Sticky fingers & KBC Custom Shop

Alexandre Gazet

Sogeti - ESEC R&D

Résumé L'utilisation d'un mot de passe `BIOS` est une mesure généralement conseillée dans un processus de durcissement de plateforme. L'implémentation d'une telle fonctionnalité est laissée à la discrétion des fabricants de machine (ou éventuellement aux vendeurs de `BIOS`). Or, pour des raisons sûrement légitimes, un procédé de recouvrement est régulièrement implémenté conjointement sous la forme d'un mot de passe maître. Si l'on s'intéresse d'un peu plus près au sujet, nous découvrons très rapidement qu'une industrie parallèle s'est formée autour de la revente de ces mots de passe maîtres. Une simple recherche avec les termes « *bios password laptop* » sur des plateformes de vente en ligne comme Ebay renvoie un nombre important de résultats. Le prix unitaire d'un mot de passe se situe le plus souvent autour de quelques dizaines de dollars américains. Un individu est toutefois venu perturber ce petit monde tranquille en mettant à disposition du public, via son blog [4], les algorithmes de génération de mot de passe maître pour plusieurs modèles d'un fabriquant de portables bien connu.

Le point de départ de cette étude est la propension de l'auteur à perdre ses mots de passe ; le mot de passe `BIOS` ne faisant pas exception. Dès lors, nous nous sommes intéressés à l'implémentation de la fonctionnalité de recouvrement, pour ensuite dériver vers des méandres plus obscurs.

1 Découverte de l'implémentation

1.1 Accès au `BIOS`

Nous savons que le `BIOS` du portable concerné implémente un mécanisme de mot de passe maître permettant le recouvrement de l'accès. Pour analyser l'implémentation, deux options s'offrent à nous. La première, orientée vers l'analyse dynamique, via la lecture des zones mémoires contenant le code du `BIOS`, au risque de passer à côté de zones mémoires chargées puis déchargées dynamiquement. Une seconde, plus orientée vers l'analyse statique, via l'analyse d'un package de mise à jour du `BIOS` proposé par le fabricant. La seconde option a été retenue pour ces travaux.

Nous ne partons pas totalement de zéro car certaines communautés se sont déjà intéressées à ces packages de mise à jour, en particulier la communauté des *modders* de `BIOS`. Leurs motivations sont multiples :

- Personnalisation du processus de démarrage
- Activation de fonctionnalités cachées du `BIOS`

– Modifications des tables `ACPI`, par exemple pour y inclure de nouvelles informations de licence (certificats)
– ...

Le sujet « *Dell bios, how to decompose/mod* » [2] contient l'essentiel des informations dont nous avons besoin. Un package de mise à jour se présente sous la forme d'un exécutable (sous Windows). Il supporte, entre autres, les options suivantes sur la ligne de commande :
– `-writeromfile` : écrit la `ROM` du `BIOS` dans un fichier
– `-writehdrfile` : sensiblement comme le fichier `ROM` mais contient un en-tête supplémentaire et utilise des *paddings* différents
– `-writekromfile` : écrit la `ROM` du contrôleur clavier dans un fichier

Comme cela est décrit dans l'article [2], il est possible de décomposer un fichier `hdr` en une suite de modules. Ces modules peuvent être de plusieurs natures (identification parfois possible grâce aux en-têtes ou aux chaînes de caractères) :
– Images
– Portion de code du `BIOS`
– Module `CompuTrace` [8]
– `ROM PCI`
– `VGA BIOS`
– Tables `ACPI`
– etc.

Nous savons que le mot de passe maître est dérivé du numéro de série du portable. Pour le modèle concerné, les numéros de séries portent le suffixe « `2A7B` ». Un *grep* de ce motif sur l'ensemble des modules extraits renvoie un module candidat.

1.2 Analyse du module candidat

Le module candidat ne possède aucun en-tête de fichier ; toutefois, si l'on excepte les zones remplies de zéros, des chaînes de caractères sont présentes et le reste des octets a l'apparence de code binaire Intel 32-bit. Une fois interprété comme tel et chargé à la bonne adresse de base, nous découvrons que ce module contient en fait le code du *handler* `SMM`. Ceci est indiqué par la présence d'instruction `RSM` (*Resume from System Management Mode*). La transition 16-bit vers 32-bit en mode `SMM` est un point intéressant, documenté indirectement via divers brevets [10]. Dès lors, à l'aide des sources contenant les algorithmes publiées initialement, il est très facile d'identifier le code responsable de la dérivation du mot de passe maître ; en particulier

à l'aide du suffixe et de l'utilisation d'un algorithme dérivé de `MD5` (en cas de doute sur ce dernier, se référer aux travaux de M. Stéphane Duverger sur le sujet). L'objectif initial, pour rappel l'étude de l'implémentation de la fonctionnalité de mot de passe maître, est donc pratiquement plié.

