Univesity of Bordeaux
College of Science & Technology
351 Cours de la Liberation
33400 Talence

- Projet M2 -

Analyse de malware statique et dynamique : Les outils et quelques cas pratiques

Erwan GRELET & Amélie GUÉMON

Master 2 CSI — Cryptology & Computer Security

4 mars 2017

Table des matières

Introduction

Non sans étonnement, l'année 2016 aura été l'année du boom des ransomwares. Avec 1,3 millions de nouveaux ransomwares détectés lors du second trimestre par les équipes de McAfee et plus de 90% des emails de phishing présentant des ransomwares [1] , la menace n'est pas à prendre à la légère. En début d'année, un hôpital californien avait été la cible d'un ransomware, chiffrant les données médicales de près de milliers de patients. L'hôpital aura finalement versé 17.000$ en bitcoins (40 bitcoins à cette époque) afin de continuer à pouvoir soigner ses clients (les auteurs ont accepté un prix plus faible, la rançon originelle était de 3.4 millions de dollars) [2]. La monétisation d'informations volées ou verrouillées peut rapporter gros aux auteurs de ces attaques.

Mais les ransomwares ne sont pas les seuls malwares à surveiller : ils ne représentent qu'une partie d'une menace bien plus importante. Des spywares (rootkits, trojans, keyloggeurs) aux downloaders et botnets, les menaces sont variées et les cibles de plus en plus diversifiées, avec l'essor des objets connectés (IoT) et des mobiles. Ces derniers ne sont pas toujours sécurisés et en font des cibles de choix.
Il est donc important de constamment étudier ces nouvelles menaces, afin de pouvoir protéger les systèmes d'informations et les données qu'ils contiennent. L'analyse de malware sert typiquement trois causes : l'obtention d'informations nécessaires à la réponse à une intrusion, l'extraction d'indicateurs de compromission ou simplement à la compréhension et la découverte des dernières techniques utilisées par les fabricants de malwares.

Dans ce papier seront donc présentées, en première partie, les techniques d'analyses de malware, puis deux études de cas pratiques : le botnet Ganiw et le ransomware SageCrypt.

1. https://phishme.com/phishing-ransomware-threats-soared-q1-2016/
2. http://khn.org/morning-breakout/california-hospital-held-hostage-by-hackers-pays-17000-ransom-to-unlock-records/

Chapitre 1

Techniques d'analyses de malwares

Lors de l'analyse d'un logiciel suspect, le code source est rarement disponible et le seul support de travail est l'exécutable fournit. Pour appréhender son comportement, il est alors nécessaire de recourir à l'utilisation de nombreux outils et astuces.

Il existe deux approches à l'étude d'exécutable au comportement inconnu : l'analyse statique et l'analyse dynamique.
L'analyse statique consiste à examiner le binaire sans l'exécuter, tandis que l'analyse dynamique se fait au cours de son exécution. Ces deux approches sont en fait complémentaires et permettent d'obtenir une vision d'ensemble sur le comportement du binaire étudié.

1.1 Analyse statique

L'analyse statique est donc une méthode d'étude reposant uniquement sur la lecture du binaire, sans l'exécuter. Celle-ci se base sur l'étude des propriétés du binaire, de son code assembleur et permet de récupérer des informations sur l'élément suspect sans risquer de contaminer ou de modifier de manière involontaire son environnement de travail.

1.1.1 Scanner antivirus

Une des premières et plus répandues techniques d'analyse, quelles que soient les connaissances en informatiques, est le passage par un antivirus. Celui-ci permet d'identifier, neutraliser et même éliminer de nombreux logiciels malveillants.

Ils reposent principalement sur 3 techniques :
- Comparaison des signatures virales ;
- Analyse du comportement des binaires suspectés ;
- Analyse de forme, à partir de règles regexp (très utilisés pour les mails).

Si le logiciel incriminé est identifié, alors celui-ci est connu et reconnu depuis déjà un certain temps par les bases de données des anti-virus : des études devraient être disponibles sur le net.

1.1.2 Hash du sample et comparaison aux bases de données

Les signatures utilisées par les anti-virus sont des identifiants propres à chaque binaire, à une portion de code ou à un comportement particulier. Les techniques et heuristiques utilisées dépendent des types de scanner mais vont du simple hash de binaire aux algorithmes d'analyse de comportement. Cette dernière permet de trier les logiciels suspects par familles, en fonction de leur comportement (phishing, modification de registres, ...) mais aussi d'en analyser et détecter certains qui sont alors encore inconnus, simplement par leurs actions.

Dans le cas de logiciels détectés comme malveillants mais encore inconnus, de nouvelles signatures doivent être générées. En effet, bien que le comportement soit détecté comme suspect, celui-ci n'est pas encore totalement étudié et peut présenté des principes de protections (comme de réplication par exemple) qu'il sera nécessaire de déterminer afin de développer de nouveaux outils et de débarrasser totalement les victimes de leurs charges virales.

1.1.3 Examen de l'exécutable

Quel que soit le binaire, celui-ci va fournir des informations relative à son exécution, comme l'architecture supportée, la version (32/64 bits), l'endianness, le point d'entrée (si celui-ci n'est pas le *main*), ...Ces informations ne sont pas stockées dans le corps du binaire mais dans une en-tête, de manière différentes en fonction des types d'exécutables. Mais les mêmes informations seront présentes, quels que soient les supports utilisés.

Parmi les formats présents, les plus répandus sont les **ELF**, **PE** et **Mach-O**.

• ELF (Extensible Linking Format, ou plus formellement, Executable and Linkable Format) est le format le plus présent dans les systèmes d'exploitation de type Unix, excepté pour Mac OS X, permettant de représenter les exécutables, les fichiers objets, les bibliothèques partagées comme les core dumps.

Les informations peuvent être récupérées par les fonctions *file*, *readelf*, ...

```
$> readelf -h ch23
En-tête ELF:
  Magique:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Classe:                            ELF32
  Données:                           complément à 2, système à octets de
poids faible d'abord (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  Version ABI:                       0
  Type:                              EXEC (fichier exécutable)
  Machine:                           Intel 80386
  Version:                           0x1
  Adresse du point d'entrée:         0x8048450
  Début des en-têtes de programme :  52 (octets dans le fichier)
  Début des en-têtes de section :    2404 (octets dans le fichier)
  [...]
```

FIGURE 1.1 – Données comprises dans l'en-tête d'un binaire ELF

D'autres informations peuvent être trouvées dans le binaire, notamment les bibliothèques et fonctions importées/exportées. Celles-ci sont extrêmement utiles pour l'analyse d'un binaire, afin d'avoir une idée de son comportement sans l'exécuter.

• PE (Portable Executable) est le format de fichier des exécutable et des bibliothèques sur les systèmes Windows. Il sert à décrire les binaires (.exe), les bibliothèques dynamiques (.dll) et les pilotes (.sys).

Ces informations peuvent être récupérées de nombreuses manières.
Voici un extrait du résultat de la commande *pedump*, sur Linux :

```
$> pedump ch22.exe
PE Header:                                NT Header:
         Magic (0x010b): 0x010b              Image Base (0x400000): 0x00400000
             LMajor (6): 0x0b            Section Alignment (8192): 0x00002000
             LMinor (0): 0x00               File Align (512/4096): 0x00000200
              Code Size: 0x00003000                   OS Major (4): 0x0004
  Initialized Data Size: 0x00003400                                 [...]
Uninitialized Data Size: 0x00000000
        Entry Point RVA: 0x00004f3e
    Code Base RVA: 0x00002000
  Data Base RVA: 0x00006000
```

FIGURE 1.2 – Données comprises dans l'en-tête d'un binaire PE

• Mach-O pour les systèmes Mac OS X.

1.1.4 Recherche de chaînes de caractères

Une des manières d'analyser un exécutable est aussi d'observer les caractères imprimables présents au milieu du code. En effet, si celui-ci n'est pas obfusqué, les chaînes de caractères écrites "en brut" ou les fonctions utilisées comme les sections peuvent être retrouvées assez facilement. Avec la commande *strings*, il est possible d'afficher toutes les chaînes d'au moins 4 caractères ascii.

```
$> /tmp/HB3$ strings ch23
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
strncpy
__stack_chk_fail
printf
strlen
memset
__libc_start_main
[...]
Usage : %s <your name>
[...]
.init
.text
[...]
.data
.bss
[...]
main
_init
```

FIGURE 1.3 – Résultat de la commande *strings* sur un binaire

Il est également possible d'utiliser d'autres outils comme *nm*, qui peuvent fournir des informations supplémentaires comme le nom des fonctions et librairies utilisées, ainsi que les variables globales utilisées.

1.1.5 Packer et techniques de protection du binaire

Mais l'analyse statique peut faire face à un certain nombre d'obstacles. En effet, celle-ci repose essentiellement sur le principe de lecture de code et en est donc entièrement dépendante. Si le binaire se trouve, d'une quelconque manière, obfusqué, que son code est modifié au cours de l'exécution ou simplement que les données originales sont altérées, alors sa lecture et sa compréhension sont bien plus compliquées.

Ainsi, il est possible de se retrouver face à un logiciel qualifié de "packé". Cette partie s'attardera uniquement sur ceux obfusquant le comportement du binaire, même si ils n'ont pas tous vocation à le faire (comme ceux se basant uniquement sur un principe de compression de données).

Une de ces techniques consiste à chiffrer la charge utile, malveillante (le .text ou le .data par exemple). Le code suspect est alors composé de la routine de déchiffrement et du payload chiffré. La clef, quant à elle, pourra être stockée dans la routine de déchiffrement, le payload ou encore récupérée depuis l'extérieur : depuis un fichier ou par une requête vers un serveur.

De même, il existe plusieurs méthodes de chiffrement, allant du XOR à celles utilisant les courbes elliptiques, du chiffrement unique aux chiffrés imbriqués, bloc par bloc ou du payload entier (voir figure 1.4).

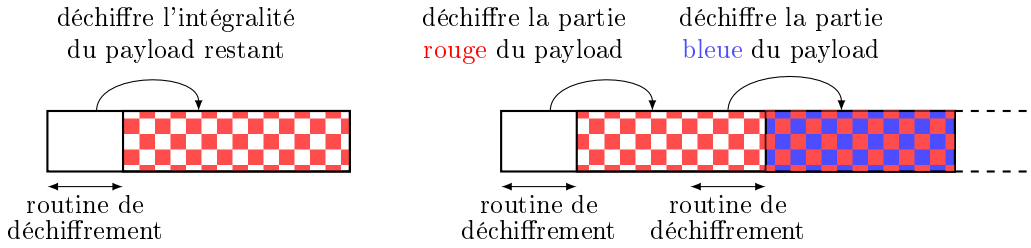


FIGURE 1.4 – Différentes formes de packer chiffrant

En règle générale, le clair obtenu lors du déchiffrement sera écrit dans un nouveau fichier (avec les appels systèmes correspondant, pour chque système) ou dans le binaire lui-même, si celui-ci est libre en écriture.

