Analyse et détection de logiciels malveillants

Aurélien Palisse et Jean-Louis Lanet

LHS ⋆ – PEC ⋆⋆ INRIA
[aurelien.palisse, jean-louis.lanet]@inria.fr

Résumé L'accroissement du nombre de logiciels malveillants ainsi que les techniques de furtivité employées permettent d'infecter de nombreuses cibles. Les mécanismes de protection de plus en plus complexes les rendent difficiles à analyser et détecter. Nous proposons d'étudier une méthode permettant de contrôler les accès mémoires d'un programme et ainsi en extraire la charge utile en cas de paquetage du binaire. La détection d'un éventuel comportement malveillant s'appuie sur l'apprentissage automatique. Nous garantissons une intégration transparente pour les applications dans l'espace utilisateur ainsi qu'un faible impact sur les performances. Cette approche bas niveau a pour objectif de minimiser les effets de bord sur le système.

Mots-clés : programmes malveillants, analyse dynamique, gestion mémoire, dépaquetage, apprentissage automatique

1 Introduction

Les logiciels malveillants utilisent le plus souvent un "loader" ou chargeur permettant de contourner les techniques de détection des antivirus. Ce chargeur encapsule le binaire original et contient des techniques d'obfuscation qui rendent l'analyse statique du binaire très difficile. Le paquetage d'un binaire permet d'obfusquer à la fois les données qu'il contient mais aussi son code en y ajoutant par exemple un moteur métamorphique ou du code auto-modifiant. Il n'est plus nécessaire d'utiliser un compilateur dédié à la génération de codes polymorphiques ou métamorphiques en amont de la phase d'empaquetage, les empaqueteurs actuels tels que *Themida* sont capables d'agir directement sur le binaire. La possibilité d'obtenir facilement et à faible coût de nombreux empaqueteurs pour les développeurs de logiciels malveillants complique la tâche des entreprises d'antivirus.

Les protections mises en place par les logiciels malveillants sont telles qu'il est préférable de laisser le binaire s'exécuter tout en contrôlant à son insu l'environnement. Lors de son exécution le chargeur extrait le code malveillant de manière progressive par "couches" ou d'une seule traite. Les techniques employées par les empaqueteurs sont pour la plupart connues mais il est difficile d'automatiser

⋆. Laboratoire de Haute Sécurité
⋆⋆. Pôle d'Excellence Cyber

l'extraction du binaire original notamment à cause de la volatilité des paramètres pris en compte lors du paquetage. Pour ne pas être détecté il est nécessaire de conserver une liste à l'état de l'art de l'ensemble des protections utilisées et des contre-mesures associées, le travail de veille doit être permanent.

L'utilisation d'une plateforme d'analyse dynamique ne doit pas modifier le comportement d'un binaire. Celui-ci peut contenir des techniques anti-vm, anti-reverse et anti-debug fastidieuses à éliminer manuellement qui biaise l'analyse du binaire et la rend difficile. De plus, utiliser un environnement virtualisé laisse de nombreux artefacts qu'il est impossible d'éliminer entièrement, notamment pour des raisons d'architecture matérielle. Il est donc nécessaire de se placer au plus proche du matériel et du système d'exploitation dans un environnement non virtualisé. On espère ainsi observer le comportement malveillant une fois le dépaquetage en RAM effectué. On peut ensuite, à l'aide d'un mécanisme de gestion de la mémoire "dumper" l'espace mémoire correspondant au binaire malveillant. Un problème reste malgré tout d'actualité, le code ainsi obtenu peut encore contenir de nombreuses traces d'obfuscation rendant le désassemblage couteux en temps.

2 Travaux connexes

De nombreux travaux mettent en avant l'utilisation de la virtualisation (e.g, *Ether* [4], [5, 8]) avec des logiciels comme QEMU ou Xen. Le problème majeur de cette approche est qu'il est souvent très complexe et couteux en temps de s'engager dans le développement d'une telle solution. De plus, il est possible de détecter un environnement virtualisé sans qu'aucune véritable contre-mesure n'existe, c'est le cas pour les attaques par "timing detection" même si des solutions ad-hoc émergent [7]. Une nouvelle approche est apparue avec l'utilisation d'un hyperviseur s'exécutant directement sur le matériel sans qu'il soit hébergé par un système d'exploitation [18]. Malgré une nouvelle étape franchie vers plus de transparence, l'interception de certaines instructions par l'hyperviseur qu'elles soient privilégiées ou non permettent de détecter un tel environnement [19].

