VirtDbg

Damien Aumaitre
Christophe Devine

Plan

Plan

Génèse du projet ?

Envie(s)

- Étudier Windows 7 64 bits, en particulier PatchGuard, la signature des drivers, les DRM, ...
- Réutiliser le framework de manipulation de la mémoire physique présenté lors de l'édition 2008 du SSTIC

Comment ?

- Utiliser un débogueur ring 0
- Mais lequel ?

Plan

Panorama des débogueurs ring 0

Un peu de terminologie

Cible	Machine contrôlée par le débogueur
Débogueur	Contrôle la cible
Utilisateur	Celui qui agit sur le débogueur

Deux grandes familles :

- Débogueurs locaux : SoftICE, Syser, rr0d, ...
- Débogueurs distants : WinDbg, gdb

Débogueurs locaux

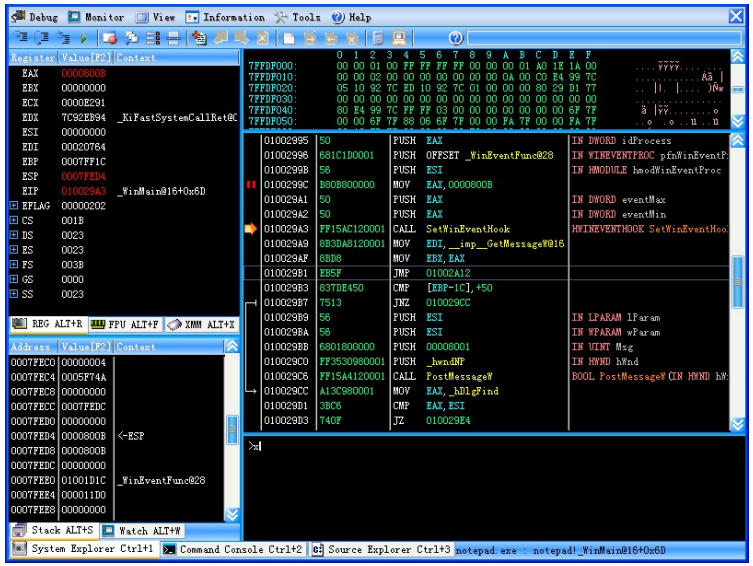
SoftICE, Syser, rr0d ...

Avantages

- Nécessite une seule machine

Inconvénients

- Difficilement extensibles, scriptables (copier-coller ?)
- Communications avec l'extérieur ?
- Manipulent directement le framebuffer, le clavier et la souris ← pas portable



Débogueurs distants

WinDbg, gdb ...

Avantages

- Extensibles
- Nombreuses fonctionnalités

Inconvénients

- Besoin de deux machines, interface spartiate
- WinDbg demande à booter en mode /DEBUG ⇒ désactive PatchGuard
- gdb nécessite un stub sur la machine cible (par exemple : VMware)
- gdb est peu adapté au déboguage noyau (vue centrée sur un processus, manque de primitives standardisées)

Plan

Comment fonctionne un débogueur noyau ?

Interception de certaines interruptions du processeur

Cas de l'architecture IA-32

- Les vecteurs d'interruption sont contenus dans l'IDT (Interruption Descriptor Table)
- L'interruption 1 est utilisée par les points d'arrêts matériels (single step et utilisation des registres de debug DR)
- L'interruption 3 est utilisée par les points d'arrêts logiciels
- Ils peuvent aussi intercepter l'interruption 14 (fautes de pages) pour poser des points d'arrêts mémoire

Comment fonctionne un débogueur noyau ?

Deux modes de fonctionnement :

- Mode de debug où le débogueur attend des requêtes de l'utilisateur
- Sinon fonctionnement normal du système

Cycle de vie

- Passage en mode debug ("Breakin"), le système est alors arrêté et le débogueur attend des requêtes de l'utilisateur
  - Inspection des registres du processeur
  - Inspection de la mémoire
  - Pose de points d'arrêts
- La requête "Continue" arrête le mode debug et le système d'exploitation continue alors son exécution

Plan

Revenons à nos moutons

Comment procéder ?

Utiliser WinDbg ?

- Nombreuses fonctionnalités, très simple à utiliser
- Interface spartiate $\Rightarrow$ pas trop grave
- Mais PatchGuard et les DRM se désactivent en mode DEBUG

Créer son propre stub de debug ?

Problématiques :

- Comment passer en mode noyau sous Windows 7 ?
- Comment hooker l'IDT avec PatchGuard activé ?
- Comment interagir avec le stub de debug ?

Problème n° 1

Drivers signés

- Existent sous les plateformes 64 bits depuis Vista
- Protègent l'accès au noyau
- Désactivable mais effets de bord préjudiciables à notre analyse

Contournons la signature

- Exécution de code arbitraire avec des requêtes DMA sur le bus PCI (cf. SSTIC 2008, 2009)
- Les requêtes DMA se font à l'insu du processeur et donc de l'OS

Bilan

Exécution de code arbitraire ⇒ chargement d'un driver noyau à l'insu de l'OS

Problème n° 2

PatchGuard ?

