2017 2nd International Conference on Computer Science and Technology (CST 2017)
ISBN: 978-1-60595-461-5

Thin Hypervisor-Based User Authentication Mechanism for Linux Security Modules

Bin YAN[1,2], Pei ZHAO[1], Heng-tai MA[1,a*] and Jian ZHAI[1]

[1]Institute of Software Chinese Academy of Sciences, Beijing, China

[2]University of Chinese Academy of Sciences, Beijing, China

[a]hengtai@iscas.ac.cn

*Corresponding author

Keywords: User authentication, RTA, Thin hypervisor, Bit visor, LSM.

Abstract. LSM (Linux Security Modules) has been developed as a lightweight, general purpose, access control framework for the mainstream Linux kernel, many tools employ LSM to implement mandatory access control of processes. However, when administrators intend to employ LSM to control a user's behavior instead of just a process's, things become more complicated. Since a user's behavior is reflected by a variety of processes, the control of a user turns into the control of processes associated with the user, which needs the ability to match up a process's identity to a particular user. Unfortunately, without a strong user authentication mechanism, malicious users can easily bypass the behavior control framework by juggling the identity of a process. In this paper, a practical, efficient, secure mechanism, namely RTA (Real-Time Authentication) is proposed to add real-time user authentication support for traditional LSM. The proposed mechanism employs the ID management framework in a thin hypervisor, BitVisor. At last, a new security module called EWL (Executable White List) is designed and implemented based on RTA and LSM, the experimental results show that EWL ensures security and has small system overhead.

Introduction

LSM [1] framework provides lots of security hooks all over Linux kernel, which enable the security modules to do access control in an elegant way with very fine granularity. After the birth of LSM, many well-designed security modules have been developed based on it. There are four standard Linux Security Modules: Security Enhanced Linux (SELinux) [2], AppArmor [3], SMACK and TOMOYO, all of them focus on controlling processes' behavior to achieve better security. In contrast, we decided to provide administrators a tool to control users' behavior for the following reasons:

(1) Users with different privileges run an application in different ways. It's not wise to apply the same restriction of a process to all users. For example, a developer may require higher Internet bandwidth and more memory than an ordinary user when using the same web browser.

(2) A particular user may not be allowed to run some particular applications under special circumstances. For instance, children are not allowed to play computer games when they are supposed to study online.

In a word, it is insufficient to control processes' behavior without attaching users' identity to them. Current tools like AppArmor and SELinux are not able to ensure an ideal run time environment when administrators want to treat each user individually.

To implement such a tool, we need to attach a user to a particular process, that is, we have to authenticate the user and get the identity of the process.

Linux authentication was traditionally dependent on the *passwd* and *shadow* files. Applications like *login* and *su* run with root privilege in order to access authentication information, and set or alter the uids of a process. There are currently three uids in a process's *task_struct*: (1) the real user ID (taken from the entry in the *passwd* file when users log in), (2) the effective user ID (determines users' file access permissions), (3) the saved set-user-ID (used for restoring privileges). When a user enters his name and password after system boots up, the *login* application creates a shell process for the user and sets all three uids to the uid it gets from *passwd* file.

Applications like *login* and *su* are called setuid-to-root binaries, which indicate such applications belong to the root user and their setuid bit is set. Once such applications are executed by ordinary users, the processes' effective user ID will be set to root so that processes can run with root privilege. Malicious users can turn themselves into super user as long as they exploit any kind of security flaw in setuid-to-root binaries. Trusted, setuid-to-root binaries have been a substantial, long-lived source of privilege escalation vulnerabilities on UNIX systems. Chen et al. [4] show thatmany privilege escalation attacks go through setuid-to-root binaries, even on SELinux or AppArmor. Once a malicious user obtains the root privilege, he can easily change the identity of a process through setuid functions. As a result, the real user ID we get from task_struct can never be trusted and the policy rules we made in Linux security modules will be invalid.

In this paper, a novel mechanism to add user authentication for Linux security modules is introduced, supported by the ID management framework of a thin hypervisor, BitVisor [5]. Also, the implementation of a new security module called EWL is briefly presented. In this new module, administrators make policy rules to restrict users to execute programs. By employing the user information provided by BitVisor, EWL can ensure better enforcement of the security policy with just little impact to Linux kernel.

