Architectural design for a secure Linux operating system

Conference Paper · March 2017

DOI: 10.1109/WiSPNET.2017.8299902

4 authors, including:

Hari Narayanan
Amrita Vishwa Vidyapeetham
5 PUBLICATIONS **20** CITATIONS
SEE PROFILE

Jayaraj Poroor
Amrita Vishwa Vidyapeetham
19 PUBLICATIONS **42** CITATIONS
SEE PROFILE

Architectural design for a secure linux operating system

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

Jayaraj Poroor
EnergiMate
Gandhi Nagar, Gujarat, India

***Abstract*— Operating system security is a hot research area for the past several decades. Various security mechanisms have been introduced till now to secure the operating system. In this paper we are focusing on securing Linux operating system. Even though Linux is open source and large numbers of people are involved in developing kernel patches for security holes, there are still many malwares to exploit the existing vulnerabilities. Using our architecture we are trying to minimize the damage done by the malwares if not blocking them altogether. Our architecture is designed to ensure the principle of least privilege. Principle of least privilege guarantees that a process will get the privileges just enough to perform its task. This ensures that even if the process is compromised it can do the least damage to the system as it is running in a sandbox. Major chunk of our system is running in the user level to make it portable across the distributions. Our system uses a specially structured security ticket to provide fine grained authorization to user processes which is not currently possible in the traditional linux architecture. The security ticket is designed in such a way that it can be inherited by a child process, can be shared and is unforgeable. The core module in the system is called Secd (Secure Daemon) which authorizes all the requests and also manages the security tickets.**

Keywords— Linux; authorization; principle of least privilege; security ticket; sandbox

I. INTRODUCTION

Authorization is an essential part of any access control mechanism. Authorization is the process of specifying the access rights. There are two important stages in access control. (1) Defining the access policy and (2) Enforcing the policy. The process of defining the access policy is authorization. Here a policy is defined which says who all can access what all objects or resources. In the second step a decision is taken when an actual access request comes, either to allow or deny the access. Also the users are supposed to be authenticated i.e., their identity is verified. If authentication is failed then the subject is not allowed to access any of the system resources. If authentication is successful then the subject is provided with controlled access.

Traditionally Linux operating system uses identity based authorization. Authorization is based on the userid and the groupid of the user. More specifically the effective userid and the effective group id are used for the authorization. When the user executes a program or command, which will result in a fork and exec combination. If the setuid bit of the executable is set then the effective userid of the process becomes the owner of the executable. So is the effective groupid. If the setuid bit is not set then the effective userid of the process is same as that of the real userid. Similar is the case of the effective groupid. The problem with the identity based authorization is that all the processes executing on the behalf of the user gets the ambient privilege i.e. all the privileges of the user. To understand the problem more clearly let us consider the case in which the user downloaded a malware affected game program. Outwardly it looks very harmless but the remote attacker can control the program through a back door setup in the program. Now the attacker gets the ambient privilege and can do unlimited damages to the system.

Currently there are well known solutions to tackle this type of problems like SELinux, Capsicum, Apparmor etc. The one thing common to all the existing solution is that it involves instrumenting the Linux kernel. So for every kernel upgradations the implementation has to be modified. But in our case the major chunk of the implementation is in the user space. So as long as the system call interfaces remains unchanged we need not modify the implementation. Also our solution is platform independent and can be ported to all the Linux distributions. What we are trying to do is to set up an authorization mechanism on top of the identity based authorization so that the principle of least privilege is established. The program will not get the ambient privilege rather it would be given just enough privileges needed to perform its task. So if we take the example of the game program it cannot access any other system resources like files, network etc. We cripple the ability of the attacker to do extensive damage to the system. In a way we are running the process in a restricted environment or in a sandbox.

The remainder of this paper is organized as follows. Section II presents the current and proposed architecture. Section III describes the user level process secure daemon (Secd). Section IV describes the modified fork. Section V describes unforgeability of security tickets. Section VI presents our conclusion. Section VII describes the related works.

