VOL. 12, NO. 12, JUNE 2017 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2017 Asian Research Publishing Network (ARPN). All rights reserved.

www.arpnjournals.com

SYSTEM CALL AUTHORIZATION IN LINUX BY A SECURE DAEMON

Vivek Radhakrishnan, Hari Narayanan and Shiju Sathyadevan
Amrita Center for Cybersecurity Systems and Networks Amrita School of Engineering, Amritapuri Amrita Vishwa Vidyapeetham
Amrita University, India
E-Mail: hari@am.amrita.edu

ABSTRACT

Compromises on data integrity and confidentiality have exposed the vulnerability of security architectures of traditional Linux-based operating systems against malicious attacks. Minimized functionality and increased complexity restrict the effectiveness of traditional approaches such as sandboxing in handling attacks. We proposed architecture based on restricted user privileges and authorization to secure the Linux operating system. We developed a Secure Daemon to authorize the system calls. All the system calls invoked by user processes are redirected to secure daemon using a dynamic dispatch mechanism (wrapper functions) implemented on top of the existing libraries. Our approach ensures that critical system resources are protected in the event of an attack. Since the major elements of the proposed system operate at the user level, it is portable across all Linux distributions.

Keywords: linux, authorization, system calls, secure daemon, wrapper functions, dynamic dispatch mechanism.

INTRODUCTION

All the operating systems employ some kind of access control mechanisms. Authentication and authorization are two very important steps in an access control mechanism. Authentication is the process of verifying or validating the identity provided by a person or an entity. Access control involves access policy definition and access policy enforcement. Policy definition is the specification of access rules which is also called authorization. The next step is the policy enforcement which happens when an access request is actually received. A decision is taken whether to grant or deny the access.

In traditional Linux operating systems, discretionary access control (DAC) mechanism is used. DAC uses identity-based authorization. A process can be identified by its userid and the groupid. When a process tries to access a system resource, the kernel verifies its effective userid and the effective groupid and matches it with the resource permissions. If permission is matching, a resource handle is returned to the process. The process can perform its intended operation on the resource using this resource handle. If permissions do not match, the process is denied access and an error code is returned instead of the resource handle. If a process runs on behalf of a user, it gets all the privileges of the user; in other words, the process gets the ambient authority. If a user downloads a malware affected application from the internet and executes it, the application runs with the ambient authority. This is a dangerous situation. The application has the potential to cause significant damages to the system.

We have developed a new security architecture which builds a security ticket based authorization on top of the existing identity-based authorization. This architecture ensures that the application or the process has the exact privileges just enough to perform its intended tasks. This effectively puts the application into a restricted environment or in a sandbox. So even if the application is malware affected it can do the least amount of damage as it is running in a sandbox. This work is a continuation of our previous work [1]. We have come up with a modular design in which the entire architecture is broken down into modules for the sake of implementation. We have partially implemented few modules.

Usually, a process does not make system calls directly. It uses some API library like glibC and invokes the API with the same name as the system call. The APIs in libC are wrapper functions to system calls. They hide the lower level complexities from the user processes by providing high-level simple interfaces. There will be assembly code in these wrapper functions which causes an interrupt and context switch from user mode to kernel mode. Before that, the system call number is placed in process registers and parameters are placed in kernel CALL stack. In the kernel, authorization is done and the actual system call is invoked by the system call handler.

We have implemented another wrapper function to the glibC API, which by itself is a wrapper for the system calls. When the process invokes the glibC API, our wrapper function is executed. In the normal mode, the actual API is executed as in the traditional Linux. In the secure mode, the parameters of the APIs are transferred to a Secure Daemon via Unix Domain Socket which does an additional authorization on top of the existing authorization using the effective userid and the effective groupid of the process. The resource handle is returned back to the wrapper function by the Secure Daemon which in turn returns it to the original user process. Also, we have analyzed our system to show that the latency incurred is very small.

The remainder of this paper is organized as given below. Section II describes the design. Section III describes the implementation. Section IV presents the analysis and result. Section V presents our conclusion and section VI describes the related work.

