CrashsOS : Recherche de vulnérabilités système dans les hyperviseurs

Anaïs Gantet
anais.gantet@airbus.com

Airbus

Résumé. Les hyperviseurs occupent désormais une place importante au sein de la plupart des infrastructures. Il est donc important de s'assurer de la robustesse de ces logiciels.
CrashOS a été développé dans ce but. Il propose un moyen de rechercher des vulnérabilités dans les hyperviseurs. C'est un système d'exploitation minimaliste présentant l'avantage d'être entièrement configurable, permettant ainsi d'implémenter simplement des attaques logicielles à très bas niveau. Cet outil est *opensource*.
À l'heure actuelle, CrashOS a pu être utilisé pour éprouver principalement deux solutions de virtualisation, l'une développée en interne sous forme d'outil d'expérimentation (Ramooflax), l'autre déployée à grande échelle (VMware). Dans la plupart des cas d'attaques, les logiciels étudiés se sont montrés robustes aux tests réalisés, que ce soit au niveau de la virtualisation du processeur, de la mémoire, ou des autres périphériques. Néanmoins, certains tests ont conduit à un crash de la machine virtuelle. L'outil développé a commencé à montrer son efficacité en permettant de détecter certains dysfonctionnements des solutions de virtualisation étudiées.

1 Introduction

L'hyperviseur ou *virtual machine monitor* (VMM) est l'élément clé d'un système utilisant la virtualisation, puisqu'il contrôle l'ensemble des machines virtuelles (VM) s'exécutant sur une machine physique. Utiliser un hyperviseur ajoute une surcouche logicielle dans les systèmes et peut donc être source de vulnérabilités.

Les vulnérabilités système connues aujourd'hui dans VMware, Xen, KVM ou Virtualbox révèlent différents points critiques dans les hyperviseurs, par exemple :

- le désassemblage des instructions (CVE-2014-7155 [7], CVE-2012-0217) ;
- l'émulation des instructions sensibles ou privilégiées (CVE-2015-5307, CVE-2015-8104) ;

– la gestion de l'accès aux ressources mémoire (CVE-2016-3960, CVE-2015-7835, CVE-2016-6258) ;
– la virtualisation des périphériques (CVE-2015-3456, CVE-2015-2341).

L'exploitation d'une faille au sein d'un hyperviseur peut engendrer des conséquences critiques telles que la fuite d'information entre VM, l'élévation de privilèges au sein d'une VM, le risque de compromission de l'infrastructure complète ou encore le risque d'indisponibilité propagé à l'ensemble des VM [11, 12, 14]. Il est donc important de s'assurer de la robustesse de ce type de logiciel.

Le but de notre travail a été de proposer un moyen pour faciliter la recherche des vulnérabilités dans les hyperviseurs. Notre étude ne concerne que les architectures x86. Cet article présente l'approche proposée, ainsi que l'outil CrashOS développé et les premiers résultats obtenus grâce à cet outil.

2 Notre approche

Le code source des hyperviseurs n'est pas toujours accessible publiquement. La stratégie de recherche de vulnérabilités présentée ici ne se fonde donc pas sur une analyse de code mais plutôt sur une analyse comportementale des hyperviseurs, en observant leur réaction face à une VM malveillante.

La figure 1 illustre cette approche pour le cas de l'hyperviseur VMware installé sous Linux.

