ExaTrack

From tweet to rootkit

ExaTrack - Stéfan Le Berre (stefan.le-berre [at] exatrack.com)

Ce papier parlera de nos analyses depuis un poste sur twitter de *Florian Roth* qui nous a mené à identifier (et analyser) un **rootkit signé**, dont le **certificat n'est pas encore révoqué** et qui est **inconnu de VirusTotal**. Dans cette version publique nous vous présenterons une partie de nos analyses sur l'un des deux dumps. Bonne lecture :-)

Introduction

Le 24 juillet 2019 un poste sur twitter de *Florian Roth* a particulièrement attiré notre attention, il évoquait un rootkit de Winnti qui venait d'être envoyé sur VirusTotal.

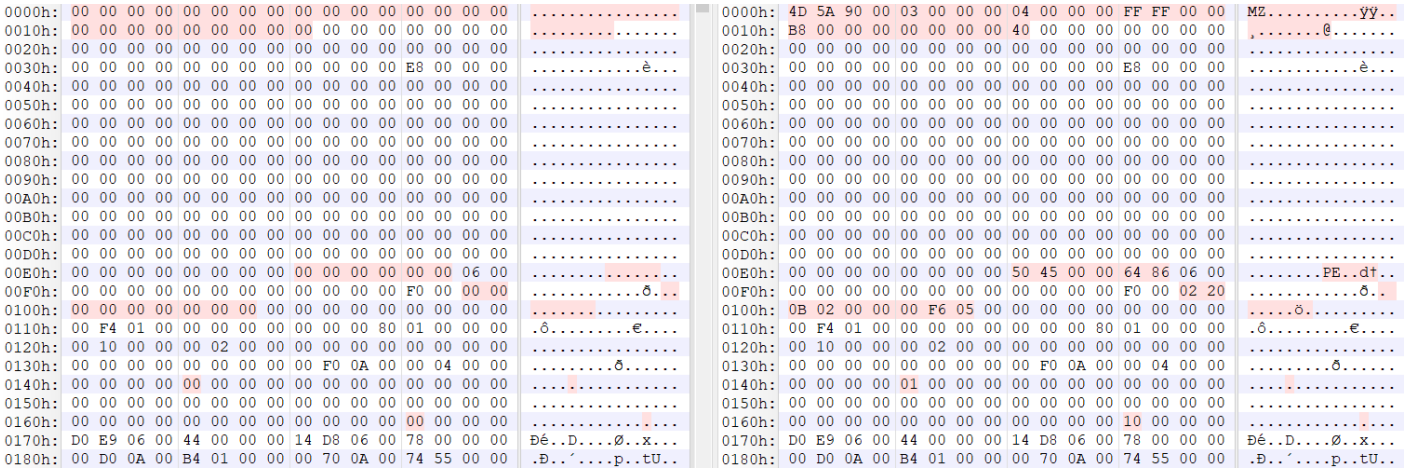
A ExaTrack nous sommes très portés sur la détection et l'analyse de rootkits, et ne voyant aucune analyse de ce dump plus d'un mois après le poste nous avons décidé d'y jeter un coup d'œil.

Le binaire que nous analyserons est le suivant :
https://www.virustotal.com/gui/file/92c37c829dac8f6d277ae4b72b926e82f54ed8fc1b61885d7d7d92fd8417b99f/analysis

L'analyse a pour objectif d'identifier les fonctionnalités majeures du rootkit et d'analyser une partie des actions du code en mode utilisateur.

Reconstruction du sample

Le fichier semble être un dump d'exécutable partiellement corrompu, des identifiants d'entête ont été supprimés. Nous avons donc reconstruit l'entête MZ et PE pour que le binaire soit chargeable par le système d'exploitation et analysable avec des outils.



A notre surprise il ne s'agit pas d'un driver, comme le laissait penser le poste twitter, mais d'un fichier DLL 64b. Nous verrons plus loin dans ce papier qu'un driver signé est effectivement présent.

Analyse du fichier DLL

Informations de contexte

Avant de commencer l'analyse technique du code malveillant plusieurs informations peuvent être récupérées avec les premiers éléments disponibles.