PatchGuard sert à protéger le noyau, il va donc :

- Empêcher la modification du code noyau et de certains drivers
- Surveille les structures les plus importantes (SSDT, . . .)
- Provoque un écran bleu si un changement est détecté
- Désactivable aussi mais ce n'est pas ce que nous voulons !

Contournons Patchguard

- Problème principal : non-modification de l'IDT
- Si l'on peut exécuter du code arbitraire, comment contrôler les différents interruptions sans hooker l'IDT ?
- Solution : utiliser la virtualisation matérielle !
  - Propose un contrôle très fin du système cible
  - Tout ce qu'il faut pour faire du debug noyau
  - Extrêmement furtif (cf. BluePill)

Problème n° 3

Communication avec l'extérieur

- Classiquement les débogueurs ring 0 utilisent une interface série, facile à programmer mais lente
- Moyens de communications rapides : Ethernet, FireWire et USB
- Ethernet, USB fortement dépendants du matériel
- FireWire serait idéal (cf. utilisation dans WinDbg)
- Que choisir ?

Solution retenue

Lectures/écritures (accès "DMA") en mémoire physique

- Pas de code dépendant du matériel ⇒ empreinte minimale sur la cible
- Fait suite aux travaux présentés au SSTIC 2009

Cahier des charges

Contraintes

Empreinte minimale sur le système cible (utiliser le moins possible les fonctions de l'OS et ne pas modifier de structures du système)

