Détection de protections Anti-VM utilisées par les malwares

Mémoire de fin d'étude

Master *Sciences et Technologies*,
Mention *Mathématiques*,
Parcours *Cryptologie et Sécurité Informatique*.

Auteur
Elouan Charbonnier <elouan.charbonnier@etu.u-bordeaux.fr>

Superviseur
Pierre Mondon <pierre.mondon@eshard.com>
Lionel d'Hauenens <lionel.dhauenens@eshard.com>

Tuteur
Emmanuel Fleury <emmanuel.fleury@u-bordeaux.com>

22 août 2025

Déclaration de paternité du document

Je certifie sur l'honneur que ce document que je soumets pour évaluation afin d'obtenir le diplôme de Master en *Sciences et Technologies*, Mention *Mathématiques* ou *Informatique*, Parcours *Cryptologie et Sécurité Informatique*, est entièrement issu de mon propre travail, que j'ai porté une attention raisonnable afin de m'assurer que son contenu est original, et qu'il n'enfreint pas, à ma connaissance, les lois relatives à la propriété intellectuelle, ni ne contient de matériel emprunté à d'autres, du moins pas sans qu'il ne soit clairement identifié et cité au sein de mon document.

Date et Signature
22 août 2025

Remerciements

J'aimerais tout d'abord remercier mes superviseurs Pierre Mondon et Lionel d'Hauenens pour leur accompagnement, leurs conseils et leur soutien tout au long de mon stage.

Je remercie également l'ensemble de l'équipe d'eShard pour leur accueil chaleureux et leur aide précieuse durant cette période.

Enfin, je remercie Emmanuel Fleury, mon tuteur de stage, pour l'opportunité d'effectuer ce stage au sein d'eShard.

Résumé

Afin d'analyser un logiciel malveillant, l'utilisation d'un environnement *jetable* et isolé permet d'expérimenter le logiciel susmentionné sans prendre de risques. Pour cela, nous utilisons des machines virtuelles qui, au besoin, peuvent être réinitialisées à un état antérieur (*snapshot*), souvent isolées de tout réseau ou de toute machine vulnérable. Ainsi, tout développeur de maliciel prend cela en compte et intègre dans son produit des techniques dites anti-VM. C'est-à-dire des techniques permettant de détecter la présence de machines virtuelles qui vont modifier le comportement du logiciel afin de rendre plus difficile son analyse.

Ce mémoire porte sur ces méthodes, leur fonctionnement et comment les contrer. La première partie explique les principes de virtualisation et d'émulation ainsi que l'utilisation de l'émulateur QEMU. La deuxième partie explique un certain nombre de protections anti-VM, qui se basent sur des API Windows et des instructions assembleur. Puis, la troisième partie explique comment contrer ces techniques à l'aide de configuration de machines virtuelles, de *hooks* et de WinDbg. Enfin, la quatrième partie montre l'utilisation de ces techniques dans deux cas concrets, le maliciel *PoC* (Proof of Concept) `al-khaser` et un cheval de Troie bancaire nommé *QakBot*.

Table des matières

Introduction **1**

I Virtualisation et émulation **2**

I.1 Définitions **3**
I.1.1 Machine virtuelle 3
I.1.2 Hyperviseur 3
I.1.2.1 Types d'hyperviseur 3
I.1.3 Virtualisation 4
I.1.3.1 Virtualisation complète 4
I.1.3.2 Paravirtualisation 5
I.1.3.3 Virtualisation assistée par le matériel 5
I.1.4 Émulation 6

I.2 QEMU **7**
I.2.1 Définition et modes d'utilisation 7
I.2.1.1 Présentation générale 7
I.2.1.2 Modes d'utilisation 7
I.2.2 Fonctionnement d'un traducteur 9
I.2.2.1 Traduction d'instruction 9
I.2.2.2 Utilité d'un traducteur 9
I.2.2.3 Bloc de traduction (`Translation Block`) 10
I.2.3 TCG, Traducteur de QEMU 10
I.2.4 Emulation des composants 14
I.2.4.1 Unité de gestion de mémoire (*MMU*, *Memory Management Unit*) 14
I.2.4.2 Processeur, régions mémoires et disque dur 14

II Techniques d'anti-virtualisation/anti-émulation **16**

II.1 API Windows **17**
II.1.1 Interaction matérielle 17
II.1.1.1 Résolution de l'écran 17
II.1.1.2 Taille du stockage 18
II.1.1.3 Taille de la mémoire 19
II.1.1.4 Adresse MAC 20
II.1.1.5 Nombre de CPU 21
II.1.1.6 Fonctionnalités d'alimentation 22
II.1.2 Interaction des artéfacts 24
II.1.2.1 Services 24
II.1.2.2 `RegKey` 26
II.1.3 Interaction humaine 27
II.1.3.1 Mouvement de la souris 27
II.1.3.2 Changement de fenêtre 27

II.2 Instruction machine **29**
II.2.1 `cpuid` 29
II.2.1.1 Bit d'hyperviseur 30
II.2.1.2 Chaîne d'identifiant de l'hyperviseur 30
II.2.2 `rdtsc` 31
II.2.3 `sidt` 32
II.2.3.1 `IDT` 32

II.2.3.2 Explication de la technique . . . 33

II.3 Utilisation de ces techniques dans un maliciel . . . 34

III Méthodes de contre-mesures . . . 36

III.1 Modification de configuration . . . 37
III.1.1 Augmenter la taille du disque dur . . . 38
III.1.2 Modifier le nombre de CPU et la quantité de RAM . . . 38
III.1.3 Augmenter la résolution de l'écran . . . 38
III.1.4 Modifier l'adresse MAC . . . 38
III.1.5 Modifier les informations que `cpuid` va récupérer . . . 39

III.2 *Hooks* des fonctions de l'API Windows . . . 40
III.2.1 Injection de DLL . . . 40
III.2.2 Installation d'un *hook* . . . 40
III.2.2.1 Explication du *Inline Hooking* . . . 40
III.2.2.2 Exemple de code . . . 41
III.2.3 Mise en place de substitut . . . 42

III.3 Appliquer un correctif . . . 43
III.3.1 Mise en place de l'environnement . . . 43
III.3.2 Rentrer dans le contexte d'un processus . . . 44
III.3.3 Retrouver le point d'entrée d'un exécutable . . . 45
III.3.4 Exemple : Contournement de la technique `SIDT` . . . 46

IV Cas pratiques . . . 48

IV.1 *Al-Khaser* . . . 49
IV.1.1 Techniques d'Anti-VM pour QEMU . . . 49
IV.1.1.1 `ACPI` . . . 50
IV.1.1.2 `SMBIOS` . . . 50
IV.1.2 Techniques d'Anti-VM pour Hyper-V . . . 51
IV.1.3 Techniques d'Anti-VM générales . . . 52
IV.1.3.1 *Windows Management Instrumentation* (WMI) . . . 52
IV.1.3.2 Entrée utilisateurs . . . 53

IV.2 *QakBot* . . . 54
IV.2.1 Échantillon . . . 54
IV.2.2 Analyse . . . 54
IV.2.2.1 `sub_4033fc()` \ `avm_in_vmware_version()` . . . 55
IV.2.2.2 `sub_40349a()` \ `avm_in_vmware_memory_size()` . . . 56
IV.2.2.3 `sub_4035b6()` \ `avm_devices_check()` . . . 56
IV.2.2.4 `sub_40385e()` \ `avm_process_check()` . . . 57
IV.2.2.5 `sub_40336e()` \ `call_cpuid()` . . . 58
IV.2.3 Résultat de l'analyse . . . 58

Conclusion . . . 59

Annexes . . . 61
Annexe 1 — Script permettant de retrouver le code assembleur de la traduction du code ARM . . . 61
Annexe 2 — Fonction `SERVICES_NAME_LIST` et `belong_to` — Détection des services liés à des machines virtuelles . . . 62
Annexe 3 — Liste de clefs de registre standard pour les gestionnaires de machines virtuelles . . . 63
Annexe 4 — Fonction `qemu_firmware_ACPI` du maliciel `al-khaser` . . . 64
Annexe 5 — Fonction `get_system_firmware` — récupération des tables micrologicielles . . . 66
Annexe 6 — Fonction `enumerate_object_directory` — Permet de retrouver la liste des noms des objets présent dans un répertoire du gestionnaire d'objets. . . . 67
Annexe 7 — Fichier `Generic.h` de `al-khaser` — En-tête contenant toutes les fontions d'anti-VM génériques. . . . 68
Annexe 8 — Fonction `ExecWMIQuery` — Permet d'exécuter une requête WMI. . . . 69

Annexe 9 — Fonctions `DetectSnxhkWindow` et `EnumWindows_find_match_class` - Teste la présence d'une fenêtre windows ayant un certains nom de classe 70
Annexe 10 — Fonction `sub_403ef7` — Implémente un certains nombre de protection anti-VM. 71
Annexe 11 — Fonction `sub_403ef7` renommée — Implémente un certains nombre de protection anti-VM. 72
Annexe 12 — Listes des descriptions et noms de classe testées par *QakBot* 73
Annexe 13 — Syntaxe de la fonction `SetupDiGetDeviceRegistryPropertyA` 74
Annexe 14 — Fonction `avm_devices_check` 75
Annexe 15 — Listes des processus testés par *QakBot* 77
Annexe 16 — Fonction `avm_process_check` 78
Annexe 17 — Fonction `cpuid_call()` 80

Bibliographie **82**

Introduction

Le développement d'un logiciel malveillant requiert une connaissance approfondie de la cible (API, architecture, etc.), mais aussi une compréhension des méthodes d'analyse. Les objectifs d'un maliciel varient, mais les principaux sont l'exfiltration de données, le contrôle à distance ou le chiffrement des données dans le cadre d'une demande de rançon.. Ces objectifs nécessitent une persistance au sein d'un système, ce qui accroît leur exposition aux outils d'analyse.

Plus un logiciel laisse de traces, plus il est facile de comprendre son fonctionnement et de s'en prémunir. Pour cette raison, il est courant que les analystes créent des environnements isolés et spécialisés dans l'étude de logiciels malveillants (*sandboxes*), équipés d'outils d'analyse qui facilitent la compréhension de leur fonctionnement.

Afin de contrer ces analyses, beaucoup de maliciels implémentent des mécanismes qui permettent de détecter ces environnements d'analyse. Ces techniques, dites d'anti-VM, sont essentielles pour des logiciels de grande envergure. Ces mécanismes peuvent mener à différents comportements pour le logiciel, mais dans la plupart des cas, ces techniques mènent à la mise en sommeil du logiciel ou à la modification de son comportement en quelque chose de bénin. Par exemple, le cheval de Troie *Blackmoon* [1] adopte la forme d'une application anodine. Lors de son exécution, il emploie des méthodes d'anti-VM et d'anti-debug afin de savoir s'il doit continuer à s'exécuter ou renvoyer un message expliquant que cette application ne peut être lancée dans une machine virtuelle.

Il existe de nombreuses méthodes possibles pour détecter un environnement virtualisé (ou émulé), les plus répandues étant des instructions en assembleur qui permettent d'interroger le processeur, l'hyperviseur ou encore des composants clefs du système. Certaines méthodes sont réservées à certains systèmes, par exemple, l'API Windows permet d'accéder à de nombreux paramètres du système tels que la position de la souris, la résolution de l'écran, la taille du stockage, etc. Comme nous le verrons, ces informations peuvent être utilisées afin de repérer la présence d'une machine virtuelle.

Il est néanmoins important de noter que ces protections peuvent aussi se retourner contre leurs développeurs. En effet, la plupart des entreprises ont tendance à avoir un parc de machines virtuelles pour gérer tout un tas de tâches différentes. Bien que ce soient des machines virtuelles, toutes ne sont pas dédiées à l'analyse et à la détection de maliciels. Par ailleurs, des techniques d'anti-VM ayant des critères trop stricts amèneront le maliciel à éviter les machines avec des configurations plus anciennes. Ainsi, il est important de savoir quelles sont les limites à poser pour ces mécanismes. Bien que très utiles, il sera montré que ces protections anti-VM restent bien moins répandues que d'autres techniques et lorsqu'elles sont implémentées, elles ont tendance à être assez rudimentaires.

Dans ce mémoire, nous allons présenter les principes de virtualisation et d'émulation, présenter des techniques d'anti-VM couramment utilisées et montrer comment contrer ces dernières. Nous allons aussi présenter deux cas concrets de maliciels utilisant ces méthodes anti-virtualisation (ou anti-émulation).

1. Plus communément appelé *KrBanker*, le nom *Blackmoon* provient d'une chaîne de caractères de débogage retrouvée dans le premier échantillon découvert de ce maliciel.

Première partie

Virtualisation et émulation

Chapitre I.1

Définitions

I.1.1 Machine virtuelle

Une machine virtuelle [1], ou machine invitée, est un environnement logique qui imite ou reproduit le comportement d'une machine physique. En général, une machine virtuelle est créée par un processus de virtualisation, reposant sur un hyperviseur chargé de gérer et d'isoler les systèmes invités [1]. Toutefois, par abus de langage, on désigne parfois également comme « machine virtuelle » un environnement émulé, c'est-à-dire entièrement simulé par logiciel, sans recourir à un hyperviseur ni aux extensions de virtualisation matérielle. Dans ce cas, le système hôte reproduit complètement un nouvel ensemble de composants « émulés » pour la machine invitée.

I.1.2 Hyperviseur

Un hyperviseur est un *logiciel* qui permet de créer et de gérer les machines virtuelles. C'est un élément indispensable de la virtualisation, qui met à disposition, pour chaque invité, un ensemble de ressources (processeurs, mémoire, périphériques).

I.1.2.1 Types d'hyperviseur

Il existe deux types d'hyperviseurs [2] :

- Type 1 (*bare-metal*) : L'hyperviseur est installé sur le matériel et communique directement avec ce dernier.

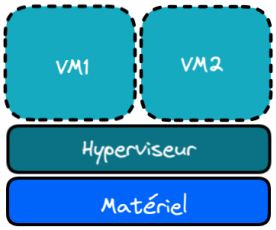

FIGURE I.1.1 – Représentation logique d'un hyperviseur de type 1

- Type 2 (*hosted*) : L'hyperviseur va communiquer avec le matériel par le biais du système d'exploitation hôte.

1. On peut aussi mentionner les environnements d'exécution de langage tels que la JVM pour Java qui, malgré l'appellation « machine virtuelle », ne relèvent pas exactement de la virtualisation.

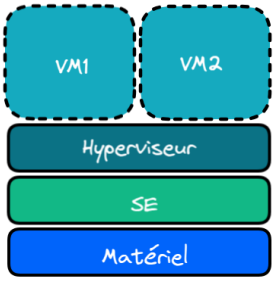

FIGURE I.1.2 – Représentation logique d'un hyperviseur de type 2

Il est théoriquement possible d'utiliser chacun des deux types d'hyperviseurs pour les différents modes de virtualisation que nous allons présenter.

I.1.3 Virtualisation

La virtualisation est un ensemble de techniques permettant d'isoler des ressources matérielles ou logicielles afin d'exécuter des environnements séparés (machines virtuelles). Comme mentionné précédemment, la virtualisation repose généralement sur un hyperviseur et, par extension, peut également être associée à une émulation logicielle. Dans cette section, nous allons nous concentrer sur la virtualisation dans son sens le plus strict. On distingue trois types de virtualisation :

- la virtualisation complète : virtualisation complète du matériel physique par l'hyperviseur.
- la paravirtualisation : modification de l'invité afin qu'il soit conscient de l'existence de l'hyperviseur.
- la virtualisation assistée par le matériel [2] : virtualisation par les extensions matérielles (Intel VT-x, AMD-V) aidant l'hyperviseur.

Dans les prochaines sections, nous allons examiner plus en détail le fonctionnement de chaque technique.

I.1.3.1 Virtualisation complète

Dans ce mode de virtualisation, le système invité n'est pas conscient de sa virtualisation. L'hyperviseur lui fournit une version virtualisée du matériel physique. Cela garantit une compatibilité avec le système invité sans avoir besoin d'une quelconque modification.

Pour isoler correctement les systèmes invités, l'hyperviseur intercepte les communications entre l'invité et le matériel physique (par exemple, les instructions privilégiées telles qu'un accès au CPU), puis il les traduit via des mécanismes de virtualisation matérielle afin d'assurer la compatibilité avec le matériel physique non virtualisé. Lors de la traduction, l'hyperviseur doit, pour chacune de ces communications, les intercepter, les traiter puis les renvoyer ; cela va mener à une surcharge importante pour chaque communication et donc à un ralentissement du système invité.

2. Ce n'est pas vraiment une technique à part entière de la virtualisation, cependant, cela reste une optimisation qui diverge grandement des autres techniques.

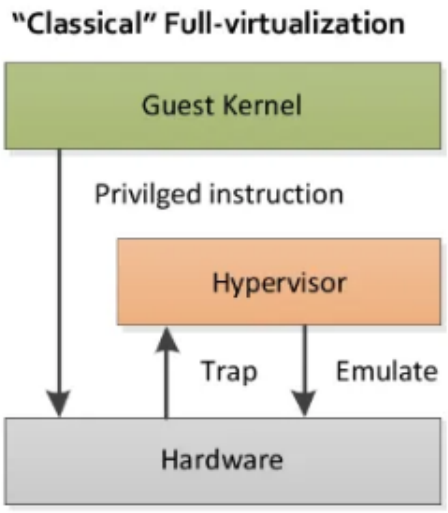

FIGURE I.1.3 – Schéma représentant la virtualisation complète [3]

I.1.3.2 Paravirtualisation

La paravirtualisation est un mode de virtualisation dans lequel le système invité conscient de sa virtualisation. Dans ce cas, le système invité communique directement avec l'hyperviseur (à l'aide d'*hyperappels*), ce qui ne nécessite plus de traduction des communications ; cependant, cela nécessite des modifications sur le système invité.

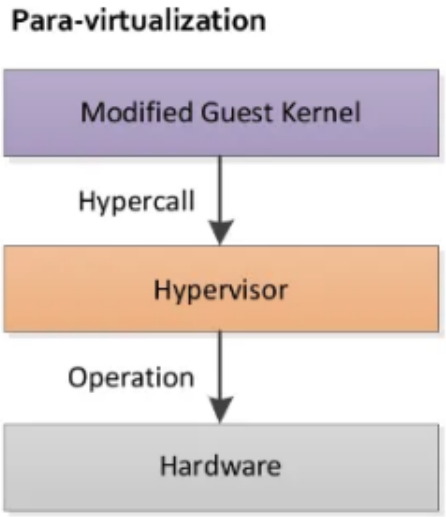

FIGURE I.1.4 – Schéma représentant la paravirtualisation [3]

I.1.3.3 Virtualisation assistée par le matériel

Nous allons présenter le troisième type de virtualisation, qui n'est pas vraiment un type à part de virtualisation (ou d'émulation) mais plutôt une amélioration. Cette technique résout l'un des plus grands problèmes de la virtualisation, la vitesse d'exécution.

Cette virtualisation utilise des extensions intégrées aux processeurs, telles qu'Intel VT-x ou AMD-V. Ces extensions permettent de faire en sorte qu'un processeur, initialement conçu pour ne faire fonctionner qu'un seul système d'exploitation, puisse directement aider à la virtualisation.

Pour cela, ces extensions ajoutent un mode d'exécution au processeur, le mode invité. Ce mode sera utilisé pour exécuter le code de l'invité. L'utilisation de nouvelles instructions permettra le changement de mode en cas de ren-

contre avec une instruction privilégiée, qui nécessitera une action du mode noyau.

I.1.4 Émulation

L'émulation consiste à générer un ensemble de composants matériels (processeur, bus, périphériques, etc.) afin d'exécuter un système invité d'une architecture différente de l'hôte. Contrairement à la virtualisation, l'émulation a besoin de traduire les instructions venant de l'invité afin que l'hôte puisse les traiter. Cette fonctionnalité permet la compatibilité multi-architecture au prix de performances réduites.

Dans le chapitre suivant, nous examinerons le fonctionnement de l'émulation plus en détail en prenant l'exemple de l'émulateur QEMU.

Chapitre I.2

QEMU

Dans ce chapitre, nous expliquons en détail le fonctionnement d'un émulateur. Pour cela, nous allons prendre l'exemple de QEMU [4].

I.2.1 Définition et modes d'utilisation

I.2.1.1 Présentation générale

QEMU, Quick EMULator, est un émulateur de systèmes et de binaires. Il permet d'émuler (ou de virtualiser à l'aide d'un hyperviseur) une architecture pouvant être différente de celle de l'hôte. Cela constitue l'une des plus grandes forces de QEMU avec le fait qu'il soit libre. La grande polyvalence de QEMU lui permet d'émuler un très grand nombre de systèmes. Il est également possible de l'utiliser avec des hyperviseurs tels que *KVM* afin de passer de l'émulation à la virtualisation.

QEMU a deux modes d'utilisation principaux : le mode utilisateur, où il exécute des exécutables compilés pour une architecture différente de celle de l'hôte et le mode système, où il émule un système dans son entièreté, incluant donc les composants physiques tels que le processeur, la mémoire vive, les différents périphériques, etc.

Dans le cadre de la rétro-ingénierie et de l'analyse de maliciels, on utilise souvent *PANDA* [5], basé sur QEMU. *PANDA* est un logiciel d'analyse dynamique qui permet d'explorer en détail le fonctionnement interne du système observé. De manière plus détaillée, *PANDA* donne à l'utilisateur accès à l'entièreté des instructions exécutées par le système ciblé. Cela permet d'enregistrer absolument tout ce qui s'est déroulé dans le système durant un laps de temps choisi.

Comme nous le verrons plus tard, il est également possible de récupérer l'état interne de QEMU (plus particulièrement de TCG) à tout moment. En mode utilisateur, certaines options permettent, par exemple, de retrouver le code assembleur dans l'architecture de l'invité, une fois interprété par TCG, puis comment TCG le traduit pour l'hôte.

I.2.1.2 Modes d'utilisation

Nous allons détailler le fonctionnement ainsi que l'utilité de chaque mode de QEMU, notamment comment fonctionne la traduction entre deux architectures différentes (invité et hôte), et ce que cela implique dans l'architecture interne de QEMU.

Mode utilisateur

Comme exprimé précédemment, ce mode permet d'exécuter un binaire compilé pour une architecture différente de celle de la machine hôte. Il est actuellement possible d'émuler un binaire compilé pour Linux (`qemu-linux-user`) et BSD (`qemu-bsd-user`)[1]. Le schéma ci-dessous représente l'état d'une machine lorsque QEMU est utilisé en mode émulation :

1. Pour une architecture particulière, il est possible de faire appel à `qemu-<arch>` avec `arch` une architecture supportée par QEMU. Par exemple, pour une architecture ARM [6], on fera `qemu-arm`.

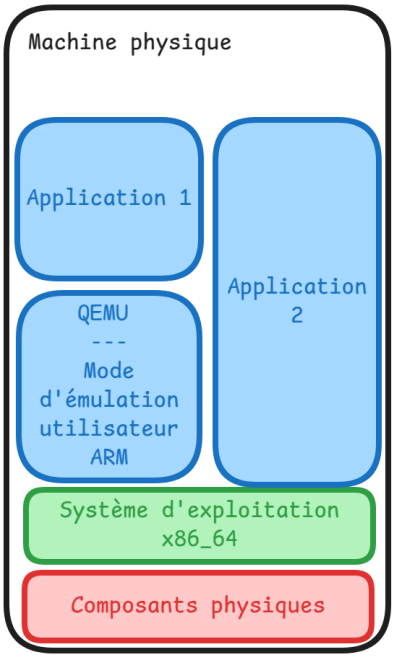

FIGURE I.2.1 – Schéma du mode d'émulation utilisateur de QEMU

Comme indiqué dans la documentation de QEMU, l'émulation en mode utilisateur présente plusieurs caractéristiques :

- Traduction des appels systèmes : Système de traduction des appels systèmes qui permet traduire les paramètres des instructions afin que l'hôte et l'invité puisse communiquer.
- Gestion des signaux POSIX : Redirige les signaux venant de l'hôte vers le programme invité.
- Gestion des threads : Permet de gérer un thread hôte (processeur virtuel) pour chaque thread invité.

