Contrôle d'intégrité de la séquence de démarrage d'un ordinateur

Noël Cuillandre[1]*& F. Chabaud[2]

1: CFSSI
51, boulevard Latour-Maubourg
75700 Paris Cedex SP, France
noel.cuillandre@wanadoo.fr
2: SGDN/DCSSI/SDS/LTI
51, boulevard Latour-Maubourg
75700 Paris Cedex SP, France
florent.chabaud@polytechnique.org

Résumé

La séquence de démarrage d'un ordinateur commence à la mise sous tension de la machine et se termine lorsque le système d'exploitation est complètement chargé. On peut voir cette séquence comme une succession d'étapes qui mèneront l'ordinateur d'un état de repos (ordinateur éteint) à un état utilisable (système d'exploitation activé et prêt à exécuter les applications des utilisateurs).

Cet article propose une méthode de contrôle d'intégrité de la fin de la séquence de démarrage. Cette méthode a été développée sur un PC mono-processeur de type Intel équipé des systèmes d'exploitations Windows 95 et Linux, mais elle est généralisable à tout autre système d'exploitation, ainsi qu'à des architectures matérielles différentes.

1. Introduction

La sécurisation d'un ordinateur consiste à renforcer sa configuration pour assurer en particulier un cloisonnement entre les différents utilisateurs et entre les données auxquelles ils peuvent accéder. Toutefois, cette configuration est amenée à évoluer dans le temps, notamment suite à l'installation de nouveaux logiciels. Cette configuration peut aussi être altérée par un attaquant ayant réussi un instant à prendre le contrôle de la machine, pour lui permettre d'y accéder ultérieurement, même si la faille utilisée a disparu par suite de la mise à jour du logiciel vulnérable. Le contrôle de l'intégrité de la configuration de la machine est donc important pour permettre de vérifier régulièrement que cette configuration n'a pas été altérée. *A contrario*, détecter une altération de la configuration peut constituer un signal indiquant que la machine a subi une attaque. L'objectif ici est donc d'empêcher qu'un attaquant ayant réussi un instant à prendre le contrôle total de la machine protégée, puisse avoir la possibilité d'effectuer des modifications de configuration indétectables sur cette machine.

La configuration d'un ordinateur est principalement contenue dans le système de fichiers utilisé par le système d'exploitation. Pour contrôler cette configuration, de nombreux outils existent dont le plus connu est TRIPWIRE [KS94]. Ce logiciel permet en effet de sauvegarder l'empreinte cryptographique d'un ensemble de fichiers à un instant t. Ultérieurement, il est alors possible de contrôler que les fichiers considérés n'ont pas été modifiés. Toutefois, pour que ce contrôle apporte une certaine garantie, il faut qu'il soit effectué indépendamment du système d'exploitation contrôlé. En effet, si le système d'exploitation utilisé est altéré,

* Mis à disposition par le Ministère de la Défense (Armée de l'Air) pour un stage au centre de formation à la sécurité des systèmes d'information.

il peut très bien camoufler les modifications effectuées sur sa configuration. Un logiciel comme TRIPWIRE est donc à utiliser en initialisant la machine sur un système d'exploitation sûr.

Reste que même si le logiciel de type TRIPWIRE est utilisé dans un système d'exploitation sûr, rien ne prouve que la machine dont la configuration a été contrôlée va effectivement initialiser cette configuration. En effet, la séquence de démarrage d'un ordinateur, qui est partie intégrante de cette configuration, n'est pas située dans le système de fichiers et n'est donc pas directement contrôlable. Cette séquence est enregistrée au niveau de certains secteurs du disque et n'est pas accessible à travers le système de fichiers de haut niveau du système d'exploitation. Une altération de cette séquence pourrait ainsi être parfaitement indétectable par des outils comme TRIPWIRE, tout en permettant à un attaquant d'initialiser un autre système d'exploitation que celui initialement installé et contrôlé.

