TRACKWATCH

GATEWATCHER

Shikata Deshita

Romain LESTEVEN

7 août 2015

Table des matières

1 Introduction

Une personne qui veut s'en prendre à un système d'information peut avoir comme objectif soit l'extraction de données, soit la prise de contrôle du système. Dans ce cas il doit dans un premier temps trouver une faille de sécurité, puis exploiter cette faille en forçant la machine à exécuter un code fourni par l'attaquant qui va exécuter une série d'actions et dont le résultat dépend de l'objectif de l'attaquant et des possibilités offertes par la machine. Ce genre de code est appelé "shellcode".

Les shellcodes ne permettaient, à leurs débuts, "que" de donner l'accès à un shell (ou invite de commande) à l'attaquant qui avait alors un certain contrôle sur la machine (en fonction des droits du shell sur la machine). De nos jours ces shellcodes permettent de faire des choses beaucoup plus intéressantes comme augmenter les droits de l'attaquant afin de le rendre root (admin).

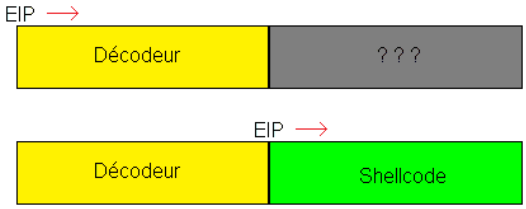
Bien évidemment, avec le temps les antivirus ont apprit à se défendre contre la plupart des shellcodes par des méthodes de signatures sur les shellcodes connus ou sur des modèles de shellcode classiquesou bien en évaluant la dangerosité d'un code. Mais les "hackers" ont eux aussi apprit à bypasser les antivirus ; il existe principalement deux solutions : les shellcodes "fais maison" qui n'ont pas comportement "agressif" et les encodages de shellcodes.

2 Encodage de shellcode

Si les antivirus ne peuvent identifier un code exécutable, comment peuvent-ils le catégoriser comme dangereux ? Ils ne peuvent pas ! C'est l'objectif des encodages de shellcodes : modifier le code afin de le rendre complètement différent. Mais viennent alors deux questions :

— Comment exécuter un shellcode encodé ?
— Quel type d'encodage utiliser ?

En effet, un shellcode encodé ne peut être exécuté en l'état sur la machine, c'est pour cela qu'un "décodeur" est ajouté en préfixe du shellcode. Ainsi, à l'exécution du code, le shellcode est d'abord décodé avant d'être exécuté :



Si on utilisait un chiffrement, par exemple AES, on serait confronté à un problème de taille (au sens propre du terme). Les shellcodes ont souvent une taille limitée par la vulnérabilité exploitée. Si on utilisait un chiffrement "standard", comme AES, le décodeur aurait une taille tellement grande qu'on ne pourrait quasiment jamais l'utiliser. De plus, le chiffrement possède deux objectifs : modifier le code, et être "impossible" à déchiffrer sans la clé. Or nous ne recherchons que l'aspect modification de code puisque la clé doit se trouver dans le décodeur afin que le shellcode puisse être décodé à l'exécution (sauf dans certains cas particuliers qu'on abordera plus tard). Des encodages spécialisés pour les shellcodes ont donc vu le jour.

Maintenant que les shellcodes sont impossible à identifier, que pouvons-nous faire ? Bien que le shellcode soit encodé, le décodeur lui est en clair, on peut donc facilement répertorier les décodeurs afin de les détecter par signature et catégoriser le code comme dangereux ! Mais il existe bien évidemment des solutions pour outrepasser ces signatures :

— les encodages "fais maisons", créés de zéro et donc non répertoriés dans les signatures.
— les encodages "réchauffés maison", créés à partir d'un encodage déjà existant de sorte que le décodeur ne correspond plus à la signature.
— les décodeurs polymorphiques, c'est à dire qui a une forme pseudo-aléatoire tout en ayant le même résultat, par exemple Shikata Ga Nai.

3 Shikata Ga Nai

3.1 Présentation

Shikata Ga Nai signifie "On ne peut rien y faire". Cette expression a été utilisée lors de la Seconde Guerre Mondiale lorsque les américains occupaient le Japon.

