Démarche d'analyse collaborative de codes malveillants

Stefan Le Berre, Adrien Chevalier, et Tristan Pourcelot
stefan.le-berre@sogeti.com
adrien.chevalier@ssi.gouv.fr
tristan.pourcelot@ssi.gouv.fr

Agence Nationale de la Sécurité des Systèmes d'Informations (ANSSI)
Sogeti ESEC lab

Résumé. L'analyse de codes malveillants dans un cadre opérationnel pose de nombreuses problématiques. Il est ainsi nécessaire de combiner les différents problèmes propres à la rétro-ingénierie, tels que la comparaison de binaires, la possibilité d'automatiser au maximum les tâches, d'identifier les points d'intérêts au sein du code, mais également des contraintes plus génériques telles que le partage d'informations, la communication et le travail efficace en équipe.
Nous proposons une approche opérationnelle de ces besoins, en répondant séparément à chaque partie du problème.
Par la suite, nous détaillons une méthode basée sur le graphe de flot de contrôle pour effectuer des comparaisons de code, aussi bien sur un binaire complet que sur un sous-ensemble de fonctions.
Nous présentons également une plateforme répondant à ces problématiques en permettant de stocker et d'organiser un corpus d'échantillons avec les métadonnées associées. Enfin, cette plateforme facilite la rétro-ingénierie collaborative en partageant les informations produites par l'analyse automatique et par les autres intervenants de l'analyse.

1 Introduction

Au sein de l'ANSSI, le bureau failles et signatures (BFS) du centre opérationnel de sécurité des systèmes d'informations (COSSI) analyse quotidiennement de nombreux codes malveillants. Ceux-ci sont issus de la veille en source ouverte assurée par le bureau ou transmis par des partenaires et sont liés à des attaques informatiques d'importance.

La multiplication de ces attaques et le volume important de codes à analyser a nécessité de revoir les pratiques existantes au sein du bureau afin de gagner en efficacité et d'éviter une duplication des efforts.

De façon récurrente, le bureau est amené à devoir analyser des centaines de codes en quelques jours, en tenant compte des contraintes en termes de ressources humaines, nécessairement limitées, et en termes de temps

consacré aux analyses. Le degré d'exhaustivité de ces dernières est donc directement influencé par les contraintes évoquées.

Lors de l'étude d'une famille de codes comportant de nombreux échantillons, le travail collaboratif est soumis à différents enjeux : être capable de capitaliser les résultats généraux, caractérisant la famille en elle-même, ainsi que les résultats spécifiques à un échantillon.

La démarche vise à limiter les travaux redondants et faciliter le partage au sein d'une équipe de résultats déjà obtenus.

Cet article se découpe en deux grandes parties : la première couvrira les différentes problématiques identifiées lors de l'analyse de codes malveillants, qui sont la capitalisation d'informations et le travail collaboratif efficace. La seconde présentera *Polichombr*, un outil développé pour répondre à ces problématiques.

2 Problématiques liées à l'analyse de codes malveillants

Dans le cadre de l'étude des menaces informatiques, il est nécessaire d'effectuer un suivi des outils et codes malveillants utilisés par des groupes d'attaquants. Ceux-ci sont susceptibles, comme tout programme informatique, d'évoluer au fur et à mesure de leur développement et de leur utilisation. Le volume constitué par l'ensemble de ces échantillons est donc relativement considérable.

L'analyse d'un tel corpus de codes malveillants suit un processus itératif. Les informations obtenues par ce processus regroupent des éléments techniques (tels que des indicateurs de compromission), mais également des éléments d'ordre plus général, tels que les capacités d'un groupe d'attaquant (méthodes utilisées, variété d'échantillons...) ou encore l'état d'avancement de l'analyse.

Ce cycle d'analyse est composé des étapes suivantes :

- stockage des échantillons ;
- extraction des caractéristiques de ces échantillons ;
- analyse automatique de chaque échantillon ;
- catégorisation des échantillons ;
- si besoin, analyse manuelle (collaborative dans le cadre de binaires de taille conséquente) ;
- propagation des résultats obtenus ;
- extraction et partage des résultats obtenus.

2.1 Stockage des échantillons

Lors du traitement de grands volumes d'échantillons, l'analyste est très rapidement confronté au problème de la capitalisation de ces échantillons et des informations qui leur sont liées.