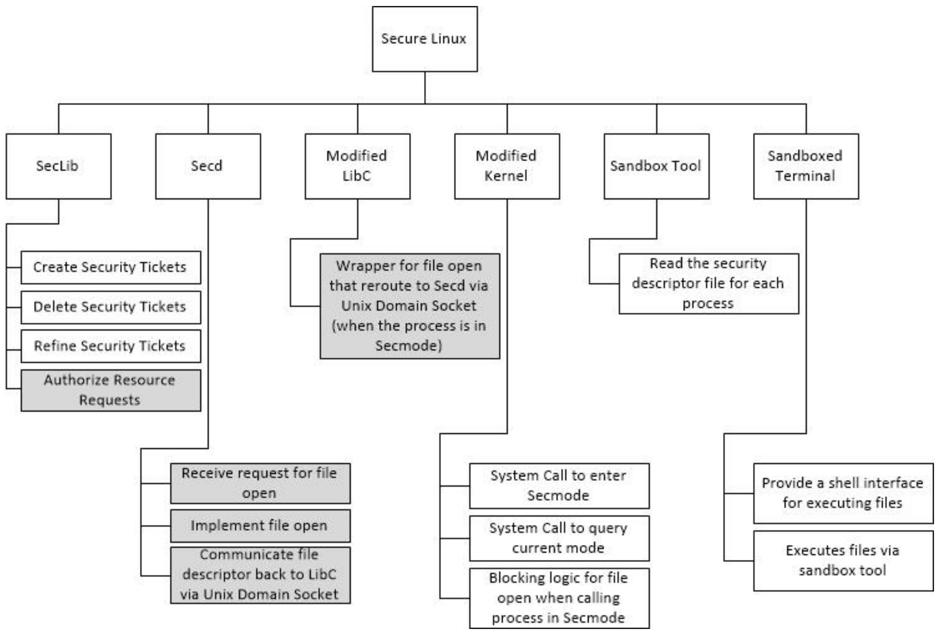

Secure Linux
SecLib
Secd
Modified LibC
Modified Kernel
Sandbox Tool
Sandboxed Terminal
Create Security Tickets
Delete Security Tickets
Refine Security Tickets
Authorize Resource Requests
Wrapper for file open that reroute to Secd via Unix Domain Socket (when the process is in Secmode)
Read the security descriptor file for each process
Receive request for file open
Implement file open
Communicate file descriptor back to LibC via Unix Domain Socket
System Call to enter Secmode
System Call to query current mode
Blocking logic for file open when calling process in Secmode
Provide a shell interface for executing files
Executes files via sandbox tool

Figure-1. Top down modular design of the security architecture.

DESIGN

The proposed secure Linux architecture [1] is broken down into several modules in a top-down fashion for the sake of implementation. Each module is modifiable without affecting the other modules. The major modules are SecLib, Secd, Modified LibC, Modified kernel, Sandbox Tool and Sandboxed terminal as shown in the Figure-1. The shaded boxes at the bottom of the Figure-1 represent modules which are already implemented. Implementation of the unshaded boxes is in progress.

SecLib

Authorization in our security architecture is based on Security Tickets. When a process issues a system call, it should be accompanied by the Security Ticket. Secd will verify the Security Ticket for authorizing the system call. SecLib is a library which contains procedures to create, delete and refine Security Tickets. Also, it contains procedures to authorize resource requests.

SecD

Secd is a security daemon which always runs on startup. It has a unique userid and groupid for its protection. All the critical system calls are rerouted to Secd for authorization. We have implemented the authorization of open system call which opens a file from the hard disk. When a process invokes open or fopen API, it will be rerouted to the Secd for authorization via Unix domain socket by the modified LibC. Secd will verify the Security Ticket for authorization. If authorization is successful, then Secd would open the file and return the file descriptor back to the process via Unix domain socket.

Modified LibC

Wrapper functions are written for the glibC APIs open and fopen which would reroute the system call parameters to the Secd for authorization if the process is in the secure mode. In the normal mode, the wrapper function invokes the original system call itself directly. Wrapper function implementation is in progress for the other critical system calls like socket create, socket bind, file create etc.

Modified kernel

The Linux kernel is modified by adding new system call which switches a process from normal mode to the secure mode. Also, we need another system call to query the current mode of the process which invoked the glibC API.