II. SYSTEM ARCHITECTURE

The authorization mechanism in the traditional Linux is as shown in the figure 1. If the user process wants to access a

system resource, it would invoke a POSIX API which is defined in the LibC. LibC will convert that into a system call. The system call dispatcher at the interface invokes the correct

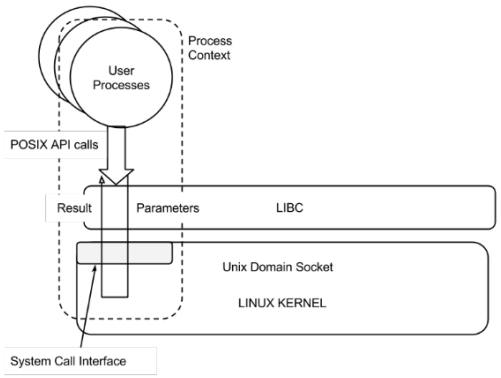

Fig. 1. System interface architecture of unmodified Linux.

system call handler which executes the original system call. The system call handler checks the effective userid and effective groupid for authorization. If the process is authorized to access the resource, the resource handle is returned to the LibC API and back to the user function which invoked the LibC API.

The proposed architecture is shown in the figure 2. There are two modes in which a process can execute: (1) Normal mode and (2) Secure mode. In the normal mode, everything is similar to traditional Linux as shown in the figure 1. There is a secure daemon (Secd) which automatically runs in the background when the system is booted. In the secure mode all the LibC APIs are diverted to the Secd via unix domain socket, instead of being converted into a system call. The Secd will verify the security credentials of the initiated process. If the process has the necessary privileges then the Secd will convert the API in to a system call. That will result in a context switch to the kernel mode from the user mode. The requested resource would be allocated and the resource handle will be returned to the Secd. The resource handle is given back to the LibC via unix domain socket. LibC will finally return it to the calling function in the process.

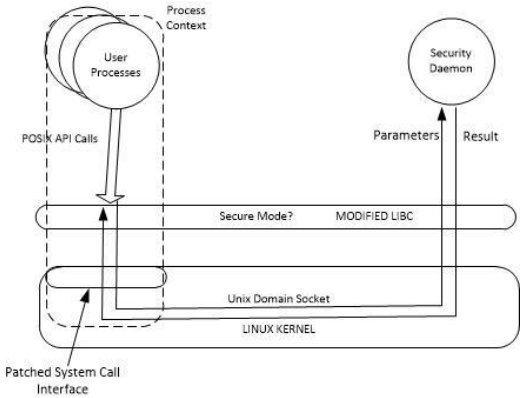

Fig. 2. System interface architecture of proposed system.

Once the process switches from the normal mode to the secure mode there is no going back to normal mode. Also all the children spawned by the process would run in the secure mode.

III. SECURE DAEMON (SECD)

The Secd is a user level process which always runs in the background when the system is booted, automatically. Secd is responsible for creating, deleting and refining security ticket as well as authorization of resource access requests. The resource access requests are done by invoking system calls. In order to protect the Secd from any attack, it is run using a unique userid and groupid.

The two main tasks Secd does are (1) Issuing the security ticket and (2) Authorization of system calls as shown in the figure 3.

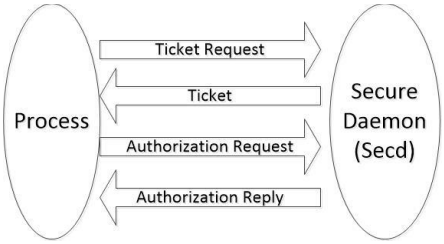

Fig. 3. Communication between a process and Secd.

All the processes should request for a security ticket to run in the secure mode. Secd will issue the ticket to the process. This ticket should be send as a parameter with any system calls. Authorization is done by the Secd on verifying this ticket which is shareable and unforgeable.

