Pourquoi la protection de CloneCD a été cassée.

jB
resrever@gmail.com

Mai 2006

A Lise_ Grim,
à qui j'avais parlé de ce texte il y a bien longtemps.

Cet article vous permettra au final de créer un générateur de clés pour CloneCD, mais ce n'est pas son but premier. La protection étudiée ici a été abandonnée depuis plusieurs années par Elaborate Bytes : le schéma des versions actuelles, distribuées par Slysoft, est totalement différent. Lire cet article ne vous sera d'aucune aide si vous voulez attaquer des versions récentes. La version 3.0.7.2, sur laquelle se base cet article, est très dépassée (elle a déjà 5 ans), et de nombreux numéros de série sont disponibles sur internet. Je doute fort que quelqu'un pirate ce logiciel après avoir lu cet article.

Le but de ce texte est d'expliquer comment un algorithme réputé très sûr a pu être mis en échec. En cryptographie l'erreur est très souvent humaine. Nous verrons qu'une fois de plus c'est le cas ici.

Introduction

CloneCD fut pendant longtemps la bête noire de bien des keygeners. La cryptographie commençait à pointer le bout de son nez au niveau des protections logicielles, avec des implémentations souvent très mauvaises. Ici le logiciel utilise un algorithme de chiffrement fort, censé incassable, et des notions mathématiques plutôt complexes.

CloneCD fut l'un des premiers logiciels à utiliser la cryptographie à courbes elliptiques (ECC) pour vérifier la validité d'un numero de série. Microsoft l'a fait avant, dès Windows 98, en utilisant deux algorithmes utilisant des courbes elliptiques sur $GF(p)$. Ceux ci sont très peu sûrs, et peuvent être cassés avec un ordinateur personnel en quelques jours ou heures. La différence ici est que l'ordre des points utilisés est de 241 bits. A l'époque, le plus gros logarithme discret jamais calculé était de 97 bits. En 2002, Chris Monico et son équipe ont calculé un logarithme discret sur $GF(p)$ de 109 bits. Il aura fallu une puissance équivalente à celle de à 10000 ordinateurs fonctionnant pendant 549 jours pour ce calcul. Aujourd'hui, un logarithme de cette taille est toujours très loin d'être calculable. Par exemple, Armadillo utilise ECC sur $GF(2^m)$ avec un ordre premier de 113 bits, et, si l'on oublie la première version utilisant ce schéma, l'algorithme est considéré comme incassable par un groupe restreint de personnes.

CloneCD, Restorator et WinRar furent à ma connaissance les premiers logiciels à utiliser ECC sur $GF(2^m)$. Tous trois utilisent le code source de Pegwit[1],

une alternative à PGP développée par George Barwood. Et tous trois ont été cassés par Dimedrol. Pegwit correctement utilisé est pourtant réputé sûr : WinRar l'utilise toujours, et il n'y a plus eu de générateur de clés pour ce logiciel depuis des années.

Dans un premier temps sera décrit le système de vérification des numéros de série. Puis la faille sera expliquée. Et enfin un générateur de clefs fonctionnel pourra être codé.

1 Analyse de l'algorithme

CloneCD est compressé avec ASProtect. On peut obtenir un dump avec Stripper 2.07, non fonctionnel mais suffisant pour une analyse avec IDA. Toutes les fonctions ayant trait à la cryptographie sont contenues dans la bibliothèque *ElbyCrypt.dll*.

Une licence est formée d'un nom et d'un numéro de série. En premier lieu le format du nom et du numéro de série sera présenté, puis la routine de vérification sera présentée.

1.1 Format du nom et du numéro de série

Après une brève analyse on trouve facilement la routine de vérification du numéro de série.

```
.text:00419708                push    50h             ; int len_max
.text:0041970A                lea     eax, [ebp-16Ch]
.text:00419710                push    eax             ; char *src
.text:00419711                lea     edx, [ebp-11Ch]
.text:00419717                push    edx             ; char *dest
.text:00419718                call    ElbyCrypt_CookText
.text:0041971D                add     esp, 0Ch
.text:00419720                lea     ecx, [ebp-11Ch]
.text:00419726                push    ecx
.text:00419727                call    _strlen
.text:0041972C                pop     ecx
.text:0041972D                cmp     eax, 3
.text:00419730                jnb     short length_ok
```

