Int. J. Communications, Network and System Sciences, 2012, 5, 105-110
http://dx.doi.org/10.4236/ijcns.2012.52014 Published Online February 2012 (http://www.SciRP.org/journal/ijcns)

Research on User Permission Isolation for Multi-Users Service-Oriented Program

Li Yu[1,2,3], Jiang Wei[1,2,3*], Lin Li[1], Zhan Jing[1,4], Liang Peng[1], Yingxu Lai[1], Shupo Bu[5]
[1]College of Computer Science and Technology, Beijing University of Technology, Beijing, China
[2]State Key Laboratory of Information Security, Institute of Software, Chinese Academy of Sciences, Beijing, China
[3]Key Laboratory of Information and Network Security, 3rd Research Institute, Ministry of Public Security, Shanghai, China
[4]Key Lab of Network Security and Cryptology, Fujian Normal University, Fuzhou, China
[5]Department of Electronic Engineering, Suzhou Insititue of Industrial Technology, Suzhou, China
Email: *j8w8@sohu.com

Received November 30, 2011; revised January 14, 2012; accepted January 26, 2012

ABSTRACT

For the super user privilege control problem in system services, a user permission isolation method is proposed. Based on virtualization technology, the permission limited environments are constructed for different users. According to privilege sets, the users, mapping relations are built among users, isolated domains and program modules. Besides, we give an algorithm for division of program permissions based on Concept Lattices. And the security strategies are designed for different isolated domains. Finally, we propose the implications of least privilege, and prove that the method eliminates the potential privileged users in system services.

Keywords: Least Privilege; Virtualization; Isolation; Privileged User; Domain; System Service

1. Introduction

Currently, most of the operating systems (OSs) use identity-based authorization mechanism, and super user has all the permissions of the whole OS [1]. Once intruders get the identity of super user, they will get complete control of OS [2,3]. For example, a flaw in SENDMAIL prohibits the *setuid* operation, which results in the revocation of root privileges fails and intruders may use the flaw [3].

In OS, users access resources through processes. However, the system services have the specificity of running in the privileged kernel space. Once there are security vulnerabilities in service programs, the guest users will gain access to the privileged kernel space through system services. As a result, guest users will be able to access all system resources. In this case, guest becomes a potential super user. According to statistics, there will be a vulnerability at least per thousand lines of codes, so the potential super user is inevitable ξ.

For potential super user problem, a user permission isolation method is proposed in the paper. The permission limited environments are constructed for users with different permissions. Then we give a formal security policies and an algorithm for division of program permissions. Finally, we propose the implications of least privilege, and prove that the method eliminates the potential super users in system services.

The rest of the paper is organized as follows. Section 2 introduces the basic definition of the system. Section 3 presents user space isolation model (*USIM*), designs security policies and the algorithm for division of program permissions. Section 4 analyzes the proposed method. Section 5 describes related work. Finally, conclusions and future work are presented in the last section.

2. Basic Definitions

The basic formal definitions are as follows.

Definition 1 A system M is composed of

- a set S of states, with an initial state s_0.
- a set U of users, with an user is not only the OS user, but also the application's user. For example, Samba, Remote Desktop, etc. The super user is denoted as U_R, while guest user is denoted as U_R.
- a set P of processes.
- a set A of actions.
- a set O of objects.
- together with the functions $host$ and pri:
- $host$: $host(\xi)=\delta$, if $\xi \in P$, then the parent process of ξ is δ or user δ starts process ξ. If $\xi \in O$, then the owner of ξ is δ.
- pri: $pri_s^p(\xi,\delta)=A$, said in state s, from the view of the process $p \in P$, the subject has the permission

*Corresponding author.

A to access the object $\delta \in O$. The set of permissions for processes is denoted as $pri(p)$, while the user's is denoted as $pri(u)$.

Together with the relation $\rightarrow$:

- $\rightarrow$ denote the permission inheritance relation:

$$\begin{cases} pri(u) \rightarrow pri(p), & \text{if } host(p) = u,\ u \in U \\ pri(q) \rightarrow pri(p), & \text{if } host(p) = q,\ p, q \in P \end{cases}$$

When a user *u* starts a process *p*, *p* inherits the permissions from *u*. When a process *q* forks a child process *p*, *p* inherits the permissions from parent process *q*.