Mode système

En mode système, QEMU va émuler un système à part entière, c'est-à-dire émuler l'entièreté des ressources physiques dont le système a besoin. Comme évoqué précédemment, QEMU peut être couplé avec un hyperviseur issu de la liste suivante [7] :

Accelerator	Host OS	Host Architectures
KVM	Linux	Arm (64 bit only), MIPS, PPC, RISC-V, s390x, x86
Xen	Linux (as dom0)	Arm, x86
Hypervisor Framework (hvf)	MacOS	x86 (64 bit only), Arm (64 bit only)
Windows Hypervisor Platform (whpx)	Windows	x86
NetBSD Virtual Machine Monitor (nvmm)	NetBSD	x86
Tiny Code Generator (tcg)	Linux, other POSIX, Windows, MacOS	Arm, x86, Loongarch64, MIPS, PPC, s390x, Sparc64

FIGURE I.2.2 – Liste des hyperviseurs supportés par QEMU ainsi que les hôtes compatibles [7]

Si, lors de l'utilisation de QEMU, aucun hyperviseur n'est précisé, QEMU utilise un moteur de traduction dynamique (JIT) nommé Tiny Code Generator (TCG). Comme défini dans la documentation QEMU relative à TCG :

> *The Tiny Code Generator (TCG) exists to transform target insns (the processor being emulated) via the TCG frontend to TCG ops which are then transformed into host insns (the processor executing QEMU itself) via the TCG backend.*

Ci-dessous un schéma représentant le fonctionnement de QEMU en mode système :

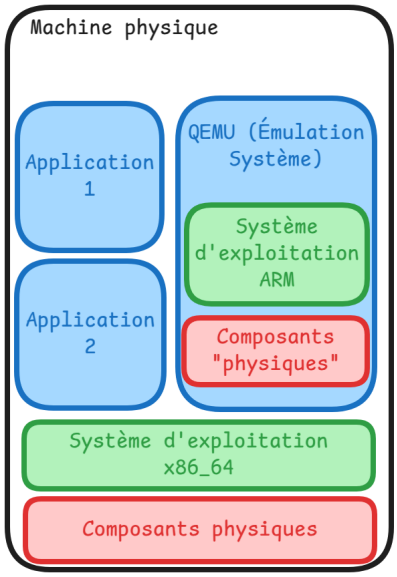

FIGURE I.2.3 – Schéma du mode d'émulation système de QEMU

I.2.2 Fonctionnement d'un traducteur

Avant de se pencher sur le fonctionnement interne de QEMU, nous allons d'abord nous intéresser au fonctionnement d'un traducteur, tel que TCG [8] [9].

I.2.2.1 Traduction d'instruction

Lorsque l'on souhaite émuler un système ayant une architecture différente de l'hôte, il est nécessaire de traduire les instructions. En effet, deux architectures différentes ont deux jeux d'instructions (*ISA*, Instruction Set Architecture) distincts, ce qui signifie que deux codes faisant la même chose ne sont pas représentés de la même manière en assembleur. Par exemple, si l'on prend un code en assembleur ARM et un en x64, et que l'on souhaite effectuer une simple addition, nous aurons les codes suivants :

```
1 mov rax, 5
2 mov rbx, 3
3 add rax, rbx
```

```
1 mov r0, #5
2 mov r1, #3
3 add r0, r0, r1
```

Comme on peut le voir, les noms des registres, la façon de charger une valeur dans un registre ainsi que la syntaxe pour additionner sont différentes entre le jeu d'instructions x64 (à gauche) et le jeu d'instructions ARM (à droite).

Donc, si nous voulons faire en sorte d'exécuter un binaire compilé dans une architecture ARM sur un système x64, nous devons ainsi être capables de faire comprendre au système hôte (x64) ce que fait le code invité (ARM). Bien que nous puissions simplement faire une traduction de ARM vers x64, il est également possible (et recommandé) d'utiliser un traducteur d'instructions.

I.2.2.2 Utilité d'un traducteur

Pratiquement parlant, il est possible d'émuler un système sur un autre sans utiliser de traducteur générique [2]. Le problème est que cette approche ne fonctionne qu'au cas par cas. Par exemple, si l'on souhaite émuler un système invité 1 sur le système hôte, on peut traduire le langage de l'invité 1 afin que l'hôte le comprenne.

Cependant, si l'on souhaite émuler d'autres architectures (MIPS, x64, etc.) en plus de celle que nous avons déjà (ARM), on ne peut pas garder la solution précédente qui est conçue que pour traduire une seule sorte d'architecture. En effet, dans ce cas, le traducteur va essayer de traduire les instructions des autres architectures sans jamais réussir.

2. https://www.usenix.org/legacy/publications/library/proceedings/usenix-nt97/full_papers/chernoff/chernoff.pdf

Pour pallier cela, il faut utiliser un traducteur tel que TCG qui va recevoir les instructions, les traduire dans son propre ensemble d'instructions (pseudo architecture), puis les retraduire pour l'architecture ciblée. Ainsi, on a le fonctionnement suivant :

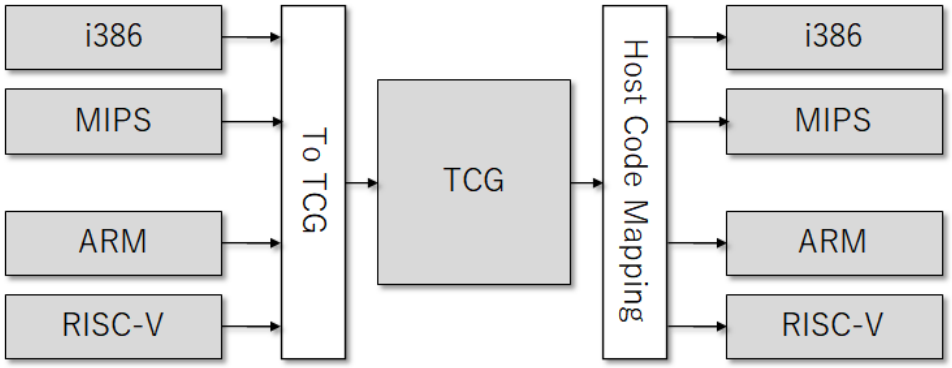

FIGURE I.2.4 – Exemple du traducteur TCG, [10]

I.2.2.3 Bloc de traduction (`Translation Block`)

Dans le code assembleur, TCG va séparer des morceaux de ce code en blocs de traduction. Ces blocs sont séparés soit par des instructions de saut, soit par une taille définie par TCG. TCG va traduire le code invité, l'interpréter dans son propre langage, puis générer le code pour l'hôte.

Un bloc de traduction est composé des éléments suivants :

- L'adresse de la première instruction du bloc.
- Le code machine après traduction.

Ces blocs de traduction peuvent être réutilisés par TCG, par ailleurs, avant de traduire toute instruction, TCG regarde s'il n'a pas déjà traduit ce bloc.

I.2.3 TCG, Traducteur de QEMU

Nous allons maintenant nous pencher sur le fonctionnement exact de l'architecture interne à QEMU. Nous allons voir plus en détail comment est gérée la traduction avec TCG.

Comme expliqué précédemment, pour émuler une architecture différente de celle de l'hôte, nous allons nous aider d'un traducteur, dans le cas de QEMU, c'est *Tiny Code Generator* (TCG).

Tout au long de cette partie, nous allons prendre pour exemple le code suivant en ARM et l'exécuter à l'aide de QEMU sur une machine x86-64 :

```
@ Compilation :
@ arm-linux-gnueabi-as -o add.o add.s
@ arm-linux-gnueabi-ld -o add add.o

.section .text
.global _start

_start:
    mov r0, #5
    mov r1, #7
    add r0, r0, r1
    mov r7, #1
    svc #0
```

Nous allons émuler l'exécution de ce binaire à l'aide de la commande suivante :

```
qemu-arm -d in_asm,op,out_asm -D add.log -singlestep ./add
```

Voici une explication de la commande et de ses paramètres [11] :

- `qemu-arm` : Lance un programme ARM en mode émulation utilisateur.
- `-d` : Permet de remplir un journal de l'exécution du binaire avec différentes options, qui sont définies comme suit dans la documentation :
  - `in_asm` : Montre le code assembleur invité (ARM).
  - `op` : Montre le code de la représentation interne de TCG.
  - `out_asm` : Montre le code généré en assembleur pour l'hôte (x86-64).
- `-D` : Crée un fichier pour le journal récupéré.
- `-singlestep` : Lance le programme en s'arrêtant pour remplir le journal pour chaque instruction.

Cette commande va récupérer les traductions effectuées par TCG lors de l'exécution du binaire, puis va les écrire dans un fichier. Nous allons donc avoir le code assembleur en ARM, le code de la représentation interne de TCG et le code généré pour l'hôte.

Nous allons par la suite étudier l'extrait suivant du journal afin de comprendre comment fonctionne la traduction :

```
IN:
0x00010054:
OBJD-T: 0500a0e3

OP:
 ld_i32 loc3,env,$0xfffffffffffffff0
 brcond_i32 loc3,$0x0,lt,$L0
 st8_i32 $0x1,env,$0xfffffffffffffff4

 ---- 0000000000010054 0000000000000000 0000000000000000
 mov_i32 loc6,$0x5
 mov_i32 r0,loc6
 mov_i32 pc,$0x10058
 call lookup_tb_ptr,$0x6,$1,tmp9,env
 goto_ptr tmp9
 set_label $L0
 exit_tb $0x736b64000043

OUT: [size=64]
  -- guest addr 0x0000000000010054 + tb prologue
0x736b64000100:
OBJD-H: 8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfd
OBJD-H: ff1512000000ffe0488d0514ffffffe9e4feffff
  -- tb slow paths + alignment
0x736b64000134:
OBJD-H: 90909090
  data: [size=8]
0x736b64000138:  .quad  0x000059c6dad02c20
```

Il y a trois sections par bloc de code :

- `IN` : Section de code de l'invité contenant l'adresse de l'instruction et un *opcode* de l'instruction
- `OP` : Section contenant la traduction du code invité en instructions de la représentation interne de TCG.
- `OUT` : Section contenant le code x86-64 généré à l'aide de la représentation interne de TCG.

Étudions en détail ces sections, en commençant par la première section qui est le code de l'invité.

Code de l'invité

```
IN:
 0x00010054:
 OBJD-T: 0500a0e3
```

On retrouve dans le journal, dans la section `IN`, le code ARM exécuté par l'invité qui est à l'adresse `0x10054`, il n'est pas représenté en assembleur dans le journal, mais en *opcode*.

L'*opcode* `0500a0e3` correspond à l'instruction :

```
mov         r0, #5
```

Cette instruction va être récupérée par TCG, puis transformée afin de correspondre à la syntaxe de la représentation interne de TCG.

Code de la représentation interne de TCG

```
1  OP:
2   ld_i32 loc3,env,$0xfffffffffffffff0
3   brcond_i32 loc3,$0x0,lt,$L0
4   st8_i32 $0x1,env,$0xfffffffffffffff4
5
6   ---- 0000000000010054 0000000000000000 0000000000000000
7   mov_i32 loc6,$0x5
8   mov_i32 r0,loc6
9   mov_i32 pc,$0x10058
10  call lookup_tb_ptr,$0x6,$1,tmp9,env
11  goto_ptr tmp9
12  set_label $L0
13  exit_tb $0x736b64000043
```

De la ligne 7 à 8, il y a la traduction dans la représentation interne de TCG de l'instruction en ARM. On a donc l'instruction `mov_i32 loc6,$0x5` qui va placer `5` dans `loc6` puis l'instruction `mov_i32 r0,loc6` va placer la valeur que contient `loc6` dans `r0`.

Finalement, de la ligne 9 à 13, le code sert à sauter au prochain bloc de traduction (*Translation Block*).

Code généré pour l'hôte

```
1  OUT: [size=64]
2    -- guest addr 0x0000000000010054 + tb prologue
3  0x736b64000100:
4  OBJD-H: 8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfd
5  OBJD-H: ff1512000000ffe0488d0514ffffffe9e4feffff
6    -- tb slow paths + alignment
7  0x736b64000134:
8  OBJD-H: 90909090
9    data: [size=8]
10 0x736b64000138:  .quad  0x000059c6dad02c20
```

Dans cette partie, le code généré est donné sous la forme d'une chaîne hexadécimale qui est donnée aux lignes 4 et 5 :

```
8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfdff1512000000ffe0488d0514ffffffe9e4feffff
```

On peut en avoir le cœur net en utilisant le script python en Annexe 1 qui écrit le code traduit dans un fichier.

On va retrouver dans le fichier `add_x86_64` le code assembleur qui a été généré par TCG pour l'hôte :

```
1  mov ebx,[rbp-0x10]
2  test ebx,ebx
3  jl near 0x28
4  mov byte [rbp-0xc],0x1
5  mov dword [rbp+0x0],0x5
6  mov dword [rbp+0x3c],0x10058
7  mov rdi,rbp
8  call [rel 0x38]
9  jmp rax
```

```
10  lea rax,[rel 0xffffffffffffff43]
11  jmp 0xffffffffffffff18
```

On peut retrouver à la ligne 5 la version x86_64 de l'instruction `mov r0, #5` que nous avions en ARM.

Ci-dessous, un schéma résumant la traduction de l'instruction `mov r0, #5` :

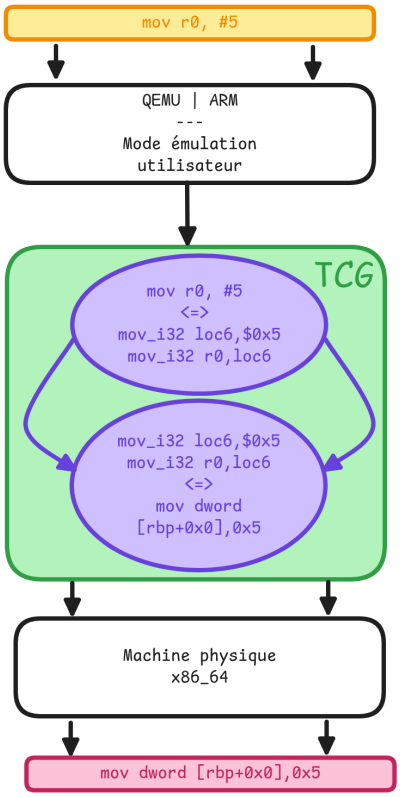

FIGURE I.2.5 – Schéma résumant la traduction d'une instruction ARM

I.2.4 Emulation des composants

Nous avons vu comment QEMU fait pour traduire des instructions d'une architecture à une autre. Cependant, QEMU est aussi capable d'émuler l'entièreté d'un système, ce qui nécessite des composants matériels tels qu'un processeur, de la mémoire vive ou un disque dur. Il faut donc que QEMU soit capable de les émuler. Dans cette section, nous allons voir les composants que QEMU émule et comment il procède.

I.2.4.1 Unité de gestion de mémoire (*MMU, Memory Management Unit*)

Comme indiqué dans la documentation de QEMU [12], la MMU, qui est responsable de la traduction des adresses virtuelles vers des adresses physiques, est nommée *softmmu* dans QEMU. *softmmu* est utilisé par le TCG pour émuler la mémoire virtuelle.

Pour réaliser cela, il implémente un TLB (Translation Lookaside Buffer) qui est un cache mémoire qui va enregistrer les dernières pages mémoires auxquelles le processeur a dû accéder. Cela va traduire l'adresse en une valeur qui, combinée avec l'adresse non traduite, va pointer vers une adresse de l'espace utilisateur de QEMU.

I.2.4.2 Processeur, régions mémoires et disque dur

Nous allons maintenant voir comment QEMU émule trois des composants les plus importants d'une machine, le processeur, la mémoire vive et un NVME.

Processeur

En émulation, le processeur émulé est TCG qui va traduire les instructions de l'invité vers des instructions que l'hôte comprend. Lors de la virtualisation, c'est l'hyperviseur qui fait office de processeur du point de vue de l'invité.

Régions mémoires

Les régions mémoires sont modélisées par l'API mémoire de QEMU [13]. Cette API permet de représenter différents types de régions mémoires, elles sont représentées par une unique classe C `MemoryRegion`. Les régions mémoires modélisées sont les suivantes :

- Mémoire vive (RAM) : Plage de mémoire de l'hôte reservée pour l'invité, `memory_region_init_ram()`.
- MMIO (*Memory-mapped I/O*) : Plage de mémoire invité, elle est gérée par l'hôte, chaque écriture ou lecture fait appel à l'hôte, `memory_region_init_io()`
- ROM (*Read Only Memory*) : En lecture, l'API accède directement à une région de la mémoire de l'hôte. En écriture, elle utilise le même principe que pour la région MMIO, `memory_region_init_rom_device()`
- IOMMU (*Input–Output Memory Management Unit*) : Traduit l'adresse des accès réalisés pour cette région vers une autre région, `memory_region_init_iommu()`

La modélisation de la mémoire est sous forme de graphe acyclique de `MemoryRegion`, les feuilles correspondent à la RAM et MMIO tandis que le reste des nœuds correspond aux bus, contrôleurs mémoires et aux autres régions mémoires.

Disque dur

Par défaut, QEMU utilise le format de fichier `qcow2` [14] pour ses images de disques, `qcow` signifie *QEMU Copy On Write*. Comme son nom l'indique, ce type de fichier utilise l'allocation paresseuse afin de n'utiliser le stockage que si c'est nécessaire. On peut générer une image de ce type de disque à l'aide de la commande `qemu-img`.

Pour accéder au stockage, qui ne constitue qu'une représentation logique d'un espace de stockage physique de l'hôte, QEMU recourt à des contrôleurs de mémoire qui, à leur tour, exploitent des *Block Devices* [15] facilitant l'écriture et la lecture de données, ainsi que le transfert de données vers le stockage permanent, entre autres.

Nous avons présenté un aperçu du fonctionnement interne de QEMU, ainsi que des possibilités offertes par ce logiciel. Dans la prochaine partie, nous allons voir une liste de techniques d'anti-émulation qui peuvent être utilisées

pour détecter une machine virtuelle. La plupart de ces techniques se concentrent sur une machine Windows que nous avons générée à l'aide de QEMU.

Deuxième partie

Techniques d'anti-virtualisation/anti-émulation

Chapitre II.1

API Windows

L'API Windows [16] offre un grand nombre de fonctions permettant d'interagir de manière efficace avec le système. Beaucoup de ces fonctions sont utilisables pour détecter si un système est virtualisé. Nous pouvons soit examiner le matériel qui est émulé, soit les logiciels qui sont en train de tourner au sein de la machine virtuelle.

Note importante

Les techniques présentées ne sont pas uniques, il existe de nombreuses autres façons de réaliser chacune des techniques. De plus, les extraits de code présentés sont des fonctions qui retournent `true` si une machine virtuelle est détectée.

II.1.1 Interaction matérielle

Comme expliqué dans le Chapitre I.1, soit la machine virtuelle émule les composants de la machine hôte, soit elle fournit de nouveaux composants virtuels en se servant des ressources de l'hôte. Pour chaque technique présentée, nous allons voir pourquoi cette technique fonctionne dans un contexte d'anti-virtualisation et comment un maliciel peut l'utiliser pour détecter une machine virtuelle.

II.1.1.1 Résolution de l'écran

La première technique concerne la résolution de l'écran. Elle repose sur le principe suivant : la majorité des écrans utilisés de nos jours possèdent une résolution qui descend rarement en dessous d'un certain seuil. Cependant, lors de la création d'une machine virtuelle, ou plus précisément d'un environnement de test, il n'est généralement pas nécessaire de rendre l'environnement particulièrement complet. ous nous limitons souvent à une configuration basique : cela inclut une résolution d'écran par défaut de 1024×768 dans QEMU (voire inférieure).

Nous cherchons donc à récupérer la résolution et à la comparer avec une taille standard. Pour cela, nous allons utiliser, pour Windows, la fonction `GetSystemMetrics`. Cette fonction va retrouver la mesure ou le paramètre de configuration demandé, voici la syntaxe :

```
int GetSystemMetrics(
  [in] int nIndex
);
```

Le paramètre `nIndex` peut prendre un grand nombre de valeurs, chacune de ces valeurs entières correspond à un paramètre. Parmi celles-là, seulement deux nous intéressent [17] :

- `SM_CXSCREEN` (0) : La largeur de l'écran en pixels
- `SM_CYSCREEN` (1) : La hauteur de l'écran en pixels

Il ne reste plus qu'à les comparer avec une résolution standard, dans notre cas, nous allons prendre **1024×768**.

Ci-dessous, une fonction en C qui montre comment utiliser cette technique pour Windows :

```
#include <windows.h>
#include <stdio.h>

bool check_RESOLUTION()
{
    int length = GetSystemMetrics(SM_CXSCREEN);
    int height = GetSystemMetrics(SM_CYSCREEN);

    return length <= 1024 || height <= 768;
}
```

II.1.1.2 Taille du stockage

Cette technique est identique en principe à la précédente, à savoir l'exploitation de caractéristiques matérielles courantes. Toutefois, au lieu de s'intéresser à la résolution de l'écran, nous allons cette fois porter notre attention sur la capacité de stockage de la machine.