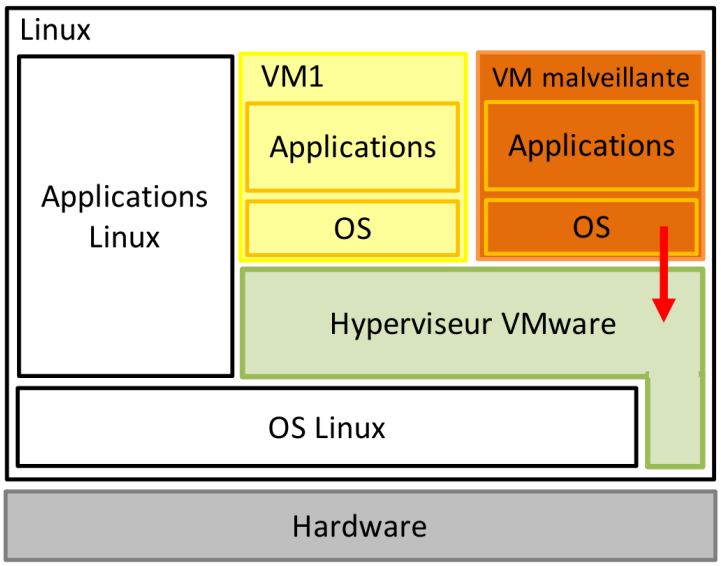

Fig. 1. Architecture de recherche de vulnérabilités – cas de Vmware sous Linux

De manière générale, les hyperviseurs fonctionnent correctement lorsque les systèmes d'exploitation qu'ils gèrent se comportent de façon classique. Cependant, notre expérience dans l'analyse et l'écriture d'hyperviseurs nous a montré que traiter de manière exhaustive et correcte toutes les combinaisons de cas plus ou moins complexes que l'OS peut requérir est un véritable casse-tête pour les auteurs des hyperviseurs et constitue donc un potentiel point faible.

Le but est de construire une VM malveillante adaptée à la recherche de vulnérabilités. Le travail d'un hyperviseur se concentre sur les couches basses du système. L'étude se focalise donc à ce niveau.

L'idée n'est pas de rechercher des vulnérabilités en effectuant du *fuzzing* aveugle. En effet, les configurations système à tester n'ont d'intérêt que dans la mesure où elles mettent en jeu un traitement particulier de l'hyperviseur, que ce soit au niveau de la virtualisation de la vision mémoire, de la virtualisation du processeur ou encore de la communication avec les périphériques.

Les grandes étapes de notre démarche peuvent donc se résumer ainsi :

- imaginer des cas particuliers de configuration système pouvant mettre en difficulté les hyperviseurs ;
- disposer d'un outil permettant de les mettre en œuvre et de les embarquer dans une VM unique ;
- observer la réaction des hyperviseurs face à cette VM ;
- évaluer l'exploitabilité des éventuelles failles détectées.

3 Mise en œuvre

3.1 Fonctionnalités attendues

Afin de tenter de mettre en difficulté les hyperviseurs de manière efficace, la VM de tests doit remplir les critères suivants :

- posséder un système d'exploitation permettant de faciliter la communication avec le matériel et l'écriture des tests ;
- posséder un framework regroupant des tests à lancer en série sur divers hyperviseurs ;
- permettre de visualiser le résultat de chaque test.

3.2 Outils existants

Concernant les outils dédiés à la recherche de vulnérabilités dans les hyperviseurs, il existe l'outil VESPA [8], développé par Securimag. Cependant, cet outil n'est pas entièrement *opensource* et ne cible que les attaques via les périphériques. Il n'a donc pas été utilisé pour notre étude.

Concernant le système d'exploitation (OS) de la VM, l'OS nécessaire à notre étude n'a pas besoin d'être spécialement sophistiqué. Nous souhaitions cependant disposer d'un OS nous donnant le maximum de privilèges afin d'interagir avec le matériel de la manière la plus simple et la plus directe possible. Les OS classiques tels que Windows et Linux ne sont pas adaptés à ce genre d'utilisation, ne serait-ce que pour simplement faire basculer le processeur en mode réel, ou pour désactiver facilement la pagination. De nombreux petits OS existent (Simple Operating System [15], Redox [5], OSv [4], BareMetal-OS [2], B2G [1], par exemple) mais la plupart ne disposent pas des fonctionnalités requises, d'où l'idée de développer notre propre outil de tests des hyperviseurs : CrashOS.

3.3 Présentation de CrashOS : aspect général

L'outil *opensource* [6] développé en C et assembleur est un OS minimaliste configurable, pouvant provoquer des crashs en cas de dysfonctionnement de l'hyperviseur, d'où son nom : CrashOS.

Le premier avantage de l'outil est sa compacité. En moins de 2 000 lignes de code source, l'outil permet de :

- démarrer une machine ;
- définir et utiliser des structures de pagination (PGD, PTB, CR3, etc.) ;
- définir et utiliser des structures de segmentation (GDT, LDT, etc.) ;
- bénéficier d'un adressage direct vers la mémoire physique ;
- écrire n'importe quel handler d'interruptions ou d'exceptions ;
- interagir avec n'importe quel périphérique par IN et OUT sur le port d'I/O concerné ;
- lancer plusieurs tests les uns après les autres ;
- visualiser le résultat de chaque test (à l'écran ou sur le port série).

