Stratégies de défense et d'attaque : le cas des consoles de jeux

Ryad Benadjila et Mathieu Renard
prenom.nom@ssi.gouv.fr

ANSSI

Résumé Depuis plus de vingt ans, l'industrie du jeu vidéo investit du temps de recherche et développement pour lutter contre la production de contrefaçons et le piratage. L'importance de cet investissement est proportionnel au potentiel manque à gagner qui se compte en plusieurs dizaines de millions d'euros. Les consoles de jeux sont donc des vitrines technologiques sur le plan de la sécurité matérielle et logicielle. Le présent article propose un tour d'horizon de l'évolution de la sécurité de ces équipements, alliant un intéressant mélange de choix techniques innovants (tant pour la partie matérielle que logicielle) à de subtiles attaques permettant de les contourner. Il est d'ailleurs intéressant d'observer que les fabricants sont passés d'un modèle de menace où l'objectif était de se protéger des pirates de jeux à un modèle où les hobbyistes adeptes de plateformes maîtrisées sont devenus les principaux ennemis. L'analyse menée montre par ailleurs que les fabricants de consoles ont eu une avance certaine de plusieurs années sur les techniques de protection matérielle et logicielle aujourd'hui appliquées – ou pas – à d'autres produits grand public comme les smartphones et les Set Top Box.

1 Introduction

Ce document décrit l'utilisation d'outils et de techniques de Jailbreak dans un cadre purement défensif. L'objectif de cet article est de comprendre les raisons qui ont poussé les attaquants à privilégier certains chemins d'attaque, ainsi que les actions qui les auraient déjouées. Cet article ne doit pas être une incitation à l'exploitation de vulnérabilité, ou à Jailbreaker un équipement. Pour mémoire l'ANSSI incite les éditeurs à corriger systématiquement toute vulnérabilité identifiée dans les plus brefs délais. Les utilisateurs sont quant à eux invités à appliquer ces correctifs dès leur publication.

L'histoire des consoles de jeux vidéo débute avec la mise sur le marché de l'Odyssey mais c'est avec le succès de la console Atari Pong en 1975 qu'une branche d'activité spécifique voit le jour. Depuis la fin des années 80, ce marché est le théâtre d'une guerre économique que se livrent encore les trois grands constructeurs qui subsistent : Nintendo, Microsoft et Sony.

À cette guerre économique vient s'ajouter la guerre contre la contrefaçon de jeux qui débute réellement en 1994 avec la PlayStation produite par Sony Computer Entertainment. Même si le théâtre de la lutte contre la contrefaçon semble se distinguer du monde de la sécurité des systèmes, l'analyse de l'architecture des consoles modernes montre qu'un certain nombre de systèmes de sécurité mis en œuvre par les industriels du jeu vidéo auraient vocation à être utilisés dans le domaine de la sécurité (au sens strict).

Dans ce contexte, ce document commence par rappeler les enjeux et les menaces liés à l'industrie du jeu vidéo avant d'établir un lien entre les mondes de la protection de la propriété intellectuelle et de la sécurité. Un état des lieux de l'architecture et des fonctionnalités de sécurité des consoles de jeux vidéo modernes est ensuite déroulé avec pour objectif la mise en évidence de concepts et techniques pouvant être généralisés au monde de la sécurité des systèmes d'information, et aux systèmes embarqués en particulier.

1.1 Modèle d'attaquant, enjeux et menaces

Afin de comprendre la présence ou l'absence des mécanismes de sécurité intégrés dans une console de jeux vidéo, il est nécessaire de rappeler brièvement le modèle économique de ce secteur d'activité. Il y a vingt ans, les consoles de jeux étaient généralement vendues à perte pour faciliter leur déploiement dans les foyers ; de nos jours ce n'est plus le cas. Le modèle économique reste toutefois similaire : c'est la vente des jeux et des licences de développement associées qui compensent les éventuelles pertes et génèrent les bénéfices.

Il est donc essentiel pour les fabricants de garder sous contrôle les développements réalisés sur leurs plateformes, au risque de voir leurs consoles ouvertes à la contrefaçon. Cette problématique explose lorsque les cartouches de jeux (difficilement reproductible de par leur facteur de forme) font place au CD-ROM dans les années 90 : l'émergence et la vulgarisation des graveurs de CD-ROM devient une menace réelle pour l'industrie vidéoludique.

Au cours des années 2000 s'ajoute une nouvelle problématique : pour augmenter l'interactivité, les fabricants décident de connecter leurs consoles de jeux sur des plateformes accessibles depuis Internet (via le Xbox Live ou le PlayStation Network). C'est lors de cette ouverture des consoles au jeu en communauté que Microsoft et Sony prennent conscience des risques de compromission inhérents à cette interactivité nouvelle. Il en résulte deux consoles (Xbox 360 et PS3) dont l'architecture matérielle et logicielle

a clairement été pensée pour empêcher la moindre exécution de code non maîtrisé. Le paradigme des consoles de jeux vues comme des « DRM de luxe » empêchant simplement le piratage de jeux a dès lors changé : les consoles dites « nextgen » sont de parfaits exemples de **systèmes durcis** qui reposent sur des technologies que certaines industries ont mis plusieurs années à intégrer (voire qu'elles sont encore loin d'intégrer).

Les consoles d'ancienne génération faisaient plutôt face à un marché de la contrefaçon qui visait d'abord à pirater massivement des jeux et à en faire un marché noir parallèle. De manière intéressante, Microsoft et Sony se retrouvent désormais face à des attaquants qui se sont adaptés : des hobbyistes et autres *hackers* dont le niveau technique n'a d'égal que la créativité. L'objectif principal de ces passionnés est maintenant de faire tourner GNU Linux et des *homebrews* sur une plateforme fermée : la console est devenue un jouet qui offre un *défi* excitant.

1.2 Contribution

Par manque d'information publique sur les mécanismes de sécurité des nouvelles plateformes de Sony et Microsoft (PS4 et Xbox One), notre analyse se limite à certaines consoles ayant marqué l'histoire du jeu vidéo.

La documentation concernant les consoles de jeux et leur piratage est abondante mais est en général :

— Désorganisée : car constituée entre autres de contributions parcellaires et parfois contradictoires sur des forums de discussion.
— Incomplète : les attaquants publient souvent des exploits avec des explications sibyllines, sans forcément les relier aux protections contournées ni à l'architecture de sécurité de la console (ou alors de manière parcellaire).
— Complexe : car chaque nouvelle attaque présentée suppose une connaissance préalable des exploits précédents et du fonctionnement des mécanismes de sécurité de la console.

Une des contributions du présent article est d'essayer de compiler pour les quelques consoles sélectionnées une vue d'ensemble détaillée et complète de l'architecture, de leur sécurité et des attaques. Tous les éléments présentés sont **publics**. Notons que parmi ces documents on trouve une référence académique concernant la Xbox 360 et la PS3 [43] qui donne une bonne vision d'ensemble des attaques sur ces deux consoles. Toutefois, ce document souffre de quelques lacunes :

— L'énumération et l'historique des attaques, bien que très complètes, ne sont pas vraiment reliées au modèle de menace et à l'architecture

de sécurité des consoles (ces deux derniers éléments sont d'ailleurs introduits trop succinctement, avec peu d'explications).
— La description de l'exploitation de certaines attaques (comme la *King Kong attack* sur Xbox 360 ou la *glitch attack* de la PS3) n'est pas assez détaillée. Or il nous semble que le niveau de détail que nous tentons d'amener dans le présent article bénéficie à la compréhension des erreurs commises par le fabricant (et donc à l'analyse globale de la sécurité des consoles de jeux avec quelques années de recul, et au regard d'autres systèmes).
— Il se cantonne à une analyse purement objective et historique de la sécurité des deux consoles et n'apporte aucune analyse des fonctionnalités de sécurité.

Dans les sections suivantes, nous passons ainsi au crible l'architecture, les techniques de défense et d'attaque sur quatre consoles qui incarnent une évolution majeure dans le monde vidéoludique en dix ans.

— La PlayStation : elle a été la première à faire face au piratage de masse et accessible à tous. Nous verrons que cela s'explique par le peu d'investissement en termes de sécurité ;
— La Xbox : c'est la première console de Microsoft, et à ce titre elle a essuyé quelques plâtres d'inexpérience de la compagnie dans ce domaine. Malgré cela, nous verrons que cette console contenait les prémices (encore immatures) de la sécurité des consoles modernes.
— La Xbox 360 : seulement 10 ans après la PlayStation, nous verrons comment les ingénieurs de Microsoft ont réussi à créer un système durci en appliquant, avec plusieurs années d'avance, des techniques défensives logicielles et matérielles. Nous verrons aussi l'ingéniosité qu'il a fallu aux attaquants pour compromettre ce système.
— La PlayStation 3 : elle est à plusieurs titres remarquable. Du point de vue de la conception, d'une part, car Sony a fait le pari d'une plateforme exotique innovante (le processeur Cell d'IBM) qui apporte des éléments de sécurité intéressants. Du point de vue de l'attaque d'autre part, car malgré de bonnes bases la console a souffert de petit oublis lourds de conséquences pour sa sécurité.

2 Analyse

2.1 Du DRM à la sécurité : histoire de l'échec de la PlaySation

Produite par Sony Computer Entertainment à partir de 1994, la PlayStation originale fut la première machine à avoir subi la contrefaçon

de masse. L'installation d'un *modchip* permettait aux utilisateurs de prendre le contrôle de la plateforme matérielle et d'exécuter du code personnel ou des contrefaçons de jeux gravés sur des CD-ROM vierges. La production en masse de ces *modchips* a véritablement marqué le début de la contrefaçon des jeux vidéo.

Voyage dans le passé : sécurité de la PlayStation

Comme en témoigne la décision d'embarquer un processeur `MIPS R3000` *custom* dépourvu d'unité de gestion mémoire (MMU) [47,51] (alors que les RS3000E classiques en sont pourvus), la sécurité semble être bien loin des préoccupations des ingénieurs de Sony de l'époque. Ceux-ci, semblent en effet avoir privilégié les fonctionnalités de zonage des jeux et d'anti-copie comme le **wobble-groove**. Ce sont ces deux éléments qui ont donné naissance à un tout nouveau type d'attaque : les **modchips**.

PlayStation *Zonage des jeux et des consoles (disc region)*
Lors de la conception de la PlayStation, Sony a développé un système de zonage des jeux [40] et des consoles. Ainsi, il n'est pas possible de lancer un jeu acheté en Asie ou aux USA sur une console européenne. Ce système s'appuie sur un code régional enregistré à la fois dans le `BIOS` des consoles et sur la piste d'amorçage des CD-ROM légitimes.

Ces codes régionaux se présentent sous la forme d'une chaîne de caractères ayant la forme : `SCEx`, où `SCE` correspond aux initiales de Sony Computer Entertainment et `x` correspond à la région du disque :
- `A` pour l'Amérique (`SCEA`)
- `E` pour l'Europe (`SCEE`)
- `I` pour le Japon (`SCEI`)
- `W` pour Net Yarozee (`SCEW`)

Cette information est vérifiée par le système lors du démarrage du jeu. Lorsque l'information obtenue à la lecture du CD-ROM est absente ou erronée, le CD-ROM inséré dans la console est rejeté.

PlayStation *Tatouage numérique : wobble-groove*
Cette technologie est développée par Royal Philips Electronics NV. Elle permet en complément d'un tatouage numérique (*watermarking*) de s'assurer que seuls les disques authentiques peuvent être joués. En effet, la protection vise à remodeler la rainure gravée sur le disque en faisant varier son orientation (voir Figure 1). La piste du *Lead-IN* n'est plus rectiligne mais présente des ondulations. À la lecture, ces ondulations créent une

oscillation de la lumière du laser réfléchie sur le disque. Cette oscillation est trop rapide pour pouvoir être suivie et enregistrée par le capteur de la tête de lecture. Cependant, le capteur d'erreur utilisé pour le maintient en phase de l'objectif de la tête de lecture sur le mouvement de la piste est capable d'acquérir les signaux produits par le *wobble-groove*. Seuls les lecteurs CD-ROM équipés d'un module capable de filtrer et traiter le signal provenant du capteur d'erreur peuvent donc interpréter les données stockées avec cette protection.

En raison des technologies sophistiquées nécessaires à la fabrication de lecteurs capable de lire ou reproduire les informations stockées avec le *wobble-groove*, le piratage des disques authentiques est presque impossible. Les attaquants ont donc cherché d'autres méthodes permettant d'exécuter du code sur la console.

Playstation 1 : les attaques

En l'absence de fonctions de sécurité, les attaques réalisées sur la PlayStation avaient pour principal objectif de contourner les fonctions de zonage de la console. L'historique de ces attaques est présenté dans la présente section, et sa vue d'ensemble sur la figure 2.

PlayStation *Histoire courte : Action replay* 1995

Une technique employée par les attaquants pour exécuter du code sur la PlayStation a consisté à connecter un module sur le port d'extension (*parallel I/O*, cf. figure 3). Ce port, présent sur les consoles avec une version antérieure à `SCPH-900x`, est connecté sur le même BUS que la *bootROM* et dispose d'un accès au signal `/OE` (Output Enable) de la *bootROM*. Un module est donc en mesure de se faire passer pour une *bootROM* et servir un code de démarrage personnalisé (aucun mécanisme de signature de code ou de chaîne de démarrage de confiance n'est intégré à la PlayStation).

Cette technique a été popularisée par la production et la commercialisation du module *Action Replay* dont l'objectif principal était de fournir un moyen aux utilisateurs de tricher. Le code de l'*Action Replay* était exécuté à la place du code de la *bootROM* et présentait une interface de saisie de code en hexadécimal permettant à l'utilisateur de saisir le code de l'astuce lui permettant de gagner plus facilement. Le code saisi par l'utilisateur correspondait à une séquence d'instruction sous forme d'*op-codes*, utilisée par l'*Action Replay* pour modifier le jeu en mémoire lors du démarrage. L'*Action Replay* à également été utilisé pour lancer des contrefaçons. La réaction de Sony a ces actions a été de supprimer le

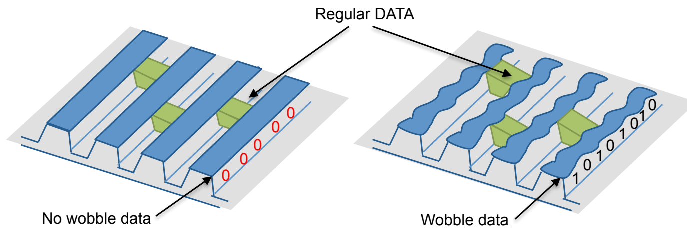

Figure 1. Playsation 1 : concept du wobble-groove

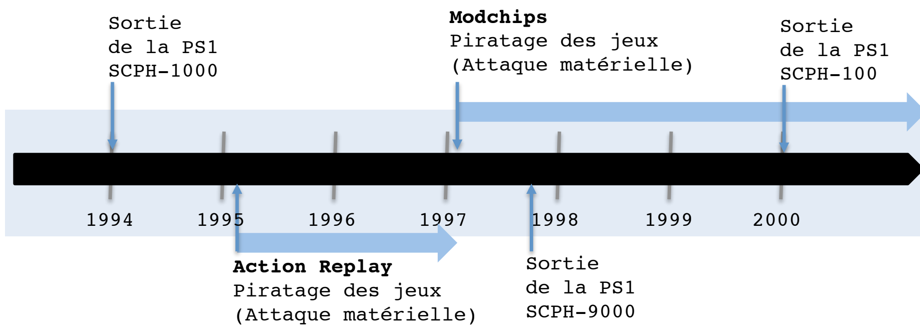

Figure 2. PlayStation : chronologie des attaques

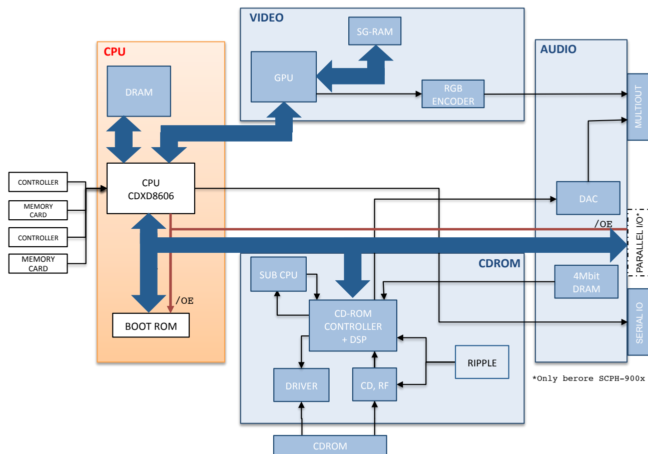

Figure 3. Playsation 1 : architecture de la PlayStation

port *parallel I/O* sur les nouveaux modèles et d'ajouter des mesures de détection aux nouveaux jeux, rendant ainsi cette technique obsolète.

PlayStation *Naissance des modchips* 1997
L'information de région n'étant pas reproductible à partir d'un graveur de CD-ROM/DVD-ROM du commerce, les attaquants ont dû trouver d'autres méthodes pour lancer leurs copies de jeux. C'est ainsi qu'un groupe décide de leurrer le système en reproduisant l'information à l'aide d'un micro-contrôleur connecté entre la sortie du capteur et le composant en charge du traitement de l'information, afin de permettre le lancement des contrefaçons [44].

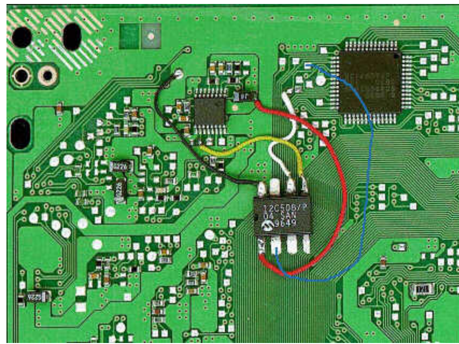

Figure 4. Playsation 1 : *modchip* installé sur une carte mère de Playstation version PU-18

Dans l'impossibilité d'ajouter des fonctionnalités de sécurité à sa plateforme (le BIOS et le noyau sont stockés en ROM), Sony a tenté d'identifier les consoles modifiées en intégrant dans le code des jeux des systèmes de détection. Toutes ces tentatives ont été déjouées par les attaquants qui, grâce aux *modchips*, ont pris le contrôle du matériel.

Playstation 1 : conclusions sur la sécurité
Il y a 20 ans, avec la PlayStation, Sony semble avoir refusé de payer le prix de la sécurité, faisant ainsi exploser l'industrie du piratage. Fort de cette expérience, les fabricants de consoles de jeux vidéo ont augmenté le budget de développement alloué à la sécurité. C'est dans ce contexte que les architectures des consoles modernes ont vu le jour. Les chapitres suivants proposent donc une analyse détaillée des succès et des échecs des mécanismes de sécurité (au niveau matériel et logiciel) de ces architectures.

2.2 Les prémices de la sécurité : la Xbox

Microsoft fait ses premiers pas dans le monde des consoles de jeux avec Nvidia. Ils développent la Xbox, lancée aux États-Unis le 15 novembre 2001. À cette époque les principales concurrentes sont la PlayStation 2 de Sony et la Gamecube de Nintendo. Pour se différencier et devenir leader du marché, Microsoft fait évoluer la Xbox avec la plateforme Xbox Live. La Xbox devient donc une station multimédia permettant le jeu multi-joueurs distants. C'est sous cette forme que la Xbox est commercialisée dans le reste du monde. Le support de la console par Microsoft s'est arrêté en mars 2009 aux États-Unis et en avril 2010 en France.

La Xbox est l'une des premières plateformes à intégrer dès sa sortie plusieurs fonctionnalités de sécurité qui seront détaillées tout au long de ce chapitre. Ces mécanismes ayant été mises en défaut par différentes équipes d'attaquants nous présenterons une analyse des différentes faiblesses et des techniques mise en œuvre pour prendre le contrôle du système. Mais avant de plonger dans le vif du sujet il est nécessaire de faire un point sur l'architecture de cette console.

Xbox : architecture générale

Xbox **Architecture Matérielle**

L'architecture de la Xbox est proche de celle d'un PC. En effet, la carte mère a été conçue par NVIDIA autour de son *chipset* NForce. Ce composant, destiné à l'origine pour l'architecture PC, a été modifié afin de répondre au besoin de la Xbox. Bien que la Xbox ait été conçue à l'origine pour embarquer un processeur AMD, c'est un processeur Intel Pentium III modifié, cadencé à 733 MHz qui est embarqué sur la carte mère lors de la sortie commerciale. Ce CPU hybride, entre la gamme Pentium III et la gamme Celeron est au centre d'une architecture mémoire unifiée ou *Unified Memory Architecture (UMA)* : le CPU est connecté au *Graphics Processing Unit (GPU)* via un *Front Side Bus (FSB)* de 64 bits cadencé à 133 MHz. Le GPU est intégré à la puce `NV2A` de NVidia qui intègre également le *Northbridge*. Cette architecture permet au CPU et au GPU de partager l'accès à la mémoire. Il n'y a pas de puce mémoire vidéo dédiée. Cependant la RAM (64 Mo) est divisée en plusieurs banques qui peuvent être consultées indépendamment à la fois par le GPU et le CPU.

Le *Northbridge* et les autres périphériques sont reliés à la puce `MCPx` de NVidia qui joue le rôle de *Southbridge*. L'interconnexion est faite au travers d'un bus *HyperTransport* 2x8 bits, cadencé à 200 MHz.

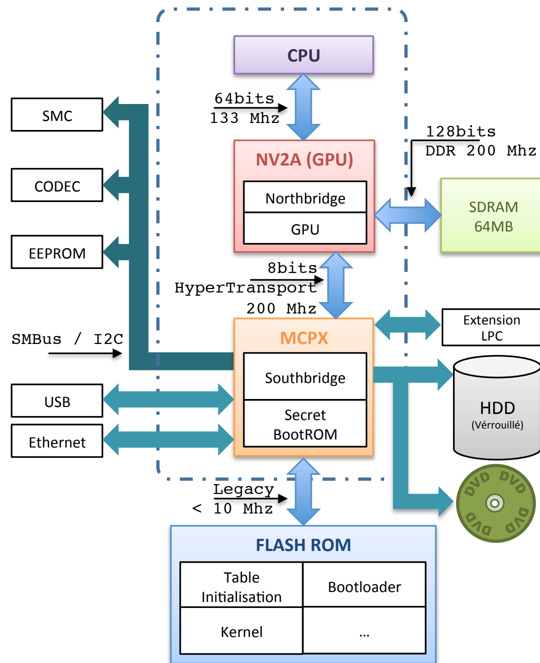

Figure 5. Xbox : architecture matérielle de la Xbox

Le `MCPx` combine la puce *Southbridge* ainsi que presque tous les périphériques Xbox :

— les contrôleurs USB ;
— la *bootROM* ;
— le processeur audio numérique Dolby ;
— le contrôleur Ethernet ;
— le contrôleur IDE ;
— ...

La Xbox est la première console à intégrer un disque dur. Cette zone de stockage permet aux joueurs d'enregistrer leurs sauvegardes de jeux ainsi que le contenu multimédia téléchargé depuis la plateforme *Xbox live*. Les jeux emploient également une partie de l'espace comme zone de cache, accélérant ainsi le temps de chargement. Le disque dur est un disque IDE standard formaté avec une variation de FAT16, appelée FAT-X [12]

La Xbox intègre également un lecteur DVD pour que les joueurs puissent exécuter leurs jeux. Les jeux Microsoft Xbox sont distribués au format DVD9 (DVD Double couche) et les données sont organisées sur le support en suivant un format **propriétaire**, le format XDVD [38] Enfin, la connexion à la plateforme *Xbox Live* se fait au travers du port RJ45 présent sur la carte mère grâce à une connexion TCP/IP standard.

Avec la Xbox Microsoft fournit donc une plateforme PC/Multimédia à moindre coût (La Xbox est vendue 199$ quelques mois après sa sortie).

Xbox Architecture logicielle

L'architecture logicielle repose sur un noyau Windows 2000 simplifié. Ce noyau est responsable de l'exécution de l'écran d'accueil (*Dashboard*) et des différents jeux ou applications Microsoft. Tous les jeux et applications Xbox fonctionnent en mode noyau. Les raisons de ce choix restent obscures. Nous pouvons imaginer un possible ralentissement des jeux lorsqu'ils sont exécutés en espace utilisateur, ou encore par facilité de développement (les développeurs des jeux contrôlent l'intégralité du système). En tout cas ce choix fait peser sur l'architecture un risque de compromission élevé.

Xbox : Fonctionnalités de sécurité Afin de garder la mainmise sur sa console de jeux Microsoft a intégré plusieurs fonctionnalités de sécurité présentées dans les paragraphes suivants.

Xbox Signature de code

Les exécutables Xbox ont un format proche du format `PE` de Microsoft appelé `XBE` [37]. Ces binaires sont signés avec une clé privée appartenant à

Microsoft. Il est donc *a priori* impossible d'exécuter un binaire non signé sur la Xbox.

Xbox Contrôle d'accès au contenu du disque dur

Afin d'interdire la modification des données stockées sur le disque à froid, Microsoft utilise les commandes définies par la norme `ATA Security` [5] pour verrouiller le disque et interdire l'accès au contenu depuis un équipement autre que la Xbox [39]. Lorsque le disque est verrouillé toutes les demandes de lecture et d'écriture sont rejetées. En effet lorsque la console est hors tension, le disque est verrouillé par un mot de passe de 32 octets unique pour chaque Xbox. Le mot de passe est généré en deux phases distinctes [11]. Une clé unique (dénommée HDKey) est extraite de l'EEPROM. Cette clé est ensuite utilisée pour générer le mot de passe de déverrouillage du disque (HMAC-SHA-1 de la HDKey, du numéro de modèle du disque ainsi que de son numéro de série). Ainsi lors du démarrage, le *bootloader* lit les données de l'EEPROM, calcule le mot de passe et transmet le résultat au disque dur à l'aide d'une commande ATA Security. Le disque dur reste déverrouillé jusqu'à l'arrêt du système.

Xbox Chaîne de démarrage de confiance

Afin de garder le contrôle du code exécuté sur sa console de jeux, Microsoft a intégré une chaîne de démarrage de confiance à la Xbox. Ce paragraphe en décrit le fonctionnement.

Initialisation

L'analyse de la phase de démarrage de la Xbox conduite par la communauté Xbox Linux montre qu'avant même le démarrage de la console, le MCPx va lire des données dans la mémoire flash externe. Il semble donc qu'une phase de pré-initialisation se déroule avant le démarrage du système. Les étapes de cette phase ainsi que la raison de cette opération ne sont toujours pas claires. Nous pouvons alors supposer que le MCPx initialise ses différentes interfaces et tente de valider la présence de la mémoire flash externe avant le démarrage de la console.

Démarrage

Lors de la mise sous tension, la première instruction récupérée et exécutée est située à l'adresse physique `0xFFFF:FFF0`. Cette adresse est située 16 octets en dessous de l'adresse physique la plus élevée du processeur. Sur un PC, le BIOS qui se trouve dans une mémoire flash externe est mappé à cette adresse.

Ainsi, à la mise sous tension d'un PC, le code du BIOS est exécuté. Ce code spécifique à chaque carte mère est généralement stocké en clair, sur un composant externe de type flash reprogrammable (ou EEPROM).

Il est donc possible de remplacer le composant, d'en extraire le contenu ou encore de le reprogrammer.

Comme sur une architecture PC, les versions de développement de la Xbox sont conçues pour démarrer à partir d'une mémoire flash externe connectée sur le bus standard ou sur le bus *Low Pin Count* (LPC). Par conséquent, il est impossible de garantir ni l'authenticité, ni l'intégrité et encore moins la confidentialité du code exécuté par le processeur.

BootROM et racine de confiance
Microsoft souhaitant garder le contrôle du code exécuté sur sa console de jeux, a intégré une racine de confiance. À l'époque, les technologies de type TPM tel que Palladium ne sont encore que des concepts en cours de définition. C'est pourquoi, pour la Xbox, Microsoft a tenté de créer sa propre chaîne de confiance [36,34]. La figure 6 en présente le concept.

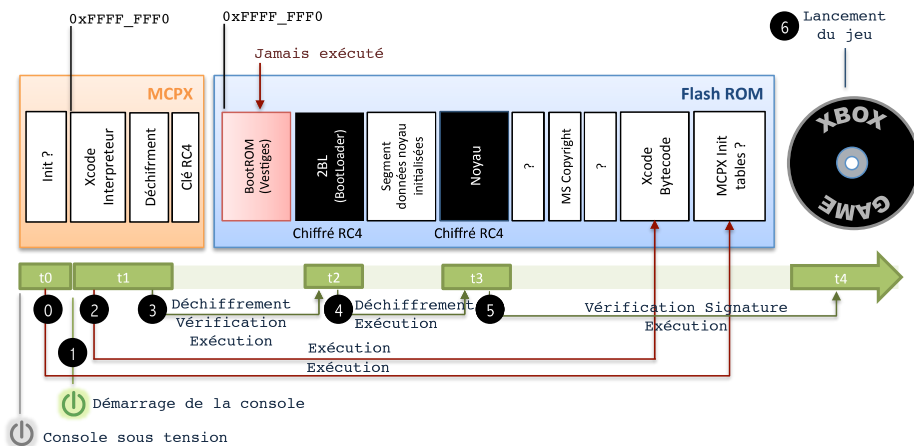

Figure 6. Xbox : séquence de démarrage et chaîne de confiance d'une Xbox

Microsoft a donc demandé à NVidia de graver le code du *bootloader* dans le silicium de son composant MCPx spécialement conçu pour la Xbox afin de jouer le rôle de *Southbridge*. L'intégration d'une zone mémoire dans un composant de type ASIC [4] ayant un coût important et dépendant de l'espace disponible, Microsoft et Nvidia ont donc limité la taille de la zone de stockage de la *bootROM* à 512 octets.

bootROM et Initialisation
La limitation de la taille (512 octets) de la *bootROM* va jouer un rôle

déterminant dans l'architecture de la chaîne de démarrage. Le code de la *bootROM* est responsable de l'initialisation des différents périphériques du système. Il inclut le code de *training* de la mémoire RAM, et dans le cas de la Xbox cela pose un problème d'espace de stockage. En effet, le code de *training* nécessaire à l'initialisation des modules de mémoire RAM de la Xbox requiert plus de 1 Ko. Il est donc nécessaire d'avoir recours à un espace de stockage externe.