Definition 2 $\forall \xi \in (U \cup P)$,

$$space(\xi) = \left\{ \delta \middle| pri_s^M(\xi, \delta) = a \neq \varphi,\ \delta \in O \right\}$$

In state *s*, from the view of *M*, $space$ represent set of objects which subject ξ has non-empty permissions to access. We call it permission space (PS). In the classical Windows or Linux OS, PS can be divided into two categories, one is privileged space $SPACE_R$, which users and processes have all privileges to access all system resources, the other is non-privileged user-space $SPACE_R$, where the permission is limited and privileged operations are forbidden.

Definition 3 $classify(U, \ell) = \{cf\}$, $cf \subset U$.

We classify the users *U* into several sets according to the rule ℓ. If $pri(cf_1) \subseteq pri(cf_2)$, we call cf_2 dominates cf_1, denoted as $cf_2 \geq cf_1$.

Definition 4 The inclusion relation $\prec$:

If $pri(\xi) \rightarrow pri(\delta)$, then $space(\delta) \prec space(\xi)$.

That is, the permission space must satisfy the inclusion relation if two subjects meet the inheritance relation. Obviously, the permissions of a process must inherit from the parent process or the user who starts it.

Theorem 1 If u satisfies $pri_s^p(u, o) \neq \varphi$, $host(p) = u_0$, $u_0 \in U_R$, then $space(u) \prec_P SPACE_R$.

Proof: According to permission inheritance relation, $host(p) = u_0 \Rightarrow pri(u_0) \rightarrow pri(p)$, since $u_0 \in U_R$, then $pri(U_R) \rightarrow pri(p)$, since Definition 4, we have

$$space(p) \prec space(U_R) = SPACE_R \qquad (1)$$

Since Definition $\prec$, we have

$$pri_s^p(u, o) \neq \varphi \Leftrightarrow space(u) \prec_p space(p) \qquad (2)$$

Hence, we get $space(u) \prec_P SPACE_R$. The proof is completed.

The theorem indicates that if user u has access to a program which runs in privileged space, u elevates the privileges of his own. In this case, if u is a guest user, he will become a potential super user. The theorem proved that the potential super user does exist. Similarly, a guest user who has less permissions can get more authority from the guest user which has more permissions through penetration into the program.

In proof of theorem 1, if $u_0 \in U_R$ does not hold, the conclusion will not hold. Hence, we get a new idea that we can create a copy of the program which runs in unprivileged space. And the user access to the program will be redirected to the copy; the potential super user problem will be solved. That is the core idea of this paper.

3. User Isolation Method

Multi-user remote logon procedure in Linux OS is as follows. First, login process authenticates users' identity. Then, login process creates the shell for different users. Based on remote login procedure, this section gives the user space separation model (*USIM*).

3.1. User Space Separation Model

As shown in **Figure 1**, *USIM* is divided into three layers: basic function layer, isolation management layer, virtual execution layer.

As the basis of division of the system program, *USIM* constructs isolated domains for different categories of users and provides security management functions. The functions of three layers in *USIM* are as follows.

1) Basic function layer

Basic function layer processes the access request first, for example, the user identity authentication. Then, encapsulate the layer-related security context information. At last, send the information to isolation management layer.

2) Isolation management layer

This layer is responsible for context management, virtualization management and policy management. It is mainly to address how to construct user isolated domains which does not interfere each other, and to ensure the user's permissions are restricted in the domain.

- Context Management

Context Management maintains current context information about active users, manages the mapping relation between the users and isolated domains. Moreover,

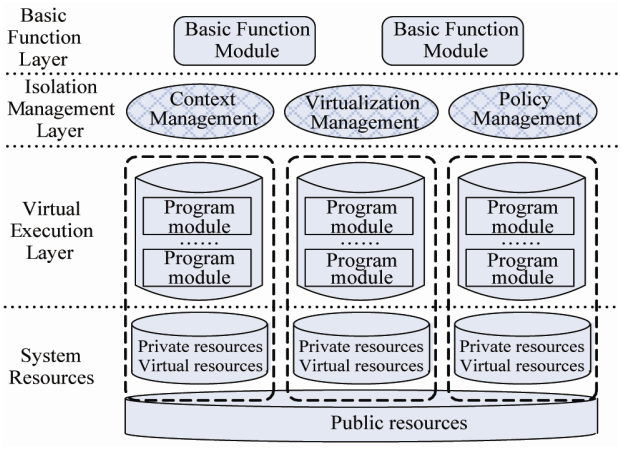

Figure 1. User space separation model.

it encapsulates the necessary security context again, and sends to the virtualization layer.

