Pertanika J. Sci. & Technol. 21 (2): 507 - 520 (2013)

PERTANIKA JOURNALS

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

Rootkit Guard (RG) - An Architecture for Rootkit Resistant File-System Implementation Based on TPM

Teh Jia Yew[1]*, Khairulmizam Samsudin[1], Nur Izura Udzir[2] and Shaiful Jahari Hashim[1]

[1]Department of Computer Systems and Communications Engineering, Faculty of Engineering, Universiti Putra Malaysia, 43400 Serdang, Selangor, Malaysia
[2]Department of Computer Science, Faculty of Computer Science and Information Technology, Universiti Putra Malysia, 43400 Serdang, Selangor, Malaysia

ABSTRACT

Recent rootkit-attack mitigation work neglected to address the integrity of the mitigation tool itself. Both detection and prevention arms of current rootkit-attack mitigation solutions can be given credit for the advancement of multiple methodologies for rootkit defense but if the defense system itself is compromised, how is the defense system to be trusted? Another deficiency not addressed is how platform integrity can be preserved without availability of current RIDS or RIPS solutions, which operate only upon the loading of the kernel i.e. without availability of a *trusted boot* environment. To address these deficiencies, we present our architecture for solving rootkit persistence – *Rootkit Guard* (RG). RG is a marriage between *TrustedGRUB* (providing trusted boot), *IMA* (Integrity Measurement Architecture) (serves as RIDS) and *SELinux* (serves as RIPS). TPM hardware is utilised to provide total integrity of our platform via storage of the *aggregate of the clean snapshot* of our platform OS kernel into TPM hardware registers (i.e. the PCR) – of which no software attacks have been demonstrated to date. RG solves rootkit persistence by leveraging on one vital but simple strategy: the mounting of rootkit defense via prevention of the execution of configuration binaries or build initialisation scripts. We adopted the technique of rootkit persistence prevention via thwarting the initialisation of a rootkit's installation procedure; if the rootkit is successfully installed, proper deployment via thwarting of the rootkit's configuration is prevented. We had subjected the RG to 8 real world Linux 2.6 rootkits and the RG was successful in solving rootkit persistence in all 8 evaluated rootkits. In terms of performance, the RG introduced a maximum of 11% overhead and an average of 4% overhead, hence permitting deployment in production environments.

Article history:
Received: 24 September 2012
Accepted: 14 January 2013

Keywords: Trusted Computing (TC), Trusted Platform Module (TPM), Malware and rootkits, SELinux, Linux Integrity Measurement Architecture (Linux-IMA), TrustedGRUB, rootkit detection and prevention

E-mail addresses:
jyteh@yahoo.com (Teh Jia Yew),
khairulmizam@upm.edu.my (Khairulmizam Samsudin),
izura@upm.edu.my (Nur Izura Udzir),
sjh@upm.edu.my (Shaiful Jahari Hashim)
*Corresponding Author

ISSN: 0128-7680

INTRODUCTION

Recent research into rootkit-attack mitigation focusses upon two major categories: detection and prevention. Recent works from the first category includes Nguyen *et al*. (2007), Doug *et al*. (2007) and Riley *et al*. (2007).

Two collective traits are identified within the RIDS works. The first is that the method employed rests on the fact that detection is based on kernel integrity. Violation of kernel integrity signifies rootkit compromise. The second is that there exists no mechanism for the guarantee of platform integrity from the moment the terminal is booted until the kernel loads. With the RIDS codes or tool deployed and functioning at the kernel level, we can assert that vulnerabilities exist even from the moment the BIOS boot block code loads until the OS kernel becomes available. Overcoming this vulnerability requires a trusted boot process, where integrity can be preserved from the moment the BIOS code loads until the kernel loads.

Realisation of the trusted boot is achievable via the availability of a boot-loader with Trusted Computing Group (TCG) support, which mandates the utilisation of hardware-based anchorage for a stage by stage integrity measurement, starting from the BIOS, boot-loader and finally the OS. In guaranteeing platform integrity, execution of the next stage is only permitted after the preceding stage has been guaranteed of its safety. Highly reliable integrity of RIDS solutions is attainable if existing RIDS solutions are complemented with *TrustedGRUB* (Ulrich Kühn, 2007).

In the detection category, we discovered that no mechanism is available to provide a truly reliable *guard* (the detection codes or tool) i.e. if the integrity of the guard is compromised, how do we trust the guard any further and who can we trust after a compromise occurs? Furthermore, how can we ensure that the guard can never lie about its current state, even in the compromised state?

