La rétro-ingénierie de code malveillant dans la CTI - Analyse de l'évolution d'une chaîne d'infection

Charles Meslay
charles.meslay@sekoia.io

Sekoia.io

Résumé. Cet article a pour but de présenter le rôle de la rétro-ingénierie des codes malveillants dans le domaine du renseignement sur la menace cyber ou Cyber Threat Intelligence (CTI). Après un bref rappel de ce qu'est que la CTI, nous partirons d'un cas réel d'investigation autour de la chaîne d'infection du code FlowCloud lié à un mode opératoire adverse nommé « TA410 ». Cet article présentera les mécanismes de protection de ce code : mécanismes anti-analyse et chiffrement des différentes étapes de la chaîne d'infection. Les résultats de ces travaux seront ensuite utilisés afin de créer une nouvelle règle de détection YARA qui permettra de trouver un nouveau variant disposant de zéro de détection sur VirusTotal.

1 Introduction

1.1 Qu'est-ce que la CTI ?

La Cyber Threat Intelligence (CTI) ou renseignement sur la menace cyber correspond à l'étude des groupes d'attaquants informatique. Ce domaine de recherche, multidisciplinaire, a pour but d'étudier les modes opératoires adverses (MOA) dans le but de les détecter au plus tôt pour se protéger de ceux-ci. La CTI s'articule autour de plusieurs domaines d'expertise. Pour faciliter la compréhension, nous nous limitons ici à uniquement trois d'entre eux.

Le premier n'est pas technique, mais lié à la connaissance stratégique. Le but de ce domaine est d'étudier le contexte dans lequel sont effectuées les attaques, qu'il s'agisse de campagnes étatiques ou cybercriminelles. L'analyse des enjeux politiques entre pays peut ainsi permettre d'expliquer certaines attaques et donc d'anticiper de futures opérations adverses. Cette compétence intègre aussi une connaissance de l'organisation des groupes d'attaquants. Cela peut aussi bien correspondre à l'étude de l'organisation de groupes cybercriminels que l'organisation cyber d'un Etat.

Un deuxième domaine lié à la CTI est l'étude des groupes d'attaquants via le suivi de leur infrastructure. Comme présenté lors de l'édition 2023

du SSTIC par Charles Hourtoule et Simon Msika [2], un MOA comme Mustang Panda, réputé d'origine chinoise, peut être suivi via des spécificités dans la configuration de leurs serveurs : l'identification d'un motif dans le certificat d'un serveur permet d'illuminer une partie de l'infrastructure de l'attaquant. Ainsi, un suivi continu de cette infrastructure permettra de détecter une compromission non pas via les traces laissées sur un poste mais via les connexions réseaux.

Enfin, un dernier domaine est l'analyse des codes malveillants utilisés par un MOA. Bien que les objectifs soient multiples, cette analyse permet de documenter de façon précise les codes utilisés afin :

- d'en extraire des indicateurs de compromissions ;
- d'identifier des moyens de remédiations ;
- de rapprocher des menaces entre elles et ainsi de suivre une menace dans le temps.

Cet article a pour but d'explorer ce dernier domaine. Nous tenterons d'expliquer comment l'analyse de code via la rétro-ingénierie permet de suivre l'évolution de codes malveillants via l'exemple du code FlowCloud associé au mode opératoire TA410.

1.2 Présentation du MOA TA410 et de la chaîne d'infection de FlowCloud

TA410 est un MOA supposé d'origine chinoise actif depuis au moins 2019. Ce MOA a été pour la première fois documenté en 2020 par ProofPoint à l'occasion d'une campagne d'attaque contre des fournisseurs d'énergie aux États Unis [4]. La description de cette campagne par Proofpoint fait état d'un implant nommé FlowCloud, qui donne un accès complet au système compromis.

En avril 2022, les chercheurs d'ESET, ont décrit de façon détaillée la chaîne d'infection et les fonctionnalités de FlowCloud [1]. La chaîne d'infection est assez complexe mais est typique des chaines d'infection habituellement rencontrées. Sans rentrer dans l'ensemble de la chaîne d'infection, nous pouvons retenir que celle-ci commence par la création d'un service qui exécutera une application vulnérable à du « DLL-side-loading » afin de charger une DLL malveillante. Cette DDL, nommée `XXXModule_dlcore0` charge une autre DLL qui elle-même déchiffre et exécute un autre code qui chargera FlowCloud.

1.3 YARA, un outil de signature de code

Dans les annexes de l'article de blog, ESET fournit des hashs de fichiers qu'il est possible de récupérer sur VirusTotal ainsi que plusieurs « règles YARA » dont une sur les techniques anti-analyses de FlowCloud. YARA est un outil permettant d'effectuer des recherches dans des fichiers à partir de règles dans lesquelles des recherches sur des motifs spécifiques sont définis (qu'il s'agisse de chaînes de caractères ou directement basés sur une suite d'octets). Des règles YARAs sont ainsi utilisées afin de « signer » des codes (ou des sous-parties de codes).

Un des objectifs du travail de rétro-ingénierie est d'identifier des motifs spécifiques aux maliciels afin de pouvoir les détecter et les classer.