Or, l'utilisation d'une mémoire flash externe augmente le risque de voir son contenu extrait ou modifié par un attaquant. C'est pourquoi les concepteurs de la plateforme ont décidé d'en chiffrer le contenu à l'aide de l'algorithme de chiffrement par flot `RC4` et d'une clé de 128 bits enfouie dans le silicium du `MCPx`. Ainsi, sans connaissance de la clé il est impossible de déchiffrer et d'interpréter le code extrait de la flash externe.

Ce choix technique introduit également problème dans le séquencement des opérations de démarrage : le code stocké dans le `MCPx` doit déchiffrer le contenu de la mémoire flash externe pour initialiser la RAM. Or le déchiffrement requiert l'utilisation de la RAM.

Les équipes de Microsoft ont donc intégré un interpréteur minimaliste stocké dans la *bootROM* du `MCPx`. Cet interpréteur, qui porte le nom de *Xcode*, réalise les fonctions d'initialisation requises pour placer le système dans un état stable (*training* mémoire par exemple) et permettre le déchiffrement du *bootloader* de second niveau. Pour mener à bien ces opérations *Xcode* exécute du *bytecode* qui est stocké en clair dans la mémoire flash externe (128 octets).

Interpréteur Xcode

Xcode est un interpréteur minimal qui supporte douze instructions. Le listing 1 présente le pseudo code de l'interpréteur Xcode. Un mnémonique *Xcode* se compose d'une déclaration de 8 bits, toujours suivie par deux opérandes 32 bits.

```
struct {
    char opcode;
    int op1;
    int op2;
} *p;

int acc;
p = 0xFFF00080;
while(1) {
    switch(p->opcode) {
    case 2:
        acc = *((int*)p->op1);
        break;
    case 3:
        *((int*)p->op1) = p->op2;
```

```
            break;
        case 4:
            outl(p->op1, 0x0CF8);
            outl(p->op2, 0x0CFC);
            break;
        case 5:
            ...
        case 0xEE:
            goto end;
        }
        p++;
}
end:
```

Listing 1. Xbox - Interpreteur Xcode en C

Le tableau 1 résume les instructions supportées par la machine virtuelle, tous les autres codes sont considérés comme des `NOP` (opérations neutres).

Opcode	Nom	Opération
0x02	PEEK	ACC := MEM[OP1]
0x03	POKE	MEM[OP1] := OP2
0x04	POKEPCI	PCICONF[OP1] := OP2
0x05	PEEKPCI	ACC := PCICONF[OP1]
0x06	AND/OR	ACC := (ACC & OP1) | OP2
0x07	*prefix*	Exécute l'intruction stockée dans OP1 avec OP1 := OP2, OP2 := ACC
0x08	BNE	IF ACC = OP1 THEN PC := PC + OP2
0x09	BRA	PC := PC + OP2
0x10	AND/OR	ACC2 (non utilisé) ACC2 := (ACC2 & OP1) | OP2
0x11	OUTB	PORT[OP1] := OP2
0x12	INB	ACC := PORT(OP1)
0xEE	END	

Table 1. Xcode bytecode

Les instructions exécutées par *Xcode* sont stockées dans la mémoire flash externe. Aucun contrôle d'intégrité n'est réalisé sur cette partie de la flash. Le *bytecode* peut donc être modifié par un attaquant, les ingénieurs de Microsoft ont donc mis en place plusieurs restrictions. Par exemple, à l'aide de l'instruction `PEEK` il est possible d'accéder à la mémoire et de lire l'état des entrées/sorties. Ainsi, un attaquant disposant d'un accès physique à la mémoire flash pourrait altérer le *bytecode* et exposer le code de la mémoire secrète dans la mémoire ou sur un bus non chiffré simple à espionner comme le LPC ou le bus I^2C. L'interpréteur `Xcode` s'assure donc que les instructions ne peuvent pas lire la ROM secrète qui est mappée sur les 512 derniers octets de l'espace d'adressage. Cette opération est réalisée

en forçant la mise à zéro des 4 bits de poids fort de l'adresse passée en paramètre à l'instruction PEEK.

```
    and ebx, 0FFFFFFFh   ; Masquage des 4 bits de poids fort
    mov edi, [ebx]       ; Lecture de la memoire a l'adresse fournis
        par OP1
    jmp next_instruction
```

Listing 2. Xbox - code exécuté par l'instruction PEEK

Déchiffrement du bootloader de second niveau (2BL)
Une fois la mémoire initialisée et le système placé dans un état stable, le code de la *bootROM* est en mesure de déchiffrer en RAM, le *bootloader* de second niveau (2BL) et le noyau stockés sur la mémoire flash sous forme chiffrée. Le chiffrement est réalisé à l'aide le l'algorithme de chiffrement par flot RC4 et une clé de 128bits stockée sur le silicium de la puce du MCPx. Le code de la fonction de déchiffrement est développé de sorte que la clé de déchiffrement n'est jamais écrite dans la mémoire principale, déjouant ainsi toute tentative d'espionnage du bus mémoire. Toutefois, le bus *HyperTransport* situé entre le *Southbridge* et le *Northbridge* en direction du processeur n'est pas chiffré, car supposé protégé par sa vitesse de transmission élevée : l'horloge du bus est cadencée à 200 MHz mais les données sont transmises sur les deux fronts, ce qui permet d'obtenir une vitesse de 400 Mb/s. Pour mener à bien l'acquisition il faut disposer d'un équipement de mesure qui avait à l'époque un coût assez important. Microsoft a donc délibérément choisi de ne pas chiffrer ce bus.

Vérification du 2BL
L'intégrité du code déchiffré du 2BL est vérifié par le code de la *bootROM*. Encore une fois Microsoft va se heurter à une problématique d'espace : le code de déchiffrement RC4 représente 150 octets, la clé de déchiffrement 16 octets, l'interpréteur *Xcode* 175 octets et l'initialisation du processeur 145 octets. Il reste donc uniquement 30 octets pour vérifier l'intégrité du code déchiffré. Les ingénieurs de Microsoft ont alors implémenté une fonction de vérification d'intégrité minimaliste.

Comme présenté dans le listing 3, cette fonction se contente de vérifier une constante de 32 bits située à la fin des données déchiffrées en RAM à l'adresse 0x0009_5FE4. Lorsque la valeur à cette adresse est celle attendue (0x7854794A) le code de la *bootROM* saute à l'adresse 0x0009_0000, l'emplacement où le *bootloader* de second niveau déchiffré a été placé.

Désactivation de l'accès à la bootROM
Pour masquer l'utilisation de la zone mémoire secrète, Microsoft utilise le concept de superposition mémoire (*memory overlay*, voir figure 8. Ainsi, au

démarrage, la zone mémoire secrète est mappée à l'adresse 0xFFFF:FFF0 (voir figure 7). Lorsque le 2BL est intègre, c'est à lui de désactiver l'accès à la mémoire secrète avant d'exécuter toute autre instruction.

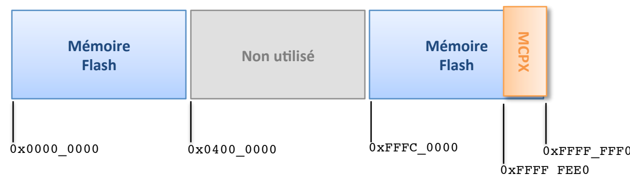

Figure 7. Xbox : *mapping* mémoire au démarrage de la Xbox

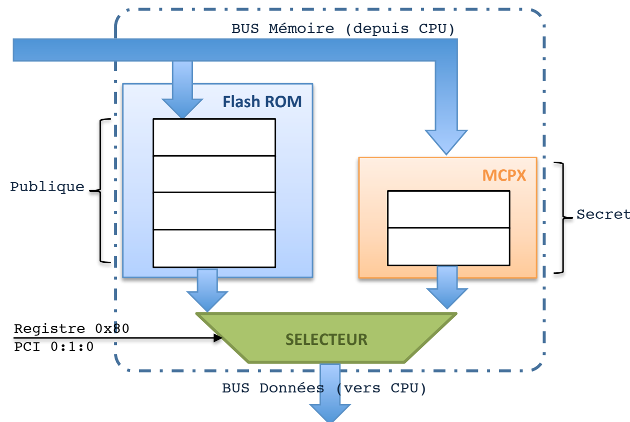

Figure 8. Xbox : concept de superposition mémoire utilisé par la Xbox

Dans le cas où le code du 2BL n'est pas intègre, l'accès à la mémoire secrète est désactivée et l'exécution est stoppée. Comme exposé précédemment, désactiver l'accès à la mémoire secrète n'est pas si simple. Lorsque le processeur est arrêté, il n'est plus possible de désactiver l'accès à la *BootROM* ; et lorsque l'accès a la *BootROM* est désactivé il n'est plus possible d'agir sur le flot d'exécution et de stopper le processeur. En effet, à cause de l'*overlay* mémoire, lors de la désactivation de l'accès à la mémoire secrète, le processeur bascule sur la zone mémoire externe et continue l'exécution du code.

Les équipes de Microsoft ont alors eu recours à une astuce *a priori* ingénieuse pour désactiver l'accès à l'espace mémoire secret et arrêter

proprement le système. Cette astuce consiste à conduire le flot d'exécution en haut de l'espace d'adressage et désactiver l'accès à la zone secrète à la toute dernière adresse de l'espace mémoire `0xFFFF:FFFA`. Ainsi, après l'exécution de cette instruction le compteur programme déborde à l'adresse `0x0000:0000` et provoque une exception. Comme aucun gestionnaire d'interruption n'est défini pour gérer ce débordement, le processeur est censé générer une double faute et s'arrêter. Ainsi il n'est pas possible d'avoir accès au contenu de la mémoire secrète. Le code de la fonction en charge de l'extinction de la Xbox en cas de corruption de la flash est présenté dans le listing 3.

```
mov eax, ds:95FE4h
cmp eax, 7854794Ah        ; Magic : dechiffrement OK
jnz short bad_checkcode
mov eax, ds:90000h
jmp eax
                        ; Saut vers le 2BL
                        ; Preparation pour la desactivation de la
                            bootROM
bad_checkcode:
   mov eax, 80000880h
   mov dx, 0CF8h
   out dx, eax
jmp far ptr 8:0FFFFFFFAh    ; Saut a la fin de la bootROM,

[..]

FFFA:                       ;  Adresse 0FFFFFFFAh (Fin de la bootROM
    )
   add dl, 4
   mov al, 2
   out dx, al
                            ; Adresse 0x00000000
```

Listing 3. Xbox : désactivation de la mémoire secrète en cas de corruption de la flash

SVC Watchdog

Afin de détecter tout comportement anormal lors de l'exécution de la *bootROM* Microsoft a ajouté un *watchdog* sur le circuit de la carte mère. Ce *watchdog* est mis en œuvre à l'aide d'un micro-contrôleur de type PIC connecté sur le SMBus et qui embarque des fonctions basiques telles que le contrôle des LED de la console. Le *watchdog* est implémenté sous la forme d'un *challenge* de type question/réponse dont les échanges sont réalisés sur le SMBus lors du démarrage. Le PIC génère quatre octets aléatoires, lues à partir de registres `0x1C` à `0x1F`. Pour désactiver le compte à rebours, le `2BL` hache ces quatre octets et écrit le résultat sur le SMBus aux adresses `0x20` et `0x21`. Le fonctionnement détaillé de ce *watchdog* est présenté sur

le site de la communauté Xbox Linux [18]. Ainsi, lorsque la *bootROM* met plus de 200 millisecondes pour démarrer le PIC, cela provoque le *reset* du processeur. Le `2BL` est chargé de répondre au challenge pour désactiver le *watchdog*.

Déchiffrement du noyau

Le noyau est également chiffré par l'algorithme de chiffrement `RC4`. Le déchiffrement se fait via le `2BL` qui emploie une clé de chiffrement elle aussi stockée dans la mémoire flash externe. La sécurité de cette implémentation repose donc sur la clé de chiffrement utilisée pour déchiffrer le `2BL`, et sur l'incapacité à extraire le code stocké dans la zone mémoire sécurisée (*BootROM*).

Synthèse

En résumé, le code de la *bootROM* enfoui dans le MCPx initialise le CPU, interprète les *Xcodes* sauvegardées en mémoire flash externe pour initialiser les différents périphériques dont la mémoire RAM ; puis déchiffre en RAM et exécute le code `x86` du second *bootloader*. Le chiffrement repose sur l'algorithme de chiffrement par flot `RC4` ainsi qu'une clé de 128 bits stockée sur le silicium de la puce du MCPx. Le `2BL` continue donc l'initialisation du matériel, charge le noyau, le déchiffre avec une clé RC4 embarquée (chiffrée) dans la flash et l'exécute. La gestion de l'intégrité du système en fonctionnement est déléguée au noyau qui est responsable de la vérification de la signature des jeux.

On déplore l'absence de contrôle d'intégrité sur le *bytecode* et sur le noyau ainsi que la quasi-inexistance du contrôle d'intégrité du `2BL`. À notre sens, ces choix techniques peuvent s'expliquer par le maintient électrique permanent de du signal `RW#` (lecture seule) sur la mémoire flash. Le passage en lecture et écriture requiert une intervention sur la carte mère. Il n'est *a priori* pas possible de modifier le noyau. D'autant que pour obtenir un noyau valide, il est nécessaire d'avoir connaissance de la clé de chiffrement RC4.

Xbox Réduction des risques d'exploitation

Aucune fonctionnalité de réduction des risques d'exploitation n'est intégrée au noyau. À l'époque le support du bit `XD` (*eXecute Disable*) sur les processeurs Intel est inexistant. Cette fonctionnalité n'est apparue sur les processeurs Intel qu'à partir de 2004. Toute la sécurité logicielle de la Xbox repose donc sur la signature des jeux et la chaîne de confiance.

Xbox Maintien en condition de sécurité

Le système est stocké sur une mémoire externe. Cette mémoire flash est électriquement forcée en lecture seule. Il n'est donc pas possible de

mettre à jour le système au travers de la connexion à la plateforme *Xbox Live*. Toutefois le 11 septembre 2003, Microsoft force une mise à jour du *dashboard* sur toutes les consoles connectées à Internet incluant des consoles non enregistrées sur la plateforme *Xbox Live* [17]. Sans avertissement, Microsoft supprime donc la possibilité pour la communauté Xbox Linux d'ajouter un menu de lancement de GNU Linux sur le *dashboard*.

Xbox **DRM Xbox**

Comme toutes les consoles de jeux la Xbox intègre des fonctionnalités anti-copie. Celles-ci sont en partie intégrées dans le *firmware* du lecteur de DVD de la console et interdisent tout chargement de code autre que le code des jeux de Microsoft, ce qui interdit à la communauté Xbox Linux d'exécuter du code à partir d'un DVD ou un CD-ROM.

Fimware du lecteur de DVD
Les DVD de jeux Xbox sont organisés en suivant une structure particulière : XDVD. Le lecteur DVD de la Xbox embarque donc un *firmware* personnalisé permettant de lire ces DVD En effet, ces DVD sont physiquement identiques à un DVD double couche standard, mais embarquent plusieurs mécanismes propriétaires interdisant la lecture de certaines informations [7].

La figure 9 présente la structure d'un DVD Xbox. La zone de données d'un jeu est divisée en deux parties, ou « partitions ». La première est une partition de type DVD vidéo. Celle-ci est librement accessible. La seconde correspond à la partie jeu et n'est accessible qu'à partir d'un lecteur original de Xbox. C'est pourquoi lorsque qu'un DVD Xbox est inséré dans un lecteur DVD standard, seule une vidéo ($\tilde{7}$ Mo) indiquant qu'il s'agit d'un jeu Xbox est accessible. Autrement dit, les données du jeu ne sont pas atteignables et ne peuvent donc pas être copiées (en théorie).

Parmi les éléments ajoutés par Microsoft il faut noter la présence de secteurs de sécurité, la présence d'une fausse zone de *Lead OUT* et surtout la présence d'un secteur d'authentification permettant entre autres au lecteur de la Xbox d'identifier qu'il s'agit d'un DVD de jeu Xbox. Ce secteur est positionné dans le *Lead OUT* du média, c'est-à-dire à la fin de la zone de données. Dans le cas des DVD Xbox, il se situe sur la deuxième couche du DVD. Ainsi, ce secteur dont la position est fixe n'est pas reproductible à l'aide d'un graveur DVD simple couche qui correspond au standard de l'époque. D'autre part, la lecture de ce secteur ne peut être faite sans modification préalable du *firmware* des lecteurs du commerce.

Les secteurs de sécurité (*SS*) sont ajoutés entre les blocs des fichiers (figure 10). Leur position est aléatoire. Ils contiennent diverses informations

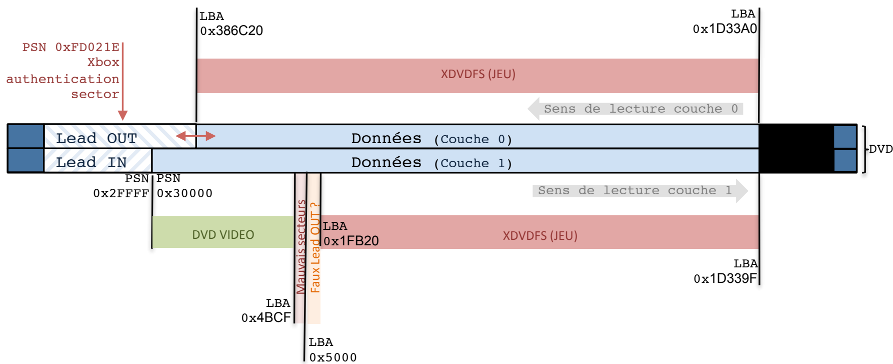

Figure 9. Xbox : particularités de la structure des DVD Xbox et Xbox 360

dont l'authenticité est vérifiée à l'aide d'informations stockées dans le binaire du jeu, lequel est signé avec la clé RSA privée de Microsoft. L'index de la position de chacun de ces secteurs est stocké dans le secteur d'authentification du DVD, qui est quant à lui enregistré dans la zone *Lead OUT* placée sur la deuxième couche du DVD. En ajoutant ces secteurs non référencés par l'index standard (*TOC*) au milieu des données valides, Microsoft tente donc d'interdire la copie des jeux. Pour réaliser une copie il suffit ainsi d'obtenir la liste des secteurs de sécurité pour les identifier et le lire.

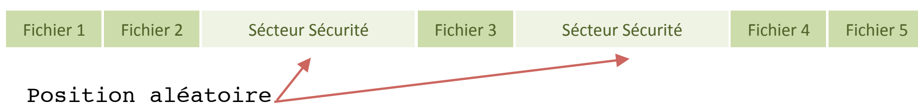

Figure 10. Xbox : organisation des données sur un DVD Xbox et Xbox 360

Format des jeux

Le code exécutable d'un jeux de Xbox se présente sous la forme d'un fichier `.XBE` signé. Ils sont exécutés à partir d'un DVD ou téléchargés depuis la plateforme *Xbox Live*. La signature du binaire inclut la signature de deux champs de données particuliers intégrés à sa structure [37]. Le premier champ : `media flag` permet de définir le type de média sur lequel le

binaire peut être exécuté et le second la région de la console pour laquelle le jeu a été produit : `region code`.

Le *media flag* indique à la Xbox depuis quel média l'exécutable est autorisé à être lancé (CD-R/RW, DVD-R/RW, DVD-XBOX, etc.). Les jeux originaux Xbox contiennent un *media flag* n'autorisant leur lancement que depuis un DVD avec le *media flag* : DVD-XBOX. Cette information n'étant pas reproductible avec le matériel grand public et les binaires étant signés il paraît impossible de réaliser une copie parfaite et de l'exécuter sur une console non modifiée.

Xbox : Attaques

L'intégration de fonctionnalités de sécurité dans un produit de grande consommation implique un budget restreint. Ainsi, il semble que Microsoft ait fait des compromis sur la couverture de certains risques.

Parmi les risques non couverts par les mesures techniques mises en œuvre par Microsoft il faut noter :

— Modification du contenu des modules de sauvegarde ;
— Vulnérabilités dans un jeu ou sur l'écran d'accueil ;
— Extraction du code contenu dans la mémoire flash ;
— Modification du *bytecode* dans la mémoire flash ;
— Ajout d'une mémoire flash alternative sur le bus LPC ;
— Écoute du bus HyperTransport ;
— Modification du *firmware* lecteur de DVD ;
— Analyse, extraction, modification du *firmware* du disque dur ;
— Modification du contenu du disque dur ;
— ...

Ce chapitre présente les attaques résultantes de ces choix techniques et explique les techniques employées par les attaquants pour compromettre la sécurité du système dans le but d'en prendre le contrôle. Seules les vulnérabilités les plus intéressantes sont présentées dans ce document. Le lecteur intéressé se reportera aux travaux de Michael Steil [50] qui apportent une vue détaillée des vulnérabilités identifiées sur la Xbox. Nous présentons sur la figure 11 la chronologie des attaques sur la console.

Xbox *Extraction du contenu de la mémoire flash* 2001

Andrew « bunnie » Huang, alors étudiant doctorant au MIT, a étudié les fonctions de sécurité de la Xbox [46] dès sa sortie aux États-Unis en 2001. Il commence alors par démonter sa Xbox, identifier et dessouder la mémoire flash pour en extraire le contenu à l'aide d'un lecteur/programmateur standard du marché. L'analyse de l'image présentée sur la figure 6 confirme

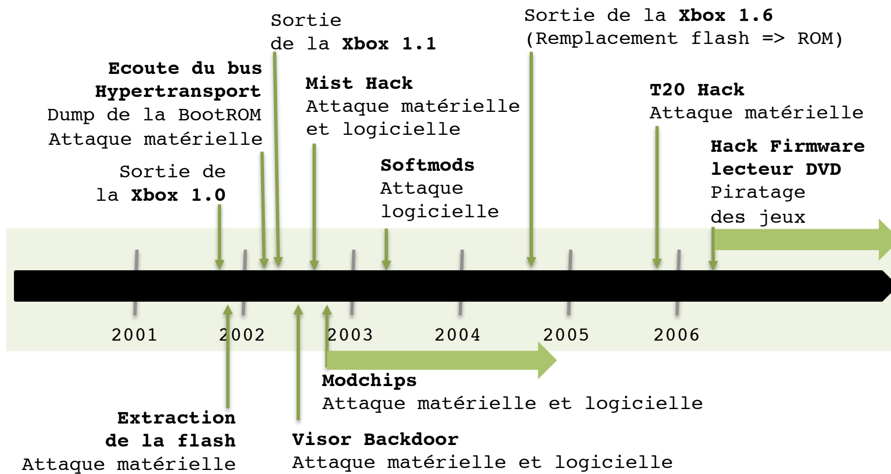

Figure 11. Xbox : chronologie des attaques

qu'une grande partie de la mémoire flash est chiffrée. Bunnie identifie également la présence de code `x86` dans une zone de 512 octets situé au début de l'image et s'attelle alors à son analyse. Il découvre alors le code d'un interpréteur de *bytecode*, une fonction de déchiffrement ressemblant à RC4 et des données ou du code chiffrés. Il se lance alors dans la réécriture de la fonction de déchiffrement pour retrouver le reste des éléments stockés dans la flash externe. Les résultats alors obtenus sont peu probants. En effet, le code obtenu n'est pas valide et ne correspond à aucun *op-code* supporté par l'interpréteur trouvé dans la flash. Ce code apparaît donc comme inutilisé et provenant probablement des plateformes de développement. Cette hypothèse est rapidement confirmée : la Xbox reste fonctionnelle après avoir remplacé ces 512 octets par des zéros. Le code d'initialisation ne se trouve donc pas dans la mémoire flash.

Plus tard, il sera confirmé que le code stocké dans les 512 premiers octets de la mémoire flash correspondent à une version historique du *bootloader* de premier niveau oubliée par les ingénieurs de Microsoft lors de la génération de l'image de production. Cette erreur *a priori* anodine associée à l'absence de chiffrement des données sur le bus HyperTransport s'est avérée décisive lors de la mise au point de l'extraction de la *bootROM*.

Xbox *Écoute du bus HyperTransport* 2001

Bunnie décide alors d'analyser les flux de données accessibles sur la carte

mère à la recherche du code d’initialisation. Il commence par le bus HyperTransport. Ce bus est composé de 16 paires différentielles (8 pour l’émission et 8 pour la réception), de 2 signaux d’horloge (émission et réception) et d’une ligne *strobe*. Les signaux de données sont transmis sur les fronts montant et descendant de l’horloge cadencée à 200 MHz. À l’époque le matériel nécessaire à l’acquisition de ces signaux est jugé trop onéreux pour que l’attaque soit réalisable par un attaquant disposant de moyens limités. Ainsi, les ingénieurs de Microsoft décident de ne pas chiffrer les flux de données transitant sur le bus HyperTransport. Or, le doctorant réussit à développer un module d’acquisition pour $50 (le module est présenté figure 12). Il procède alors au décapage des pistes et connecte le module d’acquisition entre la Xbox et une plateforme à base de FPGA développée initialement pour son projet de thèse (le marquage présent sur la carte mère facilite l’identification du rôle de chacune des pistes composant le bus).

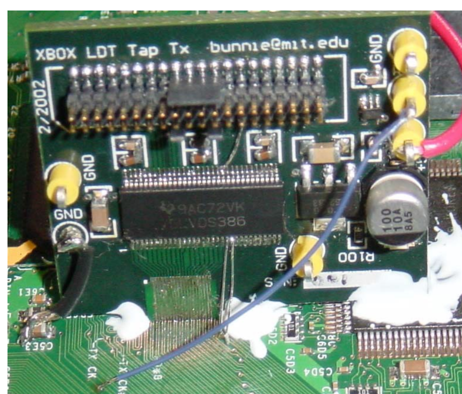

Figure 12. Xbox : acquisition des données transitant sur le bus Hyper-Transport

Une fois l’acquisition des données réalisée, il est nécessaire de les réordonner pour obtenir du code valide. Pour réaliser cette étape il part du principe que le code stocké dans la *bootROM* secrète est similaire au code trouvé dans les 512 premiers octets de la mémoire flash. Il réussit ainsi à reconstruire l’image de la mémoire secrète et extraire la clé de chiffrement RC4 requise pour déchiffrer le code du `2BL`. En effet, lors du démarrage de la console, le code stocké dans le MCPx est transféré au processeur pour exécution. Ainsi, la communauté a pu analyser de manière

détaillée la chaîne de démarrage et notamment la procédure en charge du contrôle de l'intégrité du *2BL*. Cette procédure ne consistant qu'en la vérification d'une constante à une adresse fixe, il fut possible de modifier le *2BL* pour exécuter simplement du code non signé comme le système d'exploitation Linux sur la Xbox.

Réponse de Microsoft
Pour limiter l'impact de cette attaque, Microsoft a sorti une nouvelle version de la Xbox courant 2002. Cette version embarque une nouvelle clé de chiffrement RC4 et une nouvelle procédure de vérification d'intégrité pour le `2BL`, basée sur l'algorithme *Tiny Encryption Algorithm* (TEA). TEA est un algorithme de chiffrement par bloc dont l'implémentation ne fait que quelques lignes de code. Microsoft utilise alors cet algorithme en tant que fonction de hachage autorisant ainsi des attaques par collision [49,50]. Il devient alors une nouvelle fois possible de forger un `2BL` valide permettant de prendre le contrôle du flot d'exécution.

Xbox *Visor Backdoor* 2002
Un des membres de la communauté (Visor) a étudié le code exécuté pour désactiver l'accès à la zone mémoire secrète en cas d'échec de la vérification de l'intégrité du `2BL` après déchiffrement [50]. Comme expliqué dans les chapitres précédents, en cas de code invalide l'accès à la zone mémoire est désactivé. À cause de l'*Overlay* de la mémoire, la désactivation de l'accès à la zone mémoire secrète entraine le basculement du flot d'exécution vers la mémoire externe. Un attaquant disposant d'un accès physique et capable de reprogrammer la mémoire flash externe serait donc en mesure de prendre le contrôle du flot d'exécution en désactivant l'accès à la zone mémoire secrète de façon prématurée.

