# Etude bas niveau de la mémoire et porte dérobée de VMware

Christian Abegg — IN3

19 avril 2010

## 1 Objectifs

Cette étude a pour but de concevoir un noyau de système d'exploitation afin de pouvoir lire et écrire la mémoire vive sans aucune restriction. Cela permet de voir si la mémoire vive est remise à zéro par VMware. Dans un deuxième temps, quelques fonctionnalités de la porte dérobée de VMware seront testées.

## 2 Analyse de la problématique

Dans un serveur de virtualisation, par exemple ESX, la mémoire vive physique est partagée entre toutes les machines virtuelles. Il est donc absolument indispensable que l'isolation de la mémoire soit garantie. En clair, une machine virtuelle ne devrait pas avoir accès aux données d'une autre machine. Cela correspond à la pagination de la mémoire vive qui est pratiquée par les systèmes d'exploitation modernes.
De plus, lors du démarrage d'une machine virtuelle, cette dernière va se voir allouer une certaine quantité de mémoire vive qui a peut être été utilisée par une autre machine auparavant (ballooning, etc.). Il est donc de première importance que cette mémoire soit réinitialisée avant d'y donner accès à la nouvelle machine virtuelle. Sans quoi, il serait possible de relire et donc de compromettre des informations confidentielles.

Une autre brèche dans le système de sécurité de VMware, pourrait être la porte dérobée nécessaire à la communication entre l'hyperviseur et la machine virtuelle. Ce canal de communication est notamment utilisé pour l'intégration de la souris, la copie du presse-papier entres-autres.

# 3 Détails techniques

## 3.1 Choix de la base logicielle

Afin de pouvoir accéder à notre guise à la mémoire vive, il est impératif d'être en mode privilégié, soit le mode *ring0*. L'utilisation d'un système d'exploitation classique tel que FreeBSD, GNU/Linux ou Windows, n'est pas possible puisque tous les programmes tournent en mode *ring3* et la mémoire est segmentée. A l'exécution de la première instruction privilégiée, le programme sera immédiatement tué par le système d'exploitation afin de ne prétériter la stabilité et la sécurité de ce dernier. La seule possibilité serait d'écrire un pilote qui lui sera automatiquement en mode privilégié. Cette méthode étant de loin pas évidente.

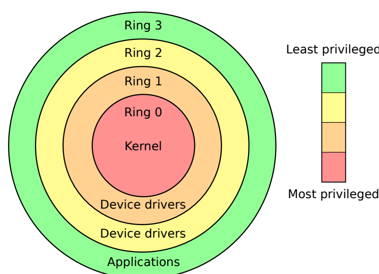



FIGURE 1 – Anneaux de protections de l'architecture x86. Image tirée de Wikipédia : `http://en.wikipedia.org/wiki/File:Priv_rings.svg`

L'environnement de développment retenu, est le micro-système d'exploitation *SOS : Simple Operating System* conçu par David Decotigny & Thomas Petazzoni[5] dans le cadre d'une série d'articles sur la conception de systèmes d'exploitation dans la revue *Linux Magazine France*.

Le travail effectué dans cette étude a eu comme base le code de l'article 2 de SOS. Cette base est constituée d'un secteur d'amorce, d'un code d'initialisation de la sortie vidéo et d'une méthode `main()`. A l'exception du secteur d'amorce qui est écrit en assembleur x86, tout est codé en C. Il est donc très aisé de modifier et d'ajouter le code souhaité.

Après le démarrage, la méthode `main()` est exécutée en mode privilégié. On peut donc utiliser sans restriction le jeu d'instructions x86 complet. De plus, toute la mémoire peut être accédée.

## 3.2 Secteur d'amorce

Comment un système d'exploitation démarre-t-il ? Voici une explication de l'amorçage, sans entrer dans les détails. Pour une explication exhaustive, l'article 1 de SOS est une très bonne référence.