The answer is of course to have a guard that can never be tampered with. We present the use of an *IMA*-based (Riener Sailer *et al.*, 2004) RIDS (Integrity Measurement Architecture) with TPM hardware anchorage, which provides total reliability of the OS kernel via storage of digitally signed aggregate of the *clean snapshot* of the OS kernel into TPM hardware registers (i.e. PCRs), the theft of which is possible only with physical attacks mounted on the TPM chip. To date no theft or attacks have been found to be viable through software. The TPM PCR was able to anchor clean snapshots of the OS kernel, ensuring the availability of a truly reliable guard in the event of compromise.

Recent works in the second category include *Secvisor* by Arvind Seshadri *et al*. (2007), NICKLE by Riley *et al*. (2007), *HookSafe* by Zhi Wang *et al.* (2009) and *IFEDAC* by Ziqing Mao *et al.* (2011) . *Secvisor* employs enforcement of *Write + Execute* in memory pages of guest OS, barring non-authorized codes from being executed with kernel-level privileges. NICKLE employs *memory shadowing* technique utilising shadow physical memory in VMM (Virtual Machine Monitors) for performing authentication of kernel code in real-time execution, ensuring only trusted codes be permitted for execution , in turn ensuring freedom from rootkit codes. In *Hooksafe*, rootkit defense is mounted via protection of kernel hooks in guest OS of hypervisors from being hijacked via relocation of dedicated page-aligned memory, and hardware-based page-level protection is utilised for access regulation of kernel hooks. In

IFEDAC, the marriage of DAC and MAC is utilised for achieving the best of both worlds for the aim of the development of a malware-resistant DAC-MAC system, which guarantees security via permitting user-defined objects (e.g. files) in the OS to be trusted, and employing MAC policies for ensuring malware such as Trojan Horses fails to remain persistent in the event the malware is successful in its deployment.

Although advanced techniques such as memory shadowing (as per NICKLE) and kernel-hook protection (as per *HookSafe*) were employed, the majority of rootkit-mitigation works neglected one simple but vital factor in rootkit defense i.e. *rootkit defense can be mounted via prevention of the execution of configuration binaries or build initialisation scripts.*

Taking this into consideration, we utilised *SELinux* (Richard Haines, 2010) MAC (Mandatory Access Control) mechanism *where all files (objects) are assumed as a threat unless otherwise specified*. The availability of the MAC configuration mechanism of *SELinux*, with its dynamic programming language like *SELinux* policies, enables the labelling of OS objects and files via *file type enforcement labelling*, which labels files into trusted and non-trusted objects, granting *rwx* (read, write and execute) permissions to objects deemed trustable by the OS administrator. In order to preserve normal OS operations, policies were written for the trusted and permitted execution of binaries only in the */bin* and */sbin* directories. Our experiment with 8 real-world Linux kernel 2.6 rootkits demonstrates that all these 8 rookits require some form of configuration prior to deployment and for proper operation and that *SELinux* is effective and successful in the prevention of the execution of configuration binaries and build initialisation scripts.

A collective trait of *Secvisor*, *NICKLE* and *HookSafe*, all rootkit defenses, was mounted in hypervisors (also known as Virtual Machine Monitors or *VMM*) using guest OS. We wish to point out that recent and the majority of rootkit prevention works are carried out in *hypervised* environments, hence, there is no benchmark to evaluate both the performance and effectiveness of the published rootkit-defense methodologies. IFEDAC, while deployed in real time, neglects to address the integrity of the *guard* itself, i.e. how does IFEDAC ensure that the DAC itself remains trustworthy in the event malware targets the DAC?

In an attempt to complement existing rootkit defenses, we present the *RG* (Rootkit Guard), an architecture of merged *SELinux* MAC (Mandatory Access Control), an *IMA*-based RIDS (Rootkit Intrusion Detection System), with TPM hardware-based anchorage and *SELinux*-based RIPS (Rootkit Intrusion Prevention System), *deployed in real time*. To ensure the integrity of our platform from boot-up until the kernel loads, whereupon SELinux would be available, we leverage such guarantee using *TrustedGRUB*. In short, RG possesses these features: ability to detect presence of rootkit, the ability to never lie about its compromised state and ability to prevent the manifestation of rootkits via the prevention of the execution of configuration binaries and build initialisation scripts. RG further provides an *encrypted loopback partition* for storage of vital data if a compromise is detected. The partition's private key is stored in the TPM hardware register, hence, the impossibility of theft or software-mounted assaults.

Currently, to demonstrate the viability of the RG, we installed 8 real-world rootkits and attempted the prevention of the execution of both: i) the configuration binaries deemed essential for the proper deployment of the tested rootkits and ii) the build initialisations scripts (written in *bash, perl*). We demonstrated the successful thwarting of both via utilisation and enforcement

of *SELinux* MAC policies. In terms of performance, evaluation utilising the *UnixBench 5.1.3* micro-benchmarking tool introduced only a maximum of 11.3 % of system overhead and an average of 3.78 % of system overhead. The reasonable overhead has minimal impact on a running OS and, hence, we believe, practical, real-world deployment is feasible.