Tout d'abord nous constatons que le nom original de la DLL semble être `workdll64.dll`, comme déclaré dans l'`Export Address Table`. Ce nom doit certainement être une appellation interne.

En recherchant certaines chaînes de caractères marquantes sur Internet nous avons identifié un lien avec un fichier disponible sur *Hybrid Analysis* : http://ww.hybrid-analysis.com/sample/a5d6139921576c3aedfc64e2b37ae1a64f3160bd1bb70d4fc7fce956029e7d55

Le nom de soumission du binaire est `rasppp_decrypt.dat_fixed by r0cu3`, nous pouvons donc suspecter que le nom original était `rasppp.dll` classé par la plateforme comme *ambiguous*. Le fichier PDB associé est `I:\DrvDev\Works\NdisReroute\X64\NdisRerouteD.pdb`, il nous indique donc un probable lien avec les drivers NDIS, lien qui se confirmera par la suite. Le fichier a été envoyé pour la première fois sur VirusTotal le 2015-08-13, nous sommes donc en présence d'une suite de codes malveillants actifs depuis au moins 4 ans.

Entrypoint avec des arguments spécifiques

La première caractéristique assez rare du malware est son initialisation. Lors du chargement d'une DLL la fonction `DllMain` est exécutée, celle-ci est définie par *Microsoft* comme suit :

```
BOOL WINAPI DllMain(
  _In_ HINSTANCE hinstDLL,
  _In_ DWORD     fdwReason,
  _In_ LPVOID    lpvReserved
);
```

Nous avons donc l'argument `lpvReserved` qui n'est pas clairement défini. Normalement cette valeur est sensée être à 0, mais dans des cas particuliers, comme évoqués par j00ru (https://j00ru.vexillium.org/2009/07/dllmain-and-its-uncovered-possibilites/) il peut pointer sur une structure CONTEXT.

Dans notre cas nous avons DllMain qui commence en vérifiant cet argument :

```
undefined8 DllMain(undefined8 hinstDLL,int fdwReason,CONTEXT *lpvReserved)

{
  int iVar1;
  registers_dump local_58;

  if (((fdwReason == 1) && (_DAT_1800845e0 != 1)) &&
     (_DAT_1800845e0 = fdwReason, lpvReserved != (CONTEXT *)0x0)) {
```

Si le module est chargé par une plateforme d'analyse nous aurons donc cet argument à 0 et le malware n'effectuera aucune action particulière. Comme évoqué dans l'article : « If fdwReason is DLL_PROCESS_ATTACH, lpvReserved is NULL for dynamic loads and non-NULL for static loads. »

Validation du processus

Dans la suite du code plusieurs validations sont effectuées pour valider le contexte d'exécution du code par rapport à l'initialisation du système d'exploitation lui-même. Tout d'abord que le processus hôte se nomme bien `svchost.exe`.

```
  GetModuleFileNameA((HMODULE)0x0,&local_128,0x104);
  _strlwr(&local_128);
  strstr(&local_128,"svchost.exe");
```

Entre la validation des arguments (faite dans une autre partie du code) et le nom de l'exécutable, le module a relativement peu de chances d'être détecté par son exécution dans une *sandbox*.

Interrogation des devices réseaux

Le malware cherche à identifier l'`AdapterName` de l'interface réseau ethernet en se basant sur les fonctions `GetAdaptersInfo` et `GetIfTable`. Une fois cette sélection faite, la DLL va la vérifier via la clé de registre `HKLM\SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318}` et identifier la sous-clé `Linkage` ayant la valeur `RootDevice` associée.

