2012 4th International Conference on Computer Research and Development
IPCSIT vol.39 (2012) © (2012) IACSIT Press, Singapore

Runtime Detection of Software Modification Using RSCA Method

Sathaporn Sa-ngounwong [1] and Pornsiri Muenchaisri [2]

Chulalongkorn University, Bangkok, Thailand
[1] sathaporn.sangounwong@gmail.com
[2] pornsiri.mu@chula.ac.th

Abstract. Runtime Software Modification (RSM) is one of software modification that can be used for both positive and negative purposes. Positive RSM (PRSM) is the legal software modification used for changing dynamic software architecture to improve its functions while they are running on different environments. On the contrary, Negative RSM (NRSM) is illegal software modification used for software piracy or software attack. We propose a method called "Restricted System Calls Analysing" (RSCA) for evaluating the PRSM and NRSM. The RSCA is a method for detecting running software modification by analysing the restricted system calls calling. We categorise the restricted system calls into three groups: other-process modifies system calls, self-modification system calls and encouraged modification system calls. We evaluate the PRSM and NRSM by analysing the relationship among three kinds of restricted system calls. Our RSCA method can efficiently and safely detect the NRSM without fault negative result.

Keywords: Computer security; Runtime detection; Software modification; System calls analysis

1. Introduction

The software modification could be a major problem in the software security field. Software modification can be categorised into two groups: Static Software Modification (SSM) and Runtime Software Modification (RSM).

The SSM focuses on the modifications on file-system rather than memory system. The modification will perform on an executable file and a configuration file. The SSM is normally used by a virus or static file hacker. Examples of the SSM protections include Metamorphic Software Protection [1] and Obfuscation [2] methods. Additionally, the SSM detection is intrusion detection via static analysis [3] method. However, both of the SSM protection and detection cannot work properly in the runtime state. While software is running, it has to convert all protected code to the normal runnable code. Thus, every software symbol can be readable and be directly modified in that state. On the contrary, the RSM focuses on the running software being called by process or task. Since the running software is on memory, the RSM will interact with memory rather than the file-system.

In general, the RSM has a negative meaning because most of software modifications have been used for the software piracy and system attack purpose. But some dynamic software architectures [4][5][6][7] were designed to modify their structure while it is running to adapt themselves to different situations. This type of RSM called Positive RSM (PRSM). We assume the PRSM is a normal behaviour because it is a basic concept that software is able to modify its structure by itself.

We distinguish another type of RSM as Negative RSM (NRSM) which is the software modification that this paper is focusing on. The NRSM is the illegal software modification used for software piracy or system attack purpose. Examples of NRSM include Buffer overflow [8] and Software parasite [9]. The Buffer overflow attacks the running software's stack via vulnerable functions that may be introduced to the software itself or on shared libraries. The examples of buffer overflow protection are StackGuard [10] and SecureBit

[11]. The parasite software modifies the running software via common shared library vulnerability such as the ld command shared library that was mapped to process with the same memory address [12].

There are a large number of runtime software security researches that focus on software protection such as LIDS [13] and rootkits detection [14][15][16] which are the methods for protecting operating system from the unauthorised usage. In addition, kernel memory protection [17] is the method for protecting the memory using the W^X [18] algorithm. The system protection method is a one-way protection. It is designed to enable the system to be stronger but it cannot guarantee that the system cannot be modified. If the modification occurs, how it can be detected.

The efficient and easy method for detecting RSMD is software's integrity value checking [19][20]. The integrity value can check software modification accurately. However, the integrity value can be pinned out by hacker and it is necessary to pay for the software footprint and overhead. The other interesting RSMD method is the Host-based Intrusion Detection System (HIDS) [21][22] which proposed many solutions for protecting, detecting and correcting the running system from misuse applications. There are many methods were proposed in HIDS such as the running software behaviour logging [23][24][25], anomaly software detection [26][27][28] and system calls sequence analysis [29][30] which our proposed method is based on. These kinds of RSMD methods focus on system intrusion detection rather than software modification detection. These methods are efficient but time-consuming to compile the statistics of running software.

We use the Linux OS [31] as a platform for software development because Linux is an open source OS that a user can customise any requirements as needed. Kprobes [35] is a system-call probing tool that was embedded into kernel to assist the developer tracking the behaviour of running process. Since we are not so familiar with the logic/methods of data retrieval and filtering of Kprobes, we choose SystemTap [36] as a tool for implementing our RSCA method.