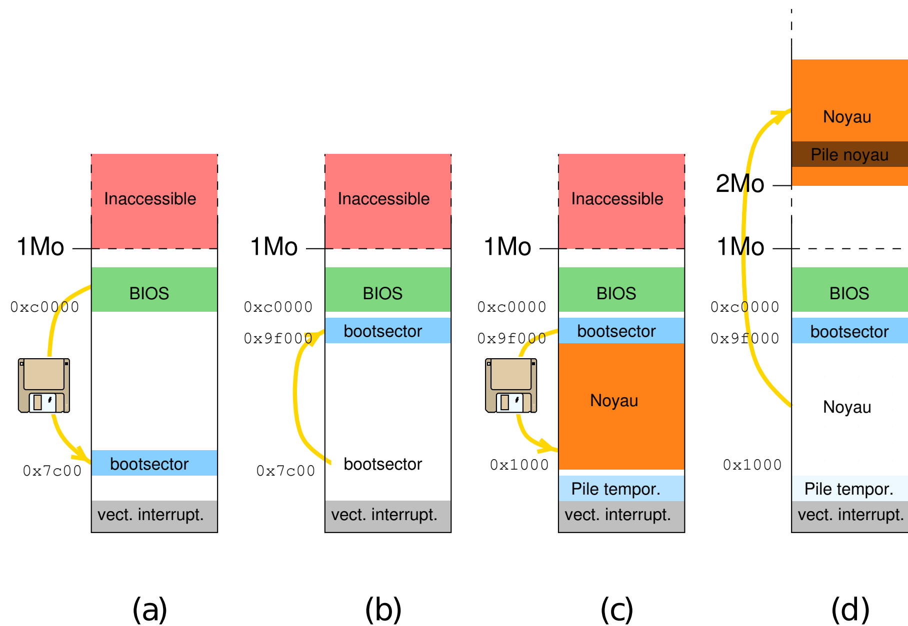



FIGURE 2 – Etapes de chargement du noyau par le secteur d'amorce. Source : Article 1 de SOS [5]

Tout d'abord, il faut savoir que la manière de démarrer un système d'exploitation n'a guère évolué depuis les premiers processeurs x86. Cela permet de garantir la compatibilité. Ce démarrage comporte quatre parties distinctes, expliquées dans les points suivants en correspondance avec la figure 2

(a) Après exécution, le BIOS [1] va chercher un secteur d'amorce parmi les disques accessibles. Il s'agit toujours du premier secteur d'un disque qui a comme signature `0x55AA`. Une fois trouvé, ce secteur est copié à l'adresse `0x7C0`. Dès la copie terminée, le BIOS passe la main au processeur.

(b) Pour l'instant, seul le premier méga-octet de mémoire vive est adressable. Cela vient du fait que le processeur se trouve en mode réel [2]. Le BIOS et la mémoire de la carte vidéo se trouvant également dans ce segment mémoire, il reste exactement 640 ko pour y placer le noyau. Cette étape consiste donc à déplacer le secteur d'amorce à une adresse plus élevée.

(c) Il s'agit maintenant de copier le noyau depuis le disque dans la mémoire vive. Cette opération s'effectue à l'aide de l'interruption `int 0x13`. Cette interruption est en fait un appel au BIOS permettant de lire et d'écrire sur un disque.

(d) Le code chargé à l'étape précédente, va s'occuper d'activer le mode protégé du processeur, soit le mode 32 bits. De plus, le noyau sera déplacé à une zone mémoire plus élevée (au-dessus de 2 Mo). A ce moment-là, le noyau va lancer la méthode `main()` du système d'exploitation SOS.

1. Basic Input/Output System, est le système d'entrée/sortie élémentaire fournissant des méthodes telles que l'affichage en mode texte ou encore l'accès aux disques.
2. Il s'agit du mode 16 bits du x86 tel que défini avec le 8086 d'Intel en 1978.