- Virtualization Management

Virtualization Management is responsible for managing the virtual environment. In a virtual environment, three main aspects should be considered, including the executive program virtualization management, network connectivity reconstruction management and resources management. The three mechanisms are discussed in another paper, which are not the focus of the paper.

- Policy Management

Policy Management is responsible for policy formulation and configuration, including the permission set of users and programs, the communication policy and security policy adjustment.

3) Virtual execution layer

It constructs the executive subject for the isolated domain. Each domain is corresponding to an active process in OS. It realizes the users' rights isolation through the isolated processes.

We give the formal definition of the isolated domain as follows.

Definition 5 A user isolated domain *DOM* is composed of

- *ID* is the unique identifier of a domain,
- a set *CONTEXT* of security context information, including user credentials, sessions, etc.
- a set *PRI* of permissions, which denotes $pri(DOM)$,
- a set *C* of program modules,
- a set *RES* of resources in the domain.

together with the function *communicate*: *communicate* function denotes the communication among different *DOMs*. *Communicate* (*SourceDOM*, *TargetDOM*) = {*TRUE*, *FALSE*} indicates whether the two domains communication is permitted.

Before constructing a *DOM*, in the actual application process, we classify the users to several sets according to user categories. For example, we do $classify(U,R)=\{cf\}$ operation to classify the users in accordance with user role. We give the algorithm for constructing a *DOM* as shown in **Figure 2**.

3.2. Security Policy

Information flow among domains may elevate the privileges. However, normal information flow does exist to complete the task. This section gives the security policies to prevent the privilege elevation.

Rule 1 Inter-domain communication rule

$\forall dom_1, dom_2 \in DOM$, $dom_1 \neq dom_2$, if $pri(dom_1)=cf_1$, $pri(dom_2)=cf_2$, Then

$$communicate(dom_1, dom_2)=\begin{cases} TRUE, & \text{if } cf_1 \geq cf_2 \\ FALSE, & \text{else} \end{cases}$$

```
Dom DomConstruct( )
{
    //get the security context
    get(CONTEXT);
    //get user information in context
    UserID = getUid(CONTEXT);
    //get user's permissions
    cf_i = search({cf}, UserID);
    //search whether the target exists
    for each dom_i
        if (pri(cf_i) == pri(dom_i))
            return dom_i;
    ComCluster = null;
    //solving the modules set according to the permissions
    for each c_i
        if ( pri(c_i) ⊆ pri(cf_i) )
            insert(ComCluster);
    // encapsulate the modules to program
    P = encapsulate(ComCluster);
    //start the process using identity UserID
    dom = fork(UserID, P);
    return dom;
}
```

Figure 2. *DOM* constructing algorithm.

Inter-domain communication is permitted only if the source *DOM*'s permission set dominate the target set. The communication initiated by target *DOM* which is dominated, may result in the privilege elevation, so must be forbidden.

Rule 2 Communication rule between *DOM* and unrelated process

If $C \in dom$, $C \subset p$, $\forall C' \subset p'$, $p' \neq p$, $C' \in dom'$, then $communicate(dom, dom')=FALSE$.

That is, communication between *DOM* and unrelated process should be prohibited.

Rule 3 Access rule to *DOM*

$\forall dom \in DOM$, if $\exists u \in U$, *u* satisfies $pri_s^{dom}(u,o) \neq \varphi$, then $u \in dom.CONTEXT$.

The rule limits the users' access range in the corresponding *DOM*. In addition, inverse negative proposition shows that, except the users in *DOM*, other users are prohibited from access to *DOM*.

