement vérifier l'authenticité du fichier image, via le script ci-dessous. La totalité de la sortie du script est reproduite ci-dessous afin d'expliciter la structure du tag, de récapituler les calculs intermédiaires pour reconstituer les données d'authentification, et enfin, d'afficher des valeurs test, permettant au lecteur qui le souhaite de reproduire l'opération. Ce script sera mis à disposition **[15]**.

```
$ python odd_verif.py IMG_0006.CR2
hmac_key=1493d665407600fa692d05b06cee9df3ee8e2e3f9faa3685c65eac2ca9a0df29

0xffffffff , version = 0x00000003
0x00000014 fd86c9edc422e36914a68036ba9825e840064c9b (file hmac)
0x00000014 ac0fd27d41e00cbb768000500231e8dfc9b6c6cf (ODD hmac)
tag len=0x00000228  4=0x00000004  rand=0x745a9b08  3=0x00000003
filesize=0x01ce1d99
vhash=0x00000003  keyid=0x00000004  boardid=0x820fd801  hmac_rand=0x68f00e32

n_area = 8
 1   4  salt=0x0bee8168 0x14  63569076b9283b0e64c5a7a7042c1f42871d0fc9  1
        0x004e63fa 0x017fb99d
     sha256=f498ae4693f7518ef6bf9350f1723278ac65dceb57726200dfa03f274a501462
     hmac=63569076b9283b0e64c5a7a7042c1f42871d0fc9
 2   4  0xd584fc0d  0x14  026db83cb6181d6bbbf77ef95002282b4d26ef11 n_other = 9
        0x00000000 0x0000006e,  0x00000072 0x000005c6,  0x0000063c 0x00000704,
        0x00000fa0 0x0000a4cc,  0x0000b574 0x00002202,  0x00010de8 0x0000000e,
        0x0033e3f6 0x00000002,  0x004e63f8 0x00000002,  0x01ce1d97 0x00000002,
         sha256=fb1a01710223fd236dc20db2296dda063dc8900748a8b9e8f28df8e2827bb586
         hmac=026db83cb6181d6bbbf77ef95002282b4d26ef11
 3   4  salt=0x7526256b 0x14  23a93a3aa7d3a8bbe3753ecfbc571166e196ba0e  1
        0x0000006e 0x00000004
     sha256=67abdd721024f0ff4e0b3f4c2fc13bc5bad42d0b7851d456d88d203d15aaa450
     hmac=23a93a3aa7d3a8bbe3753ecfbc571166e196ba0e
 4   4  salt=0x4d3694e3 0x14  0a245bdd4c366931cb08e16eda749e4ddc1ea6a0  1
        0x0000b46c 0x00000108
     sha256=44b8aa4d28701168922acf61435ea4bb442f97b0b14ad7a2510ed68874ee2a72
     hmac=0a245bdd4c366931cb08e16eda749e4ddc1ea6a0
 5   4  salt=0xb582f1fe 0x14  03a013ab309475c210d2aa9f0a3d5641209f2ee3  1
        0x00000638 0x00000004
     sha256=df3f619804a92fdb4057192dc43dd748ea778adc52bc498ce80524c014b81119
     hmac=03a013ab309475c210d2aa9f0a3d5641209f2ee3
 6   4  salt=0x0e87fbf4 0x14  aa374cb98d24bbae9e004fa462ded4297a6f5e63  1
        0x0000d776 0x00003672
     sha256=5e7756ad6be5f2c06555d7ce5721b15803da965bfa68dcc8f47616be8ec67456
     hmac=aa374cb98d24bbae9e004fa462ded4297a6f5e63
 7   4  salt=0x65fea7fb 0x14  954a9ee5009dcfe9e8d1872d514baf05fb2e6491  1
        0x00010df6 0x0032d600
     sha256=4cdc2e2f1d7b06aab6d8b1392d8c2d0c536cce59e42400b21628b301d5f17a1a
     hmac=954a9ee5009dcfe9e8d1872d514baf05fb2e6491
 8   4  salt=0x51f4c551 0x14  cbd2fef0fbd60871e31c9812e2ba5b71308b357c  1
     0x0033e3f8 0x001a8000
     sha256=9cc97295fc75896feb377b29637fa297a2cdce5045078cb67830de10192dba13
     hmac=cbd2fef0fbd60871e31c9812e2ba5b71308b357c

ODD size = 0x228, offset = 0xd78
 sha256=b535910463913964 7ea152c648f04046795d62f3362d33dd0c5e2a8090f9d4a9
 hmac=ac0fd27d41e00cbb768000500231e8dfc9b6c6cf
file hmac=  fd86c9edc422e36914a68036ba9825e840064c9b ok
```

La valeur **file_hmac** calculée, à la dernière ligne, est bien identique au HMAC inclus par le firmware dans le tag ODD, autrement dit : le fichier est considéré comme authentique. On peut remarquer également que la valeur est disponible en clair dans l'en-tête des données ODD.