Cet algorithme utilise une clé de 4 octets pour encoder le shellcode par le résultat de l'opération XOR entre les deux opérandes pré-citées. Après chaque opération la clé est modifiée, soustraite par les prochains 4 octets à encoder (soustraction octet par octet).

Le décodeur de Shikata Ga Nai est polymorphique, la plupart de ses instructions peuvent être remplacées ou même déplacées. En fait il n'existe qu'un seul point fixe : l'instruction fnstenv [esp-0xc]. Cette instruction de 4 octets est toujours présente dans le décodeur et permet aux antivirus, associé à un comportement de modification de la mémoire, de détecter cet encodage.

A quoi bon continuer puisque les antivirus font déjà le travail ? Tout simplement parce que, dans une utilisation normale, le shellcode ne se trouvera pas dans un fichier, il est envoyé directement à un programme via le réseau. L'antivirus ne pourra donc jamais le détecter ! Par contre un IDS présent sur le réseau, lui, le pourra, mais se baser sur une signature de 4 octets impliquera tellement de faux-positifs (un IDS en coeur de réseau doit gérer entre plusieurs centaines de Mo et plusieurs Go de données par seconde) qu'il est impossible d'utiliser cette technique (le travail sur le comportement du code après la détection de la signature sera compliqué à gérer).

Je vais donc vous présenter une solution que j'ai mit au point pour détecter l'encodage et décoder le shellcode, mais d'abord il faut comprendre comment fonctionne Shikata Ga Nai !

3.2 Le décodeur

Voici un exemple de shellcode (il ne s'agit pas d'un shellcode en vérité mais on supposera que si !) encodé par Shikata Ga Nai avec le décodeur en préfixe :

```
00: b8 b8 23 0f f8    mov eax, 0xf80f23b8
05: da d1             fcmovbe st,st(1)
07: d9 74 24 f4       fnstenv [esp-0xc]
0b: 5d                pop ebp
0c: 31 c9             xor ecx,ecx
0e: b1 24             mov cl, 0x24
10: 31 45 12          xor DWORD PTR [ebp+0x12], eax
13: 83 ed fc          sub ebp, 0xfffffffc
16: 03                ???
17: 45 0e e2 f5       ???
1b: 3c 20 a2 1a       ???
...
```

Voici la ligne de commande permettant d'obtenir cela (shellcode.bin contient le code binaire du shellcode) :

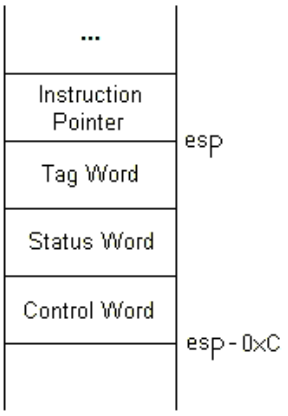
```
cat shellcode.bin | \
msfvenom -e x86/shikata_ga_nai -i 1 -a x86 --platform linux -f elf
```

Il y a deux points importants à remarquer pour la suite de l'analyse :

- `mov eax, 0xf80f23b8` : la clé est écrite en clair dans le code, une fois qu'on aura comprit le décodeur il sera très simple de décoder le shellcode.
- `fnstenv [esp-0xc]` : le point fixe dont je vous parlais précédemment

Essayons maintenant de comprendre le décodeur :

- `mov eax, 0xf80f23b8` : la clé (0xf80f23b8) est écrite dans le registre EAX.
- `fcmovbe st,st(1) ; fnstenv [esp-0xc]` : fnstenv est généralement utilisée pour gérer les exceptions (erreurs) lors d'une exécution, l'instruction enregistre des informations à la position pointé par l'opérande et notamment la valeur du registre EIP sauvegardée lors du dernier appel de fonction ou dernière instruction du type "FP-instruction" exécutée. fcmovbe est justement une de ces instructions particulières.