All RSMD methods that we reviewed have the advantage and disadvantage but none focuses on the PRSM and NRSM. We propose the RSCA method that not only detects the RSM but also can separate the PRSM and NRSM by analysing relationship among "Restricted system calls" callings of all running processes. Our RSCA method is simple and has low effect to operating system performance in which the results is shown in the Experiments and Results section.

2. RSCA Method

2.1. The Restricted System Calls

All restricted system calls that can lead to the runtime software modification. The RSCA method will specifically be focuses on these system calls. We categorise restricted system calls into three groups. First, other-process modifies system calls which we shortly name it to "OM". Second, the self-modification system calls which we call "SM" and the last one is encouraged modification system calls which we call "EM".

The OM system calls are system calls that any process (caller process) used for either controlling or modifying another process (target process). "*ptrace*" system call is used for debugging the running processes. Attacker can use this *ptrace* system call to attack the target process such as the software parasite [9].

The SM system calls are system calls that caller process uses to modify its structures, such as "*mremap*" and "*munmap*". They are used to adjust the process's memory map for process's memory allocating or de-allocating purposes. These SM system calls always lead to PRSM by default.

The EM system calls do not modify the running process's contents directly, but it can increase the chance for modification of running processes. In this paper, we only warn users about the modification chance that may be occurred. We plan to implement the memory contents modification detection in future work.

2.2. RSCA Overview

Fig. 1 shows the overview of the proposed RSCA method that can be separated into four parts. Part A is "Restricted System Calls Monitoring" used for monitoring and filtering the restricted system calls calling from running processes. If a running process calls to some restricted system calls, the RSCA will categorise that call and pass it to the related parts. Part B is OM system calls analysing which will response to the system calls that can modify other processes. Part C is SM system calls analysing. This part not only

responses to the self-modification system calls but also analyses the PRMS and NRSM of caller processes. Part D is EM system calls analysing. This part is used for warning a chance of modification that may be occurred. The detail of each part implementation will be explained in the next subsection.

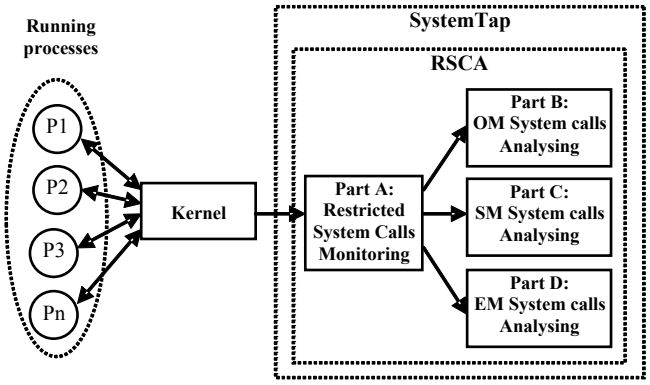

Fig. 1: RSCA approach overview

2.3. RSCA Algorithm

NRSM analysing is a key of the proposed RSCA method. The NRSM will be occurred in case of the OM system calls are called and the target process of the OM system calls is modified. If process's contents are changed without any OM system calls calling, the RSCA will evaluate it as PRSM.

We can separate the algorithm into four parts which is equivalent to RSCA overview in Fig. 1. The major responsibility of Part A is to capture all system calls usage which is information needed for the following parts. Part B is OM system calls filtering and managing. If there is a process calls to OM system calls, the RSCA will keep the target process ID in Target list in which Part C will use for NRSM evaluating.

Part C is SM system calls filtering and analysing part. This part uses information in the Target List and in the process that calls to SM system calls for evaluating the NRSM. If the caller process is already in Target list, this means it is modified by other processes and RSCA will report the NRSM is founded. On contrary, if the caller process is not in the Target list, this means the caller process modifies itself and RSCA will report that the PRSM is founded.

The final part is Part D. This part is EM system calls filtering and analysing. The EM system calls do not directly modify any information of running processes but it increases a chance for modification. Thus, RSCA will issue warning about these system calls calling.