Une approche que nous jugeons plus pragmatique consiste à se placer au niveau de l'OS sous la forme d'un driver. Il est ensuite possible de contrôler les accès mémoire en utilisant des techniques similaires à celles d'un rootkit [10, 13, 14]. On bénéficie également de l'API offerte par le DDK [1] lors du développement et de davantage de support. Cette approche est limitée à la détection de logiciels malveillants dans l'espace utilisateur ou du moins ceux qui utilisent un chargeur qui s'exécute dans cet espace (ring 3). Un exploit zero-days qui permettrait d'injecter directement du code dans le noyau ne peut pas être contrecarré de manière certaine avec une telle solution.

1. Driver Development Kit

Les systèmes d'exploitation moderne utilisent la pagination comme mécanisme de gestion mémoire, chaque processus peut ainsi adresser l'ensemble de la mémoire vive disponible et même davantage. La résolution d'une adresse logique vers une adresse physique n'est pas uniquement réalisée par le processeur (MMU), en cas de "page fault" le système d'exploitation intervient pour charger la page en mémoire. Pour travailler en collaboration, la MMU et le système doivent maintenir des structures de données en mémoire. La segmentation étant quasi inexistante sur Windows avec un modèle dit "flat", la protection de la mémoire repose en grande partie sur la pagination (U/S, W).

Il est possible d'activer une protection supplémentaire pour les versions 32-bits de Windows présente "nativement" en 64-bits. Il s'agit du bit NX pour *Never eXecute* qui permet de marquer une zone mémoire comme non exécutable. Les processeurs x86 prennent en charge cette fonctionnalitée depuis une dizaine d'années, pour l'utiliser il faut charger un noyau Windows prenant en charge PAE [2]. Pour des raisons de compatibilité les versions récentes de Windows n'activent pas le bit NX par défaut pour les exécutables 32-bits tiers (OptIn) même si le processeur prend en charge cette protection [17].

Le contrôle des accès mémoires est utilisé par l'ensemble des solutions de dépaquetage pour observer le comportement d'un binaire lors de son exécution. L'objectif est en fait de simuler la protection W⊕X (voir figure 1) pour l'ensemble des processus et ce quelque soit l'architecture. En marquant chaque page comme étant accessible uniquement en écriture ou exécutable on devrait assister aux différentes phases de dépaquetage d'un binaire.

La détection d'un comportement malveillant dans la littérature repose sur des heuristiques plutôt simples telles que la surveillance des appels systèmes ou la détection d'un éventuel saut vers une adresse mémoire précédemment extraite par le chargeur ou lointaine. Une approche alternative consiste à utiliser l'apprentissage automatique [16] lors de l'exécution mais aussi directement sur des binaires [6, 22].

3 Propositions

Notre objectif est de concevoir un framework embarquant un module responsable du contrôle des accès mémoire via un driver et un second capable de détecter une menace. Les modules étant indépendants, ils favorisent la réutilisation et la portabilité du framework. Nous visons le système d'exploitation Windows 7, celui-ci étant le plus utilisé actuellement. Nous choisissons de développer notre solution directement sur "bare-metal", les techniques employées seront similaires à celles d'un rootkit. Nous allons nous focaliser davantage sur les interactions d'un programme avec son environnement plutôt que sur sa manière de les réaliser.

2. Physical Address Extension

Nous incluons également la possibilité de “dumper” un binaire suspect qui requiert une analyse manuelle approfondie et ce en approximant au mieux la fin du dépaquetage.

La détection quant à elle va utiliser l'apprentissage automatique et ce sur du code dépaqueté lors de l'exécution. Plutôt que de placer ce composant dans l'espace utilisateur comme c'est souvent le cas, nous choisissons de l'intégrer dans un “export driver”, il s'agit simplement d'une librarie présente dans l'espace d'adressage du noyau (ring 0). Nous investiguons également la possibilité d'utiliser du code présent dans le firmware en développant un driver runtime UEFI qui pourrait apporter des garanties de sécurité supplémentaires de par son isolement de l'OS [3, 9]. Il est nécessaire pour invoquer un tel service d'utiliser une interface fournie par le firmware, celle-ci reste présente en mémoire même après le chargement de l'OS.