— `pop ebp` : récupère la valeur de l'EIP, plus précisément l'adresse de l'instruction fcmovbe précédente (ici l'offset 0x05)
— `xor ecx,ecx ; mov cl, 0x24` : le registre ECX est appelé "compteur", il contient la valeur qui défini le nombre d'itération à effectuer lors d'une instruction loop. Il est initialisé à 0 puis contient le nombre d'itérations à effectuer pour décoder entièrement le shellcode
— `xor DWORD PTR [ebp+0x12], eax` : c'est l'opération de décodage. Un XOR est effectué entre le registre EAX (qui contient la clé) et les 4 octets positionnés à `ebx+0x12`. Le registre EBX contient l'adresse de l'offset 0x05, additionnés on obtient 0x17 ce qui correspond au début de la partie encodée.
— `sub ebp, 0xfffffffc` : est équivalent à `add ebp, 0x4`, cette instruction déplace en fait le pointeur 4 octets plus loin pour que la prochaine itération décode les 4 octets suivants.

Ceux qui auront bien suivi l'histoire se diront peut-être "Mais, il manque des choses ? !", essayons tout d'abord d'exécuter ces instructions. [...] Les 4 premiers octets sont maintenant décodés, nous obtenons alors :

```
00: b8 b8 23 0f f8    mov eax, 0xf80f23b8
05: da d1             fcmovbe st,st(1)
07: d9 74 24 f4       fnstenv [esp-0xc]
0b: 5d                pop ebp
0c: 31 c9             xor ecx,ecx
0e: b1 24             mov cl, 0x24
10: 31 45 12          xor DWORD PTR [ebp+0x12], eax
13: 83 ed fc          sub ebp, 0xfffffffc
16: 03 45 0e          add eax, DWORD PTR [ebp+0xe]
19: e2 f5             loop 0x10
1b: 3c 20 a2 1a       ???
...
```

Il s'agit bien de la fin de notre décodeur !

Étant donné que l'encodage se fait 4 octets par 4 octets, si le shellcode avait une taille non multiple de 4 alors on obtiendrait une erreur lors de l'exécution de l'encodage. Parmi les solutions possible, celle consistant à compléter le shellcode à encoder par la fin du décodeur a été choisi afin d'ajouter en protection contre la détection antivirus.
En effet si ce n'était pas le cas il y aurait un second point fixe : `loop 0x10`. Dans le cas où le shellcode a une taille multiple de 4, les 4 derniers octets du décodeur sont d'abord encodés afin de toujours avoir au moins le dernier octet du décodeur encodé, c'est le cas que nous avons ici. Nous avons donc deux nouvelles instructions :

- `add eax, DWORD PTR [ebp+0xe]` : la clé (registre EAX) est additionnée avec les 4 octets décodés, octet par octet. A noter que pour le XOR nous avions `ebx+0x12`, mais entre temps le registre EBX a été incrémenté de 4, il s'agit donc bien des 4 octets décodés !
- `loop 0x10` : le registre ECX est décrémenté et on recommence à partir de l'offset 0x10, ce qui correspond aux étapes de décodage, jusqu'à ce que le registre ECX ait la valeur 0.
- shellcode : une fois le registre ECX à 0 on arrête les itérations et les instructions suivantes, c'est-à-dire le shellcode, sont exécutées.

Modification de la clé J'insiste à nouveau sur le fait que la clé doit être additionnée octet par octet. Dans notre cas la clé vaut `b8 23 0f f8` et les octets décodés sont `45 0e e2 f5`, si on additionne les 4 octets en une fois on obtient `3d 22 a2 1d` mais voici ce qu'on doit obtenir :

```
f8 + 45 = (1)3d     13 + 0e + 01 = 22
bf + e2 = (1)a1     28 + f5 + 01 = (1)1e
```

Nouvelle clé : `3d 22 a1 1e`

Polymorphisme Parlons maintenant de ce qui différencie Shikata Ga Nai de la plupart des encodage. Voici le cas général du décodeur :

```
Mov reg1, Key              # Possible position de la cle : 5 octets
FP-instruction             # Sauvegarde EIP : 2 octets
Mov reg1, Key              # Possible position de la cle : 5 octets
Fnstenv [esp-0xc]          # Recupere l environnement : 4 octets
Mov reg1, Key              # Possible position de la cle : 5 octets
Pop reg2                   # Extrait la sauvegarde de EIP: 1 octet
Mov reg1, Key              # Possible position de la cle : 5 octets
Xor ecx, ecx               # ECX initialise a 0 : 2 octets
Mov reg1, Key              # Possible position de la cle : 5 octets
Mov cl, nb_loop            # Nombre d iterations : 2 octets
Mov reg1, Key              # Possible position de la cle : 5 octets
Xor DWORD PTR[reg2 + N], reg1  # Decode 4 octets : 3 octets
Add reg2, 4                # Passe aux 4 octets suivants : 3 octets
Add reg1, DWORD PTR[reg2 + N-4] # Modifie la cle : 3 octets
Loop 0x10
```