C'est ce qui est fait ici par ESET en fournissant à la communauté une règle YARA qui signe les mécanismes anti-analyse utilisés dans FlowCloud (figure 1).

```
rule apt_Windows_TA410_FlowCloud_malicious_dll_antianalysis
{
    meta:
        description = "Matches anti-analysis techniques used in TA410 FlowCloud hijacking DLL."
        reference = "https://www.welivesecurity.com/"
        source = "https://github.com/eset/malware-ioc/"
        license = "BSD 2-Clause"
        version = "1"
        author = "ESET Research"
        date = "2021-10-12"
    strings:
    /*
        33C0                xor eax, eax
        E8320C0000              call 0x10001d30
        83C010              add eax, 0x10
        3D00000080              cmp eax, 0x80000000
        7D01                jge +3
        EBFF                jmp +1 / jmp eax
        E050                loopne 0x1000115c / push eax
        C3                  ret
    */
        $chunk_1 = {
            33 C0
            E8 ?? ?? ?? ??
            83 C0 10
            3D 00 00 00 80
            7D 01
            EB FF
            E0 50
            C3
        }
    condition:
        uint16(0) == 0x5a4d and all of them
}
```

Fig. 1. Règle YARA fournie par ESET Research

Une règle YARA est généralement divisée en trois sections :

— la section `condition` est une expression booléenne qui définit la logique de la règle en utilisant les variables définies dans la section `strings` ;

— la section `strings` permet de définir des variables utilisables dans la section `condition` ;

— la section `meta` permet de définir des metadonnées.

Cette règle vérifie la présence de deux choses :

— `uint16(0) == 0x5a4d` : qui correspond à la présence des octets `0x4d 0x5a` au début du fichier. Il s'agit ici de vérifier la présence de l'en-tête `MZ` spécifique aux fichiers PE sous Windows.

— la présence de la valeur `$chunk_1` (via les mots clés « `all of them` »). Cette variable contient une suite d'octets correspondant à une séquence particulière de code assembleur. L'auteur de cette règle a précisé le sens de ces octets en commentaire.

L'intérêt de créer des règles YARA est multiple. Tout d'abord, d'un point de vue détection, l'objectif est de pouvoir analyser les fichiers d'un système au regard de ses règles afin d'identifier des codes malveillants. Cela peut par exemple être utilisé par des EDR (« Endpoint Detection Response » ou des antivirus). Un autre intérêt est de pouvoir suivre dans le temps l'évolution des codes malveillants ou tenter de trouver de nouveaux variants en soumettant ces règles à des bases de données de fichiers.

Il est important de noter que la création d'une règle YARA nécessite toujours d'effectuer un compromis entre :

— définir des critères suffisamment précis pour être sûr qu'une règle est spécifique à un code particulier, c'est à dire réduire au maximum le nombre de faux positifs.

— être suffisamment générique afin de détecter, si possible, tous les codes de cette famille, c'est à dire réduire au maximum le nombre de faux négatifs.

Dans la suite de cet article nous montrerons la démarche que nous avons adopté afin d'analyser des variants du chargeur de FlowCloud ainsi que la création d'une règle YARA pertinente.

2 Suivi de la menace dans le temps via le chargeur de FlowCloud

L'article d'ESET contient des Indicateurs de Compromissions (« *Indicator Of Compromises* » ou « *IoC* »). Parmis ces IoCs, des hashs de

fichiers sont présents. L'un d'eux étant disponible sur VirusTotal,[1] nous décidons de débuter notre investiguation par l'analyse de ce fichier.

Dans ce chapitre, nous présenterons les principales étapes de la rétro-ingénierie de ce code ainsi que d'autres variants connus dans le but de comprendre les spécificités de ces codes. Nous pourrons ensuite en extraire des motifs spécifiques pour créer notre propre règle YARA. Celle-ci sera enfin utilisée pour tenter identifier de nouveaux variants.

2.1 Analyse du fichier initial

Après avoir récupéré le fichier sur VirusTotal, nous pouvons procéder à son analyse. Lors de l'ouverture du logiciel malveillant à l'intérieur d'un outil tel que IDA Pro, on constate qu'il échoue à l'analyse. En effet, de nombreuses portions de code sont considérés par IDA Pro comme des données brutes (IDA Pro n'a pas réussi à désassembler ces octets, en gris sur la figure 2) ou comme des instructions en dehors d'une fonction (IDA Pro n'a pas été capable de reconstruire la fonction associée, en rouge sur la figure 2).

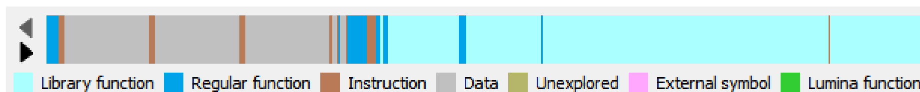

Fig. 2. Vue schématisée du résultat de la décompilation du maliciel par IDA Pro

Comme nous allons le voir, l'analyse dans IDA Pro a échoué à cause d'une technique d'obfuscation utilisée dans ce code. Cette technique anti-analyse correspond à la portion de code signée par la règle YARA de la figure 1. Plus précisément la technique d'obfuscation utilisée est visible dans la figure 3.

Pour comprendre ce qu'il se passe ici, prenons le temps de comprendre comment fonctionne IDA Pro. Le désassembleur d'IDA Pro traite tous les octets les uns à la suite des autres. Ainsi, lorsqu'il arrive à l'adresse `0x100010B7`, le désassembleur :

1. identifie que cette instruction est codée sur deux octets et qu'il s'agit d'un saut conditionnel vers l'adresse `0x100010BA` ;
2. désassemble l'instruction suivante, qui commence à l'adresse `0x100010B9` dont la taille est de deux octets. Cette instruction aura pour effet d'effectuer un saut vers, elle aussi, l'adresse `0x100010BA`.