En particulier, dans le cas d'échantillons de codes malveillants, de nombreuses métadonnées peuvent être extraites du fichier et nécessitent d'être stockées. Les métadonnées usuellement associées à un échantillon sont ses condensats cryptographiques, sa taille, et les métadonnées dérivées du format exécutable comme l'adresse de point d'entrée, le nombre de sections et les noms qui leur sont associés, etc.

Ces différentes caractéristiques permettent à un analyste de rechercher rapidement des échantillons en utilisant leur condensat ou des métadonnées associées, mais également d'effectuer des recoupements entre ces échantillons.

2.2 Automatisation des analyses

La rétro-ingénierie d'un code malveillant passe régulièrement par des étapes répétitives, telles que l'identification des fonctions gérant le réseau ou le chiffrement.

Ces étapes sont nécessaires pour obtenir une bonne compréhension du programme, mais néanmoins chronophages et sujettes à des oublis. L'automatisation de celles-ci est donc indispensable dans un contexte opérationnel et permet d'obtenir :

- un gain de temps initial ;
- l'assurance d'oublier le moins d'étapes possibles dès le début de l'analyse ;
- faciliter une éventuelle analyse manuelle ultérieure en identifiant les points d'intérêts au sein du code.

Dans cette optique d'identification de points chauds au sein d'un échantillon, plusieurs approches complémentaires sont possibles :

- une approche descendante, en parcourant chaque fonction depuis les points d'entrée du programme ;
- une approche montante, qui marque les fonctions d'intérêt en se basant sur les API utilisées au sein du programme (par exemple `*CreateFile`, `WinHTTPCrackUrl`, ...) ;
- la reconnaissance de motifs, qui permet d'identifier les fonctions (par leur prologue par exemple), ainsi que des éléments d'intérêts comme les clés de registre, les données `HTTP`, les fichiers embarqués ou les chaînes de caractères.

2.3 Catégorisation de codes

Le but principal de cette étape est de trouver les liens entre les différents échantillons, afin de réduire le nombre d'analyses complètes à réaliser. Cette problématique de la classification de codes malveillants est déjà connue du monde académique. Les méthodes existantes se basent sur des isomorphismes de graphes ([3,5,7]), sur des techniques de *clustering* ([13]), ou encore sur des comparaisons de graphes d'appels systèmes ([14]).

Une première approche est de regrouper les échantillons en familles, à l'aide d'une structure arborescente où chaque noeud de l'arbre représente une famille.

Deux principaux cas sont rencontrés durant les analyses :

- comment classifier en familles un ensemble de n échantillons ;
- comment rattacher un échantillon seul à une famille.

Différentes heuristiques et méthodes peuvent être appliquées pour obtenir cette classification, en particulier l'application de signatures (*Yara* par exemple) issues d'autres analyses, la comparaison de graphes de flot de contrôle, ou encore la comparaison de métadonnées sur l'échantillon.

Un exemple d'heuristique qui peut être employée pour regrouper rapidement des échantillons est la classification basée sur les condensats de table d'imports (*Imphash* de *Mandiant* [12]).

2.4 Capitalisation des analyses effectuées

Une des premières étapes lors de l'étude d'un code est de récupérer les informations déjà disponibles sur celui-ci. Ces informations peuvent provenir de sources extérieures (éléments de contexte dans un rapport externe par exemple), mais aussi de précédentes analyses sur ce code. Cette étape vise à limiter au maximum le temps perdu à réassembler des informations déjà disponibles. Deux catégories d'informations peuvent donc être distinguées, celles concernant un seul échantillon de code, et celles, plus globales, associées à une famille ou sous-famille.

Eléments associés à un échantillon de code particulier Dans le cadre des éléments de connaissance associés à un échantillon en particulier, le plus important est de ne pas répéter inutilement les opérations d'analyse. Il est donc essentiel de conserver la connaissance apportée par les différentes passes d'analyse précédentes.

Eléments associés à une famille de codes La connaissance d'un échantillon ne pouvant se résumer aux seuls éléments techniques associés, il est vital de conserver les informations de contexte qui s'y rapportent.

Ces informations peuvent servir à relier des échantillons qui possèdent des caractéristiques communes, mais également à apporter un éclairage plus général, que ce soit sous la forme de rapports sur la famille de codes, ou sous la forme de *scripts* utilitaires.

