Dissection de l'hyperviseur VMware

Brice L'helgouarc'h
brice.lhelgouarch@amossys.fr

Amossys
11 Rue Maurice Fabre, 35000 Rennes

Résumé. L'utilisation croissante de la virtualisation des systèmes d'exploitation a contribué à l'émergence de nouvelles vulnérabilités spécifiques aux hyperviseurs, telles que les *VM escape*. Il peut donc être intéressant de comprendre les mécanismes internes des hyperviseurs afin de réduire au maximum leur surface d'attaque et se prémunir de telles vulnérabilités. Dans cette optique, ce document décrit des travaux de rétro-ingénierie menés sur l'hyperviseur VMware afin d'en expliquer les mécanismes internes, et plus particulièrement ceux de son mode Unity, jusqu'alors non documenté publiquement.

1 Introduction

La virtualisation des systèmes d'exploitation s'est aujourd'hui imposée sur la quasi-totalité des systèmes d'information. Dans le cas des processeurs x86, cette virtualisation était initialement mise en œuvre à travers des techniques peu efficaces (émulation, virtualisation totale) ou très contraignantes (paravirtualisation). L'arrivée des jeux d'instructions de virtualisation a rendu la virtualisation plus simple à mettre en œuvre, mais aussi plus efficace. Elle a aussi permis l'émergence de nouveaux usages de la virtualisation, par exemple pour isoler plusieurs applications d'un même système d'exploitation, comme le fait Windows 10 avec la protection *Virtualization-Based Security*.

Cependant, dans le cas d'une infrastructure virtualisée, l'isolation entre machines n'est plus physique mais logicielle. Cela induit de nouveaux risques, car la présence d'une vulnérabilité dans l'hyperviseur utilisé pourrait avoir de graves conséquences, parmi lesquelles une fuite d'information, une élévation de privilèges ou encore le *crash* d'une machine virtuelle. Par ailleurs, la surface d'attaque d'un hyperviseur est importante : un moniteur de machines virtuelles (VMM) peut potentiellement être vulnérable au niveau de la gestion des instructions sensibles, de l'interaction avec les systèmes invités ou encore de la connexion des machines virtuelles au réseau.

De plus, le fonctionnement interne des hyperviseurs propriétaires est peu documenté, et, par conséquent, peu connu de la communauté. Afin d'éclaircir ce sujet, les principes généraux de la virtualisation seront tout d'abord expliqués. Il s'agira ensuite de décrire certains mécanismes intrinsèques de l'hyperviseur VMware. Une attention plus particulière sera portée à son mode Unity, permettant de manipuler les fenêtres d'un système invité directement sur le système hôte, et dont le fonctionnement interne n'est pas documenté publiquement à ce jour.

2 Fonctionnement général des hyperviseurs

La virtualisation est un concept qui, dans l'absolu, consiste à rendre une application indépendante du matériel sur lequel elle s'exécute. La virtualisation trouve son intérêt dans de nombreuses branches de l'informatique, et permet, par exemple, de rendre un logiciel indépendant de la plateforme matérielle utilisée (Java Virtual Machine), de créer un lien direct entre deux machines à travers Internet (réseaux privés virtuels), ou encore de faire cohabiter plusieurs systèmes d'exploitation sur une même machine (machines virtuelles).

Par abus de langage, on désigne souvent par « virtualisation » le fait d'exécuter des machines virtuelles uniquement, sans se soucier des autres applications de la virtualisation. Nous nous concentrerons d'ailleurs sur cet aspect de la virtualisation à travers ce document.

2.1 Prérequis formels à la virtualisation

En 1974, Popek et Goldberg publient l'article *Formal Requirements for Virtualizable Third Generation Architectures* [9], décrivant un ensemble de conditions suffisantes (mais non nécessaires) pour permettre la virtualisation sur une architecture de processeur donnée. Cet article spécifie tout d'abord trois propriétés attendues d'un hyperviseur, à savoir la *fidélité*, la *performance* et le *contrôle sur les ressources matérielles*. Popek et Goldberg définissent ensuite les notions d'instruction privilégiée et d'instruction critique, en considérant un processeur disposant d'un mode superviseur et d'un mode utilisateur. Dans ce contexte, le terme français *lancer une exception* sera utilisé de manière équivalente au terme anglais *to trap*.

Définition 1. Une instruction est dite *privilégiée* si l'exécution de celle-ci lance une exception lorsque le processeur n'est pas en mode superviseur.

Ainsi, l'exception lancée par l'exécution d'une instruction privilégiée en mode utilisateur va donner lieu à un changement de contexte afin de pouvoir traiter de manière appropriée cette instruction.

Définition 2. Une instruction est dite *critique* ou *sensible* si celle-ci interagit avec le matériel.

Une fois les définitions précédentes établies, trois théorèmes sont énoncés, dont le suivant :

Théorème 1. Pour tout ordinateur de troisième génération conventionnel, un hyperviseur peut être construit si l'ensemble de ses instructions sensibles est un sous-ensemble de ses instructions privilégiées.