## 3.3 Utilisation de la sortie vidéo

Afin de pouvoir afficher à l'écran, plusieurs méthodes sont à disposition, notamment par le BIOS, le *framebuffer* ou en faisant directement appel à la carte graphique. La méthode retenue est l'utilisation de l'interruption `int 0x10` [3] du BIOS. La palette d'options disponible est très spartiate. Il s'agit de mettre dans les registres `EAX`, `EBX`, `ECX` et `EDX` du processeur les informations que l'on souhaite afficher. Deux modes d'adressage sont disponibles : par caratère ou par pixel. Cela permet au choix de dessiner des graphiques ou d'afficher du texte. Le jeu de caractères disponible est le `cp437` [4]. Ce jeu regroupe tous les caractères ASCII, la plupart des diacritiques ainsi que des éléments graphiques. Ces derniers seront utilisés pour afficher la « carte » de la mémoire.

Pour permettre l'affichage dans le cadre de cette étude, les méthodes d'affichage de chaînes de caractères fournies par SOS seront utilisées. Leur comportement est similaire à la méthode `printf()` du langage C.

## 3.4 Accès mémoire

Maintenant que la base logicielle est en place, il est possible d'exécuter toutes les instructions x86 et d'accéder à toute la mémoire sans restriction aucune. Cette partie est gérée en trois étapes :

(a) Il s'agit de lire les 36 premiers Mo de la mémoire vive. La « carte » de la mémoire s'étend sur 16 lignes et 72 colonnes. Chaque caractère représente 32768 octets. Le programme va lire chacun de ces blocs, octet par octet, en comptant le nombre de bit à `'1'` trouvés. Ce nombre sera utilisé pour le choix du caractère représentant le bloc. Plus le caractère [5] est rempli, plus il y a de bit à `'1'` dans le bloc.

(b) La suivante partie consiste à écrire un motif quelconque dans toute cette partie de la mémoire vive. Pour des raisons évidentes, rien n'est écrit en dessous de la barre des 2 Mo. Ecrire à cette adresse, reviendrait à écraser la mémoire vidéo, le BIOS ainsi que le noyau ce qui planterait immédiatement l'ordinateur. Il faut donc être prudent, surtout qu'aucun mécanisme empêche cet écrasement.

(c) Finalement, toute la mémoire est relue et la « carte » de la mémoire est mise à jour. L'exécution tourne ensuite en boucle.

La figure 3 montre l'architecture d'un hyperviseur. Dans le cas de l'isolation mémoire, aucun des OS invités ne doit pouvoir accéder à des informations d'autres OS invités. Il est intéressant de noter que la mémoire vive lue dans la machine virtuelle [6] est toujours égale à zéro comme le montre la figure 4. La mémoire est donc initialisée par l'hyperviseur avant toute utilisation dans la machine virtuelle. En guise de comparaison, sur une machine physique, ce n'est pas le cas, voir la figure 5. Si l'on redémarre à chaud la machine, le contenu de la mémoire vive reste intacte. Ainsi, la « carte » de la mémoire est rigoureusement la même après

---

3. Une explication des différentes commandes est disponible à l'adresse suivante : `http://en.wikipedia.org/wiki/Interrupt_0x10`
4. Code page 437, il s'agit du jeu de caractères du PC d'IBM. Un tableau exhaustif est disponible à l'adresse suivante : `http://en.wikipedia.org/wiki/Code_page_437`
5. Pour la légende complète, voir la dernière ligne de la figure 4
6. tests effectués avec VMware et VirtualBox