2.5 Travail collaboratif en temps réel

Lors de l'analyse d'un binaire de taille conséquente ou d'une famille de codes complexes, il est fréquent que plusieurs analystes travaillent en parallèle sur le même binaire. Afin d'éviter les actions redondantes et donc une perte de temps, il est souhaitable que les analystes puissent disposer en temps réel des résultats produits par chacun d'entre eux.

Ces résultats sont généralement obtenus à l'aide de la rétro-ingénierie. Les éléments les plus importants composant ces informations sont, à notre sens, les suivants :

- les noms de fonctions qui donnent un sens au code désassemblé et aux différents appels ;
- des commentaires permettant de déduire le rôle d'une suite d'instructions ou d'une adresse spécifique ;
- des structures, permettant de comprendre le fonctionnement intrinsèque d'une fonction ou d'un objet ;
- le typage des fonctions ;
- les définitions de type de données (chaîne de caractères, données, code).

2.6 Propagation des résultats

Afin de réduire le nombre d'analyses par groupe d'échantillons, un analyste doit être en mesure de propager ses résultats sur un sous-ensemble de binaires.

Un cas d'usage fréquent est le renommage d'une fonction de chiffrement au sein d'un binaire. Si cette fonction est utilisée dans d'autres échantillons, ce nom doit être propagé, ce qui évitera à l'analyste de réétudier cette fonction et facilitera sa recherche par la suite.

Cette fonctionnalité est également utile dans le cadre de la reconnaissance de fonctions de bibliothèques, permettant de les déduire rapidement lors de l'analyse des codes.

De même, si une signature telle qu'une règle *Yara* est dérivée d'une souche, il est important de l'appliquer sur l'ensemble des échantillons stockés pour établir des rapprochements. L'application de ces signatures permettra de recatégoriser, le cas échéant, les échantillons, ou de mettre en évidence une signature trop générique.

2.7 Export et partage des résultats

Les résultats des analyses doivent fréquemment être transmis à différents interlocuteurs. En fonction du profil de ces interlocuteurs (analyste au sein de ou extérieur à l'équipe, RSSI, ...), différents critères de sélection doivent être appliqués pour fournir des résultats cohérents le plus facilement possible.

Les besoins identifiés en fonction des profils d'interlocuteurs sont les suivants :

- un analyste de l'équipe est intéressé par toutes les informations disponibles, aussi bien les résultats techniques que les éléments de contexte ;
- un analyste extérieur sera principalement intéressé par les éléments techniques concernant les binaires, des contraintes supplémentaires s'appliquant, comme la confidentialité et la sensibilité des binaires ;
- un RSSI sera intéressé par des indicateurs de compromission liés à des éléments de contexte limités, notamment en termes de confidentialité et de sensibilité.

2.8 Moyens existants et limites

Les outils et méthodes actuellement utilisés dans le monde de la sécurité montrent assez rapidement leurs limites dans un mode opérationnel. Ces limites peuvent être décomposées sous deux formes, d'un côté les points durs au niveau organisationnel, rencontrés lorsque plusieurs analystes travaillent ensemble sur un sujet, et d'un autre côté les limites techniques, rencontrées lors de l'utilisation d'outils censés répondre aux besoins de la rétro-ingénierie.

Limites organisationnelles Une personne étant en charge d'analyser un code ou une famille de codes aura tendance à conserver ses échantillons, ses résultats et ses scripts d'analyse sur son poste personnel, avant de rédiger un rapport contenant :

- ses résultats ;
- une description des différentes techniques utilisées par l'échantillon ;

- les différents outils réalisés pour en extraire des éléments (déchiffrement de configuration par exemple) ;
- les éléments de détections associés ;

Cette méthode permet de transmettre une partie de la connaissance acquise sur une famille de codes, toutefois la transition vers un nouvel intervenant qui reprendrait les analyses en cours peut se révéler difficile, en particulier lors de l'étape de récupération des échantillons, ou lors d'une réanalyse d'un échantillon en particulier.

De même, le partage d'échantillons via des dossiers / sous dossiers atteint rapidement ses limites puisque tous ces échantillons perdent leur contexte.

Limites des outils existants Il ne s'agit pas ici d'effectuer une liste des outils existants, qui apportent chacun leur pierre à l'édifice d'une meilleure rétro-ingénierie. Il existe ainsi de nombreuses applications de stockage d'échantillons ([2, 8]), d'autres applications d'analyse automatique ([15]), des briques enfichables permettant de combiner des outils et d'effectuer des partages d'informations ([1, 4, 6, 11]), etc.