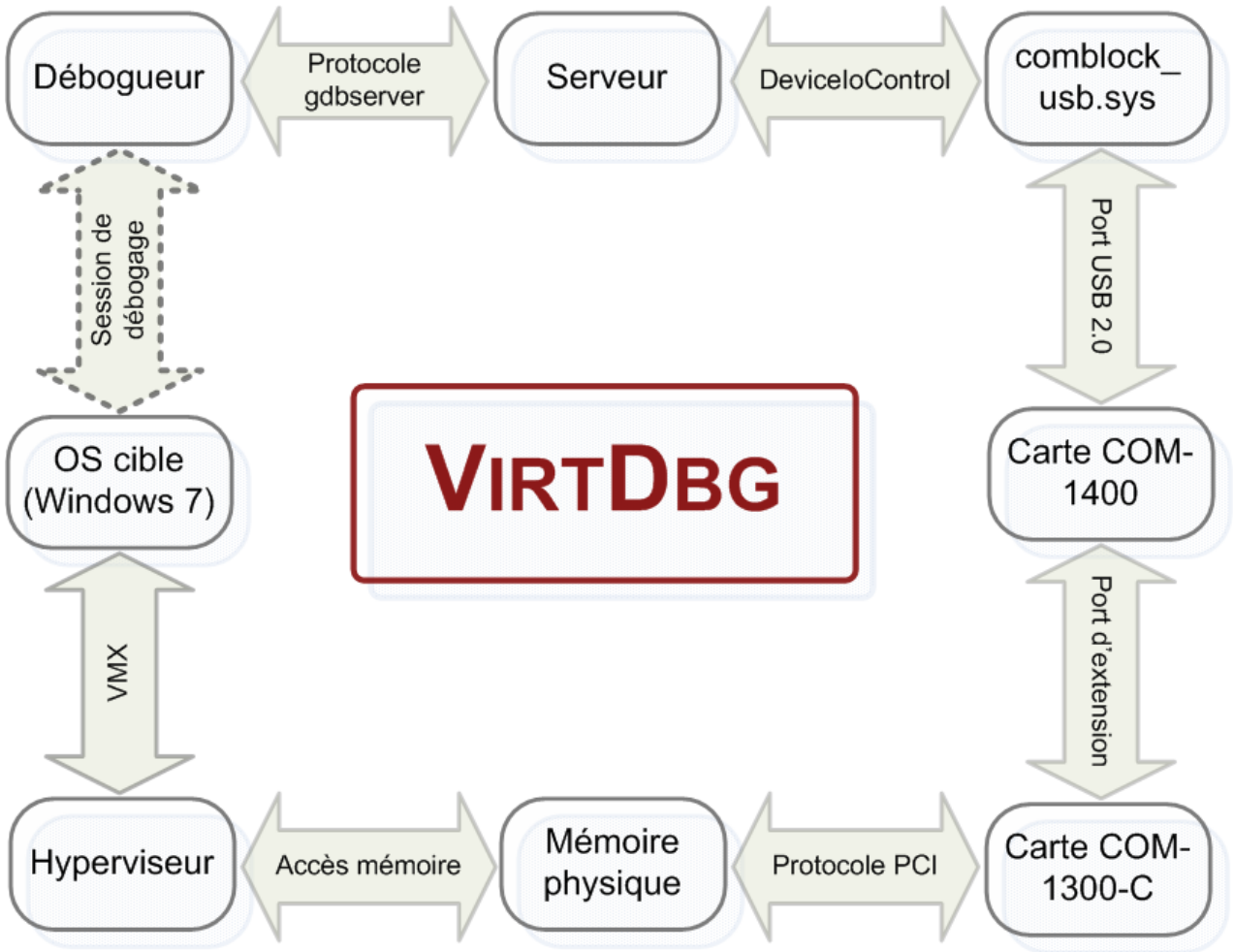
Fonctionnalités

- Furtif par conception
- Déporter les fonctionnalités lourdes du coté client
- Utilisable en tant que stub gdb

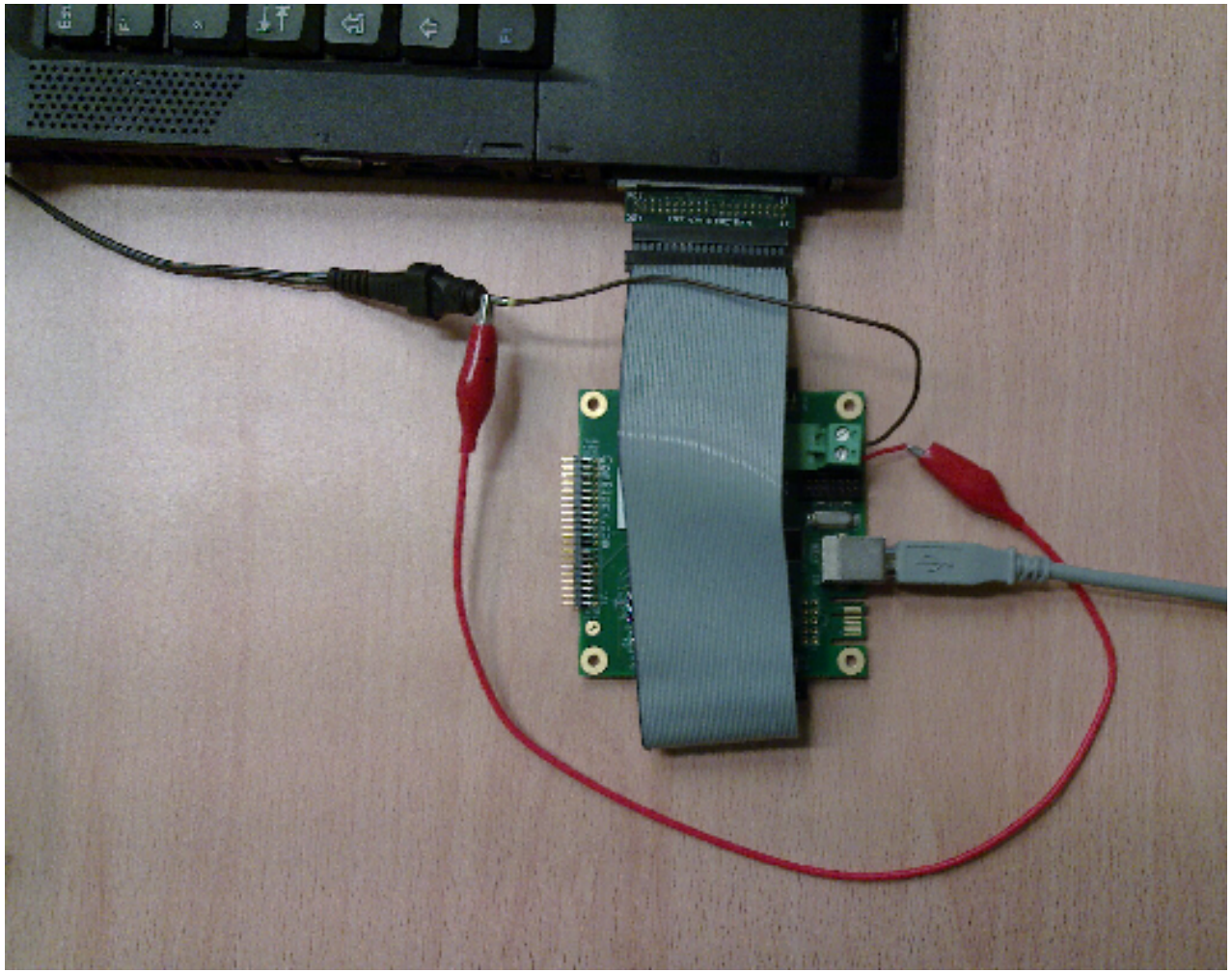
Finalités

- Etude de composants dépendants du matériel (DRM par exemple)
- Analyse de malwares dotés de protections (anti-VM et anti-émulation)
- Débogueur "ring -1" scriptable, sur du code théoriquement non déboguable (gestionnaire d'interruptions)

Virtdbg


Débogueur
Protocole gdbserver
Serveur
DeviceIoControl
comblock_ usb.sys
Session de débogage
Port USB 2.0
OS cible (Windows 7)
VIRTDBG
Carte COM-1400
VMX
Port d'extension
Hyperviseur
Accès mémoire
Mémoire physique
Protocole PCI
Carte COM-1300-C

Photo



Plan

Plan

Rappels : le DMA (Direct Memory Access)

DMA ?