La taille des disques durs a grandement augmenté depuis les premiers modèles. Dans les années 90, la taille d'un disque dur dépassait rarement les 20 Go. Ce n'est que depuis assez récemment que l'on trouve des disques durs ayant des tailles allant au-delà de 1 To. On peut voir grâce au schéma ci-dessous cette évolution :

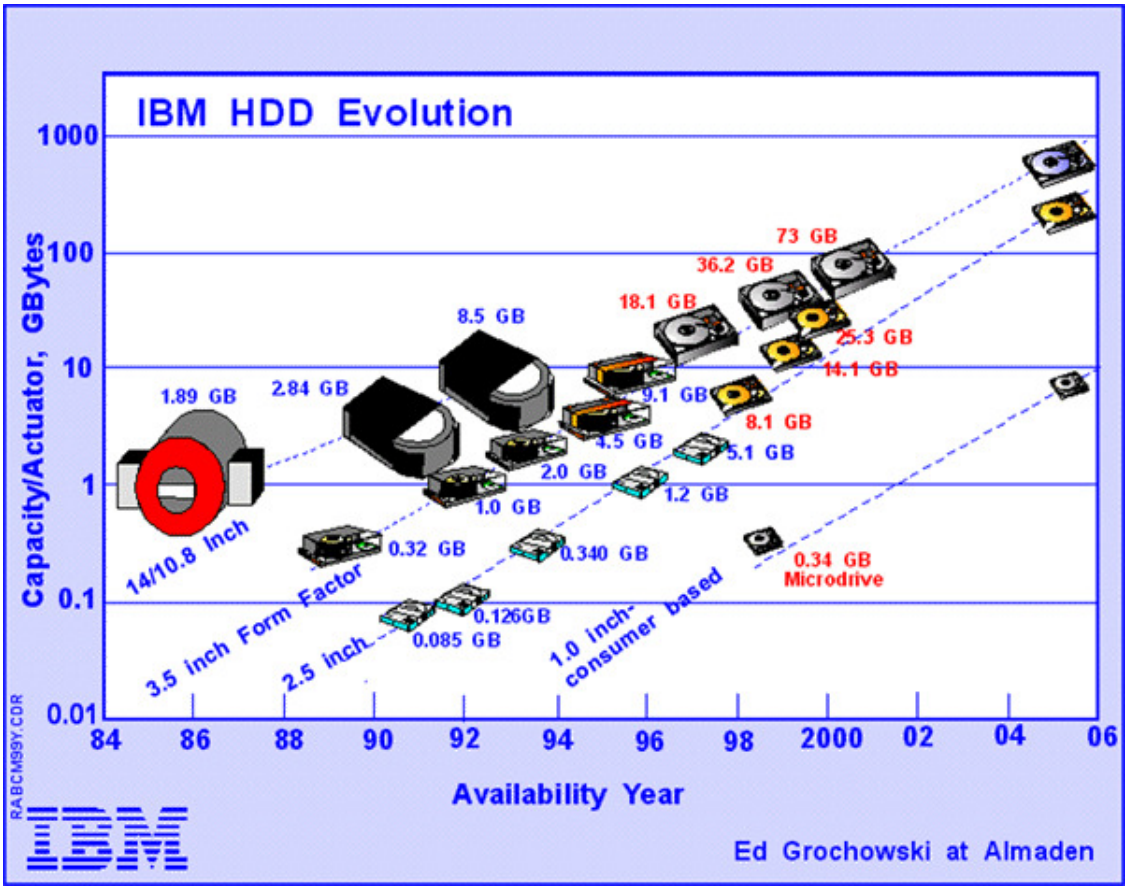

FIGURE II.1.1 – Évolution de la taille des disques durs [18]

Tout comme la résolution de l'écran, de nos jours, il est rare de trouver une machine ayant une taille de stockage "petite". Cette propriété donne lieu à deux techniques :

- Tester la taille totale du stockage du disque dur
- Tester la taille de l'espace restant du stockage du disque dur

Un logiciel peut détecter s'il est utilisé dans une machine virtuelle en s'aidant de cela. En effet, lors de la création d'une machine virtuelle, la taille de son disque virtuel est limitée par le disque dur que nous possédons [1]. Cela signifie que la capacité totale du disque dur ainsi que celle de l'espace restant du stockage auront des valeurs inférieures à ce que l'on peut trouver dans des machines physiques.

Pour utiliser cela, nous allons avoir besoin de la fonction `GetDiskFreeSpaceExA` de l'API Windows [19] :

```
BOOL GetDiskFreeSpaceExA(
  [in, optional]  LPCSTR          lpDirectoryName,
  [out, optional] PULARGE_INTEGER lpFreeBytesAvailableToCaller,
  [out, optional] PULARGE_INTEGER lpTotalNumberOfBytes,
  [out, optional] PULARGE_INTEGER lpTotalNumberOfFreeBytes
);
```

Cette fonction va aller chercher la taille totale (`lpTotalNumberOfBytes`) et l'espace restant (`lpTotalNumberOfFreeBytes`) d'un disque dur désigné par son nom (`lpDirectoryName`). Il suffit donc de comparer `lpTotalNumberOfBytes` et `lpTotalNumberOfFreeBytes` avec des valeurs de taille (et d'espace restant) afin de déterminer si l'on est dans une machine virtuelle. Un exemple ci-dessous sous la forme d'un code C :

Cette fonction va aller chercher la taille totale (`lpTotalNumberOfBytes`) et l'espace restant (`lpTotalNumberOfFreeBytes`) d'un disque dur désigné par son nom (`lpDirectoryName`). Il suffit donc de comparer `lpTotalNumberOfBytes` et `lpTotalNumberOfFreeBytes` avec des valeurs de taille (et d'espace restant) afin de déterminer si l'on est dans une machine virtuelle. Un exemple est présenté ci-dessous sous la forme d'un code C :

```c
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>

bool check_HARD_DRIVE_SIZE()
{
    ULARGE_INTEGER freeBytesAvailable, totalBytes, totalFreeBytes;
    ULARGE_INTEGER freeBytesAvailable, totalBytes, totalFreeBytes;
    GetDiskFreeSpaceExA("C:\\", &freeBytesAvailable, &totalBytes, &totalFreeBytes);

    /* 214748364800 --> 200 GB || 32212254720 --> 30 GB */
    return totalBytes.QuadPart < 214748364800 || freeBytesAvailable.QuadPart <
32212254720;
}
```

II.1.1.3 Taille de la mémoire

Identique en principe à la technique précédente, celle-ci va se concentrer sur la mémoire vive (*RAM*) disponible. Comme expliqué précédemment, une machine virtuelle a ses caractéristiques limitées par celles de la machine hôte ; par exemple, la mémoire vive allouée à l'invité ne peut pas dépasser celle disponible sur l'hôte.

De plus, tout comme la capacité de stockage des disques durs, la quantité de mémoire disponible a, elle aussi, augmenté considérablement. Ainsi, si la taille de la mémoire vive est trop petite (inférieure à 4 Go), on peut en déduire, soit que l'on est dans une machine ancienne, soit dans une machine virtuelle qui est limitée par son hôte. Pour récupérer la taille de la mémoire vive, nous allons utiliser la fonction `GlobalMemoryStatusEx` de l'API Windows [20] :

```
BOOL GlobalMemoryStatusEx(
  [in, out] LPMEMORYSTATUSEX lpBuffer
);
```

Cette fonction va remplir une structure `MEMORYSTATUSEX` qui comprend un champ `ullTotalPhys` qui représente le nombre total d'octets dans la mémoire vive. On peut donc implémenter la technique décrite ci-dessus comme suit :

```c
#include <windows.h>
#include <stdio.h>
```

1. Il est possible de contrer cela en utilisant la technique de *thin provisioning*, qui permet d'allouer plus de mémoire qu'il n'en existe physiquement.

```
bool check_MEMORY()
{
    MEMORYSTATUSEX memInfo;
    memInfo.dwLength = sizeof(MEMORYSTATUSEX);
    GlobalMemoryStatusEx(&memInfo);

    return memInfo.ullAvailPhys / 1024 / 1024 / 1024 <= 4;
}
```

II.1.1.4 Adresse MAC

La carte réseau est un composant essentiel de chaque ordinateur. De plus, chacune de ces cartes a un identifiant qui lui est unique, l'adresse MAC. Cette adresse MAC est une adresse de 6 octets, représentée sous forme hexadécimale (exemple : `12:34:56:78:9a:bc`). Elle se compose de deux parties [2] :

- *Organisationally Unique Identifier (OUI)* (trois premiers octets) : contient l'identifiant d'un constructeur de *NIC*.
- *Network Interface Controller (NIC) Specific* (trois derniers octets) : contient l'identifiant unique de la carte afin de la distinguer des autres cartes du même constructeur.

Cette adresse est un identifiant physique unique à chaque carte sur chaque ordinateur physique ; cependant, c'est aussi le cas pour les machines virtuelles.

En effet, lors de la création d'une machine virtuelle, le gestionnaire de machine virtuelle (QEMU, VirtualBox, VMWare, etc.), la machine utilisera une interface réseau fournie par le gestionnaire. Ce dernier va donc lui fournir une carte réseau émulée. Afin de respecter l'unicité de l'adresse MAC, chaque gestionnaire de machine virtuelle a son propre identifiant de constructeur (*NIC Specific*). On a par exemple :

- Pour VirtualBox : `08:00:27`
- Pour VMWare : `00:0C:29`
- Pour QEMU/KVM : `52:54:00`

Nous pouvons utiliser cette propriété afin de détecter la présence d'une machine virtuelle. Pour cela, nous allons avoir besoin d'accéder à l'adresse MAC de la machine et de la comparer avec différentes adresses MAC standard. Voyons comment implémenter cette technique pour une machine Windows.

Nous allons utiliser la fonction `GetAdaptersInfo` provenant de la librairie `iphlpapi.h`, elle est décrite comme suit par la documentation **Microsoft** :

> *La fonction GetAdaptersInfo récupère les informations de l'adaptateur pour l'ordinateur local.*

Et sa syntaxe est la suivante [21] :

```
IPHLPAPI_DLL_LINKAGE ULONG GetAdaptersInfo(
  [out]     PIP_ADAPTER_INFO AdapterInfo,
  [in, out] PULONG           SizePointer
);
```

Cette fonction va remplir une structure `PIP_ADAPTER_INFO` d'informations sur la carte réseau. Dans cette structure, nous allons nous intéresser au champ `Address` :

```
typedef struct _IP_ADAPTER_INFO {
  struct _IP_ADAPTER_INFO *Next;
  DWORD                   ComboIndex;
  char                    AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
  char                    Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
  UINT                    AddressLength;
  BYTE                    Address[MAX_ADAPTER_ADDRESS_LENGTH];
...
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

2. En réalité, on peut dire que l'adresse MAC comporte 4 parties plutôt que 2, cependant ces deux parties supplémentaires, qui sont le bit montrant si l'adresse est unique et le bit montrant si l'adresse est locale, peuvent être regroupées avec une autre partie de l'adresse (*OUI*).

Nous allons donc implémenter la technique de la manière suivante :

- On récupère la liste des interfaces réseau présentes (`GetAdaptersInfo`)
- Pour chaque interface, on formate l'adresse trouvée en une chaîne de caractères de la forme `XX:XX:XX:XX:XX:XX`
- Et enfin, on teste si l'adresse de la machine ciblée contient une adresse MAC standard pour une machine virtuelle.

Ci-dessous, une implémentation possible de cette technique :

```
#include <windows.h>
#include <iphlpapi.h>
#include <stdio.h>
#include <string.h>

#define MAX_MAC_ADDRESSES 16
static char *recurrent_mac[6] = {"08:00:27", "00:0C:29", "00:1C:14", "00:50:56",
"00:05:69", "52:54:00"};

bool check_MAC()
{
    IP_ADAPTER_INFO adapter_info[MAX_MAC_ADDRESSES];
    DWORD dwBufLen = sizeof(adapter_info);

    DWORD dwStatus = GetAdaptersInfo(adapter_info, &dwBufLen);
    if (dwStatus != ERROR_SUCCESS)
    {
        return false;
    }

    PIP_ADAPTER_INFO pAdapter = adapter_info;

    while (pAdapter)
    {
        char mac_address[18];
        sprintf(mac_address, "%02X:%02X:%02X:%02X:%02X:%02X", pAdapter->Address[0],
pAdapter->Address[1],
                pAdapter->Address[2], pAdapter->Address[3], pAdapter->Address[4],
pAdapter->Address[5]);
        for (int i = 0; i < sizeof(recurrent_mac) / sizeof(recurrent_mac[0]); i++)
        {
            if (strstr(mac_address, recurrent_mac[i]) != NULL)
            {
                return true;
            }
        }
        pAdapter = pAdapter->Next;
    }
    return false;
}
```

II.1.1.5 Nombre de CPU

Un ordinateur moderne utilise (dans la plupart des cas) plus d'un processeur logique, et assez souvent plus de 4. Cependant, lors de la création d'une machine virtuelle, il est possible d'en utiliser moins. Le but d'une machine virtuelle n'est pas forcément d'être la plus performante possible.

Cette technique se base sur ce principe, pour l'appliquer, nous allons récupérer le nombre de processeurs logiques et le comparer à une valeur arbitraire (quatre dans notre cas). Pour ce faire, nous allons utiliser la fonction `GetSystemInfo` de l'API Windows [22]. Cette fonction va remplir une structure `SYSTEM_INFO` définie comme suit :

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
```

```
      WORD wProcessorArchitecture;
      WORD wReserved;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

Dans cette structure, nous allons utiliser le champ `dwNumberOfProcessors` qui représente le nombre de processeurs logiques d'un système. Il suffit de comparer ce champ avec la valeur arbitraire choisie. Ci-dessous, un exemple d'implémentation :

```
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>


bool check_NB_CPU()
{
    SYSTEM_INFO info;
    GetSystemInfo(&info);
    int nb_cpu = info.dwNumberOfProcessors;
    return nb_cpu < 4;
}
```

II.1.1.6 Fonctionnalités d'alimentation

L'API de Windows permet à un utilisateur de récupérer la plupart des informations du système. Il est notamment possible d'obtenir les informations concernant les fonctionnalités d'alimentation du système, ces fonctionnalités sont conformes à la spécification *ACPI* (*Advanced Configuration and Power Interface*).

Pour récupérer ces informations, nous allons utiliser la fonction `GetPwrCapabilities` [23] qui va remplir une structure `SYSTEM_POWER_CAPABILITIES` contenant des informations sur les fonctionnalités d'alimentation du système :

```
typedef struct {
... /* rest of the field */
  BOOLEAN                 SystemS1;
  BOOLEAN                 SystemS2;
  BOOLEAN                 SystemS3;
  BOOLEAN                 SystemS4;
  BOOLEAN                 SystemS5;
... /* rest of the field */
  BOOLEAN                 ThermalControl;
  BOOLEAN                 ProcessorThrottle;
  BYTE                    ProcessorMinThrottle;
  BYTE                    ProcessorThrottleScale;
... /* rest of the field */
} SYSTEM_POWER_CAPABILITIES, *PSYSTEM_POWER_CAPABILITIES;
```

On peut utiliser cette structure afin d'implémenter deux techniques d'anti-émulation, le contrôle de la température des composants et les états de veille.

Contrôle de la température

Un système Windows va gérer la température de ses composants en utilisant des zones thermiques. Une zone thermique est une partition logique qui englobe un ou plusieurs composants. Ces zones sont représentées dans le matériel par des capteurs qui vont récupérer la chaleur et la transmettre au système, qui les reliera avec les zones thermiques. Lorsqu'un composant va surchauffer, le capteur va transmettre la température au système qui va la faire correspondre à une zone thermique. Cela permet au système de correctement agir afin de refroidir le composant en question.

Cette technique va utiliser le champ `ThermalControl` qui va être utilisé par le système pour savoir si le système est capable de contrôler la température des composants. Si c'est le cas, alors le champ sera mis à `True` ; sinon il sera mis à `False`.

Une machine physique est obligée de contrôler la température de ses composants, sinon, elle risque de surchauffer et de s'endommager. À l'inverse, une machine virtuelle n'a pas cette obligation, en effet, même en émulation (ou en virtualisation complète), le système ne simule qu'une version "virtuelle" des composants. Ces composants "virtualisés" (ou émulés) ne peuvent donc pas surchauffer et de ce fait ne peuvent pas être reliés à des zones thermiques.

Ainsi, pour cette technique, il suffit de savoir si le champ `ThermalControl` est présent, dans ce cas, nous sommes dans une machine physique ou non. En voici une implémentation :

```
#include <windows.h>
#include <powrprof.h>
#include <stdbool.h>


bool check_THERMAL_CONTROL()
{
    SYSTEM_POWER_CAPABILITIES power_caps;

    if (GetPwrCapabilities(&power_caps) == true)
    {
        return !power_caps.ThermalControl;
    }
    return false;
}
```

États de veille

Dans un système, il existe plusieurs états de veille définis comme états APCI de S0 à S5 avec certaines variantes :

- S0 (Travaillant) : Le système utilisable par l'utilisateur (affichage graphique, interactions, etc)
- S0 inactif (Veille) : Équivalent à un état S0 mais inactif à faible puissance, cependant il est capable de s'activer rapidement et sortir de l'état inactif.
- S1 à S3 (Dormant) : Le système semble désactivé et consomme moins d'énergie que S0, seulement un seul de ces trois états peut être activé dans un système.
- S4 (Hibernation) : Le système paraît désactivé, la consommation d'énergie est minimale. Le système sauvegarde le contenu de la mémoire dans un fichier.
- S5 (*Soft off*) : Le système semble désactivé, c'est un arrêt complet qui va redémarrer la session utilisateur à chaque redémarrage.

Ces modes de veille sont standardisés pour des machines physiques. Dans le cas d'une machine virtuelle, certains modes de veille peuvent être absents. L'objectif de cette technique va donc être de récupérer des informations sur ces modes de veille et de regarder s'ils sont présents dans le système ciblé.

Dans la structure `SYSTEM_POWER_CAPABILITIES`, il existe cinq champs, `SystemS1` à `SystemS5` qui correspondent aux cinq modes de veille décrits ci-dessus. Nous allons donc tester chacun de ces modes de veilles et si aucun n'est présent, voici une implémentation possible pour cette technique :

```
#include <windows.h>
#include <powrprof.h>
#include <stdbool.h>
```

```
bool check_SLEEP_MODES()
{
    SYSTEM_POWER_CAPABILITIES power_caps;

    if (GetPwrCapabilities(&power_caps) == true)
    {
        return !(power_caps.SystemS1 || power_caps.SystemS2 || power_caps.SystemS3
|| power_caps.SystemS4);
    }
    return false;
}
```

II.1.2 Interaction des artéfacts

Afin de correctement virtualiser un système, l'invité peut faire appel à des périphériques de virtualisation (par exemple *Virtiofs*). Ces périphériques peuvent avoir des services qui leur sont attribués et apparaîtront donc dans la liste des services actifs de Windows. De plus, dans Windows, il existe aussi des clefs de registre (`RegKeys`) attribuées à des extensions de virtualisations ou à des gestionnaires de machines virtuelles.

II.1.2.1 Services

Comme expliqué, dans une machine virtuelle, il peut y avoir des services qui signalent la présence d'un environnement émulé (virtualisé). Par exemple, lorsque l'on crée un dossier partagé ou qu'on choisit de modifier la résolution de l'écran de la machine virtuelle, on ajoute des services supplémentaires. Par exemple, pour les dossiers partagés, on peut utiliser `Virtiofs` qui créera un service actif nommé `VirtioFsSvc`. On peut aussi prendre l'exemple d'une machine créée par VirtualBox et qui aura donc des services qui contiendront la chaîne de caractères `VBox`.

Pour réaliser cette technique, nous allons devoir construire un tableau de chaîne de caractères contenant des noms possibles pour les services recherchés. Pour lister les services actifs, nous allons utiliser des fonctions de l'API Windows :

- `OpenSCManager` : permet de se connecter au gestionnaire de contrôle de service en fournissant un *handle*.
- `EnumServicesStatusA` : énumère les services dans le gestionnaire de contrôle de service, nécessite l'utilisation d'un *handle* de connexion au gestionnaire fourni par `OpenSCManager`.

Tout d'abord, nous allons récupérer le *handle*, voici la syntaxe de `OpenSCManager` [24] :

```
SC_HANDLE OpenSCManagerA(
  [in, optional] LPCSTR lpMachineName,
  [in, optional] LPCSTR lpDatabaseName,
  [in]           DWORD  dwDesiredAccess
);
```

Nous allons récupérer le *handle* sans passer en argument le nom de la machine (`lpMachineName`) et le nom de la base de donnée du gestionnaire de service (`lpDatabaseName`).

Nous allons cependant préciser le dernier argument qui lui est nécessaire, à savoir le droit d'accès au gestionnaire de services. Comme nous souhaitons énumérer les services, nous allons utiliser le droit `SC_MANAGER_ENUMERATE_SERVICE`, nécessaire pour appeler les fonctions `EnumServicesStatus` ou `EnumServicesStatusEx` [25] qui permettent de lister les services.

La syntaxe de `EnumServicesStatus` [26] est la suivante :

```
BOOL EnumServicesStatusA(
  [in]                SC_HANDLE              hSCManager,
  [in]                DWORD                  dwServiceType,
  [in]                DWORD                  dwServiceState,
  [out, optional]     LPENUM_SERVICE_STATUSA lpServices,
  [in]                DWORD                  cbBufSize,
```

```
  [out]               LPDWORD             pcbBytesNeeded,
  [out]               LPDWORD             lpServicesReturned,
  [in, out, optional] LPDWORD             lpResumeHandle
);
```

La fonction `EnumServicesStatusA` va remplir une structure `LPENUM_SERVICE_STATUSA` qui est un tableau d'éléments contenant des informations concernant chaque service telles que leur nom. Voici sa syntaxe :

```
typedef struct _ENUM_SERVICE_STATUSA {
  LPSTR          lpServiceName;
  LPSTR          lpDisplayName;
  SERVICE_STATUS ServiceStatus;
} ENUM_SERVICE_STATUSA, *LPENUM_SERVICE_STATUSA;
```

Il suffit par la suite de comparer le nom de chaque service trouvé avec notre liste de noms de service prédéfinie. Si on trouve une correspondance, on peut alors conclure que nous sommes dans une machine virtuelle.

Voici une implémentation possible :

```
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#define NB_SERVICES 15

/* Definition of belong_to and SERVICES_NAME_LIST */
...


bool check_SERVICES()
{
    SC_HANDLE hSCManager = OpenSCManager(NULL, NULL, SC_MANAGER_ENUMERATE_SERVICE);
    if (hSCManager == NULL)
    {
        printf("OpenSCManager failed! Error: %lu\n", GetLastError());
        return false;
    }

    ENUM_SERVICE_STATUS serviceStatus[1024];
    DWORD bytesNeeded = 0, servicesReturned = 0, resumeHandle = 0;

    if (!EnumServicesStatus(hSCManager, SERVICE_WIN32, SERVICE_STATE_ALL,
serviceStatus, sizeof(serviceStatus),
                            &bytesNeeded, &servicesReturned, &resumeHandle))
    {
        printf("EnumServicesStatus failed! Error: %lu\n", GetLastError());
        CloseServiceHandle(hSCManager);
        return false;
    }

    for (DWORD i = 0; i < servicesReturned; i++)
    {
        if (belong_to(serviceStatus[i].lpServiceName))
            return true;
    }

    CloseServiceHandle(hSCManager);

    return false;
}
```

La fonction `belong_to` et le tableau `SERVICES_NAME_LIST` peuvent être trouvés en Annexe 2.

II.1.2.2 `RegKey`

Avant d'expliquer la technique d'anti-émulation, il est bon de rappeler ce qu'est une `RegKey`, diminutif de *Registry Key*, ou *Clef de Registre*. Une `RegKey` est une entrée du registre Windows qui comprend toutes sortes d'informations sur le système. Ces clefs comportent trois champs, un nom, un type et les données correspondant à la clef. Les données diffèrent selon la clef. Cela peut être une valeur hexadécimale ou une chaîne de caractères représentant un chemin, une description, etc. Ce registre va donc contenir un tas de clefs concernant certaines applications présentes dans le système.

Lorsque QEMU génère une machine virtuelle, il va installer dans l'invité un *daemon* nommé *QEMU Guest Agent*. Ce logiciel est utilisé pour permettre à l'invité de communiquer entre l'hôte et l'invité. Faisant partie du système, ce logiciel aura une `RegKey` qui lui sera attribuée, on peut la retrouver au chemin suivant dans l'éditeur de registre :

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\QEMU-GA
```

Chaque gestionnaire de machines virtuelles possède son propre logiciel qui va permettre ces échanges hôte-invité, il est donc possible de les détecter en s'aidant des clefs de registre. Cette technique se base sur la possibilité de récupérer une clef de registre. Nous allons essayer d'accéder à une clef de registre. Si c'est possible, alors cela signifie qu'un logiciel lié à de la virtualisation (ou à l'émulation) existe.