La désactivation de l'accès à la mémoire secrète est réalisée en définissant le bit 1 à 1 dans le registre `0x80` de l'espace de configuration PCI du périphérique `0:1:0`, ce qui se traduit par la valeur `0x80000880`. L'encodage de l'instruction envoyée sur le port PCI est construite comme présenté dans le tableau 2. Ainsi, l'exécution de l'instruction présentée dans le listing 4 pourrait permettre d'exécuter cette attaque. C'est pourquoi, comme le montre le listing 5, Microsoft interdit l'accès au contrôleur d'*overlay* à l'aide de l'instruction `POKEPCI` avec une valeur de configuration PCI à `0x80000880`.

```
POKEPCI 0x80000880, 2
```

Listing 4. Xbox - Désactivation de la mémoire secrète Xcode bytecode

Bit	Valeur
0-7	Registre
8-10	Fonction
11-15	Matériel
16-23	Bus
24-30	Réservés
31	Toujours 1

Table 2. Xbox - Instruction PCI

```
cmp ebx, 80000880h  ; Configuration PCI = Desactivation bootROM
    ?
jnz short not_mcpx_disable  ; non
and ecx, not 2              ; bit 1 force a 0
not_mcpx_disable:
mov eax, ebx
mov dx, 0CF8h
out dx, eax                 ; Port de configuration PCI Adresse
add dl, 4
mov eax, ecx
out dx, eax                 ; Port de configuration PCI Donnees
jmp short next_instruction
```

Listing 5. Xbox - code exécuté par l'instruction POKEPCI

Pour conduire cette attaque il est nécessaire d'avoir un accès physique à la mémoire flash externe et d'en modifier le contenu en ajoutant du *bytecode Xcode* pour générer l'instruction `jmp 0xFFFF0000` à l'adresse `0x0000:0000` de la mémoire RAM puis corrompre les quatre derniers octets du `2BL` dans la mémoire flash. Au lieu de générer une double faute (comportement attendu par les équipes de Microsoft) et arrêter l'exécution de code, le processeur *wrap* et continu (exécution de l'instruction NOP) jusqu'au début de l'image du `2BL` situé à l'adresse `0x0000:0000`. Afin de garantir la bonne initialisation de la mémoire le *payload* est ajouté à la fin du *bytecode* de Microsoft. Ainsi, il devient possible d'exécuter du code non signé sur la Xbox sans avoir recours à l'utilisation de la clé RC4 secrète. Autrement dit, en ajoutant un saut vers la fin du *bytecode* de Microsoft et en remplaçant le code du `2BL` par celui d'un *bootloader* Linux il devient possible d'exécuter automatiquement GNU Linux sur la Xbox.

La raison pour laquelle le processeur Intel de la Xbox ne génère pas de double faute lors du débordement du compteur d'adresse est historique. Dans les années 1970, les processeurs (8080...) commencent par exécuter le code situé en haut de l'espace d'adressage moins 16 octets, soit à l'adresse `0xFFF0`. Mais certains fabricants d'ordinateurs font pression sur les fabricants de processeurs pour placer leur ROM en bas de l'espace d'adressage, soit à l'adresse `0x0000`. Ainsi Intel, s'est vu contraint de

trouver une solution à moindre coût à ce problème, et a fini par désactiver l'interruption en cas de débordement de l'espace d'adressage et interpréter comme un `NOP` les `0xFF` générés par les erreurs de lecture dues à l'absence de mémoire flash à l'adresse en cours d'exécution. Ainsi, lorsqu'aucune mémoire n'est mappée à l'adresse de démarrage standard `0xFFF0`, le CPU exécute l'instruction `NOP` jusqu'au débordement du compteur de programme et repasse à l'adresse `0x0000` où est mappée la mémoire.

Ce comportement présentant un risque de sécurité et n'étant plus nécessaire au moment où AMD est rentré sur le marché des processeurs, les ingénieurs d'AMD ont décidé de supprimer ce comportement.

Il est donc surprenant que les ingénieurs de Microsoft aient fait cette extrapolation. Mais cela peux s'expliquer par le changement d'architecture. Ainsi que par l'absence de test de cette fonctionnalité de sécurité.

Xbox MIST Hack 2002

Peu de temps après la publication de l'attaque de Visor [50], une autre vulnérabilité [50] a été identifiée dans le code de la fonction responsable de la désactivation de la zone mémoire secrète dont le code est présenté dans le listing 5. Au début de la fonction `POKEPCI` l'instruction de configuration PCI est stockée dans le registre `EBX`. Cette instruction est ensuite envoyée au port d'entrée/sortie situé à l'adresse `0x0CF8`, et les données (32 bits) sont envoyées sur le port d'entrée/sortie situé à l'adresse `0x0CFC`.

L'attaque est assez évidente : seuls 8 bits sont utilisés pour définir l'adresse du registre, et la comparaison est faite sur une valeur unique de 32 bits. Il est donc possible de contourner le test en modifiant une partie des bits 24 à 30. Ainsi il est possible d'utiliser l'instruction `POKEPCI(C0000880h, 2)` pour forcer la désactivation de la mémoire secrète à la place de l'instruction `POKEPCI(80000880h, 2)`, et dévier le flot d'exécution vers la mémoire flash externe, entrainant ainsi l'exécution de code non authentifié. Il est intéressant de noter que cela aurait également fonctionné sur une architecture AMD car cette attaque ne requiert pas l'aide du débordement du compteur de programme.

Xbox Modchips 2002

Les attaques décrites jusqu'à ici nécessitent toutes une interaction avec le matériel, ainsi qu'un équipement dédié ou une intervention risquée sur la carte mère (extraction/remise en place de la mémoire flash). Les membres de la communauté ont donc cherché à identifier le moyen d'ajouter un composant dans la console pour prendre le contrôle du flot d'exécution.

L'analyse du circuit de la carte mère met alors en évidence l'emplacement du bus Low Pin Count (LPC) [16]. Ce bus est généralement utilisé

pour connecter des périphériques à faible bande passante au processeur au travers du *Southbridge*. La liste des périphériques couramment connectés à ce bus est variable : port série RS232, BIOS secondaire...

Le bus LPC est multiplexé sur 4-bits synchronisé sur une horloge (33,3 MHz) pour transférer les adresses et les données. Le bus est donc constitué à minima de sept signaux :

— LCLK : horloge à 33,3 MHz (fourni par l'hôte) ;
— LRESET # : ré-initialisation du bus ;
— LFRAME # : indique le démarrage des échanges LPC (commandé par l'hôte uniquement) ;
— CONT [3 : 0] : ces quatre signaux bidirectionnels sont utilisés pour transmettre les adresses et les données.

Ces signaux sont directement accessibles sur la carte mère des premières générations de consoles, voir figure 13. Il suffit alors de souder un *header* pour y connecter une mémoire flash supportant le protocole LPC type SST49LF020. Les tests réalisés ont montré que lorsqu'une mémoire flash secondaire était connectée au bus et que la mémoire externe n'était pas disponible, le processeur démarre sur la mémoire secondaire. La mémoire secondaire est donc mappée lors du démarrage.

La plupart des *modchips* sont donc connectés à ce bus. Toutefois, pour que le code installé dans une puce soit exécuté il est nécessaire de corrompre les données stockées dans la mémoire flash externe. Cette opération est réalisée en forçant l'état du signal `D0` au niveau logique 0. Ainsi, le système pense que la mémoire flash est vide ou absente.

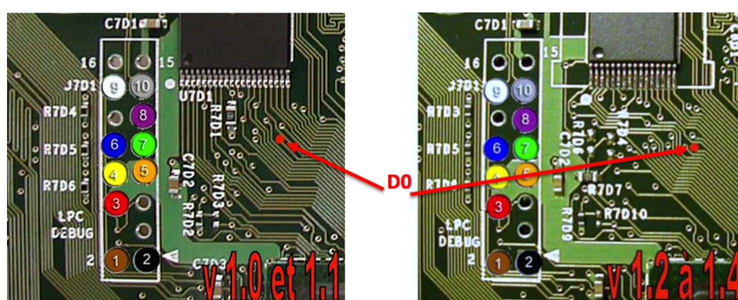

Figure 13. Xbox : identification des points de connexion de *modchip*

En combinant cette technique avec la vulnérabilité de Visor ou de Mist il devient alors possible de déjouer le système de sécurité de la Xbox de façon systématique et transparente pour l'utilisateur. Il suffit donc

de programmer la mémoire flash embarqué sur la puce avec un des deux exploits et les éléments nécessaires au démarrage d'un *Homebrew* ou de Xbox Linux.

Réponse de Microsoft

Face au développement de l'industrie des *modchips* sur sa console, Microsoft a fini par supprimer certaines broches du bus LPC (Xbox révision 1.5), rendant la conception de *modchip* plus difficile. Toutefois, le bus LPC étant employé pour programmer la mémoire flash sur la chaîne de production il fut impossible de désactiver complètement le bus LPC sans revoir le processus de fabrication de la console. Il a fallu attendre la version 1.6 de la console pour voir apparaître des modifications majeures et notamment la suppression de la mémoire flash externe. En effet, dans la version 1.6 Microsoft remplace la mémoire flash reprogrammable par une mémoire ROM intégrée à la puce du *codec* vidéo. Ces modifications rendent ainsi le bus LPC inutile. Toutefois, Microsoft n'ayant pas revu la conception du *MCPx* de Nvidia le démarrage à partir d'une mémoire Flash alternative reste possible : il suffit de reconstituer le câblage du connecteur LPC. La figure 14 présente les modifications réalisées pour reconnecter le port LPC sur une Xbox version 1.6.

Figure 14. Xbox : reconstitution du port LPC sur une Xbox version 1.6

Xbox SoftMod 2003

Jeux et vulnérabilités

Sur la Xbox la surface d'attaque d'un jeu est limitée aux sauvegardes qui sont enregistrées sur des modules externes. Les modules de sauvegarde

Xbox sont des périphériques de stockage USB standards, il est donc possible d'accéder et de modifier le contenu des sauvegardes de l'utilisateur.

Ensuite, plusieurs vulnérabilités ont été identifiées par la communauté :

- 007 : Agent under Fire - Savegame Stackoverflow Exploit ;
- Mechassault - Savegame Stackoverflow Exploit.

L'exploitation de ces vulnérabilités se révèle facilitée car sur la Xbox, tous les jeux Xbox fonctionnent en mode noyau. Un attaquant ayant pris le contrôle du flux d'exécution est en mesure d'accéder au contenu de la mémoire flash externe et d'en modifier les valeurs. Toutefois, cette opération nécessite de souder sur la carte mère pour désactiver le mode lecture seule sur le composant.

Vulnérabilités Dashboard

Il n'est pas possible sans modification matérielle de *jailbreaker* la console de façon permanente, au travers de l'exploitation de vulnérabilité dans un jeu. Il est nécessaire d'insérer le jeu et d'exploiter la vulnérabilité à chaque fois que l'on souhaite exécuter du code non signé. Ce qui n'est pas en faveur de l'expérience utilisateur. La communauté Xbox Linux s'est donc intéressée à la recherche de vulnérabilités en modifiant le contenu des éléments stockés sur le disque dur. Ainsi, plusieurs vulnérabilités ont été identifiées dans le menu d'accueil (*dashboard*) qui est lancé au démarrage de la console lorsqu'aucun jeu n'est présent dans le lecteur DVD :

- Dashboard Font Loader Integer/Heap Overflow ;
- Dashboard Audio CD Ripper Integer/Stack Overflow.

Le tableau de bord charge plusieurs types de fichiers depuis le disque dur (audio, image). Or pour ajouter ce type de fichier sur le disque il est nécessaire de démonter la Xbox et d'accéder physiquement au disque. Le démontage de la console n'est pas une option envisageable pour les membres de la communauté Xbox Linux. De plus le disque dur est **verrouillé** lorsque la console est hors tension. La solution retenue consiste à exploiter une vulnérabilité dans le gestionnaire de sauvegarde d'un jeux pour obtenir l'accès au disque dur et écrire la charge active qui permet alors d'exécuter du code non signé résilient sur le système sans avoir recours à une modification matérielle.

Xbox *T20 Hack* 2005

Cette attaque s'appuie sur la présence d'une fonctionnalité permettant d'assurer la compatibilité des processeurs Intel avec l'ancien processeur Intel 8086. Notons que jusqu'en 2005 Microsoft n'avait pas connaissance de cette technique utilisée par les membres du projet Xbox Linux [50].

À l'origine, le processeur 8086 disposait d'un bus mémoire de 20 bits qui ne lui permettait pas d'adresser plus de 1 Mo. Toutefois, l'adressage en mode réel permettait l'utilisation d'adresses légèrement supérieures à cette limite. Les adresses étaient alors « tronquées » et projetées au début de la mémoire. Par exemple, l'adresse logique `0xFFFF:FFFF` correspond à l'adresse physique `0x0010:FFEF`. La limitation physique du bus faisait que le bit 20 était ignoré et que l'adresse physique effective était `0xFFEF`.

Pour des raisons d'optimisation et de performance, des développeurs utilisaient cette particularité dans leurs programmes. Quand le 80286 a introduit un bus de 24 bits, IBM a ajouté une *gate* permettant de désactiver le bit 20 du bus mémoire pour assurer une compatibilité parfaite avec ses ordinateurs 8086. Comme le 8042 des contrôleurs clavier avait une *pin* vacante, celle-ci a été utilisée pour contrôler la *gate* de la ligne `A20` du bus.

Depuis le 80486, les processeurs Intel n'emploient plus de cache externe mais embarquent un cache interne. Il n'est donc plus possible d'utiliser un contrôleur externe pour piloter le *gate* de la ligne `A20`. Ainsi, Intel a ajouté une entrée spéciale sur ces processeurs. Cette entrée nommée `A20#`, permet de forcer l'état du bit numéro 20 à 0 pour tous les caches internes et pour les bus mémoire externes, transformant ainsi l'adresse de la première instruction exécutée par le processeur initialement définie à `0xFFFF:FFF0` en `0xFFEF:FFF0`

L'attaque T20 consiste donc à envoyer un signal sur la *pin* `A20#` du processeur Pentium afin d'activer le signal `A20`. La Xbox démarre alors en exécutant l'instruction en `0xFFEF:FFF0`. Or, dans le cas de la Xbox, cette adresse correspond à une projection de la mémoire de la flash externe dont le contenu est maîtrisé par l'attaquant.

Ceci permet donc de démarrer la Xbox avec du code non signé, en contournant complètement la chaîne de démarrage de confiance. Cette technique est d'autant plus intéressante que la zone mémoire secrète reste accessible. L'équipe du projet Xbox Linux a employé cette technique pour extraire le contenu de la *bootROM* au travers du bus I^2C.

Xbox Modification du firmware du lecteur DVD de la Xbox 2006

Une partie des informations ajoutées par Microsoft dans la structure du DVD ne sont pas reproductibles simplement à partir des équipements grand public. Les contrefacteurs ont alors opté pour la modification du *firmware* du lecteur DVD afin qu'il retourne les valeurs attendues par la Xbox indépendamment du type de média. Parmi les modifications apportées [13], il faut noter le support du secteur d'authentification du

média placé dans un *Lead OUT* gravé sur un DVD mono couche et la falsification du *media flag.*

Xbox : conclusions sur la sécurité

L'analyse de la stratégie de défense de la Xbox montre que Microsoft a pensé à sécuriser sa plateforme. On note la présence d'une chaîne de confiance, qui est encore rare à l'époque. Toutefois, les analyses et attaques réalisées (notamment par la communauté Xbox Linux) montrent que les fonctionnalités de sécurité ont été réalisées à la va-vite. Ainsi, l'implémentation de la chaîne de confiance sur la Xbox est un échec total.

La présence de code mort dans la mémoire flash externe a permis aux attaquants de comprendre rapidement le fonctionnement de la chaîne de confiance. L'absence de chiffrement a permis l'extraction du code de la *bootROM* et de la racine de confiance (clé RC4). Ainsi il fut possible d'exécuter du code non signé. L'analyse du code de la *bootROM* a autorisé la validation du fonctionnement des mécanismes identifiés lors de l'analyse du code mort et de trouver plusieurs vulnérabilités critiques, permettant de prendre le contrôle du flot d'exécution.

L'absence de tests de sécurité poussés lors du passage d'un processeur AMD à Intel a également permis à la communauté Xbox Linux d'exploiter une différence de comportement entre les deux architectures afin d'obtenir l'exécution de code non authentifié lorsque l'intégrité du code de la mémoire flash externe est altéré. Nous déplorons également :

- L'utilisation de fonctionnalités de contrôle d'intégrité contournable (*Magic*) et vulnérable (`TEA`) ;
- La présence du connecteur LPC ainsi que le support du démarrage sur une mémoire flash LPC alternative ;
- La présence de bus de communication apparent sur une face apparente du circuit ;
- L'impossibilité de mettre à jour le code stocké dans la mémoire flash externe ;
- L'absence de mesure de réduction des risques d'exploitation.

Malgré de bonnes hypothèses de départ, le niveau de sécurité résultant de l'implémentation des fonctions de sécurité sur la Xbox est un échec. La raison réelle de cet échec réside dans la non-maturité du concept de chaîne de démarrage de confiance, le manque de maîtrise du niveau technique de l'attaquant et la probable absence (ou le déficit) de communication entre l'équipe en charge de l'architecture matérielle et l'équipe en charge de l'architecture logicielle. Tous ces facteurs ont été aggravés par une

politique de réduction des coûts de fabrication ayant notamment entrainé des modifications de dernière minute sur le matériel de la plateforme.

2.3 L'âge de raison : la Xbox 360

La Xbox 360 succède à la Xbox et offre une rétro-compatibilité avec une partie des titres parus sur cette dernière. Elle est développée par Microsoft, en coopération avec IBM, ATI, Samsung et SiS. La Xbox 360 est la première console de septième génération, sortie en compétition avec la PlayStation 3 de Sony et la Wii de Nintendo. Elle est d'abord disponible dans une version standard, sortie mondialement en 2005. Une version HDMI voit le jour en 2007 en Europe. En 2010, Microsoft dévoile la Xbox 360 S : comparable à l'ancien modèle, elle bénéficie cependant d'un nouveau design et intègre désormais un disque dur de 250 Go, elle se dote du Wi-Fi et d'un port dédié au nouveau système à détection automatique de mouvement Kinect. La Xbox One lui succède fin 2013.

Avec la Xbox 360 Microsoft répond aux problématiques de sécurité et aux différentes vulnérabilités exploitées par la communauté Xbox Linux pour prendre le contrôle de la plateforme. Ainsi, l'architecture de la Xbox a été pensée pour intégrer de nombreuses fonctions de sécurité. L'attaque de la Xbox 360 s'est donc révélée être un défi de choix pour la communauté qui a fait preuve d'astuce pour finalement en prendre le contrôle.

Xbox 360 : architecture générale

Xbox 360 **Architecture matérielle**

L'architecture de la Xbox 360 est fondée sur un cœur Xenon, un PowerPC adapté par IBM pour Microsoft [41]. La micro-architecture de base est celle d'un PowerPC 970 (G5) 64 bits comme on pouvait en trouver dans les iMac Apple de l'époque. La figure 15 présente une vue simplifiée de l'architecture de la Xbox360. Deux points importants sont à noter :

- Le CPU supporte un mode **hyperviseur** en sus des classiques modes superviseur et utilisateur (seuls les modes hyperviseur et superviseur sont néanmoins utilisés) ;
- En terme d'architecture matérielle, des accélérateurs cryptographiques ont été ajoutés pour interagir avec le contrôleur de cache L2 comme nous le détaillons dans les fonctions de sécurité.

Par ailleurs, le contrôleur mémoire qui pilote la RAM (équivalent d'un Northbridge) est contenu dans le GPU, ce dernier étant directement en

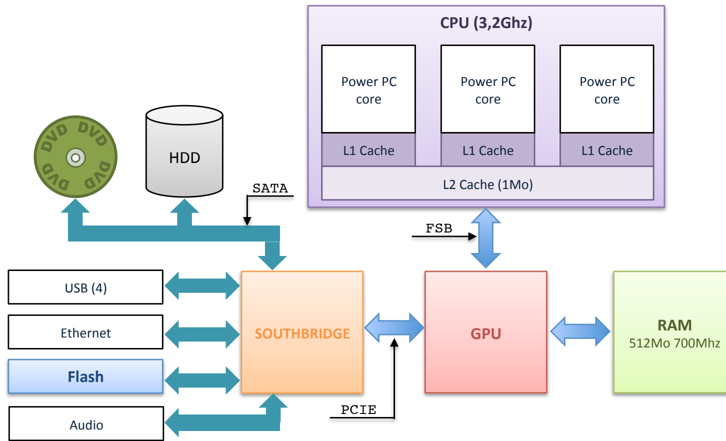

Figure 15. Xbox 360 : architecture matérielle de la Xbox 360

communication avec le FSB du CPU. Cette architecture (qui s'éloigne légèrement de celle des PC classiques) permet notamment des accès mémoire directs entre CPU et GPU optimisés et sans passage par la RAM. Un Southbridge permet les interactions des autres périphériques (périphériques sur le bus PCIExpress et autres) avec le Northbridge.

À des fins d'optimisation, le GPU a un accès non restreint à la RAM et peut donc faire des accès DMA sans limitation. Il n'existe notamment pas d'**IOMMU** au niveau du contrôleur d'I/O, ou de MMU ou MPU interne au GPU comme il peut en exister sur les processeurs graphiques modernes. Les *shaders* exécutés dans le GPU ont notamment un accès illimité à la RAM, qui s'explique *a priori* par un besoin d'accélération des traitements graphiques mais qui a des conséquences sur la sécurité. Microsoft a tenté d'apporter une réponse à ce risque au travers du chiffrement et de la vérification d'intégrité de RAM. Ces contre-mesures sont présentées dans la suite du document.

Xbox 360 *Architecture logicielle*

D'un point de vue logiciel, Microsoft sépare trois éléments présentés sur la figure 16 :

— Le code, les données des jeux et les données des autres applications (multimédia par exemple), qui vivent dans un contexte mémoire non privilégié **superviseur** : Microsoft n'utilise pas le mode **utilisateur** du PowerPC ;

— Le code et les données du noyau (un dérivé du noyau de la première Xbox, lui-même dérivé du noyau Windows 2000) qui vivent dans un contexte non privilégié **superviseur** ;
— L'hyperviseur qui vit dans un contexte privilégié **hyperviseur**.

Les trois cœurs du CPU sont accessibles par les programmeurs de jeu. L'hyperviseur est le seul à gérer directement la MMU via les registres associés. Le noyau est paravirtualisé afin d'utiliser des **hypercalls** vers l'hyperviseur lorsqu'il a besoin de faire des traitements « bas niveau », comme des créations de nouveaux contextes d'exécutions (nouvelles tables de page) ou des transferts en mémoire physique. Cela permet un contrôle d'accès au niveau de l'hyperviseur. Par ailleurs, l'hyperviseur laisse le noyau gérer directement la plupart des périphériques via des zones de mémoire virtuelle *iomappées*.

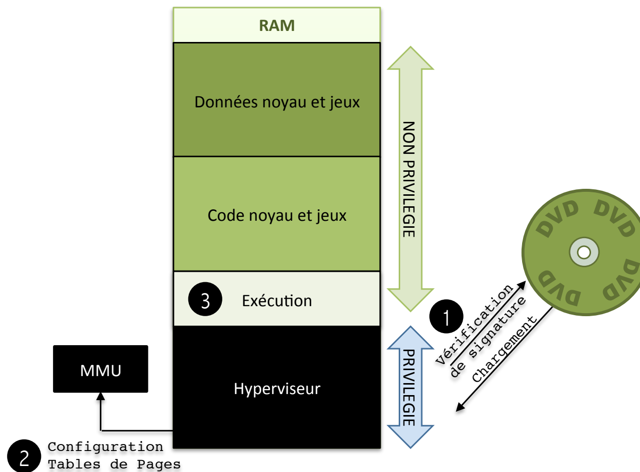

Figure 16. Xbox 360 : architecture logicielle de la Xbox 360

Lorsqu'un DVD de jeu est chargé dans la console, les étapes d'un point de vue macroscopique sont les suivantes :

1. Le code et les données du jeu sont chargés en RAM, l'hyperviseur en vérifie la signature en utilisant la clé publique de Microsoft ;
2. Si la vérification de signature échoue, le jeu n'est pas chargé. Sinon, un nouveau contexte d'exécution est créé pour le jeu : l'hyperviseur crée des nouvelles tables de pages adéquates pour le code et les données (voir plus bas pour une discussion sur les droits liés à ces pages) ;

3. Le nouveau contexte d'exécution est validé et le code du jeu est exécuté.

En termes de format binaire, Microsoft utilise le **Xex**, un conteneur qui encapsule des éléments signés, chiffrés, et compressés. Il contient entre autres des ressources et des binaires PE (*Portable Executable*) classiques [9]. Nous n'entrerons pas plus dans les détails de ce format car il n'est pas nécessaire à la compréhension du modèle de sécurité et des attaques menées sur la console.

Xbox 360 : fonctionnalités de sécurité

Xbox 360 ***Vue globale des mécanismes de sécurités offerts***

Les éléments importants à retenir concernant la sécurité de la Xbox 360 sont :

— Hyperviseur de confiance qui centralise la gestion de la MMU ;
— `W⊕X` sur le code et les données ;
— Intégrité et chiffrement de la RAM ;
— Clé de chiffrement unique par console ;
— Démarrage sécurisé ;
— Mécanismes anti-*downgrade* via eFuse ;
— Signature Microsoft de tout code chargé pour exécution dans la console.

Ces éléments sont détaillés dans la section suivante.

Xbox 360 ***Intégrité et chiffrement de la RAM***

Le processeur PowerPC G5 original implémente une logique de caches très classique : les caches L1 et L2 sont physiquement *taggés* (ils sont accédés avec des adresses physiques). Lorsque la MMU est activée, le CPU manipule des adresses virtuelles [1] sur 64 bits qui sont traduites en adresses physiques sur 64 bits. S'il y a un *cache hit* dans le L1, la donnée remonte dans le CPU, sinon le *cache miss* induit une requête au L2. De manière similaire, un *hit* dans le L2 remonte la donnée au L1 et au CPU alors qu'un *miss* va le chercher en RAM.

C'est au niveau du contrôleur de cache L2 qu'IBM a introduit pour Microsoft des coprocesseurs cryptographiques permettant de gérer l'intégrité et le chiffrement transparent. Comme nous l'avons mentionné, la

1. En fait, ce sont des adresses logiques (effectives dans la nomenclature PowerPC). Nous confondons malgré tout, par souci de simplification, adresses logiques/effectives et adresses virtuelles (qui sont normalement issues de l'unité qui gère la segmentation).

RAM fait 512 Mo, et un bus d'adresses de 32 bits sur le FSB suffit donc amplement à gérer la mémoire physique ainsi que les accès MMIO aux divers périphériques. Ainsi, sur les 64 bits d'adresse physique, les 32 bits de poids fort sont astucieusement **utilisés pour encoder des métadonnées permettant de savoir si la donnée ou le code concerné doit être chiffré et/ou vérifié en intégrité**.

Lorsqu'une donnée quitte le cache L2, si le bit concernant la vérification d'intégrité est positionné dans les 32 bits de poids fort de l'adresse physique, alors un condensat est calculé et stocké dans une SRAM du CPU. De manière similaire, lorsqu'une donnée est récupérée depuis la RAM et vers le L2, le condensat est calculé et l'intégrité est vérifiée en le comparant à celui stocké en SRAM. L'unité de base du calcul est en fait la *cache line*, qui correspond à l'unité atomique d'échange entre L2 et RAM lors d'opérations de *fetchs* de la RAM vers le L2 ou de *write-back* du L2 vers la RAM. La taille des *cache lines* sur le Xenon est de 128 octets.

Cela pose évidemment le problème de stocker les condensats dans la SRAM de toutes les lignes de cache possibles en RAM (plus exactement en mémoire configurée comme *cacheable*). Pour rester sur une taille de SRAM raisonnable, IBM a implémenté une vérification d'intégrité seulement sur une zone **de l'ordre du mégaoctet en mémoire physique basse**. Il semblerait que Miscrosoft utilise une fonction de hachage basée sur des codes linéaires, mais aucune information officielle ne le précise. Notons que pour des condensats de 20 octets à stocker par ligne de cache de 128 octets, cela fait 160 Ko de SRAM nécessaires pour un mégaoctet de mémoire physique (ce qui est raisonnable, sachant que le code de l'hyperviseur fait 128 Ko).

D'un point de vue logiciel, seuls le code et les données de l'hyperviseur sont vérifiés en intégrité. Il faut noter enfin qu'au sens strict, ce n'est pas l'intégrité de la RAM qui est assurée, mais plutôt l'intégrité de la vue CPU de celle-ci (cf. figure 17).

Le chiffrement de la RAM est piloté via les métadonnées des 32 bits de poids fort des adresses physiques de la même manière que pour l'intégrité. Il semblerait que l'algorithme de chiffrement par bloc AES-128 soit utilisé, et la contrainte de stockage de condensats en SRAM disparaît. Il est donc possible de chiffrer toute la RAM. Par ailleurs, la clé de chiffrement AES est tirée aléatoirement au démarrage de la console (via une composition de la sortie d'un bloc RNG matériel, de l'état des *eFuses* de la console, ainsi que d'un *timestamp*), de sorte à éviter les attaques par rejeu. Notons enfin que la propriété de diffusion du chiffrement par bloc (un octet modifié a

une conséquence sur le bloc) ne pose pas de problème du fait de la taille des lignes de cache qui en sont un multiple.

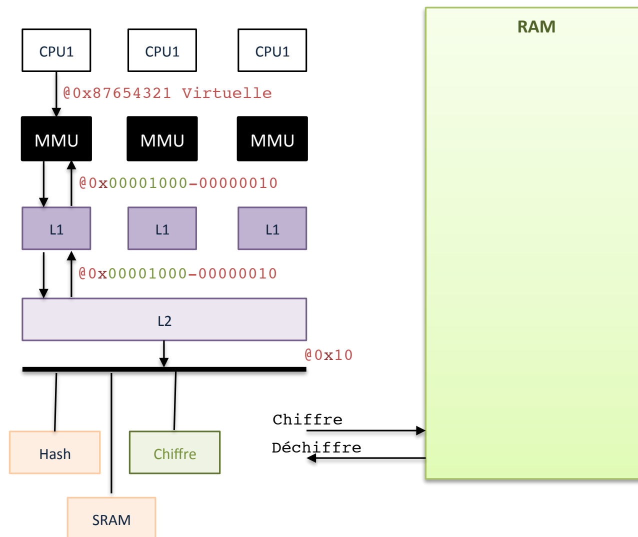

Figure 17. Xbox 360 : chiffrement de la RAM

La décision de chiffrer ou vérifier l'intégrité d'une zone en mémoire virtuelle est prise par l'hyperviseur : celui-ci l'encode dans les entrées de tables de page lorsqu'il configure les projections mémoire.