A process wishing to enter a security sandbox will first acquire security tickets from the Secd via Secd APIs implemented by the modified LibC. After this the process enters the secure mode by invoking the new system call as shown in figure 4.

Now onwards, all POSIX API calls are automatically routed to the Secd for authorization and resource allocation. The Secd will check if the originating process has the necessary privileges to acquire the resource. If so, Secd will

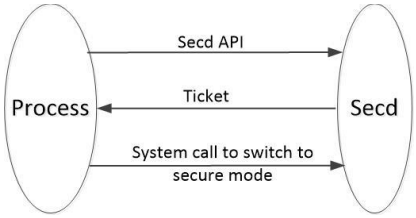

Fig. 4. Entering in to a security sandbox.

create the appropriate resource handle (e.g. by opening the target file) and supplies the handle back via unix domain socket to the modified LibC executing in the context of the originating process. The modified LibC then supplies the handle as the result of the API call to the calling function. This process is shown in the figure 5.

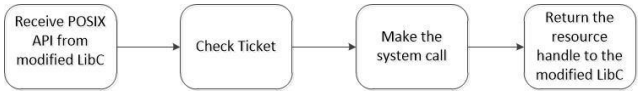

Fig. 5. Actions taken by Secd when it receives a POSIX API

The API calls are converted to remote method invocation by modified LibC and the API parameters are marshalled in to

messages and send via unix domain socket to the Secd as shown in the figure 6.

The Secd authorizes the resource access requests and if access is permitted invokes the system call. The results and the resource handle are marshalled and send back to the modified LibC as messages. Modified LibC will unmarshall the results and resource handles and return to the first function which invoked the LibC API. Any attempt by a process which is running in the secure mode to bypass the LibC to make a system cal will be blocked at the patched system call interface as shown in the figure 6.

Unix Domain Socket

The above security mechanism relies on the following features of the Unix Domain Sockets:

- Ability to send the file handles between processes so that Secd can create a file handle after authorization and pass to the calling process.
- Ability to pass process credentials (process id, uid, gid) which are validated by the kernel from the calling process to Secd.
- Ability to restrict creation/bind permission for unix domain sockets via file system permissions. The creating process must have ”wx” permission to the directory of the socket’s path. This can be restricted to only the Secd’s uid/gid. Connecting processes require ”rw” permission to the directory.

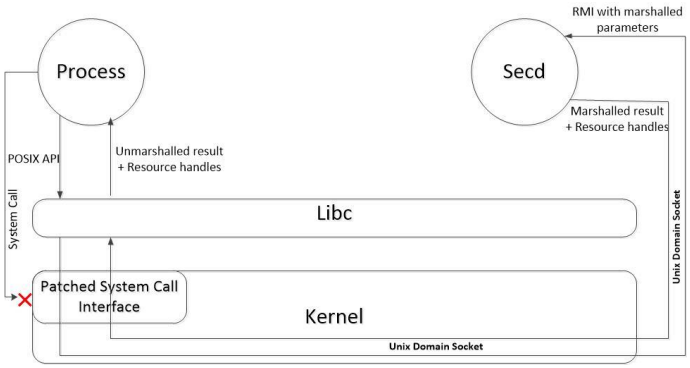

Fig. 6. Conversion of POSIX API to RMI by LibC

IV. FORK

All the POSIX APIs follow the same sequence if the process is running in the secure mode, the call is diverted by the LibC to the Secd transparently via unix domain socket. But fork API is different as it creates a new child process. If the parent is running in the secure mode, the child would also run in the secure mode.

The secure daemon has to supply an access token to the process which has initiated the fork API. When the process makes a fork call the following sequence takes place which is also shown in the figure 7:

1. LibC will send a request for access token to secure daemon.
2. Secure daemon will issue an access token back to LibC which is running in the context of the process.
3. LibC will issue a modified fork call together with the access token to the kernel.