Pour réaliser cette technique, nous allons utiliser la fonction `RegOpenKeyExW` de l'API Windows [27], cette fonction a la syntaxe suivante :

```
LSTATUS RegOpenKeyExW(
  [in]           HKEY    hKey,
  [in, optional] LPCWSTR lpSubKey,
  [in]           DWORD   ulOptions,
  [in]           REGSAM  samDesired,
  [out]          PHKEY   phkResult
);
```

Le paramètre `lpSubKey` est le chemin de la clé de registre à laquelle on souhaite accéder. Nous allons donc créer un tableau de chemins de clés de registre qui indiquent la présence de logiciels de virtualisation et accéder à chacune de ces clés. Si l'on réussit à accéder à l'une d'elles, alors on peut en conclure que l'on est dans une machine virtuelle. En voici une implémentation, la liste des clés de registre testées peut être trouvée en Annexe 3 :

```
#include <stdbool.h>
#include <stdio.h>
#include <windows.h>

...

bool check_REGS() {
  HKEY regkey;
  DWORD i;
  char value[1024];
  DWORD size;
  DWORD type;

  bool result = false;

  for (i = 0; i < sizeof(RegValuePath) / sizeof(RegValuePath[0]); i++) {
    if (RegOpenKeyExW(HKEY_LOCAL_MACHINE, RegValuePath[i], 0, KEY_READ,
                      &regkey) == ERROR_SUCCESS) {

                        result = true;
    }

  }
  RegCloseKey(regkey);
  return result;
}
```

II.1.3 Interaction humaine

La manière la plus évidente de détecter une machine virtuelle est de repérer toute trace de comportement inhabituel pour une machine physique. Cependant, il est aussi possible de regarder les réactions de la machine en fonction de l'activité de son utilisateur. En effet, dans la majorité des cas, un utilisateur effectue des actions attendues, comme bouger la souris, interagir avec les fenêtres, utiliser le clavier, etc.

Dans cette section, nous allons présenter deux techniques basées sur ce principe.

II.1.3.1 Mouvement de la souris

Dans certains environnements d'analyse, l'étude d'un exécutable peut se faire par un enregistrement du flot d'exécution. Lors de cet enregistrement, il est courant que l'interface graphique se fige et ne permette plus à l'utilisateur d'interagir avec elle. Cela permet alors d'éliminer dans l'enregistrement les instructions superflues, telles que celles servant à déplacer la souris.

Cependant, un utilisateur déplacera régulièrement la souris de manière plus ou moins volontaire, peu importe le contexte. Cette technique exploite ce comportement pour déterminer si le code s'exécute dans un environnement virtuel. Pour cela, nous allons utiliser une fonction de l'API Windows : `GetCursorPos` [28]. Cette fonction a la syntaxe suivante :

```
BOOL GetCursorPos(
  [out] LPPOINT lpPoint
);
```

Cette fonction remplit une structure `POINT` qui a deux champs :

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT, *NPPOINT, *LPPOINT;
```

La structure `POINT` récupérée correspond à la position du curseur de la souris à l'écran.

Nous allons donc devoir comparer la position du curseur de la souris à deux intervalles de temps différents. Pour cela, il suffit d'appeler une première fois `GetCursorPos`, de laisser passer un certain temps (avec `Sleep` par exemple) puis `GetCursorPos`. Le code suivant représente une implémentation possible de cette technique :

```
#include <stdbool.h>
#include <stdio.h>
#include <windows.h>

bool check_MOUSE() {
  POINT start, end;

  GetCursorPos(&start);
  Sleep(8000);
  GetCursorPos(&end);

  return start.x == end.x && start.y == end.y;
}
```

II.1.3.2 Changement de fenêtre

Comme évoqué précédemment, dans certains environnements d'analyse, l'interface graphique peut se figer. Cela signifie que la fenêtre actuellement au premier plan ne changera pas pendant un certain laps de temps. Nous pouvons exploiter ce comportement en appelant la fonction `GetForegroundWindow` [29] à deux reprises, espacées par un appel à `Sleep`. Si les deux fenêtres récupérées sont identiques, alors nous pouvons en déduire que l'interface est restée figée, ce qui est typique de certains environnements d'analyse.

La fonction `GetForegroundWindow` est décrite comme suit :

Récupère un descripteur dans la fenêtre de premier plan (la fenêtre avec laquelle l'utilisateur travaille actuellement).

Nous allons donc récupérer un premier descripteur à l'aide de la fonction `GetForegroundWindow`, puis attendre un peu avec la fonction `Sleep`. On récupère un deuxième descripteur et enfin, on compare les deux descripteurs. S'ils sont différents, alors il est peu probable que ce soit dans un environnement d'analyse. Ci-dessous, une implémentation possible de cette technique :

```
#include <windows.h>
#include <stdbool.h>
#include <stdio.h>


bool check_FOREGROUND()
{
    HWND foregroundHandle1 = GetForegroundWindow();
    Sleep(2500);
    HWND foregroundHandle2 = GetForegroundWindow();

    return foregroundHandle1 == foregroundHandle2;
}
```

Chapitre II.2

Instruction machine

Nous avons vu certaines techniques d'anti-émulation basées sur l'API Windows, qui ne fonctionneront que sur un système Windows. Nous allons maintenant présenter quelques techniques basées sur des instructions assembleur. Cela permet (en partie) de rendre ces techniques interopérables entre différents systèmes d'exploitation [1].

II.2.1 `cpuid`

L'instruction `cpuid` est une instruction assembleur introduite par Intel en 1993 [2]. Cette instruction est utilisée pour récupérer des informations sur le processeur. Pour accéder à ces informations, il est nécessaire d'accéder à une *feuille* (*leaf*) de l'instruction, c'est-à-dire un comportement particulier de l'instruction selon une entrée.

Dans le cas de `cpuid`, l'instruction prend en entrée le registre `eax` [3] et retourne dans `eax`, `ebx`, `ecx` et `edx` les informations demandées. La page Wikipédia de `cpuid` détaille précisément pour chaque entrée (valeur de `eax`) quelles sont les informations qui vont être retournées et dans quels registres. Le schéma ci-dessous résume l'utilisation de cette instruction :

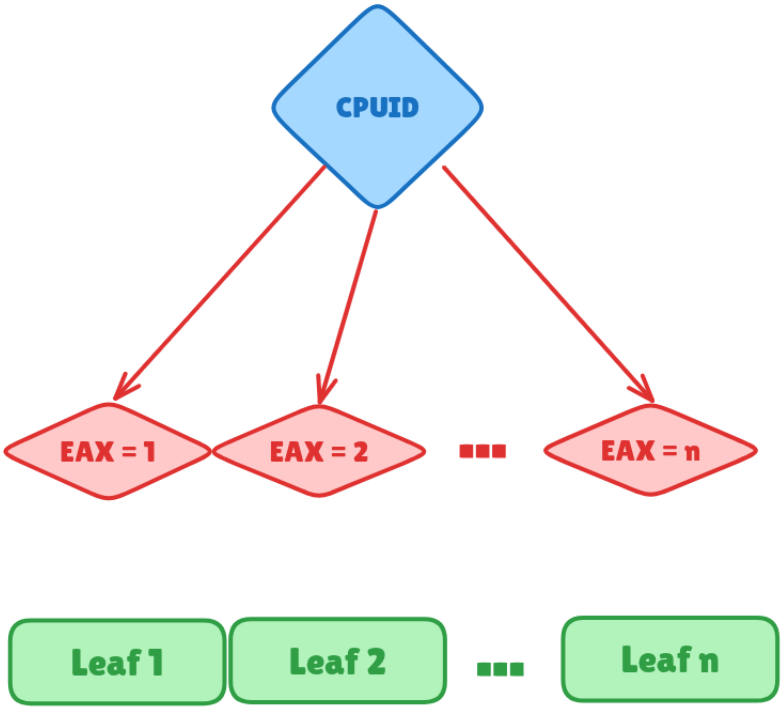

FIGURE II.2.1 – Résumé de l'utilisation de l'instruction `cpuid`

1. Il faut cependant prendre soin d'adapter chacune de ces instructions selon l'architecture du système ciblé, les conventions d'appels peuvent changer et certaines de ces instructions sont communes à des processeurs récents.
2. La plupart des processeurs modernes, qu'ils soient Intel ou pas, implémentent cette instruction.
3. L'instruction `cpuid` ne prend que les registres de 32 bits, même dans une architecture 64 bits.

Cette instruction peut être utilisée par tous les utilisateurs, c'est donc un excellent moyen pour un maliciel de savoir s'il est virtualisé. Il existe plusieurs techniques découlant de cette instruction.

II.2.1.1 Bit d'hyperviseur

La première technique consiste à détecter la présence d'un hyperviseur. Pour cela, nous allons interroger le processeur via l'instruction `cpuid`, en utilisant la leaf `0x1`. Si nous sommes dans une machine virtuelle, l'instruction va récupérer la valeur 1 depuis le bit d'hyperviseur dans `ecx`, plus particulièrement le bit 31 du registre. À l'inverse, si nous ne sommes pas dans une machine virtuelle, alors le bit 31 du registre `ecx` sera égal à 0, il suffit donc de comparer la valeur retrouvée et renvoyer `true` si elle vaut "1" et `false` sinon. Ci-dessous un exemple d'implémentation en C et assembleur :

```
#include <stdio.h>
#include <stdbool.h>

bool check_HYPERVISOR_BIT()
{
    bool isVM = false;

    __asm__ volatile (
        "mov $1, %%eax\n\t"
        "cpuid\n\t"
        "bt $31, %%ecx\n\t"
        "setc %0\n\t"
        : "=r"(isVM)
        :
        : "eax", "ebx", "ecx", "edx", "cc"
    );

    return isVM;
}
```

II.2.1.2 Chaîne d'identifiant de l'hyperviseur

La seconde technique consiste à récupérer la chaîne d'identifiant de l'hyperviseur. Cet élément est une chaîne de caractères qui permet d'authentifier l'hyperviseur utilisé. Lors de l'utilisation de l'instruction `cpuid` dans une machine virtuelle, l'hyperviseur va intercepter la demande afin de retourner ses informations et non les informations du processeur physique. Cependant, ces informations comportent des indices sur la présence de ce même hyperviseur, notamment la "marque" de l'hyperviseur.

Cette "marque", aussi appelée chaîne d'identifiant de l'hyperviseur, est une chaîne de 12 caractères représentant quel gestionnaire de machines virtuelles fournit cet hyperviseur. Par exemple, la chaîne d'identifiant de l'hyperviseur de :

- VirtualBox est `VBoxVBoxVBox`
- VMWare est `VMwareVMware`
- QEMU est `TCGTCGTCGTCG`

Ceux-ci ne sont que des exemples, il en existe plein d'autres.

Dans une machine physique, il n'y a pas d'hyperviseur, la chaîne d'identifiant de l'hyperviseur sera donc vide. Pour retrouver cette chaîne, il faut utiliser l'instruction `cpuid` avec `eax = 0x40000000`, ce qui renverra les quatre premiers caractères dans `ecx`, les quatre suivants dans `ebx` et les quatre derniers dans `edx`. Il suffit ensuite de tester si la chaîne est vide ou si elle correspond à un identifiant connu. On peut l'implémenter comme suit en C :

```
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#define NB_BRAND_STRING 6
```

```
const char *known_hypervisor_id[] = {
    "KVMKVMKVM",    /* KVM */
    "Microsoft Hv", /* Hyper-V */
    "VMwareVMware", /* VMware */
    "XenVMMXenVMM", /* Xen */
    "VBoxVBoxVBox", /* VirtualBox */
    "TCGTCGTCGTCG"  /* QEMU */
};

static char hypervisor_id[13];

bool check_HYPERVISOR_ID() {
  unsigned int eax, ebx, ecx, edx;
  char hypervisor_id[13];

  __asm__ volatile("movl $0x40000000, %%eax;"
                   "cpuid;"
                   : "=b"(ebx), "=c"(ecx), "=d"(edx)
                   :
                   : "%eax"
    );

  *(unsigned int *)&hypervisor_id[0] = ebx;
  *(unsigned int *)&hypervisor_id[4] = ecx;
  *(unsigned int *)&hypervisor_id[8] = edx;
  hypervisor_id[12] = '\0';

  for (int i = 0; i < NB_BRAND_STRING; i++) {
    if (strstr((char *)hypervisor_id, known_hypervisor_id[i])) {
      return true;
    }
  }
  return false;
}
```

II.2.2 rdtsc

L'instruction `rdtsc` [30] (signifiant *read time stamp counter*) est utilisée pour récupérer le nombre de *cycles* du processeur depuis sa dernière remise à zéro. Cette instruction va lire le registre *Time Stamp Counter* et placer le contenu dans `edx` et `eax`.

On peut utiliser `rdtsc` pour compter le nombre de *cycles* nécessaires à l'exécution d'une instruction sérialisée. Dans une machine physique, chaque instruction prend un certain nombre de *cycles*, par exemple, en moyenne, l'instruction `cpuid` prend entre 150 et 500 cycles. Cependant, sur une machine virtuelle, le nombre de cycles de l'instruction `cpuid` se situe entre 3 000 et 6 000.

On peut constater une telle différence pour plusieurs raisons :

- Lorsque nous allons chercher des informations auprès du processeur avec `cpuid`, nous allons causer une sortie de virtualisation (**VMEXIT**, c'est-à-dire arrêter le code s'exécutant dans la machine invité) et passer la main à l'hyperviseur qui va gérer l'interruption.
- Si la machine virtuelle est émulée, alors il y a de forte chance pour que le nombre de cycles requis pour l'exécution d'une instruction soit plus élevé que dans une machine physique.
- La machine virtuelle étant dépendante des composants de la machine physique, il est possible d'observer des ralentissements au sein de la machine si l'hôte utilise trop de ressources.

Nous allons donc implémenter la détection de la manière suivante :

1. Répéter l'opération un certain nombre de fois (ici 1000) pour obtenir une moyenne représentative.
2. À chaque itération :

a) Lire et enregistrer le nombre de cycles processeur écoulés depuis le démarrage dans une variable `start` à l'aide de l'instruction `rdtsc`.

b) Exécuter l'instruction `cpuid`, qui agit comme une instruction sérialisée.

c) Lire une nouvelle fois le nombre de cycles et l'enregistrer dans `end`.

d) Calculer la différence `end - start`, représentant le nombre de cycles pris par `cpuid`, et ajouter cette valeur à un total.

3. Calculer la moyenne des cycles mesurés sur l'ensemble des itérations (delta).
4. Retourner `true` si cette moyenne dépasse 500 cycles, ce qui peut indiquer un environnement virtualisé.

On obtient donc le code en C suivant :

```c
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>

#if defined(_WIN32) || defined(_WIN64)
#include <intrin.h>
#pragma intrinsic(__rdtsc)
#pragma intrinsic(__cpuid)
#else
#include <cpuid.h>
#include <x86intrin.h>
#endif

static inline void cpuid__() {
  asm volatile("cpuid" : : "a"(0) : "ebx", "ecx", "edx");
}

bool check_RDTSC() {
  const int test_nb = 1000;
  unsigned int cycles = 0;
  unsigned int start = 0;
  unsigned int end = 0;

  for (int i = 0; i < test_nb; i++) {

    start = __rdtsc();
    cpuid__();
    end = __rdtsc();

    cycles += end - start;
  }

  unsigned int delta = cycles / test_nb;
  return delta > 500;
}
```

II.2.3 `sidt`

Dans cette technique, nous allons voir comment utiliser la *Table des Descripteurs d'Interruptions* d'un système (en anglais, *IDT - Interrupt Descriptor Table*). Regardons d'abord à quoi ressemble cette structure.

II.2.3.1 `IDT`

L'IDT est une structure de données (sous forme de table) qui est utilisée par les processeurs d'architecture x86. Elle sert à répertorier l'ensemble des descripteurs d'interruptions système, il y a donc 256 entrées, chacune de huit octets (une entrée par interruption). L'emplacement de cette structure est enregistré dans un registre spécifique nommé `IDTR` (*IDT Register*), en cas d'interruption, le processeur n'a qu'à retrouver l'adresse de l'*IDT* en lisant l'*IDTR* et par

la suite retrouver l'interruption dans cette table. Cependant, cette table est unique à chaque cœur d'un processeur et donc à chaque processeur. Cela signifie que lorsque l'on crée une machine virtuelle, l'IDT doit être à un endroit différent de celui du processeur physique. C'est sur ce principe que se base la technique que nous allons présenter.

II.2.3.2 Explication de la technique

Comme expliqué ci-dessus, lorsque l'on crée une machine virtuelle, la table des interruptions (IDT) est souvent placée à une adresse différente de celle utilisée sur une machine physique. En général, cette adresse est située plus haut dans la mémoire physique. Un maliciel peut ainsi détecter s'il s'exécute dans une machine virtuelle en récupérant l'adresse de l'IDT à l'aide de l'instruction `SIDT [31]`, puis en analysant les octets de l'adresse obtenue. Si l'un des octets de poids fort est suffisamment élevé — typiquement proche de `0xFF` — cela peut indiquer une exécution dans un environnement virtualisé. Cette technique, connue sous le nom de **Red Pill** [32], a été présentée par Joanna Rutkowska [4].

Plusieurs implémentations de cette technique existent. Elle dépend majoritairement du système d'exploitation ciblé : en effet, toutes les IDT ne sont pas placées au même endroit en mémoire selon le système. Dans l'exemple ci-dessous, la fonction a pour but de cibler l'IDT d'un système d'exploitation Windows x64. L'adresse de l'IDT dans ce cas se trouve souvent autour de `0x80ffffff` [33].

L'implémentation présentée n'est qu'une des nombreuses variantes possibles de cette technique :

```
#include <stdio.h>
#include <stdbool.h>

bool check_SIDT()
{
    struct
    {
        uint16_t limit;
        uint64_t base;
    } __attribute__((packed)) idtr;

    __asm__("sidt %0" : "=m"(idtr));

    uint8_t third_byte = (idtr.base >> 16) & 0xFF;
    return third_byte != 0xFF;
}
```

4. Blog de Joanna Rutkowska : The Invisible Things Blog

Chapitre II.3

Utilisation de ces techniques dans un maliciel

Comme nous l'avons vu précédemment, il existe un grand nombre de techniques, cependant, la plupart des maliciels n'utilisent qu'une petite partie de celles-ci. On retrouve un très grand nombre d'utilisations de `cpuid` (II.2.1) et `rdtsc` (II.2.2), d'autres techniques sont assez populaires (récupération de l'adresse MAC, analyse de l'espace disponible du stockage, etc.), bien que moins que les deux précédentes. L'article *Longitudinal Study of the Prevalence of Malware Evasive Techniques* [34] expose ce comportement dans le tableau suivant :

Malware [2016, 2020]	APT
`ErasePEHeader` (8.9%)	`IsDebuggerPresentAPI` (7.5%)
`IsDebuggerPresentAPI` (7.1%)	`disk_getdiskfreespace` (5.2%)
`RDTSC` (5.9%)	`RDTSC` (4.1%)
`process_enum` (5.2%)	`CanOpenCsrss` (3.0%)
`vm_check_mac` (4.1%)	`process_enum` (2.7%)
`IsDebuggerPresentPEB` (4.1%)	`IsDebuggerPresentPEB` (2.5%)
`disk_getdiskfreespace` (3.8%)	`time_stalling` (2.3%)
`SizeOfImage` (3.8%)	`SizeOfImage` (2.2%)
`memory_space` (2.4%)	`vm_check_mac` (2.1%)
`CanOpenCsrss` (2.3%)	`NSIT_ThreadHideFromDebugger` (1.4%)

FIGURE II.3.1 – Liste des taux de présence des techniques d'évasions (anti-VM, anti-analyse, etc) les plus fréquentes dans les maliciels génériques et les APT.

D'après l'article *Advanced or not ? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware* [35], les maliciels, en moyenne, préfèrent recourir à de l'anti-debug pour des usages plus généraux et à l'anti-VM lorsqu'il y a une cible particulière [1], comme le confirme le tableau suivant issu de leur rapport :

1. Nous avons pu observer ce comportement dans le premier échantillon de *QakBot*, la plupart des techniques d'anti-VM étaient faites pour détecter *VMware*.

Family	% Anti-debug.	% Anti-VM	Family	% Anti-debug.	% Anti-VM
Sality	89.6%	76.2%	Ramnit	85.8%	71.6%
Zbot	72.9%	39.7%	Zeroaccess	41.6%	50.4%
Winwebsec	80.0%	52.9%	Reveton	74.8%	62.8%
Targeted (APT)	68.6%	84.2%			

Table 4: Percentage of samples using anti-debugging/anti-VM techniques in each malware family

Ce qui confirme l'idée avancée par le premier article [34]. Cependant, cette tendance évolue. En effet, le graphique ci-dessous, issu de l'article de *Advanced or not ? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware* [35], le confirme :

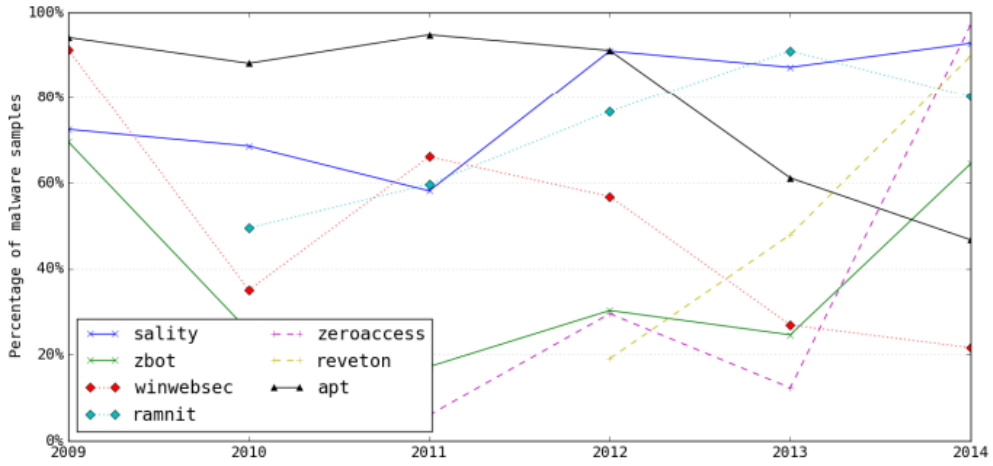

Figure 5. Evolution of the use of anti-VM techniques

Dans l'ensemble, les techniques anti-VM se sont démocratisées dans les maliciels. Malgré une baisse de leur présence dans les échantillons plus récents, ces techniques restent néanmoins utiles pour les développeurs de maliciels et dérangeantes pour les analystes.

Nous allons donc voir comment se protéger face à ces techniques.

Troisième partie

Méthodes de contre-mesures

Chapitre III.1

Modification de configuration

Lorsque l'on crée une machine virtuelle dans un gestionnaire de VM, il est possible de configurer l'ensemble des caractéristiques techniques de la machine, comme la taille du disque, la quantité de mémoire vive, le type de processeur, etc.

Nous allons présenter dans cette partie comment configurer sa machine virtuelle pour contrer les techniques suivantes :

- Taille du stockage
- Nombre de CPU
- Taille de la mémoire
- Résolution de l'écran
- Adresse MAC
- `cpuid`

Nous allons, tout au long de ce chapitre, présenter les modifications à effectuer afin de contrer certaines méthodes. Il est important de préciser que certaines modifications permettant de contrer ces techniques sont assez contraignantes et peu pratiques. C'est pourquoi nous présenterons dans les prochains chapitres (III.2 & III.3) d'autres méthodes permettant de les contrer différemment. Nous allons utiliser la commande `qemu-system-x86_64` [36] afin de générer des machines virtuelles

Chacune des valeurs choisies dans les prochaines sections n'est pas la seule qui fonctionne, nous nous basons sur une norme présente au sein des constructeurs d'ordinateurs. Par exemple, le nombre de cœurs et la taille de la mémoire ont grandement évolué ces 20 dernières années, ce qui permet d'établir un seuil de "normalité" pour ces valeurs.

Voici la commande utilisée pour lancer notre machine virtuelle [1] :