Le nom entré est modifié par la procédure *ElbyCrypt_ CookText*, qui le convertit en minuscules et supprime les espaces. La longueur du nom une fois converti doit être supérieure ou égale à trois caractères.

```
.text:004197AE                push    edx             ; char *key_out
.text:004197AF                lea     eax, [ebp-0CCh]
.text:004197B5                push    eax             ; char *key_in
.text:004197B6                call    CookKey
.text:004197BB                add     esp, 0Ch
.text:004197BE                lea     ecx, [ebp-0CCh]
.text:004197C4                push    ecx             ; char *key
.text:004197C5                lea     eax, [ebp-1B8h]
.text:004197CB                push    eax             ; cpPair sig
.text:004197CC                call    AssembleKeys
.text:004197D1                add     esp, 8
.text:004197D4                test    eax, eax
```

```
.text:004197D6                  jz      short wrong_key_format
.text:004197D8                  push    0
.text:004197DA                  lea     edx, [ebp-1B8h]
.text:004197E0                  push    edx             ; cpPair sig
.text:004197E1                  lea     ecx, [ebp-16Ch]
.text:004197E7                  push    ecx             ; char *name
.text:004197E8                  call    VerifyKey
.text:004197ED                  add     esp, 0Ch
.text:004197F0                  mov     ebx, eax
.text:004197F2                  jmp     short loc_4197F6
```

Le numéro de série est modifié par la fonction *CookKey*, qui convertit le numéro de série en minuscules et ne conserve que les caractères hexadécimaux (0-9, a-f).

La procédure *AssembleKeys* crée ensuite une signature (r, s), composée de deux bignums, à partir du numéro de série. Le premier octet correspond au nombre de mots de 16 bits utilisés par le bignum. Suivent ensuite les mots, puis un autre octet pour le nombre de mots du deuxième bignum, et la suite de mots. Voyons un exemple à partir d'un numéro de série *cardé* (le nom a été enlevé pour que ce numéro ne soit pas utilisé) :

```
10F6101C94A78AF0B2AE709395408175C6983C7B51E31855452
82C92AA088FBEB10F3D9328ED23E7E44A04CA5F4B9D067CC8AB
69F2E2B4BFB6710252D1011BBF
```

Le numéro de série est interprété comme ceci :

- `10` Nombre de mots utilisés par la première partie du numéro de série.
- `F6101C94A78AF0B2AE709395408175C6983C7B51E3185545282C92AA088FBEB1` Première partie de la signature.
- `0F` Nombre de mots utilisés par la deuxième partie du numéro de série.
- `3D9328ED23E7E44A04CA5F4B9D067CC8AB69F2E2B4BFB6710252D1011BBF` Deuxième partie de la signature.

Le premier bignum sera appelé du numéro de série sera appelé s, et le deuxième sera appelé r. Pour des questions d'implémentation, les bignums sont représentés par mots de 16 bits, de droite à gauche. La signature est alors, après avoir renversé les chaines de caractères :

$$\begin{cases} r = \texttt{1BBFD1010252B671B4BFF2E2AB697CC89D065F4B04CAE44A23E728ED3D93} \\ s = \texttt{B1BE8F08AA922C28455518E3517B3C98C6758140959370AEB2F08AA7941C10F6} \end{cases}$$

Une licence est donc composée d'un nom d'au moins trois caractères, espaces non compris, et d'une numéro de série composé de deux bignums, précédés de leur longueur exprimée en mots de 16 bits.

1.2 Algorithme de vérification

Si le nom est assez long et que le format du numéro de série est correct, la licence est vérifiée avec la fonction *VerifyKey*. Elle prend en entrée le nom, la licence est un troisième paramètre (ici 0) qui ne nous intéresse pas.