As at the time of writing of this paper, RG has been implemented in virtualised environments using the *VirtualBox* VMM (Virtual Machine Monitor). We shall, at a later stage, port our RG implementation to real-time execution .Our work (conducted in real time) provides the ***actual scenario*** of rootkit prevention operating in real time. In terms of deployment in robustness, we dare assert that our work supplements the clearest and most accurate results for the consideration of the adoption of our RG in production environments or in environments where the use of virtualised guest OS is neither possible nor practical.

The rest of this paper is organised as follows: we present next the *design* of our RG, followed by details of implementation (under the section *Materials and methods*), proceeding to the evaluation of our RG in the section *Results and discussion* where we demonstrate the RG's effectiveness in preventing rootkit configuration and performance benchmarking and, finally, end with the section *Conclusion*.

RG DESIGN GOALS AND ASSUMPTIONS

Design goals

The main design goal of RG was to merge TPM hardware-anchored RIDS i.e. IMA with the implementation of MAC-based Linux security i.e. *SELinux* for RIPS purposes with a trusted boot guaranteed by *TrustedGRUB*, the integrity of which is also guaranteed by the TPM hardware. We found that recent rootkit-defense work neglected to consider the importance of ensuring integrity from system boot until the deployment of rootkit defense mechanisms. The novelty of our work is that this is the first of its kind: we integrated *TrustedGRUB*, IMA and *SELinux* into a single entity, yielding a TPM hardware-anchored RIDS i.e. a RIPS rootkit defense mechanism with *trusted boot* feature. To summarise, we developed a rootkit-mitigation methodology or tool that can never lie about its state (feature provided by the TPM hardware), and for the effective implementation of rootkit-attack mitigation, permits only trusted objects to be granted security clearance (feature provided by *SELinux*).

As mentioned in the introduction, recent efforts in overcoming rootkit persistence have yielded numerous solutions to the variety of methods to overcome rootkit persistence. Our work discovered that rootkits would fail to be persistent if its installation is thwarted or if it failed to be properly configured prior to deployment. Hence, another goal of our RG was to attempt the prevention of rootkits before they can even begin installation procedure, and if the rootkit has been successfully installed, to prevent its deployment via thwarting its configuration.

Assumptions

Our RG design and operation are based on these assumptions:

i. Rootkits may infiltrate, but will fail to accomplish proper deployment if their binary fails to execute for initialising the rootkit or for configuration purposes.

ii. A platform may be compromised but it will reveal that it is in compromised state i.e. the platform *never lies* on the integrity of its state.

iii. The RG's RIDS, although employing a commonly deployed method of comparison of healthy with altered hash values in the IMA, operates without the availability of intelligent heuristics algorithms. Hence, the RG serves to alert the OS administrator of a possible compromise and the user can then decide if the alert is one of false positive or vice versa.

iv. Our incorporation of *SELinux* as rootkit prevention mechanism with MAC implementation operates based on the presumption that *all files* are treated as malicious unless otherwise specified in the *SELinux* policies. Hence, *rwx* file permission privileges are granted only to files permitted by deployed *SELinux* policy, configured by the OS administrator.

v. RG serves to complement the multiple, existing rootkit-defense methodologies available, especially *Secvisor*, NICKLE and Hooksfe. No one single tool can claim 100% effectiveness in rootkit-attack mitigation.

MATERIALS AND METHODS

We present the RG architecture (Fig.1) as the proposed solution to the malware threat as mentioned in the introduction. Our proposed solution is effective against solving *binary rootkit persistence* and any rootkit that operates via initialisation or configuration of a binary file, for example, to run the *SuckIT* rootkit, the rootkit will have to be configured and executed using *./sk c* command on the victim's machine (Phrack Magazine, 2012). *Lrk5* i.e. (Linux Rootkit Documentation, *Lrk5*, 2012) and *Adore-ng* i.e. (Linux Rootkit Documentation, Adore , 2012) are other examples of rootkits with deployment methods similar to *SuckIT*.