3. Experiments and Results

After the RSCA method is implemented, two activities are subsequently performed. First one is how to test the proposed method and the latter is how the proposed method affects the operating system performance. Thus, in the experiments section, we will present both of the RSCA testing and the effect to operating system performance.

3.1. Software modification simulation

Before we launch the RSCA, we have to specify the method to simulate the RSM first. We choose the “Software parasite” [9] for testing the NRSM detection. The software parasite is the method that other process can modify the other process by using the shared library vulnerability. More information on parasite software usage and the parasite software preparing steps can be found in [9].

3.2. RSCA Test case

We start by testing the NRSM and PRSM detections. We use the parasite software to simulate NRSM. The test steps can be done by launch the RSCA script then run the parasite software to infect the most used software one by one and monitor the reporting result.

3.3. RSCA Test Result

We experiment by infecting software parasite into the most frequently used software such as OpenOffice, Gnome-terminal and Mozilla-firefox etc. We found only the Gnome-terminal that does not calls to any Restricted System Calls while executing. The RSCA can detect all parasite infected software, except the Mozilla-firefox and Google-chrome browsers which are not infected by parasite because they do not use the specific common library of kernel while running.

The overall result indicates that the RSCA approach can distinguishes the NRSM and PRSM well without fault negative (FN) results. We encounter some unexpected fault positive (FP) results in case of the Integrated Development Environment (IDE) Software such as Netbeans [32] and QT Designer [33]. The RSCA evaluates the debugged software as NRSM.

3.4. Performance evaluation

All software security methods have to be paid with some performance of running system. To prove the proposed RSCA performance, we use the "Load testing" [34] method to evaluate the effect of the proposed method to the operating system's performance. We implement the load testing to perform the common function usage such as memory allocation/free, arithmetic calculation and the function call/return. The performance is evaluated by comparing the same test software result on the operating system with RSCA and the system with no RSCA. The result shows that our RSCA method decreases the overall performance of operating system by approximately 7%.

4. Conclusions and Future Work

In short, we can conclude that the proposed RSCA method is a simple method for the runtime software modification detection by using only system calls calling information. The RSCA is easy to use (only run single script), easy to add the other software modification patterns by adding only SystemTap probe. There is no need to modify any part of running software. It is safe and has little effect to system performance.

The proposed RSCA method has limited application. Currently, we can only detect the modification of "Software parasite" and "Self modification". We cannot detect the other special runtime modification method, for instance, a buffer overflow or process's memory contents modification. In our next research, we will simulate other runtime software modifications and implement a new detection method.

5. References

[1] Birrer, B. D., Raines, R. a, Baldwin, R. O., Mullins, B. E., & Bennington, R. W. (2007). Program Fragmentation as a Metamorphic Software Protection. *Third International Symposium on Information Assurance and Security*, 369-374. IEEE. doi:10.1109/IAS.2007.28

[2] Fukushima, K., Kiyomoto, S., & Tanaka, T. (2009). Obfuscation Mechanism in Conjunction with Tamper-Proof Module. *2009 International Conference on Computational Science and Engineering*, 665-670. IEEE. doi:10.1109/CSE.2009.20

[3] Wagner, D., & Dean, R. (n.d.). Intrusion detection via static analysis. *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, 156-168. IEEE Comput. Soc. doi:10.1109/SECPRI.2001.924296

[4] Oreizy, P. (1996). Issues in the Runtime Modification of Software Architectures, 1-8.

[5] Goldman, K. J. (2004). Live Software Development with Dynamic Classes. *Engineering*, 1-19.

[6] Shahabuddin, M., Murray, A., & Carson, V. (2008). Component-Based, Run-Time Flight Software Modification. *2008 IEEE Aerospace Conference*, 1-16. IEEE. doi:10.1109/AERO.2008.4526467

[7] Gabba, T. (2011). MODIFICATION OF SOFTWARE ON THE FLY.

[8] Ward, Craig E. (2005), "C/C++ Buffer Overflows", *Unix Users Association of Southern California,* Orange County, California.

[9] O'Neill R.,Modern Day ELF Runtime infection via GOT poisoning, 2009, http://www.vxheavens.com/

[10] Perry, W., & Crispin, C. (2003). StackGuard: Simple Stack Smash Protection for GCC. *Proceedings of the GCC Developers Summit*.

