ICEBOX : analyse de Malwares par introspection de machine virtuelle

Benoit Amiaux[1], Luca Farey[1], Jean-Marie Borello[1]

ThalesGroup Rennes
benoit.amiaux@thalesgroup.com luca.farey@thalesgroup.com
jean-marie.borello@thalesgroup.com

Résumé Cet article présente un projet d'introspection de machine virtuelle extensible et performant, dénommé `IceBox`, offrant non seulement les fonctionnalités de reporting classiques des *sandboxes* actuelles mais aussi un contrôle total du système d'exploitation invité via une modification du projet open-source `VirtualBox` [16]. De part sa furtivité liée au fait qu'aucun agent ne tourne dans l'invité, ce projet se prête particulièrement bien à l'analyse dynamique de malwares.

Keywords: introspection, machine virtuelle, malware, analyse dynamique

1 Introduction

Avec la dématérialisation continue de l'information, la virtualisation poursuit son développement et devient aujourd'hui omniprésente. Si ce contexte offre de nouvelles possibilités aux attaquants, il permet aussi le développement de solutions d'analyse tirant profit de ces évolutions. En particulier, le sujet d'introspection de machine virtuelle, une technique permettant de suivre et d'inspecter directement l'exécution d'une machine virtuelle, et qui permet des analyses furtives et efficaces. Dans cet article, nous introduisons notre solution de VMI [1] dénommée `IceBox` [2] qui offre non seulement les fonctionnalités de reporting et de monitoring classique d'une sandbox d'analyse de malwares, mais aussi un contrôle total et furtif de l'invité.

Sans rentrer dans les détails techniques qui seront expliqués plus loin, nous énumérons ci-après les contributions `IceBox` qui offrent une position privilégiée pour l'analyse de l'exécution d'un malware :

- des **performances élevées** permettant une analyse en "temps réel" et offrant une interactivité forte avec le système analysé ;
- une **furtivité** vis-à-vis du système invité contrairement à d'autres sandbox commerciales ou open-sources ;
- une solution **générique portable** pour architecture Intel, c'est-à-dire, supportant les systèmes d'exploitation "classiques" aussi bien en hôtes qu'en invités (Windows, Linux et MacOS).

1. Virtual Machine Introspection
2. `https://github.com/thalium/icebox`

— une solution **moderne**, qui supporte des systèmes d'exploitation invités récents (Windows 10 build 1809 et Ubuntu 19.10) ;

Le reste de l'article s'articule de la façon suivante : la section 2 présente la problématique d'analyse de malwares et les principales approches associées, en particulier les *frameworks* de VMI les plus répandus. La section 3 introduit ensuite notre solution à travers ses choix de design. La section 4 détaille ensuite une vue générale de son architecture. La section 5 propose un exemple d'analyse de malware menée avec `IceBox` pour en illustrer les capacités. Enfin, la section 6 termine cet article par les limitations actuelles du projet et les évolutions envisagées.

2 État de l'art

L'analyse dynamique de malwares est un sujet connu mais qui demeure toutefois difficile car :

— les malwares sont par design non-coopératifs vis-à-vis d'un analyste, ils emploient à ce titre un ensemble de techniques connues visant à complexifier leur analyse tel que de l'anti-debug, de l'obfuscation et diverses techniques d'auto-modification de code tels que le chiffrement, le polymorphisme ou encore des packers (`Themida` [15], `VMProtect` [17], etc) ;
— l'intégrité de l'environnement d'analyse doit pouvoir être garantie (par un mécanisme de restauration ou de *snapshot* par exemple) car l'exécution de code malveillant est par essence compromettante ;
— l'environnement d'analyse doit aussi être le plus furtif possible afin de ne pas trahir sa présence vis-à-vis du malware qui pourrait réagir en conséquence ;
— le contrôle sur le système d'exploitation invité doit être suffisamment avancé pour permettre une analyse la plus fine possible. Ainsi, il existe sous windows un mécanisme nommé `Kernel Patch Protect` [7] qui protège l'intégrité du système. Par exemple les modifications de certaines parties du noyau sont impossibles sans un accès hyperviseur.

Parmi l'ensemble des solutions existantes pour répondre à ces contraintes, l'introspection de machine virtuelle apparaît comme la plus répandue actuellement. Ce type d'introspection existe sous deux formes :

1. **dans l'invité** mais qui se retrouve au même niveau de privilège que le système et est, de fait, détectable par un malware. C'est par exemple le cas de la *sandbox* `cukoo` [4] ;
2. **hors invité**, dans ce cas, l'introspection est faite au niveau de l'hyperviseur au moyen d'un jeu d'instructions spécialisé du processeur et qui offre un plus haut niveau de furtivité.