Ceci a pour objectif de valider la configuration réseau associée à l'interface réseau ethernet de la machine.

```
if (AdaptaterName != (char *)0x0) {
                    /* (Network adapters) {4D36E972-E325-11CE-BFC1-08002BE10318} */
  result = RegOpenKeyExA((HKEY)0xffffffff80000002,

                         "SYSTEM\\CurrentControlSet\\Control\\Class\\{4D36E972-E325-11CE-BFC1-0800
                         2BE10318}"
                         ,0,0x20019,(PHKEY)&lHkey);
  if (result == 0) {
    lSubKeys = 0;
    LVar1 = RegQueryInfoKeyA(lHkey,(LPSTR)0x0,(LPDWORD)0x0,(LPDWORD)0x0,&lSubKeys,(LPDWORD)0x0,
                             (LPDWORD)0x0,(LPDWORD)0x0,(LPDWORD)0x0,(LPDWORD)0x0,
                             (PFILETIME)0x0);
    if (LVar1 == 0) {
      null_ptr = 0;
      memset(local_137,0,0x103);
      if (lSubKeys != 0) {
        do {
          lValue = 0x104;
          LVar1 = RegEnumKeyExA(lHkey,dwIndex,(LPSTR)&null_ptr,&lValue,(LPDWORD)0x0,(LPSTR)0x0,
                                (LPDWORD)0x0,(PFILETIME)0x0);
```

Extraction du driver signé

Au cours de son initialisation le module peut charger un driver en fonction de la version du système d'exploitation.

```
win_version = set_global_win_version();
get_temp_filename(&filename);
if (win_version < 4) {
  driver_size = 0x9400;
  driver_datas = &DrvDatas;
}
else {
  driver_size = 0x9c50;
  driver_datas = &Drv2Datas;
}
uVar2 = write_to_disk(driver_datas,driver_size,&filename);
```

La valeur 4 désigne un noyau de version 6.0, soit *Windows Vista*. Nous nous sommes intéressés au driver chargé dans le cas d'un système de version 6.0 et supérieur. Pour charger le driver les différentes clés registres sont créées par le malware, puis le chargement est effectué par un `NtLoadDriver` (résolu dynamiquement).

Driver

Signature

Le driver est signé, nous sommes donc probablement en présence d'un certificat volé, utilisé pour charger le rootkit sur les systèmes Windows 64b.

```
       Verified:         A required certificate is not within its validity period when
verifying against the current system clock or the timestamp in the signed file.
       Link date:      06:10 11/04/2016
        Signing date:   n/a
        Catalog:        C:\rootkit.sys
        Signers:
           *****
                Cert Status:      This certificate or one of the certificates in the
certificate chain is not time valid.
                Valid Usage:    Code Signing
                Cert Issuer:    VeriSign Class 3 Code Signing 2010 CA
                Serial Number:  F0 87 74 64 EC F2 AA 94 E0 4B 84 25 4D ED B5 4E
                Thumbprint:     117F5C5B276C2805D69A48F8B23C25883FCF5BE6
                Algorithm:      sha1RSA
                Valid from:     02:00 28/03/2012
                Valid to:       01:59 14/04/2015
```

Hook du driver NULL.SYS

Lors de son initialisation le rootkit cherche à poser un hook sur le Device `\Device\Null`. Pour ce faire il doit dans un premier temps obtenir son `DEVICE_OBJECT`, puis son `DRIVER_OBJECT` afin de pouvoir directement modifier sa table IRP.

```
RtlInitUnicodeString((PUNICODE_STRING)&dev_null,L"\\Device\\Null");
uVar1 = IoGetDeviceObjectPointer
                  ((PUNICODE_STRING)&dev_null,1,&PFILE_OBJECT_NULL,&PDEVICE_OBJECT_NULL);
[...]
NullDrvObj = PDEVICE_OBJECT_NULL->DriverObject;
if (NullDrvObj == (_DRIVER_OBJECT *)0x0) {
  ZwClose(EventHandle);
  EventHandle = (HANDLE)0x0;
  ObfDereferenceObject(PFILE_OBJECT_NULL);
  PFILE_OBJECT_NULL = (PFILE_OBJECT)0x0;
  PDEVICE_OBJECT_NULL = (PDEVICE_OBJECT)0x0;
  return 0xc0000034;
}
func_irp_deviceIoCtl_nullDrv = NullDrvObj->MajorFunction[0xe];
NullDrvObj->MajorFunction[0xe] = (PDRIVER_DISPATCH)0x140003d30;
```

Cette action est assez risquée de la part du rootkit, il est en effet fréquent de voir des codes malveillants modifier les objets de `\Device\Null`. L'entrée `0xe` du tableau `MajorFonction` correspond à la fonction `IRP_MJ_DEVICE_CONTROL`.