[1] `https://www.virustotal.com/gui/file/c0568d6c0aa6d019454c9613b1a9b0ef`

```
.text:100010A8 33 C0                                       xor     eax, eax
.text:100010AA E8 81 0C 00 00                              call    sub_10001D30
.text:100010AF 83 C0 10                                    add     eax, 10h
.text:100010B2 3D 00 00 00 80                              cmp     eax, 80000000h
.text:100010B7 7D 01                                       jge     short near ptr loc_100010B9+1
.text:100010B9
.text:100010B9                             loc_100010B9:
.text:100010B9 EB FF                                       jmp     short near ptr loc_100010B9+1
.text:100010B9                             ; ---------------------------------------------------
.text:100010BB E0                                          db 0E0h
.text:100010BC                             ; ---------------------------------------------------
.text:100010BC 50                                          push    eax
.text:100010BD C3                                          retn
.text:100010BD                             ; ---------------------------------------------------
.text:100010BE 75 E8                                       dw 0E875h
.text:100010C0 8C 0C 00 00 83 F8 01 0F…                    dd 0C8Ch, 0F01F883h, 8F3384h, 2086800h
```

Fig. 3. Exemple du premier motif utilisé pour mettre en défaut IDA Pro

Ici, nous voyons que le désassembleur, qui réalise une analyse séquentielle des octets, est mis en défaut puisque la prochaine instruction exécutée est à l'adresse `0x100010BA` qui est au milieu d'une instruction. Cette technique est ainsi appelée, « *Jump In The Middle* ».

À partir de cette adresse, nous pouvons considérer le désassembleur comme « désynchronisé » par rapport au flux d'exécution réel du programme : les instructions suivantes ne correspondent alors plus aux instructions réellement exécutées par le programme.

Dans ce cas, le travail du rétro-analyste est d'aider le désassembleur à gérer cette situation. Pour cela, une solution est de supprimer l'instruction à l'adresse `0x100010B9` et d'en créer une nouvelle à l'adresse `0x100010BA`. Cela permet de voir que cette nouvelle instruction est un saut inconditionnel, « `jump eax` » où la valeur de `eax` dépend du retour de la fonction `sub_10001d30`.

```
.text:100010A8 33 C0                                       xor     eax, eax
.text:100010AA E8 81 0C 00 00                              call    sub_10001D30
.text:100010AF 83 C0 10                                    add     eax, 10h
.text:100010B2 3D 00 00 00 80                              cmp     eax, 80000000h
.text:100010B7 7D 01                                       jge     short loc_100010BA
.text:100010B7                             ; ---------------------------------------------------
.text:100010B9 EB                                          db 0EBh
.text:100010BA                             ; ---------------------------------------------------
.text:100010BA
.text:100010BA                             loc_100010BA:
.text:100010BA FF E0                                       jmp     eax
.text:100010BA                             ; ---------------------------------------------------
.text:100010BC 50                                          db  50h ; P
.text:100010BD C3                                          db 0C3h
```

Fig. 4. Première passe de modifications pour corriger l'analyse effectuée par IDA Pro

Nous ne rentrerons pas dans le détail de la fonction `sub_10001D30`, mais celle-ci est utilisée pour vérifier que la DLL ne s'exécute pas dans une *sandbox*. L'exécution dans un débogeur permet de voir que la valeur de retour, `eax` vaut :

— `1` si le code détecte un environnement d'analyse. Dans ce cas, le `jmp eax` effectue un saut vers l'adresse `0x11`, ce qui génère une erreur.

— L'adresse de retour de la fonction `sub_10001D30`, c'est à dire `0x100010AF` dans le cas présent. Le `jmp eax` effectuera donc un saut vers l'adresse `0x100010AF+0x10=0x100010BF`.

Nous pouvons alors simplifier le code en remplaçant certains octets par la valeur 0x90 qui correspond à l'instruction « `no operation` » :

— les octets qui correspondent aux instructions de saut (« `jge` » ansi que le premier octet du « `jmp` ») ;

— les trois octets à partir de l'adresse `0x100010BC` car ces octets ne sont jamais « exécutés ».

En résultat, nous obtenons la figure 5 qui est une version simplifiée mais équivalente à ce qui est réellement exécuté.

```
.text:100010A8 33 C0                              xor     eax, eax
.text:100010AA E8 81 0C 00 00                     call    sub_10001D30
.text:100010AF 83 C0 10                           add     eax, 10h
.text:100010B2 3D 00 00 00 80                     cmp     eax, 80000000h
.text:100010B7 90                                 nop
.text:100010B8 90                                 nop
.text:100010B9 90                                 nop
.text:100010BA
.text:100010BA                    loc_100010BA:
.text:100010BA FF E0                              jmp     eax
.text:100010BC                    ; ---------------------------------------------------
.text:100010BC 90                                 nop
.text:100010BD 90                                 nop
.text:100010BE 90                                 nop
.text:100010BF E8 8C 0C 00 00                     call    sub_10001D50
.text:100010C4 83 F8 01                           cmp     eax, 1
.text:100010C7 0F 84 33 8F 00 00                  jz      near ptr ExitProcess
```

Fig. 5. Seconde passe de modifications pour corriger l'analyse effectuée par IDA Pro

Notons que la fonction `sub_10001D50` est une autre anti-debug. Si le résultat est 1, le processus se termine.

Dans cet exemple, nous avons manuellement aidé IDA Pro, mais ce motif (ainsi que les fonctions anti-debug) est répété de multiples fois (32 fois dans cet exécutable). Il est alors nécessaire d'automatiser le processus de suppression de ces motifs et des appels aux fonctions anti-debug. Dans ce cas précis, il est possible de remplacer l'ensemble des octets de la

figure 5 par des octets « *no operation* ». En effet, le rôle de ces octets est uniquement lié à de la détection d'environnement d'analyse.