Il est conseillé à partir de maintenant de lire le manuel de Pegwit, étant donné que le coeur de la vérification se fait via les routines de Pegwit.

```
VerifyKey       proc near               ; CODE XREF: sub_417394+17F
                                        ; .text:004197E8 ...
```

```
hash_buffer     = dword ptr -30h
var_22          = dword ptr -22h
var_1E          = dword ptr -1Eh
var_1A          = word ptr -1Ah
var_18          = dword ptr -18h
crc             = dword ptr -4
name            = dword ptr  8
sig             = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                add     esp, 0FFFFFF80h
                push    ebx
                push    esi
                push    edi
                mov     esi, [ebp+sig]
                mov     [ebp+crc], 0FFFFFFFFh
                mov     bl, 1
                call    ElbyCrypt_Init
                test    eax, eax
                jz      short loc_417B61
                xor     eax, eax
                jmp     loc_417D8B
; ---------------------------------------------------------------------------

loc_417B61:                             ; CODE XREF: VerifyKey+1C
                push    26h             ; n
                push    0               ; c
                lea     edx, [ebp+hash_buffer]
                push    edx             ; hash_buffer
                call    _memset
                add     esp, 0Ch
                mov     ecx, [ebp+name]
                push    ecx             ; char *str
                call    CalcUserCRC
                pop     ecx
                mov     edi, eax
                push    esi             ; cpPair *sig
                lea     eax, [ebp+hash_buffer]
                push    eax             ; vlPoint vlMac
                push    offset vlPublicKey1 ; vlPoint vlPublicKey
                call    ElbyCrypt_DecodePublic
                add     esp, 0Ch
                push    18h             ; len
                lea     edx, [ebp+crc]
                push    edx             ; unsigned int *crc
                lea     ecx, [ebp+hash_buffer]
                push    ecx             ; unsigned char *data
                call    DoCRCData
                add     esp, 0Ch
                mov     eax, [ebp+hash_buffer+18h]
                mov     edx, [ebp+crc]
```

```
cmp     eax, edx
jz      short crc1_ok
[...]
call    ElbyCrypt_Quit
xor     eax, eax
jmp     wrong_serial
```

Il est à noter que le code ici n'est pas complet. La portion de code *ElbyCrypt_DecodePublic / DoCRCData* est présente trois fois, avec trois clés publiques différentes. On va s'intéresser seulement à la première clé. Je ne sais pas à quoi servent les autres clés, et je n'ai pas pu trouver de licence générée avec ces clés.

La fonction *VerifyKey* appelle *ElbyCrypt_Init*, qui calcule des tables pour accélérer les calculs sur $GF(2^m)$ (voir [2] pour plus d'informations). C'est en fait la fonction *gfInit* de Pegwit qui est renommée.

Le contenu d'un buffer est mis à 0. Nous verrons que ce buffer contiendra un bignum représentant le hash décrypté par la procédure *ElbyCrypt_DecodePublic*. Puis le nom de l'utilisateur est hashé avec *CalcUserCRC*. Cette procédure appelle d'abord *CookText* pour convertir en minuscules le nom entré et supprimer les espaces ; elle calcule ensuite un hash de 32 bits, appelé CRC8 (je ne sais pas si c'est un hash standard) avec une procédure DoCRCString. Ces procédures sont dans le code de l'exécutable, mais sont aussi présentes dans la bibliothèque de cryptographie d'Elby. J'ai trouvé leur nom en regardant le nom des procédures exportées par la bibliothèque.

ElbyCrypt_DecodePublic prend 3 arguments en entrée :

- Une clé publique, qui est un point sur la courbe du système ; La clé publique est un point de la courbe compressé : étant donné que sur une courbe elliptique seuls deux points peuvent avoir la même abscisse, on peut représenter un point par son abscisse et y ajouter un bit (soit 0 soit 1) pour le distinguer. La clé publique est donc un bignum.
- Un bignum vlMac, le message à vérifier (ici un buffer initialisé à 0) ;
- Une signature composée de deux bignums (r, s), ici le numéro de série.

Cette procédure calcule un bignum représentant le message chiffré, et le copie ensuite dans *vlMac*. Nous verrons l'algorithme utilisé, Nyberg-Rüeppel, dans la section suivante. Il suffit pour le moment de savoir que ce système de signature permet de retrouver le message signé à partir de la signature ; c'est pourquoi *vlMac* peut être généré.

Les 24 (0x18) premiers caractères du message vlMac obtenu sont ensuite hashé par la procédure *DoCRCData*. Le hash est comparé aux 4 octets suivants du message, ce qui permet une première vérification de la validité du numéro de série.