- Historiquement les I/O sont faites par le processeur : très lent car cela monopolise le CPU.
- Principe : utiliser un contrôleur dédié pour faire les transferts
- Implémenté au niveau du bus PCI
- Tout périphérique relié au bus PCI peut donc les utiliser (FireWire, PCMCIA, ExpressCard, . . .)

Faille de conception

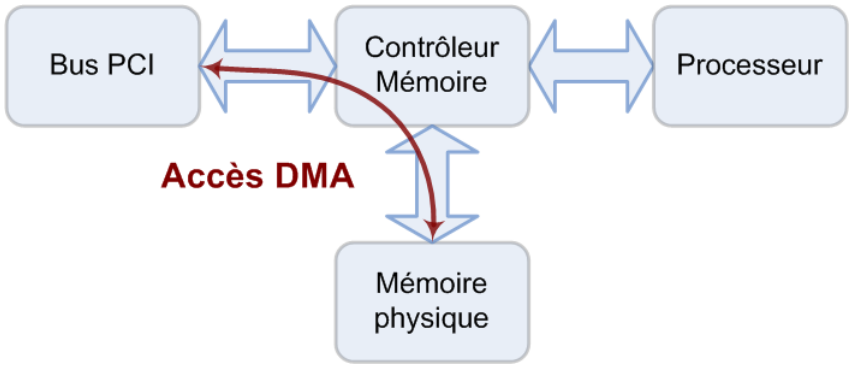
- Le processeur (et l'OS) ne sont pas conscients des transferts DMA !

Rappels : le DMA (Direct Memory Access)

Conséquences

- Lecture/Écriture dans la mémoire **physique**
- Equivaut à lire la mémoire du système d'exploitation
- Permet de contourner tous les dispositifs de sécurité du processeur et du système d'exploitation

Exemple d'accès DMA :



Attaque par DMA

Principe de l'attaque

- Utiliser un périphérique malicieux (iPod, CardBus, etc.)
- Obtenir un accès Bus Master sur le bus PCI (permet d'initier des transferts DMA)

Travaux précédents

- Utilisation du Firewire pour accéder à la mémoire physique :
  - 2004 – Maximillian Dornseif (Mac OS X)
  - 2006 – Adam Boileau (Windows XP)
  - 2008 – Damien Aumaitre (Windows & Mac)
- Utilisation d'une carte CardBus avec FPGA :
  - 2009 – Christophe Devine et Guillaume Vissian

FPGA sur CardBus : nouveaux développements (2010)