2. Séquence de démarrage d'un ordinateur

La séquence de démarrage d'un ordinateur est un processus très variable selon le matériel utilisé. Certaines cartes de communication peuvent par exemple piloter la mise sous tension de l'ordinateur sur réception d'un signal en provenance du réseau (Ethernet, modem, etc.). Toutefois, dans le cas des ordinateurs de type PC, la séquence de démarrage proprement dite reste telle que spécifiée dans la description d'architecture [PCat84, PCxt86] et peut être découpée en trois étapes successives.

2.1. Du processeur au BIOS

La première étape est caractérisée par la prédominance d'événements de type matériel :

1. À la mise sous tension de la machine, la carte mère envoie des signaux électriques à destination du processeur afin de le mettre dans un état particulier `RESET HARDWARE` (cf. [IntelSys99][page 8.1]).
2. Le processeur Intel entre alors en phase d'initialisation matérielle. Cette phase (cf. [IntelSys99][page 8.1]) a pour but de mettre les registres du processeur dans un état prédéfini et de basculer le processeur en mode d'adressage réel.
3. Le processeur peut alors exécuter un test interne, le "*Built-In Self-Test*" (BIST), sur demande de la carte mère. À l'issue du test, une valeur nulle dans le registre EAX indique qu'aucune erreur n'a été détectée dans le processeur (cf. [IntelSys99][page 8.2]).
4. Le processeur va ensuite exécuter la première instruction qui se trouve à l'adresse physique `0xFFFFFFF0`. À cette adresse se trouve une instruction de saut vers le code d'initialisation du BIOS (cf. [Phoenix91][pages 95-100]).

Bien que la séquence ici décrite soit celle d'ordinateurs de type PC, ce type de séquence est assez universel puisque la mise sous tension provoque en règle générale le lancement d'un programme à une adresse fixée. Les idées développées ci-après peuvent donc être adaptées en fonction des spécifications de telle ou telle machine, même si l'utilisation, par exemple, d'un processeur au jeu d'instructions différent nécessitera une adaptation.

2.2. Le rôle du BIOS

La seconde étape de la séquence de démarrage commence ensuite, elle a pour but de mettre en place les structures de données nécessaires aux fonctions de gestion élémentaires de la machine. Cette étape se caractérise par l'exécution en mode réel de code externe au processeur. Ainsi le BIOS peut se voir comme une interface élémentaire entre le matériel et le logiciel (système d'exploitation et/ou application indépendante).

Une des tâches les plus importantes attribuée au BIOS est de mettre en place la table des vecteurs d'interruption. Cette table, qui débute à l'adresse physique `0x00000000`, est constituée de 256 mots de 32 bits. Chaque mot représente, sous la forme d'un couple (segment,offset), l'adresse physique d'une routine qui sera exécutée lorsque l'interruption ou l'exception concernée est déclenchée. Les interruptions sont des opérations provoquées soit par les composants matériels de la machine, soit par le logiciel en cours d'exécution. Elles permettent de suspendre l'exécution d'une instruction ou d'une application afin d'effectuer une action particulière puis de reprendre la suite normale de l'exécution de l'instruction ou de l'application.

Modifier cette table de vecteurs d'interruption permet par exemple de détourner les modes d'accès aux différents périphériques (pour poser un espion clavier, par exemple). Il est donc indispensable, dans notre approche, que le BIOS soit lui-même intègre puisque nous allons, au cours de la simulation, utiliser certaines de ces interruptions.