Une fois ce hook effectué il suffit d'ouvrir un Handle sur `\\.\NUL` pour communiquer via IoCtl avec le rootkit.

Communication IoCtl

Comme la majorité des rootkits, une communication est établie avec la DLL, celle-ci via deux numéros d'IoCtl :

```
ioctl_code != 0x156003 && (ioctl_code != 0x15e007)
```

Si l'IoCtl est correctement positionné le buffer associé servira à passer les commandes à exécuter. La structure est la suivante :

```
struct ioclt_buffer_struct {
    uint CodeId;
    uint DataSize;
    char Datas[];
};
```

Nous allons évoquer plusieurs commandes pouvant être appelées depuis le mode utilisateur.

getMagicNumber (0x200)

Ici certainement le code le plus simple.

```
*(undefined4 *)out_buffer = 0x41126;
*out_buffer_len = 4;
```

Il s'agit certainement d'un tag de vérification ou d'un numéro de version.

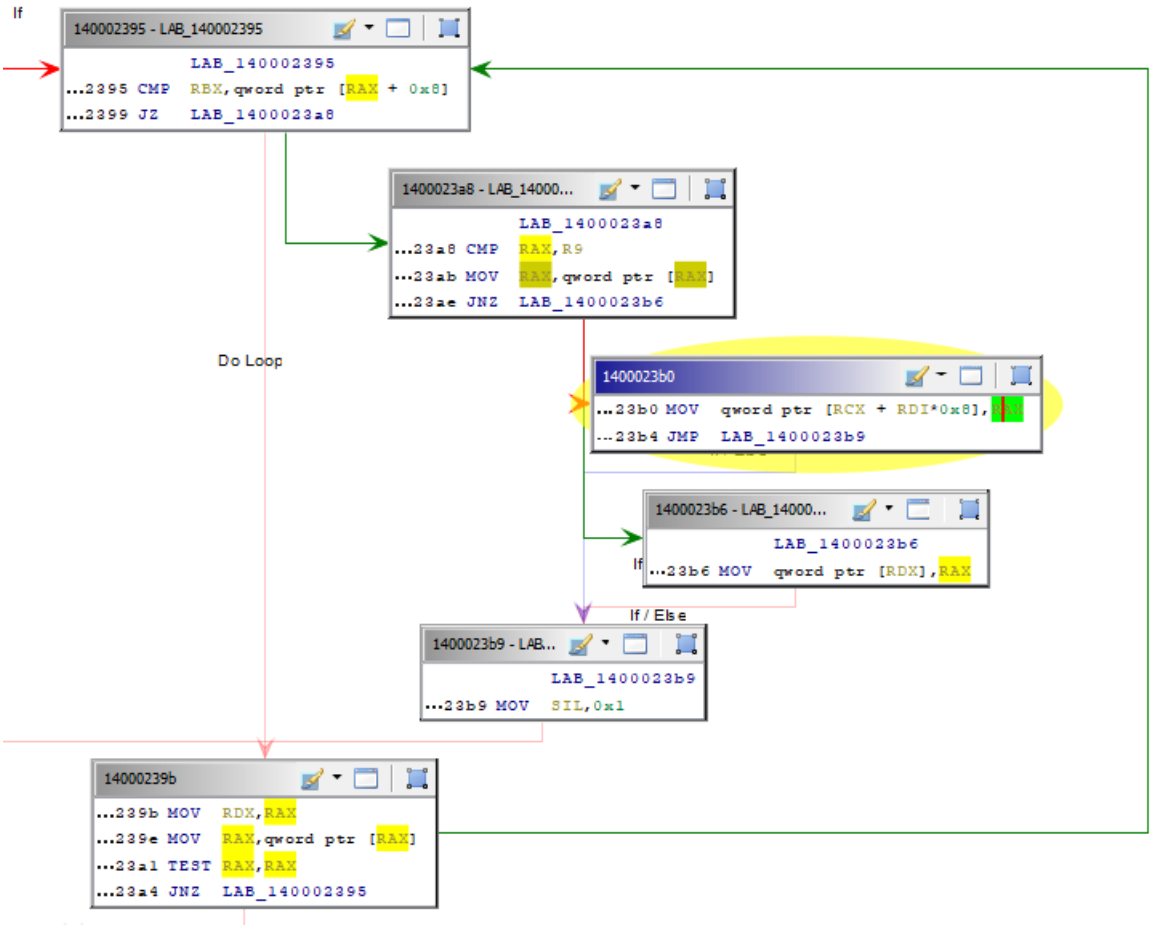
hideDriver (0x100)