Ici, le terme *ordinateur de troisième génération conventionnel* désigne un modèle formel d'ordinateur défini dans l'article dont le but est de décrire les ordinateurs contemporains à la date de rédaction de l'article. Cependant, ce modèle peut aussi s'appliquer aux ordinateurs actuels. Ainsi, l'architecture x86 n'est pas virtualisable au sens de Popek et Goldberg, car celle-ci possède des instructions sensibles mais non privilégiées [12]. Cette caractéristique de l'architecture x86 rend difficile l'implémentation d'un hyperviseur satisfaisant la propriété de contrôle sur les ressources matérielles : il faudrait en effet remplacer les instructions sensibles mais non privilégiées par un traitement approprié (traduction binaire).

L'utilisation des instruction de virtualisation (technologie VT-x) permet toutefois de répondre aux trois propriétés énoncées précédemment. En effet, cette technologie permet à l'hyperviseur d'intercepter automatiquement les instructions sensibles, ce qui satisfait la troisième propriété et facilite le respect de la seconde.

2.2 Gestion des instructions critiques

Les instructions critiques sont définies comme étant les instructions interagissant avec le matériel. Ces instructions doivent être traitées d'une manière particulière pour que l'hyperviseur puisse garantir son contrôle total sur les ressources matérielles et isoler les machines virtuelles. Dans le cadre de la virtualisation totale, l'hyperviseur doit intercepter par lui même les instructions critiques, ce qui rend la réalisation d'un hyperviseur complexe. Les instructions de virtualisation simplifient cette réalisation en permettant à l'hyperviseur d'intercepter automatiquement les instructions critiques. De plus, la technologie VT-x fournit des instructions permettant de sortir d'une machine virtuelle automatiquement et de passer de l'hyperviseur à une machine virtuelle en une opération atomique. De ce fait, le fonctionnement d'un hyperviseur basique peut être décrit par le pseudo-code donné dans le listing 1.

```
vmxon
init VMCS
vmlaunch

while(1) {
    exit_code = read_exit_code(VMCS);
    switch(exit_code) {
        case TRIPLE_FAULT_EXIT:
            // Traitement en cas de triple fault
        case IO_INSTRUCTION:
            // Traitement des instructions I/O
        case VMXOFF_INSTRUCTION:
            // Traitement de l'instruction vmxoff
        [...] // Autres exit codes
    }
    vmresume
}

vmxoff
```

Listing 1. Pseudo-code d'un hyperviseur exploitant les instructions de virtualisation [4].

L'hyperviseur active tout d'abord les extensions de virtualisation (*Virtual Machine eXtensions*) à l'aide de l'instruction `vmxon`, ce qui introduit deux nouveaux modes d'exécution. Le mode *VMX root* est destiné aux opérations de l'hyperviseur, alors que le mode *VMX non-root* est destiné aux instructions des systèmes invités. Le moniteur de machines virtuelles initialise ensuite une *Virtual Machine Control Structure* (VMCS) pour tout processeur virtuel/logique de chaque machine virtuelle. Cette structure permet de contrôler les transitions entre les modes *VMX root* et *VMX non-root* survenant à l'exécution des instructions `vmlaunch`/`vmresume` ou d'une instruction critique [6, Vol. 3C, p. 23-2]. Les données contenues dans une VMCS sont divisées en six groupes logiques [6, Vol. 3C, p. 24-3], dont les champs de contrôle de sortie de la machine virtuelle et les champs d'information de sortie de la VM.

Ainsi, lorsqu'une instruction sensible est piégée dans l'hyperviseur, ce dernier peut déterminer la raison pour laquelle l'exécution de la machine virtuelle a été suspendue (sortie de VM, *VM exit*) en lisant les champs d'information de *VM exit*. Cette instruction sensible peut ensuite être traitée de manière adaptée par le moniteur de machines virtuelles. Ce dernier peut ensuite reprendre l'exécution de la machine virtuelle (*VM entry*) à travers l'instruction `vmresume`, et ne regagne le contrôle sur le processeur qu'en cas de *VM exit* ou de fin du *VMX preemption timer* (mécanisme permettant d'interrompre une machine virtuelle de manière périodique et automatique). L'utilisation des extensions de virtualisation peut éventuellement être suspendue en exécutant l'instruction `vmxoff`.

3 Étude des mécanismes de l'hyperviseur VMware

L'entreprise VMware propose sur le marché deux hyperviseurs adaptés à des besoins distincts : *Workstation* pour le marché bureau et *ESXi* pour le marché serveur. Ces deux hyperviseurs ayant un fonctionnement semblable, le terme « hyperviseur VMware » sera utilisé à travers ce document pour les désigner sans distinction.

L'une des caractéristiques principales de l'hyperviseur VMware est sa forte utilisation d'extensions de système invité, permettant par exemple de partager le presse-papiers ou des dossiers du système hôte avec les machines virtuelles. La mise en œuvre de telles extensions est rendue possible par des communications entre les machines virtuelles et le VMM, mais aussi entre les différents processus de l'hyperviseur sur le système hôte. Ces communications, qui représentent par ailleurs l'un des éléments les plus importants de la surface d'attaque de l'hyperviseur (après la gestion des périphériques), sont détaillées ci-dessous.