Sandbox tool

This module reads the Security Descriptor File of a process and creates a sandboxed child with the privileges specified in the descriptor file.

Sandboxed terminal

This is a shell interface for the user to execute programs. Sandboxed Terminal uses Sandbox Tool internally.

IMPLEMENTATION

The proposed architecture [1] for securing Linux is shown in Figure-2. When the system is booted, the secure daemon (Secd) automatically gets invoked in the background. The LibC APIs are redirected to the Secd via Unix domain socket in secure mode, in lieu of being converted into a system call. Our system uses a dynamic dispatch mechanism (Wrapper function) at the user level to redirect the system call to a Secure Daemon. Dynamic dispatch mechanism overrides the existingLibC APIs. Our current implementation comprises of two main components: Modified LibC and a Secure Daemon.

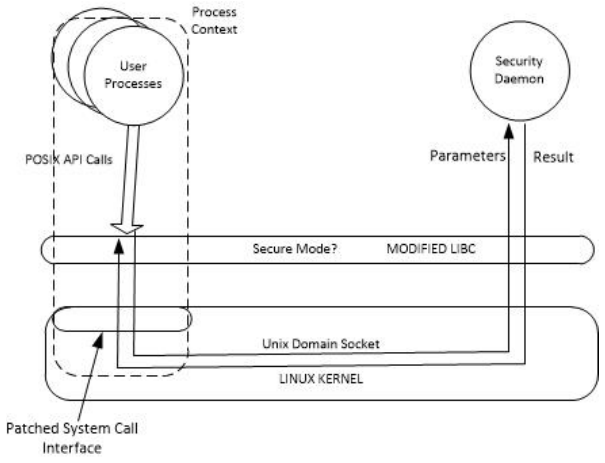
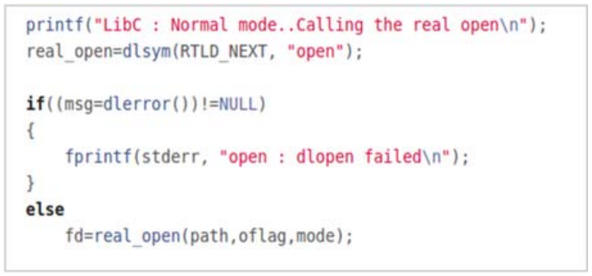


Figure-2. System interface architecture of proposed system.

Modified LibC

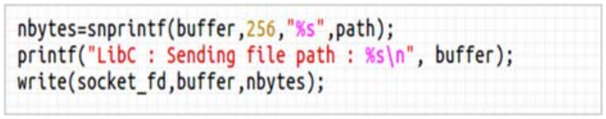
In traditional Linux architecture, when a system call gets invoked, LibC stores the system call name along with its corresponding arguments in the system register. This generates an interrupt which enables the system to switch from the user mode to the kernel mode.

In our approach, a dynamic dispatch mechanism implemented at the user level redirects the system call to a Secure Daemon. The dynamic dispatch mechanism is done by wrapper functions that override the existing LibC APIs. A shared library is created that contains all the wrapper functions. We have to preload the shared library in order to execute the wrapper function instead of the original API. To preload the shared library, we use a shell environment variable called LD_PRELOAD. To call back the original function in the event of a system call override, we use dlsym() function. dlsym() function is used to obtain the address of the original glibC function as shown in Figure-3. This address is required to invoke the original function if the process is running in the normal mode. The wrapper function in the shared library reroute the glibC API to the secure daemon via Unix Domain Socket (UDS) if the process is running in the secure mode. The parameters are send to the Secd for authorization as shown in Figure-4.



```
printf("LibC : Normal mode..Calling the real open\n");
real_open=dlsym(RTLD_NEXT, "open");

if((msg=dlerror())!=NULL)
{
    fprintf(stderr, "open : dlopen failed\n");
}
else
    fd=real_open(path,oflag,mode);
```

Figure-3. Code snippet of calling real open using dlsym().



```
nbytes=snprintf(buffer,256,"%s",path);
printf("LibC : Sending file path : %s\n", buffer);
write(socket_fd,buffer,nbytes);
```

Figure-4. Code snippet of parameter passing via UDS in secure mode.

