Security Policies for Linux Operating System

Adrian Airinei, Dorin Smaranda
Faculty of Automatic Control and Computer Science
University "Politehnica" of Bucharest
airinei.adrian@gmail.com, smarandadorin@gmail.com

***Abstract*—This paper presents an approach to the pressing matter of security. For a multi-user machine to be secure a lot of precaution measures are taken. Presuming that the integrity of the physical machine cannot be compromised, only the OS and programs running on the secured machine can be attacked. Therefore any kind of attempt to gain unauthorized information or access will be software. To keep more than a few users in line security policies must be set. But for every user a different set of rules will be applied as each individual can have a certain degree of clearance in the system. Our approach simply takes the protection to a lower level. Using the Linux OS as base, the protection intended is as close as possible to the core of the OS. Therefore the restrictions will be enforced at the kernel level while the root user uses a friendly text interface to create complex custom policies by combining simple rules.**

***Index Terms*—kernel, policy, security, syscall**

I. Introduction

Through the years, the importance of security has anything but diminished [8][9]. The systems and applications became more complex so the security measures needed to be enforced so that the privacy needed could be satisfied.

Typical approaches used in improving computer security can be described as following : physically limit access to computers to only those who will not compromise security, hardware mechanisms that impose rules on computer programs, thus avoiding depending on computer programs for computer security, operating system mechanisms that impose rules on programs to avoid trusting computer programs and programming strategies to make computer programs dependable and resist subversion.

For most multi-user systems, we can easily provide the physical access limitation, but hardware mechanism or operating system mechanisms can be costly and even hard to implement. Current strategies for protecting the integrity of any system is based on programs that try to block unauthorized access. For such a strategy to work, the firewalls, passwords, etc. must work closely with the operating system, because otherwise a skilled user could just find a workaround at a lower level of functionality in the system.

In that matter it is clear that the lower you get into a system, you can better protect it. Going by that idea it is this simple approach that can bring order to a multi-user system, by allowing easy management of their available resources. This means files, applications, disk space, etc, everything that a user has rights on that machine. Even if they are many in count they all have something in common when making use of them : the utilization of the operating system. For example any read, write, even execution of a file must pass through the OS, because that is how it was built.

For the present paper we have chosen the Linux operating system. As following, a kernel module that would monitor the system calls could provide the user management needed through security policies enforced by authorized users.

The remainder of this paper is organized as follows. Chapter II takes a snapshot on the current general situation of the work already done connected to this paper. In chapter III it is presented a short architecture description, followed by a closer look in the implementation detail in chapter IV. The following chapter presents the evaluation on the actual project and chapter VI draws the conclusions and enumerate ideas for future work.

II. Related work

The idea of implementing a way to monitor any action (syscalls, resource access, etc.) that could occur in a Linux-kernel in order to prevent any security issue is not new. Several approaches have been made [1][3][4][6], every one in its characteristic way. Despite this, few of them have implemented an approach which helps the system administrator to define security policies in an easy way. In the next paragraphs the most important previous approaches will be outlined.

A. Linux-PAM

Linux-PAM [1] is a free implementation of Pluggabale Authentication Modules (PAM) [2] for Linux. Basically it is a flexible mechanism for authenticating users. PAM provides a way to develop programs that are independent of authentication scheme. The authentication modules are installed in these programs at run-time in order to work. Which module is installed is dependent upon the local system and is configurable by the system administrator.

PAM actually does not apply general security policies but is more concerned on the policy of authenticating the users. The system administrator is free to choose how individual service-providing applications will authenticate users.

B. Syscalltrack

The syscall tracker [10] allow the system administrator to track invocations of system calls across Linux kernel. Rules may be defined to specify which system call invocations to

be tracked, and what to do when a rule matches a system call invocation.

Syscalltrack interests is on determining what process or what program is doing a certain action such as: deletes a file, sends a certain signal to another process, updates a file content, changes permission on a file/directory, etc. As it can be seen it cannot be used to limit resource access regarding a certain user: limit the memory usage, limit the number of open files or limit the CPU usage.