Il faut enfin noter qu'il est simple de détecter un problème d'intégrité en RAM (via un condensat non valide) : le CPU émet un *reset* dans ce cas. Il n'est par contre pas possible de détecter qu'une donnée chiffrée a été compromise en intégrité.

Xbox 360 *Protection de l'hyperviseur, du code et des données*

Une particularité de l'hyperviseur de la Xbox 360 est qu'il tourne en mode **réel**(cf. figure 18). Les raisons de ce choix ne sont pas très claires (peut être afin de ne pas vouloir gérer les tables de pages qui concernent l'hyperviseur lui-même, ou des raisons de performances). Afin de gérer l'application de l'intégrité et le chiffrement de la mémoire au code et aux données de l'hyperviseur, Microsoft utilise une astuce liée à la programmation du mode hyperviseur du PowerPC : le registre de 64 bits `HRMOR` applique un masque `OR` à l'adresse physique émise par le CPU avant le *fetch* sur le L1 si l'hyperviseur est en mode réel. Il suffit donc de fixer `HRMOR` à une valeur

ayant les 32 bits de poids fort bien positionnés pour s'assurer que le code et les données du mode hyperviseur réel sont chiffrés et intègres.

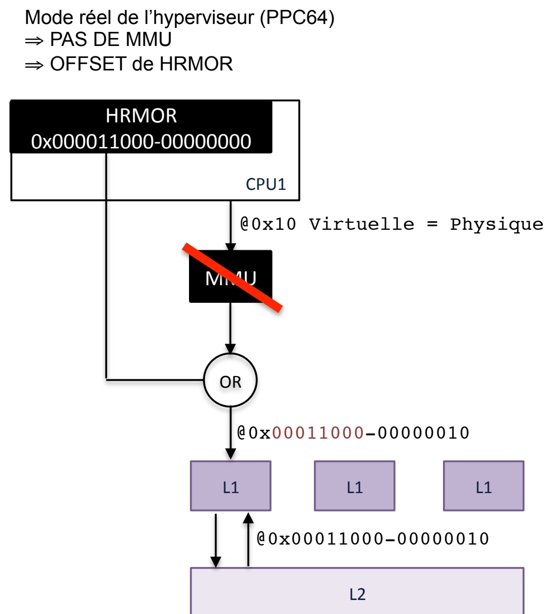

Figure 18. Xbox 360 : mode réel de l'hyperviseur

Par ailleurs, **tout code exécuté est signé avec la clé privée de Microsoft** et dûment vérifié, que ce soit le code chargé lors du démarrage sécurisé ou le code chargé et exécuté depuis un DVD. Ainsi, il est en théorie impossible d'exécuter du code qui n'a pas été écrit par Microsoft ou un développeur tiers de confiance.

Si l'hyperviseur est chiffré et intègre via le `HRMOR`, les pages de code du noyau et des jeux sont chiffrées et configurées en `RX` via la MMU (simplement en les forçant en adresses logiques adéquates) ; les pages de données sont non chiffrées mais en `RW` (sans exécution). Ainsi, la Xbox 360 profite de :

— `W⊕X` interdisant la modification de pages de code et l'exécution de données.

— Une protection contre les attaques DMA venant des périphériques contre la mémoire de l'hyperviseur :

  — La modification de code et de données de l'hyperviseur (dont les tables de page) est protégée par l'intégrité de RAM ;

  - La récupération du code de l'hyperviseur à des fins de *reverse engineering* est empêchée par le chiffrement de RAM.
- Une protection contre les attaques DMA sur le code (noyau et jeux) via le chiffrement de la mémoire.

Notons que le chiffrement des données du noyau et des jeux n'est pas activé car des accès DMA doivent pouvoir être possibles de manière légitime depuis les périphériques comme le GPU.

Xbox 360 Technologie eFuse, clé de CPU, downgrade et KeyVault

La Xbox 360 innove en utilisant la technologie eFuse d'IBM : un ensemble de 768 bits de fusibles matériels que le logiciel peut « griller » électriquement de 0 à 1 de manière irréversible (à savoir sans retour possible de 1 à 0). Ces eFuse sont dans le CPU, donc illisibles depuis l'extérieur de celui-ci.

Microsoft subdivise les 768 bits d'eFuse [52] en 12 **fusesets** [45] de 8 octets qui vont servir trois principaux usages :

- Les *fusesets* 00 et 01 servent à définir l'environnement matériel : console de développement ou de production. Une console de développement a son *fuseset* 01 positionné à `0x0F0F0F0F0F0F0F0F` alors que pour une console de production celui-ci est positionné à la valeur `0x0F0F0F0F0F0F0FF0`. Le fait que le dernier octet de ces deux valeurs soient complémentaires a pour conséquence de bloquer le passage d'une console de développement à une console de production, et réciproquement.
- Les *fusesets* 03 et 05 sont grillés aléatoirement au premier démarrage de la console, ce qui fournit **une clé CPU unique et aléatoire de 128 bits par console**. Cette clé sert lors du démarrage sécurisé comme nous l'expliquons dans le paragraphe qui suit, ainsi que comme graine pour diverses dérivations cryptographiques. Les *fusesets* 04 et 06 sont une copie de cette clé pour avoir de la redondance.
- Les *fusesets* 02 et 07 à 11 servent au mécanisme anti-*downgrade* :
  - Les *fusesets* 07 à 11 servent de LDV (LockDown Values) : à chaque mise à jour du *dashboard* (système principal de la console), un des bits du *fuseset* associé est grillé à un et le compteur LDV associé, stocké en NAND (dans la section du `6BL/CF`) est incrémenté. Lors du démarrage, si l'état du *fuseset* et du LDV en NAND ne sont pas cohérents, c.à.d si le poids de Hamming (nombre de bits à un) du *fuseset* est strictement supérieur au LDV, le *bootloader* *brick* la console (avec pour

conséquence le fameux `RROD`, ou *Red Ring Of Death*). Notons que plutôt que de griller un unique bit à chaque mise à jour, un groupe de quatre bits `0xF` est en fait grillé. Cela signifie que toutes les Xbox 360 sont limitées à environ $64 \times 5/4 = 80$ mises à jour au maximum.

— Le *fuseset* 02 sert pour les mises à jour et la **révocation** du *bootloader* de plus bas niveau (`2BL`) : Microsoft le réserve aux mises à jour critiques de celui-ci (impliquant des mises à jour de sécurité) vu le faible nombre de bits disponibles dans ce fuseset. Le fonctionnement de la vérification de ce *fuseset* diffère légèrement de celui du *dashboard*. Le *fuseset* 02 contient la version minimale admise du `2BL` (modifiée lors d'une mise à jour, les fuses sont grillés par groupe de 4 bits). Le code du `2BL` contient de son côté un numéro de version ainsi qu'un masque de bits permettant aussi de gérer **des versions admises**. Lors du démarrage, le `2BL` lit le *fuseset* 02 et vérifie que cette valeur est bien cohérente avec sa version et le masque de versions admises. Un LDV est aussi stocké en NAND dans la section du `2BL` et incrémenté à chaque mise à jour de celui-ci. Enfin, la vérification LDV $\geq$ *fuseset* 02 est effectuée. Un `RROD` est généré s'il y a eu un problème durant ces vérifications.

Notons que la clé CPU sert notamment à chiffrer le **KeyVault** qui est une zone de mémoire NAND qui sert à stocker divers éléments dont les certificats de la console, ses clés privées d'authentification, les clés d'appairage DVD, le numéro de série, la date de fabrication... Le KeyVault sert donc à authentifier la console sur le Xbox Live, et à détecter/gérer le bannissement de consoles considérées comme piratées. Le chiffrement du KeyVault avec la clé CPU permet d'appairer celui-ci à la console.

Xbox 360 Démarrage sécurisé

Afin de s'assurer que la phase de démarrage est intègre, un mécanisme à plusieurs étages reposant sur divers algorithmes cryptographiques a été mis en œuvre [8]. Cette chaîne de démarrage inclut également les fonctionnalités de protection et détection de *downgrade*.

Description des étapes de démarrage :
Il faut tout d'abord noter que Microsoft gère le chargement de l'hyperviseur et du noyau d'une manière assez singulière : la version de base en sortie d'usine du noyau (1888) est présente dans la NAND durant toute la vie de la console, et les mises à jour sont en fait stockées sous forme de *patchs* différentiels appliqués en RAM sur la version de base lors du démarrage.

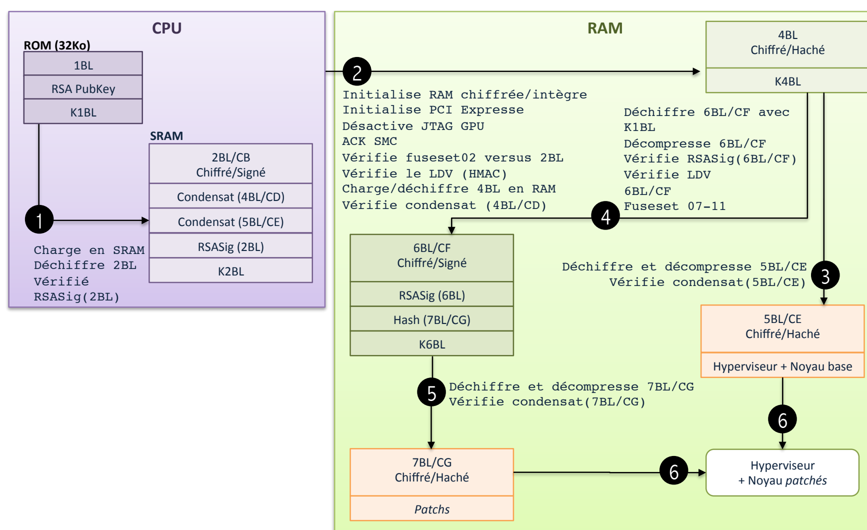

Figure 19. Xbox 360 : démarrage sécurisé

Voici les différents étages de démarrage, leur protection cryptographique, et leur rôle :

1. Le premier étage de démarrage, `1BL`, est dans une *bootROM* de 32 Ko au sein du CPU. Il contient le code de l’algorithme RSA (avec la clé publique de Microsoft), ainsi qu’une clé secrète RC4 de 16 octets notée `K1BL` dans la suite du document (attention à ne pas confondre cette clé commune à toutes les consoles avec la clé CPU précédemment décrite). Il déchiffre le second étage `2BL` (alias CB) – récupéré depuis la NAND – dans la SRAM interne de 64 Ko, et en vérifie la signature (présente dans le *header* du `2BL` déchiffré).

2. Le `2BL` réalise l’initialisation de divers composants tel que : le coprocesseur en charge de la gestion de la mémoire chiffrée intègre, le bus PCIExpress, le port série et la mémoire RAM. Il est également en charge de la désactivation du JTAG du GPU et de la vérification de la cohérence du *fuseset* 02 avec la version et le masque de bits encodés dans son *header*. Il vérifie aussi la cohérence de son *pairing* : à l’aide d’un HMAC, il s’assure que son LDV qui est stocké en NAND est cohérent avec celui stocké dans le *fuseset* 02 (voir figure 19). Si l’un des précédents élément échoue, le `2BL` provoque un `RROD`. Sinon, il récupère le `4BL` (alias CD) stocké en NAND et le déchiffre en RAM. Ensuite, il en vérifie l’intégrité via un condensat SHA-1 qu’il stocke. Notons pour être complet que le SHA-

1 employé par Microsoft, nommé *RotSumSha1* par la communauté, diffère légèrement du SHA-1 classique (vraisemblablement à des fins d'obfuscation).

3. Le contrôle est transféré au `4BL`. Celui-ci déchiffre et décompresse le `5BL` (alias CE) qui correspond à un noyau de base (le noyau d'origine de la console, de numéro 1888) auquel seront appliqués des *patchs*. Lorsque l'on parle de noyau, on parle en fait de code comportant à la fois l'hyperviseur et le noyau qui tourne en mode superviseur. Par ailleurs, le `4BL` récupère le condensat du `5BL` dans le `2BL` et en vérifie l'intégrité avant son exécution.
4. Une fois que le `5BL` a décompressé et préparé en mémoire les structures de l'hyperviseur et du noyau, il rend la main au `4BL` qui charge, déchiffre, vérifie la signature et décompresse le `6BL`. Il est à noter que le `6BL` est chiffré avec la clé `K1BL` du `1BL`, et non avec la clé `K4BL` comme c'est le cas pour le `5BL`.
5. Le `6BL` (alias CF) s'occupe de patcher en mémoire ce qu'a chargé le `5BL`, les données du « *patch* » étant dans le `7BL` (CG) dont il vérifie l'intégrité avant d'appliquer les différences. Le `6BL` vérifie de plus les données de pairing et le LDV qui lui sont associés dans sa section en NAND correspond avec le *fuseset* 07-11 : comme pour le `2BL` un HMAC avec la clé CPU est utilisé (voir ci-après pour plus de détails).
6. Le `6BL` exécute l'hyperviseur via un saut sur son vecteur d'interruption de *reset*. Ce dernier configure ce qu'il faut et charge avec le noyau le reste du système depuis le système de fichiers en NAND (le *dashboard* de la Xbox 360 est lancé).

Le déchiffrement RC4 de chaque *bootloader* ne se fait pas directement avec la clé RC4 présente dans le *bootloader* précédent. La clé du *bootloader* courant permet de dériver une clé diversifiée à partir de la clé RC4 via un HMAC-SHA-1. Seuls 16 octets sur 20 du résultat sont retenus et constituent la clé RC4 effective de (dé)chiffrement. Ainsi, si l'on prend l'exemple du `1BL` et du `2BL` : la clé effective `K2BL` de déchiffrement du `2BL` est calculée comme les 16 premiers octets de l'opération `HMAC-SHA-1(K2BL_HEAD, K1BL)` où `K2BL_HEAD` correspond à la valeur de 16 octets stockée dans le *header* du `2BL` avant déchiffrement. De même, `K4BL` sera égal à `HMAC-SHA-1(K4BL_HEAD, K2BL)`...

Remarquons qu'il est complexe pour un attaquant externe de compromettre ce démarrage sécurisé, par exemple via des attaques DMA. En

effet, le `1BL` et le `2BL` s'exécutent dans le CPU (ROM et SRAM). Après le `2BL`, tous les autres étages s'exécutent en RAM chiffrée et intègre.

Il faut aussi noter que les clés de (dé)chiffrement des étages de *bootloaders* sont embarquées dans ceux-ci, chaque étage déchiffre le suivant. Elles sont **communes à toutes les consoles** (cela est du moins vrai sur les premières générations de Xbox 360, mais a été modifié sur les versions récentes comme nous l'expliquons dans la section consacrée aux attaques).

Mécanisme d'appairage de la console, anti-downgrade :
Dans le flot d'exécution du démarrage sécurisé que nous venons de décrire, nous n'avons pas complètement abordé la gestion du *downgrade*. En effet, les LDV sont des compteurs stockés en NAND, et *a priori* rien n'empêche un attaquant qui maîtrise la NAND d'aller les modifier à une valeur supérieure à celle des *fusesets* 02 et 07-11 pour passer les tests effectués par le `2BL` et le `6BL`. C'est ici qu'intervient la notion de **pairing** : il est nécessaire d'appairer les LDV en NAND aux versions de *bootloaders* associés lors des mises à jour. Il est aussi nécessaire d'appairer ces éléments à la console elle-même, afin d'éviter des attaques par rejeu via l'image de la NAND d'une autre console. C'est ainsi qu'intervient la clé CPU : l'appairage se fait **via un HMAC-SHA-1** comprenant une partie du `xBL` (`2BL` ou `6BL`) et le LDV associé, avec la clé CPU comme clé secrète. Seuls 16 octets du résultat sur 20 octets sont retenus et stockés en NAND avec le code du `xBL` dans la section *ad hoc* (le tout chiffré avec la clé dérivée d'un *bootloader* précédent comme déjà décrit). Voici par exemple la structure des informations pour le `2BL` (celle pour le `6BL` est équivalente pour les données de pairing) :

```
0x00: "CB", length, version, mask ...
0x10: RC4 Key (correspond à \texttt{K2BL})
0x20: Pairing Block (correspond aux Pairing data + LDV)
0x30: CB Hash (correspond au HMAC-SHA-1 de pairing)
```

Le **Pairing Block** contient en fait 4 octets de la forme `XXXXXXYY` où `XXXXXX` est une donnée servant au pairing de la console (dérivée de la clé CPU), et `YY` est le LDV incrémenté à chaque mise à jour. Les informations de version stockées au début sont celles utilisées par le `2BL` lors de la vérification des versions révoquées.

Résumé :
Les différents étages de *bootloaders* sont liés entre eux via des signatures, des condensats et des clés de (dé)chiffrement. Il n'est donc pas aisé par exemple de « mélanger » un `4BL` ancien avec un `2BL` récent (le condensat ne sera pas le bon). En sus, les protections anti-*downgrade* assurent de manière matérielle que la version des éléments principaux du système

(2BL : premier élément exécuté depuis la NAND, et 6BL : gestion des mises à jour du noyau) ne peuvent être *downgradés*.

Xbox 360 Modèle de sécurité sur Xbox 360

Pour résumer, les seules classes d'attaque que l'attaquant va avoir à disposition sont :

— Le ROP (*Return Oriented Programming*) et les *ret2lib* pour exécuter du code déjà existant ;

— Le DMA vers les zones non chiffrées, ou le DMA « aveugle » vers les zones chiffrées sans intégrité.

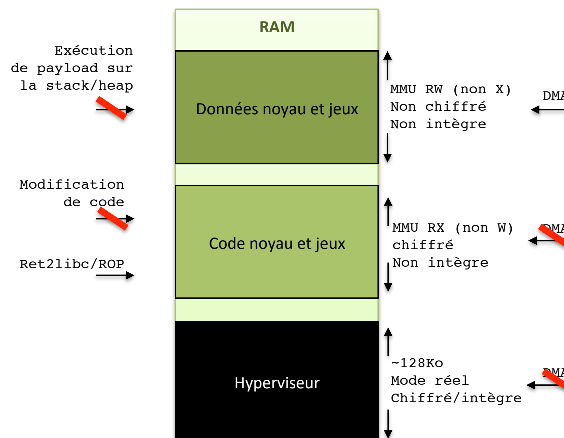

Figure 20. Xbox 360 : modèle de sécurité

Un exemple de chemin d'attaque serait l'injection d'adresses de retour dans la pile d'un processus via DMA permettant ainsi, de faire du ROP et de faire exécuter un *payload* avec le code existant. Malheureusement, il existe deux freins à cette attaque :

— L'intérêt d'une telle attaque serait d'injecter un *payload* exécutable maîtrisé (par exemple injecté via DMA aussi), or toute nouvelle projection mémoire est sous le contrôle de l'hyperviseur, et celui-ci vérifie soigneusement que chaque nouvelle page exécutable contient du **code signé**.

— Pour mener à bien l'attaque DMA, il faut pouvoir contrôler un périphérique qui en est capable.

Malgré ces contraintes, la première attaque référencée sur la Xbox 360 a été purement logicielle et a tiré parti des deux seules faiblesses (présentées sur la figure 20) de la protection mise en place par Microsoft.

Xbox 360 : attaques

La présente section a pour objectif de dresser l'historique des attaques sur la Xbox 360, tout en donnant les détails techniques de ces attaques. La figure 21 donne une vue d'ensemble de leur chronologie.

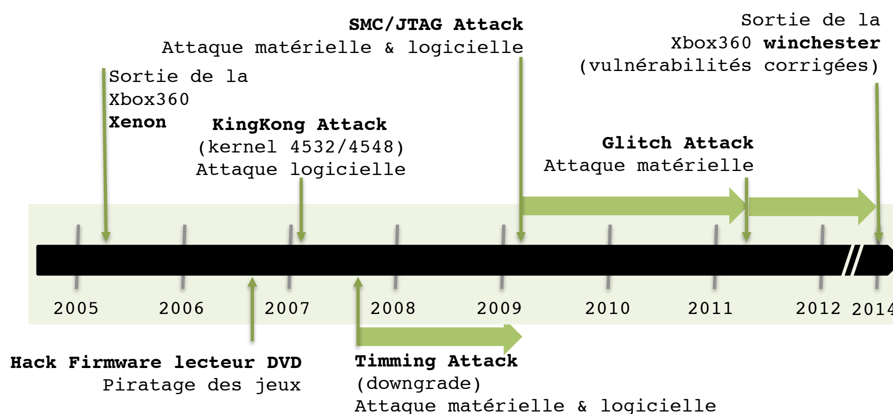

Figure 21. Xbox 360 : chronologie des attaques

Xbox 360 *Piratage de jeux* 2006

Paradoxalement (et contrairement au cas de sa concurrente la PS3), Microsoft n'a pas implémenté de protection très poussée contre le piratage de jeux. En effet, le lecteur de DVD a une interface SATA standard (il peut donc être branché sur un PC). Les seules protections implémentées sont :

— Un appairage entre le lecteur de DVD et la console via une clé partagée (la DVD Key) qui est d'une part contenue dans le KeyVault de la console chiffrée avec la clé CPU, et d'autre part contenue dans le *firmware* du lecteur DVD. Ainsi, il n'est pas possible d'utiliser un autre lecteur DVD que celui d'origine sur une console ;

— L'envoi de chaînes de caractères d'identification spécifiques lorsque le lecteur reconnaît un DVD original inséré.

Il suffit alors de **modifier le firmware** [7] du lecteur DVD afin qu'il envoie les bonnes chaînes de caractères d'identification de DVD originaux pour pouvoir exécuter des copies de jeux (le fait de conserver le lecteur d'origine permet de garder l'appairage via la DVD Key). Une protection aussi peu poussée s'explique par le fait qu'à l'époque de la conception de la Xbox 360, peu d'alternatives existaient. En effet, la seule alternative viable était le chiffrement de disque à la AACS tel qu'apparu sur le Blu-Ray (et dont la PS3 profite par ailleurs). À l'époque, Microsoft militait pour l'adoption du HD-DVD, concurrent du Blu-Ray, qui implémentait de telles protections. Celui-ci n'a jamais vraiment été adopté comme standard, et cela a joué en défaveur de la Xbox 360.

Notons malgré tout que les consoles piratées avec un *firmware* de DVD modifié sont détectées par l'OS, et peuvent être bannies du Xbox live lors de campagnes régulières. Le piratage reste donc sous contrainte.

Le piratage de jeux au sens strict ayant été résolu assez vite, la communauté s'est focalisée sur la problématique d'exécution de code non signé (dont GNU Linux) qui présente un challenge autrement plus élevé.

Xbox 360 *La « King Kong Attack » (noyaux 4532/4548)* 2007

Cette attaque est la première attaque a avoir été conduite sur la Xbox 360 [48]. Elle porte ce nom car le *payload* de l'attaque a été implémenté dans un *shader* du jeu King Kong qui n'était pas couvert par la signature RSA du jeu. Cette attaque rendu possible par la modification du *firmware* du lecteur DVD, a permis une **élévation de privilège au niveau de l'hyperviseur**, permettant donc une prise de contrôle complète, l'exécution de code non signé, ce qui a donné naissance à Free60, un Linux modifié pour s'exécuter sur Xbox 360.

La vulnérabilité [42] exploitée concerne exclusivement les noyaux en versions 4532 et 4548, ce détail est important car il explique pourquoi beaucoup d'attaques ultérieures se sont focalisées sur le *downgrade*.

Le principe de l'attaque est présenté sur la figure 22. La première étape de l'attaque consiste donc à faire des écritures mémoire en RAM non chiffrée via des accès DMA depuis le *shader* maîtrisé qui s'exécute dans le GPU : cela est possible car comme nous l'avons précédemment décrit, le GPU a une vision complète et non restreinte de la RAM. Les données intéressantes à manipuler pour détourner le flot d'exécution sont les adresses de retour des fonctions. Vu l'aspect dynamique des piles lors de la vie du système, y injecter du code par des accès DMA pour détourner

le flot d'exécution présente un risque d'instabilité. S'ajoute à cela la problématique de maîtrise des données dans les registres une fois la prise de contrôle effectuée (elle pourrait se gérer avec du ROP mais constitue aussi une source d'instabilité). Par contre, d'autres données plus stables du noyau sont intéressantes : ce sont les données de contexte des *threads* sauvegardés et restaurés lors des changements de contexte du *scheduler*. Le contexte des *threads* contient notamment l'état des registres (dont le PC) d'un *thread*. Il suffit donc de modifier le contexte sauvegardé d'un *thread* endormi pour avoir la maîtrise du processeur (PC et registres) lorsque le *thread* est réveillé. Or le contexte du « *thread idle* » est sauvegardé à une adresse connue en mémoire physique.

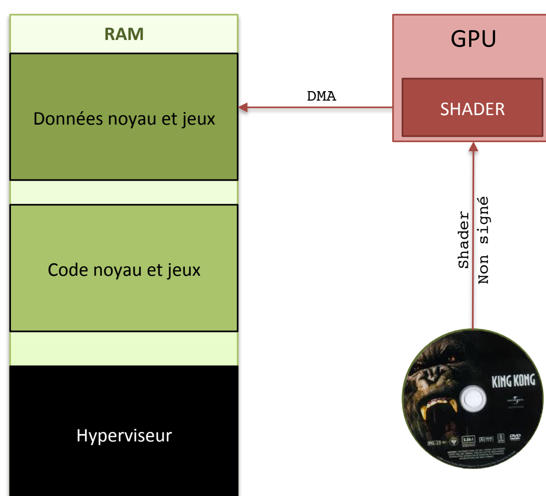

Figure 22. Xbox 360 : principe de l'attaque

La seconde étape (l'élévation de privilèges à proprement parler) consiste en l'exploitation d'une vulnérabilité dans le *syscall handler* (voir figure 23) des hypercalls de l'hyperviseur dont le code assembleur PowerPC et le pseudo-code C équivalent sont donnés. Lors d'un appel système via l'instruction `sc` depuis le mode superviseur, la routine d'interruption de l'hyperviseur prend la main, bascule en mode réel, et appelle le *syscall handler*. Ce dernier récupère le numéro de *syscall* dans `r0`, les arguments du *syscall* sont dans les registres à partir de `r3`. Le code du *handler* récupère le numéro de *syscall*, vérifie s'il est supérieur à `0x61` (nombre d'entrées dans la table des *syscalls*), et lève une exception si c'est le cas.

Sinon, il récupère l'entrée correspondante dans la table et l'appelle avec les arguments fournis.

La vulnérabilité exploitée se situe dans l'utilisation de l'instruction de comparaison `cmplwi` qui compare les 32 bits de **poids faible** de l'opérande `r0` à `0x61`. Or le déréférencement dans la table des *syscalls* se fait avec les 64 bits de l'opérande. Ainsi, l'attaquant réussit à faire passer le test en maîtrisant la partie haute.

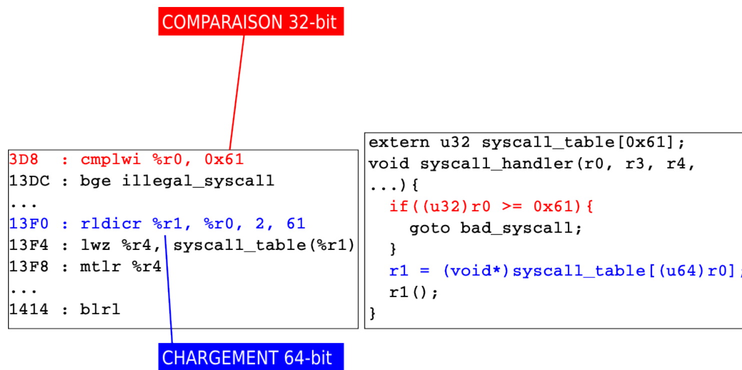

Figure 23. Xbox 360 : la vulnérabilité dans le *syscall handler* de l'hyperviseur

Il faut alors d'imposer un appel vers une zone mémoire maîtrisée. Malheureusement, comme nous l'avons vu, la partie haute des 64 bits de `r0` n'encode que des métadonnées, et pour passer le test il faut que la partie basse soit inférieure à `0x61`. Par ailleurs, l'application du masque logique OU du `HRMOR` imposera (vu qu'on a un ou logique) les bits de *fetch* en mémoire intègre et chiffrée. L'astuce consiste alors à mettre à contribution une autre subtilité de l'architecture PowerPC : lorsque le bit de poids fort des 64 bits d'adresse est positionné en mode hyperviseur réel, le `HRMOR` **n'est pas appliqué**. L'attaquant arrive alors à détourner l'exécution du syscall vers une adresse de l'espace hyperviseur dont les bits d'intégrité et de chiffrement ne sont pas positionnés (cf. figure 24). Le contenu de la case du *syscall* concerné sera interprété par le CPU sans déchiffrement ni vérification d'intégrité : c'est une primitive d'exploitation idéale en complément d'une attaque DMA.