De plus, CrashOS se présente comme une sorte d'API qui permet d'interagir directement avec le matériel, en ring 0. Par exemple, si l'on veut savoir comment se comporte un hyperviseur face à une VM avec des structures de pagination configurées de manière atypique et que l'on veut pour cela activer la pagination, il suffit d'utiliser la fonction `enable_paging()` définie dans le listing 1.

```
#define enable_paging()
    asm volatile (                  \
    "mov %%cr0, %%eax       \n"     \
    "or $0x80000000, %%eax  \n"     \
    "mov %%eax, %%cr0":::"eax" )
```

Listing 1. Fonction d'activation de la pagination de CrashOS

3.4 CrashOS : architecture et utilisation

CrashOS se présente sous la forme d'un binaire *crashos.bin* à installer sur une partition bootable. Ce binaire suit le standard multiboot, ce qui permet d'utiliser Grub [10] comme *boot loader* de notre OS. Utiliser Grub permet non seulement de simplifier l'implémentation du démarrage de CrashOS mais aussi de profiter du fait que Grub s'occupe de passer le processeur en mode protégé. Ceci est intéressant car la majorité des tests que l'on souhaite lancer nécessite d'être en mode protégé 32 bits[1].

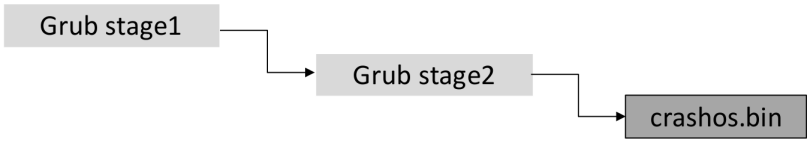

Fig. 2. Chaine de démarrage de CrashOS

Le contenu du binaire *crashos.bin* découle de la compilation du code C de CrashOS et dépend de l'utilisateur. En effet, hormis l'étape de démarrage, la vie de l'OS dépend entièrement des tests que l'utilisateur désire lancer.

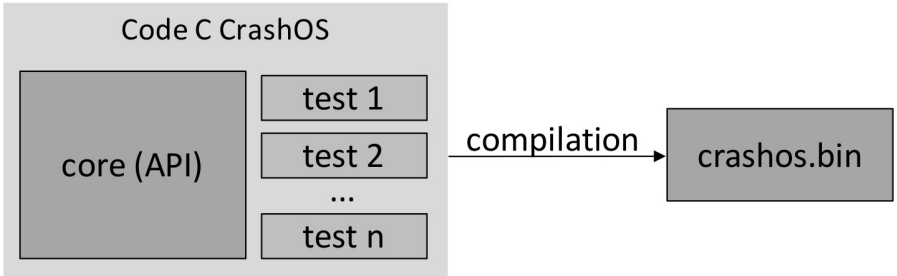

Fig. 3. Construction de crashos.bin

[1] Cela n'empêche pas de basculer à tout moment l'OS dans un autre mode, notamment le mode 64 bits, pour un test en particulier.

La spécification des tests s'effectue à la compilation. Il suffit à l'utilisateur de lister les noms des tests dans le Makefile. Cette liste de noms est gérée par une structure interne à CrashOS, qui stocke un pointeur sur chaque test dans la section `.tests` du binaire *crashos.bin*. Le listing 2 contient la définition de cette section telle que passée à l'édition de lien.

```
    __tests_start__ = .;
    .tests      . : { *(.tests) }
    __tests_end__   = .;
```

Listing 2. Définition de la section tests dans CrashOS

Cette section permet de regrouper l'ensemble des tests marqués par la macro `DECLARE_TEST(test)`, définie dans le listing 3.

```
#define DECLARE_TEST(xXx)   test_t * __attribute__((section(".tests"
    ))) xXx##_ptr = &xXx;
```

Listing 3. Définition de la macro DECLARE_TEST tests dans CrashOS

Enfin la fonction principale de CrashOS parcourt la liste des tests et exécute chacun des tests en itérant entre les symboles `__tests_start__` et `__tests_end__`.

3.5 CrashOS : Rédaction d'un nouveau test

Dans CrashOS, chaque test a besoin de définir exactement le contexte dans lequel il va s'exécuter afin de tester des configurations particulières. Par exemple, un test aura besoin d'utiliser la pagination, alors qu'un autre n'aura besoin que de la segmentation avec des segments particuliers.