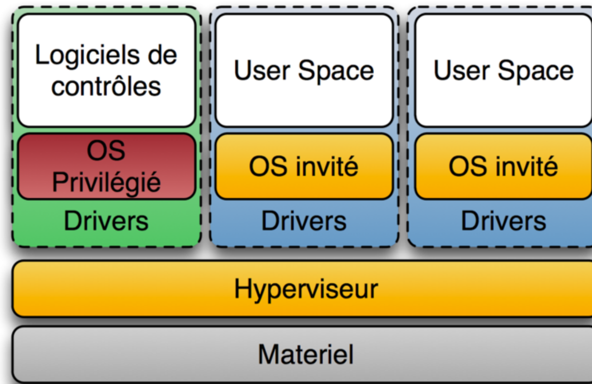
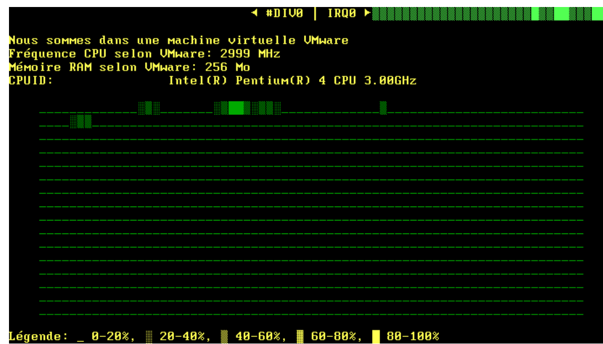



FIGURE 3 – Architecture de virtualisation avec un hyperviseur. Image tirée de Wikipédia : http://fr.wikipedia.org/wiki/Fichier:Diagramme_ArchiHyperviseur.png

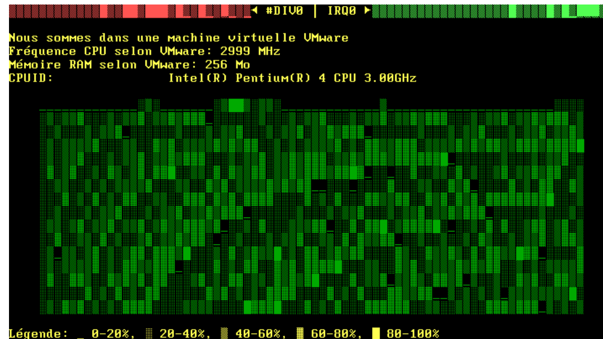
redémarrage. Il est même possible d'éteindre la machine pendant quelques minutes, avant de la rallumer et de retrouver la mémoire vive en grande partie non altérée (!). Ce délai de conservation est lié à la température des barettes de mémoire [7].
Extrait de code pour l'accès à la mémoire vive :

```
/*
 * Définition d'un pointeur vers un char non signé
 * La variable adresse est un entier qui représente
 * l'adresse mémoire que l'on souhaite atteindre.
 */
unsigned char *addr = adresse;

/*
 * L'instruction suivante permet de lire le contenu de
 * la case mémoire directement dans la variable valeur.
 * En langage C, cela revient à lire la valeur pointée
 * par le pointeur addr.
 */
unsigned char valeur = *addr;

/*
 * Pour écrire dans cette case mémoire, il suffit d'inverser
 * les opérandes. Ici, on écrit 42 dans cette case mémoire.
 * En langage C, cela revient à écrire dans la case mémoire
 * pointée par le pointeur addr.
 */
*addr = 42;
```

7. Une étude très intéressante sur le sujet : http://citp.princeton.edu/memory/, avec un article très complet : http://citp.princeton.edu/pub/coldboot.pdf




(a) Etat de la mémoire vive avant écriture




(b) Etat de la mémoire vive après écriture

FIGURE 4 – « Carte » de la mémoire dans VMware. Les blocs visibles sur la première ligne, sont dans l'ordre : mémoire vidéo, BIOS et noyau SOS. A chaque démarrage, l'état de la mémoire vive est le même.

Le code source complet des méthodes d'accès à la mémoire en lecture et écriture se trouve dans le fichier `sos/laboesx/memoire.c` de l'archive.

## 3.5 Porte dérobée VMware

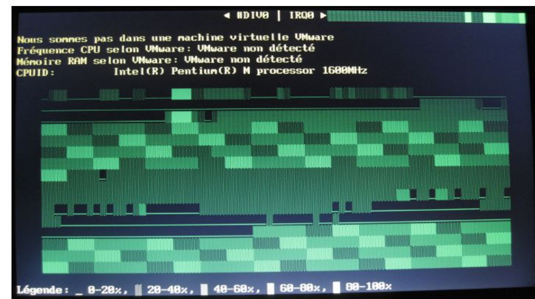
La porte dérobée est le « canal de service » entre l'hyperviseur et la machine virtuelle. Les outils VMware se basent exclusivement sur ce canal. La grande majorité des méthodes ne sont pas documentées ou alors elles le sont de manière officieuse, c'est-à-dire par rétro-ingénierie [3]. Un moyen de découvrir ces méthodes serait d'analyser le code source des *Open Virtual Machine Tools* [8], il s'agit de la mise en œuvre libre des outils VMware.