Mettons bout à bout l'utilisation des différentes primitives de l'attaque, cela donne :

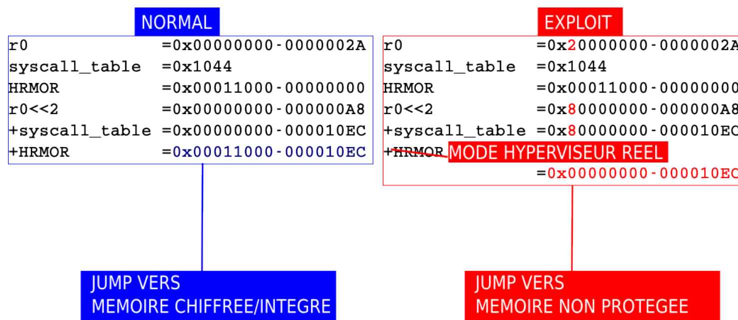

Figure 24. Xbox 360 : exploitation de la vulnérabilité dans le *syscall handler* de l'hyperviseur

1. L'attaquant modifie depuis une attaque DMA le contenu d'une des cases de la table des *syscalls* avec une adresse qu'il maîtrise. Il faut absolument écraser un *syscall* rarement utilisé (au risque que celui-ci soit utilisé avant la fin de l'exploitation...) ;
2. L'attaquant modifie ensuite le contexte du *thread idle* qui est à une adresse physique connue : il en profite notamment pour faire pointer le PC sauvegardé vers une adresse du noyau qui déclenche un *hypercall* via l'instruction `sc`. Le registre `r0` est positionné au numéro de *syscall* dont l'entrée a été compromise en (1) avec une partie haute permettant l'exploitation ;
3. Lorsque le *scheduler* rend la main au *thread idle* qui était suspendu, l'exécution est redirigée vers l'instruction `sc` ;
4. Le *handler* d'interruption de l'hyperviseur prend la main, passe en mode réel et appelle le *syscall handler*. La vulnérabilité est exploitée et l'appel est redirigé vers l'adresse maîtrisée par l'attaquant sans déchiffrement ni vérification d'intégrité.

L'association des ces primitives est présentée sur la figure 25. Les choses sont en fait un peu plus complexes pour la dernière étape : en effet, une fois l'exécution redirigée en mode hyperviseur, le `HRMOR` sera encore appliqué (et l'attaquant ne maîtrise qu'une adresse de 32 bits car les cases mémoire de la table des *syscalls* font cette largeur). Ainsi, l'attaquant ne peut que déclencher une chaîne de ROP dans l'espace chiffré et intègre de l'hyperviseur. Il faut alors rediriger via un saut l'exécution vers une instruction de l'hyperviseur permettant un `jump` vers un registre maîtrisé, dont le bit de poids fort est fixé à 1 afin d'annuler l'effet du `HRMOR` une seconde fois. Le *payload* maîtrisé (et injecté via des accès DMA par une

étape précédente) peut alors en premier lieu désactiver l'application du HRMOR pour les instructions suivantes. Le *payload* est donc exécuté en mode hyperviseur réel en mémoire non chiffrée et non intègre.

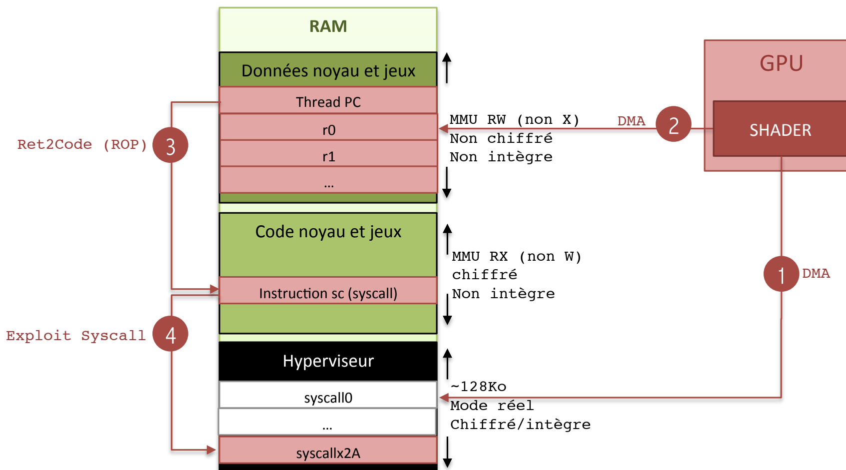

Figure 25. Xbox 360 : la « King Kong Attack » complète

Microsoft a très rapidement patché cette vulnérabilité ; (ils avaient en fait été avertis de celle-ci bien avant son annonce publique). Le correctif consistait en la correction de la comparaison sur 32 bits, mais aussi en l'introduction de la protection par chiffrement des structures mémoire critiques du noyau (dont les contextes d'exécution des *threads* font partie). La conséquence concrète fut que la King Kong attack n'a pas pu être exploitée directement, car la mise à jour a été diffusée rapidement via le Xbox Live (à l'exception des consoles non connectées).

Xbox 360 *Autres vulnérabilités sur l'hyperviseur* Aucune

Il est intéressant de noter qu'aucune autre vulnérabilité n'a été rendue publique, sur l'hyperviseur de Microsoft. Cet hyperviseur est minimaliste, il a été pensé pour être robuste. Les efforts de la communauté *homebrew* se sont donc portés sur le *downgrade*, afin de pouvoir revenir sur un noyau 4532 ou 4548 vulnérable et exploiter la King Kong attack et exécuter du code non signé.

Début de la course au downgrade :

Suite au correctif des noyaux vulnérables à la *King Kong attack*, le *down-*

grade **est devenu une stratégie plausible pour exécuter du code non signé**. L'attaquant gagne s'il *downgrade* directement la version du noyau vers une version vulnérable (4532/4548), ou s'il *downgrade* dans une version inférieure (par exemple la version de base 1888) pour ensuite appliquer « à la main » les mises à jour publiques vers les versions vulnérables.

Xbox 360 *Downgrade : neutralisation des fusesets via le R6T3* 2007

La communauté a découvert assez tôt qu'il était possible sur les premières révisions de la console de neutraliser les mises à jour des *eFuses* via la suppression d'une résistance de 10 KOhms (nommée R6T3 sur le PCB) [20]. Cette résistance de *pull up* permet d'activer le circuit de charge fournissant la tension nécessaire aux opérations de destruction des *eFuses* par le CPU (ce dernier ne génère pas directement cette tension, un circuit dédié de type MBT3904 est utilisé pour les eFuses).

La suppression de cette résistance a pour effet de ne plus faire évoluer les eFuses, et ainsi de garder la possibilité de *downgrader* sa console. Cette modification peut néanmoins s'avérer dangereuse car le mauvais fonctionnement des *eFuses* est détectable de manière logicielle, et Microsoft peut ainsi bannir une console repérée comme telle.

Cette modification a donc été que très peu employée, et les attaquants ont dû trouver d'autres vulnérabilités et moyens plus efficaces pour exploiter le *downgrade* comme nous le verrons ci-après.

Xbox 360 *Downgrade : la timing attack à la rescousse (2BL 1920)* 2007

Une idée naïve de *downgrade* serait de reflasher la NAND d'une Xbox 360 avec l'image d'une chaîne `2BL`/`4BL`/Noyau vulnérable. Cette idée n'aboutit à rien (si ce n'est un *brick* de la console) puisque le mécanisme anti-*downgrade* le détectera via la vérification des LDV et des *fusesets* 02/07-11 associés. Par ailleurs, sans avoir la clé CPU, il n'est pas possible de forger un HMAC entre un `xBL` et un LDV donnant un appairage valide.

Même si elle n'a pas été exploitée en masse, les avantages de la *King Kong attack* sont doubles :

— Les attaquants ont pu récupérer le code des divers *bootloaders* impliqués dans la chaîne de démarrage sécurisé. Ils ont notamment pu récupérer la clé de chiffrement RC4 mise en dur dans la ROM du `1BL`, ce qui leur permet de déchiffrer toutes les mises à jour de la chaîne de démarrage. Cela a été un atout non négligeable pour découvrir une nouvelle vulnérabilité dans le `2BL` à l'origine de la **timing attack**. Le fait que le `1BL` soit en ROM rend impossible toute opération de mise à jour.

— L'exploitation de la *King Kong attack* sur une console vulnérable permet de récupérer la clé CPU d'une console via la lecture des *fusesets* 03 à 05. À partir de là, le contrôle des mises à jour de la console est complet. Il est notamment possible d'outrepasser les contrôles de LDV et pairing de la console.

Cet état de fait aboutit à un cercle vicieux sur les consoles qui ont été mises à jour sans avoir exploité la King Kong attack : pour avoir la clé CPU, il faut exploiter l'attaque. Or pour exploiter l'attaque, il faut *downgrader* le noyau, et donc forger un appairage valide, ce qui n'est pas possible sans clé CPU.

Néanmoins, les attaquants se sont rendus compte au travers de l'ingénierie inverse du `2BL` en version 1920 que la fonction de HMAC-SHA-1 qui vérifie son appairage avec sa LDV utilise une implémentation en **temps variable** (voir figure 26). En effet, une fois le HMAC calculé en utilisant la clé CPU, la comparaison entre la valeur calculée et la valeur stockée en NAND se fait avec un `memcmp` **octet par octet**. Ainsi, le temps mis pour cette comparaison dépend du nombre de premiers octets égaux entre le HMAC calculé et le HMAC en NAND (voir figure). La vérification de HMAC faite dans le `6BL`, pour vérifier son LDV ; souffre de la même faiblesse.

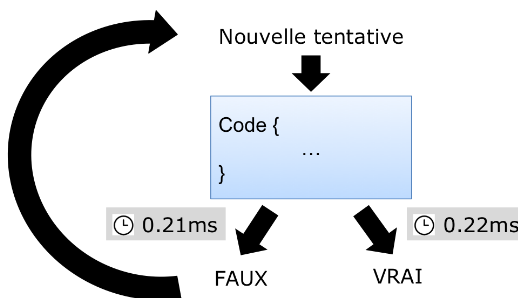

Figure 26. Xbox 360 : la timing attack

Il est donc possible **sans connaître la clé CPU** de forger un HMAC valide d'une combinaison `2BL` et LDV choisie. L'algorithme est le suivant :

1. En réalisant une image de la NAND, on récupère la valeur de LDV courante et cohérente avec le fuseset (que l'on ne peut évidemment pas changer) ;
2. On écrase dans le header du `2BL` en NAND le HMAC stocké (que l'on initialise avec des zéros) ainsi que le LDV récupéré lors de la

première étape. On écrase également tous les *bootloaders* (2BL, 4BL et 5BL) avec la version du noyau de base de la Xbox 360 (la *release* 1888).

3. Ensuite, pour chaque octet i de 0 à 15 du HMAC à forger, on mesure un écart dans le temps de démarrage de la console en faisant varier cet octet entre 0 et 255. Si le temps mis à démarrer est légèrement supérieur au temps moyen mesuré, c'est que l'on a deviné l'octet en question. On peut alors passer aux tentatives sur l'octet suivant $i + 1$.

Cet algorithme converge vers un HMAC valide avec un maximum de $16 \times 256 = 4096$ tentatives, et 2048 tentatives en moyenne. Il existe une forte probabilité de créer une collision entre l'octet à deviner et un octet aléatoire avec 128 tentatives. Il faut par ailleurs un mécanisme de synchronisation pour une mesure assez précise du temps.

L'analyse du circuit de la carte mère de la Xbox 360 met en évidence la présence du port de *Power On Self Test* (POST) (voir figure 27). Les valeurs codées sur 8 bits représentent l'état du système au cours des différentes étapes de démarrage. Les tests réalisés ont permis d'établir que la comparaison du HMAC était réalisée entre le POST 0x21 et les POST 0xA4 en cas d'échec de comparaison ou POST 0x22 en cas de succès.

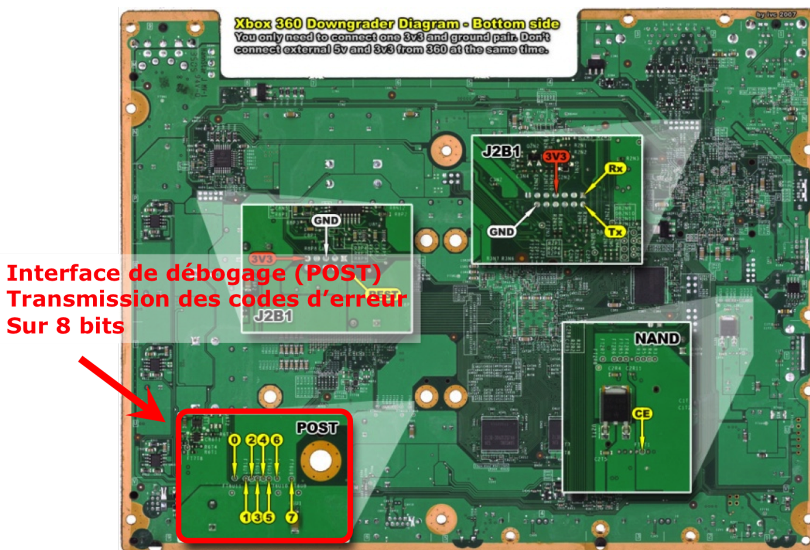

Figure 27. Xbox 360 : Identification du port POST sur la carte mère

Une fois un HMAC valide trouvé en `2BL` et LDV associé, il ne reste plus qu'à forger une image de NAND valide avec la chaîne `2BL`/`4BL`/noyau d'origine de la XBox 360 (la version 1888). Il suffit alors d'appliquer les mises à jour vers le noyau 4532 vulnérable à la *King Kong attack*, et enfin exploiter celle-ci.

La timing attack [35] a été implémentée en pratique dans des *downgraders*, kits matériels comportant un PIC relié à un programmateur de NAND (Infectus par exemple).

Xbox 360 *Une attaque a priori inutile : le `MfgBootLauncher`* 2007

Un peu avant la découverte de la *timing attack*, le *reverse engineering* du `2BL` a aussi permis de découvrir le mode **Zero-Pairing**. Ce mode semble principalement avoir été implémenté par Microsoft à des fins de test de consoles en sortie d'usine. Lorsque les données d'appairage du LDV du `2BL` sont à zéro (le Pairing Block du *header* précédemment décrit), celui-ci bascule en mode usine : il ne vérifie pas le HMAC de son LDV avec le *fuseset* 02. Par ailleurs, le `2BL` lève un *flag* pour que le `4BL` charge la version minimaliste de base de l'hyperviseur et du noyau (`5BL` en version 1888) **sans appliquer les possibles correctifs** présents en `7BL` ; le `6BL` n'est pas exécuté. La version de base charge ensuite un binaire de test en lieu et place du *dashboard* usuel : le `MfgBootLauncher`, celui-ci mettant la console dans un mode de « self-test » (Mfg étant certainement l'acronyme de *Manufacturing*). Les éléments d'appairage ne sont pas vérifiés dans ce mode car la console est supposée être dans un état non personnalisé (les *fusesets* n'ont pas encore été initialisés).

Le mode `MfgBootLauncher` est inexploitable pour deux raisons :

— Le binaire `MfgBootLauncher` ne présente pas grand intérêt pour la scène *homebrew* (le mode usine débridé qu'il laisse entrevoir ne permet pas de faire grand-chose) ;

— Le fait de lancer la version de base 1888 sans possibilité de mise à jour vers 4532/4548 est inutile, car cette version n'est pas vulnérable à la *King Kong attack*.

Le *Zero-Pairing* est donc resté longtemps inexploité par la communauté. Il a néanmoins été fort utile, comme nous le verrons, pour l'attaque SMC/JTAG apparue en 2009.

Xbox 360 *La course au downgrade continue : l'attaque SMC/JTAG* 2009

La *timing attack* découverte en 2007 a été la première exploitation de vulnérabilité à grande échelle pour cette console (la *King Kong attack* avait été corrigée avant sa publication). Elle concerne principalement la première

révision (Xenon) de la Xbox 360. Microsoft a patché la timing attack en implémentant logiquement une fonction `memdiff` de **comparaison à temps constant** pour le HMAC dans le `2BL` et le `6BL`. Notons néanmoins que du fait de l'existence même de la faille dans ces *bootloaders*, une simple mise à jour logicielle ne suffit pas puisque le *downgrade* implique le rejeu et l'exploitation des dits *bootloaders* vulnérables !

Microsoft a donc par ailleurs réagi contre la *timing attack* et la découverte du *Zero-Pairing* grâce à trois éléments :

— L'utilisation de mises à jour logicielles qui utilisent le *fuseset* 02 et la version minimale du `2BL` vérifiée par celui-ci, afin d'empêcher la réinstallation d'une chaîne de *bootloaders* vulnérables ;

— La mise à jour de sa plateforme matérielle, notamment de la carte mère (avec Zephyr, Falcon, Opus et Jasper entre 2007 et 2009) qui reposent sur des *bootloaders* corrigés donc non *downgradables* d'origine (car non compatibles matériellement) ;

— Microsoft a changé la manière dont le `4BL` est déchiffré à partir du `2BL` en version 1920. À l'installation d'une mise à jour, le `2BL` **utilise la clé CPU** pour dériver une clé de (dé)chiffrement du `4BL` stocké en flash. Ainsi, le *downgrade* entre versions de `2BL` à partir de ce point de mise à jour nécessite de garder des *snapshots* de sa NAND ou de connaître sa clé CPU. Cela permet de rendre les images NAND des *bootloaders* personnalisées par console, et non partageables entre attaquants (alors que ce n'était le cas que pour les Pairing data jusqu'ici). La chaîne de démarrage présente en NAND devient dépendante (à partir du `4BL`) de la clé CPU.

Notons que le chiffrement du `4BL` par la clé CPU à partir de la version 1920 est surtout une réponse de Microsoft à la découverte du Zero-Pairing et du `MgfBootLauncher` (et non spécifiquement à la *timing attack*). La mise à zéro du bloc d'appairage dans le *header* du `2BL` bascule toujours en mode Zero-Pairing, mais le `2BL` n'utilise pas la clé CPU pour déchiffrer le `4BL` (il bascule sur l'ancien mode de démarrage où seule la clé de chiffrement embarquée dans le `2BL` déchiffre le `4BL`). De cette manière, Microsoft s'assure qu'un attaquant mettant à zéro ses données d'appairage ne pourra pas démarrer en mode `MfgBootLauncher` puisque son `4BL` en NAND sera mal déchiffré. Par ailleurs, les `4BL` de mise à jour (non encore chiffrés avec la clé CPU donc) ne sont pas accessibles à l'attaquant tant qu'il n'a pas attaqué une console. On en revient à une situation de cercle vicieux similaire à celle de la *timing attack*…

La *timing attack* de 2007 a néanmoins eu un gros intérêt : elle a permis aux attaquants de récupérer des clés CPU de consoles vulnérables à cette

période. Cela leur a permis de déchiffrer toutes les mises à jour du `4BL` sur ces consoles. De manière surprenante, l'analyse du `4BL` en version 1920 (la première à utiliser la clé CPU) a notamment révélé une **modification importante dans la gestion du Zero-Pairing** : Microsoft, se croyant certainement à l'abri d'attaques sur le mode `MfgBootLauncher` grâce au chiffrement utilisant la clé CPU, a débridé la mise à jour en mode Zero-Pairing. En effet, dans ce mode le `4BL` exécute le `6BL` qui applique les correctifs présents au-dessus de la version de base 1888 si les données d'appairage de ceux-ci sont à zéro aussi (i.e. l'appairage des LDV du `6BL`). L'intérêt d'appliquer les mises à jour pour Microsoft réside certainement dans la possibilité d'effectuer en usine des tests poussés sur différentes versions de l'OS avec des versions non personnalisées des consoles.

Ce changement de comportement dans le *Zero-Pairing* fut une erreur fatale pour Microsoft, car c'est ce qui a permis aux attaquants de créer un kit de *downgrade* générique qui fonctionne sur toutes les révisions de consoles de l'époque. Le plus gros challenge des attaquants a été, la création d'une chaîne de démarrage vérifiant le Zero-Pairing pour toutes les variantes de consoles, ce qui s'est révélé possible grâce à d'ingénieuses astuces.

Production d'une chaîne de démarrage vérifiant le Zero-Pairing :
A l'époque de la découverte du changement de comportement du *Zero-Pairing*, le panorama des révisions de consoles était le suivant :

- La Xenon, première révision de la console, sur laquelle avait été exploitée la timing attack. Celle-ci a été corrigée avec la révision 1921 du `2BL`, le nouveau comportement du *Zero-Pairing* ayant été introduit avec la révision 1920 du `2BL` ;
- Les autres révisions matérielles de la console, à savoir la Zephyr, la Falcon, l'Opus et la Jasper. Ces révisions sont sorties directement avec des `2BL` non vulnérables à la timing attack et qui utilisent la clé CPU pour déchiffrer le 4BL. D'autre part, vu les divergences matérielles des révisions, le code du `2BL` n'est pas exactement le même entre ces consoles (ainsi qu'avec la Xenon).

Les attaquants ont donc mené leur attaque de deux manières différentes en fonction de la révision de console concernée :

1. **Attaque de la Xenon :** il est relativement aisé d'exploiter le *Zero-Pairing* sur Xenon. Il suffit de prendre le `2BL` en version 1920 ou 1921, de mettre à zéro le Pairing Block en NAND, et de chiffrer le `4BL` associé avec l'ancienne méthode (qui utilise exclusivement la clé embarquée dans le `2BL` pour déchiffrer le `4BL`). Cela fut possible

car les attaquants avaient le code du 2BL grâce à la clé du 1BL, et le code du 4BL déchiffré grâce à la clé CPU sur les consoles exploitées. Grâce au *Zero-Pairing*, le MfgBootLauncher se lance avec application des mises à jour sans vérification des *fusesets*. Par ailleurs, la vérification des versions révoquées du 2BL (via le *fuseset* 02), qui n'est pas évitée par le *Zero-Pairing*, n'est pas bloquante puisque le 2BL 1921 utilisé est celui qui est le plus à jour sur la Xenon.

2. **Attaque de Zephyr, Falcon, Opus, Jasper :** sur les consoles plus récentes, l'exploitation du *Zero-Pairing* s'avère plus complexe. En effet, le 4BL étant chiffré avec la clé CPU, il n'a pas été possible d'en récupérer le code pour le chiffrer avec l'ancienne méthode. Pour mémoire, cette étape est nécessaire pour l'exploitation du *Zero-Pairing*. Les attaquants avaient malgré tout à disposition tous les 2BL en clair de toutes les révisions des consoles (tous les 2BL étant chiffrés avec la même clé connue du 1BL en ROM). Pour chaque révision de console, ils ont comparé les différences de code entre le 2BL 1920 et le 2BL de la révision matérielle associée (par exemple 4558 pour Zephyr, 5770 pour Falcon et Opus, 6712 pour Jasper). Les différences étaient mineures et notamment liées à la timing attack : nouvelle implémentation du memcpy, et une mise à jour des numéros de versions. Leur idée fut de générer, pour chaque 2BL, le 4BL associé **verifiant le condensat SHA-1** présent dans le 2BL **en portant dans le binaire du 4BL 1921** (dont ils avaient le clair) les modifications observées entre les 2BL [1]. Les attaquants se sont aussi aidés de la taille des 4BL chiffrés visés, qui font la même taille que les 4BL déchiffrés (du fait de l'utilisation du RC4), pour confirmer ces différences binaires [2].

Le résultat fut qu'en août 2009, les attaquants avaient une chaîne de démarrage *downgradée* dédiée et complète pour chaque révision de console de l'époque.

Utilisation du SMC/JTAG comme support de l'attaque : La mise en place « manuelle » et naïve de l'attaque qui tire parti du *Zero-Pairing* est en fait laborieuse. Le noyau 4532 vulnérable à la *King Kong attack* est par exemple mal supporté par Jasper (il provoque un

2. Pour être complet, l'extrapolation du code pour la version 6712 du 4BL de Jasper ne fut pas simple. Ce 4BL est en effet le premier à intégrer une modification plus « profonde » du code en ajoutant le support des mémoires flashs à larges blocs. Il n'est donc pas possible de deviner ce code, et les sources publiques ne donnent pas de détails sur l'astuce utilisée par les attaquants.

« *kernel panic* » lorsque le GPU est sollicité) du fait d'un changement majeur de GPU. Dès lors, il semble difficile, voire impossible, de gérer l'exploitation à la main en insérant le DVD du jeu King Kong contenant le *payload* comme c'était le cas sur la timing attack.

Les attaquants ont donc mis au point une automatisation [15] de l'exploitation de la vulnérabilité à chaque démarrage de la Xbox 360 [27]. Le vecteur principal de la King Kong attack originale, et qui en fait source d'instabilité, est le DMA depuis le GPU. Les attaquants ont donc trouvé un moyen de mener la même attaque DMA depuis un autre périphérique qui en avait les capacités : **le contrôleur de flash NAND**. Celui-ci se trouve sur le bus PCI et ses registres sont mappés en `0xEA00C000` dans l'espace d'adressage physique du bridge PCI :

```
...
0xEA00C008: Operation (read DMA/write DMA)
0xEA00C00C: Adresse source/destination dans la NAND
0xEA00C01C: Adresse RAM source/destination de la page
0xEA00C020: Adresse RAM source/destination de l'ECC
...
```

Le type d'opération décrit une opération NAND, un transfert de la NAND vers la RAM (`read`) ou de la RAM vers la NAND (`write`) dans le cas particulier du DMA. Le transfert se fait par unité de page NAND de 512 octets, et les données de correction d'erreur ECC font 16 octets par page. Le contrôleur permet de transférer les données d'une page NAND et l'ECC à/depuis des adresses en RAM différentes.

Le contrôleur NAND est par ailleurs parfait pour une exploitation de la console puisqu'il permet de tirer parti d'un stockage du *payload* d'exploitation dans la NAND elle-même. Ce contrôleur est néanmoins passif en ce sens qu'un élément extérieur doit programmer ses registres pour qu'un accès DMA soit déclenché. Un bon candidat pour programmer le contrôleur de NAND est le SMC (*System Management Controller* [26]). Ce micro-contrôleur de type Intel 8051 est présent derrière le Southbridge sur le bus PCI et sert à gérer certains étages d'alimentation de la carte mère, les LED frontales, le RTC (*Real Time Clock*, horloge temps réel), les capteurs de température, les ventilateurs, ainsi que le chargeur de DVD-ROM. Le CPU communique avec le SMC via des FIFO mappées en mémoire sur le bridge PCI. Les commandes envoyées par le CPU servent notamment à récupérer l'heure, commander l'ouverture du chargeur de DVD, déclencher un `RROD`...

Mais plus intéressant : certains registres du contrôleur de NAND sont aussi mappés dans les SFR (*Special Function Registers*) du SMC. L'intérêt d'accéder à la NAND pour ce dernier réside dans le fait d'y lire son code

à exécuter (un premier *stage* présent en EEPROM interne au 8051 charge le second *stage* en NAND), ainsi que certains éléments de configuration et de calibration (des données thermiques pour la gestion des ventilateurs par exemple). Le SMC n'ayant pas de raison de provoquer des accès DMA entre la NAND et la RAM, les adresses de source et destination DMA `0xEA00C01C` et `0xEA00C020` ne sont pas mappées dans ses registres (le micro-contrôleur lit la NAND via des opérations de lecture lentes classiques de mémoire flash SPI par mots). Notons que le même registre du contrôleur de NAND `0xEA00C008` gère à la fois les opérations de lecture/écriture classiques et les opérations DMA (seul l'opcode de commande change). De même, le registre `0xEA00C00C` sert aux opérations classiques et au DMA. Il en résulte que le SMC peut **provoquer** une transaction DMA en mettant l'ordre adéquat en `0xEA00C008` avec une adresse maîtrisée en NAND, mais **sans maîtriser** les adresses source/destination en RAM (pour la page et l'ECC).

C'est là qu'intervient un dernier élément exploité par les attaquants : le JTAG du GPU. L'analyse de celui-ci a exposé des ordres permettant de faire exécuter au GPU des commandes vers le bus PCI. Il est donc possible d'adresser le contrôleur de NAND avec ces ordres. Alors pourquoi ne pas utiliser directement le JTAG GPU comme maître provoquant les transactions DMA de la NAND (elle aussi sur le bus PCI) ? Les choses ne sont en fait pas aussi simples : ce JTAG est désactivé par le `2BL` assez tôt durant le démarrage pour des raisons évidentes de sécurité, et il n'est plus disponible lorsque le *payload* doit être copié en RAM. L'idée est donc d'utiliser le JTAG tôt durant la phase de démarrage (avant sa désactivation par le `2BL`), pour programmer les registres d'adresses de destination DMA du contrôleur de NAND. De cette manière, le SMC peut déclencher le DMA depuis une adresse maîtrisée en NAND et vers une adresse maîtrisée via le JTAG au moment adéquat. Pour que cette technique fonctionne, il est impératif qu'aucun autre composant ne programme le contrôleur de NAND pour faire des accès DMA. Ceci est le cas dans la fenêtre d'exploitation concernée.

Il ne manque plus que deux éléments notables pour mener à bien l'attaque :

— La synchronisation : il est nécessaire de détecter le bon moment du côté du SMC pour déclencher le transfert DMA. Lors du démarrage, le noyau fait une requête au SMC pour récupérer l'heure sur le RTC. Cette requête est donc utilisée pour la synchronisation et le déclenchement de la *King Kong attack* ;

— L'accès et le contrôle des ordres JTAG : il faut câbler le JTAG du GPU sur la carte mère et envoyer les bonnes commandes avant que celui-ci ne soit désactivé. Le SMC dispose de suffisamment de GPIO pour câbler le JTAG (une adaptation de la tension : 3.3V vers 1.8V est néanmoins nécessaire) il est donc utilisé pour envoyer les ordres JTAG au GPU. Cette fois, la synchronisation est réalisée par la détection sur le bus JTAG d'un `IDCODE` signifiant que le GPU est démarré et répond aux commandes. Lorsque cet `IDCODE` est identifié, le SMC lance les ordres suivants (`write32(a, b)` écrit la valeur `b` à l'adresse `a`) :

```
write32(0xd0000018, 0x20100); //activation du bridge PCI
write32(0xd0140010, 0xea00c000); //activation du BAR0: le
    controleur NAND (PCI space en 0xEA00C000)
write32(0xd0140004, 0x6); //activer la NAND et le
    busmastering du controleur
write32(0xea00c00c, PAYLOAD); //addresse de l'exploit en
    flash
write32(0xea00c01c, 0x00130360); //adresse cible du contexte
    de thread idle
write32(0xea00c020, 0x00002080); //address cible de la table
    de syscalls de l'hyperviseur (ECC)
```

Nous ne l'avions pas précisé jusqu'ici, mais parmis les tâches du `2BL` il y a la vérification de l'intégrité du code en NAND du SMC via un HMAC-SHA-1 réalisé avec la clé CPU (comme pour les LDV). Le *Zero-Pairing* permet néanmoins de désactiver cette vérification.