Toutefois la plupart de ces produits ne fonctionnent que dans certains cas d'usage ne correspondant pas aux besoins identifiés, que ce soit en termes de capitalisation ou de collaboration.

3 Polichombr

Les deux grandes problématiques liées à l'analyse de codes malveillants qui ont été identifiées sont donc la capitalisation des résultats issus de ces analyses et le travail efficace en équipe.

Les outils existants permettent de répondre séparément à ces différentes problématiques, toutefois aucun ne permet de les combiner ensemble pour obtenir des résultats opérationnels.

Un outil spécifique permettant de répondre à ces critères a donc été développé : *Polichombr*.

Polichombr est un applicatif serveur auquel peuvent venir se greffer plusieurs parties clientes, dont le script *IDAPython skelenox*.

De même, un nouveau type de condensat (nommé *Machoc*), basé sur le graphe de flot de contrôle (*CFG*), a été développé et intégré au sein de cet outil pour réaliser des comparaisons de binaires.

3.1 Architecture

Briques *Polichombr* se découpe en plusieurs briques :

- une interface utilisateur sous forme d'interface *Web* ;
- une base de données qui stocke les métadonnées associées aux samples, ainsi que les résultats d'analyse ;
- un moteur de désassemblage, d'analyse et d'émulation (scripts Ruby utilisant la bibliothèque de désassemblage *Metasm* [9]) ;
- un script *IDAPython*, qui effectue le lien entre la base de données et l'outil *IDA Pro* de l'analyste.

Interface Web L'interface *Web* expose aux utilisateurs une vue macroscopique du travail des analystes. Par exemple, elle permet de disposer d'une vue par familles et sous-familles de codes, d'une vue sur les analyses en cours, de notes des analystes, de marqueurs résultant des analyses, etc. Elle permet également d'exporter ces marqueurs dans divers formats.

Le module de désassemblage Ruby/Metasm Ce module effectue plusieurs tâches : tout d'abord, lors de la soumission d'un nouvel échantillon, il extrait les métadonnées et les condensats *Machoc* du binaire. Ensuite, lors de la consultation de l'échantillon dans l'interface *Web*, il est possible de faire appel à cette brique pour émuler certaines parties du code (seuls les exécutables au format `PE x86` et `PE x64` sont supportés) ou pour afficher des fonctions désassemblées sous forme de graphe.

Skelenox Enfin, le script *IDAPython skelenox* effectue l'interface entre l'outil *IDA Pro* et *Polichombr* via une interface `HTTP` de type *web service*. Cette interface permet aux analystes de remonter les éléments qu'ils produisent en local (un renommage de fonction par exemple), mais également de récupérer les éléments déjà en base de données tels que les points d'intérêts identifiés par l'analyse automatique et les informations produites par les autres contributeurs.

3.2 Organisation des échantillons

Les échantillons de codes malveillants sont regroupés en arborescence de familles et sous-familles. En prenant l'exemple de la famille *PlugX*, on peut définir des sous-familles v1, v2 et v3. Ces sous-familles peuvent elles mêmes être subdivisées, en différenciant par exemple la branche `A` au sein de la sous-famille `PlugX.v2`.

Chaque famille (peu importe son niveau) peut posséder un descriptif (à la discrétion de l'analyste), des règles de détection (*Yara*, *Machoc*, *Snort* . . .), des documents de référence, des indicateurs de compromission, un niveau de confidentialité, etc.

Un binaire, quant à lui, est associé à des informations variées, qui vont de ses métadonnées (extraites par le script d'analyse Ruby) aux opérations effectuées par les analystes dans le logiciel *IDA Pro*, en passant par un bloc-note, un aide-mémoire d'opérations à effectuer, des condensats *Machoc*, des règles *Yara* ayant identifié le code, etc.

Enfin, tous ces éléments sont liés aux utilisateurs de l'applicatif, permettant ainsi à ces derniers de disposer de zones de travail, ainsi que de pouvoir déterminer rapidement quel utilisateur peut être compétent sur une famille de codes donnée.

3.3 Algorithme *Machoc*

Afin de comparer les échantillons, une méthode permettant non seulement d'identifier des codes similaires, mais également de comparer des échantillons ayant subi des modifications succinctes a été développée.

Les modifications que peuvent subir un échantillon peuvent être dûes à une recompilation du code, une mise à jour de fonctions, un changement de configuration, etc.

