Cryptographie et reverse engineering en environnement Win32

Kostya Kortchinsky

Responsable du CERT RENATER
kostya.kortchinsky@renater.fr

Résumé De façon illustrée et didactique, l'article mettra l'accent sur des résultats que l'on peut obtenir par application de l'analyse et du reverse-engineering dans le contexte particulier de la cryptographie en environnement Win32, sans toutefois s'attarder trop sur les techniques de reverse engineering ni la théorie cryptographique.

1 Introduction

L'un des inconvénients majeurs de logiciels à vocation cryptographique en environnement Windows est sans doute l'absence quasi totale de visibilité de l'utilisateur vis à vis de ce qui lui est vendu. Comment s'assurer que ce que l'on achète fait correctement ce qu'il est sensé faire, et ni plus, ni moins ? Il existe une alternative à une confiance aveugle en ce que nous racontent les éditeurs, et qui permettra à toute personne un tant soit peu motivée et techniquement compétente, de se faire une idée par soi-même : le reverse-engineering. Compliqué en temps normal, l'analyse et la rétroconception gagnent une nouvelle dimension lorsque l'on se trouve confronté à des algorithmes imposants et complexes, mais les résultats sont souvent à la hauteur des efforts investis.

L'actualité fait écho des achèvements de certains dans ce domaine, avec l'exemple parlant du système de génération et de vérification de clés produits Microsoft. Utilisant un algorithme de signature sur courbes elliptiques, les parties privées de ses clés ont récemment été déterminées suite à un travail conséquent sur les binaires concernés.

2 Cryptos, ou pourquoi il ne faut pas croire ce que l'on vous raconte (sauf moi)

Cryptos [1] est un logiciel de chiffrement de fichier simple et ergonomique sous environnement Win32, utilisant en version 1.0 – d'après ses auteurs – un cryptage de type DES de 128 bits (!) pour la version française, jusqu'à 4096 bits (! !) sur la version internationale. Courant 2002, un fichier chiffré "testcryptage.rtf.CR1" (avec la version internationale) a été mis en ligne, et une somme de 1000 euros promise à toute personne arrivant à obtenir le fichier original.

2.1 Analyse

Une analyse sommaire du logiciel en question montre rapidement que :
- le type de chiffrement utilisé n'est en rien du DES (même de loin sous la pluie par temps de brouillard) ;
- l'espace des clés possible est ridiculement faible (100 clés au maximum pour la version française à cause d'une erreur de programmation) ;
- une attaque par texte clair avec une dizaine de caractères connus devrait mener à des résultats rapidement ;
- il parait peu probable que les clés soient uniques pour un même fichier chiffré.

Ce type de renseignements peut être obtenus sans recours à la décompilation, grâce à des manipulations simples susceptibles d'être effectuées dans le cadre d'une utilisation régulière du logiciel sur sa version d'évaluation.

A titre d'exemple, le fait qu'un même fichier, chiffré avec les clés " 12 ", " 123 ", " 1234 " et " 12345 ", donne 4 fichiers chiffrés en tout points identiques est plus que suspect. Quelques essais supplémentaires mettent en évidence que seuls les deux premiers chiffres de la clés sont pris en compte, d'où les 100 clés possibles dans la version françaises.

La version internationale introduit deux paramètres supplémentaires, de type petits entiers (16 bits), augmentant l'espace possible des clés.

Des essais sur des fichiers de petite taille, dans le cadre d'une utilisation légitime du logiciel, permettent de mettre en évidence l'usage d'une opération binaire simple de type XOR (ou-exclusif) pour le chiffrement des fichiers, accompagné d'une opération arithmétique simple sur les paramètres constituant la clé.

2.2 Déchiffrement

Le fichier fournit pour le challenge est de type Rich Text Format (RTF), comme le laisse supposer sa première extension. Ce type de document débute toujours par une séquence de caractères identique : `\rtf1\ansi()`.

En s'aidant des précédentes observations, et ayant à disposition le début du texte clair et la totalité du texte chiffré, l'implémentation d'un programme parcourant de façon exhaustive l'espace des clés ne posera pas de problème majeur.