Dans le cadre de cette étude, trois instructions de service ont été testées. Il s'agit des méthodes permettant d'obtenir la vitesse du processeur, la quantité de mémoire vive allouée au système invité ainsi que la version de l'hyperviseur. Cette dernière est en réalité utilisée pour déterminer si l'on a affaire à VMware ou non.

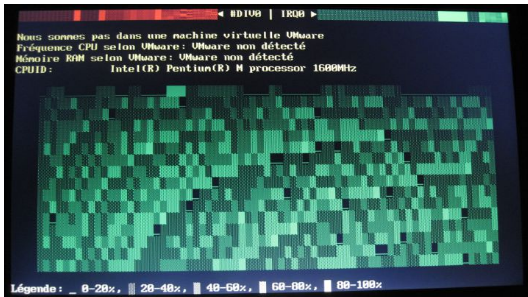
Ces commandes n'étant pas des accès mémoires, mais des accès à des périphériques d'entrée/sortie, il est nécessaire de passer par du code assembleur dont voici l'extrait permettant

8. `http://open-vm-tools.sourceforge.net/`

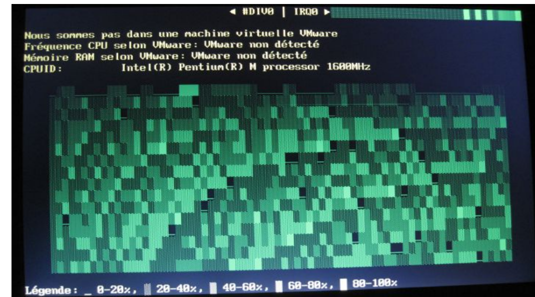
(a) Etat de la mémoire vive avant écriture, après démarrage à froid




(b) Etat de la mémoire vive avant écriture, mais après redémarrage à chaud depuis Linux




(c) Etat de la mémoire vive après écriture




(d) Etat de la mémoire vive après écriture, arrêt de la machine pendant 30 s. La mémoire n'a pas changé.

FIGURE 5 – « Carte » de la mémoire d'un PC IBM. Les blocs visibles sur la première ligne, sont dans l'ordre : mémoire vidéo, BIOS et noyau SOS. On peut parfaitement constater que la mémoire n'est jamais initialisée. La mémoire laisse apparaître des motifs intéressants.

d'obtenir la taille de la mémoire vive :

```
/*
 * Retourne la taille de la RAM en Mo selon VMware
 * Si VMware n'est pas détecté, 0xFFFFFFFF sera retourné
 * Uniquement testé en mode 32 bits.
 */
int vmware_get_memsize() {
  unsigned int mem = 0;

  /* Ecriture de la valeur magique dans l'accumulateur */
  asm(" movl $0x564D5868, %eax; ");

  /* Ecriture de la commande désirée dans le registre CX */
  asm(" movw $0x0014, %cx;");

  /* Choix du port I/O de VMware dans le registre DX */
  asm(" movw $0x5658, %dx;");
```

```
  /*
   * Lecture du port I/O, c'est à cet instant que VMware
   * répond à la demande
   */
  asm(" inl %dx, %eax;");

  /*
   * Lecture de l'accumulateur contenant la taille de la
   * mémoire vive en Mo.
   * Il s'agit de la syntaxe de GNU as qui permet de lire
   * un registre directement dans une variable, ici mem.
   */
  asm(" movl %%eax, %0;" :"=r"(mem));

  return mem;
}
```

Le code source complet des méthodes permettant d'appeler les trois méthodes de la porte dérobée se trouve dans le fichier `sos/laboesx/vmware.c` de l'archive.