Voici donc les possibilités utilisées :

- registres : 2 utilisés parmi 6
- `Mov reg1, key` : 6 positions
- FP-instruction : 4 instructions
- `Xor ecx, ecx` : 4 écritures
- `Mov cl, nb_loop` : 2 écritures (explications ci-dessous)
- `Xor - Add - Mod` : 3 permutations (Add-Xor-Mod, Xor-Add,Mod, Xor-Mod-Add)

Au total on obtient 17280 décodeurs différents pour Shikata Ga Nai, autant dire qu'il est compliqué de faire une signature pour chacun ...

Registre ECX Dans notre exemple le décodage s'effectue en 0x24 (36) itérations, le sous-registre CL qui est utilisé est alors suffisant (1 octet). Si par contre le shellcode avait une taille supérieure à 1024 octets il aurait fallu 1024/4 = 256 itérations, soit 0x100 : il faut 2 octets pour contenir ce nombre ! Dans ce cas on utilise le sous-registre CX qui peut contenir 2 octets et on obtient l'instruction `mov cx, nb_loop` dont la valeur hexadécimale fait 4 octets (la taille des instructions aura de l'importance pour la suite).

Maintenant que tout est dit sur le fonctionnement du décodeur on va passer à la méthode que j'ai mit au point pour décoder en statique le shellcode et qui en même temps permet de vérifier qu'il ne s'agit par d'un faux positif.

4 Shikata Deshita

La première étape consiste à trouver l'instruction `fnstenv [esp-0xc]` et de s'assurer qu'il s'agit bien d'une instruction (cette vérification est important et sera effectuée pour chaque élément recherché). En effet, le décodeur contient une clé aléatoire de 4 octets, on peut donc très bien imaginer (avec un peu de malchance) que les octets de la clé correspondent aux octets de l'instruction recherchée. De la même façon, le nombre d'itérations de décodage (1 ou 2 octets) peut contenir les octets des instructions recherchées.

```
bed97424f4          mov esi,0xf42474d9      33c9                xor ecx,ecx
dbda                fcmovnu st,st(2)        66b933c9            mov cx,0xc933
d97424f4            fnstenv [esp-0xc]
```

La vérification du `fnstenv` consiste à s'assurer que l'instruction suivante soit `mov, reg, key` ou `pop reg`. En effet, la recherche d'instruction se fait par incrémentation de la position, on ne pourra donc confondre la clé avec la vraie instruction que si le `mov reg, key` se trouve avant celle-ci, ce qui implique que l'instruction qui suivra sera soit la `FP-instruction` soit le vrai `fnstenv`.

Dans les étapes suivantes nous rechercherons les instructions à des positions fixes par rapport au `fnstenv` pour que, si l'instruction recherchée n'est pas présente là où elle est sensée être dans le décodeur, on puisse affirmer qu'il s'agit d'un faux-positif. Si toutes les instructions et informations ont pu être récupérées alors on put effectuer la procédure de décodage.

Voici le code écrit en C qui effectue cette première étape :

```c
long isItShikata (unsigned char *encoded, long file_size)
{
   long i;
   int j;
   unsigned char registers[6] = {0x8, 0xa, 0xb, 0xd, 0xe, 0xf};
   unsigned char mov_key = 0xb0, pop = 0x50;

   for (i = 0; i < file_size - 4; i++) // fnstenv : 4 bytes
      if (encoded[i] == 0xd9 && encoded[i + 1] == 0x74 &&
          encoded[i + 2] == 0x24 && encoded[i + 3] == 0xf4)
         for (j = 0; j < 6; j++)
            if (encoded[i + 4] == mov_key + registers[j] ||
                encoded[i + 4] == pop + registers[j])
               return i;
   return -1; // Error
}
```

La deuxième étape consiste à récupérer le nombre d'itérations à effectuer pour décoder complètement le shellcode, qui est l'argument de l'instruction `mov cl, X` ou `mov cx, X`. Ces instructions peuvent se trouver à 7 ou 12 octets du fnstenv (`fnstenv`(4) + `pop reg`(1) + `xor ecx, ecx`(2) + `mov reg, key`(5)).