Par la suite, nous ne considérons que l'introspection au niveau de l'hyperviseur car jugée plus en adéquation avec notre problématique d'analyse de malwares de part sa furtivité.

Nous présentons ici les principales solutions de VMI pour l'analyse de malwares :

— `PyREBox` [12], qui s'appuie sur `QEMU` [13] et `Volatility` [18] ;
— `Drakvuf` [9], qui s'appuie sur `Xen` [20] et `LibVMI` [10] ;
— `applepie` [1] qui s'appuie sur `HyperV` [6] et `Bochs` [2]. Solution initialement conçue pour du fuzzing mais adaptable pour l'analyse de malwares ;
— `Sandbagility` [8] qui est une sandbox écrite en python s'appuyant sur le projet `Winbagility` [3] qui est un framework de VMI sur `VirtualBox` [16].

Le tableau 1 synthétise les principales caractéristiques des différentes solutions de VMI présentées :

	PyREBox	**Drakvuf**	**applepie**	**Sandbagility**	**IceBox**
Hyperviseurs	QEmu	Xen	HyperV+Bochs	VirtualBox	VirtualBox
Hôtes supportés	Linux+Mac	L+M	Win10	Win10	W+L+M
Guest supportés	W+L	W+L	Win10	Win10	W+L
API	Python	Python	C	Python	C++
Performances	médiocres	élevées	élevées	correctes	élevées

Table 1. tableau comparatif des différentes solutions de VMI dédiées adaptées à l'analyse de malwares.

3 IceBox

Dans cette partie, nous justifions notre choix vis-à-vis de l'ensemble des solutions présentées dans la section précédente.

Le premier choix est celui de repartir du projet `Winbagility` [3]. Ainsi, nous tirons profit de `VirtualBox`, une solution open-source (license BSD), portable (Windows, Linux et MacOS), et relativement facile à déployer. De plus, le patch de VirtualBox fourni par `Winbagility` est relativement court et donc jugé plus simple à comprendre et maintenir pour les versions futures. Enfin, `Winbagility` est la solution de VMI offrant les performances les plus élevées (lecture de l'ordre de 2,5 Go/s, points d'arrêt de 5K à 11K/s)

Le deuxième choix est d'avoir écrit une bibliothèque en code natif (C++) afin de bénéficier pleinement de la vitesse de `Winbagility` lui-même développé en C. `Sandbagility`, écrit en Python, n'a pas été réutilisé pour des raisons de performances. En effet les tests menés sur un outil de suivi des allocations mémoire mettait en évidence une vitesse d'exécution trop faible pour être utilisable. Le système n'était plus interactif et certaines opérations sur le binaire analysé pouvaient prendre plusieurs heures au lieu de quelques secondes. Deux problèmes ont été identifiés :

— Le surcoût lié au langage Python, du code interprété, appelé chaque fois qu'un point d'arrêt est activé ne serait-ce que pour filtrer le processus ciblé ;

— Les choix d'architecture de `Sandbagility` qui favorisent la simplicité d'utilisation au détriment des performances. Sur les points d'arrêt, cette bibliothèque inspecte énormément de propriétés du processus courant, comme son nom, alors qu'il ne suffit parfois que de lire un seul registre pour filtrer le processus qui nous intéresse ;

Pour finir, `IceBox` apporte des fonctionnalités supplémentaires assez fondamentales comme la gestion de la pile des appelants (*callstack*), fonctionnalité que nous détaillons un peu plus loin.

4 Architecture

La figure 1 représente de manière schématisée l'architecture d'`IceBox`.

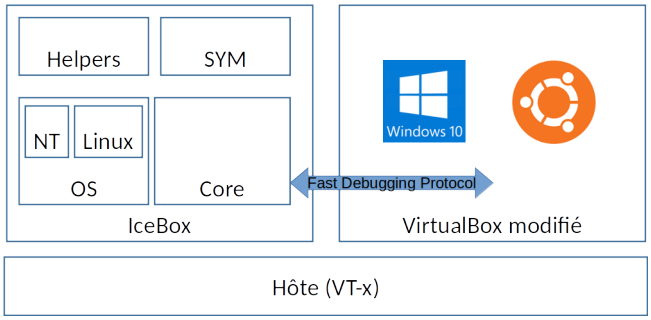

Figure 1. Schéma d'architecture d'`IceBox`.