Our RG comprises two major arms: *Rootkit IDS* (detection and alert) and *Rootkit IPS* (prevention). Effective implementation mandates our RG be compiled into the Linux kernel, hence, part of the Linux file-system. Availability of a TPM guarantees the integrity of a clean kernel state, as such state is utilised by RG IDS and serves as the essence of the RG's reliability. The *Rootkit IDS* comprises:

i. IMA databases (*clean and runtime*) – protects kernel files and modules

ii. SELinux Policy database – protects user-defined critical files

iii. IMA database SHA1Comparison Engine

iv. IMA database Compromise Alert Mechanism

v. SELinux Security Policy Violation Alert Mechanism

while the *Rootkit IPS* encompasses the:

i. Encrypted Loopback Partition

ii. Deny File Access Features - to the infected file (via removal of *rwx* privileges)

SHA1 collision attacks occurring since 2004 (Bruce Schneier, 2005), Michael Szydlo, 2005), Xiaoyun Wang *et al*., 2005) can be employed for the aim of defeating the SHA1 digital finger-printing. In the event such attacks do occur, contamination only affects the

runtime database of hashes. The clean database remains reliable as the *aggregate hash* of the clean database is *extended* (i.e. using *TPM_Extend*) into the TPM hardware PCR (Platform Configuration Register) no. 10 as per the technique adopted in Integrity Measurement Architecture or *IMA* i.e. as per IMA Wikipage Main (2012). Should alterations occur to the aggregates, the occurrence would signify compromised integrity of the clean database.

Operational-wise, two RG vital components are developed:

1. RG Module (Kernel Space)

Both the Rootkit IDS and IPS features are integrated into a kernel module called the *RG Kernel Module*, which loads simultaneously with kernel loading, i.e. initialising from *init level 1*. Such ensures detection of integrity violation at the earliest possible stage. We further include an option for the module to be built into the kernel.

Rootkit detection and prevention are simultaneously executed by the RG Module. It is, hence, imperative that the module interfaces to both the *IMA* and the *SELinux Security Policy Violation Alert Mechanism*. Fig.2 illustrates the arms and functions of the RG Module. A rootkit compromise alert is triggered via discrepancies in the IMA database and violations in SELinux Security Policies. The RG Module is protected from rootkits targetting it via the sealing of the SHA1 of the clean RG module into the TPM PCR.