```
qemu-system-x86_64 \
 -enable-kvm \
 -m 8G \
 -smp 6 \
 -cpu qemu64,-hypervisor,kvm=off \
 -drive file=dummy_windows.qcow2,format=qcow2 \
 -netdev user,id=net0 \
 -device e1000,netdev=net0,mac=52:55:44:33:22:11 \
 -vga virtio \
 -device usb-ehci,id=usb \
 -device usb-tablet,bus=usb.0
```

1. Nous l'avons virtualisée avec KVM afin d'obtenir de meilleures performances.

Note importante

La commande ci-dessus est faite pour lancer la machine une fois créée (sans système d'exploitation). Pour rajouter un système d'exploitation, il suffit de rajouter la ligne suivante à la commande ci-dessus :
`-drive file=<PATH_TO_ISO>,media=cdrom`
Avec `PATH_TO_ISO`, le chemin vers l'image du système souhaité.

III.1.1 Augmenter la taille du disque dur

La première technique de détection que nous allons contrer est l'analyse de la taille du disque dur (ou SSD). La majorité des disques durs ont une taille comprise entre 512 Go et 1 To et en général, il ne reste que 100 à 300 Go sur le disque dur d'un ordinateur. La taille du disque dur créé doit donc avoir au moins 200 Go (valeur arbitraire) de libre en permanence. Par exemple, pour une machine virtuelle utilisée pour tester l'exécution d'un binaire, un stockage de 512 Go sera suffisant. Tout d'abord, 512 Go est une taille raisonnable pour le stockage d'une machine physique. De plus, cette machine ne contiendra que le système d'exploitation et quelques fichiers, tout au plus, il restera donc au moins 450 Go.

Pour réaliser cela, nous allons créer une image de disque suffisamment grande à l'aide de la commande `qemu-img` :

```
qemu-img create -f qcow2 dummy_windows.qcow2 512G
```

III.1.2 Modifier le nombre de CPU et la quantité de RAM

Lors de la création de la machine virtuelle, les premières caractéristiques à configurer sont le nombre de processeurs et la quantité de mémoire vive. Notre but va donc être de donner à notre machine virtuelle un nombre suffisant de processeurs et de mémoire afin de la faire ressembler à une machine physique. Cependant, cette machine sera contrainte par les ressources de l'hôte.

Nous avons vu précédemment que si l'on utilise un montant trop faible de processeurs ou de mémoire vive, il est possible qu'un code détecte qu'il tourne dans une machine virtuelle en utilisant les techniques utilisant la quantité de mémoire (II.1.1.3) ou le nombre de processeurs (II.1.1.5). C'est pour cela que nous avons choisi, dans la configuration, six processeurs et 8 Go de RAM. Nous allons utiliser les options `-smp 6` qui va générer une machine avec 6 processeurs et `-m 8G` qui va allouer 8 gigaoctet pour notre machine.

III.1.3 Augmenter la résolution de l'écran

Afin de pouvoir modifier la résolution, une fois dans la machine virtuelle, nous avons besoin d'ajouter l'option `-vga` qui va nous permettre d'émuler une carte VGA. Nous allons utiliser les pilotes `virtio` afin de pouvoir modifier au maximum la résolution du système invité. Il suffit ensuite d'accéder aux paramètres de la machine et d'augmenter manuellement la résolution. Voici l'option à ajouter à la commande :

```
-vga virtio
```

III.1.4 Modifier l'adresse MAC

Comme nous l'avons vu, l'adresse MAC (cf. II.1.1.4) fournie par défaut de QEMU est reconnaissable par ses deux premiers octets `52:54`. Nous devons modifier l'adresse. Pour cela nous allons utiliser l'option `-device` qui va nous permettre de créer un nouveau périphérique. La syntaxe de cette option est la suivante :

```
device driver[,prop[=value][,...]]
```

Dans notre cas, nous allons créer une carte réseau avec l'option `e1000`. Pour modifier l'adresse MAC de cette carte il suffit d'utiliser la propriété `mac=`. Voici l'option complète :

```
-device e1000,netdev=net0,mac=52:55:44:33:22:11
```

III.1.5 Modifier les informations que `cpuid` va récupérer

Nous allons maintenant nous occuper des techniques utilisant l'instruction `cpuid`. Pour cela, nous allons utiliser l'option `-cpu` avec deux propriétés différentes, `-hypervisor` [2] pour masquer le bit d'hyperviseur et `kvm=off` pour masquer le nom de l'hyperviseur. Nous avons donc l'option suivante :

```
-cpu host,-hypervisor,kvm=off
```

2. L'option `-hypervisor` n'existe pas dans la documentation officielle de QEMU.

Chapitre III.2

Hooks des fonctions de l'API Windows

Dans cette partie, nous allons voir un moyen de contourner les protections anti-émulation basées sur l'API Windows en utilisant des *hooks*. L'approche présentée est celle d'un projet personnel disponible sur GitHub, `https://github.com/HalfTimeOfLife/Panoptiv`, tous les morceaux de code que l'on peut trouver ci-dessous sont issus de ce dépôt.

III.2.1 Injection de DLL

Il existe plusieurs façons d'injecter une DLL dans un processus, on peut :

- Utiliser un code spécialement fait pour injecter une DLL dans un programme comme ce qui est utilisé dans le projet.
- Détourner une DLL afin de la remplacer par une version trafiquée.
- Utiliser une extension d'un débogueur (par exemple `!injectdll` de WinDbg [1]).

III.2.2 Installation d'un *hook*

Plusieurs techniques existent pour installer un *hook*, on peut faire appel à une librairie déjà existante, par exemple Detours, ou on peut implémenter une solution plus explicite : modifier les octets de la fonction visée (*Inline Hooking*). C'est-à-dire que nous allons remplacer le prologue de la fonction par une instruction `jmp` vers notre fonction *hooked*. Nous allons voir ci-dessous comment installer un *hook* depuis une DLL.

III.2.2.1 Explication du *Inline Hooking*

Comme expliqué ci-dessus, pour installer un *hook*, nous allons devoir remplacer le prologue d'un appel de fonction. Dans Windows, le prologue d'une fonction est souvent le même :

```
mov edi, edi
push ebp
mov ebp, esp
```

Les instructions `push ebp` et `mov ebp, esp` sont utilisées pour créer une nouvelle pile d'appel pour la fonction appelée. L'instruction `mov edi, edi` est un `nop` sur deux octets, cette fonctionnalité est utilisée pour permettre de déboguer une fonction plus facilement. Cette suite d'instructions correspond à l'hexadécimal suivant : `89FF5589E5`.

On va donc chercher à remplacer ce prologue (cinq octets) par un nouvel ensemble de cinq octets qui va correspondre à un `jmp <ADRESSE_FONCTION>`.

1. Article de l'auteur : !injectdll – a WinDbg extension for DLL injection

III.2.2.2 Exemple de code

Tout d'abord, voici la syntaxe qu'aura notre fonction :

```
void InstallHook(char *moduleName, char *functionName, void *hookFunction, BYTE
*backupBytes, void **originalFunction);
```

Les paramètres sont les suivants :

1. `moduleName` : Nom du module contenant la fonction que l'on souhaite *hook*.
2. `functionName` : Nom de la fonction que l'on souhaite *hook*.
3. `hookFunction` : Pointeur vers la fonction qui va remplacer la fonction que l'on va *hook*.
4. `backupBytes` : Pointeur vers une variable qui va sauvegarder les octets originaux.
5. `originalFunction` : Pointeur vers une variable qui va sauvegarder l'adresse de la fonction d'origine que l'on souhaite *hook*.

Pour que l'installation du *hook* se fasse correctement, nous avons besoin de plusieurs informations. Tout d'abord, il nous faut l'adresse de la fonction que l'on souhaite *hook*. Pour cela, nous allons utiliser la fonction `GetProcAddress` :

```
/* Get handle of module containing function to hook */
HMODULE hModule = GetModuleHandleA(moduleName);
if (hModule == NULL) {
    printf("[-] Failed to get module handle for %s\n", moduleName);
    return;
}

/* Get address of function to hook */
FARPROC pFunction = GetProcAddress(hModule, functionName);
if (pFunction == NULL) {
    printf("[-] Failed to get function address for %s in %s\n", functionName,
moduleName);
    return;
}
```

Par la suite, on peut commencer à installer le *hook*. Cette opération s'effectue en plusieurs étapes. Nous allons commencer par sauvegarder les 5 premiers octets de la fonction d'origine et modifier les protections mémoire de la fonction ciblée (`VirtualProtect`) :

```
DWORD oldProtect;
memcpy(backupBytes, pFunction, 5);

DWORD relAddr = ((DWORD)hookFunction - (DWORD)pFunction) - 5;

VirtualProtect(pFunction, 5, PAGE_EXECUTE_READWRITE, &oldProtect);
```

On va ensuite créer un tableau de 5 octets qui va correspondre au nouveau prologue de la fonction que l'on a *hook*, ces 5 octets représenteront l'instruction `jmp <ADRESSE_HOOK>`, ensuite à l'aide de `memcpy`, on va remplacer l'ancien prologue par ces nouveaux octets qui vont détourner le flot d'exécution du programme vers notre fonction :

```
BYTE patch[5] = { 0xE9 };
memcpy(patch + 1, &relAddr, 4);
memcpy(pFunction, patch, 5);

/* Set back the memory protection as it was */
VirtualProtect(pFunction, 5, oldProtect, &oldProtect);
```

Ensuite, on enregistre le pointeur de la fonction d'origine dans une variable, afin d'avoir la possibilité de l'appeler dans le *hook* :

```
if (originalFunction)
    *originalFunction = (void*)pFunction;
```

III.2.3 Mise en place de substitut

Une fois que nous sommes capables de remplacer une fonction de l'API Windows, il ne nous reste plus qu'à implémenter des substituts pour chaque fonction. Par exemple, si l'on souhaite contourner la technique de détection de la taille du disque dur (cf. II.1.1.2), on peut créer une nouvelle version de la fonction `GetDiskFreeSpaceExA` telle que ci-dessous :

```
BOOL WINAPI HookedGetDiskFreeSpaceExA(LPCSTR lpDirectoryName, PULARGE_INTEGER
lpFreeBytesAvailableToCaller, PULARGE_INTEGER lpTotalNumberOfBytes, PULARGE_INTEGER
lpTotalNumberOfFreeBytes) {

        if (!lpDirectoryName || !lpFreeBytesAvailableToCaller ||
!lpTotalNumberOfBytes || !lpTotalNumberOfFreeBytes) return FALSE;

    lpTotalNumberOfBytes->QuadPart = 1000000000000;
    lpTotalNumberOfFreeBytes->QuadPart = 500000000000;

    return TRUE;
}
```

Cette technique est censée détecter si la quantité de stockage est suffisamment grande pour prouver qu'elle appartient à une machine physique. Si ce n'est pas le cas, alors on peut déduire que le programme tourne dans une machine virtuelle.

La fonction `HookedGetDiskFreeSpaceExA` va tout le temps donner la même quantité de stockage totale et restante, une valeur suffisamment grande a été choisie pour contourner la technique décrite ci-dessus. Il est possible de faire cela avec n'importe quelle fonction de l'API Windows [2].

Nous allons maintenant voir comment contrer des techniques anti-VM en utilisant un débogueur ce qui permet de contrôler en temps réel le comportement d'un maliciel.

2. Tous les substituts des fonctions de l'API Windows peuvent être trouvés dans le fichier `PanoptivDLL/dllmain.c` dans le dépôt Panoptiv.

Chapitre III.3

Appliquer un correctif

Dans cette partie, nous allons voir comment il est possible de contourner les protections anti-émulation d'un exécutable en utilisant un débogueur. Ici, nous allons utiliser `WinDbg` [37] en débogage noyau à distance afin d'essayer de rentrer dans le contexte d'un exécutable qui s'appelle `AVMBinary_SIDT_no_stub.exe`, c'est un exécutable qui implémente la protection anti-émulation `sidt` (cf. II.2.3).

III.3.1 Mise en place de l'environnement

Pour déboguer un noyau à distance, il est nécessaire d'utiliser une machine virtuelle en plus de la machine que l'on souhaite déboguer. Cette machine doit se situer dans le même réseau que la machine cible, car elle doit pouvoir s'y connecter en utilisant, par exemple, une passerelle (logicielle).

Sur la machine cible

Avant de s'attarder sur cette machine, nous devons d'abord correctement configurer la machine cible afin "d'autoriser" et de permettre le débogage. Pour cela, il faut utiliser les commandes suivantes :

- `bcdedit -set TESTSIGNING ON` : Active la possibilité d'ajouter des drivers non signés
- `bcdedit /debug on` : Active le débogage pour la machine cible
- `bcdedit /dbgsettings net hostip:<ip> port:<port>` : Spécifie le port et l'adresse IP de la machine de débogage, afin qu'elle puisse connecter le débogueur à la machine cible [1]

Sur la machine de débogage

Sur la machine de débogage, il suffit d'activer WinDbg ainsi que comment établir un pont entre la machine de débogage et la machine cible.

Il existe différentes techniques pour établir le pont entre la machine de débogage et la machine cible [2]. Par exemple, nous avons les solutions suivantes possibles :

- Redirection de port via une pipe : `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/attaching-to-a-virtual-machine--kernel-mode-`
- VirtualKD pour VirtualBox et VMWare : `https://sysprogs.com/legacy/virtualkd/tutorials/install/`
- Ntoseye : `https://github.com/dmaivel/ntoseye`, différent de WinDbg

1. On peut également spécifier la clef que l'on veut utiliser avec `key:<key>`. Si cette option n'est pas utilisée, une clef aléatoire est générée.
2. Ici, nous avons utilisé l'outil `RvnKdBridge` qui permet de se connecter à une machine virtuelle sur la plateforme **esReverse** de *eShard*

III.3.2 Rentrer dans le contexte d'un processus

Lorsqu'on débogue le noyau, le débogueur (WinDbg) ne peut pas interagir directement avec un exécutable (pour cela, on pourrait déboguer directement un exécutable). De plus, étant en débogage à distance, le débogueur n'a pas accès à l'exécutable ; cependant, il est possible de s'attacher à n'importe quel processus depuis le contexte noyau.

Pour cela, nous avons besoin de savoir quand un processus est créé et trouver le processus qui nous intéresse dans la liste des processus en cours. Nous allons utiliser la fonction `NtCreateUserProcess` de la librairie `ntdll.dll`. Cette fonction est la fonction de plus bas niveau appelée par la fonction `CreateProcess` qui permet de créer un nouveau processus, le schéma ci-dessous montre ce comportement :

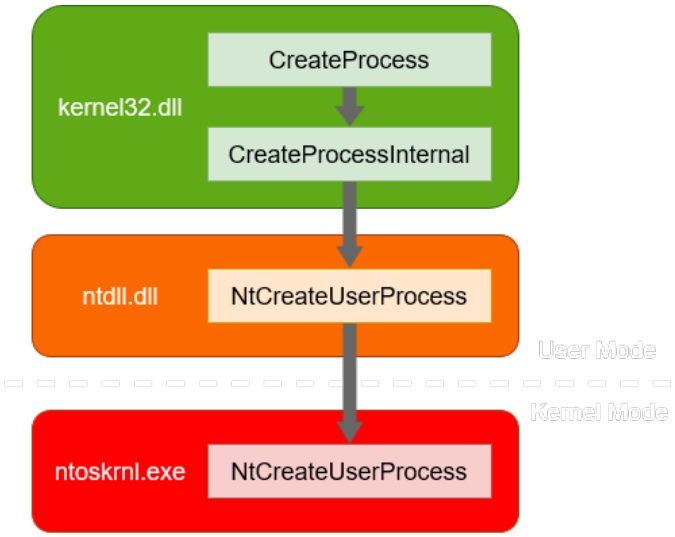

FIGURE III.3.1 – Chaîne d'appels de la fonction `CreateProcess`

Nous allons donc placer un point d'arrêt sur cette fonction et lancer notre exécutable. Cependant, à ce point, notre processus n'est pas encore présent en mémoire, il faut ainsi utiliser une autre fonction afin de trouver le point d'entrée du processus ciblé.

La fonction `NtMapViewOfSection` est la fonction de plus bas niveau appelée par `ZwMapViewOfSection` pour mapper l'exécutable du processus en mémoire. À l'issue de cette fonction, nous pouvons trouver la liste des processus présents en mémoire à l'aide de la commande `!process 0 0` :

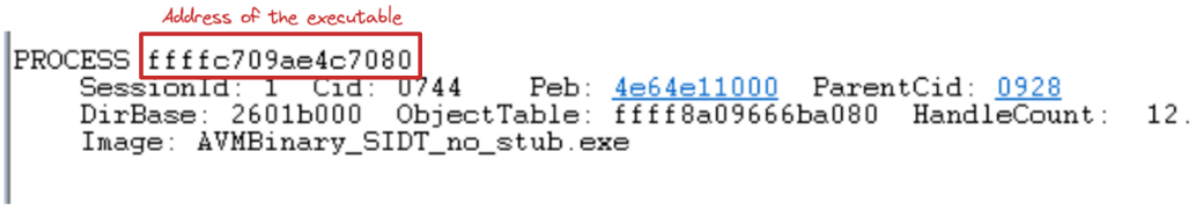

FIGURE III.3.2 – Présence de l'exécutable dans la liste des processus en mémoire

Une fois chargé en mémoire, il est possible de s'attacher à ce processus afin de rentrer dans son contexte. Nous avons besoin de récupérer le champ `PROCESS` de l'exécutable et de l'utiliser avec la commande `.process /p /r <PROCESS>`. Les options `/p` et `/r` sont définis comme suit dans la documentation :

> **/p** Traduit toutes les entrées de table de pages de transition (PTE) de ce processus en adresses physiques avant l'accès, si vous utilisez /p et que Process est différent de zéro. Cette traduction peut entraîner des ralentissements, car le débogueur doit trouver les adresses physiques de toute la mémoire utilisée par ce processus. De plus, le débogueur peut avoir à transférer une quantité importante de données à travers le câble de débogage. (Ce comportement est identique à celui de .cache forcedecodeuser.)
>
> Si vous incluez l'option /p et que Process est égal à zéro ou que vous l'omettez, la traduction est désactivée. (Ce comportement est identique à celui de .cache noforcedecodeptes).
>
> **/r** Recharge les symboles en mode utilisateur après que le contexte du processus a été défini, si vous utilisez les options /r et /p. (Ce comportement est identique à celui de .reload /user.)

III.3.3 Retrouver le point d'entrée d'un exécutable

Lorsqu'on s'attache à un processus, nous avons la possibilité de voir la liste des modules chargés pour ce processus. Grâce à cela, on peut récupérer l'adresse de la base de l'exécutable, que l'on nommera `BASE_ADDRESS`. Pour cela, on utilise la commande `lm` :

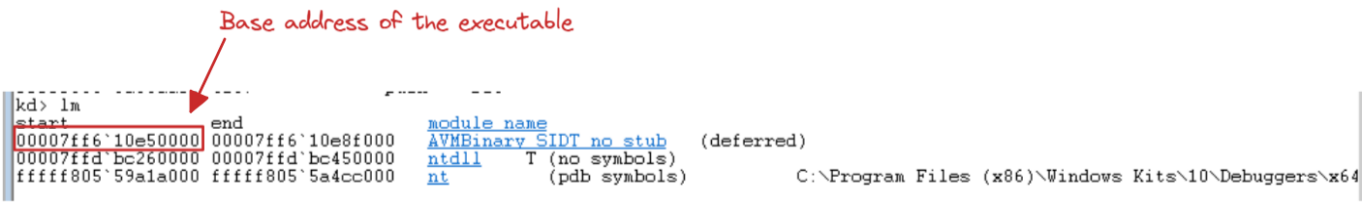

FIGURE III.3.3 – Récupération de l'adresse de la base de l'exécutable

On peut maintenant récupérer l'adresse du point d'entrée (`OFFSET_ENTRYPOINT`) à l'aide de la commande `!dh`[3] :

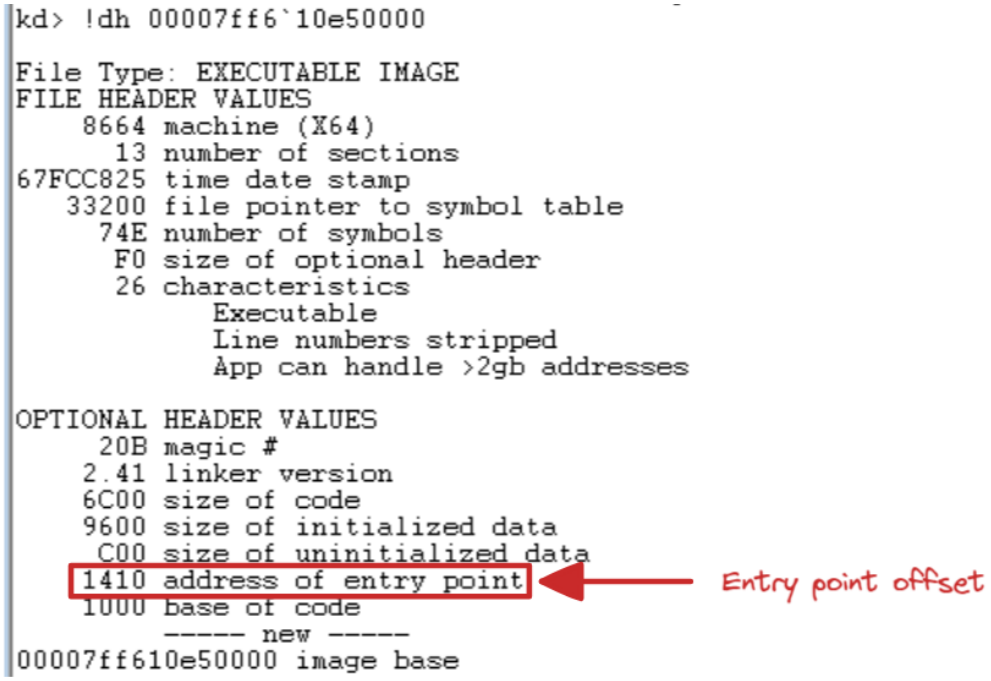

FIGURE III.3.4 – Récupération du point d'entrée de l'exécutable

Ainsi, on sait que le point d'entrée va se situer à `BASE_ADDRESS` + `OFFSET_ENTRYPOINT`, il suffit donc de placer un point d'arrêt à cet endroit afin de commencer à modifier notre exécutable :

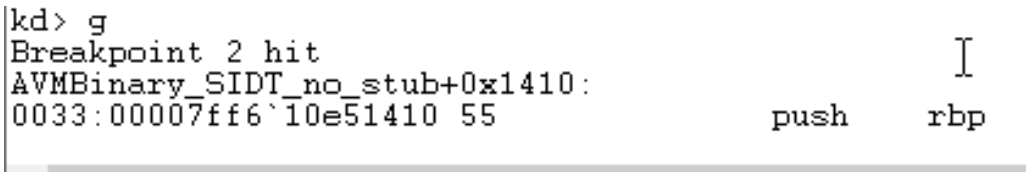

FIGURE III.3.5 – Point d'arrêt au point d'entrée de l'exécutable

Nous avons montré comment accéder au contexte d'un exécutable, il faut maintenant réussir à contourner les protections anti-VM et donc corriger (modifier) le comportement de l'exécutable, pour cela, nous allons voir un exemple avec l'exécutable étudié précédemment. C'est-à-dire, nous allons montrer comment, avec WinDbg, faire en sorte que l'adresse de la table des interruptions systèmes (IDT) ne correspond pas à ce que la technique de détection attend. Nous allons montrer comment contourner la technique `sidt` (cf. II.2.3).

3. Signifiant *display headers*, cette commande permet d'afficher les en-têtes d'une image donnée.

III.3.4 Exemple : Contournement de la technique `SIDT`

La technique `sidt` (cf. II.2.3) consiste à retrouver l'adresse de la table des appels systèmes et à déterminer si elle se situe à une adresse suffisamment haute pour être considérée comme faisant partie d'un système virtuel.

Afin de contourner cette technique, nous avons besoin de récupérer l'adresse de l'instruction et de placer un point d'arrêt à cette adresse. Pour retrouver l'adresse de l'instruction, nous allons construire un script qui va parcourir notre exécutable et retrouver toutes les instructions `sidt`. Nous avons cependant deux contraintes, la première étant que le pont (bridge) que l'on utilise, n'autorise pas les modifications de registre [4], dans ce cas cela ne nous dérange pas trop mais on aurait pu directement modifier le registre de l'IDT afin de contrer la technique. La deuxième est bien plus contraignantes, la recherche en mémoire via la commande `s` ne fonctionne pas [5], nous ne pouvons donc pas chercher dans la mémoire du processus les appels à certains opcodes afin de retrouver `sidt`.

Nous allons donc devoir créer un script capable de retrouver l'instruction. Cependant, elle comporte une spécificité qui va compliquer sa recherche exacte. L'opcode de base de l'instruction `sidt` est `0f 01`, cependant il est suivi d'un octet `ModR/M` dont la valeur dépend notamment du registre utilisé. Nous allons donc devoir nous baser sur les deux octets de base afin de détecter `sidt`, ce qui va mener à un certain nombre de faux positif.

Le script va donc :

- Parcourir toutes les adresses de 4 en 4, entre l'adresse de base et de fin de notre processus (utiliser la commande `lm` pour obtenir ces adresses), nous allons utiliser une boucle while itérant sur toutes les adresses .
- Tester si l'instruction courante est une instruction `sidt` qui a pour opcode `0f01`. Pour ce faire nous allons comparer les deux premiers octets de l'instruction avec `010f` [6].
- Si les octets correspondent, alors on affiche l'instruction et on place un point d'arrêt à ce niveau.

Lorsque l'on récupère l'opcode de l'instruction courante, il va être nécessaire de vérifier que l'on n'accède pas à des parties de la mémoire non-allouées. Pour cela, nous allons utiliser la commande `.catch` afin de capturer les erreurs, cette commande va englober l'exécution d'un ensemble de commandes, si l'une d'elles échoue alors la commande qui a échoué est ignorée (aucune variable n'est modifiée) et elle saute celles d'après. La prochaine commande exécutée sera la première en dehors du contexte de `.catch`.

Nous allons utiliser la commande `.catch` pour englober les commandes `r @$t5 = poi(@$t3)` et `r @$t6 = @$t3`. La commande `poi` permet de déréférencer un pointeur afin d'accéder à la valeur à l'adresse ciblée, dans notre cas, nous allons déréférencer le pointeur d'instruction (sauvegardée dans une variable intermédiaire `@$t3`). C'est cette commande qui va potentiellement échouer, en effet, il se peut qu'en essayant d'accéder à la mémoire à une certaine adresse, la zone en question ne soit pas allouée et donc qu'une erreur se produise.

La deuxième commande va servir à enregistrer l'état de la variable contenant le pointeur de l'instruction courante, cela permettra de la comparer avec l'état précédent que nous aurons enregistré afin de savoir si les commandes dans le `.catch` ont été effectuées. Nous aurons donc la commande suivante :