The remainder of this paper is organized as follows. In the 2nd section, the existing solutions to the problem of authentication mentioned earlier are discussed. Then in the 3rd section, the proposed authentication mechanism namely RTA is introduced. Section 4 shows the implementation of RTA and the prototype of EWL. Section 5presents the experimental results and security analyses of our designed architecture. Some future works is mentioned in section 6. Section 7 presents the conclusions.

Related Works

Privilege escalation problem associated with setuid-to-root binaries is also called setuid problem. Actually, with proper usage and careful programming, the risk of setuid-to-root binaries can be contained. For example, programmers can use uid-setting system calls (system calls that modify user IDs) to drop and restore privilege to avoid privilege escalation. However, uid-setting system calls such as *setuid* function are poorly designed, insufficiently documented, and widely misunderstood and misused, which has caused many security vulnerabilities in application programs.

There have been some solutions to the setuid problem, one of which is Linux capability system [6]. Linux capability system was introduced to subdivide the actions typically associated with the super user to limit the security risk of applications which required a particular privileged operation. However, this approach does not reduce the

risk for applications such as *login* and *su* which require the capability to change the user identity of a process. The existing capability system does not govern which user a process may change its identity to. In other words, the same vulnerability described earlier exists for all applications involved with authentication such as *login* and *su*. By the way, the same problem exists in project Protego [7], which spares no effort to deprivilege privileged codes.

Hao Chen, David Wagneret al. [8] provided some advices for programmers to avoid the setuid problem when they are using uid-setting system calls. They also came up with an improved API for privilege management. The API was implemented as wrapper functions to the uid-setting system calls and offered the ability to perform each of privilege management tasks directly and easily, programs with such API should have a secure usage of uid-setting system calls. But to use this implementation, an application must meet some requirements, one of which is that the process does not make any uid-setting system calls that change any of the three user IDs. This limitation makes the API incompatible with applications like *login* and *su* that need the ability to change user IDs of a process, which means the API cannot solve the setuid problem mentioned earlier.

Plan 9 [9] is a distributed operating system designed at Bell Labs to be a next generation improvement over UNIX. It provides the ability to pass a setuid capability - a token which may be used by a task owned by one userid to switch to a particular new userid only once - through the /*dev*/*caphash* and /*dev*/*capuse* files. Thus, applications like *login* and *su* need not run as the super user. Also, capabilities to change identity are restricted to specific authenticated identities by the authentication system. However, the complexity is high because there are various components need to be implemented before using the authentication mechanism of Plan 9. Also, the security of the authentication server is not guaranteed.

Proposed Authentication Mechanism

We propose a strong user authentication mechanism for Linux security modules, so that the user identity problem mentioned earlier will not cause danger to the enforcing security. This mechanism depends on the ID management framework of BitVisor. The EWL module we developed employs this mechanism to achieve a better security of operating system.

BitVisor is designed to enhance the security of computing systems by providing some transparent security functions. As a Virtual Machine Monitor (VMM), the lightweight BitVisor can provide protection from unintentional circulation of information with minimal overhead. Besides the secure data services, BitVisor also has a complete ID management framework [10,11,12,13], providing strong user authentication between a VMM and its users by employing smart cards. The framework mainly employs X.509 public key certificates (PKC) to manage an employee’s identity of an organization.