Les condensats cryptographiques usuels tels que MD5, SHA1 ou SHA256 ne sont par nature pas résistants aux changements et ne sont donc clairement pas adaptés à ce problème. Les condensats SSDEEP et SDHash tolèrent quant à eux des modifications partielles des données, mais ne sont pas spécifiquement adaptés aux exécutables.

Enfin, ces mécanismes ne permettent pas non plus de pouvoir identifier des changements entre plusieurs versions.

Or, dans le cas précis de l'analyse de codes malveillants, seules certaines zones de l'échantillon sont utiles et devraient être ciblées par la méthode de comparaison. La définition d'un nouveau type de condensat, résistant à des changements minimes au sein des codes, a donc été nécessaire.

Choix de conception L'algorithme *Machoc* se base sur le graphe de flot de contrôle des fonctions, à savoir la relation entre les blocs élémentaires au sein d'une même fonction. Ce graphe représentant la forme d'une fonction, il est moins susceptible de changer que les instructions composant cette fonction lors d'une modification.

Un condensat *Machoc* est donc représentatif d'une seule fonction. Néanmoins, la concaténation de l'ensemble des condensats de fonctions permet d'obtenir un condensat représentatif de chaque échantillon.

Le comparaison des condensats entre deux binaires permet d'obtenir simplement un pourcentage de similarité.

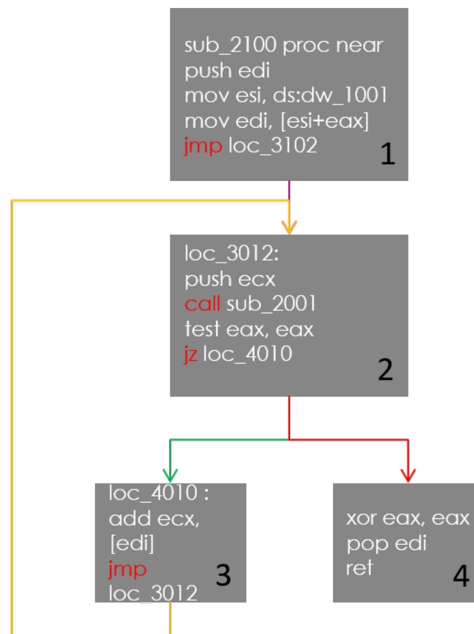

Fig. 1. Exemple de graphe de flot de contrôle

Exemple d'application Afin de décrire le fonctionnement de cet algorithme, nous nous baserons sur le graphe de flot de contrôle d'une fonction représenté dans la figure 1.

Le calcul du condensat *Machoc* suit les étapes suivantes :

1. numéroter les blocs élémentaires, soit en parcourant le flux d'exécution de la fonction soit en suivant le chaînage logique des blocs ;
2. transcrire pour chaque bloc les appels et les sauts par une chaîne de caractère simple ;
3. concaténer l'ensemble des représentations de chaque bloc pour obtenir la représentation d'une fonction ;
4. condenser la représentation de cette fonction en condensat de type *Murmurhash3*.

On représente chaque bloc sous la forme d'un texte suivant une syntaxe comprenant son identifiant (numéro), un indicateur de présence d'appel (identifiant `c`), et les liens vers d'autres blocs (identifiant du bloc cible).

Le bloc 2 de la figure 1 est donc représenté par la chaîne de caractères suivante :

```
2:c,3,4;
```

La représentation complète de la fonction consiste à itérer le processus pour chaque bloc puis concaténer les résultats.

Pour la fonction représentée en 1, la représentation complète serait :

`1:2;2:c,3,4;3:2;4:;`

Cette représentation peut être extrêmement longue pour certaines fonctions volumineuses, c'est pourquoi un condensat *MurmurHash3* est appliqué sur la chaîne résultante. Ce condensant a été sélectionné pour sa rapidité, puisque la qualité cryptographique du condensat n'est pas importante pour ce cas.

Utilisation pour la comparaison de fonctions Chaque condensat de fonction est stocké avec l'adresse de la fonction correspondante. Cette relation est importante pour d'autres fonctionnalités de *Polichombr*, notamment pour l'identification de fonctions identiques entre plusieurs échantillons proches.

Utilisation pour la comparaison de binaires Le condensat *Machoc* complet d'un binaire étant constitué de la concaténation de l'ensemble des condensats de ses fonctions, il est possible d'identifier un pourcentage de similarités entre deux codes en comptant simplement les condensats communs.