Pour effectuer la vérification on s'assurera qu'il y a l'instruction `xor, ecx, ecx` 2 ou 5 octets plus haut. En effet, pour que la clé trompe la recherche (7 octets après le `fnstenv`), elle doit se trouver avant cette instruction, on ne pourra donc pas la trouver avant s'il s'agit de la clé.

```
d97424f4
58
bd00b14203
33c9
b142
```

```
long nbLoop (unsigned char *encoded, long location)
{
   int i, j, j_key;
   unsigned long concatCX, concatXOR1, concatXOR2;
   unsigned long XOR[4] = {0x31c9, 0x29c9, 0x33c9, 0x2bc9};

   j = location + 7;
   if (encoded[j] == 0xb1)
   {
      concatXOR1 = (encoded[j - 2] << 8) + encoded[j - 1];
      concatXOR2 = (encoded[j - 7] << 8) + encoded[j - 6];
      for (i = 0; i < 4; i++)
         if (concatXOR1 == XOR[i] || concatXOR2 == XOR[i])
            return encoded[j + 1];
   }
   j_key = location + 12;
   if (encoded[j_key] == 0xb1)
      return encoded[j_key + 1];
   concatCX = (encoded[j] << 8) + encoded[j + 1];
   if (concatCX == 0x66b9)
   {
      concatXOR1 = (encoded[j - 2] << 8) + encoded[j - 1];
      concatXOR2 = (encoded[j - 7] << 8) + encoded[j - 6];
      for (i = 0; i < 4; i++)
         if (concatXOR1 == XOR[i] || concatXOR2 == XOR[i])
            return (encoded[j + 3] << 8) + encoded[j + 2];
   }
   concatCX = (encoded[j_key] << 8) + encoded[j_key + 1];
   if (concatCX == 0x66b9)
      return (encoded[j_key + 3] << 8) + encoded[j_key + 2];
   return 0;
}
```

On va ensuite rechercher la clé. L'instruction qui la contient, `mov reg, key`, a 6 positions possibles (le schémas explicitant le polymorphisme dans la partie 3.2 les répertorie) et peut utiliser l'un des 6 registres disponibles : EAX (`0xb8`), EDX (`0xba`), EBX (`0xbb`), EBP (`0xbd`), ESI (`0xbe`), EDI (`0xbf`).

Étant donné qu'on ne recherche qu'un seul octet pour identifier l'instruction, on pourrait imaginer confondre avec le nombre d'itérations (étape précédente). Ceci ne peux arriver que si on recherche la clé en pensant que l'instruction est `mov cx, X` (4 octets) alors qu'il s'agit en fait de `mov cl, X` (2 octets) :

```
d97424f4              fnstenv [esp-0xc]
5b                    pop ebx
33c9                  xor ecx,ecx
66b9b800              mov cx,0xb8
b801020304            mov eax,0x4030201
```

Ce cas est donc facilement évité en précisant à la fonction si le nombre d'itération était supérieur ou inférieur à 255.

Voilà le code écrit en C qui s'occupe de cette partie :

```
int findKey (unsigned char *encoded, long location, int extend)
{
   unsigned char mov_key[6] = {0xb8, 0xba, 0xbb, 0xbd, 0xbe, 0xbf};
   int pos_key[6] = {-7, -5, 4, 5, 7, 9};
   int i, j;

   // nb_loop > 255
   if (extend == 1)
      pos_key[5] += 2;

   for (i = 0; i < 6; i++)
      for (j = 0; j < 6; j++)
         if (encoded[location + pos_key[i]] == mov_key[j])
            return location + pos_key[i] + 1;
   return -1;
}
```

La prochaine étape va nous donner la position des premiers 4 octets à décoder, on va d'abord rechercher la `FP-instruction` pour obtenir la position de référence, puis l'instruction `xor` du décodage pour obtenir la position relative, la somme des deux donnera le résultat attendu.