Mais il existe aussi d'autres formes de packer, comme ceux dits transformant qui vont, par exemple, transformer leur code originel en code à fournir à une VM embarquée, qui est donc incompréhensible sans l'exécuter ou connaître les instructions de l'architecture utilisée.

Enfin, il existe aussi des techniques de protections, ralentissant ou empêchant la rétro-ingénierie d'un binaire, de manière passive sur le code :

- Binaire "stripped" : aucun symbole de débugage ;
- Détection de débugueur, de breakpoints (opcode 0xcc) ;
- Anti-virtualisation (VM) ;
- Anti-dumping ;
- Anti-tampering (vérification de checksum, ...) ;
- ...

1.1.6 Analyse du code assembleur

Le plus gros travail se porte sur la lecture de code. Bien souvent, le sample a étudier ne fournit pas son code source et il faut donc lire son code assembleur.

Celui-ci n'est pas donné nativement avec l'exécutable et il est nécessaire de le récupérer à l'aide d'un désassembleur, à partir du langage machine. De nombreux outils existent, quelles que soient les plateformes utilisées : *OllyDbg* sur Windows, *gdb* et *objdump* sur Linux, ...Certains sont multi-plateformes et proposent des services supplémentaires pour la rétro-ingénierie. Parmi les plus connus, sont disponibles *Radare2* et *IdaPro*, ce dernier proposant par exemple, un décompilateur pour les codes en C/C++.

```
|  ,======< 0x08048664      741c           je 0x8048682
|  ||||||   0x08048666      8d442438       lea eax, dword [esp + local_38h] ; 0x38 ; "4" ; '8' ; "4"
|  ||||||   0x0804866a      89442408       mov dword [esp + local_8h], eax
|  ||||||   0x0804866e      8b442430       mov eax, dword [esp + local_30h] ; [0x30:4]=0x19001c ; '0'
|  ||||||   0x08048672      89442404       mov dword [esp + local_4h], eax
|  ||||||   0x08048676      8b44242c       mov eax, dword [esp + local_2ch] ; [0x2c:4]=0x280009 ; ','
|  ||||||   0x0804867a      890424         mov dword [esp], eax
|  ||||||   0x0804867d      e8b2feffff     call sym.insert
|  ||||||   ; JMP XREF from 0x08048611 (sym.main)
|  `````--> 0x08048682      8b442434       mov eax, dword [esp + local_34h] ; [0x34:4]=6 ; '4'
|     | ||   0x08048686      89442408       mov dword [esp + local_8h], eax
|     | ||   0x0804868a      c74424041400.  mov dword [esp + local_4h], 0x14 ; [0x14:4]=1
|     | ||   0x08048692      8d8424381000.  lea eax, dword [esp + local_1038h] ; 0x1038 ; ".5"
|     | ||   0x08048699      890424         mov dword [esp], eax
|     | ||   0x0804869c      e843fdffff     call sym.imp.fgets          ; char *fgets(char *s, int size, FILE *stream);
|     | ||   0x080486a1      85c0           test eax, eax
|     `====< 0x080486a3      0f856affffff   jne 0x8048613
|       ||   ; JMP XREF from 0x080485bf (sym.main)
|       ``-> 0x080486a9      b800000000     mov eax, 0
|            0x080486ae      8b94244c1000.  mov edx, dword [esp + local_104ch] ; [0x104c:4]=0x7368732e ; ".shstrtab"
|            0x080486b5      653315140000.  xor edx, dword gs:[0x14]
|         ,=< 0x080486bc      7405           je 0x80486c3
|         |   0x080486be      e881fdffff     call sym.imp.__stack_chk_failvoid);
|         `-> 0x080486c3      c9             leave
\            0x080486c4      c3             ret
[0x08048460]>
```

FIGURE 1.5 – Capture d'écran de Radare2

1.2 Analyse dynamique

L'analyse dynamique, en opposition à l'analyse statique, repose donc sur l'exécution du binaire suspect et l'observation des différentes actions et modifications apportées au système hôte. Cette technique peut être dangereuse pour l'environnement de travail et doit être correctement exécutée.

1.2.1 Machine virtuelle

L'élément à étudier peut ne pas correspondre à votre environnement et votre architecture : un malware Windows ne pourra affecter votre système si celui-ci est un GNU/Linux. C'est pourquoi il est essentiel de mettre en place un système de machine virtuelles.

De nombreux outils comme *VMware* ou *VirtualBox* (gratuit) permetent de pouvoir virtualiser un grand nombre d'architectures, sans pour autant modifier son environnement et offrent la possibilité de choisir , de configurer le système qui sera infecté (utile pour forcer certaines versions du noyau, la persistance de certaines failles de sécurité, ...).
Un autre avantage à l'utilisation de machines virtuelles est la capacité de pouvoir utiliser des snapshots. Ces "instantanés" vont capturer l'état de la machine virtuelle (mémoire, configurations, état des disques, ...) à un moment précis. Les modifications alors apportées à la machine seront indépendantes du snapshots, préservant l'état de la VM original. Si l'utilisateur a besoin de revenir en arrière sur ses actions, il lui suffit de restaurer son snapshot, autant de fois que nécessaire. Ceci est particulièrement utile lors de l'étude d'un malware, afin de pouvoir étudier plusieurs fois la phase d'infection ou la première communication avec un serveur C&C, par exemple.

1.2.2 Simulation de réseaux

Un autre avantage à l'utilisation de machines virtuelles comme environnement d'étude est la possibilité de relier plusieurs machines virtuelles entre elles, comme dans un vrai réseau, tout en gardant le contrôle sur chacune d'elles. Il est alors possible de créer totalement son environnement, de l'ordonner, de le gérer (création de serveur DNS par exemple), afin d'adapter l'environnement d'infection au malware et de pouvoir étudier les communications circulant sur le réseau.

INetSim est un outil open-source offrant ce service. Celui-ci permet de simuler des services internet qui sont habituellement utilisés par les malwares, dans un environnement de travail confiné. Parmi les modules proposés et qui peuvent être utiles, il est possible de trouver la mise en place de service DNS, FTP, HTTP, Daytime, ...

1.2.3 Analyse de trafic

Une fois l'environnement mis en place et que l'élément à analyser pense communiquer normalement avec l'extérieur, il est bien souvent intéressant d'observer et d'étudier ces communications. Ceci peut permettre d'inférer ou de confirmer un comportement vu lors de la lecture de code, ou au contraire, d'observer une interaction encore inconnue du malware avec son environnement.

L'analyse de trafic ethernet peut être effectué à l'aide d'un outil de capture de paquets libre comme *tcpdump* ou *Wireshark*, disponibles sur plusieurs architectures et systèmes. L'utilisateur peut alors utiliser un filtre (BPF pour tcpdump, par exemple) ou les fonctionnalités de *Wireshark* afin de disséquer plus simplement les paquets, de travailler sur des protocoles déjà connus ou encore, d'en découvrir et de les étudier.
Ce type d'étude peut être très pratique lors de la communication avec un serveur C&C.

1.2.4 Listing des processus

Une autre facette importante de l'infection à étudier, outre la communication réseau, est l'état de la machine au cours de l'infection.
La première chose qui vient à l'esprit est donc de regarder les processus actifs et leurs actions. C'est souvent ainsi que l'infection est détectée, lorsqu'un processus qui ne devrait pas être présent, l'est. Ainsi, une connection ssh ouverte depuis l'extérieur, vers la machine alors que seul un navigateur a été ouvert, peut témoigner de la présence d'une backdoor.

De nombreux outils sont disponibles, nativement, sur les systèmes d'exploitation, comme *ps* ou *pstree* sur Linux. Mais il existe aussi des outils propriétaires aussi bien utiles, comme *Process Explorer*, sur Windows, proposant des analyses plus complètes des processus actifs sur le système de l'utilisateur.

Quoi de plus naturel que d'ouvrir la liste des processus actifs afin de voir la consommation en CPU/mémoire, lorsque le système semble plus lent qu'à son habitude ?

1.2.5 Tracing

Mais il est aussi important de savoir les actions effectuées par le logiciel suspicieux (et ses différents threads et daemons, s'il y a).

Il est possible de tracer les appels systèmes effectués et les signaux reçus par un exécutable, sous linux, avec la commande *strace*. De même pour Windows, il est possible d'utiliser l'outil *Process Monitor*, gratuit, qui se charge d'afficher et de gérer l'état des fichiers du système, les interactions avec les registres Windows ainsi que l'utilisation et les appels de DLL.

1.2.6 Débugueur

Enfin, une des étapes de l'analyse de malware consiste à analyser celui-ci à l'aide d'un débugueur. Ces outils permettent de suivre l'exécution du malware, pas à pas, si aucune technique anti-debug n'est présente dans le binaire étudié. Cette technique est particulièrement utile lorsqu'il s'agit de connaître le contenu d'un prédicat opaque, d'une variable qui ne peut être appréhender facilement durant l'analyse statique, comme lors d'un embranchement dépendant de ce prédicat ou d'une phase de déchiffrement dépendant d'une variable contenant la clef.

Parmi les outils existant, la plupart des outils décrits dans la partie Analyse du code assembleur sont des débugueurs. Il peut être aussi utile de citer *ADB* (Androi Debug Bridge) pour les systèmes android et *x64dbg* pour Windows, qui est open-source.

Chapitre 2

Ganiw

2.1 Introduction

Le *sample* porté à l'étude est appelé **Ganiw** mais est aussi connu sous les nom de **BillGates** ou de combinaisons comme **LINUX.BACKDOOR.GATES** par les antivirus .
Celui-ci semble avoir été étudié pour la première fois en Février 2014 dans un post de *ValdikSS* sur le site *Хабрахабр* : *Studying the BillGates Linux Botnet*[1], mais continue tout de même à être étudié et à être actif.

Le malware Ganiw a été réalisé en C++ et pensé de manière à être modulaire et portable. Ce malware visait, à l'origine, uniquement les systèmes Linux, mais a été porté plus tard sur les systèmes Windows[2]
. Le choix des systèmes Linux n'est pas sans lien avec le développement toujours plus important des objets connectés dans l'"Internet of Things" (IoT). En effet, une grande partie des systèmes embarqués prenant part à l'IoT fonctionnent sur des distributions de GNU/Linux (voir OpenWrt[3] ou le projet Yocto[4]). Ces objets connectés sont, de manière générale, peu ou mal protégés (absence de firewall, mots de passe faibles,...) et non surveillés ou peu maintenus. Ce sont donc des cibles de choix pour la création de *botnets*, comme tente de le faire le malware **Ganiw**, pour mener des attaques de types DDOS[5] par exemple.

2.2 Analyse

2.2.1 Les outils

Les principaux outils utilisés pour réaliser cette analyse ont été le logiciel de virtualisation *VirtualBox*, le débugueur GNU *GDB*, l'utilitaire *strace*, l'analyseur de paquets *Wireshark*, le désassembleur *Radare2* et et le simulateur de services internet *INetSim* qui sont des logiciels libres.