En se basant sur l'exemple de plugin IDA Pro de Rolf Rolles,[2] il suffit alors de modifier la fonction `ev_ana_insn` comme le montre la figure 6.

```
def ev_ana_insn(self, insn):
    cur = idaapi.get_byte(insn.ea)
    #if the current byte is 0x33, we can check for the pattern
    if cur == 0x33:
        a = idaapi.get_byte(insn.ea+1)
        b = idaapi.get_byte(insn.ea+2)
        # 0x33 0xC0: xor eax, eax
        # 0xe8 : indicates a call
        if a == 0xC0 and b == 0xe8:
            # if next bytes (after the call) corresponds to :
            # add eax, 10
            # cmp eax, 80000000
            pattern_eset =  b'\x83\xc0\x10=\x00\x00\x00\x80}\x01'
            if idaapi.get_bytes(insn.ea+7, 10) == pattern_eset:
                #we replace all these bytes by 0x90
                idaapi.patch_bytes(insn.ea, b"\x90"*0x25)
    return False
```

Fig. 6. Adaptation de la fonction `ev_ana_insn`

Ce script est assez simple : il vérifie que l'octet courant est le début d'un bloc à supprimer. Si c'est le cas, nous remplaçons ce bloc par 37 instructions « `no operation` » (`0x25` dans la figure 6). IDA Pro est alors en capacité de reconstruire la fonction et de produire un code décompilé automatiquement. Après quelques renommages de variables et fonctions, retypages triviaux et l'ajout de quelques commentaires, on obtient le code C présenté dans la figure 7.

Il est ainsi assez aisé d'identifier les principales étapes de cette fonction :

1. récupération du chemin actuel de l'exécutable ;
2. ouverture et lecture du fichier `setlangloc.dat`. Le contenu est copié dans une zone mémoire nouvellement allouée ;
3. création d'un patch pour modifier certaines instructions dans ce processus afin d'appeler le code à l'intérieur de `setlangloc.dat` ;
4. application ce patch.

La figure 8 présente les données du patch dans le segment `rdata`.

Une première piste identifiée pour créer une nouvelle règle YARA est d'utiliser les données du patch. Ces octets pourront être utilisés pour créer un nouveau motif de détection dans la notre règle YARA.

[2] `https://github.com/RolfRolles/FinSpyVM/blob/master/FinSpyDeob.py`

```
int load_setlangloc_dat()
{
  v6 = 0;
  FileContent = 0;
  dwSize = 0;
  memset(Filename, 0, sizeof(Filename));
  // Current path
  GetModuleFileNameW(0, Filename, 0x104u);
  // Remove the filename from the path
  PathRemoveFileSpecW(Filename);
  // add "setlangloc.dat" to the path
  PathAppendW(Filename, L"setlangloc.dat");
  // Open and ReadFile
  FileContent = OpenAndReadFile(Filename, &dwSize);
  if ( !FileContent )
    return v6;
  hmem = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  // Copy file content into new memory
  memcpy(hmem, FileContent, dwSize);
  // retrieve the address to patch
  addressToPatch = (GetModuleHandleW(0) + 0x2D7CE);
  // Create the patch
  v2[0] = 0x68;                              // 0x68: push opcode
  *&v2[4] = 0x9090C312;                      // 0xc3: ret opcode
  *&v2[1] = hmem;                            // with "0x68" => Push hmem on the stack
  // Temporarily modify memory protection to allow writing
  VirtualProtect(addressToPatch, 0xBu, 0x40u, &flOldProtect);
  // Apply the patch
  *addressToPatch = *v2;
  addressToPatch[1] = *&v2[4];
  *(addressToPatch + 4) = 0x9090;
  *(addressToPatch + 10) = 0x90;
  // Restore original memory protection
  VirtualProtect(addressToPatch, 0xBu, flOldProtect, &flOldProtect);
  return v6;
}
```

Fig. 7. Résultat final suite à l'automatisation de la désobfuscation

```
.rdata:1000BB58 68 78 56 34          dword_1000BB58  dd 34567868h
.rdata:1000BB5C 12 C3 90 90          dword_1000BB5C  dd 9090C312h
.rdata:1000BB60 90 90 90 00          dword_1000BB60  dd 909090h
```

Fig. 8. Octets du patch dans le segment `rdata`

2.2 Analyse de variants

À partir de la règle YARA sur le motif anti-analyse, deux autres fichiers sont trouvés sur VirusTotal que nous nommerons « `Variant A` »[3] et « `Variant B` ».[4]

Ces fichiers sont connus comme étant également liés à FlowCloud. Dans le but d'améliorer notre signature, nous allons analyser ces fichiers afin d'identifier leur caractéristiques et spécificités.

Variant A. Ce variant présente de nombreuses similarités avec le chargeur initial de FlowCloud. Néanmoins, lors de son chargement dans IDA Pro,

[3] `https://www.virustotal.com/gui/file/1e3baba3b7261eb9441fce16a1310532`
[4] `https://www.virustotal.com/gui/file/1c1ad9fd655ee80447f7bb38a570313b`

et malgré le script précédemment créé, le désassemblage échoue. Nous observons rapidement qu'un autre motif anti-analyse est présent, comme le montre la figure 9.

```
.text:10002F00
.text:10002F00                          loc_10002F00:
.text:10002F00 8B 44 24 FC                            mov     eax, [esp-4]
.text:10002F04 50                                     push    eax
.text:10002F05 33 C0                                  xor     eax, eax
.text:10002F07 58                                     pop     eax
.text:10002F08 74 01                                  jz      short near ptr loc_10002F0A+1
.text:10002F0A
.text:10002F0A                          loc_10002F0A:
.text:10002F0A E8 33 C0 E8 4E                         call    near ptr 5EE8EF42h
.text:10002F0F 04 00                                  add     al, 0
.text:10002F11 00 83 C0 10 3D 00                      add     [ebx+3D10C0h], al
.text:10002F11                          ; ---------------------------------------------------
.text:10002F17 00                                     db 0
.text:10002F18 00 80 7D 01 EB FF E0 50…               dd 17D8000h, 50E0FFEBh, 0D9E875C3h, 830
.text:10002F2C D0 B0 00 00 89 44 24 FC…               dd 0B0D0h, 0FC244489h, 24648D58h, 0FC81
```