# 4 Utilisation du logiciel

Cette partie explique comment utiliser le fichier fourni ainsi que la compilation du code source, par exemple si l'on souhaite effectuer des modifications.

## 4.1 Image disque `sos_qemu.img`

Instructions pour exécuter le programme avec VMware

(a) Extraire le fichier `sos/extra/sos_qemu.img` de l'archive fournie,

(b) Créer une nouvelle machine virtuelle, avec au moins 64 Mo de mémoire vive, sans disque dur et avec un lecteur de disquette,

(c) Envoyer le fichier `sos_qemu.img` dans le *datastore* du serveur ESX,

(d) Choisir « Connected » et « Connected at power on » dans les paramètres du lecteur disquettes. Indiquer comme image disque, le fichier `sos_qemu.img` qui se trouve dans le *datastore*.

La machine virtuelle est à présent prête à exécuter le code.

## 4.2 Compilation

Instructions pour compiler le programme. Un environnement de développement GNU/Linux avec GCC est nécessaire.

(a) Décompresser intégralement l'archive fournie,

(b) Se placer dans le répertoire `sos/extra`,

(c) Exécuter `make` qui s'occupera de compiler le code source et de créer l'image disque.

L'image disque fraîchement créée est disponible ici : `sos/extra/sos_qemu.img`. Les fichiers contenant le code de cette étude sont :
– tous les fichiers dans le répertoire `sos/laboesx`
– le fichier contenant la méthode `main()` : `sos/sos/main.c`

# 5 Conclusion

Cette étude a permi de découvrir le fonctionnement bas niveau d'un système d'exploitation, surtout son démarrage. Le programme écrit a pu confirmer le fait que VMware initialise la mémoire vive avant que la machine virtuelle puisse y accéder, ce qui ne remet pas en cause l'isolation.

Pour une étude plus appronfondie, l'utilisation d'un vrai système exploitation couplé à un débogueur *ring0* tel que `rr0d` [9] ou WinDbg [10] serait une voie intéressante à explorer. Une autre approche plus bas niveau est d'effectuer un débogage par IEEE-1394 [11]

L'utilisation de la porte dérobée de VMware confirme la possibilité de communication entre une machine virtuelle et l'hyperviseur. On pourrait imager effectuer du *fuzzing* [12] et observer le comportement de VMware.

La solution d'utiliser la base de code du système SOS s'est révélée être le bon choix. D'une part les articles détaillés que l'on peut que recommander ainsi que le code très bien documenté ont été une aide précieuse pour ce travail.

# Références


[1] Intel Software Developer's Manuals. `http://www.intel.com/products/processor/manuals/`.

[2] Wikipédia. Consultation de divers articles techniques : `http://www.wikipedia.org/`.

[3] Ken Kato. VMware Backdoor I/O Port. `http://chitchat.at.infoseek.co.jp/vmware/backdoor.html`.

[4] Pierre Maurette. *Assembleur*. Micro Application, 2003.

[5] David Decotigny & Thomas Petazzoni. Croisière au cœur d'un OS. 2004-2009. Tous les articles et codes sources sont disponibles à l'adresse suivante : `http://sos.enix.org/`.


9. Rasta Ring 0 Debugger : `http://rr0d.droids-corp.org/`
10. Débogueur de drivers Windows de Microsoft : `http://www.microsoft.com/whdc/devtools/debugging/default.mspx`
11. Le contrôleur IEEE-1394 permet un accès total à la mémoire vive car le protocole a été conçu pour communiquer avec le contrôle DMA. Il est donc possible de lire et d'écrire la mémoire vive sans même que le processeur puisse le savoir. Papier sur le sujet : `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf` Sous GNU/Linux : `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob_plain;f=Documentation/debugging-via-ohci1394.txt;hb=HEAD`
12. Méthode de test des logiciels en injectant des données aléatoires : `http://fr.wikipedia.org/wiki/Fuzzing`