3.1 Communications entre VM et hyperviseur

Backdoor. Les communications entre VM et hyperviseur reposent sur un mécanisme bas niveau nommé Backdoor [5]. Celui-ci consiste en un traitement particulier des *VM exits* lancées par la communication sur les ports I/O `0x5658` et `0x5659`. Par choix d'architecture, l'hyperviseur ne lance pas d'exceptions lorsque le mécanisme Backdoor est appelé depuis le ring 3. Cela permet de minimiser les privilèges des extensions fonctionnant au sein des machines virtuelles, mais empêche en contrepartie l'hyperviseur de connaître le niveau de privilèges du procesus ayant lancé l'appel à Backdoor. Il est à noter que ce mécanisme est actif par défaut et ne semble pas désactivable.

```
mov rax, 0x564D5868 ;magic number
mov rbx, 0x16645186 ;params commande
mov rcx, 0x01 ;num commande
mov dx, 0x5658 ;port

in rax,dx
```

Listing 2. Exemple de communication par Backdoor: récupération de la fréquence du CPU de la machine virtuelle.

Comme le montre le listing 2, l'appel à Backdoor nécessite de passer par l'assembleur. Le registre `rcx` permet de spécifier un numéro correspondant à une commande spécifique : ces commandes sont variées, et permettent par exemple de récupérer des informations système ou de manipuler le

presse-papiers de la VM [3]. Une utilisation « brute » de Backdoor n'est toutefois pas suffisante pour permettre des communications complexes et sécurisées entre les machines virtuelles et l'hyperviseur, puisqu'un utilisateur non privilégié pourrait *spoof* d'autres utilisateurs aux yeux de l'hyperviseur [11]. VMware définit donc deux protocoles fonctionnant au-dessus de Backdoor : RPCI et TCLO, repsectivement en charge des communications de la machine virtuelle à l'hyperviseur et inversement.

RPCI. Ce protocole permet d'acheminer les messages en provenance de la machine virtuelle vers l'hyperviseur. Ce protocole, essentiellement textuel, fonctionne au travers d'un ensemble de commandes permettant au système invité d'annoncer ses capactités, son adresse IP, ou encore de déclencher des logs sur le système hôte. Ces commandes sont exploitées par les extensions de système invité pour transmettre des informations à l'hyperviseur au cours de l'exécution de la machine virtuelle.

Il est possible de capturer des messages RPCI en plaçant des *breakpoints* appropriés lors de l'exécution du processus `vmware-vmx.exe` [5]. Le listing 3 donne un exemple de message capturé à l'aide de cette méthode. Un administrateur soucieux de limiter la surface d'attaque d'un hyperviseur peut désactiver certaines commandes RPCI à travers la configuration des machines virtuelles (fichiers `.vmx`).

```
74 6f 6f 6c 73 2e 75 6e-69 74 79 2e 70 75 73 68  tools.unity.push
2e 75 70 64 61 74 65 20-6d 6f 76 65 20 36 36 34  .update move 664
35 32 20 31 33 34 33 20-31 30 31 32 20 31 34 30  52 1343 1012 140
32 20 31 30 38 30 00 00-00                       2 1080...
```

Listing 3. Exemple de *hex dump* de message RPCI: déplacement d'une fenêtre en mode Unity. Remarquons que le message se termine par trois *null bytes*.

Il est possible d'énumérer les différentes commandes RPCI existantes en inspectant les *xrefs* d'une commande connue (comme `vmx.set_option`). La chaîne de caractères correspondante est en général référencée dans une unique fonction, que nous nommerons `RegisterRPCICommandHandler(id, vmdbKey, command, handler, a5)`, où :

- `id` est un identifiant numérique ;
- `vmdbKey` est une clé de la VMDB (détaillé ci-dessous) ;
- `command` est la commande RPCI à enregistrer ;
- `handler` est un pointeur vers la fonction prenant en charge la commande spécifiée ;
- `a5` est un paramètre valant 0 la plupart du temps.

TCLO. Les messages TCLO permettent à l'hyperviseur de donner des ordres aux extensions d'une machine virtuelle. Tout comme RPCI, TCLO est en grande partie textuel, et permet entre autres d'ordonner un changement de résolution de la console (voir listing 4) ou un passage en mode Unity (commande `unity.enter`). Le système invité répond ensuite à l'hyperviseur à travers un message `OK` ou `ERROR`, auquel peuvent être jointes des données [11] : ces données peuvent par exemple être une image PNG en réponse à une commande `unity.get.icon.data`.

```
44 69 73 70 6c 61 79 54-6f 70 6f 6c 6f 67 79 5f  DisplayTopology_
53 65 74 20 31 20 2c 20-30 20 30 20 31 39 32 30  Set 1 , 0 0 1920
20 31 30 38 30 00                                 1080.
```

Listing 4. *Hex dump* d'un message TCLO ordonnant au système invité de passer la définition de la console à 1920×1080.