La configuration système de l'OS fait donc partie intégrante des tests car elle est spécifique à chacun. À chaque test sont donc associées les trois fonctions suivantes :

- une fonction d'initialisation : pour la sauvegarde de l'état courant et l'initialisation de l'OS spécifique à notre test ;
- la fonction de test proprement dite ;
- une fonction de restauration : pour restaurer l'état sauvegardé.

À chaque test est également attribuée une structure `test_t` contenant les informations du test : son nom, une courte description et les pointeurs des trois fonctions nommées ci-dessus. Enfin, pour qu'un test puisse être pris en compte par la structure de gestion des tests de CrashOS, il faut le marquer de la macro `DECLARE_TEST(test)`. Le listing 4 donne la forme générale que doivent suivre les tests de CrashOS.

```
#include <core/test.h>
#include <core/init.h>
#include <core/segmentation.h>
#include <core/interrupts.h>
#include <core/page.h>
#include <core/print.h>
#include <core/start.h>

static int init_test_x() {
    // Save
    // Init
    return 0;
}

static int do_test_x() {
    // Test
    return 0;
}

static int restore_test_x() {
    // Restore
    return 0;
}

test_t test_x = {
    .name    = "test name",
    .desc    = "test description",
    .init    = init_test_x,
    .test    = do_test_x,
    .restore = restore_test_x,
};
DECLARE_TEST(test_x)
```

Listing 4. Template d'un test dans CrashOS

Pour l'implémentation des fonctions `init`, `test` et `restore`, l'utilisateur peut s'appuyer sur l'API de CrashOS. L'ensemble des fonctions de l'API est documenté (`crashos$ make documentation`).

3.6 CrashOS : Visualisation du résultat des tests

Le lancement d'un test peut aboutir aux trois comportements suivants :

- le test se passe bien et le comportement de l'hyperviseur est identique à celui d'une machine physique ;
- le test se passe bien mais le comportement de l'hyperviseur est différent de celui d'une machine physique ;
- le test conduit à un crash : la VM ou l'hyperviseur ont cessé de fonctionner de façon inattendue.

Dans les deux premiers cas, la visualisation des résultats peut être gérée par CrashOS. Dans le troisième cas, seules les informations remontées

avant le crash sont maîtrisables. Pour cela, l'API de CrashOS permet de communiquer avec deux types de périphériques suivants : la mémoire vidéo et le port série.

La communication avec la mémoire vidéo s'effectue par *memory mapped I/O* [3]. La figure 4 illustre un exemple de l'écran de CrashOS :

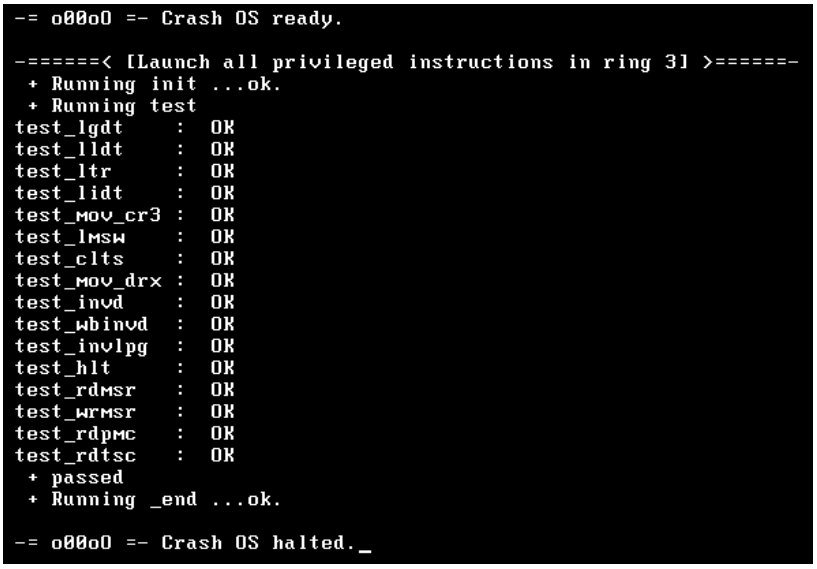

Fig. 4. Aperçu de la visualisation du résultat des tests à l'écran de CrashOS

CrashOS permet également de récupérer des messages via le port série, avec lequel la communication s'effectue par *port I/O*. En cas de crash, cela permet de savoir à quel endroit s'est arrêté l'OS.