```
.catch { r @$t5 = poi(@$t3); r @$t6 = @$t3 }
```

Cette commande va cependant créer un grand nombre d'erreurs mémoires et les afficher. Pour éviter cela, nous allons utiliser la commande `.foreach` [7]. Cette commande permet de répéter une série de commandes (`OutCommands`) pour chaque élément (`Variable`) d'une liste générée par un ensemble de commandes en entrée (`InCommands`). Voici sa syntaxe :

```
.foreach [Options] ( Variable { InCommands } ) { OutCommands }
```

Nous allons faire un `.foreach` avec :

- `Variable` = `ignored`
- `InCommands` = `.catch { r @$t5 = poi(@$t3); r @$t6 = @$t3 } }`
- `OutCommands` = `{}`

On fait ensuite un test afin de savoir si les instructions en entrée ont été exécutées :

4. C'est dû à l'outil `RvnKdBridge` que nous utilisons qui ne supporte cette fonctionnalité.
5. Dans ce cas, il est possible que ce soit dû à la manière dont nous nous attachons au contexte du processus.
6. Sur Windows, les adresses sont petit-boutistes (octet de poids le plus faible en premier).
7. `https://stackoverflow.com/questions/52477397/windbg-possible-to-suppress-output-for-outmask-1-or-outmask-d`

```
.if (@$t6 == @$t4) {
   r @$t3
   r @$t3 = @$t3 + 1
   .continue
}
```

Si ce n'est pas le cas, on incrémente l'instruction regardée et on continue. Il ne reste plus qu'à tester si l'instruction actuelle utilise sidt. Pour cela, nous allons déréférencer l'adresse avec `poi` et comparer le résultat avec l'opcode de `sidt`. Voici le code complet :

```
.printf "[+] Searching for sidt instruction between %p and %p\n", ${$arg1}, ${$arg2}
.printf "-------------------------------------------------------------------------\n\n"

r @$t0 = ${$arg1}
r @$t1 = ${$arg2}
r @$t3 = @$t0
r @$t4 = 0
r @$t5 = 0
r @$t6 = @$t3

.while (@$t3 < @$t1) {

    .foreach (ignored { .catch { r @$t5 = poi(@$t3); r @$t6 = @$t3 } }) {}

    .if (@$t6 == @$t4) {
       r @$t3
       r @$t3 = @$t3 + 1
       .continue
    }

    .if ((poi(@$t3) & 0xFFFF) == 0x010F) {
        .printf "[+] Found SIDT instruction at %p\n", @$t3
        bp @$t3
        .printf "[+] Breakpoint set at %p\n", @$t3
        u @$t3 L1
    }

    r @$t4 = @$t3
    r @$t3 = @$t3 + 1
}
```

Une fois que tous les point d'arrêt ont été posé, il faut modifier l'adresse récupérée par l'instruction. Pour cela, à l'aide de la commande `db DEST L8`, nous allons regarder la destination dans laquelle `sidt` a placé l'instruction (nous allons appeler cette donnée `DEST`) et modifier la valeur à cet endroit. Nous allons réécrire cette adresse en utilisant la commande `eb` qui écrit un octet en mémoire à la destination choisie. Nous allons faire cela pour tous les octets de l'adresse. Par exemple si l'instruction `sidt` a placé l'adresse de l'IDT à l'adresse `rbp-0xb` :

```
.for (r @$t4 = @$t4; @$t4 >= 0x6; r @$t4 = @$t4 - 1) {
  eb @rbp - @$t4 ff;
};
```

Quatrième partie

Cas pratiques

Chapitre IV.1

Al-Khaser

Note importante

Dans cette partie, nous allons utiliser une machine virtuelle créée à l'aide de QEMU qui a les caractéristiques suivantes :

- Taille de la mémoire vive : 2 Go
- Taille du disque dur : 60 Go
- CPU : 1 cœur, 1 processeur logique
- Adresse MAC : `52-54-00-12-34-56`

Al-Khaser est un maliciel *PoC*, c'est-à-dire un exécutable fait pour tester la furtivité d'un système virtuel, il implémente de nombreuses techniques d'anti-débogage, d'anti-VM et d'anti-analyse. Le code source de ce maliciel est disponible sur le dépôt GitHub `al-khaser` [38].

Le maliciel se présente sous la forme d'un exécutable Windows qui a plusieurs options disponibles pour tester différents types de techniques. Nous allons donc nous concentrer sur les techniques anti-VM implémentées. Voici celles que nous allons utiliser :

- `QEMU` : Détection de machines virtuelles basées sur QEMU
- `HYPERV` : Détection de machines virtuelles, virtualisées par Hyper-V
- `GEN_SANDBOX` : Technique générique d'anti-VM

La commande utilisée pour exécuter ce maliciel sera la suivante :

```
.\al-khaser_x64.exe --check QEMU --check HYPERV --check GEN_SANDBOX
```

Cet ensemble d'options permet de tester si un système se fait détecter par les techniques anti-VM implémentées, dans ce cas, nous avons pris les techniques qui détectaient l'environnement que nous avons testé.

Ce maliciel n'étant qu'une *Proof of Concept*, il n'y aura pas d'analyse détaillée du fonctionnement menant à certains comportements, nous n'allons nous concentrer que sur les techniques implémentées.

IV.1.1 Techniques d'Anti-VM pour QEMU

L'option `QEMU` va utiliser les techniques suivantes :

```
VOID qemu_reg_key_value();
VOID qemu_reg_keys();
VOID qemu_processes();
VOID qemu_dir();
BOOL qemu_firmware_ACPI();
BOOL qemu_firmware_SMBIOS();
```

Parmi celle-là, deux techniques détectent notre machine virtuelle : `ACPI` et `SMBIOS`.

IV.1.1.1 `ACPI`

La première technique que nous allons regarder est nommée `qemu_firmware_ACPI`. Elle fait appel à une fonction nommée `get_system_firmware` qui va interroger les tables ACPI. *ACPI, Advanced Configuration and Power Interface*, est une norme de gestion de la consommation électrique. Les informations concernant le système sont enregistrées dans plusieurs tables ACPI auxquelles le système peut accéder via le BIOS.

La fonction va tout d'abord faire appel à une fonction de l'API Windows `EnumSystemFirmwareTables` [39] pour obtenir la liste des tables ACPI. Voici la syntaxe de la fonction :

```
UINT EnumSystemFirmwareTables(
  [in]  DWORD FirmwareTableProviderSignature,
  [out] PVOID pFirmwareTableEnumBuffer,
  [in]  DWORD BufferSize
);
```

Afin de récupérer les tables ACPI, nous allons devoir fixer la signature, `FirmwareTableProviderSignature`, à `APCI`. La fonction va remplir une liste de tables ACPI, le paramètre `pFirmwareTableEnumBuffer` pointe vers cette structure. Par la suite, le programme va récupérer une par une ces tables à l'aide de la fonction `get_system_firmware` et rechercher dans ces tables des chaînes de caractères caractéristiques des environnements virtuels, ces chaînes de caractères sont les suivantes :

- `"FWCF"`
- `"QEMU0002"`
- `"BOCHS"`
- `"BXPC"`

Le code complet de la fonction `qemu_firmware_ACPI` se trouve en Annexe 4.

IV.1.1.2 `SMBIOS`

La fonction qui réalise cette technique s'appelle `qemu_firmware_SMBIOS`, pour cela elle appelle à la fonction nommée `get_system_firmware` qui va retourner un pointeur vers un buffer contenant la table SMBIOS. Le SMBIOS, *System Management Basic Input/Output System* est une norme qui établit un ensemble de structures de données permettant au micrologiciel de communiquer au système d'exploitation les caractéristiques de la machine (processeur, mémoire vive, carte mère, etc).

Comme énoncé ci-dessus, c'est la fonction `get_system_firmware` qui va se charger de retrouver la table SMBIOS et qui va retourner un pointeur (cf. Annexe 5). Cette fonction va faire appel à la fonction `GetSystemFirmwareTable` [40] qui a la syntaxe suivante :

```
UINT GetSystemFirmwareTable(
  [in]  DWORD FirmwareTableProviderSignature,
  [in]  DWORD FirmwareTableID,
  [out] PVOID pFirmwareTableBuffer,
  [in]  DWORD BufferSize
);
```

Le premier paramètre correspond à la table de micrologiciel que l'on souhaite récupérer. Dans le cas d'`alkhaser`, pour retrouver la table SMBIOS, il faut que `FirmwareTableProviderSignature` vaille `RSMB`. Une fois le micrologiciel récupéré, le programme vérifie s'il contient l'une des deux chaînes de caractères suivantes : *QEMU* ou *qemu*. Si c'est le cas, la fonction renverra `TRUE`. Voici le code de la fonction :

```
BOOL qemu_firmware_SMBIOS()
{
        BOOL result = FALSE;

        DWORD smbiosSize = 0;
```

```
        PBYTE smbios = get_system_firmware(static_cast<DWORD>('RSMB'), 0x0000,
&smbiosSize);
        if (smbios != NULL)
        {
                PBYTE qemuString1 = (PBYTE)"qemu";
                size_t StringLen = 4;
                PBYTE qemuString2 = (PBYTE)"QEMU";

                if (find_str_in_data(qemuString1, StringLen, smbios, smbiosSize) ||
                        find_str_in_data(qemuString2, StringLen, smbios, smbiosSize))
                {
                        result = TRUE;
                }

                free(smbios);
        }

        return result;
}
```

IV.1.2 Techniques d'Anti-VM pour Hyper-V

L'option `-check HYPERV` va faire en sorte que le maliciel utilise les fonctions suivantes :

```
BOOL check_hyperv_driver_objects();
BOOL check_hyperv_global_objects();
```

Dans notre cas, une seule de ces fonctions détecte notre environnement virtuel : `check_hyperv_global_objects`. Cette fonction fait tout d'abord appel à une autre fonction : `enumerate_object_directory`, le code de cette fonction peut être trouvé en Annexe 6. Cette fonction sert à retrouver la liste des noms des objets présents dans un répertoire du gestionnaire d'objets de Windows. La fonction prend en paramètre le nom du répertoire et retourne la liste des objets contenu dans ce répertoire.

Ces répertoires sont représentés par une structure `OBJECT_DIRECTORY` qui est une table de hachage qui pointe vers des en-têtes d'objets existants. Ces objets sont des liens symboliques qui font correspondre des noms de périphériques accessibles dans l'espace utilisateur (`C:`) à leur équivalent dans l'espace noyau (`\Device\HarddiskVolume1`).

Dans notre cas, `enumerate_object_directory` est appelé pour récupérer les objets du répertoire `GLOBAL??`. Ce répertoire contient des liens symboliques pour des périphériques DOS du système.

Une fois le nom de ces objets récupérés, la technique va chercher à savoir si un des noms suivants est présent :

- `VMBUS#` : Bus virtuel qui permet de créer un canal de communication entre l'hôte et l'invité, utilisé par Hyper-V pour faire de la paravirtualisation
- `VDRVROOT` : Permet à Windows de gérer les disques virtuels
- `VmGenerationCounter` : Permet de détecter la création d'instantanés (*snapshot*) ou la restauration de machine virtuelle.
- `VmGid` : Génere et assigne un identifiant unique à chaque machine virtuelle

Si c'est le cas alors le programme en déduira qu'il tourne dans une machine virtuelle.

Voici le code de la fonction `check_hyperv_global_objects` :

```
BOOL check_hyperv_global_objects()
{
        auto globalObjs = enumerate_object_directory(L"\\GLOBAL??");
        if (globalObjs == nullptr)
        {
                return FALSE;
        }
        for (wchar_t* globalObj : *globalObjs)
        {
```

```
                if (StrStrW(globalObj, L"VMBUS#") != NULL)
                {
                        return TRUE;
                }
                if (StrCmpCW(globalObj, L"VDRVROOT") == 0)
                {
                        return TRUE;
                }
                if (StrCmpCW(globalObj, L"VmGenerationCounter") == 0)
                {
                        return TRUE;
                }
                if (StrCmpCW(globalObj, L"VmGid") == 0)
                {
                        return TRUE;
                }
        }
        return FALSE;
}
```

IV.1.3 Techniques d'Anti-VM générales

`al-khaser` implémente aussi une longue liste de techniques anti-VM plus générales, la liste de toutes ces techniques se trouve en Annexe 7, le code complet de ces techniques peut être trouvé sur le dépôt dans le fichier `Generic.cpp`[1], nous allons nous concentrer sur celles qui détectent notre environnement et qui n'ont pas déjà été expliquées.

IV.1.3.1 *Windows Management Instrumentation* (WMI)

`WMI` [41] est une interface permettant d'accéder aux données d'un système Windows, il est possible de récupérer des informations sur le matériel au travers de cette interface.

Pour exécuter des requêtes WMI en C, on utilise la fonction `ExecQuery`, le maliciel `al-khaser` utilise cette fonction via sa propre fonction `ExecWMIQuery` (cf Annexe 8). Voici la syntaxe de `ExecQuery` :

```
HRESULT ExecQuery(
  [in]  const BSTR           strQueryLanguage,
  [in]  const BSTR           strQuery,
  [in]  long                 lFlags,
  [in]  IWbemContext         *pCtx,
  [out] IEnumWbemClassObject **ppEnum
);
```

`al-khaser` implémente un certain nombre de ces techniques basées sur WMI, nous allons décrire le fonctionnement général des deux fonctions suivantes :

```
BOOL manufacturer_computer_system_wmi();
BOOL cpu_fan_wmi();
```

`manufacturer_computer_system_wmi`

Cette technique va utiliser la requête WMI `SELECT * FROM Win32_ComputerSystem` dont le programme va extraire le champ `Manufacturer` qui correspond au nom du fabricant de l'ordinateur. Dans une machine virtuelle, le champ sera égal au nom du gestionnaire de machine virtuelle utilisé (ou émulateur), par exemple, si l'on génère une machine virtuelle en utilisant QEMU, le champ vaudra `QEMU`. Il suffit de comparer (à l'aide de `StrStr`) le nom récupéré à une liste prédéfinie de noms de gestionnaire de machines virtuelles.

1. https://github.com/ayoubfaouzi/al-khaser/blob/master/al-khaser/AntiVM/Generic.cpp

cpu_fan_wmi

Toutes les machines physiques ont un certain nombre de ventilateurs, on peut le confirmer (sur un système Windows) en regardant les instances de ventilateurs présentes à l'aide de la classe `Win32_Fan` que l'on peut récupérer à l'aide d'une requête WMI. Si on récupère cette classe depuis une machine virtuelle, la requête n'est censée rien retourner vu que les ventilateurs d'une machine ne sont pas virtualisés/émulés [2].

IV.1.3.2 Entrée utilisateurs

Cette technique se base sur le même principe que la détection de mouvement de la souris (cf. II.1.3.1), l'idée étant que dans une machine physique, un utilisateur va en permanence utiliser son clavier, ou tout du moins la plupart du temps. Il suffit donc de regarder le temps de la dernière utilisation d'une touche du clavier. Si la date est trop éloignée par rapport à un seuil défini, alors le programme peut en déduire qu'il est probablement exécuté dans une machine virtuelle

Voici l'implémentation de cette technique dans `al-khaser` :

```
BOOL lack_user_input()
{
        int correct_idle_time_counter = 0;
        DWORD current_tick_count = 0;
        LASTINPUTINFO last_input_info; // Contains the time of the last input
        last_input_info.cbSize = sizeof(LASTINPUTINFO);

        for (int i = 0; i < 128; ++i) {
                Sleep(0xb);
                // Retrieves the time of the last input event
                if (GetLastInputInfo(&last_input_info)) {
                        current_tick_count = GetTickCount();
                        if (current_tick_count < last_input_info.dwTime)
                                // impossible case unless GetTickCount is manipulated
                                return TRUE;
                        if (current_tick_count - last_input_info.dwTime < 100) {
                                correct_idle_time_counter++;
                                if (correct_idle_time_counter >= 10)
                                        return FALSE;
                        }
                } else // GetLastInputInfo must not fail
                        return TRUE;
        }
        return TRUE;
}
```

Le maliciel `al-khaser` est un très bon moyen de vérifier si une machine virtuelle est correctement configurée afin d'éviter ces techniques. Il implémente un très grand nombre de techniques, nous n'avons vu ici qu'une petite partie des capacités de ce programme.

Nous allons maintenant prendre exemple sur un "vrai" maliciel nommé *QakBot* [3] qui va nous permettre de voir un exemple de l'utilisation des protections anti-VM implémentées par un maliciel.

2. Il est possible de contrer cette technique à l'aide de la technique expliquée dans cet article :`https://wbenny.github.io/2025/06/29/i-made-my-vm-think-it-has-a-cpu-fan.html`
3. Plus précisément, *QakBot* est un cheval de Troie bancaire qui a grandement évolué depuis sa première apparition (supposée) en 2008.

Chapitre IV.2

QakBot

QakBot est un cheval de Troie bancaire qui a évolué pour multiplier ses capacités, cependant, ses nombreuses variantes semblent s'orienter vers un rôle de chargeur de maliciel.

IV.2.1 Échantillon

Au cours de cette section, nous allons présenter la version suivante de *QakBot* :

Date	Version SHA-256
Mars 2023	`f5ff6dbf5206cc2db098b41f5af14303f6dc43e36c5ec02604a50d5cfecf4790`

TABLE IV.2.1 – Échantillons SHA-256 de la version de mars 2023

Les protections anti-VM implémentées dans cet échantillon vont se servir de techniques rudimentaires basées sur `cpuid`, cependant *QakBot* implémente aussi deux méthodes de détections qui concernent spécifiquement VMWare.

IV.2.2 Analyse

Au point d'entrée du maliciel, on retrouve le premier bout de code qui va allouer un tas pour le programme (`HeapCreate`) et importer toutes les DLL nécessaires (`setup_imports`) :

```
00401a28    void _start() __noreturn
00401a28    {
00401a28        int32_t numArgs = 0;
00401a44        PWSTR* argv = CommandLineToArgvW(GetCommandLineW(), &numArgs);
00401a4e        int32_t eax_2;
00401a4e
00401a4e        if (argv)
00401a4e        {
00401a4e            hHeap = HeapCreate(HEAP_NONE, 0x80000, 0);
00401a72            eax_2 = setup_imports(3);
00401a4e        }
00401a4e
...
```

Pour la suite, nous avons utilisé un outil de *Time Travel Analysis* (Analyse par Voyage dans le Temps) qui permet de conserver l'intégralité du flot d'exécution d'un programme. On peut donc regarder les embranchements empruntés par un programme selon les arguments donnés au programme, les instructions effectuées, etc. Cette technique combine les méthodologies de l'analyse statique et dynamique.

Grâce à cela, nous n'avons pas eu besoin de comprendre toute les fonctions du maliciel, seulement celles que nous rencontrons dans notre analyse. Après l'initialisation, on voit que le programme teste si les arguments `argv` sont absents ou si l'importation des DLL a échoué. Si c'est le cas pour ces deux conditions, le programme quitte :

```
00401a7a        void* uExitCode;
00401a7a
00401a7a        if (!argv || eax_2 < 0)
00401a50            uExitCode = 1;
```

Sinon, le programme commence à repérer l'environnement afin de s'y installer. Il va tout d'abord repérer si une fenêtre Windows ayant pour nom de classe `snxhk_border_mywnd` existe :

```
00401a9a            if (DetectSnxhkWindow())
00401a9c                data_410514 = 0x27f3;
```

Le code complet de cette fonction peut être trouvé en Annexe 9. Cette fonction fait appel à la fonction `EnumWindows` afin de chercher parmi toutes les fenêtres si l'une d'elles a pour nom de classe la chaîne de caractères recherchée. Dans notre cas, cette fonction est utilisée pour savoir si une des fenêtres ouvertes a pour nom de classe `snxhk_border_mywnd`.

Cette technique peut être qualifiée d'anti-VM ou tout du moins permettre d'empêcher le maliciel de se lancer dans un environnement spécifique. Malheureusement, nous n'avons pas trouvé de source fiable concernant l'environnement ciblé par cette technique.

Il continue à effectuer quelques tests sur le nombre d'arguments donnés à l'exécutable et les valeurs des arguments. Dans notre cas, sans passer aucun argument, nous sommes arrivés à l'embranchement suivant :