[11] Piromsopa, K., & Enbody, R. (2006). Secure Bit: Transparent, Hardware Buffer-Overflow Protection. *IEEE Transactions on Dependable and Secure Computing*, *3*(4), 365-376. doi:10.1109/TDSC.2006.56

[12] Bartolich A., (2002), "The ELF Virus Writing HOWTO" ,http://www.ouah.org/virus-writing-HOWTO/index.html

[13] http://www.lids.org

[14] Wampler, D. R. (2007). Methods For Detecting Kernel Rootkits. *UMI*.

[15] Wampler, D., & Graham, J. H. (2008). A normality based method for detecting kernel rootkits. *ACM SIGOPS Operating Systems Review*, *42*(3), 59. doi:10.1145/1368506.1368515

[16] Xianghe, L., Liancheng, Z., & Shuo, L. (2006). Kernel rootkits implement and detection. *Wuhan University Journal of Natural Sciences*, *11*(6), 1473-1476. doi:10.1007/BF02831800

[17] Liakh, S., & Grace, M. (2010). Analyzing and Improving Linux Kernel Memory Protection□: A Model Checking Approach. *ACSAC*, (Section 4), 6-10.

[18] Igor, B. (n.d.). Integration of Security Measures and Techniques in an Operating System. *Security*.

[19] Myles, G. (n.d.). The Use of Software-Based Integrity Checks in Software Tamper Resistance Techniques. *IBM*. IBM Almaden Research Center.

[20] Chen, Venkatesan, Cary, Pang, Sinha, and Jakubowski, Oblivious Hashing: A Stealthy Software Integrity Verification Primitive, Proc. of 5th International Workshop on Information Hiding, 2002.

[21] G. Vigna and C. Kruegel, Host-based Intrusion Detection Systems, The Handbook of Information Security, Volume III John Wiley & Sons December 2005.

[22] Chris PeterSen, An Introduction To Network and Host based Intrusion Detection, LogRhythm, Inc., 2006

[23] Shahzad, F., Bhatti, S., Shahzad, M., & Farooq, M. (n.d.). In-Execution Malware Detection using Task Structures of Linux Processes. *Security*.

[24] Nguyen, N., Reiher, P., & Kuenning, G. H. (2003). Detecting Insider Threats by Monitoring System Call Activity. *IEEE Workshop on information assurance*, (June 2001).

[25] Das, S., Chattopadhayay, A., Kalyani, D. K., & Saha, M. (2009). File-system Intrusion Detection by preserving MAC DTS□: A Loadable Kernel Module based approach for LINUX Kernel. *CSIIRW*, *April*.

[26] Andrew P. Kosoresow, Steven A. Hofmeyr, "Intrusion Detection via System Call Traces," IEEE Software, vol. 14, no. 5, pp. 35-42, Sep./Oct. 1997, doi:10.1109/52.605929

[27] Sekar, R., Bendre, M., Dhurjati, D., & Bollineni, P. (2001). A fast automaton-based method for detecting anomalous program behaviors. *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, *9*(C), 144-155. IEEE Comput. Soc. doi:10.1109/SECPRI.2001.924295

[28] Feng, H. H., Kolesnikov, O. M., Fogla, P., & Lee, W. (n.d.). Anomaly detection using call stack information. *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*, 62-75. IEEE Comput. Soc. doi:10.1109/SECPRI.2003.1199328

[29] Federico Maggi, Stefano Zanero, and Vincenzo Iozzo. 2008. Seeing the invisible: forensic uses of anomaly detection and machine learning. *SIGOPS Oper. Syst. Rev.* 42, 3 (April 2008), 51-58. DOI=10.1145/1368506.1368514

[30] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. 2006. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.* 9, 1 (February 2006), 61-93. DOI=10.1145/1127345.1127348

[31] Bovet, D. P., & Cesati, M. (2000). *Understanding the Linux Kernel*. (First.). O'Reilly.

[32] http://netbeans.org

[33] http://qt.nokia.com

[34] Choosing A Load Testing Strategy. (2005). *Segue Software, Inc*.

[35] http://sourceware.org/systemtap/kprobes/

[36] *SystemTap Language Reference*. (2010). (Version 1., pp. 1-46). Boston, USA.: Red Hat Inc., IBM Corp., Intel Corp. Retrieved from http://www.gnu.org/licenses/fdl.html