Hormis les messages relatifs aux lancements successifs d'une série de tests, consistant en l'affichage des informations du test courant (nom, description, lancement de la fonction `init`, `test` ou `restore`), la remontée d'information est laissée totalement libre à l'utilisateur. Le rédacteur d'un test dispose des trois fonctions suivantes :

- `printf(...)` : pour afficher des messages à l'écran ;
- `log(...)` : pour envoyer un message à travers le port série ;
- `printf_log(...)` : pour afficher à la fois un message à l'écran et le rediriger vers le port série.

Par exemple, dans le cas de la figure 4, le rédacteur du test *Launch all privileged instructions in ring 3* a choisi d'afficher « OK » si le comportement du l'hyperviseur est identique à celui d'une machine physique, « NOK » sinon.

4 Dans la pratique

Cette partie illustre des exemples d'attaques ayant pu être mise en œuvre avec CrashOS ainsi que les résultats actuels obtenus sur les deux hyperviseurs testés, Ramooflax et Vmware.

4.1 Exemples d'attaques

Les idées d'attaques à lancer sur les hyperviseurs peuvent être regroupées en fonction des points des hyperviseurs que l'on cherche à mettre en défaut. L'implémentation de certains tests décrits ci-dessous est fournie avec l'outil.

Accès à la mémoire physique

Objectif Un premier vecteur d'attaque est la gestion de la vision de la mémoire physique que l'hyperviseur donne aux VM. Par exemple, l'hyperviseur doit s'assurer qu'une VM ne peut accéder qu'à la plage d'adresses physiques qui lui a été attribuée, pour éviter d'avoir accès à la mémoire d'une autre VM ou encore à celle de l'hyperviseur lui-même.

Exemples de mises en œuvre En général, les tailles de RAM attribuées sont inférieures à 4GB. Or, en mode protégé 32 bits, l'adressage physique peut aller de 0 à 4GB. Une manière de tester est donc d'essayer d'écrire, de lire ou d'utiliser la mémoire physique au delà de la taille de la RAM.

Difficultés des traductions d'adresses en mode protégé

Objectif Un autre point crucial à vérifier est le mécanisme de traduction d'adresses, surtout lorsque les adresses virtuelles sont utilisées par des instructions sensibles, instructions interceptées et virtualisées par l'hyperviseur.

Exemples de mises en œuvre Un cas de test a été de configurer les tables de manière à ce que 2 pages virtuelles consécutives soient mappées sur 2 pages physiques non contiguës en mémoire, comme illustré sur la figure 5, et de placer des données ou une instruction à cheval sur ces deux pages.