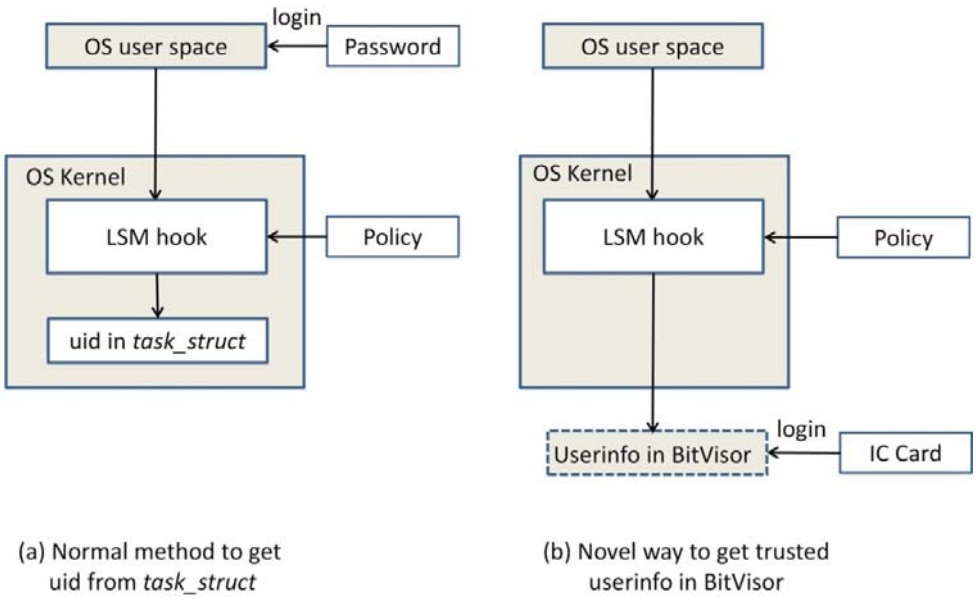

Figure 1. Comparison between two ways of getting user info for LSM hooks.

Fig. 1 illustrates the comparison between two ways of getting user info for LSM hooks. There are high risks of getting uid from *task_struct* for LSM hooks because of the setuid problem we mentioned earlier. Instead of logging in directly into guest OS user space, we let users login into BitVisor firstly and store their user info in VMM address space. After a second boot up, users log in the guest OS with identities recorded in *passwd* file. The EWL we implemented will always take the user as the user logging in BitVisor regardless of the changing of identities in OS user space. This means in RTA authentication mechanism, LSM hooks can always get trusted user info every time in need of the identity of current process.

We call the proposed authentication mechanism Real-Time Authentication, because we don't cache the user identity info in any place of guest OS address space, which means there are no way to juggle the user identity after a user logged in. Since EWL can always get the true identity of the invoker of target process, the proper execution of our policy can be ensured.

Prototype Implementation

A prototype of EWL based on proposed RTA is implemented. This prototype has the ability to restrict the execution of programs invoking by some particular users, according to policy rules made by system administrators. The experiment was taken on Ubuntu 14.04 with the kernel version of 4.3.3, the latest stable version BitVisor 1.4 is chosen for user authentication. The result shows that EWL in the guest OS works fine with the user info from BitVisor.

Implementation of RTA

Fig. 2 shows the procedure of user authentication in BitVisor and the usage of RTA by guest OS. The process of user authentication is a simple challenge and response authentication procedure. A user inputs a PIN number, and an IC card authenticates the user. An authentication handshake is executed between the IC card and the ID management system of BitVisor. BitVisor sends a challenge as R, and the IC card

then returns the user's Public Key Certificate (PKC_{USER}) and a signature of $R((R)_{USER})$ generated by the user's private key($PrivateKey_{USER}$). The ID management system validates the certificate chain using a trust anchor certificate for BitVisor ($PKCT_{TRUST\ ANCHOR\ (VMM)}$). In addition, the ID management system checks certificate revocation status by a Certificate Revocation List (CRL). Once authentication is successful, the user info is stored in BitVisor address space and waits for the call from guest OS.