La figure 2 illustre l'organisation des régions et leur contenu. Le lecteur voulant comprendre l'intégralité du format du tag ODD en version 3 trouvera son bonheur dans les planches 17 à 21 de **[1]**.

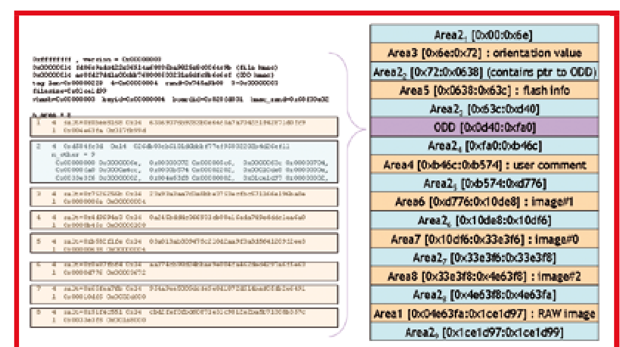


Figure 2

Pour information, des brevets Canon **[14]** décrivent comment créer des données de vérification d'authenticité d'une image et comment générer une clé secrète, la stocker sur un média « tamper resistant » : il s'agit sans doute de la carte Secure Mobile nécessaire au chiffrement des images.

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

Le format TIFF/CR2

Le format de fichier TIFF (pour *Tagged Image File Format*), dans sa déclinaison CR2 pour Canon, contient 4 sections, chacune comportant un répertoire de *tags* (*Image File Directory*, IFD), une image et des méta-données. Les tags décrivent par exemple les dimensions de l'image, le modèle du boîtier, le nom du photographe. La plupart des tags sont officiellement documentés, sauf ceux présents dans le sous-répertoire MakerNotes (notes constructeur). Certains tags comme le tag EXIF pointent sur un sous-répertoire (SubIFD), ou un bloc de données lorsque sa taille est supérieure à 4 octets. Pour atteindre les données ODD, il faut donc parcourir l'IFD#0, l'IFD Exif, l'IFD MakerNotes, et enfin, trouver le tag 0x0083, qui stocke l'*offset* vers ces données.

9 Discussion

Nous sommes donc capables, pour la version 3 du tag, lorsque l'on dispose de `IKBoardID`, de vérifier l'authenticité du fichier image par rapport au contenu du tag ODDv3. Mais est-il facile de falsifier cette vérification : produire un tag valide pour une image modifiée ?

Usurper le boîtier d'origine d'une image reste, pour la même raison, difficile : comme l'algorithme pour générer le secret `IKBoardID` en fonction de valeurs publiques comme `BoardID` et `KeyID` est inconnu, il faut être capable d'extraire cette valeur secrète et unique. Cela nécessite un accès physique au boîtier visé.

Par contre, forger un tag valide est plus simple, on peut en effet réutiliser le secret `IKBoardID` extrait d'un boîtier, ainsi que ses valeurs liées `BoardID` et `KeyID`. C'est d'ailleurs ce que l'on peut observer pour quatre images modifiées par ElcomSoft : `Sakura.jpg`, `Liberty.jpg`, `Stalin.jpg` et `Fuji.jpg`, provenant selon les métadonnées (forgées) de quatre boîtiers différents. Les mêmes valeurs `BoardID`=0x82e933a7 et `KeyID`=1 sont utilisées à chaque fois, donc un même secret `IKBoardID` a certainement été réutilisé pour ces quatre images : c'est une attaque de type « replay ». On peut imaginer alors black-lister ces valeurs, mais selon quels critères ? Et pour la version 2 du tag, il faut révoquer la clé utilisée pour tous les boîtiers d'un même modèle, la fonctionnalité est ici condamnée.