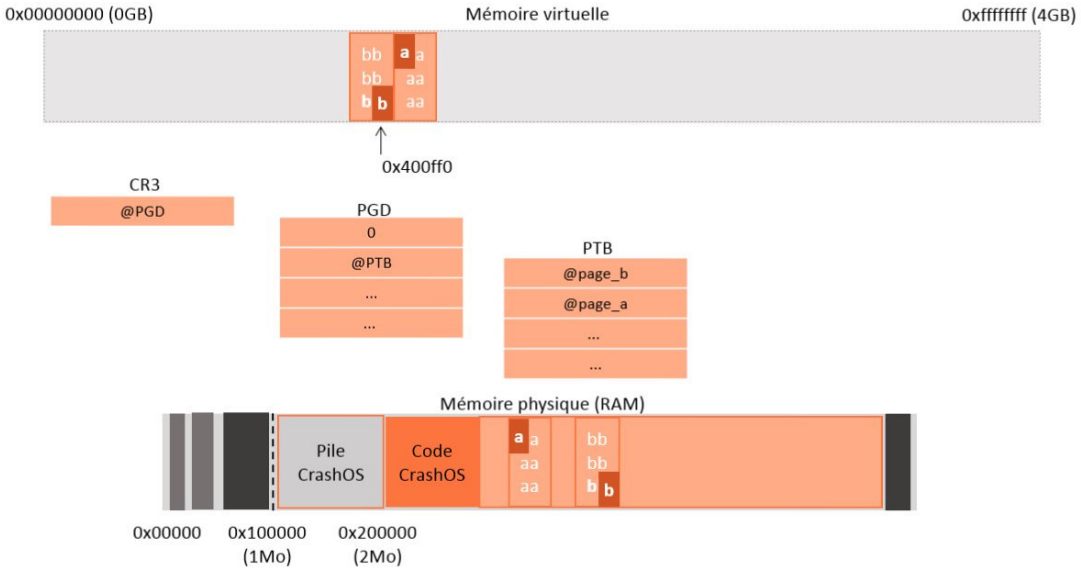

Fig. 5. Configuration mémoire avec mémoire de 2 pages physiques non contiguës

```
static int init_0xcf9() {
    init_interrupts(idt, &idtr);
    init_work_mem();
    init_segmentation(gdt, &tss, r3esp, &gdtr);
    return 0;
}

static int test_0xcf9() {
    int i;
    uint32_t port = 0xcf9;

    // OUTSB instruction - port 0xcf9 - value 0x00-0xff
    for(i=0; i < 256; i++) {
        asm volatile (
        "mov %0, %%edx          \n"
        "lea %1, %%esi          \n"
        "outsb (%%esi), %%dx    \n"
        ::"m"(port), "m"(i):);
    }
    printf_log("\noutsb done\n");
    return 0;
}
```

Listing 5. Implémentation de l'écriture de valeurs arbitraires sur le port `0xcf9`

Vérification des droits en mode protégé

Objectif Concernant les droits de pagination et de segmentation, il faut s'assurer que lorsque l'hyperviseur traite une instruction sensible, il vérifie bien les droits d'écriture, de lecture ou d'exécution de pages mémoire, ou les droits utilisateur/superviseur.

Exemples de mise en œuvre Des tests ont été effectués en tentant d'exécuter en ring 3 chaque instruction privilégiée, ou de lire ou écrire sur des segments n'ayant pas les bons droits.

Gestion du changement de mode d'exécution

Objectif Une difficulté pour un hyperviseur est de bien gérer la virtualisation des différents modes d'exécution du processeur.

Exemples de mise en œuvre On peut par exemple tenter de mettre en difficulté les hyperviseurs en mélangeant du code 16 bits dans du code 32 bits en utilisant les préfixes d'instructions `0x66` ou `0x67`, ou en basculant du mode protégé au mode réel.

Périphériques

Objectif Enfin, la communication avec les périphériques fait également partie des instructions sensibles que doit traiter un hyperviseur, que ce soit par *port I/O* ou *memory mapped I/O*.

Exemples de mise en œuvre Afin de parcourir tous les cas possibles d'écriture sur les périphériques, un test exhaustif sur l'ensemble des ports d'I/O possibles avec des valeurs de `0x00` à `0xff` a été notamment effectué. Le listing 5 donne un exemple de test rédigé dans CrashOS. Ici, il s'agit simplement de tester l'écriture de `0x00` à `0xff` sur un port d'I/O particulier.

4.2 Hyperviseurs testés

Tous les hyperviseurs n'utilisent pas les mêmes techniques de virtualisation du CPU, de la mémoire et des périphériques.

Il existe deux grandes techniques de virtualisation. Les hyperviseurs utilisant la technique de *full virtualization* supportent les OS non modifiés.

Cela signifie que lorsque l'OS invité veut exécuter une instruction sensible (par exemple mettre une valeur arbitraire dans un registre de contrôle du processeur : `MOV CRx, 0xXX`), l'hyperviseur doit l'en empêcher, intercepter cette instruction et s'occuper de son traitement. L'interception peut s'effectuer par *binary translation* en triant les instructions à la volée, ou à l'aide de l'*hardware-assisted virtualization*, via des instructions spécialisées telles que VT-X sur les processeurs Intel.
Les hyperviseurs utilisant la technique de *para-virtualization* ne font pas ce travail d'interception et n'interviennent que lorsque les OS font appel à eux. En contrepartie, les OS invités doivent être nécessairement modifiés.

Pour la gestion des ressources mémoire, l'hyperviseur doit virtualiser la MMU *(Memory Management Unit)* et donner à chaque VM une vision correcte de la mémoire physique. Il peut pour cela utiliser les *shadow page tables*, entièrement gérées par le logiciel, ou les *nested page tables* (EPT sous Intel), assistées par le matériel. Quelle que soit la méthode utilisée, il est essentiel que ces tables de pages soient correctement configurées pour assurer l'isolation des machines virtuelles entre elles et vis-à-vis de l'hyperviseur. Il doit notamment faire en sorte qu'une VM ne puisse ni lire ni modifier la mémoire d'une autre VM ou de l'hyperviseur lui-même.