2.2.2 Analyse basique

Avant toute chose, voilà, ci-dessous, le SHA-256 du sample analysé :

```
94f5fd896a526427a5ef1de37725e6eae3a06af3da098547f0adcfdd34fbfd2a
```

Pour commencer, il est bon d'utiliser l'utilitaire *file* pour en apprendre un peu plus sur l'exécutable :

```
ganiw: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked,
       for GNU/Linux 2.2.5, not stripped
```

C'est donc un exécutable au format ELF (Executable and Linkable Format) 32-bit pour Linux.
Il est intéressant de remarquer que l'exécutable est lié statiquement et surtout qu'il n'a pas été strip (ce qui va nous permettre d'obtenir des informations utiles telles que les noms de fonctions, méthodes ou variables potentiellement utilisées par le programme et donc nous permettre de comprendre plus facilement comment fonctionne le malware).

Un petit coup de *strings* nous permet de trouver quelques indices sur ce qui se passe lors de l'exécution du binaire, par exemple :

1. Traduit depuis le russe
2. `https://thisissecurity.net/2015/09/30/when-elf-billgates-met-windows/`
3. `https://openwrt.org/`
4. `https://www.yoctoproject.org/`
5. Distributed Denial-Of-Service

```
$> strings ganiw | less
[...]
/proc/meminfo
MemTotal:          %d kB
/proc/stat
cpu %llu %llu %llu %llu
/proc/net/dev
%7s %llu %lu %lu %lu %lu %lu %lu %lu %llu %lu %lu %lu %lu %lu %lu %lu
/proc/cpuinfo
processor
/proc/net/arp
%16s 0x%d 0x%d %20s %s
%2x:%2x:%2x:%2x:%2x:%2x
/proc/net/route
%5s %8x %8x %s
cpu MHz
cpu MHz          : %d.%d
vector::_M_insert_aux
%5s %s
/bin/netstat
/bin/lsof
/bin/ps
/bin/ss
/usr/bin/netstat
/usr/bin/lsof
/usr/bin/ps
/usr/bin/ss
/usr/sbin/netstat
/usr/sbin/lsof
/usr/sbin/ps
/usr/sbin/ss
GLIBCXX_FORCE_NEW
update_temporary
mkdir -p %s
cp -f %s %s
/tmp/notify.file
/usr/bin/
.lock
[...]
```

L'outil *strings* renvoit un grand nombre de chaînes de caractères, dont certaines intéressantes. Il y a peu de chance que l'exécutable soit packé ou compressé étant donné le nombre élevé de chaînes compréhensibles et claires présentes.

L'utilisation de *nm* permet en revanche de voir clairement les symboles présents dans le binaire. Puisque le malware a été réalisé en C++, il est préférable d'utiliser l'option *-C* de *nm* pour pouvoir lire facilement le nom des méthodes.
Voici, par exemple, ce qu'il est possible de trouver en utilisant *nm* :

```
$> nm -C ganiw | grep Main
080623f2 T MainBeikong()
08061c48 T MainMonitor()
080620ac T MainProcess()
08061d3c T MainSystool(int, char**)
08062304 T MainBackdoor()
08089398 T CThreadTns::ProcessMain()
08083ffe T CThreadDoFun::ProcessMain()
080883fc T CThreadShell::ProcessMain()
08089a3e T CThreadUpdate::ProcessMain()
0807fb5c T CThreadAtkCtrl::ProcessMain()
080865b4 T CThreadHttpGet::ProcessMain()
08087552 T CThreadLoopCmd::ProcessMain()
08087ac0 T CThreadRecycle::ProcessMain()
08087966 T CThreadMonGates::ProcessMain()
08086f2c T CThreadKillChaos::ProcessMain()
08088740 T CThreadTaskGates::ProcessMain()
08083aae T CThreadConnection::ProcessMain()
080848d8 T CThreadFakeDetect::ProcessMain()
08083870 T CThreadClientStatus::ProcessMain()
08083e22 T CThreadFXConnection::ProcessMain()
08088628 T CThreadShellRecycle::ProcessMain()
080808a6 T CThreadKernelAtkExcutor::ProcessMain()
08080db8 T CThreadNormalAtkExcutor::ProcessMain()
08066a8e T CManager::MainProcess()
```

```
08066788 T CManager::ZXMainProcess()
08100900 r MainSystool(int, char**)::C.1203
081008c0 r MainSystool(int, char**)::C.1206
```

Quelques noms intéressants apparaissent, il ne reste plus qu'à aller jeter un oeil de plus près.

Une analyse plus poussée est nécessaire pour avoir une idée concrète de ce que fait le malware étudié. Pour l'instant, impossible de savoir quels sont les liens entre les différentes fonctions et méthodes repérées précédemment. L'analyse continue donc avec Radare2, pour désassembler le binaire, VirtualBox, GDB et strace pour obtenir des valeurs à des endroits clés ou la liste des appels systèmes utilisés pendant l'exécution du malware.

Afin de faciliter la compréhension des relations qu'entretiennent les différents modules entre eux, une figure récapitulative est présentée ci-dessous :

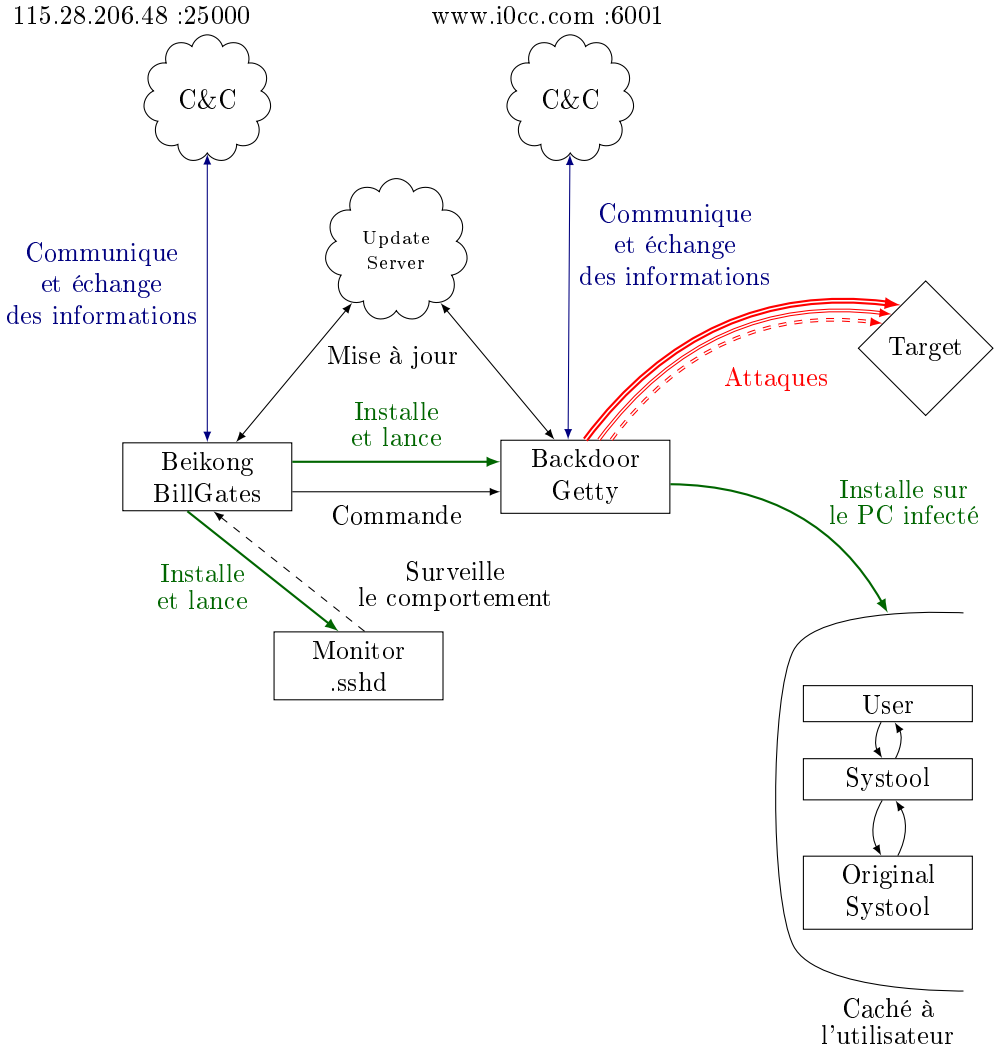


FIGURE 2.1 – Fonctionnement schématique de Ganiw

2.2.3 Main

Le point d'entrée du binaire se fait au niveau de la fonction *main*. Sans trop entrer dans les détails, il est aisé de remarquer la structure modulaire du malware 2.2.3.

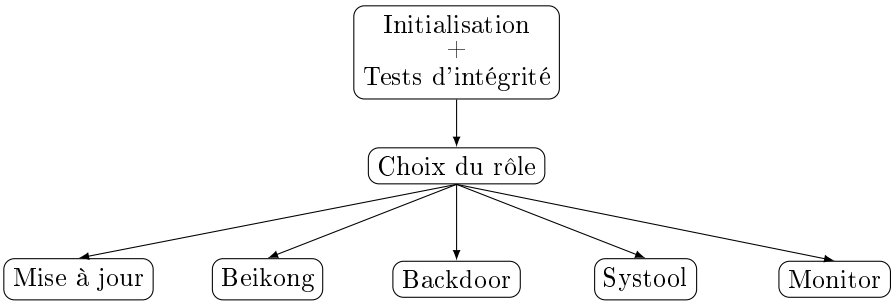


FIGURE 2.2 – Modules mis à disposition du malware

La fonction main commence donc par un ensemble de tests et d'initialisations de variables globales, afin de mettre en place le système.
Ci-dessous sont décrites précisément chacune de ces étapes, mais un pseudo-code récapitulatif peut être trouvé en annexes 4.

Fermer les descripteurs de fichiers

Tout d'abord, le malware va fermer les descripteurs de fichier compris entre 3 et 1023, laissant l'entrée standard, la sortie standard et l'erreur standard (stdin, stdout et stderr) accessibles.

Première initialisation de variables

La deuxième partie consiste en l'initialisation de certaines variables globales, depuis une chaîne de caractères quelque peu étrange. Dans le sample étudié, celle-ci correspond à :
"681A1C1543072E0140491F162F0B55545C55775F55565E57745E5D545652705D5E55585F70585C 5659577D5 C5F565C0423575B025A51720A56".

Une fonction de décodage (un simple XOR avec les octets de la clé), avec la clé 'Google', est appliquée à cette chaîne. Le code C++ associé est présent en annexes 4. Les informations obtenues sont alors stockées dans des variables (figure 2.3), qui serviront à définir le comportement futur du binaire.

g_Iud77	681A1C154 [...] 0423575B025A51720A56		
g_strMonitorFile	/usr/bin/.sshd	g_uHiddenPt	30000
g_iFileSize	1223123	g_iHardStart	772124
g_iSoftStart	773152	g_strDoFun	3010ad84e645e9

FIGURE 2.3 – Valeurs des premières variables initialisées

Chemin absolu du module

Ensuite, le chemin absolu du binaire va être récupéré à l'aide de l'appel système *readlink* et du fichier *\proc**XX***\exe* où ***XX*** correspond au pid du processus courant.