C. StMichael LKM

StMichael is a LKM (Loadable Kernel Module) [11] that attempts to provide a level of protection against kernel-module root kits. It monitors various portions of the kernel for modifications that may indicate the presence of a malicious kernel module. If a malicious activity is detected, StMichael will attempt to recover the kernel's integrity by rolling back the changes made to a previously known-good state. The malicious actions may be detected through different ways: an MD5 check sum of the base kernel or at kernel loading which requires the StMichael to be loaded early in the boot process using initrd.

Furthermore it can protect immutable files from having their immutable attribute removed and disable write-access to kernel memory through the */dev/kmem* device. Anyway, StMichael does not go on user permission and authorization.

D. REMUS (Reference Monitor for Unix Systems)

Remus [3] is a loadable kernel module that can be added to the standard Linux kernel to monitor and possibly block "critical" system call invocations. The rules for a system call to be executed are kept in an Access Control Database (ACD) within the kernel. These rules specify whether the syscall invoking process and the value of the arguments comply with the policy applied.

One major attack that REMUS is able to block is buffer overflow attack, by stopping it before it completes. Nonetheless, the project intends to protect against any technique that allows an attacker to hijack the control of a privileged process.

As it can be seen, in general REMUS tries to secure the OS by means of lightweight extensions of the kernel. Still, the principle that guides it is the same: trying to monitor every system call made by every process.

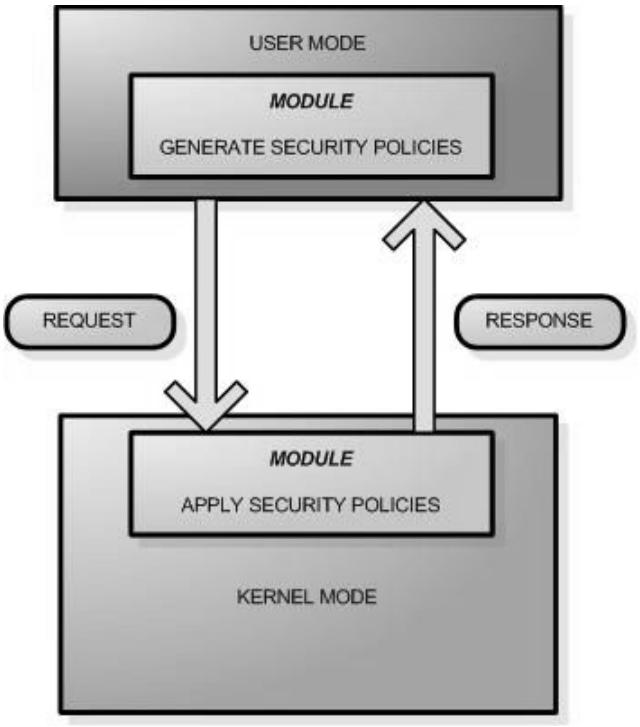
All the projects mentioned above have followed more or less the ideas that guided the SePoL project and all of them influenced somehow the implementation of it. Linux-PAM uses the same way of defining rules, but it focuses more on authentication part; syscalltrack is administration oriented, but not necessarily focused on users; StMichael and REMUS are more kernel-protection oriented.

What the project brings new is the idea of designing a way to enable the setting of multiple and not-trivial security policies to be applied. Therefore enhancing the kernel with management capabilities, easily used by the administrator without having to know all the founding principles.

III. Architecture

The architecture of the application is comprised of three main elements : the kernel-side module, the used side module and the communication module. The kernel-side module is usually referred to as the kernel module as in common knowledge, the applications that run at the kernel level are in fact loadable modules. The user side module is a command-line application that runs in the user space, not a module, but will seldom be referred to as the user module, since it is independent of the kernel module. The communication part is in fact not a module. It is a just a design concept used to describe in simpler words the way the whole project works.

In short, we have the kernel module and the user module, which communicate through a series of functions that by design could form the communication module.