2.3 Source

Fichier decryptos.c :

```
/*
 * gcc -Wall -O3 -o decryptos decryptos.c
 */

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

int main(int argc, char *argv[])
{
  unsigned int j;
  unsigned short int i, k, l, m;
  unsigned short int key, tmp;
  unsigned char encrypted_buffer[8],
    decrypted_buffer[8], key_buffer[8];
  unsigned char buffer[8], file_name[64];
  FILE *encrypted_file, *decrypted_file;

  if ((encrypted_file
      = fopen("testcryptage.rtf.CR1", "rb"))
      == NULL)
    exit(EXIT_FAILURE);

  /* Texte clair */
  memcpy(decrypted_buffer, "{\\rtf1\\ansi\\",
    sizeof(decrypted_buffer));
  /* Texte chiffré */
  if (fread(encrypted_buffer, 1,
      sizeof(encrypted_buffer), encrypted_file)
      != sizeof(encrypted_buffer))
    exit(EXIT_FAILURE);

  for (i = 0; i < sizeof(key_buffer); i++)
    key_buffer[i] = encrypted_buffer[i]
      ^ decrypted_buffer[i];

  /* Premier chiffre de la clé */
  for (i = '0'; i <= '9'; i++)
  {
    /* Premier petit entier */
    for (j = 0; j <= 0xffff; j++)
    {
      tmp = ((key_buffer[0] << 8) | i);
      tmp = (tmp + encrypted_buffer[0]) * j;
      k = (key_buffer[1] - (tmp >> 8) - 1) << 8;
      /* Second petit entier */
      for (l = 0; l <= 0x1ff; k++, l++)
      {
        key = tmp + k;
        for (m = 1; m < 8; m++)
        {
```

```
          if((unsigned char)(key >> 8)
              != key_buffer[m])
            break;
          key = (key + encrypted_buffer[m])
            * j + k;
        }
        /* Y a-t-il concordance ? */
        if (m != 8)
          continue;
        /* Oui, on déchiffre tout le fichier */
        printf("[!] Probable good key found: %04x %04x %04x\n",
          (key_buffer[0] << 8) | i, j, k);
        sprintf(file_name,
          "decryptos_%04x_%04x_%04x.rtf",
          (key_buffer[0] << 8) | i, j, k);
        /* Déchiffrement du fichier avec la clé probale trouvée */
        if ((decrypted_file
             = fopen(file_name, "w+b")) == NULL)
          return EXIT_FAILURE;
        fseek(encrypted_file, 0, SEEK_SET);
        key = (key_buffer[0] << 8) | i;
        while (fread(buffer, 1, 1,
          encrypted_file) > 0)
        {
          /* XOR et opération arithmétique */
          decrypted_buffer[0] = (key >> 8)
            ^ buffer[0];
          key = (key + buffer[0]) * j + k;
          fwrite(decrypted_buffer, 1, 1,
            decrypted_file);
        }
        fclose(decrypted_file);
      }
    }
  }

  fclose(encrypted_file);
  return EXIT_SUCCESS;
}
```

2.4 Résultats

Pour ce fichier, sensé être protégé par un chiffrement de type DES, les conséquences sont plutôt dramatiques. Exécution sur un Intel Pentium 4 à 2.26GHz :

```
kostya@icare:~/tools/cryptos$ time ./decryptos
[!] Probable good key found: 3130 0dfc 89a2
[!] Probable good key found: 3131 0dfc 7ba7
[!] Probable good key found: 3132 0dfc 6dac
[!] Probable good key found: 3133 0dfc 5fb1
[!] Probable good key found: 3134 0dfc 51b6
[!] Probable good key found: 3135 0dfc 43bb

real    0m2.254s
user    0m2.250s
sys     0m0.010s
```

Il nous faut moins de 3 secondes pour parcourir l'ensemble des clés, trouver 6 clés probables, ainsi que les fichiers déchiffrés associés.

```
Fichiers déchiffrés :
kostya@icare:~/tools/cryptos$ md5sum *.rtf
1c401e97695ad4b7bd48b0381f90986f  decryptos_3130_0dfc_89a2.rtf
1c401e97695ad4b7bd48b0381f90986f  decryptos_3131_0dfc_7ba7.rtf
e4b0521ee9e8e05e6c96bf68eea80c27  decryptos_3132_0dfc_6dac.rtf
e4b0521ee9e8e05e6c96bf68eea80c27  decryptos_3133_0dfc_5fb1.rtf
e4b0521ee9e8e05e6c96bf68eea80c27  decryptos_3134_0dfc_51b6.rtf
e4b0521ee9e8e05e6c96bf68eea80c27  decryptos_3135_0dfc_43bb.rtf
```

Le fichier original étant au format Rich Text, il est assez simple de vérifier la validité des fichiers déchiffrés avec nos clés probables. Les deux premières clés mènent à un fichier déchiffré globalement compréhensible, les 4 clés suivantes, au fichier original (pas d'erreur RTF).