Definition 6 Resources in a *DOM* are

$$RES_{DOM} = RES_{PUB} + RES_{PRI} + RES_{CALL}$$

- a set RES_{PUB} of public resources, which are accessed by all *DOM*s in service program, such as files corresponding to FTP program.
- a set RES_{PRI} of private resources, which are created in running process, such as temporary files, etc.
- a set RES_{CALL} of resources, which are called in host OS during the run-time, such as dynamic link libraries in Windows OS.

Rule 4 Resource access control rules

1) If $C \in dom$, $C \subset p$, $\forall res \in RES_{PUB}$, $RES_{PUB} \subset RES_{dom}$, $\forall p' \in P$ and $p' \neq p$, p' satisfies $pri_s^M(p', res)=\varphi$.

Public resources RES_{PUB} only can be accessed by the corresponding process, while other processes are refused.

2) $\forall dom, dom' \in DOM$, $dom \neq dom'$, $\forall res \in RES_{PRI}$, $RES_{PRI} \subset RES_{dom}$, res satisfies $pri_s^P(dom', res) = \varphi$.

Only users in *DOM* can access the private resources RES_{PRI} in the same *DOM*.

3) $\forall res \notin (RES_{PUB} \bigcup RES_{PRI} \bigcup RES_{CALL})$, $\forall dom, dom.C \subset p$, dom satisfies $pri_s^P(dom, res) = \varphi$.

Except public resources, private resources and called resources, any resource can be accessed by the corresponding *DOM*.

3.3. Module Partition Based on Concept Lattice

Information flow among domains may elevate the privileges. However, normal information flow does exist to complete the task. This section gives the security policies to prevent the privilege elevation.

A few existing researches have provided some methods for constructing an environment to prevent privilege elevation. For example, Price [4] used the way that directly running a copy of the program in the virtual environment. However, the above method is not suitable to multi-user services programs, such as FTP. First, each user runs the same program; some users will get more permissions than necessary. Second, several copies of the program run simultaneously will result in performance lost in OS. Hence, the paper proposes the division method for program permissions based on Concept Lattice. It is the premise of building a virtual environment.

Concept lattice is a conceptual hierarchy according to binary relation between objects and properties. We define a program fragment as the object in concept lattice, the property as permission. By looking for the same permissions in program fragments, we achieve the division aim. The basic concepts of concept lattice are as follows [5].

Definition 7 Permission context is a triple (I, PRI, R). I is the set of program fragments which are composed of one or more functions. R is the relation between I and PRI .

$$\lambda(I) = \{pri \mid pri \in PRI \bigcap \forall i \in I, (i, pri) \in R\}.$$

λ is the public permissions set in fragments.

$$\eta(PRI) = \{i \mid i \in I \bigcap \forall pri \in PRI, (i, pri) \in R\}.$$

η is the public program fragments set in permissions.

Definition 8 If $\lambda(I) = PRI$ and $\eta(PRI) = I$ does not hold, then $C = (I, PRI)$ is called a concept. $\uparrow$ denotes the top concept, and $\downarrow$ denotes the bottom concept.

Partial order relation $\propto$ between concepts: $c_1 = (I_1, PRI_1)$, $c_2 = (I_2, PRI_2)$, $c_1 \propto c_2 \Leftrightarrow I_1 \subseteq I_2 \bigcup PRI_2 \subseteq PRI_1$. c_1 is called sub-concept, c_2 is called parent concept. $cchild(c)$ is the function seeking the sub-concept, and $cparent(c)$ is the function seeking the parent concept.

$L: (C, \propto)$ is called a concept lattice.

Definition 9 If $c_1 = (I_1, PRI_1)$, $c_2 = (I_2, PRI_2), \ldots,$ $c_n = (I_n, PRI_n)$, they satisfy $\sum_{j=1}^{n} I_j = I$, and $I_j \bigcap I_k = \varphi (j \neq k)$, the set of C is called a Concept Division D .

A concept corresponds to a program module which is composed of fragments. Hence, concept division is also bound to the corresponding to module partition. Therefore, to achieve a reasonable program partition, we should just find the concept division. Algorithm for module partition is as shown in **Figure 3**.

The solution based on the above algorithm is often not unique. Each concept division corresponds to a module partition. Reference [6] gives a method to distinguish which partition method is more targeted, and we will not go further on this issue.