Check d'intégrité

Un test d'intégrité va être effectué à cette étape. La taille de l'exécutable va être récupérée à l'aide de l'appel système *stat* et va être comparée à la valeur de la variable définie un peu plus tôt : *g_iFileSize*. Si les deux tailles sont différentes, alors le binaire produit une erreur de segmentation.

Chemin absolu du père

Puis, le chemin absolu vers le processus père de l'exécutable est récupéré, de la même manière que précédemment, mais avec la fonction *getppid*.

Protection anti-debug

Un test est de nouveau effectué, mais, cette fois-ci, afin d'"empêcher" le débugage du malware. Si la chaîne de caractères "gdb" est présente dans le chemin absolu du processus père, alors le binaire produit une erreur de segmentation.

Seconde initialisation de variables

La deuxième série d'initialisation de variables n'est pas obfusquée : le nom des variables et leurs contenus sont présentés dans le tableau suivant (figure 2.4).

g_strSN	DBSecuritySpt	g_strML	/tmp/moni.lod
g_strBDSN	selinux	g_strGL	/tmp/gates.lod
g_strBDG	getty		

FIGURE 2.4 – Valeurs des secondes variables initialisées

Définition du comportement futur du binaire

Cette étape va définir le comportement du binaire, en fonction de son chemin absolu. En effet, celui-ci va être comparé au différentes valeurs présentes dans les variables définies plus tôt, afin de définir le contenu d'une nouvelle variable : *g_iGatesType* (voir algorithme 2.2.3).

Algorithm 1 Initialisation de la variable g_iGatesType

```
1:  path := GetModuleFullPath
2:  uaSystools := ['/bin/netstat', '/bin/lsof', '/bin/ps', '/bin/ss',
3:                 '/usr/bin/netstat', '/usr/bin/lsof', '/usr/bin/ps', '/usr/bin/ss',
4:                 '/usr/sbin/netstat', '/usr/sbin/lsof', '/usr/sbin/ps', '/usr/sbin/ss']
5:
6:  if (path = g_strMonitorFile) then
7:      g_iGatesType := 0                                          ▷ Module MainMonitor
8:  else
9:      if (path = "/usr/bin/bsd-port/getty") then                 ▷ Issu de g_strBDG
10:         g_iGatesType := 2                                      ▷ Module MainBackdoor
11:     else
12:         if (path ∈ uaSystools) then
13:             g_iGatesType := 3                                  ▷ Module MainSystool
14:         else
15:             g_iGatesType := 1                                  ▷ Module MainBeikong
16:         end if
17:     end if
18: end if
```

Troisième initialisation de variables

Tout comme lors de l'étape Première initialisation de variables, un ensemble de variables va être initialisé en fonction d'une chaîne de caractères, rentrée en clair dans le binaire.
En ce qui concerne ces variables, il y a deux configurations possibles : une première configuration pour le module Backdoor et une seconde pour le module Beikong. Ces deux configurations sont stockées chiffrées (à l'aide de l'algorithme de chiffrement RSA) dans le binaire et sont déchiffrées avant d'être parsées.

Un test d'intégrité est fait sur les configurations en comparant la valeur des 14 premiers octets du hash MD5 de la chaîne de caractères contenant les deux configurations chiffrées à la valeur stockée dans la variable globale g_strDoFun.

Ces deux configurations sont les suivantes :

g_strConnTgts	115.28.206.48	g_iIsService	1
g_iGatsPort	25000	g_strForceNote	-== Love AV ==-
g_iGatsFx	1	g_iDoBackdoor	1

FIGURE 2.5 – Valeurs des dernières variables globales initialisés pour le module Bill

g_strConnTgts	www.i0cc.com	g_iIsService	1
g_iGatsPort	6001	g_strForceNote	-== Love AV ==-
g_iGatsFx	1	g_iDoBackdoor	1

FIGURE 2.6 – Valeurs des dernières variables globales initialisés pour le module Backdoor

Choix du module

Enfin, le malware va charger un des modules, en fonction de la valeur de la variable *g_iGatesType*, ou lancer une mise à jour si l'exécutable porte le nom **update_temporary**.
Tous les modules à l'exception des modules Monitor et Systool commencent par invoquer la fonction *daemon* (avec les arguments (1, 0)) qui détache le processus actuel du terminal actif en effectuant un *fork* puis en redirigeant stdin, stderr et stdout dans /dev/null.

2.2.4 Module Beikong/Bill

Le module Beikong (ou Bill) est le module d'installation mais également le premier module principal du botnet.
Son rôle est, d'abord, de nettoyer le système en arrêtant et supprimant les potentielles instances de lui-même qui tourneraient déjà sur le système, puis de mettre en place le démarrage automatique de son exécutable au lancement du système (en manipulant la crontab) et, enfin, de réinstaller et démarrer les modules Backdoor et Monitor. Il communique également avec un serveur "command and control" (C&C).

Algorithm 2 Pseudo-code Beikong

```
1: Arrête le module Monitor
2: Arrête le module Bill
3: if IsService then
4:     Créé un script /etc/init.d/DbSecuritySpt et des liens symboliques /etc/rci.d/S97DbSecuritySpt (avec i allant de 1 à 5) vers ce script, permettant de lancer l'exécutable actuel au démarrage
5: end if
6: if DoBackdoor then
7:     Arrête le module Backdoor, le réinstalle (dans /usr/bin/bsd-port/getty) et le relance
8: end if
9: if IsRoot() then
10:    Indique la localisation du fichier du module Beikong dans /tmp/notify.file et installe/lance le module Monitor (dans /usr/bin/.sshd)
11: end if
12: Exécute MainProcess()
```

2.2.5 Module Backdoor

Le module Backdoor est le deuxième module principal du botnet. Il s'occupe de remplacer quelques outils systèmes classiques par son exécutable pour cacher ses traces. Il communique lui aussi avec un serveur C&C.

Algorithm 3 Pseudo-code Backdoor

1: Inscrit le pid du processus actuel dans le fichier **/usr/bin/bsd-port/getty.lock** et lock le fichier
2: Créer un script **/etc/init.d/selinux** et des liens symboliques **/etc/rc*i*.d/S99selinux** (avec i allant de 1 à 5) vers ce script, permettant de lancer l'exécutable actuel (**/usr/bin/bsd-port/getty**) au démarrage
3: Copie et remplace les outils systèmes *netstat*, *lsof*, *ps* et *ss* dans les dossiers **/bin/**, **/usr/bin/** et **/usr/sbin/**
4: Exécute MainProcess()

2.2.6 MainProcess, capacité d'attaque et communication avec les serveurs C&C

Dans les modules précédents on finit dans les deux cas par exécuter la fonction MainProcess(). Cette fonction constitue la partie active du botnet.

Algorithm 4 Pseudo-code MainProcess()

1: Initialise DNSCache (parse **/etc/resolv.conf**, utilisé pour la résolution de noms de domaine)
2: Initialise ConfigDoing (parse **conf.n** qui se trouve dans le dossier courant)
3: Initialise CmdDoing (parse **cmd.n** qui se trouve dans le dossier courant)
4: Initialise StatBase (GetOS, GetCpuSpd, CpuUse, NetUse, GetMemSize)
5: Initialise ProvinceDns (liste de DNS utilisés pour l'amplification)
6: Essaye de charger **/usr/lib/xpacket.ko** avec system("insmod /usr/lib/xpacket.ko")
7: Initialise CAmpResource (en lisant le fichier **/usr/lib/libamplify.so**)
8: Initialise CManager (démarrage des threads principaux et de la communication au C&C)

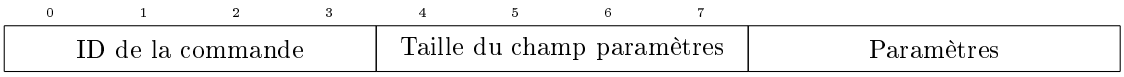
Communication avec le serveur C&C

Le botnet à besoin de communiquer avec un ou plusieurs serveurs C&C pour pouvoir agir.
La communication au C&C peut se faire dans les deux sens : soit le client se connecte au serveur C&C, soit le serveur C&C se connecte au client. Le choix est fait en fonction du paramètre de configuration g_GatsIsFx ; dans notre cas le client se connecte au serveur (ce qui est préférable étant donné que beaucoup de routeurs font du NAT par défaut, ce qui empêche les connexions sur des ports arbitraires depuis l'extérieur).

La communication s'effectue par le biais d'une ou plusieurs connexions TCP/IP en direction ou depuis les IPs définies dans g_strConnTgts et sur le port défini dans g_iGatsPort.
Dans le cas où g_iGatsIsFX est vrai, un thread représenté par la classe "CThreadFXConnection" est crée pour établir et gérer chaque connection aux différents serveurs C&C.
En revanche, dans le cas où g_iGatsIsFX est faux, alors le module exécute la méthode CManager::ZXMainProcess() qui va simplement créer une socket, la bind sur le port g_iGatsPort et se mettre en écoute, de manière à accepter toutes les connexions entrantes sur ce port.
Dans les deux cas, le module fini par appeler la méthode CManager::ConnectionProcess() qui s'occupe de communiquer avec le serveur C&C et de faire passer les commandes reçues au thread en charge de l'exécution des commandes, par le biais d'un objet de type "CThreadSignaledMessageList<CCmdMsg>".

Le protocole de discussion est simple : après avoir établi une connexion avec un serveur C&C, le client commence par envoyer des informations sur lui-même (à savoir, une copie de son objet CConfigDoing actuel) ainsi que sur la machine (telles que le nombre de cœurs présents sur la machine, la fréquence de fonctionnement des cœurs, la quantité de RAM disponible sur la machine, la version du noyau linux ou encore l'IP de la machine dans le réseau local) en appelant la méthode CManager::MakeInitResponse(). Ensuite le client se met en attente de commandes en appelant la méthode CManager::RecvCommand().

Les paquets de commande envoyés par le serveur C&C sont de la forme :



Plusieurs commandes sont disponibles :

Id de la commande	Description
0x1	Démarre une attaque sur une ou plusieurs cibles
0x2	Arrête les attaques ou mises à jour en cours
0x3	Modifie la configuration du module
0x5	Démarre une mise à jour du client
0x7	Mets à jour l'actuel objet de type CCmdDoing
0x8	DoFakeDetect (Contrôler l'IP source)
0x9	Demande l'accès à un reverse shell en tant que root sur le client

FIGURE 2.7 – Liste des commandes disponibles

Le thread représenté par la classe "CThreadTaskGates" s'occupe, en parallèle, d'exécuter la liste de commandes qu'il peut trouver dans la liste des commandes mentionnée précédemment. Il vérifie d'abord que des commandes ont été reçues et, si c'est le cas, exécute le handler associé à la commande, puis recommence indéfiniment.