Toutefois, une étude comparée plus minutieuse du code et des sources publiées, nous amène à la conclusion suivante : certaines fonctions, présentes dans les sources publiées, ne sont tout simplement pas présentes dans le code du *handler* `SMM`. Comment expliquer ceci ? Une nouvelle étude du code met en exergue le fait suivant : là où devraient se trouver les fonctions manquantes, nous trouvons des instructions assembleur `IN` et `OUT` sur les ports 0x910, 0x911. Ces ports sont non standards et aucune documentation les concernant n'a alors été trouvée.

Après un échange de mail avec l'auteur du blog, la boîte de Pandore s'entrouvit : le *handler* `SMM` communique avec un composant hardware, le contrôleur clavier !

2 Another world

2.1 Analyse du protocole de communication

Nous savons que la communication avec le contrôleur clavier se fait via les ports 0x910, 0x911. Nous cherchons maintenant à reconstruire le protocole de communication. Dans cette démarche, il est utile de prendre un peu de recul et de regarder l'existant, en particulier le protocole de communications avec la `Real Time Clock` [7] (`CMOS RTC`).

```
cli
mov al, index
out 0x70,al
in al,0x71
sti
```

Lecture d'un octet

```
cli
mov al,index
out 0x70,al
mov al,value
out 0x71,al
sti
```

Écriture d'un octet

Un port indique l'adresse, le second la donnée (octet) à lire/écrire. Nous appliquons ces concepts aux ports 0x910/0x911 et utilisons une démarche ascendante classique. Certains octets lus/écrits influencent le comportement du code : nous différencierons alors les octets de contrôle (`ctrl`) des octets de données (`data`). À partir des primitives de base `readbyte`, `writebyte`, nous composons :

- plusieurs `readbyte` successifs : `readbuffer`
- plusieurs `writebyte` successifs : `writebuffer`

– boucle sur un octet de contrôle : `loopbyte`

Au plus haut niveau, nous retrouvons une composition qui n'est pas dénuée de charme :
`writebuffer(data)+writebyte(ctrl)+loopbytectrl+readbuffer(data)`

Comme dans un épisode de Scoubidou, voici un des premiers mystères de l'histoire élucidé. Même si l'hypothèse n'est pas encore validée, les fonctions manquantes dans le code `SMM` sont déportées sur le contrôleur clavier et le *handler* `SMI` interroge le contrôleur via un protocole de communication dédié. Les données sont tout d'abord envoyées au contrôleur, puis l'envoi d'un octet de contrôle (qui correspond au numéro de la commande) déclenche l'exécution. Le *handler* `SMI` se met alors en attente (boucle) sur un octet de contrôle lu depuis le contrôleur. Lorsque cet octet est activé, il signale la complétion de l'exécution de la commande, le résultat est disponible (tampon data). Le *handler* `SMI` exécute donc des commandes à distance sur le contrôleur clavier.

2.2 Identification du matériel

Pour valider notre hypothèse, il nous faut analyser le code présent sur le contrôleur. Nous avons accès à la `ROM` du contrôleur, pour rappel à l'aide de la commande `-writekromfile`, il nous faut l'interpréter. Il est possible de trouver sur les sites de plusieurs réparateurs de portables des schématiques (au niveau matériel) de modèles de portables proches de celui sur lequel a été réalisée cette étude. Ces schématiques nous informent que le contrôleur clavier serait un modèle de la famille `MEC5035` ou `MEC5025`. Ce composant serait fabriqué par SMSC. Toutefois, à notre connaissance, aucune spécification technique supplémentaire n'est disponible, c'est aussi la raison de l'emploi du conditionnel dans les phrases précédentes.