4. Security Analysis

Buyens [6] gives a standard to test whether a program design meets the principle of least privilege (PLP), that is if and only if each component meets PLP. *DOM* is the

```
Void SubPartition (C Cf, D M )
//recursive algorithm
{
    if (Cf ==↑)
        return;
    for each ci ∈ Cf
    {
        //get parent concept
        c' = cparent(ci);
        //replace the parent concept with its //sub-concepts
        Cf'= Cf + c' –cchild(c');
        if (C'f ∈ D)
            insert Cf' into M ;
        //calculate recursively
        SubPartition (Cf', M );
    }
}
Void Partition (D M )
//solving concept division
{
    //get parent concept set of ↓
    Cj = cparent(↓);
    if (Cj ∈ D)
        Insert Cj into M ;
    SubPartition (Cj, M );
}
```

Figure 3. Algorithm for module partition.

basic unit of service programs, so we think it should include three aspects in PLP.

1) Isolation Requirements. Users complete their task in an isolated environment, and the information exchange between internal and external environment must be controlled.

2) Permissions Limited. The users' permissions are necessary, and there are no more authorizations than necessary.

3) Resource Access Restricted. Resources in isolated environment should be protected to prohibit external access. In addition, users in isolated environment can not access resources outside to prevent privilege elevation.

For isolation requirements, any two different *DOM*s belong to two different processes, so OS realizes the isolation of *DOM*. Communication between different *DOM*s relies on rules 1 and rule 2 to prevent external interference. Therefore, *USIM* achieves the isolation requirements.

For permissions limited, *USIM* realizes program partition based on concept lattice. First, $\forall c \in C$, $C \in dom$, c satisfies $pri(c) \subseteq pri(cf_i)$, hence the set C must satisfy $pri(C) = pri(cf_i)$. Second, the permissions of a user u in *DOM* satisfy $pri(u) \subseteq cf_i$. Third, in *DOM*, $host(dom.C) = u'$, according to inheritance relation we get $pri(u') \to pri(dom)$, hence $pri(dom) = pri(u')$. Since $pri(u') = pri(cf_i)$, $pri(dom) \subseteq pri(cf_i)$ holds. Therefore, users, *DOM*s and program modules are interconnected by the set of permissions cf_i. As a result, *USIM* just gives the user the appropriate permissions, and it achieves the Permissions Limited requirement.

For Resource Access Restricted requirement, security policies prohibit external users accessing resources in *DOM*. Besides, rules 3 and rule 4 ensure that users in *DOM* can only access resources corresponding to *DOM*. Therefore, *USIM* meets the resource access restricted requirement.

Finally, *USIM* meets the requirements of PLP, and therefore can eliminate the potential super users in a service program.

However, in order to realize the program partition, we have to clear the source codes of the program, which is a limitation of *USIM*.

5. Related Work

Least privilege is the classic method to achieve permissions restriction [7,8]. Chen proposed a check method for against PLP [9]. It achieves policy compliance checks by intercepting system call. However, in OS, the user in process context is the one who starts the process. For the case multiple users process a system service, it is impossible to distinguish the different users through users' identity.

Privilege separation is a privilege restricted method by partitioning the program into several modules [10]. Douglas proposed an idea that partitioning an application into two parts [11], a privilege server and the main application without privilege respectively. However, it is difficult to develop appropriate partitioning strategy. David inherited the idea, and partitioned the program into the privilege monitor and slave without privilege [12].

Virtualization provides each user a runtime environment by virtualization technology. Jail [13,14] provides an operating system virtualization layer technology for FreeBSD. The access is limited to Jail, and the information flow inside and outside of Jail is forbidden. However, it is a full-virtualization technology, for each Jail must have a copy of system resources. It reduces the efficiency of the OS. Similarly, Solaris Zone [4] took this idea. FVM [15] is a feather-weight Windows based virtual machine. It achieves the isolation by the namespace virtualization. However, these studies can not solve users' permissions isolation in the same program problem.

6. Conclusion and Future Work

For potential super user problem, the paper introduces a user permission isolation method. Based on Concept Lattices, the algorithm for division of program permissions is proposed. Using virtualization technology, *USIM* constructs the permission limited environments for different users. Besides, we develop security strategies for *USIM*. Finally, we prove that *USIM* meet the principle of least privilege, and the method eliminates the potential privileged users in system services.