Au niveau de la virtualisation des périphériques, l'hyperviseur peut soit rediriger la lecture ou l'écriture sur le périphérique physique correspondant *(pass through)*, soit émuler le périphérique concerné. Dans ce dernier cas, les périphériques étant très diversifiés et complexes, un traitement spécifique doit être effectué et la virtualisation des périphériques est souvent source de vulnérabilités dans le code des hyperviseurs.

Enfin, certains hyperviseurs supportent la *nested virtualization*, c'est-à-dire le fait d'exécuter un hyperviseur dans un hyperviseur.

Pour l'instant, CrashOS a été utilisé pour tester la robustesse des deux hyperviseurs suivants :

- Ramooflax : un hyperviseur *opensource* présenté au SSTIC en 2011 [13] reposant sur l'*hardware-assisted virtualization* ;
- VMware Workstation 12 Pro ([9, 16]) : un hyperviseur de *full virtualization* largement utilisé sur le marché, pouvant utiliser la *binary translation* ou l'*hardware-assisted virtualization*, et qui supporte également la *nested virtualization*.

4.3 Premiers résultats

Dans la plupart des cas, les deux hyperviseurs se sont montrés robustes face aux tests réalisés. Cependant, quelques cas ont provoqué des comportements inattendus.

Ramooflax

En ce qui concerne Ramooflax, l'utilisation de CrashOS a en effet permis de détecter certains dysfonctionnements de l'hyperviseur.

- mauvaise traduction d'adresse en cas de mélange de code 16 bits dans du code 32 bits (avec utilisation du préfixe 0x67) ;
- buffer overflow dans le buffer d'UART du port série virtualisé par Ramooflax lors d'écriture massive sur le port série COM1.

Le deuxième point est critique puisqu'il permet de corrompre arbitrairement la mémoire de l'hyperviseur, notamment les champs de contrôle des VM. Les dysfonctionnements sont aujourd'hui corrigés.

VMware Workstation 12 Pro

En ce qui concerne VMware, voici un résumé des cas ayant provoqué un crash de la VM ou de l'hyperviseur :

Lors de l'écriture de valeurs arbitraires sur les périphériques, nous obtenons les résultats suivants :

N° de port	Donnée	Comportement de VMware et message d'erreur
`0x1042`	valeur impaire	Arrêt de la VM – Monitor panic `VERIFY bora/devices/chipset/piix4SM.c:437`
`0x2000` `0x2040`	valeur impaire	Arrêt de la VM – Monitor panic `NOT_IMPLEMENTED bora/devices/e1000/e1000.c:2752`

`0x102c`
`0x102d`
`0x102e`
	valeur paire	Arrêt de la VM – Monitor panic
		`NOT_REACHED bora/devices/chipset/piix4PM.c:1982`

`0xcf9`	4	Échec du redémarrage de la VM (no bootable disk found)

De plus, dans le cas où on utilise la *nested virtualization* supportée par VMware, deux cas d'échecs supplémentaires ont pu être observés au niveau des périphériques :

N° de port	Donnée	Comportement de VMware en *nested virtualization*
`0xcf9`	4	Monitor panic EPT misconfiguration : PA `fffffff0`
`0x64`	254	Monitor panic EPT misconfiguration : PA `fffffff0`

Écrire `254` sur le port `0x64` ou `4` sur le port `0xcf9` provoque normalement le redémarrage du système. Lors du deuxième bug décrit ci-dessus, il semble que VMware ne réussisse pas à redémarrer la VM avec la *nested virtualization*.

Par ailleurs, un cas de test de configuration de la pagination a conduit au bug illustré sur la figure 6.