Si ce premier test est passé, une deuxième vérification a ensuite lieu, afin de vérifier le nom de l'utilisateur :

```
crc1_ok:                                ; CODE XREF: VerifyKey+6D
                                        ; VerifyKey+A4 ...
                call    ElbyCrypt_Quit
                mov     ecx, dword_4DC708
                xor     edx, edx
                mov     dl, bl
                mov     [ecx+280h], edx
                mov     eax, [ebp+hash_buffer+12h]
```

```
cmp     edi, eax
jz      short crc2_ok
xor     eax, eax
jmp     wrong_serial
```

Les 4 octets de vlMac à partir du 18e (0x12) caractère sont comparés au hash du nom calculé précédemment (cf. plus haut, le hash du nom est stocké dans edi). C'est la deuxième vérification. Si ce test passe, la licence est validée par le programme.

64 bits sont donc utilisés pour déterminer si la licence est valide : 32 bits vérifiant la validité de vlMac, et 32 autres bits vérifiant la validité du nom d'utilisateur. Une attaque exhaustive sur la signature (r, s) afin de générer une licence valide parait infaisable.

Voyons maintenant plus précisément comment est généré *vlMac*.

1.3 Nyberg-Rüeppel en détail

A partir de la signature (r, s) passée en entrée à *ElbyCrypt_DecodePublic*, Nyberg-Rüeppel permet de calculer le message original qui a été signé. C'est pourquoi cette signature est appelée signature *"à recouvrement"*. CloneCD utilise Nyberg-Rüeppel sur une courbe elliptique avec des éléments sur $GF(2^m)$. Cette signature est utilisable sur un groupe fini plus simple, comme $\mathbb{Z}_p^*$. Si vous avez du mal à comprendre son fonctionnement je vous recommande d'étudier son principe $\mathbb{Z}_p^*$ (par exemple avec [3]). L'adaptation sur une courbe elliptique devrait vous paraitre plus simple à appréhender ensuite. Ceux qui sont intéressés par l'utilisation des courbes elliptiques en cryptographie peuvent se référer à [4], qui est une très bonne introduction.

ElbyCrypt_DecodePublic est en fait la procédure *cpVerify* de Pegwit, légèrement modifiée. Voilà le code C de la procédure :

```
void ElbyCrypt_DecodePublic
(const vlPoint vlPublicKey, const vlPoint vlMac, cpPair * sig )
{
    ecPoint t1,t2;
    vlPoint t3,t4,p_order;

    vlCopy2( p_order, prime_order );
    ecCopy( &t1, &curve_point );
    ecMultiply( &t1, sig->s );
    ecUnpack( &t2, vlPublicKey );
    ecMultiply( &t2, sig->r );
    ecAdd( &t1, &t2 );
    gfPack( t1.x, t4 );
    vlRemainder( t4, p_order );
    vlCopy( t3, sig->r );
    if ( vlGreater( t4, t3 ) )
        vlAdd( t3, p_order );
    vlSubtract( t3, t4 );
    vlCopy( vlMac, t3 );
    vlClear( p_order );
    vlClear( t4 );
    vlClear( t3 );
}
```

Dans la procédure de Pegwit, *cpVerify*, vlMac est un paramètre d'entrée, et la procédure retourne 1 si le message signé est bien vlMac, et 0 sinon. Ici, vlMac est au départ un buffer initialisé à zéro. La procédure ne fait que récupérer le message signé et le copie dans vlMac. Les deux vérifications utilisant CRC8, expliquées précédemment, servent ensuite à valider la signature.

1.3.1 Paramètres

Le code source brut n'est pas très intuitif. Une description des calculs, avec les paramètres utilisés, ne sera pas de trop.

- La courbe utilisée est la courbe par défaut de Pegwit, une courbe elliptique sur $GF(2^{255})$. Le groupe est représenté avec des polynômes de degré 17 dont les coefficients sont dans $GF(2^{15})$.
- *curve_point* est un point de la courbe, qu'on appellera P par la suite. Son ordre *prime_order*, noté n par la suite, est un nombre premier de 241 bits.
- *vlPublicKey* est un bignum représentant un point de la courbe, sous forme compressée. On notera ce point Q. Son ordre est le même que celui de P.
- *sig* est la signature, doublet de bignums (r, s), permettant de retrouver le message signé.
- Le message qui a été signé par le doublet (r, s) est un bignum que l'on notera h.

1.3.2 Vérification

L'algorithme de la procédure de vérification est le suivant :

- Calculer $X = sP + rQ$.
- Convertir l'abscisse du point X obtenu en un entier x.
- Calculer ensuite $v = x \bmod n$.
- Retourner le message qui a été signé : $h = r - v \bmod n$.