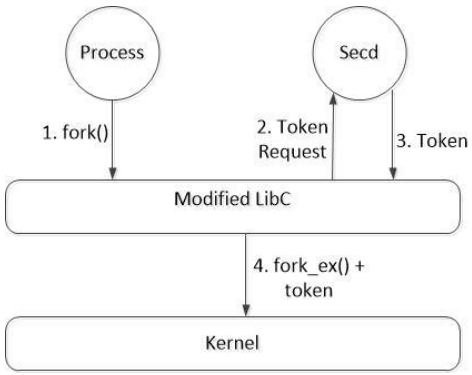

Fig. 7. Issuing fork API by a process in the secure mode

V. UNFORGEABILITY

In order to ensure unforgeability of the security tickets issued by the Secd, security tickets are essentially indices to a per process security array (similar to the way open file handlers are maintained by the OS Kernel) maintained by the Secd as shown in the figure 8. When a process supplies a security ticket, it is used as an index (with bound checking) into this array. The array entries contain pointers to the global security objects maintained by the Secd.

Security data structures are maintained by the Secd process. Security objects are immutable objects. Two processes can share a security object. Modification of a security object will result in the creation of a new object. A security object contains a bunch of <name, value>pairs or <attribute, value>pairs.

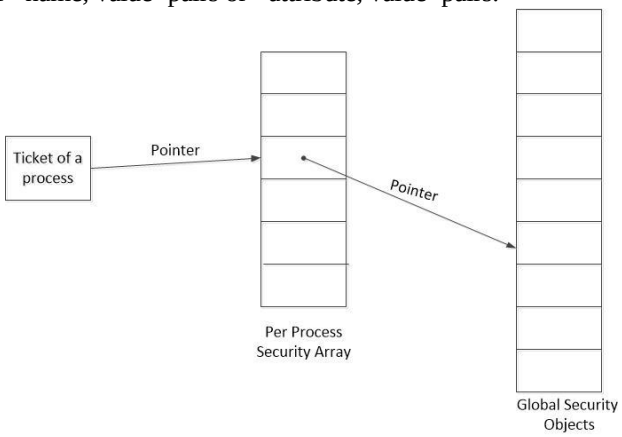

Fig. 8. Data structures to store security ticket information

In the case of a file system the attribute can be a file name or a pattern like temp/*. The value can be permissions like read/write/execute. In the case of a network the attribute can be ip-address or port. The value can be operations like bind/connect. In general we can say that security ticket is a bunch of <resource spec, operation spec>pairs. Both resource spec and the operation spec are <attribute, value>pairs.

VI. CONCLUSION

We have designed a new architecture for securing the Linux operating system. Identity based authorization of traditional Linux systems is not fine grained. We have set up another layer

of authorization on top of the identity based authorization based on specially structured security tickets which are unforgeable and sharable. There is a user level process called secure daemon (Secd) that will mediate all access to system resources and authorizes the request. Secd also creates and deletes the security tickets. The LibC is modified to transparently divert the API calls to the secure daemon via unix domain socket. Fork system call is an exception as it creates a child process. It is handled in a different way. If the parent is in the secure mode then the child is also in the secure mode. The child process inherits the security ticket of the parent either in its entirety or in a refined fashion. Security tickets can be inherited as well as shared. Security tickets issued to the process are indices in to an array which contains the pointers to the actual security objects. This is to make sure that the security tickets are not forged.

In addition to providing fine grained authorization the proposed system is platform independent as Secd is running in the user space. Even if we download a malware affected program from the internet it can cause only minimum damage as the process is running in a sandbox as long as the system call interface remains the same. One problem that can happen is that every program should have a resource description file that specifies the resource requirements in order for it to run in the system. Even if the kernel version is upgraded our system can run with the least amount of changes. We are in the process of implementing this architecture in a Linux machine and evaluate the performance. Our future research would be to handle concurrent requests from the same process by using multiple threads.