Attaques normales

Le botnet peut mener plusieurs attaques de type DDOS en utilisant des sockets "raw" depuis le mode utilisateur.
Tout se déroule dans la méthode CThreadAtkCtrl::StartNormalSubTask().
Au total, 11 types d'attaques sont recensées dans ce sample, dont 3 qui ne semblent pas être totalement implémentées :

- CAttackCompress : Attaque TCP flood avec header TCP choisi (utile pour les fragment attack / Tear-drop)
- CAttackSyn : Attaque TCP type SYN flood
- CAttackUdp : Attaque type UDP packet flood
- CAttackDns : Attaque type DNS flood (pour l'attaque de sous-domaines DNS)
- CAttackAmp : Attaque type DNS amplification
- CAttackPrx : Attaque de type indéterminé faisant usage de requêtes DNS
- CAttackIcmp : Attaque type ICMP-Request flood
- CTcpAttack : Attaque type TCP flood (connexion, envoi de 5000 octets, déconnexion)
- CAttackCc : Attaque pas entièrement implémentée (une attaque de type HTTP request flood d'après les seules traces visibles)
- CAttackIe : Attaque non implémentée
- CAttackTns : Attaque non implémentée

Le déroulement des attaques est similaire pour toutes ces attaques. Les classes sont toutes descendantes d'une classe CPacketAttack qui implémente des méthodes virtuelles $UpdateCurVariant$ (chargée de mettre à jours certains paramètres tels que l'IP source, le port source ou le numéro de séquence du prochain paquet), $MakePacket$ (qui s'occupe de forger entièrement le paquet à envoyer) et Do (qui exécute une itération de l'attaque, en appelant $UpdateCurVariant$, $MakePacket$, $SendPacket$ typiquement) que chaque classe d'attaque personnalise en fonction des besoins.

Attaques noyaux

Le botnet est également capable de lancer des attaques depuis le noyau grâce à l'outil *pktgen*, qui permet de générer des paquets très rapidement.
Tout se déroule dans la méthode CThreadAtkCtrl::StartKernalSubTask() (et la faute de frappe pour Kernal n'est pas de nous).

La configuration de l'outil se fait en trois étapes :

- Pour chaque cœur du CPU de la machine, un fichier **/proc/net/pktgen/kpktgend_*i*** (où *i* est le numéro du cœur) est créé. Le contenu de ces fichiers est le suivant :

```
rem_device_all
add_device eth%d
max_before_softirq 10000
```

- Pour chaque cœur du CPU de la machine, un fichier **/proc/net/pktgen/eth*i*** (où *i* est le numéro du cœur) est créé. Le contenu de ces fichiers est le suivant :

```
count 0
clone_skb 0
delay 0
TXSIZE_RND
min_pkt_size %d
max_pkt_size %d
IPSRC_RND
src_min %s
src_max %s
UDPSRC_RND
udp_src_min %d
udp_src_max %d
dst %s
udp_dst_min %d
udp_dst_max %d
dst_mac %02x:%02x:%02x:%02x:%02x:%02x//adresse MAC de la passerelle obtenue de g_statBase
is_multi %d //nombre de cibles
multi_dst %s //si l'attaque se fait vers plusieurs adresses, elles sont specifiees ici
pkt_type %d
dns_domain %s
syn_flag %d
is_dns_random %d
dns_type %d
is_edns %d
edns_len %d
is_edns_sec %d
```

- Enfin, le malware créé un fichier **/proc/net/pktgen/pgctrl**, dans lequel il écrit la chaîne de caractères "start".

La plupart des valeurs utilisées dans la configuration de pktgen sont obtenues depuis les paramètres de la commande d'attaque.

2.2.7 Les autres modules

Monitor

Le module Monitor s'occupe de vérifier que le module Bill reste en vie, et de le relancer si il ne l'est plus.

while True **do**
    Écrit le pid du processus actuel dans le fichier **/tmp/moni.lod** et lock le fichier
    Récupère la localisation du fichier Beikong dans **/tmp/notify.file** et supprime le fichier
    Démarre un thread "CThreadMonGates" qui vérifie, toutes les 60 secondes,
que le fichier **/tmp/gates.lock** à un lock actif (et donc que le module Bill est toujours vivant)
et relance le module Bill si ce n'est pas le cas
end while

Systool

Le module Systool est le module qui s'exécute lorsque l'exécutable se trouve à la place d'un des outils système *netstat*, *lsof*, *ps* ou *ss*. Son rôle est de filtrer les sorties des outils qu'il remplace pour cacher les parties qui pourraient révéler la présence du module Backdoor.

Algorithm 5 Pseudo-code Systool

Déduit le chemin de l'outil système original associé, en le dérivant du nom de l'exécutable actuel
Si l'outil a été trouvé, le chemin complet de l'exécutable du module Backdoor est récupéré ainsi que la valeur du HiddenPort. L'outil système est alors exécuté avec les arguments passés en paramètres et la sortie est filtrée, n'affichant pas les références aux chemins et ports récupérés précédemment.

Update

La partie mise à jour du botnet est composée de deux morceaux : une première partie, handler de la commande *DoUpdateCommand*, qui peut être envoyée par le serveur C&C et une deuxième qui fonctionne comme les modules précédents, dans le sens où, si l'exécutable porte le nom **update_temporary**, alors il exécute la méthode *DoUpdate* et se termine ensuite.

Algorithm 6 Pseudo-code Update

switch update_type **do**
    case 0x1 :
        Download, move to libamplify.so, ReinitReadResources (libamplify.so)
    case 0x5 :
        Copy self, execute (update_temporary, executes DoUpdate)
    case 0x4 :
        Download, move, execute if needed

Algorithm 7 Pseudo-code DoUpate()

if argc == 5 **then**
    Prépare une update pour cfg1
end if
if atoi(argv[1]) == 5 **then**
    Modifie le fichier argv[3] en HardStart et SoftStart
end if

Chapitre 3

Windows – Sage 2.0

3.1 Introduction

Les ransomwares sont des malwares qui ont pour but d'extorquer de l'argent à leurs victimes en chiffrant les données personnelles qui se trouvent sur la machine sur laquelle ils sont exécutés et en demandant une certaine somme d'argent à leur propriétaire en échange de la clé qui permet le déchiffrement.
Le nombre de ransomwares augmente très rapidement depuis leur première apparition, il y a de cela quelques années, et pour cause ; avec un minimum de notions en programmation, il est possible de réaliser un ransomware et le gain potentiel d'argent en intéresse plus d'un. Un autre avantage du ransomware est qu'il peut mener à bien sa mission avec peu de droits sur la machine victime, puisque le seul chiffrement des fichiers de l'utilisateur exécutant le ransomware (qui n'est pas forcément administrateur) peut avoir des conséquences désastreuses pour la victime.

Sage est une nouvelle famille de ransomware, dérivée de CryLocker.
Son fonctionnement est semblable à celui de CryLocker : envoi d'informations à un serveur C&C à l'aide du protocole UDP, géolocalisation à l'aide d'une API Google, suppression des sauvegardes de fichiers *Shadow Copy*, persistance assurée à l'aide d'une tâche planifiée, format de la note de rançon et paiement sur un service TOR. La seule différence étant au niveau des algorithmes de chiffrement utilisés, probablement dans le but de faire perdre un peu de temps aux personnes analysant le malware.

3.2 Analyse

3.2.1 Les outils

Les principaux outils utilisés pour réaliser cette analyse ont été le logiciel de virtualisation *VirtualBox*, le débugueur *x64dbg*, l'utilitaire *ProcessMonitor*, l'éditeur hexadécimal *HxD* et le désassembleur *Radare2*.

3.2.2 Analyse basique

SHA-256 du sample :

```
50624b1338349dcab4ad8345e0100ea75d3b643ef1e3a487b32fd711418b281b
```

Résultat renvoyé par l'utilitaire *file* :

```
sage: PE32 executable (GUI) Intel 80386, for MS Windows
```

C'est donc un exécutable au format PE (Portable Executable) 32-bit pour Windows.

Côté *strings*, rien d'intéressant :

```
[...]
xcludeIe
MANDize
%04X%iArc
;*.baon-fiMAND
iarc.in
\Tran
ot s
clude
ange
efault)
%COMMANas receut w
IDPomes
ting
X%04Error
mmandd probl
ossiblips_rt
You auted coutpeck youges,on f
le typeuration
_ABO regi
std@@
Conf
ma.t
[...]
```

Aucune chaîne de caractères intelligible, ni aucune chaîne concernant une rançon quelconque. On peut supposer que l'exécutable fait donc usage d'obfuscation au moins pour les chaînes de caractères.

Pour avoir une meilleure idée de ce que peut faire le sample, dresser une liste des fonctions qu'il importe (de KERNEL32.dll et NTDLL.dll en l'occurence, bibliothèques incontournables pour les exécutables Windows) se révèle utile.
Radare2 fait cela très bien :

```
[0x0041d1e0]> ii | grep KERNEL32
ordinal=001 plt=0x0041e03c bind=NONE type=FUNC name=KERNEL32.dll_CloseHandle
ordinal=002 plt=0x0041e040 bind=NONE type=FUNC name=KERNEL32.dll_ResumeThread
ordinal=003 plt=0x0041e044 bind=NONE type=FUNC name=KERNEL32.dll_VirtualAlloc
ordinal=004 plt=0x0041e048 bind=NONE type=FUNC name=KERNEL32.dll_ReadProcessMemory
ordinal=005 plt=0x0041e04c bind=NONE type=FUNC name=KERNEL32.dll_SetConsoleMode
ordinal=006 plt=0x0041e050 bind=NONE type=FUNC name=KERNEL32.dll_SetProcessWorkingSetSize
ordinal=007 plt=0x0041e054 bind=NONE type=FUNC name=KERNEL32.dll_ReadFileEx
ordinal=008 plt=0x0041e058 bind=NONE type=FUNC name=KERNEL32.dll_GetModuleHandleA
ordinal=009 plt=0x0041e05c bind=NONE type=FUNC name=KERNEL32.dll_lstrcpynA
ordinal=010 plt=0x0041e060 bind=NONE type=FUNC name=KERNEL32.dll_GetProcAddress
ordinal=011 plt=0x0041e064 bind=NONE type=FUNC name=KERNEL32.dll_GetStartupInfoA
ordinal=012 plt=0x0041e068 bind=NONE type=FUNC name=KERNEL32.dll_SetEnvironmentVariableA
ordinal=013 plt=0x0041e06c bind=NONE type=FUNC name=KERNEL32.dll_Sleep
ordinal=014 plt=0x0041e070 bind=NONE type=FUNC name=KERNEL32.dll_SetStdHandle
ordinal=015 plt=0x0041e074 bind=NONE type=FUNC name=KERNEL32.dll_SetEvent
ordinal=016 plt=0x0041e078 bind=NONE type=FUNC name=KERNEL32.dll_ProcessIdToSessionId
ordinal=017 plt=0x0041e058 bind=NONE type=FUNC name=KERNEL32.dll_GetModuleHandleA
ordinal=018 plt=0x0041e080 bind=NONE type=FUNC name=KERNEL32.dll_LoadLibraryA

0x0041d1e0]> ii | grep NTDLL
ordinal=001 plt=0x0041e100 bind=NONE type=FUNC name=NTDLL.dll_isprint
ordinal=002 plt=0x0041e104 bind=NONE type=FUNC name=NTDLL.dll_iswdigit
ordinal=003 plt=0x0041e108 bind=NONE type=FUNC name=NTDLL.dll_RtlUnwind
ordinal=004 plt=0x0041e10c bind=NONE type=FUNC name=NTDLL.dll___toascii
ordinal=005 plt=0x0041e110 bind=NONE type=FUNC name=NTDLL.dll_strtol
ordinal=006 plt=0x0041e114 bind=NONE type=FUNC name=NTDLL.dll__CIsqrt
ordinal=007 plt=0x0041e118 bind=NONE type=FUNC name=NTDLL.dll_RtlMoveMemory
```