Résumé de l'attaque :
Les étapes de l'attaque SMC/JTAG sont les suivantes :

1. L'attaquant soude les GPIO du SMC sur le JTAG du GPU ;
2. En fonction de la révision de console visée, une image flash comprenant la bonne chaîne `2BL`/`4BL`/noyau est choisie, les données de *Zero-Pairing* étant forcées dans les *headers* des *bootloaders* associés. Le script [15] permet de les générer. Le *layout* de la mémoire flash ainsi composée est le suivant :

```
00000000..00100000: SMC, KeyVault, CB, CD, CE, CF, CG,
    Bootloader post-exploit de backup
00100000..00140000: Bootloader post-exploit
00140000..00f7c000: Padding
00f7c000          : Bloc de configuration du SMC
00ffc000          : Buffer d exploitation
```

Le code du SMC est modifié afin d'intégrer la configuration JTAG et l'exécution du transfert DMA à la réception de la requête RTC

par le noyau. Le *buffer* d'exploitation contient le *payload* de la *King Kong attack* en deux morceaux : d'une part le contexte du *thread idle* contenu dans une page de NAND, de l'autre le contenu de la table des *syscalls* de l'hyperviseur dans l'ECC associé. Les deux écritures DMA nécessaires à la King Kong attack se font donc en une unique requête DMA du contrôleur NAND. Le *bootloader* post-exploit peut être adapté, mais de base il s'agit de XeLL.

Réaction de Microsoft :
La réaction de Microsoft face à la publication de l'attaque SMC/JTAG fut une mise à jour massive sur **toutes les consoles** [2] de toute la chaîne de *bootloaders* à partir du `2BL` (la mise à jour 849x) en août 2009 (le même mois que l'attaque). Le principal élément modifié fut la suppression du mode `MfgBootLauncher` en *Zero-Pairing* tel qu'il était utilisé dans l'attaque, et surtout l'utilisation du *fuseset* 02 afin que la vérification de version minimale du `2BL` échoue lors d'une tentative de *downgrade*. Il est intéressant de noter que cette mise à jour, la première ayant une telle envergure, ne fut pas anodine : environ une console sur mille fut *brickée*.

Xbox 360 Le Reset Glitch Hack (RGH) 2011
Suite à l'attaque SMC/JTAG, et au-delà de la mise à jour massive d'août 2009, Microsoft a réagi de diverses manières afin d'éradiquer afin d'éradiquer les attaques par *downgrade* :

- Les mises à jour ont continué à exploiter le *fuseset* 02 (version minium du `2BL`) ;
- Microsoft a **mis en dur dans les** `4BL` des nouvelles révisions de consoles les versions de noyau vulnérables à la King Kong attack (4532/4548), de sorte à ne **jamais exécuter ces noyaux** ;
- Microsoft a changé la manière dont le `2BL` est déchiffré sur les nouvelles révisions matérielles de consoles (dites « slim », correspondant aux révisions Trinity/Valhala). Le `2BL` est découpé en deux sous-parties : le `2BL-A` et le `2BL-B` (le premier déchiffre, vérifie l'intégrité et charge le second). Le `2BL-B` charge ensuite le `4BL` comme avant. La différence majeure réside dans le fait que `2BL-A` utilise la clé CPU pour déchiffrer `2BL-B` (lors d'une mise à jour, le CPU chiffre le `2BL-B` comme il faut avant le stockage en NAND, comme cela est fait pour le `4BL`). Ainsi, il devient encore plus complexe de *downgrader* un `2BL` puisque celui-ci dépend aussi de la clé CPU ;
- Enfin, l'évolution matérielle des nouvelles révisions de consoles rend assez complexe l'exploitation de *downgrade* sur les nouvelles

consoles : les anciens `2BL`/`4BL` contiennent du code qui peut se terminer de façon inopinée sur les nouvelles révisions (du fait de divergences parfois pas si mineures des plateformes matérielles).

La communauté s'est donc retrouvée pendant plusieurs années (de 2009 à 2011) sans nouvelle attaque à exploiter sur les nouvelles révisions de consoles ou sur les anciennes consoles mises à jour post 849x. C'est alors qu'en désespoir de cause, des attaquants tentèrent une attaque matérielle longtemps discutée et réputée trop complexe : la *glitch attack* [3].

Principe de l'attaque par glitch :
L'idée est assez simple : lors démarrage sécurisé, le `2BL` déchiffre le `4BL` et en vérifie l'intégrité via un SHA-1. Un `memcmp` est alors utilisé pour comparer le condensat calculé et le condensat du `4BL` stocké en dur dans le `2BL`. Si la comparaison échoue, le `2BL` arrête l'exécution. Sinon, il passe la main au `4BL`. La comparaison prend la forme d'une instruction de branchement conditionnelle `beq` qui effectue un branchement si le résultat du `memcmp` est nul. Ainsi, une faute (ou *glitch*) insérée au bon moment peut forcer ce branchement. L'injection de faute s'avérer très complexe sur un CPU dont l'horloge est à 3.2 GHz, l'astuce consiste à le ralentir pour faciliter l'insertion d'une impulsion sur la piste de `reset` du CPU (voir figure 28). L'effet du *glitch* est assez surprenant, puisqu'au lieu de redémarrer, le CPU continue d'exécuter du code, mais avec certains registres **remis à zéro**, ce qui simule un résultat à zéro du `memcmp` et une **exécution du `4BL` malgré un mauvais condensat**. Il est ainsi possible d'utiliser un `4BL` non intègre maîtrisé par l'attaquant. L'exécution de celui-ci est alors réalisé avec le niveau de privilèges le plus élevé de la machine (l'exploitation se fait très tôt durant le démarrage, avant l'activation d'autres protections). Une fois l'attaque réussie, le CPU reprend une fréquence normale.

Le *glitch* en lui-même doit être assez précis, et nécessite l'utilisation d'un composant externe de type FPGA ; (un CPLD Xilinx CoolRunner II est utilisé dans la preuve de concept). De même que pour la *timing attack*, il est impératif de se synchroniser avec le code pour envoyer l'impulsion au bon moment. Ce sont encore les transitions de `POST` qui sont utilisées (Microsoft ne l'a pas supprimé malgré la timing attack, car celui-ci sert au diagnostic technique lors des retours de consoles non fonctionnelles).

Le *glitch* étant par nature assez aléatoire, le SMC est utilisé via une modification de son code pour détecter une réussite ou non, et redémarrer indéfiniment jusqu'à réussite de l'attaque.

Les détails de l'attaque divergent entre les révisions « fat » (aussi nommées « phat » : Xenon, Zephyr, Falcon/Opus, Jasper) et « slim »

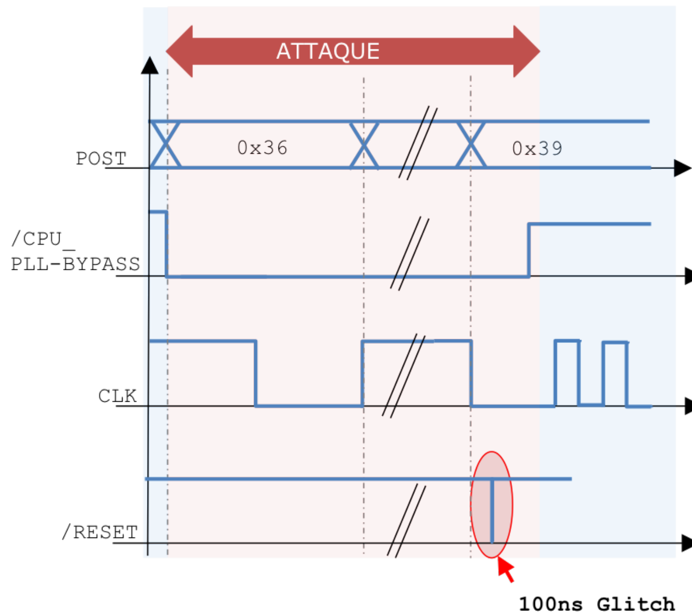

Figure 28. Xbox 360 : chronograme de l'attaque par *glitch*

(Trinity/Valhala et Corona) de Xbox 360. Nous donnons ci-après des explications précises sur ces divergences.

Attaques sur les consoles dites « fat » :
Sur les consoles fat, c'est le signal `CPU_PLL_BYPASS` découvert sur la carte mère qui permet au niveau bas de diviser l'horloge du CPU par 128. Il est inséré au `POST 0x36`. Le `POST 0x39` caractérise le début du `memcmp` du condensat du `4BL`, et le *glitch* est généré peu après. Le `CPU_PLL_BYPASS` est alors remonté pour remettre le CPU à sa fréquence d'origine.

Un problème lié au `4BL` n'a pas été évoqué jusqu'ici : en effet, comme expliqué précédemment, le `4BL` a été chiffré avec la clé CPU après la timing attack. Malgré la neutralisation de la vérification d'intégrité, il faut donc pouvoir générer un `4BL` chiffré pour exploiter le *glitch*.

Les attaquants ont en fait réutilisé le *Zero-Pairing* : malgré la limitation du `MfgBootLauncher` suite à l'attaque SMC/JTAG, le mode *Zero-Pairing* a été laissé par Microsoft (toujours pour des raisons de validation matérielle des consoles en sortie d'usine). Pour rappel, dans ce mode le `2BL` n'utilise plus la clé CPU pour déchiffrer le `4BL`, mais bascule sur l'algorithme *legacy* qui utilise une clé dans le `2BL` (connue de manière indirecte via la clé `1BL`).

Il est dès lors possible, en mode *Zero-Pairing*, de chiffrer avec cette clé connue un 4BL modifié et maîtrisé. Le *Zero-Pairing* a une autre vertu : l'intégrité du code du SMC n'est pas vérifiée ce qui est utile puisque son code en NAND est modifié. L'attaque par *glitch* est ainsi complète et réussit en moyenne au bout de 30 secondes.

Attaques sur les consoles récentes dites « slim » :
Sur ces nouvelles révisions des cartes mères, l'attaque menée sur les consoles les plus anciennes (*fat*) pourrait parfaitement s'adapter : il suffit d'appliquer le *glitch* au moment où le 2BL-B vérifie le condensat du 4BL. Néanmoins, il y a un énorme intérêt à appliquer le *glitch* au 2BL-A lors de la vérification du condensat du 2BL. En effet, la données d'appairage et de révocation du 2BL sont vérifiées dans le 2BL-B, ce qui signifie qu'une exploitation de *glitch* avant son exécution implique une **impossibilité de patcher** cette vulnérabilité via le *fuseset* 02 !

Les attaquants ont dès lors relevé ce défi : le 2BL-A utilise la clé CPU pour déchiffrer le 2BL-B. Mais comment forger un 2BL-B valide qui exécute un *payload* sans avoir la clé CPU ? Le *Zero-Pairing* ne peut être utilisé ici car il est justement appliqué trop tard (par le 2BL-B). Ils ont astucieusement utilisé **la faiblesse de RC4** (des chiffrements à flot en général) qui permet de retrouver le *keystream* lorsque le clair est connu :

1. Le 2BL des versions *fat* sont disponibles en clair ;
2. Sur console *slim*, les 2BL-B sont chiffrés indirectement par la clé CPU avec l'algorithme RC4 : un keystream est généré et le 2BL-B chiffré correspond à <code2BL-B⊕keystream>.
3. Les attaquants ont deviné le début de code2BL-B en inférant qu'un 2BL de *fat* correspondait grossièrement à une concaténation de 2BL-A et 2BL-B de *slim*. Cela permet de retrouver le keystream via l'opération <(code2BL-B⊕keystream)⊕code2BL_connu>. Le keystream permet alors en théorie de chiffrer un 2BL-B maîtrisé, exploitable via le *glitch*.
4. Néanmoins, comme la partie commune du 2BL *fat* et du 2BL-B *slim* ne faisait que quelques octets, il n'était pas possible de forger un 2BL-B chiffré complet. Les attaquants ont donc astucieusement employé les quelques octets disponibles pour mettre un petit *payload* permettant de lire les *fusesets* et dumper la clé CPU, et par là même de récupérer un 2BL-B clair.
5. Enfin, une fois l'analyse du 2BL-B achevée, un *patch* binaire a été générée pour ajouter au 2BL-B un *payload* nécessaire à l'attaque post-*glitch* (à savoir charger un 4BL complètement maîtrisé sans en

vérifier l'intégrité ni le déchiffrer). Ce *patch* binaire est appliqué de manière générique à toutes les consoles via l'astuce de récupération du *keystream* via le `xor` avec le clair.

Microsoft utilise la vérification d'intégrité comme contremesure à la malléabilité inhérente au RC4, mais c'est cette vérification d'intégrité qui est justement visée par l'attaque...

Un autre élément diverge au niveau de l'exploitation des consoles « slim » par rapport aux consoles « fat » : la piste du `CPU_PLL_BYPASS` n'a pas été trouvée. La communauté a alors exploité le HANA : c'est un composant contenant des convertisseurs analogique/digital ainsi que des PLL permettant de configurer l'horloge du CPU et du GPU. Or les registres du HANA sont accessibles via un bus I^2C sur lequel le SMC se trouve. Le code du SMC est donc modifié pour envoyer une commande I^2C de diminution de fréquence CPU lors du `POST 0xD8`. Le SMC attend le `POST 0xDA`, moment du `memcmp` du condensat du `2BL-B`, pour déclencher le *glitch*. La fréquence d'origine est enfin restaurée via HANA.

Réaction de Microsoft :
Comme nous l'avons expliqué, sur les consoles « slim » l'attaque ne peut avoir de correction purement logicielle utilisant la révocation via le *fuseset* 02. Microsoft a d'ailleurs attendu assez longtemps pour patcher cette vulnérabilité puisqu'elle nécessitait une révision matérielle majeure : seule la version Winchester sortie en 2014 implémente une **protection matérielle** contre le *glitch* (la version Corona sortie juste après l'attaque n'en bénéficie pas, car son design matériel s'était fait trop tard).

Sur les consoles « fat », les choses sont légèrement plus complexes. La mise à jour 14719 du *dashboard* a tenté de corriger l'attaque par *glitch* en implémentant une comparaison `memcmp` plus robuste et en révoquant les `2BL` vulnérables via le *fuseset* 02. La communauté a néanmoins réussi à implémenter une nouvelle variante de l'attaque (dite RGH2) mettant à mal cette protection.

Xbox 360 : conclusions sur la sécurité

L'analyse de l'architecture et des attaques précédemment décrites sur la Xbox 360 est riche d'enseignements, notamment de par la rétrospective des quelque dix ans qui nous séparent de la conception de ce système.

Les éléments positifs de la Xbox 360, encore d'usage (voire d'avant-garde) aujourd'hui, sont les suivants :

— Une architecture logicielle fondée sur un hyperviseur **minimaliste** et **vérifié**, situé dans une RAM intègre et chiffrée ;

- Du `W⊕X` complet ;
- De la vérification de signature de tout code exécuté sur le CPU, sous le contrôle de l'hyperviseur ;
- Des clés embarquées dans le CPU ;
- Une chaîne de démarrage sécurisé ;
- Un système anti-*downgrade*.

Tous ces éléments n'ont malheureusement pas suffit à protéger la console contre les attaques. Rétrospectivement, voici quelques éléments qui ont été autant de fissures exploitées par les attaquants :

- Des protections très insuffisantes contre les attaques DMA : comme nous l'avons vu, le point d'entrée de la *King Kong attack* se fait depuis un périphérique pouvant faire des accès DMA. On pourrait penser que la faute incombe exclusivement au *shader* non signé du jeu King Kong, mais des variantes de cette attaque existent et utilisent du des accès DMA depuis le contrôleur NAND. Par ailleurs, le chiffrement de la mémoire n'est qu'une pâle protection contre ce type d'attaques. La raison réelle de cet échec réside dans la non-maturité des **technologies d'IOMMU de l'époque** (la virtualisation en était à ses débuts).
- Le système de démarrage sécurisé est intéressant pour l'époque, mais pêche par quelques éléments :
  - Le fait que le `1BL` soit accessible en lecture, avec une clé embarquée en dur, compromet la sécurité des mises à jour futures dès lors qu'une compromission a lieu. Une vulnérabilité en *bootROM* est très grave car non corrigeable. Apple l'a récemment compris, et a finalement rendu sa *bootROM* non lisible après le démarrage du système.
  - Le fait que les clés CPU soient en *eFuse* (et donc lisibles depuis du code exécuté) est une énorme lacune. La compromission sur une console donnée implique le contrôle de la console en question pendant toute la durée de vie de celle-ci. À titre de comparaison, les SoC modernes emploient des coprocesseurs avec des clés embarquées non lisibles ou extractibles par d'autres périphériques ou le CPU.
  - Certaines primitives cryptographiques utilisées par Microsoft présentent des faiblesses qui ont été exploitées (HMAC en temps non constant, utilisation du RC4 permettant une récupération du *keystream*). L'utilisation de primitives comme le chiffrement authentifié aurait pu éviter plusieurs attaques.

— Enfin, la Xbox 360 n'a clairement pas été pensée pour résister aux attaques par faute et autres attaques par canaux auxiliaires. Cela lui a été fatal (avec la timing attack et la *glitch* attack). Notons qu'en dehors du monde de la carte à puce et des composants sécurisés, les SoC modernes n'intègrent pas forcément non plus des contre-mesures contre ces attaques.

Malgré les éléments positifs mis en œuvre dans l'architecture de sécurité de la Xbox 360, Microsoft a payé cher les quelques lacunes présentes à l'origine dans son système puisque via le jeu des *downgrades*, une vulnérabilité a suffi à alimenter des exploits de plus en plus complexes pendants plusieurs années.

2.4 La sécurité sur une plateforme exotique : la Playsation 3

Dans tout le reste de la présente section, nous emploierons « PlayStation 3 » ou « PS3 » de manière équivalente pour parler de la troisième console de salon de Sony.

PlayStation 3 : architecture générale

PlayStation 3 Architecture matérielle

Tout comme Microsoft, Sony a choisi IBM comme partenaire pour la PlayStation 3. La plateforme résultante fut néanmoins plus exotique que celle de la Xbox 360. Bien qu'également fondée sur une architecture PowerPC 64 bits de type G5, le **Cell Broadband Engine** [19] (voir figure 29) est en fait bien plus qu'un simple CPU. C'est une nouvelle micro-architecture parallèle qui s'éloigne des PC standards (alors que la Xbox 360 peut être grossièrement vue comme un PC avec un CPU tri-cœurs). Le Cell est composé d'un PPE (*PowerPC Element*) et de sept SPE [3] (aussi nommés SPU pour *Synergistic Processor Element* ou *Unit*) :

— Le PPE est un CPU PowerPC 64-bit classique avec deux *threads*, une fréquence de 3.2 GHz, 32Ko de cache L1 d'instructions et de données, et 512Ko de cache L2 unifié. Comme pour la Xbox 360, le PPE supporte les trois modes hyperviseur, superviseur et utilisateur ;

— Les SPE sont des CPU vectoriels conçus pour des calculs parallèles de type SIMD (*Single Instruction Multiple Data*). Ils s'exécutent à

3. En fait huit SPE, dont un est désactivé sur le die afin d'optimiser le rendement des *yields*.

3.2 GHz, ont 128 registres génériques de 128 bits, et ont accès à 256 Ko de stockage local privatisé (qu'ils peuvent partager ou non avec les autres unités sur le bus mémoire, nous y reviendrons). Cette zone est une SRAM dénommée LS (pour *Local Storage*). Les SPE ne supportent qu'un mode d'exécution « utilisateur » : ils n'ont pas de MMU, mais une unité MFC (*Memory Flow Control*) qui leur permet de gérer leur LS. Les SPE ont un contrôleur DMA leur permettant d'échanger des données avec le reste du système.

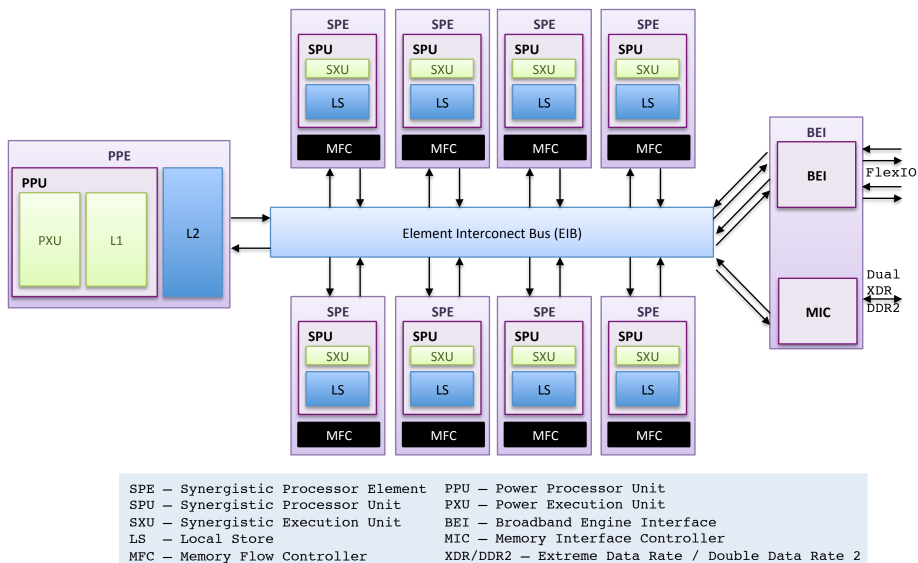

Figure 29. PS3 : architecture matérielle

La communication entre les SPE et les autres unités (dont le PPE) se fait via le bus EIB (*Element Interconnect Bus*), au travers de :
— Transferts DMA entre la RAM et les LS ;
— Mailbox entre le PPE et les SPE (chaque SPE dispose de deux mailbox de messages de 32 bits) ;
— Signaux envoyés via l'écriture dans des registres SPE spécifiques de 32 bits (accessibles depuis les autres périphériques).

Les autres éléments notables de l'architecture de la PlayStation 3 sont :
— Sa RAM de 256 Mo ;
— Son GPU, le RSX (un dérivé de la 7800GTX de NVIDIA) ;

— Sa mémoire flash qui stocke les *bootloaders* et l'OS. C'est de la NAND ou de la NOR en fonction des révisions de la console ;
— Son lecteur Blu-Ray ;
— Son disque dur.

PS3 Le mode SPE isolé, la clé CPU

IBM a introduit dans le Cell un nouveau paradigme permettant de faire du démarrage sécurisé et de l'exécution dynamique sécurisée de code. L'idée part de deux constats :

1. La plupart des technologies de *boot* sécurisé développées jusqu'ici se focalisaient sur les premiers instants du *boot* : l'intégrité du *firmware* n'étant assurée qu'au démarrage, toute exploitation post-*boot* rend cette vérification caduque. Une **vérification dynamique** de l'authenticité et de l'intégrité de modules chargés post-démarrage apporte alors un réel bénéfice ;
2. Il faut partir du principe qu'un attaquant aura accès au niveau de privilège le plus élevé du CPU sur lequel il exploitera une vulnérabilité. Il faut donc isoler le code critique sur des unités d'exécution qui restent intègres malgré la prise de contrôle d'autres unités.

IBM a donc conçu un mode spécifique des SPE nommé **mode d'isolation des SPE** [25] permettant de verrouiller une unité vis-à-vis de l'extérieur, à savoir le PPE ainsi que tout élément connecté sur le bus EIB. Dans ce mode particulier, le SPE bloque via son MFC les accès DMA à une grande partie de sa mémoire locale. L'espace mémoire de 256 Ko `0x0-0x40000` est alors segmenté en deux parties : la zone mémoire basse `0x0-0x3e000` est réservée au SPE, et la zone mémoire haute `0x3e000-0x40000` reste disponible sur le bus EIB. Cette dernière zone permet une discussion via des RPC à destination du SPE avec des arguments passés sur une pile dans cette zone.

C'est le PPE qui est « maître » et décide de faire passer le SPE en mode isolation via un registre accessible en mode PPE superviseur. Lorsque le SPE est en mode isolé, le PPE ne peut que le recharger avec un nouveau code ou arrêter le mode isolé, auquel cas la mémoire locale du SPE est effacée avant toute autre action pour éviter la fuite d'information ou de secrets.

Le PPE peut donc charger du code et le faire exécuter par le SPE depuis la mémoire locale de celui-ci. Si le SPE passe en mode isolation, le PPE n'aura plus accès à cette mémoire, même s'il est en mode privilégié

(mode hyperviseur par exemple). Sa seule action possible est l'arrêt et la reprogrammation du SPE qui s'est isolé.

Bien évidemment, le chargement de code SPE par le PPE n'a pas grand intérêt si l'authenticité, l'intégrité et la confidentialité de ce code ne sont pas assurées. Or, le PPE ne fait pas partie de la zone de confiance dans le modèle que nous considérons. La réponse d'IBM à ce problème a été l'utilisation d'une **graine de confiance matérielle** au sein du processeur Cell. Lorsqu'un SPE est chargé pour la première fois avec du code destiné à être exécuté en mode isolé, ce code est déchiffré et vérifié en intégrité (via un HMAC vraisemblablement) avant d'être exécuté.

C'est l'AES qui semble être utilisé, et la clé CPU ancrée dans le matériel repose sur la même technologie eFuse que celle des Xbox 360. Cette clé est **unique par console**.

PS3 Architecture logicielle

Les composants logiciels de la PlayStation 3 peuvent être catégorisés en quatre types :

1. Les **loaders**, qui interviennent lors du *boot* et lors du chargement d'autres composants jugés critiques : ceux-ci s'exécutent sur les SPE en mode isolation vu leur criticité (nous y revenons lorsque nous décrivons le *boot* sécurisé). On distingue :
   — Les *loaders* associés au démarrage qui sont chargés dans le SPE en mode isolation les uns à la suite des autres ;
   — Les *loaders* qui peuvent être appelés durant le cycle de fonctionnement nominal de la console, et qui sont exécutés ponctuellement sur un SPE. C'est par exemple le cas des applications qui gèrent le lecteur Blu-Ray, ou des applications qui s'occupent des déchiffrements de DRM lors de la lecture de fichiers audio ou vidéo protégés.
2. L'**hyperviseur**, qui s'exécute au niveau de privilège le plus élevé, et qui gère la MMU sur le PPE (par exemple la création de nouvelles tables de pages) ;
3. L'**OS**, qui s'exécute en mode superviseur. Sony a prévu à la sortie de la PlayStation 3 la possibilité de choisir au démarrage de la console d'exécuter un OS alternatif, nommé **OtherOS** par opposition à l'OS natif de Sony dédié au jeu nommé **GameOS**. L'intérêt du Cell était évidente pour la communauté scientifique, et le fait de pouvoir transformer la console de jeu en machine de calcul massivement parallèle a ouvert à Sony un marché jusqu'ici inexploité et insolite pour sa branche multimédia. De son côté IBM a joué le jeu en

mettant à disposition des documents techniques complets, des SDK ainsi que des *patchs* pour Linux [24] et autres FreeBSD permettant d'exécuter plusieurs distributions open source sur une PlayStation 3 de l'époque (Debian, Ubuntu...) ;

4. Les **applications**, qui s'exécutent en mode utilisateur sur le PPE. Les jeux font partie de cette catégorie (après qu'ils ont été déchiffrés par un *loader*).

Tout ce qui s'exécute sur le PPE, quel que soit le mode, est en RAM. Les éléments qui s'exécutent en mode SPE isolé sont cloisonnés dans le LS du SPE concerné comme précédemment expliqué.

Le paquetage formé par les *loaders* (hormis les deux *loaders* de plus bas niveau), l'hyperviseur, le GameOS et les applications natives (lecteur de vidéo, lecteur de musique...) de la console est nommé **CoreOS**. Il est stocké dans la mémoire flash de la console, et correspond au « *firmware* ».

PS3 *EID, Chiffrement de disque, protection Blu-Ray*

EID et pairing par console :
Les EID (*Encrypted Individual Data*) sont une zone de données en flash chiffrées par la clé CPU et contenant des données propres à chaque console, qui servent indirectement à l'appairage d'éléments comme le disque dur et le lecteur Blu-Ray.

Chiffrement de disque dur :
Tout comme la Xbox, la PlayStation 3 contient un disque dur. Celui-ci sert principalement de zone de stockage des données annexes de CoreOS (typiquement les images de thème de la console), des sauvegardes de jeu, de données multimédia (musique, films) ainsi que des jeux (téléchargés *online*).

Afin d'éviter l'utilisation de ce disque comme vecteur d'attaque, Sony fait appel à l'algorithme de chiffrement AES-CBC sur les anciennes révisions, et d'AES-XTS sur les nouvelles (ce qui a amené des vulnérabilités sur le chiffrement de disque des anciennes consoles). Une unité de chiffrement/-déchiffrement (ENCDEC) se situe dans le Southbridge afin d'augmenter les performances. Cette unité est chargée avec des clés AES lors du démarrage de la console par un des *loaders*.

Il faut de noter que pour éviter qu'une unité malveillante ne fasse une attaque *Man in the Middle* sur le bus EIB partagé, le *loader* et ENCDEC établissent un canal sécurisé à partir de clés AES pré-partagées (ils en dérivent des clés de session).