Ce type de composant, contrôleur clavier (*keyboard controller*, `KBC`), ou SuperIO (`SIO`), est une unité de calcul indépendante du processeur principal mettant en œuvre un processeur (alors inconnu) autonome. Nous avons vu précédemment qu'il était possible d'extraire la `ROM` depuis le package de mise à jour du `BIOS`. Nous nous retrouvons donc à résoudre le problème suivant : comment identifier un processeur à partir du code binaire. Si la plupart des contrôleurs clavier ou `SIO` sont construits autour d'un processeur Intel 8051, ce n'est pas notre cas.

Après de (très) nombreuses recherches infructueuses, le processeur utilisé par le composant a pu être identifié via un fil de discussion initié par une personne utilisant

le pseudonyme Alzou[1]. C'est donc un processeur de type `ARCompact`[2]. Ce processeur est également utilisé par certaines cartes wifi Intel. Il possède plusieurs particularités parmi lesquelles :

- de pouvoir entrelacer un encodage 16-bit et 32-bit des instructions
- jeu d'instructions éventuellement personnalisé

Une nouvelle fois peu d'informations techniques précises sont disponibles ; néanmoins un livre blanc décrivant le jeu d'instruction est disponible sur le site. Le processeur n'étant pas supporté par IDA, la solution la plus immédiate, la plus simple et la plus élégante est évidement l'ajout du support de ce processeur au merveilleux outil qu'est Metasm[3][4], développé par le non moins merveilleux Yoann Guillot. Nous avions déjà illustré l'implémentation d'un processeur via un exemple[5], il a été procédé d'une manière similaire pour le support `ARCompact`.

2.3 Brave a new world

Suffisament équipés, nous sommes maintenant en mesure de mener un raid contre ce composant. Plusieurs difficultés sont toutefois présentes :

- Espace mémoire du contrôleur clavier globalement inconnu (`RAM`/`ROM`, types de mémoires, permissions, etc.)
- Fonctionnalités du composant inconnues ; en particulier les capacités d'entrées/sorties

Pour rappel, l'objectif est maintenant de vérifier la présence dans le code du contrôleur clavier des fonctions que nous n'avions pu retrouver dans le code du *handler* `SMI`. Ce premier objectif est finalement trivial, à l'aide de constantes présentes dans le code ces portions de code sont très rapidement identifiées et par la même occasion, notre hypothèse validée.

À ce stade nous avons donc validé le fait que le *handler* `SMI` est capable d'exécuter des commandes sur le contrôleur clavier via un protocole de communication dédié (pour simplifier, il est possible de faire une analogie avec `RPC`). Il est alors assez naturel de se demander comment est gérée cette exécution de commande au niveau du contrôleur. Les zones mémoires correspondantes au protocole de communication décrit précédemment sont assez facilement identifiables. De là, nous découvrons le

1. http://topic.csdn.net/u/20090709/13/66a41828-aaeb-427a-90ab-b5e0136cf511.html
2. http://www.synopsys.com/IP/ConfigurableCores/ARCProcessors/Pages/default.aspx
3. http://metasm.cr0.org/
4. L'auteur assume pleinement son prosélytisme.
5. http://esec-lab.sogeti.com/dotclear/index.php?post/2009/10/02/71-how-to-implement-a-new-process-in-metasm

reste de la sémantique du protocole. Une requête peut se décomposer de la manière suivante : un octet sert à décrire le numéro de la commande à exécuter, un octet sert de contrôle ou statut tandis que le reste forme les données.

Nous avons développé une interface permettant de communiquer avec le contrôleur clavier via les ports dédiés. Pour cela, notre code doit utiliser les instructions `IN` et `OUT` ; ces instructions ne sont normalement pas accessibles depuis le mode utilisateur du système d'exploitation. Le drapeau `IOPL` (`IO` *Privilege level*) du registre `EFLAGS` définit le niveau de privilèges requit pour réaliser des `IO` (3 : mode utilisateur, 0 : mode noyau). Une solution est de développer un pilote, chargé dans le noyau. Cette solution serait satisfaisante si nous ne possédions pas un magnifique outil tel que Metasm [6]. En effet, notre implémentation repose sur un script Ruby, qui modifie son propre `IOPL` afin de pouvoir réaliser des `IO` depuis le mode utilisateur. Ceci est rendu possible par le démarrage du système d'exploitation en mode *debug* et en chargeant un pilote de Windbg qui nous ouvre l'accès à la toute la mémoire physique/virtuelle depuis le mode utilisateur.