Première remarque, la liste est courte. Deuxième remarque, avec les fonctions ici listées, difficile d'imaginer comment un ransomware pourrait mener à bien sa mission (c'est-à-dire chiffrer des fichiers sans fonctions liées à la gestion de fichiers). Là encore, probable obfuscation. *VirtuallAlloc* et *GetProcAddress* sont importées, ce qui permet tout à fait de charger du code (stocké compressé ou chiffré) lors de l'exécution.

3.2.3 Obfuscation

La première partie du code de Sage utilise une simple technique pour rendre la lecture du code un peu plus pénible. Toutes les 4 à 10 instructions, un appel à une fonction est fait, s'occupant de modifier la valeur du registre EIP, à la manière d'une instruction *jmp*. La "longueur" du saut est définie à l'aide d'un paramètre passé à la fonction.

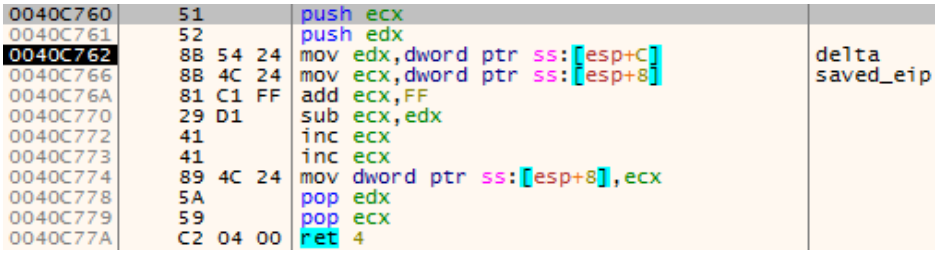


FIGURE 3.1 – Fonction utilisée pour exécuter une instruction *jmp* de manière détournée

Il y a 11 fonctions comme celle-ci, contenant le même code et qui sont utilisées à tour de rôle.
Dans la figure ci-dessous, la fonction *sage.404924* est une de ces fonctions.

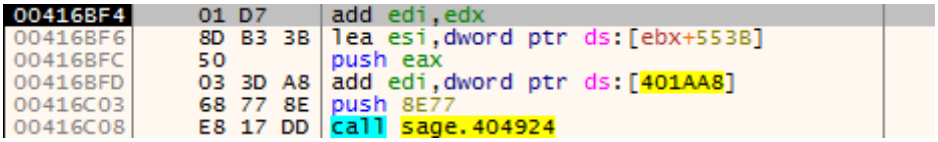


FIGURE 3.2 – Utilisation de la fonction pour rendre le flot d'exécution plus difficile à suivre

L'essentiel du code du ransomware est stocké chiffré en mémoire. Le chargement du code se fait en 2 parties. D'abord, un *stub* (qui est stocké à l'adresse 0x004210BA, chiffré par des combinaisons de *xor* et de *add*) [1] est déchiffré, copié dans un espace mémoire alloué dynamiquement puis exécuté.
Ce stub se charge ensuite de re-allouer les pages à partir de l'adresse 0x00400000 (adresse à laquelle est chargée l'image de l'exécutable en mémoire), supprimant toutes traces des headers et de la section *.text* originale. Il déchiffre ensuite le véritable code du ransomware (qui est stocké à l'adresse 0x0042166A, chiffré à l'aide de RC4), puis passe l'exécution au code fraîchement déchiffré. Après avoir fait un dump de la partie packée, on retrouve la véritable liste des fonctions importées ainsi que le point d'entrée du programme original, qui est à l'adresse 0x00406020.

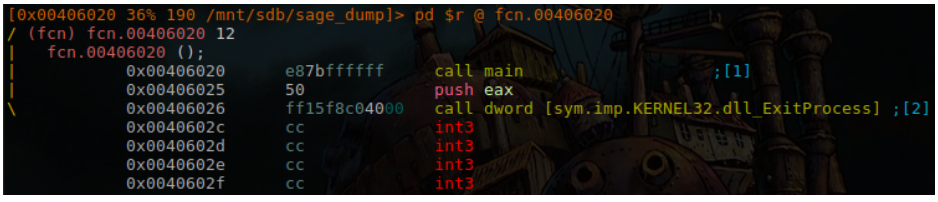


FIGURE 3.3 – Point d'entrée du code original, après déchiffrement

Sage n'intègre pas de véritables fonctionnalités d'anti-debug mais tente simplement de gêner les utilisateurs de débugueurs en simulant un fork au début de son exécution. Pour se faire plus discret, Sage tente également de dissimuler sa présence en faisant une copie de lui-même dans le dossier **%appdata%\Roaming**. Cette copie est nommée aléatoirement (nom composé de 8 caractères alphanumériques) et c'est elle qui s'exécute à chaque démarrage de session utilisateur pour assurer la persistance.

1. Dans le domaine du packing, un stub est un morceau de code minimal dont le rôle est de charger une partie plus importante du code qui est chiffrée ou compressée

3.2.4 Main

Grâce au dump du code original du ransomware, une analyse statique plus poussée est possible. Bonne nouvelle, la fonction principale est plutôt courte et sa structure est simple :

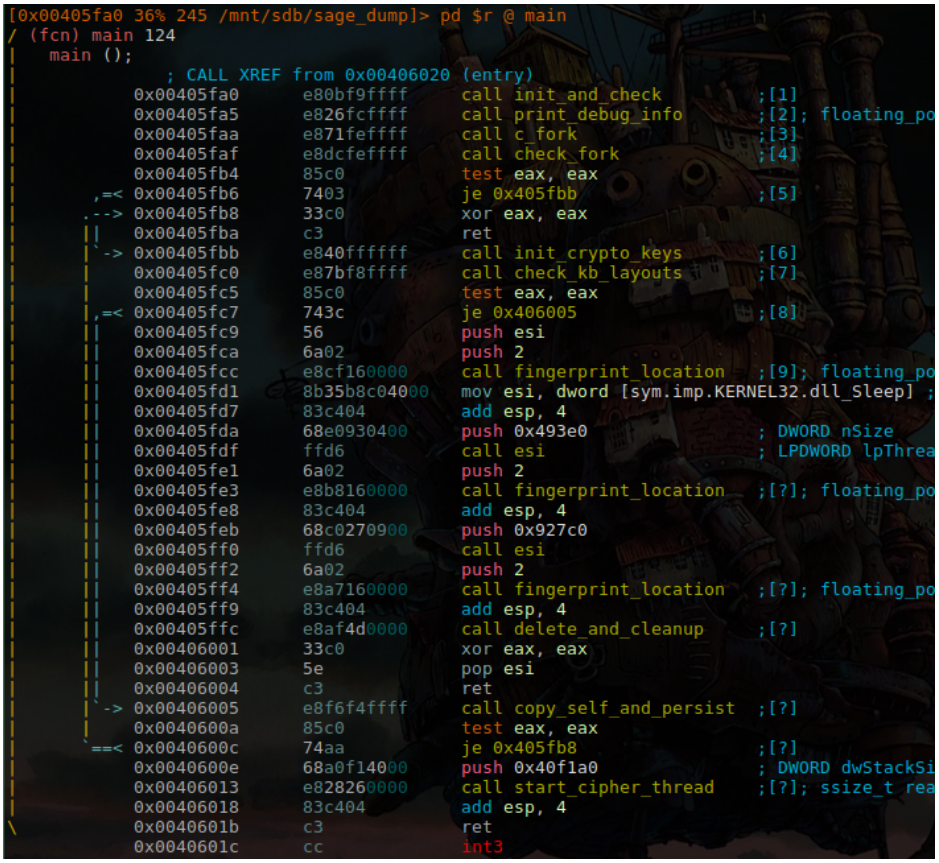


FIGURE 3.4 – Code assembleur de la fonction principale après déchiffrement

Ce qui donne, en pseudo-code :

```
int main()
{
  init_and_chek();
  print_debug_info();
  c_fork(arg);
  if (check_fork())
    return 0;
  init_crypto_keys();
  if (check_kb_layouts())
  {
    fingerprint_location(2);
    Sleep(0x493E0u);
    fingerprint_location(2);
    Sleep(0x927C0u);
    fingerprint_location(2);
    delete_and_cleanup();
    result = 0;
  }
  else
  {
    if (!copy_self_and_persist())
      return 0;
    result = start_cipher_thread(&encryption_key);
  }
  return result;
}
```

Vérification de la langue Sage estime la nationalité de la victime, d'après la liste des layouts clavier utilisés sur la machine, qu'il obtient grâce à la fonction *GetKeyboardLayoutList*.

```
bool check_kb_layouts()
{
  localeCount = GetKeyboardLayoutList(10, (HKL *)&List);
  if (localeCount <= 0)
    return false;
  i = 0;
  if (localeCount <= 0)
    return false;
  while (true)
  {
    next = (int)(&List)[i] & 0x3FF;
    if (next == 0x23 || next == 0x3F || next == 0x19 || next == 0x22 ||
        next == 0x43 || next == 0x85)
      break;
    if (++i >= localeCount)
      return false;
  }
  return true;
}
```

Ces codes correspondent aux layouts clavier suivants :

- Biélorusse
- Kazakh
- Russe
- Ukrainien
- Ouzbek
- Sakha

Si un de ces layouts clavier est utilisé sur la machine victime, alors Sage s'arrête avant de chiffrer quoi que ce soit.

Estimation de la localisation Si c'est possible (c'est-à-dire, si des bornes Wi-Fi sont détectées), Sage tente également de localiser plus précisément la machine sur laquelle il s'exécute en utilisant l' API de géolocalisation Google.
Cette API permet d'estimer la localisation d'une borne Wi-Fi à l'aide de son adresse MAC et de son SSID.