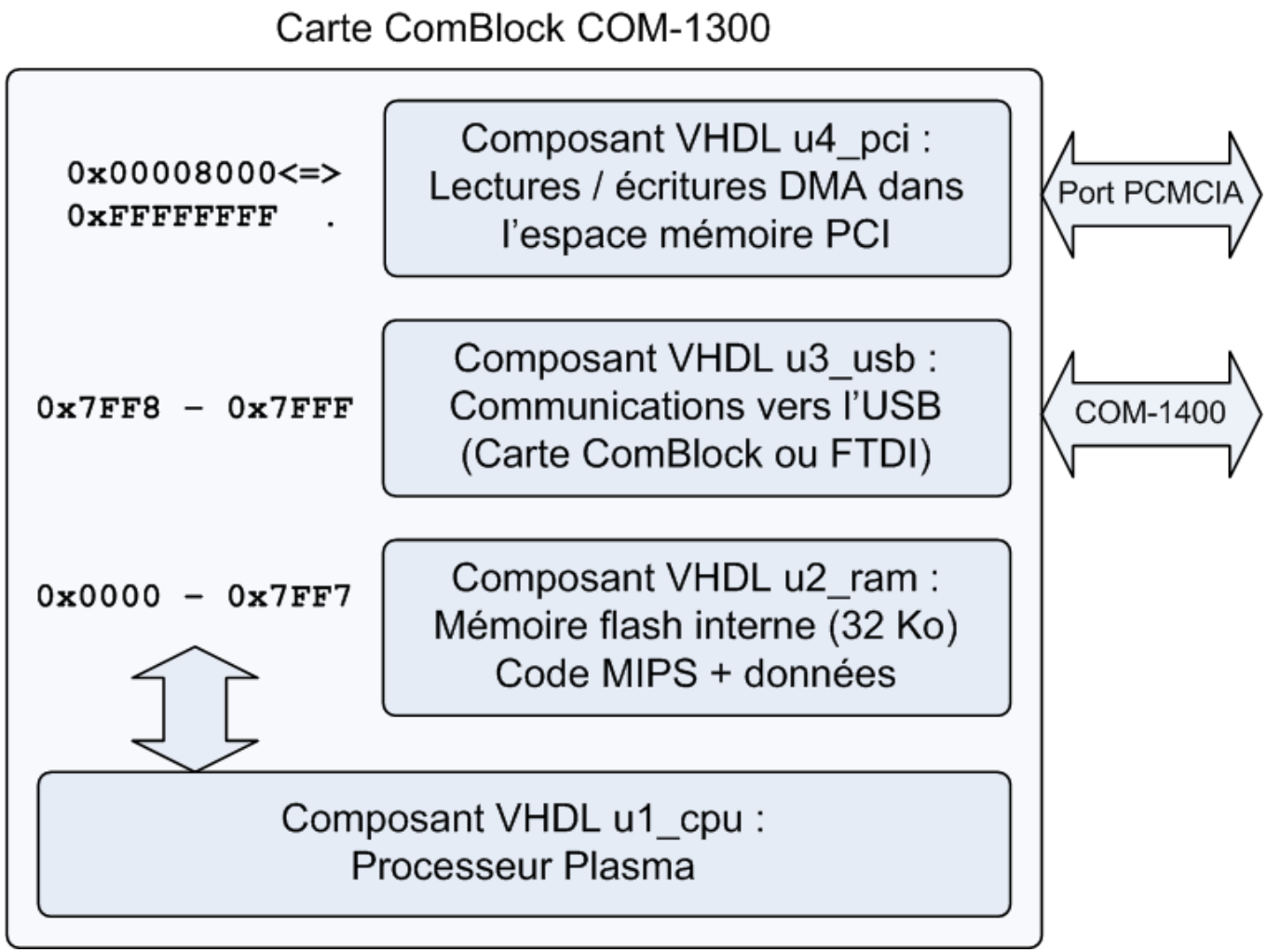
Errata

- Correction d'un bogue majeur lié aux lectures DMA :
  - "A master which is target terminated with Retry must unconditionally repeat the same request until it completes"
- Non-désactivation par le driver PCMCIA avec deux astuces :
  - Lecture à vide tous les 1000 cycles ⇒ pas de mise en veille
  - "Randomisation" du subsystem id ⇒ nouveau périphérique détecté à chaque insertion, le DMA reste autorisé

Réécriture "from scratch"

- Utilisation du code VHDL d'un processeur MIPS sous domaine public ("plasma", opencores.org)
- Permet la programmation en C du micro-code sur le FPGA

FPGA sur CardBus : vue d'ensemble



FPGA sur CardBus : exemple de programmation

Déverouillage d'un portable sous Windows 7 x64

Recherche de signature dans toute la mémoire physique

```
for (i = PHYS_MEM_START; i < PHYS_MEM_SIZE; i += 0x1000)
{
    DMA_PAUSE
    l = (unsigned char*)(i + 0x290);
    if (*(unsigned int*)l == 0x850fc63b)
    {
        DMA_PAUSE
        if (*(unsigned int*)(l + 4) == 0xb8c0)
        {
            DMA_PAUSE
            *(unsigned int*) l = 0x840fc63b;
            for (;;);
        }
    }
}
```