The kernel module is, as stated before, a loadable module. That means that it can be started after the operating system has been completely loaded and functioning. Of course it can be loaded in the OS loading procedure but that aspect is not one of the points of interest to this paper. In this paper it is considered that the authorized user is in fact the root user (or administrator), that has full access on all the system's resources. Therefore, the administrator can manually load or unload the module as wanted. When loaded the admin (administrator) cannot interact directly with the module since there are no such methods implemented. To do so, the admin must start the user module.

The user module represents the user interface. It is a command-line like application itself and recognizes a series of commands. Through it the admin can load, unload the kernel module without needing the proper commands to manually accomplish that task.

After the kernel module is loaded and the user module loaded, the administrator can begin the input of the rules to be applied, or can load them from an older session, saved in the appropriate file and format. When the administrator is satisfied with the rules he/she designed, then it is time to put them in practice. This is a simple command, which communicate the kernel module all the rules currently available to the user module. For future use, the administrator can save the current rules to the configuration file.

When the restrictions are no longer needed, the kernel module can be stopped, directly from the user module.

IV. Implementation details

As described in the previous chapter, the project is composed of two main parts: the user-module and the kernel-module. Each of them has its own independent implementation except the communication part where there must be an agreement on how the information is sent from one side to the other. The following sections present the way the modules were implemented.

A. User-module

This module is the user interaction interface. It allows the user to start and stop the kernel module, input and apply the desired rules. It is written as a .lex file, and must be compiled using Flex compiler. After obtaining the C file, this file must be further compiled as a normal C program.

This part of the application uses the Flex rules to recognize the text input by the user. Then decides whether it was correct and after that it saves the information in temporary variables until the user is satisfied with the parameters, and the rule is then saved in memory. To reuse the saved rules, the administrator can save them in the config file.

There are many variables that can be set by the user. This is accomplished through the console-like interface that the application provides. After carefully selecting the desired parameters, the user can save the rules. These rules are stored into structures that will be kept until the application ends or the user decides to apply them or load others from file.

Some parameters are mutually exclusive. For example: the user and group setting, and if one is set, then the input of the other will reset the first. All the other settings can coexist. The time setting is optional, it can be set if desired, and after the expiration of the time set, the rule will be removed. If the time is not set, the rule will remain until the administrator stops the program. The saved configuration will be stored in a predefined file named “config.txt”. It can be found in the same directory with the user-module application. Well thought policies [7] can save a lot of time in a crowded system.

The communication between the user component and the kernel component is achieved by using the OS’s signaling procedures. The user application uses a specific signal and the function is called using the parameters saved as the configuration rules. And so, little by little all the rules are transmitted to the kernel module. Since the kernel module already monitors the system calls, it is an easy task to validate the transmission and extract the information.

B. Kernel-module

The kernel-module is the part of the project which actually applies the security policies. It receives them from the user-mode-module through the SIGUSR2 system call as parameters. The policies are then stored in local kernel memory which actually limits the number of the policies that may be applied to a certain number. However, memory storage involves a higher speed when searching for data and thus the impact on kernel speed will not be so high. The evaluation chapter will analyze this impact more deeply.

In order to apply the policies defined in user-mode the system calls are intercepted. Here a issue may appear if the kernel on which the module is inserted does not have the sys_call_table symbol exported. In that case the kernel-module is not able to install any hook for syscalls. Usually the sys_call_table symbol is not exported starting kernel version 2.6 due to security issues. Anyway, the kernel may be modified and then recompiled in order to work with SePoL.