`FP-instruction` se trouve 2 octets ou 7 octets avant le `fnstenv`. Cette instruction pose problème étant donné qu'elle est en début du décodeur, si on commence à chercher à la position "-2" mais que la clé se trouve entre les deux instructions il n'y a pas de méthode évidente pour le détecter. De même, si on commence à chercher en "-7" mais que la clé se troue après le `fnstenv`, on tombe en dehors du décodeur (en supposant qu'il s'agit d'octets aléatoire on pourrait tomber sur les octets de l'instruction et donc fausser la position du décodage). La solution consiste à transmettre à la fonction la position de la clé (étape précédente) pour déterminer dans quel cas on se trouve.

L'instruction `xor` se trouve à 9 ou 14 octets (nombre d'itération <= 255) ou à 11 ou 16 octets (nombre d'itération > 255). Si c'est l'instruction `add` (déplacement) qui s'y trouve alors le `xor` est 3 octets plus loin. Si on a l'information sur le nombre d'itération et sur la position de la clé alors il n'y qu'une seule position possible et il devient impossible de se tromper.

Voilà le code écrit en C qui s'occupe de cette partie :

```
int findBegin (unsigned char *encoded, long location, int extend,
    long key)
{
   int i;
   long pos, begin = 0, tmp = 0;
   unsigned char FPi[4] = {0xd9, 0xda, 0xdb, 0xdd};

   pos = location - 2;
   if (key - location == -4)
      pos -= 5;

   for (i = 0; i < 4; i++)
      if (encoded[pos] == FPi[i])
      {
         begin = pos;
         break;
      }
   if (begin == 0)
      return -1;
```

```
   pos = location + 9 + 2 * extend;
   if (key > location)
      pos += 5;

   if (encoded[pos] == 0x31)
      tmp = encoded[pos + 2];
   else
      if (encoded[pos] == 0x83 &&
           encoded[pos + 3] == 0x31)
         tmp = encoded[pos + 5] + 4;

   if (tmp == 0)
      return -1;
   else
      return begin + tmp;
}
```

Maintenant que toutes les informations ont pu être récupérées avec succès nous somme passé d'une signature de 4 octets (`fnstenv`) à une signature entre 8 et 13 octets (en fonction de la position de la clé et du nombre d'itération) qui ont tous une position précise les uns par rapport aux autres sur une surface de 19 à 24 octets. La probabilité d'avoir un faux-positif n'est pas nul (une image ou tout autre contenu aléatoire pourrait correspondre au décodeur sans en être un) mais elle est devenue suffisamment faible pour que le processus d'analyse de shellcode, beaucoup plus couteux en temps de calcul, se charge d'identifier les faux résultats.

On va donc s'occuper de décoder le shellcode en n'oubliant pas de modifier la clé après chaque itération de décodage.

Il ne faut pas oublier que 1 à 4 des derniers octets du décodeur sont encodés, j'ai donc prit soin de ne stocker le résultat du décodage qu'une fois l'instruction `loop 0x10` (0xe2f5) décodée.

```
long decode (unsigned char *encoded, long begin, unsigned char
    *decoded, long nb_loop, unsigned char *key)
{
   int i, j = 0, loop, decoding = 0, carry, tmp;
   unsigned char result[4];

   if (encoded[begin - 1] == 0xe2)
      decoding++;

   for (loop = 0; loop < nb_loop; loop++)
   {
      for (i = 0; i < 4; i++)
      {
         result[i] = encoded[begin + i] ^ key[i];
         if (decoding == 2)
         {
            decoded[j] = result[i];
            j++;
         }
         else
            if (decoding == 0 && result[i] == 0xe2)
               decoding++;
            else
               if (decoding == 1 && result[i] == 0xf5)
                  decoding++;
      }
      carry = 0;
      for (i = 0; i < 4; i++)
      {
         tmp = key[i] + result[i] + carry;
         if ((tmp >> 8) != 0)
            carry = 1;
         else
            carry = 0;
         key[i] = tmp & 0xFF;
      }
      begin += 4;
   }

   return j;
}
```

5 Résultats & Améliorations

Que le shellcode fasse 20 ou 1000 octets, il ne faudra que quelques millisecondes (3-4 ms) pour s'assurer qu'il ne s'agit pas d'un faux-positif et le décoder ! Cette vitesse d'exécution nous permet de lui transmettre tout flux de données contenant les octets `0xd9 0x74 0x24 0xf4` (on peut donc imaginer pouvoir vérifier 300 fichiers par seconde en utilisant un seul thread) avant de transmettre les résultats positifs à l'analyseur de shellcode.