Une fonctionnalité de Metasm, `DynLdr`, nous permet de compiler dynamiquement du code assembleur puis de l'appeler directement depuis notre script, la conversion des types `Ruby` vers `C` est effectuée automatiquement. Voici par exemple la définition de méthodes d'`IO` `readbyte` et `writebyte`.

```
class MEC
 def init
  parse_c ''
  new_func_c("__fastcall int readbyte(int) __attribute__((zero_not_fail)){
   asm(\"mov edx, 0x910  movzx eax, cl out dx, al mov edx, 0x911 in al, dx
       movzx eax, al\"); }")

  new_func_c("__fastcall int writebyte(int, int) __attribute__((zero_not_fail))
      {
   asm(\"push edx mov edx, 0x910  movzx eax, cl out dx, al pop eax mov edx, 0
       x911 out dx, al movzx eax, al\"); }")
 end
end
```

Parmi les fonctions exposées par le contrôleur, il est intéressant de noter qu'une nous permet de lire la `ROM` du contrôleur (les 0x28000 premiers octets). Le reste de la mémoire du processeur nous est toutefois inaccessible via cette fonction. Le code est visible sur la figure 1 ; le test, à l'aide de l'instruction de saut conditionnel `brlo` (*branch if lower*) et de la valeur numérique 0x28000 est clairement visible. Si l'adresse demandée est supérieure à cette limite, la fonction renvoie -1. Enfin, à titre

6. Pour rappel : http://metasm.cr0.org/

d'information, il est possible de déduire que les `IO` sont mappées à partir de l'adresse 0xff0120 dans la mémoire du contrôleur.

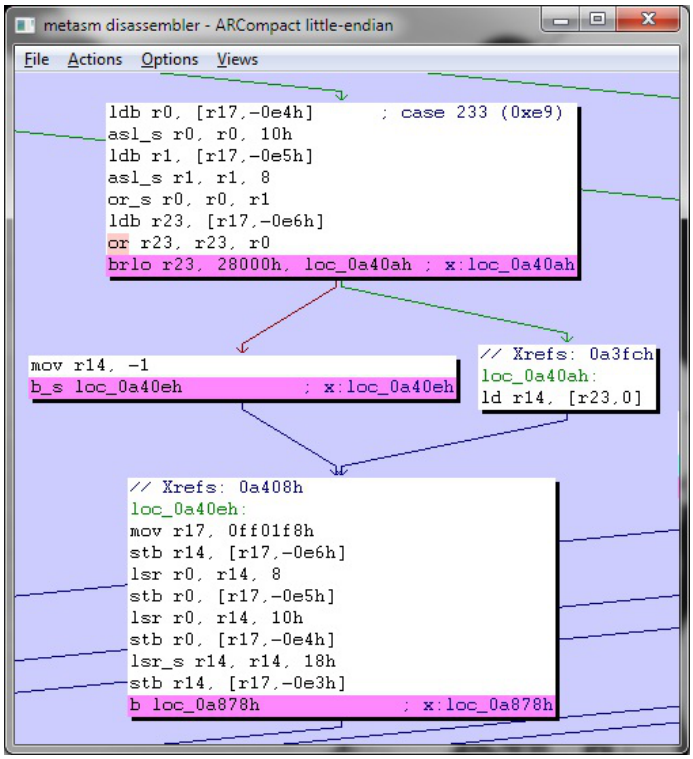

Figure 1. Fonction de lecture de la mémoire du contrôleur.

3 All your base are belong to us

3.1 Dressed to kill

Le contrôleur clavier est une unité de calcul indépendante, à coté du processeur principal. De fait, du point de vue de l'attaquant, il présente un intérêt certain : stocker des données ou du code totalement à l'insu du système d'exploitation, détournement de fonctionnalités existantes. L'idéal serait évidement de pouvoir exécuter du code arbitraire sur le contrôleur.

Pour ce faire, nous avons étudié le processus de mise à jour du contrôleur clavier et par la suite avons été en mesure de développer notre propre outil de mise à jour du

code du contrôleur : nous sommes libres de modifier/injecter le code du contrôleur. Les premiers redémarrages étant l'occasion de grands moments de solitude et d'angoisse, il est l'heure de faire appel à M. Crowley pour conjurer la peur de l'écran noir.