La distance utilisée pour comparer deux binaires est celle de *Jaccard*, et le taux de similarité (déterminé de manière empirique) permettant d'établir un lien quasi sûr entre deux binaires est de 80%.

Remarques Plusieurs implémentations sont disponibles :

- dans le moteur de désassemblage Ruby, reposant sur le cadriciel *Metasm* ;
- dans le script *Skelenox*, reposant sur *IDAPython*.

En pratique, certaines différences (de l'ordre de 10%) peuvent subvenir entre les deux formes de calcul, principalement liées à la reconnaissance des fonctions, effectuée différemment par les outils *Metasm* et *IDA Pro*. Les deux formes sont donc conservées en base.

3.4 Analyse des échantillons

Analyse automatisée Lors de la soumission d'un exécutable à *Polichombr* plusieurs tâches sont effectuées. Le but de ces tâches est de réaliser des actions chronophages pour un analyste et d'éviter d'oublier des actions importantes. Elles permettent également une extraction de métadonnées susceptibles d'être recherchées ou comparées.

Les tâches suivantes sont effectuées pour chaque binaire au format `PE` (*Portable Executable*) :

- calcul des condensats cryptographiques MD5, SHA1 et SHA256 du fichier, ainsi que ceux de sa table d'importations (*Import Address Table*) ;
- extraction des métadonnées issues du format `PE` (`IMAGE_NT_HEADERS` par exemple) ;
- calcul d'un condensat MD5 des sections et des ressources de l'exécutable ;
- désassemblage du binaire et recherche de fonctions non découvertes ;
- création d'une arborescence des appels entre les fonctions ;
- reconstruction des arguments d'appels d'environ 60 fonctions de la bibliothèque *Win32* (API *Win32*) "sensibles" ; Ces API concernent la création de processus (`WinExec`, `CreateProcess*`, ...), mais aussi la gestion du registre (`Reg*`), ou encore la manipulation de fichiers (`DeleteFile`, `CreateFile`, ...).
- production d'un graphe d'appel depuis le point d'entrée (ou depuis les fonctions exportées) vers ces fonctions "sensibles" ;
- renommage des fonctions par thématique identifiée (par exemple : `Net_sub_123` ou `Reg_File_sub_123`) ;
- identification d'algorithmes cryptographiques connus ;
- identification de boucles cryptographiques ;
- émulation d'une pile pour identifier des écritures de chaînes de caractères à l'intérieur de celle ci ;
- énumération de toutes les chaînes de caractères présentes avec les instructions liées et catégorisation des chaînes par thématique ;
- calcul du condensat *Machoc* de chaque fonction ;
- application des règles *Yara* ;
- comparaison des condensats *Machoc* spécifiques à ceux du binaire ;
- catégorisation automatique (voir ci-après).

Certaines des informations liées à une adresse spécifique (boucle cryptographique, chaîne reconstruite, etc.) sont notamment enregistrées comme commentaires, qui seront visibles par l'analyste lors de l'utilisation d'*IDA Pro* avec le script Skelenox.

Enfin, les informations plus génériques (reconstruction d'appels, identification de fonctions sensibles...) font l'objet d'un rapport consultable sur l'interface *Web*, permettant de donner des pistes à l'utilisateur afin de gagner un certain temps lors de la première étude d'un code.

Catégorisation automatique Lors de la soumission d'un nouvel échantillon, différentes actions sont effectuées pour tenter une catégorisation automatique de celui-ci.

En particulier, si l'échantillon correspond à une signature *Yara* déjà vérifiée et associée à une famille, le binaire est rattaché directement à cette famille. De même, dans le cas d'une comparaison *Machoc* signante (algorithme de compression personnalisé par exemple), l'association automatique du binaire à la famille correspondante est effectuée.

Il est à noter que ces types de signatures "sûres" sont également rétroactifs : lors de l'association d'une telle signature à une famille, tous les codes identifiables par cette signature hériteront de la famille concernée. Dans le cas de résultats plus incertains, comme un condensat *Machoc* global proche d'un second binaire ou d'un même condensat de section ou de table d'importation, un bref rapport d'analyse est affiché à l'utilisateur lui résumant ainsi les propositions. A lui de choisir par la suite s'il souhaite ou non faire hériter le binaire de ces familles.