Les machines virtuelles n'étant pas en capacité de recevoir des interruptions en provenance de l'hyperviseur, celles-ci sont dans l'obligation de recevoir les messages TCLO à travers un mécanisme de *polling*. Tout comme pour RPCI, il est possible de capturer les messages TCLO transitant entre hyperviseur et VM en plaçant des *breakpoints* dans les fonctions d'envoi et de réception TCLO de l'exécutable `vmware-vmx.exe`.

3.2 VMDB

Les communications entre VM et VMM contiennent souvent des informations devant être mémorisées par le système hôte, telles que les capacités du système invité, l'état des périphériques ou l'IP de la VM.

VMware apporte une réponse à cette problématique à travers la VMDB, base de données relative à l'exécution courante de la machine virtuelle et maintenue par le processus `vmware-vmx.exe`. Bien que ce mécanisme ne soit pas officiellement documenté, il est possible de le mettre en évidence par analyse statique. Cette dernière se retrouve ici grandement facilitée par la verbosité de l'hyperviseur : il est possible, pour un grand nombre de fonctions, de déterminer leur nom à l'aide de *format strings* passées en paramètres à des fonctions de log, comme le montre le listing 5.

```
lea     rdx, aVmxvmdb_setdev ; "VMXVmdb_SetDevPresent"
lea     rcx, aSDeviceNotFoun ; "%s: device not found!\n"
call    Debug
jmp     short loc_7FF7F15D63FF
```

Listing 5. Exemple de log permettant de déterminer le nom d'une fonction.

La recherche de chaînes de caractères débutant par le motif « `%s:` » permet donc d'identifier des noms de symboles dans le binaire, et ce de manière potentiellement automatisée. L'utilisation de cette méthode sur les fonctons relatives à la VMDB permet d'en identifier les caractéristiques principales. La VMDB présente tout d'abord un fonctionnement analogue à celui d'un système de fichiers : les informations y sont ordonnées sous forme arborescente et la notion de chemin courant y est définie (`Vmdb_SetCurrentPath`). Il est d'ailleurs possible d'observer des exemples de chemins VMDB en recherchant les chaînes de caractères présentes dans un *dump* de la mémoire du processus `vmware-vmx.exe`.

La VMDB présente tout de même des caractéristiques propres aux bases de données : des données peuvent y être écrites à l'aide des fonctions `Vmdb_Get` et `Vmdb_Set`, ces données étant organisées selon des schémas (exports `Vmdb_AddSchema` et `Vmdb_GetSchema` dans `vmwarebase.dll`).

Au final, la VMDB inclut un système de *callbacks*, c'est à dire de fonctions appelées à la suite d'une modification de la base. Ces *callbacks* sont tout d'abord enregistrés dans la VMDB à travers la fonction `VMXVmdbCb_RegisterCallbacks`, qui appelle `VMXVmdbCb_RegisterCallback` pour plusieurs fonctions et chemins de la VMDB. L'hyperviseur peut ensuite réagir à un évènement lancé par une machine virtuelle à travers les mises à jour de la VMDB déclenchées par le traitement des messages RPCI reçus.

3.3 Communications inter-processus

Bien qu'un hyperviseur simple puisse être décrit en une petite portion de pseudo-code, les hyperviseurs du marché sont bien plus complexes : ils doivent par exemple supporter de nombreux périphériques virtuels, présenter une interface graphique à l'utilisateur, ou encore faciliter les interactions avec les machines virtuelles à travers des extensions de système invité. L'hyperviseur VMware découpe ces différentes tâches en plusieurs processus spécifiques. Ainsi, chaque machine virtuelle active sur le système hôte fonctionne à travers un processus `vmware-vmx.exe`. Ce processus interagit avec un *driver* spécifique nommé `vmx86` en charge de gérer les VMCS des machines virtuelles. Remarquons qu'à ce jour, VMware n'utilise pas les hyperviseurs intégrés aux systèmes d'exploitation (*Windows Hypervisor Platform*, *macOS Hypervisor Framework* et KVM) [2, 7, 10]. Cela empêche par exemple un utilisateur Windows d'utiliser VMware Workstation en même temps que *Virtualization-Based Security*.

Les *VM exits* reçues par `vmx86` sont ensuite dispatchées au *VM exit handler* approprié à l'aide des champs de sortie de machine virtuelle

contenus dans la VMCS. Ces *VM exit handlers*, eux aussi contenus dans le processus `vmware-vmx.exe`, lui permettent de gérer les instructions critiques et donc les périphériques. Le mécanisme Backdoor reposant sur l'interception des *VM exits* lancées par les communications sur les ports I/O `0x5658` et `0x5659`, le processus VMX est aussi en charge de la gestion des interactions avec les machines virtuelles. De ce fait, et bien que l'hyperviseur soit en réalité découpé en plusieurs processus, l'essentiel de sa surface d'attaque réside dans les processus `vmware-vmx.exe`.