VII. RELATED WORKS

In Discretionary Access Control (DAC) [1], a process running as a user (UID or SUID) has the user's permissions to objects. So the Unix way of authorization can't be considered as a convenient methodology for many applications. System wide policies have to be enforced in order to restrict malicious code as well as careless users. One way of minimizing the consequence of malicious code being executed is running applications in the sandbox [1] [2] [3]. Sandboxing has been immensely accepted in various applications for providing security from malicious content. But in many of the context, sandbox does not provide an infallible bullet against malware detection [3]. Some prototype has been implemented as an executable shared library for dune [4] which provides isolation to native Linux process.

SELinux [2] is a security-enhanced Linux, which implements Mandatory Access Control mechanism which differentiates information based on confidentiality and integrity requirements. The MAC architecture used in SELinux is FLASK [5]. Each subject, as well as the object in SELinux, is associated with a security context. Security context depends on security policies, which usually consists of user id, role and type enforcement domain.

Apparmor [6] is a Linux Security module which uses Mandatory Access Control. Profiles can be created for each application that has to be protected. This profile defines the security context for that application. Each profile may consist of the path to the program that is confined; also a path entry defining the part of the file system the program can access.

As a solution for dynamic policy modification based on user preference, a system was put forward known as Pluggable User space Linux Security Environment (PULSE) [7], which implements MAC enforced dynamic, user-level POLA [7] implementation. Since security policies are stipulated using user-space plugins, PULSE provides a high degree of flexibility and usability which is not common in existing security architectures.

Capsicum [8] is an existing security module, which extends existing UNIX APIs, rather than creating new APIs. Capsicum adds new Capabilities [9] to UNIX. Capsicum implements compartmentalization [10] there by running applications in independent sandboxes. Compartmentalization ensures the principle of least privilege [11] by granting each compartment only the rights it requires.

Janus [12] can be considered as a firewall sitting in between an application and the operating system. Janus contains a kernel module and a user-level program. Secure system call interposition [12] is carried out through the kernel module, while the user-level program decides which system calls are to be allowed or denied.

TOMOYO [13] Linux is a security-enhanced Linux, which uses Mandatory Access control, with automatic policy generation technique. It doesn't require any labeling and has CUI [13] based policy editor. Tomoyo provides methods for the existing programs to discard unnecessary privileges voluntarily.

New systems have been implemented and under research which uses hybrid [14] approach that enables fail-safe operation, which maintains less performance overhead. Systrace [14] facility is efficiently used to give fine grained process confinement, which gives application elevated privileges as determined by policy generator, rather than having root privileges throughout the lifetime of an application.

Many organizations are very keen about their security, but present deployments are too costly and not that flexible. There are some implementations of Firewall built on Open Source Linux Operating System. These implementations take advantages of the Netfilter [15] framework and make use of IP tables which communicates the firewall policies to the kernel.

Newer trends arising portrays the development of web browsers as Mobile operating systems. Common platforms such as Mozilla Firefox and Google Chrome has already initiated this trend. Firefox OS [16] uses Linux Kernel which provides various security measures to overcome various policy violations. The security measures include running the program in Sandboxed environments; creating content security policy and permission management system [16].

Modern kernels require decomposition as an effective means of restricting the consequences of individual attacks. Lightweight capability domains [17] are present which develops principles, mechanisms, and tools based on Linux kernel which facilitates effective decomposition of an operating system kernel.

REFERENCES

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

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

[3] 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.

[4] 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.

[5] NSA Peter Loscocco. Integrating flexible 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] 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.

[7] 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.

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

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

[10] Khilan Gudka, Robert NM Watson, Steven Hand, Ben Laurie, and Anil Madhavapeddy. Exploring compartmentalisation hypotheses with soaap. In Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2012 IEEE Sixth International Conference on, pages 23{30. IEEE, 2012.

[11] Khilan Gudka, Robert NM Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G Neumann, and Alex Richardson. Clean application compartmentalization with soaap. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 1016-1031. ACM, 2015.

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

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

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

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

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

[17] 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.