De fait, avec surprise et satisfaction nous constatons que le processus implémenté semble fiable, aucun portable n'ayant été tué durant les tests. Nous sommes en mesure de modifier le code du contrôleur ; dans l'implémentation actuelle, les modifications sont persistantes tant que le portable a accès à une source d'énergie (batterie ou secteur). Voici donc les modifications apportées au code du contrôleur :

- Débridage de la commande de lecture : cela nous permet alors de lire tout l'espace mémoire du contrôleur et affiner notre connaissance : zone de `RAM`, *mapped* `IO`, etc. C'est anecdotique, mais cela nous permet également de retrouver une forme légèrement encodée du mot de passe maître, accompagnée du service tag, dans la `RAM` (Fig. 2).
- Ajout d'un *hook* sur l'exécution de commande : enregistrement du numéro de commande pour chaque requête. Nous sommes ainsi en mesure d'enregistrer les interactions entre le mode `SMM` et le contrôleur même lors d'un redémarrage de la machine
- Ajout d'une commande à l'interface, permettant l'écriture d'un octet dans la `RAM` du contrôleur à une adresse arbitraire
- *Hook* de la fonction permettant l'envoi d'un tampon : ce point sera explicité dans les paragraphes suivants

Dès lors, nous pouvons dire que le contrôleur clavier est totalement compromis. À ce stade, le résultat est déjà intéressant d'un point de vue offensif. Un attaquant peut cacher et exécuter du code sur une unité de calcul autonome, à l'insu du processeur principal et donc du système d'exploitation de l'hôte. Toutefois, et c'est le revers de la médaille, notre isolement du reste du système limite nos capacités d'actions malveillantes. Ralf-Philipp Weinmann [12] a proposé l'implémentation d'un *keylogger*. Nous avons opté pour une voie différente.

Sachant que le contrôleur échange des données avec le *handler* `SMI`, un utilisateur malveillant pourrait-il en tirer profit de la compromission du contrôleur pour attaquer le *handler* `SMI` depuis le contrôleur ?

3.2 You shall not pass

D'un point de vue théorique, le processeur entre en mode `SMM` lorsqu'une *System Management Interrupt* (`SMI`) est levée, elle est typiquement générée par le chipset

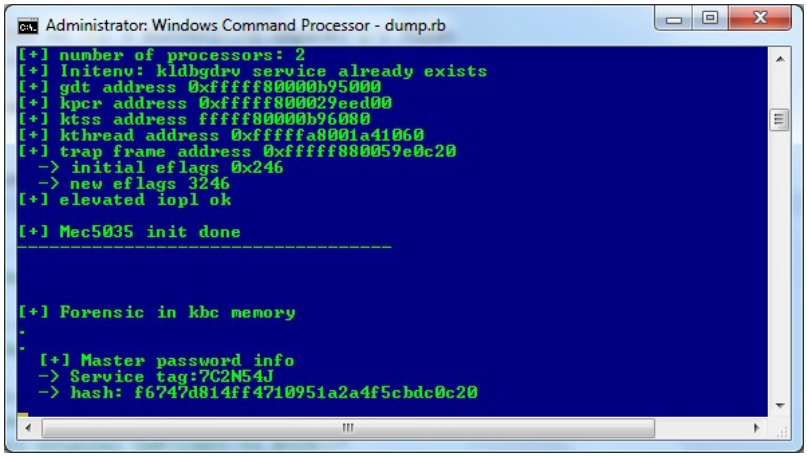

Figure 2. Forme encodée du mot de passe maître et service tag retrouvés dans la RAM du contrôleur.

(*northbridge*). Le handler `SMI` est situé dans la `SMRAM`, c'est une zone de mémoire spéciale située dans la `RAM` mais qui ne peut être accédée que si le processeur est en mode `SMM`. Une fois le handler chargé, l'accès à la `SMRAM` est donc contrôlé, le bit `D_LOCK` situé dans le registre de configuration `SMRAMC`, est ainsi systématiquement activé sur les systèmes récents (y compris celui utilisé pour ces travaux). Lorsque le processeur entre en mode `SMM`, il sauvegarde le contexte courrant dans la `SAVE_STATE_MAP`, et le recharge lorsqu'il quitte le mode `SMM`.

Pour faire court, théoriquement c'est donc *no pasaran*. Notons que le contrôle d'accès est clairement pensé de manière à prevenir un accès à la `SMRAM` depuis le processeur principal. En pratique, le contrôleur contrôlé, intéressons-nous aux échanges de données entre le mode `SMM` et le contrôleur.