Il s'agit d'une architecture classique de type client/serveur. La partie serveur représentée à droite sur la figure correspond au projet `Winbagility` [3]. Ce projet modifie l'hyperviseur `VirtualBox` afin de lui rajouter des primitives de débogue furtives à travers un protocole spécialement conçu pour les performances. Les primitives de bases offertes par ce protocole, dénommé *FDP*[3] sont les suivantes :

— lire/écrire de la mémoire de l'invité (physique ou virtuelle) ;
— lire/écrire des registres de l'invité ;
— installer/désinstaller des points d'arrêt ;
— sauvegarder/restaurer l'état de l'invité ;
— suspendre/reprendre l'exécution de l'invité ;
— redémarrer l'invité.

La partie cliente est `IceBox` à proprement parler dont les principaux composants sont :

3. Fast Debugging Protocol, voir [3] pour les détails

Le *core* qui utilise l'API de débogue fournie par `Winbagility`. Ce dernier permet d'accéder à la mémoire, aux registres ainsi qu'à l'état de l'invité (gestion des *breakpoints*). Concrètement, il fournit les primitives "bas niveau" exposées par le protocole FDP citées précédemment.

L'OS qui est une interface de manipulation générique du système d'exploitation invité. Au moment de l'initialisation du *core*, l'implémentation de cette interface est faite en fonction du système d'exploitation détecté (pour le moment, Windows ou Linux). Cette interface permet de manipuler (récupérer, énumérer) les éléments classiques d'un système d'exploitation tels que les processus, les threads, les modules (utilisateurs et noyaux) ainsi que d'être notifié sur certains évènements comme la création/chargement, suppression/déchargement d'un processus, thread ou module.

Le composant SYM qui permet la gestion des symboles. Par exemple, la lecture des fichiers `PDB` [4] qui contiennent les informations de débogue sous Windows afin d'associer un nom de symbole (fonction, chaîne de caractères, variable globale) à une adresse et réciproquement.

Sous Linux, `IceBox` est capable d'extraire ces informations de débogue directement à partir des binaires lorsqu'elles sont présentes. Pour les symboles du noyau lui-même, `IceBox` nécessite la compilation préalable d'un module dédié pour la distribution ciblée. Une fois ce module compilé, un profil local à cette distribution est généré sur l'hôte et réutilisé ensuite pour permettre l'introspection du noyau Linux.

Les *helpers* qui sont des composants spécifiques complémentaires pour chaque système d'exploitation mais qui apporte une aide précieuse pour l'analyse :

- la *callstack* qui permet de récupérer la pile d'appels lorsque l'invité est en arrêt, par exemple lorsqu'un *breakpoint* est atteint ;
- l'introspection dans les objets du noyaux NT comme les devices, les drivers, les `HANDLES`, etc.
- la possibilité de poser un point d'arrêt en début et/ou en fin de fonction et d'accéder aux arguments ou la valeur de retour.

5 Analyse de malwares

Muni d'`IceBox`, nous disposons maintenant d'un environnement restaurable, furtif et suffisamment maîtrisé pour pouvoir mener une analyse dynamique. Reste maintenant à *analyser* un malware, qui, rappelons le, cherche à masquer son activité vis-à-vis d'un analyste.

Nous prendrons ici pour exemple NotPetya, qui est un ransomware datant de 2017 particulièrement virulent car exploitant la vulnérabilité `EternalBlue`

4. Program DataBase

ciblant le protocole SMB (CVE-2017-0144). Ce malware est un cas d'étude intéressant pour plusieurs raisons :

- rechargement du malware en mémoire ;
- création de plusieurs sous-processus ;
- modification du MBR [5] ;
- chiffrement des fichiers (activité forte sur le système de fichier) ;
- propagation via le réseau avec différentes techniques ;
- exploitation de la vulnérabilité CVE-2017-0144 plus connue sous le nom d'`EternalBlue` ;
- backdoor kernel `DoublePulsar` [5] sur la machine cible.