Conclusion

Dans cet article, nous avons vu comment sont produites des données permettant de vérifier l'authenticité des données d'un fichier image (le tag ODD donc) via une combinaison de primitives cryptographiques standards : SHA256 et HMAC-SHA1, ici pour le modèle 60D. Les hypothèses faites par le constructeur étaient que la clé **IKBoardID** resterait secrète, puisque présente uniquement en mémoire, et que l'algorithme de signature est inconnu.

Ceci n'est plus vrai. Canon n'a sans doute pas utilisé le « attack resistant device » pour protéger la clé **IKBoardID** pour des raisons de coût.

Nous avons également constaté qu'avant de se précipiter sur un désassembleur, on peut faire beaucoup d'hypothèses justes par observation de la structure des données du tag ODD et via les chaînes de caractères contenues dans le firmware.

Est-il finalement envisageable de résoudre la dernière énigme dont nous parlions dans le chapitre précédent : comment le kit de vérification de Canon **[6]** calcule-t-il ou vérifie-t-il la valeur `IKBoardID` de l'appareil par rapport à ses valeurs « publiques » `KeyID` et `BoardID` ? Lors de la vérification des images, il est nécessaire de connecter la carte Secure Mobile au PC **[16]** où sont stockés les fichiers images. L'appareil photo n'étant pas connecté au système, le kit OSK-E3 utilise uniquement des données provenant des images. Dans le meilleur des cas pour Canon, la décision de la vérification est prise par la carte, masquant les secrets utilisés. Le brevet US 7650511 **[14]** contient peut-être des éléments de réponse, semble-t-il à base de chiffrement. Peut-être un lien avec les fonctions `CRP_Crypto3DesWrap` du firmware ? ■

REMERCIEMENTS

Merci au projet Magic Lantern d'offrir une plate-forme d'expérimentation inattendue. Je tiens à remercier également Dmitry Sklyarov pour m'avoir désigné la dernière pièce du puzzle (c'est-à-dire où se trouve `hmac_rand`) et pour la clé du 20D. Merci enfin au rédacteur en chef et aux deux relecteurs pour leurs conseils.

Les références de cet article sont disponibles sur : http://www.unixgarden.com/index.php/misc-hs/references-misc-hs-n6/7

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

POUR RENFORCER
LA SÉCURITÉ
DE VOTRE ENTREPRISE,
GLISSEZ-VOUS DANS
LA PEAU D'UN HACKER !

FORMATIONS INTRUSIONS

Cours SANS Institute
Certifications GIAC

SEC 542
Tests d'intrusion applicatifs
et hacking éthique

SEC 560
Network Penetration Testing and
Ethical Hacking

SEC 660
Tests d'intrusion avancés, exploits,
hacking éthique

Dates et plan disponibles
Renseignements et inscriptions
par téléphone +33 (0) 141 409 700
ou par courriel à : formations@hsc.fr

www.hsc-formation.fr

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015

esiea
Grande École d'Ingénieurs

EN PARTENARIAT AVEC

QUARKSLAB
INNOVATIVE SECURITY

LANCE 2 FORMATIONS
COMPATIBLES AVEC UNE ACTIVITÉ PROFESSIONNELLE

REVERSE ENGINEERING
TEST D'INTRUSION

DIPLÔMES ACCRÉDITÉS PAR LA CONFÉRENCE DES GRANDES ÉCOLES

Volume horaire total : 220 h de cours - projets - ateliers - conférences | Ouverture : janvier 2013 | Durée totale : 7 mois | Lieu : Paris

Clôture des inscriptions : 3 décembre 2012

BADGE REVERSE ENGINEERING

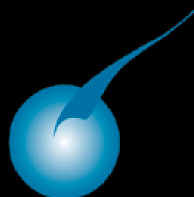
De l'analyse de malwares à la rétro-conception de protocoles,
un BADGE pour être capable d'étudier tous les programmes, protégés ou non

- Analyse de malwares
- Protections logicielles
- Reverse de protocoles
- Reverse de cryptologie

BADGE TEST D'INTRUSION

Des outils indispensables à une approche rigoureuse,
un BADGE pour être capable d'identifier les faiblesses dans un réseau

- Collecte d'informations de sources ouvertes
- Cartographie de réseaux
- Recherche et exploitation de vulnérabilités
- Post-exploitation et furtivité



www.esiea.fr/badges
badges@esiea.fr

www.quarkslab.com/fr-badges
badges@quarkslab.com

Ce document est la propriété exclusive deJohann Locatelli (johann@gykroipa.com) le 1 Mars 2015