The security policies implemented by SePoL and how where specifically implemented are:

- **write access on a file or directory** - the *write* syscall is intercepted;
- **read access on a file or directory** - the *read* syscall is intercepted;
- **execute access on a file or directory** - the *execve* syscall is intercepted;
- **disk space usage** - every time a *write* syscall is done, it is checked whether the current process is allowed to write more blocks on disk. The space disk used is counted since the insertion of the module;
- **memory usage** - whenever a malloc() call is made in user-mode, the *mmap* syscall is called. Consequently intercepting it allows to decide whether the process gets or not the needed amount of memory;
- **number of open files** - in *open* syscall hook it is tested using *current* structure how many files has the current process opened. This feature may also be used a a security measure against a possible attack that tries to make a denial of service attack by opening many files on the machine;
- **number of created processes** - *fork* and *clone* syscall are intercepted;
- **time constraint** - the policy to be applied may have time constraints: what time should the module start monitor it, and for how long. Also periods may be scheduled.

The kernel-module implementation is quite modularized and it is composed from different sub-modules, each of them supplying one type of policy. This structured implementation has a real advantage in extending the functionalities of the project. Every new feature should only add a new sub-module without making any major changes to the existing code.

V. Evaluation

The project tried to bring a new and valuable feature for a Linux administrator who wants to have more and easier control

over all the processes and users that are using a system. Instead of applying his security policy [5] by using many different means which may actually not satisfy all his requests, the system administrator will only have to use a single service and customize his needs in a very simple and efficient way.

For example if he wants to deny user *user1* to write in file *foo.txt* he cannot do this explicitly. With **chmod** you can only set permission for the owner, for the users in the same groups with the owner and for others. If you try to move the user in the group with the owner and give them all the permission to write the file, the rule is not quite good, because also all the other persons in the group will have access to the file and maybe we don't want that.

Furthermore there are not trivial tools to set the maximum disk space for a certain user. You may define some quotas to restrict disk space usage, but the administrator should have some knowledge regarding this. Using our kernel module is much easier to make this restriction and only basic knowledge is required for the administrator.

Another nice feature was the possibility to set the availability of the policy for a certain period, which was not so hard to implement and brought a helpful facility. Actually this feature enables you to schedule when some policies should apply and which conditions must be met.

When the project was designed it was thought how many benefits will the project bring to an ordinary Linux administrator. However when it got to the testing and evaluation part it was tried to see whether the project is viable for slightly different purposes. For example if the administrator has more than 20 users on the same machine and must set 100 rules at least in order to keep the security at a medium level. In this case some problems may appear. Because the information from the configuration file is quite large the user-module cannot send all the security policy at once through the syscall. The information is divided into equal items of 1024 bytes maximum each and sends them one by one to the kernel-module. On the other hand the kernel module stores all this information in memory and if the received information is too large (in extreme case), it may happen that not all of the security policies may be applied.

Furthermore it is known that intercepting a system call should not be too much time consuming. We tried to test the kernel-module response time when a large number of security policies are applied. It is obvious that the system should respond slower than usual because more tests are needed to see whether the current processes are allowed to take place.

In order to test the delay that a syscall gains after the SePoL is inserted into kernel and security policies are applied, it was wrote a simple program in user-mode that call the same syscall 100 times in the same conditions and counts the time flow. Then it was divided the time measured to 100 and obtained the average time for one syscall. Also there were set different number o policies regarding open access so that their impact could be measured. It was arbitrary chosen the open syscall to test because any of the other syscalls would have led to the same results. Indeed the syscall time depends from syscall to syscall, but the overhead brought by SePoL is the same for each of them. The test results are shown in the graphic diagram from Figure 1.

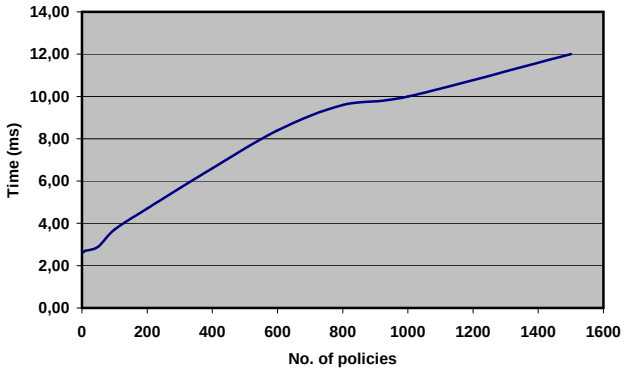


Fig.1. SePoL Overhead

From the results above it was concluded that the impact was not so big as expected. Actually it can be seen that the kernel responses are only slightly longer than usual and only if the number of security policies are really high (over 400). The syscall response time is not very affected. For 1000 rules, 12 milliseconds is 500% overhead, but considering that the syscalls are not very often called it is acceptable. Indeed, if a program opens many files in a very short period, the execution time will increase by 5 times, which is not a pleasant case. For these situations the administrator may remove SePoL from the system.