Communications PC débogueur ⇔ FPGA

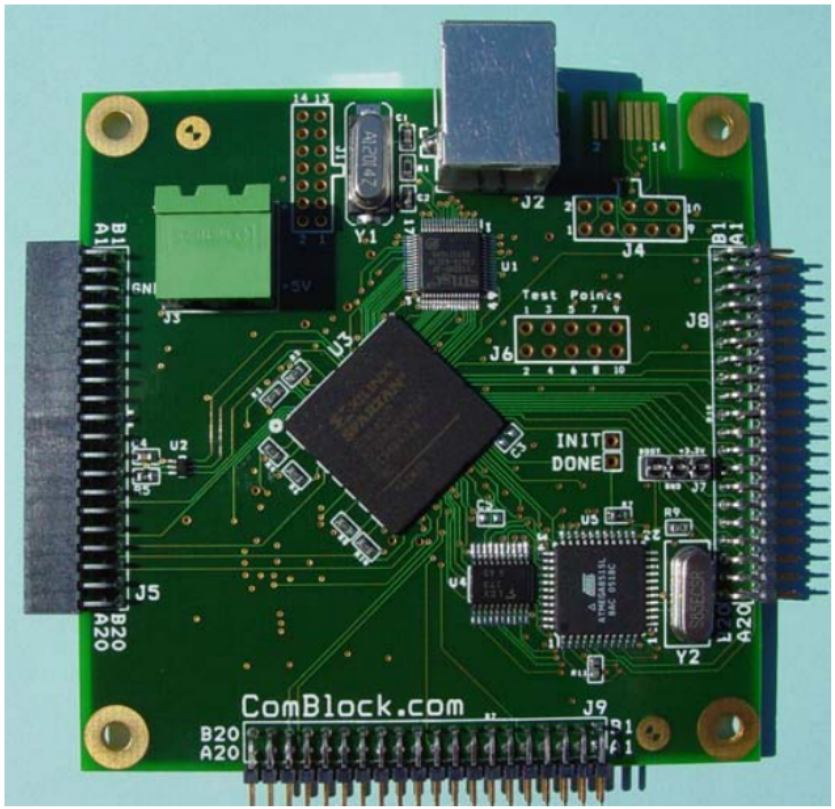
Problématique

- Classiquement : les débogueurs noyau utilisent un port série (fiable, mais lent)
- Ici : FPGA à 33 MHz, VIO : 3.3V (non compatible série)

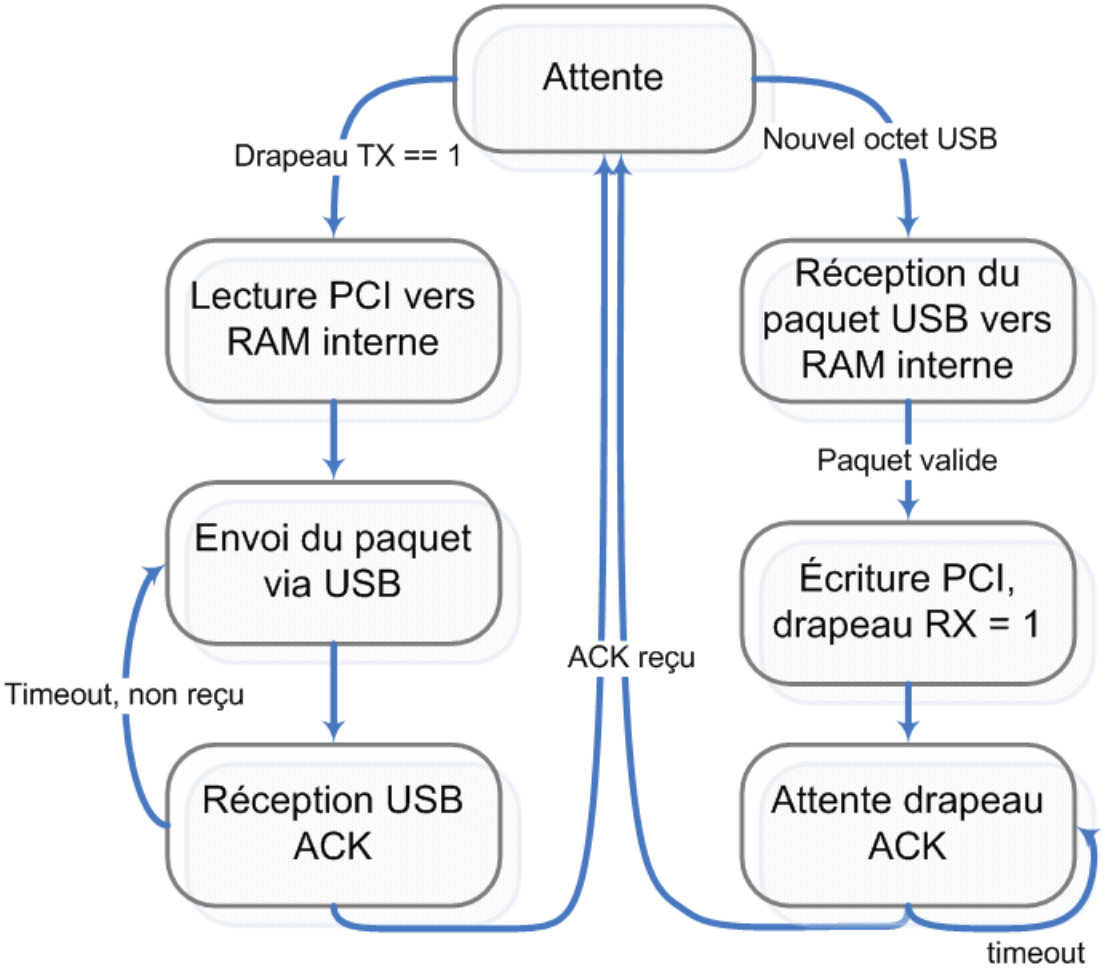
Pistes

- Port parallèle : plus disponible sur les portables récents
- Port série : puce MAX3232, convertit 3.3V ⇔ 12V
- FTDI : puce d'interfaçage USB 2.0 ⇔ RS232 en 3.3V
- COM-1400 (choix retenu) : USB 2.0 ⇔ protocole ad-hoc

COM-1400, vue de l'espace



Machine à état pour les communications USB⇔PCI



Recul sur la carte USB COM-1400

- Choix de cette carte pour son interfaçage avec la carte COM-1300
- Driver de ComBlock porté sous Windows XP seulement, buggé et non maintenu
- Débits limités à 300 Ko/s au lieu de 12 Mo/s théorique
- Au final, mauvais choix. Solution : FTDI ?