Dans le cas de VMware Workstation, les machines virtuelles sont administrées par l'utilisateur à l'aide d'une interface graphique présentée par le procesus `vmware.exe`. L'interface devant récupérer des informations concernant l'exécution de la machine virtuelle, elle utilise les données contenues dans la VMDB maintenue par le VMX au cours de son exécution.

L'utilisation d'un moniteur système tel que *Process Hacker* permet de repérer les *handles* exploités par ces processus, et de remarquer l'utilisation de *named pipes* telles que `vmx-mks-fd`, `vmx-live-fd`, ou encore `vmx-vmdb-fd`. Nous focaliserons notre attention sur cette dernière puisque que comme son nom l'indique, elle fait transiter les interactions avec la VMDB associée à une machine virtuelle, et occupe par conséquent un rôle central dans le fonctionnement du mode Unity.

Les *named pipes* se manipulant comme des fichiers sur Windows, il suffit de placer des *breakpoints* aux appels à `ReadFile` et `WriteFile` pour intercepter les communications : en x64, `lpBuffer` est pointé par `rdx` et `nNumberOfBytesToRead/Write` est contenu dans `r8`. Le script PyKD donné dans le listing 6 permet d'intercepter automatiquement les appels à ces fonctions pour un *handle* donné, et donc de visualiser l'intégralité des communications passant par un *named pipe* de *handle* `n_handle`.

```
import pykd

kernel32 = pykd.module("kernel32")

def OnWriteFile():
    n_handle = ###

    if pykd.reg("rcx") == n_handle:
        msg_str = ""
        msg_hex = pykd.loadBytes(pykd.reg("rdx"), pykd.reg("r8"))
        for b in msg_hex:
            msg_str = msg_str + chr(b)
        print(msg_str)

b0 = pykd.setBp(kernel32.WriteFile, OnWriteFile)
pykd.go()
```

Listing 6. Script d'interception des messages transmis sur les *named pipes*.

L'exécution du script du listing 6 sur le *named pipe* `vmx-vmdb-fd` permet d'observer des messages semblables à celui donné ci-dessous.

```
6 TUPLESe /vm/#_VMX/vmx/1
1 21 021 unity/unityUpdate/#19/updateData/1 228
    bW92ZSA2NjQ1MiAzODIgNDgxIDQzMiA1MzkAAAA=1
1 22 220 1 10 1
1
```

Listing 7. Exemple d'interaction avec la VMDB: propagation d'un message `unityUpdate` vers la GUI.

Nous nous sommes ici restreints aux échanges entre les processus `vmware-vmx.exe` et le processus `vmware.exe` via le *named pipe* `vmx-vmdb-fd`. Cependant, VMware Workstation exploite d'autres processus, tels que `vmware-usbarbitrator64.exe`, processus permettant à l'utilisateur d'attribuer un périphérique USB à l'hôte ou à une machine virtuelle, ou `vmnetdhcp.exe`, en charge de l'attribution des adresses IP des machines virtuelles. De même, il pourrait être intéressant d'analyser les messages transmis sur d'autres *named pipes*. La description de ces interactions n'étant pas nécessaire à la compréhension du fonctionnement du mode Unity, celles-ci ne seront pas détaillées dans un souci de concision.

4 Recherche de vulnérabilités sur le mode Unity

L'hyperviseur VMware Workstation propose, dans ses versions Windows et macOS, une fonctionnalité nommée *mode Unity* et permettant à un utilisateur de manipuler les fenêtres d'une machine virtuelle comme si il s'agissait de fenêtres du système hôte, comme le montre la figure 1. L'implémentation d'une telle fonctionnalité, au premier abord, semble très complexe : il faut en effet gérer un grand nombre d'interactions utilisateur, telles que le déplacement des fenêtres ou le lancement de nouveaux programmes.

Cette complexité semble *a priori* intéressante pour un attaquant, puisqu'une quantité importante de code est souvent à l'origine de vulnérabilités. De plus, les mécanismes internes du mode Unity ne sont pas documentés, et celui-ci ne semble à ce jour pas exploité par des vulnérabilités connues. Étudier de tels mécanismes à travers la rétro-ingénierie peut donc s'avérer très intéressant, autant d'un point de vue défensif qu'offensif.

Il est à noter que VirtualBox propose de telles fonctionnalités à travers son *seamless mode*. Cette recherche de vulnérabilités s'est cependant concentrée sur VMware Workstation en raison de la piètre politique de correction de vulnérabilités d'Oracle [8].