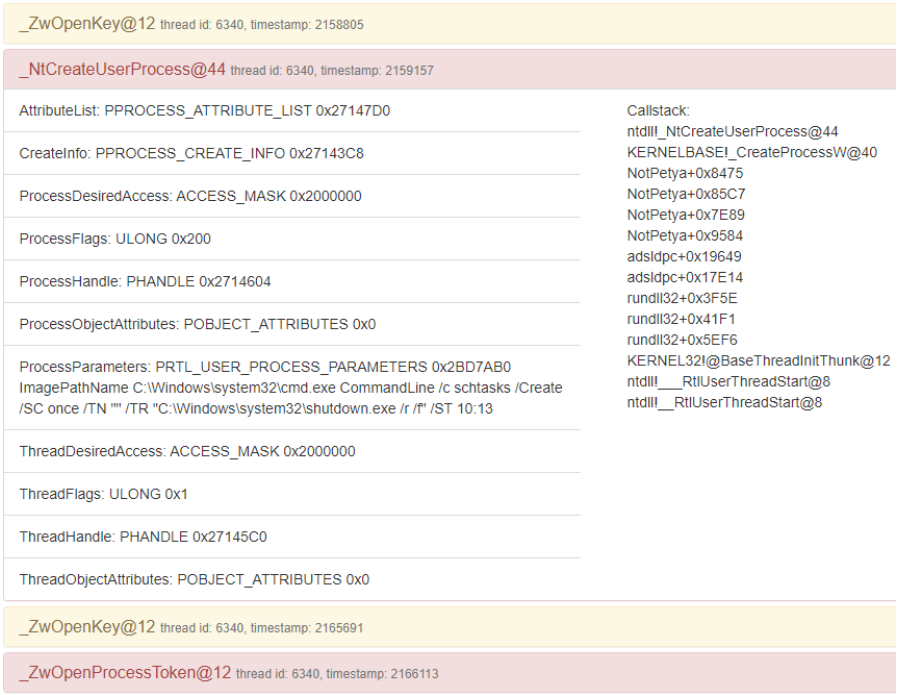

Figure 2. Exemple de trace sur une création de processus

Pour cela, un premier plugin dénommé `strace_nt` a été créé afin de *tracer* chaque appel système fait par un programme. Techniquement il s'agit de mettre un point d'arrêt (ou *breakpoint*) sur chaque appel système et lorsque ce dernier est atteint, de récupérer les valeurs concrètes des paramètres passés à cet appel. Ceci nous permet de rapporter les interactions du malware avec le système d'exploitation comme le font les *sandboxes* classiques, à savoir les actions sur :

5. Master Boot Record

— le système de fichiers (accès en lecture et surtout en écriture) ;
— le registre (clés créés, valeurs ajoutées, etc)
— le réseau (requêtes DNS, connexions entrantes et sortantes, etc) ;
— les processus créés (avec la command line associée) ;

Nous avons donc lancé une trace sur NotPetya et généré une visualisation HTML simplifiée des résultats, présentée sur la figure 2. On y voit notamment le malware utiliser la fonction système `_NtCreateUserProcess` qui lui permet d'enregistrer une tâche à exécuter. Cette tâche, qui utilise l'exécutable `shutdown`, forcera le redémarrage du système après un délai. Ce n'est qu'après ce redémarrage que NotPetya informera l'utilisateur que ses fichiers ont été chiffrés et la demande de rançon.

Ensuite, `IceBox` dispose, en plus des fonctionnalités des *sandboxes* classiques, d'un *helper* particulièrement utile : la récupération de la pile des fonctions appelantes (ou *callstak* user et kernel en 32 et 64-bits). On peut voir, toujours sur la figure 2, la `callstack` de l'appel de `_NtCreateUserProcess` et notamment constater que quatre fonctions du malware sont présentes. Ces informations sur le graphe d'appels des fonctions, généré dynamiquement, sont une aide précieuse à l'analyste afin d'améliorer sa compréhension d'un malware, surtout lorsque celui-ci fait des appels de fonctions dynamiques qui ne sont pas visibles lors d'une analyse statique.