3 ASProtect, un cas concret de faille d'implémentation de l'algorithme RSA

ASProtect est un système de protection logicielle d'applications, offrant de nombreuses fonctionnalités, parmi lesquelles le chiffrement et la compres-sion de l'application, l'ajout de contre-mesures variées compliquant débugage et désassemblage, ainsi qu'un système d'enregistrement de l'application utilisant un algorithme à clé publique.

Ce système d'enregistrement a connu de multiples évolutions au fil des versions du produit, l'une d'entre elles ayant été motivée par la sortie de générateurs de clés pour une trentaine de produits protégés avec ASProtect, dont ASPack et ASProtect eux-même. Cet événement laissait suggérer la présence d'une faille dans l'algorithme utilisé, du RSA, dont les paramètres avaient cependant une taille conséquente, 1024 bits. Les travaux effectués pour cet article l'ont été sur une version 1.1 de ASProtect qui pourra être mise à disposition du lecteur.

3.1 Processus de vérification d'une clé d'enregistrement

Une clé d'enregistrement valide pour une application protégée par ASProtect se présente généralement sous forme d'un fichier ".key", pouvant être importé au Registre de Windows, y ajoutant une nouvelle valeur dont le nom est "Key" et dont le contenu est une chaîne binaire de 1024 bits (au plus) encodée en Base64.

```
Exemple de clé : REGEDIT4

[HKEY_CURRENT_USER\Software\Asprotect]
"Key"="rfk1ksv3o8WSSYZDmYZpjIFpmd8t5b38J
       Q/26lVoGLruotu/tYygUKwpSlPSkoYnc0
       Gj3o2vUiLQaBavC0SvG2xdnkYiEFK3hTT
       Mpgp92p+EAb46ibXK8Eoc78y8ajf/NXj0
       GGJMfV6AXQrUOYFaGB5S7YvkzTL9oyKLh
       rbmPRh="
```

Le traitement appliqué à cette chaîne par le squelette d'ASProtect peut être suivi grâce à un simple debugger de ring3 - OllyDbg [2] pour le citer, néanmoins les quelques trucs et astuces utilisés par ASProtect à l'encontre de ce type d'activité peuvent désorienter des débutants. Le reverse-engineering n'étant pas l'objet premier de cet article, je ne détaillerai pas le procédé mais le décrirai brièvement.

Une fois l'exécutable chargé sous OllyDbg, il nous faut modifier un peu UnhandledExceptionFilter dans kernel32.dll afin que les exceptions ne soient pas passées au debugger, forcer le saut après NtQueryInformationProcess fait l'affaire, puis lancer le programme à proprement parler. La première exception arrêtera l'exécution du code dans le squelette d'ASProtect une fois celui-ci déchiffré, une aubaine. Un breakpoint sur l'API RegQueryValueExA, quelques shift-F9, et nous atteignons la lecture de la valeur de la clé de Registre voulue. Des "breakpoint on access" sur les zones mémoires impliquées nous mènerons jusqu'à la procédure de décodage Base64, puis jusqu'à la première opération arithmétique effectuée sur une partie de la chaîne décodée.

L'instant semble être tout indiqué pour effectuer un dump de la plage mémoire dans laquelle nous évoluons : nous ne disposons pas du code du squelette d'ASProtect puisqu'il est déchiffré dynamiquement à l'exécution, et un dead-listing se révèlera indispensable à l'usage ; de plus il y a de grandes chances que les paramètres de l'algorithme de chiffrement soient présents à ce moment en mémoire. Nous suivons ensuite quelques "ret" en notant les adresses qui nous seront utiles pour le désassemblage, et regardant de temps à autre le contenu des buffers. Un oeil habitué discernera rapidement les paramètres importants, d'autant plus que l'auteur d'ASProtect nous facilite la tache en utilisant des buffers statiques. Un extrait du code (chargement sous IDA [3] du dump de la mémoire en tant que fichier binaire, l'offset peut varier) :

```
; modulo (n)
seg000:0092D0E7 mov     ecx,  offset byte_9326D4
; exposant public (e)
```

```
seg000:0092D0EC     mov edx,  offset byte_932654
; message (m)
seg000:0092D0F1     mov eax,  offset byte_9325D4
seg000:0092D0F6     call      sub_92C568
```