Fig. 9. Première nouvelle technique anti-analyse

Cette technique consiste à sauvegarder la une valeur de `eax` sur la pile, mettre `eax` à zéro puis restaurer la valeur de `eax`. La mise à zéro de `eax` a pour effet que le saut conditionnel « `jz` » est pris. Le flux d'exécution saute alors au milieu du « `call` » suivant.

```
.text:10002F00
.text:10002F00                     loc_10002F00:
.text:10002F00 8B 44 24 FC                         mov     eax, [esp-4]
.text:10002F04 50                                  push    eax
.text:10002F05 33 C0                               xor     eax, eax
.text:10002F07 58                                  pop     eax
.text:10002F08 74 01                               jz      short loc_10002F0B
.text:10002F08                     ; ------------------------------------------------
.text:10002F0A E8                                  db 0E8h
.text:10002F0B                     ; ------------------------------------------------
.text:10002F0B
.text:10002F0B                     loc_10002F0B:
.text:10002F0B 33 C0                               xor     eax, eax
.text:10002F0D E8 4E 04 00 00                      call    sub_10003360
.text:10002F12 83 C0 10                            add     eax, 10h
.text:10002F15 3D 00 00 00 80                      cmp     eax, 80000000h
.text:10002F1A 7D 01                               jge     short near ptr loc_10002F1C+1
.text:10002F1C
.text:10002F1C                     loc_10002F1C:
.text:10002F1C
.text:10002F1C EB FF                               jmp     short near ptr loc_10002F1C+1
.text:10002F1C                     ; ------------------------------------------------
.text:10002F1E E0                                  db 0E0h
.text:10002F1F 50                                  db  50h ; P
.text:10002F20 C3                                  db 0C3h
```

Fig. 10. Désobfuscation de la nouvelle technique anti-analyse

La figure 10 présente le résultat après une correction manuelle de cette obfuscation. Nous remarquons qu'elle est immédiatement suivie du motif anti-analyse présenté précédemment. Nous pouvons donc mettre à jour notre script de désobfuscation afin de prendre en compte cette nouvelle technique.

Notons, qu'une autre technique d'obfsucation est présente. Celle-ci n'empêche pas l'analyse par IDA Pro, mais a pour effet de rajouter des variables au code obtenu après décompilation. Cette technique est présentée dans la figure 11.

```
.text:10001833 89 44 24 FC                          mov     [esp-2F9Ah+arg_2F8E], eax
.text:10001837 58                                   pop     eax
.text:10001838 8D 64 24 FC                          lea     esp, [esp-4]
.text:1000183C 81 FC 00 10 00 00                    cmp     esp, 1000h
.text:10001842 77 06                                ja      short loc_1000184A
.text:10001844 81 C4 D6 06 00 00                    add     esp, 6D6h
.text:1000184A
.text:1000184A                          loc_1000184A:
.text:1000184A 8B 44 24 FC                          mov     eax, [esp-3670h+arg_3664]
```

Fig. 11. Deuxième nouvelle technique d'obfuscation

Après avoir traité cette technique de la même façon que les précédentes (en remplaçant toutes ces instructions par des « `no operation` » dans notre plugin), nous pouvons recharger ce fichier dans IDA Pro. Nous observons que le désassemblage est correctement effectué comme le montre la figure 12.

```
int sub_10002990()
{
  v4 = 0;
  Src = 0;
  dwSize = 0;
  memset(name_setlangloc_dat, 0, sizeof(name_setlangloc_dat));
  // decrypt filename
  decrypt_filename(setlangloc_dat, 2048, name_setlangloc_dat);
  memset(Filename, 0, sizeof(Filename));
  // Get current path
  GetModuleFileNameW(0, Filename, 0x104u);
  // Remove the filename from the path
  PathRemoveFileSpecW(Filename);
  // add the decrypted filename to the path
  PathAppendW(Filename, name_setlangloc_dat);
  // Open and ReadFile
  Src = OpenAndReadFile(&dwSize, Filename, &dwSize);
  if ( !Src )
    return v4;
  // Decrypt the file
  DecryptFile(Src, Src, dwSize, 0xD3u, 0);
  v1 = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  memcpy_0(v1, Src, dwSize);
  memset(Src, 0, dwSize);
  operator delete(Src);
  Src = 0;
  dwSize = 0;
  *&name_setlangloc_dat[55] = 0;
  // Patch the current process
  PatchFile(v1, v1);
  return v4;
}
```

Fig. 12. Résultat de la décompilation du premier variant

Comme précédemment, cette figure n'a fait l'objet que de quelques renommages et ajouts de commentaires. Nous pouvons assez rapidement observer que ce code est similaire au code obtenu dans la figure 7.

Néanmoins, bien que de nombreuses similarités existent, nous observons trois différences avec le fichier initial :

— la première correspond au nom de fichier utilisé pour l'étape suivante. Auparavant, celui-ci était récupéré via plusieurs appels de fonctions (`GetModuleFileNameW`, `PathRemoveFileSpecW` et `PathAppendW`). Maintenant, ce nom de fichier est déchiffré dans une fonction dédiée.