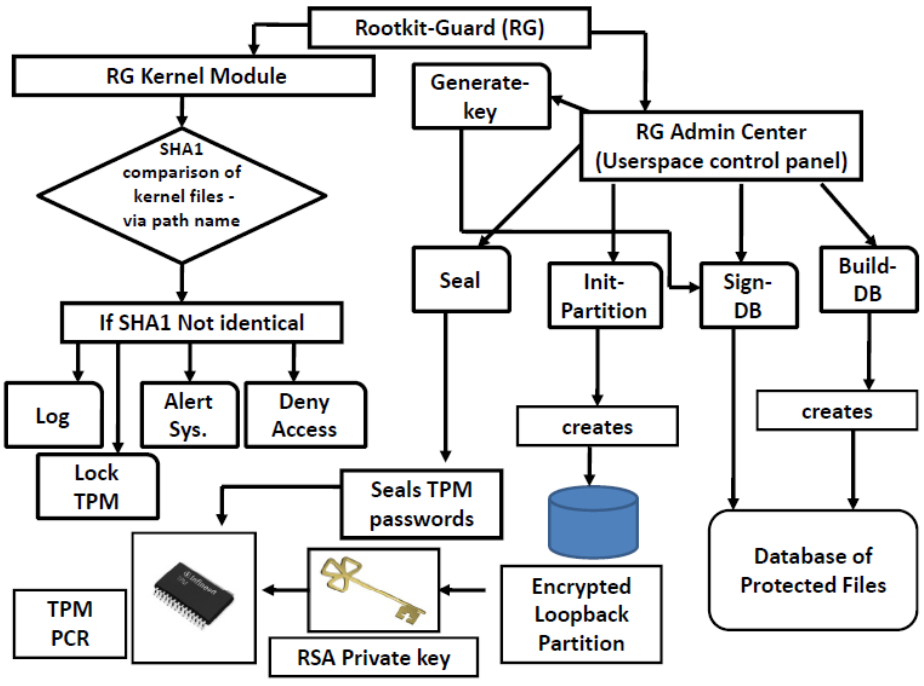

Fig.1: RG architecture

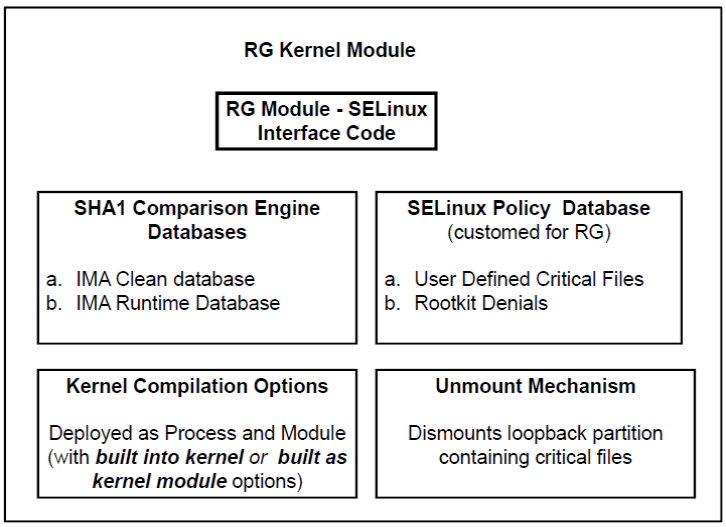

Fig.2: RG Kernel module components

2. RG Admin Center(User Space)

The RG is equipped with a control panel i.e. the *RG Admin Center*, a user-space application permitting users to configure RG, at pre- and post-deployment stages; this is detailed in the next section. The list of RG Admin Center features is given here:

i. Creation of a database of kernel files and modules *(Build -DB)*

ii. Digitally signing the database in *i)* above using a private-key created by GPG *(Sign-DB)*

iii. Creation of an Encrypted Loopback Partition for storage of user-defined critical files (e.g. missile launch codes) *(Init-Partition)*

iv. Seal the key in *ii)* above to the TPM PCR *(Seal)*

TPM-protected Encrypted Loopback Partition

The RG incorporates an encrypted loopback partition (i.e. part of Rootkit IPS) for storage of user-defined critical files e.g. nuclear warhead launch codes. Protection of this partition is provided by 1024-bit RSA Encryption, whose private key is stored in the TPM; hence, the impossibility of theft.

IMPLEMENATION & TEST –BED

Implementation is conducted in two stages. See Fig.3:

a. Pre-Deployment Configurations: Essential set-ups and configurations necessary for the proper and effective operation of the RG

b. Actual, Real-Time Deployment of the RG: The RG in action, implemented in a production environment

Unlike most malware research work which performs implementation in VM environments and deployment in guest OSes (Ryan Riley *et. al.*, 2008; Arvind Seshadri *et. al*. ,2007), our RG was deployed in an actual platform with no hypervisors utilised. RG deployment was ported and executed on a Dell Latitute E5400 laptop, with the following specifications: 3 *GHz* Core 2 Duo CPU, 4GB DDR-2 RAM, running Fedora Core Linux 16, with kernel 3.1.7. Currently, our implementation and test-bed is conducted on the same laptop as above, albeit in the *VirtualBox* Hypervisor (an open source hypervisor). We allocated 1.3 GB of RAM to the same Linux OS (installed as Guest OS in VirtualBox) running identical kernel. Each stage is detailed here:

a. Pre-Deployment Configurations

The pivotal part of the RG is the availability of a clean database of the IMA-measured SHA1 list of kernel files and modules from a freshly configured platform .We term this the *Clean IMA Database*. RG Admin's Build-DB feature is utilised for this purpose. Upon deployment, the clean database is compared to a *runtime database* of similar SHA1 list for rootkit detection.

Next, SELinux is utilised to establish security context for user-defined critical files via the writing of a dynamic programming language like *SELinux policies* (SELinux rules) . A *critical files and objects domain* is created and these critical files are labelled by SELinux with file type enforcement: *rg_secured_t*. Only files and also objects labelled with *rg_secured_t* are granted execution privileges. A database of clean SELinux policies for the critical files-objects is established. TrustedGRUB is configured to preserve the integrity of boot essential files (esp. *initrd* and *vmlinuz*). Finally, the creation of an Encrypted Loopback Partition for storage of critical files accomplishes this stage.

b. Actual, Real Time Deployment of the RG

This section considers the rootkit infiltration events after a *SuckIT*-type *rootkit* was successfully planted on the victim's machine (*V*). Configuration of the *SuckIT* binary is essential prior to the execution of a backdoor to permit access to *V* machine by a remote attacker (*A*) machine.

Upon rootkit binary execution (i.e. *./sk c*), two mechanisms in the Rootkit IDS kick in to alert the user: the IMA database's *SHA1 Comparison Engine* (see Fig.2) via SHA1 anomalies in the clean and runtime databases, and the *SELinux Policy Violation Alert Mechanism* issues an alert on the user's Desktop due to policy violation for two possible actions: either i) the rootkit attempts to access files in the *critical files domain* with *rg_secured_t* file type enforcement or ii) files not labeled *rg_secured_t* file-type enforcement attempt execution.

The Rootkit IPS steps in and unmounts the Encrypted Loopback Partition, preventing possible data theft (e.g. missile launch codes) by the rootkit. Instinctively, the detected rootkit is denied *rwx* privileges by SELinux as a decontamination measure. Fig.3 details real-time deployment in a procedural illustrative view.