1.3.3 Signature

L'algorithme de signature peut se déduire facilement de l'algorithme de vérification. Il est présent dans Pegwit est dans la librairie d'Elby.

- Choisir un entier k dans l'intervalle $[1, n-1]$.
- Calculer $kP = (x_k, y_k)$ et convertir x_k en un entier x.
- Calculer $r = x + h \bmod n$.
- Calculer $s = k - rd \bmod n$.
- Retourner (r, s).

d est la clé privée du système. C'est l'entier de l'intervalle $[1, n-1]$ tel que $Q = dP$. Sans lui il n'est pas possible de signer de message. Résoudre le logarithme discret, c'est trouver d. C'est un problème difficile, et compte tenu de l'ordre de P et Q (241 bits), aucune méthode n'est envisageable pour calculer directement d. C'est pourquoi on pensait la protection incassable. Nous verrons qu'elle peut être calculée autrement.

1.4 Petit récapitulatif

Pour le moment, on a vu qu'une licence était composée d'un nom et d'une signature NR. Les bignums utilisés pour la signature sont précédés de leur taille, en nombre de mots de 16 bits. Une fois que le message signé a été reconstitué,

deux vérifications sont effectuées sur ce message : le CRC des 24 premiers octets doit être égal au double mot suivant ces octets, et le CRC du nom doit être égal au double mot située au 18e octet du message.

Pour pouvoir générer un numero de série valide, il faut pouvoir signer un message, et pour cela il faut calculer la clé privée du système.

2 Pourquoi le système est cassable

Une analyse des numéros de série *cardés* va révéler une grosse faille, permettant de calculer facilement la clé privée du programme. Elle permettra également de faire apparaitre un mécanisme de protection oublié jusqu'ici.

2.1 Analyse de deux numéros de série *cardés*

Plusieurs numéros de série ont été *cardés* ou volés avant qu'un générateur de clés soit diffusé. J'ai pu m'en procurer deux, le dernier datant de la version 2.8.4.1. Ces numéros ont été *blacklistés* dans la version étudiée ici : la bibliothèque *CCDDriver.dll* exporte une procédure *CCDDriver_GetTable* qui renvoie une liste de buffers de 64 bits. Chacun de ces buffers est comparé avec les octets 15 à 22 du message signé. Si on trouve un buffer identique à celui du message signé, le numéro de série est détecté comme étant volé, et n'est pas validé.

Voici les deux numéros de série (les noms ont été retirés pour que ces numéros de série ne soient pas utilisés directement sur des versions antérieures) :

```
10542B69A0FDB37CBDD65A6D826C7763C0E1ED5F5EFF
CB6DCCA9CF844F2284BEB10F68BF6FFC23E6E44A04CA
5F4B3CF57DCFB1DE01FFB4C0457E57BBD1001BBF
```

et :

```
10F6101C94A78AF0B2AE709395408175C6983C7B51E3
185545282C92AA088FBEB10F3D9328ED23E7E44A04CA
5F4B9D067CC8AB69F2E2B4BFB6710252D1011BBF
```

On peut extraire alors les signatures (r_1, s_1) et (r_2, s_2), ainsi que les messages signés h_1 et h_2. On obtient :

$$\begin{cases} r_1 = \texttt{1BBFD10057BB457EB4C001FFB1DE7DCF3CF55F4B04CAE44A23E66FFC68BF} \\ s_1 = \texttt{BEB12284844FA9CF6DCCFFCB5F5EE1ED63C06C776D82D65A7CBDFDB369A0542B} \\ h_1 = \texttt{28012E44091E018B7132B3893A55BE7721656D206B637553330DAA9F} \end{cases}$$

$$\begin{cases} r_2 = \texttt{1BBFD1010252B671B4BFF2E2AB697CC89D065F4B04CAE44A23E728ED3D93} \\ s_2 = \texttt{BEB1088F92AA282C5545E3187B51983C75C640819395AE70F0B2A78A1C94F610} \\ h_2 = \texttt{2801D8DB7A11018B6215AD14394F1E8821656D206B637553EBFE7F73} \end{cases}$$