3.3 L'enfer c'est les autres

Une fonction présente dans le code du handler `SMM` a retenu toute notre attention : celle responsable de la lecture d'un tampon en provenance du contrôleur et à destination du handler `SMI`. En pseudo-code nous aurions les fonctions suivantes :

- du coté `SMM` : `readbuffer(tampon, taille)`
- du coté contrôleur : `sendbuffer(tampon, taille)`

La taille des données échangées est codée en dur pour chaque partie, a priori tout va bien. Or, probablement du fait de contraintes matérielles, la transmission du tampon se fait par *chunk* d'une taille maximum de 8 octets. À chaque envoi, un octet précise la taille du *chunk* courant. Une forme très simplifiée de la fonction `readbuffer` est visible sur la figure 3.

```
readbuffer(char *tampon, unsigned short taille)
{
  while(taille != 0)
  {
    taille_chunk, chunk = readchunk()
    break if taille_chunk == 0

    tampon += chunk
    taille -= taille_chunk
  }
}
```

Figure 3. Fonction `readbuffer` simplifiée.

Si nous contrôlons la taille des *chunk* envoyés, il est possible de réaliser un *underflow* sur la variable taille, dont la conséquence directe est un *buffer overflow* sur le tampon. Nous contrôlons alors totalement la taille du tampon lu par le *handler* `SMI`. La fonction `readbuffer` est appelée à plusieurs endroits, toutefois un en particulier semble plus prometteur : le tampon passé en paramètre se situe sur la pile. *Smashing the stack*, à l'ancienne. Nous ne savons pas à quelle adresse se situe le *handler* `SMI`, en écrasant uniquement les deux octets de poids faible de l'adresse de retour stockée sur la pile ; nous sautons alors sur du code proche connu et re-dirigeons l'exécution vers le tampon contenant notre exploit avant de finalement restaurer sur l'adresse de retour originale. Nous disposons alors d'un petit *stager* de 20 octets (qui est exécuté en `SMM` (la fonction vulnérable échange un tampon de 16 octets et 4 supplémentaires sont obtenus en écrasant le *frame pointer* qui précède l'adresse de retour sur la pile).

Se pose un dernier problème : comment déclencher un appel à cette fonction vulnérable ?

Pour cela repartons des travaux de Loïc Duflot sur le mode `SMM` [6]. Il est précisé qu'un appel à l'instruction `OUT 0xB2` déclenche l'entrée en mode `SMM`. C'est en fait une

véritable interface exposée par le *handler* `SMI`, au système d'exploitation. De la même manière que pour le contrôleur, un ensemble de fonctions est exposé derrière cette interface. Lors de l'entrée en mode `SMM` le *handler* tente de déterminer la raison de l'entrée en `SMM`. Il vérifie notamment si la dernière instruction exécutée correspond à l'instruction `OUT 0xB2` ; dans ce cas la valeur contenue dans le registre `AL` sert d'index dans une table de pointeurs (les fonctions exportées).

La fonction vulnérable se trouve être accessible via cette interface. Lors de l'entrée en mode `SMM`, le contexte du processeur est sauvegardé dans la `SAVE_STATE_MAP`, qui est rechargée lors de la sortie du mode `SMM`. Les fonctions exposées par le *handler* peuvent éventuellement modifier cette `SAVE_STATE_MAP` afin de renvoyer des résultats (l'inverse est aussi vrai pour le passage d'argument).

4 It's a tarp !

Assemblons maintenant tous les éléments décrits précédemment. L'attaque a été implémentée de manière stable sur un portable muni d'un système d'exploitation Windows Seven 64-bit.

Pré-requis de l'attaque

- Droits administrateurs : exécution d'instruction d'entrées/sorties (`IN`/`OUT`)
- L'implémentation actuelle de l'attaque requiert également que le système d'exploitation soit démarré en mode *debug*, car nous tirons parti d'un pilote de `Windbg` [1].