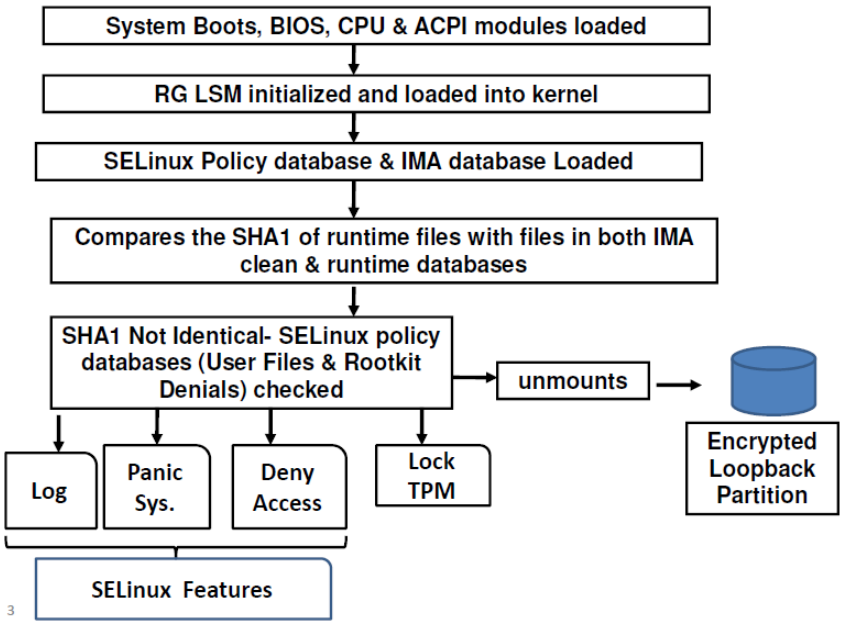

Fig.3: Deployment of the RG – procedural flow illustration

RESULTS AND DISCUSSION

Effectiveness

We subjected our RG to 8 real-world Linux 2.6 rootkits to gauge its effectiveness and attempted the prevention of both the execution of the configuration binaries and build scripts, both actions deemed essential for the proper deployment and installation of the tested rootkits. We demonstrated the successful thwarting of a rootkit binaries configuration via utilisation and enforcement of *SELinux* MAC policies enforcing *rg_secured_t* file type enforcement in our platform. The results are summarised in *Table 1*. The prevention of the execution of the configuration binary of one of the rootkits experimented on i.e. *SUCKIT* is shown in *Figure 4*.

TABLE 1
RG Effectiveness in the Prevention of Real World Linux 2.6 Rootkits

Rootkit	Pre-deployment means	Prevention successful?
Kbeast-v1 Medusa– 0.7.1 Sebek-3.2 Lrk5 Mood-nt	Installation via build script execution	Yes
SuckIT Superkit Adore-ng	Execution of configuration binary	

Performance

The *Rootkit IDS*: IMA integrity assessment via SHA1 Comparison Engine of all kernel files consumes 70s was measured using *Bootchart* (Bootchart, 2012). For the *Rootkit IPS*: we utilised UnixBench 5.1.3 for micro-benchmark evaluation of our RG. This tool is capable of providing a fine-grain performance impact of RG. The type and nature of tests performed are shown in *Figure 5*. Our results listed two highest overheads as the price of running our RG: 11% - from Filecopy 4KB (buffer size 8000 maxblocks) test and 9% from the Dhrystone test. The average overall performance overhead for our RG wass 3.78 %.

The performance penalty is due to the operation of the RG kernel module executing security checks on the kernel files in the Linux file-system. The SHA1 Comparison Engine executes via comparing hashes of *runtime* kernel files with hashes in the *clean* database. This requires the use of string handling and comparison functions (which explains the Dhrystone test overhead) and procurement of hashes of *all* kernel files (which explains the Filecopy test overhead). Near negligible overheads are possible due to the power of contemporary user-computing platform hardware i.e. 4GB RAM and *7,200 rpm* HDD plus 3GHz Core 2 Duo CPU.

In terms of overhead, the performance of our RG was relatively on par with HookSafe, which reports a maximum overhead of 7% and an average overhead of 4%. Next, compared with another rootkit-attack mitigation model which is similar to our RG but without hardware-based anchorage for guaranteeing platform integrity i.e. the IFEDAC (Ziqing Mao *et al.,* 2011), our RG showed better performance. The IFEDAC introduced an overhead of a maximum of 19% and an average of 5.4%. NICKLE rakes up a maximum of 13% of overhead and an average of 5.45%. All reported performance results were obtained using UnixBench.