Les paramètres sont stockés sous forme d'un tableau de 128 octets, l'octet de poids faible en première position. Le résultat de la fonction est stocké dans le tableau contenant le message initial. Récupérés du dump de la mémoire, les paramètres sont les suivants :

```
m  = 0x183de6b6 868b22a3 fd32cde4 8bed521e
       185a8139 d40a5d80 5e7d4c62 18f47835
       ff376abc ccef1c4a f0cab589 3abe0184
       9fda7d0a a6cc3485 b7521022 469e5d6c
       1baf440b af1668d0 2252af8d dea34173
       278692d2 534a29ac 50a08cb5 bfdba2ee
       ba186855 eaf60f25 fcbde52d df996981
       8c698699 43864992 c5a3f7cb 9235f9ad

e  = 0x 11

n  = 0xeb1d4ead a4815f62 77519791 bffa8b4c
       0b872d1c 436515ab d9572b22 bf6a03fe
       cb4e5cc4 9af1ee35 c3134461 7a12106c
       30569052 9b9ce7f1 3ed2d37c d7034a3e
       dd096853 ec61243b ccac5a58 800b0330
       a4dd85e9 aa237f2f 2ae60ca0 49b1d277
       7b2e0c5f f51e0583 82a86c3e c12f7ab4
       16420227 72ff2a2d 3dba7047 25702199
```

L'utilisation d'un logiciel de calcul sur les grands nombres confirmera le contenu du message déchiffré à la sortie de la fonction.

Message déchiffré :

```
m' = 0x00000000 256453f4 b5e2e2d7 d775aa2c
       b87d6273 64f523a1 14030000 00000000
       00000000 00000000 00000000 00000000
       00000000 00007468 67696c69 77540a0d
       0a0d2d20 65736e65 63696c20 65746973
       202d0a0d 594b534e 49484354 524f4b20
       61797473 6f4b0000 00000000 0000c21c
       f68cadec cc1a8fd2 7072bd9d 839d1002
```

Une séance de débuggage complémentaire contribuera à déterminer le format d'une clé d'enregistrement déchiffrée (je ne rentrerai pas dans les détails) :

0 2 octets identifiant le champ suivant ;

2 24 octets contenant un hash 128 bits complété de caractères nuls ;
26 76 octets contenant le nom d'utilisateur et éventuellement des informations supplémentaires (type de licence, nombre de licences, ...) ;
102 2 octets identifiant le champ suivant ;
104 24 octets contenant un hash 160 bits (?) complété de caractères nuls.

Le premier hash se révèlera être un hash MD5 du nom d'utilisateur, dont la validité sera vérifiée par le code de protection ajouté à l'application. La nature du second hash est plus floue pour le moment, on ne peut que constater qu'il est hashé de nouveau et comparé à un hash précalculé.

Voila pour le processus de vérification d'une clé d'enregistrement. Nous disposons à présent d'une quantité d'informations non négligeable pour s'attaquer au système d'enregistrement de ASProtect.

3.2 Processus de génération d'une clé d'enregistrement

Le logiciel ASProtect lui-même inclut un module de génération de clé d'enregistrement : une paire de clé est susceptible d'être générée pour chaque projet, et des clés d'enregistrement produites en fonction des informations fournies. Dans un premier temps, il nous faudra disposer d'un exécutable déprotégé susceptible de fournir une base à tout désassemblage ultérieur. Je vous invite pour cela à appliquer les méthodes décrites par Nicolas Brulez dans son article. Un rapide parcours de l'exécutable (désassemblé ou en hexadécimal) pourrait faire naître en vous un intérêt particulier pour la chaîne "*_rsaKeyGen@16*", typiquement un nom de fonction importée ou exportée. Grâce à un bon désassembleur (IDA Pro [3] pour ne citer que lui), il apparaît que cette fonction appartient à une portion de code chargée dynamiquement depuis l'une des ressources de l'application, TRSA, présente sous forme chiffrée dans l'exécutable.