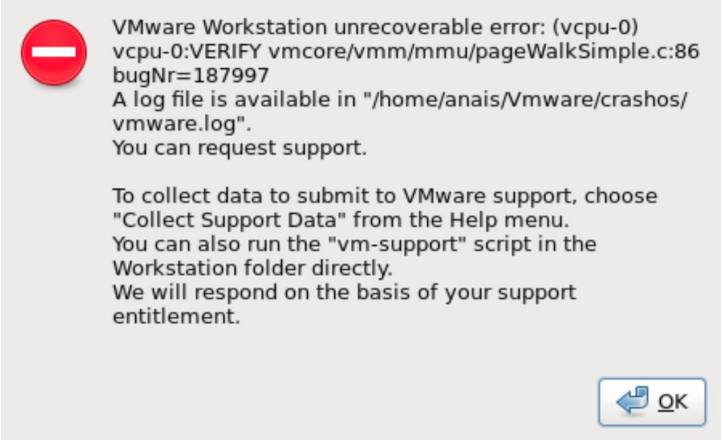

Fig. 6. Message d'erreur de VMware

Enfin, un test relatif à la modification des structures de pagination n'a provoqué aucun message d'erreur mais le résultat obtenu avec VMware diverge par rapport au comportement sur une machine physique. En effet, les modifications des structures de pagination ne sont normalement prises en compte que lorsque le registre CR3 est rechargé. VMware, quant à lui, semble mettre à jour ses structures sans attendre le rechargement de CR3.

Exploitabilité Les vulnérabilités trouvées conduisent en général à un crash de la VM, avec un message d'erreur précisant d'où provient le crash. En termes d'exploitabilité, ces crashs peuvent être utilisés pour des dénis de service sur la machine virtuelle. Cependant, ils ne sont pas exploitables pour réaliser par exemple une élévation de privilèges. Le dernier test mentionné pourrait néanmoins éventuellement être utilisé par un malware pour détecter s'il s'exécute dans un environnement virtualisé.

5 Discussion et conclusion

Le moyen proposé pour rechercher des vulnérabilités dans les hyperviseurs prend la forme d'un système d'exploitation entièrement configurable. Dans son état actuel, ce système d'exploitation, CrashOS, a pu être utilisé pour rédiger et lancer un certain nombre de tests. Cela a permis de détecter des dysfonctionnements voire des vulnérabilités dans les deux hyperviseurs testés.

CrashOS est cependant en cours de développement. Les prochaines fonctionnalités à implémenter en priorité sont les suivantes :

– gérer les hypercalls pour pouvoir disposer d'un moyen de tester les hyperviseurs de para-virtualisation tels que Xen ;
– ajouter des fonctionnalités d'hyperviseur pour maîtriser finement les tests de *nested virtualization*.

L'idée est également d'une part d'étendre notre étude à d'autres hyperviseurs des architectures x86 (Virtualbox, Xen, KVM, etc.) et d'autre part d'enrichir notre base de tests pour détecter le maximum de vulnérabilités possible.

Références

1. B2G. https://github.com/mozilla-b2g.
2. BareMetal-OS. https://github.com/ReturnInfinity/BareMetal-OS.
3. Driver de la carte graphique. http://www-igm.univ-mlv.fr/~dr/XPOSE2004/rleonard/codeVideomem.php.
4. OSv. https://github.com/cloudius-systems/osv.
5. Redox. https://github.com/redox-os/redox.
6. Airbus (A. Gantet). CrashOS. https://github.com/airbus-seclab/crashos.
7. Common Vulnerabilities and Exposures website. CVE. http://cve.mitre.org/cve/cve.html.
8. David Decotigny, Thomas Petazzoni. SOS. http://sos.enix.org/fr/SOSDownload.
9. Edouard BUGNION, Scott DEVINE, Mendel ROSENBLUM, Jeremy SUGERMAN, Edward Y. WANG. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. http://www.cs.columbia.edu/~cdall/candidacy/pdf/Bugnion2012.pdf.
10. Erich Stefan Boleyn. GNU GRUB. https://www.gnu.org/software/grub/index.html.
11. Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, Berk Sunar. Fine grain Cross-VM Attacks on Xen and VMware are possible. https://eprint.iacr.org/2014/248.pdf.
12. Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, Yuriy Bulygin. Attacking Hypervisors via Firmware, BHUSA 2015. http://www.intelsecurity.com/advanced-threat-research/content/AttackingHypervisorsViaFirmware_bhusa15_dc23.pdf.
13. S. Duverger. Ramooflax. https://www.sstic.org/media/SSTIC2011/SSTIC-actes/virtualisation_dun_poste_physique_depuis_le_boot/SSTIC2011-Article-virtualisation_dun_poste_physique_depuis_le_boot-duverger_1.pdf.
14. Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, Jacob R. Lorch. SubVirt : Implementing malware with virtual machines. http://web.eecs.umich.edu/~pmchen/papers/king06.pdf.
15. Securimag. VESPA. https://github.com/Orange-OpenSource/vespa-core.
16. VMware®. VMware® Workstation 12 Pro. http://www.vmware.com/fr/products/workstation/workstation-evaluation.html.