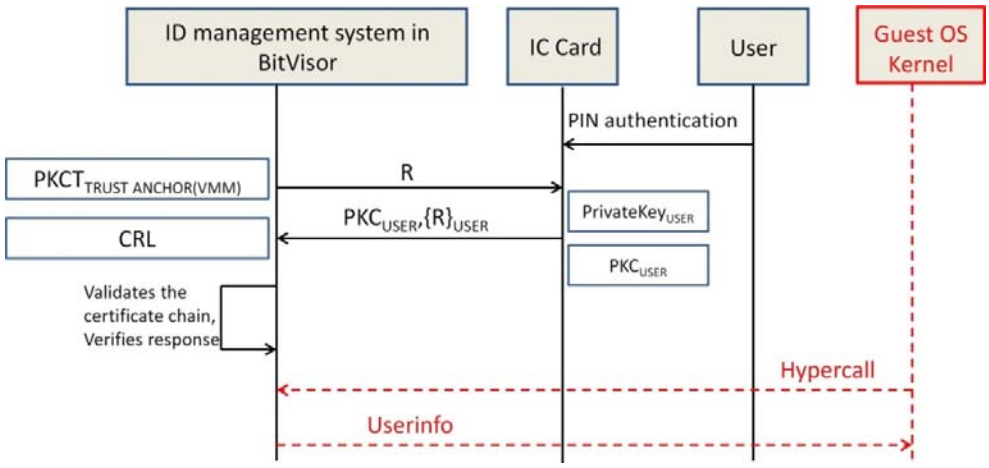

Figure 2. Implementation of RTA.

Intel VT-X technology is employed to make BitVisor and guest OS talk to each other. More concretely, we use *vmmcall* instruction from VMX instruction set to implement a hypercall. A hypercall to a VMM is like a syscall to Linux kernel. In this way, guest OS can call a hypercall to get trusted user info stored in BitVisor.

Architecture of EWL

Fig. 3 shows the architecture of EWL. There are three main parts of EWL: (1) an ID management framework in BitVisor, (2) the behavior control system in guest OS, (3) the external service modules. The ID management framework in BitVisor provides trusted user info for EWL. The behavior control system includes a EWL application (for administrators to make policy rules), a start/stop module (starts the enforced security or shut it down), and a Linux security module (intercepts and verifies the behavior according to policy rules). The external service modules include a CA (signs ELF files), a Time Server (provides trusted time), and a User Manage Server (manages users).

In EWL project, all users' names registered in User Manage Server are collected for administrators of guest operating system to make policy rules of enforced security. When a user executes a program, the LSM hook in *fork* function would catch the action. At this time, the current user's information stored in BitVisor should be called and verification according to the policy rules would be executed. In this way, administrators can control the behavior of an ordinary user.

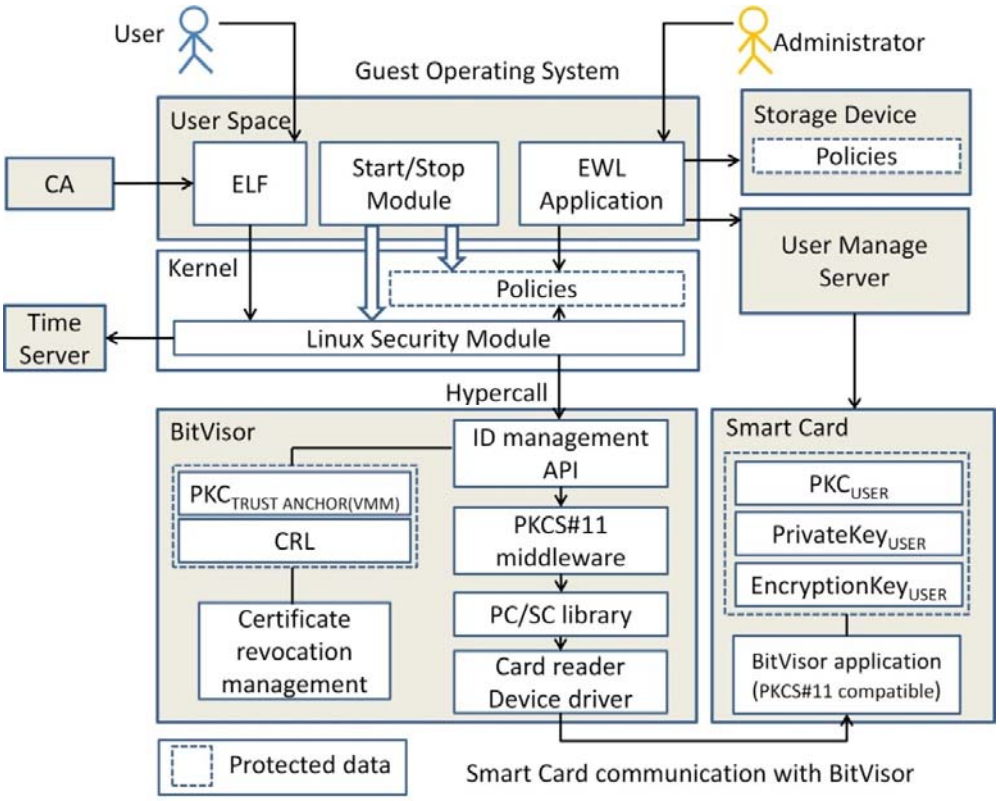

Figure 3. Architecture of EWL.