Afin d'obtenir une image fonctionnelle de la bibliothèque RSAVR.dll et dont une partie est (c) 1986 Philip Zimmermann si l'on en croît deux chaînes de caractères, deux possibilités : extraire la ressource en question et appliquer la procédure de déchiffrement (fastidieux), ou bien dumper la mémoire une fois la DLL déchiffrée, l'extraire et la reconstruire (d'une facilité déconcertante).

La fonction rsaKeyGen est appelée dans ASProtect grâce aux quelques lignes suivantes :

```
; modulo (n)
seg000:0044BA82     push    offset byte_45EC50
; exposant privé (d)
seg000:0044BA87     push    offset byte_45EBD0
; exposant public (e)
seg000:0044BA8C     push    offset byte_45EB50
; 1024bits
seg000:0044BA91     push    400h
; _rsaKeyGen@16
seg000:0044BA96     call    [ebp+var_C]
```

Quelques minutes de reverse-engineering permettent de mettre en évidence l'algorithme de génération de la clé RSA :

1. initialisation du générateur de nombres aléatoires,
2. génération d'un nombre premier P de 512bits :
   - génération d'un nombre aléatoire P de 510bits,
   - mise à 1 des 511ème et 512ème bits (poids fort),
   - détermination d'un nombre premier probable supérieur (immédiatement) au nombre généré grâce à un test de Miller-Rabin [4],
3. génération d'un nombre premier Q de 512bits,
   - génération d'un nombre aléatoire Q de 510bits,
   - mise à 1 des 511ème et 512ème bits (poids fort),
   - détermination d'un nombre premier probable supérieur (immédiate-ment) au nombre généré grâce à un test de Miller-Rabin [4],
4. si P - Q (ou Q - P en fonction du signe) a une longueur strictement inférieure à 505bits, on retourne à l'étape 1.
5. génération de n, e (fixé à (1 ¡¡ 4) + 1 soit 17), et d grâce aux nombres précédemment calculés.

3.3 Identification de la vulnérabilité

Il convient maintenant de détailler les portions de code pertinentes de la fonction de génération de la clé RSA. Initialisation du générateur de nombre aléatoires :

```
.text:10003358     push    0
.text:1000335A     call    _time
.text:1000335F     add     esp, 4
.text:10003362     mov     esi, eax
.text:10003364     call    ds:GetCurrentThreadId
.text:1000336A     add     esi, eax
.text:1000336C     call    ds:GetTickCount
.text:10003372     xor     esi, eax
; unsigned int
.text:10003374     push    esi
; srand((time(NULL) + GetCurrentThreadId())
  ^ GetTickCount());
.text:10003375     call    _srand
```

Génération d'un nombre aléatoire :

```
.text:1000138B     movsx   eax, bx
.text:1000138E     shr     eax, 5
.text:10001391     push    edi
; byteLength = bitLength / 32;
.text:10001392     mov     edi, eax
```

```
.text:10001394     neg     eax
.text:10001396     shl     eax, 5
; remainingBits = bitLength % 32;
.text:10001399     add     ebx, eax
.text:1000139B
.text:1000139B loc_1000139B:
.text:1000139B     call    generate32bitNumber
; for (i = 0; i < byteLength; i++)
;   bigNumber[i++] = generate32bitNumber();
.text:100013A0     mov     [esi], eax
.text:100013A2     add     esi, 4
.text:100013A5     dec     edi
.text:100013A6     jnz     short loc_1000139B
.text:100013A8     pop     edi
.text:100013A9
.text:100013A9 loc_100013A9:
.text:100013A9     test    bx, bx
.text:100013AC     jz      short loc_100013C1
.text:100013AE     call    generate32bitNumber
.text:100013B3     mov     edx, 1
.text:100013B8     mov     cl, bl
.text:100013BA     shl     edx, cl
.text:100013BC     dec     edx
.text:100013BD     and     eax, edx
; if (remainingBits != 0)
;   bigNumber[i] = generate32bitNumber() & ((1 << remainingBits) - 1);
.text:100013BF     mov     [esi], eax
Génération d'un nombre aléatoire de 32 bits :
.text:100013D2     xor     esi, esi
.text:100013D4     mov     edi, 4
.text:100013D9
.text:100013D9 loc_100013D9:
; for (result = 0, i = 4; i > 0; i--)
;   result = (result << 8) + rand();
.text:100013D9     call    j__rand
.text:100013DE     shl     esi, 8
.text:100013E1     add     esi, eax
.text:100013E3     dec     edi
.text:100013E4     jnz     short loc_100013D9
.text:100013E6     mov     eax, esi
```