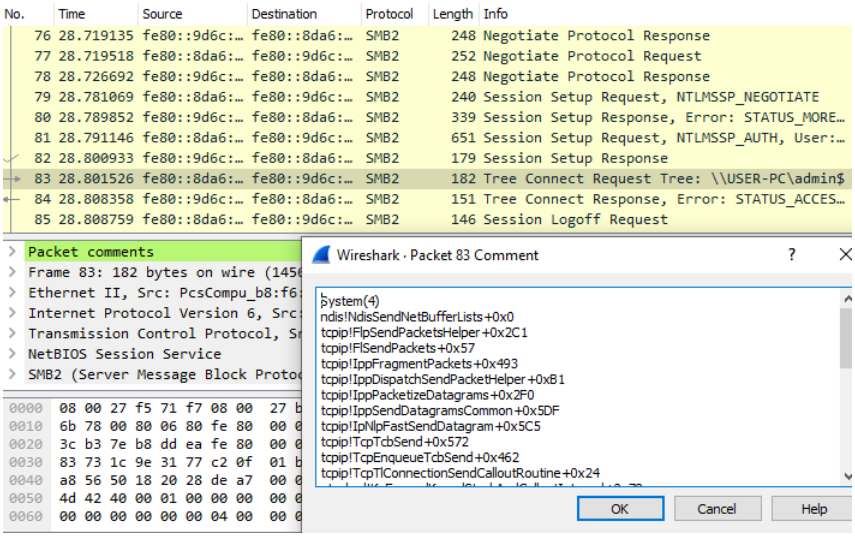

Figure 3. Trace noyau sur un message réseau

Un autre intérêt d'une solution basée sur de l'introspection hyperviseur est de permettre de poser des points d'arrêt sur des fonctions à priori inaccessibles

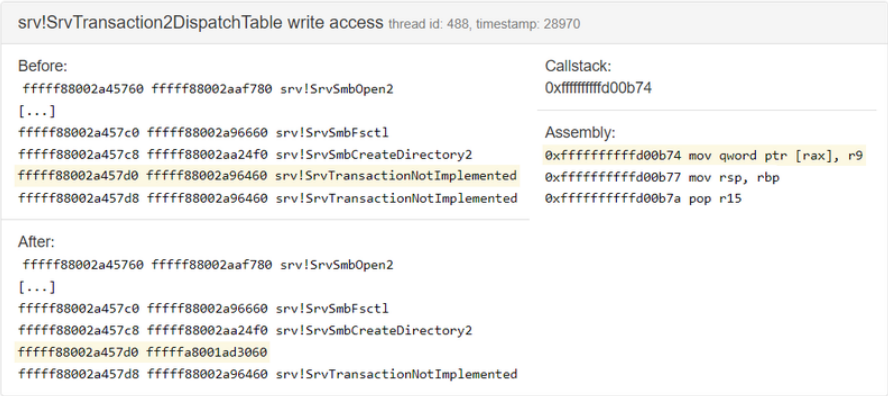

Figure 4. Trace noyau sur la vulnérabilité DoublePulsar

du monde *user*. Comme par exemple tracer les lectures et écritures de paquets réseaux directement au niveau de la pile IP du noyau.

Pour ce faire, un autre plugin, dénommé `wireshark` comme l'outil de référence, a été développé afin de faire le lien entre une capture réseau au format `pcap-ng` (visualisable dans l'outil `Wireshark` [19]) et l'origine d'un paquet réseau intercepté dans le programme émetteur.

Plus précisément, ce plugin pour `IceBox` capture des paquets réseaux (émis et reçus) au niveau du driver NDIS (juste avant le driver matériel de la carte réseau). C'est ce qui est représenté par la figure 3 ou l'on visualise la *callstack* sur un paquet SMB [6]. En effet, le malware NotPetya est capable de contaminer d'autres machines connectées en exploitant une vulnérabilité dans le partage de fichiers Windows.

Enfin, un dernier outil a été développé dans le but de capturer des écritures au niveau des pages noyau. Cet outil a permis de mettre en évidence l'installation de la *backdoor* `DoublePulsar`, visible sur la figure 4. On y voit qu'un pointeur de fonction externe a été écrit dans la table des fonctions du driver SMB. Ce nouveau pointeur de fonction permettra l'exécution de code arbitraire sur le système. Notre outil intercepte l'instruction exacte qui effectue cette écriture dans le noyau, du code non-référencé provenant de `DoublePulsar`.

6 Perspectives et évolutions

En l'état actuel, le projet `IceBox` est déjà clairement utilisable comme le montre l'exemple traité précédemment. Néanmoins, ce projet ne constitue qu'un socle sur lequel de nombreuses améliorations peuvent encore être apportées afin de simplifier le travail d'un analyste parmi lesquelles :

6. Server Message Block

Augmenter la furtivité. Bien que l'introspection en elle même soit furtive (le système invité n'a pas conscience que sa mémoire est lue, que des points d'arrêt sont placés, etc), `VirtualBox`, comme la plupart des systèmes de virtualisation, n'est absolument pas conçu dans l'objectif d'être furtif vis-à-vis des systèmes d'exploitation invités. De fait il est possible pour le système invité de détecter que l'environnement sur lequel il s'exécute est virtuel. C'est par exemple le but du projet `pafish` [11] qui permet de détecter la plupart des systèmes de virtualisation dont `VirtualBox`.