The operation of UDS is similar to that of remote procedure calls. In contrast to other data communication models, UDS is capable of sending file descriptors and process credentials (process id, user id, group id) between processes.

Secure daemon

As discussed in section II, SecD is responsible for the issue of security tickets and the system call authorization. It has a unique userid and groupid. When Secd receives an open request from the wrapper function, it retrieves the process id, effective user id (EUID) and effective group id (EGID) of the initiated process from the UDS and is compared against the file status. If the file permissions are matching with the process credentials, SecD authorizes the system call and returns the file handle to wrapper function via Unix Domain Socket.

To further examine the working of the proposed system, we consider the sequence of events which occurs when a user process attempts a call open() as shown in Figure-5.

1. When the user makes an open() system call, it is overridden by the wrapper function
2. If the process is in secure mode, the wrapper function passes the arguments of the open() call to SecD via Unix Domain Socket
3. On receiving the arguments, SecD opens the file requested by the user and returns the file descriptor to LibC wrapper via Unix Domain Socket.
4. LibC wrapper returns the authorized file descriptor to the user process.
5. If the process is in normal mode, the wrapper function opens the file and returns the file descriptor to the user process.

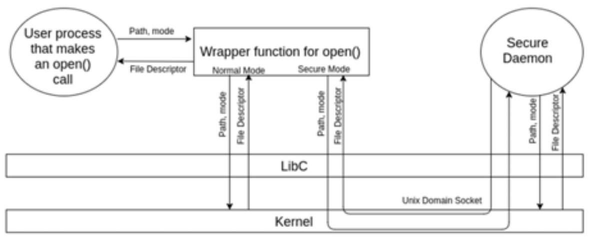
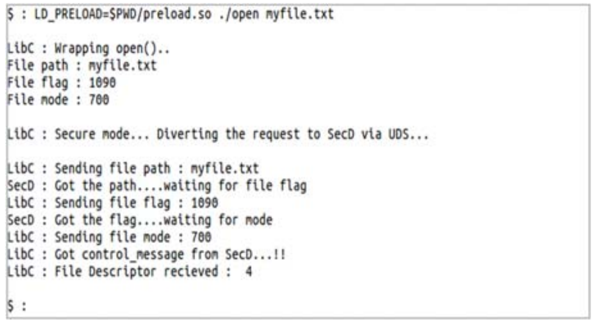


Figure-5. Dynamic dispatching mechanism of open() call.

ANALYSIS AND RESULTS

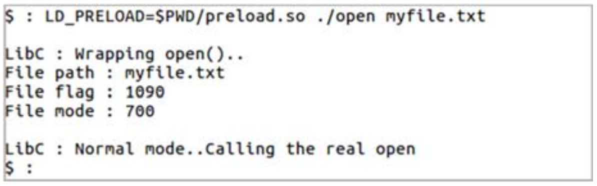
Figure-6 shows the actual screen shot of the communication between the modified LibC and Secure

Daemon in Secure mode and Figure-7 shows the screen shot of the execution of the real open() in the normal mode. No connection to secure daemon is established in the normal mode.



```
$ : LD_PRELOAD=$PWD/preload.so ./open myfile.txt

LibC : Wrapping open()..
File path : myfile.txt
File flag : 1090
File mode : 700

LibC : Secure mode... Diverting the request to SecD via UDS...

LibC : Sending file path : myfile.txt
SecD : Got the path....waiting for file flag
LibC : Sending file flag : 1090
SecD : Got the flag....waiting for mode
LibC : Sending file mode : 700
LibC : Got control_message from SecD...!!
LibC : File Descriptor recieved :  4

$ :
```

Figure-6. Screen shot of the secure mode operation in the proposed system.



```
$ : LD_PRELOAD=$PWD/preload.so ./open myfile.txt

LibC : Wrapping open()..
File path : myfile.txt
File flag : 1090
File mode : 700

LibC : Normal mode..Calling the real open
$ :
```

Figure-7. Screen shot of the normal mode operation in the proposed system.