Le constat est simple, l'ensemble des opérations de génération de nombres aléatoires s'effectue grâce aux fonctions standard du C srand et rand. srand se contente d'affecter la valeur passée en paramètre à l'aléa, variable globale.

```
Contenu de la fonction rand :
.text:10003916     mov     ecx, [eax+14h]
```

```
.text:10003919     imul    ecx, 343FDh
.text:1000391F     add     ecx, 269EC3h
; globalSeed = (globalSeed * 0x343fd) + 0x269ec3;
.text:10003925     mov     [eax+14h], ecx
.text:10003928     mov     eax, ecx
.text:1000392A     shr     eax, 10h
; result = (globalSeed >> 16) & 0x7fff;
.text:1000392D     and     eax, 7FFFh
```

Nous voici donc confronté à un cas pratique de génération de clé RSA basée sur l'utilisation d'un générateur d'aléa faible (environ 32 bits) [5].

3.4 Implémentation de l'attaque

Certes, les vulnérabilités théoriques d'implémentation de tels algorithmes existent, mais les possibilités qu'elles offrent à un particulier ne sont souvent pas claires, et les attaques à leur encontre restent généralement l'apanage d'organisations gouvernementales ou de gros centres de recherche ... qu'en est-il dans le présent cas ?

Le principe de l'attaque à mener est simple : un parcours exhaustif de l'espace des graines accompagné des calculs adéquats devrait nous permettre de retrouver les nombres P et Q utilisés pour calculer le paramètre N en notre possession. Cette solution n'est certainement pas unique, mais permettra d'aboutir à un résultat de façon efficace et peu demandant en puissance de calcul et en temps.

Plusieurs éléments de la procédure de génération de la clé RSA méritent d'être regardés en détails.

Initialisation du générateur de nombres pseudo aléatoires La graine de 32 bits initialisant le générateur de nombres pseudo aléatoires est issue d'une opération binaire simple sur les résultats de trois appels à des fonctions standard en environnement Win32 :

- *time(NULL)*, renvoie le nombre de secondes écoulées depuis le 1er janvier 1970 à `00:00:00 UTC`. Aspect particulièrement intéressant pour nous, sa valeur se situe entre *0*x386d3570 pour le 1er janvier 2000 et *0*x3a4fba6f pour le 31 décembre 2000 ;
- *GetCurrentThreadId()*, retourne l'identifiant de la "thread" (processus léger) ayant appelé la fonction, valeur entière guère supérieure à 5000 dans les environnements impliqués ;
- *GetTickCount()*, dont le résultat est le nombre de millisecondes écoulées depuis le démarrage du système, qui, sans jugement de valeur, dépassait rarement *0*x5265C00 (24 heures) sur un système Windows personnel en 2000.

De ces observations, on peut déduire un intervalle approximatif de l'expression

```
(time(NULL) + GetCurrentThreadId()) ^ GetTickCount()
```

dont le résultat devrait être compris entre *0*x38000000 et *0*x3fffffff, soit un espace de 2^27 valeurs, une réduction significative (bien que pas forcément exacte) des graînes possibles.

Travail sur les nombres premiers Bien qu'ayant réduit, de façon plausible, l'espace des valeurs à parcourir, la quantité d'opérations à effectuer reste conséquente, notamment à cause des opérations sur les grands nombres et les tests de primalités.

Dans un premier temps, il est nécessaire de s'affranchir de ces fameux tests. L'écart moyen entre le nombre généré P et ce fameux nombre premier immédiatement supérieur (appelons le P') est grossièrement de l'ordre de log(P) [6], qui vaut environ 355 dans le cas qui nous intéresse ($2^512 < P < 2^513$). Il est donc raisonnable de penser que la majeure partie des bits de poids fort de P et P' est identique. Il en va bien évidemment de même pour Q.

En conséquence, une très grande partie des bits de poids fort du produit $P * Q$ sera similaire à $P' * Q'$ (largement supérieure à 512 bits).