Dans ce cas, un malware serait capable d'adapter son comportement pour se rendre lui-même furtif vis-à-vis des analyses que l'on pourrait faire. Cependant, il est possible d'envisager des modifications directement au niveau du `VirtualBox` inclus dans `IceBox` afin de rendre inopérantes ces différentes détections proposées par `pafish`. On peut imaginer ainsi, rendre aléatoire les identifiants retournés par le matériel émulé par l'hyperviseur ou encore mentir sur les valeurs retournées par les instructions CPU de *profiling* comme RDTSC.

Améliorer la furtivité d'un hyperviseur est un sujet à part entière, complexe, et ouvert vers de multiples solutions.

Support de multiples CPU virtuels. Actuellement, `IceBox` ne supporte qu'un seul CPU virtuel. Cette limitation, détaillée dans un article de Xen [14], est utilisée comme élément de détection de *sandbox* et nuit de fait à la furtivité du projet. Elle nécessite une modification profonde dans l'hyperviseur afin de pouvoir être gérée de manière fiable.

Ajout de plugins d'analyse. Des travaux complémentaires sont envisagés afin de faciliter le travail de l'analyste. Outre les capacités de reporting des *sandboxes* classiques, des fonctionnalités avancées tirant pleinement profit des capacités des VMI sont en cours de développement parmi lesquelles :

- l'aide à l'"unpacking" automatique de malwares ;
- l'amélioration du plugin `wireshark` ;
- la complétude du plugin `strace` ;
- une API permettant d'accéder aux `mapping` mémoire d'un processus ;
- le monitoring des allocations mémoire ;
- des `bindings` python pour permettre un prototypage rapide ;
- le support de MacOS en système invité ;

7 Conclusion

Dans cet article, nous avons présenté notre solution open-source de VMI dénommée `IceBox`. C'est une solution portable, accessible sur les principaux systèmes hôtes Windows, Linux et MacOS. Les systèmes invités actuellement supportés sont Windows et Linux.

`IceBox` est aussi une solution performante, grâce à l'utilisation d'un hyperviseur et d'une architecture orientée temps-réel. Ce *framework* propose néanmoins

une API de haut-niveau permettant de manipuler simplement les entités du système invité comme les processus, les threads, leur évènements ou encore les symboles associés.

Enfin `IceBox` est un projet naissant, un socle permettant la création de multiples outils. Plus que l'analyse de malwares, c'est un outil puissant d'introspection du fonctionnement interne d'un système d'exploitation invité et des applicatifs exécutés afin d'en comprendre précisément les rouages.

Références

1. Applepie, `https://github.com/gamozolabs/applepie`
2. Bochs, `http://bochs.sourceforge.net`
3. Couffin, N. : Winbagility : Débogage furtif et introspection de machine virtuelle. SSTIC (2016)
4. Cuckoo Sandbox, `https://cuckoosandbox.org`
5. Double Pulsar Backdoor, `https://shasaurabh.blogspot.com/2017/05/doublepulsar-backdoor.html`
6. Microsoft Hyper-V, `http://www.microsoft.com/hyper-v`
7. Kernel patch protect, `https://en.wikipedia.org/wiki/PatchGuard`
8. Khourbiga, F., Deligne, E. : Sandbagility : un framework d'introspection en mode hyperviseur pour microsoft windows. SSTIC (2018)
9. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A. : Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In : Proceedings of the 30th Annual Computer Security Applications Conference (2014)
10. LibVMI, `http://libvmi.com`
11. Paranoid Fish, `https://github.com/a0rtega/pafish`
12. PyREBox, `https://talosintelligence.com/pyrebox`
13. QEMU, `https://www.qemu.org`
14. Stealthy Monitoring with Xen, `https://xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m`
15. Oreans Themida, `https://www.oreans.com/Themida.php`
16. Oracle VM VirtualBox, `https://www.virtualbox.org`
17. VMProtect, `https://vmpsoft.com`
18. Volatility Foundation, `https://www.volatilityfoundation.org`
19. Wireshark · Go Deep., `https://www.wireshark.org`
20. Xen Project, `https://xenproject.org`