Les clés de chiffrement de disque sont diversifiées par console (dérivée du contenu des EID) afin d'empêcher l'échange de disques entre consoles.

Protection du lecteur Blu-Ray :
De la même manière que pour le disque dur, le lecteur Blu-Ray est appairé à la console. Un *loader* dédié exécuté sur un SPE isolé établit un canal chiffré via une clé AES partagée avec le *firmware*, cette clé étant dérivée des EID.

Cela a pour conséquence de ne pas pouvoir récupérer le *firmware* du lecteur Blu-Ray pour l'analyser (car celui-ci sera chiffré), et de ne pas pouvoir échanger un lecteur entre deux consoles.

Au-delà de la protection apportée par Sony au niveau du lecteur, le chiffrement standard AACS des disques Blu-Ray est utilisé ainsi que la technologie anti-copie BD ROM-Mark. Ces deux éléments rendent difficiles la contrefaçon par clonage de disques. Nous ne donnons pas plus de détails sur ces technologies, le lecteur intéressé peut se référer à [31,32].

PS3 *Boot et chargement dynamique sécurisé*

Le démarrage de la PlayStation 3 utilise en grande partie le mode isolé des SPE comme éléments de sécurité matérielle. Trois types de *bootloaders* peuvent être distingués :

— Le *bootloader* de niveau zéro, ou **secure bootloader**, est celui ancré dans la *bootROM* du Cell. C'est le seul à pouvoir accéder à la clé CPU en eFuse, et c'est lui qui déchiffre et vérifie l'intégrité d'un *bootloader* de premier niveau chargé dans un SPE et mis en mode isolation. C'est la racine de confiance matérielle et la base de la sécurité de la PS3. Notons que contrairement au cas de la Xbox 360, la *bootROM* du Cell et sa clé CPU ne sont **jamais accessibles** depuis du code exécuté, et ce quel que soit le niveau de privilèges ou l'unité d'exécution (PPE et SPE).

— Les *bootloaders* de premier niveau : `bootldr` et `metldr`. Ils sont présents en flash et chiffrés avec la clé CPU. Ce processus de chiffrement se fait en usine, lorsque la clé CPU est générée en eFuse. Pour cette raison, ces deux *bootloaders* **ne peuvent être mis à jour** sur une console produite. Ces *bootloaders* sont exécutés en mode SPE isolé, ce sont les seuls à pouvoir *bootstraper* un SPE grâce à leur chiffrement par la clé CPU.

— Les *bootladers* de second niveau : `lv0`, `lv1ldr`, `lv2ldr`, `appldr`, `isoldr`. Ces *bootloaders* sont chargés par les *bootloaders* de premier niveau dans les SPE en mode isolés, et s'occupent de charger le reste du système (l'hyperviseur, le noyau du GameOS...). Nous

donnons ci-après le détail de ces chargements. Ces *bootloaders* sont stockés en flash avec le CoreOS.

Afin d'assurer une chaîne de confiance de bout en bout, chaque maillon de la chaîne de *boot* déchiffre et vérifie l'authenticité du maillon suivant. La *bootROM* fait cela via la racine de confiance matérielle. Tout le reste utilise de la cryptographie logicielle. Chaque *bootloader* au-delà de la *bootROM* contient :

— Une clé AES de 256 bits que nous nommerons `*_erk` par la suite, où le `*` sera remplacé par le nom du *bootloader* concerné (`erk` signifie *Encryption Round Key*). Par exemple `metldr_erk` est la clé contenue dans le `metldr` ;
— Un IV nommé `*_riv` (pour *Reset IV*) ;
— Une clé **publique** ECDSA que nous nommerons `*_ecdsa_pub`.

Ces éléments sont utilisés pour charger les fichiers binaires à exécuter, qui sont en fait des fichiers au format ELF modifiés par Sony pour intégrer le chiffrement et la signature.

Format SELF et cryptographie :
Le format SELF (pour Signed ELF) encapsule un ELF chiffré auquel sont apposées des métadonnées permettant de le déchiffrer et d'en vérifier la signature. Les SELF sont prévus pour être chargés avec des *loaders* qui contiennent une clé de déchiffrement, et une clé publique pour vérifier la signature. La structure d'un SELF est la suivante [14] (voir figure 30) :

— Le SCE *header* : cet entête sert à encoder des informations de typage du SELF, et quelques informations liées à taille que le *loader* utilisera. Ces informations sont en clair ;
— Des métadonnées qui contiennent la clé de chiffrement du SELF ainsi qu'un IV associé. Le déchiffrement se fait avec de l'AES-128 en mode CTR. Ces métadonnées sont elles-mêmes chiffrées avec la clé AES du *loader* ;
— Des métadonnées qui contiennent un ensemble de condensats SHA-1 des segments et sections du ELF encapsulé. Ces métadonnées sont chiffrées avec la clé AES-128 du SELF ;
— La signature ECDSA, qui couvre principalement les éléments précédents ;
— Le fichier ELF chiffré.

Ainsi, lorsqu'un *loader* `x` charge un SELF, il déchiffre les métadonnées avec sa clé `x_erk` et son IV `x_riv` avec l'AES-256 en mode CBC. Il en extrait la clé AES-128 ainsi que l'IV, déchiffre les métadonnées qui contiennent les condensats et la signature avec l'AES-128 CTR. Il vérifie enfin la signature ECDSA avec `x_ecdsa_pub`. Il déchiffre et *parse* ensuite

le fichier ELF, vérifie ses condensats SHA-1 issus des métadonnées, le charge à l'adresse spécifiée dans le *program header* et saute sur le point d'entrée.

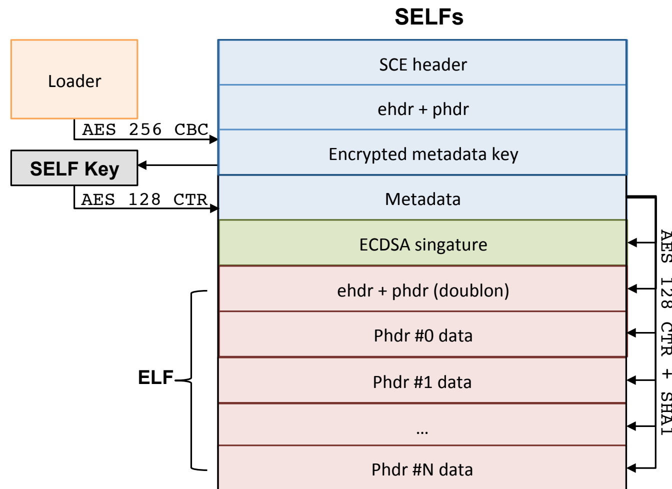

Figure 30. PS3 : anatomie d'un Signed ELF

Chaîne de *boot* complète :
Les fichiers SELF peuvent être déchiffrés et chargés en RAM ou directement dans les SPE en fonction de leur type. Ainsi, les *loaders* `lv1ldr`, `lv2ldr`, `appldr`, `isoldr` sont des SELF chargés et exécutés dans les SPE. Les *bootloaders* et applications `lv0`, `lv1`, `lv2`... sont des SELF chargés en RAM et exécutés sur le PPE.

Notons que le fait qu'un *loader* qui s'exécute dans un SPE isolé charge un SELF explique l'utilisation de multiples condensats lors de la vérification du ELF : vu le peu de mémoire disponible dans le LS, une vérification et un chargement fragmentés permettent d'éviter les attaques de type TOCTOU (*Time Of Check Time Of Use*).

Notons que les *bootloaders* de premier niveau `bootldr` et `metldr` ne sont pas des SELF. Ce sont des binaires « bruts » chargés dans les SPE, et déchiffrés par la *bootROM*.

La chaîne de *boot* de la PS3 est la suivante (voir la figure 31) :

1. La *bootROM* charge, vérifie, déchiffre et exécute le `bootldr` qui se situe à l'adresse 0 en flash. Celui-ci est exécuté sur le SPE0 ;

2. `bootldr` charge, déchiffre, vérifie le `lv0` et l'exécute sur le PPE. Ce *bootloader* très bas niveau initialise le matériel ;
3. Le `lv0` charge ensuite le `metldr` dans le SPE2 en mode isolation. Comme nous l'avons précisé, le `metldr` n'est pas un SELF. Il est déchiffré et vérifié par la *bootROM*. Il porte ce nom car c'est un **méta-loader**, qui va charger tous les autres *loaders* ;
4. Le `lv0` fait charger au `metldr` le SELF `lv1ldr` qui s'exécute dans le SPE2 ;
5. Le `lv1ldr` charge le SELF `lv1` pour qu'il s'exécute sur le PPE. Le `lv1` est le binaire de **l'hyperviseur**.
6. Après avoir fait quelques initialisations, le `lv1` passe en mode superviseur, recharge le `metldr` dans le SPE2 et lui fait charger `lv2ldr`. Ce dernier charge le SELF `lv2` sur le PPE qui correspond au **GameOS** (qui s'exécute donc en mode superviseur).

Nous n'entrerons pas plus dans les détails, mais la même logique est employée pour charger les SELF `appldr` et `isoldr` (via le `mtldr`) qui se chargent entre autres d'exécuter les jeux et d'autres applications sur le PPE. Les jeux sont en l'occurrence déchiffrés par `appldr` et exécutés sur le PPE (en mode utilisateur).

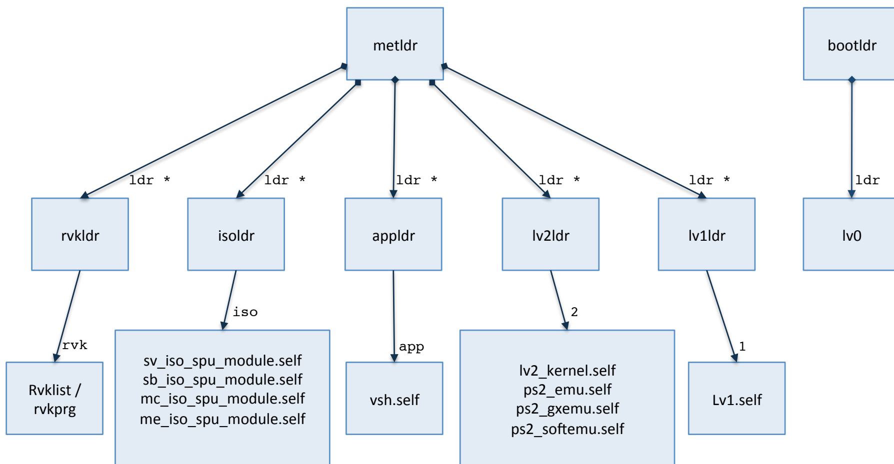

Figure 31. PS3 : chaîne de *boot* sécurisé

En résumé, la chaîne de démarrage développée par Sony pour la PS3 utilise de manière centrale le mode isolé des SPU via le `metldr`, et la

propagation de la chaîne de confiance se fait via la vérification et le déchiffrement des SELF chargés.

PS3 Protection anti-downgrade

Il faut distinguer deux éléments lorsqu'il s'agit de mise à jour et de protection anti-*downgrade* : les éléments qui peuvent être mis à jour, et les éléments qui peuvent être révoqués.

Sur la PS3, deux éléments ne peuvent être mis à jour (et sont *a fortiori* non révocables) : le `bootldr` ainsi que le `metldr`. C'est le fait qu'ils sont chiffrés par la clé CPU qui l'implique (contrairement au cas de la Xbox 360, la clé CPU n'est pas directement accessible au logiciel qui s'exécute sur les CPU). Hormis ces deux éléments, tout le reste peut être mis à jour (et fait d'ailleurs partie du « package » CoreOS).

La PS3 utilise une liste de révocation de logiciels stockée en flash (dans le CoreOS, voir le tableau 3). Le `lv2ldr` vérifie l'authenticité de cette liste via sa signature ECDSA, et le `lv2` la *parse* pour arrêter le processus de démarrage si un élément révoqué est détecté.

Remarquons trois éléments qui limitent fortement ce système :
- La moindre vulnérabilité dans les *bootloaders* non révocables peut être fatale ;
- Il n'y a pas vraiment d'ancrage matériel de la révocation (contrairement aux *fusesets* de la Xbox 360) : un rejeu de l'état de la flash ne peut pas être détecté ;
- Le fait que le `lv2` vérifie la liste de révocation arrive tard durant le *boot*.

Élément	CPU/Mode	MàJ	Révocation	Cryptographie/déchiffré par
bootROM	Cell	Non	Non	-
`bootldr`	SPE0	Non	Non	*bootROM*
`lv0`	PPE/HV	Oui	Non	`bootldr`
`metldr`	SPE2	Non	Non	*bootROM*
`lv1ldr`	SPE2	Oui	Non	`metldr`
`lv1`	PPE/HV	Oui	Non	`lv1ldr`
`lv2ldr`	SPE2	Oui	Non	`metldr`
`lv2`	PPE/SP	Oui	Oui	`lv2ldr`
`isoldr`	SPE2	Oui	Non	`metldr`
`appldr`	SPE2	Oui	Oui	`metldr`
jeux/applications	PPE/USR	Oui	Oui	`appldr`

Table 3. Mise à jour et révocation des *bootloaders* de la PS3

PS3 Fonctionnalités et modèle de sécurité

Il est utile de noter ici la différence de philosophie entre Microsoft et Sony concernant la protection du code critique malgré l'utilisation d'un même fournisseur (IBM). Microsoft a de son côté misé sur le « classique » chiffrement et intégrité de la RAM avec l'hyperviseur comme TCB (*Trusted Computing Base*). Sony a plutôt misé sur un nouveau paradigme d'isolation dynamique de SPE, en mettant le PPE et l'hyperviseur (quasi-)hors TCB.

Ainsi, et contrairement à l'hyperviseur de Microsoft, l'hyperviseur `lv1` de Sony n'a pas vraiment été pensé pour la sécurité. C'est lui qui gère les tables de page de GameOS et OtherOS, mais un élément notable (qui servira d'ailleurs aux attaquants) est qu'il **n'oblige pas à avoir du W⊕X activé**. Par ailleurs, il n'est pas en charge de la vérification de l'authenticité du code exécuté sur le PPE : il crée des tables de pages (potentiellement exécutables) à la demande de l'OS qui s'exécute en mode superviseur sans rien vérifier, si ce n'est que l'OS est bien cloisonné dans son espace de mémoire virtuelle. Comme nous l'avons déjà souligné, Sony a misé sur les SPE pour la vérification d'authenticité du code.

Par ailleurs, les éléments critiques étant exécutés dans les SPE, Sony ne chiffre pas la RAM et se prémunit via le mode isolation des attaques DMA depuis les périphériques. L'établissement de canaux chiffrés avec quelques périphériques (lecteur de disque dur par exemple) limite aussi la surface d'attaque via ces vecteurs.

La clé de chiffrement unique par console et non accessible depuis le logiciel est en un sens un avantage car cette clé ne pourra jamais être extraite, voire directement utilisée. C'est d'un autre côté un désavantage car cela oblige les *bootloaders* de plus bas niveau à ne jamais être mis à jour.

Concernant la protection contre la découverte de vulnérabilités, Sony a visiblement pensé que le chiffrement de code suffirait à « obfusquer » et à cacher les potentiels vecteurs d'exploitation (via des *buffer overflows* par exemple).

Enfin, le mécanisme anti-*downgrade* a été négligé et semble peu efficace. Nous verrons dans les attaques présentées ci-après que la sécurité de la PS3 n'a finalement tenu plus longtemps que pour deux raisons :

- La présence de Linux possible comme OtherOS : les *hackers* avaient peu d'intérêt à casser les protections de CoreOS ;
- La difficulté à « faire un premier pas » dans la partie propriétaire de la console. Mais une fois ce pas franchi, les attaques sont assez vite venue à cause de choix malheureux et d'erreurs grossières.

PlayStation 3 : attaques
Sony a introduit dès la sortie de la PS3 la possibilité d'installer un OtherOS sur le disque dur, Linux par exemple. Cet OtherOS s'exécute en mode superviseur au-dessus de l'hyperviseur `lv1` et a finalement assez peu de limitations imposées. Il a par exemple accès à tous les SPE, peut demander la création de tables de pages avec les droits qu'il veut, et a accès à la plupart des périphériques sur le bus EIB. La seule limitation est liée au GPU RSX : pour que sa console ne soit pas utilisée pour du jeu sous Linux, l'hyperviseur bride l'accès au mode 3D, seul le mode 2D du RSX est accessible. OtherOS et GameOS ne s'exécutent pas en même temps : l'utilisateur sélectionne au *boot* sur quel système la console démarre. Durant la chaîne de démarrage précédemment décrite, `lv1` charge OtherOS plutôt que de continuer sur la chaîne de *boot* classique.

La possibilité d'exécuter OtherOS explique le répit de quelques années (de 2006 à 2010) qu'a eu la PS3 avant sa première attaque réussie. La chronologie des attaques sur la console est présentée sur la figure 32.

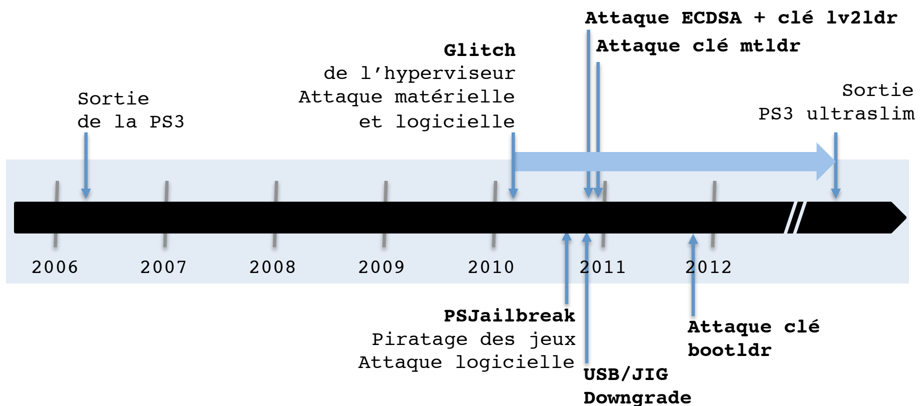

Figure 32. PS3 : chronologie des attaques

PS3 « Hello hypervisor, I'm geohot » : accès à l'hyperviseur 2010

Le premier *hack* sur le système de la Playsation 3 n'est pas vraiment une attaque. Vu que l'hyperviseur bloquait l'accès au GPU RSX depuis OtherOS, aucune accélération graphique n'était disponible (seul le mode *framebuffer* était supporté). En 2007, des développeurs du module noyau Nouveau se sont néanmoins rendus compte que le bridage du RSX fait

par l’hyperviseur pouvait être contourné via un *integer overflow* dans l’hypercall `lv1_gpu_memory_allocate` qui alloue la zone DMA d’échange avec le GPU (on pourra se référer à [22] pour plus de détails).

La première attaque réelle sur PS3 est une attaque par *glitch* [6]. Bien qu’elle ressemble de loin à ce qui a été fait sur Xbox 360, nous allons voir que le détail de l’attaque, le vecteur utilisé et ses implications sont bien différents.

Geohot voulait attaquer l’hyperviseur `lv1` depuis OtherOS. Malheureusement, du fait du chiffrement de tout le code de CoreOS, aucune analyse de l’hyperviseur n’était possible. L’attaquant n’avait au final accès qu’à son interface avec `lv1`, à savoir les **hypercalls exportés** par celui-ci.

Description de l’attaque par glitch :
Parmi les hypercalls notables figurent ceux liés à la manipulation des tables de pages de la mémoire virtuelle. Sur l’architecture PowerPC, la gestion de la traduction de mémoire virtuelle en mémoire physique se fait via des tables particulière nommées `HTAB`. À des fins de simplification (nous n’avons pas besoin de plus de détail pour expliquer l’attaque), une `HTAB` est une table qui contient pour un espace mémoire virtuel donné les traductions des adresses virtuelles vers les adresses physiques (de manière grossière, chaque entrée de la table associe l’adresse virtuelle d’une page à l’adresse physique de cette page). La paravirtualisation faite via l’hyperviseur permet à l’OS qui s’exécute au-dessus de créer des `HTAB` et de les manipuler (typiquement lorsque l’OS créé des nouveaux contextes pour des nouveaux processus, il crée de nouvelles tables) sans avoir accès la mémoire physique directement.

L’hyperviseur ne permet évidemment pas à l’OS de manipuler ces tables comme bon lui semble. Les `HTAB` fournissent normalement un accès à toute la mémoire physique, mais l’hyperviseur doit protéger son espace mémoire. De ce fait, il ne laisse pas l’OS gérer **directement** ses tables. Il existe des hypercalls permettant de créer et de manipuler les `HTAB` depuis l’OS, et l’hyperviseur applique un contrôle d’accès lors de ces appels. Ainsi l’OS peut :

- Créer un nouveau *mapping* pour un nouvel espace mémoire virtuel d’identifiant fixé (un nouveau `vsid`) via `lv1_construct_virtual_address_space` ;
- Ajouter ou invalider une traduction entre mémoire virtuelle et mémoire physique dans une table associée à un `vsid` déjà créé via `lv1_write_htab_entry` et `lv1_invalidate_htab_entries` ;

Lors de la création et la manipulation des `HTAB` via ces hypercalls, l’hyperviseur s’assure qu’aucun *mapping* demandé par l’OS ne permet

d'accéder aux zones de mémoire physique qui ne lui sont pas alloués. Par ailleurs, les `HTAB` créés sont mappés en **lecture seule** en mémoire virtuelle de l'OS (afin que celui-ci puisse y accéder par lecture directe sans passer par un hypercall).

L'objectif de Geohot est de pouvoir créer un *mapping* `HTAB` auquel il aurait accès malgré tout en **écriture**, ce qui lui donnerait un moyen d'accéder où bon lui semble en mémoire physique via des traductions, virtuel vers physique qu'il maîtrise. Afin de réaliser cela, il utilise deux éléments :

— Un *glitch* permettant de cacher à l'hyperviseur qu'un *mapping* vers une zone mémoire physique n'a en fait pas été invalidée ;
— Une fuite d'information de la part de l'hyperviseur : celui-ci fournit les adresses physiques de tables `HTAB`.

Afin de décrire un peu mieux l'attaque, nous adoptons les notations suivantes : soit `HTAB` une table contenant des traductions d'adresses virtuelles `va` vers des adresses physiques `pa`. L'hyperviseur a sa propre table de traductions pour son espace mémoire virtuel `HTAB_HV`, et l'OS a ses tables de traductions $\texttt{HTAB_OS}_i$ (la sienne $\texttt{HTAB_OS}_0$ ainsi que celles des processus qu'il gère). Par commodité, nous nommerons `pa(E)` l'adresse physique d'un élément `E`, et `va(E)` son adresse virtuelle.

Voici le détail de l'attaque par *glitch* (dont le code source est disponible en [28]) :

1. Une zone de mémoire physique `Z` est allouée pour OtherOS par l'hyperviseur via l'hypercall `lv1_allocate_memory`. Cette fonction retourne l'adresse physique de cette zone, soit `pa(Z)`. OtherOS n'a aucune maîtrise sur les adresses physiques retournées : c'est l'hyperviseur via sa gestion de la mémoire physique, son contrôle d'accès et son allocateur qui fixe cette adresse ;

2. OtherOS remplit avec `lv1_write_htab_entry` sa table $\texttt{HTAB_OS}_0$ de traductions vers cette zone. $\texttt{HTAB_OS}_0$ est donc remplie de traductions $\texttt{va(Z)}_j$ vers `pa(Z)` (ce sont des alias en mémoire virtuelle qui pointent vers la même adresse physique) ;

3. OtherOS demande ensuite à l'hyperviseur de désallouer la zone `Z` via l'hypercall `lv1_release_memory`. L'hyperviseur est alors censé **invalider** toutes les traductions dans les tables $\texttt{HTAB_OS}_i$ à `pa(Z)` (il a connaissance de toutes tables puisque c'est lui qui les a allouées à l'OS via des hypercalls, il les parcourt donc toutes) ;

4. L'invalidation par l'hyperviseur des traductions vers `pa(Z)` se fait dans le cache du CPU. L'attaquant force des *flushs* de celui-ci,

et lance un **glitch sur le bus mémoire XDRAM** en espérant perturber le *writeback* en RAM. De cette manière, si le *glitch* est bien placé, l'invalidation d'une des traductions vers `pa(Z)` dans `HTAB_OS`$_0$ ne sera pas prise en compte en RAM ;

5. Ainsi, si le *glitch* a réussi, OtherOS possède toujours une traduction `va(Z)`$_j$ vers `pa(Z)` valide au sens de la MMU, mais dont l'hyperviseur n'est pas au courant. L'objectif de l'attaquant est alors de faire allouer à l'hyperviseur une nouvelle table `HTAB_OS`$_Z$ située à une adresse physique qui *overlappe* `pa(Z)`. Pour arriver à cette fin, l'attaquant :
   — Utilise l'hypercall `lv1_construct_virtual_address_space` pour créer des nouvelles tables `HTAB_OS`$_{i,i\geq 1}$;
   — Utilise l'hypercall `lv1_map_htab` qui fournit l'adresse physique d'une table allouée par l'hyperviseur, et grâce à cette fuite d'information l'attaquant peut tester si l'adresse physique de la table `pa(HTAB_OS`$_i$`)` coïncide avec `pa(Z)` ;
   — Cela a de fortes chances d'arriver car les nouvelles tables sont allouées par l'hyperviseur dans les zones de mémoire physique libre (dont la zone `Z` fait partie puisqu'elle a été désallouée).
6. Lorsque l'étape précédente a réussi à trouver une table `HTAB_OS`$_Z$ adéquate, l'attaquant possède un *mapping* valide en mémoire virtuelle `va(Z)` vers `pa(Z)`=`pa(HTAB_OS`$_Z$`)`. Il peut donc modifier à sa guise depuis OtherOS les entrées de la table `HTAB_OS`$_Z$ **sans contrôle de l'hyperviseur** (puisqu'il ne passe plus par l'hypercall `lv1_write_htab_entry`, mais par des accès directs en mémoire). Une fois les *mappings* intéressants configurés dans la table `HTAB_OS`$_Z$, OtherOS change son espace mémoire virtuel pour l'utiliser à la place de `HTAB_OS`$_0$. Il a dès lors accès sans restriction à **toute la mémoire physique**, y compris celle de l'hyperviseur.

Cette attaque peut demander plusieurs tentatives du fait de l'imprécision inhérente au *glitch*. Elle finit néanmoins par fonctionner au bout de quelques essais (de l'ordre d'une dizaine). Elle a donc permis une élévation de privilèges d'OtherOS au niveau hyperviseur.

Possibilités offertes par l'attaque par glitch :
Grâce à cette attaque, il a tout d'abord été possible de *dumper* les binaires de l'hyperviseur afin d'analyser celui-ci plus finement. Il est aussi possible d'accéder à la flash afin d'en étudier le contenu.

La communauté s'est aussi rendue compte qu'elle pouvait utiliser les SPE, notamment pour y charger les *loaders* via `metldr`. Il est ainsi possible de charger par exemple `lv1ldr` et `lv2ldr`, et de les utiliser pour

charger `lv1` et `lv2` en RAM. Les SPE ont donc été utilisés depuis OtherOS comme **oracles de déchiffrement du code** de tous les fichiers SELF prévus pour s'exécuter sur les PPE (ceux qui s'exécutent sur les SPE sont inaccessibles du fait du mode isolation). Notons que pour `lv0` les choses sont un peu plus complexes en ce sens que `bootldr` ne peut être chargé facilement dans le SPE depuis OtherOS (ce *bootloader* est très bas niveau et suppose qu'il est chargé lors des premiers instants de démarrage du Cell).

Une partie du code de CoreOS, que Sony croyait à l'abri grâce au chiffrement, fut ainsi exposé. Finalement, l'accès hyperviseur fourni par le *glitch* de Geohot a permis la mise en place d'un bac à sable permettant d'expérimenter avec les binaires propriétaires de Sony depuis OtherOS.

Limitations de l'attaque par glitch :
Malgré son accès au plus haut niveau de privilèges du PPE, l'attaque par *glitch* ne permet pas le piratage de jeux, et ne permet pas de compromettre le *boot* sécurisé de la console. Cela explique qu'elle n'ait jamais été industrialisée sous forme de *modchip*, mais a plutôt été reproduite au cas par cas par les hackers voulant expérimenter avec l'hyperviseur.

En effet, GameOS (le `lv2`), bien que déchiffré, n'est pas compromis, de même que l'hyperviseur `lv1` lorsqu'il démarre sur CoreOS. Il en va de même pour la chaîne de *boot* pour laquelle aucun des *loaders* SPE (`bootldr`, `metldr`, `lv1ldr`, `lv2ldr`) n'a été exposé. Bien qu'il soit possible de charger ces *loaders* SPE et les binaires SELF pour des expérimentations basiques, les faire exécuter sans initialisation (propriétaire) du matériel pour charger un jeu par exemple n'est pas possible.

Réaction de Sony :
Sony avait depuis peu (fin 2009) sorti la version « slim » de sa console, sans possibilité d'installer OtherOS (en prétendant que le matériel n'était plus compatible). L'attaque de Geohot en 2010 a pour effet de supprimer complètement l'ouverture à OtherOS des PS3 en révision « fat » via la mise à jour 3.21 de CoreOS.

PS3 *Premier Jailbreak : PSJailbreak* 2010

Quelques mois après la publication de l'attaque par *glitch* fut publié un premier *Jailbreak* sous la forme d'un *modchip* particulier : une clé USB commerciale nommée **PSJailbreak**. Cette clé a rendu possible l'exécution de jeux piratés sauvegardés sur le disque dur de la console. Vu le marché potentiel, les créateurs de la clé n'ont rien communiqué sur l'exploit utilisé.