Experiment and Analysis

Some experiments are done after the implementation of EWL, the results show that the RTA mechanism can ensure proper execution of policy rules made in EWL with just small system overhead. Also, the security of RTA is analyzed from different perspectives.

Experimental Results

There are some typical policy rules made to verify the function of RTA and EWL. Also, some common applications are executed to test the system overhead when running over BitVisor. Table 1 shows details of the policy rules.

Table 1. Policy rules made to verify the function of RTA and EWL.

Rule ID	Target User	Target Program	Action
1	Susan	Gedit	allow to execute
2	Susan	Firefox	forbid to execute
3	Tim	Gedit	forbid to execute
4	Tim	Firefox	allow to execute

In Table 1, each row contains the information of a single rule. The rule ID is the number of a rule, target user represents the user logged in BitVisor, target program is what a user tries to execute on the operating system, and the action indicates the intention of a rule. For example, the rule represented in the first row allows Susan to

execute program Gedit, but the rule in the second row forbids Susan to execute Firefox.

Table 2. The execution results of policy rules

Program	BitVisor User	OS User	Execution Status
Gedit	Susan	Lily	success
Gedit	Susan	root	success
Gedit	Tim	Lily	fail
Gedit	Tim	root	fail
Firefox	Susan	Lily	fail
Firefox	Susan	root	fail
Firefox	Tim	Lily	success
Firefox	Tim	root	success

The execution results of the above policy rules are listed in Table 2. For user Susan, after she logs in BitVisor, she can always execute Gedit, this is because the policy rule 1 allows her to. But Susan can never execute Firefox according to policy rule 2, even if she logs in OS as root user. Because of employing user info in BitVisor, the policy rule won't be invalid due to the identity tampering in OS, which means the setuid problem mentioned earlier cannot affect the execution results of policy rules. As for user Tim, he can always execute Firefox but can never execute Gedit according to policy rules.

Table 3. Average system overhead when running over BitVisor.

Program	Without BitVisor[μs]	With BitVisor[μs]	Overhead[μs]
Gedit	362.7	3549.3	3186.6
Firefox	229.7	2818.7	2589
VLC	288.2	2704.8	2416.6
gzip	319.7	3089.9	2770.2
Avg.	-	-	2740.6

A program starts up by *fork* function to get a process from its parent process and employs *do_execve* function to execute the program. To control the start of an application, we employ the LSM hook hidden in *do_execve*function, in which a hypercall is used to get user info from BitVisor and make decisions according to policy rules about whether the application can be run by a particular user. The time cost of *do_execve* function is tested in order to show the overhead cost caused by employing RTA mechanism.

The statistical data obtained is listed in Table 3. Some common applications such as Gedit and Firefox are chosen to do the experiment. To avoid the influence from temporary factors, we ran each application ten times and recorded the average time costs. The result shows that RTA mechanism costs an average value of 2740.6 microseconds system overhead when running applications, which is very small and acceptable.

Security Analysis

In this section, we focus on the enhanced security about our proposed user authentication mechanism RTA, which is reflected in the following aspects:

Lightweight VMM. The VMM we choose to use is the lightweight BitVisor, also called parapass-through hypervisor. By using device drivers of the guest OS to handle devices and eliminating the components for sharing and protecting system resources among VMs [14], BitVisor is able to minimize the code size of itself. Compared to traditional hypervisors, BitVisor is at a lower risk of being attacked and has a smaller system overhead.