2.3. Du BIOS au système d'exploitation

Lorsque le BIOS a terminé son initialisation, il exécute l'instruction INT $19h$. Le processeur va alors lancer l'exécution du code pointé par l'entrée $19h$ de la table des vecteurs d'interruption. Ce sous-programme, appelé "*boostrap loader*", a pour but de chercher le premier périphérique bootable selon un ordre prédéfini et modifiable par le menu de configuration du BIOS. Si un disque bootable est trouvé, le premier secteur du disque est lu et copié en mémoire système à l'adresse `0000:7C00h` (cf. [Phoenix91][page 101]). Le BIOS exécute alors la première instruction située à cette adresse. Cette première instruction fait partie d'un bloc exécutable d'au plus 512 octets, qui a pour but de trouver le système d'exploitation sur le périphérique, de le charger en mémoire et de lancer son exécution. Ce bloc exécutable est mis en place soit lors de l'installation d'un système d'exploitation, soit lors de l'installation d'une application de gestion de démarrage.

Dans le cas d'une machine MS-DOS, le bloc exécutable de 512 octets est le "*Master Boot Record*" ou DOS-MBR, qui va ensuite charger le premier secteur de la partition active, lequel va charger le fichier noyau `command.com`. Pour une machine linux, on trouve souvent installé le logiciel LILO [Alm98t] dans une configuration qui remplace le DOS-MBR par un LILO-MBR, lequel appelle ensuite une seconde phase de chargement stockée à d'autres endroits du disque, mais toujours inaccessible au système de fichiers de haut niveau. La finalité de ces processus est justement de permettre à ces petits programmes de chargement d'accéder au programme principal, le noyau du système, sans disposer du système de fichiers.

Quel que soit le BIOS utilisé, c'est au départ ce MBR qui est chargé et qui constitue donc la première étape de la séquence d'initialisation à contrôler.

3. Intégrité de la séquence de démarrage

Nous avons vu que la séquence de démarrage d'un ordinateur fait intervenir successivement trois acteurs principaux : le processeur, le BIOS et le système d'exploitation. Toute modification de l'intégrité de l'une de ces étapes est susceptible de permettre le contournement par un attaquant éventuel des dispositifs de sécurité du système d'exploitation lancé, comme l'ont démontré les virus de boot [Bon91, Bis92] et les détournement des appels systèmes [Ben01].

Dans [AKS97] puis dans [AKFS98] a été proposée une méthode de construction d'une séquence d'initialisation sécurisée par chaînage des différentes opérations. Chaque étape vérifie l'intégrité de l'étape suivante par une signature cryptographique. Moyennant l'hypothèse que le début de la séquence est intègre, on parvient ainsi à garantir l'intégrité de l'ensemble de la séquence. Cette approche présente toutefois l'inconvénient de ne pas pouvoir être appliquée à une séquence d'initialisation existante. Elle suppose au minimum le *flashage* du BIOS et l'installation d'outils de lancement spécifiques.

Une nouvelle approche est donc proposée pour contrôler la dernière partie de la séquence de démarrage, le lancement du système d'exploitation par le BIOS, sans avoir à modifier celle-ci. L'hypothèse initiale reste nécessaire, à savoir que le code du BIOS est supposé intègre.

3.1. Principe

L'idée de base est de démarrer un système sain extérieur (sur une disquette et/ou sur cédérom) et de contrôler l'intégrité des périphériques bootables de la machine. Cette idée est similaire à celle d'un contrôleur de fichiers comme TRIPWIRE [KS94]. Le mode d'utilisation de notre programme sera donc similaire à celui de TRIPWIRE et il sera même intéressant de réaliser les deux contrôles de façon simultanée. Comme dans le cas de TRIPWIRE, un premier calcul permet d'obtenir l'empreinte de la séquence d'initialisation supposée intègre. Cette empreinte sera stockée sur un média amovible garantissant son intégrité. Par la suite, c'est par comparaison à cette empreinte que l'intégrité de la séquence d'initialisation sera vérifiée. Comme dans la cas de TRIPWIRE, un contrôle périodique et automatique peut être envisagé, mais il ne permet pas de garantir l'intégrité de la séquence. Seul un redémarrage de la machine contrôlée avec un système d'exploitation sain (boot sur disquette par exemple) permet un contrôle valide.