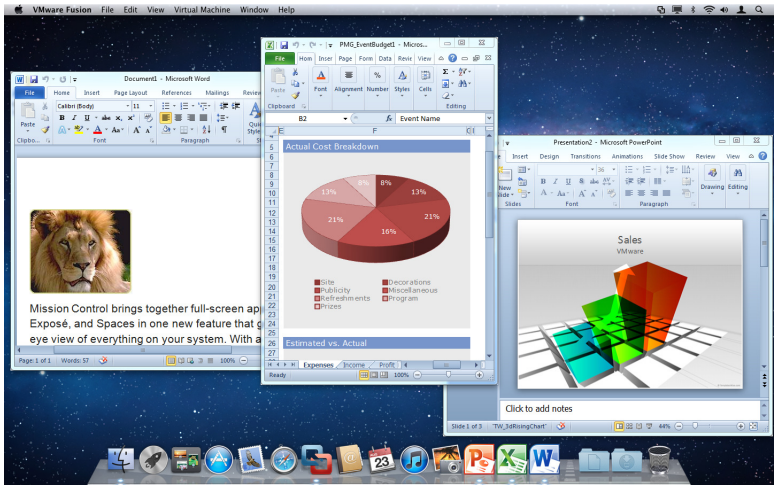

Fig. 1. Exemple d'utilisation du mode Unity : manipulation de fenêtres Windows sur un système Mac OS X (source : VMware).

4.1 Fonctionnement interne du mode Unity

Le mode Unity repose en majorité sur les mécanismes présentés ci-dessus. Le basculement en mode Unity se fait depuis l'interface graphique à l'initiative de l'utilisateur. Une fois ce basculement effectué, les systèmes invité et hôte s'échangent des mises à jour `unityUpdate` correspondant à un évènement tel qu'un ajout, un déplacement ou une réduction de fenêtre via RPCI et TCLO.

Du côté du système hôte, les mises à jour reçues en provenance de la machine virtuelle sont propagées vers la VMDB à travers des messages tels que montré sur le listing 7. Ces mises à jour sont ensuite lues par l'interface graphique (processus `vmware.exe`) et parsées par la fonction `UnityWindowTracker_ParseUnityUpdate`, contenue dans `vmwarebase.dll`. À la suite de ce *parsing* est extrait un type de mise à jour codé de manière numérique (sur Workstation 15, ces types sont au nombre de 20). Cette valeur est ensuite utilisée par la méthode `cui::UnityMgrBasic::ProcessUnityUpdate` afin de pouvoir dispatcher la mise à jour vers la fonction appropriée.

Le système invité réceptionne quant à lui les messages en provenance de l'hyperviseur via TCLO. Ces messages sont ensuite traités par le démon `vmtoolsd.exe`, qui comporte une instance ayant des privilèges utilisateur et une autre ayant des privilèges administrateur. Le démon fonctionne à l'aide d'un système de plugins, pouvant s'exécuter à différents niveaux de

privilèges (`vmusr` ou `vmsvc`). Les messages TCLO reçus peuvent alors être dispatchés vers le plugin approprié via un système de *callbacks*.

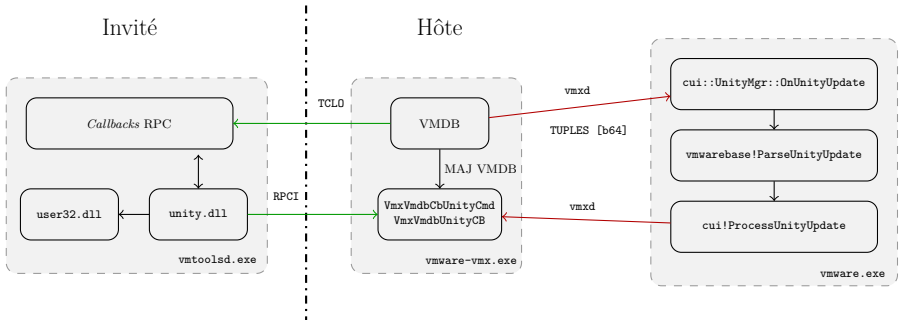

Fig. 2. Schéma récapitulatif du fonctionnement du mode Unity.

Dans le cas du mode Unity, le plugin utilisé est `vmusr/unity.dll`. Ce dernier met en place des *callbacks* en réaction aux commandes TCLO `unity.*` et `ghi.guest.*`. Comme lors de l'étude de la VMDB, la recherche du motif « `%s:` » facilite grandement la rétro-ingénierie, et permet de mettre en avant les fonctions `vmware::tools::*` enregistrées comme *callbacks* TCLO. Ces dernières appellent ensuite leur fonction `UnityPlatform*` respective (par exemple `UnityPlatformMoveResizeWindow` pour `vmware::tools::UnityTcloMoveResizeWindow`), qui va interagir avec les fenêtres de la VM via les fonctions de `user32.dll`.

4.2 Problématique

Comme montré précédemment, le mode Unity fonctionne à l'aide de communications entre l'hyperviseur et un système invité via RPCI et TCLO. De plus, le traitement des données échangées donne lieu à la création de nombreux objets pouvant potentiellement causer des vulnérabilités relatives à la gestion de la *heap*. Ainsi, un attaquant voulant utiliser cette fonctionnalité comme vecteur d'attaque contrôle les messages RPCI qu'il envoie à l'hyperviseur, mais aussi les réponses aux ordres reçus via TCLO.