La représentation de $h1$ et $h2$ en mémoire est :

```
0E 00 9F AA 0D 33 53 75 63 6B 20 6D 65 21 77 BE  ..§ł.3Suck me!w¿
55 3A 89 B3 32 71 8B 01 1E 09 44 2E 01 28        U:Ľş2qŇ...D..(

0E 00 73 7F FE EB 53 75 63 6B 20 6D 65 21 88 1E  ..s.þëSuck me!Ĺ.
4F 39 14 AD 15 62 8B 01 11 7A DB D8 01 28        O9.ŋ.bŇ..zÛ∅.(
```

La chaine de caractères *Suck me !* dans les messages signés, que l'on avait pas vue précédemment, saute aux yeux ici. Cette chaine sert à décrypter les portions de code sécurisées par ASProtect et qui permettent l'écriture du numéro de série dans la base de registres. Etant donné que c'est une protection inhérente à ASProtect et non à CloneCD, et que les *Secured sections* sont encore utilisées par ASProtect, je ne rentrerai pas dans les détails sur ce point. Il faut juste retenir que cette chaine est fixe, et que si une autre chaine est utilisée, la portion de code ne sera pas décryptée et les informations d'enregistrement ne seront pas sauvées. Sans numéro de série valide, il est difficile de retrouver cette chaine.

En entrant un de ces numeros de série, et en modifiant un saut conditionnel afin que le numero entré ne soit pas considéré comme volé, on peut debugger le processus afin d'étudier la portion de code qui va être décryptée. On s'aperçoit alors que les clés générées peuvent expirer ou non : une dernière vérification sur le numero de série est faite, ce seulement si la période d'évaluation de 21 jours est terminée :

```
loc_417CC0:                             ; CODE XREF: VerifyKey+12F
                xor     ebx, ebx
                xor     edi, edi
                mov     eax, isNotExpired
                test    eax, eax
                jz      short loc_417D2E
                call    dword_4C9890 ; ASProtect decryption
                jmp     loc_417D28
                ...
```

Le fonctionnement de cette routine est simple. Elle peut être recodée facilement, ou directement rippée. Elle calcule un octet à partir du double mot commençant au 15^{e}octet du message chiffré. Cet octet est comparé au 6^{e}octet du message chiffré. Afin de ne pas alourdir le générateur de clés, et de ne pas allonger encore cet article, je ne commente pas cette routine. A partir du double mot 0x87654321, l'octet généré est 0x7A. On peut fixer cette valeur dans le générateur de clés. Les clés générées auraient alors été facilement *blacklistées* par l'éditeur à l'époque, mais ce n'est pas un problème ici.

2.2 Calcul de la clé privée

Revenons à l'étude de la signature NR. $r = x + h \bmod p$, x étant l'abscisse du point kP et k un nombre aléatoire. Si $r1$ et $r2$ peuvent être aussi proches, compte tenu de la taille de h (très inférieure à celle de r) c'est que pour les deux signatures le même x a été utilisé, i.e le même k (P étant constant). Après vérification on a bien :

$$r_1 - h_1 = r_2 - h_2$$

Elby utilise une valeur de k fixe pour signer ces messages, ce qui est une grave erreur. Ceci est valable pour quasiment toutes les signatures basées sur un logarithme discret. k et d jouent un rôle symétrique pour signer un message. En trouvant kP on peut retrouver d (la clé secrète), et en trouvant d on peut retrouver kP. C'est pourquoi la valeur de k ne doit jamais être divulguée quand on signe un message. De plus, utiliser deux fois la même valeur pour k compromet toute la sécurité du système. Revenons aux équations de la signature. On est en possession de deux messages et leur signature, (h_1, r_1, s_1) et (h_2, r_2, s_2). On a :

$$\begin{cases} s_1 = k_1 - r_1 d \bmod n \\ s_2 = k_2 - r_2 d \bmod n \end{cases}$$

Or $k_1 = k_2$, donc :

$$k_2 = s_1 + r_1 d \bmod n$$

La deuxième équation du système peut alors s'écrire :

$$s_2 = s_1 + r_1 d - r_2 d \bmod n$$

On en déduit la valeur de d :

$$d = (s_2 - s_1)(r_1 - r_2)^{-1} \bmod n$$

```
d = 5F4BA0012F2BCDA1CAC967DA0D004BF9F4AD25A28647433599FB23D991AC
```

Le calcul de d est instantané, et le système est entièrement cassé. On peut maintenant générer des licences valides.