Mais à la différence de TRIPWIRE, il est nécessaire pour le contrôle de la séquence d'initialisation d'adopter une approche dynamique. En effet, le simple contrôle statique du MBR (d'ailleurs effectué de façon native par certains BIOS) ne suffit pas, puisque le programme contenu dans ce MBR va appeler d'autres programmes situés dans d'autres secteurs du périphérique, voire dans d'autres périphériques. Contrôler globalement ces périphériques n'est pas possible puisqu'ils contiennent les données utilisateurs et sont en perpétuelle modification. Il convient donc de ne contrôler que le code effectivement exécuté depuis la première instruction à l'adresse `0000:7C00h` jusqu'au lancement du système d'exploitation. Or, selon les programmes chargeurs et leurs configurations, ce code peut se retrouver à tout endroit. Une configuration statique n'est donc pas envisageable et seule une interprétation du code va nous permettre de recoller les différents morceaux de la séquence d'initialisation.

3.2. Simulateur

Pour savoir exactement quel code est impliqué dans la séquence d'initialisation, un simulateur capable d'interpréter toutes les instructions utilisables en mode de fonctionnement réel du processeur a été développé. L'emploi d'un simulateur permet d'exécuter le code dans un environnement sûr (principe du bac à sable employé sous Java [Gon98]). Ainsi, même si le code interprété a été modifié par un virus ou un autre moyen, aucune action dangereuse pour le système ne sera réellement accomplie. En outre, l'utilisation d'un simulateur s'impose aussi pour interpréter le code dans le contexte d'initialisation de la machine, qui est celui d'un processeur en mode réel.

Afin d'avoir un bac à sable sûr, il est nécessaire de simuler l'action de certaines interruptions, telles que la lecture du clavier, et d'isoler les données qui sont en mémoire. Ainsi, une structure de données représentant la mémoire vive (RAM) est mise en place et contient toutes les données lues depuis le périphérique et/ou qui doivent être écrites en RAM.

Par conséquent, la seule action autorisée sur les périphériques réels est la lecture de secteurs du périphérique concerné. Cette action utilisant une interruption au niveau du BIOS, il est indispensable que ce dernier soit intègre, tout du moins en ce qui concerne les accès en lecture aux disques.

3.2.1. La machine virtuelle

La machine virtuelle est constituée d'un processeur et d'une mémoire principale. Le processeur est une structure de données qui représente les registres internes nécessaires au fonctionnement en mode réel d'un processeur Intel [IntelArc97, chapitre 3]. La mémoire principale est représentée par une liste doublement chaînée de tableaux d'octets qui représentent chacun une page mémoire de 512 octets consécutifs. Les pages mémoires sont classées dans la liste par adresse physique croissante. Lors de la simulation, toutes les opérations de lecture et d'écriture en mémoire se font dans cette liste. Pour les opérations d'écriture, les résultats sont sauvegardés dans la liste avec insertion d'une nouvelle page mémoire si nécessaire. Pour les opérations de lecture, une erreur est déclenchée si aucune des pages mémoires existantes ne contient l'adresse recherchée.

3.2.2. Simulation du code

L'initialisation du simulateur consiste principalement à mettre le processeur virtuel en mode de fonctionnement réel et à copier le MBR du périphérique concerné dans sa mémoire principale. Dans le cas d'une séquence d'initialisation d'un autre type que la séquence étudiée ici, le chargement s'effectuerait de façon similaire au niveau du périphérique contenant le début de la séquence de boot (EEPROM, cédérom, disquette, etc.) puisque ce périphérique doit de toute façon être accessible au processeur et au BIOS lors du démarrage.