Plan

Méthode utilisée

- Reconstruction de l'espace d'adressage virtuel (cf. SSTIC 2008)
- Copie d'un premier payload (stager) chargé d'allouer suffisamment de mémoire pour pouvoir copier l'hyperviseur
- Ce payload écrit l'adresse physique du buffer alloué
- Le loader copie le code de l'hyperviseur comme le ferait Windows
  - Copie des sections
  - Résolution des imports
  - Application des relocations
  - Redirection de l'exécution vers le point d'entrée du driver
- Se reporter à l'article pour savoir quel pointeur écraser ;)

Plan

Hyperviseur

- Virtualise à "chaud" le système cible (à la BluePill)
- Implémenté sous la forme d'un driver compilé avec le WDK
- Virtualisation activée par l'utilisation des extensions VMX des processeurs Intel dernière génération (VT-x)

VMX (Virtual Machines eXtensions) en 1 slide (voir deux)

Terminologie

Host L'hyperviseur, appelé aussi VMM (Virtual Machine Monitor), a accès à toutes les ressources du système

Guest Machine virtuelle, soumise à l'arbitrage de l'hyperviseur

Deux modes de fonctionnement

VMX root l'hyperviseur s'exécute dans ce mode

VMX non-root le mode du système invité

Remarques

- Par conception, pas moyen de différencier ces modes
- Les transitions entre les deux modes sont appelées VM-Entry (VMM → Guest) et VM-Exit (Guest → VMM)
- Le fonctionnement du processeur en mode VMX non-root est modifié : certains évènements/instructions provoquent des transitions

VMX (Virtual Machines eXtensions) en 1 slide (voir deux)