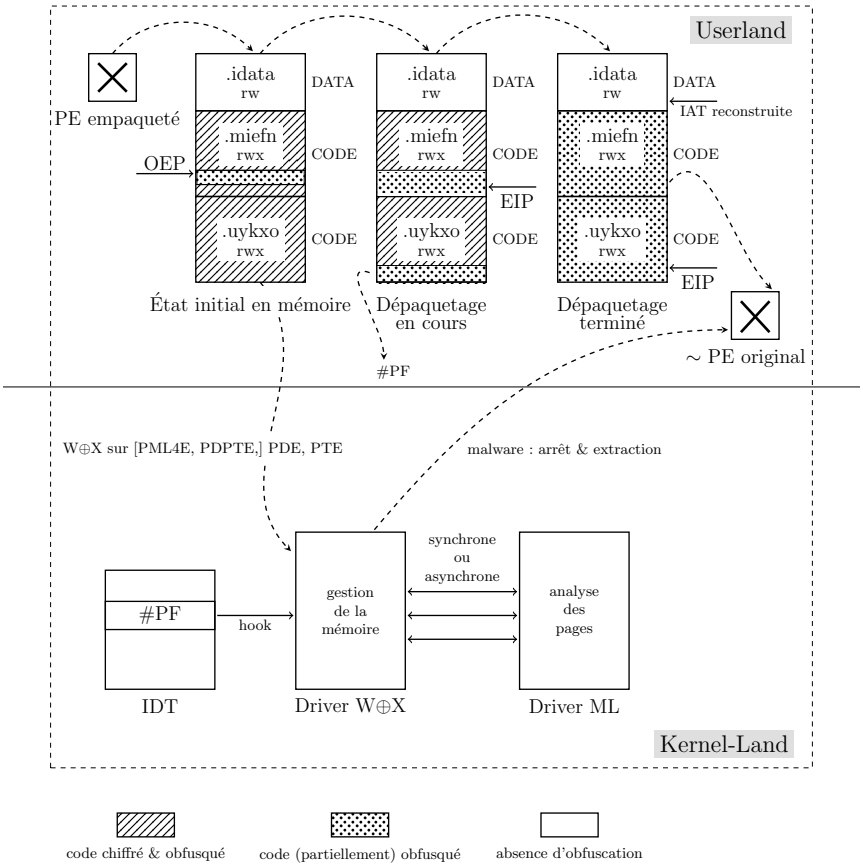

FIGURE 1: Détection d'un binaire malveillant empaqueté

La figure 1 illustre l'intégration de notre framework dans le système, il s'agit de mettre en place la protection W⊕X puis de monitorer les défauts de page ainsi engendrées. La gestion de ces défauts de page est réalisée grâce à la modification du flux d'exécution, on remplace la routine du système. La routine fournie fait appel à notre module de détection puis à la routine originale.

4 Heuristiques

Nous souhaitons focaliser notre algorithme de détection sur les différentes couches de codes dépaquetés et ce de manière incrémentale. L'utilisation de l'apprentissage automatique dans un environnement dynamique va permettre de restreindre les zones mémoires à analyser à la taille d'une page. De plus, il s'agit de zones mémoire effectivement atteinte par le programme et non d'une approximation. La séquence d'instructions ainsi d'observées devra être traitée et formatée afin que nous puissions stocker ces informations pour entrainer l'algorithme.

Il semble que deux types d'algorithmes d'apprentissage automatique peuvent correspondre à nos besoins [15, 16, 21], les algorithmes de regroupement et de classification. L'avantage des algorithmes de regroupement est qu'ils permettent de trouver des similarités (comportemental) entre un logiciel malveillant "inconnu" n'appartenant à aucune classe et les classes existantes. Les algorithmes de classification ne connaissent que des classes prédéfinies auxquelles le logiciel malveillant "inconnu" peut ne pas appartenir. De nombreux tests devront être menés afin de valider : les caractéristiques extraites des couches observées, la taille de la base d'entrainement, l'efficacité des algorithmes et de leurs paramètres.

L'utilisation de l'apprentissage automatique s'accompagne d'opération sur des réels qui peuvent être coûteuses dans l'espace noyau. De plus, à l'intérieur du noyau nous ne disposons pas de librairies mathématiques comme dans l'espace utilisateur. Il est nécessaire d'utiliser le langage assembleur pour certaines opérations (exp, pow, ...). Une solution est d'utiliser une représentation des nombres en virgule fixe si jamais les performances sont trop impactées [1]. Cette technique est employée dans les systèmes embarqués ne disposant pas d'unité à virgule flottante.

5 Expérimentations

La collecte des programmes malveillants est une tâche primordiale, nous utilisons des bases de données [2, 11, 12, 20] mises à jour régulièrement disposant de la quasi-totalité des programmes malveillants présent dans la nature. Une fois les binaires collectés il est nécessaire de les tester pour déterminer s'ils sont encore actifs mais aussi si leur comportement malveillant est susceptible de se déclencher dans notre environnement. Une approche manuelle est choisie pour tester le comportement de ces programmes, principalement pour les observer en

détail. Nous privilégions une approche qualitative plutôt que quantitative pour le moment.

L'environnement dans lequel nous menons les expérimentations dispose d'un accès direct à Internet sans filtrage et les PCs infectés sont équipés d'une adresse IP publique afin d'être joignable même de l'extérieur (C&C, RAT). L'installation d'un système de restauration automatique est réalisée ainsi que la collecte de l'ensemble des flux réseaux via un miroir de port sur un commutateur. Les systèmes infectés ne sont pas virtualisés et sont arrangés de manière similaire à celui d'un utilisateur lambda (musiques, cookies, applications, etc...).