Le nombre de couples possibles (P, Q) étant de l'ordre de 2^32 et normalement assez bien répartis, il est raisonnable de penser qu'un comparaison du nombre N de ASProtect avec le nombre N que nous allons être amenés à calculer sur leurs 64 bits de poids fort, ou plus, donnera des résultats satisfaisants.

Les tests de primalités en moins, le facteur limitant des itérations devient la multiplication des nombres non premiers P et Q de 512 bits. Un détail va nous permettre d'y remédier. Ces fameux nombres se fait par blocs de 32 bits. En multipliant les 32 bits de poids fort de P à ceux de Q, on obtiendra un nombre de 64 bits dont les 32 bits de poids fort correspondront à ceux du produit complet $P * Q$ aux retenues près.

Cette comparaison sur 32 bits permettra de déterminer si de plus amples calculs sont nécessaires, ou si l'on peut passer directement à la graine suivante.

On en déduit le contenu des itérations à effectuer lors du parcours de l'espace des graines :

1. initialisation de la graine ;
2. génération du nombre P de 512 bits avec 2 bits de poids fort à 1, par 16 appels successifs à la fonction de génération d'un nombre de 32 bits issue du code de RSAVR.dll ;
3. génération du nombre Q de 512 bits avec 2 bits de poids fort à 1, de façon similaire ;
4. si P et Q sont trop proches, retour à l'étape 3 ;
5. calcul du produit des entiers de poids fort de P et Q ;
6. si les 32 bits de poids fort du produit ne correspondent pas aux 32 bits de poids fort de N (aux retenues près), on incrémente la graine et on continue en 2. ;
7. s'il y a correspondance, on calcul le produit complet de P et Q, et on compare davantage de bits ;

8. si il y a de nouveau correspondance, on calcul P' et Q', sinon on passe à la graine suivante.

3.5 Code source

Fichier aspr.c :

```
/*
 * FreeLIP v1.1 is needed
 * gcc -Wall -O3 -lm -o aspr aspr.c lip.c
 */

#include <stdio.h>
#include <stdlib.h>
#include "lip.h"

unsigned long int asprSeed = 0;

/* Modulus de ASProtect */
unsigned long int asprN[32] =
{
  0x25702199, 0x3dba7047, 0x72ff2a2d,
  0x16420227, 0xc12f7ab4, 0x82a86c3e,
  0xf51e0583, 0x7b2e0c5f, 0x49b1d277,
  0x2ae60ca0, 0xaa237f2f, 0xa4dd85e9,
  0x800b0330, 0xccac5a58, 0xec61243b,
  0xdd096853, 0xd7034a3e, 0x3ed2d37c,
  0x9b9ce7f1, 0x30569052, 0x7a12106c,
  0xc3134461, 0x9af1ee35, 0xcb4e5cc4,
  0xbf6a03fe, 0xd9572b22, 0x436515ab,
  0x0b872d1c, 0xbffa8b4c, 0x77519791,
  0xa4815f62, 0xeb1d4ead
};

unsigned long int asprRandom(void)
{
  register unsigned long int r, s;

  /* Réimplémentation des appels successifs à rand(),
     retournant directement l'entier de 32 bits */
  r = asprSeed;
  r = (r * 0x343fd) + 0x269ec3;
  s = (r >> 16) & 0x7fff;
  r = (r * 0x343fd) + 0x269ec3;
  s = (s << 8) + ((r >> 16) & 0x7fff);
  r = (r * 0x343fd) + 0x269ec3;
  s = (s << 8) + ((r >> 16) & 0x7fff);
```

```
  r = (r * 0x343fd) + 0x269ec3;
  s = (s << 8) + ((r >> 16) & 0x7fff);
  asprSeed = r;

  return s;
}

int main(int argc, char *argv[])
{
  unsigned long int i, j;
  unsigned long int p[16], q[16];
  unsigned long int x, y, z;
  verylong vlP = 0;
  verylong vlQ = 0;
  verylong vlN = 0;
  verylong vlAsprN = 0;

  zultoz(asprN, 32, &vlAsprN);
  for (i = 0x3fffffff; i >= 0x30000000; i--)
  {
    /* Notre équivalent de srand(i) */
    asprSeed = i;
    /* Génération de P de 512 bits */
    for (j = 0; j < 16; j++)
      p[j] = asprRandom();
    /* 2 bits de poids fort à 1 */
    x = (p[15] |= 0xc0000000);
    do
    {
      /* Génération de Q de 512 bits */
      for (j = 0; j < 16; j++)
        q[j] = asprRandom();
      /* 2 bits de poids fort à 1 */
      y = (q[15] |= 0xc0000000);
      z = (x > y) ? x - y : y - x;
      /* Tant que P et Q sont trop proches */
    } while (z < 0x01000000);
    /* Multiplication des deux entiers de poids fort */
    z = asprN[31]
      - ((((unsigned long long)x) * y) >> 32);
    if (z <= 2)
    {
      fprintf(stdout, "[!] Probable good seed found : 0x%lx
                       (%lu)\n", i, z);
      fflush(stdout);
```

```
      /* Passage en verylong de P et Q */
      zultoz(p, 16, &vlP);
      zultoz(q, 16, &vlQ);
      /* Multiplication des deux grands nombres */
      zmul(vlP, vlQ, &vlN);
      /* Vérification de la similitude de 4 * 31 bits de
         poids fort du N calculé et du N de ASProtect */
      if (vlN[32] == vlAsprN[32]
          && vlN[31] == vlAsprN[31]
          && vlN[30] == vlAsprN[30]
          && vlN[29] == vlAsprN[29]) {
        fprintf(stdout, "[+] Good seed found : 0x%lx\n", i);
        /* Si concordance, calcul de P et Q */
        if (zodd(vlP) == 0)
          zsadd(vlP, 1, &vlP);
        zmod(vlAsprN, vlP, &vlN);
        while (ziszero(vlN) == 0)
        {
          zsadd(vlP, 2, &vlP);
          zmod(vlAsprN, vlP, &vlN);
        }
        zdiv(vlAsprN, vlP, &vlQ, &vlN);
        fprintf(stdout, "p = "); zwriteln(vlP);
        fprintf(stdout, "q = "); zwriteln(vlQ);

        return 0;
      }
    }
  }
  fprintf(stdout, "[-] Good seed not found\n");

  return 1;
}
```