À l'issue de la procédure d'initialisation, le pointeur d'instruction du processeur virtuel pointe sur le premier octet du MBR. Dès lors, le simulateur va entrer dans une boucle d'évaluation des instructions que l'on peut résumer par la procédure suivante :

```
/*
 * Procédure d'évaluation d'une instruction:
 * [IN]
 *   pip: pointeur sur le processeur virtuel.
 * [IN/OUT]
 *   ptr: pointeur sur le bloc de code à simuler.
 */

int sim(intel_processor * pip,byte ** ptr)
{
  intel_instruction ii;         /* instruction à simuler     */
  byte bytes_eaten;             /* nombre d'octets consommés *
                                 * par l'instruction         */

                                /* DECODAGE de l'instruction */
  memset(&ii,0,sizeof(intel_instruction));
  get_intel_instruction(SEG_16BITS,*ptr,&ii,&bytes_eaten,ptr);
                                /* MISE À JOUR du pointeur   *
                                 * d'instruction virtuel EIP */
  pip->EIP+=bytes_eaten;

  do_signature(pip,&ii);        /* SIGNATURE du code simulé  */

                                /* EVALUATION de l'inst.     */
  eval_intel_instruction(pip,&ii);
                                /* MISE À JOUR du pointeur   *
                                 * d'instruction réel        */
  intel_set_pointer_to_CSEIP(pip,(void **)ptr); /* ptr=CS:EIP*/

  return 1;
}
```

Dans cette procédure, la fonction *get_intel_instruction* a pour but de décoder le bloc d'octets pointés par *ptr* en un code d'opération valide suivi éventuellement d'une à trois opérandes, conformément au format défini par Intel [IntelRef97]. Puis la fonction *eval_intel_instruction* simule l'exécution de l'instruction décodée par le processeur virtuel. À la différence d'un désassembleur standard, le décodage ne se fait pas de façon linéaire lorsqu'une instruction de branchement est évaluée :

branchement inconditionnel : l'interprétation se poursuit à l'adresse indiquée en paramètre.

branchement conditionnel : le registre d'état EFLAGS est d'abord évalué, puis le décodage se poursuit à l'adresse indiquée en paramètre si la condition est vérifiée. Sinon, le décodage se poursuit avec l'instruction qui suit le branchement.

Cette façon de procéder permet d'une part de n'évaluer que les chemins d'exécutions possibles et d'autre part de travailler dans des cas où le code exécutable est fragmenté au milieu de données, ce qui est souvent le cas des séquences d'initialisation.

3.3. Empreinte Cryptographique

Bien entendu, tout le problème consiste à générer de façon déterministe une empreinte cryptographique de la séquence de démarrage malgré les branchements conditionnels présents dans le programme. Cette empreinte cryptographique servira ensuite à contrôler l'intégrité du programme de démarrage, plus précisément à garantir que seuls les chemins d'exécutions initialement prévus peuvent être exécutés. Pour cela, à chaque interprétation d'instruction, une fonction *do_signature* est appelée, qui calcule au fur et à mesure l'empreinte.

Cette empreinte cryptographique peut être calculée en utilisant une fonction de hachage (par exemple [Riv92, SHA95]). Ainsi, toute modification dans le chemin d'exécution, telle qu'un branchement conditionnel non pris ou une itération en plus ou en moins dans une boucle, impliquera une modification de l'empreinte cryptographique.

3.4. Quand s'arrêter ?

3.4.1. Explosion combinatoire

Un des problèmes principaux est de faire face à l'explosion combinatoire dans le cas où les chemins autorisés sont nombreux (cas typique d'une boucle de saisie). En effet, dans ce cas, un unique chemin d'exécution ne peut être déterminé. Afin de pallier cette difficulté, l'ensemble des chemins possibles est pris en compte. Dans un premier temps, le simulateur passe en mode exhaustif lorsque l'interruption clavier est utilisée. Ensuite, une procédure cartographie les chemins d'exécutions possibles :

```
BOUCLE {
    SI instruction de branchement {
        SI non déterministe {
            SI non déjà visité OU contexte différent {
                ->sauvegarde du contexte de simulation
                  (structure processeur, mémoire) sur
                  la pile de chemins
            }
        }
    }

    simulation instruction courante
    instruction suivante
}
```

Il faut remarquer que seules les instructions de branchement non déterministes, c'est-à-dire dépendant d'une interaction extérieure, sont susceptibles de poser problème. Si, par exemple, le chargeur calcule un CRC sur le noyau qu'il est sur le point de charger, les données utilisées seront disponibles à notre simulateur et le fonctionnement sera déterministe.