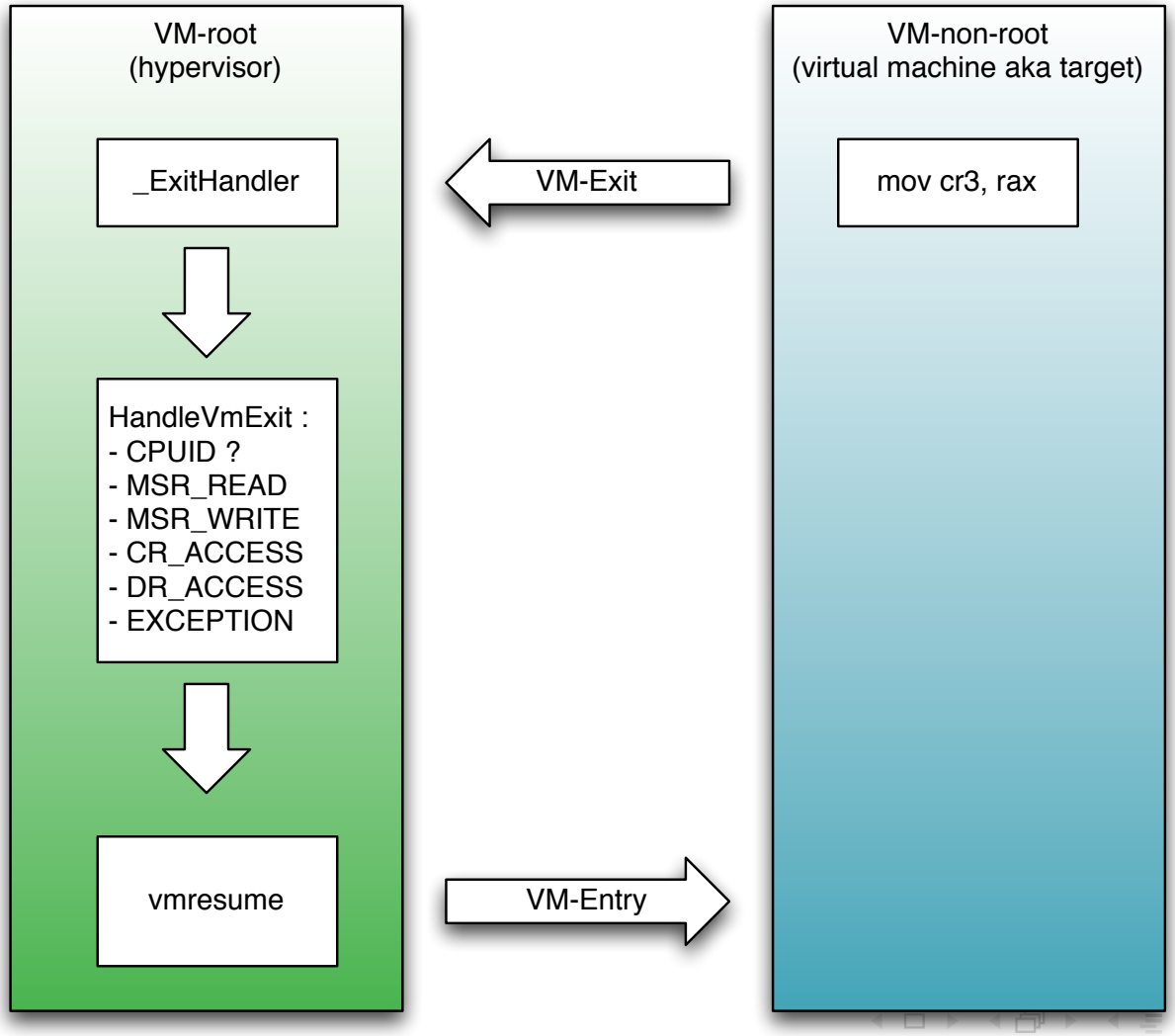
Cycle de vie

- Passage en mode VMX avec l'instruction `VMXON`
- Lancement des machines virtuelles avec l'instruction `VMLAUNCH`
- L'hyperviseur prend le contrôle à chaque VM-Exit
- L'hyperviseur redonne la main au système avec l'instruction `VMRESUME` qui provoque une VM-Entry
- L'hyperviseur peut s'arrêter en utilisant l'instruction `VMXOFF`

VMCS (Virtual Machine Control Structure)

- Contrôle les transitions et le fonctionnement en mode VMX non-root
- Manipulée par de nouvelles instructions (`VMPTRST`, `VMPTRLD`, `VMREAD`, `VMWRITE` et `VMCLEAR`)

Transitions



Plan

Déboguons dans la joie et la bonne humeur !

Modes de fonctionnement

- Mode **HALTED** : cible arrêtée, appelé aussi mode debug
- Mode **RUNNING** : cible en fonctionnement

Sens de communication

- Vers l'hyperviseur : manipulation de l'état de la cible
- Vers le client : indication d'un changement d'état de la cible (typiquement un point d'arrêt a été atteint)

Primitives de debug

- Arrêter / Reprendre l'exécution de la cible
- Inspecter les registres
- Inspecter la mémoire
- Singlestep

Primitives

Breakin et Continue

Des VM-Exit se produisent lors des changements de contexte : `mov cr3, rax` par exemple. L'hyperviseur regarde si le client a demandé l'arrêt ;

- Si oui : passage en mode debug
  - Arrêt des autres processeurs logiques
  - Passage en single step en modifiant le registre `RFLAGS` du guest
  - Après la VM-Entry, obtention d'une VM-Exit à cause de l'interruption
  - Attente des requêtes de manipulation
- Si non : modification du contexte et reprise de l'exécution

Lecture/écriture dans les registres

- Une partie du contexte est sauvegardé dans la VMCS
- Le reste est sauvegardé avant de rentrer en mode VMX root
- Et sera rétablit avant d'exécuter l'instruction `VMRESUME`

Primitives

Lecture/écriture mémoire

- Pour ne pas être interrompu, exécution à une irql `DPC_LEVEL`
- L'hyperviseur ne doit pas "crasher"
  - Lectures arbitraires $\Rightarrow$ faute de page $\Rightarrow$ écran bleu
- Parcours des PTE pour valider l'adresse demandée

Single Step

- Très simple étant donné l'accès au registre `RFLAGS`

Plan

Communications avec l'hyperviseur

Mémoire partagée

- Mémoire partagée entre l'hyperviseur et le FPGA sur CardBus
- Aucun mécanisme de synchronisation
- Deux zones de données : une pour l'émission et une pour la réception

Paquets de communication

- Formé d'un en-tête et de données
- Présence d'un `Id` incrémenté à chaque envoi/réception
- Somme de contrôle sur les données
- Type de paquets indiqués dans l'en-tête

Démo

DÉMO

Plan

Conclusion

Pourquoi VirtDbg ?

- Les débogueurs noyaux actuels utilisent des fonctions de l'OS qui sont du coup impossibles à déboguer (par exemple : gestionnaire d'interruption, fonctions réseau)
- Ils modifient des structures de l'OS et nécessitent sa coopération
- Parce que c'est fun :)

Pistes futures

- Publier une release !
- Porter l'hyperviseur en 32 bits, support des processeurs AMD
- Rajouter des évènements (interruptions, changement de contexte, entrées/sorties matérielles, etc.)
- Rajouter des primitives (modifications MSR, mémoire physique, etc.)
- Remplacer le COM-1400 par un FTDI

Questions ?

Pour nous contacter

Laboratoire **Sogeti-ESEC**
6-8 rue Duret
75016 Paris - France

damien.aumaitre@sogeti.com

christophe.devine@sogeti.com