Cette commande prend un argument en plus qui permet de déterminer la sous-action à effectuer :

- 1 : cacher le driver
- 2 : connaitre l'état du driver (caché ou visible)

Le driver est caché de plusieurs manières, tout d'abord les entêtes de l'exécutable sont réécrits avec des 0 afin d'éviter d'être identifié depuis une simple recherche des magic `MZ` et `PE`.

Puis le driver va énumérer les entrées de `\Driver` jusqu'à trouver son `DRIVER_OBJECT` affecté lors de son chargement. Une fois l'objet identifié le pointeur `FLINK` (objet suivant) de la liste chaînée est remplacé par le driver suivant. Cette action permet de cacher l'instance du driver dans le répertoire `\Driver` de l'object manager.



La même opération est effectuée dans `\Device` avec le `DEVICE_OBJECT` du driver (bien qu'il ne créé pas de `DEVICE_OBJECT` lui étant associé).

Bien que le driver soit supprimé de la liste dans le « Directory Object » il détruit également plusieurs informations lui étant liées pour éviter d'être révélé par certains modules d'analyse forensics de la mémoire.

```
driver_base = (short *)ownDrvObject->DriverStart;
BVar1 = MmIsAddressValid(driver_base);
SizeOfHeaders = 0;
if (((BVar1 != '\0') && (SizeOfHeaders = 0, *driver_base == 0x5a4d)) &&
   (SizeOfHeaders = 0,
   *(int *)((longlong)*(int *)(driver_base + 0x1e) + (longlong)driver_base) == 0x4550)) {
  SizeOfHeaders =
       (ulonglong)*(uint *)((longlong)*(int *)(driver_base + 0x1e) + 0x54 + (longlong)driver_base)
  ;
}
wipe_datas_with_mdl(ownDrvObject->DriverStart,SizeOfHeaders);
ownDrvObject->Type = 0;
ownDrvObject->DriverStart = (PVOID)0x0;
ownDrvObject->DriverSize = 0;
```

Cette même action est effectuée avec le `DEVICE_OBJECT`, bien qu'encore une fois aucun « device » ne lui a été affecté.

```
ownDeviceObject = ownDrvObject->DeviceObject;
if (ownDeviceObject != (PDEVICE_OBJECT)0x0) {
  ownDeviceObject->Type = 0;
  ownDrvObject->DeviceObject->Size = 0;
  ownDeviceObject = ownDrvObject->DeviceObject;
  ownDeviceObject->DeviceType = 0;
}
is_wiped = 1;
```

SetIpAndPort (0x600)

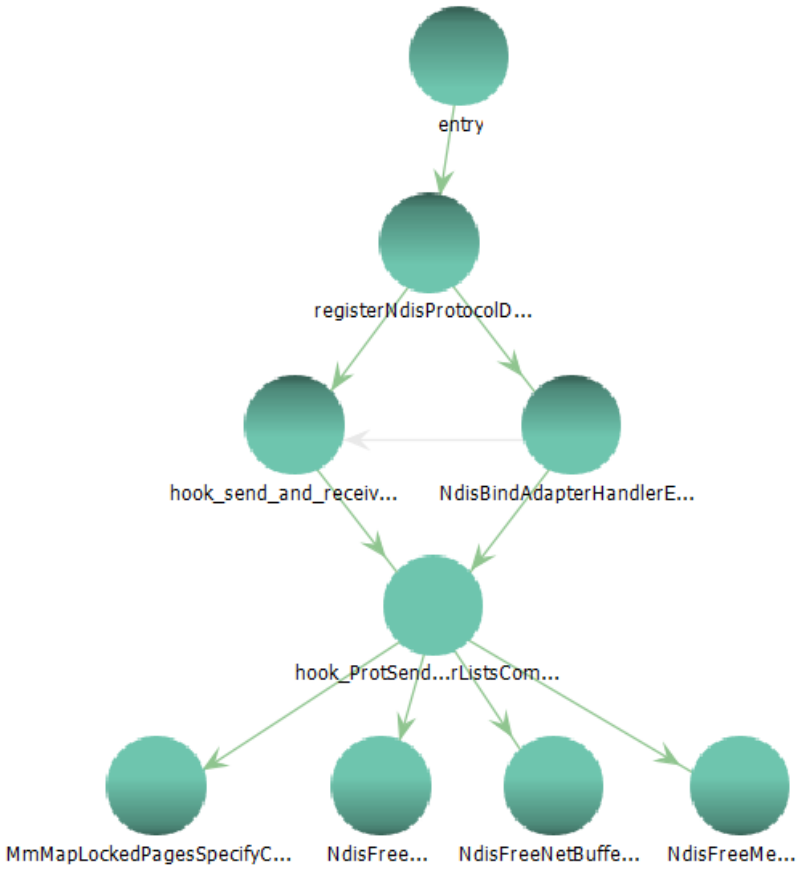
Ce code permet d'annoncer le server à contacter pour valider l'utilisation des injections réseaux via NDIS. Nous reviendrons sur son utilité dans la suite du papier.