Déroulement de l'attaque

- Injection du code malveillant sur le contrôleur :
  - Ajout d'une nouvelle fonction à l'interface, permettant l'écriture dans la `RAM` du contrôleur depuis le système d'exploitation
  - La fonction `send_buffer` est modifiée pour utiliser une taille de *chunk* différente
  - Un *hook* est installé sur la fonction `send_buffer` ; il est activé à la demande via un drapeau (situé dans la `RAM` du contrôleur). Lorsque le drapeau est activé, la fonction sert la charge utile malveillante de l'exploit (dont la taille provoque un dépassement de tampon), en lieu et place du tampon attendu

- La mise à jour requiert un redémarrage de la machine

- Du point de vue du système d'exploitation :
  - La charge utile de l'exploit est écrite dans la `RAM` du contrôleur clavier
  - Le drapeau précédemment décrit est activé
  - Une instruction `OUT 0xB2` (`AL` initialisé avec le numéro de la commande) est exécutée afin de déclencher l'appel, par le *handler* `SMI`, de la fonction vulnérable

L'exploitation est tout à fait stable, le *stager* est utilisé pour injecter nos propres commandes dans l'interface exposée par le *handler* `SMI`, nous permettant alors de communiquer directement avec le mode `SMM` sans passer par le contrôleur. Les trois commandes ajoutées correspondent aux primitives *read*, *write*, *execute*. Un petit wrapper Ruby de plus, nous permet d'interfacer un objet `Disassembler` de Metasm avec la `SMRAM`, et nous voilà un train d'introspecter dynamiquement la `SMRAM`, depuis un simple script exécuté en mode utilisateur.

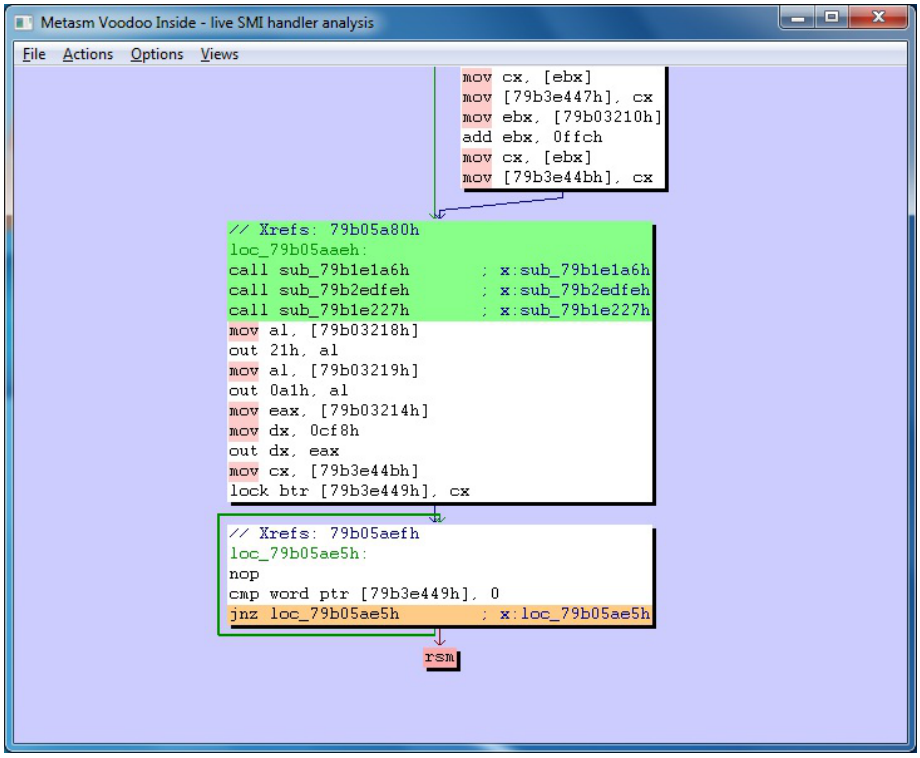

Figure 4. Introspection dynamique du handler `SMI`, depuis le mode utilisateur.

Limitations La portée de l'attaque doit être mise en perspective avec les contraintes relativement importantes portant sur l'environnement :

- Forte dépendance matérielle : chipset `ARC`, ports de communication non standards, protocoles de communication
- Forte dépendance logicielle : révisions du `BIOS` et de la `ROM` du contrôleur clavier
- Nécessite un redémarrage de la machine et des droits administrateurs

Toutefois les concepts mis en jeu sont eux tout à fait génériques. Problème de confiance surtout : le mode `SMM` fait confiance au contrôleur clavier (échange des données). Or, ce dernier est accessible avec des droits différents (et en un sens inférieurs) à ceux du mode `SMM`.