Cependant, les messages échangés entre la machine virtuelle et l'hyperviseur sont, pour la plupart, relativement complexes. Déterminer le rôle de chacun des champs de message dans le but de disposer d'un *fuzzer* aussi intelligent que possible s'avérerait donc extrêmement chronophage. D'autre part, le *fuzzer* devrait idéalement avoir un *code coverage* important, en sachant que les messages invalides sont directement rejetés par le

VMX : les messages générés par le *fuzzer* doivent donc être « réalistes », bien que leur spécification initiale ne soit pas connue en détail. Finalement, l'utilisation du *fuzzer* ne devrait pas nécessiter d'intervention de la part de l'utilisateur : il est donc nécessaire de trouver un moyen de générer des interactions automatiquement. L'ensemble des réponses TCLO impliquées dans le mode Unity étant relativement restreint, nous nous focaliserons ici sur les messages RPCI provenant de la VM. La démarche décrite peut cependant être facilement adaptée à TCLO.

4.3 Solution retenue

La première problématique se présentant est le réalisme des messages générés. La mutation puis le rejeu de captures existantes et le *fuzzing* en mémoire se trouvent alors être deux solutions viables. Cependant, la problématique d'automatisation des interactions semble avantager le rejeu de messages. La mise en œuvre de ces deux approches de *fuzzing*, qui peuvent être complémentaires ou interdépendantes, est décrite ci-dessous.

Mutation et rejeu de captures RPCI. Les messages RPCI sont capturés d'une manière semblable à celle décrite par ZDI à ZeroNights [5]. Ils peuvent ensuite être rejoués, avec ou sans mutation, et dans l'ordre de capture ou non. Les mutations appliquées peuvent être multiples : inversion d'un bit du message (*bitflip*), modification d'un paramètre textuel ou numérique, etc. Une telle approche présente toutefois une limite importante : il n'est possible de rejouer qu'un ensemble restreint de commandes (correspondant aux interactions capturées), dont les paramètres sont pour la plupart déterminés à l'avance. Il est à noter qu'il ne semble pas possible de rejouer des interactions Unity réellement cohérentes d'une telle manière. On pourrait alors penser à enregistrer, en plus des messages, leur espacement temporel afin de gagner en réalisme.

MemITM. MemITM [1] est un outil développé par Amossys permettant d'intercepter et modifier des messages situés dans la mémoire d'un processus Windows directement depuis Python. Cet outil se découpe en plusieurs composants :

- un script IDAPython `generate.idapython.py` qui génère des fichiers de configuration indiquant l'adresse de la fonction dans laquelle se placer et les registres pointant sur le *buffer* pertinent ;
- une DLL pour placer le *hook* dans la mémoire du processus cible ;
- un injecteur de DLL ;

— un script Python `memitm.py` communiquant avec la DLL injectée.

Une fois le fichier de configuration généré depuis IDAPython, il suffit de démarrer `memitm.py` en lui spécifiant le fichier de configuration et un PID (ou un nom de processus). Le script `memitm.py` expose alors deux fonctions, `logger` et `fuzzer`. Par défaut, la fonction `fuzzer` effectue un *bitflip* sur un vingtième des messages interceptés. La fonction `logger` permet d'enregistrer les messages mutés afin de pouvoir les rejouer après un éventuel *crash* du programme cible. Ces deux fonctions peuvent être modifiées à la guise de l'utilisateur en exploitant les fonctions et bibliothèques de l'écosystème Python.

Placement des *hooks*. L'utilisation du fuzzing en mémoire nécessite de connaître à l'avance l'endroit où se placer dans la mémoire du processus cible pour y modifier les messages. Dans le cas de l'hyperviseur VMware, plusieurs possibilités s'offrent à nous : le choix d'un *fuzzing* sur l'intégralité des messages RPCI à destination de l'hyperviseur, ou, au contraire, d'une partie restreinte de ces derniers. La sélection de l'une de ces posssibilités se fait à travers le choix de l'adresse à laquelle est placée le *hook*. Le démon `vmtoolsd` reposant sur un système de plugins, il suffit de placer le hook avant un appel à `RpcChannel_Send` (importée depuis `vmtools.dll`) dans un plugin donné pour *fuzz* une catégorie ou un message précis. Au contraire, si l'on désire pouvoir modifier l'intégralité des messages RCPI à destination de l'hyperviseur, il est nécessaire de se placer à l'intérieur de la fonction `RpcChannel_Send`, exportée par la DLL `vmtools.dll`.

Les mutations appliquées aux messages sont semblables à celles employées lors du rejeu de captures, à savoir l'utilisation du *bitflip* et la substitution de paramètres.

4.4 Résultats

Mode Unity. L'utilisation des méthodes de *fuzzing* expliquées ci-dessus a donné lieu à divers dysfonctionnements du mode Unity. Les résultats du fuzzing en mémoire se sont avérés peu convaincants, puisque `vmtoolsd` semble suspendre l'envoi de mises à jour Unity dès lors que le suivi d'une fenêtre est perdu. De plus, nous avons pu remarquer des cas de boules infinies sur l'interface graphique (non exploitables) s'apparentant à des corruptions de la VMDB. Enfin, le rejeu de captures RPCI avec *bitflip* a donné lieu à un dysfonctionnement relativement intéressant, s'apparentant à une injection XPath, comme le montre la figure 3. Une telle injection ne cause cependant aucune corruption mémoire, elle n'est donc pas exploitable.