La plupart des programmes malveillants étant empaquetés nous les utiliserons pour tester l'efficacité de notre solution. De plus, ceux-ci sont susceptibles d'utiliser des versions faite-maison. Afin de mieux assimiler les techniques employées par les empaqueteurs (*Themida*[3], *UPX*[4]) nous obfusquerons nos propres binaires. Nous espérons également mettre en évidence les caractéristiques d'un binaire empaqueté malveillant grâce aux échantillons collectés mais aussi en jouant abondamment sur le comportement des binaires empaquetés par nos soins.

6 Conclusion

Notre objectif est de déployer un framework transparent, fiable et efficace. Sa compatibilité est assurée grâce aux spécifications matérielles des processeurs pour accéder à la mémoire. L'utilisation de l'apprentissage automatique promet des résultats intéressants. Les performances du système seront impactées uniquement si des applications abusent la protection W⊕X. Le framework agit comme un véritable bac à sable pour les binaires empaquetés et ce sans notification pour les applications présentes dans l'espace utilisateur pour le moment.

3. www.oreans.com
4. upx.sourceforge.net

Références

1. Anguita, D., Boni, A., Ridella, S. : Svm learning with fixed-point math. In : Neural Networks, 2003. Proceedings of the International Joint Conference on. vol. 3, pp. 2072–2076. IEEE (2003)
2. AVCaesar : Malware analysis engine and repository, http://avcaesar.malware.lu
3. Bailey, D. : Winning the race to bare metal. BH (2008)
4. Dinaburg, A., Royal, P., Sharif, M., Lee, W. : Ether : malware analysis via hardware virtualization extensions. In : Proceedings of the 15th ACM conference on Computer and communications security. pp. 51–62. ACM (2008)
5. Egele, M., Scholte, T., Kirda, E., Kruegel, C. : A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys (CSUR) 44(2), 6 (2012)
6. Gavriluţ, D., Cimpoeşu, M., Anton, D., Ciortuz, L. : Malware detection using machine learning. In : Computer Science and Information Technology, 2009. IMCSIT'09. International Multiconference on. pp. 735–741. IEEE (2009)
7. Hotz, CMU : Qira is a timeless debugger, http://qira.me
8. Kang, M.G., Poosankam, P., Yin, H. : Renovo : A hidden code extractor for packed executables. In : Proceedings of the 2007 ACM workshop on Recurring malcode. pp. 46–53. ACM (2007)
9. Lazarowitz, Matthew : Anti-malware support for firmware (Oct 21 2014), uS Patent 8,869,282
10. Maaser, C., Baier, H. : A concept for monitoring self-transforming code using memory page access control management. In : Security Technology (ICCST), 2011 IEEE International Carnahan Conference on. pp. 1–7. IEEE (2011)
11. Malekal : Malware repository, http://malwaredb.malekal.com
12. Malwr : Malware analysis by cuckoo sandbox, http://malwr.com
13. Martignoni, L., Christodorescu, M., Jha, S. : Omniunpack : Fast, generic, and safe unpacking of malware. In : Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. pp. 431–441. IEEE (2007)
14. Quist, D., Smith, V. : Covert debugging circumventing software armoring techniques. Black hat briefings USA (2007)
15. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P. : Detection of Intrusions and Malware, and Vulnerability Assessment : 5th International Conference, DIMVA 2008, Paris, France, July 10-11, 2008. Proceedings, chap. Learning and Classification of Malware Behavior, pp. 108–125. Springer Berlin Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70542-0
16. Rieck, K., Trinius, P., Willems, C., Holz, T. : Automatic analysis of malware behavior using machine learning. Journal of Computer Security 19(4), 639–668 (2011)
17. Russinovich, M.E., Solomon, D.A., Ionescu, A. : Windows Internals - Parts 1 and 2. Microsoft Press, 6th edn. (2012)
18. Shinagawa, T., Eiraku, H., Tanimoto, K., Omote, K., Hasegawa, S., Horie, T., Hirano, M., Kourai, K., Oyama, Y., Kawai, E., et al. : Bitvisor : a thin hypervisor for enforcing i/o device security. In : Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 121–130. ACM (2009)

19. Thompson, C., Huntley, M., Link, C. : Virtualization detection : New strategies and their effectiveness. http://www.cs.berkeley.edu/∼cthompson/papers/virt-detect.pdf
20. VirusShare : Repository of malware samples, http://virusshare.com
21. Wagener, G., State, R., Dulaunoy, A. : Malware behaviour analysis. Journal in Computer Virology 4(4), 279–287 (2007), http://dx.doi.org/10.1007/s11416-007-0074-9
22. Wong, W., Stamp, M. : Hunting for metamorphic engines. Journal in Computer Virology 2(3), 211–229 (2006)