Il ne s'agit néanmoins que d'une version basique qui ne décode que les shellcodes encodés une fois par cet algorithme de Shikata Ga Nai. En effet un shellcode peut être encodé plusieurs fois et par plusieurs encodages différents. Il faut donc prendre en compte le fait que ce qu'on a décodé peut encore contenir un binaire encodé en recherchant la présence d'un autre décodeur.

Un autre genre de problème que l'on peut rencontré, plus spécifique à Shikata Ga Nai, est la génération de clé par contexte. Si l'attaquant a accès à une information fixe (qui ne va pas changer d'ici à l'exécution du shellcode) sur la machine ciblée, alors il peut s'en servir pour générer une clé non-aléatoire. Il devra alors mettre en préfixe du décodeur son générateur de clé qui ira chercher cette information sur la machine.

Cette méthode a deux avantages pour l'attaquant : le fait que la clé ne soit plus écrite en dur dans le décodeur, et que le shellcode ne puisse être décodé (et donc exécuté) que sur la machine ciblée.

De notre point de vu on peut supposer se trouver dans un tel cas si la clé est le seul élément qu'on ne parvient pas à trouver mais on se verra confronté à deux difficultés : trouver quelle est l'information recherchée par la génération de clé et récupérer cette information pour générer la clé et effectuer le décodage.

Il faut aussi savoir que l'agencement du décodeur est légèrement différent et, même si les sets d'instructions sont les mêmes, l'ordre peut être différent et il faudra revoir le code précédent pour être sûr de les récupérer.

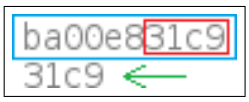
Nous avions évoquer dans la deuxième partie que l'une des façons de bypasser les antivirus était la modification de l'existant. On peut imaginer trois manières de modifier l'algorithme de Shikata Ga Nai sans pour autant changer ses fondements :

— Modifier les sets d'instructions
— Modifier l'ordre des instructions
— Ajouter de l'obfuscation (ajout d'instructions n'ayant aucun impact sur le résultat)

Dans le premier cas il faudrait ajouter ces nouvelles instructions dans les étapes décrites précédemment, mais, pour cela, encore faut il savoir lesquelles !

Le problème que l'on va rencontrer avec l'obfuscation n'est pas les instructions qui sont ajoutées, mais surtout le fait qu'elles vont décaler toutes celles que l'on recherche ! Ainsi, pour les deux derniers cas, il faudra complètement revoir la méthode et passer à une recherche par récursivité.

Cette méthode consiste en fait à rechercher les instructions une à une, n'importe où à une certaine distance autour du `fnstenv`, et, par un système de "visualisation" du code, repérer si les instructions trouvées sont superposées.



Nous avons ici identifié les instructions `mov, reg, key` (cadre bleu) et `xor ecx, ecx` (cadre rouge) mais on remarque qu'elles partagent un ou plusieurs octets (en terme de position). On comprend que les octets qu'on pensait être l'instruction `xor ecx, ecx` sont en fait ceux de la clé, on va donc continuer la recherche et finir par trouver la bonne (flèche verte).

Maintenant que vous savez comment venir à bout des encodages de type Shikata Ga Nai, il ne vous reste plus qu'à faire de même pour tous les autres !

Quelques liens sur les shellcodes :

- **Tutoriel** http ://www.vividmachines.com/shellcode/shellcode.html
- **Tutoriel plus évolué** http ://hackerforhire.com.au/part-1-disassembling-and-understanding-shellcode/
- **DB de shellcodes** http ://shell-storm.org/shellcode/
- **DB de shellcodes** https ://www.exploit-db.com/
- **Encodage** http ://www.bases-hacking.org/xored-shellcode.html

Et concernant Métasploit & Shikata Ga Nai :

- **Métasploit** http ://www.metasploit.com/
- **MSFVenom** https ://www.offensive-security.com/metasploit-unleashed/msfvenom/
- **Github** https ://github.com/rapid7/metasploit-framework/tree/master/
- **Explication des contextes** http ://census.gr/media/context-keying-slides.pdf