```
00401aed            else if (eax_6 == 0x43)
00401b25                    uExitCode = sub_00403ef7(2);
```

On arrive par la suite à une fonction qui nous intéresse, le code complet de la fonction `sub_403ef7` peut être trouvé en Annexe 10 et sous forme renommée en Annexe 11. Elle implémente des fonctions d'anti-VM qui sont les suivantes [1] :

- `sub_4033fc()` \ `avm_in_vmware_version()`
- `sub_40349a()` \ `avm_in_vmware_memory_size`
- `sub_4035b6()` \ `avm_devices_check()`
- `sub_40385e()` \ `avm_process_check()`
- `sub_40336e()` \ `call_cpuid()`

Chacune de ces fonctions retourneront 0 si elles ne détectent pas un environnment virtuel et une valeur supérieur dans le cas inverse. Nous allons par la suite étudier toutes ces fonctions.

IV.2.2.1 `sub_4033fc()` \ `avm_in_vmware_version()`

La première fonction est une fonction d'anti-VM qui cible le gestionnaire de machine virtuelle *VMWare*.

Afin de communiquer avec l'hyperviseur, *VMware* utilise le port d'entrée/sortie `0x5658`. Pour réaliser cette communication, il faut utiliser l'instruction `in`, qui nécessite les registres `eax` et `edx`. Cette instruction copie la valeur lue à partir du port spécifié (contenu dans `edx`) vers `eax`.

Dans le cadre de *VMware*, le gestionnaire va intercepter tout appel à l'instruction `in` avec comme port d'entrée/sortie `0x5658`, le programme va donc placer cette valeur dans `edx`. `eax` va prendre la valeur magique `0x564D5868` qui va permettre à `VMware` d'authentifier la requête de communication. Avant de faire appel à l'instruction `in`, nous devons placer la commande que l'on souhaite envoyer à `VMware` dans le registre `ecx`.

Dans ce cas, la technique place la valeur `0xa` dans `ecx`. Comme on peut le voir à l'aide du header de la backdoor [42], elle essaie de récupérer des informations sur la version de `VMware`. Voici comment cette technique est implémentée en assembleur dans le maliciel :

```
mov     dx, 0x5658
mov     ecx, 0x564d5868
mov     eax, ecx
mov     ecx, 0xa
in      eax, dx
```

1. La fonction implémente aussi des techniques d'anti-analyse que nous n'aborderons pas

Si la machine virtuelle dans laquelle cette instruction est exécutée n'est pas générée par *VMware*, alors l'instruction provoquera une erreur :

```
0x403439    66 ba 58 56              mov  dx, 0x5658
0x40343d    b9 68 58 4d 56           mov  ecx, 0x564d5868
0x403442    8b c1                    mov  eax, ecx
0x403444    b9 0a 00 00 00           mov  ecx, 0xa
0x403449    ed                       general protection while executing in  eax, dx
```

FIGURE IV.2.1 – Erreur de l'exécution de l'instruction `in`

Sans erreur, la valeur magique `0x564d5868` qui était dans `eax` sera déplacée dans `ebx`. Le code de la technique présentée va donc comparer le contenu de `ebx` avec la valeur magique. S'il sont égaux alors l'instruction a fonctionné et le programme se trouve dans une machine virtuelle de *VMware*, il va donc retourner une valeur différente de 0 (`0x6e`).

IV.2.2.2 `sub_40349a()` \ `avm_in_vmware_memory_size()`

Cette fonction reprend le principe de la technique précédente, le code va faire appel à l'instruction `in` mais cette fois-ci, il va récupérer la mémoire utilisée (`ecx` = 0x14) [42] :

```
mov     dx, 0x5658
mov     ecx, 0x564d5868
mov     eax, ecx
mov     ecx, 0x14
in      eax, dx
```

Si cette instruction réussit, on récupérera la mémoire, elle va ensuite être comparée avec une borne inférieure (`0x10 = 16 Ko`) et une borne supérieure (`0x2000 = 8192 Ko`). Si la taille récupérée est comprise entre ces bornes, alors la fonction retournera la taille récupérée, sinon 0. Si cette instruction échoue, la fonction renverra 0.

IV.2.2.3 `sub_4035b6()` \ `avm_devices_check()`

Cette fonction va remplir deux listes d'octets écrits en brut dans le code, ces octets formeront une chaîne de caractères qui sera déchiffrée plus tard dans le code. On verra que la première liste `var_8c` est une liste de descriptions de périphériques que les gestionnaires de machine virtuelles peuvent utiliser (VMware, VirtualBox, etc.). La deuxième `var_28` est une liste de noms de classes de périphériques utilisés par ces mêmes gestionnaires de machines virtuelles. La liste de ces descriptions et classes peuvent être retrouvée en Annexe 12.

Puis, la fonction va utiliser la fonction `SetupDiGetClassDevsA` de l'API Windows qui renverra un *handle* d'un ensemble d'information de périphériques, voici sa syntaxe :

```
WINSETUPAPI HDEVINFO SetupDiGetClassDevsA(
  const GUID *ClassGuid,
  PCSTR       Enumerator,
  HWND        hwndParent,
  DWORD       Flags
);
```

Les trois premiers paramètres seront ignorés dans le code, le quatirème, nommé `Flags`, va être mis à `6`, ce qui correspond, d'après l'en-tête de la librairie `SetupAPI.h`[2] aux valeurs suivantes :

```
#define DIGCF_PRESENT           0x00000002
#define DIGCF_ALLCLASSES        0x00000004
```

Ces valeurs sont définies de la manière suivante dans la documentation :

> DIGCF_ALLCLASSES Retournez la liste des appareils installés pour toutes les classes d'installation d'appareil ou toutes les classes d'interface d'appareil.
> DIGCF_PRESENT Retournez uniquement les appareils actuellement présents dans un système.

2. https://github.com/tpn/winsdk-10/blob/master/Include/10.0.16299.0/um/SetupAPI.h

Cette fonction va donc retourner un *handle* vers un ensemble d'informations de tous les périphériques sans distinction de leur classe.

La fonction `sub_4035b6()` va ensuite utiliser une autre fonction de l'API Windows, `SetupDiEnumDeviceInfo`, qui permet de récupérer les informations de tous les périphériques, ces informations seront placées dans une structure `PSP_DEVINFO_DATA`. Voici la syntaxe de cette dernière :

```
WINSETUPAPI BOOL SetupDiEnumDeviceInfo(
  [in]  HDEVINFO         DeviceInfoSet,
  [in]  DWORD            MemberIndex,
  [out] PSP_DEVINFO_DATA DeviceInfoData
);
```

Une fois la structure `DeviceInfoData` remplie par la fonction, le programme entre dans sa phase finale qui est une boucle réalisant deux actions différentes :

- Premièrement, le programme va récupérer la propriété de description du périphérique en utilisant la fonction `SetupDiGetDeviceRegistryPropertyA`[3] avec l'identifiant de propriété 0, la syntaxe de cette fonction se trouve en Annexe 13. Puis, il va déchiffrer les chaînes de caractères des descriptions de périphérique (`var_8c`) et tester, à l'aide de `StrStrIA`, si une de ces chaînes est comprise dans la description d'un des périphériques récupérés grâce à `SetupDiEnumDeviceInfo` alors on retourne 1, sinon 0.
- Deuxièmement, le programme aura le même comportement mais cette fois, il va récupérer la propriété de classe du périphérique et déchiffrer les chaîne de caractères des classes de périphérique (`var_28`). La comparaison et le retour sont identiques à l'action précédente.

Le code complet de cette fonction se trouve en Annexe 14.

IV.2.2.4 `sub_40385e()` \ `avm_process_check()`

Comme la fonction précédente, celle-ci déchiffre une liste de chaînes de caractères, dans ce cas des noms de processus disponibles en Annexe 15. Par la suite, elle récupère des informations sur le processus actuel, pour cela elle utilise la fonction `CreateToolhelp32Snapshot` qui va faire une capture du processus ciblé et de son environnement, sa syntaxe est la suivante :

```
HANDLE CreateToolhelp32Snapshot(
  [in] DWORD dwFlags,
  [in] DWORD th32ProcessID
);
```

Le maliciel l'appel comme suit :

```
HANDLE hObject = CreateToolhelp32Snapshot(2, GetCurrentProcessId());
```

Le premier argument `dwFlags` qui permet d'établir la portion du système que l'on souhaite "capturer" est mis à `2`, ce qui correspond à la constante `TH32CS_SNAPPROCESS` qui est décrit dans la documentation comme suit :

> Includes all processes in the system in the snapshot. To enumerate the processes, see Process32First.

Une fois la capture effectuée, le programme va récupérer le premier processus rencontré dans la *snapshot* à l'aide de la fonction `Process32First` :

```
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

Cette fonction va placer les informations sur le processus dans la structure `LPPROCESSENTRY32 lppe`. Une fois le premier processus obtenu, le programme rentre dans une boucle `do while`. Cette boucle va récupérer le prochain processus à étudier à l'aide de la fonction `Process32Next` :

3. Cette fonction va récupérer les propriétés d'un périphérique demandé par l'utilisateur.

```
BOOL Process32First(
  [in]      HANDLE           hSnapshot,
  [in, out] LPPROCESSENTRY32 lppe
);
```

Dans la boucle `do while`, on retrouve deux autres boucles `for` :

- La première boucle `for` parcourt la liste des noms de processus précédemment déchiffrés (`var_14c`) et les compare avec le nom du processus actuel (`var_110`). La comparaison est réalisée par la fonction `__beep` qui est un alias vers `lstrcmpiA`, cette fonction permet de comparer deux chaînes de caractères et retourne une valeur négative si la première chaîne est inférieure à la deuxième, 0 si les chaînes sont égales et une valeur positive sinon.
- La deuxième boucle reprend le même principe, mais utilise `StrStrIA` pour savoir si une chaîne de caractères est incluse dans l'autre.

À l'issue de la boucle `do while`, la fonction renverra 1 si l'un des processus a été identifié comme étant lié à de l'émulation. Le code complet de la fonction peut être trouvé en Annexe 16.

IV.2.2.5 `sub_40336e()` \ `call_cpuid()`

Cette technique ne fait pas partie de la condition (`if`) permettant de détecter une machine virtuelle. Elle est appelée à l'issue de l'échec de la détection d'un environnement émulé (virtualisé). De plus, elle n'est pas utilisée avec une comparaison ou un test, nous n'avons donc pas réussi à expliquer la présence de cette dernière. Néanmoins, elle implémente une technique qui peut constituer une protection anti-VM utilisant `cpuid`II.2.1 :

- Tout d'abord, le programme va détecter la chaîne d'identifiant du processeur en utilisant `cpuid` avec `eax = 0` et qui sera comparée par la suite à la chaîne de caractères `GenuineIntel` à l'aide de la fonction `lstrcmpiA`. Il semble que cela soit utilisé afin de ne cibler que certaines machines. En effet, la plupart des émulateur émulent un processeur avec la même chaîne d'identifiant du processeur que celui présent sur la machine hôte.
- La technique d'anti-VM va également utiliser `cpuid`, mais avec `eax = 0x1` (cf II.2.1.1) afin de déterminer si un hyperviseur est présent.

Le code de cette fonction est disponible en Annexe 17.

IV.2.3 Résultat de l'analyse

Comme nous avons pu le constater, la diversité de protections anti-VM dans le maliciel *QakBot* est suffisante pour détecter une bonne partie des gestionnaires de machines virtuelles, bien qu'il semble se concentrer sur *VMware*. Toutefois, ces méthodes restent assez basiques et ciblent des solutions de virtualisation simples et peu défendues. Des machines virtuelles suffisamment configurées permettent d'éviter la plupart de ces protections anti-VM.

Pour conclure, *QakBot* utilise un arsenal de protections anti-VM varié et efficace face à des outils standards, notamment *VMware*, mais il reste cependant vulnérable à des environnements plus sophistiqués.

Conclusion

L'objectif de ce mémoire était d'étudier les mécanismes mis en place par les logiciels malveillants afin de détecter et de contourner les environnements virtualisés. Comme nous l'avons vu, il existe de nombreuses techniques. La plupart se basent sur des paramètres du système qui semblent anormaux pour une machine physique. La plupart des méthodes présentées sont implémentées à l'aide de l'API Windows, elles ne sont donc utilisables qu'au sein d'un environnement Windows. Il existe des équivalents pour chacune de ces techniques dans d'autres systèmes.

En dépit de la grande diversité de techniques, le nombre de protections anti-VM demeure assez faible dans la plupart des logiciels malveillants. Ces techniques sont souvent rudimentaires, la plupart du temps, elles sont basées sur des instructions en assembleur telles que `cpuid` ou `rdtsc`. L'exemple de *QakBot* confirme cette idée : ce logiciel malveillant implémente des tests simples (utilisation du port I/O de *VMware*, cpuid, etc.).

Bien que simples, les protections anti-VM de *QakBot* lui permettent de se protéger efficacement contre des environnements virtualisés spécifiques. Il est également possible que la simplicité de ces techniques suive la logique d'une campagne particulière, *QakBot* n'aura donc pas vocation à être réutilisé sous cette forme. Il existe aussi d'autres méthodes qui vont retarder le moment de l'utilisation des protections anti-VM le plus possible. Par exemple, la campagne d'hameçonnage réalisée par APT29 [43] utilise un logiciel nommé `WINELOADER` qui permet notamment d'utiliser des techniques d'anti-VM. Ce logiciel est téléchargé par un autre composant nommé `GRAPELOADER` qui va faire appel à un serveur afin de récupérer le `WINELOADER`. Cette méthode permet d'empêcher toute analyse du logiciel sans vérification de l'environnement au préalable.

Nous avons présenté des méthodes efficaces dans le but de contrer ces protections anti-VM. Cependant, ces contre-mesures ne prennent pas en compte d'autres facteurs souvent présents dans ces logiciels, tels que des techniques d'anti-débogage qui vont contrer l'utilisation d'un débogueur. Comme montré au cours de ce mémoire, les logiciels malveillants contiennent en moyenne plus de techniques d'anti-débogage que de protections anti-VM.

Les protections anti-VM peuvent toutefois être à double tranchant. Tout d'abord, si elles sont mal définies, ces techniques peuvent amener le logiciel à ignorer des machines ayant une configuration inhabituelle. Par exemple, bien que rare, il est encore possible de trouver des machines ayant moins de 512 Go de stockage. De plus, ces techniques vont aussi faire en sorte d'ignorer toutes les machines virtuelles, ce qui réduira grandement la surface d'attaque du maliciel. Beaucoup d'entreprises utilisent des parcs de machines virtuelles afin d'héberger des postes de travail, des serveurs, etc. Ces machines restent intéressantes à attaquer bien qu'elles soient virtualisées.

Pour conclure, les protections anti-VM implémentées par les maliciels sont des éléments essentiels à la pérennité de ces logiciels, elles empêchent l'analyse de ces derniers et permettent d'augmenter leur durée de vie. Bien que parfois excessives, parfois même contre-productives, ces techniques sont fiables lorsqu'elles fonctionnent. Elles forcent les analystes à développer des outils et des environnements toujours plus performants et sûrs.

Annexes

Annexe 1 — Script permettant de retrouver le code assembleur de la traduction du code ARM

```
import os

string = "8b5df085db0f8c1d000000c645f401c7450005000000c7453c58000100488bfdff15120000
00ffe0488d0514ffffffe9e4feffff"

with open("add_x86_64_bytes", "wb") as f:
    f.write(bytes.fromhex(string))

os.system("ndisasm -b 64 add_x86_64_bytes > add_x86_64")
```

Annexe 2 — Fonction `SERVICES_NAME_LIST` et `belong_to` — Détection des services liés à des machines virtuelles

```
static char *SERVICES_NAME_LIST[] = {
/* VM services used in QEMU */
"QUEM-GA", "VirtioFsSvc", "BalloonService", "spice-agent", "vmi", "vds",

/* VM services used in VBOX */
"VBoxSVC", "VBoxDrv", "VBoxUSBMon", "VBoxNetFlt", "VBoxNetAdp",

/* VM services used in VMWare */
"VGAuthService", "vmvss", "vm3dservice", "VMTools"};

bool belong_to(char *service_name)
{
    for (int i = 0; i < NB_SERVICES; i++)
    {
        if (strstr(service_name, SERVICES_NAME_LIST[i]))
        {
            return true;
        }
    }
    return false;
}
```

Annexe 3 — Liste de clefs de registre standard pour les gestionnaires de machines virtuelles

```
/* Sample of regkey path */
LPCWSTR RegValuePath[] = {

  /* VMWare */
    L"SOFTWARE\\VMware, Inc.\\VMware Tools",

  /* VirtualBox */
    L"SOFTWARE\\Oracle\\VirtualBox Guest Additions",
    L"HARDWARE\\ACPI\\DSDT\\VBOX__",
    L"HARDWARE\\ACPI\\FADT\\VBOX__",
    L"HARDWARE\\ACPI\\RSDT\\VBOX__",
    L"SYSTEM\\ControlSet001\\Services\\VBoxGuest",
    L"SYSTEM\\ControlSet001\\Services\\VBoxMouse",
    L"SYSTEM\\ControlSet001\\Services\\VBoxService",
    L"SYSTEM\\ControlSet001\\Services\\VBoxSF",
    L"SYSTEM\\ControlSet001\\Services\\VBoxVideo",
    L"HKEY_LOCAL_MACHINE\\HARDWARE\\ACPI\\DSDT\\VBOX__",

    /* QEMU and other software */
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\QEMU Guest Agent VSS
Provider",
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\QEMU-GA",
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Services\\BALLOON",
    L"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Enum\\ROOT\\vdrvroot"
};
```

Annexe 4 — Fonction `qemu_firmware_ACPI` du maliciel `al-khaser`

```
BOOL qemu_firmware_ACPI()
{
        BOOL result = FALSE;

        PDWORD tableNames = static_cast<PDWORD>(malloc(4096));

        if (tableNames) {
                SecureZeroMemory(tableNames, 4096);
                DWORD tableSize =
enum_system_firmware_tables(static_cast<DWORD>('ACPI'), tableNames, 4096);

                // API not available
                if (tableSize == -1)
                        return FALSE;

                DWORD tableCount = tableSize / 4;
                if (tableSize < 4 || tableCount == 0)
                {
                        result = TRUE;
                }
                else
                {
                        const char* strings[] = {
                                "FWCF",                 // fw_cfg name
                                "QEMU0002",         // fw_cfg HID/CID
                                "BOCHS",         // OEM ID
                                "BXPC"                  // OEM Table ID
                        };

                        for (DWORD i = 0; i < tableCount; i++)
                        {
                                DWORD tableSize = 0;
                                PBYTE table =
get_system_firmware(static_cast<DWORD>('ACPI'), tableNames[i], &tableSize);

                                if (table) {

                                        for (DWORD j = 0; j < sizeof(strings) /
sizeof(char*); j++)
                                        {
                                                if
(!find_str_in_data((PBYTE)strings[j], strlen(strings[j]), table, tableSize))
                                                {
                                                        free(table);
                                                        result = TRUE;
                                                        goto out;
                                                }
                                        }

                                        free(table);
                                }
                        }

                        DWORD tableSize = 0;
                        PBYTE table =
get_system_firmware(static_cast<DWORD>('ACPI'), static_cast<DWORD>('PCAF'),
&tableSize);

                        if (table) {
                                if (tableSize < 45)
```

```
                                {
                                        return FALSE; // Corrupted table
                                }

                                // Format: [HexOffset DecimalOffset ByteLength]
FieldName  : FieldValue (in hex)
                                //                      [02Dh      0045          001h
   ]         PM Profile : 00 [Unspecified] - hardcoded in QEMU src
                                if ((BYTE) table[45] == (BYTE)0)
                                {
                                        result = TRUE;
                                }

                                free(table);
                        }
                }
        out:
                free(tableNames);
        }
        return result;
}
```

Annexe 5 — Fonction `get_system_firmware` — récupération des tables micrologicielles

```
PBYTE get_system_firmware(_In_ DWORD signature, _In_ DWORD table,
                                                    _Out_ PDWORD pBufferSize)
{
        if (!API::IsAvailable(API_IDENTIFIER::API_GetSystemFirmwareTable)) {
                return NULL;
        }

        DWORD bufferSize = 4096;
        PBYTE firmwareTable = static_cast<PBYTE>(malloc(bufferSize));

        if (firmwareTable == NULL)
                return NULL;

        SecureZeroMemory(firmwareTable, bufferSize);

        auto GetSystemFirmwareTable = static_cast<pGetSystemFirmwareTable>(
                API::GetAPI(API_IDENTIFIER::API_GetSystemFirmwareTable));

        DWORD resultBufferSize =
                GetSystemFirmwareTable(signature, table, firmwareTable, bufferSize);
        if (resultBufferSize == 0) {
                printf("First call failed :(\n");
                free(firmwareTable);
                return NULL;
        }

        // if the buffer was too small, realloc and try again
        if (resultBufferSize > bufferSize) {
                PBYTE tmp;

                tmp = static_cast<BYTE *>(realloc(firmwareTable, resultBufferSize));
                if (tmp) {
                        firmwareTable = tmp;
                        SecureZeroMemory(firmwareTable, resultBufferSize);
                        if (GetSystemFirmwareTable(signature, table, firmwareTable,

resultBufferSize) == 0) {
                                printf("Second call failed :(\n");
                                free(firmwareTable);
                                return NULL;
                        }
                }
        }

        *pBufferSize = resultBufferSize;
        return firmwareTable;
}
```

Annexe 6 — Fonction `enumerate_object_directory` — Permet de retrouver la liste des noms des objets présent dans un répertoire du gestionnaire d'objets.

```
std::vector<wchar_t *> *enumerate_object_directory(const wchar_t *path)
{
        if (!API::IsAvailable(API_NtOpenDirectoryObject) ||
                !API::IsAvailable(API_NtQueryDirectoryObject)) {
                return nullptr;
        }

        UNICODE_STRING usPath = { 0 };
        usPath.Buffer = const_cast<wchar_t *>(path);
        usPath.Length = static_cast<USHORT>(lstrlenW(path) * sizeof(wchar_t));
        usPath.MaximumLength = usPath.Length;

        OBJECT_ATTRIBUTES objAttr = { 0 };
        InitializeObjectAttributes(&objAttr, &usPath, OBJ_CASE_INSENSITIVE, NULL,
                                                                NULL);

        auto ntOpenDirectoryObject = static_cast<pNtOpenDirectoryObject>(
                API::GetAPI(API_NtOpenDirectoryObject));
        auto ntQueryDirectoryObject = static_cast<pNtQueryDirectoryObject>(
                API::GetAPI(API_NtQueryDirectoryObject));

        const int DIRECTORY_QUERY = 0x0001;
        HANDLE hDirectory = 0;
        NTSTATUS status =
                ntOpenDirectoryObject(&hDirectory, DIRECTORY_QUERY, &objAttr);
        if (status != 0) {
                // printf("\nNTODO failed: %x\n", status);
                return nullptr;
        }

        auto pObjDirInfo =
                static_cast<OBJECT_DIRECTORY_INFORMATION *>(calloc(0x800, 1));
        ULONG returnedLength = 0;
        ULONG context = 0;
        auto results = new std::vector<wchar_t *>();
        while (ntQueryDirectoryObject(hDirectory, pObjDirInfo, 0x800, TRUE, FALSE,
                                                                &context,
&returnedLength) == 0 &&
                   returnedLength > 0) {
                // wprintf(L"\nobject: %s\n", pObjDirInfo->Name.Buffer);
                wchar_t *name = static_cast<wchar_t *>(
                        calloc(pObjDirInfo->Name.Length + 1, sizeof(wchar_t)));
                memcpy(name, pObjDirInfo->Name.Buffer,
                           pObjDirInfo->Name.Length * sizeof(wchar_t));
                results->push_back(name);
        }

        free(pObjDirInfo);

        return results;
}
```