3.6 Résultats

Exécution un Intel Pentium 4 à 2.26GHz :

```
kostya@icare:~/tools/aspr$ time ./aspr
[!] Probable good seed found : 0x399bacc4 (0)
[+] Good seed found : 0x399bacc4
p = 126288019099230838424474746472287405436197229176055513682050 5\
    384518153966116669245699179134519091648338851029999841022348 6\
    052599507362956069915523128660647

q = 130735286696710563710145518127522117222698023352396111601094 3\
```

```
80195977835183387039889438926902676765164520587929708355155 33\
67228808904290449109145094425 4399

real    1m46.711s
user    1m46.710s
sys     0m0.000s
```

Le principe est simple, les résultats obtenus tout à fait satisfaisants. Le temps moyen de cassage d'une clé RSA de 1024 bits générée par ASProtect est de l'ordre de 2 minutes.

4 Conclusion

Cas particulier ? Libre à chacun de le penser. Néanmoins ce n'est pas l'avis d'un certain nombre de 'crackers' qui a amplement parcouru le sujet avec un succès pour le moins surprenant : récupération de clés privées, falsification de signatures quel que soit l'algorithme, exploitation de débordement de buffer, et ce avec un unique point commun : une faiblesse d'implémentation trouvée et exploitée. Du petit éditeur de logiciels, au géant du système d'exploitation, personne n'est l'abri du reverse-engineering de ses binaires, dans des conditions plus ou moins légales. La sécurité de l'implémentation du système cryptographique par l'obscurité est alors une bien piètre protection ? La technique d'exploitation de la faiblesse du générateur de nombres aléatoires pour la génération des clés RSA de ASProtect présente l'avantage de s'adapter facilement à d'autres problèmes sous des environnements variés [7].

Références

1. Cryptos, `http://www.palmsoft.fr/`
2. OllyDbg, `http://home.t-online.fr/home/Ollydbg/`
3. Interactive Disassembler Pro, `http://www.datarescue.com/`
4. Miller-Rabin's Primality Test, `http://www.security-labs.org/index.php3?page=5`
5. Randomness Recommendations for Security, `http://www.ietf.org/rfc/rfc1750.txt`
6. How Many Primes Are There ? : `http://www.utm.edu/research/primes/howmany.shtml`
7. Enigme : problème d'aléa en JAVA MISC n°13.