— La seconde différence correspond au chargement du contenu de l'étape suivante. Dans le premier fichier, un simple `memcpy` était utilisé pour copier le contenu du fichier. Dans ce nouveau variant, une fonction est appelée afin de déchiffrer l'étape suivante : Cette seconde fonction de déchiffrement prend en paramètre une graine utilisée pour le calcul de la clé de déchiffrement. Comme la valeur `0xd3` est passée en paramètre de cette fonction, la valeur de la clé est alors `0x7b`. La figure 13 présente cette fonction chargée de dériver la clé est d'appliquer le déchiffrement.

```
unsigned int __cdecl DecryptFile(_BYTE *input, unsigned int size, unsigr
{
  Key = seed % 0x46B - 0x58;
  for ( i = 0; i < size; ++i )
  {
    if ( encrypt )
    {
      // input[i] = ((input[i] - Key) ^ Key) % 256
      *input -= Key;
      *input ^= Key;
    }
    else
    {
      // input[i] = ((input[i] ^ Key) + Key) % 256
      *input ^= Key;
      *input += Key;
    }
    ++input;
  }
  return i;
}
```

Fig. 13. Fonction dérivant la clé et déchiffrant l'étape suivante

— Enfin, une troisième différence correspond à l'application du patch. Ici, l'application du patch est déportée dans une autre fonction. La

figure 14 présente cette fonction. Celle-ci correspond à ce qui était effectué précédemment.

```
int __cdecl PatchFile(int addr)
{
  result = new_entrypoint;
  lpAddress = new_entrypoint;
  if ( !new_entrypoint )
    return result;
  // Configure the patch
  v2[0] = 0x68;
  *&v2[4] = 0x9090C312;
  *&v2[1] = addr;
  // Set memory protection to PAGE_EXECUTE_READWRITE
  VirtualProtect(new_entrypoint, 0xBu, PAGE_EXECUTE_READWRITE, &flOldProtect);
  // Apply the patch
  *lpAddress = *v2;
  lpAddress[1] = *&v2[4];
  *(lpAddress + 4) = 0x9090;
  *(lpAddress + 10) = 0x90;
  // Restore memory protection to original value
  return VirtualProtect(lpAddress, 0xBu, flOldProtect, &flOldProtect);
}
```

Fig. 14. Fonction qui applique le patch

Variant B. Le deuxième variant est très proche du premier variant. Il dispose par exemple des mêmes fonctions de chiffrement et du même mécanisme de dérivation de clé. Cela nous indique qu'il peut être intéressant de signer aussi ces mécanismes pour les inclure dans une règle YARA.

Un fait de comparaison intéressant entre ces trois versions est le nombre d'occurences des motifs d'obfuscation. Les résultats sont présentés dans le tableau 1

Fichier	motif ESET	motif obfuscation 1	motif obfuscation 2
Fichier initial	48	0	0
Variant A	69	136	136
Variant B	1	64	128

Tableau 1. Nombre d'occurences des motifs dans chaque fichier

Ces résultats montrent que le dernier variant ne dispose que d'une seule occurence du motif initial signé par la règle YARA d'ESET. Une hypothèse est que cette obfuscation a vocation à disparaitre dans les prochaines versions. Cela peut donc inciter à réaliser une nouvelle règle YARA dont la présence de ce motif est optionnelle.

2.3 Création de nouvelles signatures

À ce stade, nous disposons d'une règle YARA qui se base uniquement sur un motif anti-analyse. Or, comme nous avons pu le voir, toutes ces variantes disposent de plusieurs similarités (fonctions cryptographiques) mais aussi de deux nouvelles techniques anti-analyse.

Nous pouvons donc tenter de créer une nouvelle règle YARA basée sur de nouvelles heuristiques :

— les mécanismes de déchiffrement (du nom du fichier et du contenu de celui-ci)
— le mécanisme de dérivation de la clé pour le déchiffrement du contenu du fichier
— les deux nouveaux motifs anti-analyse
— les octets du patch appliqué

Signer des algorithmes cryptographique peut être intéressant si l'implémentation est spécifique au code étudié. Dans le cas présent, le déchiffrement du contenu du fichier correspondant juste à une addition suivant d'un xor ne semble pas opportun. En effet, cela pourrait mener à de nombreux faux positifs.

Par contre, bien que nous n'ayons pas abordé cette partie dans cet article par soucis de clarté, le déchiffrement du nom du fichier semble plus spécifique. La figure 15 présente un extrait de cette fonction.

```
for ( i = 0; i < v7; ++i )
  out[i] ^= (i + 38) ^ input[i % 4] ^ input[-(i % 4) + 7];
```

Fig. 15. Extrait de la fonction de déchiffrement du nom du fichier

De cet algorithme nous avons extrait les octets suivants :

```
$decryption_function = {8A C8 80 C1 26 32 D1 30 14 38}
```

Le mécanisme de dérivation des clés peut lui être signé par cette variable :

```
$derivation_key = {6B 04 00 00 F7 ?? 81 c2 a8 01 00 00}
```

Enfin, nous avons les deux nouveaux motifs anti-analyses ainsi que les octets du patch :

```
$new_pattern_1 = {50 33 c0 58 74 01 e8}
$new_pattern_2 = {89 44 24 fc 58 8D 64 24 fc 81 fc 00 10 00 00 77
↪ 06 81 c4 ?? ?? ?? ?? 8B 44 24 FC}
$patch_bytes = {68 78 56 34 12 C3 90 90 90 90 90 00}
```