L'exhaustivité du parcours de l'arbre est toutefois difficile à assurer si le nombre d'options possibles est trop grand. On peut toutefois légitimement supposer que sur une machine sécurisée, le nombre de configurations possibles devra être restreint ; il est même probable que la séquence de démarrage soit parfaitement déterministe.

3.4.2. Détection de fin de programme

Le programme qui est interprété dans le simulateur est une séquence d'initialisation d'un système d'exploitation, c'est-à-dire un programme qui a vocation à ne jamais s'arrêter. Il est donc nécessaire de trouver un point d'arrêt pertinent à notre programme de calcul d'empreinte cryptographique de la séquence d'initialisation.

Dans le cas des composants Intel, le choix a été fait d'arrêter le calcul de l'empreinte cryptographique lorsque le système d'exploitation fait passer le processeur en mode protégé.

La documentation Intel[IntelSys99][section 8.8.1], indique que le passage du processeur en mode protégé se fait par l'exécution d'une instruction MOV CR0 qui active le drapeau PE du registre d'état EFLAGS. C'est donc lorsque cette séquence est détectée que la signature calculée est considérée comme définitive.

L'utilisation du mode protégé permet un contrôle plus aisé de la mémoire par le système d'exploitation. Il est quasi-indispensable à tout système d'exploitation d'utiliser ce mode et le passage en mode protégé est donc un bon indicateur du fait que le simulateur est en train d'exécuter le code propre du noyau. Ce code est normalement contenu dans un fichier accessible, quant à lui, au niveau du système de fichiers de haut niveau. Le recouvrement ainsi obtenu entre le contrôle d'intégrité de la séquence de démarrage et le contrôle d'intégrité du système de fichiers réalisé à l'aide d'outils comme TRIPWIRE permet, en théorie, de garantir l'intégrité de l'ensemble de la chaîne d'initialisation.

3.4.3. Intégrité du noyau

Même si en théorie, il est ainsi possible de contrôler l'intégrité de la séquence d'initialisation du noyau, en pratique, si le noyau charge dynamiquement des modules ou pilotes qui ne sont pas contrôlés en intégrité, alors il n'y a plus de garantie d'intégrité. C'est la raison pour laquelle des approches complémentaires comme LOMAC [LOMAC] ont été tentées.

Le projet LOMAC [LOMAC] représente une tentative de mise en place d'un contrôle d'accès obligatoire (MAC) non pénalisant pour les utilisateurs. Le contrôle d'accès obligatoire est dérivé du modèle "*Low Water-Mark*" [Bib77] et est implanté par un module chargeable dans une version 2.2 du noyau Linux (LKM). Sous LOMAC, le système est divisé en deux niveaux d'intégrité : haut et bas. L'idée est que les processus et fichiers de niveau bas représentent une menace potentielle pour l'intégrité du système. Les mécanismes de protection de LOMAC et du noyau s'ajoutent : la protection offerte par LOMAC est uniquement basée sur un niveau d'intégrité et non sur l'identification d'un utilisateur. Une opération sera permise seulement si LOMAC et le noyau décident de l'autoriser. Depuis juin 2001, un essai d'implémentation d'un LOMAC avancé avec plus de 2 niveaux est en cours de réalisation par NAI Labs [NAI]. Dans [Fra00] et [Fra01], Timothy Fraser, le principal concepteur de LOMAC, mentionne ses avantages et inconvénients par rapport à d'autres projets existants, dont KS OS [CD79], DTE [BSSW96], UCLA Secure Unix [PKKSUW79], et Secure-Enhanced Linux [LS01, SELINUX].