Persistance En ce qui concerne la persistance (l'exécution de lui-même au démarrage), Sage fait usage des tâches planifiées de Windows. Après s'être copié dans le dossier **AppData\Roaming**, Sage ajoute une tâche dans la liste de manière à relancer l'exécution de son binaire à chaque nouvelle connexion d'un utilisateur sur la machine. Ceci permet de s'assurer que la machine reste inutilisable après l'infection et ce jusqu'à ce que la rançon soit payée.

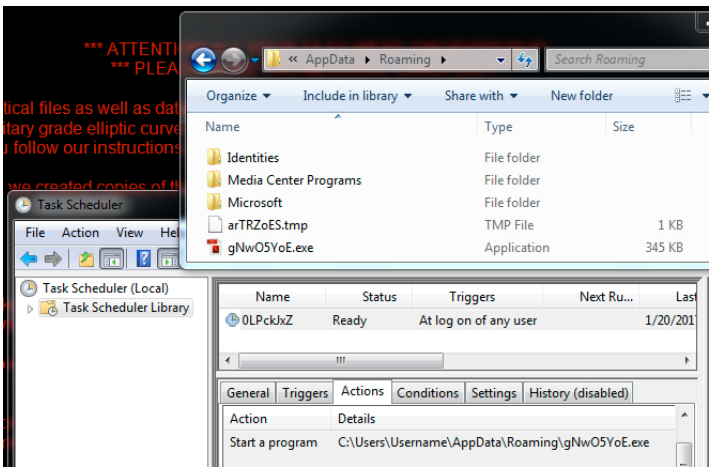


FIGURE 3.5 – Binaire de sage et tâche planifiée utilisée pour maintenir l'exécution au démarrage.

Information de debug & fichier canary Certains restes de fonctionnalités de debug sont trouvables dans l'exécutable. Notamment, la gestion d'un paramètre "d", qui permet d'afficher une information concernant la configuration du ransomware.

```
C:\Users\Marjorie\Desktop>sage.exe d

C:\Users\Marjorie\Desktop>{"b":"22A0F11342191EFD"}
```

FIGURE 3.6 – Résultat renvoyé par Sage en lui passant l'argument "d"

Également, vérification de la présence d'un fichier qui permet d'éviter le lancement du ransomware, certainement utile pour éviter tout lancement accidentel durant le développement du ransomware :

```
if (CreateFileW(L"C:\\Temp\\lol.txt", 0x80000000, FILE_SHARE_READ, NULL,
    OPEN_EXISTING, 0, NULL) == INVALID_HANDLE_VALUE)
{
        // Chiffrement des fichiers
}
```

Communication C&C Sage tente de communiquer à un serveur C&C durant son exécution. Il essaye d'abord d'obtenir une adresse IP pour le nom de domaine **mbfce24rgn65bx3g.rzunt3u2.com**, et si il n'y parvient pas, envoie un paquet chiffré à des milliers d'adresses IP différentes en utilisant le protocole UDP. Le choix du protocole UDP est probablement dû au fait que le protocole UDP permet d'envoyer un paquet à un serveur sans établir de connexion ou même attendre de réponse du serveur avec lequel on communique. Cela permet à Sage de garder la véritable IP du serveur C&C cachée parmi les milliers d'IP auxquelles sont envoyés ces paquets.

Voici une partie du trafic réseau produit par Sage :

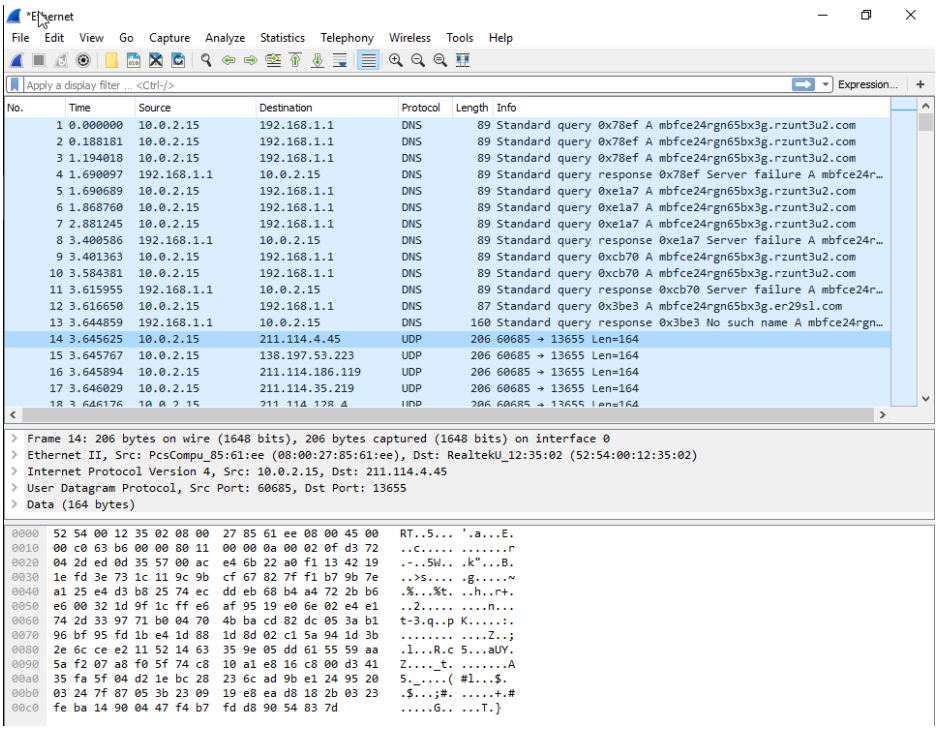


FIGURE 3.7 – Paquets envoyés par Sage lors de son exécution.

3.2.5 Extensions ciblées

Pour éviter de rendre le système d'exploitation inutilisable en chiffrant des fichiers critiques, les ransomwares se contentent de chiffrer uniquement les fichiers dont les noms contiennent les extensions les plus couramment utilisées pour les fichiers de données utilisateur.

En ce qui concerne Sage, cette liste est la suivante :

```
.dat .mx0 .cd .pdb .xqx .old .cnt .rtp .qss .qst .fx0 .fx1 .ipg .ert .pic .img
.cur .fxr .slk .m4u .mpe .mov .wmv .mpg .vob .mpeg .3g2 .m4v .avi .mp4 .flv
.mkv .3gp .asf .m3u .m3u8 .wav .mp3 .m4a .m .rm .flac .mp2 .mpa .aac .wma .djv
.pdf .djvu .jpeg .jpg .bmp .png .jp2 .lz .rz .zipx .gz .bz2 .s7z .tar .7z .tgz
.rar .zip .arc .paq .bak .set .back .std .vmx .vmdk .vdi .qcow .ini .accd .db
.sqli .sdf .mdf .myd .frm .odb .myi .dbf .indb .mdb .ibd .sql .cgn .dcr .fpx
.pcx .rif .tga .wpg .wi .wmf .tif .xcf .tiff .xpm .nef .orf .ra .bay .pcd .dng
.ptx .r3d .raf .rw2 .rwl .kdc .yuv .sr2 .srf .dip .x3f .mef .raw .log .odg .uop
.potx .potm .pptx .rss .pptm .aaf .xla .sxd .pot .eps .as3 .pns .wpd .wps .msg
.pps .xlam .xll .ost .sti .sxi .otp .odp .wks .vcf .xltx .xltm .xlsx .xlsm
.xlsb .cntk .xlw .xlt .xlm .xlc .dif .sxc .vsd .ots .prn .ods .hwp .dotm .dotx
.docm .docx .dot .cal .shw .sldm .txt .csv .mac .met .wk3 .wk4 .uot .rtf .sldx
.xls .ppt .stw .sxw .dtd .eml .ott .odt .doc .odm .ppsm .xlr .odc .xlk .ppsx
.obi .ppam .text .docb .wb2 .mda .wk1 .sxm .otg .oab .cmd .bat .h .asx .lua .pl
.as .hpp .clas .js .fla .py .rb .jsp .cs .c .jar .java .asp .vb .vbs .asm .pas
.cpp .xml .php .plb .asc .lay6 .pp4 .pp5 .ppf .pat .sct .ms11 .lay .iff .ldf
.tbk .swf .brd .css .dxf .dds .efx .sch .dch .ses .mml .fon .gif .psd .html
.ico .ipe .dwg .jng .cdr .aep .aepx .123 .prel .prpr .aet .fim .pfb .ppj .indd
.mhtm .cmx .cpt .csl .indl .dsf .ds4 .drw .indt .pdd .per .lcd .pct .prf .pst
.inx .plt .idml .pmd .psp .ttf .3dm .ai .3ds .ps .cpx .str .cgm .clk .cdx .xhtm
.cdt .fmv .aes .gem .max .svg .mid .iif .nd .2017 .tt20 .qsm .2015 .2014 .2013
.aif .qbw .qbb .qbm .ptb .qbi .qbr .2012 .des .v30 .qbo .stc .lgb .qwc .qbp
.qba .tlg .qbx .qby .1pa .ach .qpd .gdb .tax .qif .t14 .qdf .ofx .qfx .t13 .ebc
.ebq .2016 .tax2 .mye .myox .ets .tt14 .epb .500 .txf .t15 .t11 .gpc .qtx .itf
.tt13 .t10 .qsd .iban .ofc .bc9 .mny .13t .qxf .amj .m14 ._vc .tbp .qbk .aci
.npc .qbmb .sba .cfp .nv2 .tfx .n43 .let .tt12 .210 .dac .slp .qb20 .saj .zdb
.tt15 .ssg .t09 .epa .qch .pd6 .rdy .sic .ta1 .lmr .pr5 .op .sdy .brw .vnd .esv
.kd3 .vmb .qph .t08 .qel .m12 .pvc .q43 .etq .u12 .hsr .ati .t00 .mmw .bd2 .ac2
.qpb .tt11 .zix .ec8 .nv .lid .qmtf .hif .lld .quic .mbsb .nl2 .qml .wac .cf8
.vbpf .m10 .qix .t04 .qpg .quo .ptdb .gto .pr0 .vdf .q01 .fcr .gnc .ldc .t05
.t06 .tom .tt10 .qb1 .t01 .rpf .t02 .tax1 .1pe .skg .pls .t03 .xaa .dgc .mnp
.qdt .mn8 .ptk .t07 .chg .#vc .qfi .acc .m11 .kb7 .q09 .esk .09i .cpw .sbf .mql
.dxi .kmo .md .u11 .oet .ta8 .efs .h12 .mne .ebd .fef .qpi .mn5 .exp .m16 .09t
.00c .qmt .cfdi .u10 .s12 .qme .int? .cf9 .ta5 .u08 .mmb .qnx .q07 .tb2 .say
.ab4 .pma .defx .tkr .q06 .tpl .ta2 .qob .m15 .fca .eqb .q00 .mn4 .lhr .t99
.mn9 .qem .scd .mwi .mrq .q98 .i2b .mn6 .q08 .kmy .bk2 .stm .mn1 .bc8 .pfd .bgt
.hts .tax0 .cb .resx .mn7 .08i .mn3 .ch .meta .07i .rcs .dtl .ta9 .mem .seam
.btif .11t .efsl .$ac .emp .imp .fxw .sbc .bpw .mlb .10t .fa1 .saf .trm .fa2
.pr2 .xeq .sbd .fcpa .ta6 .tdr .acm .lin .dsb .vyp .emd .pr1 .mn2 .bpf .mws
.h11 .pr3 .gsb .mlc .nni .cus .ldr .ta4 .inv .omf .reb .qdfx .pg .coa .rec .rda
.ffd .ml2 .ddd .ess .qbmd .afm .d07 .vyr .acr .dtau .ml9 .bd3 .pcif .cat .h10
.ent .fyc .p08 .jsd .zka .hbk .mone .pr4 .qw5 .cdf .gfi .cht .por .qbz .ens
.3pe .pxa .intu .trn .3me .07g .jsda .2011 .fcpr .qwmo .t12 .pfx .p7b .der .nap
.p12 .p7c .crt .csr .pem .gpg .key
```