7. Acknowledgements

This research is funded by 863 National High Tech Research and Develop Plan Project (2009AA01Z437), 973 National Key Fundamental Research Development Plan Project (2007CB311100), Opening Project of State Key Laboratory of Information Security (Institute of Software, Chinese Academy of Sciences) (No. 04-04-1), Opening Project of Key Lab of Information Network Security, Ministry of Public Security (No. C11610) the program "Core Electronic Devices, High-End General Purpose Chips and Basic Software Products" in China (No. 2010ZX01037-001-001), Funds of Key Lab of Fujian Province University Network Security and Cryptology (2011009) and Doctor Launch Fund in Beijing University of Technology (X00700054R1764), the natural science foundation of No. X0007211201101 Beijing City under Grant No. 4123093, National Soft Science Research Program (No. 2010GXQ5D317), Opening Project of Jiangsu Province Web TV Research and Development Center for Engineering Technology (No. SIIT111002).

REFERENCES

[1] R. Stevens, "Advanced Programming in the UNIX Environment," Addison-Wesley Publishing Company, 1992.

[2] H. Chen, D. Wagner and D. Dean, "Setuid Demystified," *Proceedings of the* 11*th USENIX Security Symposium*, San Francisco, 05-09 August 2002, pp. 171-190.

[3] Sendmail Inc. Sendmail Workaround for Linux Capabilities Bug, 2009. http://www.Sendmail.org/Sendmail.8.10.1.LINUX-SECURITY.txt

[4] D. Price and A. Tucker, "Solaris Zones: Operating System Support for Consolidating Commercial Workloads," *USENIX* 18*th Large Installation System Administration Conference* (*LISA*'04), Atlanta, 14-19 November 2004, pp. 241-254.

[5] C. Lindig and G. Snelting, "Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis," *Proceedings of the* 19*th International Conference on Software Engineering*, Boston, May 1997, pp. 349-359.

[6] K. Buyens, B. D. Win, and W. Joosen, "Resolving Least Privilege Violations in Software Architectures," *Proceedings of the* 5*th International Workshop on Software Engineering for Secure Systems*, Vancouver, 19 May 2009, pp. 9-16.

[7] T. E. Levin, C. E. Irvine and T. D. Nguyen, "A Least Privilege Model for Static Separation Kernels," Technical Report NPS-CS-05-003, Center of Information Systems Security Studies and Research, Naval Postgraduate School, October 2004.

[8] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, Vol. 63, No. 9, 1975, pp. 1278-1308. doi:10.1109/PROC.1975.9939

[9] S. Chen, J. Dunagan, C. Verbowski and Y.-M. Wang, "A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities," *Proceedings of Network and Distributed System Security Symposium*, San Diego, 3-4 February 2005, pp. 42-53.

[10] K. Buyens, R. Scandariato and W. Joosen, "Composition of Least Privilege Analysis Results in Software Architectures," *Proceeding of the* 7*th International Workshop on Software Engineering for Secure Systems*, Waikiki, 22 May 2011, pp. 29-35.

[11] D. Kilpatrick, "Privman: A Library for Partitioning Applications," *Proceedings of Freenix*, San Antonio, 12-14 June 2003, pp. 273-284.

[12] D. Brumley and D. Song, "Privtrans: Automatically Partitioning Programs for Privilege Separation," *Proceedings of the* 13*th conference on USENIX Security Symposium*, San Diego, 9-13 August 2004, p. 5.

[13] P. H. Kamp and R. N. Watson, "Jails: Confining the Omnipotent Root," 2*nd International System Administration and Network Engineering Conference* (*SANE*'00), Maastricht, 2000, pp. 1-15.

[14] S. Evan, "Securing FreeBSD Using Jail," *System Administration*, Vol. 10, No. 5, 2001, pp. 31-37.

[15] Y. Yu, F.-L. Guo, S. Nanda, L.-C. Lam and T.-C. Chiueh, "A Feather-Weight Virtual Machine for Windows Applications," *Proceedings of the Second ACM/USENIX Conference on Virtual Execution Environments* (*VEE*'06), Ottawa, 14-16 June 2006, pp. 24-34.