From the ordinary user point of view, the project offers a nice interface for adding and managing the security policies. Though there are some negative parts. The main disadvantage is that every time when the securities are modified (added, updated or deleted) the kernel-module must be re-inserted and all the policies retransmitted through the system call SIGUSR2. Then there is the 2.6 kernel incompatibility which has only the not-user friendly solution of modifying the kernel source and recompiling it. Furthermore there might be situations when the input of an inappropriate policy rule will may corrupt the operating system irreversibly. For example if the administrator sets a rule that denies access to the admin group for a very long time then the whole group will be unable to do anything in the system beginning with the moment of enforcement. Either all the users in that group must wait the set period before regaining access or the operating system must be reinstalled because no one else can remove the module (considering only the root is allowed to do this).

In any case, it is considered that the system administrator is responsible enough to avoid the case mentioned above and that he is aware about the risks the usage of SePoL involve. From the point of view of the system administrator it is not of primary importance to understand the internal behavior. The important point to understand is that the main rules that will remain active throughout all the online period, must be carefully thought out, and also the usefulness of the time setting.

VI. Conclusions & further work

In this paper it was presented the design and implementation of SePoL, a loadable kernel module that helps the system administrator to have more and easier control over what a user is authorized do or use. The evaluation of SePoL took into consideration two main criteria as follows.

First, what is the impact on the Linux-kernel performance regarding the response time of the syscalls. From the tests it resulted that the impact is quite small even if there are 100 security policies defined by the system administrator. Off course an overhead of about 1000 security policies will indeed bring a considerable time increase. The second is the project usage. Our overall conclusion is that SePoL is not yet developed enough for real usage, but the advantages and innovative approach it uses makes it a good candidate for further studies and improvements, both in the efficiency and the interface sections. Thus, we believe that using SePoL can, in practice and not just in theory, be used as a management tool for the Linux operating system.

As further work the project may be extended by adding more security policies, by adding a feature that notices the system administrator if the rules have been broken in some way. Then a feature that enables to define an action if a rule is broken could be added such as syscalltrack. Furthermore a nice feature would be to add support for updating the rules without having to reload the module and also to improve the search algorithm for policies, this way considerably reducing the overhead time in case many security policies are applied.

VII. References

- [1] Linux-PAM http:// www.kernel.org/ pub/ linux/ libs/ pam/
- [2] Pluggable Authentication Modules for Linux http:// linux.die.net/ man/ 8/ pam.d
- [3] REMUS - REference Monitor for Unix Systems http:// remus.sourceforge.net/
- [4] Improving File System Reliability with I/O Shepherding - Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
- [5] Security Policies and Security Models / Access Control http:// www.cs.purdue.edu/ homes/ ninghui/ courses/ Spring06/ lectures/ lecture05.pdf
- [6] Enforceable Security Policies - Fred B. Schneider
- [7] Analyzing Consistency of Security Policies http:// www1.cs.columbia.edu/ ~locasto/ projects/ spcl/ docs/ research/ laurence97analyzing.pdf
- [8] Flask: Flux Advanced Security Kernel http:// www.cs.utah.edu/ flux/ fluke/ html/ flask.html
- [9] Raksha: A Flexible Information Flow Architecture for Software Security http:// csl.stanford.edu/ ~christos/ publications/ 2007.raksha.isca.pdf
- [10] Syscalltrack http://syscalltrack.sourceforge.net/
- [11] StMichael LKM http://sourceforge.net/projects/stjude