Le modèle de sécurité du `SMM` (en particulier le contrôle d'accès à la `SMRAM`) est mis en échec à cause d'une implémentation défaillante et d'une confiance indirecte accordée à un composant matériel du système. Comparée à l'état de l'art actuel, l'exploitation de la vulnérabilité est réellement triviale. Le code du *handler* `SMI` ne tire aucun profit de techniques de prévention d'exploitation qui tendent à se généraliser au niveau du système d'exploitation et de l'espace utilisateur.

Une fois le contrôleur clavier compromis et la vulnérabilité identifiée, les conditions d'exploitation sont idéales : un dépassement de tampon maîtrisé et aucune protection. Un attaquant obtient dès lors un contrôle total du système et peut réaliser des actions offensives telles que l'installation d'un *rootkit* [11] en tirant profit des caractéristiques du mode `SMM`. Nous avons ici le cocktail parfait avec une zone de stockage persistante particulièrement furtive dans le contrôleur, et une exécution toute aussi furtive dans la `SMRAM`).

5 Conclusion

La portée de l'attaque présentée est limitée car liée à de fortes contraintes au regard de la configuration matérielle et logicielle du système cible. Néanmoins les concepts étant génériques, rien n'interdit de penser que des vulnérabilités similaires pourraient être découvertes sur d'autres configurations. L'impact opérationnel est fort, offrant tout ce dont un rootkit avancé aurait besoin pour s'assurer furtivité et persistance.

Après les présentations de Loïc Duflot [5,6], Guillaume Delugré [3] ou encore Joanna Rutkowska [9], nous pouvons nous interroger sur la connaissance même que nous avons de notre propre matériel et la prise en compte des unités de calcul indépendantes dans les différents modèles de sécurité établis.

Je tiens à remercier Yoann Guillot pour l'incroyable support Metasm, Damien Aumaitre, Yann Le Glouahec et Damien Millescamps pour l'aide précieuse qu'ils m'ont apportée durant ce projet.

Références

1. Gazet Alexandre and Guillot Yoann. Metasm feelings. In *RECON*, 2010. http://metasm.cr0.org/docs/2010-recon-bintrace.pdf.
2. Apokrif. Dell bios, how to decompose/mod, 2010. http://forums.mydigitallife.info/threads/12962-Dell-bios-how-to-decompose-mod./page48.
3. Guillaune Delugré. Closer to metal : Reverse engineering the broadcom netextreme's firmware. In *Hack.Lu*, 2010. http://esec-lab.sogeti.com/dotclear/public/publications/10-hack.lu-nicreverse_slides.pdf.
4. dogber1, 2010. http://dogber1.blogspot.com/2010/05/dell-2a7b-keygen.html.
5. Loic Duflot and Olivier Grumelard. Et si les fonctionnalités des processeurs et des cartes mères pouvaient servir à contourner les mécanismes de sécurité des systèmes d'exploitation ? In *SSTIC*, 2006. http://www.sstic.org/media/SSTIC2006/SSTIC-actes/Contournement_des_OS_avec_les_fonctionnalites_des_/SSTIC2006-Article-contournement_des_OS_avec_les_fonctionnalites_des_processeurs_et_des_cartes_meres-duflot_grumelard.pdf.
6. Loic Duflot, Benjamin Morin, Olivier Levillain, and Olivier Grumelard. Getting into the smram : Smm reloaded. In *CanSecWest*, 2009. http://cansecwest.com/csw09/csw09-duflot.pdf.
7. Intel. Accessing the real time clock registers and the nmi enable bit. http://edc.intel.com/Link.aspx?id=1218.
8. Alfredo Ortega and Anibal Sacco. Deactivate the rootkit, 2009. http://www.blackhat.com/presentations/bh-usa-09/ORTEGA/BHUSA09-Ortega-DeactivateRootkit-SLIDES.pdf.
9. Joanna Rutkowska. A quest to the core. In *BlackHat*, 2009. http://www.invisiblethingslab.com/resources/misc09/Quest%20To%20The%20Core%20%28public%29.pdf.
10. General Software. Method for executing a 32-bit flat address program during a system management mode interrupt, patent no. : 7,444,5000 b1, 2001.
11. Filip Wecherowski. A real smm rootkit : Reversing and hooking bios smi handlers. In *Phrack 66*, 2009. http://www.phrack.org/issues.html?issue=66&id=1.
12. Ralf-Philipp Weinmann. The hidden nemesis : Backdooring embedded controllers. In *ToorCon*, 2010. http://sandiego.toorcon.org/index.php?option=com_content&task=view&id=49&Itemid=9.