Il n'a pas fallu longtemps à la communauté pour analyser la clé PSJailbreak, en comprendre le fonctionnement et en faire des clones open

source [29] (PSGroove, PSFreedom...). L'attaque par *glitch* ayant permis la récupération du code du GameOS (`lv2`), l'analyse de celui-ci fut possible. Il s'avère que le *dongle* exploitait une erreur d'implémentation de la pile USB découvert lors de cette analyse [21].

Description de l'attaque :
Le PSJailbreak exploite une erreur d'implémentation de la pile USB de la PS3, et est exécuté durant les premiers instants de démarrage (*boot* du `lv2`) lorsque la console recherche la présence d'une clé JIG permettant d'effectuer des opérations de maintenance (la console rentre dans ce mode lors d'un appui long sur *eject* juste après l'allumage).

Nous pouvons observer le fait que la communauté n'a pas eu besoin d'explorer les détails de la vulnérabilité dans le code du `lv2` : il a suffi d'un *sniffer* USB pour récupérer les trames échangées, le clonage par mimétisme du PSJailbreak étant alors parfaitement possible. Notons à ce propos quelques divergences dans les sources concernant le détail de l'attaque, par exemple entre [21] et [33]. Nous détaillons dans la présente section la description de l'attaque faite dans [33], car elle nous semble plus cohérente.

L'erreur d'implémentation concernerait la gestion des descripteurs de configuration USB. Ces descripteurs sont des données envoyées par les périphériques USB à l'hôte lorsqu'ils sont branchés afin de négocier plusieurs éléments comme la puissance consommée maximale, le nombre d'interfaces... Un même périphérique peut avoir plusieurs descripteurs de configuration. Les descripteurs USB contiennent un champ de longueur `wTotalLength` sur deux octets. Sachant que l'OS ne peut pas savoir *a priori* la longueur totale du descripteur, la récupération de celui-ci se fait en deux passes : une première passe qui récupère un *header* de taille connue où `wTotalLength` est positionné, une seconde passe où cette taille a été allouée et, où, le descripteur complet avec ses données est récupéré. C'est lors de cette phase que la pile USB est vulnérable : si le périphérique présente une taille non nulle à la première passe puis une taille nulle à la seconde passe, une allocation mémoire est bien effectuée par la pile USB mais le descripteur n'est **pas copié** en mémoire par celle-ci. Par conséquent, de la **mémoire non initialisée** se trouve à la place du descripteur. La pile USB stocke les descripteurs récupérés dans un même tampon les uns à la suite des autres (ce tampon est augmenté au fur et à mesure de la récupération). Lorsque la pile USB récupère le descripteur de configuration suivant, elle *parse* les champs de taille des descripteurs déjà récupérés pour savoir où copier le nouveau descripteur. Si l'attaquant maîtrise le champ `wTotalLength` du descripteur précédent, il a

une primitive d'écriture en dehors du tampon alloué pour les descripteurs déjà récupérés : si `wTotalLength` est positionné à `0xAABB`, le nouveau descripteur écrasera les éléments positionnés à la position du dernier descripteur plus `0xAABB`.

Deux octets laissent peu de marge de manœuvre, mais permettent néanmoins d'écraser des éléments intéressants dans le tas si celui-ci a bien été préparé. C'est justement ce que fait le PSJailbreak en présentant un *hub* USB derrière lequel six périphériques sont exposés (voir figure 33). Voici l'ordre dans lequel les éléments sont exécutés :

1. Le périphérique 1 contient le *payload* qui sera au final exécuté après l'exploitation ;
2. Le périphérique 2 permet d'initialiser le tas avec des données afin que le tampon concernant les descripteurs du périphérique 4 soit initialisé avec des valeurs maîtrisée. Le périphérique 2 est donc débranché avant que le 4 ne soit branché ;
3. Le périphérique 3 ne contient pas vraiment de code ou de descripteurs utiles, il n'est là que pour préparer dans une zone (juste après le tampon de descripteurs du 4) des objets C++, dont les VTABLES (tables contenant les méthodes) seront écrasées ;
4. Le périphérique 4 est celui qui exploite réellement la faille : il expose trois descripteurs de configuration. Le premier est normal. Le second expose une taille non nulle à la première passe, mais nulle à la seconde. Ce second descripteur n'est donc pas copié, et utilise la mémoire initialisée par le périphérique 2 précédemment débranché. Enfin, le troisième descripteur de configuration contient le *payload* qui écrase la VTABLE d'un objet associé au périphérique 3. Le destructeur de la classe pointe alors sur le *payload* présent dans le descripteur du périphérique 1 ;
5. Le *payload* est exécuté lorsque le périphérique 3 est retiré (ce qui a pour effet d'appeler le destructeur écrasé).

Deux autres périphériques sont utilisés de manière annexe :

— Le périphérique 5 émule la JIG USB de maintenance de Sony (via les `VendorID` et `ProductID` associés). Elle ne semble être là que pour que le code du `lv2` qui gère la JIG s'exécute et permette au reste de l'exploit d'avoir lieu ;

— Le périphérique 6 sert à la suite du PSJailbreak. Le payload envoie notamment des ordres post-exploitation à ce périphérique afin de faire clignoter des LED pour signifier que le jailbreak s'est bien déroulé.

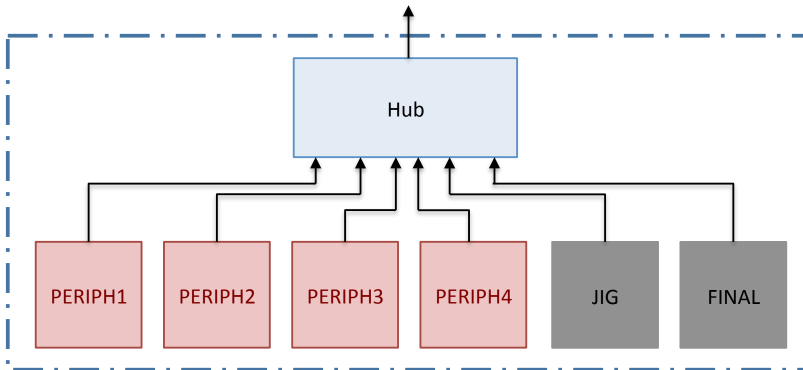

Figure 33. PS3 : *hub* du PSJailbreak

Enfin, notons que l'exécution du *payload* présent dans le tas est possible car le `lv2` **laisse le droit d'exécution sur ses pages de données** (et l'hyperviseur `lv1` n'assure pas le cloisonnement entre code et données).

Conséquences du PSJailbreak :
Comme nous l'avons mentionné, le PSJailbreak permet une prise de contrôle du `lv2` au niveau superviseur. Il ne permet pas de :

— Prendre le contrôle de l'hyperviseur `lv1` ;
— Compromettre la chaîne de *boot* en flash puisque les éléments sont signés.

Le détournement de l'exécution du GameOS permet néanmoins **d'exécuter des jeux pirates**. Le PSJailbreak utilise astucieusement le disque dur de la console pour y stocker des sauvegardes de disque Blu-Ray. Vu qu'il s'exécute sur le PPE, il peut passer outre les protections liées au chiffrement du disque dur, et peut lire les fichiers présents sur le Blu-Ray de manière transparente.

L'unique contrainte est en fait d'avoir un Blu-Ray original (n'importe lequel) avant de lancer une copie de sauvegarde : c'est l'hyperviseur `lv1` qui s'occupe de vérifier la BD-ROM Mark avant le lancement d'un jeu. Celle-ci n'étant pas falsifiable, il faut que le `lv1` détecte une marque valide avant que le `lv2` modifié ne puisse charger les éléments du jeu depuis le disque dur.

Le PSJailbreak permet aussi **d'exécuter du code non signé** puisque l'hyperviseur ne vérifie pas l'authenticité du code avant de l'exécuter. Cela a permis à une équipe de hackers de mettre une variante de Linux (AsbestOS) sur les consoles « fat » et « slim » [33].

Réaction de Sony :
Le PSJailbreak concerne les *firmwares* en versions 3.41 et inférieures. Comme précédemment détaillé, le `lv2` peut être mis à jour et est révocable. Sony est donc passé par la révocation du `lv2` dans sa mise à jour 3.42.

PS3 *Downgrade via JIG USB* 2010
Depuis la sortie du PSJailbreak, le *downgrade* vers un *firmware* 3.41 ou inférieur vulnérable est devenu possible.

Vu que Sony n'utilise pas d'appairage matériel pour la révocation, il était possible d'utiliser des *downgraders* matériels permettant d'écraser la flash. La communauté a néanmoins trouvé une solution plus simple qui passe par la JIG USB.

Comme nous l'avons décrit dans la description du PSJailbreak, la PS3 peut entrer dans mode de maintenance lors de son démarrage. Dans ce mode, elle s'attend à trouver une clé USB JIG qui partage un secret avec la console permettant de faire une authentification à base de HMAC. Une fois l'authentification passée, le `lv2` exécute le code qui lui est fourni (mais en vérifie la signature avant).

Deux problèmes se posaient donc (et ont été résolus) :
- Sans la clé secrète, il n'est pas possible de créer une clé JIG authentique : cela a été résolu via le *reverse engineering* de l'hyperviseur dans lequel cette clé était en dur ;
- Même en ayant la clé secrète, il faut pouvoir exécuter du code signé. Cela fut résolu grâce à la fuite d'un binaire de Sony permettant d'installer n'importe quel *firmware* sur une console.

Des nouveaux *modchips* de *downgrade* sont donc apparus. La réaction de Sony a été de changer la clé de JIG et de révoquer l'application qui avait fuité.

PS3 *Attaque des clés privées de signature ECDSA* 2010
Les créateurs d'AsbestOS ont présenté au *Chaos Computer Congress* de 2010 une attaque dévastatrice à l'origine de la **compromission complète et définitive de la chaîne de *boot* des consoles** déjà sorties.

Le fait de pouvoir exécuter du code depuis le GameOS a permis aux hackers d'analyser finement le fonctionnement de l'OS de Sony et ses interactions avec l'hyperviseur. Ils ont notamment pu décortiquer le format SELF et comprendre exactement comment le *boot* s'effectuait.

Vulnérabilité dans `lv2ldr` :
Le PPE avait jusqu'ici été complètement exploité (le `lv1` via le *glitch*, le `lv2` via le PSJailbreak). Seuls les SPE en mode isolation n'avaient pas encore été attaqués (hormis en tant qu'oracles pour déchiffrer des SELF).

Les créateurs d'AsbestOS ont justement découvert (vraisemblablement via du *fuzzing*) une vulnérabilité dans le `lv2ldr`. Le `lv2ldr` s'exécute intégralement dans le SPE2. Lors du *boot*, il doit vérifier l'intégrité de la liste de révocation `rvk_shared` signée avec de l'ECDSA. Le PPE charge donc cette liste dans la zone mémoire commune du LS (`0x3e000-0x40000`). Le `lv2ldr` ne vérifie pas la signature « en place », car une attaque de type TOCTOU pourrait être menée par le PPE. Le `lv2ldr` copie donc cette liste depuis la zone commune du LS vers la zone privée. Il récupère la taille de la liste de révocation dans le *header* de celle-ci, mais ne fait aucune vérification de taille (voir figure 34). Un *buffer overflow* est donc possible, permettant de déborder sur le code du `lv2ldr` et d'y placer du code maîtrisé.

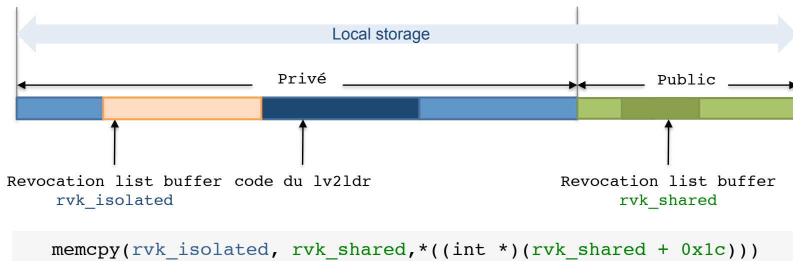

Figure 34. PS3 : vulnérabilité dans le `lv2ldr`

Grâce à cette vulnérabilité, il a été possible de récupérer `lv2ldr_erk`, `lv2ldr_riv` et `lv2ldr_ecdsa_pub`.

La vulnérabilité découverte est très intéressante car elle permet via la création en flash d'une liste de révocation adéquate (avec un *payload* adapté) de compromettre le `lv2ldr` à chaque *boot*, et de supprimer par là même la vérification de révocation. Sachant que le `lv2ldr` n'est pas révocable (puisque c'est lui qui vérifie ladite révocation), la chaîne de confiance est dès lors **totalement brisée** sur les consoles produites à la découverte de cette vulnérabilité.

La même équipe de hackers est encore allée plus loin en donnant les moyens de casser complètement la chaîne de *boot* de la console.

Signature ECDSA sans nonce :
Grâce à leur vulnérabilité sur le `lv2ldr`, les attaquants ont pu analyser les signatures de différents SELF normalement signés avec la même clé privée `lv2ldr_ecdsa_priv`.

Ils se sont rendus compte d'une faille cryptographique assez grossière que Sony a laissé passer : le standard ECDSA spécifie l'utilisation de *nonces* lors de la génération d'une signature [30]. Si ce n'est pas le cas et qu'un même nombre est utilisé pour plusieurs signatures, **seules deux signatures** suffisent à retrouver la clé privée !

Les attaquants ont donc pu retrouver la clé privée `lv2ldr_ecdsa_priv` à partir de deux signatures de deux SELF signés avec cette clé. Dès lors, il est possible de signer n'importe quel SELF destiné à être vérifié par cette clé. Il est possible de **produire des SELF parfaitement valides aux yeux du `lv2ldr`**. Connaissant la clé de chiffrement `lv2ldr_erk`, il devient aussi possible de déchiffrer n'importe quel SELF destiné à ce *loader*.

Tous les *loaders* semblent affectés par cette vulnérabilité. Il y a donc moyen de casser complètement la chaîne de confiance de Sony utilisée lors du *boot*. Le seul rempart à cela reste la récupération des clés `*_eik` ainsi que de signatures produites par `*_ecdsa_pub` (rappelons que les signatures sont chiffrées dans les SELF). Il faut donc trouver des vulnérabilités dans les *loaders* qui sont en mode SPE isolé afin de récupérer ces éléments : c'est cette « course au *dump* de clés et signatures » qui agite la communauté durant deux ans.

Réaction de Sony :
Les *firmwares* vulnérables à l'attaque ECDSA sont ceux inférieurs ou égaux au 3.55. Sony a réagi en corrigeant la vulnérabilité sur la signature et en mettant à jour sa liste de révocation, mais cela était trop tard pour les consoles déjà sorties. Seules des nouvelles révisions flashées avec la correction des `metldr` et `bootldr` en usine corrigent cela.

PS3 *Clés des bootloaders : la course au dump* 2010-2012

Après le `lv2ldr`, la cible de choix est le `metldr` puisque s'il est compromis, tous les autres *loaders* qui en dépendent (`lv1ldr`, `appldr`, `isoldr`...) le seront aussi. Il est dès lors possible de créer des versions CFW (*Custom Firmwares*) de toutes les mises à jour officielles fournies par Sony sur les consoles vulnérables.

Attaque du `metldr` :
De même que pour le `lv2ldr`, il est nécessaire de trouver une vulnérabilité dans le `metldr` pour en récupérer la clé secrète `metldr_erk` ainsi que des signatures permettant de calculer la clé privée ECDSA.

La manière dont s'y sont pris les attaquants (décrite en [23]) est remarquable. L'attaque se fait en deux étapes :

— Récupération de la signature : ils se sont rendus compte que `metldr` utilise un offset dans le *header* SCE du fichier SELF afin de calculer l'endroit où le déchiffrement des métadonnées a lieu. De plus, le *header* SCE est le seul élément non chiffré du SELF, il est donc possible de le modifier sans connaître `metldr_erk`. Par ailleurs, vu que la vérification de la signature se fait **après** le déchiffrement des métadonnées (puisque la signature ECDSA est chiffrée), le *bootloader* s'en rend compte trop tard. Il est ainsi possible en jouant sur l'offset précédemment décrit de faire déchiffrer les métadonnées dans la zone commune du LS accessible au PPE. À partir de là, la clé locale AES-128 de déchiffrement du SELF est récupérée. Le SELF est déchiffré et sa signature aussi. La clé privée `metldr_ecdsa_priv` est calculée.

— Récupération de la clé `metldr_erk` : grâce à l'étape précédente, il est possible de signer un SELF valide avec du code modifié (puisque la clé locale de chiffrement est connue). Le chargement d'un SELF qui *dump* tout le contenu de la partie privée du LS montre qu'en fait `metldr` efface toutes les données avant de sauter sur le point d'entrée du SELF qu'il a chargé. Il n'est donc pas possible de récupérer `metldr_erk` par cette méthode. L'astuce consiste à écraser au chargement (en jouant sur les adresses de chargement du ELF) le code responsable de l'effacement de la mémoire du `metldr`. Ce dernier vérifie néanmoins l'adresse de chargement, et refuse de charger du code qui écrase le code du `metldr`. Une seconde astuce consiste à passer outre la vérification en mettant une adresse plus grande que la taille maximale du LS `0x40000` et qui est congrue à l'adresse cible modulo cette taille.

Attaque du `bootldr` :
Une fois les clés du `metldr` découvertes, il ne restait plus beaucoup d'alternatives à Sony pour protéger sa console. Dans une ultime tentative de correction, Sony modifie radicalement la chaîne de *boot* sur le *firmware* 3.60 pour tout ancrer au `bootldr` dont les clés n'ont pas été compromises. Ainsi, le `metldr` est supprimé et son rôle est intégré au `lv0` qui devient un méta-*loader*.

La difficulté pour les attaquant vis-à-vis des clés du `bootldr` réside dans le fait qu'ils ne peuvent pas le charger à la volée sur un SPE comme c'était le cas avec le `metldr`. Or l'avantage de charger un *bootlader* depuis un OS maîtrisé est de pouvoir le *fuzzer* et en étudier le comportement facilement sans devoir redémarrer la console. Ce n'est pas le cas avec le

`bootldr` qui est écrit pour s'exécuter à très bas niveau et nécessite un *reboot* dès qu'il plante.

La communauté a donc mis plusieurs mois avant de trouver une vulnérabilité exploitable permettant de récupérer les dernières clés qui permettaient à Sony de garder un peu de maîtrise sur sa console. Peu d'éléments publics ont été divulgués concernant l'exploitation du `bootldr`, mais il semblerait qu'il ait fallu du matériel dédié permettant de gérer les *reboot* de la machine, ainsi que l'utilisation d'une autre vulnérabilité permettant post-*reboot* de récupérer une RAM non effacée (voir [10]).

Réaction de Sony :
Sony ne pouvait plus faire grand-chose sur les consoles déjà sorties et vulnérables : toutes les consoles dont le *firmware* usine est antérieur à 3.60 sont vulnérables et le seront pour toujours. Ce n'est par contre pas le cas des consoles fabriquées depuis 2012, c'est-à-dire une partie des PS3 « slim » et toutes les PS3 « super-slim ».

PlayStation 3 : conclusions sur la sécurité

Avec les quelques années qui nous séparent de la sortie de la PS3, nous pouvons faire une analyse du modèle de sécurité et des attaques qui ont été menées sur cette console.

Une plateforme matérielle intéressante :
La plateforme matérielle sur laquelle est fondée la PlayStation 3, à savoir le Cell BE, a amené un nouveau paradigme concernant le démarrage sécurisé et l'exécution dynamique sécurisée d'applications. L'idée de pouvoir exécuter du code sur des unités d'exécution isolées du processeur principal est très intéressante, mais a visiblement amené Sony à négliger certains concepts assez basiques de défense en profondeur comme nous le rappelons ci-après.

Il n'en demeure pas moins que le concept de SPE isolé avec une graine de confiance située en ROM et une clé CPU en eFuse non lisible est très utile. Aucune clé CPU de PlayStation 3 n'a été compromise, contrairement au cas de la Xbox 360.

Un élément manquait néanmoins à la plateforme d'IBM : une MMU dans les SPE, qui aurait permis une limitation de l'exploitation de vulnérabilités (si tant est que le code en tire correctement partie).

Un hyperviseur limité :
Sony n'a pas conçu l'hyperviseur `lv1` comme un hyperviseur de confiance destiné à la sécurité (contrairement à ce qu'a fait Microsoft pour sa Xbox 360). Bien que la prise de contrôle de cet hyperviseur ne compromette pas

complètement la console, sa position centrale au sein du système en fait un élément important à consolider. L'absence par exemple de concepts aussi fondamentaux que le `W⊕X` a eu des conséquences non négligeables sur la mise au point d'attaques comme le PSJailbreak.

Une chaîne de *boot* à la sécurité limitée :
Sony a fait le choix de diversifier les clés des premiers *bootloaders* (`metldr` et `bootldr`) par console. Ce choix peut sembler judicieux au premier abord car il permet d'appairer fortement le système à la console. Il a néanmoins un prix à payer : l'impossibilité de mettre à jour ces éléments. La moindre vulnérabilité sur ces *bootloaders* a une conséquence implacable : ils resteront vulnérables *ad vitam*. C'est d'ailleurs ce qu'il s'est passé avec la vulnérabilité ECDSA et la récupération des clés desdits *bootladers*.

Par ailleurs, comme nous l'avons présenté, la révocation des éléments du système n'est mise en œuvre que tard dans la chaîne (à partir des `lv2ldr` et `lv2`). Il faut donc considérer que tous les éléments précédents dans la chaîne de *boot* sont *downgradable* si une vulnérabilité est découverte.

La sécurité par l'obscurité :
Sony a pensé que le chiffrement de leurs binaires cacherait de potentielles vulnérabilités aux attaquants. Il n'en n'est rien, et les attaquants ont été assez malins pour réussir malgré tout à exploiter des vulnérabilités « à l'aveugle » dans les *loaders* en mode SPE isolés dont ils n'avaient pas le code. Les codes exécutés dans les SPE ont d'ailleurs montrés des failles assez basiques de non-vérification ou mauvaise vérification de bornes.

Des petits oublis aux conséquences lourdes :
Les *nonces* ECDSA (qui n'en sont pas) de Sony sont devenus un cas d'école en cryptographie. Il est intéressant de voir l'impact d'un si petit oubli...

OtherOS or not OtherOS ?
Nous abordons enfin la question de l'OS alternatif. Sa suppression a-t-elle précipité la découverte de vulnérabilités sur la PlayStation 3 ? Plusieurs sources, dont [33], affirment que oui. Nous nuançons malgré tout cette affirmation (mais c'est un avis très subjectif).

Il est vrai que la disparition complète d'OtherOS suite au *glitch* de Geohot a provoqué une levée de boucliers unanime chez les *hackers* et autres scientifiques qui utilisaient le Cell à des fins académiques.

Néanmoins, le *glitch* fut possible principalement grâce à l'accès à OtherOS. Et c'est à partir du *dump* des SELF fait depuis OtherOS que des attaques comme le PSJailbreak ont pu se faire. OtherOS a donc, à notre sens, *bootstrapé* l'analyse du système propriétaire de Sony. Cela est à nuancer avec le fait que de 2006 à 2010, la PS3 n'a souffert de presque aucune vulnérabilité...

Références

1. 1920to1921. https://github.com/gligli/tools/blob/master/imgbuild/1920to1921.py.
2. 849x System Update. http://www.free60.org/wiki/849x_System_Update.
3. 849x System Update. http://www.free60.org/wiki/Reset_Glitch_Hack.
4. Application-specific integrated circuit. http://fr.wikipedia.org/wiki/Application-specific_integrated_circuit.
5. ATA 3 Specification. http://www.t13.org/project/d2008r7b-ATA-3.pdf.
6. Attaque par glitch de la Playstation 3. http://www.eurasia.nu/wiki/index.php/PS3_Glitch_Hack.
7. Background DVD and 360 Game Info. http://dark.ellende.eu/public/360DVDfirmwareRelatedInfo.pdf.
8. Boot Process, howpublished = http://www.free60.org/wiki/Boot_Process,.
9. Building an Xbox 360 Emulator. http://www.noxa.org/blog/2011/08/13/building-an-xbox-360-emulator-part-5-xex-files/.
10. Cell Reset Exploit. http://www.psdevwiki.com/ps3/CELL_Reset_Exploit.
11. Details of the Xbox Hard Drive Locking Mechanism. https://web.archive.org/web/20030203172300/http://xbox-linux.sourceforge.net/articles.php?aid=2002224023814.
12. Differences between Xbox FATX and MS-DOS FAT. https://web.archive.org/web/20051110034740/http://www.xbox-linux.org/wiki/Differences_between_Xbox_FATX_and_MS-DOS_FAT.
13. Firmware lecteur DVD Samsung Xbox hacké. https://web.archive.org/web/20100114065747/http://gx-mod.com/xbox/modules/news/article.php?storyid=4707.
14. Format SELF de la Playstation 3. http://www.psdevwiki.com/ps3/SELF_File_Format_and_Decryption.
15. imgbuild. https://github.com/gligli/tools/blob/master/imgbuild/build.py.
16. Intel® Low Pin Count (LPC) Interface Specification. http://www.intel.com/design/chipsets/industry/25128901.pdf.
17. Microsoft deleted my data - remotely, without my permission, and... without even bothering to ask ! https://web.archive.org/web/20040427031653/http://xbox-linux.org/docs/remotedelete.html.
18. PIC Challenge Handshake Sequence. https://web.archive.org/web/20030608101810/http://xbox-linux.sourceforge.net/articles.php?aid=2002173101523.
19. Processeur Cell BE. http://sebug.net/paper/phrack/66/p66_0x0D_Power_cell_buffer_overflow_by_BSDaemon.txt.
20. Protecting eFuses and JTAG From Updates (R6T3, U6T1, U6T2 Methods). http://team-xecuter.com/forums/showthread.php?62331-Protecting-eFuses-and-JTAG-From-Updates-(R6T3-U6T1-U6T2-Methods).
21. Reverse engineering du PSJailbreak. http://www.eurasia.nu/wiki/index.php/PS_Jailbreak_reverse_engineering.

22. RSX Hack. http://rvlution.net/thread/138-rsx-driver-for-the-ps3-linux/.
23. Récupération des clés du metldr. http://www.psdevwiki.com/ps3/Dumping_Metldr.
24. SDK Cell et Linux. http://www.powerdeveloper.org/platforms/cell/playstation.
25. Securité du Cell et isolation des SPE. http://www.ibm.com/developerworks/power/library/pa-cellsecurity.
26. SMC. http://www.free60.org/wiki/SMC.
27. SMC Hack. http://www.free60.org/wiki/SMC_Hack.
28. Sources de l'attaque par glitch de la Playstation 3. http://xorloser.com/blog/wp-content/uploads/2010/03/exploit.c.
29. Sources du PSFreedom, clone du PSJailbreak. https://github.com/kakaroto/PSFreedom.
30. Standard ECDSA. http://csrc.nist.gov/groups/ST/toolkit/digital_signatures.html.
31. Système AACS. http://cacr.uwaterloo.ca/techreports/2007/cacr2007-25.pdf.
32. Système BD-ROM Mark. http://www.cd-info.com/blu-ray/bd-rom/.
33. Sécurité de la Playstation 3 (Chaos Computer Congress 2010). http://events.ccc.de/congress/2010/Fahrplan/events/4087.en.html.
34. The Hidden Boot Code of the Xbox. https://web.archive.org/web/20050814022350/http://www.xbox-linux.org/wiki/The_Hidden_Boot_Code_of_the_Xbox.
35. Timming Attack. http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack.
36. Understanding the Xbox boot process/Flash structures. http://hackspot.net/XboxBlog/?p=1.
37. .XBE File Format. https://web.archive.org/web/20040229204243/http://xbox-linux.sourceforge.net/docs/xbe.html.
38. xbfuse source code. https://github.com/multimediamike/xbfuse/blob/master/src/xdvdfs.c.
39. Xbox hard disk lock. http://skaya.enix.org/docs/xbox-hard-disk-lock.txt.
40. ACID_SNAKE. How PS1 security works. http://wololo.net/2012/12/10/how-ps1-security-works/, 2012.
41. Alexander McCaleb. An Introduction to Computer Architecture and the Xbox 360. http://people.ucsc.edu/~amccaleb/documents/Arch_360_Intro.pdf, 2012.
42. Anonymous Hacker. Xbox 360 hypervisor privilege escalation vulnerability. http://www.securityfocus.com/archive/1/461489, 2007.
43. Eric DeBusschere and Mike McCambridge. Modern game console exploitation. 2012.
44. Gatchan. MultiMode 3 pour 12f629MultiMode 3 for 12f629. http://www.gatchan.net/2012/08/27/multimode-pour-12f629/, 2012.
45. Gligli. Fusesets. http://free60.org/wiki/Fusesets, 2011.

46. Andrew Huang et al. Hacking the xbox. 2003.
47. Sony Computer Entertainment Inc. *PlayStation Hardware*. Sony Computer Entertainment Inc., Sony Computer Entertainment America 919 E. Hillsdale Blvd., 2nd floor Foster City, CA 94404, 1 edition, 8 1998.
48. IVC Wiki. King Kong Shader Exploit for the XELL Loader. `http://beta.ivc.no/wiki/index.php/Xbox_360_King_Kong_Shader_Exploit`, 2006.
49. John Kelsey, Bruce Schneier, and David Wagner. Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea. *Information and Communications Security*, pages 233–246, 1997.
50. Michael Steil. 17 mistakes microsoft made in the xbox security system. *CCC22*, 2005.
51. D Sweetman and N Stephens. Idt r30xx family software reference manual. *IDT Manual*, 1994.
52. William R Tonti. efuse design and reliability. In *Integrated Reliability Workshop Final Report*, page 114, 2008.