3.2.6 Chiffrement

Sage se démarque des autres ransomwares en ce qui concerne le chiffrement puisqu'il fait usage de cryptographie sur les courbes elliptiques ainsi que de l'algorithme ChaCha pour chiffrer les fichiers de l'utilisateur, ce qui n'est pas commun.
ChaCha est un algorithme de chiffrement à flot dérivé de Salsa20 ; Sage l'utilise pour chiffrer le contenu des fichiers.
Chaque fichier cible est renommé et chiffré avec une clé ChaCha choisie aléatoirement, et cette clé est ensuite stockée à la fin du fichier chiffré. Cette clé n'est évidemment pas stockée tel quel dans le fichier, ce sont en fait deux parties qui permettent, si on possède une valeur secrète, de retrouver la clé utilisée pour chiffrer le fichier, qui sont stockées.
Il est important de noter que Sage prend soin de supprimer les sauvegardes de fichiers faites avec *Shadow Copy* avant de commencer le chiffrement.
Les calculs sont faits sur la courbe elliptique *Curve25519* (d'équation $y^2 = x^3 + 486662 * x^2 + x$, sur $\mathbb{F}_p = GF(2^{255} - 19)$), qui est une courbe populaire pour plusieurs raisons (sécurité, rapidité des calculs, propriétés liées à la génération d'éléments aléatoires) et qui est conçue pour être utilisée par le protocole ECDH (Elliptic Curve Diffie-Hellman).
Dans la suite, on notera $\mathbb{F}_p = GF(2^{255} - 19)$ et G le point de la courbe $Curve25519$ admettant $x = 9$. Enfin, le ransomware a connaissance de $Q_s = k_s * G$, la partie publique de l'extorqueur utilisée lors de la création du secret partagé.

Le chiffrement d'un fichier se déroule comme ceci :

- On génère $k_c \in \mathbb{F}_p$ aléatoirement
- On calcule le point $Q_c = k_c * G$
- On calcule le secret partagé $S = k_c * Q_s = (k_c * k_s) * G$
- On dérive du secret partagé, un entier $sh \in \mathbb{F}_p$
- On calcule le point $P = sh * G$
- On génère $n \in \mathbb{F}_p$ aléatoirement
- On calcule les points $chacha\_key = n * P = (n * sh) * G$ et $chacha\_pub = n * G$
- On chiffre le fichier en utilisant ChaCha avec une clé dérivée de $chacha\_key$
- On ajoute les valeurs de Q_c et $chacha\_pub$ à la fin du fichier

Ce qui produit, en pratique, un fichier de la forme :

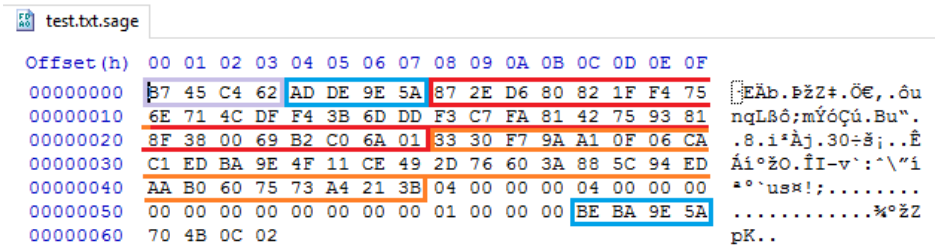


FIGURE 3.8 – Contenu d'un fichier après avoir été chiffré par Sage

Légende :

- Mauve : Contenu chiffré du fichier original
- Rouge : Q_c
- Orange : $chacha\_pub$
- Bleu : Valeurs constantes

Petite note sur le déchiffrement : il est nécessaire de connaître la valeur de k_s pour pouvoir déchiffrer les fichiers. Cette valeur n'est stockée nulle part dans le binaire et on peut donc supposer que seul l'auteur du ransomware la possède, évidemment. Étant donné qu'il n'est donc pas possible, en pratique, de calculer k_s en un temps raisonnable, le seul moyen de déchiffrer les fichiers est donc, soit de payer, soit d'attendre que la clé privée fuite ou soit retrouvée sur des serveurs saisis par la police (comme ce fut le cas pour le ransomware ICEPOL).

Le déchiffrement d'un fichier se déroule comme ceci :

- On récupère les valeurs Q_c et $chacha\_pub$
- On calcule le secret partagé $S = k_s * Q_c = (k_s * k_c) * G$
- On dérive du secret partagé, l'entier $sh \in \mathbb{F}_p$
- On calcule le point $chacha\_key = sh * chacha\_pub = (sh * n) * G$
- On déchiffre le fichier en utilisant ChaCha avec la clé dérivée de $chacha\_key$

Conclusion

Les études du botnet Ganiw et du ransomware SageCrypt ont permis d'effleurer la diversité des malwares présents aujourd'hui. Ces deux malwares sont très différents dans leurs structures et dans les protections mises en place.
Ganiw est codé de manière très modulaire et ses protections contre le reverse-engineering sont assez faible. C'est un malware facilement analysable mais qui pourra être retrouvé sous de nombreuses formes, avec des modules d'attaques supplémentaires. À l'inverse, SageCrypt est bien plus protégé, avec la présence d'un packer par exemple, mais son code et sa structure sont plus ou moins connus, car dérivés d'un autre ransomware, CryLocker.

À la suite de cette étude, il est facile de remarquer que du temps peut être économisé en automatisant certaines parties de l'étude, qui plus est si l'élément a inspecté ou que ses mécanismes d'infection et de propagation sont déjà, en partie connus, et si de nombreux éléments sont à étudier. Parmi les outils existants sont disponibles les sandboxs *Limon*[2] et *Cuckoo*[3], très utiles pour fournir rapidement de nombreuses informations sur l'élément analysé et de manière sécurisé, sans compromettre son environnement de travail.

Mais ceci ne permet pas de classifier les malwares et nécessite encore une grande partie de travail manuel. Pour cela, il faudrait pouvoir les discriminer en fonction de leur comportement, de leurs appels à diverses librairies, ...Ceci pourrait être fait en se basant sur le graphe du flot d'exécution (CFG : Control Flow Graph), par exemple.
Quels que soient les moyens mis en place, l'intervention humaine sera toujours nécessaire pour la détection d'infection par un malware car le problème est connu pour être indécidable (se réduisant au problème de l'arrêt)[4].

2. `https://github.com/monnappa22/Limon`
3. `https://github.com/cuckoosandbox/cuckoo`
4. *Fundamentals of Computation Theory : 18th International Symposium, FCT 2011, Oslo – Norway*, Olaf Owe, Martin Steffen, Jan Arne Telle

Chapitre 4

Annexes

Algorithm 8 Main Pseudocode

```
1: Fermer les descripteurs de fichiers
2: Première initialisation de variables          ▷ Initialise g_strMonitorFile, g_iFileSize, g_iHardStart, ...
3: Chemin absolu du module
4: if (GetTgtFileSize!= g_iFileSize) then   ▷ Test d'intégrité : vérifie que le binaire n'a pas changé de taille
5:     SegmentationFault
6: end if
7: Chemin absolu du père
8: if ("gdb" in chemin du père) then          ▷ Anti-Debug : vérifie que le processus père n'est pas gdb
9:     Segmentation Fault
10: end if
11: Seconde initialisation de variables                ▷ Initialise g_strSN, g_strBDSN, g_strBDG, ...
12: Définition du comportement du binaire              ▷ Initialise g_iGatesType en fonction du module activé
13: Troisième initialisation de variables     ▷ Initialise g_strConnTgts, g_iGatsPort, g_iDoBackdoor, ...
14:
15: if (IsUpdateTemporary) then                                                          ▷ Choix du module
16:     DoUpdate
17: else if (g_iGatesTypes = 1) then
18:     MainBeikong
19: else if (g_iGatesType > 1) then
20:     if (g_iGatesType = 2) then
21:         MainBackdoor
22:     else if (g_iGatesType = 3) then
23:         MainSystool
24:     end if
25: else if (! g_iGatesType) then
26:     MainMonitor
27: end if
```

Algorithm 9 Décoder et initialiser les premiers flags

```
#include <string>
#include <cstring>

using namespace std;

int
ascii_to_dec (char * c)
{
    char v2 = 0;
    if (*c <= 47 || *c > 57)
    {
        if ( *c > 64 && *c <= 70)
            v2 = *c - 55;
    }
    else
    {
        v2 = *c - 48;
    }
    return v2;
}

int
treatment (char * c1, int c2, char * c3, int c4)
{
    char tmp1;
    char tmp2;
    int result = 0;

    for (signed int i = 0; c2 / 2 > i; i++)
    {
        tmp1 = 16 * ascii_to_dec (c1 + 2 * i);
        tmp2 = tmp1 + ascii_to_dec (c1 + 2 * i + 1);

        if (i < c4)
        {
            *(c3 + i) = tmp2;
            ++result;
        }
    }
    return result;
}
int
main ()
{
  string hash_s = "681A1C1543072E0140491F162F0B55545C55775F55565E57745E5D545652705D5E5\
5585F70585C5659577D5C5F565C0423575B025A51720A56";
    const char * hash_c = hash_s.c_str ();
    string param_s = "Google";
    const char * param_c = param_s.c_str ();
    char * tokens;
    char val[512], v4[512];
    double d = 0;
    std::memcpy (val, &d, 512);
    std::memcpy (v4, &d, 512);
    int res = treatment ((char *) hash_c, strlen (hash_c), val, 511);

    for (int i = 0; i < res; i++)
    {
        v4[i] = val[i] ^ param_c[i % (signed int) strlen (param_c)];
    }
    printf ("Result: %s\n\nFlag values:\n", v4);

    tokens = strtok (v4, ":");
    while (tokens)
    {
        printf ("%s\n", tokens);
        tokens = strtok (NULL, ":");
    }
    return EXIT_SUCCESS;
}
```