To evaluate the performance of our system we use the metric; time latency. Time latency comprises of three metrics; the real metric, the user metric, and the sys metric. Real metric is the wall clock time measured from the beginning to end of the call. User metric is the CPU time spent for executing the program in user mode. Sys metric is the CPU time spent for executing the system calls within the kernel space. Analysis of the user metric and the sys metric indicates the CPU time the process has used.

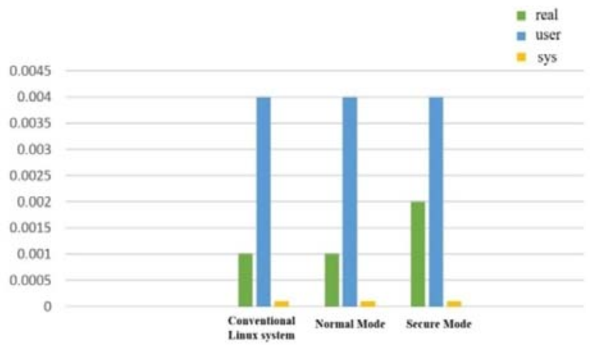


Figure-8. Comparison of the time latency of the conventional Linux system with the proposed system.

From Figure-8 it is clear that the wall clock time utilization of our system in the secure mode exceeds that of the conventional Linux system and the normal mode in our system by 1 millisecond. This additional time penalty incurred by the proposed system can be neglected keeping in view of the additional security it offers. Wall clock time remains the same across the normal mode of the proposed system and the conventional Linux system. The user metric and the sys metric remains the same across all the three systems.

CONCLUSIONS

In this paper, we have come with a top-down modular design to implement the security architecture proposed by us in our previous paper [1]. The tasks that have to be done by each of the modules are also clearly defined. The implementation of some of the modules has begun. We have started to create a new SecLib library by adding a new API to it to authorize the open() system call. We have modified the glibC by writing wrapper functions for the APIs open and fopen. In the secure mode, these APIs will be routed to the Secd via Unix domain socket by the wrapper function. Secd is the daemon that actually does the authorization for open and fopen and returns the file descriptor back to the wrapper function. We performed an analysis of the performance in terms of time latency and found that there is a very insignificant delay by employing the additional authorization.

What we have achieved is an additional level of authorization with least overhead in the processing time. Our future work is to complete all the modules and replace the current authorization with a more fine-grained one using the security tickets. Also, we will be writing wrapper functions and authorization modules for all the major critical system calls commonly used in Linux.

RELATED WORKS

Security of both data and host machine has been a paramount concern for a long time. Consequently, Linux Security model framework was introduced, which is designed to implement mandatory access control (MAC) by imposing minimal changes to Linux Kernel.

In our previous paper [1], we proposed architecture to secure the Linux operating system. It uses a Security Ticket to provide a fine-grained authorization for the user process ensuring that the user process gets only the least privilege to execute the intended task. System calls from user processes are authorized by Secd based on Security Tickets.

CliffeSchreuder *et al* [2] depicted a generalized view on some Linux Security models using mandatory access control, which include SELinux, AppArmor, TOMOYO, and FBAC-LSM.

SELinux differentiates information based on confidentiality and integrity requirements. Michael Wikberg [3] presented a full study on the policies used to secure the operating system using mandatory access control (MAC) mechanism. The MAC architecture used in SELinux is FLASK. AppArmor uses the concept of creating profiles for each application which has to be secured. Security context for each application is defined by this profile.

Toshiharu HARADA *et al* [4] presented a Security-enhanced Linux named TOMOYO Linux which uses Mandatory Access Control [5] with an automatic policy generator. TOMOYO discards unnecessary privileges for the existing programs. Mandatory Access Control mechanism has been widely adopted since it increases the security of an operating system.

For securing Linux, other mechanisms like system call interposition and capability based authorizations are used. Janus, a sandboxing system, based on system call interposition was developed by Ian Goldberg *et al* [6]. Janus is visualized as a firewall in between application and the Operating system, which filters the system call. The initial version of Janus was noteworthy since it does not require any kernel modification. An extended version of Janus was introduced in which system call interposition is carried out through the kernel module.