send_packet (0x400)

Sous certaines conditions le driver peut autoriser l'envoi de paquets Ethernet sur l'interface réseau. Les données envoyées sont contenues dans le buffer transmis au noyau. Les conditions de cet envoi seront présentées dans la partie suivante.

Hooks NDIS et injections réseaux

Le rootkit possède d'intéressantes fonctionnalités réseau, en effet le rootkit se positionne au niveau NDIS pour communiquer au plus proche de la carte réseau. Globalement le positionnement des hooks NDIS depuis l'EntryPoint se fait comme suit :



La fonction `registerNdisProtocolDriver` va dans un premier temps rechercher dans les protocoles l'instance nommée `TCPIP`. Le parcours se fait via une liste simplement chaînée.

```
uVar2 = NdisRegisterProtocolDriver(0,&ProtocolCharacteristics,&ndis_registration);
if (-1 < (int)uVar2) {
  if (_struct_version < 0x6001e) {
    ndisProtocols = (NDIS_PROTOCOL_BLOCK *)ndis_registration->NextProtocol;
    while( true ) {
      cp_OpenQueue = (_NDIS_OPEN_BLOCK *)0x0;
      ndis_current_protocol = (NDIS_PROTOCOL_BLOCK *)0x0;
      if (ndisProtocols == (NDIS_PROTOCOL_BLOCK *)0x0) break;
      BVar1 = RtlEqualUnicodeString((UNICODE_STRING *)&ndisProtocols->Name,&tcp_ip_ustr,'\x01');
      if (BVar1 != '\0') {
        cp_OpenQueue = ndisProtocols->OpenQueue;
        ndis_current_protocol = (NDIS_PROTOCOL_BLOCK *)0x0;
        ndis_register_struct_getted = ndisProtocols;
        break;
      }
      ndisProtocols = (NDIS_PROTOCOL_BLOCK *)ndisProtocols->NextProtocol;
    }
  }
```

Nous avons ici le parcours des protocoles enregistrés. Une fois `TCPIP` (ici `tcp_ip_ustr`) trouvé, deux fonctions vont être remplacées : `ReceiveNetBufferLists` et `ProtSendNetBufferListsComplete`. Ces fonctions sont utilisées pour la réception et l'envoi des paquets du protocole.

```
*tmp_network_obj.pReceiveNetBufferLists = hook_ReceiveNetBufferLists;
*tmp_network_obj.pProtSendNetBufferListsComplete =
hook_ProtSendNetBufferListsComplete;
```

Nous allons nous attarder sur la fonction `hook_ReceiveNetBufferLists`. Cette fonction reçoit les paquets réceptionnés depuis la carte réseau. Chaque paquet sera analysé et suivant l'initialisation du driver et les données contenues dans le paquet certaines fonctions du rootkit seront activées.

Il est intéressant de noter que le rootkit possède un parseur de paquets réseau.