CONCLUSION

This paper presents an architectural model i.e. Rootkit Guard (RG) for solving rootkit persistence. Our RG utilises the current trend in security solutions in the computing industry today, the TPM, in providing a complementing security solution with TPM-platform integrity guaranteed by the TPM hardware. RG incorporates Tripwire-like features and blends both *IMA* as the RIDS and *SELinux* security features as the RIPS in providing one weapon in the armory of tools/solutions against rootkit persistence. Our RG's inclusion in the kernel ensures that RG is a limb of the platform; hence, RG is part of the platform's 'biological' immune-defense system against rootkits. Evaluation of the RG, both in terms of effectiveness i.e. prevention of deployment of 8 real-world Linux 2.6 rootkits and performance i.e. with average overall performance overhead of only 3.78% underscores the fact that RG is viable for deployment in real-time due to its near-negligible consumption of system resources. The RG complements other existing rootkit-attack mitigation solutions in rootkit defense for OSs.

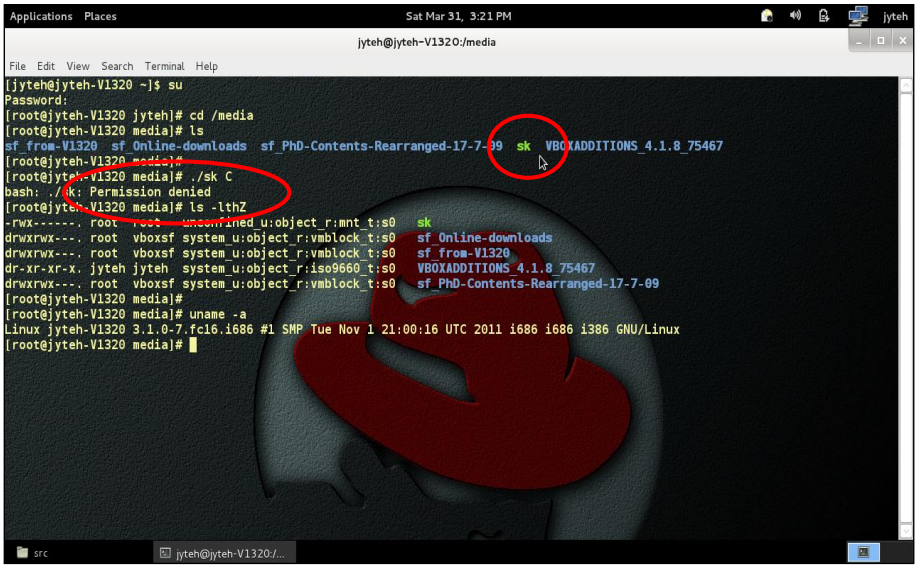

Fig.4: Prevention of the SuckIT Configuration Binary, ./sk, from Execution

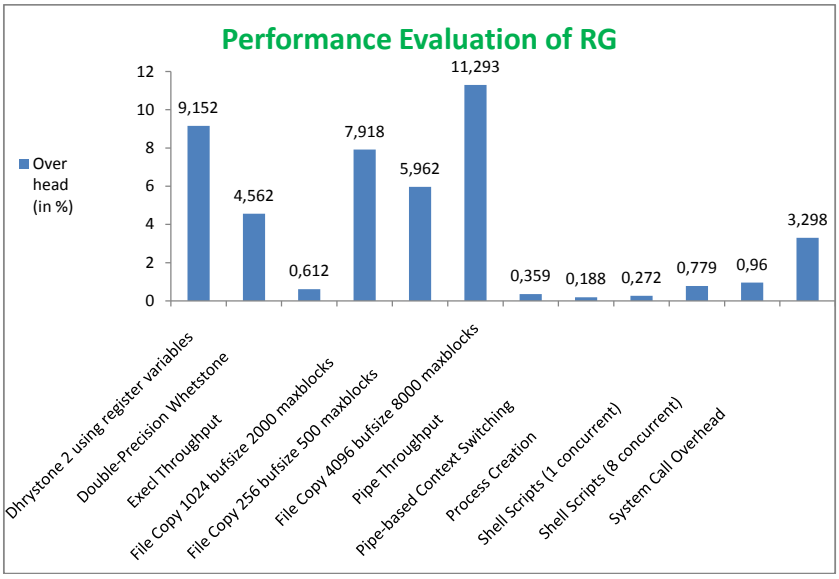

Fig.5: Performance Evaluation Results of RG Using UnixBench 5.1.3

ACKNOWLEDGEMENT

We are indebted to the Research Management Centre (RMC), UPM, for the financial support on our work via Grant No. : UPM/700-2/1/RUGS/05-01-12-1638RU. The authors further wish to thank the Malaysian Ministry of Higher Education (MOHE) for the award of the *myBrain – myPhD* Scholarship to the corresponding author in support of this work.