Robert N. M. Watson *et al* [7] presented Capsicum, a lightweight operating system which extends UNIX API's. Capsicum imparts additional capabilities to existing Linux. It was introduced to include in FreeBSD 9. Capsicum allows applications to self-compartmentalize by which monolithic applications are decomposed to run in independent sandboxes to form logical applications. Sandboxing architecture was proposed by Muhammad Shams Ul haq *et al* [8] which uses reference monitor as a shared library to load the applications which have to be executed.

NielsProvos [9] presented a hybrid approach which attains a fine-grained process confinement utilizing Systrace facility. The presented approach supports interactive policy generation and intrusion detection which applies to both the system services and user applications.

Decomposition of Kernel is a powerful way for reducing the consequence of individual attacks. Charles Jacobsen [10] provided the concept of Lightweight Capability Domains, which aids effective decomposition of an operating system kernel.

Adwitiya Mukhopadhyay *et a*l [11] presented an implementation of a firewall based on Linux operating system, which utilizes the Netfilter framework, and the IP tables that communicates the firewall policies with the kernel. Several researchers put forward a variety of Kernel space as well User space [12] security implementations based on sandboxing [13-15], system call interposition, capability systems [16], and Mandatory access control mechanism. All of them are good in one aspect but deficits in another.

REFERENCES

[1] Hari Narayanan, Vivek Radhakrishnan, Shiju Sathyadevan, and Jayaraj Poroor. Architectural design for a secure linux operating system, accepted at International Conference on Wireless Communications Signal Processing and Networking (WISPNET), Chennai, 2017.

[2] Z Cliffe Schreuders, Tanya McGill, and Christian Payne. Empowering end users to confine their own applications: The results of a usability study comparing selinux, apparmor, and fbac-lsm. ACM Transactions on Information and System Security (TISSEC), 14(2): 19, 2011.

[3] Michael Wikberg. Secure computing: Selinux. http://www.tml.tkk./Publications/C/25/papers/Wikberg nal. pdf, 2007.

[4] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Task oriented management obviates your onus on linux. In Linux Conference, volume 3, 2004.

[5] NSA Peter Loscocco. Integrating exible support for security policies into the linux operating system. In Proceedings of the FREENIX Track. USENIX Annual Technical Conference, page 29. The Association, 2001.

[6] Tal Garfinkel *et al*. Traps and pitfalls: Practical problems in system call interposition based security tools. In NDSS, volume 3, pages 163-176, 2003.

[7] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In USENIX Security Symposium, volume 46, page 2, 2010.

[8] Muhammad Shams Ul Haq, Lejian Liao, and Ma Lerong. Design and implementation of sandbox technique for isolated applications. In Information Technology, Networking, Electronic and Automation Control Conference, IEEE, pages 557-561. IEEE, 2016.

[9] Niels Provos. Improving host security with system calls policies. In Usenix Security, volume 3, page 19, 2003.

[10] Charles Jacobsen, Muktesh Khole, Sarah Spall, Scotty Bauer, and Anton Burtsev. Lightweight capability domains: towards decomposing the linux kernel. ACM SIGOPS Operating Systems Review, 49(2):44-50, 2016.

[11] Adwitiya Mukhopadhyay, V Srinidhi Skanda, and CJ Vignesh. An analytical study on the versatility of a linux based rewall from a security perspective. International Journal of Applied Engineering Research, 10(10): 26777-26788, 2015.

[12] AP Murray and Duncan A Grove. Pulse: A pluggable user-space linux security environment. In Proceedings of the sixth Australasian conference on Information security-Volume 81, pages 19-25. Australian Computer Society, Inc., 2008.

[13] S. P and K. P. Jevitha. Static analysis of firefoxos privileged applications to detect permission policy violations. International Journal of Control Theory and Applications, 9(7):3085-3093, 2016.

[14] Jan Hurtuk, Anton Balaz, and Norbert Adam. Security sandbox based on rbac model. In Applied Computational Intelligence and Informatics (SACI), 2016 IEEE 11th International Symposium on, pages 75-80. IEEE, 2016.

[15] Misha Mehra and Dhawal Pandey. Event triggered malware: A new challenge to sandboxing. In India Conference (INDICON), 2015 Annual IEEE, pages 1-6. IEEE, 2015.

[16] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. Proceedings of the IEEE, 63(9):1278-1308, 1975.