En premier lieu il vérifie que la taille du paquet est bien supérieure à 0x35 octets, tous les paquets TCP étant plus grands ceci permet un premier filtrage. Puis il vérifie le type de protocole suivant, 0x800 étant IP. Le rootkit vérifie ensuite la version de IP (4 pour IPv4) et que le protocole suivant est bien TCP.

```
if ((0x35 < data_length) &&
   ((ushort)((ushort)packet_buffer->EthernetNextType >> 0x8 |
            packet_buffer->EthernetNextType << 0x8) == 0x800)) {
  start_ip_protocol_buffer = (byte *)&packet_buffer->ip_VersionHlen;
  if (((packet_buffer->ip_VersionHlen & 0xf0U) == 0x40) &&
     (packet_buffer->ip_next_protocol == '\x06')) {
```

Ensuite, l'adresse IP source (donc du serveur distant) est comparée à une variable globale. Cette variable peut être initialisée depuis une commande IoCtl `SetIpAndPort`. Il est donc nécessaire d'annoncer l'adresse IP du C&C pour décoder totalement le paquet.

L'enregistrement dans l'IoCtl est fait comme suit :

```
code_from_buffer = in_buffer->field_0x8;
if (code_from_buffer == 0x0) {
  return 0x0;
}
 [...]
_trusted_tcp_dest_port =
     (ushort)*(byte *)&in_buffer->field_0xd + *(short *)&in_buffer->field_0xc * 0x100;
_trusted_ip_2 = code_from_buffer;
```

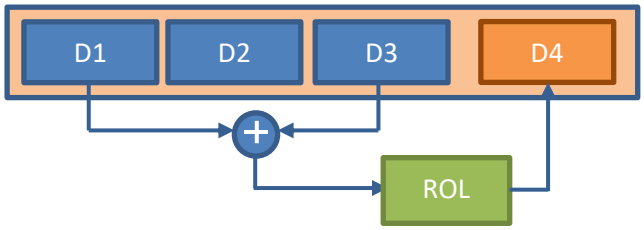
Et le contrôle de l'adresse IP source :

```
if ((_trusted_ip_2 == packet_buffer->ip_src) &&
   (_trusted_tcp_dest_port == start_tcp_protocol_buffer->tcp_dst_port)) {
```

Puis un « checksum » est effectué sur les données :

```
 ((uVar7 = start_datas_buffer[0x2] ^ *start_datas_buffer,
  (uVar7 << 0x10 | uVar7 >> 0x10) == start_datas_buffer[0x3] &&
  (uVar8 = check_ethernet_addr(packet_buffer), (char)uVar8 != '\0')))) {
_trusted_tcp_src_port = start_tcp_protocol_buffer->tcp_src_port;
_trusted_ip_src = packet_buffer->ip_src;
```

Ce « checksum » est très simple et consiste à effectuer une opération XOR entre le premier DWORD et le troisième DWORD, puis faire une rotation de 0x10 du résultat et le stocker dans le quatrième DWORD.



Si cette vérification est validée le rootkit va référencer le handle (`OpenQueue`) courant dans une variable globale. Cette variable sera utilisée pour envoyer des paquets en brut sur le réseau.

Nous pensons que l'objectif visé par toutes ces vérifications est d'identifier si le serveur distant peut bien être contacté et identifier l'interface avec laquelle émettre des paquets bruts par la suite.

Une fois toutes ces conditions remplies nous pouvons émettre des paquets réseaux sur l'interface identifiée, pour ce faire il faut envoyer les données via une commande IoCtl `send_packet`. Ci-dessous nous avons donc émis un paquet forgé avec des données arbitraires.