REFERENCES

Arvind, S., Mark, L., Ning, Q., & Adrian, P. (2007). *SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes, SOSP'07*. October 14–17, Stevenson, Washington, USA. *ACM*.

Bickford, J., O'Hare, R., Baliga, A., Ganapathy, & V. Iftode, L. (2010). *Rootkits on smartphones: attacks, implications and opportunities*. In the Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications. *ACM*. New York. pp. 49–54.

Bootchart. (2012). *Open Source Boot-time Measurement Tool*. Retrieved on March 15, 2012 from http://www.bootchart.org.

Bruce, S., (2005). *Schneier on Security- The blog covering security and security technology*. Retrieved from http://www.schneier.com/blog/archives/2005/02/sha1_broken.html.

Doug, W., & James, H. G. (2007). A Normality Based Method for Detecting Kernel Rootkits. *ACM*.

IMA Wikipage Main. (2012). *Integrity Measurement Architecture (IMA)*. Retrieved on March 15, 2012 from http://domino.research.ibm.com/comm/research_people.nsf/pages/sailer.index.html.

Jonathan, M., McCune, B., Parnoy, A., Perrigy, M., Reiteryz, K., & Hiroshi, I. (2008), Flicker: An Execution Infrastructure for TCB Minimization. *EuroSys'08,* April 1.4, 2008, ACM.

Kevin, R. B. B., Stephen, M., & Patrick, D. M. (2008). Rootkit-Resistant Disks. *ACM CCS Journals - Conference on Computer and Communications Security in the Proceedings of the 15th ACM Conference on Computer and Communications Security*, October 27–31.

Linux Rootkit Documentation, *Lrk5* (2012). *Linux Rootkit 5 Technical Documentation*. Retrieved on March 22, 2012 from http://www.phrack.org/issues.html?issue=63&id=18.

Linux Rootkit Documentation, Adore (2012). *Linux Rootkit Technical Documentation*. Retrieved ofrom http://www.phrack.org/issues.html?issue=61&id=10.

McAfee Inc. (2006). *Rootkits, part 1 of 3*, *The growing threat*, White Paper. Retrieved on August 15, 2012 from http://download.nai.com/Products/mcafee-avert/whitepapers/akapoor_rootkits1.pdf.

Michael, S.(2005). SHA1 Collisions can be Found in 2^63 Operations, RSA Labs. Retrieved on December 15, 2012 from http://www.rsa.com/rsalabs/node.asp?id=2927.

Nguyen, A. Q., & Yoshiyasu, T. (2007). Towards a Tamper Resistant Kernel Rootkit Detector. *ACM.*

Phrack Magazine. (2012). *SuckIT* Rootkit Technical Documentation, Volume 0x0b. Issue 0x3a. Phile #0x07 of 0x0e. Linux on-the-fly kernel patching without LKM. Retrieved on March 11, 2012 from http://www.phrack.org/issues.html?id=7&issue=58.

Reiner, S., Zhang, X., Trent, J., & Leendert, v. D. (2004), *Design and Implementation of a TCG based Integrity Measurement Architecture.* In the Proceedings of the 13th USENIX symposium, CA , USA, USENIX Association.

Riley, R., Jiang, X., & Xu, D. (2008). Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In Lippmann, R., Kirda, E., & Trachtenberg, A. (Eds.) *RAID 2008. LNCS*, vol. 5230, pp. 1–20. Springer, Heidelberg.

SANS Institute. (2012). *Security Predict.* Retrieved on March 15, 2012 from http://www.sans.edu/research/security-laboratory/article/security-predict2011.

Richard, H. (2010). *The SELinux Notebook, Volume 1 – The Foundations*. GNU Free Documentation, pp. 14 – 30.

Tripwire Homepage. (2010). *Tripwire*. Retrieved on April 5, 2010 from http://sourceforge.net/projects/tripwire/.

Ulrich, K., Marcel, S., & Christian, S. (2007). *Realizing Property-Based Attestation and Sealing with Commonly Available Hard- and Software.* STC'07, November 2, 2007, ACM, pp. 50 – 57.

Xiaoyun, W., Yiqun, Y., Hongbo, Y. (2005). *Finding Collisions in the Full SHA-1.* Advances in Cryptology-Crypto 05, LNCS Springer, 3621, pp. 17-36.

Zhi, W., XuXian, J., Weidong, C., Peng, N. (2009). *Countering Kernel Rootkits with Lightweight Hook Protection*, CCS'09, November 9–13, ACM.

Ziqing, M., NingHui, L., Hong, C., & XuXian, J. (2011). Combining Discretionary Policy with Mandatory Information Flow in Operating Systems. *ACM Transactions on Information and System Security*. Vol. 14. No. 3. Article 24.