Secure Authentication. BitVisor employs smart cards to authenticate users, the challenge and response authentication procedure is hard for malicious users to crack into. Also, with the unreadable users' private key stored in the smart card, it is safer to use compared to the traditional password way.

Isolated Storage. The user information is stored securely in the safe zone of BitVisor address space, which is unreachable from guest OS. The hooks employ hypercall to directly get user information in real time, leaving no room for malicious users to subvert the user identity.

Safe from OS Attack. BitVisor is in the VMM layer under guest OS, a guest OS cannot even notice the existence of BitVisor, which means the attacks on guest operating system are in vain.

Future Work

We focused on providing a trusted user authentication mechanism for Linux security modules. Based on our proposed authentication mechanism RTA and LSM framework, an access control project namely EWL was implemented for administrators so that policy can be put on ordinary users to limit their authority when execute an ELF file. However, we currently employ just the *fork* hook in LSM framework to achieve basic access control, additional hooks applied in EWL project could make it more functional and helpful.

Conclusion

In this paper, a novel user authentication mechanism for Linux security modules has been proposed, which is supported by BitVisor ID management framework. By employing user info from VMM layer in real time, it can be ensured that the security module get trusted user identity to enforce policy.

We have completed a prototype of access control tool called EWL. Administrators can configure policy rules through EWL application and put limits on an ordinary user's authority to execute an ELF file.

We believe that with more improvements added to the EWL project, it can be better used by administrators to provide enforcement security to the computer system.

Acknowledgement

The authors would like to thank the reviewers, Kai Li and Gangru Xuefor insightful comments on earlier drafts of this paper, Jianping Wang for helping the implementation of RTA mechanism, and Huan Liu for discussing the design of the architecture of EWL.

This work was supported by National Science and Technology Major Projects of China (Grant No.2014ZX01029101-002).

References

[1] C. Wright, C. Cowan, S. Smalley, J. Morris, et al. Linux Security Modules: General Security Support for the Linux Kernel, in: USENIX Security Symposium, 2002, pp. 1-14.

[2] S. Smalley, C. Vance, and W. Salamon Implementing SELinux as a Linux security module, NAI Labs Report. 1 (2001) 139.

[3] M. Bauer. Paranoid penguin: AppArmor in Ubuntu 9, Linux Journal. 2009 (2009) 9.

[4] H. Chen, N. Li, and Z. Mao Analyzing and Comparing the Protection Quality of Security Enhanced Operating Systems, in: NDSS, 2009, pp. 11-16.

[5] Information on http://www.bitvisor.org/

[6] S. E. Hallyn and A. G. Morgan. Linux capabilities: Making them work, in: Linux Symposium, 2008.

[7] B. Jain, C.-C. Tsai, J. John, and D. E. Porter Practical techniques to obviate setuid-to-root binaries, in: Proceedings of the Ninth European Conference on Computer Systems, 2014, p. 8.

[8] H. Chen, D. Wagner, and D. Dean. Setuid Demystified, in: USENIX Security Symposium, 2002, pp. 171-190.

[9] A. Ganti. Plan 9 authentication in Linux, ACM SIGOPS Operating Systems Review. 42 (2008) 27-33.

[10] M. Hirano, T. Okuda, E. Kawai, and S. Yamaguchi. Design and Implementation of a Portable ID Management Framework for a Secure Virtual Machine Monitor, Journal of Information Assurance and Security (JIAS), Dynamic Publishers. 2 (2007) 211-216.

[11] M. Hirano, T. Shinagawa, H. Eiraku, S. Hasegawa, et al. Introducing role-based access control to a secure virtual machine monitor: security policy enforcement mechanism for distributed computers, in: Asia-Pacific Services Computing Conference, 2008. APSCC'08. IEEE, 2008, pp. 1225-1230.

[12] M. Hirano, E. Kawai, H. Eiraku, K. Kato, et al. Portable ID Management Framework for Security Enhancement of Virtual Machine Monitors, INTECH Open Access Publisher, 2009.

[13] M. Hirano, D. W. Chadwick, and S. Yamaguchi. Use of role based access control for security-purpose hypervisors, in: 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 2013, pp. 1613-1619.

[14] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, 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, 2009, pp. 121-130.