Au final, notre nouvelle règle YARA est présentée dans la figure 16.

```
rule APT_FlowCloud_Loader{
    meta:
        id = "60792b78-8e22-4a52-9917-a39a769087d4"
        version = "1.0"
        malware = "FlowCloud"
        intrusion_set = "TA410"
        description = "Detects FlowCloud Loader"
        source = "Sekoia.io"
        creation_date = "2023-12-07"
        classification = "TLP:WHITE"
    strings:
        $decryption_function = {8A C8 80 C1 26 32 D1 30 14 38}
        $derivation_key = {6B 04 00 00 F7 ?? 81 c2 a8 01 00 00}
        $new_pattern_1 = {50 33 c0 58 74 01 e8}
        $new_pattern_2 = {89 44 24 fc 58 8D 64 24
                          fc 81 fc 00 10 00 00 77
                          06 81 c4 ?? ?? ?? ?? 8B
                          44 24 FC}
        $patch_bytes = {68 78 56 34 12 C3 90 90 90 90 90 00}
    condition:
        uint16be(0) == 0x4d5a and filesize < 4MB and 2 of them

}
```

Fig. 16. Nouvelle règle YARA signant les variants

Ici, notre objectif est de trouver de nouveaux fichiers. Nous décidons donc de faire en sorte que cette règle soit assez laxiste. C'est pourquoi nous décidons d'utiliser la directive « `2 of them` » et non « `all of them` » afin que cette règle vérifie les fichiers ne disposant que deux de ces cinq motifs.

La création d'une règle YARA n'est pas une fin en soi. En effet, ces règles doivent être capitalisées afin qu'elles soient confrontés aux nouveaux fichiers soumis à détection. De plus, il est généralement conseillé de confronter ces règles à des bases de données de fichiers existantes. Cette dernière fonctionnalité est souvent appelée « *RetroHunt* »

2.4 Nouveau variant

Cette nouvelle règle YARA nous a permis de récupérer un nouveau fichier sur VirusTotal nommé `msedgeupdate.dll`,[5] téléversé en septembre 2023. En novembre 2023, lorsque nous avons réalisé cette investigation, ce fichier n'était détecté par aucun antivirus (la dernière analyse effectuée sur VirusTotal datant du 15 novembre 2023).

L'objectif de ce chapitre est donc d'analyser ce fichier pour s'avoir s'il s'agit bien d'un nouveau variant ou pas et de s'assurer que notre règle est bien pertinente.

Analyse de `msedgeupdate.dll`. Tout d'abord, nous pouvons vérifier que ce fichier vérifie quatre des motifs créés :

- `$decryption_function`
- `$patch_bytes`
- `$new_pattern_1`
- `$new_pattern_2`

Le seul motif non présent dans cet implant est celui relatif au mécanisme de dérivation de clé (`$derivation_key = {6B 04 00 00 F7 ?? 81 c2 a8 01 00 00}`). Notons de plus que le motif identifié dans la règle YARA d'ESET n'est pas présent.

Le tableau 2 est une mise à jour du tableau 1 prenant en compte ce nouveau fichier.

Fichier	Motif ESET	`$new_pattern_1`	`$new_pattern_2`
Fichier initial	48	0	0
Variant A	69	136	136
Variant B	1	64	128
Nouveau fichier	0	280	280

Tableau 2. Nombre d'occurences des motifs dans chaque fichier

Notre plugin de désobfuscation nous permet de facilement supprimer les mécanismes anti-analyse et de retrouver très rapidement une fonction ressemblant à ce que nous avons observé précédemment. Celle-ci est présentée dans la 17.

Contrairement aux précédents variants, nous pouvons voir que le nom du fichier à charger n'est pas chiffré mais est issu du nom du fichier courant.

[5] `https://www.virustotal.com/gui/file/c0a29416705997d796b94cdce648348d`

```
int global_func()
{
  // Get module filename and replace the extension by ".dat"
  memset(Filename, 0, sizeof(Filename));
  GetModuleFileNameW(hModule, Filename, 0x104u);
  PathRemoveExtensionW(Filename);
  PathAddExtensionW(Filename, L".dat");
  // check if <filename>.dat exists
  if ( !PathFileExistsW(Filename) )
    return 0;
  dwSize = 0;
  // Open & Read the file
  v0 = readfile(Filename, &dwSize);
  if ( !v0 )
    return 0;
  // Decrypt the file
  decrypt_file(v0, dwSize);
  // Alloc & copy the decrypted file
  v3 = VirtualAlloc(0, dwSize, 0x1000u, 0x40u);
  memcpy(v3, v0, dwSize);
  free(v0);
  // Patch the current process to call the new file
  patch_file(v3);
  return 0;
}
```

Fig. 17. Fonction principale du nouveau fichier

L'extension `dll` est remplacée par `dat`. Ainsi, le nom du fichier utilisé pour l'étape d'après est `msedgeupdate.dat`. Notons que même si le nom de fichier n'est pas chiffré, la fonction de déchiffrement est présente dans la DLL. C'est pourquoi les octets de la variable `$decryption_function` sont bien présents.