Annexe 7 — Fichier `Generic.h` de `al-khaser` — En-tête contenant toutes les fontions d'anti-VM génériques.

```
VOID loaded_dlls();
VOID known_file_names();
VOID known_usernames();
VOID known_hostnames();
VOID other_known_sandbox_environment_checks();
BOOL NumberOfProcessors();
BOOL idt_trick();
BOOL ldt_trick();
BOOL gdt_trick();
BOOL str_trick();
BOOL number_cores_wmi();
BOOL disk_size_wmi();
BOOL setupdi_diskdrive();
BOOL mouse_movement();
BOOL lack_user_input();
BOOL memory_space();
BOOL dizk_size_deviceiocontrol();
BOOL disk_size_getdiskfreespace();
BOOL accelerated_sleep();
BOOL cpuid_is_hypervisor();
BOOL cpuid_hypervisor_vendor();
BOOL serial_number_bios_wmi();
BOOL model_computer_system_wmi();
BOOL manufacturer_computer_system_wmi();
BOOL current_temperature_acpi_wmi();
BOOL process_id_processor_wmi();
BOOL power_capabilities();
BOOL hybridanalysismacdetect();
BOOL cpu_fan_wmi();
BOOL caption_video_controller_wmi();
BOOL query_license_value();
BOOL cachememory_wmi();
BOOL physicalmemory_wmi();
BOOL memorydevice_wmi();
BOOL memoryarray_wmi();
BOOL voltageprobe_wmi();
BOOL portconnector_wmi();
BOOL smbiosmemory_wmi();
BOOL perfctrs_thermalzoneinfo_wmi();
BOOL cim_memory_wmi();
BOOL cim_numericsensor_wmi();
BOOL cim_physicalconnector_wmi();
BOOL cim_sensor_wmi();
BOOL cim_slot_wmi();
BOOL cim_temperaturesensor_wmi();
BOOL cim_voltagesensor_wmi();
BOOL pirated_windows();
BOOL registry_services_disk_enum();
BOOL registry_disk_enum();
BOOL number_SMBIOS_tables();
BOOL firmware_ACPI();
BOOL hosting_check();
VOID looking_glass_vdd_processes();
```

Annexe 8 — Fonction `ExecWMIQuery` — Permet d'exécuter une requête WMI.

```
BOOL ExecWMIQuery(IWbemServices **pSvc, IWbemLocator **pLoc,
                                IEnumWbemClassObject **pEnumerator, const TCHAR
*szQuery)
{
        // Execute WMI query
        BSTR strQueryLanguage = SysAllocString(OLESTR("WQL"));
        BSTR strQuery = SysAllocString(szQuery);

        BOOL bQueryResult = TRUE;

        if (strQueryLanguage && strQuery) {

                HRESULT hres = (*pSvc)->ExecQuery(strQueryLanguage, strQuery,
                      WBEM_FLAG_FORWARD_ONLY |
                          WBEM_FLAG_RETURN_IMMEDIATELY,
                      NULL, pEnumerator);

                if (FAILED(hres)) {
                        bQueryResult = FALSE;
                        print_last_error(_T("ExecQuery"));
                        (*pSvc)->Release();
                        (*pLoc)->Release();
                        CoUninitialize();
                }
        }

        if (strQueryLanguage)
                SysFreeString(strQueryLanguage);
        if (strQuery)
                SysFreeString(strQuery);

        return bQueryResult;
}
```

Annexe 9 — Fonctions `DetectSnxhkWindow` et `EnumWindows_find_match_class` - Teste la présence d'une fenêtre windows ayant un certains nom de classe

```
int32_t DetectSnxhkWindow()
{
    void* decrypted_targeted_name = decrypt_ressource_into_buf(0x27ca);
    void* targeted_name = decrypted_targeted_name;
    int32_t match_found = 0;
    int32_t match_confirmed = 0;
    int32_t name_length = lstrlenA(decrypted_targeted_name);

    if (EnumWindows(EnumWindows_find_match_class, &targeted_name))
    {
        custom_free(&targeted_name);

        if (!match_found || match_confirmed > 0)
            return 1;
    }
    else
    {
        GetLastError();
        custom_free(&targeted_name);
    }

    return 0;
}

int32_t __stdcall EnumWindows_find_match_class(HWND hwnd, int32_t* user_data)
{
    uint8_t className[0x100] = {0};
    memset(className, 0, sizeof(className));

    GetClassNameA(hwnd, (LPSTR)className, 0x100);

    user_data[3] += 1;

    int32_t i = 0;
    int32_t length = user_data[1];  // Length of the string to compare
    uint8_t* target = (uint8_t*)*(void**)user_data;

    if (length > 0)
    {
        int32_t offset = data_410664;  // Likely a static XOR or transformation key

        do
        {
            uint8_t transformed_target =
                *(uint8_t*)(*(target + i) + offset);
            uint8_t transformed_classname =
                *(uint8_t*)(&className[i] + offset);

            if (transformed_target != transformed_classname)
                return 1;

            i += 1;
        } while (i < length);
    }

    user_data[2] = 1;  // Mark a match
    return 1;
}
```

Annexe 10 — Fonction `sub_403ef7` — Implémente un certains nombre de protection anti-VM.

```
int32_t __fastcall sub_403ef7(void* arg1)
{
    void* var_8 = arg1;
    void* var_c = arg1;

    while (false)
        /* nop */

    if (sub_4033fc() <= 0 && sub_40349a() <= 0 && sub_4035b6() <= 0
        && sub_40385e() <= 0 && sub_403bdf() <= 0 && sub_403d22() <= 0)
    {
        sub_403deb();
        sub_403e6f();
        sub_40336e();

        while (false)
            /* nop */

        return 0;
    }

    int32_t result = 1;
    var_8 = decrypt_ressource_into_buf(0x1ac6);

    while (false)
        /* nop */

    if (var_8)
    {
        if (test_file_exists(var_8))
        {
            while (false)
                /* nop */

            result = 0;
            sub_4033fc();
            sub_40349a();
            sub_4035b6();
            sub_40385e();
            sub_403bdf();
            sub_403d22();
            sub_403deb();
            sub_403e6f();
            sub_40336e();
        }

        custom_free(&var_8);
    }

    if (sub_403aa0() > 0)
        result = 0;

    return result;
}
```

Annexe 11 — Fonction `sub_403ef7` renommée — Implémente un certains nombre de protection anti-VM.

```
int32_t __fastcall anti_vm_and_analysis_checks(void* arg1)
{
    void* var_8 = arg1;
    void* var_c = arg1;

    while (false)
        /* nop */

    if (avm_in_vmware_version() <= 0 && avm_in_vmware_memory_size() <= 0
        && avm_devices_check() <= 0 && avm_process_check() <= 0
        && dll_injected_check() <= 0 && filename_check() <= 0)
    {
        sub_403deb();
        sub_403e6f();
        call_cpuid();

        while (false)
            /* nop */

        return 0;
    }

    int32_t result = 1;
    var_8 = decrypt_ressource_into_buf(0x1ac6);

    while (false)
        /* nop */

    if (var_8)
    {
        if (test_file_exists(var_8))
        {
            while (false)
                /* nop */

            result = 0;
            avm_in_vmware_version();
            avm_in_vmware_memory_size();
            avm_devices_check();
            avm_process_check();
            dll_injected_check();
            filename_check();
            sub_403deb();
            sub_403e6f();
            call_cpuid();
        }

        custom_free(&var_8);
    }

    if (sub_403aa0() > 0)
        result = 0;

    return result;
}
```

Annexe 12 — Listes des descriptions et noms de classe testées par *QakBot*

```
Descriptions :

VMware Pointing
VMware Accelerated
VMware SCSI
VMware SVGA
VMware Replay
VMware server memory
CWSandbox
Virtual HD
QEMU
Red Hat VirtIO
srootkit
VMware VMaudio
VMware Vista
VBoxVideo
VBoxGuest


Noms de classe :

vmxnet
vmscsi
VMAUDIO
vmdebug
vm3dmp
vmrawdsk
vmx_svga
ansfltr
sbtisht
```

Annexe 13 — Syntaxe de la fonction `SetupDiGetDeviceRegistryPropertyA`

```
WINSETUPAPI BOOL SetupDiGetDeviceRegistryPropertyA(
  [in]            HDEVINFO         DeviceInfoSet,
  [in]            PSP_DEVINFO_DATA DeviceInfoData,
  [in]            DWORD            Property,
  [out, optional] PDWORD           PropertyRegDataType,
  [out, optional] PBYTE            PropertyBuffer,
  [in]            DWORD            PropertyBufferSize,
  [out, optional] PDWORD           RequiredSize
);
```

Annexe 14 — Fonction `avm_devices_check`

```
int32_t avm_devices_check()
{
    int32_t result = 0;
    int32_t descriptions;
    __builtin_memcpy(&descriptions,

"\xd4\x05\x00\x00\x38\x02\x00\x00\xc4\x02\x00\x00\x5e\x1d\x00\x00\xee\x05\x00\x00\x"

"a1\x26\x00\x00\xc1\x32\x00\x00\xf5\x20\x00\x00\x17\x35\x00\x00\xdf\x22\x00\x00\x54"
    "28\x00\x00\x57\x25\x00\x00\xf2\x01\x00\x00\xe5\x1d\x00\x00\x62\x36\x00\x00",
        0x3c);
    int32_t classes;
    __builtin_memcpy(&classes,

"\x9b\x0d\x00\x00\xfc\x26\x00\x00\x20\x02\x00\x00\x3a\x19\x00\x00\x68\x00\x00\x00\x"
    "c5\x26\x00\x00\x1a\x00\x00\x00\x05\x0c\x00\x00\xaa\x2e\x00\x00",
        0x24);

    HDEVINFO DeviceInfoSet = SetupDiGetClassDevsA(nullptr, nullptr, nullptr, 6);

    if (DeviceInfoSet == 0xffffffff)
    {
        while (false)
            /* nop */

        return 0xffffffff;
    }

    struct SP_DEVINFO_DATA DeviceInfoData;
    DeviceInfoData.cbSize = 0x1c;
    uint32_t MemberIndex = 0;

    while (SetupDiEnumDeviceInfo(DeviceInfoSet, MemberIndex, &DeviceInfoData))
    {
        uint8_t* var_94 = nullptr;
        uint8_t* var_98 = nullptr;
        var_94 = retrieve_property_device(DeviceInfoSet, &DeviceInfoData, 0);

        if (var_94)
        {
            for (int32_t i = 0; i < 0xf; i += 1)
            {
                void* var_9c = decrypt_ressource_into_buf((&descriptions)[i]);

                if (var_9c)
                {
                    if (StrStr(var_94, var_9c))
                    {
                        while (false)
                            /* nop */

                        result = 1;
                        custom_free(&var_9c);
                        break;
                    }

                    custom_free(&var_9c);
                }
            }
```

```
            custom_free_(&var_94, 0);
        }

        var_98 = retrieve_property_device(DeviceInfoSet, &DeviceInfoData, 4);

        if (var_98)
        {
            for (int32_t i_1 = 0; i_1 < 9; i_1 += 1)
            {
                void* var_a0 = decrypt_ressource_into_buf((&classes)[i_1]);

                if (var_a0)
                {
                    if (StrStr(var_98, var_a0))
                    {
                        while (false)
                            /* nop */

                        result = 1;
                        custom_free(&var_a0);
                        break;
                    }

                    custom_free(&var_a0);
                }
            }

            custom_free_(&var_98, 0);
        }

        if (result > 0)
            break;

        if (result > 0)
            break;

        MemberIndex += 1;
    }

    SetupDiDestroyDeviceInfoList(DeviceInfoSet);

    return result;
}
```

Annexe 15 — Listes des processus testés par *QakBot*

```
Processus :

Fiddler.exe
sample.exe
sample.exe
runsample.exe
lordpe.exe
regshot.exe
Autoruns.exe
dsniff.exe
VBoxTray.exe
HashMyFiles.exe
ProcessHacker.exe
Procmon.exe
Procmon64.exe
netmon.exe
vmtoolsd.exe
vm3dservice.exe
VGAuthService.exe
pr0c3xp.exe
CFF Explorer.exe
dumpcap.exe
Wireshark.exe
idaq.exe
idaq64.exe
TPAutoConnect.exe
ResourceHacker.exe
vmacthlp.exe
OLLYDBG.EXE
windbg.exe
bds-vision-agent-nai.exe
bds-vision-apis.exe
bds-vision-agent-app.exe
MultiAnalysis_v1.0.294.exe
x32dbg.exe
VBoxService.exe
Tcpview.exe
```

Annexe 16 — Fonction `avm_process_check`

```
int32_t avm_process_check()
{
    int32_t result = 0;
    int32_t var_144 = 0xffffffff;
    int32_t* var_c = nullptr;
    int32_t* var_8 = nullptr;
    void* var_140 = nullptr;
    int32_t var_138 = 0;
    void* var_14c = nullptr;
    void* var_13c = nullptr;

    while (false)
        /* nop */

    var_140 = decrypt_ressource_into_buf(0x1ad8);
    void* eax_1 = decrypt_ressource_into_buf(0x1ad8);
    var_14c = parse(var_140, 0x3b, 0, &var_c);
    var_13c = parse(eax_1, 0x3b, 0, &var_8);
    custom_free(&var_140);
    custom_free(&var_140);

    if (!var_14c || !var_13c)
    {
        while (false)
            /* nop */

        return 0;
    }

    HANDLE hObject = CreateToolhelp32Snapshot(2, GetCurrentProcessId());

    if (hObject != 0xffffffff)
    {
        int32_t lppe;
        memset(&lppe, 0, 0x128);
        lppe = 0x128;

        if (Process32First(hObject, &lppe))
        {
            int32_t i;

            do
            {
                void var_110;

                for (int32_t j = 0; j < var_c; j += 1)
                {
                    if (!__beep(*(uint32_t*)((char*)var_14c + (j << 2)), &var_110))
                    {
                        while (false)
                            /* nop */

                        result = 1;
                        break;
                    }
                }

                if (!result)
                {
                    for (int32_t j_1 = 0; j_1 < var_8; j_1 += 1)
```

```
                    {
                        if (StrStr(&var_110, *(uint32_t*)((char*)var_14c + (j_1 <<
2))))
                        {
                            while (false)
                                /* nop */

                            result = 1;
                            break;
                        }
                    }
                }

                if (result)
                    break;

                i = Process32Next(hObject, &lppe);
            } while (i);

            CloseHandle(hObject);
        }
    }

    cleanup(&var_c, &var_14c);
    cleanup(&var_8, &var_13c);
    return result;
}
```

Annexe 17 — Fonction `cpuid_call()`

```
int32_t call_cpuid()

{
    int32_t var_c = 0;
    int32_t var_8 = 0;
    char string1 = 0;
    void var_2f;
    int32_t ecx;
    int16_t* edi;
    edi = __builtin_memset(&var_2f, 0, 0x1c);
    *(uint16_t*)edi = 0;
    edi[1] = 0;
    void* lpString2 = nullptr;

    while (false)
        /* nop */

    char* var_3c = &string1;
    int32_t edx;
    cpu_brand_string(&string1, edx, ecx, var_3c);
    int32_t eax_1;
    int32_t ecx_2;
    int32_t edx_1;
    int32_t ebx;
    eax_1 = __cpuid(1, var_3c);
    lpString2 = decrypt_ressource_into_buf(0x42c);

    if (lpString2)
    {
        if (ecx_2 == 1 && !lstrcmpiA(&string1, lpString2))
        {
            while (false)
                /* nop */

            int32_t var_c_1 = 1;
        }

        custom_free(&lpString2);
    }

    return 0;
}

...

int32_t __convention("regparm") cpu_brand_string(int32_t arg1,
  int32_t arg2, int32_t arg3, void* arg4)

{
    while (false)
        /* nop */

    int32_t eax;
    int32_t ecx;
    int32_t edx;
    int32_t ebx;
    eax = __cpuid(0, arg3);
    int32_t var_c = ebx;
    int32_t var_8 = ecx;
    int32_t var_10 = edx;
```

```
    memcpy(arg4, &var_c, 4);
    memcpy((char*)arg4 + 4, &var_10, 4);
    memcpy((char*)arg4 + 8, &var_8, 4);
    *(uint8_t*)((char*)arg4 + 0xc) = 0;
    return 0;
}
```

Bibliographie

[1] GOOGLE. *Qu'est-ce qu'une machine virtuelle ?* URL : https://cloud.google.com/learn/what-is-a-virtual-machine?hl=fr.

[2] Red HAT. *Un hyperviseur, qu'est-ce que c'est ?* 2025. URL : https://www.redhat.com/fr/topics/virtualization/what-is-a-hypervisor.

[3] WIKIBOOKS. *Les systèmes d'exploitation/Virtualisation et machines virtuelles*. URL : https://fr.wikibooks.org/wiki/Les_syst%C3%A8mes_d%27exploitation/Virtualisation_et_machines_virtuelles.

[4] QEMU. *QEMU*. URL : https://www.qemu.org/.

[5] PANDA-RE. *Dépôt GitHub de panda*. URL : https://github.com/panda-re/panda.

[6] QEMU. *Arm System emulator*. URL : https://www.qemu.org/docs/master/system/target-arm.html.

[7] QEMU. *System Emulation : Introduction*. URL : https://www.qemu.org/docs/master/system/introduction.html.

[8] QEMU. *TCG Emulation*. URL : https://www.qemu.org/docs/master/devel/index-tcg.html.

[9] Alexander GRAF. *Vortrag: QEMU's recompilation engine*. URL : https://chemnitzer.linux-tage.de/2012/vortraege/1062/.

[10] Formation DEVOPS. *QEMU*. URL : https://docker.uptime-formation.fr/KVM/Theorie-QEMU/.

[11] QEMU. *QEMU User space emulator*. URL : https://www.qemu.org/docs/master/user/main.html.

[12] QEMU. *Translator Internals*. URL : https://www.qemu.org/docs/master/devel/tcg.html.

[13] QEMU. *The memory API*. URL : https://www.qemu.org/docs/master/devel/memory.html.

[14] QEMU. *Qcow2 Image File Format*. URL : https://www.qemu.org/docs/master/interop/qcow2.html.

[15] Ceph FOUNDATION. *QEMU and Block Devices*. URL : https://docs.ceph.com/en/reef/rbd/qemu-rbd/.

[16] MICROSOFT. *Windows API index*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list.

[17] MICROSOFT. *GetSystemMetrics function (winuser.h)*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getsystemmetrics.

[18] PC TROUBLESHOOTING et MAINTENANCE. *EVOLUTION OF HARD DISK*. 2008. URL : https://pctroubleshootingandmaintenance.blogspot.com/2008/11/evolution-of-hard-disk.html.

[19] MICROSOFT. *GetDiskFreeSpaceExA function (fileapi.h)*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-getdiskfreespaceexa.

[20] MICROSOFT. *GlobalMemoryStatusEx function (sysinfoapi.h)*. 2024. URL : https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-globalmemorystatusex.

[21] MICROSOFT. *GetAdaptersInfo function (iphlpapi.h)*. 2021. URL : https://learn.microsoft.com/en-us/windows/win32/api/iphlpapi/nf-iphlpapi-getadaptersinfo.

[22] MICROSOFT. *GetSystemInfo, fonction (sysinfoapi.h)*. 2021. URL : https://learn.microsoft.com/fr-fr/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsysteminfo.

[23] MICROSOFT. *GetPwrCapabilities function (powerbase.h)*. 2024. URL : https://learn.microsoft.com/en-us/windows/win32/api/powerbase/nf-powerbase-getpwrcapabilities.

[24] MICROSOFT. *OpenSCManagerA function (winsvc.h)*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-openscmanagera.

[25] MICROSOFT. *Sécurité du service et droits d'accès*. 2023. URL : https://learn.microsoft.com/fr-fr/windows/win32/services/service-security-and-access-rights.

[26] MICROSOFT. *EnumServicesStatusA function (winsvc.h)*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-enumservicesstatusa.

[27] MICROSOFT. *RegOpenKeyExW, fonction (winreg.h)*. 2024. URL : https://learn.microsoft.com/fr-fr/windows/win32/api/winreg/nf-winreg-regopenkeyexw.

[28] MICROSOFT. *GetCursorPos, fonction (winuser.h)*. 2022. URL : https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getcursorpos.

[29] MICROSOFT. *GetForegroundWindow, fonction (winuser.h)*. 2024. URL : https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-getforegroundwindow.

[30] Félix CLOUTIER. *RDTSC — Read Time-Stamp Counter*. 2023. URL : https://www.felixcloutier.com/x86/rdtsc.

[31] Félix CLOUTIER. *SIDT — Store Interrupt Descriptor Table Register*. 2023. URL : https://www.felixcloutier.com/x86/sidt.

[32] stack OVERFLOW. *Red Pill detect virtualization*. 2017. URL : https://stackoverflow.com/questions/46267618/red-pill-detect-virtualization.

[33] Tom Liston et ED SKOUDIS. "On the Cutting Edge : Thwarting Virtual Machine Detection". In : (2006). URL : https://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf.

[34] Lorenzo Maffia et DARIO NISI† ET PLATON KOTZIAS ET GIOVANNI LAGORIO ET SIMONE AONZO ET DAVIDE BALZAROTTI†. "Longitudinal Study of the Prevalence of Malware Evasive Techniques". In : (2021). URL : https://arxiv.org/abs/2112.11289.

[35] Ping Chen et CHRISTOPHE HUYGENS ET LIEVEN DESMET ET WOUTER JOOSEN. "Advanced or Not? A Comparative Study of the Use of Anti-debugging and Anti-VM Techniques in Generic and Targeted Malware". In : (2016). URL : https://inria.hal.science/hal-01369566/document.

[36] *qemu-system-x86_64(1) - QEMU emulator*. URL : https://man7.org/linux/man-pages/man1/qemu-system-x86_64.1.html.

[37] MICROSOFT. *WinDbg Preview*. 2025. URL : https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools.

[38] AYOUBFAOUZI. *Al-Khaser v0.81*. 2025. URL : https://github.com/ayoubfaouzi/al-khaser.

[39] MICROSOFT. *EnumSystemFirmwareTables function (sysinfoapi.h)*. 2021. URL : https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-enumsystemfirmwaretables.

[40] MICROSOFT. *GetSystemFirmwareTable function (sysinfoapi.h)*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemfirmwaretable.

[41] MICROSOFT. *Windows Management Instrumentation*. 2023. URL : https://learn.microsoft.com/en-us/windows/win32/wmisdk/wmi-start-page.

[42] Inc. VMWARE. *backdoor_def.h*. 2023. URL : https://github.com/vmware/open-vm-tools/blob/stable-13.0.0/open-vm-tools/lib/include/backdoor_def.h.

[43] Checkpoint RESEARCH. *Renewed APT29 Phishing Campaign Against European Diplomats*. 2025. URL : https://github.com/kernelwernel/VMAware.

[44] CURIOUSLEARNERBLOG. *Qemu-TCG instruction emulation*. 2016. URL : https://curiouslearnerblog.wordpress.com/2016/05/04/qemu-tcg-instruction-emulation/.

[45] Unprotect PROJECT. *Sandbox Evasion*. URL : https://unprotect.it/category/sandbox-evasion/.

[46] Michael Sikorski et ANDREW HONIG. *Practical Malware Analysis*. San Francisco : no starch press, 2012. URL : https://www.kea.nu/files/textbooks/humblesec/practicalmalwareanalysis.pdf.

[47] A0RTEGA. *Dépôt GitHub de pafish*. URL : https://github.com/a0rtega/pafish.

[48] KERNELWERNEL. *Dépôt GitHub de VMAware*. URL : https://github.com/kernelwernel/VMAware.