3 Ecriture d'un générateur de clés

Un algorithme de génération de clés sûres (dans le sens où la clé secrète ne pourra pas être calculée à partir de licences volées) sera présenté, suivi du code source en C.

3.1 Algorithme de génération d'une licence

Revenons à l'étude de $h2$:

```
0E 00 73 7F FE EB 53 75 63 6B 20 6D 65 21 88 1E  ..s.þëSuck me!Ĺ.
4F 39 14 AD 15 62 8B 01 11 7A DB D8 01 28        O9.ŋ.bŇ..zÛØ.(
```

Voici le format du message :

- Les deux premiers octets correspondent au nombre de mots qu'occupe le message, i.e toujours 0x000E.
- Les octets 3 à 5 sont des caractères aléatoires.
- Le 6^{e}octet est un octet de contrôle, fixé à 0x4B ici
- Les octets 7 à 14 sont utilisés pour décrypter les portions de code protégées par ASProtect.
- Les octets 15 à 18 sont des caractères que l'on a fixés à 0x87654321.
- Les octets 19 à 22 ont pour valeur le CRC8 du nom d'utilisateur.
- Les octets 23 et 24 sont des caractères aléatoires.
- Les octets 25 à 28 ont pour valeur le CRC8 des 24 premiers caractères du message.
- Les octets 29 et 30 semblent avoir pour valeur 0x01 et 0x28, d'après les deux messages que l'on possède.

L'algorithme de génération d'une licence est alors le suivant. Notons que la valeur de k utilisée pour la signature du message ne doit pas être retrouvée. Il faut utiliser un générateur de nombres aléatoires sûr.

- Créer un bignum h, de longueur 0x000E.
- Copier la clé AsProtect dans h, à partir du 7e octet.

- Copier les deux octets finaux dans h à partir du 29e octet.
- Calculer le hash *crcUser* du nom d'utilisateur.
- Copier *crcUser* dans h à partir du 19e octet.
- Initialiser les octets 3-5 et 23-24 avec des valeurs aléatoires.
- Copier les valeurs fixées aux octets 6 et 15-18.
- Calculer le hash *crcSerial* des 24 premiers octets de h.
- Copier *crcSerial* dans h à partir du 25e octet.
- Calculer la signature (r, s) de h en utilisant Nyberg-Rüeppel.
- Retourner r et s précédés de leur taille.

3.2 Code source C

CloneCD utilise Pegwit, autant le réutiliser pour coder le générateur de clés. Il est à noter que si k est fixé, aucun calcul sur $GF(2^m)$ n'est nécessaire si on précalcule l'abscisse de kP. Dans mon implémentation k varie, pour rendre le système sûr. Toutes les fonctions de hash ont été recodées.