Pour traiter de gros volumes de codes, il est également possible de soumettre des échantillons via l'interface de type *web service* et d'obtenir un résultat de la classification automatique. Dans ce cas d'utilisation, deux cas de figures sont possibles :

- Dans le cas où le code soumis est détecté par une règle "sûre" (*Yara* ou condensat *Machoc* spécifique), le binaire est conservé et hérite des familles identifiées ;
- le code soumis n'est pas détecté par une règle "sûre" : le binaire n'est pas conservé pour éviter de polluer la base de données, sauf sur demande explicite de l'utilisateur, à qui il appartiendra alors d'affecter le binaire à une famille, ou de soumettre une règle de détection "sûre" a posteriori.

Propagation d'informations entre échantillons Le condensat *Machoc* permet d'obtenir des similarités entre les codes. Dans la mesure où un travail de nommage de fonctions a été effectué sur un code spécifique et où nous disposons d'un triplé adresse / nom / condensat *Machoc*, il est possible de propager ce travail sur un second code très proche, sur demande de l'utilisateur.

Le but de cette propagation est double : premièrement, l'utilisateur cherchant à identifier une donnée en particulier dans un code pourra la retrouver aisément dans un second code proche (par exemple, une constante d'une routine de chiffrement). Deuxièmement, un analyste pourra également identifier rapidement les fonctions ayant évolué entre deux versions d'un même code.

Pour ce faire, *Polichombr* va comparer les condensats *Machoc* des deux échantillons, et traiter les trois cas suivants :

- si le condensat d’une fonction est présent de manière unique dans les deux binaires, la fonction cible est renommée ;
- si le condensat d’une fonction est présent de manière multiple dans un des deux binaires, la recherche est étendue aux deux fonctions précédentes et suivantes de la fonction cible, ce qui fait donc un ensemble de cinq condensats à comparer. Si cet ensemble est unique dans les deux binaires, la cible est renommée ;
- si le condensat d’une fonction n’est pas dans le binaire “cible”, alors la recherche est étendue aux trois condensats au dessus et au dessous de la fonction. Si ces condensats sont trouvés et que leur suite est unique dans chaque binaire (suite ne comprenant donc pas le condensat manquant), les fonctions ainsi encadrées sont considérées comme étant deux versions d’une même fonction et le renommage est également effectué.

Désassembleur Le module de désassemblage permet de visualiser dans l’interface *Web* le flot de contrôle d’une fonction du binaire. Il est également possible de renommer les fonctions, ajouter des commentaires et se déplacer dans les fonctions en cliquant sur les appels. De même, les résultats fournis par les analystes tels que les noms de fonctions et les commentaires sont pris en compte et affichés.

Emulateur Un émulateur pour le format `PE` en architecture x86/x64 a été développé pour permettre de simuler des parties du binaire, tout en minimisant les risques. Cette émulation est basée sur la symbolique fournie par *Metasm* pour chaque instruction. En utilisant la simulation des registres et de l’espace mémoire (de manière similaire à celle utilisée pour le mécanisme de *backtracking* intégré dans *Metasm*), il est possible de simuler chaque instruction, et donc d’obtenir l’émulation complète d’une fonction.

De même, afin de maximiser l’identification des fonctionnalités d’un code malveillant, plusieurs fonctions standards sont simulées (`CreateFile` ou `WinHttp*` par exemple). Il est aussi possible d’injecter du code `C` ou assembleur dans l’espace mémoire de l’échantillon.

Il est ainsi possible d’émuler une fonction du programme, pour, par exemple, réaliser un déchiffrement de zone protégée ou de chaînes de caractères.

3.5 Travail collaboratif

Capitalisation depuis *IDA Pro* : *skelenox* Lors du démarrage de *skelenox*, une première passe d'analyse du binaire est effectuée. Cette passe a pour but de récupérer les points d'intérêt produits par l'analyse automatique et de se synchroniser avec les informations contenues dans *Polichombr*.

En effet, dans un but d'efficacité, les actions d'analyse les plus courantes sont capitalisées via un script *IDAPython* (*Skelenox*) vers le *web service* de *Polichombr*.

Ces différentes actions permettent de partager en temps réel les actions effectuées, et ainsi d'éviter de perdre un temps précieux.

Les actions capitalisées sont les suivantes :

- renommage de fonction ;
- typage d'une fonction ;
- définition d'une structure ;
- écriture d'un commentaire ;
- affectation d'un type de donnée à une adresse (structure, donnée, fonction).