4. Exemple d'utilisation sur LILO

Lors de la mise au point du simulateur, de nombreux essais ont été réalisés sur le gestionnaire de démarrage pour Linux, LInux LOader [Alm98t]. Ce gestionnaire de démarrage permet de lancer jusqu'à 16 systèmes d'exploitations.

Lorsque LILO est installé comme gestionnaire de démarrage principal, le MBR contient le *first stage boot loader*. Cette portion de code a pour but de se transférer en mémoire, de charger et d'exécuter le *second stage boot loader* qui est la partie la plus importante puisqu'elle permet l'interaction avec l'utilisateur et le choix du système d'exploitation à lancer.

Nous avons vu que certains BIOS offrent une protection en écriture du MBR. Dans le cas de LILO, on ne peut garantir l'intégrité de la troisième étape de la séquence de démarrage avec ce type de protection. En effet, LILO charge en mémoire de nombreux secteurs du périphérique qui contiennent, en plus du code exécutable, les données nécessaires à la localisation des secteurs de démarrage des systèmes d'exploitation.

L'examen réalisé des deux étapes de LILO par le simulateur a permis de mettre en évidence une incohérence dans la programmation du menu utilisateur en mode texte :
- lorsque l'utilisateur commande l'affichage des systèmes d'exploitations disponibles, le programme ne parcourt pas les 16 champs possibles, mais s'arrête au premier champ vide détecté ;
- lorsque l'utilisateur entre le nom d'un système d'exploitation, le programme vérifie systématiquement les 16 champs.

Cette incohérence permet d'insérer une séquence d'initialisation invisible à l'utilisateur (non proposée dans l'interface) mais malgré tout accessible (l'option correspondante peut être saisie au clavier, elle sera activée). Par exemple, l'attaquant peut insérer une option `toto` qui va charger un noyau qu'il a lui-même modifié pour y avoir accès en toute circonstance. Ce noyau sera bien entendu situé dans la zone des données utilisateurs non contrôlable en intégrité.

L'activation de cette vulnérabilité nécessite un redémarrage de la machine et le code LILO pourra facilement être corrigé. Toutefois, même sans correction, l'utilisation de la méthode de contrôle proposée permettrait de détecter le branchement conditionnel supplémentaire qu'un attaquant pourrait ajouter en utilisant cette faiblesse, puisque l'empreinte finale de la séquence d'initialisation en serait altérée.

5. Conclusion

Nous avons présenté une méthode de contrôle de l'intégrité de la séquence de démarrage d'un ordinateur de type PC depuis le lancement par le BIOS d'un secteur de démarrage MBR jusqu'au passage en mode protégé du processeur, supposé effectué par le noyau du système d'exploitation. Cette méthode suppose que le BIOS de la machine est intègre. Elle utilise un simulateur de processeur qui offre l'avantage de "suivre" l'exécution du code jusqu'au lancement du système d'exploitation, sans effectivement réaliser les opérations demandées. Un code viral qui se serait inséré dans la séquence d'initialisation ne serait donc pas réellement exécuté et ne pourrait empêcher sa détection.

Avec l'utilisation simultanée d'un système de contrôle d'intégrité au niveau du système de fichiers, il est ainsi possible de détecter toute modification de la séquence de démarrage. À titre d'exemple, une incohérence dans le programme LILO a été mise en évidence, qui permettrait à un attaquant d'insérer une option d'initialisation non présentée à l'utilisateur au niveau de l'interface de LILO. Cette option de branchement invisible serait toutefois détectée par la méthode proposée.

Remerciements

Nous tenons à remercier les referees pour l'ensemble de leurs commentaires qui ont grandement contribué à l'amélioration espérée de cet article.

Références

[AKFS98] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, et J. M. Smith. Automated recovery in a secure bootstrap process. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security (SNDSS)*, pages 155-167, Mars 1998.