La fonction de déchiffrement du contenu du fichier est un peu différente. Comme le montre la figure 18, contrairement aux variants précédents, le mécanisme de dérivation de clé n'est plus présent. La clé (sur un octet) est ainsi codée en dur et vaut `0x7b`. Il est intéressant de noter cette valeur correspond au résultat de l'algorithme de dérivation de clé utilisé précédemment. Une hypothèse ici peut être que le compilateur a optimisé le code en remplaçant l'algorithme de dérivation de clé par sa valeur finale.

```
int __fastcall decrypt_file(_BYTE *encrypted_data, int size )
{
  for ( *(&v5 - 1) = v3; size; --size )
  {
    *encrypted_data ^= 0x7Bu;
    *encrypted_data++ += 0x7B;
  }
  return v6;
}
```

Fig. 18. Fonction de déchiffrement du nouveau variant

La fonction relative au patch partage aussi des caractéristiques identiques avec les précédents fichiers, notamment la même séquence d'octets (figure 19).

```
int __stdcall patch_file(int a1)
{
  *&v7[-4] = v1;
  lpAddress = ::lpAddress;
  result = *&v7[4430];
  if ( !::lpAddress )
    return result;
  // Configure the patch
  patch[0] = 0x68;
  *&patch[4] = 0x9090C312;
  *&patch[8] = 0x909090;
  *&patch[1] = a1;
  // set memory protection to PAGE_EXECUTE_READWRITE
  VirtualProtect(::lpAddress, 0xBu, PAGE_EXECUTE_READWRITE, &flOldProtect);
  // apply patch
  *v4 = *&patch[4];
  *v5 = *&patch[8];
  *lpAddress = *patch;
  v6 = patch[10];
  *(lpAddress + 1) = *v4;
  *(lpAddress + 4) = *v5;
  *(lpAddress + 10) = v6;
  // restore initial memory protection
  VirtualProtect(lpAddress, 0xBu, flOldProtect, &flOldProtect);
  return *&patch[8];
}
```

Fig. 19. Fonction appliquant le patch au nouveau variant

À ce stade, nous pouvons avoir une confiance élevée sur le fait qu'il s'agit bien d'un nouveau variant associé à FlowCloud. Mais essayons d'aller plus loin.

Analyse de `msedgeupdate.dat`. Nous avons vu que le fichier `msedgeupdate.dll` a pour objet le chargement et le déchiffrement d'un fichier nommé `msedgeupdate.dat`. Par chance, nous pouvons retrouver sur VirusTotal un fichier nommé `msedgeupdate.dat` téléversé en même temps que ce nouveau variant.[6]

Une première ouverture de ce fichier dans un éditeur de texte montre que ce fichier semble être un fichier de données inintelligible. Cela est cohérent avec notre analyse puisque l'étape précédente commençait par déchiffrer ce fichier. Lorsque nous déchiffrons manuellement ce fichier nous obtenons bien un shellcode qui s'autodéchiffre afin d'obtenir une nouvelle DLL protégée par VMProtect.

[6] `https://www.virustotal.com/gui/file/6d5bcb74a284d119dd32f7b09d37369f`

Un élément intéressant ici, est que la fonction de déchiffrement dispose du même mécanisme de dérivation de clé que précédemment. Ainsi, dans ce nouveau fichier déchiffré nous retrouvons le seul motif de notre règle YARA qui n'était pas présent dans le fichier `msedgeupdate.dll`. La figure 20 présente la fonction de déchiffrement avec l'algorithme de dérivation de la clé précédemment utilisé.

```
int __stdcall decrypt(byte *encrypted_payload, int size, int seed)
{
  result = seed / 0x46Bu;
  v4 = seed % 0x46Bu - 0x58;
  for ( i = 0; i < size; ++i )
  {
    encrypted_payload[i] ^= v4;
    encrypted_payload[i] += v4;
  }
  return result;
}
```

Fig. 20. Fonction de déchiffrement présente dans le fichier `msedgeupdate.dat` déchiffré

La dernière étape est plus compliquée à analyser du fait des protections assurées par VMProtect. Néanmoins, nous avons une confiance très élevée sur le fait que ce fichier est une nouvelle variante de la chaîne d'infection de FlowCloud d'autant plus que :

— l'utilisation de VMProtect par FlowCloud est déjà documentée [3].

— le serveur de commande & contrôle (C2) avec lequel communiquait l'implant disposait de similarités avec d'anciens C2 connus de FlowCloud.

3 Conclusion

En conclusion, cet article avait pour but de présenter la CTI sous l'angle de la création de règles YARA via la rétro-ingénierie de logiciel malveillant. Bien que la création de règles YARA n'est pas l'unique but du travail d'un analyste CTI, il n'en reste pas moins un élément de base dans le suivi des MOA permettant le suivi de leurs codes et de leur évolution dans le temps. L'exemple pris avait pour but de proposer une approche pas à pas de l'étude d'un code et de la création d'une règle YARA. Enfin, nous espérons avoir intéressé le lecteur via l'analyse d'exemples réels d'une famille de codes malveillants.

Références

1. ESET Alexandre Côté Cyr, Matthieu Faou. A lookback under the TA410 umbrella : Its cyberespionage TTPs and activity. 2022. https://www.welivesecurity.com/2022/04/27/lookback-ta410-umbrella-cyberespionage-ttps-activity/.
2. Simon Msika Charles Hourtoule. Comment anticiper la menace : l'exemple de Mustang Panda. *SSTIC*, 2023.
3. MACNICA Hiroshi Takeuchi. USB flows in the Great River : classic tradecraft is still alive. 2023. https://www.virusbulletin.com/conference/vb2023/abstracts/usb-flows-great-river-classic-tradecraft-still-alive/.
4. Michael Raggi and the ProofPoint Threat Insight Team. LookBack Forges Ahead : Continued Targeting of the United States' Utilities Sector Reveals Additional Adversary TTPs. 2019. https://www.proofpoint.com/us/threat-insight/post/lookback-forges-ahead-continued-targeting-united-states-utilities-sector-reveals.