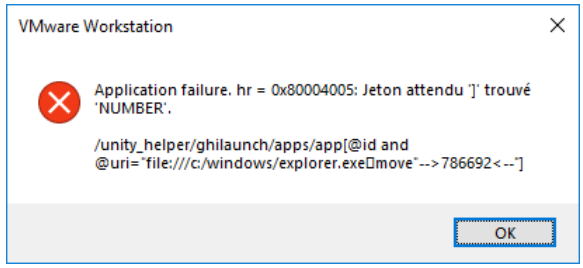

Fig. 3. Erreur lancée par l'interface graphique à la suite d'une injection de caractères invalides dans une requête XPath.

Le mode Unity, bien que relativement complexe, semble donc présenter un intérêt limité pour un potentiel attaquant. La nécessité d'un basculement manuel dans ce mode restreint par ailleurs l'exploitabilité d'une potentielle vulnérabilité le concernant. Cependant, l'étude de ses mécanismes internes (VMDB et IPC) montre qu'il est possible de viser des processus autres que `vmware-vmx.exe` (comme `vmware.exe`), dans le cas où les validations des messages à destination de la VMDB seraient insuffisantes.

Résultats globaux. Le *fuzzing* du mode Unity présente parfois des effets de bord, déclenchant des *bugs* relatifs aux mécanismes sur lesquels il repose. Ainsi, la méthode de rejeu de captures a permis de déclencher des *bugs* bien plus intéressants : de rares *crashs* relatifs à la gestion de commandes RPCI (toutefois non relatives à Unity) ont été observés. Cette phase de recherche a aussi donné lieu à la découverte d'une vulnérabilité relative à la gestion du son de la version Linux de VMware Workstation. Cette vulnérabilité a été reportée à VMware et est toujours en cours de correction. Des détails seront donnés lors de la présentation si VMware nous y autorise.

5 Conclusions

Nous avons décrit à travers ce document le fonctionnement général des hyperviseurs et détaillé celui de l'hyperviseur VMware. L'étude du mode Unity permet de mettre en avant des mécanismes internes de l'hyperviseur VMware jusqu'alors non connnus de l'état de l'art. Bien que la recherche de vulnérabilités menée sur celui-ci montre une certaine résistance à la méthode de *fuzzing* employée, les différents *bugs* rencontrés mettent en avant deux nouveaux éléments de la surface d'attaque, à savoir l'interface

graphique et la gestion du son. Il serait par ailleurs intéressant de procéder à une étude détaillée de ce dernier afin d'en évaluer la robustesse.

Les différentes vulnérabilités recensées dans l'état de l'art montrent que la présence de *bugs* dans un hyperviseur peut avoir un impact important en termes de disponibilité (*crash* d'une VM), mais aussi de confidentialité et d'intégrité (*VM escape*). De ce fait, certaines utilisations actuelles de la virtualisation, telles que l'analyse de *malwares* ou le *cloud computing* nécessitent de porter une attention particulière à la sécurité afin de limiter les risques encourus. Dans cette optique, un administrateur peut limiter la surface d'attaque d'un hyperviseur en désactivant des fonctionnalités ou périphériques non nécessaires au bon fonctionnement de la machine virtuelle à risques.

Références

1. GitHub AMOSSYS. MemITM : Tool to make in memory man in the middle. `https://github.com/AMOSSYS/MemITM`.
2. Apple. macOS Hypervisor. `https://developer.apple.com/documentation/hypervisor`.
3. VM Back. VMware Backdoor I/O Port. `https://sites.google.com/site/chitchatvmback/backdoor`.
4. Oleksandr Bazhaniuk, Mikhail Gorobets, Andrew Furtak, and Yuriy Bulygin. Attacking hypervisors through hardware emulation. `https://www.troopers.de/downloads/troopers17/TR17_Attacking_hypervisor_through_hardwear_emulation.pdf`, 2017.
5. Hariri, Abdul-Aziz and Spelman, Jasiel and Gorenc, Brian. Leveraging VMware's RPC Interface for Fun and Profit. `https://ruxcon.org.au/assets/2017/slides/ForTheGreaterGood.pdf`, 2017.
6. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2016.
7. Microsoft. Windows Hypervisor Platform. `https://docs.microsoft.com/en-us/virtualization/api/`.
8. MorteNoir1. VirtualBox E1000 Guest-to-Host Escape. `https://github.com/MorteNoir1/virtualbox_e1000_0day`.
9. Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 1974.
10. The KVM project. Kernel Virtual Machine. `https://www.linux-kvm.org/page/Main_Page`.
11. Julien Raeis and Nicolas Collignon. VMware et sécurité. `https://www.ossir.org/sur/supports/2008/OSSIR_VMware_20080807.pdf`, 2008.
12. John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. *Proceedings of the 9th USENIX Security Symposium*, 2000.