[AKS97] W. A. Arbaugh, D. J. Farber, et J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the IEEE Security and Privacy Conference*, pages 65-71, Mai 1997.

[Alm98t] W. Almesberger. LILO *Technical Overview*, version 21, December 98.

[Alm98u] W. Almesberger. LILO *User's Guide*, version 21, December 98.

[Ben01] W. C. Benton. Loadable kernel module exploits. *Linux Journal*, septembre 2001.

[Bib77] K.J. Biba. *Integrity Considerations for Secure Computer Systems*, Electronic Systems Division, Hanscom Air Force Base, Bedford, MA, pages 27-31, Avril 1977.

[Bis92] M. Bishop. *An Overview of Computer Viruses in a Research Environment*, 1992.

[Bon91] V. Bontchev. The Bulgarian and Soviet virus factories. In *Proceedings of the First International Virus Bulletin Conference*, Buckinghamshire, 1991.

[BSSW96] L. Badger, D.F. Sterne, D.L. Sherman, et K.M. Walker. A domain and type enforcement UNIX prototype. *USENIX Computing Systems*, 1996.

[CD79] E.J. McCamley et P.J. Drongowski. KSOS—The design of a secure operating system. In *Proceedings of the National Computer Conference*, vol. 48, AFIPS Press, pages 345-353, 1979.

[Fra00] T. Fraser, LOMAC : Low water-mark integrity protection for COTS environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, 2000.

[Fra01] T. Fraser, LOMAC : MAC you can live with. In *Proceedings of the FREENIX Track : USENIX Annual Technical Conference*, Boston, Massachusetts, juin 2001.

[Gon98] L. Gong. *Java 2 Platform Security Architecture*, Sun Microsystems Inc., 1998.

[IntelArc97] *Intel Architecture Software Developer's Manual*, Volume 1 : Basic Architecture, 1997.

[IntelRef97] *Intel Architecture Software Developer's Manual*, Volume 2 : Instruction Set Reference, 1997.

[IntelSpe96] *Pentium Pro Family Developer's Manual*, Volume 1 : Specifications, 1996.

[IntelSys99] *Intel Architecture Software Developer's Manual*, Volume 3 : System Programming, 1999.

[KS94] G. H. Kim, E. H. Spafford. The design and implementation of Tripwire : A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*. ACM Press, New York, 1994.

[LOMAC] T. Fraser, Low water-mark mandatory access control (LOMAC) v1.1.2. `http://opensource.nailabs.com/lomac/`, février 2002.

[LS01] P.A. Loscocco, S.D. Smalley, Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track : USENIX Annual Technical Conference*, Juin 2001.

[NAI] Network Associates Inc., `http://opensource.nailabs.com/`.

[PCat84] International Business Machines Corporation. *IBM Personal Computer AT Technical Reference*, Boca Raton, FL, 1984.

[PCxt86] International Business Machines Corporation. *IBM Personal Computer XT and Portable Personal Computer Technical Reference*, Boca Raton, FL, 1986.

[Phoenix91] Phoenix Technologies Ltd. *System BIOS for IBM PCs, Compatibles and EISA Computers*. Addison Wesley, deuxième édition, 1991.

[PKKSUW79] G.J. Popek, M. Kampe, C.S. Kline, A. Soughton, M. Urban, E.J. Walton, UCLA secure UNIX. In *Proceedings of the National Computer Conference*, vol. 48, AFIPS Press, page 355-364, 1979.

[RC95] M. Russinovich et B. Cogswell. *Examining the Windows 95 Layered File System*, Dr Dobb's Journal, décembre 1995.

[Riv92] R.L. Rivest. *RFC 1321 : The MD5 Message-Digest Algorithm*, Internet Activities Board, 1992.

[SELINUX] P. Loscocco. *Security-Enhanced Linux*, Information Assurance Research Group, National Security Agency, Avril 2001.

[SHA95] FIPS 180-1. *Secure Hash Standard*, US Department of Commerce/NIST, 1995.