Le script requête régulièrement le serveur, afin de récupérer les dernières actions effectuées et ainsi reconstituer l'enchaînement complet. De même, il est possible de visualiser l'enchaînement de ces actions dans l'interface *Web*.

Ainsi, un analyste peut profiter en temps réel des actions effectuées par ses collègues.

Support à l'analyse L'interface *Web* permet de capitaliser des informations de plus haut niveau, aussi bien sur les binaires que sur les familles associées.

Ces informations ont besoin d'être capitalisées afin de les utiliser pour des actions autres que l'analyse pure, telles que de la recherche de compromission, documentation, la compréhension de modes opératoire, etc.

Ces informations étant assez génériques, elles sont capitalisées au sein de deux modules :

- un résumé par échantillon et par famille de code, sous forme de bloc-notes à discrétion de l'analyste ;
- un système d'aide-mémoire permettant de vérifier que la procédure d'analyse a été bien suivie. Cet aide-mémoire, sous forme de *checklist*,

est variable et peut être configurée. Elle inclut aussi bien des éléments techniques devant être renseignés (mécanismes de persistance, algorithmes de chiffrement utilisés, etc.), que des éléments de procédure d'analyse à effectuer (vérification des comparaisons, etc.).

Aide au pilotage Le système d'aide-mémoire permet de garantir une certaine cohérence des analyses et des résultats malgré les différences entre les habitudes propres à chaque intervenant.

Un autre élément important de cette capitalisation est la capacité de fournir un rapport générique permettant de mettre en exergue un état des familles en cours de traitement, ainsi qu'une vue sur l'avancée des travaux en cours.

3.6 Consultation et export de données

Export de données Afin de répondre à la problématique du partage efficace des résultats d'analyse, les données contenues dans *Polichombr* peuvent être exportées dans plusieurs formats, en fonction des destinataires.

De même, les données à exporter sont sélectionnées en fonction des familles de codes concernées, ainsi que d'un niveau de sensibilité maximum.

Export OpenIOC L'export *OpenIOC* permet d'exporter tout ou partie des indicateurs de compromission concernant une famille de codes.

Il est possible d'exporter les condensats des différents échantillons, ainsi que les règles de détection (*Snort*) associées.

Export Yara L'export de règles *Yara* consiste à exporter des corpus de règles (*rulesets*), en leur joignant des métadonnées spécifiques contenant les informations de l'applicatif.

Le but est ici de pouvoir déployer rapidement des règles *Yara* hors de *Polichombr*, que ce soit pour utilisation immédiate ou pour communication à une tierce partie.

Export Machoc L'export *Machoc* vise à exporter les associations nom de fonction/condensat *Machoc* par échantillon pour transmission à une entité tierce.

Plus simple qu'une transmission de fichier d'analyse *IDA Pro* (idb), cela permet également de limiter la diffusion de certaines informations pouvant être sensibles. Les données sont exportées sous la forme de fichiers au format CSV.

Production de synthèses Enfin, l'export de synthèses vise quant à lui à exporter l'ensemble des informations concernant des codes, dans un format texte brut.

Le but est de pouvoir générer rapidement des descriptifs humainement compréhensibles concernant des familles de codes, et ce afin de pouvoir communiquer le plus d'informations possibles en un minimum d'actions.

Sont donc compris dans ce rapport les notes des analystes, les aide-mémoires, ainsi que les différents types de marqueurs (*OpenIOC*, *Machoc* et *Yara*) en annexe.

Références

1. aaronportnoy. Ida toolbag.
2. Adlice. malware : Malware repository framework.
3. Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Control flow graphs as malware signatures. In *International workshop on the Theory of Computer Viruses*, 2007.
4. CubicaLabs. Idasynergy.
5. Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5 :1–3, 2005.
6. Chris Eagle. collabreate.
7. Halvar Flake. Structural comparison of executable objects. 2004.
8. Claudio Guarnieri. Viper.
9. Yoann Guillot. Metasm.
10. SA Hex-Rays. Ida pro, 2012.
11. CrowdStrike Inc. Crowdstrike crowdre, 2012.
12. Mandiant. Tracking malware with import hashing, 2014.
13. Swathi Pai, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. Clustering for malware classification. *Journal of Computer Virology and Hacking Techniques*, pages 1–13, 2016.
14. Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, pages 45 :1–45 :4, New York, NY, USA, 2010. ACM.
15. Daniel Plohmann. simplifire.idascope - an ida pro extension for easier (malware) reverse engineering.