Le code est compilable avec Visual Studio 2005.

```
#define _CRT_RAND_S

#include <stdio.h>
#include <windows.h>
#include "ec_crypt.h"

const vlPoint private_key={15U, 0x91AC, 0x23D9, 0x99FB,
    0x4335, 0x8647, 0x25A2, 0xF4AD, 0x4BF9, 0x0D00, 0x67DA,
    0xCAC9, 0xCDA1, 0x2F2B, 0xA001, 0x5F4B
};
const char asprKey[]="Suck me!";

void bigToAscii(char *str, vlPoint n)
{
    int i;
    sprintf(str, "%02X", n[0]);
    for(i = 0; i < n[0]; i++)
        sprintf(str + 2 + 4 * i, "%04X", n[i + 1]);
}

void CookText(char *dest, char *src)
{
    while(*src != 0)
    {
        if(*src != ' ')
        {
            if(*src >= 'A' && *src <= 'Z')
                *dest = *src - ('A' - 'a');
            else *dest = *src;
            *dest++;
        }
        *src++;
    }
    *dest = 0;
}
```

```
void DoCRC8(char letter, unsigned int *crc)
{
    int i;
    if(crc == NULL)
        return;
    for(i = 0; i < 8; i++)
    {
        char c = *crc;
        *crc >>= 1;
        if((c ^ letter) & 1 != 0)
            *crc ^= 0xC050A963;
        letter >>=1;
    }
}

void DoCRCString(char *str, unsigned int *crc)
{
    *crc = ~0L;
    while(*str != 0)
    {
        DoCRC8(*str, crc);
        str++;
    }
    DoCRC8(0, crc);
}

unsigned int CalcUserCRC(char *name)
{
    unsigned int crc;
    char *str=malloc(strlen(name)+1);
    CookText(str, name);
    DoCRCString(str, &crc);
    free(str);
    return crc;
}

void DoCRCData(int *crc, char *data, int len)
{
    int i;
    *crc = ~0L;
    for(i = 0; i < len; i++)
        DoCRC8(*(data+i), crc);
}

void CreateLicense(char *name, char *serial)
{
    vlPoint message={
    14U, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x2801
    };
    vlPoint secret;
    cpPair signature;
    unsigned int crcUser, crcSerial, random_number;
    int i;
```

```
    char serialR[75], serialS[75];

    for(i = 1; i < 24 / 2; i++)
    {
        rand_s(&random_number);
        message[i] = random_number;
    }
    /* On fixe les valeurs pour la dernière vérification */
    message[2] = 0x7A00;
    message[7] = 0x4321;
    message[8] = 0x8765;

    /* Crée k, nombre aléatoire */
    secret[0] = 15U;
    for(i = 1; i <= secret[0]; i++)
    {
        rand_s(&random_number);
        secret[i] = random_number;
    }
    /* Calcule le CRC du nom et l'intègre dans le message */
    crcUser = CalcUserCRC(name);
    memcpy((char *)message + 18, &crcUser, sizeof(unsigned int));
    /* Rajoute la clé AsProtect dans le message */
    memcpy((char *)message + 6, asprKey, strlen(asprKey));
    /* Calcule le CRC des 24 premiers octets du message */
    DoCRCData(&crcSerial, (char *)message, 24);
    memcpy((char *)message + 24, &crcSerial, sizeof(int));
    /* Signature NR du message */
    gfInit();
    cpSign(private_key, secret, message, &signature);
    gfQuit();
    /* Formate la signature */
    bigToAscii(serialS, signature.s);
    bigToAscii(serialR, signature.r);
    strcpy(serial, serialS);
    strcat(serial, serialR);
}

int main()
{
    char serial[150];
    char name[]="test";
    srand(time(NULL));
    CreateLicense(name, serial);
    printf("%s\n", serial);
    return 0;
}
```

Conclusion

Cette étude permet de montrer une fois de plus qu'il est essentiel de bien comprendre tous les paramètres utilisés dans un algorithme si l'on veut correctement protéger son logiciel. Ici la protection a été réfléchie : la taille des clés

a été bien choisie, un système de *blacklist* a été mis en place. Il est dommage qu'un seul paramètre ait pu détruire toute la sécurité de l'algorithme de licence.

Aujourd'hui beaucoup de développeurs utilisent la cryptographie à clé publique pour sécuriser leur logiciel. C'est souvent une bonne idée, mais il faut faire très attention à son implémentation et à son utilisation. Il n'est vraiment pas rare qu'une faille soit présente. Beaucoup d'éditeurs, pour obtenir des clés courtes, diminuent la taille des paramètres à utiliser pour obtenir des algorithmes sûrs. Certains laissent trainer les clés privées dans le programme. Il faut veiller à vérifier tous les paramètres utilisés, à évaluer le temps nécessaire pour casser le système, avant de distribuer son produit protégé. Enfin, un algorithme de ce type est bien entendu inefficace si un *patch* permet de contourner facilement la protection...

Pour finir je tiens à remercier SeVeN et pusher qui m'ont fourni les numéros de série cardés et diffusés sur internet à l'époque. Sans ces numéros ce texte n'aurait pas été possible. Merci également aux membres de FRET qui ont soigneusement relu cet article.

Références

[1] G. Barwood, **Pegwit** : `www.george-barwood.pwp.blueyonder.co.uk/hp/v8/pegwit.htm`

[2] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, J. Vandewalle : *A Fast Software Implementation for Arithmetic Operations in* $GF(2^n)$, Asiacrypt'96

[3] A. Menezes, P. van Oorschot, S. Vanstone : *Handbook of Applied Cryptography*, CRC Press, 1996

[4] D. Hankerson, A. Menezes, S. Vanstone : *Guide to Elliptic Curve Cryptography*, Springer-Verlag, 2004