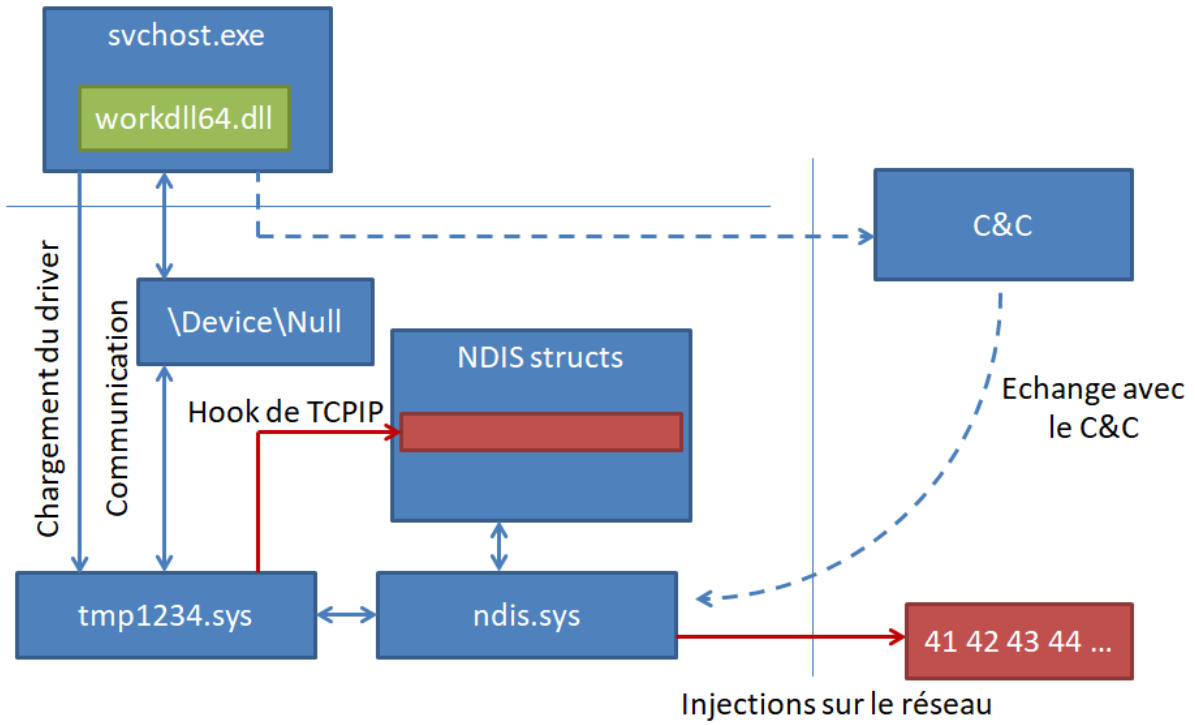
```
    23 15.853324     30:31:32:33:34:35    41:42:43:44:45:46    0x8666    1408 Ethernet II
> Frame 23: 1408 bytes on wire (11264 bits), 1408 bytes captured (11264 bits) on interface 0
> Ethernet II, Src: 30:31:32:33:34:35 (30:31:32:33:34:35), Dst: 41:42:43:44:45:46 (41:42:43:44:45:46)
v Data (1394 bytes)
    Data: 000045786154726163 6b2073656e642061207261772070617...
    [Length: 1394]

0000  41 42 43 44 45 46 30 31  32 33 34 35 86 66 00 00   ABCDEF01 2345.f..
0010  45 78 61 54 72 61 63 6b  20 73 65 6e 64 20 61 20   ExaTrack  send a
0020  72 61 77 20 70 61 63 6b  65 74 00 00 00 00 00 00   raw pack et......
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
```

Pour résumer, si l'on veut envoyer des paquets bruts sur le réseau, il faut :

1. Charger le driver
2. Communiquer avec \Device\Null
3. Envoyer un IOCTL pour configurer l'adresse IP/port du C&C
4. Echanger avec le C&C pour récupérer le checksum, qui validera l'interface réseau à utiliser
5. Envoyer un IOCTL pour émettre un paquet en brut

Le schéma suivant résume le processus d'envoi de paquets en brut sur le réseau :



Conclusions

Nous avons vu que cet attaquant prépare ses opérations probablement longtemps à l'avance étant donné la signature du rootkit et les fonctionnalités NDIS utilisées. Le rootkit possède également des techniques pour cacher sa présence, mais celles-ci ne sont certainement plus utilisées aujourd'hui, à cause de PatchGuard.

Références

[1] Partage de Florian Roth :
https://twitter.com/cyb3rops/status/1153983440871669761

[2] Présentation de Takahiro Haruyama évoquant le rootkit :
https://hitcon.org/2016/pacific/0composition/pdf/1201/1201%20R2%201610%20winnti%20polymorphism.pdf